ASSIGNMENT 1

Due 11:59 p.m. November 19, 2017

IMPORTANT! For this assignment, you may work with another person as a team. You may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. The assignment is expected to be your work, designed and coded by you and your teammate. If you need help, please consult with your instructor or TAs. Specific policies on Academic Honesty in Computing are outlined in the course syllabus.

This assignment will give you an opportunity to "peel open" a computer and look at its internal structure by programming in machine-language, a set of instructions executed by the CPU. To make this an especially valuable experience, you will be also 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 simulated 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 Instruction Set Architecture. 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 machine instructions, each of which is an unsigned 2-byte number (defined as *word* in the VSM) comprised of the *op-code* and the *operand*:

op-code	*m*	*operand*
0 3	4	5 15

The bit pattern appearing in the op-code field indicates which of the elementary operations, such as READ or ADD, is requested by the instruction. The bit patterns found in the operand field provide more detailed information about the operation specified by the op-code. For example, in the case of an ADD operation, the information in the operand field indicates which memory location contains the data to be added to the accumulator. The middle bit *m* distinguishes between operands that are memory addresses and operands that are numbers. When *m* is set to 0, the operand represents an address; if it is set to 1, the operand represents a number.

A set of machine instructions for the VSM are listed in the table below:

Op-code	Mnemonic	Function
`0000`	`EOC`	End of code section
`0001`	`LOAD`	Load a word at a specific location in memory (or a number) into the accumulator.
`0010`	`STORE`	Store a word in the accumulator into a specific location in memory.
`0011`	`READ`	Read a word from the standard input into a specific location in memory.
`0100`	`WRITE`	Write a word at a specific location in memory to the standard output.
`0101`	`ADD`	Add a word at a specific location in memory (or a number) to the word in the accumulator, leaving the sum in the accumulator.
`0110`	`SUB`	Subtract a word at a specific location in memory (or a number) from the word in the accumulator, leaving the difference in the accumulator.
`0111`	`MUL`	Multiply the word in the accumulator by a word at a specific location in memory (or a number), leaving the product in the accumulator.
`1000`	`DIV`	Divide the word in the accumulator by a word at a specific location in memory (or a number), leaving the quotient in the accumulator.
`1001`	`MOD`	Divide the word in the accumulator by a word at a specific location in memory (or a number), leaving the remainder in the accumulator.
`1010`	`NEG`	Negate the word in the accumulator.
`1011`	`NOP`	No operation.

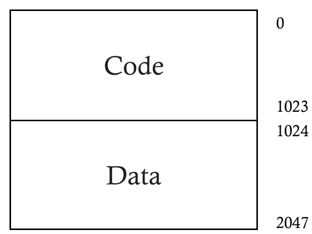
`1100`	`JUMP`	Branch to a specific location in memory.
`1101`	`JNEG`	Branch to a specific location in memory if the accumulator is negative.
`1110`	`JZERO`	Branch to a specific location in memory if the accumulator is zero.
`1111`	`HALT`	The program is terminated.

The middle bit *m* can be used only with the LOAD and five arithmetic operations (ADD, SUB, MUL, DIV, MOD). Here are some examples:

`0001100000001010`	*Load 10 into accumulator*
`0110100000000001`	*Decrement the word in the accumulator by 1*

The end of instructions of VSML programs must be indicated by the op-code `0`, followed by the input data required by the program.

The VSM Memory Layout. The VSM supports a memory system comprised of 2,048 bytes, partitioned into the *code* and *data* sections:



Before running a VSML program, it must be loaded into memory. The first instruction of every VSML program is always placed in location 0, the beginning of the code section. The data required by the program must be stored in the data section which begins at memory location 1024. All temporary data (variables) must be store in the data section.

EXAMPLE 1: The following VSML program (**sum.vsml**) reads two numbers (*x* and *y*) from the standard input, and computes and prints their sum (*z*). In this example, 30 = 10 + 20 will be output.

Location	Instruction	Comment
00	`0011010000000000`	*read x*
02	`0011010000000010`	*read y*
04	`0001010000000000`	*load x*
06	`0101010000000010`	*add y*
08	`0010010000000100`	*store z*
10	`0100010000000100`	*write z*
12	`1111000000000000`	*halt*
	`0000000000000000`	*end of code*
1024	`0000000000001010`	*10*
1026	`0000000000010100`	*20*

EXAMPLE 2: The following VSML program (**max.vsml**) reads two numbers (*x* and *y*) from the standard input, and determines and prints the larger value. In this example, 13 (max of 7 and 13) will be output.

Location	Instruction	Comment
00	`0011010000000000`	*read x*
02	`0011010000000010`	*read y*
04	`0001010000000000`	*load x*
06	`0110010000000010`	*subtract y*
08	`1101000000001110`	*branch negative to 14*
10	`0100010000000000`	*write x*
12	`1100000000010000`	*jump to 16*
14	`0100010000000010`	*write y*
16	`1111000000000000`	*halt*
	`0000000000000000`	*end of code*
1024	`0000000000000111`	*7*
1026	`0000000000001101`	*13*

Exercise 1. Write a VSML program (**prime.vsml**) which reads a number *n* from the standard input, and prints 1 if *n* is prime; 0 otherwise.

Exercise 2. Write a VSML program (**primes.vsml**) which reads a number *N* from the standard input, and prints all primes that are less than *N*.

Part II: The VSM Simulator

In this part you're going to build your own computer. You won't be soldering components together. Rather, you'll use the powerful technique of software-based simulation to create a software model of the 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.

Write a C program (**vsm.c**) which will simulate the VSM. Run your VSML programs from Part I using your simulator, namely, **sum.vsml**, **max.vsml**, **prime.vsml**, and **primes.vsml**.

When your simulator finishes running a VSML, it should display the contents of the registers and the 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 implement your dump function, a sample dump format is shown below:

```
REGISTERS:
accumulator                 0x0000
instructionCounter          0x0000
instructionRegister         0x0000
opCode                      0x0
operand                     0x0000

CODE:
     0  1  2  3  4  5  6  7  8  9
0000 00 00 00 00 00 00 00 00 00 00
0010 00 00 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00
0070 00 00 00 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 00
...

DATA:
     0  1  2  3  4  5  6  7  8  9
1024 00 00 00 00 00 00 00 00 00 00
1034 00 00 00 00 00 00 00 00 00 00
1044 00 00 00 00 00 00 00 00 00 00
1054 00 00 00 00 00 00 00 00 00 00
1064 00 00 00 00 00 00 00 00 00 00
1074 00 00 00 00 00 00 00 00 00 00
1084 00 00 00 00 00 00 00 00 00 00
1094 00 00 00 00 00 00 00 00 00 00
1104 00 00 00 00 00 00 00 00 00 00
1114 00 00 00 00 00 00 00 00 00 00
...
```

Here, **`accumulator`** represents the accumulator register. **`instructionCounter`** stores the location in memory that contains the next instruction to be executed. **`instructionRegister`** contains the current instruction being executed. You should not execute instructions directly from memory. Rather, you should transfer the next instruction to be executed from memory to **`instructionRegister`**. **`opCode`** indicates the operation currently being performed. **`operand`** represents the memory location on which the current instruction operates. For the memory dump, only the first 100 bytes of the code section and the data section should be displayed in hexadecimal without the prefix **`0x`**.

The input VSML programs to your simulator should consist of binary strings of length 16, each on separate line. The **scanf**() function supports both octal and hexadecimal conversions, but, unfortunately, it does not support binary conversion of the input. So use the conversion program (**binstr2hex.c**) from Lab 8 which reads binary strings from the standard input and displays their hexadecimal equivalent to the standard output. For example, the following code is the hexadecimal equivalent to **sum.vsml** converted by **binstr2hex**:

```
3400
3402
1400
5402
2404
4404
F000
0000
000A
0014
```

With this converter in place, your simulator should expect as input VSML programs entirely written in hexadecimal.

Programming Notes

1. Compile your programs with:

```
$ gcc –o binstr2hex binstr2hex.c
$ gcc -o vsm vsm.c
```

2. Run your simulator with:

```
$ ./binstr2hex < prog.vsml | ./vsm
```

   where **`prog.vsml`** is a VSML program. Note that we use a Unix pipe (`|`) to "feed" the output from **`binstr2hex`** to the simulator.

Handin

Upload C source (**binstr2hex.c** and **vsm.c**) and VSML source (**sum.vsml**, **max.vsml**, **prime.vsml**, and **primes.vsml**) to the course website.

Grading

VSM Simulator	20 points
VSML Programs	
sum.vsml	5 points
max.vsml	5 points
prime.vsml	10 points
primes.vsml	10 points

Your program will be graded based on the following criteria:

1. Correctness – produces correct results consistent with I/O specifications.
2. Design – employs a good modular design, function prototypes.
3. Efficiency – contains no redundant coding, efficient use of memory.
4. Style – uses meaningful names for identifiers, readable code, documentation.