- CPSC 275: Introduction to Computer Systems

CPSC 275: Introduction to Computer Systems

Fall 2025

- Syllabus
- Schedule
- Resources
- Upload
- Solution

Assignment 3: Simulating an Accumulator Machine – Part II

Due 5:00 p.m., Monday, October 20

> **IMPORTANT!** This is an individual assignment. You may discuss broad issues of interpretation, understanding, and general approaches to a solution. However, the development of a specific solution or program code must be your own work. The assignment is expected to be entirely your own, designed and coded by you alone. If you need assistance, please consult your instructor or the TAs. Be sure to read the specific policies outlined in the **Academic Honesty in Computing** section.

This is a continuation of Assignment 2, where you are asked to simulate an accumulator-based computer called the VSM (Very Simple Machine). The VSM runs programs written in the only language it directly understands—the VSM Language, or VSML.

The VSM instruction set architecture. The VSM contains an accumulator, a special register in which information is placed 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) of the following format:

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

The bit pattern appearing in the *op-code* field indicates which operations are requested by the instruction. The bit patterns 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 operand field indicates which memory location contains the data or a constant to be added to the accumulator. The middle bit *m* represents the type of the operand. When *m* is set to 0, the operand represents a memory address; if it is set to 1, the operand represents a constant.

All supported machine instructions for the VSM are listed below:

Op-code	Mnemonic	Action
`0000`	`EOC`	Signal the end of the program.
`0001`	`LOAD`	Load a word at a specific location in memory (or a number) into the accumulator.

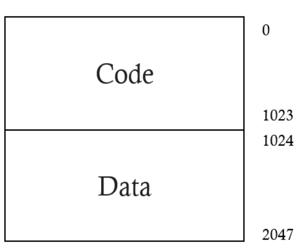
`0010`	`STORE`	Store a word in the accumulator into a specific location in memory.
`0011`	`READ`	Read a word from the standard input into a specific location in memory.
`0100`	`WRITE`	Write a word at a specific location in memory to the standard output.
`0101`	`ADD`	Add a word at a specific location 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`	Stop the program.

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

```
Instruction                       Comment
0001100000001010                  Load 10 into the accumulator
0110100000000001                  Decrement the word in the accumulator by 1
0111010000000000                  Multiply the word in the accumulator by the word at Mem[1024]
```

The end of instructions of VSML programs must be indicated by the op-code `0000` (`EOC`), followed by the input data required by the program.

The VSM memory layout. The VSM supports a memory system consisting of 2,048 bytes, divided into code and data sections:



Before running a VSML program, it must first be loaded into memory. The first instruction of every program is placed at location 0, marking 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 variables must also be stored in the data section.

Example 1. The following VSML program ([`sum.vsml`]) reads two numbers from the standard input and computes and prints their sum:

```
Location        Instruction         Comment
00              0011010000000000    read x into Mem[1024]
02              0011010000000010    read y into Mem[1026]
04              0001010000000000    load x
06              0101010000000010    add y
08              0010010000000100    store z at Mem[1028]
10              0100010000000100    write z
12              1111000000000000    halt
                0000000000000000    end of code
```

Example 2. The following VSML program ([max.vsml]) reads two numbers from the standard input, and prints the larger value:

```
Location        Instruction         Comment
00              0011010000000000    read x into Mem[1024]
02              0011010000000010    read y into Mem[1026]
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
```

The VSM Simulator. Modify your `assem.c` to simulate the VSM. Run the example VSML programs (`sum.vsml` and `max.vsml`) using your simulator. When your simulator finishes running a VSML program, it should display the contents of the registers and memory. Such a printout is often called a *computer dump*. A dump after executing a VSML program shows the actual instruction and data values at the point execution terminates. A sample dump 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** (Program Counter) 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. Instead, you must first 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`.

Compile your programs with:

```
$ gcc -Wall -o assem assem.c
```

Run your simulator with:

```
$ ./assem < prog.vsml
```

where `prog.vsml` is a VSML program. The input VSML programs to your simulator should consist of binary strings of length 16, each on a separate line.

Handin

- **Important!** ***A program with compilation errors will receive a zero***. Make sure your program compiles successfully and runs on our lab computers. Also, eliminate all compiler warnings, or points will deducted.
- Upload your C source (`assem.c`) to the course website.

- **Welcome: Sean**
    - LogOut

Trinity College
HARTFORD CONNECTICUT