- CPSC 275: Introduction to Computer Systems

CPSC 275: Introduction to Computer Systems

Fall 2025

- Syllabus
- Schedule
- Resources
- Upload
- Solution

Assignment 2: Simulating an Accumulator Machine – Part I

Due 5:00 p.m., Friday, October 10

> **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.

Introduction

An *accumulator machine* is a type of computer where a single special register, called the *accumulator*, is used as the primary location for performing calculations. Instead of using many registers like modern processors, all arithmetic and logic operations happen through the accumulator. For example, to add two numbers, the computer first loads one number into the accumulator, then adds the second number to it, and the result stays in the accumulator until it is saved. These machines were common in the early days of computing, especially from the 1940s to the 1960s, because adding more hardware registers was expensive.

In this assignment, you will simulate an accumulator machine called *VSM* (Very Simple Machine). VSM runs programs written in the only language it directly understands—VSM Language, or VSML. Your first task is to implement a decoder for VSM by completing the following two exercises:

Exercise 1

Write a function `binstr2num()` which converts a binary string of length 16 to an unsigned integer. The function must have the following prototype:

```
unsigned short binstr2num (char *);
```

For example, calling `binstr2num("0000000000001100")` should return 12.

Now write a C program (`assem.c`) which reads in one or more binary strings of length 16 from the standard input, calls the function `binstr2num` for each string, and displays the return value in hexadecimal to the standard output. Your program should behave as follows:

Sample Input

```
0011010000000000
0011010000000010
0001010000000000
0101010000000010
0010010000000100
0100010000000100
1111000000000000
0000000000000000
```

Output

```
3400
3402
1400
5402
2404
4404
F000
0000
```

Compile your program with:

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

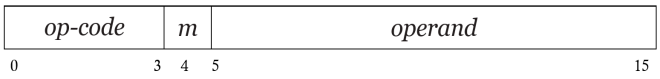
Run it with:

```
$ ./assem < assem.in
```

where `assem.in` is your input file to the program.

Exercise 2

Add a function `decode()` to `assem.c` which decodes the machine code for a 16-bit *instruction set architecture*. The instruction format is as follows:



The *op-code* field indicates the operation to be executed by the machine. The *operand* field represents the data used by the operation. The middle bit *m* represents the type of operand. When *m* is set to `0`, the operand represents a memory address; if it is set to `1`, it represents a constant. The following table describes the instructions implemented on this machine:

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.

The function `decode()` must have the following prototype:

```
unsigned short decode (unsigned short instr, unsigned short *opcode, unsigned short *opn);
```

where

- `instr`: instruction to be decoded
- `opcode`: opcode
- `opn`: operand

The function must return the value of the bit *m* (`0` or `1`). The `opcode` and `opn` must be declared in the `main()`, and their addresses must be passed to the `decode()` function. For example, when calling `decode(13312, &myopc, &myopn)`, where `myopc` and `myopn` are both integer variables, the function should return 0, and `myopc` = 3, and `myopn` = 1024, where 13312 is the first instruction in the sample input (in decimal).

Finally, modify `assem.c` so that it reads one or more 16-character binary strings from the standard input, calls the function `binstr2num()` followed by `decode()` for each string, and displays the instruction in hexadecimal, its opcode and operand. Given the sample code in Exercise 1, your program should display the following:

```
3400 READ  1024
3402 READ  1026
1400 LOAD  1024
5402 ADD   1026
2404 STORE 1028
4404 WRITE 1028
F000 HALT  0000
0000 EOC   0000
```

Note that the operands are displayed in decimal.

Documentation

1. Header comment: Every source file (`.h` and `.c`) must include a high-level comment at the top. This comment should describe the name and purpose of the program, along with your name and the date. For example:

```
/*
 * Program: hello.c
 * Purpose: Displays a friendly greeting to students in CPSC 275.
 * Author:  Peter Yoon
 * Date:    09/31/2025
 */
```

2. Function Comments: Every function must have a comment block describing its purpose, its parameters, and its return value. For example:

```
/*
 * Function: factorial
 * Purpose:  Computes the factorial of a positive number based on the formula:
 *    n! = 1 * 2 * 3 * ... n
 *
 * Parameters:
 *    n: a positive number
 *
 * Returns: the factorial of n
 */
```

Handin

Upload your source code (`assem.c`) to the course website.

- **Welcome: Sean**
  - LogOut