CS203 Lab 7 – SSAMv3.1 VM

Assigned: Wednesday October 22nd, 2025.
Due: Tuesday November 4th, 2025 by 11:59 pm.

The Lab Assignment.

Create a computer simulator, also called a Virtual Machine (VM), that implements the SSAMv3.1 assembly language (found on the lecture Moodle page in section Week 8, file "CS203 F25 calendar v1"). This can be implemented on your personal machines, but a completed VM is provided to you on a computer server. Your version will not be quite as nice, but will be easier to debug.

An assembler was also be provided to you on the lecture Moodle page for Week 7 as part of archive file "inclass examples (Mon and Wed; w/ assembler)". This assembler was written in a scripting language called Tool Command Language (TCL) and there are also usage instruction provided with the lecture Moodle Page for week 7 (file "Basic SSAM Assembler Instructions"). This assembler will generate a binary file of your assembly program, and it is this binary file where you will you're your program from.

The repository contains two assembly programs, shown in the example.

```
% tree inclass_examples
inclass_examples
├── inclassMon_function.a
├── inclassWed_jumps.a
└── ssam.tcl
```

These files after being assembled will create the following structure. You will need to install TCL on your machine, or use the server where TCL is already installed.

```
% tclsh ssam.tcl inclassMon_function.a
Loading file: inclassMon_function.a

% tclsh ssam.tcl inclassWed_jumps.a
Loading file: inclassWed_jumps.a

% tree
.
├── inclassMon_function.a
├── inclassMon_function.bin
├── inclassMon_function.la
├── inclassMon_function.lba
├── inclassWed_jumps.a
├── inclassWed_jumps.bin
├── inclassWed_jumps.la
├── inclassWed_jumps.lba
└── ssam.tcl
```

The files ending with the extension "`.a`" is the original assembly programs. Files with extension "`.la`" is a formatted assembly file created by the assembler with hex-addresses identified for each line. Files with extension "`.lba`" are fully expanded with binary encoding presented in a “friendly” way. The files with extension “.bin” are binary files containing the generated machine code.

Binary files are special files that cannot normally be opened without the user being challenged. They store information not as a character code, but the actual binary value and require some special handling to extract the information. This is described below.

For this lab, you will make a program that will take a binary file name, provided to the command line and perform the following actions.

1. Open the file.
2. Load the binary values into a character array.
3. Use the loaded character array as a program memory to run the stored program.
4. The program will accept character input to control how the program is run, line-by-line, and displayed.

Running the simulator program.

For simplicity in coding and debugging, you will be creating an interface that is somewhat similar to the previous lab. Where the program will take in a sequence of characters, with each character representing the following action.

q – will quit the program.

Q – will quit the program, dumping the current state of the VM to a log file called “dump_log.txt”, containing the detailed state of your simulation.

d – display the current state of the VM.

n – run the fetch execute cycle once.

N – run the fetch execute cycle once and display VM state to the screen. This would be equivalent to typing the “n” command and then the “d” command.

H – run the loaded program until a halt command is reached.

To allow the user more control over the processing, it should be possible to type in a sequence of commands, like the following: “nnndq”. The provided example string will run the fetch-execute cycle three times, display the computer state, and then quit. Using this string option will allow the user to run through the program, performing several steps as a sequence.

While the program is running, the actions being performed should be writing to a logging file called “log_file.txt”. The information written to the file will be each action and then a dump of the VM state. This will be one way of tracking what the program is doing in detail and will make debugging easier.

All detected errors should be written to both the logging file and the standard error-output. The easiest was to provide error processing is using the following `fprintf`. Also using `stderr` will ensure the output is even when there is a segmentation fault.

```
fprintf(stderr, “Print your error just like you would with a printf”);
```

The binary file name will be provided through the command-line as you did in your early labs.

The Assembler.

You do not need to recreate the assembler, just use the tool provide with the archive.

Program Architecture

The program will have three functional parts, the main simulation, controller, and memory. Each part will be implemented with a combined c-source file and h-header file.

`sim.c` – will contain the main and will implement the interface the user uses. This module will not have direct access to the stored data in the Memory and Controller modules. Instead, it must use getters and setters implemented in the other two parts of the program.

`memory.c` – will contain the memory array and provided getters and setters for storing/retrieving information to/from this array. This memory array should be hidden from the other parts of the program, as you have done in the previous lab. Generally, the Controller module will primarily be manipulating this Memory module, but the Simulation module will need to pass the file handler to this module so that the binary file can be loaded. The Simulation module will also need to use the getters to display information from this module. The Memory module cannot directly manipulate the data stored in the Controller module.

`controller.c` – implements the simulation of the CPU and will primarily be using the Memory getters and setters to implement retrieve program and data information from the memory. The Controller cannot directly access information stored in the Memory module. Similarly, the Simulation module can only access the Controller module through the provided functions.

The Memory Module and Little Endian

Reading from a binary file is not a complex thing, as shown in the code below. It is just like any other file, you just need to use the appropriate operations. To opening the file, use `fopen`, but instead of just an “`r`” for a read, it is a “`rb`” for read binary. Each byte is read with an `fread`.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  // int input_data_size = 0;

  unsigned char input_buffer[0xFFFF];

  FILE *fh = fopen("./binary.bin", "rb");
  if(ferror(fh)) {
    fprintf(stderr, "Error: Opening file %s.\n", "./binary.bin");
  }

  int input_data_size = 0;

  while(!feof(fh)) {
    fread((input_buffer + input_data_size),  // pointer to current location
          1,                                 // read one character
          1,                                 // width of one character
          fh);                               // pass the file handle
    input_data_size++;
  }

  fclose(fh);


  printf("bytes read %d\n", input_data_size);

  if(ferror(fh)) {
    fprintf(stderr, "Error: Reading from file %s (size %d).\n",
            "./binary.bin",
            input_data_size);
  }

  unsigned short b = 0;

  for(int i = 0x610; i < input_data_size; i += 2) {

    b |= input_buffer[i];
    b <<= 8;
    b |= input_buffer[i + 1];

    switch (b & 0xC000) {
      case 0x0000: {
        printf("0x%hx 0x%hx flow\n", i, b);
        break;
      }
      case 0x4000: {
        printf("0x%hx 0x%hx transfer\n", i, b);
        break;
      }
      case 0x8000: {
        printf("0x%hx 0x%hx manipulate\n", i, b);
        break;
      }
      case 0xC000: {
        printf("0x%hx 0x%hx jump\n", i, b);
        break;
      }
      default: {
        puts("Unknown command.");
      }
    }
  }

  return 0;
}
```

You will notice that each byte is placed individually, instead of casting a group of char values into a short. This type of manipulation is important because of how some machines store data, in little endian. By placing the values into a temporary value, things will be appropriately ordered.

This is one place where things will get tricky, if you have a machine that is Little-Endian. The challenge is that short, int, and long are stored in reverse order. So, the hex value of a short is `0x4321` and stored into a short variable, and then you try to use it as a paired character value, the encoding will change to `0x2143`. The way to avoid problems is to work with the stored data as character and use bit-wise values to load and manipulate the short value, as shown in the code above.

To ensure that there will be no problems with your final code, be sure to use the provided server account from the start of the semester, which will allow you to test your code in a consistent development environment. But to be safe, please identify what type of computer you used to develop your code at the start of your "`sim.c`" file.

<u>The Controller and the Registers</u>
All registers should be implemented as part of the Controller module. To make things a close to the provided RTN in the SSAMv3.1 design, you can implement your registers in an array and use Enums to access individual registers. With the code looking something like the following.

```
// an enum to access the registers
typedef enum  {
  R0 = 0,
  R1 = 1,
  R2 = 2,
  R3 = 3,
  AC = 4,
  SP = 5,
  BP = 6,
  PC = 7
} regnames;

// definition of the registers
unsigned short reg[REGCNT];

// register initialization
reg[R0] = 0x0000;
reg[R1] = 0x0000;
reg[R2] = 0x0000;
reg[R3] = 0x0000;
reg[AC] = 0x0000;
reg[SP] = sp_start;
reg[BP] = bp_start;
reg[PC] = pc_start;
```

Other things to keep in mind when implementing your simulated computer is that there needs to be some way of tracking when the computer is running or stopped. This is used to track when the halt operation is called, causing a running computer to change into stopped mode. When the computer is stopped it will no longer perform any more actions and will ignore any calls to perform another fetch-execute. Similarly, there needs to be a way for the computer to be initialized, starting the PC, SP, and BP are being in the correct location.

Project Requirements.

For this lab, a partial project will not be provided, and the requirements are listed here.

- The three modules (sim, controller, and memory) will only interact with each other through functions and not by direct manipulation of their data.
- You will store the state of the computer in the memory array and registers, there will not be partial results in other locations.
- You are allowed to use bit-wise operators and normal addition, just be careful to manipulate your data appropriately when converting between unsigned short and character values. You cannot use integers or any variable type larger than a short.
- You will need to implement some of your homework solutions in assembly and be able load it into your simulations. An example simulation will be provided, as will the assembler later in the first week of the lab.
- The displayed information format is left up to you, but all code for implementing this display format should be part of the Simulation module with the data pulled from the memory using a getter function.

Lab Submission.

Once done, zip or tgz the lab assignment and submit it to the Moodle.