CS-UY 2214 — Project 2

pstein, Ratan Dey

1 Introductio

This project represents ing exercise. Like all work for this class, it is to be completed individually: any form ibited, as detailed in the syllabus. This project is considered a take-home exam.

Before even reading this assignment, please read the E20 manual thoroughly. Read the provided E20 assembly language examples.

2 Assignment: Simulator

Your task is to write an E20 simulator: a program that will execute E20 machine language. Normally, machine language would be executed by a processor, but for simplicity, we will reproduce the behavior of an E20 processor in software. A correct simulator is one that will produce identical results to those produced by a real E20 processor, as described in the E20 manual.

Each E20 machine language program is a sequence of commands to be interpreted by an E20 processor, or a simulation thereof. Your simulator will need to accurately reproduce the state that is manipulated by those commands: the program counter, the general-purpose registers, and memory.

For example, consider the machine language instruction `0010100010000011`. This machine language instruction corresponds to the assembly language instruction `addi $1, $2, 3`. Therefore, in order to execute this instruction, we must first know the current value of register `$2`. We add 3 to that value, and store the sum in register `$1`. This new value may then be accessed by subsequent instructions.

The basic operation of your simulator is as follows:

1. Initialize the processor state, including the program counter, the general-purpose registers, and memory.
2. Examine the instruction pointed to by the program counter. Determine what action is to be taken.
3. Take the indicated action, updating the value of the program counter, the general-purpose registers, and memory appropriately.
4. If the executed instruction is a `halt` instruction, end the simulation.
5. Otherwise, go to step 2.

For the purposes of this simulation, the initial state of the program counter is zero, and the initial state of all registers is zero. The machine code program will be loaded into memory starting at address zero, and the value of all other memory cells is zero.

2.1 Input

The input to your simulator will be the name of an E20 machine language file, given on the command line. By convention, E20 machine language files have an `.bin` suffix.

Your program will read in the contents of the file. You may assume that the file contains well-formed E20 machine language code. The file may contains comments, which your program should ignore.

You are provided with several examples of valid E20 machine language files, which you can use to test your simulator.

Here is an example of an E20 machine language program, in a file named `loop3.bin`, which was produced by assembling the file `loop3.s`:

```
ram[0] = 16'b0000000000010000;          // add $1,$0,$0
ram[1] = 16'b[illegible]                // add $4,$0,$0
ram[2] = 16'b[illegible]                // lw $3,value($0)
ram[3] = 16'b[illegible]                // loop: slti $1,$3,20
ram[4] = 16'b[illegible]                // jeq $1,$0,skip
ram[5] = 16'b[illegible]                // add $4,$4,$3
ram[6] = 16'b[illegible]                // addi $3,$3,1
ram[7] = 16'b[illegible]                // jeq $0,$0,loop
ram[8] = 16'b[illegible]                // skip: halt
ram[9] = 16'b0000000000010000;          // value: add $1,$0,$0
```

Note that each line consists of a memory address, followed by an equals sign, followed by a 16-bit binary number in Verilog syntax, followed by a semicolon. Comments, if present, will be in Verilog syntax.

2.2 Output

Your program should print to stdout the final state of the simulated E20 processor, at the point when the simulation halts. Specifically, your program should print out the final value of the program counter (in unsigned decimal) and the eight general-purpose registers (in unsigned decimal). In addition, your program should print out the value of the first 128 memory cells (in hex).

Below is an example invocation of a simulator from Linux's `bash`. In this case, we are simulating the execution of the machine language program given above. Text in italics represents a command typed by the user.

```
user@ubuntu:~/e20$ ./sim.py loop3.bin
Final state:
    pc=     8
    $0=     0
    $1=     0
    $2=     0
    $3=    20
    $4=    70
    $5=     0
    $6=     0
    $7=     0
0010 0040 8189 2c94 c403 11c0 ed81 c07b
4008 0010 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

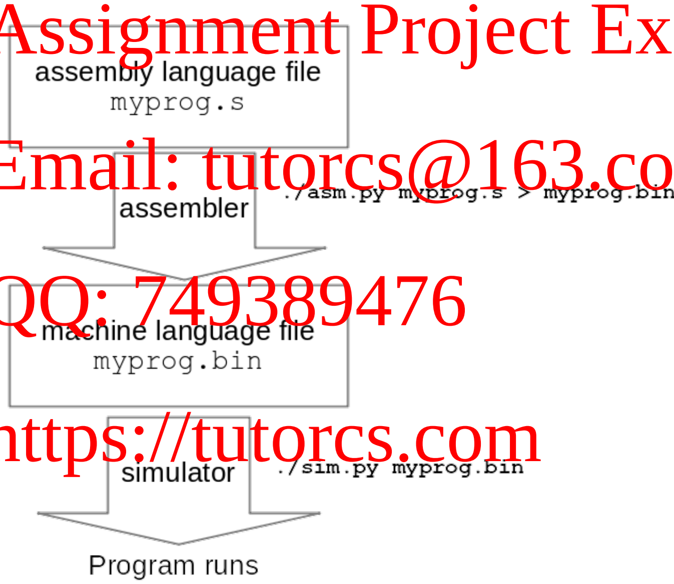

Your simulator should produce output in exactly the format shown above. Note that register values are printed as unsigned dec[illegible]mory cells are printed as 4-digit hexadecimal number.

Your solution will be [illegible] so it is important that your simulator produce output identical to the output above. P[illegible] for superficial deviations.

2.3 Testing

Several example machi[illegible] provided for you. In addition, with each machine code file you can find the corres[illegible]uage file, which includes the expected execution result. You can use these machine [illegible] correctness of your simulator. However, you should not rely exclusively on these examples, as they are not sufficient to exercise every aspect of a simulator. You are therefore expected to develop your own test cases.

Combining with assembler: Combined with your complete E20 assembler, you can now run any E20 assembly language program in your simulator. First, you must convert the assembly code into machine code with the assembler. Then, you can run the machine code in the simulator.



Your assembler can convert an assembly language file (with a `.s` suffix) into a machine language file (with a `.bin` suffix) using the following command. Text in italics represents a command typed by the user.

`user@ubuntu:~/e20$` *./asm.py myprog.s > myprog.bin*

The greater-than (>) symbol redirects the output of the assembler into the specified file.

You can then run the resulting machine code file like this:

`user@ubuntu:~/e20$` *./sim.py myprog.bin*

Alternatively, you can run your assembler and simulator together in one step like this:

`user@ubuntu:~/e20$` *./sim.py <(./asm.py myprog.s)*

2.4 Starter code

You may, but are not required to, use the provided starter code for this assignment, found in the files `sim-starter.cpp` and `sim-starter.py`. Please rename them to `sim.cpp` or `sim.py`, as appropriate.

Note that the starter code provides a function that will parse the machine code file into a memory array (`load_machine_code`), and also a function to generate output in the correct format (`print_state`). You should make use of thes[illegible]

3 Hints

- In order to run a [illegible] the Linux command line, it must first be marked executable. Otherwise, you m[illegible]nied" error message.

  To mark your Py[illegible], use the following command (assuming your file is named `sim.py`) from `bash`:

  ```
  chmod u+x sim.py
  ```

  Also make sure that the first line of the file specifies the path to the Python interpreter: it should be `#!/usr/bin/python3`. See the provided starter code. If you get an "exec format error," the problem is usually that that the first line is wrong.

  Alternatively, you can run the program by typing `python3 sim.py`.

- Your program must access its command-line parameters in order to know the name of the machine code file. In Python, you can use `sys.argv[1]`, although I recommend you use the `argparse` library, as shown in the starter code. In C++, you should use the `argv` parameter to `main`.

- The use of strings to store binary numbers is *strongly discouraged*. In short, numbers are numbers and should be stored as such; converting numbers to strings for the purpose of bit-extraction and bit-twiddling will lead to inflexible, unreadable code, and may introduce unanticipated bugs.

  Please review the hint from the earlier project, in which we advise you of techniques to manipulate bits in a number without resorting to strings.

- In order to get a good grade in this assignment, you will need to consider many edge cases in the E20 design. You are strongly advised to carefully read the entire sections of the E20 manual covering architecture and instruction set, and ensure that every detail is reflected in your program. Here is an incomplete list of questions for consideration as your read the manual:

  - What are the initial values of the registers, the program counter, and the memory cells?
  - What should happen if a program sets a register to a value outside of the range expressible as a 16-bit unsigned number? Consider both positive and negative numbers that cannot be expressed in 16 bits.
  - What should happen if a program tries to change the value of `$0`?
  - What should happen if a program uses `slt` to compare a negative number to a positive number?
  - What range of memory address are valid? What should happen if a program tries to read or write a memory cell whose address is outside of the range of valid addresses?
  - What should happen if a program sets the program counter to a value outside of the range of valid addresses?
  - What should happen if a program uses a negative immediate value in `addi` or `jeq`?
  - What should happen if a program uses a negative immediate value in `lw` or `sw`?
  - What should happen if a program modifies a memory cell containing machine code?

- What should happen if the program counter reaches the address of the last memory cell?
- When should your simulator stop?

You should be able to answer all of these questions, and others, before you start coding. You are encouraged to write test cases for each of these situations, so that you can verify that your simulator handles them cor[illegible]

- Your simulator sh[illegible] valid input. This is because your simulator should reproduce the behavior of a [illegible]at *cannot* crash.

  In this case, "vali[illegible]ine-code program (i.e. a sequence of 16-bit values) such that no invalid instruct[illegible]

4 Rules

Language You should implement this project in Python 3 or in C++.

File names and building If you are using Python 3, you must name your program `sim.py`. If your solution consists of multiple source files, submit them as well. Assume that your program will be invoked by running `sim.py` with a filename as its parameter, using Python 3.6.

If you are using C++, you must name your program's main source file `sim.cpp`. If your solution consists of multiple source files, submit them as well. Assume that your program will be built by gcc 8.3.x using the command `g++ -Wall -o sim *.cpp` and then run by the executable `sim` with a filename as its parameter. If you use C++, your program should compile cleanly (i.e. no errors or warnings) with gcc 8.3.x.

Libraries You are free to make use of all packages of the standard library of your language (that is, all libraries that are installed by default with Python 3 or C++, respectively). Do not use any additional external libraries. Do not use any OS-specific or compiler-specific extensions.

Tools Your program submission will be evaluated by running it under the GNU/Linux operating system, in particular a Debian or Ubuntu distribution. Your grade will therefore reflect the behavior of your project code when executed in such an environment. While you are welcome to develop your project under any operating system you like (such as Windows or Mac OS), you are responsible for any operating system-dependent deviations in program behavior.

Academic integrity You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else. Code may not be developed collaboratively. You may rely on publicly-accessible documentation of the language and its libraries. Please read the syllabus for detailed rules and examples about academic integrity.

Code quality You should adhere to the conventions of quality code:

- Indentation and spacing should be both consistent and appropriate.
- Names of variables, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- All functions should have a documenting comment in the appropriate style describing its purpose, behavior, inputs, and outputs. In addition, where appropriate, code should be commented to describe its purpose and method of operation.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in grading your work.

Submission You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file (not a PDF file and certainly not a Word file) containing the following information:

- Your name and N [illegible]
- The state of your [illegible] ete the assignment? If not, what is missing? Be specific. If your assignment i [illegible] wn bugs, I prefer that students let me know, rather than let me discover these [illegible].
- Any other resourc [illegible] n developing your program.
- Justify your desig [illegible] ou write your program the way you did? If you feel that your design has notable strengths or weaknesses, discuss them.

Submit your work on Gradescope. Submit all source files necessary to build and run your project. Do not submit external library code. Do not submit binary executable files.