Computer Assignment 1 **Deadline:** Friday, Oct. 27, 11:55 PM

Fall 2023 (Upload it to Gradescope.)

The goal of this computer assignment is to create a simulator for a (simple) RISC-V CPU.

Here are the important rules that we will use in all of these computer assignments:

- We will use the 32-bit version of RISC-V ISA.
- It is preferred to do all your work using C/C++ programming language, but you are free to use any other languages as long as your code can be executed and pass the tests on Gradescope. We will provide a skeleton code written in C/C++ only but you can use your own code based on this.
- Your code should be written using only the standard libraries. You may or may not decide to use classes and/or structs to write your code (C++ is preferred). Regardless of the design, your code should be modular with well-defined functions and clear comments.
- You are free to use as many helper functions/classes/definitions as needed in your project.
- There is no restriction on what data type (e.g., int, string, array, vector, etc.) you want to use for each parameter.
- Unfortunately, experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered, or if something needs to be described better. It is your responsibility to check the website often and read new versions of this project description as they become available.
- Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy. You are allowed to compare your results (only for the debug part) with others or discuss how to design your system. You are also allowed to ask questions and have discussions on Campuswire as long as no code is shared.
- You have to follow the directions specified in this document and turn in the files and reports that are asked for.

Project Description

Overview

In computer assignment 1, we will design a RISC-V single-cycle processor using a list of instructions provided later. Your task is first to design your datapath and controller based on what you have learned in class and then implement it in C/C++ (or your preferred language).

To help you get started, you can download the skeleton code from the website. Along with this description and the skeleton code, we have uploaded several test files (assembly and binary) that you could use to test your code.

Your processor should read a given trace (i.e., a text file that has the instructions), execute instructions one by one, and report the final value in `a0` and `a1` registers. Further, using your code, you should answer the questions at the end of this project description and submit your answers as a separate PDF document.

We have also uploaded a folder named “trace” which you can use to develop your project. We have provided binary files and their assembly codes. Instructions are saved in “23instMem-x” (in unsigned decimal format, where each line is one byte and stored in little-endian format). In addition to the “23instMem-X” files, we have uploaded these files: “23all.txt”, “23sw.txt”, and “23r-type.txt”. These files show the actual assembly program for each of “instMem-X” files. You see three columns in each file. These columns show the memory address (in hex), the instruction (in hex), and the actual assembly instruction. For example:

```
“14:  00f06693   ori x13 x0 15”,
```

shows that this instruction’s address is 0x14 (in hex), its binary value (NOTE: shown in hex) is `00f06693`, and the assembly representation is `ori x13 x0 15` (NOTE: immediate values can be in decimal or hex). The last line in each file indicates the correct final value for `a0` and `a1`. Use those to debug your code.

These three files are designed such that the first file (“r-type”) only contains r-type and i-type (except `lw`) instructions. It is recommended to start with this file, so you can gradually design your processor. For this part, you can safely assume that the next `PC` will be `PC+4`. The second file (“`sw`”) adds memory instructions to the mix (still no `branch`). Once you make sure your design works correctly for r-type instructions, you can start focusing on “`MEM`” stage. Finally, we added an “all” file that combines all these instructions, so you can use that as the ultimate test.

We have also uploaded “.s” files for each trace which shows the assembly program with no other details. You can use any standard RISCV compiler to compile and run this on your computers or use web-based models to run the instructions if you want to.

Also, note that there is no END instruction in our traces. The assumption is that your instruction memory will fetch an all-zero instruction after it fetches the last instruction in each trace, and when it reads an instruction with `OPCODE = zero`, it will terminate and print the results.

Instructions:

You need to implement the following list of instructions:

```
ADD, SUB, ADDI, XOR, ANDI, SRA
LW, SW
BLT, JALR
```

You should use the resources provided in class or posted on Campuswire to follow the standard format (i.e., opcode, funct3, etc.) for each instruction. Use the Datapath+Controller described in the class as your baseline and modify it as needed to support the instructions that are not supported in the lecture (e.g., BLT, SRA, and JALR). Your design should support ALL these 10 instructions.

You are free to modify the datapath+controller as you see fit. Try to be efficient (i.e., use the minimum number of additional hardware/ports to support the new instructions), but we won't deduct points if your design is not efficient. The only criterion for checking your code is that it can correctly print the final results (details below).

Code:

The entry point of your project is "`cpusim.cpp`". The program should run like this:

"`./cpusim <inputfile.txt>`",

and print the value of `a0` and `a1` in the terminal:

"`(a0,a1)`" {no space}

(for example, if `a0=10`, and `a1= -8`, then you should print (10,-8). You should use **exactly** this format, otherwise, our automated tests cannot evaluate your code.)

It is your choice how you want to structure your code (e.g., whether you want to have separate objects for each class, or you want to instantiate an object within another class, or even not use any class at all and utilize functions and structs, etc.). Our main suggestion is to use a separate function/class/struct for each unit.

The critical point to remember is that in hardware, activities happen in parallel but in C/C++, steps are sequential. As a result, **you should be very cautious about using different values from different functions/units.** A simple solution for this is to use *"current"* and *"next"* naming conventions for your (register) values. For example, `(current)_PC` and `next_PC`, or `rs1`, and `next_rs1`, etc.

You can treat the `clock` as a counter that starts from zero when the program starts. Each cycle can be modeled as one iteration of a `WHILE` loop. At the beginning of each iteration, `next_` values are updated with `current_` values. Then, every module should only use the `current_` values as *inputs* and update `next_` values. This process repeats until the `OPCODE` in all stages is `ZERO`.

What to Do/Where to Start:

You should start with first designing your datapath. Then design your controller, i.e., what control signals are needed, draw a table, and decide their respective values for each instruction (same as what we did in the lecture).

Once you have your full design on paper, start planning your functions, input/outputs to each function, etc. Use the skeleton code to get started.

Make sure to test your code at every step. The simple way to do this is to look at various steps of an instruction (e.g., fetching, decoding, reading operands, etc.) and check your code to make sure that each stage is implemented correctly before implementing the next stage.

Details/Questions/Clarification for Each Unit:

(This part will be added gradually, if needed, based on the discussions on Campuswire.)

Questions:

(to answer these questions, you may need to add additional functionality to your simulator.)

1. What is the total number of cycles for running "all" trace (ZERO instruction included)?
2. How many r-type instructions does this program ("all") have?
3. What is the IPC of this processor (for "all" trace)?

What to submit

You need to submit the following files on Gradescope.

1. Your well-commented code (all the files). Note that we will use a different trace to test your code's correctness. Your code should be compiled with the following command: "g++ *.cpp -o cpusim" (in case you are using C, it would be "gcc *.c -o cpusim"). If the code fails to compile, you will lose points. Your code should produce the results in the format described on the previous page. Failing to create the above format will result in losing points.
2. A short report (a PDF file) that shows your design (datapath with control signals), a table that shows the values for each control signal for all instructions, and finally answers to the questions on page 4.

Grading

1. 80% for passing the automated tests. This will be graded as soon as your code is uploaded on Gradescope. You can submit as many times as you would like until you get the full grade.
2. 20% for manual grading. This will be graded by me/TAs based on the PDF report that you have uploaded.