Computer Assignment 1 Deadline: Sunday, Nov. 7, 11:55 PM

Fall 2021 (Upload it to Gradescope.)

The ultimate goal of these computer assignments is to create a (simple) C/C++ simulator for a (simple) RISC-V CPU. At each step of this project, we will gradually add more units/capabilities to our processor.

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 and will focus on implementing only 10 instructions that were described in Lecture 5.
- Later on, we might add more instructions and/or capabilities to our processor.
- You should do all your work in the C/C++ programming language. 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/class/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.

Project Description

Overview

In computer assignment 1, we will design a RISC-V 5-stage pipelined processor using the 10 instructions described in the lecture. For this project, you can ignore Data and Control hazards. ***No stalling and/or forwarding is needed.*** The full datapath is shown on page 4 (see Figure 1). For this project, you don't need to implement the "`br`" instruction, and you can assume that the next `PC` is always `PC+4`.

Along with this description, we have uploaded an initial design for the project, as well as three traces.

- **GOAL:** 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 "files" folder which you can use to develop your project. We have provided binary files and their assembly codes. Instructions are saved in "instMem" (in unsigned decimal format, where each line is one byte, and stored in little endian format). In addition to "InstMem-X" files, we have uploaded these files: "test", "lw/sw", and "r-type". 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 14 (in hex), its binary value (NOTE: shown in hex) is 00f06693, and the assembly representation is ori x13 x0 15 (NOTE: imm values are all in decimal).

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 ("`lw/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 a "test" file that combines all these instructions, so you can use that as the ultimate test.

Also, note that there is no NOP and/or 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 5 consecutive instructions with `OPCODE = zero`, it will terminate and print the results.

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)`"

(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 using any class at all and utilize functions and structs, etc.). Our main suggestion is to use a separate function/class/struct for each stage. Also, use additional functions within each stage. This way, the code is much easier to track and debug.

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`.

- **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 "test" trace (ZERO instructions included)?
2. How many r-type instructions does this program ("test") have?
3. What is the IPC of this processor (for "test" trace)?

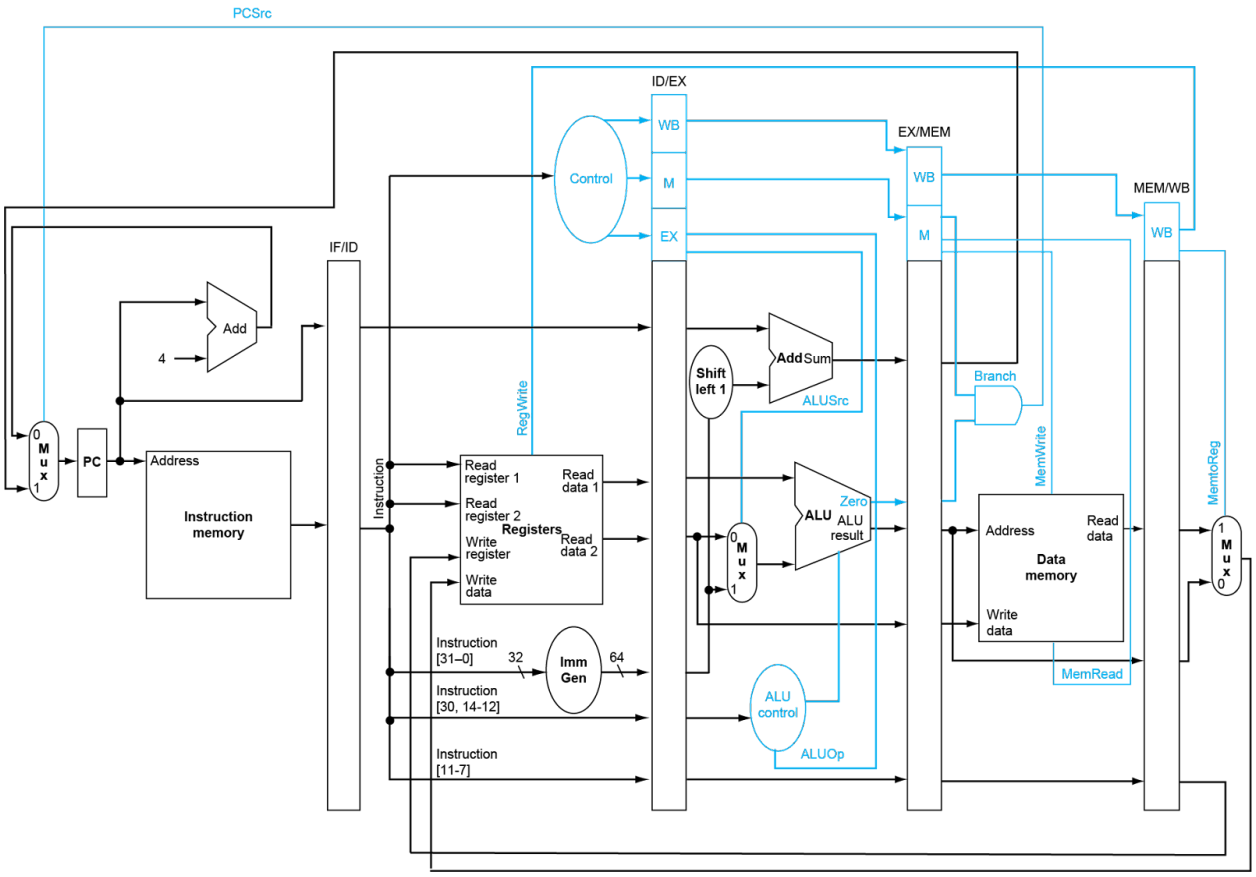

Figure 1: Datapath for the 5-stage pipeline processor that should be implemented in this project.

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 answers the questions on page 3.