

This project is a group project with two members max. You can certainly choose to work by yourself and it's actually encouraged!

Due to the deadline being closer to the end of the semester, no extension is allowed on this project. No exceptions.

1 Description

1.1 Hardware

Briefly, in this project, you will use Logisim-Evolution to create a sequential datapath, aka your own CPU! It should include all the essential parts we have discussed in class: instruction memory, register file, ALU, data memory, and of course, PC. This CPU doesn't need to be complicated, but needs to be fulfill the following basic requirements:

- ▶ At least four general purpose registers in the register file;
- ▶ At least be able to do addition and another arithmetic operation of your choice;
 - The arithmetic instructions have to read two registers and then write to another register;
 - Reading one register and using an immediate number is encouraged, but not required;
- ▶ Be able to load the data from memory, and store the data to memory;
 - Both load and store instructions need to include offset;
 - The offset can be either immediate number or a register read;
- ▶ A sequential (single-clock cycle) implementation of the datapath.

The past three labs are very useful as a start, and you can certainly reuse them in the project.

1.2 Assembly and Machine Code Design

As you have learned in class, the binary encodings of assembly instructions heavily depend on the actual hardware architecture of the CPU. Therefore, in order to make your CPU work, you have to design your own instruction set architecture. Thinking about how we mapped the instructions to hardware data signals, we can go from the low level machine code to higher level assembly code:

- ▶ For each instruction, determine how many bits you need to represent the machine code, and which field will be used for what purpose. We recommend using 8 bits, or multiples of 8 bits;
- ▶ All instruction should contain the same number of bits, but if you like challenges, feel free to do it in a more complicated way where each instruction might contain different number of bits;
- ▶ Given those instructions, you need to decide their mnemonics, and the operands.

Once you have figured these out, write them down as a manual, so other engineers ☺ can use your CPU to do great things!

Note: for a basic implementation, you don't need to design a control unit, an ALU control, or a complicated opcode field in the machine code. Since you only need to implement two types of calculation, you can just encode one bit in the machine code to decide which operation you want to do.

1.3 Simulation

Next step to test your great innovation is to write a small assembly program in the language you designed on your own. It doesn't have to be complicated like ARM; just a sequence of instructions will be fine. Then, you need to write a small program to translate that assembly into an image file (containing only hexadecimals and addresses), so we can load it to the RAM, and start executing it.

The program is basically a small assembler, and you don't need to check syntax errors. Just write a bunch of rule-based translation, so it can translate each instruction into corresponding machine code in hexadecimal. You can write this program in any of the following languages you like: C, Java, Python, JavaScript, or even ARM assembly.

Remember in the manual, provide instructions on how to use your assembler.

1.4 A Great Name!

Lastly, give your baby CPU a badass name and be proud of it! Write your CPU's name in the `.circ` file.

2 Requirements

If this is done by two people, each of you need to submit the same copy of the deliverable on Canvas. These files are required for everyone:

- ▶ Logisim-Evolution `.circ` file of your CPU, including: name and pledge, partner's name (if done in group), CPU's name;
- ▶ A compilable or executable program that can assemble your assembly program into image files. No need to do error checking; you can assume the program is always correct;
- ▶ A demo program in the assembly language you designed. This program should load the data from the main memory, execute arithmetic calculation, and store the data back to the main memory;
- ▶ A user manual (PDF file) including:
 - Name of the CPU;
 - Your name (and your partner's name if done in group);
 - A clear and detailed job description of each person in your group. If done by one person then you can skip this part;
 - How to use your assembler program (e.g., what command, what library to link, etc), and how to load it to the instruction and/or data memory;
 - The architecture description of your CPU, e.g., how many general purpose registers and how can we refer them in an assembly program, what functions your CPU can do, etc;
 - For each instruction implemented, write the general format (such as `ADD Rm, Rn, Rt`), and its binary encoding. You can follow a similar description as in Chapter 3.2 in textbook. When describing binary encoding, you need to specify why you need this number of bits for that field.

3 Grading

The basic requirement of the project will be graded based on 100 points in total:

- ▶ **50:** a correct and working CPU circuits:
 - **10:** four or more general purpose registers;
 - **10:** correct execution of addition and another arithmetic operation;
 - **10:** correct loading and storing instructions;
 - **20:** correct execution result of any combination of the instructions;
- ▶ **30:** a working assembler program:
 - **10:** be able to compile without errors (If you write the program in Python, this 10 pts is free);
 - **20:** be able to successfully translate text file that contains assembly code into image file that can be uploaded into memory and can be recognized by the circuit;
 - **Note:** you will not receive credit for this part if you didn't include an assembly demo code;
- ▶ **20:** a detailed documentation PDF file.

3.1 Extra Credits

The above is only the basic requirement for the project, but this project has so much potential to be fantastic. Extra credits will certainly be offered. Some ideas include (from easier to harder):

- ▶ **5 pts:** Making your assembler recognize data and text segment so it can generate two image files. They will be loaded into instruction and data memory separately;
- ▶ **5 pts:** Adding extra fun components, such as LED display to show calculation results;
- ▶ **15 pts:** Implementing branching related instructions. This needs you to be able to maintain a table for all the labels, and your assembler should be able to calculate offsets. The datapath is more complicated too;
- ▶ **20 pts:** Creating a pipeline version of the datapath (without hazard).

Just use your imagination and amaze me!

Note: extra credits are only considered when the basic requirements have been fulfilled. The credits will be offered from 5 pts to 20 pts, based on the work.

3.2 Early Bird Extra Credits

To encourage you to start the project early (for your own good), we have different tiers of early bird extra credits for the project. Submitting N days before the deadline will earn $N\%$ extra. Note that $N \leq 5$, meaning if you submit 6 or more days before the deadline, you will get at most 5%.

Deliverable

Zip all the files and submit on Canvas.

You should feel very rewarded after this project! If you become the CEO of a global IT giant in the future, feel free to add this course (and my name) to your biography.

Enjoy!