CSE/EE 469 Wi 2023: Lab 1

Due dates:

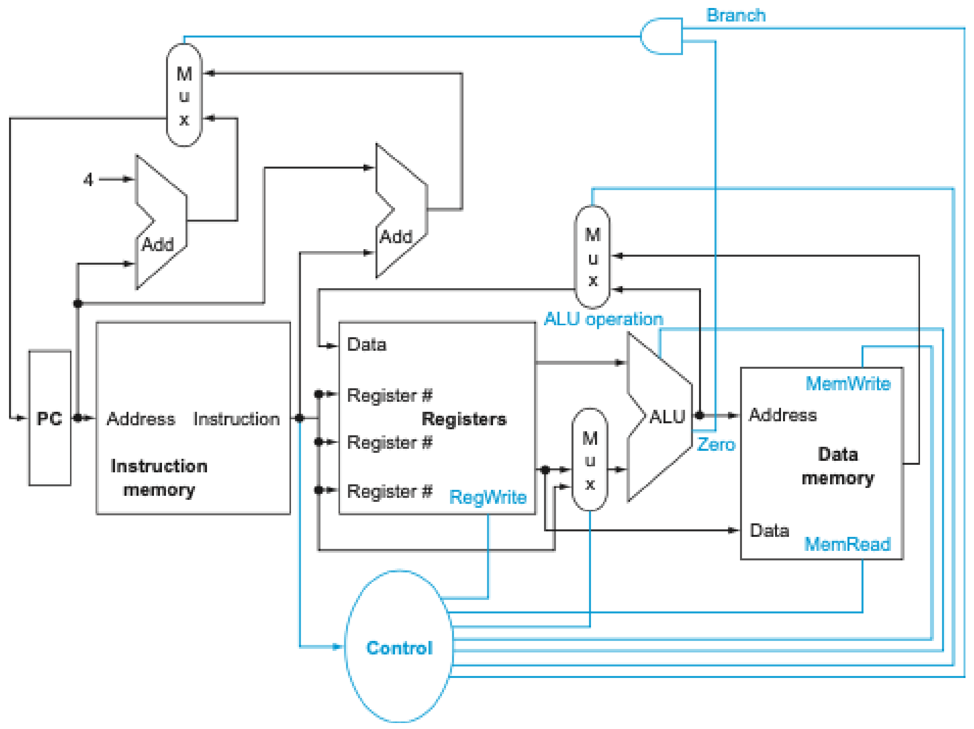
1st. Check in with TAs and show what you have completed: Feb 6th-8th
2nd Check in with TAs and show what you have completed: Feb 13th-15th
Final due date Feb 17th

Goal of This Lab

Design a 5-clock (or 6 or 7 :-) cycle 32-bit RISC-V cpu in Verilog (or SystemVerilog).
The CPU should support all rv32i instructions (except for ECALL, EBREAK, FENCE and all CSR instructions). MUL is optional. For software track we recommend it (it is easy once everything else works). For hardware track if your fpga is large enough, go for it. Otherwise just skip it and let gcc synthesize it from other instructions.

When you are done with this lab your processor should be able to execute arbitrary C code compiled for rv32i.

After this lab we will be optimizing this design by adding pipelining.
Here is a simplified diagram of the components of the RISC-V architecture from the textbook:



The main components of this design are:

- Program Counter (PC)
- Instruction memory
- Register File
- ALU
- Memory
- Writeback

All of these components will be clocked. That is at the end of the clock the results of that stage should be stored. After building these components, you will connect them together and make sure they can run instructions.

Your design will use 5-7 cycles per instruction. It will not be pipelined. These cycles correspond to “Fetch”, “decode/register read”, “execute”, “memory” and “write back”. You can use five, but you may use more too. You can get tricky and make your processor execute some instructions in less cycles. Honestly it is not worth it for this lab. Focus on correctness.

Code memory must be readable and writable from the processor itself. But you must use two memories, one for code and one for data. You will want to implement logic to route requests based on address. You need this ability if you are going to be able to load code into your processor at runtime. Gcc also depends on being able to read the code segment.

The register file should use a clocked memory with two read ports and one write port.

Starter Code

Here is the top level SystemVerilog starter code. Use this file to build the rest of your CPU onto. If you are using Verilator, here is an example C++ testbench file to get started on Verilator.

Development Environment

You can use any development environment that works for you. You can use Quartus, Verilator, Icarus Verilog, Vivado etc. ***We strongly encourage Verilator***.

You will need to write small test programs and load them into your processor at synthesis time to test it.

Software Track

Once you make sure your simulation is correct, to demonstrate your code works, we encourage you to use the "`libmc`" provided in the rv32sim.tar file. Edit the "`putc`" command to poke an address that you then interpret to print a character. Use the Verilog `$write` command to print the I/O. Then write some simple code and show that your design works. Your design does not need to load the code dynamically. Just use Verilog's built into facility to load an array.

Hardware Track

SIMULATE your design and make sure your design works, BEFORE uploading it to your board.

For this project you do not need to load code into your design dynamically, but you do need to demonstrate that it works. Code can be loaded at synthesis time. That being said, you may want to load it dynamically as it is not that challenging if you have read/write ability to/from your system to the FPGA. It may even help with debugging.

Demos

Demo TBD
During the demo, you will walk the TA through your cpu code and show that it can run printf.