In this project, you will design you own Instruction Set Architecture (ISA) and make a Multi-Cycle Processor (Datapath) that supports your ISA.

Step 0 - Set Up Your Environment

For this class, we strongly encourage you to do all your projects on a Linux distribution (Mac OS is fine too). The "easiest" distributions to deal with are probably Ubuntu and Linux Mint. Feel free to either dual boot or use VMWare or VirtualBox. For projects that require you to use logisim, **DO NOT USE BRANDONSIM**! We require you to use the official version of logisim. You can download that here.

Step 1 - Design an ISA

First, you must construct an ISA. Use the ideas you were exposed to when you studied the LC-3 in CS 2110 and the LC-2200 in CS 2200. Your ISA must conform to the following criterion:

1. Your instructions must be 32-bits long.
2. Since Logisim only supports a 24-bit wide memory systems, you should first make sure that the memory address is in the appropriate range, and then address the memory. You may ignore memory accesses that are out of range.
3. Your memory must contain 32-bit words. You should be able to fetch 32-bit words from memory.
4. Your ALU (Arithmetic Logical Unit) must support 32-bit word operations.
5. Your op-code field must be in the upper 6-bits of your 32-bit instruction.
6. DO NOT IMPLEMENT CONDITION CODES (ie. things like BRnzp in the LC-3). You are to implement BEQ (Branch EQual) style conditional instructions.

Furthermore, you must have support for the following instructions:

Table 1: Required Assembly Language Instructions

Name	Example Format	Action
add	add RZ, RX, RY	RZ ← RX + RY
nand	nand RZ, RX, RY	RZ ← RX NAND RY
addi	addi RZ, RX, IMM	RZ ← RX + SIGN_EXTEND(IMM)
lw	lw RZ, RX, OFF	RZ ← MEM[RX + SIGN_EXTEND(OFF)]
sw	sw RZ, RX, OFF	MEM[RX + SIGN_EXTEND(OFF)] ← RZ
beq	beq RZ, RX, OFF	If RZ == RX, then branch to PC + 1 + SIGN_EXTEND(OFF). Otherwise, go to the next instruction.
jalr	jalr RZ, RX	Store PC + 1 into RX, and then branch to the address in RZ.

IN ADDITION, YOU ARE ABSOLUTELY REQUIRED TO IMPLEMENT THREE NEW INSTRUCTIONS OF YOUR CHOOSING! Feel free to go wild and implement more than three, if you are so inclined. We have provided you with a file called `isa.txt` in which you should describe your ISA. Please follow the format that is given in the text file.

Step 2 - Build the Datapath

Now that you have described the ISA, you will now implement a datapath that conforms to the ISA! To keep track of where you are in a program, you should have a PC (Program Counter) register. For our purposes, all of our programs for the time being will start from address 0. To access memory, you should have a MAR

(Memory Address Register). You should have a register file with *at least* 16 32-bit registers. You should make your register file single-ported. This means that you can only read one value at a time from your register file. You are strongly advised that you layout your datapath in a linear fashion (as depicted in your book) - this will make your life a lot easier in the long run. Note that you may use anything in logisim to help you implement your datapth.

Bus Architecture

You may use one or more buses to drive values around your datapath. If any component wants to output a value to a bus, you must absolutely use a tri-state buffer, so as to avoid two or more devices driving the bus at the same time. The microcontroller you are eventually going to build will drive the appropriate control signals that will drive the various tri-state buffers. For now, you may want to have a mechanism that allows you to manually drive outputs of various components on the bus. It may be helpful to also have a way to drive your own values on to the bus.l

Branching

To support branching, you need to have some special hardware support. The instruction itself contains two registers and an offset. To determine equality, you must first subtract the value of the two registers (using the ALU) and then check to see if the output of the ALU is 0 (after driving the value of the ALU on to the bus, of course). If the difference is indeed 0, then you need to place the incremented PC value into the PC. If not, you may simply fetch the next instruction. If you would like, you may implement other branch instructions like BLE (Branch Less than or Equal), BGE (Branch Greater than or Equal), BGTZ (Branch Greater than Zero).

Step 3 - Microcontroller

Now, you will design a Finite State Machine (FSM) to control the flow of your processor. You will implement this using Read-Only Memory (ROM) and a state register. You will index into the ROM using the state register, and then the ROM will output the appropriate control signals as well as the next state to go to. You should hook up the next state bits to the state register, and you should hook up every control signal to their appropriate component in your datapath. It is up to you to determine what control signals you need. Here is a simplified version of the contents in your ROM:

Table 2: Simplified ROM Contents

State	Control Bits	Next State
00000	0100101010101001	00001
00001	0100101110101001	00010
00010	0100101011101001	00011
00011	0100101010111001	00100
⋮	⋮	⋮

Note that in this table, I have shown the Next State bits in a separate column. When you actually implement your ROM, you can simply add on extra bits to your control signals to denote the next state. You can retrieve the next state bits by simply getting the last few bits from the control signal bit string. When the processor is running the state machine does the following:

1. The state register's value will be used to index into the ROM.
2. The bits that are outputted by the ROM will be used to control the various devices on your datapath.
3. The last few bits outputted the ROM will be used as the input to the state register. This value will be latched on the last clock edge.

The trickiest instruction to implement is the BEQ instruction. This is because the microcontroller has to go to different states depending on the outcome of the register comparison. To handle this, you may use a separate ROM that is controlled by the result of comparing the two registers. The separate ROM will then be able to output different state numbers depending on the result of the comparison. You may choose between the state number outputted by your main ROM and the state number outputted by your secondary "BEQ" ROM by using a multiplexer. To help you keep track of the control signals, we have provided you with another text file called `fsm.txt` that has a template in which you can describe different states that correspond to different instructions. Note that you must have states that perform Instruction Fetching and Decoding.

Step 4 - Testing

Now that you have finished your microcontroller, it is time to hand assemble some instructions and load them into the memory of your processor. For every instruction you have, make a hex file that has a very, very tiny program that tests an instruction. Place your test programs for every instruction in a `.hex` file, where the file name should be the name of the instruction you are testing. If you fail to give us `.hex` file for an instruction you will be docked points.

Suggestions

1. As you are constructing your ISA, it might also be worthwhile to think about how you are going to be implementing your finite state machine.
2. When you first construct your datapath, do it in a way that allows you to manually input various control signals to your devices.
3. Have a connection from your clock to a pushbutton, so that you can manually switch between states once you implement your microcontroller.
4. Do not start making the microcontroller until you fully understand what control signals are needed to execute every instruction you have. Follow the control signals the FSM you described in `fsm.txt` to see if you are on the right track.

Deliverables

You are to turn in all the following:

1. `isa.txt`
2. `fsm.txt`
3. `firstname_lastname_processor.circ`
4. `instruction1.hex`
5. ⋮
6. `instructionN.hex`

in a file called `firstname_lastname.tar.gz`. This means that you have to use the `tar` command. Do not use any other format!

Please replace all occurrences of `firstname` and `lastname` with your actual first name and last name! You should submit an assembled file in the `.hex` format for *every single instruction*! Please name the file with the instruction being tested in the file.