Before reading this guide, please read every word of the homework specification so you will have context for what I will be discussing.

Here is a guide to HW4 that can help you build the processor in incremental steps and minimize bugs and make the entire experience relatively pain-free (but still time-consuming)! Please read the entire post if you want to follow it.

Intro, Read the Specifications Carefully!

To begin, make sure you understand the specifics of this project. To make a lot of the circuitry, you can reference the CPU lecture slides. But the slides **cannot** be followed exactly. You must understand that the lecture slides and the textbook specify a 32-bit, byte-addressed system, while you will be creating a *16*-bit *word*-addressed system (the project spec is very specific about this and mentions it several times). Make sure you understand what these differences mean for your project.

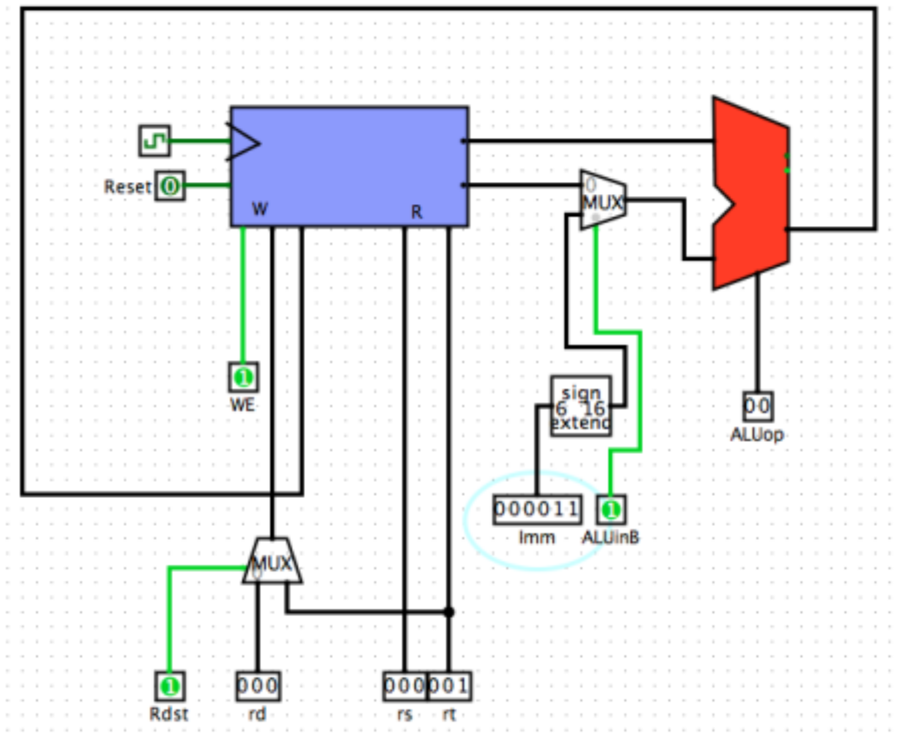
1. The Instructions on how to make theInstructions

Now, the first large phase is to implement all the instructions. You can start with the register file and ALU circuit you made for recitation 6, but you will have to make modifications. For this entire phase, remember to ***TEST EACH INSTRUCTION INCREMENTALLY!*** If you test every instruction as you create them, debugging will be much easier later. If you do not test incrementally, it will be very difficult to debug later!!!

You will want to have input pins to the read and write ports of the regfile ($rs, $rt, $rd), so you can input literal values into the pins as you test. Place the ALU functional units in the ALU and implement the "addi" instruction, referencing slides from class. Using "addi" you can now load immediates into the register file. For example, if you want to load the value "3" into register 1 (addi $r1, $r0, 3) you would set the $rt pin to be "001", $rs to be "000" and the immediate pins to be "000011".

You would also need two new control signals, "Rdst", which sets $rd to be $rt, and "ALUinB", which chooses to send the sign-extended immediate to the ALU's B operand, and set both of those signals to 1. You would also need to set the registerWriteEnable to 1, since we want to write to a register.

See as an example (this is nearly identical to slide 22):



With these signals set up, if we set the clock high and then low, “3” should be written to register 1, and you can read it out if you set either $rs or $rt to “001". Since you can now write values into the register value, you can test more instructions. Keep going down the chart of instructions in the spec and keep implementing and testing.

As you add instructions, you will have to add more control signals to the system. Keep these control signals as input pins. As you implement the instructions, you will want to maintain a spreadsheet that has instructions as the rows and control signals as the columns (see class slide for an example). This table tells you which control signals need to be on to run each instruction. As per your design, you will know which signals activate which instruction. Every instruction you add will add a new row to the table, and each instruction may or may not have several signals to control it. When you are done implementing all the instructions, your table will have 16 rows and around that many columns.

When you get to the branch and sw/lw instructions, you will have to add ROM (Read-Only-Memory, Instruction Memory) and RAM (Random-Access-Memory, Data Memory).

At the end of this phase, you should be able to run any single instruction system by turning on the appropriate control signal and register bits and clocking up and down.

Note: if you are experienced with Git, I recommend using it to version control the project. This way you can reset to an older commit if you mess up. You can checkout new branches for each instruction you create, and after testing it fully, you can merge it back into master (totally unrequired, just helpful). But do not push to Github, as that would be a violation of the Duke Community Standard :s

Note: I recommend using a custom control signal bus called ALUop (which will be a function of the instruction opcode) that controls the ALU function.

Note: Read the spec to see how to implement Instruction and Data Memory.

Note: Make sure all your muxes have “include enable” set to “no” in the object’s attributes.

Final note: The slides are all very helpful to get a basic idea of the circuits and control signals you will need for every instruction, but make sure you are implementing the instructions specified in the homework (e.g. bgt, instead of beq).

2. Controlla…

Once you have implemented all your instructions and ***EXHAUSTIVELY TESTED ALL OF THEM,*** you should now implement the control/decode unit. At this point, you should have all control signals and register file inputs as simple input pins or buses. We will now replace these pins with outputs from the control unit.

The control unit will take as an input the current instruction, and output all the control signals and register labels (e.g. $r2) that we want to read and write. The control unit will have two parts, one part that takes in the opcode as input and outputs the control signals, and a second part that takes the rest of the instruction and splits it up (literally using splitters) into rs, rt, rd, shamt, immediate, and address. For the former, you will want to make a circuit that implements the control signal table described in **Section 1**. Class slides show how to implement such a circuit, using a 4-to-16 decoder and logic gates.

Then all the input pins currently representing the control signals in your main circuit should be replaced with wires/buses output by the control/decode unit. If done correctly, each instruction will now activate the control signals for that instruction.

3. Finale Test

After doing all the above, you will have a complete system that you can now run assembly programs on. Write a few simple test programs, keeping in mind the instructions you have available and that your ALU immediates are only 6 bits. Also remember to use the “halt” command at the end of the program.

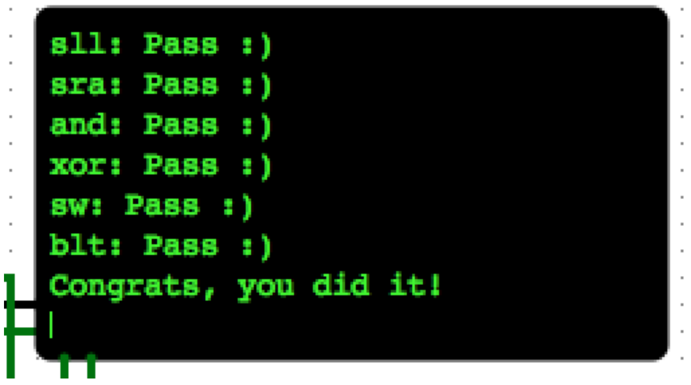
You can use the test kit (on the Teer machines, of course ☺) to assemble your programs into an .imem file, a .dmem file, and a .sim file. The .sim file you can use to see what your program does. Specifically, the output of the program on your terminal will be what your CPU ***should*** output on the TTY display. You can then load your .imem into your ROM and .dmem into your RAM (using right/two-finger/control click on the memory module and choosing **Load Image…**).

Then you can use control/cmd K to let the clock run automatically. You can choose how fast it runs in the Logisim top menu: **Simulate->Tick Frequency**.

The CPU will then output onto the TTY (make sure you write a program that uses the Output instruction). If the output matches the terminal output from the simulator, then the CPU runs that program correctly!

The holy grail program is one that tests all 16 instructions and outputs confirmation messages onto the TTY. Think about how you could use branches to check what is expected to be in a register based on the result of some test instruction, and then branch to a print function to print out a message based on whether the tested instruction ran correctly.

Like this:



```
sll: Pass :)
sra: Pass :)
and: Pass :)
xor: Pass :)
sw: Pass :)
blt: Pass :)
Congrats, you did it!
```

Of course, you can always use the python tester we give you, but remember those tests will not be exhaustive. Your CPU should be able to run arbitrary programs using all 16 instructions, so do your best to make sure it does that.

That's everything (whew). You have almost an entire month to finish, but I suggest you get started now! If you follow this guide, you should have few problems, the hardest part will be implementing the instructions correctly.

Additionally, please read the following random notes that are based on common issues that are seen:

- Have the **PC Register** and **Keyboard** clocked on the rising edge.
- Have the **RAM**, **TTY Display**, and **Register File** clocked on the falling edge. You can do this in the module's attributes or by just inverting the clock input to that module.
- You must also have no other probes other than the ones specified for the register file (r0 ... r7)
- The only modules that should be reset by the global reset are the **PC Register, Register File**, **TTY Display**, and **Keyboard**. *You should **not** reset the RAM.*
- The write enable, reset, and clock should go directly into the enable, reset, and clock inputs of all the D Flip Flops in your system, i.e. in the register file and the PC register.

You should not AND the write enable and clock together at any point, the DFFs already handle this for you.

- The reset must be asynchronous, meaning you should not have to tick the clock to reset, it should occur as soon as reset goes high.
- The reset must be a single input pin in your main circuit labeled "reset" (lowercase 'r').
- The only connected pins on the RAM should be both "D's", "A". "str" and the clock. "ld" should be left unconnected.
- Make sure the attributes on your clock source has common sense values, such as 1 low and high tick duration.
- If you want to do independent testing with your own programs, make sure to start the clock high at the very beginning of the run.

Note: To log in to the Teer machines off campus, you must the Duke VPN for which you can download the Cisco client at https://oit.duke.edu/what-we-do/services/vpn .

Good luck!