

# CSE 141L Lab #3: 8-bit CPU – Reg File, Decoder, and Preliminary Pipeline Diagram

*due Friday, February 20*

In this lab assignment, you will design the register file and the decoder (control logic) for your CPU. In order to design the decoder, you need to know what all your control signals are going to be – and to do that, you need to know what your pipeline looks like.

So the first thing you will do will be to create an overall pipeline design/strategy. What I want you to turn in is a picture analogous to Figure 6.22 in the textbook (not a Xilinx schematic, just a drawing). From that picture, we can tell how many pipeline stages you have, what you do in each stage, and what all your control signals are. The decision of how deep to make your pipeline and what to put in each stage is not an easy one (if you are concerned about performance). However, since you already have designed your ALU and fetch unit, and can have preliminary designs of your register file and decoder, you can obtain timing information for each of those components and make a fairly intelligent design choice. Assume data memory will have similar delay to your instruction ROM.

You will also design the register file (or whatever internal storage your ISA specifies). The register file will have a clock input and a write-enable input (e.g., RegWrite from the text). It will have one data input (you can only write one register per cycle), and as many outputs as you need, and of course a register specifier for each data input and output. Many of you make heavy use of implicit registers (e.g., add always reads registers 2 and 3). In that case, the decoder/control logic will be responsible for generating the read-register specifiers – that keeps your register file simple. There is one major twist -- the register file must correctly handle the case where the same register is read and written in a cycle. In this case, the read register (or registers) should supply the new value being written (this is called “forwarding”). There are two ways to handle this, with different design and performance implications.

- Method 1. Always write registers on the falling edge of the clock (all other devices in your design will latch on the rising edge of the clock). In this way, the new value will be available to the output in the second half of the clock. The textbook assumes this design.
- Method 2. Include explicit forwarding logic inside the register file. If the output register is the same as the input register (and RegWrite is high!), route the input value to the output. Otherwise, route the appropriate register value as usual.

You will turn in a high-level schematic for the register file (probably just a single module in a Xilinx schematic), and all internal verilog or schematics. You will test the register file fully (including forwarding) and turn in the timing diagrams.

After you have created your pipeline diagram, you will know what all of your control signals are, allowing you to build your decoder. The decoder is combinational logic. It's only input is your instruction opcode. The output is the full set of control signals for that instruction. If operand specifiers (either register names or memory addresses) are implicit in the opcode, then the decode logic must also produce those specifiers (either directly or possibly via a lookup table).

Unless you specifically copied the MIPS use of function code to supplement the opcode, there is no need to do decoding in two stages like we see in the book (i.e., ALU control separate from the control logic). But if you do, show and test both parts on a single schematic. For the decoder, then, please turn in the following three things. (1) A table analogous to Figure 6.25 from the book (your table is likely to be larger) for your ISA and design. (2) The high-level schematic and all internal design specifications. (3) Timing diagrams demonstrating correct operation.

No other parts of your design are due at this time; however, putting together the final design will be a major effort. If you want to get ahead, think about working ahead on a few components you'll need for the last lab – multiplexers, the forwarding control unit, bubble insertion logic.

The lab report will contain the following:

1. Introduction and general comments. Overview of report.
2. Summarize your ISA from Lab 1 (operations supported, with full detailed descriptions).
3. The overall pipeline diagram. This should clearly identify all pipeline stages and the logic within them, as well as all control signals.
4. Full schematics, hierarchically organized. Also include the table for your control signals.
5. Timing diagrams. It should be clear everything works. If your presentation leaves doubt, we'll assume it doesn't.

Answer the following questions:

6. How many pipeline stages do you support? What happens in each stage? Explain why/how you made those decisions – what were you optimizing for?
7. Describe *all* data hazards in your pipeline that require forwarding.
8. Describe all data hazards in your pipeline that require bubbles/stalling (including how many stall cycles are needed). Give an estimate of the likely overall performance impact of these bubbles (e.g., what is your expected CPI?).
9. Describe how you handle control hazards in your design. What is the expected performance impact of control hazards?
10. Describe how you expect to handle forwarding in your final design (apart from the forwarding in the register file you've already done).
11. Describe how you handle forwarding in the register file. Why did you choose the method you did? How would your performance be different if you had chosen the other option?
12. Name one thing you could have done differently in your ISA design to make your decoder design easier.
13. Name one thing you could have done differently in your ISA design to make your register file design easier.
14. Which pipeline stage do you expect to take the longest?
15. If you were unhappy with the performance of your current design, what (hardware) change would you make to improve performance? How much improvement would you expect?