E85: Digital Electronics and Computer Engineering
Lab 11: Multicycle Processor

Objective

In this lab, you will complete a multicycle RISC-V processor.

1. Multicycle RISC-V Processor

Figure 1 shows the complete multicycle processor.

Figure 2 (near the end of this lab) shows the high-level hierarchy of the single-cycle processor including the connections between the controller, datapath, instruction memory, and data memory. Your multi-cycle processor has only a single unified memory and has slightly different control signals, so you will need to modify these connections. Sketch a diagram similar to Figure 2 showing your controller, datapath, and memory modules. Draw a box for the `riscv` module that should encompass the controller and datapath. Label the signals passing between blocks.

Write a hierarchical Verilog description of the processor. The processor should have the following module declaration. The memory signals are tapped out for testing purposes. Use your controller from Lab 10 and any general Verilog building blocks you need (e.g., muxes, flops, adders, ALU, register file, immediate extender, etc.) from the single-cycle processor. NOTE: The controller microarchitecture for the ResultSrc mux has changed slightly from that shown in Lab 10 so you will need to change your ResultSrc encoding: 10 and 00 have been swapped. See Figure 3 for an updated Main FSM with changes highlighted. The testbench for Lab 10 on the website has been updated so you can double check that your updated controller is working. The single-cycle processor code (`RISCVsingle.sv`) is on the class web site and you may wish to cut and paste blocks from it.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);
```

2. Test Bench

The riscv_testbench.sv and test code (in assembly .asm and machine language .dat) are on the class web page. Study the test bench to understand how it determines if your tests succeeded or failed.

Your memory should read the test code from the memory file at startup with the line:

```
initial $readmemh("memfile.dat", RAM);
```

Before you begin simulation, predict what the processor should do while executing the first three instructions. Table 1 has been filled out for you for the first instruction.

Generate simulation waveforms at least for clk, reset, PC, Instr, state, SrcA, SrcB, ALUResult, Adr, WriteData, and MemWrite. Display the 32-bit signals in hexadecimal for ease of reading (select the signals and right click, then choose Radix). Compare against your expectations. You may wish to add other signals to help debug. Fix any problems you may find until your code executes the program as expected and the testbench reports success.

Refer to the previous lab for debugging hints. Fix all relevant warnings from Quartus and Modelsim before you debug further. It will save you much time to carefully predict what each of the signals in your waveforms should be doing on each cycle, and to systematically debug beginning with the first known discrepancy and working your way backward until you have good inputs and bad outputs and have isolated the bug.

Particularly common bugs include:

- Copying the single-cycle processor top, riscv, or datapath interface with signals that don't match the multi-cycle processor. Be sure you have a clear idea what belongs in each module. You will likely save time if you sketch a picture similar to Figure 2 and identify what signals flow between the riscv and memory modules.
- Connecting signals in different orders in a module declaration vs. in the instantiation.
- Forgetting to declare internal signals, or giving them the wrong widths.
- Inconsistent capitalization or spelling.

If you've checked these and your processor still isn't working, try adding all the outputs of the controller to your sim and make sure none are floating or X. If you still haven't found the problem, refer to your predicted waveforms in Table 1 and check that the processor is doing the right thing on each step. If the first few instructions are correct, you may need to extend the table to predict what the rest of the program should be doing. (Once you've filled out the table for several instructions, you may get the hang of the pattern and only fill out entries that are interesting…)

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Diagram showing your memory, riscv, datapath, and controller block hierarchy and names of all signals between them.
3. Hierarchical SystemVerilog for your top-level processor module (and submodules) matching the declaration given above.

4. Table 1 showing key signals for at least the first three instructions.
5. Simulation waveforms (in the order listed above) at least for the specified signals. Does your system pass your testbench? Circle or highlight the waves showing that the correct value is written to the correct address, and make sure it is legible.

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.

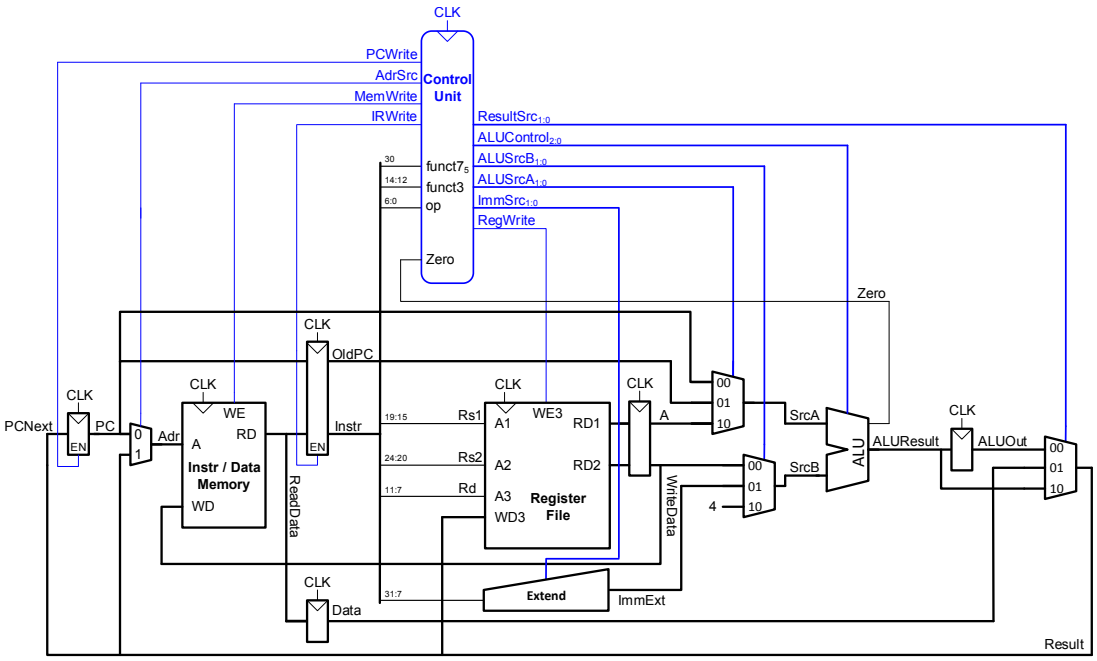


Figure 1: Complete multicycle processor

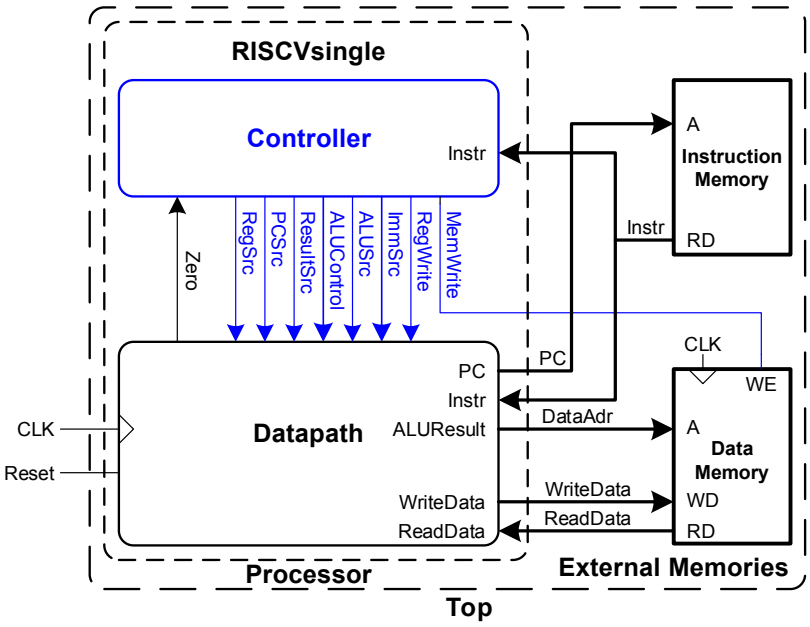
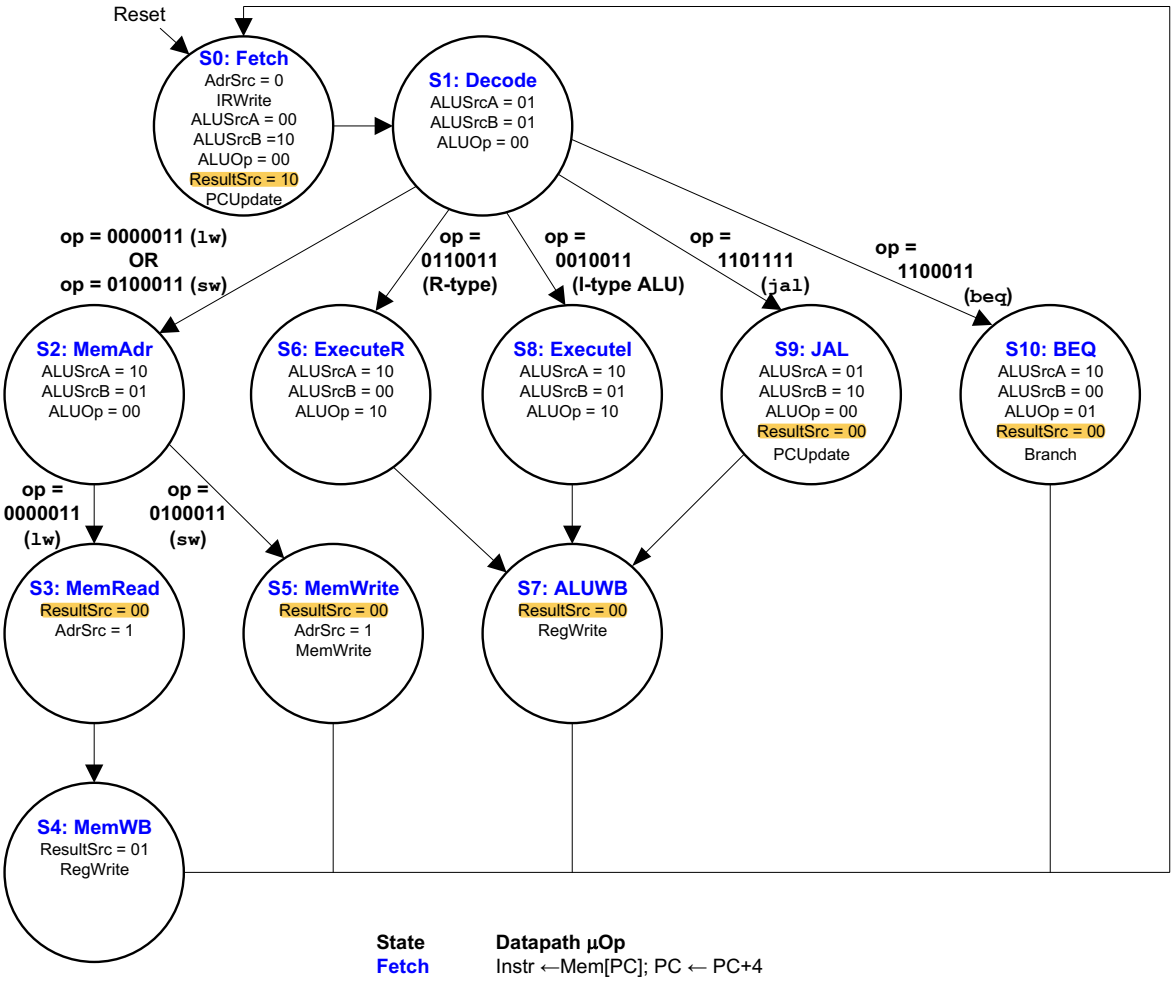


Figure 2: Single-cycle processor interfaced to external memories

Table 1: Expected Operation (after two cycles of reset)

Step	PC	Instr	State	Result	Result Notes
3	00	00500113	S0: Fetch	4	PC+4
4	04	""	S1: Decode	X	OldPC+Immediate
5	04	""	S8: ExecuteI	X	ALUResult = x0 (0) + 5 = 5
6	04	""	S7: ALUWB	5	Result = ALUOUT
7	04	""	S0: Fetch	8	PC+4
8	08	00c00193	S1: Decode	X	OldPC+Immediate
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42					
43					

(Only fill in the notes column if it is helpful to you to understand what is happening and why.)



State	Datapath μOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Fig 3. Complete multicycle control Main FSM state diagram