Lab #8: RISC-V Multicycle Processor Control

0. GOAL

In Labs 8 and 9, you will design a multicycle RISC-V processor in SystemVerilog and test it on a simple machine language program. In these labs, you will build (in SystemVerilog) and simulate the processor only (no hardware implementation on the DE2-115 board). This will tie together everything that you have learned about digital design, hardware description languages, assembly language, and microarchitecture, and give you the chance to design and debug a complex system. In this lab (Lab 8), you will build and test the controller. In Lab 9, you will build the datapath and test the whole system.

1. MULTICYCLE RISC-V CONTROLLER

Before you start developing the controller, make sure to take a look at the following diagrams. All figures and tables are provided at the end of this document.

- Multicycle controller block diagram
- Multicycle control Main FSM state diagram
- HDL Example 1: ALU Decoder logic
- HDL Example 2: Instruction Decoder logic

Write a hierarchical SystemVerilog description of the multicycle controller. When outputs are don't cares, set them to 0 so they have a deterministic value to simplify testing.

The controller should have the following module declaration and should follow the hierarchy of Figure 1. Remember that `op`, `funct3`, `funct7b5`, are bitfields of Instr and that `zero` is an output of the ALU.

```
module controller(input   logic       clk,
                  input   logic       reset,
                  input   logic [6:0] op,
                  input   logic [2:0] funct3,
                  input   logic       funct7b5,
                  input   logic       zero,
                  output  logic [1:0] immsrc,
                  output  logic [1:0] alusrca, alusrcb,
                  output  logic [1:0] resultsrc,
                  output  logic       adrsrc,
                  output  logic [2:0] alucontrol,
                  output  logic       irwrite, pcwrite,
                  output  logic       regwrite, memwrite);
```

2. TESTBENCH

Generating good test vectors is often harder than writing the code you are testing. This semester, the vectors are provided for you to increase the amount of sleep you'll get. Get the `controller_testbench.sv` and `controller.tv` from the class website. Read them and understand what they are doing.

Compile and test your controller with Modelsim. Make sure you run for long enough to get a message that all of the tests were completed with 0 errors.

3. DEBUGGING HINTS

Unless you are extraordinary unlucky, your controller won't work perfectly on the first try. If it did work, you would have missed out on the main learning objective of this lab and the next, which is how to systematically debug a complex system. You will need your controller in Lab 9, so take the time to fully debug.

Here are some tips to reduce the amount of time that debugging will take.

Minimize the number of bugs you have

Each bug takes a long time to locate, so a bit of extra time during the design phase can save you a lot of time during the debug phase.

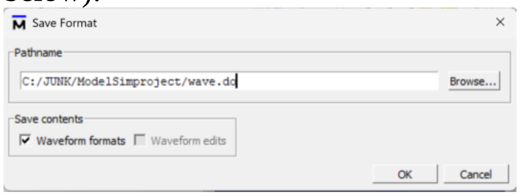
- Remember that you are building hardware, so sketch the hardware you want and write the SystemVerilog idioms that imply that hardware. Don't fall into the trap of writing SystemVerilog code without thinking of the hardware it is implying.
- Proofread your code. Make sure your signal names are spelled consistently and that module inputs/outputs are listed in the correct order.
- Simulate your design with Modelsim and look for warnings and errors when compiling. Take the warnings (as well as the errors) seriously.

Minimize the time it takes to run a test

Once you are in the debugging phase, choose a workflow that is efficient so you can make a change to your code and rerun the test in a matter of seconds rather than minutes.

- All testing will be done in Modelsim.
- Add relevant waveforms in Modelsim. It's usually worthwhile to add all the signals in a module that you are debugging so that you don't have to go through the tedious process of adding more signals and re-simulating. Change the radix to display 32-bit signals in hexadecimal.
- Remember that you don't need to restart Modelsim and re-add signals each time you change your code. Instead:
  - Compile -> Compile All
  - Make sure you have no warnings
  - At the command line, rerun the simulation by typing
    - `restart -f`
    - `run 1000` (or however long you wish to run)

- So that you don't have to dig through the hierarchy and re-add the signals, you can save a wave.do file that contains the signals of interest (resave it again if you later add more signals). To do so:
    - Start simulating your testbench.
    - Add signals to the wave window.
    - Then, click on File -> Save Format (or Ctrl-s), and save the wave.do (see figure below):



    - Then, you can reload those signals (for example, after closing Modelsim or after ending the simulation), by clicking on File -> Load Macro File (and then select the wave.do file, where ever you have saved it). Do this step after starting the simulation.
- Remember that if you close a pane (for example the "Wave" pane), you can view it again by clicking on View -> Wave (or whichever pane you've closed).

Systematically find your bugs

Inexperienced designers can waste enormous amounts of time debugging without a clear plan in mind. The following techniques can save you many hours.

- Understand what the expected inputs and outputs should be. Write down your expectations. This takes time but will usually save far more time than it takes.
- Find the first place where a signal doesn't match your expectations. One bad signal will usually trigger others downstream, so focus your debugging on the first known error and don't worry yet about subsequent errors. For example, if tests 1 and 5 fail, start debugging test 1, not test 5.
- Make sure the simulator displays all signals involved in computing the bad signal. If necessary, add them to the simulation and re-simulate as given above. If one of these inputs is bad, repeat this process to continue tracing it back.
- Once all the inputs are good and the output is bad, you've localized your bug. Examine the relevant SystemVerilog module and fix the mistake.
- Repeat this process until all bugs have been fixed.

Particularly common bugs include:

- 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.

4. WHAT TO TURN IN

Total points available: 10

1. **[1 pt]** 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. **[6 pts]** Hierarchical SystemVerilog for your controller module matching the declaration given above.
3. **[3 pts]** Does your controller pass your test vectors?

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

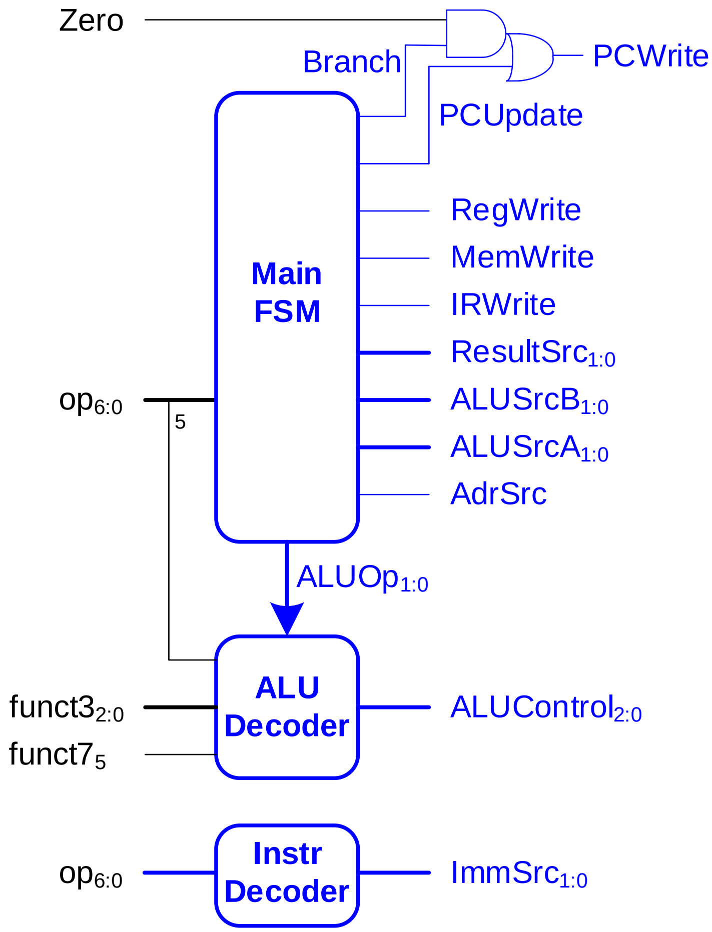
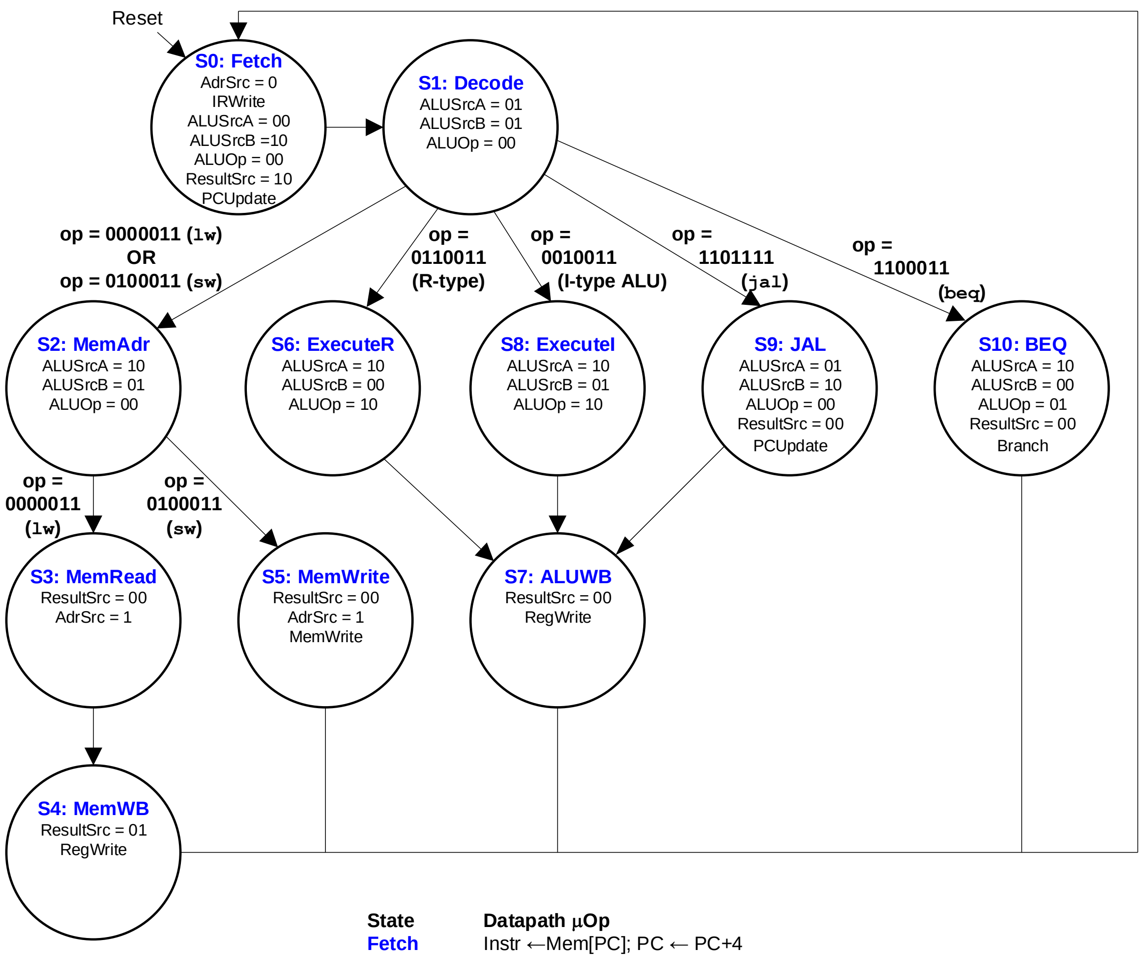


Figure 1. Multicycle control unit



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

Figure 2. Complete multicycle control Main FSM state diagram

ALU Decoder Truth Table:

ALUOp	funct3	op_5 , $funct7_5$	Instruction	$ALUControl_{2:0}$
00	x	x	**lw, sw**	000 (add)
01	x	x	**beq**	001 (subtract)
10	000	00, 01, 10	**add**	000 (add)
	000	11	**sub**	001 (subtract)
	010	x	**slt**	101 (set less than)
	110	x	**or**	011 (or)
	111	x	**and**	010 (and)

```
module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 3'b000; // addition
      2'b01:                ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
                            ALUControl = 3'b000; // add, addi
                 3'b010:    ALUControl = 3'b101; // slt, slti
                 3'b110:    ALUControl = 3'b011; // or, ori
                 3'b111:    ALUControl = 3'b010; // and, andi
                 default:   ALUControl = 3'bxxx; // ???
               endcase
    endcase
endmodule
```

HDL Example 1. ALU Decoder

Instruction Decoder TruthTable:

op	op (binary)	Instruction	ImmSrc
3	`000 0011`	`lw`	00
35	`010 0011`	`sw`	01
51	`011 0011`	**R-type**	xx
99	`110 0011`	`beq`	10
19	`001 0011`	**I-type**	00
111	`110 1111`	`jal`	11