Lab 12: Multicycle Processor Controller

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

Objective

In Labs 11 and 12, you will design a multicycle RISC-V processor in SystemVerilog and test it on a simple machine language program. This will tie together everything that you have learned in this textbook about digital design, hardware description languages, assembly language, and microarchitecture, and give you the chance to design and debug a complex system. In Lab 10, you will build and test the controller. In Lab 12, you will build the datapath and test the whole system.

1. Multi-cycle 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.

- Figure 1 shows the multicycle controller block diagram
- Figure 2 shows the multicycle control Main FSM state diagram
- Table 1 and HDL Example 1 define the ALU Decoder logic
- Table 2 and HDL Example 2 define the Instruction Decoder logic

Write a hierarchical SystemVerilog description of the multicycle controller. When outputs are don't care, 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. Test Bench

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 12, 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.
- Synthesize your design once in Quartus and look for warnings or errors. Make sure you understand which warnings are normal (e.g. no timing constraints set) and which need to be fixed. Take these warnings very seriously; they are the fastest way to detect subtle bugs in your design.
- Simulate your design with Modelsim and look for warnings when compiling. Modelsim has a different SystemVerilog analyzer and will detect types of mistakes that don't produce warnings in Quartus. Take these warnings seriously too.

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 can be done in Modelsim. You do not need to use Quartus, and recompiling in Quartus is an unnecessary time-consuming step. However, if you have made major changes, you might wish to occasionally resynthesize the design in Quartus and look for warnings hinting that you've introduced new bugs.

- 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)

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.

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. Hierarchical SystemVerilog for your controller module matching the declaration given above.
3. 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.

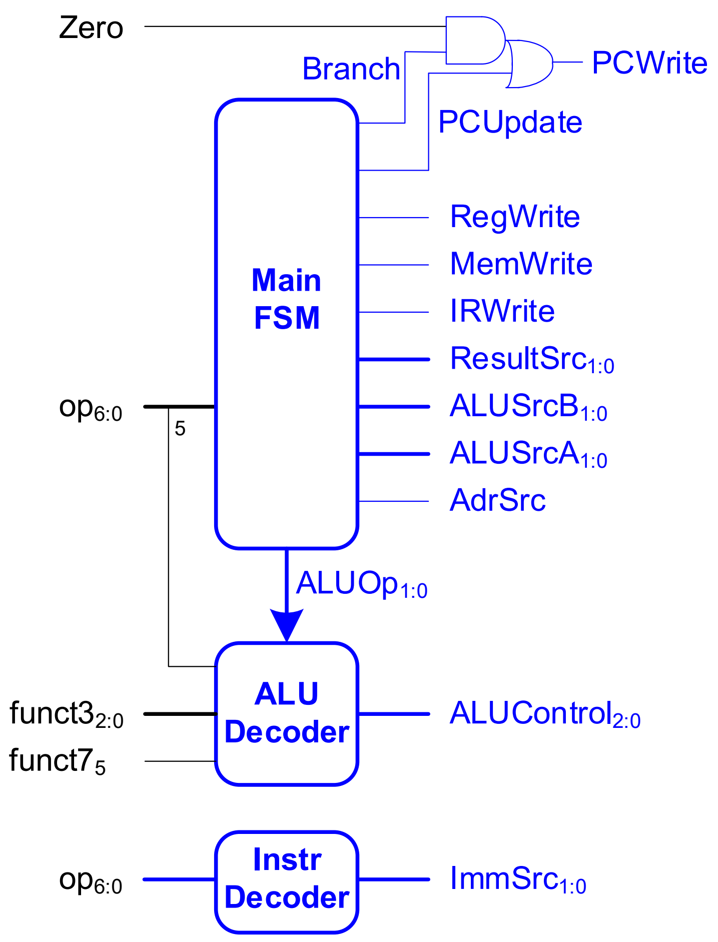
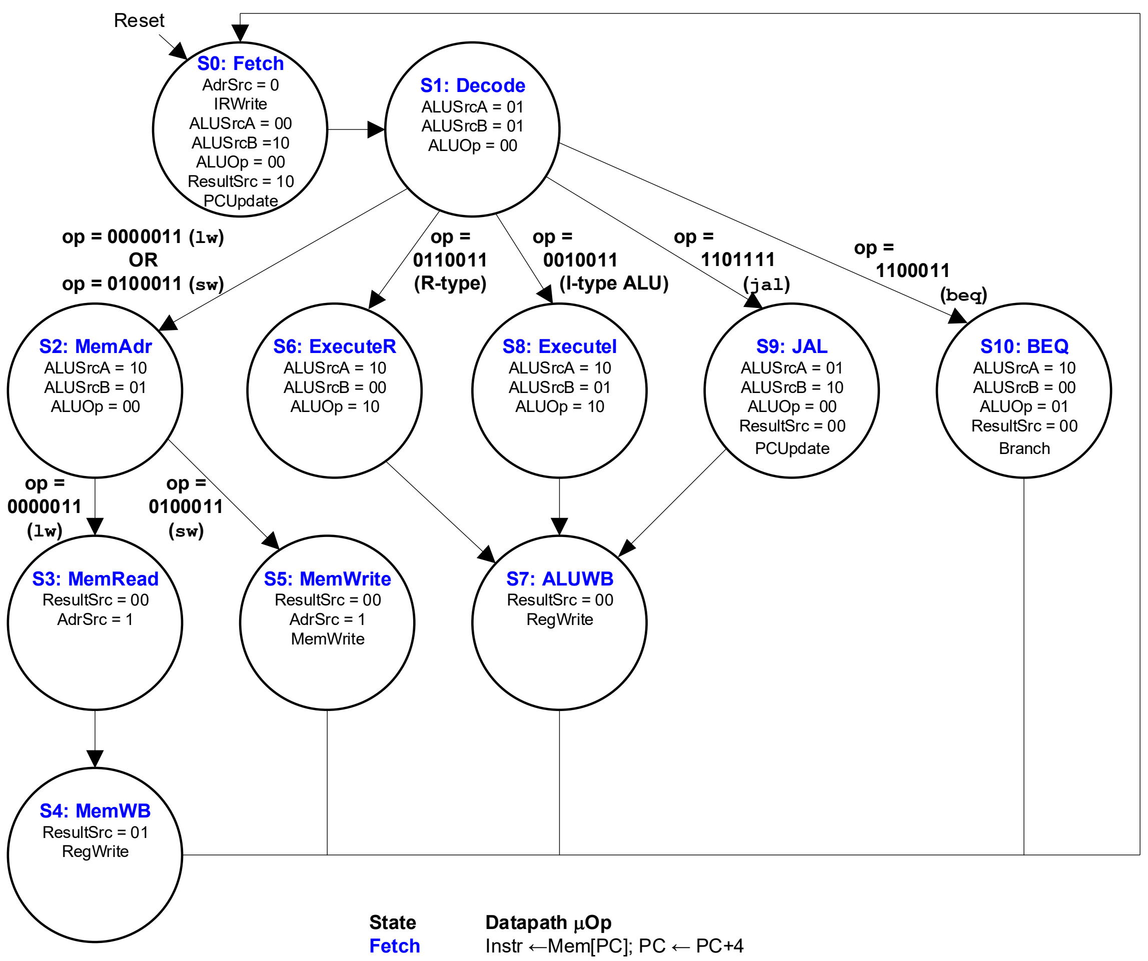


Figure 1. Multicycle control unit



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

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)

Table 1. ALU Decoder logic

```
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	Opcode (*op*)	$ImmSrc_{1:0}$
R-type	0110011	XX
I-type	0010011	00
lw	0000011	00
sw	0100011	01
beq	1100011	10
jal	1101111	11

Table 2. Instr Decoder logic for *ImmSrc*

```
module instrdec (input  logic [6:0] op,
                 output logic [1:0] ImmSrc);
  always_comb
    case(op)
      7'b0110011: ImmSrc = 2'bxx; // R-type
      7'b0010011: ImmSrc = 2'b00; // I-type ALU
      7'b0000011: ImmSrc = 2'b00; // lw
      7'b0100011: ImmSrc = 2'b01; // sw
      7'b1100011: ImmSrc = 2'b10; // beq
      7'b1101111: ImmSrc = 2'b11; // jal
      default:    ImmSrc = 2'bxx; // ???
    endcase
endmodule
```

HDL Example 2. Instruction Decoder