# Lab #5: RISC-V Multi-Cycle Processor

# Introduction

In this lab, you will design and build your own multicycle RISC-V processor. You may reuse any of your hardware (Verilog modules) from previous labs.

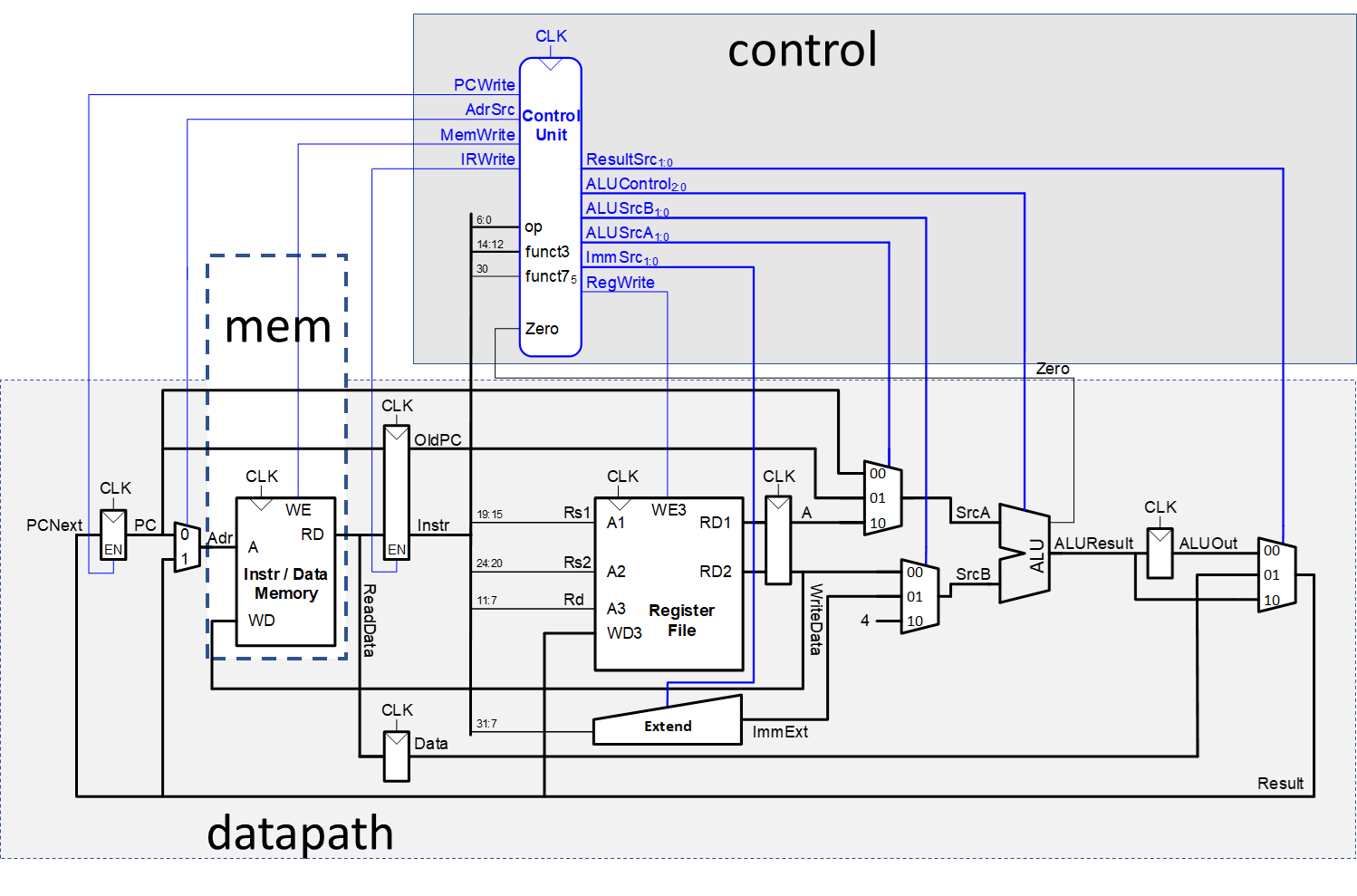
Your multicycle processor should match the design from the textbook, which is reprinted in Figure 1 for your convenience. It should handle the following instructions: add, addi, sub, and, andi, or, ori, slt, slti, lw, sw, beq, jal, and lui. The multicycle processor is divided into three units: controller, datapath, and mem (memory) units. Note that the mem unit holds both data and instructions at addresses matching actual RISC-V data and text segments. Therefore, the PC now starts at pc\_start = 0x10000, and data writes have destinations past 0x10000000.

# Getting Started

# Before you begin developing the hardware for your RISC-V multicycle processor, you’ll need to determine the correct control signals for each state in the multicycle processor’s state transition diagram. This state transition diagram is shown in Figure 7.45 of the textbook and reprinted in Figure 2 for convenience. Note that ALUControl and ImmSrc signals are generated with combinational logic, as shown in Figure 3, similarly to single-cycle processor except that jal now uses ALU unit for JTA calculation. Complete the output table of the Main Decoder in Table 1 at the end of this handout. The first two rows are filled in as examples. Be careful with this step. It takes much longer to debug an erroneous circuit than to design it correctly the first time. Also note that you will need to implement the instructions lui, which is not described in the textbook state transition diagram. Expand the state transition diagram to match your implementation of the controller that supports lui.

# Control Unit Design

The implemented design has flip-flops for every control value produced by the main FSM. This allows to remove some control signals from critical path. So, in each cycle, the main FSM computes next state \*and\* also control signals for the next state. (The alternative implementation is to generate control signals from the FSM state.) The next state and its control signals are determined by which state the processor is in, as well as op code, and reset signal. Control unit is already provided and you need to only modify portion of the code by replacing all “z” values with correct one for the FSM state transition and control signal generation, using Figure 2 and Table 1 as a reference. Be sure to use the state labels from “ucsbece154a\_defines.vh”.

****

**Figure 1. Multicycle Processor**

**Datapath Design**

After finishing the controller, you should begin implementing the datapath. Note that there are 7 sets of flip-flops in the datapath as opposed to just 1 in single-cycle design. Those are already provided in the code. Be sure to use the labels from “ucsbece154a\_defines.vh” in your mux implementations for readability. If you do not, you will lose points.

Refer to Figure 1 for the hardware modules you need to set up your datapath. Design the datapath unit in Verilog. Remember that you may reuse hardware from earlier labs (such as the ALU, multiplexers, registers, Immediate Extension hardware modules, register file, etc.) wherever possible. Pay careful attention to bus connections; they are an easy place to make mistakes.

Simulate your processor using the provided testbench. The Reset signal is set high at first. Display, at a minimum, the *PC*, *Instr*, FSM *state* (from within your controller module), *SrcA*, *SrcB, and ALUResult* (from within your datapath). You will likely want to add other signals to help debug. Check that your results match the expectations from Table 2. If there are any mismatches, debug your design and fix the errors.

# Hints for Debugging

* Be sure you thoroughly understand how the RISC-V multicycle processor is supposed to work. This system is too complex to debug by trial and error. You should be able to predict what value every signal should be at every point in time while debugging.
* In general, trace problems by finding the first point in a simulation where a signal has an incorrect value. Don’t worry about later problems because they could have been caused by the first error. Identify which circuit element is producing the bad output and add all its inputs to the simulation. Repeat until you have isolated the problem.

**Test Program**

* Use the test program “text.dat”, corresponding to the included text.s assembly file, which is similar to memfile.dat from Lab 4 but with few added instruction to match the new data segment locations.
* As in Lab 4, it is very helpful to first predict the results of a test program before running the program so that you know what to expect and can discover and track down discrepancies. Table 2, which is partially completed, lists the expected instruction trace while running the test program. Complete the table up until at least 28 cycles. Do this before you run simulations so you have a set of expectations to check your results against; otherwise, it is easy to fool yourself into believing that erroneous simulations are correct. You may specify x (don’t care) instead of an actual value, whenever the computed values are discarded and not used for instruction executuion.
* Notice that the instruction (instr) is fetched during state 0 and therefore not updated until state 1 of each instruction.

**What to Turn In**

1. Submit your RTL files to “lab5 RTL” on Gradescope.
2. Submit a single PDF file “lab5 Report” on Gradescope containing items below, clearly labeled and in the following order. Poorly organized submissions will lose points. Note that you do not need to attach Verilog files to your lab report.
   1. Please indicate how many hours you spent on this lab. This will not affect your grade, but will be helpful for calibrating the workload for next semester’s labs.
   2. A completed Main Decoder output table (Table 1).
   3. The first 28 cycles of Table 2 indicating the expected outcome of running the test program.
   4. Screenshots of the entire waveform demonstrating the controller module by providing (in the given order): CLK, Reset, PC, Inst, the state (this is an internal registered signal), and ALUControl. **Unreadable waveforms will receive no credit.** Display all signals in hexadecimal and make sure they are readable. Does it match your expectations?

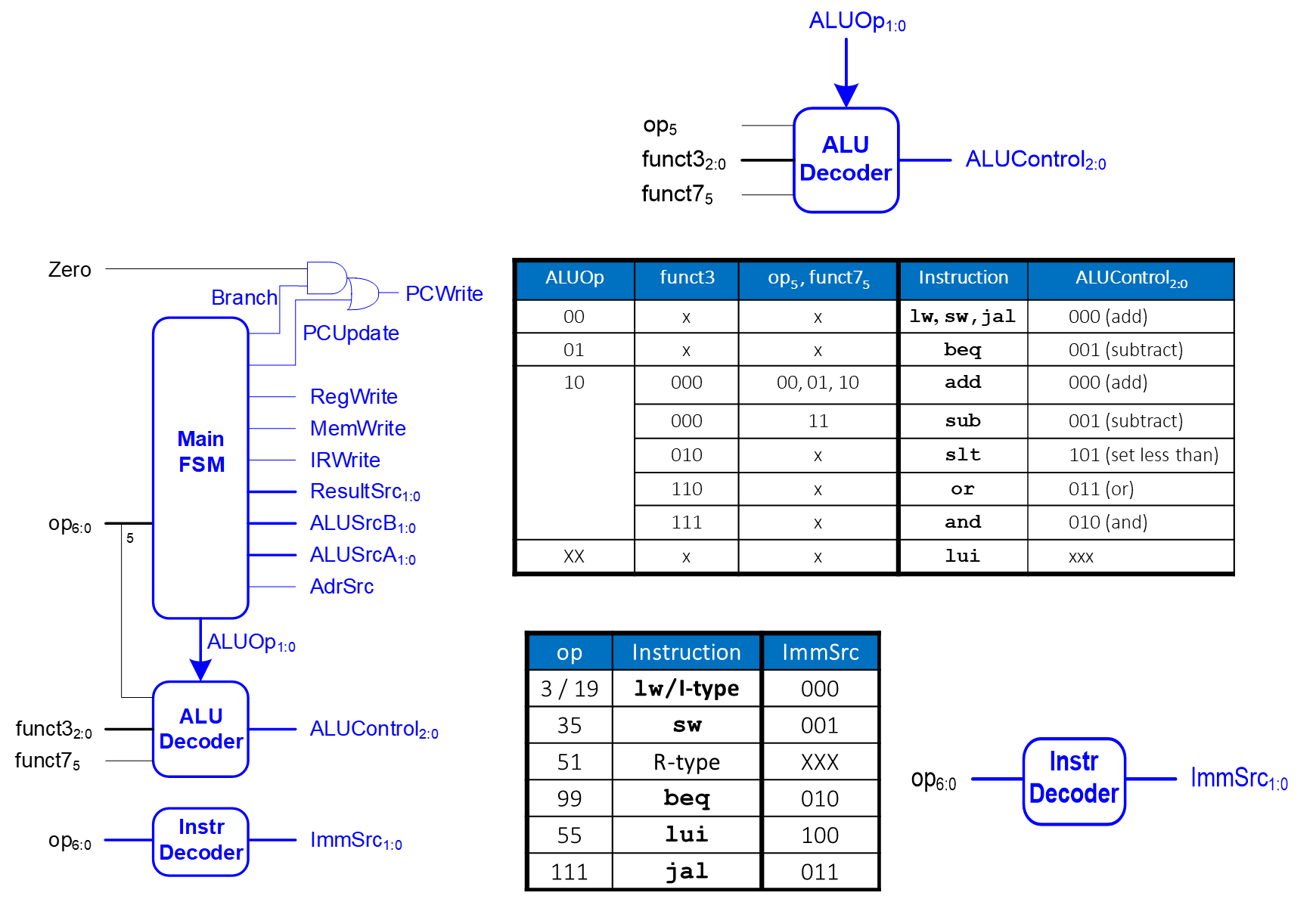
Side note, submitted designs will be synthesized for the [iCE40HX-8K CT256 FPGA](https://www.latticestore.com/products/tabid/417/categoryid/9/productid/355/searchid/1/default.aspx) and ranked on a leaderboard according to “Logic Cell Count” and “CLK Frequency (MHz)”. Your ranking has no effect on your grade, but it can give you a good idea of how your design compares to others’.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State  (Name) | | PCUpdate | MemWrite | IRWrite | RegWrite | ALUSrcA[1:0] | Branch | AdrSrc | ALUSrcB[1:0] | ResultSrc[1:0] | ALUOp[1:0] | ImmSrc[2:0] |
| 0 (Fetch) | | 1 | 0 | 1 | 0 | 00 | X | 0 | 10 | 10 | 00 | XXX |
| 1 (Decode) | | 0 | 0 | 0 | 0 | 01 | 0 | X | 01 | XX | 00 | 010 |
| 2 (MemAdr) | |  |  |  |  |  |  |  |  |  |  |  |
| 3 (MemRead) | |  |  |  |  |  |  |  |  |  |  |  |
| 4 (MemWB) | |  |  |  |  |  |  |  |  |  |  |  |
| 5 (MemWrite) | |  |  |  |  |  |  |  |  |  |  |  |
| 6 (ExecuteR) | |  |  |  |  |  |  |  |  |  |  |  |
| 7 (ALUWB) | |  |  |  |  |  |  |  |  |  |  |  |
| 8 (ExecuteI) | |  |  |  |  |  |  |  |  |  |  |  |
| 9 (JAL) | |  |  |  |  |  |  |  |  |  |  |  |
| 10 (BEQ) |  |  |  |  |  |  |  |  |  |  |  |
| 11 (LUI) |  |  |  |  |  |  |  |  |  |  |  |

**Table 1. Main Decoder Control output**



**Figure 2. Controller’s FSM**

****

**Figure 3. Controller combinational logic**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Cycle** | **Reset** | **PC** | **Instr** | **(FSM) state** | **SrcA** | **SrcB** | **ALUResult** | **Zero** |
| 1 | 1 | 10000 | xxxxxxxx | Fetch | 10000 | 4 | 10004 | 0 |
| 2 | 0 | 10004 | addi 00500113 | Decode | 10004 | x | x | x |
| 3 | 0 | 10004 | addi 00500113 | ExecuteI | 0 | 5 | 5 | 0 |
| 4 | 0 | 10004 | addi 00500113 | ALUWB | x | x | x | x |
| 5 | 0 | 10004 | addi 00500113 | Fetch | 10004 | 4 | 10008 | 0 |
| 6 | 0 | 10008 | addi 00C00193 | Decode | 10008 | x | x | x |
| 7 | 0 | 10008 | addi 00C00193 | ExecuteI | 0 | c | c | 0 |
| 8 | 0 | 10008 | addi 00C00193 | ALUWB | x | x | x | x |
| 9 | 0 |  |  |  |  |  |  |  |
| 10 | 0 |  |  |  |  |  |  |  |
| 11 | 0 |  |  |  |  |  |  |  |
| 12 | 0 |  |  |  |  |  |  |  |
| 13 | 0 |  |  |  |  |  |  |  |
| 14 | 0 |  |  |  |  |  |  |  |
| 15 | 0 |  |  |  |  |  |  |  |
| 16 | 0 |  |  |  |  |  |  |  |
| 17 | 0 |  |  |  |  |  |  |  |
| 18 | 0 |  |  |  |  |  |  |  |
| 19 | 0 |  |  |  |  |  |  |  |
| 20 | 0 |  |  |  |  |  |  |  |
| 21 | 0 |  |  |  |  |  |  |  |
| 22 | 0 |  |  |  |  |  |  |  |
| 23 | 0 |  |  |  |  |  |  |  |
| 24 | 0 |  |  |  |  |  |  |  |
| 25 | 0 | 10018 | add 004282B3 | Fetch | 10018 | 4 | 1001c | 0 |
| 26 | 0 |  |  |  |  |  |  |  |
| 27 | 0 |  |  |  |  |  |  |  |
| 28 | 0 |  |  |  |  |  |  |  |
| 29 | 0 |  |  |  |  |  |  |  |
| 30 | 0 |  |  |  |  |  |  |  |
| 31 | 0 |  |  |  |  |  |  |  |
| 32 | 0 |  |  |  |  |  |  |  |
| 33 | 0 |  |  |  |  |  |  |  |
| 34 | 0 |  |  |  |  |  |  |  |
| 35 | 0 |  |  |  |  |  |  |  |
| 36 | 0 |  |  |  |  |  |  |  |
| 37 | 0 |  |  |  |  |  |  |  |
| 38 | 0 |  |  |  |  |  |  |  |
| 39 | 0 |  |  |  |  |  |  |  |
| 40 | 0 |  |  |  |  |  |  |  |
| 41 | 0 |  |  |  |  |  |  |  |
| 42 | 0 |  |  |  |  |  |  |  |
| 43 | 0 |  |  |  |  |  |  |  |
| 44 | 0 |  |  |  |  |  |  |  |
| 45 | 0 |  |  |  |  |  |  |  |
| 46 | 0 |  |  |  |  |  |  |  |
| 47 | 0 |  |  |  |  |  |  |  |
| 48 | 0 |  |  |  |  |  |  |  |
| 49 | 0 |  |  |  |  |  |  |  |
| 50 | 0 |  |  |  |  |  |  |  |
| 51 | 0 |  |  |  |  |  |  |  |
| 52 | 0 |  |  |  |  |  |  |  |
| 53 | 0 |  |  |  |  |  |  |  |
| 54 | 0 |  |  |  |  |  |  |  |
| 55 | 0 |  |  |  |  |  |  |  |
| 56 | 0 |  |  |  |  |  |  |  |
| 57 | 0 |  |  |  |  |  |  |  |
| 58 | 0 |  |  |  |  |  |  |  |
| 59 | 0 |  |  |  |  |  |  |  |
| 60 | 0 |  |  |  |  |  |  |  |
| 61 | 0 |  |  |  |  |  |  |  |
| 62 | 0 |  |  |  |  |  |  |  |

**Table 2.** Expected Instruction Trace of “text.dat”