CMOS VLSI Design
Lab 3: Controller Design and Verification

The controller for your MIPS processor is responsible for generating the signals to the datapath to fetch and execute each instruction. It lacks the regular structure of the datapath. In the first section of the lab, you will design the ALU decoder control logic by hand. You will discover how this becomes tedious and error-prone even for small designs. For larger blocks, especially designs that might require bug fixes late in the design process, hand place and route becomes exceedingly onerous. Therefore, you will use a *PLA generator* to automatically generate the combinational logic for the control FSM.

I. ALUDec Logic

The ALUDec logic is responsible for decoding a 2-bit *ALUOp* signal and a 6-bit *funct* field of the instruction to produce three multiplexer control lines for the ALU. Two of the signals select which type of ALU operation is performed and the third determines if input B is inverted.

The function of the ALUDec logic is defined in Chapter 1 of *CMOS VLSI Design*. The Verilog code in

```
typedef enum logic [5:0] {ADD = 6'b100000,
                          SUB = 6'b100010,
                          AND = 6'b100100,
                          OR  = 6'b100101,
                          SLT = 6'b101010} functcode;

module aludec(input  logic [1:0] aluop,
              input  logic [5:0] funct,
              output logic [2:0] alucontrol);

  always_comb
    case (aluop)
      2'b00: alucontrol = 3'b010;  // add for lb/sb/addi
      2'b01: alucontrol = 3'b110;  // subtract (for beq)
      default: case(funct)      // R-Type instructions
                 ADD: alucontrol = 3'b010;
                 SUB: alucontrol = 3'b110;
                 AND: alucontrol = 3'b000;
                 OR:  alucontrol = 3'b001;
                 SLT: alucontrol = 3'b111;
                 default:   alucontrol = 3'b101; // should never happen
               endcase
    endcase
endmodule
```

Figure 1 is an equivalent description of the logic. Note that the main controller will never produce an ALUOp of 11, so that case need not be considered. The processor only handles the five R-type instructions listed, so you can treat the result of other *funct* codes as don't cares and optimize your logic accordingly.

```
typedef enum logic [5:0] {ADD = 6'b100000,
                          SUB = 6'b100010,
```

```
                        AND = 6'b100100,
                        OR  = 6'b100101,
                        SLT = 6'b101010} functcode;

module aludec(input  logic [1:0] aluop,
              input  logic [5:0] funct,
              output logic [2:0] alucontrol);

  always_comb
    case (aluop)
      2'b00: alucontrol = 3'b010;  // add for lb/sb/addi
      2'b01: alucontrol = 3'b110;  // subtract (for beq)
      default: case(funct)      // R-Type instructions
                 ADD: alucontrol = 3'b010;
                 SUB: alucontrol = 3'b110;
                 AND: alucontrol = 3'b000;
                 OR:  alucontrol = 3'b001;
                 SLT: alucontrol = 3'b111;
                 default:   alucontrol = 3'b101; // should never happen
               endcase
    endcase
endmodule
```

Figure 1: System Verilog code for ALUDec module

Make a copy of your lab2_xx directory and name it lab3_xx. Create an aludec schematic in your mips8 library. Using the logic gates from Muddlib, design a combinational circuit to compute the ALUControl[2:0] signals from ALUOp[1:0] and Funct[5:0]. As Funct[5:4] are always 10 for any instruction under consideration, you may omit them as don't cares. Try to minimize the number of gates required because that will save you time and space in the layout.

Next, create an aludec layout. Remember to use metal2 vertically and metal1 horizontally. When you are done, provide exports for VDD, GND, and the eight inputs and three outputs.

Run DRC, ERC, and NCC and fix any problems you might find. If your schematic and layout do not match, consider simulating the layout to help track down any bugs.

II. Controller Verilog

The MIPS Controller is responsible for decoding the instruction and generating mux select and register enable signals for the datapath. In our multicycle MIPS design, it is implemented as a finite state machine, as shown in Figure 2.[1] The Verilog code describing this FSM is the statelogic and outputlogic modules in the RTL mips.sv.

Look through the Verilog and identify the major portions. The next state logic describes the state transitions of the FSM. The output logic determines which outputs will be asserted in each state. Note that the Verilog also contains the AND/OR gates required to compute PCChange, the write enable to the program counter.

[1] This FSM is identical to that of the multicycle processors in Patterson & Hennessy *Computer Organization and Design* and in Harris and Harris Digital Design and Computer Architecture, save that `LW` and `SW` have been replaced by `LB` and `SB` and instruction fetch now requires four cycles to load instructions through a byte-wide interface.

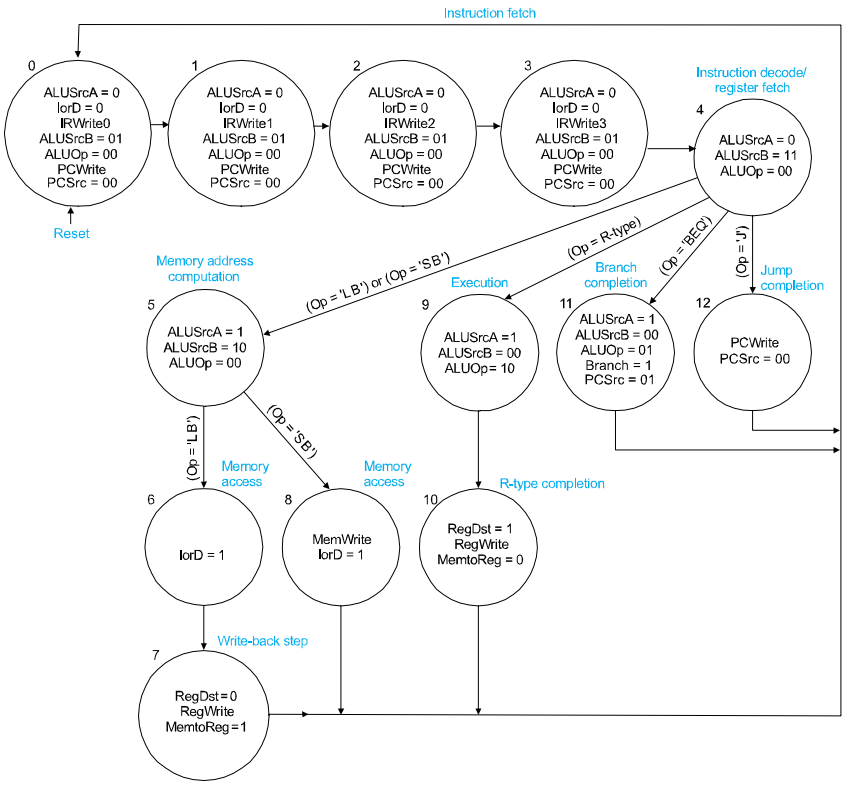

Figure 2: Controller FSM

III. Controller Synthesis

You will use a simple programmable logic array (PLA) generator to create a layout and schematic for the random logic of the controller. The PLA generator, written by Justin Gries and Danny LaValle at Harvey Mudd College, produces a PLA from a Verilog case statement. This is a primitive PLA generator, but will nevertheless save considerable manual effort.

Look in the lab directory for PLAGenerator.jar, parameters.txt, and controller_pla.v, and copy them to your own directory. The input to the PLA generator is a single `casez` statement, supporting a limited subset of the Verilog language. The inputs must be a single bus and the outputs must be another single bus. Therefore, controller_pla.v integrates the two `case` statements and uses two busses for the input and output of the case statement: `in` and `out`. `assign` statements are used to group the inputs and outputs into single busses. Pay attention to the order of bits within the bus. Recall that don't cares are indicated with a "?" in a `casez` statement.

The last two lines of the controller_pla.v `case` statement are missing for the final states of BEQ and J (states 11 and 12 in Figure 2). Using the RTL as a guide, complete controller_pla.v by adding these missing lines. Once you are satisfied the modified case statement is complete, simulate the `case` statement with the RTL to verify before generating your PLA. This involves commenting out the statelogic and outptlogic

modules, pasting in the PLA, and uncommenting the assign statements to group the inputs and outputs.

Start the PLA Generator by double-clicking on PLAGenerator.jar. It will prompt your for a parameters file; select parameters.txt. It contains generation parameters, such as the generated cell name and metal widths. You can leave these parameters as-is. It will prompt for a case.v file, select your controller_pla.v. It may take a few minutes to generate the controller_pla.jelib library. Copy the controller_pla{sch} and controller_pla{lay} cells from this library into your mips8 library.

The PLA generator produces DRC errors in certain versions of Electric. It adds extra pins that may cause DRC errors. Click on Edit • Cleanup Cell • Cleanup Pins in the layout and schematic cells to remove them. It also leaves notches in the p-select around the weak pMOS tranisistors at the left side and the top right side. Drop a huge blob of pure layer node p-select over these areas to eliminate the notches.

Generate an icon by opening the controller_pla schematic and clicking on View • Make Icon View.

IV. Controller Assembly and Simulation

Add the PLA and ALUDec icons to controller{sch} and wire them up. Open controller{lay} and place the PLA and ALUDec modules. You should be able to use auto-stitch to connect the modules to existing pins. Pins at the bottom of the controller cell should not be moved because they are pitch matched to the pins of the datapath. The state registers and the random logic in the controller for the pcen signal are already included. Verify your controller with DRC, NCC, and ERC. Write a Verilog deck for your controller{lay} and substitute it for the controller in the RTL, then simulate to verify your design.

V. What to Turn In

Please provide a hard copy of each of the following items:

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 the future.
2. A printout of the aludec schematics and layout.
3. Your completed controller_pla.v.
4. A printout of the controller schematics and layout.
5. Does your controller simulate with the RTL?
6. What are the DRC, ERC, and NCC status of each block you designed: alucontrol, controller_pla, and controller?