CMOS VLSI Design

Lab 4: Controller Design

Harris

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. Therefore, you will use a *standard cell* methodology to place and route the gates. First, you will design the ALU Control logic by hand and place and route it yourself. 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 learn about *synthesis* and *place and route*. You will complete a Verilog Hardware Description Language (HDL) description of the MIPS Controller. Then you will use the industry-standard Synopsys Design Analyzer tool to synthesize the Verilog into a gate-level netlist. You will import this netlist into Electric and use Electric's Silicon Compiler tool to place and route the design.

If you are unfamiliar with Verilog or want a review, please refer to Appendix A of CMOS VLSI Design.

1. Standard Cell Library

In your datapath cells, you used horizontal metal2 lines to route over the cells along a datapath bitslice. In a standard cell place and route methodology, you will not be doing over-the-cell routing. Therefore, you can usually achieve better cell density by running metal1 horizontally and using metal2 vertically to provide inputs to the cells. Moreover, the elementary gates in the standard cell library are less complex than the full adder in the datapath. Therefore, we will use a 60λ cell height rather than 80λ.

Copy your lab3_xx.elib library to lab4_xx.elib. The library has a number of standard cells already provided including inverters, NAND, NOR, and AOI gates, and a latch. Their names begin with std_. The layout for the 3-input NAND is missing. To become familiar with standard cell layout styles, create the **std_nand3** layout. It should be done in the same style as the **std_nor3** gate obeying the following guidelines:

- V_{DD} and GND run horizontally in 8λ-wide Metal 1 on a 60λ center-to-center spacing
- all transistors, wires, and well contacts fit between the power and ground lines
- all transistors should be within 100λ of a well contact
- avoid long routes in diffusion
- inputs and outputs appear on metal2 so that a metal2 line can be connected from above without obstruction. Do not place an input or output directly on a via because it will look funny when you look at an instance of the cell higher up in the hierarchy. Instead, attach a metal2 line to the via and place the export on the metal2 pin.

Perform the usual DRC, ERC, and NCC verification.

2. ALUControl Logic

The ALUControl logic, shown in the lower right oval in Figure 1 of Lab 1, 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 lines select which type of ALU operation is performed and the third determines if input B is inverted.

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

     // FUNCT field definitions
     parameter    ADD = 6'b100000;
     parameter    SUB = 6'b100010;
     parameter    AND = 6'b100100;
     parameter    OR  = 6'b100101;
     parameter    SLT = 6'b101010;

     // The Synopsys full_case directives are given on each case statement
     // to tell the synthesizer that all the cases we care about are handled.
     // This avoids needing a default that takes extra logic gates or implying
     // a latch.

     always @(*)
           case (aluop) // synopsys full_case
                 2'b00: alucontrol <= 3'b010; // add (for lb/sb/addi)
                 2'b01: alucontrol <= 3'b110; // sub (for beq)
                 2'b10: case (funct) // synopsys full_case
                         ADD: alucontrol <= 3'b010; // add (for add)
                         SUB: alucontrol <= 3'b110; // subtract (for sub)
                         AND: alucontrol <= 3'b000; // logical and (for and)
                         OR:  alucontrol <= 3'b001; // logical or (for or)
                         SLT: alucontrol <= 3'b111; // set on less (for slt)
                         // no other functions are legal
                 endcase
                 // aluop=11 is never given
           endcase
endmodule
```

Figure 1: Verilog code for ALUControl

The function of the ALUControl logic is defined in Figure 5.14 of Hennessy & Patterson. The Verilog code in 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.

Create an **alucontrol** schematic. Using the logic gates provided, 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 the same for any legal operation, you may omit them as don't cares. Try to minimize the number of gates required because that will save you time on the layout. Create an IRSIM .cmd file with assertions and simulate your design to ensure it is correct. You may build off the alucontrol.cmd file provided.

Next, create an **alucontrol** layout. Use metal2 vertically and metal1 horizontally. Use a routing channel above or below the cells to make the connections. For example, the figure below illustrates an SR latch created from two std_nand2 gates in the standard cell layout style with a routing channel above the cells. When you are done, provide exports for VDD, GND, and the eight inputs and three outputs.
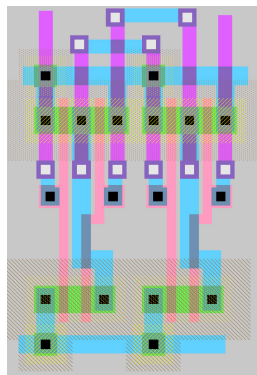

Figure 2: SR Latch using standard cell layout style

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.

3. Controller Verilog

The MIPS Controller, shown as the upper oval in Figure 1 of Lab 1, 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 3.[1] The Verilog code describing this FSM is named controller.v. Copy controller.v from the class directory to controller_xx.v in your directory.

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 Figure 5.42 of Patterson & Hennessy 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.

We would like to include support for the ADDI instruction in our processor. Consult *Patterson & Hennessy* if you are rusty on the instruction. Mark up the copy of the FSM diagram at the end of this lab with additional states to handle ADDI. Then edit the Verilog to add your new state transitions and outputs. If you are using WordPad, be careful that it does not append an undesired .txt to the end of the filename!
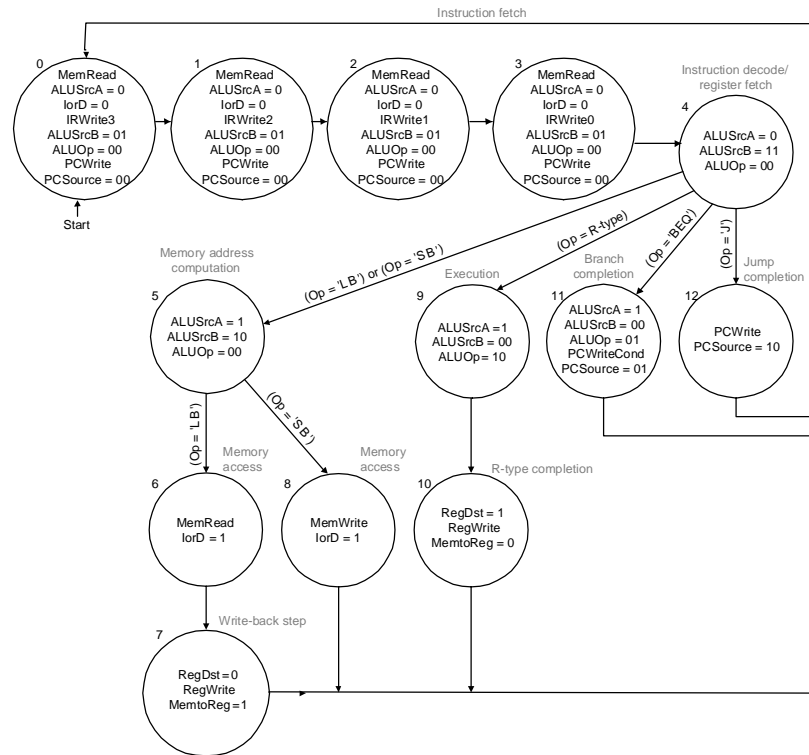

Figure 3: Controller FSM

4. Controller Synthesis

Synopsys sells a CAD tool called Design Analyzer that is widely used in industry to synthesize Verilog onto a library of logic gates. If you have access to Design Analyzer, you will synthesize your Verilog code from the previous part. The result of synthesis is a gate-level netlist in VHDL suitable for the place & route tool in Electric. If you do not have access, your instructor can provide the controller_sol.vhdl file and you can skip this step.

Use the Setup ● Defaults command to setup your libraries. You will need to include the Synopsys database file std_mudd.db defining the standard cell library. Set the search path to include the path to the std_mudd.db library as well as the other paths already there. For example, if the std_mudd.db file is at \\Charlie\Courses\Eng\E158\Labs\std_mudd.db and the J drive is mapped to \\Charlie\Courses, add J:/Eng/E158/Labs to the search path. Set the link library and target library to std_mudd.db and the symbol library to

generic.sdb. These defaults instruct Synopsys to map your Verilog controller onto the gates available in the std_mudd library.[2]

Use the File • Read to read your controller_xx.v file. Check in the command window for any errors indicating a mistake in your `addi` code. You should see two 4-bit latch memory devices inferred comprising the two latches in the state register. You should also see messages for each `case` statement. In some `case` statements, the `// synopsys full_case` directive is used to tell Synopsys that the code considers all logically possible cases and that therefore no gates are required to handle a default case.

Click on the controller in the design analyzer window and choose Tools • Design optimization. Select High effort and press ok to synthesize. Check to ensure there are no errors in the Command Window; you may see spurious warnings about improper octal values. Then double-click on the controller and descend twice to look at what was produced. You will see a tangle of gates implementing the MIPS controller. If you discovered you had a bug in your Verilog design, you would simply fix the Verilog, then resynthesize to produce a new gate-level implementation.

Finally, you will need to save the netlist describing the gates and their interconnections. The Electric Silicon Compiler wants a netlist in VHDL format. Therefore, use the File • Save As command to save in the format VHDL with the name controller_xx.vhdl.

5. Controller Place and Route

We will use the Queens University Interactive Silicon Compiler feature in Electric to place and route the controller specified with the VHDL netlist. This is a very primitive place and route tool, but will nevertheless save considerable manual effort. While in the mocmos technology, select Tools • Silicon Compiler • Silicon Compiler Options. Set the following options:

---

[2] If you wished to add more gates to the std_mudd.db Synopsys library database such as a 4-input nand, you would edit the std_mudd.lib file in the class directory to add additional gates. The gates in the library are presently modeled off LSI Logic's lsi_10k library provided with the Synopsys tools. Once you had added new gates to the .lib file, you would use the Library Compiler program (LC GUI among the Synopsys tools) to enter the following commands:

read_lib "std_mudd.lib" (expect to get some warnings about unspecified attributes)
write_lib std_mudd –output std_mudd.db

The output may appear in the Synopsys directory.

Horizontal routing arc:	Metal1
Horizontal wire width:	4
Vertical routing arc:	Metal2
Vertical wire width:	4
Power wire width:	8
Main power wire width:	20
Main power arc:	Vertical arc
P-well height:	38
P-well offset:	0.5
N-well height:	38
N-well offset:	0.5
Via size:	4
Minimum metal spacing:	4
Feed-through size:	12
Min Port distance:	8
Min Active distance:	-4
Number of rows of cells:	2

Look at the controller_xx.vhdl VHDL file in a text editor. After the library and use declarations, you will see the `entity` statement for the controller. This defines the inputs and outputs of the controller. Next, you will see the `architecture` statement for the controller. Within the architecture are `component` statements defining all the standard cells referenced within the controller. Next are a set of `signal` declarations defining the signals within the module. After the `begin` statement are a series of gate instantiations in which gates are connected together.

Use Windows • Text Options to set the text editor to point and click. Create a new facet named **controller** of view VHDL. Copy the controller_xx.vhdl file from the text editor and paste it into the new facet. The Silicon Compiler understands a very limited subset of VHDL, so you will have to make some changes to the VHDL netlist.

Specifically, Electric also does not understand the `<=` assign statement syntax, so we will have to avoid any signal renaming with such assignments. Look at the one or more assignments immediately after the `begin` statement such as

```
irwrite0 <= irwrite0_port
```

The exact list and number of signals to edit depends on how you coded your Verilog and how Synopsys happened to synthesize it. Each of these outputs will also appear in the `signal` statement. Delete them from that statement. For each of these outputs, use Edit • Special Function • Find Text to search and replace the signal name with _port with the same name without. For example, search for irwrite0_port and replace all instances of it with irwrite0 throughout the code. When you have replaced all the renamed signals, also delete the `<=` assignments that come just after the `begin` statement.

Use the Tools • Silicon Compiler • Get Network for Current Facet command to read the VHDL netlist. Open the controller{net-quisc} command file and look at how the VHDL was translated into a series of commands to create, connect, and name cells. Use the Tools • Silicon Compiler • Do Placement, Tools • Silicon Compiler • Do Routing, and Tools • Silicon Compiler • Make Electric Layout commands to place, route, and generate layout. Look at the resulting controller layout facet. You will see two rows of standard cells with all the necessary connections. Vias for metal2 connections to the inputs and outputs are provided.

Unfortunately, the Silicon Compiler is not very smart about separating metal2 lines. When you run hierarchical DRC, you may find some metal2 lines are too close together. Fix any errors you find. Run the Electrical Rule Checker to ensure the wells are correct.

While looking at the layout, select View • Make Schematic. This will produce a schematic to match the layout. Delete the vdd and gnd ports that were created in the schematic. Verify that Electric created the schematic correctly by using NCC on the controller.
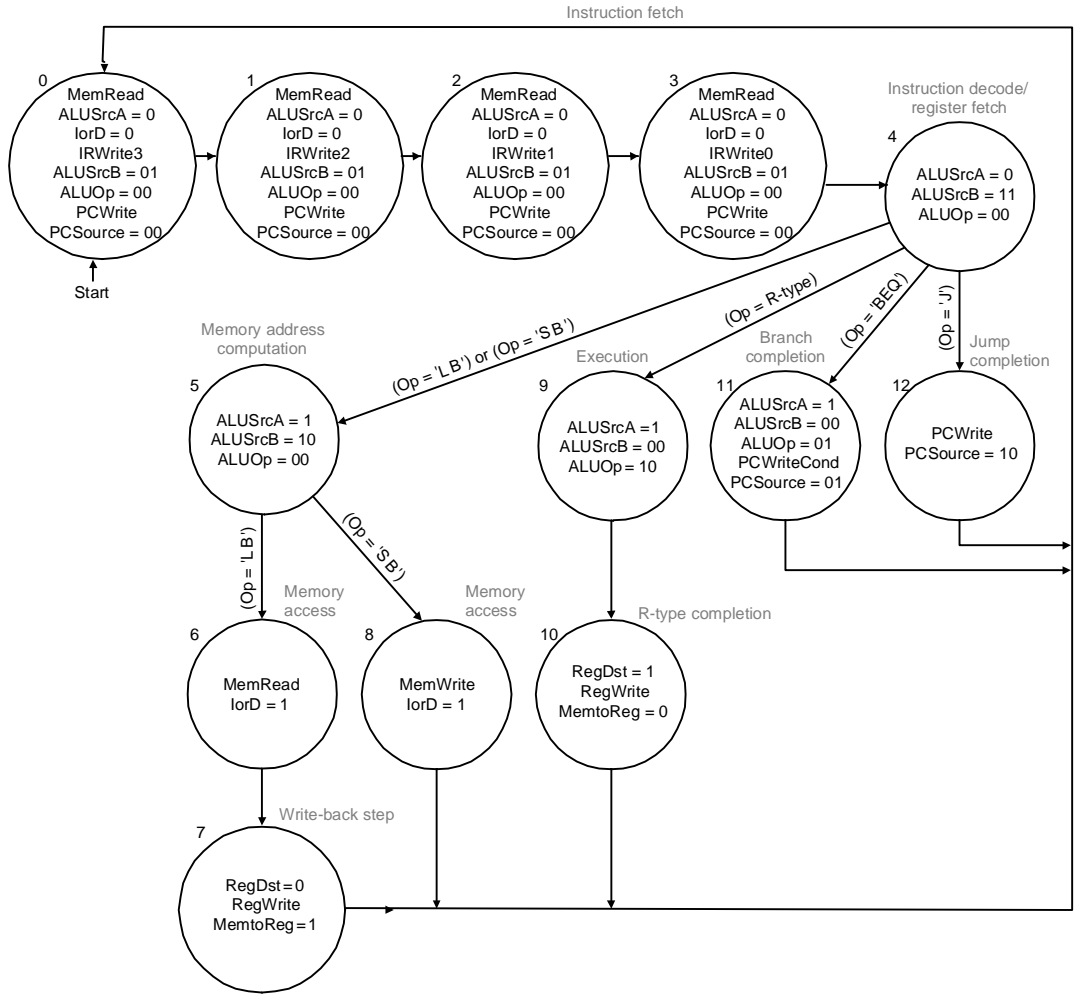
6. Controller Simulation

To verify that your controller works correctly, you can simulate it in IRSIM. controller.cmd has been provided for you. Look it over to see how vectors and clock commands are used. Look at the IRSIM Users Manual on the web page to understand the syntax of these commands. Simulate the layout and correct any assertion violations you may find.

7. 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. What are the DRC, ERC, and NCC status of each block you designed: std_nand3, alucontrol, controller?
3. A FSM diagram of your controller including the ADDI instruction
4. A printout of the alucontrol schematics and layout.
5. A printout of the alucontrol.cmd file with proper assertions. Does it simulate all legal inputs without assertion failures?
6. Does your controller.cmd file simulate without assertion failures? If not, explain.

0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite3
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

Start

1
MemRead
ALUSrcA = 0
IorD = 0
IRWrite2
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

2
MemRead
ALUSrcA = 0
IorD = 0
IRWrite1
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

3
MemRead
ALUSrcA = 0
IorD = 0
IRWrite0
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

Instruction decode/
register fetch

4
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

(Op = 'LB') or (Op = 'SB')

(Op = R-type)

(Op = BEQ)

(Op = 'J')

Memory address
computation

5
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Execution

9
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Branch
completion

11
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

Jump
completion

12
PCWrite
PCSource = 10

(Op = 'LB')

(Op = 'SB')

Memory
access

6
MemRead
IorD = 1

Memory
access

8
MemWrite
IorD = 1

R-type completion

10
RegDst = 1
RegWrite
MemtoReg = 0

Write-back step

7
RegDst = 0
RegWrite
MemtoReg = 1

Mark up this FSM to add states to support the ADDI instruction.