ECE 3450: Digital Electronics
Fall 2009, Lab #4

Dr. Pallav Gupta
`pallav.gupta@villanova.edu`

I Objectives

The goals of this lab are the following:

1. Learn standard cell design.
2. Design and layout the ALU control logic of the simplified MIPS processor.
3. Synthesize and place & route the controller logic of the simplified MIPS processor.
4. Become proficient in editing relatively large-scale designs.

II Courses Forum

The forum is located at http://pandim.ece.villanova.edu/phpbbforum. Please use it to ask questions.

III Submission Instructions

The lab must be submitted via the web at http://pandim.ece.villanova.edu. Instructions on how to submit are on the courses forum. Apply the following information on the Remository form when submitting your lab.

Filename:	`lab4-[first name]-[last name].zip`, *e.g.*, `lab4-john-doe.zip`
Location:	`ECE 3450/LAB4`
Title:	`Lab 4`
Author:	Your name, *e.g.*, `John Doe`

Note: Follow the above convention strictly. Failure to do so will result in a **zero**. We request your lab in this specific format so that the automated scripts at the back-end can run smoothly without breaking. Please adhere to it.

See Section X on how to name and organize the files that you will need to submit for this lab.

IV Collaboration

You must complete this lab independently; you are not allowed to work in pairs or a group. However, you are welcome to discuss the material with your colleagues.

V Prerequisites

You must have completed Labs #1-3 before proceeding further. If not, then it your responsibility to do that first. Otherwise, you will be totally lost in this lab.

VI Provided Files

All the files for this lab are provided in `lab4.zip`. Unpack this archive and you will see the following directory structure:

```
lab4/
┣━README                             readme
┣━lab4-xx.jelib                      Electric library
┣━alucontrol.v                       Verilog code for ALU control logic
┣━controller.v                       Verilog code for datapath controller logic
┗━std_vill.lib                       Library file for Synopsys Design Compiler
  sim/                               IRSIM test vectors
  ┣━alucontrol.cmd
  ┗━controller.cmd
```

The Electric file provides a small library of standard cells that will be necessary to complete this lab. Browse through to understand them. The Verilog files contain code for the ALU control logic and controller. The `std_vill.lib` contains area/timing information for our cells to be used during synthesis. The `sim/alucontrol.cmd` contains sample IRSIM test vectors to test the ALU control logic. It is not complete! The `sim/controller.cmd` contains complete IRSIM test vectors to test the controller.

VII Introduction

The controller of our simplified MIPS processor is responsible for generating the control signals to the datapath to fetch and execute each instruction. However, it lacks the regular structure of the datapath. Therefore, you will use a *standard cell* methodology to place & route the design.

Initially, you will design and place & route the ALU control logic by hand. You will soon discover how this becomes tedious and error-prone. For larger designs, especially those that might require bug fixes late in the design process, manual place & route becomes exceedingly cumbersome. Consequently, you will learn about *synthesis* and *place & route*. You will complete a Verilog hardware description language (HDL) description of the MIPS controller. Then you will use the industry-standard Synopsys Design Compiler to synthesize the Verilog into a VHDL gate-level netlist. You will import this netlist into Electric and use its Silicon Compiler tool to place & route the design.

If you are unfamiliar with Verilog or VHDL, consult the Appendix of the textbook.

VII.A Designing the Standard Cell Library

In designing the datapath cells, you used horizontal `metal1`, `metal3` wires to route over the cells along a bitslice. In a standard cell methodology, over-the-cell routing is not employed. Standard cells are tiled into rows separated by *routing* channels. The number of wires that must be routed sets the height of the routing channels. `metal1` runs horizontally and `metal2` vertically to provide inputs to the cells. If the fabrication technology supports multiple metal layers, then, the routing channels are unnecessary, and over-the-cell routing can be performed using the standard cells. The elementary gates in a standard cell library are less complex than the fulladder in the datapath. Therefore, you will use 60λ cell height rather than 90λ.

Automatic synthesis and place & route tools have become sophisticated enough to map entire designs onto standard cells. They tend to be larger and somewhat slower than good custom design (*i.e.*, what you did in the previous lab), but they also take an order of magnitude less design time.

Open `lab4-xx.jelib` and examine its contents. The library provides a number of standard cells (*e.g.*, `std_aoi`, `std_inv`, `std_nor2`, `std_nand2`, `std_latch`, *etc*). Study these cells in detail until you understand them. The layout of the 3-input NAND (`std_nand3`) is missing. To help you become familiar with standard cell layout styles, your job is to design the layout in `std_nand3{lay}`. Note that it should be done in the same style as `std_nor3{lay}`. Keep the following guidelines in mind:

1. Power and ground lines run horizontally in 8λ-wide `metal1` on a 60λ center-to-center spacing; well contacts should be placed on the rails every 4λ.
2. All transistors, wires, and well contacts fit between the power and ground lines.
3. All transistors should be within 100λ of a well contact.
4. Avoid long routes in diffusion.
5. You may find it necessary to add a large rectangle of `N-well` or `P-well` to surround the transistors and eliminate spacing problems.
6. Export all inputs, outputs, and power and ground lines; inputs and outputs should be aligned with the well contacts.
7. Inputs and outputs appear on `metal2` so that a `metal2` line can be connected from above without any obstruction. Do not place an input or output directly on a contact (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 contact and place the export on the `metal2` pin.
8. Create a neat and clean layout to avoid problems in the following labs.

Perform DRC, ERC, and NCC on the layout.

VII.B Designing the ALU Control Logic

The ALU control logic, shown in the MIPS block diagram of Lab #3, is responsible for decoding a 2-bit `aluop[1:0]` signal and a 6-bit `funct[5:0]` field of the instruction to produce three multiplexer control lines (`alucontrol[2:0]`) for the ALU. Two of the lines determine which type of ALU operation (*e.g.*, `ADD`, `SUB`, `AND`, *etc*) is performed while the third line determines if the second operand B is complemented or not.

The function of the ALU control logic is defined in Chapter 1 of the textbook. Refer to it for further explanation. The Verilog code shown in Fig 1 is an equivalent high-level description of the logic. Note that the controller will never produce an `aluop` of `11`, so that case need not be considered (*i.e.*, don't care). Our processor only handles the five R-type instructions shown, so you can treat the result of other `funct` codes as don't cares and optimize (simplify) your logic accordingly.

Your job is to design and layout a *combinational* circuit that implements the ALU control logic shown in Fig. 1. Use Karnaugh maps to determine the Boolean equations of `alucontrol[2:0]` as a function of `aluop[1:0]` and `funct[5:0]`. Note that since `funct[5:4]` are always the same for any legal operation, you can treat them as don't cares. Try to minimize the number of gates required because that will save you time on the layout.

Once you have determined the equations for `alucontrol[2:0]`, draw the schematic in `alucontrol{sch}`. Export all inputs and outputs. Perform DRC and IRSIM simulation. The IRSIM file `alucontrol.cmd` provides a few test vectors. It is incomplete. Add the remaining necessary test vectors and assertions to this file and simulate your design to ensure it is correct. Make sure you test for all legal instructions.

Next, create the layout in `alucontrol{lay}`. Use `metal1` and `metal2` horizontally and vertically, respectively. Use a routing channel above or below the cells to make the connections. For example, Fig. 2 illustrates an SR latch constructed from two `std_nand2` cells in the standard cell layout style with the routing channel above the

```
//                                    -*- Mode: Verilog -*-
// Filename        : alucontrol.v
// Description     : Description of the ALU control logic for our simplified MIPS processor.
// Author          : Pallav Gupta
// Created On      : Wed Oct 22 21:43:19 2008
// Last Modified On: Time-stamp: <2008-10-22 21:07:01 pgupta>
// Update Count    : 0
// Status          : Unknown, Use with caution!

module alucontrol(input [1:0] aluop, // alu op code
                  input [5:0] funct, // function code
                  output reg [2:0] alucontrol // generated control signals for the ALU
                  );

   // 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 statement to tell the
   // synthezizer that all the cases we care about are handled. This avoids
   // needing a default that takes extra logic gate 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; // sub (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 (don't care)
              endcase // case (funct)
       // aluop = 11 is never given (don't care)
     endcase // case (aluop)
endmodule // alucontrol
```

Figure 1: Verilog code for the ALU control logic.

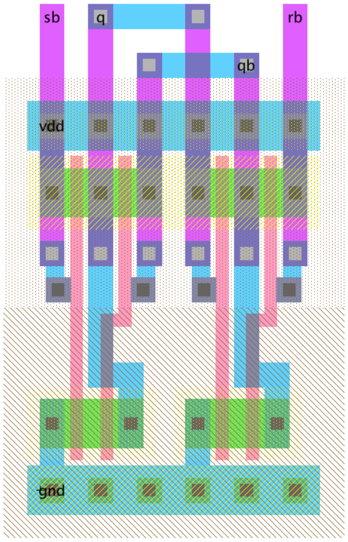

Figure 2: *SR* latch constructed from two NAND2 gates using standard cell layout style.

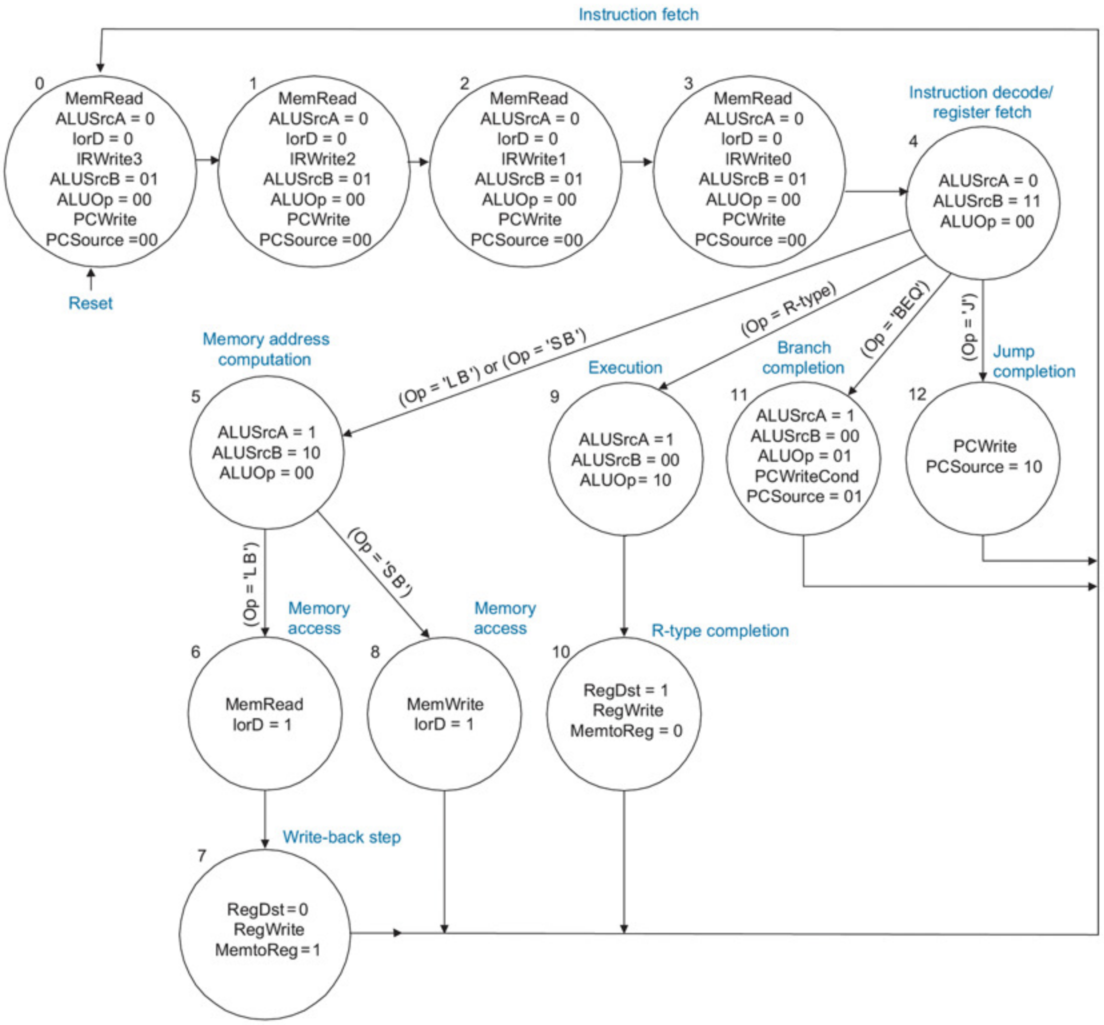

Figure 3: FSM of the MIPS controller.

cells. Export all inputs, outputs and power and ground lines. Perform DRC, ERC, and NCC on the layout and fix any errors. Perform IRSIM simulation on the layout. This may help track down bugs in your design as well.

VII.C Designing the Controller

The MIPS controller, shown in the MIPS block diagram of Lab #3, is responsible for decoding the fetched instructions and generating the multiplexer select and register enable signals for the datapath. These signals are fed into the zipper which then drives them onto the datapath. In our multicycle MIPS design, it is implemented as a finite state machine (FSM), as shown in Fig. 3.

The Verilog code describing this FSM is given in `controller.v`. Open the file and browse through the code to identify the major portions. Read the comments carefully. To further aid your understanding, try to match the code with Fig. 3. The next stage logic describes the state transitions in the FSM. The output logic determines which output signals will be asserted in each state. Note that the Verilog code also contains the AND/OR gates required to compute `pcen`, the write enable to the program counter.

Currently, the controller does not implement the add immediate (`ADDI`) instruction. Your job is to modify the Verilog code to support this instruction. The definition of the `ADDI` instruction is given in Table 1. Mark up the

Table 1: `ADDI` Instruction Semantics

Instruction	Function	Encoding	Op	Funct
`addi $1, $2, imm`	add immediate: `$1 <- $2 + imm`	I	`001000`	n/a

FSM in Fig. 3 with additional states to handle `ADDI`. Annotate the states with output signals that need to be asserted and their respective values. Then, edit the Verilog code to add the new state(s) and outputs. Comments in the code will indicate where to add things. If you are using a Windows based text editor, be careful that it does not append an undesired file extension (*e.g.*, .txt) to the end of the filename.

A testbench (`controller_tb`) has been provided in `controller.v` to test the controller logic. Use the testbench and simulate the code in ModelSim to ensure that the controller is functioning properly. If the Verilog fails to compile, fix the errors and try again. The testbench will dump all the signals to a file named `controller.vcd` which can viewed in the ModelSim Waveform Viewer. Alternatively, the testbench will also dump the results to standard output. Compare this against the test cases given in `controller.cmd` to ensure that they match.

VII.C.1 Controller Synthesis

Companies like Synopsys and Mentor Graphics sell CAD tools that can synthesize Verilog onto a library of logic gates. The result of synthesis is a gate-level netlist in VHDL suitable for the place & route tool in Electric.

The controller that you designed earlier needs to be synthesized, and the netlist needs to be generated in available in `controller.vhdl`. Refer to the Synopsys Design Compiler tutorial for instructions on how to synthesize the netlist. Once the netlist has been generated, open the file and examine its contents. Note that the only primitives used in the netlist are gates that are in the standard cell library (*i.e.*, `std_*`).

Important: Do not attempt to synthesize the netlist until you have added support for the ADDI instruction and verified that the controller is working properly. Otherwise, you will have lots of problems in the next lab.

VII.C.2 Controller Place & Route

You will use the Queens University Interactive Silicon Compiler feature in Electric to place & route the controller specified with the VHDL netlist. This is a very primitive place & route tool, but will nevertheless save considerable manual effort. Before doing place & route, you will need to set some options in Electric. Choose File → Preferences → Tools → Silicon Compiler. Set the options to their respective values as shown in Table 2. The options are self-explanatory.

Open up `controller.vhdl` in a text editor and examine the contents. After the library and use declarations, you will see the entity statement for the controller. This defines the controller's inputs and outputs. Next, you will see the architecture statement. 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.

Create a new cell with VHDL view and name it `controller`. You should end up with `controller{vhdl}`. Copy the contents of `controller.vhdl` and paste it into this facet. Save the cell. Choose Tool → Silicon Compiler → Convert Current Cell to Layout. This will produce `controller{lay}` and `controller{net.quisc}`. Examine the latter to see how the netlist was translated into a series of commands to create, connect, and name cells. Look at the controller layout and you will see two rows of standard cells with the necessary connections. Vias for `metal2` connections to the inputs and outputs are provided. All the signals have also been exported. The layout should look similar to that shown in Fig. 4.

Unfortunately, Silicon Compiler may introduce some DRC errors in the layout. Run hierarchical DRC and fix any errors you find by hand. Then, run ERC to ensure that the well contacts are correct. While looking at the layout, choose View → Make Schematic View. This will produce `controller{sch}` that is equivalent to the

Table 2: Place & Route Options

Option	Value	Option	Value
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	Metal2		
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	Minimum port distance	8
Minimum active distance	-4	Number of rows of cells	2

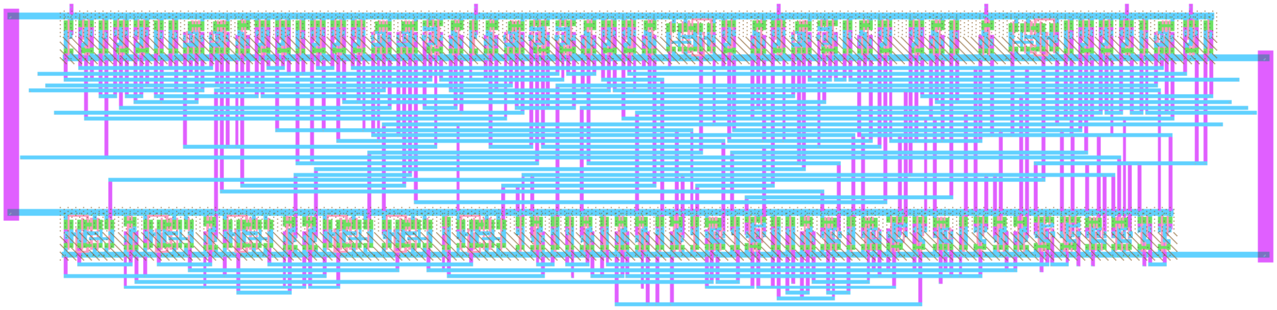

Figure 4: Automatic layout of the MIPS controller.

layout. Run DRC on the schematic and fix any errors you find (*e.g.*, dangling arcs, unnecessary pins). Verify that the layout and schematic match by running NCC.

VII.C.3 Controller Verification

To verify that your controller functions correctly, simulate the schematic and layout using IRSIM. The test vectors are provided in `controller.cmd`. Browse through the file to understand how the test vectors and clock commands are used. Refer to the user manual for help on syntax. Simulate the controller and correct any assertion violations you may find.

VIII Technical Report

A technical report (not to exceed 10 pages) is to be written that details everything you have done in this lab. You should present the design of the ALU control logic and controller. You should present the appropriate schematics and layouts. Furthermore, you should show the results of any simulations, and whether the layout passed DRC, ERC, and NCC or not. Elaborate on any difficulties faced in this lab and the employed workarounds. Summarize what you have learned from doing this lab.

The technical report must be of the highest standards. Otherwise, it runs a high risk of being rejected which will impact your lab grade. You can consult some technical publications to see how to write a good technical report. It must be written using the LaTeX template that was provided earlier. The template can be downloaded at http://pandim.ece.villanova.edu.

IX Parting Words

Congratulations on finishing this lab! Hopefully, you now have some experience in designing standard cells, controller design, and manual/automatic place & route.

X What to Submit

For this lab, you must submit the following files:

1. The Electric library. Name it `lab4-xx.jelib` where `xx` are your initials.
2. The Verilog code of the controller containing support for `ADDI`. Name it `controller.v`.
3. The synthesized VHDL netlist of the controller. Name it `controller.vhdl`
4. The IRSIM test vector file for the ALU control logic. Name it `alucontrol.cmd`.
5. The LaTeX PDF file which contains the technical report. Name it `report-xx.pdf` where `xx` are your initials.

Take all the files and archive (zip) them into a folder. Name the folder `lab4-[first name]-[last name].zip`. See Section III on how to submit the archive. Failure to follow these instructions will result in a **zero** for the lab. No ifs, buts, *etc.*

Important: The Electric library must contain the schematics, icons, and layouts of the standard cells, alucontrol, and controller. All schematics must pass DRC and IRSIM simulation. The layouts must pass DRC, ERC, NCC, IRSIM simulation, and be drawn as per specification (outlined above). Icons must be of the correct size. The `controller.v` must contain support for `ADDI` instruction. The `alucontrol.cmd` must contain the test vectors for the ALU control logic.

XI Errors

I usually write precise tutorials and bug-free code. However, I am human (do not be surprised) and do make mistakes. In addition, CAD tools get updated frequently and the interface might change, rendering parts of the writeup ineffective. If you find any mistakes or inconsistencies while doing this lab, please bring it to my attention **immediately**. You will earn extra points if what you report is indeed a bug.