***CMOS VLSI Design Course***

**LAB 3**

**Controller Design and Verification**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc48854891)

[1.1 Lab overview 1](#_Toc48854892)

[2 Learning Objectives 1](#_Toc48854893)

[3 aludecoder Logic 1](#_Toc48854894)

[3.1 Create aludecoder schematic 2](#_Toc48854895)

[3.2 Netlist and simulate 2](#_Toc48854896)

[3.3 Create aludecoder symbol and layout 3](#_Toc48854897)

[4 Controller 3](#_Toc48854898)

[4.1 Synthesis 3](#_Toc48854899)

[4.2 Place and route with Encounter 4](#_Toc48854900)

[4.2.1 Set flow environment 4](#_Toc48854901)

[4.2.2 Start SOC Encounter 5](#_Toc48854902)

[4.2.3 Import design 5](#_Toc48854903)

[4.2.4 Floorplan 5](#_Toc48854904)

[4.2.5 Power planning 6](#_Toc48854905)

[4.2.6 Special route for power planning 6](#_Toc48854906)

[4.2.7 Place standard cells 6](#_Toc48854907)

[4.2.8 Route design 7](#_Toc48854908)

[4.2.9 Add filler cells 7](#_Toc48854909)

[4.2.10 Verify connectivity and geometry 7](#_Toc48854910)

[5 Import the Controller Schematic and Layout 8](#_Toc48854911)

[5.1 Create new library 8](#_Toc48854912)

[5.2 Import synthesized Verilog 8](#_Toc48854913)

[5.3 Import DEF file from encounter 8](#_Toc48854914)

[6 What to Turn In 9](#_Toc48854915)

# Introduction

## Lab overview

The controller for your microprocessor 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 become exceedingly onerous. Therefore, you will use Synopsis’s Design Compiler to synthesize the combinational logic for the controller and Cadence’s Encounter to automatically place and route the controller.

# Learning Objectives

At the end of this lab, you should be able to:

* Design schematics and layout by hand for a small random logic
* Synthesize a random logic block with Synopsys Design Compiler
* Place and route the synthesized block with Cadence SOC Encounter
* Verify the blocks with simulation, DRC, and LVS

# aludecoder Logic

The **aludecoder** takes two inputs, a **1-bit ALUOp** signal and a **5-bits Funct** signal from which it decodes the control for the ALU. The aludecoder produces the decoded ALU control on its 2-bit ALUControl signal. The possible decoded operations of the ALU are:

* Arithmetic
	+ Add
	+ subtract
* Logic
	+ AND
	+ OR

The **aludecoder** also produces **two FlagW** signals indicating whether to update the **Z** and **C** flags in the **conditional unit**.

The Verilog code shown below is an equivalent description of the logic. The processor only handles the four R-type instructions listed, so you can treat the result of other **Funct** codes as don’t cares and optimize your logic accordingly.

module aludecoder(input logic ALUOp,

 input logic [4:0] Funct,

 output logic [1:0] ALUControl,

 output logic [1:0] FlagW);

 always\_comb

 if (ALUOp) begin // which Data-processing Instr?

 case(Funct[4:1])

 4'b0100: ALUControl = 2'b00; // ADD

 4'b0010: ALUControl = 2'b01; // SUB

 4'b0000: ALUControl = 2'b10; // AND

 4'b1100: ALUControl = 2'b11; // ORR

 default: ALUControl = 2'bx; // unimplemented

 endcase

 FlagW[1] = Funct[0]; // update Z flag if S bit is set

 FlagW[0] = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);

 end else begin

 ALUControl = 2'b00; // add for non data-processing instructions

 FlagW = 2'b00; // don't update Flags

 end

endmodule

## Create aludecoder schematic

* Create a new library named controller\_xx and attach it to the UofU\_TechLib\_ami06 tech library.
* Create the aludecoder schematic view in your controller\_xx library.
* Using the logic gates from **muddlib11**, design a combinational circuit to compute the ALUControl[1:0] and FlagW[1:0] signals from ALUOp and Funct[4:0].

|  |  |
| --- | --- |
| Lights On | Limit yourself to the **inv, nand2, nand3, nor2,** and **nor3** gates so that you gain experience designing with inverting gates.As Funct[1] is always 0 for any instruction under consideration, you may omit it as don’t cares. Try to minimize the number of gates required because that will save you time and space in the layout. Remember to name your busses with angle brackets (e.g., ALUControl<1:0>). |

* Connect individual bits to your input and output busses.
	+ Draw the busses with a wide wire and connect a pin with the appropriate name.
	+ Then, draw narrow wires from the bus to individual logic gates.
	+ Add a label for each of these wires with its name (e.g., ALUControl <0>).

## Netlist and simulate

Next, netlist and simulate your aludecoder schematic with the full chip Verilog like you simulated the datapath schematic in lab 2.

It is tricky to simulate multiple separate netlisted modules at once, so using the Verilog version of the datapath for this simulation is fine.

|  |  |
| --- | --- |
| Lights On | It is much easier to fix logic errors before you make the layout instead of afterward. |

## Create aludecoder symbol and layout

* Make a symbol for your aludecoder.

Finally, create an aludecoder layout.

* Try to create it all in one long line with only one set of power and ground.
* Remember to use metal2 vertically and metal3 horizontally.
* When you are done, provide pins for the five inputs and the four outputs.
* **Do not** provide pins for vdd! or gnd! as that will confuse the autorouter in lab 4.

Run DRC and LVS and fix any problems you might find.

# Controller

The microprocessor controller is responsible for decoding the instruction and generating mux select and register enable signals for the datapath. As detailed in the Getting Started Guide, it contains a main finite state machine for producing control signals for each step of an instruction. It also contains some conditional logic to determine if an instruction should be conditionally executed (CondEx) and to kill the write enables if it should not be executed. Finally, it contains some combinational logic for control signals that only depend on Instr.

Look through the processor\_multi.sv SystemVerilog and identify the major portions. The top-level module is called controller.

## Synthesis

You will use Synopsys Design Compiler to synthesize the controller module into a gate-level netlist. Design Compiler is the industry-standard logic synthesis tool.

* Create a new directory named synth in your IC\_CAD directory.
* Design Compiler requires a configuration file in the directory where you will run. Copy it over using:

**cp /courses/cmosvlsi/20/lab3/.synopsys\_dc.setup ~/IC\_CAD/synth**

Look over the file to see how it defines some configuration for the Design Compiler.

Design Compiler can be driven at the command line, but it is easier to place all of the commands in a script.

* Copy a generic synthesis script from the class directory

**cp /courses/cmosvlsi/20/lab3/syndc.tcl ~/IC\_CAD/synth**

Look over the script. It is written in TCL (tool control language), with extensions that Design Compiler understands. Near the beginning of the file, it sets **“myFiles”** to **processor\_multi.sv** and the **basename** (the module you want to synthesize) to **controller**.

|  |  |
| --- | --- |
| Lights On | If you were synthesizing something else, you would need to change myFile and basename. |

* Copy your processor\_multi.sv file from lab2 into the synth directory.
* While in the synth directory, invoke synthesis using the command

**syn-dc –f syndc.tcl**

You’ll get a bunch of messages that may scroll off the screen. It may be easier to pipe them to more so you get one screenful at a time:

**syn-dc –f syndc.tcl | more**

The first time that you run, Design Compiler will analyze the muddlib.db file to determine which cells are available in the cell library. Subsequent runs in the same directory will be much faster after this analysis is complete.

Check the report carefully. You should get many warnings in the first 110 lines of processor\_multi.sv because the testbench contains nonsynthesizable commands such as initial blocks, assertions, and $finish. You’ll also get warnings about driving cell attributes that you may ignore. You’ll also notice that certain unused bits are optimized out of the instruction register. Get a sense of what a good report looks like so you can recognize a bad one.

Inspect the output files named **controller\_syn.v**, **.rep**, **.pow**, **and .sdc**.

* The .v file is the structural netlist produced by synthesis.
* The .rep file is the synthesis report, including a summary of the critical path timing and the area.
* The .pow file has a power report.
* The .sdc file contains timing constraints.

## Place and route with Encounter

Now, you can import the synthesized design back into the Cadence tools and place & route it into a layout using SOC Encounter (SOC stands for System-On-Chip).

### Set flow environment

* Make another directory in **IC\_CAD** called **soc** for your SOC Encounter runs.
* Then, make a subdirectory within soc (e.g., lab3\_xx) for this particular run.
* Change into the new run directory.

You will need copies of your structural netlist, and timing constraints files from synthesis. The best way to do this is to create a symbolic link so that if you change your synthesis results, the new netlist is automatically visible. Use the following commands:

**ln -s ~/IC\_CAD/synth/controller\_syn.v .**

**ln -s ~/IC\_CAD/synth/controller\_syn.sdc .**

You’ll also need links to **muddlib.lib** and **muddlib.lef**. Use the following commands:

**ln –s /courses/cmosvlsi/20/lab3/muddlib.lib .**

**ln –s /courses/cmosvlsi/20/lab3/muddlib.lef .**

muddlib.db is the Synopsys library file containing timing information about the cells used by synthesis.

muddlib.lef is a Library Exchange Format file containing physical information about the cell sizes and pin locations.

### Start SOC Encounter

Still inside the new run directory, invoke SOC Encounter at the command line by typing:

 **cad-soc**

Note that Encounter needs your terminal window and will crash if you try to run it in the background.

Encounter can also be driven with a GUI or with a script. In this lab, we’ll use the GUI because there aren’t too many commands to enter, and you’ll be able to see what is going on.

### Import design

Now, we will import the netlist obtained from the synthesis earlier. In the Encounter window,

* **Go to:** File • Import Design.
* Enter **controller\_syn.v** for your **Verilog netlist**.
* Enter **controller** as the top-level cell.
* Enter **muddlib.lef** as the LEF file.
* **Click** on the Advanced tab.
* Click on Power and enter VDD as the Power Net and VSS as the Ground Net.
	+ Remember to capitalize, else you will get failures.

Watch for errors in the console. You can ignore the max\_capacitance attribute warnings if they appear.

|  |  |
| --- | --- |
| Lights On | Encounter’s internal state is easily corrupted when there are errors. If you get errors along the way, it is better to start over from scratch by reinvoking cad-soc rather than attempting to redo the command. |

### Floorplan

* **Go to:** Floorplan • Specify Floorplan.
* Set margins of 30 (microns) from the core to the left, right, top, and bottom sides to give room for a power ring later on.
	+ You will see a window with some rows for standard cells and some space around the edge.
* You can leave 0 spacing between pairs of rows for now.
	+ If you were building a more complex design and had trouble with insufficient space for routing, you might wish to increase the row spacing under the Advanced tab.
* File • Save Design … and save the design as controller floorplan.enc.

|  |  |
| --- | --- |
| Lights On | Encounter doesn’t allow Undo, so if you goof a later step, you’ll be able to revert to this step. In general, save often with different file names corresponding to the steps you are at so that you can revert to the last good place. If you need to reload a saved design, choose File • Restore Design …. |

### Power planning

* **Go to:** Power • Power Planning • Add Rings… to add the power rings around the cells.
* Set the **width** of the **top**, **bottom**, **left**, and **right** rings to **9.9** (microns) and the **spacing** to **1.8** to provide fat wires that can carry plenty of current to the design.
* Click Center in channel at the Offset option to center the rings in the margin around the rows of cells.

You’ll see the power rings appear in the Encounter window.

### Special route for power planning

* **Go to:** Route • Special Route… to route power and ground to each row and press OK.

Notice that Encounter automatically flips cells between rows and overlaps the power and ground wires to save space. You may wish to save again now.

### Place standard cells

* **Go to:** Place • Standard Cells….
* Turn off all optimization because you don’t want Encounter to modify your design (which would cause LVS errors later).
* Click OK to place the cells in the design.

It will appear that nothing happened. On the right end of the second row of the toolbar, click on the Physical View icon that looks like a transistor (highlighted in Figure 1). This will bring you to a new view in which you can see the gates placed in the rows. Check the console window and look for errors. Ignore warnings about the scan chain because you don’t have one. Encounter has a degree of randomness in cell placement and occasionally fails to place the cells. If you have an error, restore the last saved version and try again. If all is good, save again.



Figure 1: Physical View button

###  Route design

* **Go to**: Route • Nanoroute • Route…
* Be sure to check “Post Route Optimization: Optimize Via” and “Post Route Optimization: Optimize Wire” so the router will produce fewer design rule errors.
* Click OK to route the design.

Check the console to verify that the number of fails is 0 and the number of DRC violations is 0. Again, if it fails, restore and try again. Notice how the cells are routed together and connect to pins scattered randomly around the periphery.

### Add filler cells

* **Go to**: Place • Physical Cells • Add Filler… to add filler cells so there is a continuous nwell even where there are no logic gates.
* Click select, then choose fill\_1\_wide, and click Add. You’ll see the gaps (mostly) filled up.

### Verify connectivity and geometry

* **Go to**: Verify • Verify Geometry… to do a basic design rule check. Make sure there are no violations.
* **Go to**: Verify • Verify Connectivity… to ensure the design is really connected in the way that the structural netlist specified.

Make sure there are no violations.

You are now done with place and route.

* Save once more.
* Then, choose File • Save • DEF to save the output in Design Exchange Format that the Virtuoso Layout Editor will be able to read back in. Change to DEF version 5.5 and click OK. Close Encounter.

# Import the Controller Schematic and Layout

The last step is to import the schematic and layout for the controller back into the Cadence tools.

## Create new library

In Virtuoso, create a new library called lab3\_xx and attach the technology file used in previous labs (UofU AMI 0.60u).

## Import synthesized Verilog

In the Virtuoso window,

* **Go to:** File • Import • Verilog.
* Set the Target Library Name to lab3\_xx.
* Set reference libraries to muddlib11 basic.
* Set the Verilog Files to Import to ~/IC\_CAD/synth/controller\_syn.v.

You’ll see warnings about Verilog definitions for modules not being found. These are OK because the tool uses muddlib11 cells.

Open the controller schematic. You should be able to identify the eight latches and a rat’s nest of gates.

##  Import DEF file from encounter

In the Virtuoso window,

* **Go to:** File • Import • DEF.
* Enter your library (e.g., lab3\_xx), controller for the cell, and layout for the view.
* Enter /home/<yourusername>/IC\_CAD/soc/lab3\_xx/controller.def for the DEF file.

In the Virtuoso window, you should have warnings about failing to open the techfile.cds or the controller layout and viagen layouts and finding the *master* core. Watch for other errors.

In the Library Manager, open the controller layout that you just imported. All of the cells are imported as “abstract” views with just port information but no real layout. You wiil need to use find and replace to change these to “layout” views.

* **Go to:** Tools • Find/Replace, search for inst.
* Then, add criteria of “view name” and search for “abstract” and replace all with “layout” and be sure to save the file.
* Zoom in and inspect the layout that was just produced.
* Run DRC and LVS. There should be no errors.

# 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 aludecoder schematics and layout.
3. Does the aludecoder pass simulation?
4. A printout of the controller schematics and layout.
5. What is the DRC and LVS status of aludecoder and controller?