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 Synopsys's Design Compiler to synthesize the combinational logic for the control FSM and Cadence's SOC Encounter to automatically place and route the controller.

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 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

Create a new library named `controller_xx`. Create an aludec schematic in your `controller_xx` library. Using the logic gates from `muddlib10`, design a combinational circuit to compute the alucontrol[2:0] signals from aluop[1:0] and funct[5:0]. Limit yourself to the inv, nand2, nand3, nor2, and nor3 gates so that you gain experience designing with inverting gates. 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. Remember to name your busses with angle brackets (e.g. aluop<1:0>).

You will need to 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. aluop<0>).

Make a symbol for your aludec.

Next, create an aludec layout. Remember to use metal2 vertically and metal3 horizontally. When you are done, provide pins for vdd!, gnd!, the eight inputs and the three outputs.

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

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 3.[1] The Verilog code describing this FSM is the statelogic and outputlogic modules in the RTL `mips.sv` that you worked with in Lab 2.

Look through the Verilog and identify the major portions. The top level module is called controller. It calls the aludec, which you just designed, and a controller_synth module that you are about to synthesize. The controller_synth module, in turn, has separate modules for the next state logic and the output logic of the FSM. 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 pcen, 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.

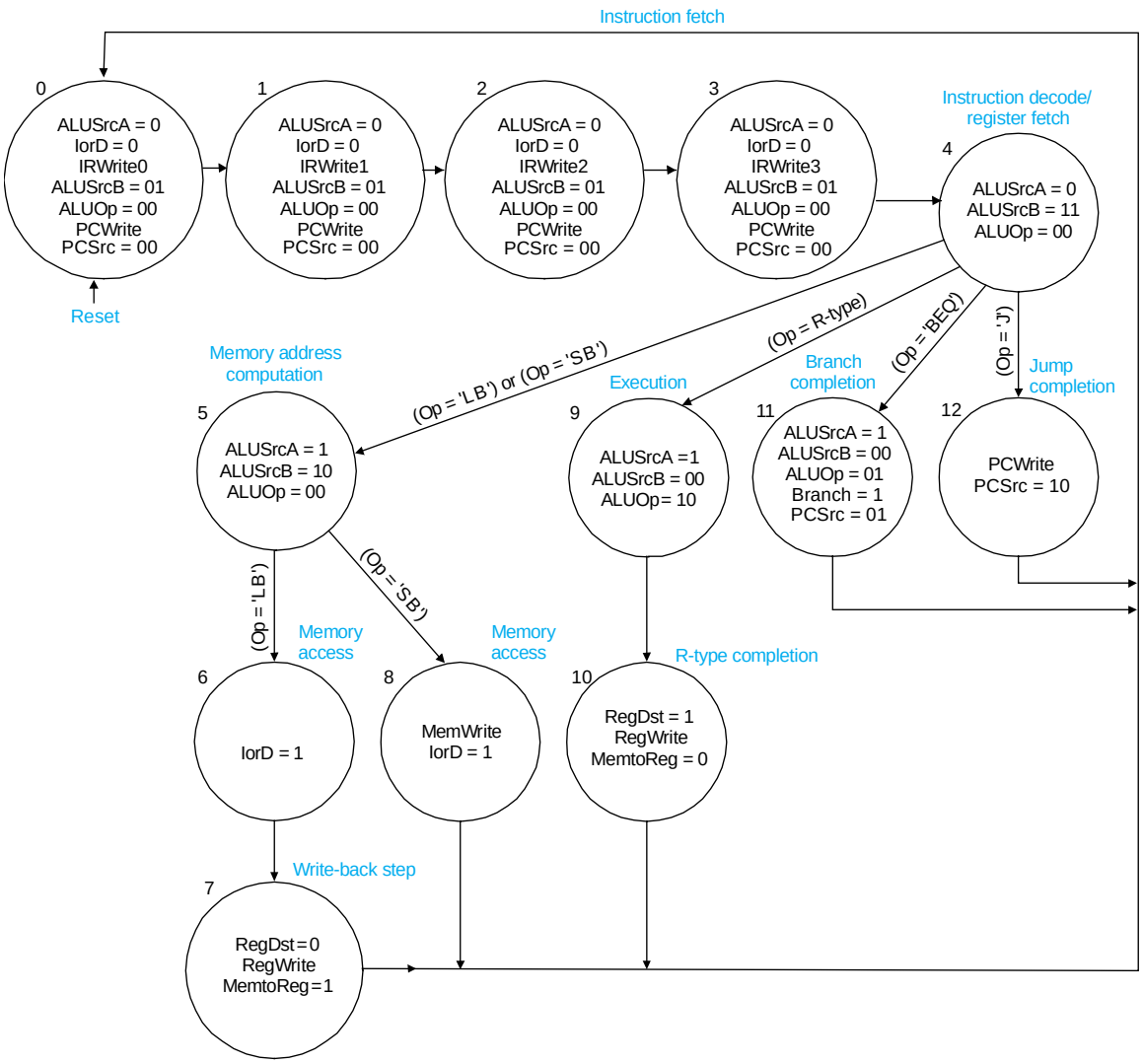

Figure 3: Controller FSM

III. Controller Synthesis

In this section, you will use Synopsys Design Compiler to synthesize the controller_synth 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/e158/10/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/e158/10/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 `mips.sv` and the basename (the module you want to synthesize) to controller_synth. If you were synthesizing something else, you would need to change these lines.

Copy your `mips.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 100 lines of `mips.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_synth_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.

IV. Controller Place & Route with SOC 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.)

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 this new run directory.

You'll 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:

```
ln –s ~/IC_CAD/synth/controller_synth_syn.v .
ln –s ~/IC_CAD/synth/controller_synth_syn.sdc .
```

You'll also need links to `muddlib.lib` and `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.

```
ln –s /courses/e158/10/lab3/muddlib.lib .
ln –s /courses/e158/10/lab3/muddlib.lef .
```

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.

Invoke Design • Import Design. Enter `controller_synth_syn.v` for your Verilog netlist and controller_synth as the top-level cell. Enter `muddlib.lib` as the Common Timing Library, `muddlib.lef` as the LEF file, and `controller_synth_syn.sdc` as the timing constraint file. Then, click on the Advanced tab. Click on Power and enter vdd! as the Power Net and gnd! as the Ground Net.

Watch for errors in the console. You can ignore the max_capacitance attribute warnings if they appear. 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.

Choose 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'll 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.

Use Design • Save Design As • SoCE… and save the design as `controller_synth_floorplan.enc`. 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 Design • Restore Design • SoCE….

Invoke 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.

Invoke 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.

Invoke 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. 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.

Invoke Route • Nanoroute • Route… and 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.

Invoke Place • Physical Cells • Add Filler… to add filler cells so there is a continuous n-well 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.

Invoke Verify • Verify Geometry… to do a basic design rule check. Make sure there are no violations.

Invoke 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 & route. Save once more. Then choose Design • 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.

V. Import the Synthesized and Placed Controller

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

In the icfb window, choose File • Import • Verilog. Set the Target Library Name to `lab3_xx`. Add `mudlib10` to Reference Libraries, along with `basic`. Set the Verilog Files to Import to `~/IC_CAD/synth/controller_synth_syn.v`. You'll see warnings about Verilog definitions for modules not being found. These are ok because the tool uses `muddlib10` cells. Open the controller_synth schematic. You should be

able to identify the eight latches and a rat's nest of gates that make you thankful the FSM was synthesized rather than designed by hand.

In the icfb window, choose File • Import • DEF. Enter your library (e.g. `lab3_xx`), controller_synth for the cell, and layout for the view. Click Use and enter `muddlib10` as the reference library to indicate where the standard cells are taken from. Enter `~/IC_CAD/soc/lab3_xx/controller_synth.def` for the DEF file. In the icfb window, you should have warnings about failing to open the techfile.cds or the controller_synth layout and viagen layouts and finding the master core. Watch for other errors.

In the Library Manager, open the controller_synth layout that you just imported. The design will initially open in "Preview" mode without the usual set of menus. Choose Tools • Layout to return to your familiar layout model.

All of the cells are imported as "abstract" views with just port information but no real layout. You'll need to use find & replace to change these to "layout" views. Choose Edit • Search… Click Add Criteria. Change the criteria to view name. Set it equal to "abstract" Click Apply, and all the abstracts will be selected. At the bottom of the window, choose Replace -> view name -> and enter "layout" in the box. Click Replace All, and close the search box. Zoom in and inspect the layout that was just produced. Be sure to save the file.

Run DRC and LVS. There should be no errors.

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. A printout of the controller_synth schematics and layout.
4. What are the DRC and LVS status of aludec and controller_synth?

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 Synopsys's Design Compiler to synthesize the combinational logic for the control FSM and Cadence's SOC Encounter to automatically place and route the controller.

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 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

Create a new library named `controller_xx`. Create an aludec schematic in your `controller_xx` library. Using the logic gates from `muddlib10`, design a combinational circuit to compute the alucontrol[2:0] signals from aluop[1:0] and funct[5:0]. Limit yourself to the inv, nand2, nand3, nor2, and nor3 gates so that you gain experience designing with inverting gates. 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. Remember to name your busses with angle brackets (e.g. aluop<1:0>).

You will need to 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. aluop<0>).

Make a symbol for your aludec.

Next, create an aludec layout. Remember to use metal2 vertically and metal3 horizontally. When you are done, provide pins for vdd!, gnd!, the eight inputs and the three outputs.

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

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 3.[1] The Verilog code describing this FSM is the statelogic and outputlogic modules in the RTL `mips.sv` that you worked with in Lab 2.

Look through the Verilog and identify the major portions. The top level module is called controller. It calls the aludec, which you just designed, and a controller_synth module that you are about to synthesize. The controller_synth module, in turn, has separate modules for the next state logic and the output logic of the FSM. 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 pcen, 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.

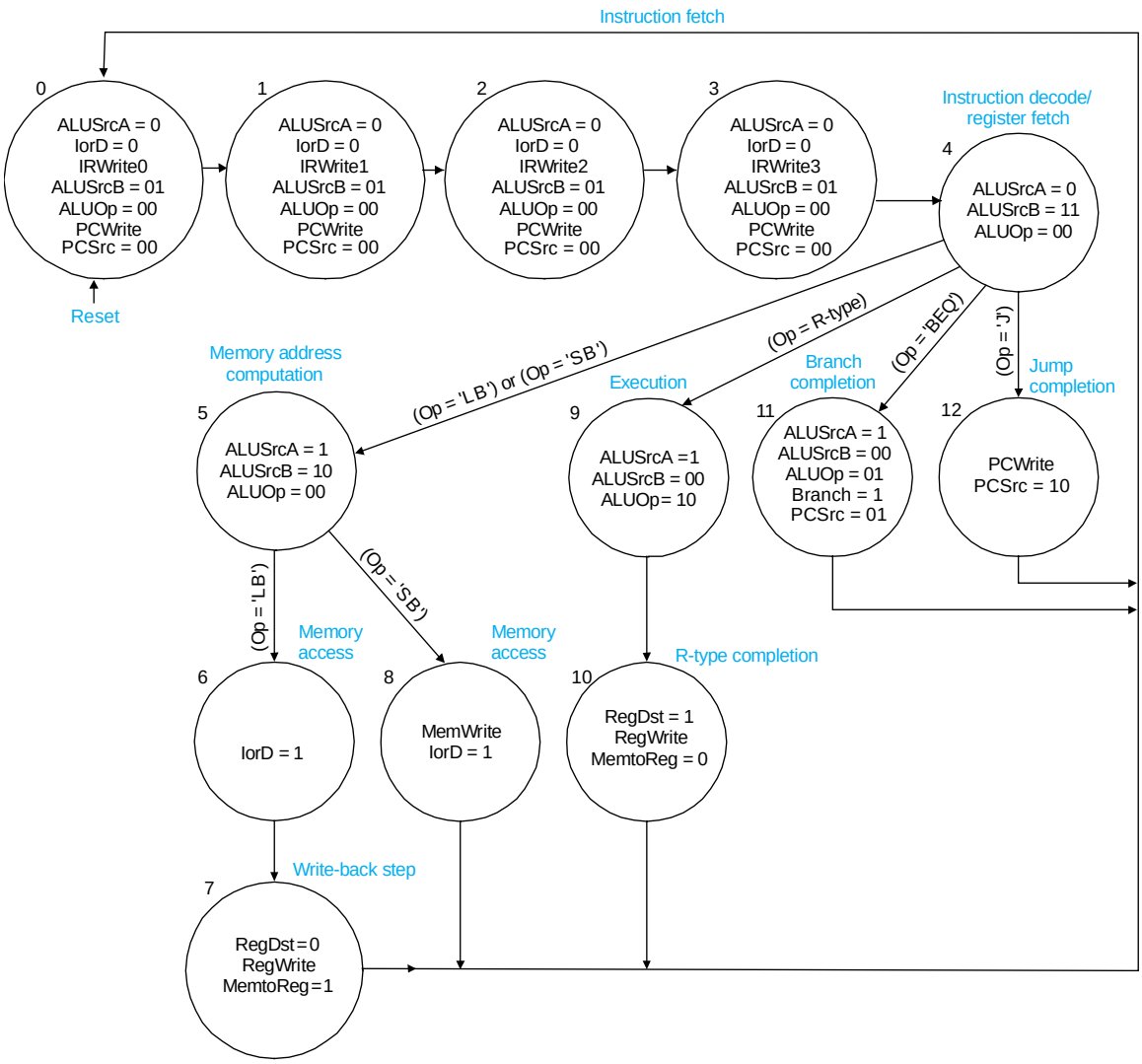

Figure 3: Controller FSM

III. Controller Synthesis

In this section, you will use ~~Synopsys Design~~ Compiler to synthesize the controller_synth 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/e158/10/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/e158/10/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 `mips.sv` and the basename (the module you want to synthesize) to controller_synth. If you were synthesizing something else, you would need to change these lines.

Copy your `mips.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 100 lines of `mips.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_synth_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.

IV. Controller Place & Route with SOC 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.)

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 this new run directory.

You'll 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:

```
ln -s ~/IC_CAD/synth/controller_synth_syn.v .
ln -s ~/IC_CAD/synth/controller_synth_syn.sdc .
```

You'll also need links to `muddlib.lib` and `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.

```
ln -s /courses/e158/10/lab3/muddlib.lib .
ln -s /courses/e158/10/lab3/muddlib.lef .
```

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.

Invoke Design • Import Design. Enter `controller_synth_syn.v` for your Verilog netlist and controller_synth as the top-level cell. Enter `muddlib.lib` as the Common Timing Library, `muddlib.lef` as the LEF file, and `controller_synth_syn.sdc` as the timing constraint file. Then, click on the Advanced tab. Click on Power and enter vdd! as the Power Net and gnd! as the Ground Net.

Watch for errors in the console. You can ignore the max_capacitance attribute warnings if they appear. 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.

Choose 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'll 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.

Use Design • Save Design As • SoCE… and save the design as `controller_synth_floorplan.enc`. 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 Design • Restore Design • SoCE….

Invoke 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.

Invoke 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.

Invoke 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. 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.

Invoke Route • Nanoroute • Route… and 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.

Invoke Place • Physical Cells • Add Filler… to add filler cells so there is a continuous n-well 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.

Invoke Verify • Verify Geometry… to do a basic design rule check. Make sure there are no violations.

Invoke 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 & route. Save once more. Then choose Design • 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.

V. Import the Synthesized and Placed Controller

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

In the icfb window, choose File • Import • Verilog. Set the Target Library Name to `lab3_xx`. Add `muddlib10` to Reference Libraries, along with `basic`. Set the Verilog Files to Import to `~/IC_CAD/synth/controller_synth_syn.v`. You'll see warnings about Verilog definitions for modules not being found. These are ok because the tool uses `muddlib10` cells. Open the controller_synth schematic. ~~You should be~~

~~able to identify the eight latches and a rat's nest of gates that make you thankful the FSM was synthesized rather than designed by hand.~~

In the icfb window, choose File • Import • DEF. Enter your library (e.g. `lab3_xx`), controller_synth for the cell, and layout for the view. Click Use and enter `muddlib10` as the reference library to indicate where the standard cells are taken from. Enter `~/IC_CAD/soc/lab3_xx/controller_synth.def` for the DEF file. In the icfb window, you should have warnings about failing to open the techfile.cds or the controller_synth layout and viagen layouts and finding the master core. Watch for other errors.

In the Library Manager, open the controller_synth layout that you just imported. The design will initially open in "Preview" mode without the usual set of menus. Choose Tools • Layout to return to your familiar layout model.

All of the cells are imported as "abstract" views with just port information but no real layout. You'll need to use find & replace to change these to "layout" views. Choose Edit • Search… Click Add Criteria. Change the criteria to view name. Set it equal to "abstract" Click Apply, and all the abstracts will be selected. At the bottom of the window, choose Replace -> view name -> and enter "layout" in the box. Click Replace All, and close the search box. Zoom in and inspect the layout that was just produced. Be sure to save the file.

Run DRC and LVS. There should be no errors.

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. A printout of the controller_synth schematics and layout.
4. What are the DRC and LVS status of aludec and controller_synth?

Gmail by Google

Sharad Seth <sharadcseth@gmail.com>

lab3 -- synthesis part

Sina Balkir <sbalkir@unl.edu> **Wed, Oct 6, 2010 at 7:29 PM**
To: Sharad Seth <seth@cse.unl.edu>

OK. Lab3 will not be exactly duplicated but to a high degree it will be similar. There are profound reasons for this which are related to specific version of Cadence utilities being used in different schools etc. Anyway, here is what the students should do for lab3.

For section III (synthesis) of lab 3, keeping in mind that we are using "mudd" instead of "IC_CAD":

1- Students first need to copy a new version of mudd.rc from /doppler/cadence/mudd_labs/ into their home and replace the old one with the new one (cp /doppler/cadence/mudd_labs/mudd.rc ~/.) . Since we do not have synopsys rtl compiler we have to use the rtl compiler of cadence, and this new file points into that in addition to previous settings.

2- ignore all the synopsys related instructions in the lab3 handout...

3- instead, we use a new .tcl script for invoking cadence rtl compiler. For this we need a cadence script, which is located at /doppler/cadence/mudd_labs/lab3. Its name is "Start.tcl"

4- Once students create a new directory called "synth" under their mudd directory (mkdir ~/mudd/synth), they will need to copy the file mentioned in step 3 to this directory. Also, copy the "mips.sv" file from lab2 to this directory.

5- Ignore any line in the labs3 handout saying syndc.tcl

6- Instead invoke rtl compiler (from under the newly formed synth directory), by typing the command " rc " from the command line. After a few moments you will be in the system prompt of the Cadence RTL compiler. It will look like this: rc1>

7- From this prompt (under the newly formed synth directory), type the command: source Start.tcl

8- This will invoke the RTL - to - logic synthesis of the mips.sv description.

9- Once successfully executed, the RTL compiler will dump two important files. One is the structural verilog netlist which is synthesized by the RTL compiler and the other one is timing constraint file for place and route. See page 4 of lab3 handouts for the names.

---------- end of synthesis ----------

lab 3 - place and route

Sina Balkir <sbalkir@unl.edu> **Wed, Oct 6, 2010 at 7:51 PM**
To: Sharad Seth <seth@cse.unl.edu>

Ok. This flow will not be 100% duplicated as the versions of tools are not fully matched at different lelves of abstraction. But, we will be able to place and route a std cell layout and call it into the icfb environment -- but no LVS will be performed, which is not really necessary, as an LVS equivalent step is already done in the place route module. Here it is:

Section IV of lab3:

1- create a new directory called soc under the ~/mudd directory.

2- There is a new cds library configuration file placed under /doppler/cadence/mudd_labs, named cds-encounter.lib

3- IMPORTANT!!!: First, this file should *not* be copied over the existing cds.lib file under the ~/mudd directory of each user. Instead, the entries for basic and analogLib should be added to their existing cds.lib file without altering anyting else. This can be done by a text editor like vi or else.

4- Follow the steps of page 4 and page 5 of lab3, keeping in mind that IC_CAD is mudd in your case.

5- Also, /courses/e158/10/lab3 path should be changed to /doppler/cadence/mudd_labs/lab3 on page 5

6- Also, we use the command "encounter" from the system prompt instead of "cad-soc".

7- Rest of the flow on page 5 and page 6 until section V should work as is.

------- end of place and route -----------

Lab 3 -- importing into the virtuoso environment (section V)

Sina Balkir <sbalkir@unl.edu> **Wed, Oct 6, 2010 at 8:27 PM**
To: Sharad Seth <seth@cse.unl.edu>

OK. This is the last step where we will have significant differences.

First, pay attention to the author's typo in writing the mudlib10; it should be muddlib10

Second, there is a profound logistic issue, when the author says to set the target library name to lab3_xx toward the bottom of page 6. This is not quite that simple. First, the lab3_xx target library is *not* the lab3_xx directory. We first have to create a new library named lab3_xx using the library manager tool of the icfb main command window and attach to it the UofUtah AMI0.5 tech library (similar steps found in lab1). Without this, the rest of the procedure will fail. To do this, first go to your top level mudd directory (cd ~/mudd), but do *not* go to soc or synth etc.

Once a new library named lab3_xx is formed by using the new library function of the icfb window, under ~/mudd, you can then invoke icfb from under the top level mudd directory as usual and follow the steps in the last paragraph of page 6.

IMPORTANT: due to a profound difference in the assign statements of the structural verilog, the schematic formation step fails to perform. So, will skip this part but it is good practice to see how it fails.

Next, on top of page 7 of lab3, execute the import steps as instructed and you should be able to see the layout eventually.

Finally, the DRC may work but LVS will fail as there is no schematic to compare against. However, keep in mind that in the encounter there was already a verification of the connectivity done so we should be OK without this step.

Note: If all the steps are done with being careful and with no typing mistakes etc. these steps should work. -- well, mine did :-) Hope this helps

-------end----------