CPEN 211 2022W1

LAB 3

DUE: `handin` submission 2022-10-23 23:59 Vancouver time; lab demo: Week 8

LOGISTICS

Partners. Lab partners (groups of 2) allowed but optional. To work with a partner you must first **register them as a partner** using `https://cpen211.ece.ubc.ca/cwl/lab_partners.php`. The deadline to sign up a partner is **96 hours before the handin deadline**. If you miss this deadline you must do the lab alone.

Submission. You must submit your code using `handin` using "Lab3" by the `handin` deadline above or your mark for this lab will be **zero**. You can find `handin` instructions on Piazza. There are template files for the design and testbenches on Piazza. All of the files you submit must have the **exact file names** required by each task below, and must be in the toplevel submission folder (no subfolders). Submitted files may be stored outside of Canada, so you should omit personal information.

If you are working with a partner, your submission **must** include a file called `CONTRIBUTIONS.txt` that describes in detail each partner's contributions to the submission. If you are missing this file, you may lose all marks in the lab, and if either partner contributed less that one third of the effort they may lose marks.

Lab demos. You will need to demo your testbench, design RTL, simulations, and working DE1-SoC to the TA who is marking you. This means you need to either bring your laptop which you can use to do this, or make sure you can use one of the lab computers. Either way, be prepared and ready to show everything when it's your turn — the TAs have many students to mark, so they don't have enough time to wait around for you to, for example, figure out how to use the lab computers.

If you are working with a partner, you **must both be present** to receive any marks for this lab.

The marks assigned by the TA are subject to validation against the code you submitted on `handin`. It is considered academic misconduct to demo different code than you submitted via `handin`.

Academic integrity. This lab **must be done individually or with the registered partner** (see above). You may not communicate with any other students about this lab. Before starting the lab, you must also read the Academic Integrity Policy (available on Piazza), and you must comply with it.

SPECIFICATIONS

In this lab you will design a digital combination lock and implement it using your DE1-SoC board.

To complete the lab, you will need to install Modelsim and Quartus using the procedure from the Verilog + FPGA tools tutorial on Piazza. This tutorial also shows you how to download a synthesized design to your DE1-SoC using a simple example.

First, let's go over the elements on the board:

- Each button, switch, and seven-segment LED display on the DE1-SoC boards has a tiny label right next to it that says KEY*n*, SW*n*, or HEX*n* (where *n* is some digit). We will use these names to refer to specific buttons, switches, etc.

- You will need to output messages to the seven-segment LEDs; these should be read left-to-right and right-aligned **with the board oriented so that the switches are on the bottom**. For example, to display CPEn, HEX3 will show C, HEX2 will show P, HEX1 will show E, and HEX0 will show n.

Your design must meet the following specification:

- The lock combination will be the **last six digits** of your UBC student number. If you are working with a partner, use the last six digits of Partner #1's student number. You can look up who Partner #1 is using `https://cpen211.ece.ubc.ca/cwl/lab_partners.php`.
- To open the lock, one resets the lock and then enters six digits, one at a time, from left to right. So if your student number were 12345678, you would enter 3, then 4, then 5, then 6, then 7, and finally 8.
- To reset the lock press KEY3 (the active-low reset). Keep holding KEY3 (reset) pressed while next pressing and releasing KEY0 (clock), then release KEY3 (reset). (Ask yourself: does this describe a synchronous or an asynchronous reset?)
- To enter each digit, set the switches SW3. . .SW0 to the binary representation of the digit and then press and release KEY0 (the clock). For example, if the digit is 5, you would set SW3=0, SW1=1, SW2=0, and SW3=1; then you would push KEY0 to commit the digit.
- Repeat the steps in the point above to enter all six digits.
- As each digit is being entered, the *current* value on SW3 to SW0 must immediately be displayed as a *decimal digit* on HEX0 (HEX5 to HEX1 must be off). If the current SW3. . .SW0 setting does not correspond to a decimal digit (0–9), then Error must be displayed on HEX4 to HEX0 (and HEX5 must be off). For example, if SW3. . .SW0 are 1001, HEX0 must show 9.
- Note that this is the *current* digit you're entering *right now* (before you press KEY0), not the *last* digit you entered. That is, as you change SW3 through SW0 to set up the next digit in the lock combination, several different numbers may be shown on HEX0 and/or the message Error might be displayed before you get all the switches in the right position.
- After the sixth digit is entered on SW3 through SW0 and KEY0 is pressed, the lock decides whether it should open. If the correct sequence of digits was entered then OPEn must be displayed on HEX3 to HEX0; otherwise CLOSEd must be shown on HEX5 to HEX0.
- Your design **must not** require the user to press KEY0 any extra times before the output is updated, beyond what is specified above. So after you reset the circuit, OPEn or CLOSEd must appear after six presses of KEY0 (one for each digit), not counting any presses while KEY3 (reset) was asserted.

IMPLEMENTATION OVERVIEW

Figure 1 illustrates the overall system. The push button KEY0 is the clock input and KEY3 is used for the reset input.

On the DE1-SoC, the push-buttons (KEY0, KEY3, etc.) are *active-low*: that is, they are 0 when the button is *being pressed*, and 1 when the button is *released*.* This is fine for reset (active-low reset is common). For the clock, however, it would be convenient for the rising edge to occur when the button is *pushed* rather

*If you are curious why, download the DE1-SoC manual from `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836&PartNo=4` and look at figure 3–14 on page 23.

than released. For this reason, we will add an inverter to the clock input; this is already done for you in the template `lab3_top.sv`.[†]

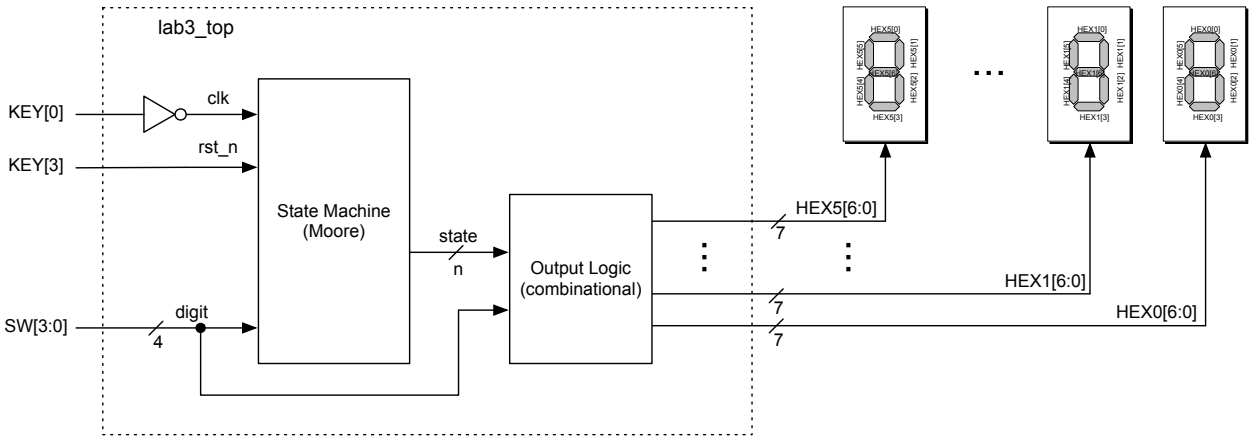


Figure 1: The circuit you will design for Lab 3

The block diagram in fig. 1 shows two parts: (i) a Moore state machine and (ii) a combinational block for the output logic that is connected to both the inputs and the current Moore machine state. In this approach, the Moore state machine checks that the sequence of values input on `SW[3:0]` matches the combination, and the output of the Moore state machine along with the values of `SW[3:0]` are fed to the output logic, which in turn displays values on the seven-segment LEDs. The output of the Moore machine is shown as n-bits-wide; you will decide the value of n based on your own design.

In combination, these form a Mealy state machine in which the output displayed on the seven-segment LEDs depends on both the current state and the current inputs. It is *recommended* that you split your solution into these two blocks, but you are free to implement your design in any way you like provided the resulting system meets the specification above.

Example

Figure 2 illustrates an example Mealy machine state diagram matching the specification assuming the last six digits of your student number are "122334."

Each edge in the figure is labelled with the input required to transition along a given edge. For example the edge "$= 4$" means that the transition occurs if the input on `SW[3:0]` is the binary representation for decimal number four; similarly, the label "$\neq 4$" means that the transition occurs if the input on `SW[3:0]` is *not* the binary representation for decimal number four. If there is no label on the transition, the transition is unconditional (i.e., happens at the next clock cycle).

Inside each state bubble is the output that should be showing on the seven-segment displays. See the specification above for details.

[†]In a real design we would *never* manually add any logic to a normal clock signal, because the clock network on a real chip is autogenerated by the synthesis tools to minimize clock skew, and logic gates screw this up. We're only doing this for this lab because we drive the clock manually, and then only in the outermost module.

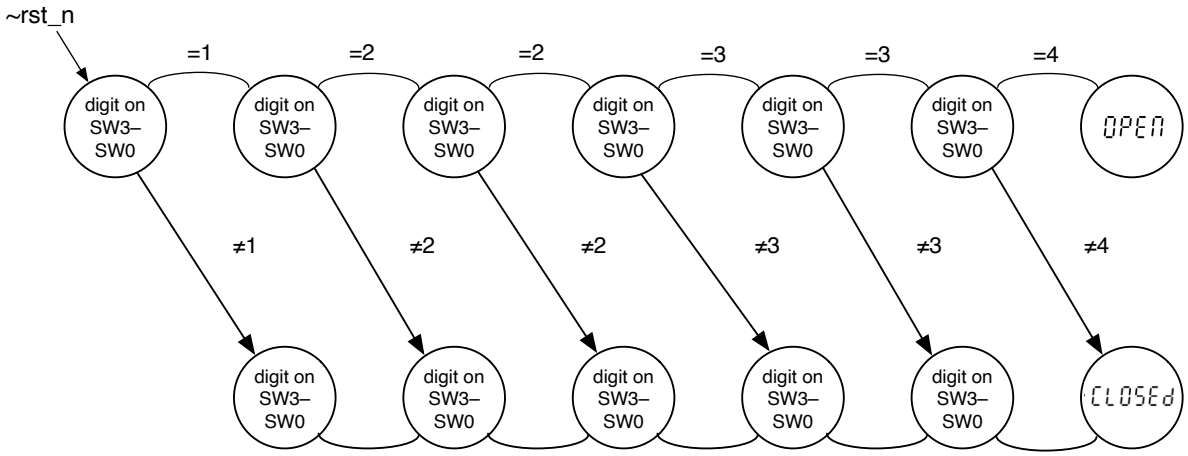


Figure 2: Sample Finite State Machine

LAB PROCEDURE

In this lab you will:

- develop a testbench in SystemVerilog based on the specification above;
- design a digital circuit in synthesizable SystemVerilog;
- simulate it *at RTL-level* in Modelsim, using your testbench to ensure the behaviour meets the specification;
- synthesize your SystemVerilog design to obtain a gate-level Verilog Output File (.vo);
- simulate your design *at gate-level* in Modelsim, using your testbench; and
- program the FPGA on the DE1-SoC with your synthesized design.

If your SystemVerilog follows the synthesis rules outlined in the lectures, the RTL-level and gate-level simulation results **should match**. If they do not, this means that you have written RTL that is *not synthesizable*, and you need to go check your RTL.

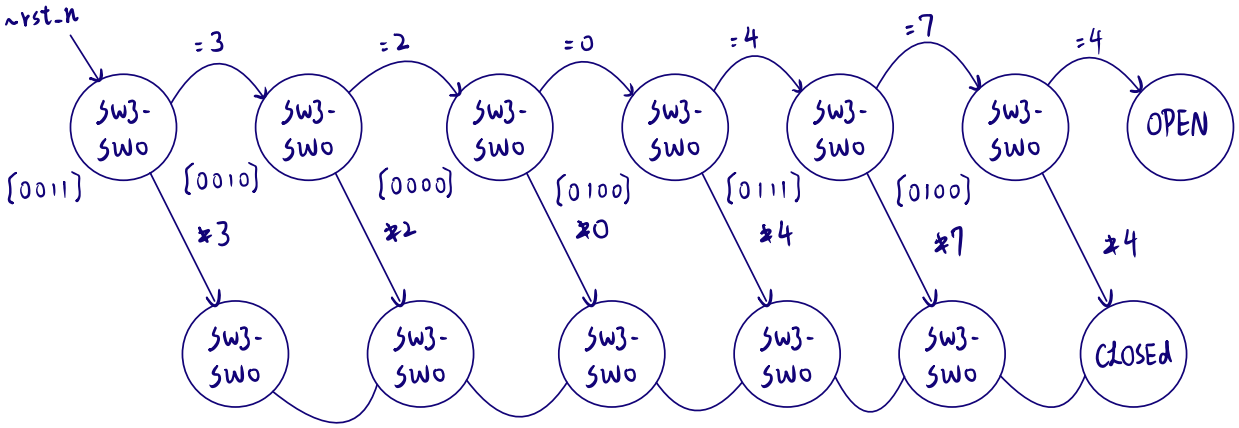
Task 1: Testbench [2 marks]

In the file `tb_lab3.sv`, develop a testbench for the design based on the Specification section above.

Note that it's not enough to generate clock and reset — at a minimum, your testbench must test not only the success case but also all possible ways to fail (i.e., failing after inputting every possible number of digits). It should also test that the HEX displays show the correct outputs not only in the success and fail states, but also how the HEX displays change as you input digits using the switches.

For full marks, your tesbench must also automatically check and report whether each test case succeeds or fails — that is, it should not require you to manually check the lock outputs and state in the Modelsim waveform to determine whether individual tests passed. Refer to the lectures to see how a testbench can do that.

SN : 74320474 [0011] → [0010] → [0000] → [0100] → [0111] → [0100]



SW				HEX0							
3	2	1	0	6	5	4	3	2	1	0	
0	0	0	0	1	0	0	0	0	0	0	□
0	0	0	1	1	1	1	1	0	0	1	1
0	0	1	0	0	1	0	0	1	0	0	2
0	0	1	1	0	1	1	0	0	0	0	3
0	1	0	0	0	0	1	1	0	0	1	4
0	1	0	1	0	0	1	0	0	1	0	5
0	1	1	0	0	0	0	0	0	1	0	6
0	1	1	1	1	1	1	1	0	0	0	7
1	0	0	0	0	0	0	0	0	0	0	8
1	0	0	1	0	0	1	1	0	0	0	9
1	0	1	0								
1	0	1	1								
1	1	0	0								
1	1	0	1								
1	1	1	0								
1	1	1	1								

HEX	Error
4	0000110
3	0101111
2	0101111
1	0100011
0	0101111

OPEN :

HEX3 1000000
HEX2 0001100
HEX1 0000110
HEX0 1001000

CLOSED

HEX5 1000110
HEX4 1000111
HEX3 1000000
HEX2 0010010
HEX1 0000110
HEX0 0100001

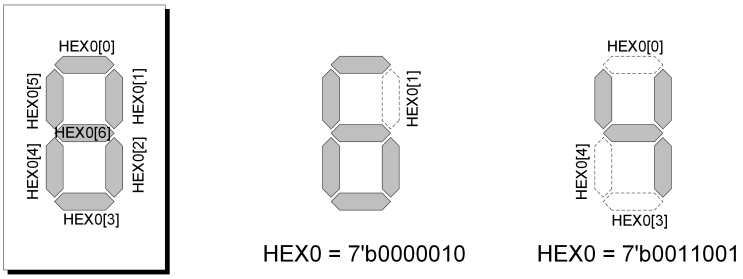


Figure 3: Examples of drawing numbers with HEX0.

Your testbench must be **entirely** within `tb_lab3.sv`. You must not modify the module name or the port list.

Handin deliverables: `tb_lab3.sv`.

Lab section deliverables: Show and explain your tests and testbench to the TA. Answer any questions the TA has for you.

Task 2: Hardware lock [2 marks]

In the file `tb_lab3.sv`, design hardware using Verilog that implements the circuit as described in the Specifications.

Keep in mind during this step is that when using synthesizable Verilog it is not possible to describe a Mealy machine using a *single* always block — this is because you can't mix combinational and sequential always-blocks. One solution is to break up the design into separate hardware blocks, as suggested in the implementation overview above.

Note that the reset described in Specifications is *synchronous* and *active-low*. This means that when the reset signal is asserted when it is at logic 0, and that the registers reset on every rising edge of the clock while the reset is being asserted.

All sequential logic in your design must be must operate on the rising edge of the clock, i.e., `@(posedge clk)`, never on the falling (negative) edge.

Your design must be **entirely** within `lab3.sv`. You **must not** modify the module name or any of the port names, widths, directions, or types in the module declaration.

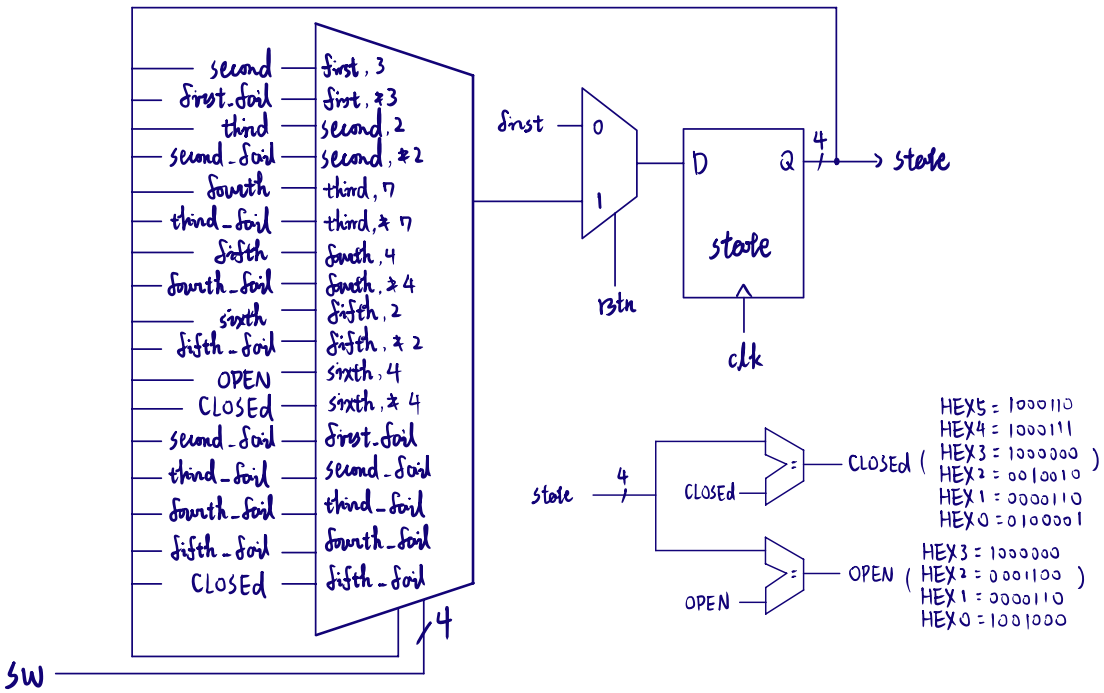
Handin deliverables: `lab3.sv`.

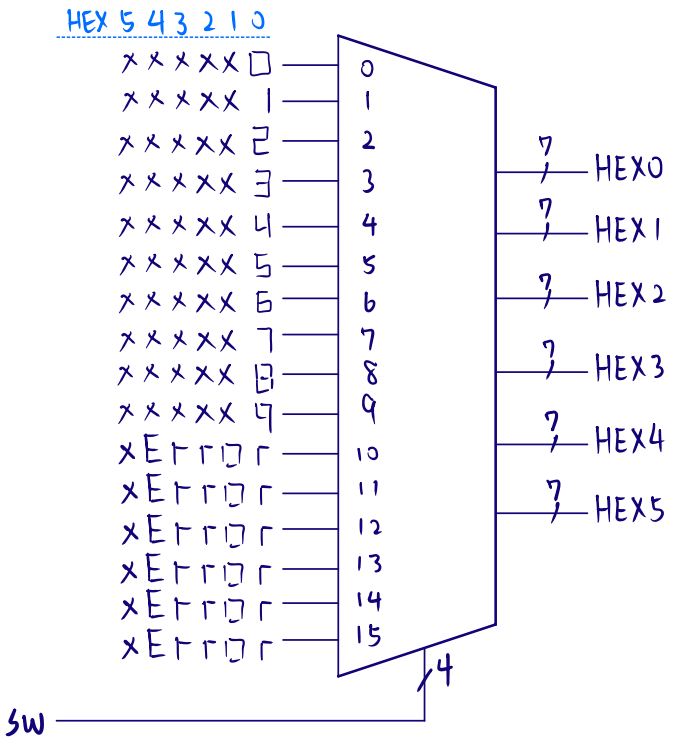
Lab section deliverables: Show and explain your hardware design (including any FSM diagrams) and how it is coded in RTL to the TA. Answer any questions the TA has for you.

Task 3: RTL-level simulation [2 marks]

Simulate your testbench and design at RTL level in Modelsim, your waveform format in file `lab3_wave_rtl.do`, and the transcript of your simulation session (i.e., the output of all `$display`, `$monitor`, or `$error` statements and such). The simulation must show all of your testcases.

Handin deliverables: `lab3_wave_rtl.do`.


second
first_foil
third
second_foil
fourth
third_foil
fifth
fourth_foil
sixth
fifth_foil
OPEN
CLOSED
second_foil
third_foil
fourth_foil
fifth_foil
CLOSED
first, 3
first, *3
second, 2
second, *2
third, 7
third, * 7
fourth, 4
fourth, *4
fifth, 2
fifth, * 2
sixth, 4
sixth, * 4
first_foil
second_foil
third_foil
fourth_foil
fifth_foil
4
SW
first
0
1
rstn
D
Q
4
state
state
clk
state
4
CLOSED
CLOSED
OPEN
OPEN
HEX5 = 1000110
HEX4 = 1000111
HEX3 = 1000000
HEX2 = 0010010
HEX1 = 0000110
HEX0 = 0100001
HEX3 = 1000000
HEX2 = 0001100
HEX1 = 0000110
HEX0 = 1001000


HEX 5 4 3 2 1 0
XXXXX0
XXXXX1
XXXX2
XXXX3
XXXX4
XXXX5
XXXX6
XXXX7
XXXX8
XXXX9
XXXXA
XXXXB
XXXXC
XXXXD
XXXXE
XXXXF
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
4
SW
7
HEX0
7
HEX1
7
HEX2
7
HEX3
7
HEX4
7
HEX5

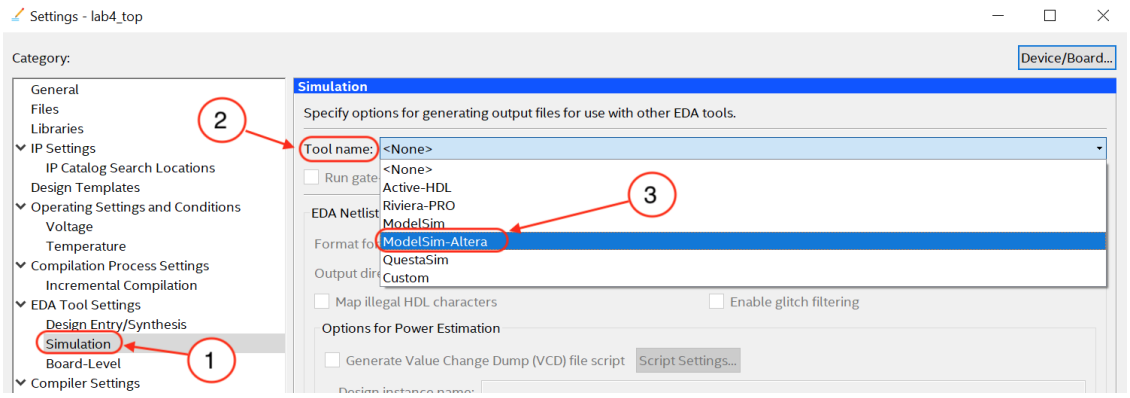


Figure 4: Configuring Quartus to generate a gate-level netlist Verilog Output File (.vo)

Lab section deliverables: Demonstrate to the TA that your design works and that your testbench fully tests your design. This means you need to show how you run Modelsim on your testbench+design. Answer any questions the TA has for you.

Task 4: Gate-level simulation [2 marks]

Modelsim simulates RTL by directly interpreting the Verilog, but Quartus is a *synthesis* tool that converts your RTL to real hardware.

In this task you will get Quartus to synthesize your design and generate a *gate-level netlist* representation. This is an intermediate step in the synthesis process: first the RTL is converted to a netlist (in structural Verilog), and then the netlist is converted to a bitstream suitable for your FPGA.

This Verilog will look *very* different from the Verilog you wrote for your design because it is the result of synthesis and a direct (but somewhat human readable) representation of the circuit that will be downloaded to your DE1-SoC when you program the device.

If you open up the ".vo" file generated in this step you will notice the "gates" in this netlist are "lookup tables" (e.g., "`cyclonev_lcell_comb`") instead of AND, OR and NOT gates — this is because FPGAs implement combinational logic by directly loading the truth table values into a small memory called a lookup table.

To start, copy your RTL testbench to `tb_lab3_gate.sv`. If you tested any internal signals in your modules (e.g., via hierarchical name references), you will have to remove those for the gate-level testbench, because these names will no longer exist in the gate-level netlist. Otherwise if you only interacted with the module via the ports, your gate-level testbench can be exactly the same as your RTL testbench.

Create a Quartus project and put your synthesizable Verilog (`lab3.sv`) into it; do not include your testbench. Next, go to "*Assignments → Settings*". In the dialog box that opens, select "*Simulation*" under "EDA Tools Settings." Then in the drop-down menu next to "*Tool name:*" select "*Modelsim-Altera*", then press "OK" (see fig. 4).

Next, recompile your design in Quartus. Quartus will generate a file with your top level module name and a ".vo" extension ("`lab3.vo`") in the directory "`simulation/Modelsim`."

Next, create a new project in Modelsim (different from the one you created in Step 2), and include just your testbench from Step 2 and the ".vo" file generated above (e.g., "`lab3.vo`"). Note that you will need to include the line "`‘timescale 1 ps/ 1 ps`" at the top of your testbench file for this step (you can leave it in for demoing Step 2). Do not include your original synthesizable verilog (e.g., "`lab3.sv`").

Compile the files. To simulate the output netlist from Quartus requires using low level libraries that model the "gates" provided on the Cyclone V FPGA on your DE1-SoC. The easiest way to do this is by starting simulation by typing the following in the transcript window. Since your testbench module is called "`tb_lab3_gate`" you would type:

```
vsim -t 1ps -L cyclonev_ver -L altera_ver -L altera_mf_ver \
     -L 220model_ver -L sgate_ver -L altera_lnsim_ver tb_lab3_gate
```

Replace the last part, "`lab3_tb`" with the name of your testbench module. The flag "`-t 1ps`" specifies the simulator time resolution is 1 picosecond. The "`-L`" flag specifies a library. As you can see, we need to specify several libraries (e.g., `cyclonev_ver`, `altera_ver`, etc…). The "\" allows us to break the single command over multiple lines. Leave "\" out if you type the above command on one line or you will get an error.

Add some signals from your testbench to the waveform viewer then run the simulation by typing "`run -all`".

Save your waveform format in a file `lab3_wave_gatelevel.do`.

Handin deliverables: `lab3_wave_gatelevel.do` and `tb_lab3_gate.sv`.

Lab section deliverables: Demonstrate to the TA that your design synthesizes correctly and that your testbench fully tests your design at gate level. This means you need to show how you run Modelsim on your testbench+netlist. Answer any questions the TA has for you.

Task 5: DE1-SoC [2 marks]

Demonstrate the design working on your DE1-SoC.

Handin deliverables: None.

Lab section deliverables: Download your hardware design works in the DE1-SoC and demonstrate to the TA that it works on the board. Answer any questions the TA has for you.

HINTS AND TIPS

Do the Verilog+FPGA tutorial before starting on this lab. It will make your life much easier.

Also watch the Modelsim debugging videos on Canvas. If you know how to effectively debug RTL using Modelsim, you will spend a lot less time doing that.

You can have the Modelsim waveform viewer display your state signals using symbolic names rather than binary suing the "`radix define`" command. For example, if your FSM has states "`Foo`", "`Bar`", and "`Baz`" encoded as `2'b00`, `2'b01` and `2'b10` respectively, you could type:

```
radix define MyStates { 2'b00 "Foo", 2'b01 "Bar", 2'b10 "Baz", -default bin }
```

in the transcript window. Then, right click on your state signals in the waveform viewer and select radix you can then select “MyStates”. Your states will show up in the waves as “Foo”, “Bar” or “Baz” instead of as binary values. More information on Page 53 here.

To avoid having to retype Modelsim commands, you can place them in a .do file and load it from the transcript window. For example if if you create a file `names.do` in your Modelsim project directory with the contents above then in the transcript window you can run the commands by typing: “`do names.do`”

After getting your RTL to compile in Modelsim, but before running any simulations in Modelsim, it is worth it compile your synthesizable modules in Quartus just to look at the warnings. Quartus provides useful warnings for many silly mistakes that Modelsim happily ignores. If you see no suspicious warnings in Quartus, then move on to simulating your testbench in Modelsim.

When using “`$stop;`” in a testbench and running with “`run -all`” in Modelsim a source window will pop-up when “`$stop`” is reached. If you use a text editor other than Modelsim to edit your files (e.g., vi or emacs), make sure to close this window before restarting simulation. If you for any reason modify your testbench outside of Modelsim (perhaps to add a test case) and you then restart simulation you will get a long set of about 50 pop-ups saying the file was modified outside of Modelsim. If you forget and this happens, note you can likely close these roughly 50 dialogs faster then restarting Modelsim by clicking on “Skip Messages” then selecting “Reload” repeatedly.