DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2024

Lab 3: Combinational Logic and State Machine Design Using Verilog

Week of Oct 21 to 25 (code must be submitted by 9:59 PM the evening before your lab session)

IMPORTANT NOTES:

- Lab partners allowed but optional. To work with a partner you must be in the same lab section and first register 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 start of your Lab 3 marking session. If you miss this deadline you must do the lab alone.
- If you require help attend TA help sessions and/or post a **private** message to instructors on Piazza.
- Any discussion of this lab with a student who is not your registered partner for Lab 3 must first be registered using https://cpen211.ece.ubc.ca/cwl/student_register_peer_help.php and must conclude before either you or the other student have started coding a solution for this lab.
- You are not permitted to share or even describe your code with other students until after all students involved have received a grade. After grades are received by both parties, sharing of code must still be registered using the above URL.
- **Either you or your partner MUST submit your code using github classroom by the deadline above or your mark for this lab will be zero. Emailed submissions will NOT be accepted.**
- **If you have never used git or github before, expect it to take several hours to learn and plan accordingly to ensure you can submit something by the deadline. NOT KNOWING HOW TO USE git OR github WILL NOT BE ACCEPTED AS AN EXCUSE FOR LATE SUBMISSIONS.**
- All submitted code will be checked for violations of the CPEN 211 Lab Academic Integrity Policy using software plagiarism detection tools.

1 Introduction

In this lab you design a state machine and connect it to the seven segment LEDs on the DE1-SoC via a combinational logic block.

Before you get started, ensure you learn how to access github classroom so you can download the Lab 3 starter code and more importantly, be prepared to submit your code when you are done. Depending upon your prior experience, it may take much longer than you expect to learn how to use git and github.com. To obtain the starter code do the following: If you do not already have one, sign up for a github account (the "free" one is all you need). Next, enter your github username using https://cpen211.ece.ubc.ca/cwl/github_info.php and then press "Submit". After submitting the page should refresh and a table with a github classroom "invite link" specific to your Lab 3 section will appear at the bottom of this same page. Click on the invite link and follow the instructions to setup your Lab 3 submission repository ("repo") on github.com. To submit your code you will need to "commit" any changes AND "push" these committed changes to github.com. After you have done this you should see the changes to the code in the repo generated using the invite link when you look at those files on github.com. Lab 3 has Visual Code Studio integration enabled on github classroom which might simplify some steps. Regardless of which tool you use you must be sure to verify you can see your submitted code on github.com before the submission deadline. If you are new to git and github you can learn about both from various resources online.

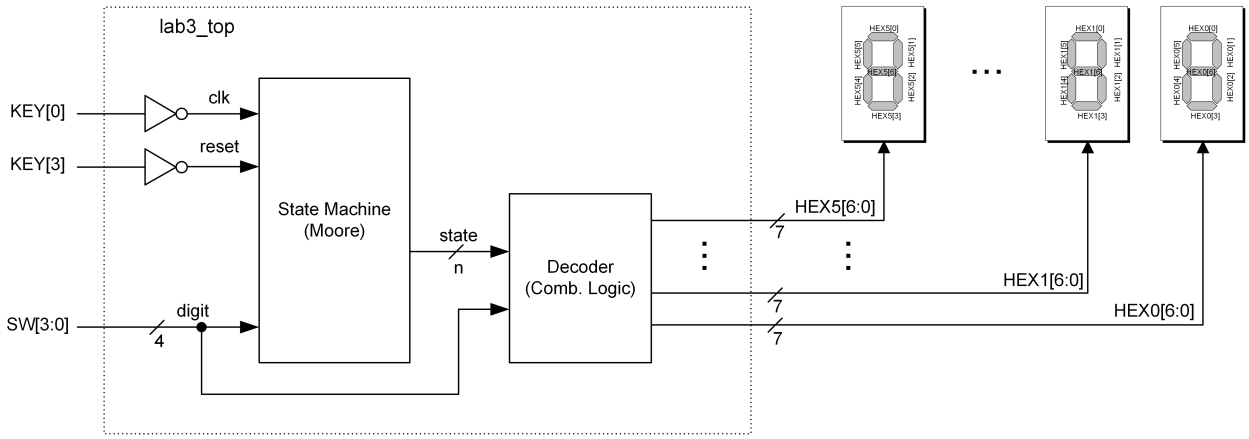

Figure 1: The circuit you will design for Lab 3

1.1 Specification

You will design a digital combination lock and implement it using your DE1-SoC. 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 lookup who Partner #1 is using https://cpen211.ece.ubc.ca/cwl/lab_partners.php.

To open the lock one resets the lock, then enters six digits, one at a time. To reset the lock press KEY3 and then keep holding KEY3 pressed while next pressing and releasing KEY0, then release KEY3. The procedure for entering an individual digit is to convert it from decimal to binary, then enter the binary code using SW3 to SW0, then press and release KEY0. Repeat the steps in the prior sentence six times to enter all digits. As each digit is being entered the value on SW3 to SW0 should immediately be displayed on HEX0 (HEX5 to HEX1 should be off). If the code does not correspond to a decimal digit (0-9), then "ErrOr" should be displayed on HEX4 to HEX0 (HEX5 should be off). Thus, as you change SW3 through SW0 to correspond to the next digit in the lock combination, multiple different numbers may be shown on HEX0 and/or the message "ErrOr" might be displayed. (By displaying the decimal version of the binary code on the slider switches immediately, the user is less likely to make an error albeit someone could then see the combination as it is entered.) After the sixth digit is entered on SW3 through SW0 and KEY0 is pressed, if the correct sequence of digits was entered then "OPEn" should be displayed on HEX3 to HEX0 and otherwise "CLOSED" should be shown on HEX5 to HEX0. Your design must not require the user to press KEY0 extra times before the output is updated, beyond what is specified above.

Figure 1 illustrates the overall system. The push button KEY0 is the clock input and KEY3 is used for the reset input. It is important to note that on the DE1-SoC push buttons (e.g., KEY0, KEY3) have inverted logic: When pressed the value is 0 and when released the value is 1. For this reason we include NOT gates between KEY0 and clock and between KEY3 and reset. In Figure 1 we show a Moore state machine the output of which is connected to a combinational logic block along with the digit inputs. The combination of these two blocks forms 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 input/output specification outlined in the paragraph above. In the suggested approach, the Moore state machine checks that the sequence of values input on `SW[3:0]` matches the combination. The output of the Moore state machine along with the values of `SW[3:0]` are input to a decoder which in turn displays values on the seven-segment LEDs. In Figure 1 the output of the Moore machine is n-bits wide where n is a parameter you decide based upon your own design.

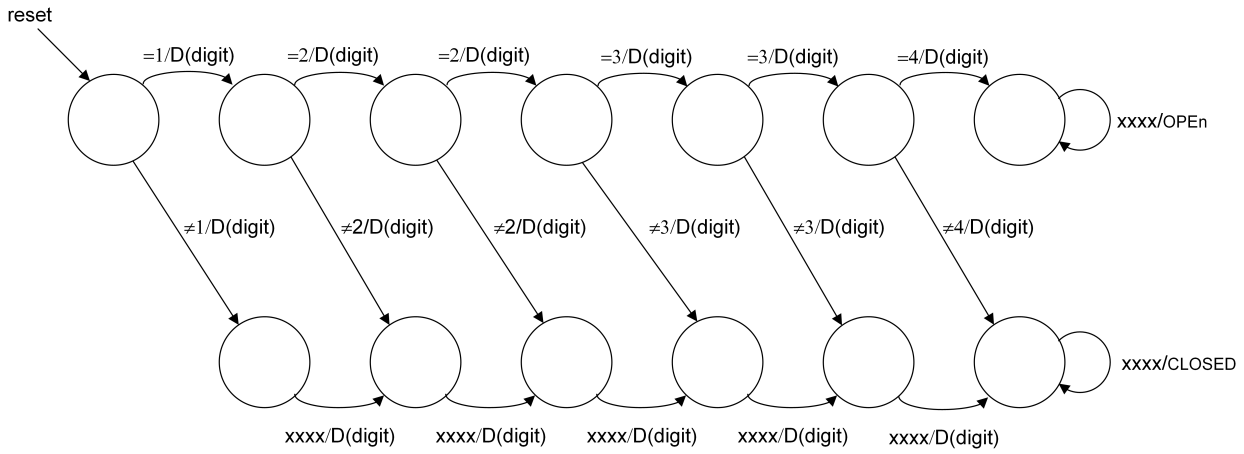

Figure 2: Sample Finite State Machine

2 Lab Procedure

In this lab you first design a digital circuit in Verilog, then simulate it *at RTL-level* in ModelSim with a testbench to ensure the behavior described is error free, then use Quartus to synthesize the Verilog to obtain a gate-level Verilog Output File (.vo) and simulate *at gate-level* in ModelSim. If your Verilog follows the synthesis rules outlined in the lectures, the results of these simulations should match. This is good because it is generally easier to find the sources of errors in simulation than after you build a chip (or download the design to an FPGA). Next, you will download your synthesized design to the FPGA your DE1-SoC. If you have not already done so, you should install ModelSim and Quartus using the procedure outlined in the tools tutorial, which also shows you how to download a synthesized design to your DE1-SoC using a simple example.

2.1 Step 1: Hardware Design

Design hardware using Verilog that implements the circuit as described in Section 1.1. Figure 2 illustrates an example Mealy machine state diagram matching the specification assuming the last six digits of your student number are "122334". Here the edges are labeled with the input required to transition along a given edge followed after a slash ("/") by the output that should be displayed. The input conditions test the input "digit" for equality with a given digit from your student number. For example the label "$= 4$" means if the input on `SW[3:0]` is the binary representation for decimal number four, then follow the corresponding transition. Similarly, the label "$\neq 4$" means if the input on `SW[3:0]` is not equal to the binary representation for decimal number four, then follow the corresponding transition. The output on some transitions is "D(digit)", which is shorthand to indicate HEX0 should display the value on `SW[3:0]` as a decimal number or HEX4 to HEX0 should display "ErrOr" as per the specification. The label "xxxx" means the transition should be followed regardless of the value on `SW[3:0]`.

An important consideration to 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 "Type 1" does not have memory and "Type 2" only updates the outputs on a clock edge whereas a Mealy machine has memory and may update its output immediately in response to changes in the inputs. The solution is to break up the design into separate hardware blocks. As suggested in Section 1.1, one approach is to split the circuit into a Moore state machine and a separate combinational logic block. Recall for a Moore machine the output depends only upon the current state and the state only changes on a rising edge of the clock, which means the output will only change on the rising edge of a clock. In this case you could create a "Type 2" always block for the Moore machine such as the examples in Slide Set 7. The combinational logic block could

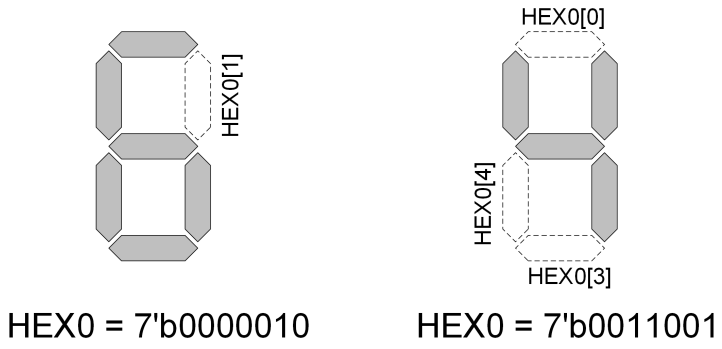

Figure 3: Examples of drawing numbers with HEX0.

then be implemented in a separate "Type 1" always block or by some combination of assign statements, module instantiations and "Type 1" always blocks. Using this approach the Moore machine would have the same states and transitions as the Mealy machine in Figure 2 and the output in each state would simply indicate which of three options should be selected by the combinational logic block in setting the output: (a) display the current value of `SW[3:0]`, decoded into decimal on HEX0, or "ErrOr" on HEX4 to HEX0 if `SW[3:0]` is not between 0 and 9; (b) display "OPEn" on HEX3 to HEX0; or (c) display "CLOSED" on HEX5 to HEX0. While option (a) has two possibilities (display a digit on HEX0 or "ErrOr" on HEX4 through HEX0) choosing between them should be done based only on the present value on `SW[3:0]`, which can be done in the combinational logic block.

Note that reset described in Section 1.1 is "synchronous". This means that when the reset signal is asserted (set to logic 1), the state machine is reset on the next rising edge of the clock. Again, remember that the actual output of KEY0 and KEY3 will be 0 when you press the key and 1 when it is not pressed.

The state machine should be positive-edge triggered. This means that the transition from one state to the next occurs on the rising edge of the clock. Given KEY0 is 0 when pressed and 1 when not pressed, you should think carefully about exactly *when* your circuit will see the rising edge.

The output of your circuit is connected to HEX5, HEX4, ..., HEX0. You should be able to figure out what should be driven on the signals to display a given number or letter given the information in Figure 3. You will need to determine the 7-bit value that will display a given number or letter. For example, `7'b0000010` will display a "6" because the one in bit position 1 turns *off* the LED corresponding to `HEX0[1]`. Similarly, `7'b0011001` will display a "4" because the ones in bit positions 0, 3 and 4 turn off the LEDs corresponding to `HEX0[0]`, `HEX0[3]`, and `HEX0[4]`. See Figure 3. When displaying "ErrOr", "OPEn" or "CLOSED" some letters may need to be approximated given the available seven-segment displays do not allow displaying curved lines. As long as what is shown on the seven segment displays looks somewhat like the desired letters that is fine.

Put your synthesizable Verilog in a file called `lab3_top.sv`. You can start with the template version on Piazza. The code provided on Piazza uses signal names that correspond with pin assignments in the pin assignments file, `DE1_SoC.qpf`, which is also provided.

2.2 Step 2: RTL-level simulation

Using the starter code `tb_lab3.sv` create a testbench that simulates inputs on the pushbutton and slider switch to verify that your design works when simulated with `lab3_top.sv`. Save your waveform format in a file `lab3_wave_rtl.do`.

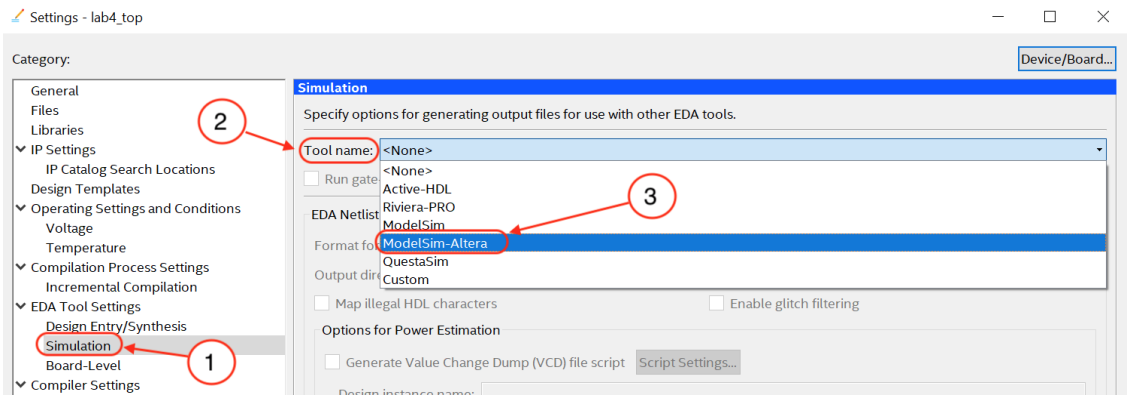

Figure 4: Configuring Quartus to generate a gate-level netlist Verilog Output File (.vo)

2.3 Step 3: Gate-level simulation

You must also create a testbench in the starter file `tb_lab3_gate.sv` that simulates inputs on the pushbutton and slider switch to verify that your design works after synthesis. Remember that you cannot use hierarchical signal names with post-synthesis RTL. (More correctly, the internal signal names of your design will all be changed during synthesis and so to use hierarchical signal names in your testbench after synthesis, you first need to know what a signal in your original design was changed into, assuming it wasn't entirely optimized away.) As a consequence, you can simply limit this test to checking the externally visible outputs.

Recall that ModelSim simulates Verilog by directly interpreting the language but that Quartus must build a circuit from the Verilog. In this step you get Quartus to synthesize your design and generate a gate-level netlist representation in Verilog. This verilog will look *very* different from the verilog you wrote in Step 1 because it is the result of synthesis and a (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 (FPGAs implement combinational logic by loading the truth table values into a small memory called a lookup table).

To start, create a Quartus project and put your synthesizable Verilog (e.g., "`lab3_top.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 Figure 4). Next, recompile your design in Quartus. Quartus will generate a file with your top level module name and a ".vo" extension (e.g., "`lab3_top.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_top.vo`"). Note that you will need to include the line "**`‘timescale 1 ps/ 1 ps`**" (type out the back tick ‘ rather than cut and pasting from this PDF) 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_top.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. As your testbench module for gate level simulation 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
```

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`.

2.4 Step 4: DE1-SoC

Demonstrate the design working on your DE1-SoC. For your pin assignments use the file `DE1_SoC.qsf` provided in the starter repo using the procedure outline in Section 2.4 in "Tutorial: Introduction to Verilog, the Tool Chain and the DE1-SoC Board" on Piazza ("To load the pin assignments, select Assignments -> Import Assignments ...").

3 Marking

This section describes the marking rubric for Lab 3. You can use either state machine coding style shown in Slide Set 7 for this lab. If working with a partner, each students mark may differ depending upon their answers to TA questions and/or their relative contributions.

3.1 Step 1: Explain your code.

Your mark for this part will be.

0/3 marks If you did not submit any code or the code has compilation errors.

1/3 marks If you submitted code that compiles, but cannot succinctly explain how it works when asked by your TA. You will also get this mark if your Verilog contains no comments or the comments are very brief or make no sense to your TA. You would get this mark if the code does not include a way to reset the state.

2/3 marks If you submitted code and can succinctly explain how it works (or is supposed to work) but it is not synthesizable (e.g., does not follow one of the patterns for how to use always blocks explained in Slide Set 6 and 7). You would get this mark if your code has warnings about inferred latches or combinational loops when synthesized in Quartus (NOTE: In later labs, the marks deducted for such errors will be much higher).

3/3 marks If you can succinctly explain your submitted code and it is synthesizable and looks to the TA like it could implement the requirements (e.g., if minor bugs were fixed).

3.2 Step 2: RTL-Level Simulation (ModelSim).

Your mark for this part will be:

0/3 marks If you don't have a testbench at all.

1/3 marks If you created a testbench and it compiles, but it does not work together with your FSM module, or you do not include a `lab3_wave_rtl.do` file.

2/3 marks If you created a testbench and it compiles and works with your lab3_top module, you include a `lab3_wave_rtl.do` file that includes both top level signals shown in Figure 1, and the state of your state machine, but your simulation output does not look correct to the TA. For example, the output does not change at the right time—e.g., it is delayed by one or more extra cycles. You will also get this mark if your testbench and/or FSM module includes no comments or very few comments (see below).

3/3 marks If your testbench extensively tests your synthesizable design, your `lab3_wave_rtl.do` includes the necessary signals mentioned above, the output looks correct in the ModelSim waveform to the TA, and your Verilog includes sufficient commenting. Specifically, that means, include at least one comment per test condition – e.g., change in inputs `SW[0]` or `KEY[1]` – in your test script and one comment per always block, module instantiation or assign statement in your sythesizable code. You do *not* need to include a comment for each change in the clock input `KEY[0]` in your test script unless you want to.

3.3 Step 3: Gate-Level Simulation (Quartus and ModelSim).

0/2 marks If you were unable to complete this portion.

1/2 marks If you could generate a ".vo" file, but cannot get it to simulate in ModelSim (note to save time you do not need to create the ".vo" file during your demo). You will also get this mark if your simulation does not look correct.

2/2 marks If you submitted a ".vo" file and can show you know how to run a simulation using it and the simulation works correctly.

3.4 Step 4: Synthesis (Quartus and DE1-SoC).

Your mark for this part will be:

0/2 marks If your your design does not synthesize in Quartus or your design synthesizes and you can download it to the board, but the HEX does not display anything, or the display does not change or the reset does not work properly on a DE1-SoC.

1/2 marks If your design synthesizes and displays something on the HEX0, which changes when the clock input is pressed but what it displays does not look like numbers to your TA; or your design displays the expected numbers on HEX0 and your reset works.

2/2 marks If your design works properly on a DE1-SoC as far as the TA can tell.

4 Lab Submission

As noted on Page 1, you will submit your code using github classroom using the invite link for your section of Lab 3 available at https://cpen211.ece.ubc.ca/cwl/github_info.php. You will first need to enter your github ID which means signing up to github.com if you do not already have an account. After you have submitted your changes (using git push) you should see the changes on github.com in the repo created using the invite link.

Check the README in the Lab 3 starter code for a list of files that need to be submitted. Briefly, ensure all your work including project files for ModelSim and Quartus are contained in the repository, along with your .sof file.

If you are working with a partner, your submission MUST include a file called "CONTRIBUTIONS.txt" that describes each student's contributions to each file that was added or modified. If either partner contributed less than one third to the solution (e.g., in lines of code), you must state this in your CONTRIBUTIONS file and verbally inform the TA. Your TA will deduct 3 marks if CONTRIBUTIONS is missing and may deduct up to this amount if it lacks in meaningful detail.

In addition, if you used any artificial intelligence tools, such as ChatGPT or github copilot you **MUST** document all prompts or other inputs in a file AI.txt. There should be enough detail in AI.txt to generate either the same (or very similar) code.

Submitted files may be stored on computer servers outside of Canada. Thus, you may omit personal information.

5 Lab Demonstration Procedure

Lookup your TA along with your marking hour and place in the TAs marking order here. Your TA will have a copy of your code when you arrive at the lab and will open it up on ModelSim/Quartus on a computer they are logged into, but you should bring your DE1-SoC as they may ask you to download your design to it. If you are working with a partner both of you must be present either in person or on Zoom (during your lab session) to receive a grade. The TAs will ask you questions about your code and ask you to demo the design on your DE1-SoC.

If you and your partner are not present when you come up in your TAs marking order and you have not told them you will be late beforehand, your name will be put to the end of the list. If this happens the TA will be under no obligation to mark you, but may do so at their own discretion and if time permits. Please note the TAs are not expected to stay past the end of their assigned lab sections and many may have classes they are attending, which start on the hour. We do not have enough TAs to allow for "make up" sessions.

6 Hints and Tips

You may find it helpful to use the Verilog include directive to define constants used in multiple files.

Instead of binary, you can have ModelSim display your state signals using symbolic names in the waveform viewer via 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". Then, 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 the command you can place it 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 Verilog to compile in ModelSim, but before running any simulations in ModelSim, it is worth try to 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.