CS/ECE 552
Spring 2022
Homework 1
Due 11:59 PM Central Time on Friday, February 11th, 2022

You should do this assignment on your own, although you are encouraged to talk with classmates in person or on Piazza about any issues you may have encountered. If you take code from another student, you will be penalized according to the UW Student Code of Conduct, as laid out in the syllabus. The standard late assignment policy applies: you may submit up to 2 days late with a 10% penalty for each late day. See *Hand in* directions for additional details.

Total Points: 10

The main objective of this problem set is to become familiar with the ModelSim program and the CS workstations. Before starting this homework, you should do the following:

1. Follow the instructions on ModelSim Setup Tutorial to get your environment setup.
2. Read the Command-line Verilog Simulation Tutorial. Additional references are on course website.
3. Read the Verilog Cheat sheet and Verilog rules pages. Everything you need to know about Verilog are in these documents.
4. Read the Verilog file naming conventions and Verilog file naming convention checking webpages and adhere to those conventions. We will be checking this in your submission to ensure you followed the rules.
5. Read the Verilog rules checking page on the course website and adhere to the conventions. This page also provides information on how you can check that your files conform to these rules.
6. Read the Handin Verification page on the course website. You will need to run this before submitting your answers.

You should simulate your solutions to verify the correct function of your designs for yourself. You also have to hand in a copy of the Verilog files, as discussed below.

This assignment will take a significant amount of time if you are not already familiar with ModelSim and hardware design languages. Get started as soon as possible. Previously, your TAs gave a tutorial on (combinational) Verilog and on Monday 2/7/22 they will give a tutorial on testbenches (although we have provided some testbenches for this assignment).

To deal with complexity, use a "divide and conquer" or hierarchical design approach. Divide the circuit into logical pieces, called blocks, which can be composed to form the larger circuit. For example, a 4-to-1 multiplexor or mux can be composed from 2-to-1 muxes. Hierarchical design reduces both the complexity faced by the designer and the complexity of the computer's representation of the schematic. While hierarchical design may seem unnecessary for something as simple as a 4-to-1 mux, remember that modern computers have billions of gates.

Problem 1 [5 points]

For all parts of this problem, we have provided templates that you can add your code to. See `/u/s/i/sinclair/public/html/courses/cs552/spring2022/handouts/verilog_code/hw1/hw1-templates.tgz`.

1. Design a 1-bit 2-to-1 multiplexer using only the provided NAND, NOR, and NOT gates (you do not have to use all three gates, you are just limited to using these gate types). Implement the circuit in Verilog using these modules. These modules can be found in the provided tarball *hw1-templates.tgz*. The input data lines of the multiplexer should be labeled *inA* and *inB*, the select line labeled *s*, and the output labeled *out*. **Your module should be named mux2_1 and your file should be named mux2_1.v.**
2. Use the 2-to-1 mux you designed in step 1 to hierarchically create a 4-to-1 mux. Label the inputs *inA*, *inB*, *inC*, and *inD*, and the output *out*. Make your select input a bus (i.e., a vector, not single wires); name it *s*(1:0) (If *s* is 00, *inA* is selected; if *s* is 01, *inB* is selected, etc.). **Your module should be named mux4_1 and your file should be named mux4_1.v.**
3. Hierarchically create a quad (4-bit) 4-to-1 mux using the symbolic version of your 4-to-1 mux. The inputs to the new mux should be four 4-bit buses (vectors) labeled *inA*(3:0), *inB*(3:0), *inC*(3:0), and *inD*(3:0). The select bus is a vector labeled *s*(1:0) and the output should be a bus (vector) labeled *out*(3:0). **Your module should be named mux4_1_4b and your file should be named mux4_1_4b.v.**
4. Use the testbench provided for testing.

Problem 2 [5 points]

For all parts of this problem, we have provided templates that you can add your code to. See `/u/s/i/sinclair/public/html/courses/cs552/spring2022/handouts/verilog_code/hw1/hw1-templates.tgz`. I also recommend consulting Appendix B.5-B.6 in your textbook if you need help getting started. You may also find B.3 (Combinational Logic) and B.4 (Using HDLs) useful.

1. Design a 1-bit full adder using only the provided NOT, NAND, NOR, and XOR gates (you do not have to use all four types of gates, you are just limited to using these gate types). Again use the provided modules. Label the inputs as *a*, *b,* and *c_in* (carry-in). Label the outputs as *s* and *c_out*. Note that this will look similar to the adder we designed in class, except we're using NOT, NAND, NOR, and XOR here. **Your module should be named fullAdder_1b and your file should be named fullAdder_1b.v.**
2. Optionally (but strongly recommended), verify the correctness of the 1-bit adder over all combinations of inputs by writing your own testbench. It is generally best to test at the smallest module level first. Given that this module is so small, it should be fairly easy to exhaustively test it. I also recommend that you create "self-checking" testbenches that compare the answer of the module you designed to the "golden" output that you get from using "+" or "-" in the testbench (again, it is ok to use these in the testbench, but not in the modules you are designing). See the testbench in Step 5 (below) for an example of this.
3. Using the 1-bit full adder you created above, design a carry lookahead adder (CLA) that adds two 4-bit binary numbers. Make the inputs and outputs 4-bit buses (vectors) labeled *a*(3:0), *b*(3:0), and *sum*(3:0), respectively. Label the carry-in *c_in* and the carry-out *c_out*. **Your module should be named cla_4b and your filename should be cla_4b.v**
4. Using the 4-bit CLA you created above, design a CLA that adds two 16-bit binary numbers. Make the inputs and outputs 16-bit buses (vectors) labeled *a*(15:0), *b*(15:0), and *sum*(15:0), respectively. Label the carry-out from the adder *c_out* and the carry-in *c_in*. **Your module should be named cla_16b and your file should be named cla_16b.v**

5. Use the testbench provided for testing.

What to Hand In

To submit this assignment, zip or tar your Verilog files together and submit them as a **single file named <netID>-hw1.tgz or <netID>-hw1.zip** on Canvas. Inside this tarball/zip, you should have a top-level folder (e.g., hw1), with two sub-folders: hw1_1 and hw1_2. All files for problem 1 should be in hw1_1 and all files for problem 2 should be in hw1_2 (including provided files). For example, my Net ID is msinclair, so my submission would be called msinclair-hw1.tgz (or msinclair-hw1.zip). If you don't have experience with tar, I recommend consulting tutorials such as this one. In addition, before submitting you should **run the Verilog check on all the files** (just the new modules you are writing, you don't need to run it on your testbenches).

Graphically your handin directory should look like:

- <netID>_hw1 (parent folder, zip/tar this)
  - hw1 (Top level folder)
    - hw1_1 (folder containing everything for Problem 1)
    - hw1_2 (folder containing everything for Problem 2)

Verifying Your Handin

We have also created a script to check that your submission correctly follows the format, located at:

```
/u/s/i/sinclair/public/html/courses/cs552/spring2022/handouts/scripts/hw1/verify_submission_format.py
```

You should run this script before submitting in order to ensure that you don't lose points for incorrectly formatting your submission! Additional details about the handin script, including how to run and examples, are available here.

Note: if you created additional Verilog files as sub-modules, this script will not check for them – it only checks for the provided files and the vcheck files for the ones you are expected to modify. However, you still need to include vcheck files for the modules/files you add.