ECE275: Sequential Logic Circuits
Lab 7

Pascal Francis-Mezger

November 1, 2021

Contents

0 Lab Overview

This lab will cover a few topics. The first two are important for code quality. These cover how to properly connect ports on a module instantiation, and how to use modules in other Verilog files. Both of these topics become more important as code complexity and size scales up, so it is a good idea for you to start understanding and utilizing it now.

1 Part 1: Connecting ports on Module Instantiation by Name

In C, reusable code is called a function, and the values passed are called variables. In Verilog, a similar idea is the module, and the values passed are ports. So far we have been passing the port connections by just putting them in the same order they are declared in when creating the module. For example:

```
module toplevel(
    input [7:0] SW,
    output [7:0] LEDG
);
    add_2 bo (SW[7:0], LEDG[7:0]);
endmodule

module add_2(
    input [7:0] binary_input,
    output [7:0] binary_output
);
    assign binary_output[7:0] = binary_input[7:0] + 2'b10;
endmodule
```

The adjustment we will make is attaching pins by name, instead of location in the module instantiation. The previous code would be adjusted from

```
add_2 bo (SW[7:0], LEDG[7:0]);
```

to the following:

```
add_2 bo (.binary_output(LEDG[7:0]), .binary_input(SW[7:0]));
```

You should be able to see from this that the port names are added, with a period before them, and then the value to connect to that port is declared inside parenthesis. You can also see that the order changed. This is not necessary, but is just intended to show that when you are passing the ports by name, the order does not matter.

This may not seem very important, but generally it is considered much better practice to pass by name instead of by location. This is because you are very likely in Verilog to write modules that have a large number of ports (sometimes hundreds). This would make it very difficult to keep track of values if passed by location, especially if you want to modify a module.

To complete this section, modify one of your previous labs to pass the variables by name to the modules instead of by location. Have your TA check off your updated code running on the board.

2 Part 2: Instantiate Modules from External Verilog Files

So far this semester, we have been including all of our Verilog modules in the same Verilog file. While this has simplified troubleshooting, it is generally considered a bad practice. This is mostly due automated tools disliking multiple modules in a single file. The better practice is to only have a single module in a Verilog file, and then naming the file modulename.v (so the module BCD_Display would be BCD_Display.v). This obviously creates the issue of how to then utilize these modules in your other Verilog code.

The method for including modules from an external verilog file is to add an include statement at the top of your Verilog code. Unfortunately Verilog does not support including an entire directory, so you have to specify each Verilog file to include.

Below is an example of lab 6 part 1, with the correct method of passing variables, and utilizing including files:

```
1  `include "../../Modules/BCD_Display.v"
2  `include "../../Modules/convert_8bit_to_BCD.v"
3  module lab7part2top(
4          input [7:0] SW,
5          output [6:0] HEX0_D,
6          output [6:0] HEX1_D,
7          output [6:0] HEX2_D
8  );
9          wire [11:0] converted_BCD;
10
11         convert_8bit_to_BCD c2bcd1 (.binary_value(SW[7:0]),
12         .BCDValues(converted_BCD));
13
14         BCD_Display(.BCDValue(converted_BCD[3:0]), .LED_Segment(HEX0_D));
15         BCD_Display(.BCDValue(converted_BCD[7:4]), .LED_Segment(HEX1_D));
16         BCD_Display(.BCDValue(converted_BCD[11:8]), .LED_Segment(HEX2_D));
17 endmodule
```

As you can see in the code, the first lines are include statements that specify Verilog modules to include. You will need to use the path relative to your project file, not the absolute path. Quartus will automatically utilize any Verilog files included in the project directory for the current project, but you should not utilize that feature for this lab. This is because it would be unrealistic in the real world for you to always copy in the Verilog files to every project, because then if a module gets a slight change you would have to do that same change in every project. If you include from a single directory of Verilog modules, you can edit the module 1 time in one place.

For a relative path, you only need to know a couple of things. A single dot denotes the current directory. Two dots denotes the parent directory. The file structure I used for the labs is shown in Figure 1. My include statement was 'include "../../Modules/BCD_Display.v" This is because I needed to go up from Lab7p2 to Lab7, and then up to Labs, and then I can go inside the Modules folder. Also, the symbol at the beginning of the include line is NOT an apostrophe. It is commonly called a backtick, and it is located on the same key as the tilde on your keyboard. Likely to the left of your 1 key.

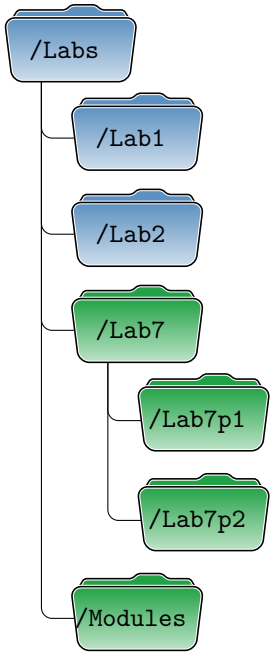


Figure 1: Example Folder Structure

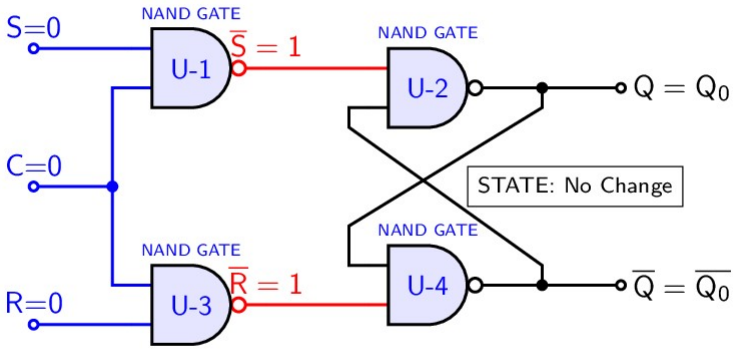
For this section of the lab, modify Lab 6 Part 1 so that the all of the Verilog modules are in separate files that are just included in your top level, similar to what is shown in the code above. It is recommended to start with just a

single module, such as your BCD_Display module, and make sure that works before moving the rest. Also, keep in mind that you will have to add an include to the verilog file for your BCD converter to your shift add 3 module. As it will be contained in the same directory as the BCD converter you can just use ./modulename.v for the include line.

3 Part 3: Simple Digital Logic Clocking and Compiler Directives

For this section, you will create a simple clocked component from combinational logic. In general you would instead do this with always blocks and procedural coding, but if there is a situation where combinational relates to your design better you can still utilize combinational Verilog.

The other goal of this section is for you to utilize your first compiler directive. These directives can be used to force the synthesizer to create logic in a method you prefer. This is especially important with simulations. For example, the synthesis will generally optimize out logic it feels is not needed to preserve hardware. In the case of the code below, the compiler directive is the synthesis keep line. This line tells the synthesis stage to not optimize out the wires. This makes the resultant logic less streamlined, but allows for more useful simulation. This is because the simulator needs an "attachment" point to read the current value of a signal. If the logic is optimized, there may be no point to attach to, to read the signal you are interested in.



```
module RS_Latch(
        input Clk,
        input R,
        input S,
        output Q
);
        wire R_g, S_g, Qa, Qb /* synthesis keep */;
        assign R_g = R & Clk;
        assign S_g = S & Clk;
        assign Qa = ~(R_g | Qb);
        assign Qb = ~(S_g | Qa);
        assign Q = Qa;
endmodule
```

To complete this section, create a new project and utilize this module from your top level. Use an LED as Q, and switches for the Clock, R, and S. The module should represent a set/reset, which will only trigger on a clock pulse (false to true conversion of switch tied to clock). Test it out to make sure it operates this way.

Lastly, examine the compiler directive /* synthesis keep */. This directive keeps the compiler from optimizing out your wire connections. Run compilation on the code as written, and take a screenshot of the RTL viewer. Next, remove the /* synthesis keep */ line and re-run compilation. Compare the new RTL view to your screenshot. What are the differences? Show the TA a screenshot of the RTL viewer with and without the synthesis keep line, and explain the differences to get checked off.