Lab 1 8-Bit Barrel Shifter
CSE 521 FPGA Design
Aaron Chamberlain ID: 003392908

I Goals:
The following lab was designed to aid in review of basic Verilog concepts, such as module design, testing, and combinational logic circuits. In order to build this simple 8-Bit barrel shifter, several components must be constructed and combined to form the working circuit and gives practice in many areas of HDLs.

II Design Process:
A. The lab asserts that the same problem to producing an 8-Bit barrel shifter can be done in two different ways. The first is to build a rotate-right circuit, a rotate-left circuit, and a 2-1 MUX that will determine which rotation circuit is used. This design is extremely simple and can then be improved upon by using a different algorithm.

In the updated algorithm, the rotate-left circuit is preserved, but the rotate-right circuit is replaced by performing a bit-reversing operation, running it through the same rotate-left circuit, and then reversing it one last time (see the circuit diagram in **Figure 1**). This algorithm can be proved to be true via a truth table in **Figure 2**.

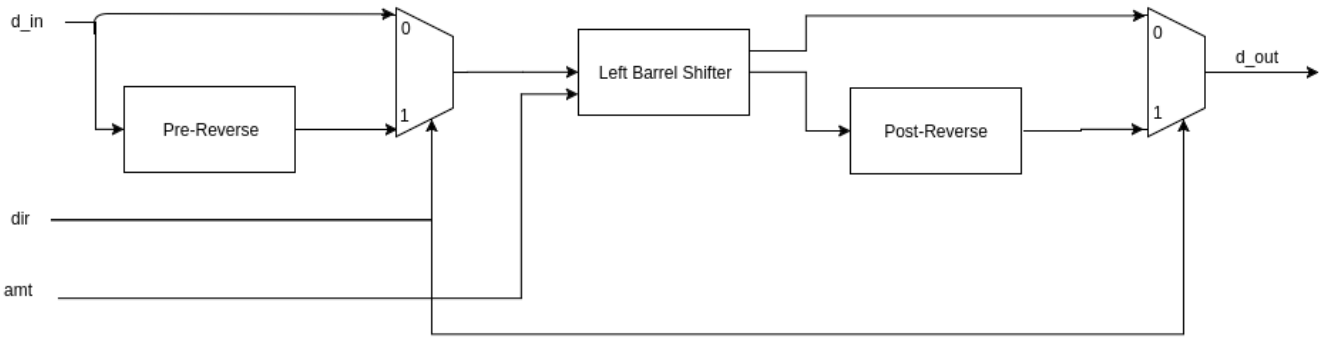

Figure 1. The circuit diagram for the 2nd Version of the bi-directional shifter.

d_in	shift_r >> 2	shift_l >> 2	pre_reverse	shift_l << 2	post_reverse/right
00000001	01000000	00000100	10000000	00000010	01000000
10000000	00100000	00000010	00000001	00000100	00100000
10101010	10101010	10101010	01010101	01010101	10101010
10010011	11100100	01001110	11001001	00100111	11100100

Figure 2. Regular shift_left/right operations on left. Reversing right shift on right.

III Program:
The source files have been submitted in the zip file alongside this report. Included are:
Lab 1:

- barrel_shifter.v – The left barrel shifter in this first case
- barrel_shift_tb.v – The test bench for barrel_shifter.v, performs two simple tests.

- bi_directional_shift.v – The top level module that combines both directional shifters to pick the results given by the shift direction.
- bi_directional_shift_tb.v – The test bench for bi_directional_shift
- right_barrel_shifter.v – A right barrel shifter
- right_barrel_shifter_tb.v – The test bench for right_barrel_shifter.
- Shifter.ucf – The User Constraint File that maps the layout of the development board to the input wires of bi_directional_shift. In my design, the left most button determines the shift direction, and the rightmost 3 buttons determine the shift amount. The input is given by the toggle switches, with each switch representing a bit in the input. Finally, the 8 LEDs are used to show the output of the circuit.

Lab 1_reverse:

This version of the circuit shares all the same code with the addition of:

- mux2_1.v – A simple MUX implementation that chooses between two inputs.
- mux2_1_tb.v – The test bench for mux2_1.v
- bit_reverse.v – A simple module that uses a for loop to reverse the order of bits.
- bit_reverse_tb.v – The test bench for bit_reverse.v

IV Results:

Test benches were written for every module included in the source, so it came as no surprise that the final shifting operation worked as expected. The test bench for the simple individual left and right shifters is given in **Figure 3.**

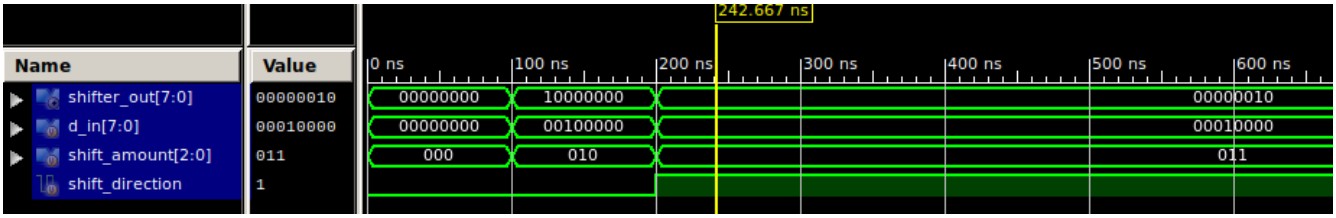

Figure 3. All values are cleared until 100ns. We test shifting left by 2, it works. At 200ns, we test shifting right by 2 with a new d_in, it works.

Next up, I switched to writing the modules required for the new shifting via reverse algorithm. Below are 3 test bench outputs displaying the manner in which they were tested. The mux module was written first to determine if two inputs could be selected in **Figure 4.**

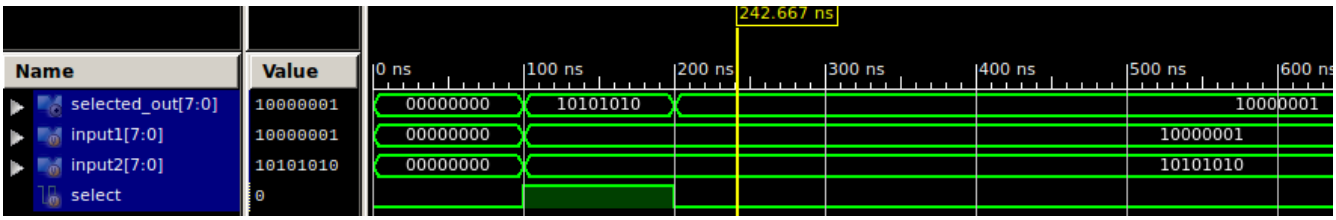

Figure 4. Mux test bench. At 100ns we select input2, it works. At 200ns we select input1 and it works.

The next module to be written and tested was the bit_reverse module. Using a simple for loop, I reversed the order of all bits within the input. The results were verified with only one input for brevity. That result can be seen in **Figure 5.**

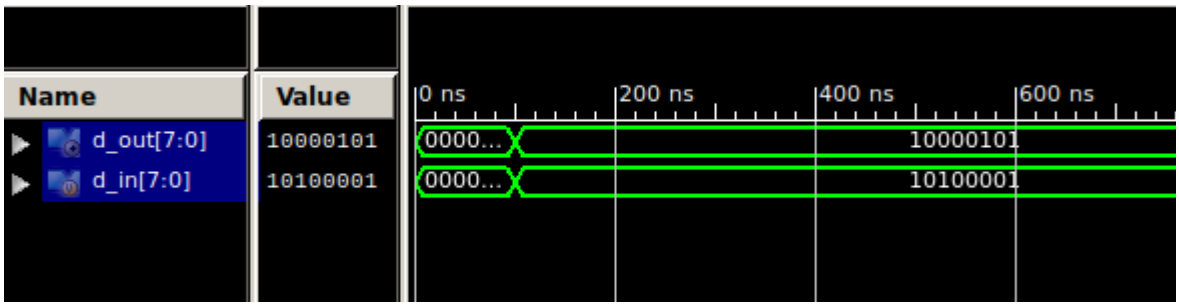

Figure 5. Bit_reverse. All inputs are cleared. At 100ns we provide the input and see it's result immediately reversed.

Now that all the required modules were written, I wired them all up in the top module by directly wiring their inputs together in the instantiation of the modules. In this way, it is as if the individual components were wired together. A few sample inputs were verified in a test bench before further tests on hardware, see **Figure 6.**

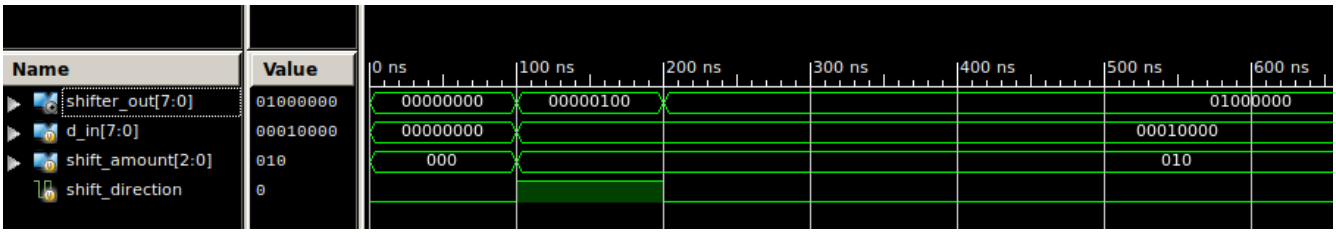

Figure 6. At 100ns the input is shifted to the right by 2 units. At 200ns the input is shifted to the left by 2 units. Both tests complete successfully.

V Problem:

Given the simple nature of all components involved in this combinational circuit, none of the implementation was difficult. The hardest part of this lab was remembering items from discrete mathematics on how to determine the logic of a combinational logic system. Whereas the lab instructions merely mentioned some of the components that would be required, I spent the majority of the lab determining the order in which they would best be arranged. After this logic circuit was drawn out and proved with truth table that is shown above, then I felt comfortable implementing the simple logic components in actual Verilog code.

VI What You Learned:

This lab was extremely helpful in allowing for the basic review of many topics in combinational logic and Verilog program structure. Even though this is not the first class taken that used Verilog, it is the first class in which it was formally taught, which resulted in much less guess work and better engineering.

The first thing that it allowed for me to learn was the testing of logic via truth tables. This is a simple task that I simply hadn't done for a while.

After this, one of the larger of the bigger things that I learned was how to make better use of the ISE simulator. I was unaware until this point that logic could be changed in the code and the simulation re-evaluated. Also of great help in this lab was the introduction to looking at nested values in the

evaluation. For example, if the top module of a Verilog module is being tested with components instantiated below it, then all values of the instantiated modules can also be seen. In this way, one can see if the error is being propagated from a lower module or not. I also learned that while the values of the modules being tested are shown, the user has the option of dragging other values in the graph to see them along side the other logic evaluation.

Finally, I learned about a few additional Verilog language features. In order to achieve the bit reversing, I utilized a for loop that I did not know was previously synthesizable. This language feature as well as things like the turnary operator, the forever block, and localparam will most likely make their way into my modules from here out, as long as they are needed.