ECE275: Sequential Logic Circuits
Lab 5

Pascal Francis-Mezger

October 17, 2021

Contents

0 Lab Overview

This lab's goal is to acquaint you with higher level Verilog funtionality, and how it relates to the created hardware on the FPGA. It is vital for you to understand the low level hardware design effects that result from your Verilog code.

The second part of this lab will show you the beginnings of utilizing sequential logic on the FPGA, which are normally instantiated from always blocks. It is possible to generate combinational logic from always blocks, but the programming is slightly different than we have utilized so far.

1 Part 1: Logic Synthesizer Comparison

For the first section, you will be testing your answers from the prelab for the Number Representation, Arithmetic, Concatenation and Replication, and Conditional Operator sections. To do this, create your answer for each as a separate module in a single Verilog file.

You will need to check off two things for this section. First, you will need to show each of the pieces working. Make sure you test them before asking a TA to check you off. Also, you will need to show a screenshot of the gates created for each answer. You can find the gate level hardware created by your Verilog code in the RTL viewer after you run compilation. For example, Figure 1 shows the gate level hardware created from lab 4. In the RTL viewer I had to right click on the box representing the 4 bit adder and choose "flatten netlist", and the select one of the full adders and choose display content to get the view shown. I did not display all of the full adders because they would have the same gates as the first and would just clutter the view.

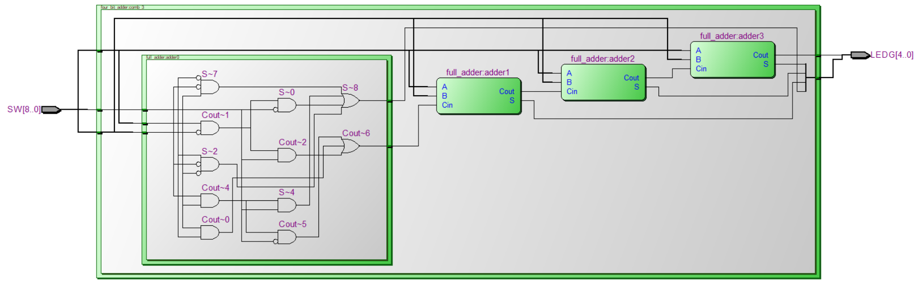

Figure 1: Gate Level Hardware Created from Quartus Synthesis of Boolean Combinational Logic Instantiation of a 4-Bit Ripple Adder

2 Part 2: Using an Always Loop

In this section you will utilize an always loop to continually count up based on an input value triggering. It will trigger based on the positive edge, so every time a transition from false to true is triggered, the always loop will run the code contained inside once. In this first case we will utilize a switch for the trigger. When the switch turns on, it will create a positive edge (false to true conversion). Another positive edge will not be triggered until the switch moves to off and then back on. Each time the switch is triggered on, the always loop will run once.

Use the code shown below for this section. Make sure to analyze the comments so that you understand what is going in the code, as you will have to slightly modify it for the next section.

```
module lab5part2top(
    input [9:0] SW,
    output [9:0] LEDG
);
//Register can only be on left hand side of equation inside an always block
reg [3:0] summed_counter; //stores current count
//Runs once when posedge SW[0] is true. This is when SW[0] goes from off to on
always @ (posedge SW[0]) begin
    //1'b1 is the representation of binary 1 in Verilog
    //This adds one to the current sum each posedge of SW[0]
    summed_counter = summed_counter + 1'b1;
end
//Display the current sum on the first four green LEDs
assign LEDG[3:0] = summed_counter[3:0];
endmodule
```

After programming the FPGA, repeatedly toggle SW[0], and watch what happens with the green LEDs. Does the value sum correctly, where each time the switch goes false to true, the binary value represented by the LEDs increases by 1?

What is the maximum value that can be represented by this code? What happens when this maximum value is exceeded?

What could you do to increase the maximum value you can represent?

3 Part 3: Using a Clock on the FPGA

In this section you will modify the code so that it counts up on a positive edge of one of the FPGA clocks instead of the switch. In the future we will look at how to reduce the frequency of the clock using a phase locked loop so that the clock is more useful, but for now we will just use the clock at the full frequency. The DE0 FPGA we are utilizing has one internal clock going to two pins, that runs at 50MHz. This means the clock will produce 50 million positive edge triggers per second. You can find information about the pin assignment of the clock on page 27 of the DE0 manual linked on the class website. You can also find its pin assignment in the QSF file. Add the clock as an input to your module (input somename, and then use the QSF to assign somename to the 50 MHz clock).

Due to the clock running so fast, we need to utilize more bits to make the counting readable. If we tried to display the count using a 9 bit register, it would roll over faster than the eye could see. This is because 9 bits can only represent $2^9 = 512$ states. $\frac{512\ counts}{50000000\frac{counts}{second}} = 10.24\mu$seconds. We can instead use a larger register to represent many more states. If we change the register summed counter to 31:0 instead of 3:0, we can represent 32 bits. This would give $2^{32} = 4.29$ billion, and $\frac{4.29 \text{ billion counts}}{50 \text{ million} \frac{\text{counts}}{\text{second}}} = 85.9$ seconds.

After adding the clock as an input, and expanding the register, you will need to make the always trigger on a positive edge of the clock. Just replace SW[0] in the alway with the name of your clock.

The last thing you will have to do is change the assign statement for the green LEDs so they display the 9 **most significant bits** of the summed_counter register. These are the slowest changing values of the register.

After you have made these changes, download your code to the FPGA.

Do the LEDs count up as you would expect? What happens after maximum value of the summed_counter is reached (all 9 green LEDs would be on)?

4 Conclusions

Save this code, as next week you will be utilizing the counter to display a timer on the seven segment LEDs. This will require you to convert the count value to BCD, and then use a multiplexer to select which seven segment should have what digit.

One more important thing to understand with this lab is the value of the speed of the FPGA in this case. It would be difficult to create a timer/counter with this fine granularity (50 million counts per second, each value individually representable) on a microprocessor, where an FPGA can do it relatively easily.