CS M152A Lab 2 Report

Jonathan Chau

UID: 705-166-732

TA: Boyuan He

February 7, 2021

1. Introduction and Requirements

For this lab, I want to demonstrate my knowledge on Xilinx by building a sequential circuit using Verilog. This sequential circuit implements and tests various clock waveforms from the basic division by 2 clock, to the odd-division clock, to the glitchy counter. To implement those different clocks, we have to utilize various techniques, from extracting bits to creating duty cycles. These techniques allow us to create clocks that are useful for synchronous data transmission, communication between devices, and checking for expected behavior. To create such clocks, I broke my overall module down to four individual modules that each take the input clock (clk_in) and the reset signal (rst) as input and the desired clock waveform as the output. The project description outlines how I built my four modules that make up the overall module.

clock_gen.v Description	
Divide by 2^n Clock	The submodule exploring clock division by power of 2
Even Division Clock	The submodule exploring even clock division
Odd Division Clock	The submodule exploring odd clock division
Glitchy counter	The submodule exploring pulse/strobe/flag

2. Design Description

My overall module takes in the input clock (clk_in) and the reset signal (rst) as input, and outputs are the clock waveforms produced by clk_div_2, clk_div_4, clk_div_8, clk_div_16, clk_div_28, clk_div_5, and toggle_counter. To make the design process easier, I broke the top module down into four sub-modules, each tested separately using their corresponding testbenches. After building each sub-module, I combine them to make the final module and test it using a testbench file.

2.1 Division by Two Clock

The first sub-module takes in clk_in and rst as input and outputs clk_div_2, clk_div_4, clk_div_8, and clk_div_16. To get started with this sub-module, I followed the project description and built a simple 4-bit counter. Building this counter is not as easy as it seems, as I will have to implement an input clock that flips after a period of time has passed. The project manual said that I needed to implement clock that has a frequency of 100 megahertz. Knowing from previous lectures, one clock cycle lasts 10 nanoseconds. Each cycle is defined as the time from one positive edge to the next positive edge. That means I have to flip the clock twice per cycle. Dividing 10 ns by two gives me the length where I need to flip the clock: 5 ns.

Implementing the reset signal is easier, since the output clocks will rest at 0 when rst is 1. The rst signal triggers and takes effect when the input clock reaches a positive edge. This was implemented using an always posedge block and an if statement.

Now that the input clock is implemented, I can see how the 4-bit counter is incremented. I noticed that each bit in the 4-bit counter is a power of two. The next step is to map each bit of the 4-bit counter to the clock outputs. The least significant bit maps to clk_div_2. The second least bit maps to clk_div_4. The third bit maps to clk_div_8, and the fourth maps to clk_div_16. Testing the sub-module again, I quickly realized that clk_div_2 flips twice as slow as clk_in. Similarly, clk_div_4 flips twice as slow as clk_div_2. The same pattern goes for clk_div_8 and clk_div_16. With this observation, I can implement clocks that divide by powers of two.

2.2 Even Division Clock

The second sub-module takes in clk_in and rst as input and outputs clk_div_28. Using the same design of clk_in, rst, and counter from the first sub-module, I declared a second variable, b, that flips when the counter reaches a certain number. For example, for a 32-division clock, b will flip once the counter reads 4’b1111, or 15. I can construct a closed-form formula on what number n will trigger the flip on b.

$$n = 0.5D - 1$$

D represents the even number in which the output clock is slower by. For instance, if D = 32, then we are dealing with a 32-division clock. We can verify that this formula works by solving for n, which is 15. This matches the number of when we need flip b. I said even numbers, not any number, because odd numbers need both positive and negative edges to function correctly. This will help build the submodule that will verify task 2, where the divide-by-32 clock is working.

One other design decision I implemented is that whenever b is flipped, the counter resets to zero. This is because we want the output clock to flip after an even number of input clock flips consistently. Otherwise, the output clock will always flip once a power of two number of clock flips has occurred.

For a divide-by-28, clock, we need to trigger the flip on b and zero the counter when the counter reaches 13, or 4'b1101. The variable b will map to the output clk_div_28, to complete the even division clock sub-module. Now I can design division clocks with any even number in mind, not just powers of two.

2.3 Odd Division Clock

The third sub-module takes in clk_in and rst as input and outputs clk_div_5. This odd division clock is constructed using two duty cycles and taking the logical OR of the two cycles. To get the general idea, I built a 50% duty cycle divide-by-3 clock with the logical OR of two 33% duty cycle clocks, one on the positive edge, and another on the negative edge. Each duty cycle was built using two variables: a and c. The first variable 'a' flips after each positive edge, and 'c' flips after each negative edge. To adjust the percentage to 33%, I had to reference an equivalent fraction, which is 1/3. A counter was implemented so that two trigger points are reached: when the counter equals the numerator value, and when the counter reads the denominator value minus 1. The final cycle accounts for the part when the variables and the counter read zero, before incrementing each variable to 1 on the next cycle. That is how the 33% duty cycle is built. The 50% duty cycle divide-by-5 clock follows the same principle, except instead of two 33% duty cycles, I used 2 40% duty cycles. This will build a clock that flips after five input clock flips. With this, I can build an odd division clock of any odd number.

2.4 Pulses and Strobes

This category contains two mutually exclusive tasks, a divide-by-200 clock using a 1% duty cycle divide-by-100 clock, and a toggle counter that adds two on the first three cycles and subtracts five on the fourth.

I built my 1% duty cycle divide-by-100 clock using the same technique used for my 33% duty cycle. The only difference is that the two triggers occur when the counter reads 1 and 99. Then I implemented an always posedge block on the 1% duty cycle to flip the divide-by-200 counter clock.

The final submodule module takes in clk_in and rst as input and outputs toggle_counter. Since each pattern lasts four clock cycles, I used a 2-bit counter that counts up to three and resets to 0 on the fourth increment. The counter is implemented using an always posedge block, where the counter is incremented each time the input clock reaches a positive edge. The first three increments would add the toggle_counter by two. The final increment subtracts the toggle_counter by five. This completes the toggle_counter submodule.

3. Simulation Documentation

I used the Xilinx ISE environment to verify and test each of the nine tasks. These tasks, except for tasks 4, 5, and 6, were tested using their own individual testbenches. It was easier for me to design a combined testbench for verification tasks 4, 5, and 6. Each test, I held the rst signal to 1 for 100 nanoseconds. For design task 8, the rst signal was held for 1 microsecond. After that, I let each of the clock outputs loose.

3.1 Verification and Design Tasks

3.1.1 Task 1

For the first task, I tested the division-by-2 clock submodule by flipping the rst signal at 100 ns and 200 ns, and flipping the input clock every 5 ns. We should observe that for every 2 input clock cycles, we have one clk_div_2 cycle. We should also observe that for every 4, 8, and 16 input clock cycles, we observe one clk_div_4 cycle, one clk_div_8, cycle, and one clk_div_16 cycle respectively. This observation shows that I can generate a clock that has a period divisible by a power of two.

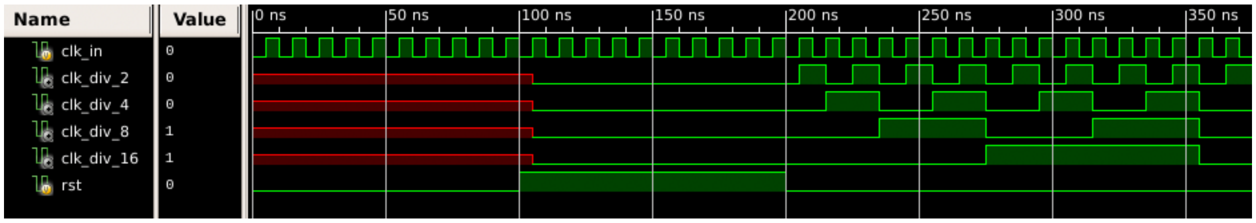

Figure 1. Simulation Waveform of Each Digit of the 4-Bit Counter

3.1.2 Task 2

For the second task, I want to see that the clk_div_32 clock's cycle lasts 32 times the length of an input clock cycle. Like task 1, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. This produces the waveform shown below. We want to see that each flip of clk_div_32 would be 160 nanoseconds apart. As we can see here, a positive edge exists at 355 ns, and a negative edge is displayed at 515 ns. 515 – 355 is 160. The next positive edge happens at 675 ns, another 160 ns after the negative edge. This observation verifies that we have a division-by-32 clock.

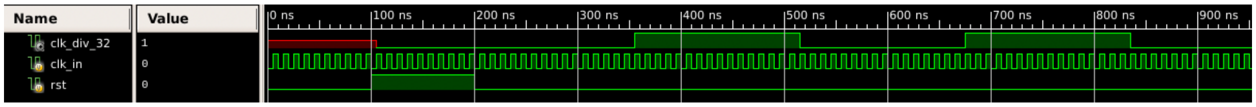

Figure 2. Simulation Waveform of the Divide-by-32 Clock

3.1.3 Task 3

For the third task, I want to see that the clk_div_28 clock's cycle lasts 28 times the length of an input clock cycle. Like task 1, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. This produces the waveform shown below. We want to see that each flip of clk_div_28 would be 140 nanoseconds apart. As we can see here, a positive edge exists at 335 ns, and a negative edge is displayed at 475 ns. 475 - 335 is 140. The next positive edge happens at 615 ns, another 140 ns after the negative edge. This observation verifies that we have a division-by-28 clock.

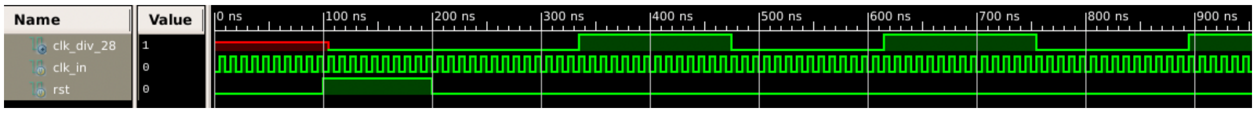

Figure 3. Simulation Waveform of the Divide-by-28 Clock

3.1.4 Task 4

For the fourth task, I want to observe a 33% duty cycle. That means each cycle, the clock is high one-third of the time. Like the previous tasks, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. The simulation waveform is produced below. The easiest way to verify this is to see that in a given three cycle clock input, the duty cycle clock is high for one clock input cycle and low for the other two. We observe that for each cycle of duty_cycle_33, it is high for one input clock cycle and low for the other two, which verifies that our 33% duty cycle is working as expected.

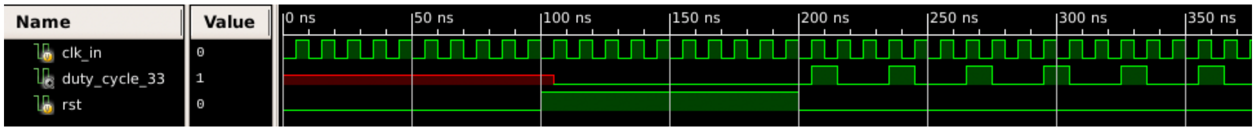

Figure 4. Simulation Waveform of a 33% Duty Cycle Clock

3.1.5 Task 5

For the fifth task, I want to observe a 33% duty cycle that activates on a negative edge. Like the previous tasks, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. The simulation waveform is produced below. We want to see that duty_cycle_33_neg outputs the same waveform as duty_cycle_33, but shifted one input clock flip apart. As we can see below, the displays same waveform as duty_cycle_33 shifted one flip to the right. We also see that duty_cycle_33_neg rises and falls on the negative edges of our input clock. This verifies that our duty_cycle_33_neg clock is working as expected.

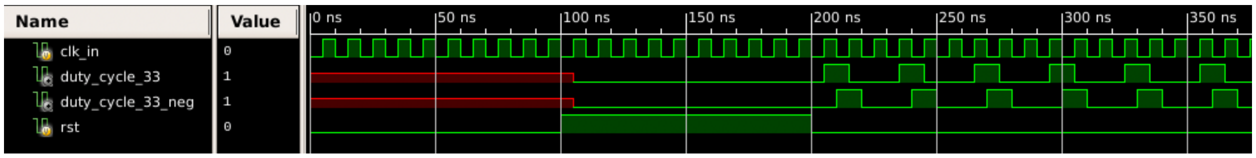

Figure 5. Simulation Waveform of a 33% Duty Cycle Clock that triggers on the Falling Edge

3.1.6 Task 6

For the sixth task, we want to see a the logical OR of the two 33% duty clocks. Like the previous tasks, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. The simulation waveform is produced below. Here, we can see that our new duty cycle rises when the duty_cycle_33 clock rises and falls when the duty_cycle_33_neg clock falls. That should verify that our new duty cycle works as intended. We should also observe that our new duty cycle is a 50% duty cycle divide-by-3 clock.

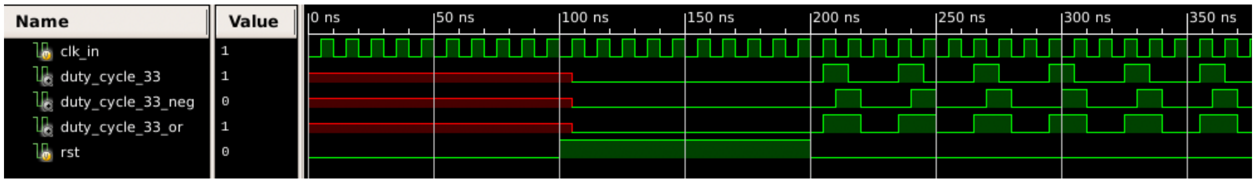

Figure 6. Simulation Waveform of the Logical Or of the two 33% Duty Cycle Clocks

3.1.7 Task 7

For the seventh task, we want to see a 50% duty cycle divide-by-5 clock. Like the previous tasks, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. The simulation waveform is produced below. We see in this simulation waveform that one clk_div_5 cycle lasts five input clock cycles. We can also see that each flip of clk_div_5 happens every 25 ns. The consistency of the timing of each flip shows that we have a 50% duty cycle. With these two in hand, we show that our 50% duty cycle divide-by-5 clock is displayed as intended.

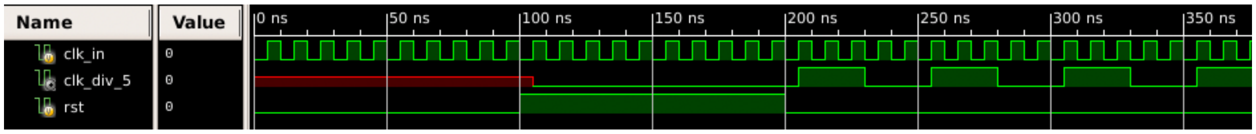

Figure 7. Simulation Waveform of the 50% Duty Cycle Divide-by-5 Clock

3.1.8 Task 8

For the eighth task, I want to observe that the divide-by-200 clock is 500 kilohertz. That means one divide-by-200 clock lasts two microseconds. Here, the rst signal is high at 1 µs and low at 2 µs, and the input clock is flipped every 5 ns. The simulation waveform is produced below. We see that our divide-by-200 clock has a positive edge at 2 µs and another one at 4µs. This means that our division by 200 clock cycle lasts 2 microseconds and thus has a frequency of 500 kilohertz. This verifies that our division by 200 clock works as intended.

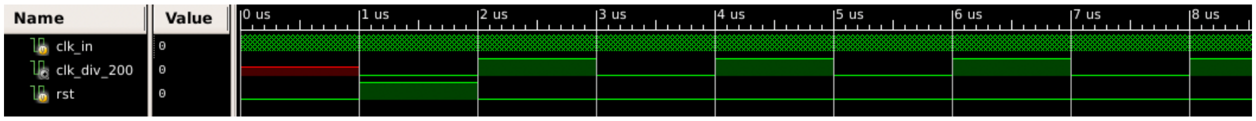

Figure 8. Simulation Waveform of a Divide-by-200 Clock

3.1.9 Task 9

For the final task, I want to observe that our toggle counter increments by two three times and subtracts by five after that. Like the first seven tasks, the rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 ns. The simulation waveform is produced below. We see at the first three input clock cycles, the toggle_counter increments by two each time the input clock rises. In the next cycle, 5 is subtracted from the toggle_counter. After that, it increments by two for the next three cycles, before being subtracted by five again. On further iterations, we see the same cycle repeated again, which verifies that our toggle counter works as expected.

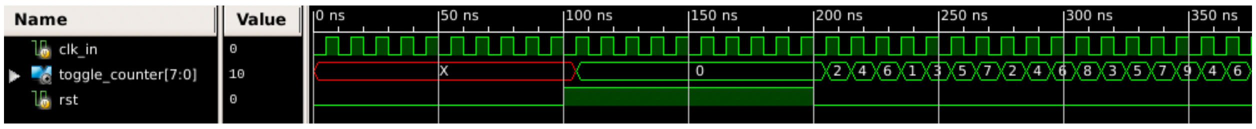

Figure 9. Simulation Waveform of the Glitchy Counter

3.2 Top Module Test

For the clock generator module test, I want to see that tasks 1, 3, 7, and 9 display as seen before. The rst signal is high at 100 ns and low at 200 ns, and the input clock is flipped every 5 nanoseconds. The simulation waveform is produced below.

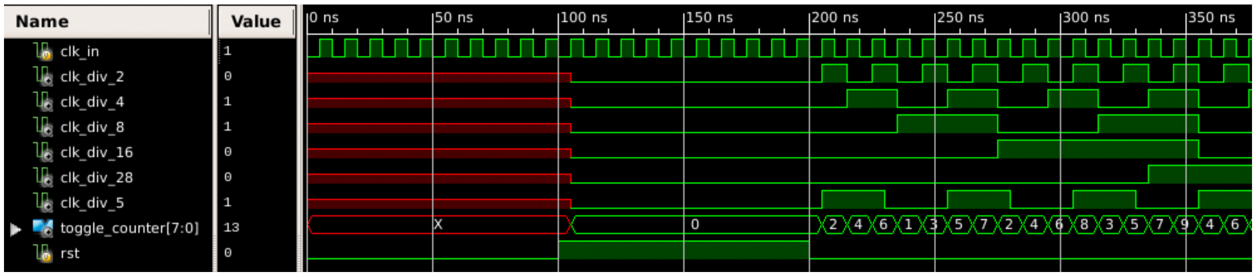

Figure 10. Simulation Waveform of the Clock Generator Module

4. Conclusion

In this lab, I learned how to construct both even and odd division clocks of different frequencies and duty cycles. I constructed an even division clock by extracting bits from the 4-bit counter, and an odd-division clock using duty cycles and triggering when the input clock reaches a positive edge or a negative edge. To build the clock strobe module, I used techniques learned from the even-division clock.

Some difficulties I faced during this lab were the unclear description the project specifications gave and the inability to get a clear sense on what to do on those unclear parts. These parts include implementing the 33% duty cycle and whether to use verification task 8 to help build design task 9. These confusions were cleared up once I approached the TA, as his answers helped guide me to build each submodule that works properly.