Megan Pham
505313300
CS M152A Lab 2
Dipti Sahu

Lab 2 Report

Introduction

Using a system clock and a reset signal, it's possible to utilize clocks within Xilinx, with purposes ranging from clock division, duty cycles, pulses, and strobes. Clock dividers are useful because they allow us to create lower frequency clock signals from an input, changing the time period. Duty cycles of signals are the percentage of the pulse duration over its total period. To calculate a duty cycle, it is equal to the pulse width in seconds multiplied by the repetition frequency in hertz multiplied by 100. A clock pulse is a signal that allows us to sync up the operations of our system, and last, strobes may act as a clock pulse/become a synchronization signal, when there is no synchronous clock in place.

Module Design

In this lab, we had to create a clock generator made up of four submodules, each with tasks. The modules were for exploring clock division by a power of 2, even clock division, odd clock division, and pulse/strobe/flag.

Let's start with the first submodule: clock divider by the power of 2's.
We notice that the least significant bit of the counter is a direct division by 2, as noted in the lab handout. As a result, to divide by the power of 2's, we just needed to assign a wire to each of the bits from the counter. I assigned clk_div_2 to the first bit, clk_div_4 to the second bit, clk_div_6 to the third bit, and clk_div_8 to the last bit.

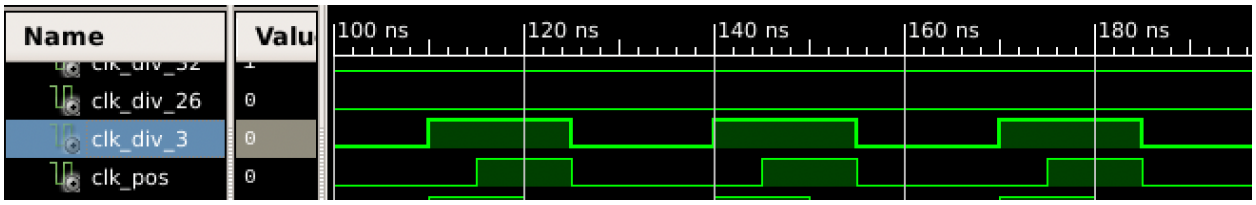
Our second submodule was creating an even division clock using counters. This is trickier than the last, though it was still a continuation since we use the same four-bit counter. To avoid overflow, we had to flip the clock each time the counter passes half of the time (32). For example, in order to divide by 32 clocks, I flipped the clock and reset it to zero every time it went past 15 (1111) since 32/2 =16, and we would have to flip by 16 as the counter starts at zero. If the counter did not hit 15, then we would continue adding one to it.

For the design task, we had to generate a clock that is 26 times smaller by modifying it when the counter resets to 0. This is essentially the same method as before: we have to flip the clock each time the counter passes half of the allotted time (in this case, 28). Every time the counter went past 12 (1101) since 26/2 =13, we would flip the clock as once the counter reaches 13, it would reset.

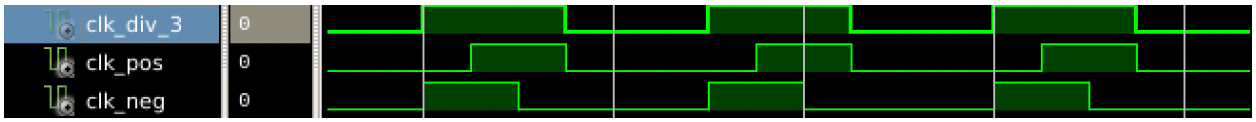
Our third submodule was an odd division clock using counters. We first had to generate a 33% duty cycle clock. This means that we need to make a clock that is active for a third of the time. That would mean we would need something tracking in 3's, and I turned towards a two-bit counter. This counter is able to generate three different scenarios 2'b00, 2'b01, 2'b10, and we are able to spot that once the most significant bit is one, then we can reset the counter back to 0. If not, we continue increasing by 1. As a result, the clock would go like this:
00 → 01 → 10 → 00 → 01 → 10 → 00 … and so on…

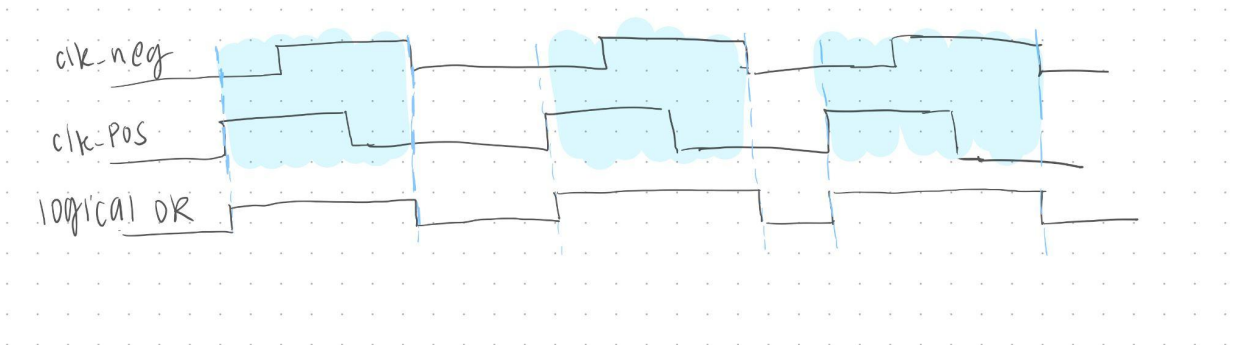
The next task was to have a block that triggers the falling edge. This was simple: I created the same always block but changed
always @ (posedge clk_in) to always @ (negedge clk_in).

Here is my verification of the waveform; you see that the clock is only active 33% of the time.



By taking the logical OR of the two 33% clocks, we get a signal that takes both the positive and negative edge duty cycles. Looking at the waveform below, we see that the negative and positive clocks mesh together when taking the logical OR.





Our design task was to create a 50% duty cycle divide-by-5- clock. This combines the past few tasks that we did. Just like before, I realized we need to make a clock active for a fifth of the time, so we would need to track in 5's. This can be done using a three-bit counter. Once the most significant bit becomes 1, we can reset the counter back to 0. If not, we continue increasing by 1. The clock would go like this:
000 → 001 → 010 → 011 → 100 → 000 → 001 … and so on …
Then, to divide by five, we just need to flip the clock every time it reaches its fifth state. Since we are not allowed to redefine clk_fiv_5 in two different always blocks, I combined both the positive and negative sides, shown in
always @ (posedge clk_in or negedge clk_in)

The last submodule was pulse/strobes. We first had to create a divide-by-100 clock with only a 1% duty cycle. Since tracking to 100 is a lot more difficult than tracking to three or five, I utilized a 100-bit counter with the starting bit as 1 and the rest of the bits were set to 0's. Every time the clock increased, we would set the "next bit" as 1. The MSB of the counter at position 99 was the output of the 1% duty cycle. As noted in the lab handout, I then created a second always block that runs on the system clock and switches the output clock every time the divide-by-100 is active. This can also be verified by a 50% duty cycle divide-by-200 clock running at 500Khz. If a[99] was equal to 1 (meaning that there were 100 cycles), we would flip the clock. The resulting waveform would be 200 times slower than the system clock. A 50% duty cycle divide-by-200 clock running at 500 kHz is also 200 times slower.

Our design task for this submodule is to create a divide-by-4 strobe to generate an 8-bit counter that counts up by 3 on every positive edge and subtracts by 5 on every strobe. I did something similar to the last part, using a 4-bit counter to track and divide by 4. Then, I utilized a second always block to check if the MSB of a[3] is equal to 1, then we have a strobe and subtract the toggle_counter by five. If a[3] was not equal to 1, then we know that we have a positive edge, and we can increase the toggle_counter by 3.

Testbench Design

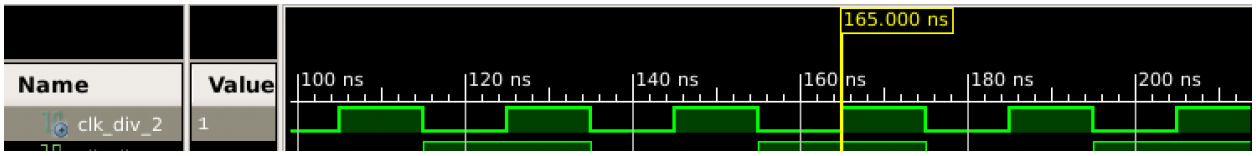
To code my testbench, I initialized the inputs by setting clk_in to 0 and the rst to 1. This starts the testbench. After 100 ns, I flipped the clk_in and rst input. The lab handout stated to set the testbench to 100 Mhz, and I kept track of this with the statement always clk_in = #5 ~clk_in;
since clk_in needs to flip every #5 for a 100Mhz.

My module passed all cases and fit every waveform. To validate these numbers and calculations, I double-checked my waveforms with the results provided in class.

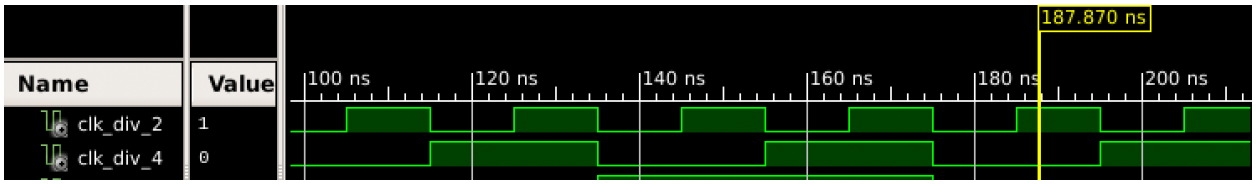
The waveforms for all verify tasks and design tasks will be shown in the next section.

Simulation Waveforms

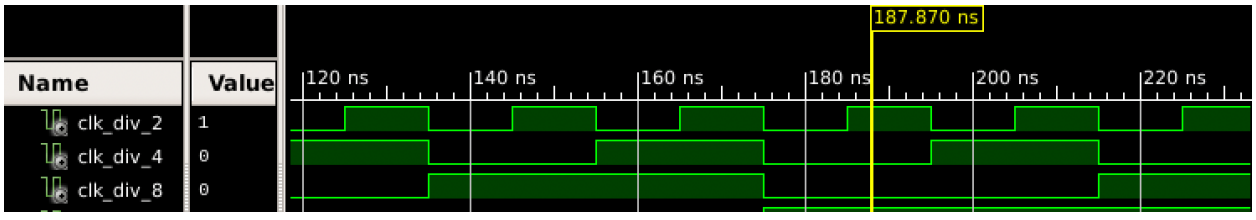
1. Divide by 2 clock (task 1)



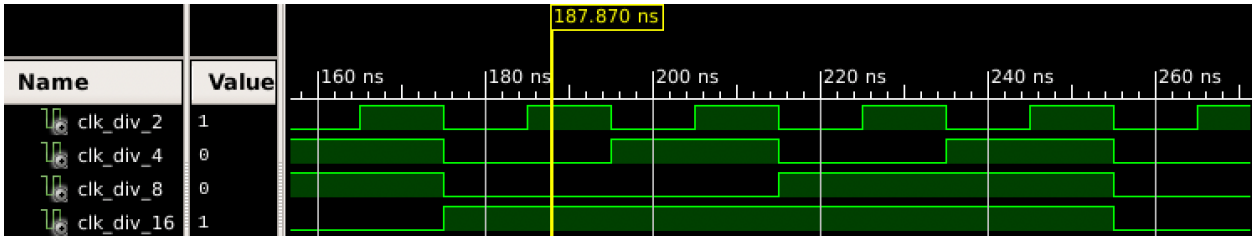
2. Divide by 4 clock (task 1)



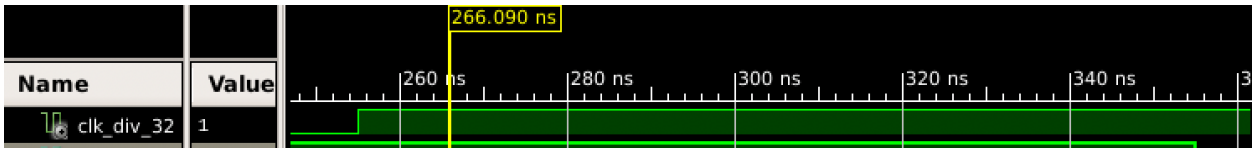
3. Divide by 8 clock (task 1)



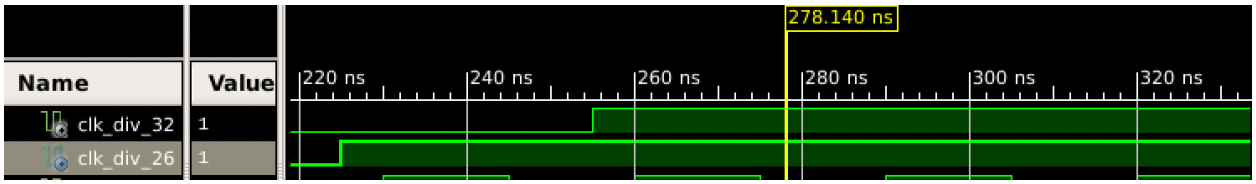
4. Divide by 16 clock (task 1)



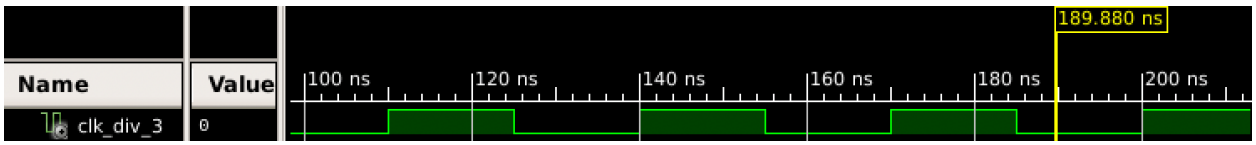
5. Divide by 32 clock (task 2)



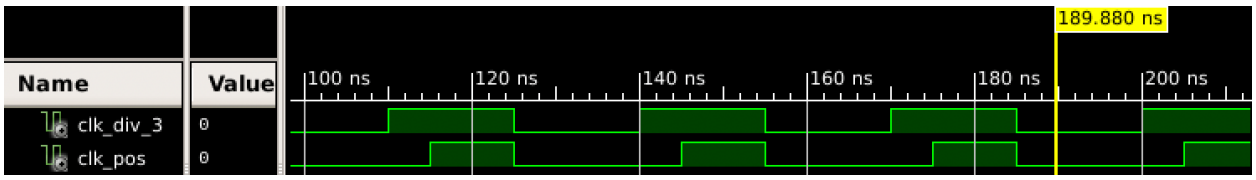
6. Divide by 26 clock (task 3)



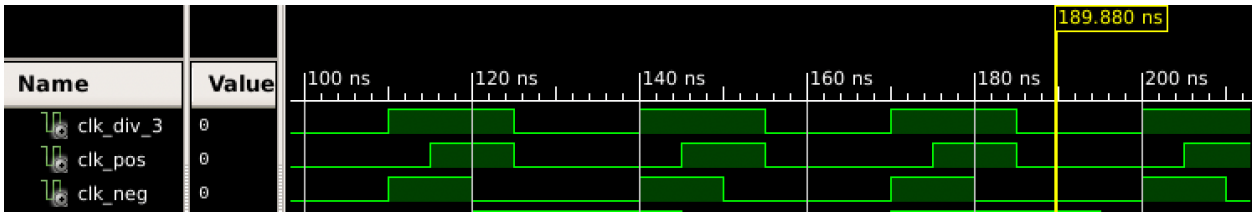
7. Divide by 3 clock (task 6)



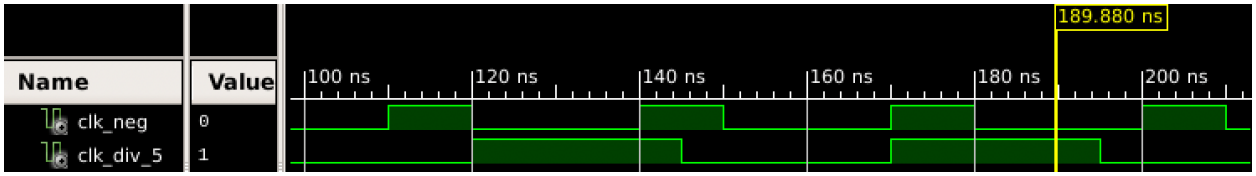
8. 33% duty clock (task 4)



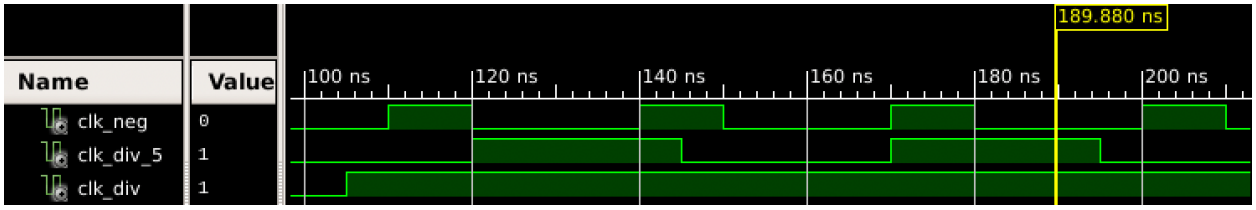
9. 33% duty clock with negative edge triggered (task 5)



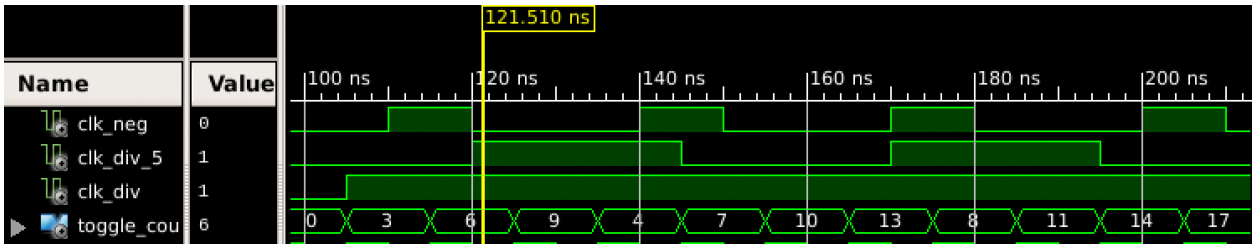
10. Divide by 5 clock (task 7)



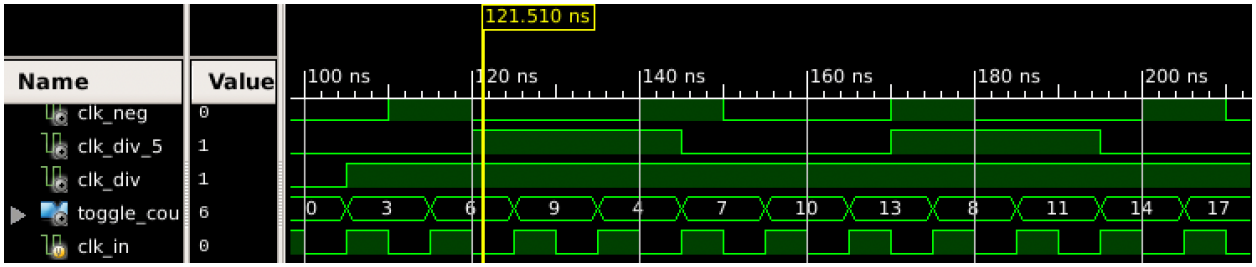
11. 500 kHz clock (task 8)



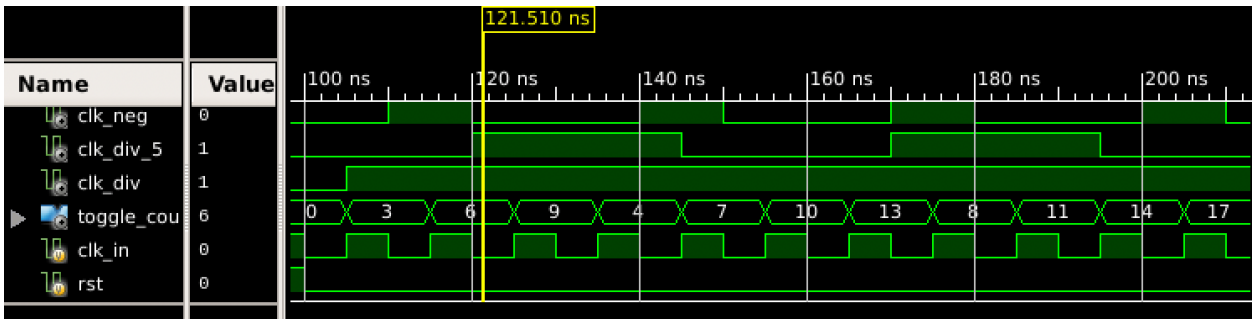
12. Glitchy counter (task 9)



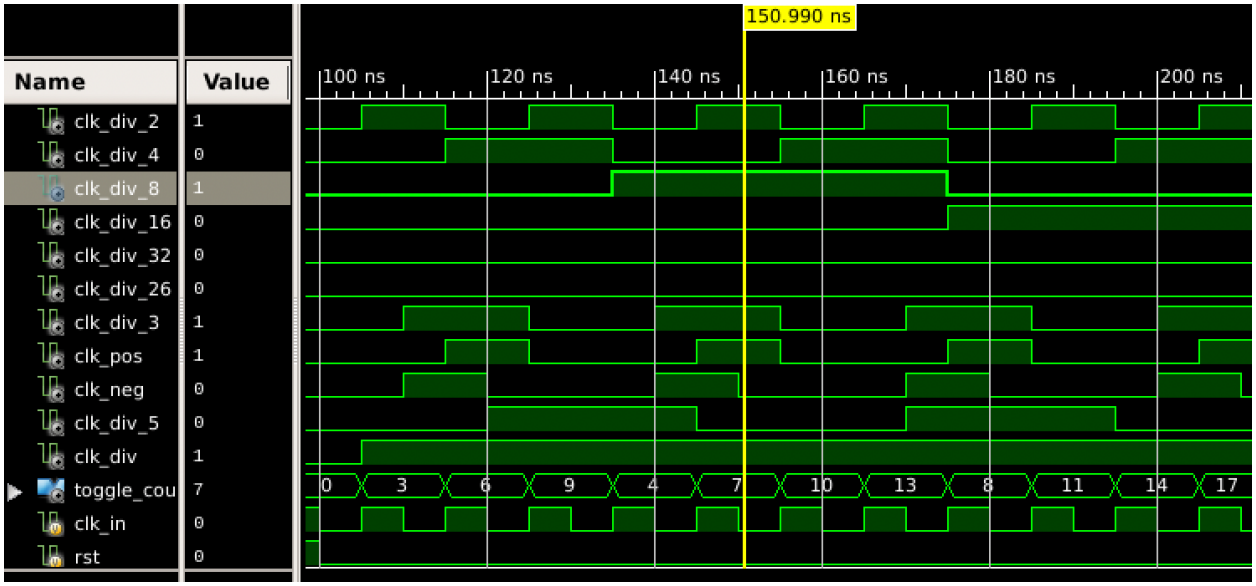
13. Input clock



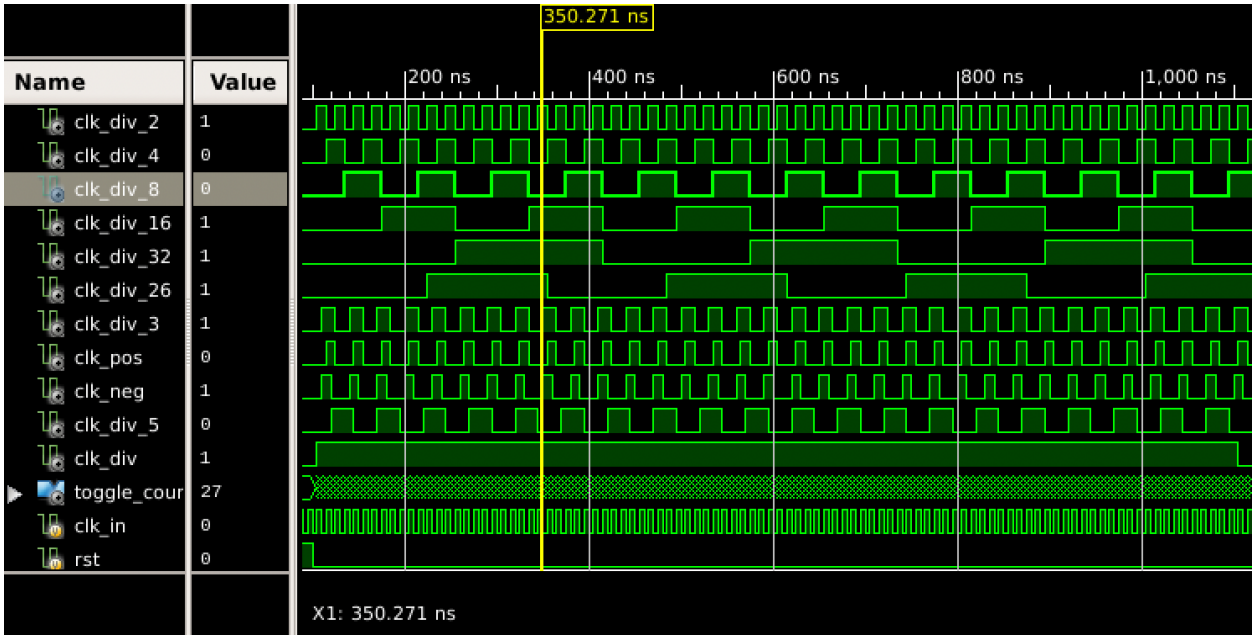
14. Reset Signal



All 14 waveforms (VM screen is small, so I could not fit 100 ns to 320 ns in the frame)



All 14 waveforms with at least 1000ns in the screen (Again, my VM window is very small and could not capture the entire 2200 ns).



ISE Design Overview Summary Report

https://drive.google.com/file/d/1E1iKXucaf2_Mc4JDahaXkZBdh3PXWFMq/view?usp=sharing

Conclusion

We experimented with multiple ways to manipulate the system clock: dividing by n, duty cycles, pulses, and strobes. It's easier once we see all 14 waveforms total the differences between each task we were required to do. I was very surprised to see almost every submodule played off of the 4-bit counter, and how clocks pretty much revolved around counters. The most difficult part of the lab was figuring out the division by n and duty cycles. I was very confused on how to divide by 100 since system clocks were such a new concept for me. However, once you break it down, draw out diagrams, and focus on the most significant bit, it becomes a lot easier to understand. In addition, I had a lot of trouble with the testbench because I had originally set the rst to 0, so my clock was starting at the wrong time (around 200ns instead of 100ns).