CS M152A Project 2 Report: Clock Design Methodology

Belle Lerdworatawee, Lab 1
TA: Ananya Ravikumar
Feb 7, 2021

Introduction and Requirement:

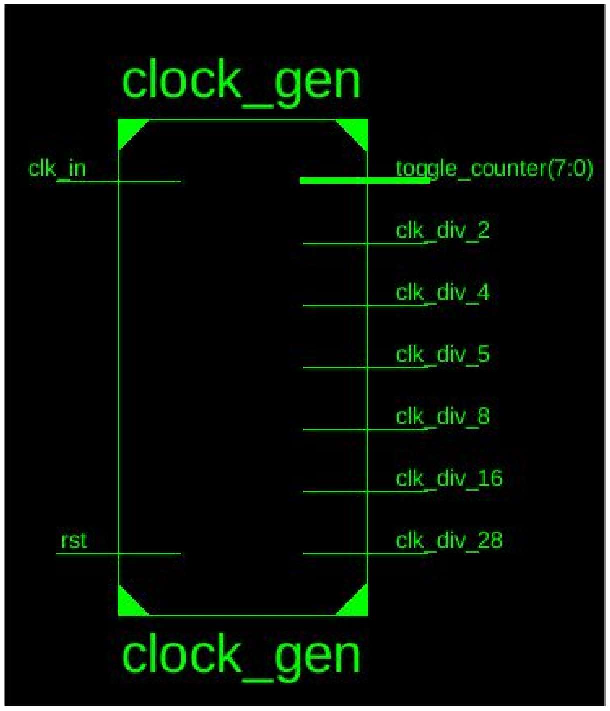
This lab focused on designing and testing clocks with various pulses and duty cycles. Clocks are crucial for synchronous designs, state machines, ensuring correct data transmission. We first needed to generate a system clock of frequency 100MHz and then use it to generate the other clocks. These other clocks included divide-by-N clocks meaning the clock period is of length N. In addition, we also created X% duty cycle clocks where the clock is high for X% of its period. Lastly, the lab required us to design a strobe which is essentially a synchronization signal when a synchronous clock is not present. Based on this strobe and system clock, we make a “glitchy” counter that decrements by 5 every 4 cycles and increments by 2 otherwise. After completing all tasks, we write a testbench and analyze all the waveforms.

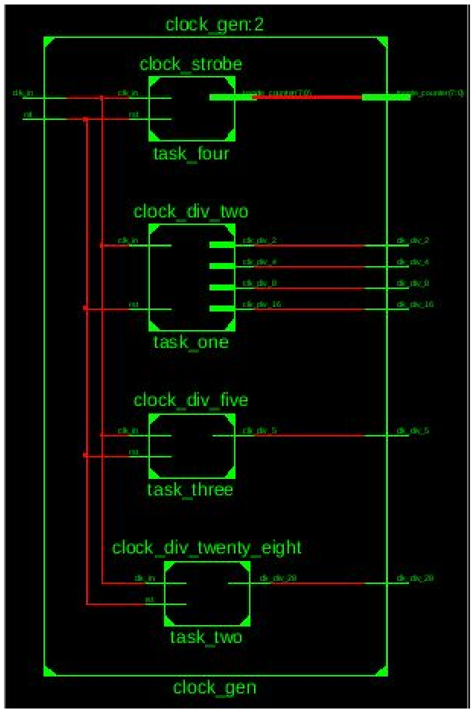
Design Description

For the design tasks, I had one top module—clock_gen—and 4 submodules within it, each generating a different clock. Additionally, I created separate modules for each verify task. Every module had the same input: clk_in and rst. For all modules, I nest an if statement inside an always @(edge) block to check if rst is high. If so, then I reset the counter and any signals to 0. I will discuss the unique outputs further in this report.

Modules:

1. <u>clock_gen</u> — This module is responsible for generating the design task waveforms. There are 7 outputs: clk_div_2, clk_div_4, clk_div_8, clk_div_16, clk_div_28, clk_div_5, and toggle_counter. I instantiated submodules and assigned the outputs of clock_gen as the outputs of the submodules. The first 4 outputs are determined by the clock_div_two submodule. The next one is generated by the clock_div_twenty_eight module. clk_div_5 was generated by the clock_div_five module and lastly, toggle_counter is generated by clock_strobe. None of these submodules interacted or depended on each other; they each only relied on the inputs clk_in and rst.





Figures 1, 2: ISE generated schematics of the top module, clock_gen

2. <u>clock_div_thirty_two</u> — This module generates a clock signal that repeats every 32 system clock cycles. Its output is clk_div_32. I implemented this module by placing a series of if statements inside an always block. The always block is triggered by the positive edge of the input clock. Inside the always block, there's also a 4-bit counter that increments every time the signal clock goes high. I use an if statement to check when the counter is about to overflow (reaches 15) and flip the signal by using the negation operator. When the rst input is high, I reset the counter and clock signal to 0.
3. <u>clock_33_duty</u> — This module has a design unlike the ones before it. It outputs 3 clock signals: clk_33_duty_pos, clk_33_duty_neg, and clk_or. The first two are implemented similarly: I nested an if statement inside of an always block and used a 2-bit register as a counter. One always block is triggered on a positive clk_in edge and one is triggered on a negative clk_in edge. The if statements check if rst is high (set the counter and signal back to 0), if the counter is 1 (flip the signal), or if the counter is 2 (set the counter to 0 and flip the signal). The default case just adds 1 to the counter. For clk_or, I took the logical OR of the two signals to generate a 50% duty cycle div-by-3 clock.
4. <u>clock_div_100</u> — This module outputs signals clk_div_100 and clk-div-200 as well as a 7-bit counter, a. The counter is optional but I included it to help confirm that the clock signals are indeed 100 or 200 times slower than the signal clock. The implementation is simple. First I use an always block that is triggered on a positive edge of clk_in. This block contains a series of if else statements that check if rst is high (reset the counter and signals to 0), if the counter is equal to 98 (flip the signal), and if the counter is equal to 99 (set the counter to 0 and flip the signal). Inside the same always block, I have a

separate if statement checking if clk_div_100 is high. If so, then I flip the clk_div_200 signal by using the negation operator.

Submodules:

1. clock_div_two — This module generates a clock signals that are divided by powers of 2. It has four outputs: clk_div_2, clk_div_4, clk_div_8, and clk_div_16. I instantiated an instance of a 4-bit counter and then assigned clk_div_2 to the 0th bit, clk_div_4 to the 1st bit, clk_div_8 to the 2nd bit, and clk_div_16 to the 4th bit. The counter received the inputs clk_in and rst. The rst input only affects the counter which will reset back to 0, and thus set the wires to 0 as well.

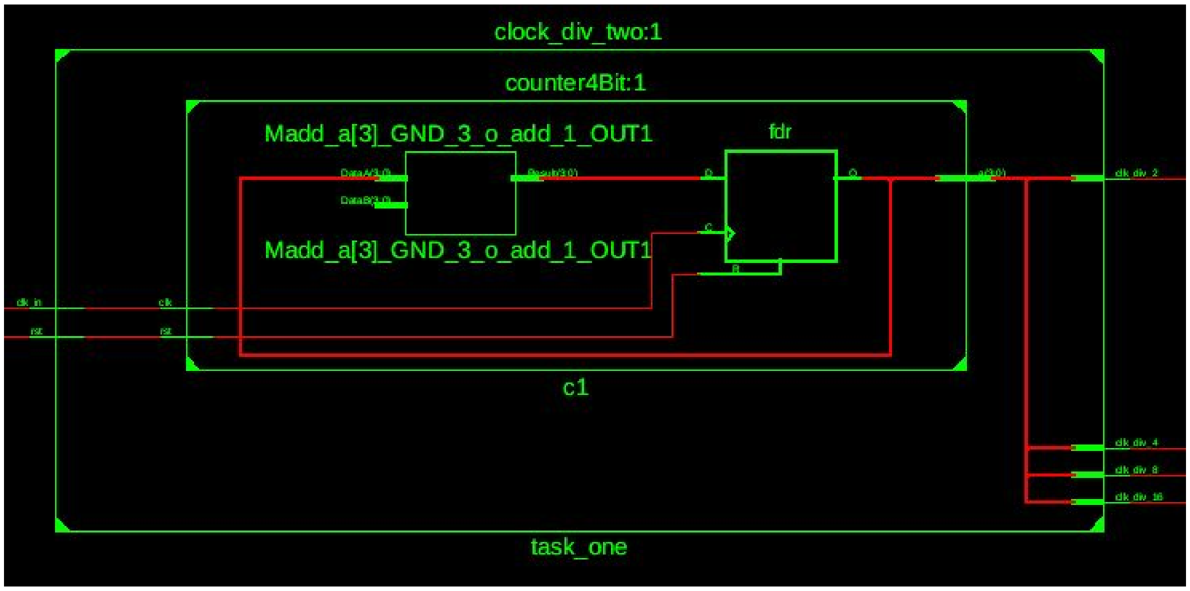
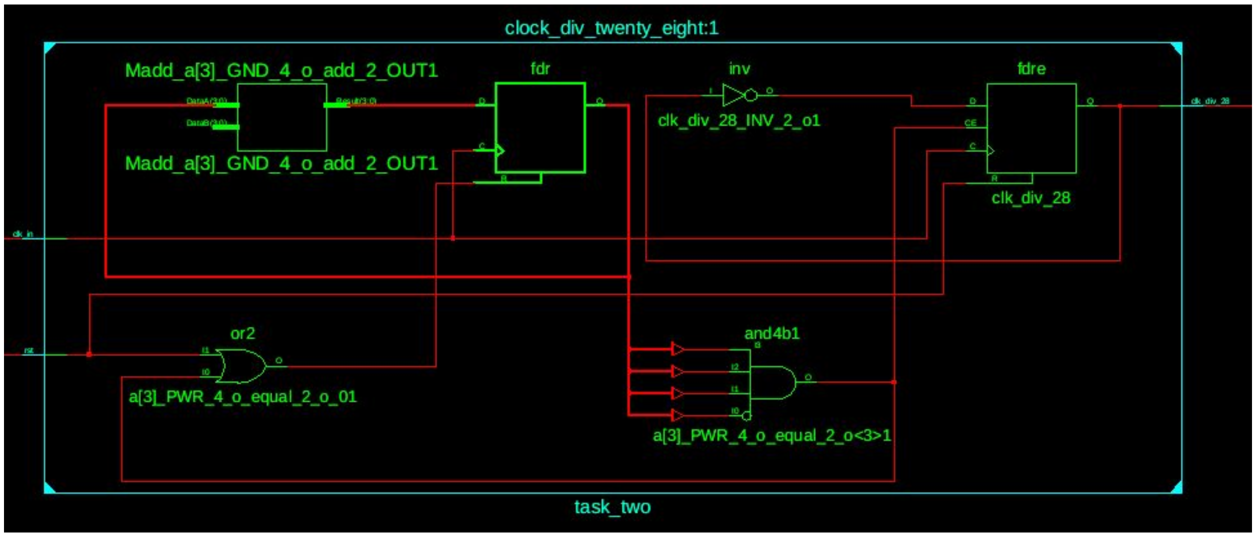


Figure 3: ISE generated schematic of clock_div_two and the inner 4-bit counter

2. clock_div_twenty_eight — Designing the clock_div_thirty_two gave me an idea for this module which generates a clock signal that is 28 times slower than the system clock. The output is clk_div_28. I implemented this module by placing a series of if statements inside a positive edge-triggered always block. Inside the block, I use a 4-bit register as a counter that increments every time the clk_in goes high. One if statement checks if the counter is 13, at which point I use the negation operator to flip the signal and set the counter to 0. Another if statement checks if the rst input is high, and then sets both the counter and clock signal to 0.


clock_div_twenty_eight:1
Madd_a[3]_GND_4_o_add_2_OUT1
fdr
inv
fdre
clk_div_28_INV_2_o1
Madd_a[3]_GND_4_o_add_2_OUT1
clk_div_28
clk_in
rst
or2
and4b1
a[3]_PWR_4_o_equal_2_o_01
a[3]_PWR_4_o_equal_2_o<3>1
task_two
Figure 4: ISE generated schematic of clock_div_twenty_eight

3. <u>clock_div_five</u> — I got the idea for this clock by designing the 33% duty clocks. The output is clk_div_5 which is a 50% duty div-by-5 clock signal. The overarching design is taking the logical OR of two 40% duty div-by-5 clock signals. One of those clock signals is positive edge-triggered and the other negative edge-triggered. These are implemented with two always blocks. They each have a 3-bit counter that counts to 4 before resetting. Both blocks have the same code inside: a series of if statements that check if rst is high (set the counter and signal to 0), if the counter is equal to 2 (flip the signal), and if the counter is 4 (reset the counter and flip the signal). The counter increases by 1 on every positive edge on clk_in. Then I took the logical OR of these two signals to generate clk_div-5.

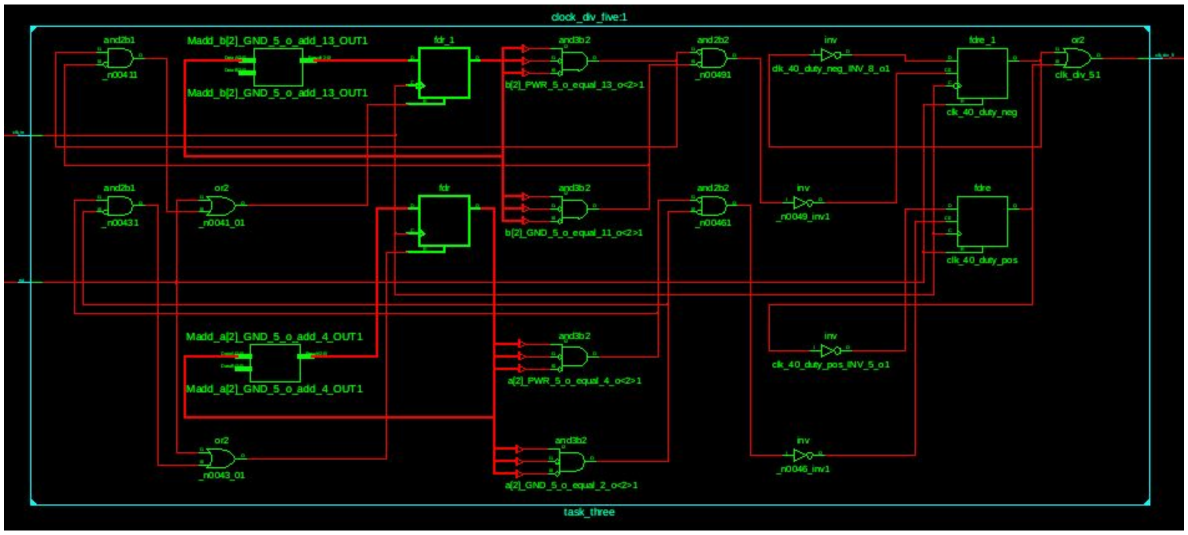
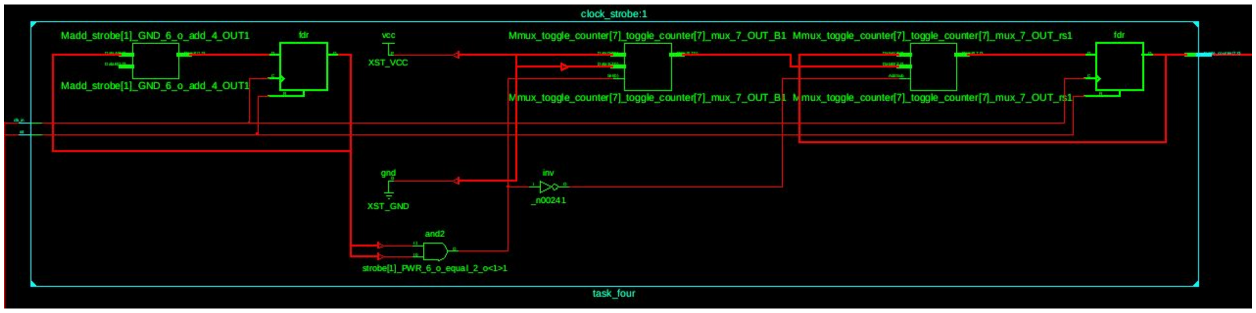


Figure 5: ISE generated schematic of clock_div_five

4. <u>clock_strobe</u> — To design this module, I combined my knowledge that I had gained from completing the previous tasks. It outputs a "glitchy" toggle_counter. First I implemented

the strobe by using a 2-bit register. Inside an always block, the strobe is incremented by 1 when clk_in is high. Everytime it becomes 3, it will become 0 on the next increment due to bit overflow and truncation. There are if else statements inside the always block that check if rst is high (set the strobe and toggle_counter to 0), and if the strobe is equal to 3. For this case, toggle_counter subtracts 5 from itself. The default case is that the strobe increments by 1 and the toggle_counter increments by 2.


clock_strobe:1
Madd_strobe[1]_GND_6_o_add_4_OUT1
fdr
vcc
XST_VCC
Mmux_toggle_counter[7]_toggle_counter[7]_mux_7_OUT_B1
Mmux_toggle_counter[7]_toggle_counter[7]_mux_7_OUT_rs1
fdr
Madd_strobe[1]_GND_6_o_add_4_OUT1
Mmux_toggle_counter[7]_toggle_counter[7]_mux_7_OUT_B1
Mmux_toggle_counter[7]_toggle_counter[7]_mux_7_OUT_rs1
gnd
XST_GND
inv
_n00241
and2
strobe[1]_PWR_6_o_equal_2_o<1>1
task_four
Figure 6: ISE generated schematic of clock_strobe

5. <u>counter4Bit</u> — I used the code from the lab description. This module uses a 2-bit register to serve as a counter and outputs numbers from 0 to 3 in order. It is implemented by placing a simple two-case if statement inside an always block. The block is triggered when clk_in becomes high. The first case is when rst input is high in which the counter is reset to 0. Otherwise, the counter is incremented by 1. Because the register used to store the counter is only 2 bits wide, the counter will become 0 again when it is 1111 and adds 1 to itself due to bit overflow and truncation. So I did not make another case to set the counter back to 0 again.

Simulation Documentation

In order to determine if the modules successfully generated the right waveforms, I created a system clock in the testbench and instantiated modules of each to test. They are all in one testbench, and to test them individually, I would comment out the modules that I wasn't testing. I made the rst input high for 10ns before keeping it low for the remainder. To make the system clock, I initialized clk_in to low and then flip it every 5ns.

<u>Design Task 1</u> Clock Divider by Power of 2s: For this task, I instantiated a sub module of the 4-bit counter and analyzed each of its bits' signals. I compared it to the signal clock as well. The image below demonstrates how the 0th bit is twice as slow as the system clock, the 1st bit is four times as slow and so on. Thus we find that we can easily generate divide-by-power of 2s clocks by assigning wires to the corresponding bits of a counter. For a divide-by-N, assign a wire to the ($\log_2$(N) - 1)th bit of the counter.

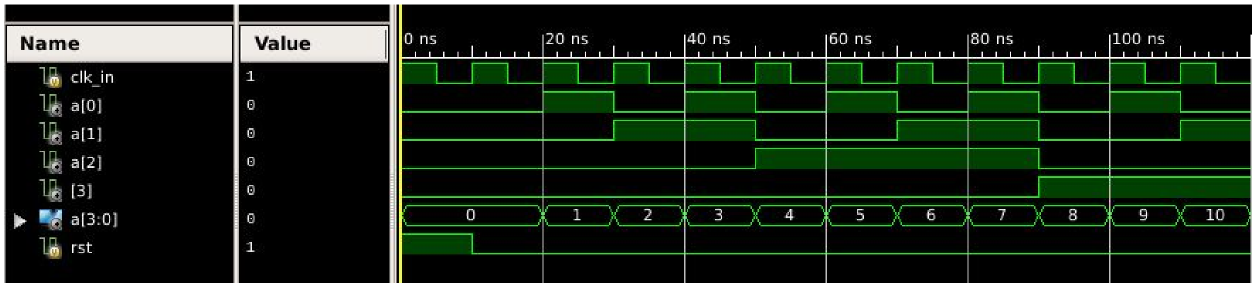


Figure 7:— waveform for each bit of the 4-bit counter

Design Task 2 Even Division Clock Using Counters: I instantiated an instance of clock_div_thirty_two in the testbench and used the clock_div_twenty_eight in the clock_gen module. When analyzing the waveform for divide-by-28 and divide-by-32, we can observe that the signal is high for 16 system clocks periods which is as expected. The waveform for the divide-by-28 clock is shown in Figure 14, and similarly, we can observe that it is high for 14 clock system clock signals. From this, we can deduce that to generate even divide-by-N clocks, we simply need to flip the signal every N/2 system clock periods.

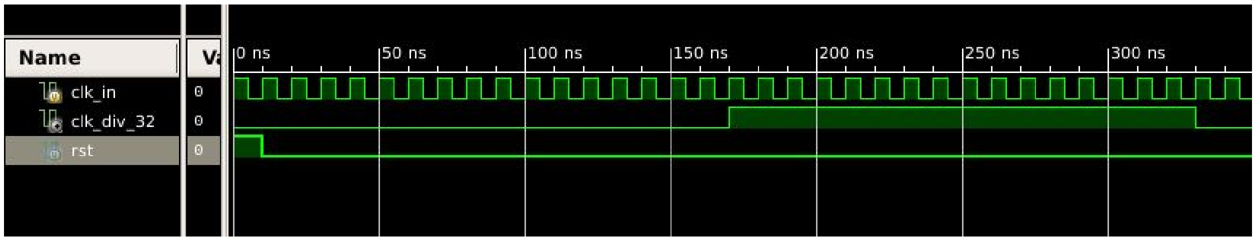


Figure 8: waveform for a divide-by-32 clock

Design Task 3 Odd Division Clock Using Counters: For this module, I first tested the positive edge-triggered 33% duty cycle clock to ensure it was correct. I had a bug where the waveform was correct for the first period and then became 50% duty. I fixed this bug during office hours and it was the result of my forgetting to increment the counter in every if statement case. It works well as we can observe from the waveforms. They both are high for 1 system clock cycle and low for 2 cycles. When taking the logical OR of them, we see that it results in an 50% duty cycle odd division clock. The waveform for the divide-by-5 clock is generated in the exact same manner; we can observe below that generating and taking the logical OR of two 40% clocks will result in a divide-by-5 50% duty clock. Therefore, odd divide-by-N clocks can be made by OR-ing one positive edge and one negative edge triggered clock.

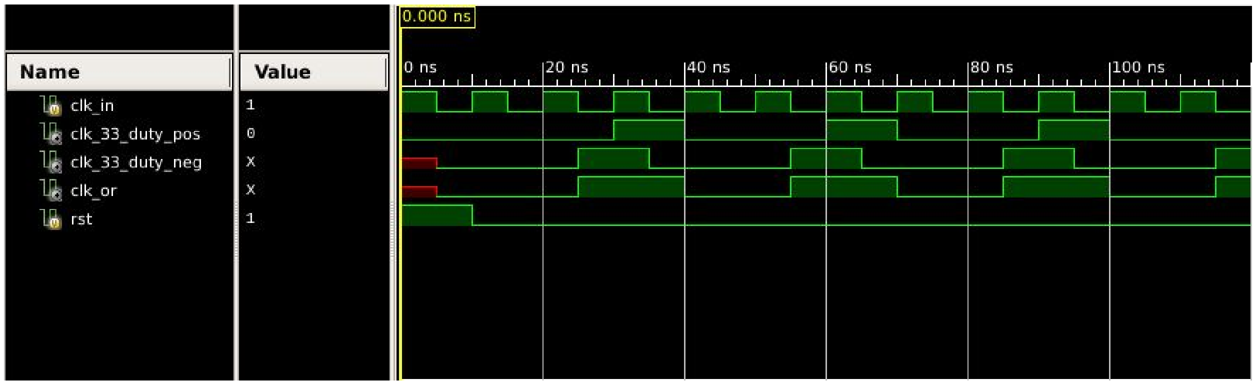


Figure 9: waveform for 33% duty clocks triggered on posedge and negedge; waveform for the logical OR of the two 33% duty clocks

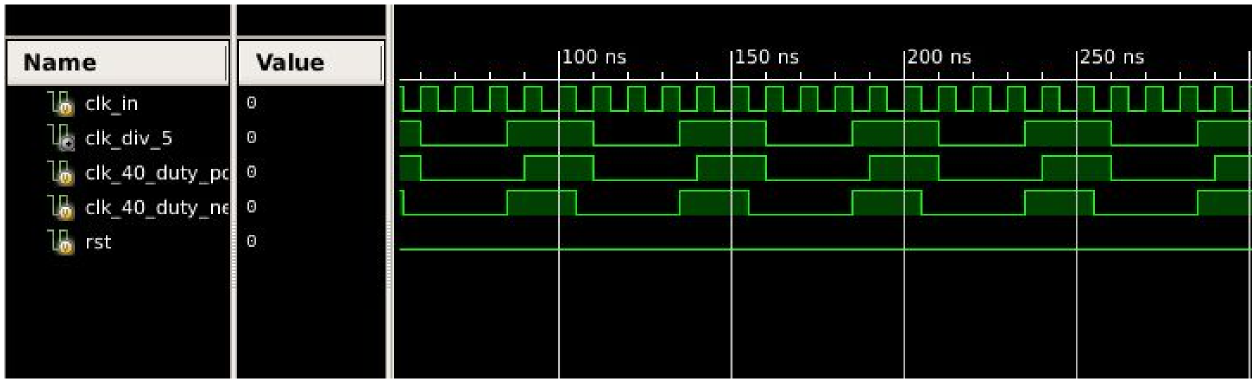


Figure 10: waveform for 40% duty clocks triggered on posedge and negedge used to create a 50% duty divide-by-5 clock

Design Task 4 Pulse/Strobes: To test this module, I included a counter to see when clk_div_100 was high for 1 clock cycle and low for 99. The counters represent how many signal clock cycles have passed. When analyzing the waveforms for clk_div_100 and clk_div_200, we can observe in Figures 12 and 13 that the first signal flips after 98 clock signals which means it is low for 99 cycles. Similarly, for clk_div_200, we can see that it becomes high when the counter is 0 and becomes low when the counter reaches 99 which is a total of 100 clock cycles. It is also low for 100 clock cycles, so this is correct.

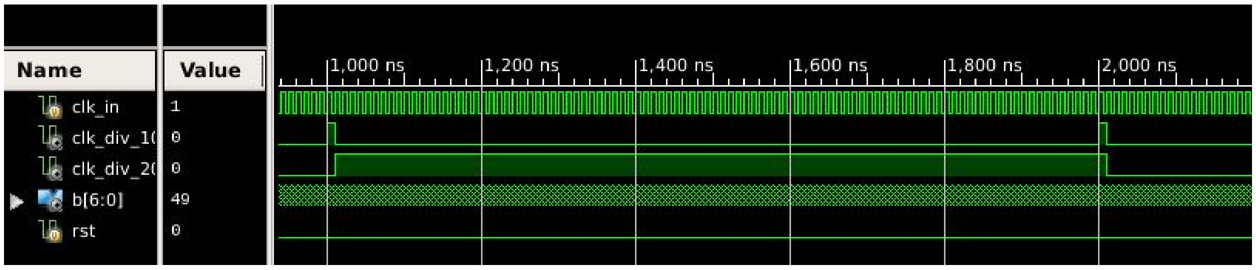
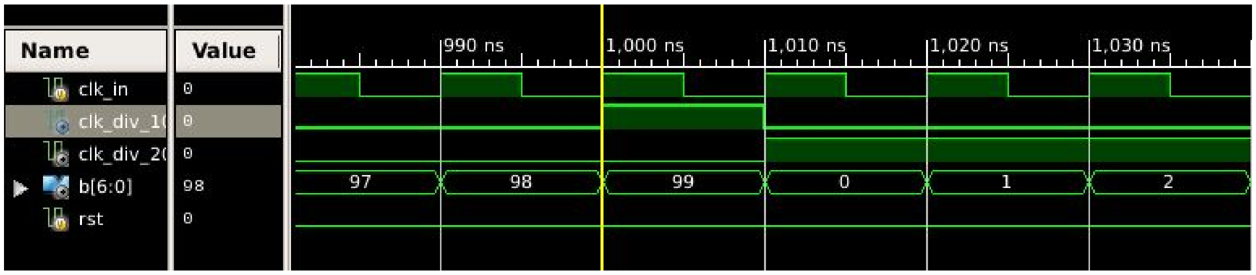
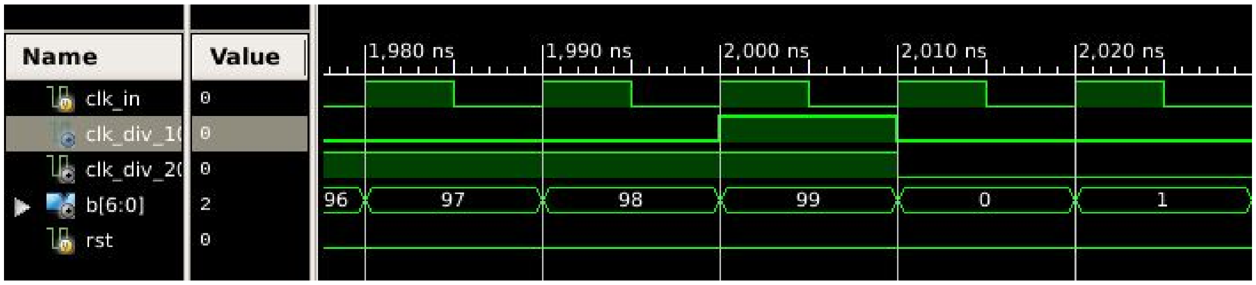


Figure 11: zoomed in waveform for a 1% duty cycle divide-by-100 clock and a 50% duty cycle divide-by-200 clock





Figures 12,13: close up waveform displaying the start and end of a period of a 1% duty cycle divide-by-100 clock; closeup of when the waveform is high for a 50% duty cycle divide-by-200 clock

clock_gen: I changed the radix of Xilinx ISim from binary to decimal to make it easier to confirm toggle_counter. I tested this module after making sure that all submodules functioned properly. There is a minor difference between this image and the example given in the lab description which is the initial low period. The example in the description flips the signal in the middle of its clock period whereas I flip the signal near the end of the period. Other than that, they are identical and we can conclude that clock_gen works as expected.

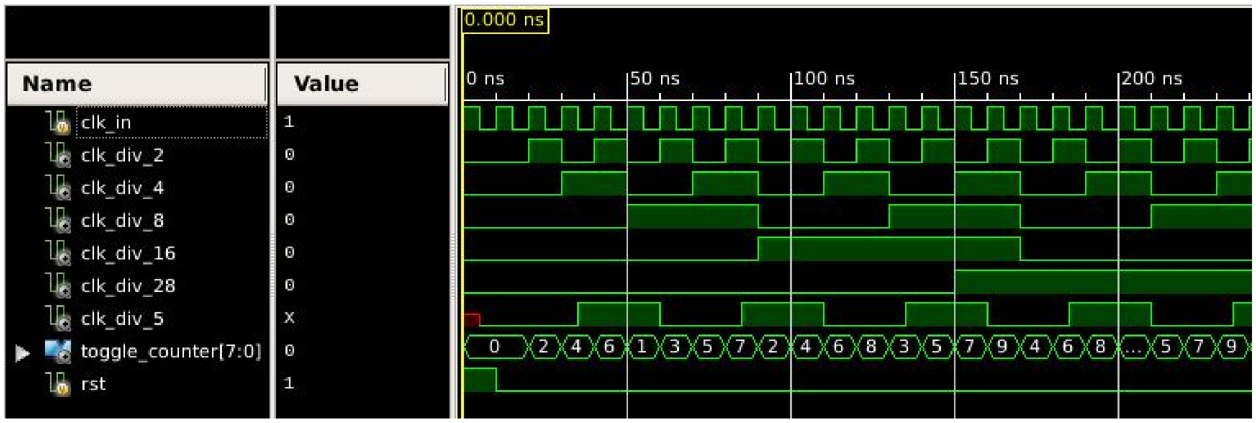
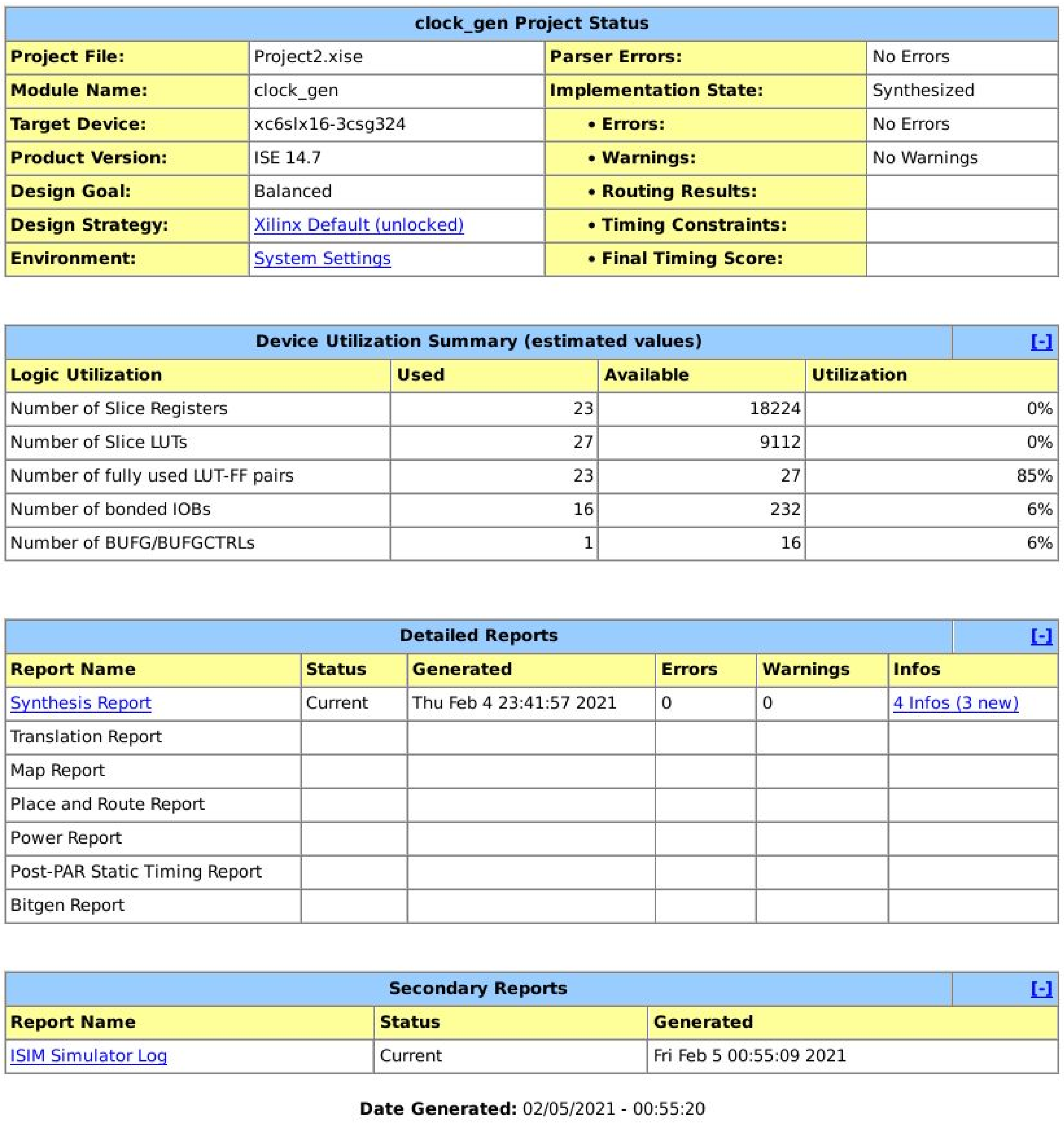


Figure 14: waveform for the final clock generator showing all 9 ports

As you can see from the design report, there are no errors or warnings. The code is completely synthesizable.



clock_gen Project Status			
Project File:	Project2.xise	**Parser Errors:**	No Errors
Module Name:	clock_gen	**Implementation State:**	Synthesized
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	No Warnings
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	23	18224	0%
Number of Slice LUTs	27	9112	0%
Number of fully used LUT-FF pairs	23	27	85%
Number of bonded IOBs	16	232	6%
Number of BUFG/BUFGCTRLs	1	16	6%

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Thu Feb 4 23:41:57 2021	0	0	4 Infos (3 new)
Translation Report					
Map Report					
Place and Route Report					
Power Report					
Post-PAR Static Timing Report					
Bitgen Report					

Secondary Reports		[-]
Report Name	**Status**	**Generated**
ISIM Simulator Log	Current	Fri Feb 5 00:55:09 2021

Date Generated: 02/05/2021 - 00:55:20

Figure 15: ISE Design Overview Summary Report

Conclusion

In summary, I designed several modules that generate new clock signals based on the 100MHz system clock. The top module uses individual modules to generate unique clock signals. Then I created other modules to handle the verification tasks. Although each module generates a unique clock signal, they all roughly follow the same implementation: if statements nested inside an always block. Each one used a counter to control when the signal flipped, and the counter resets after a period of the desired clock cycle passes. I used the Lab recording and slides to figure out how to generate specific clock pulses. The main difficulties I encountered were with figuring out how to implement the X% duty clocks and I ran into memory issues with the Xilinx software. I solved the X% duty clocks by drawing out the desired waveform compared to the system clock, and used my diagram to figure out at what value of the counter I should flip the signal on or reset the counter. I handled the Xilinx error by clearing the trash to free space. I don't have any suggestions for the lab; it was straightforward and I rewatched the discussion for reference when I had confusions about the lab spec. From this lab, I learned how to use counters and strobes to manipulate a system clock to generate different clock signals.