1. Introduction

Most digital systems are built upon a synchronous structure with a clock signal for timing. Clock signals are especially useful in synchronizing expected behavior for testing. The problem is that digital systems cannot possibly house all the possible different speeds for clocks. Some clocks run at the 4GHz while many clocks run much slower at 500KHz. Digital typically only have one or two clocks. The purpose of this project is to design hardware to enable dynamic clock behavior when limited to only one input clock.

2. Design

The top level design of this project has the following structure.

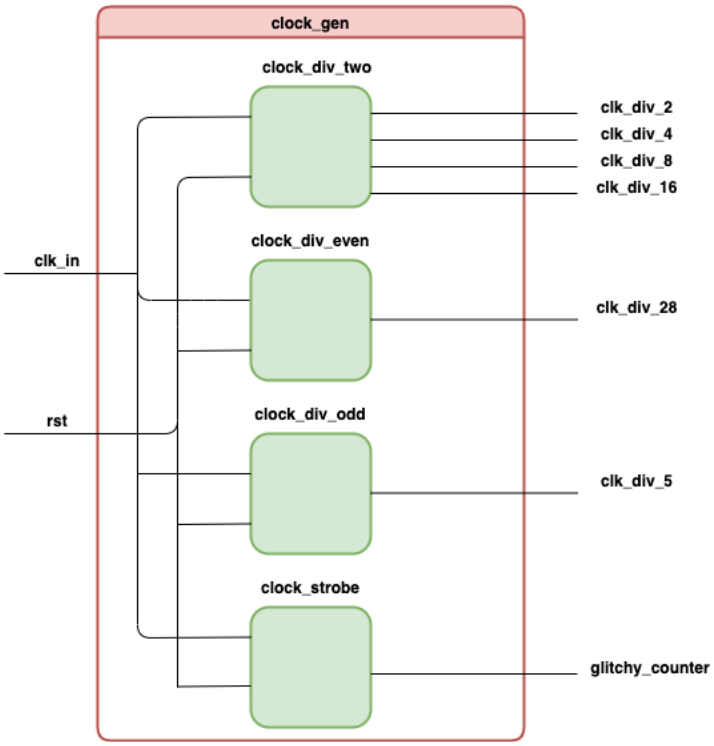

Figure 1: *Top level design of clock_gen* – The top level module, clock_gen, instantiates 4 different submodules.

Clock_gen is the top level module of the of the project and instantiates four different submodules. The clock_div_two module takes in the clock and reset signals as inputs and outputs 4 slower clock signals. The clock_div_even module takes in the same inputs and outputs a divide-by-28 clock. The clock_div_odd module outputs and divide-by-5 clock. Finally, the clock_strobe module increments a glitch counter by 2 every time the positive edge of the clock is reached. The exception is when a pulse signal is toggled on the 3rd negative edge, in which the counter is decremented by 5. The implementation details will be covered in the following section.

3. Tasks

Top Level Waveform

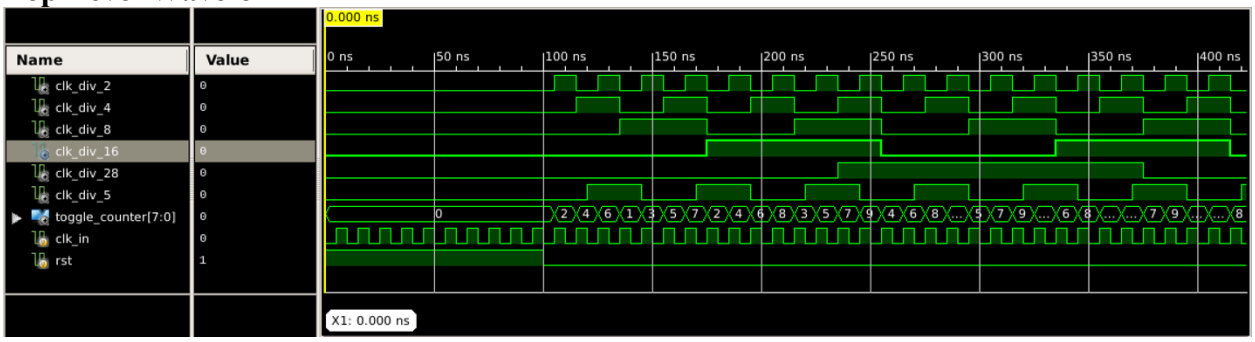

Figure 2: *Output Waveform of Top Level Module* – Stimulated clock running at 100 MHz with period of 10 ns.

The figure above shows the output of the top level module clock_gen, which is a aggregation of all the design tasks.

3.1. Design Task 1: Clock Divider by Power of 2s

Implementation

The clock divider by powers of 2s was implemented with a 4-bit counter.

Waveform

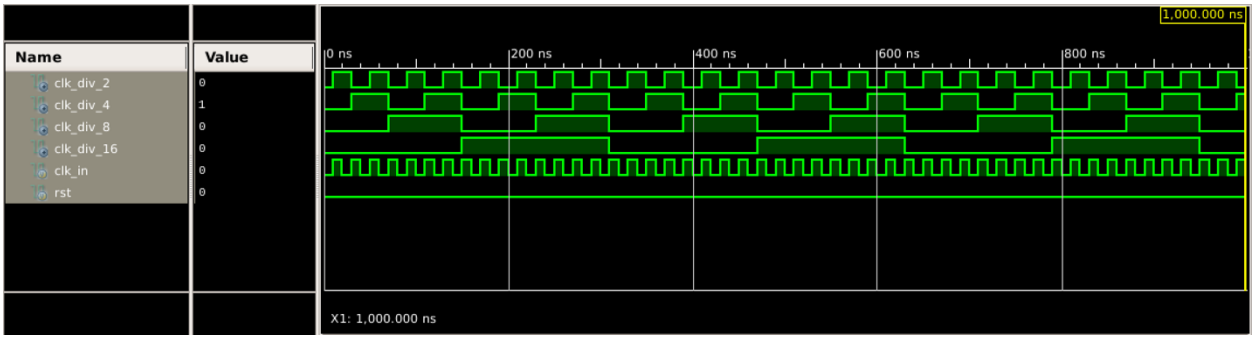

Figure 3: *Output Waveform of Clock Divider by Power of 2s* – Stimulated clock running at 50 MHz was used to test the output of the clock dividers. The period of the input stimulated input clock is 20 ns.

The clock divider by powers of 2s slows the input clock by powers of two. The original input clock has a period of 20 ns. When we zoom into the waveform we can see that each increase in division power, increases the period by a factor of two. For instance, clk_div_2 has a period of 40 ns and clk_div_4 has a period of 80 ns. For visualization, the waveform of each successive clock division (by power of 2) has a cycle that is 2 times larger than the previous clock division.

3.2 Task 2: Clock Divide-by-32 Clock

Implementation

The clock divider by 32 continues to build on the foundations of the 4-bit counter. However, when the counter overflows (i.e. back to 4'b0000), the output signal for the clock (clk_div_32) is inverted.

Waveform

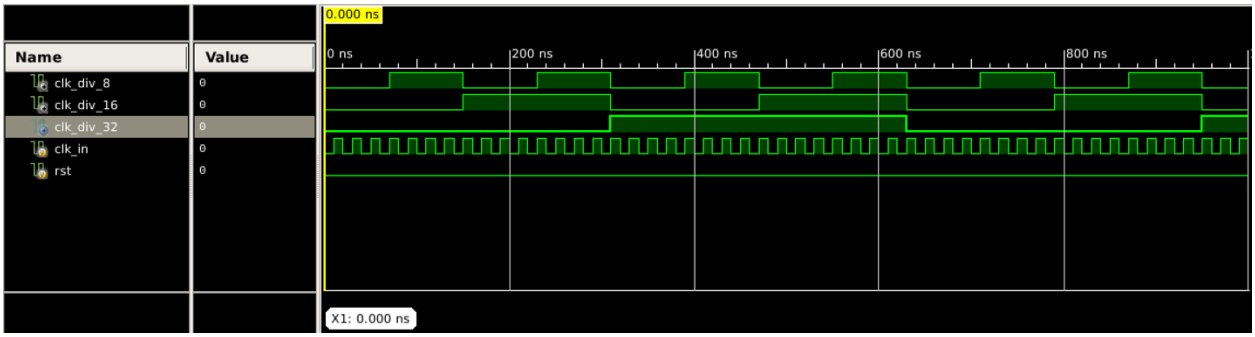

Figure 4: *Output Waveform of Divide by 32 Clock*– Stimulated clock running at 50 MHz with 20 ns periods. For reference clk_div_8 and clk_div_16 are included.

Its difficult to count out 32 positive edges of the input clock on the waveform. However, I have included the divide by 8 clock (clk_div_8) and the divide by 16 clock (clk_div_16) for simple reference. The period of the divide by 32 clock is 2 times greater than clk_div_16 and 4 times greater than clk_div_8.

3.3. Design Task 3: Divide-by-28 Clock

Implementation

The implementation for the divide by 28 clock takes inspiration from the previous clock dividers. It also uses a 4-bit counter. However, in the divide by 28 clock, the counter resets to 4'b0000, when the 14^{th} bit is reached. When the 14^{th} bit is reach, the output is also inverted.

Waveform

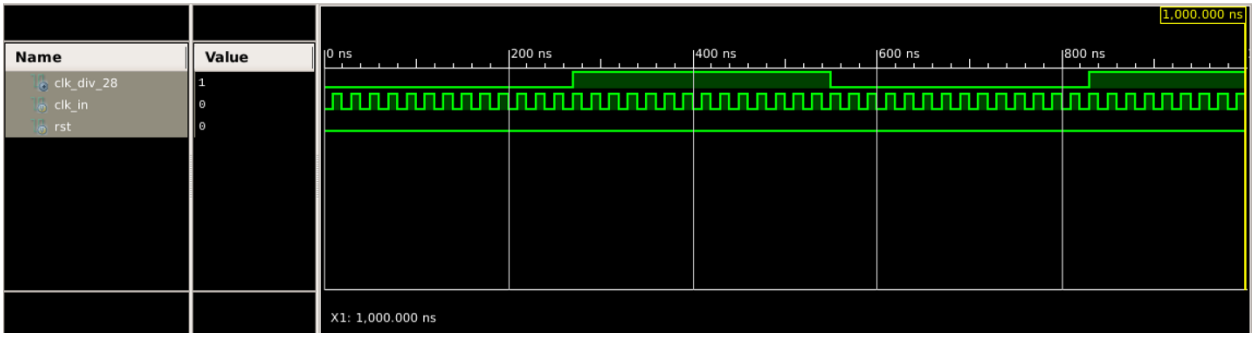

Figure 5: *Output Waveform of Divide by 28 Clock*– Stimulated clock running at 50 MHz with 20 ns periods.

The stimulated clock runs at the same rate as the previous two tasks. If we examine the waveform, we can see that 28 positive edges from the input clock fit into one cycle of the output clock. Additionally, one cycle of clk_div_28 is 560 ns, which is 28 times slower that 20 ns of the input clock.

3.4. Task 4: 33% Duty Cycle Clock

Implementation

The 33% Duty Cycle Clock was implemented with a two-bit counter. When we count to 2, our hardware toggles on the output clock. On the next cycle (i.e. third bit), we reset the counter and set the clock back to 0.

Waveform

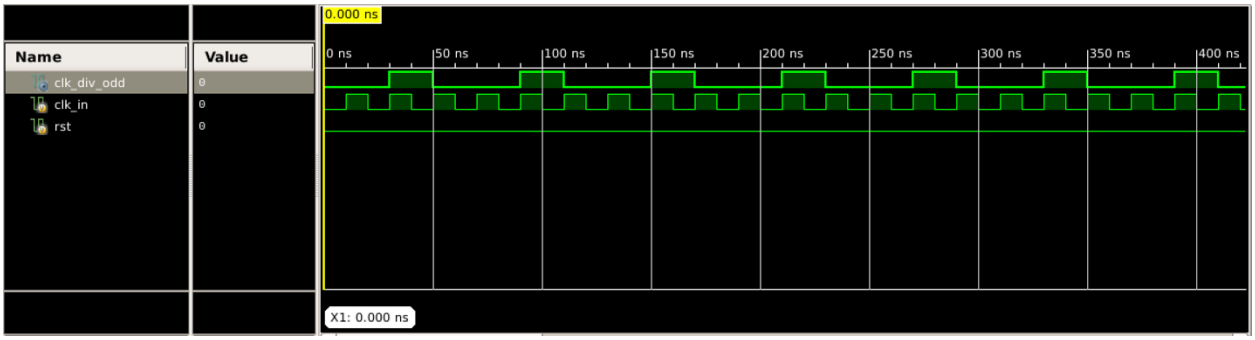

Figure 6: *Output Waveform of 33% Duty Cycle Clock*– Stimulated clock running at 50 MHz with 20 ns periods.

In the 33% duty cycle clock, we can immediately see that for every 3 clock cycles, one clock cycle is toggled on. If we do a closer examination, we can see that one period for clk_div_odd beings at 50 ns and ends at 120 ns. The period of this clock is 60 ns. Since the toggle on is equal to one clock cycle, it is toggled on for 20 ns. 20 ns divided by 60 ns is 33%, which gives us the desired duty cycle.

3.5. Task 5: Falling Edge 33% Duty Cycle Clock

Implementation

The 33% Duty Cycle Clock was implemented in the same fashion as the other clock, except triggering on the falling edge (negedge).

Waveform

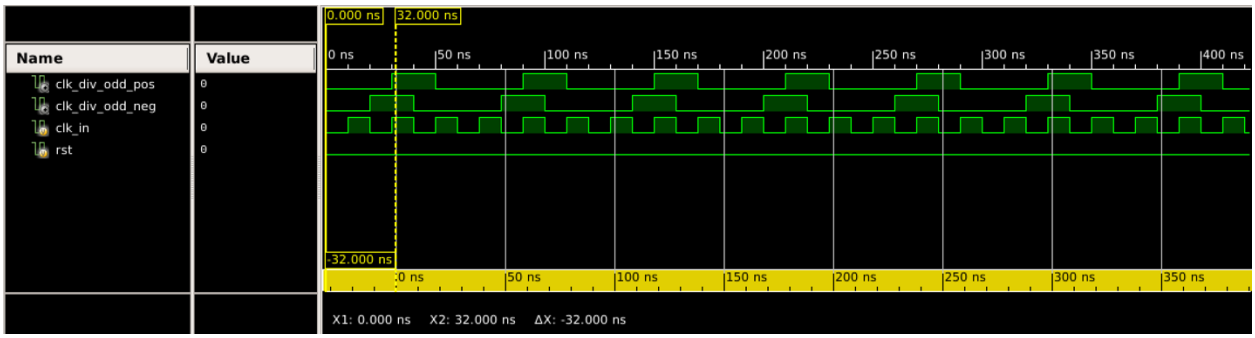

Figure 7: *Output Waveform of Falling Edge 33% Duty Cycle Clock*– Stimulated clock running at 50 MHz with 20 ns periods.

Triggering the duty cycle clock on the falling edge allows it to run half a cycle earlier than the clock that is triggered on the positive edge. In the context of this test, the negative edge duty clock is running 10 ns earlier than the positive edge clock.

3.6. Task 6: OR of Two 33% Duty Cycle Clocks

Implementation

The implementation assigns the OR the two 33% duty cycle clocks to the output clock.

Waveform

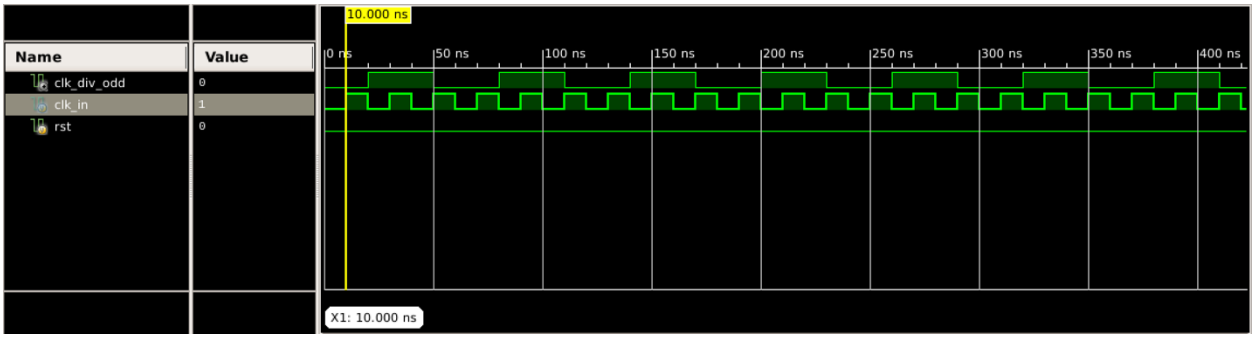

Figure 8: *Output Waveform of OR 33% Duty Cycle Clock*– Stimulated clock running at 50 MHz with 20 ns periods.

The output waveform of clk_div_odd is a 50% duty cycle divide-by-3 clock. Or the falling edge and rising edge clocks effectively allows for creating divide by odd clocks. We can see that one cycle of the clk_div_odd begins at 50 ns and ends at 110 ns. This corresponds to 3 cycles of the input clock, which means that clk_div_odd is 3 times slower than the input clock.

3.7. Design Task 7: 50% Duty Cycle Divide-by-5 Clock

3.7.1 Implementation

The 50% duty cycle divide-by-5 clock was implemented with a two 40% duty cycle clocks. This was done in a similar fashion to Task 6. However, instead of a 2-bit counter, a 3-bit counter was used for keeping track of the clock cycle. When the counter is 3, we toggle on the output clock (clk_div_odd). When the counter reaches 5, we reset the counter.

Waveform

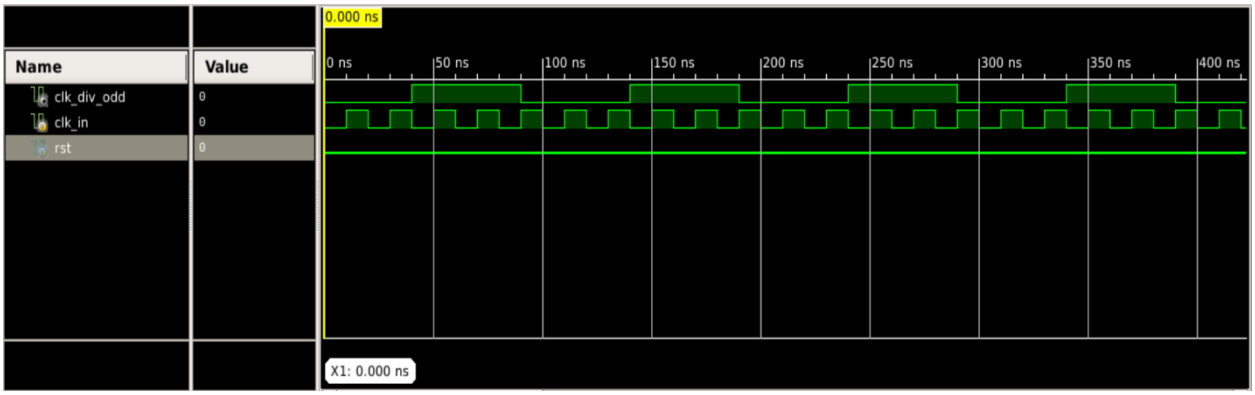

Figure 9: *Output Waveform 50% Duty Cycle Divide-by-5-Clock* - Stimulated clock running at 50 MHz with 20 ns periods.

In the graph, we can see that one cycle of the output clock begins at 90 ns and ends at 190 ns. This is equivalent to 5 cycles of the input clock. Additionally, the output clock is toggled on for half of the cycle from 140 ns to 190, which is equivalent to 50 ns.

3.8. Task 8: Divide-by-100 Strobe

Implementation

The divide-by-100 strobe was implemented with a divided by 100 clock running at 1% duty cycle. This was implemented with a 7-bit counter. When the counter reaches 99, the duty cycle clock is toggled on, and when the counter reaches 100, the duty cycle clock is toggled off. The output clock is contained in another always block and inverts when the pulse is active.

Waveform

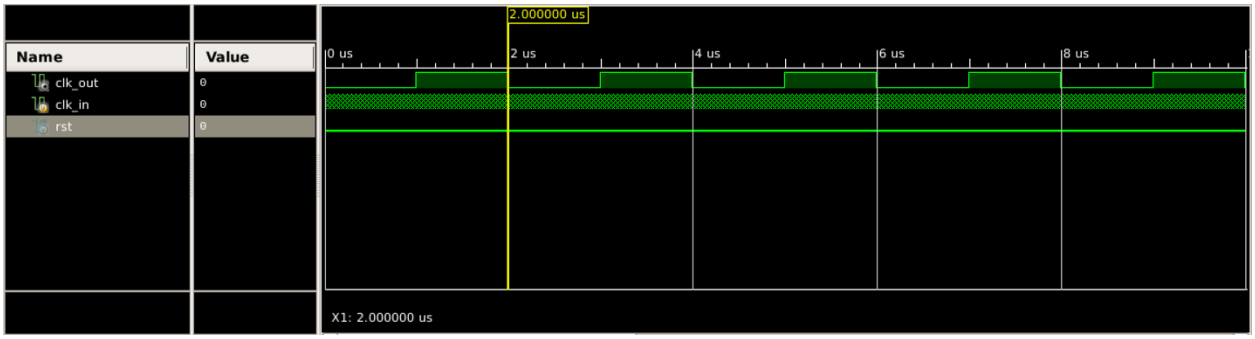

Figure 10: *Output Waveform Divide-by-100 Strobe Clock* - Stimulated clock running at 100 MHz with 10 ns periods.

We need to verify that the output clock is a divide-by-200 clock running at 500 KHz. From the graph, the period of one output clock is 2 us. This is equivalent to 2000 ns. The period of the output clock is 200 times slower than the input clock. Additionally, a 2 us period corresponds to a 500 KHz clock.

3.9. Design Task 9: Glitchy Counter

Implementation

The glitchy counter was implemented with a divide-by-4 strobe. The divide-by-4 strobe is a divide-by-4 clock running at 25% duty cycle. The clock was implemented with a 2-bit count with negative edge sensitivity. This offsets the pulse and the glitch counter so that, the counter does not detect the pulse too early. When the counter is at the falling edge of 3, we toggle on the strobe/pulse. In another always block, we increase the glitch counter by 2 on each positive edge and decrease it by 5 when the strobe is on.

Waveform

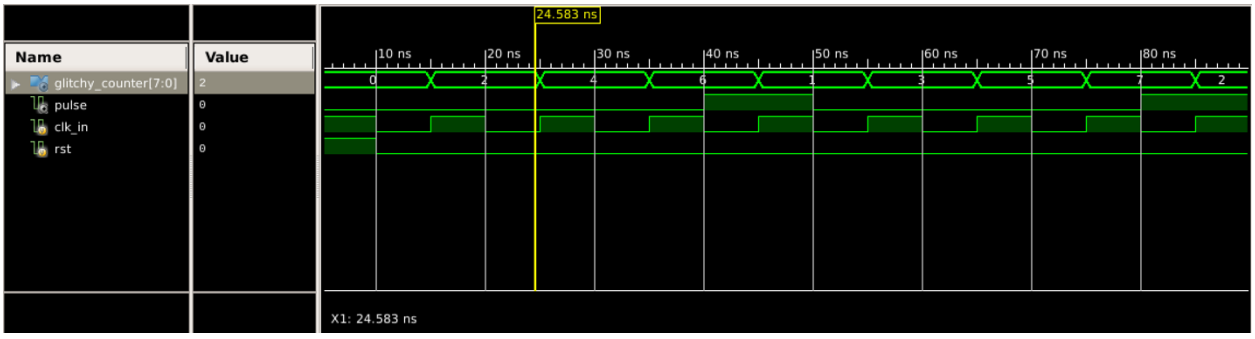

Figure 11: *Output Waveform Of Glitch Counter* - Stimulated clock running at 100 MHz with 10 ns periods.

For visualization sake, the divide-by-4 pulse has been included. We can see that at each positive edge, the counter is updated correctly. We can also see that that pulse is a divide-by-4 pulse since

4 periods of the input clock is contained in one period of the pulse signal. On the third negative edge of the input clock, the pulse is toggled. The offset ensures that the glitch counter does not detect pulse too early. If it were to be toggled on the third positive each, glitch counter would have been 0 → 2 → 4 → -1.

4. Design Summary

clock_gen Project Status (11/08/2020 - 23:09:23)			
Project File:	Project2.xise	**Parser Errors:**	No Errors
Module Name:	clock_gen	**Implementation State:**	Placed and Routed
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	7 Warnings (0 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Figure 12: *Design Summary*

When we synthesize and implement the source code, we encounter no errors in the design summary. We do, however, encounter 7 warnings which are all benign. 6 of the warnings are related to bit truncation that result from adding integers and overflowing. These edge cases have been accounted for in the code. The last warning is a mapping warning that results from one port being assigned in a different bus. This should not affect performance of the program.

5. Conclusion

This lab demonstrates that the foundation for manipulating clock speed is built upon counters. Understanding how counters can be translated into clocks is crucial since most hardware is synchronous and FPGAs often only come with a single clock speed. Additionally, even though we are working in binary, it is possible to divide clock speeds by odd powers with the use of duty cycles.