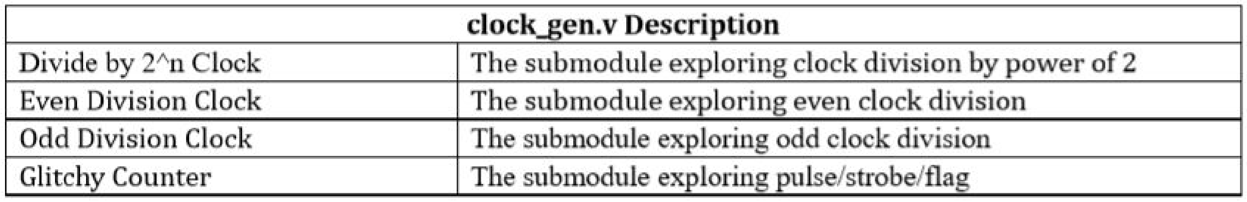
Adam Cole
UID[redacted]
Project 3 Lab Report

1. Introduction and Requirements

For project 3, our goal was to design and test various clock waveforms on a digital system. Our waveforms were to be displayed with one high level module, named clock_gen. The waveforms were encouraged to be created through the use of submodules, outlined below given through the spec:



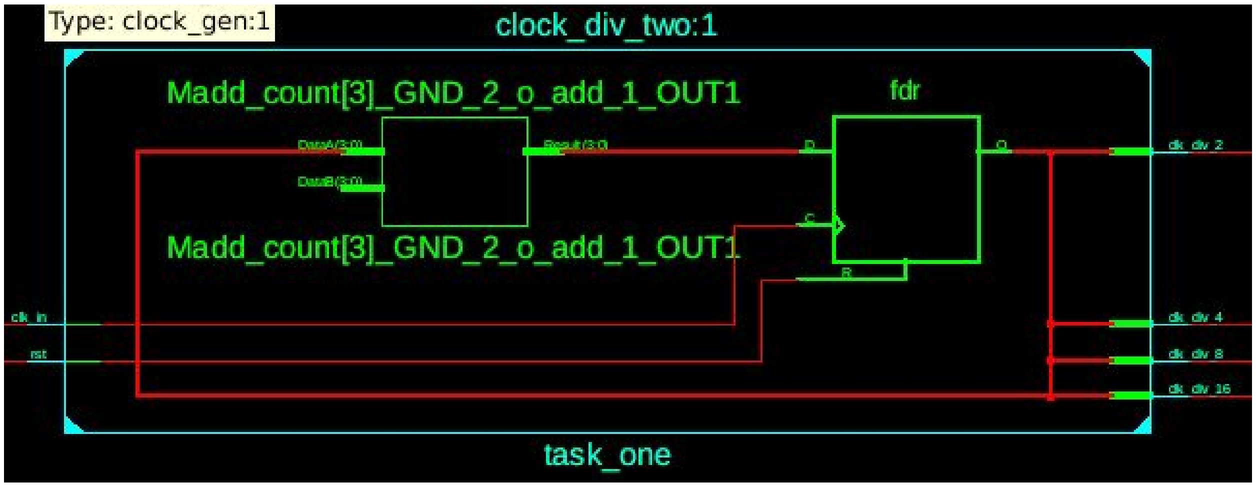
clock_gen.v Description	
Divide by 2^n Clock	The submodule exploring clock division by power of 2
Even Division Clock	The submodule exploring even clock division
Odd Division Clock	The submodule exploring odd clock division
Glitchy Counter	The submodule exploring pulse/strobe/flag

Clocks are very useful in digital systems since most sequential logic is synchronous and needs to be debugged one clock cycle at a time. Therefore, being able to implement a multitude of clocks, and derive these from the system clock, is very important. This lab seeks to give us that practice and expertise.

2. Design Description

2.1 Powers of 2 Clock Division

Our first clock divider task was to create waveforms that divide the system clock in powers of two (2, 4, 6, 8). In Project 1, we implemented a four-bit counter, and we were instructed to create our clock dividers using the same implementation. Therefore, we assigned each bit of the four-bit counter to an output wire associated with a certain power of two clock division. 2^1 was the 0th bit, 2^2 was the 1st bit, 2^3 was the 2nd bit, and finally 2^4 was the 3rd bit. Schematically, this is as easy as extending the counter wires to output, as shown by the generated RTL schematic below:



2.2 Even Clock Division

Our second clock divider task was to create clock division by any even number. In order to demonstrate this, the instructions specified that we first create a clock divider that divides an input clock by 32 - and base it off of our four-bit counter overflow. Toggling our clock each time our four bit counter overflowed caused our clock to flop every 16 clock cycles, or cycle every 32. Using this design, we were instructed to implement a clock divider that divides the system clock by 28, not 32. In order to do this, I simply changed the value at which the counter reset to 0. Now, the output clock bit toggles when the counter reaches 14 and resets, instead of 16.

2.3 Odd Clock Division

Our third clock divider task was to create clock division by any odd number. In order to demonstrate this, the instructions specified that we first create a divide by 3 clock divider using two 33% duty cycle clocks. A duty cycle is a clock that does not equally divide a period of a cycle into high and low. The formula for deriving how long a duty cycle should be high and low is:

$$\text{duty cycle} = \frac{t_{ON}}{t_{ON} + t_{OFF}}$$

Therefore, to create a 33% duty cycle, our output clock should be “on” 1 part to 2 parts “off”. This creates the ratio ⅓, or 33%. In order to do this, we used a counter to keep track of the system cycles, and made our output clock high if the counter was 1, and low otherwise. In this way, our output clock was high 1 part to 2 parts off. The second part of the instructions specified that we make a second 33% duty cycle, except counting on the negative edge of the system clock. Finally, to make our divide-by-3 clock, we logical or the two duty cycles together. In this way, we create a toggle every 1.5 system cycles, or cycling every 3 system cycles. In order to use this to create a divide-by-5 clock, we change our 3-counter to be a 5-counter, and make our output clock “high” when the counter equals 1 and 2, “low” when the counter equals anything else. In this way, each duty cycle will toggle every 2 system cycles, but a half-cycle offset from each other. By taking the logical or of these two cycles, we create a toggle every 2.5 system cycles, or cycling every 5 system cycles.

2.4 Pulsing and Strobes

Our fourth and final task was to create a glitchy-counter, which should strobe between different numbers (seemingly random). The instructions specified us to start by creating a divide-by-100 clock with a 1% duty cycle. I implemented this with the counter and clock methods we investigated in the previous parts - by using a counter to count to 100, and using if statements to toggle the output clock to “high” when the counter was 1, and “low” otherwise. This creates a divide-by-100 1% duty cycle. Our second step was to use this “pulse” as a notification to toggle a second clock. In this way, our new clock would cycle evenly every 200 system clock cycles. If our clock was cycling at 100MHz, this would be equivalent to a 500kHz divide-by-200 50% duty cycle.

Finally, we used this design to create our glitchy-counter. Our design was to start at 0, and increment the number by 2 each clock cycle. On a “pulse”, however, our design was to decrease the number by 5. A successful glitchy counter should output the following progression of numbers, given by the spec:

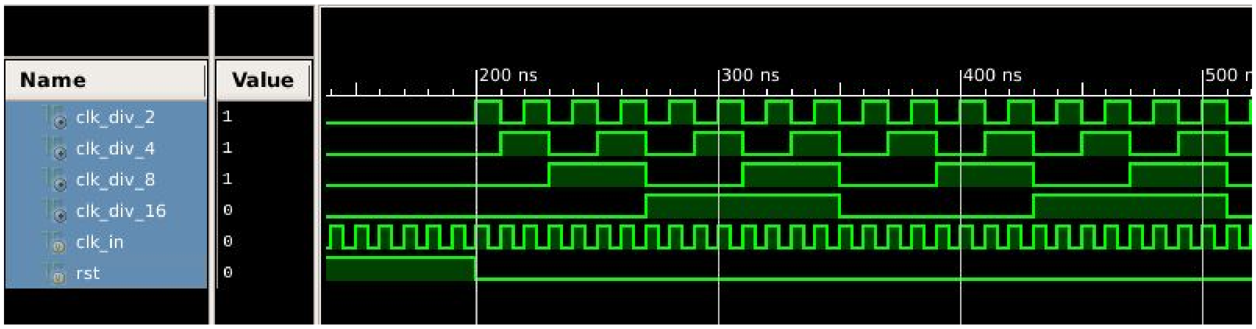
0→2→4→6→1→3→5→7→2→4→6→8→3→5→7→9→4→.....

Therefore, to implement our glitchy counter we used a divide-by-4 clock to indicate when our “pulses” should be. By doing this, our glitchy counter subtracts by 5 every fourth bit, and adds 2 on every other bit.

3. Simulation Documentation

3.1 Powers of 2 Clock Division

Running our module through Xilinx's iSim simulator, our module returns the following waveforms:

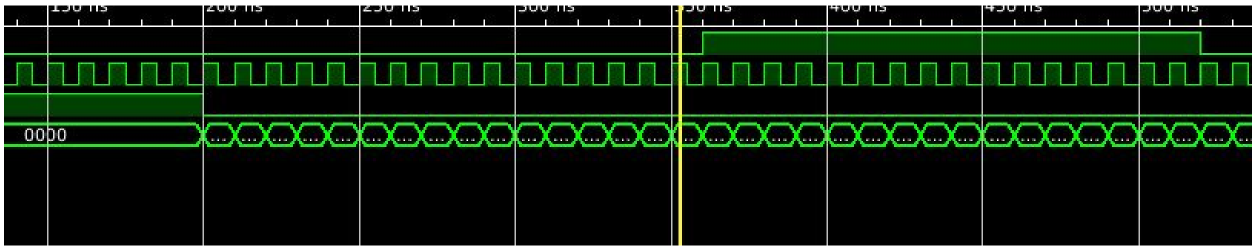


Waveform #1

Clearly, the bits in a binary system already toggle at the correct clock division, so we learn that when dividing clocks by powers of two, only the correctly associated bit is needed to divide the clock.

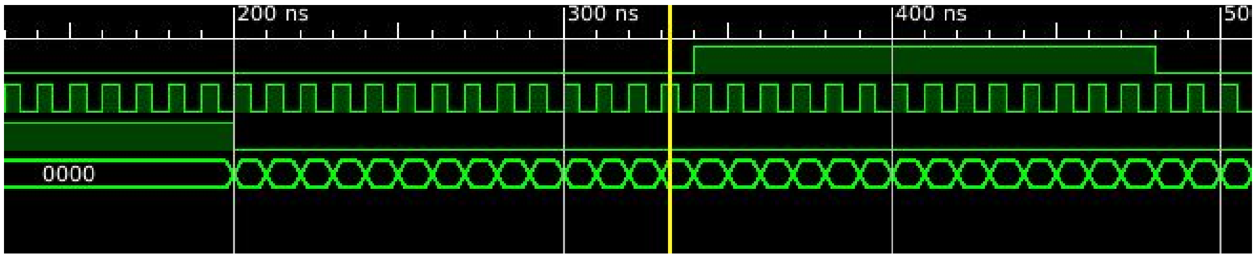
3.2 Even Clock Division

First I created and simulated the clock-divider that divided the system clock by 32. The associated waveform is shown below:



Waveform #2

After the clock resets, the output clock stays low for 16 clock cycles, and then toggles to high for 16 more clock cycles. Therefore, our method of implementing a 32-clock-divider was successful. Changing our counter to reset at 14 instead of 16, our output clock toggled every 14 cycles instead of every 16. Simulating my new design, the associated waveform is shown below:

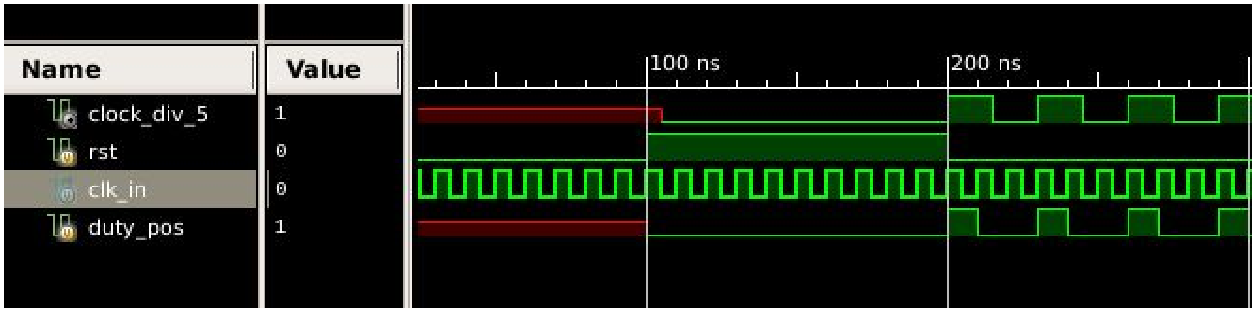


Waveform #3

After the clock resets, the output clock stays low for 14 clock cycles, and then toggles to high for 14 more clock cycles. Therefore, our method of implementing a 28-clock-divider was successful.

3.3 Odd Clock Division

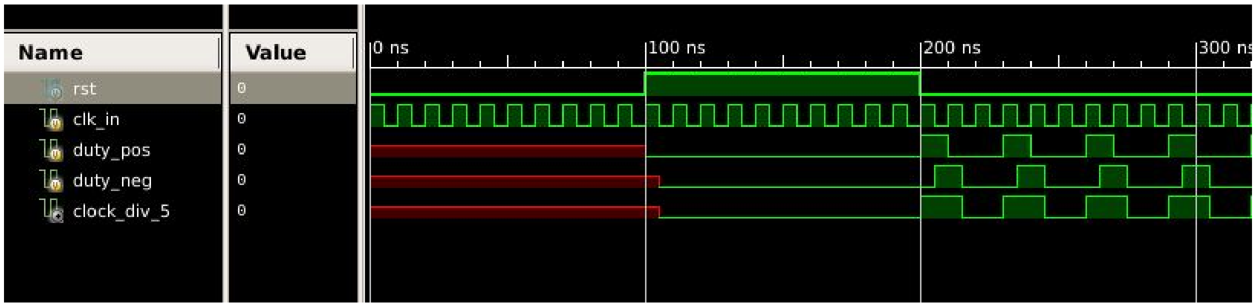
Simulating our module, I first test the first duty cycle of my divide-by-3 clock cycle. The output waveform is shown below:



Name	Value
clock_div_5	1
rst	0
clk_in	0
duty_pos	1

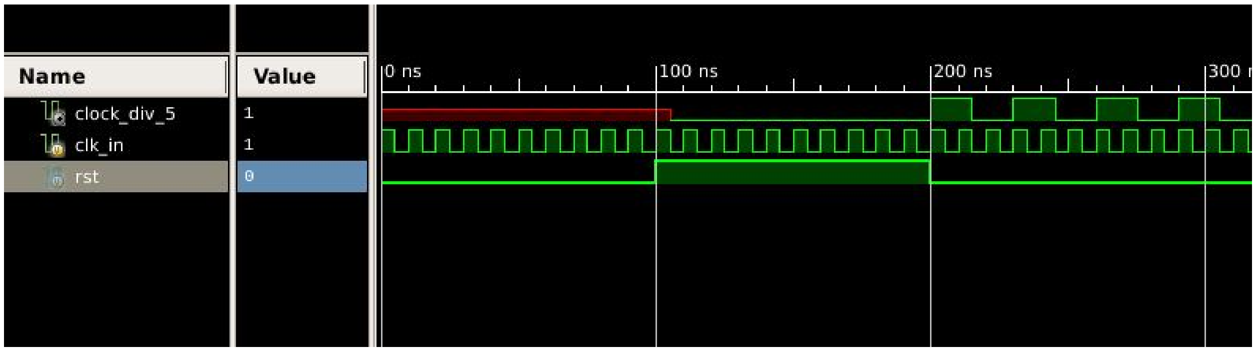
Waveform #4

After the clock resets, our duty cycle duty_pos is “high” for only 1 out of every 3 system cycles. In this way, our duty cycle is a 33% duty cycle of our system clock. Next we add the 33% negative duty cycle to the waveform. This is shown below:



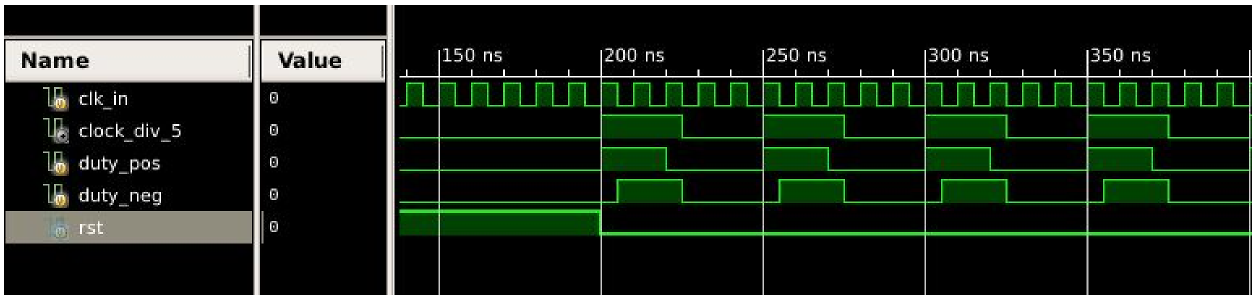
Waveform #5

After the clock resets, our duty cycle duty_neg is “high” for only 1 out of every 3 system cycles, toggling on the negative edge of our system clock. In this way, our duty cycle is a 33% duty cycle of our system clock, a half cycle offset from our first duty cycle clock. Next we logical or the two duty cycles together, creating our divide-by-3 clock This is shown below:



Waveform #6

From this we can see our successful clock division. After the clock resets, our output clock cycles in exactly 3 system clock cycles, successfully dividing the input clock by three. Now that we verified our design, we modify our module accordingly to create a divide-by-5 clock. The output waveforms are shown below:

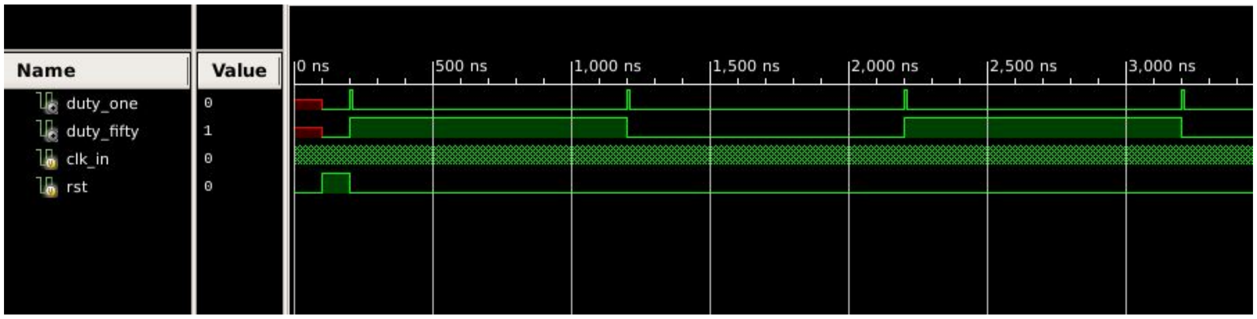


Waveform #7

Similar to the divide-by-3 clock in Waveform #6 above, each of the duty cycles take 2 system cycles to toggle, a half-cycle offset from each other. By using the logical or on both of these clock cycles, our divide-by-5 clock toggles every 2.5 system clock cycles, or cycles every 5 system clock cycles. Therefore, successfully implementing our design.

3.4 Pulsing and Strobes

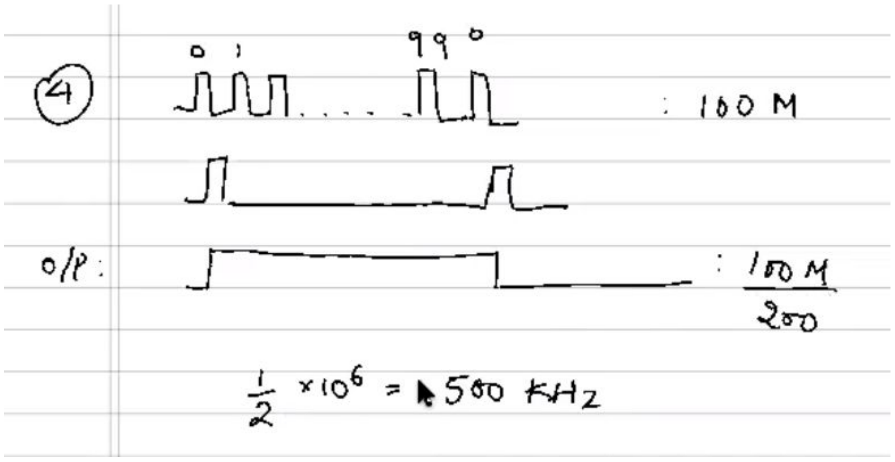
To begin the final task, I create the divide-by-100 1% duty clock and simulate the module. Then, I add the second waveform which creates the 50% duty cycle divide-by-200 clock. The resulting waveforms are shown below:



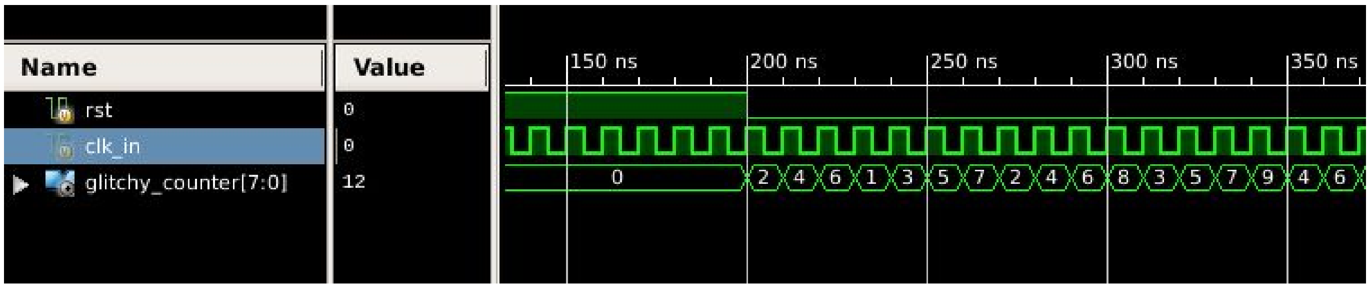
Name	Value
duty_one	0
duty_fifty	1
clk_in	0
rst	0

Waveform #8

As we can see, the top waveform is the pulse created every 100 system clock cycles, and only lasting 1 clock cycle. Furthermore, this pulse clearly lines up with the toggling of the divide-by-200 counter. Since the system clock used was 100MHz, it takes the second waveform 200 system clock cycles for it to cycle once. Therefore, it is a divide-by-200 clock, and the ratio of "high" to "low" is equivalent - making it a 50% duty cycle. A further mathematical explanation is shown below, given in lab:



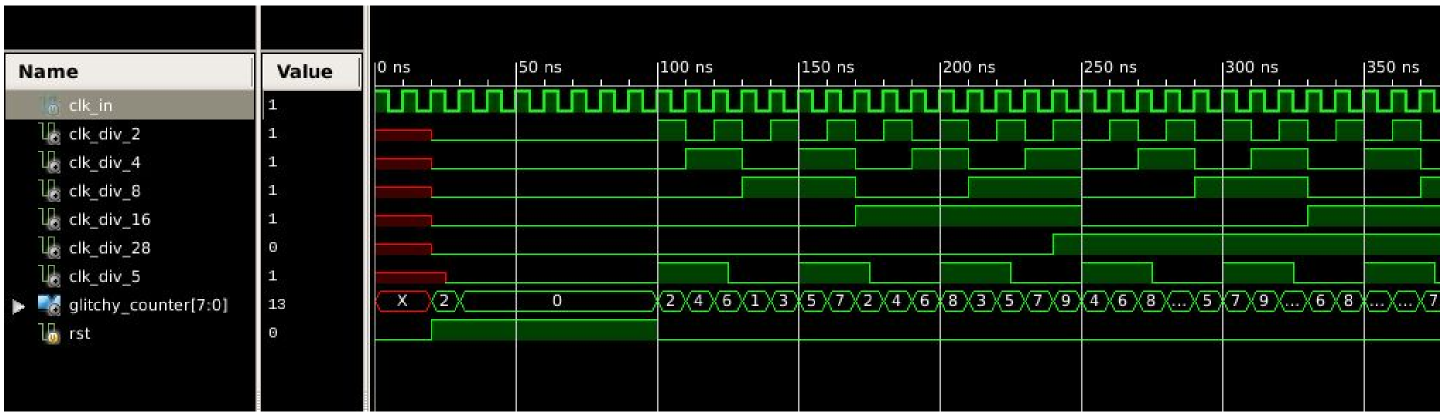
After verifying our design using the last waveform, we implement our glitchy counter using a divide-by-4 25% duty cycle. On every system clock cycle when the wave is "low", we add 2 to the value in our glitchy counter. On every system clock cycle when the wave is "high", we subtract 5 from the value in our glitchy counter. After running our module through simulation, our values match the sequence of values defined in the spec. The associated waveform is shown below:



Waveform #9

4. Conclusion and Synthesis

After creating all of the submodules outlined in this report, we combine them into one high-level module named clock_gen. We run the overall module through simulation and present the waveform below, matching the given one in the Project specification.



Waveform #10

When synthesizing my verilog files, the synthesis report passed without errors - but the report did contain some warnings that detailed inconsistencies with some sensitivity lists. While the simulation proved successful, without a FPGA Board to implement my design on, my solutions cannot be truly verified as accurate.