CS M152A Project 3 Report

Melody Chen

May 11, 2020

1 Introduction and Requirement

The focus of this lab is for students to learn to use the Xilinx ISE software to design and test various waveforms on a digital system. From the manual, we learned about basic concepts behind clocking a system and techniques to generate them from a system clock. Clocks are very important for synchronous data transmission, allowing signals to be communicated between different devices. In this lab, we focus on creating clocks and pulses of different frequency and duty cycles, such as even division and odd division clocks. For this project, I designed and built many smaller sub-modules of individual types of clock, and combined them into one single `clock_gen` top module to allow side-by-side comparisons of their waveforms and further analysis. All of my modules and sub-modules take system clock as an input in addition to a reset signal and outputs the derived clock. A brief summary of my top module and sub-modules are outlined below(from the project manual):

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

Outline of Top Module and Sub-modules

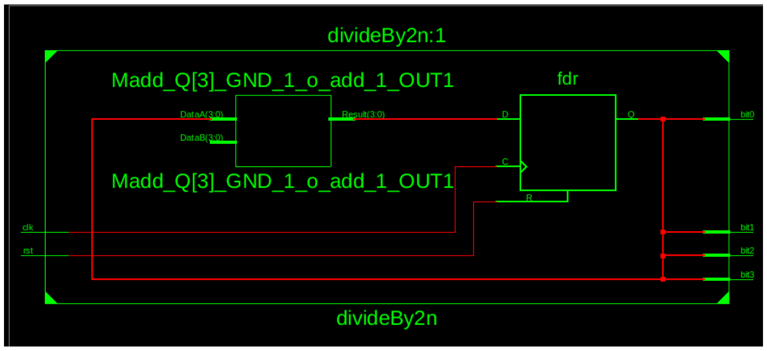
2 Design Description

For the design and implementation of the varying types of clocks, I followed guidance provided in the project specifications. I designed and implemented a total of 7 sub-modules and 1 top modules that combines the sub-modules. Below, I will talk about my design and implementation of the modules based on the category of clocks it belongs to. In this project, we implement and work with the following categories of clocks: Clock Divider by Power of 2s, Even Division Clock Using Counters, Odd Division Clock Using Counters, Pulse/Strobes. All of my sub-modules take two inputs, the system clock `clk_in` and reset signal `rst`. Output depends on each sub-module, but generally one or more derived clock signal will be outputted.

a. **Category 1:** Clock Divider by Power of 2s
In this category of clock, I designed one sub-module `clock_div_two` that outputs four clock signals: `clk_div_2`, `clk_div_4`, `clk_div_8`, and `clk_div_16`. I based my design of this sub-module on the 4-bit Counter I built for Project 1. The idea is to have a 4-bit register `Q[3:0]` that increments by 1 on each `posedge` of the clock. I then observe that the clock divider can be obtained by extracting the proper bit from the counter. Now, each bit represents a clock divider of a certain power of 2. The LSB represents the `clk_div_2` signal, as its signal is 1 for every 2 `posedge` of the clock, thus it is twice as slow as the `clk_in` signal. By the same logic, the 1st bit then represents the `clk_div_4` signal as the first bit changes signal from 0 to 1 every four clock cycles, so it is four times as slow as `clk_in` signal. The same logic is applied to obtain the `clk_div_8` and `clk_div_16` signals.

To implement this in Verilog, I replicate my 4-bit counter made in Project 1 which has an `always` block sensitive to the `posedge` of the clock signal. Inside, I increment register `Q[3:0]` by 1 unless rst is high. To extract each bit as a separate signal, I used four `assign` statements to assign each bit of `Q` to the corresponding output signal.

The RTL schematic generated by the Xilinx ISE of my `clock_div_two` module is shown below. My Verilog code results in this schematic being generated as I used an adder to increment register `Q` and extracted the 4 bits as shown by the four wires to the right of the schematic.



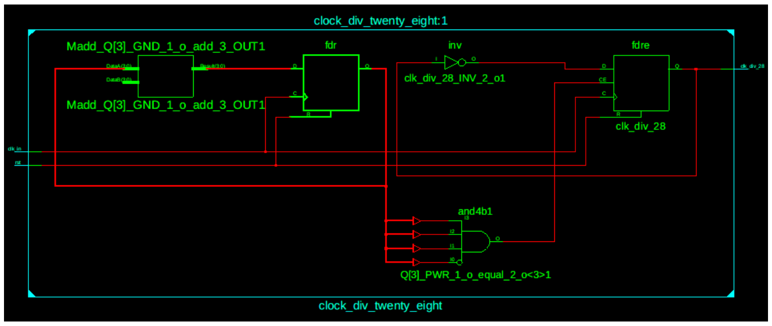
RTl Schematic of `clock_div_two` Module

b. **Category 2:** Even Division Using Counters

In this category of clock, I designed two sub-modules that corresponds to two even division clock dividers: a divide-by-32 clock `clock_div_thirty_two` module and a divide-by-28 clock `clock_div_twenty_eight` module. For the `clock_div_thirty_two` sub-module, I based my design again on the 4-bit counter, but instead of extracting bits, I check for overflow when the 4-bit register is incremented by 1. When overflow is detected which will be on the 16th `posedge`, our output clock signal will flip allowing us to have a output clock signal that is 32 times slower than the system clock. My design for the divide-by-28 clock `clock_div_twenty_eight` module is then based off of this idea of detecting overflow. But since we want signal to flip on the 14th `posedge`, instead of detecting when the 4-bit register overflow as now we want a slightly faster clock, I detect when 4-bit register is 1 step before overflow, i.e. equal to `4'b1101`. This slight modification allows me to design a clock that is 28 times slower than the system clock.

The implementation of these two sub-modules in Verilog is very similar. The `clock_div_thirty_two` module outputs signal `clk_div_32` and `clock_div_twenty_eight` outputs signal `clk_div_28`. I have an `always` block sensitive to the `posedge` of the provided clock, and then I use if statements to detect when `rst` is high and when our 4-bit register `Q[3:0]` is equal to `4'b1111` for divide-by-32 or `4'b1101` for divide-by-28 clock. When `rst` is high, we reset `Q[3:0]` and our output clock signal to 0. When `Q` is equal to specified number `4'b1111` or `4'b1101`, we flip the output clock signal.

The RTL schematic generated by the Xilinx ISE of my `clock_div_twenty_eight` module is shown below. I chose to only show RTL schematic of one module in this category as the two implementations are very similar. My Verilog code results in this schematic being generated as I used an adder to increment register `Q`, a comparator to compare value of `Q` to specified number, and another register to store our output signal.



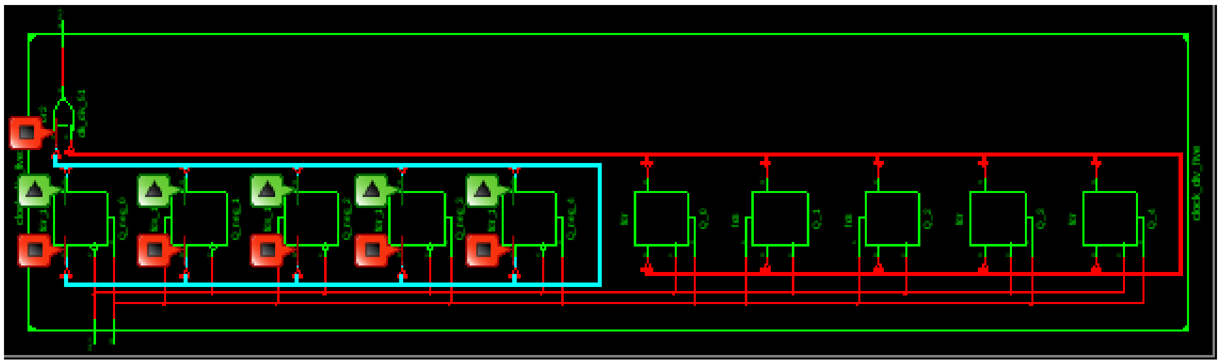
RTl Schematic of `clock_div_twenty_eight` Module

c. **Category 3:** Odd Division Clock Using Counters

In this category of clock, I designed two sub-modules that corresponds to the two tasks assigned: a 33% duty cycle clock `clock_div_thirty_three` module, and a divide-by-5 clock `clock_div_five` module. 33% duty cycle clock means that for one third of a period it is 1 and 0 for the rest. To accomplish this, instead of using concept of 4-bit counter, I rotate a the 3-bit value `3'b100` to the left at each `posedge` and set the MSB as the output `clk_div_33` signal. This allows us to have a output clock signal that is 1 one-third of the time, as only 1 bit in `3'b100` is 1. To implement `clock_div_thirty_three` module in Verilog, I used an `always` block sensitive to `posedge` of the system clock that uses concatenation to rotate our register `Q[2:0]` with initial value `3'b100` to the left. When `rst` is high, we reset `Q[2:0]` to `3'b100`. The MSB `Q[2]` is then assigned to output `clk_div_33` signal.

Following the task instructions, I duplicated another `always` block in the same module that is sensitive to `negedge` of the system clock. I created an extra output `clk_div_33_neg` that holds the extra output. I assigned the logical `or` of `clk_div_33_neg` and `clk_div_33` signal to `clk_div_33_or` and observed that it is now a 50% duty cycle divide-by-3 clock. With this idea, I can now design my divide-by-5 clock `clock_div_five` module with the same idea. Instead of `3'b100`, I rotate the value `5'b11000` because once I `or` the MSB from the output of `posedge` and `negedge`, I will have a 50% duty cycle clock and it is a divide-by-5 clock because our rotated value is 5 bits which means our output clock signal will begin repeating after 5 clock cycles .

The RTL schematic generated by the Xilinx ISE of my `clock_div_five` module is shown below. I chose to only show RTL schematic of one module in this category as the two implementations are very similar. My Verilog code results in this schematic being generated as I used two 5-bit registers `Q[4:0]` and `Q_neg[4:0]`, and an or gate to get my final clock output.



RTl Schematic of `clock_div_five` Module

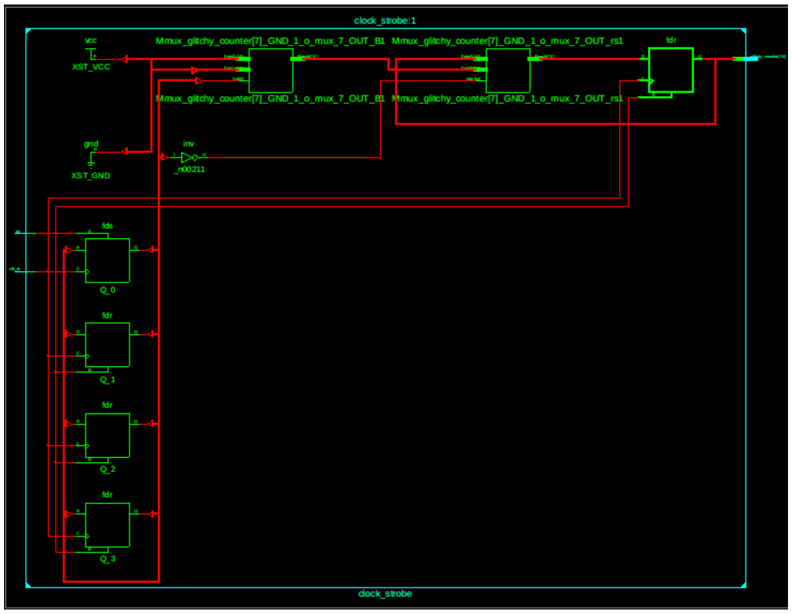
d. **Category 4:** Pulses/Strobes

For this clock category, I designed two sub-modules that corresponds to the three tasks assigned: a divide-by-200 clock using a divide-by-100 clock with 1% duty cycle and `clock_strobe` module that outputs a glitchy counter. To build the divide-by-100 clock with 1% duty cycle, I used the same idea from building a 33% duty cycle, but instead of rotating `3'b100`, I rotate {`1'b1`,{`99`{`1'b0`}}} stored in register `Q[99:0]` as we want clock to be 1 only 1% of the time. The MSB is our output clock. To implement this in Verilog, I have one `always` block sensitive to `posedge` of system clock and it rotates `Q[99:0]` to the left each time. I assign `Q[99]` as output clock signal `clk_div_100`.

Now, I want to use the divide-by-100 clock with 1% duty cycle to create an output clock that is 50% duty cycle divide-by-200 clock running at 500kHz. I can build this by adding an additional `always` block to my divide-by-100 clock module that flips the output clock `clk_div_200` every time `clk_div_100` signal is 1. With this our new output signal will flip every 100 system clock cycles, creating a 50% duty cycle divide-by-200 clock that is 200x slower than system clock. Since system clock inputted is 100Mhz, our output clock will run 200x slower, resulting in $100MHz/200 = 500kHz$ output clock signal.

Finally, we want to design our `clock_strobe` module that generates an 8-but counter that counts up by 2 on every `posedge` of master clock and subtracts by 5 on every strobe. To detect when the strobes occur, I rotate the 4-bit value `4'b0001` stored in register `Q[3:0]` to the right on every `posedge` in the first `always` block. In another `always` block, I detect whether `Q[3]` is 1. If it is, I subtract 5 from 8-bit counter, if not, I add 2 to 8-bit counter.

The RTL schematic generated by the Xilinx ISE of my `clock_strobe` module is shown below. I chose to only show RTL schematic of one module in this category as the `clock_div_hundred` module is very similar to our previous modules. My Verilog code results in this schematic being generated as I used one 4-bit register `Q[3:0]` and the muxes represents the if statements in my `always` block to determine when to add 2 or subtract 5.


clock_strobe:1
vcc
XST_VCC
gnd
XST_GND
Mmux_glitchy_counter[7]_GND_1_o_mux_7_OUT_B1
Mmux_glitchy_counter[7]_GND_1_o_mux_7_OUT_rs1
Mmux_glitchy_counter[7]_GND_1_o_mux_7_OUT_B1
Mmux_glitchy_counter[7]_GND_1_o_mux_7_OUT_rs1
fdr
inv
_n00211
fds
Q_0
fdr
Q_1
fdr
Q_2
fdr
Q_3
clock_strobe

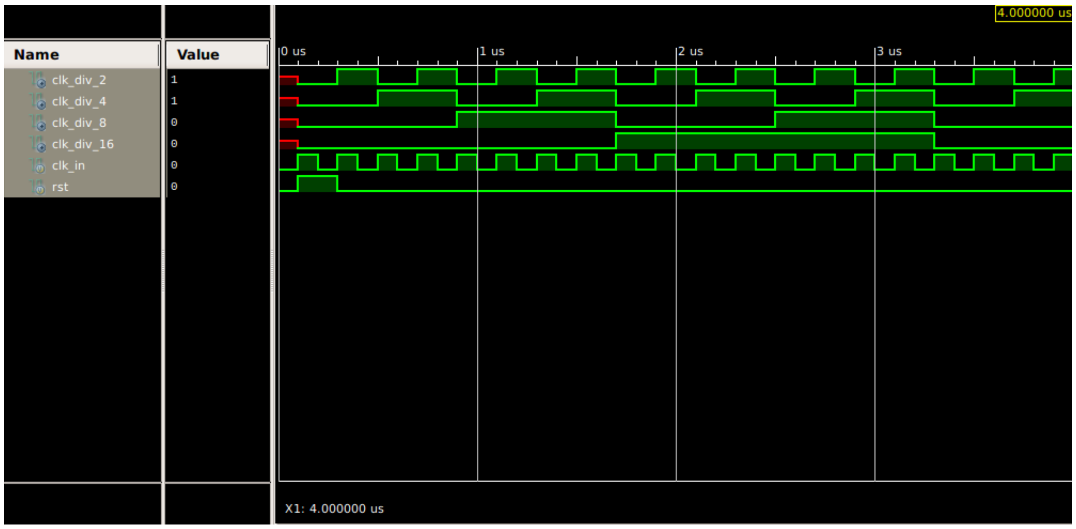
RTl Schematic of `clock_strobe` Module

e. **Top Module:** Clock Generator Module
This module is the top module and combines the 4 deliverable tasks mentioned in the project manual. This top module takes the following inputs: input clock signal `clk_in` and reset signal `rst`. It outputs the following clock signals: `clk_div_2`, `clk_div_4`, `clk_div_8`, `clk_div_16`, `clk_div_28`, `clk_div_5`, `[7:0] glitchy_counter`. In this top module, I instantiated modules we previously built that provided us with the above outputs. The schematic of this module is not shown, as it is simply a combination of modules we previously described above.

3 Simulation Documentation

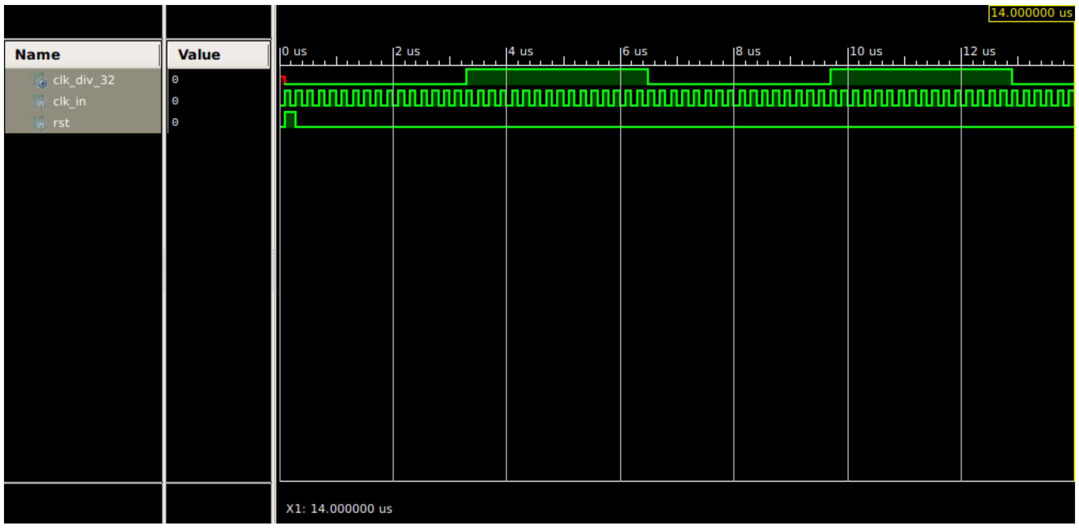
To test whether the different types of clocks we designed for this project work as expected, I wrote separate test benches to produce waveforms for each module in addition to the test bench for the final clock generator module which combines many of the sub-modules we wrote. The test bench for all of my modules and final module looks very similar as all of the modules take two inputs: `clk_in` and `rst`. In all my test benches, `rst` is only high in the very start to make sure all values are initialized correctly. Then, my `clk_in` is flipped every certain amount of time depending on the module and the instructions.

a. **(1)** Clock Divider by Power of 2s
For the `clock_div_two` module, in my test bench I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We can clearly observe that the `clk_div_2` clock signal is twice as slow as the `clk_in`, because for two `clk_in` cycles, we only observe one `clk_div_2` cycle. The same can be observed for the `clk_div_4`, `clk_div_8`, `clk_div_16` clock signals. For example, one `clk_div_16` cycle takes 16 `clk_in`, showing that `clk_div_16` clock is 16 times slower than the provided clock signal. Thus, we observe that using counters, we can easily generate clocks that have periods that are even multiple of the original.



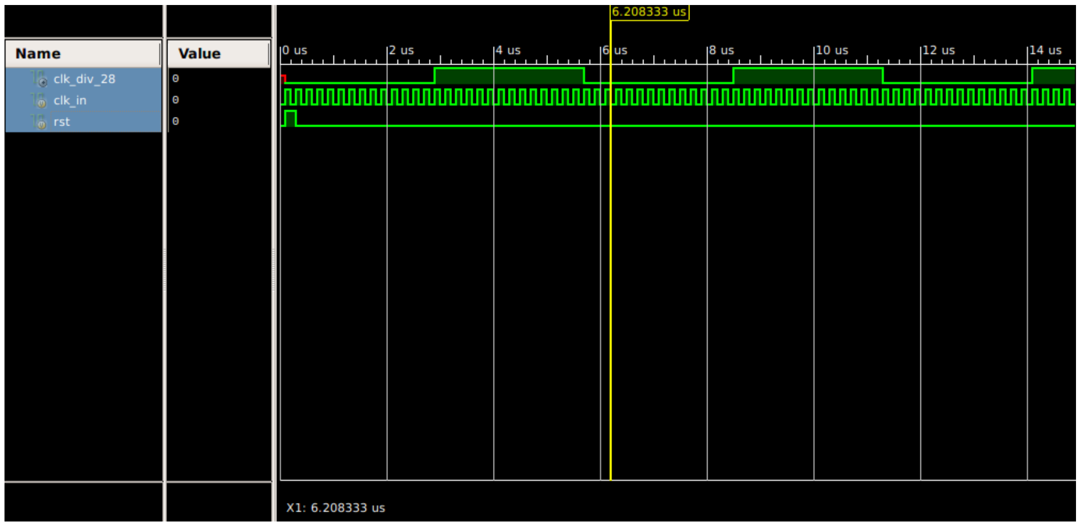
(1) Simulation Waveform for `clock_div_two`

b. **(2)** Divide-By-32 Clock
For the `clock_div_thirty_two` module, in my test bench I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We can clearly observe that the `clk_div_32` signal is 32 times slower than the input `clk_in` signal, so our module works as expected.



(2) Simulation Waveform for `clock_div_thirty_two`

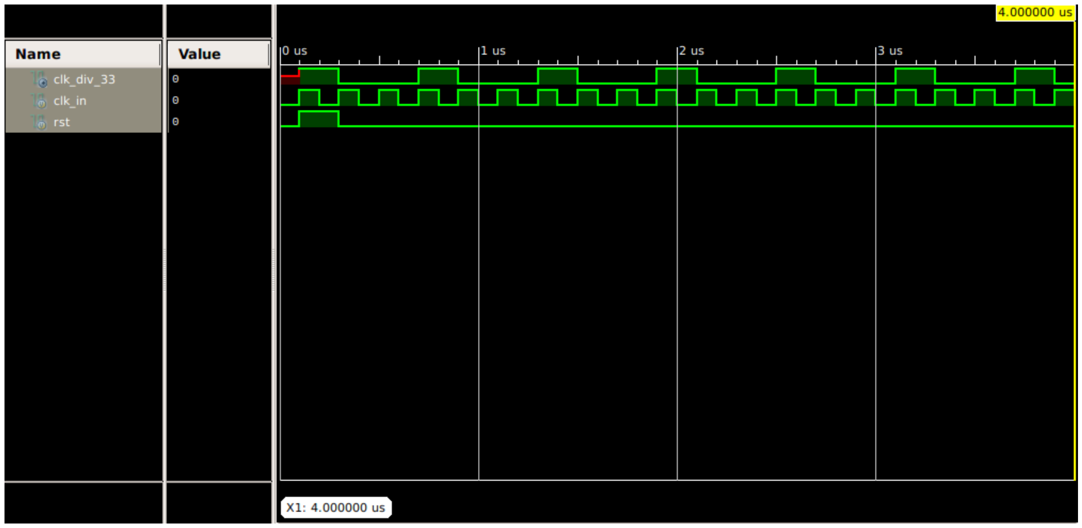
c. **(3)** Divide-By-28 Clock
For the `clock_div_twenty_eight` module, in my test bench I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We can clearly observe that the `clk_div_28` signal is 28 times slower than the input `clk_in` signal, so our module works as expected. One cycle of our `clk_div_28` clock signal takes 28 input clock cycles.



(3) Simulation Waveform for `clock_div_twenty_eight`

d. **(4)** 33% Duty Cycle Clock

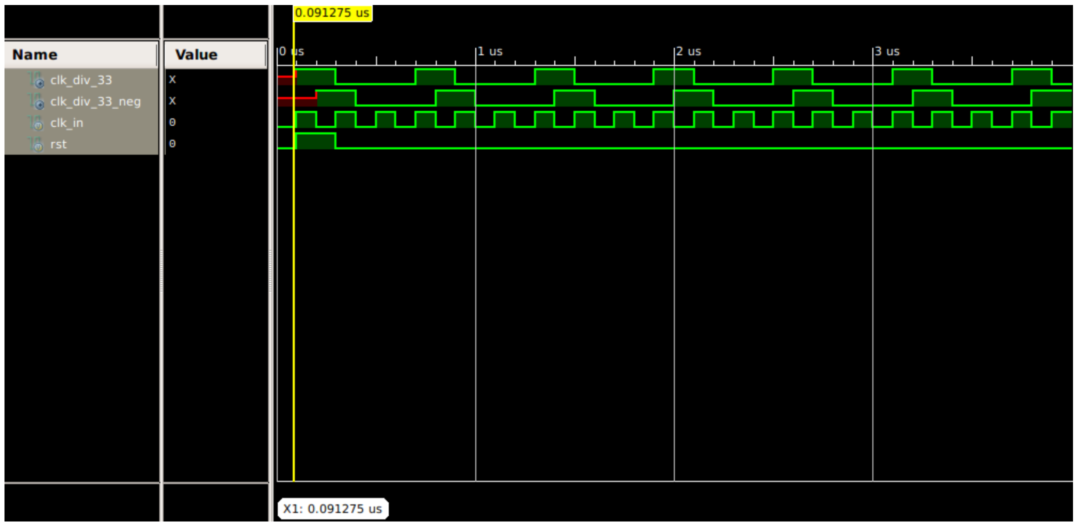
For the `clock_div_thirty_three` module, in my test bench I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We can clearly observe that 1/3 of the time our output `clk_div_33` clock signal is 1 and the rest 2/3 of the time our output `clk_div_33` clock signal is 0. To illustrate, we can observe for one cycle of `clk_in`, `clk_div_33 = 1` and following that cycle for two cycles of `clk_in`, `clk_div_33 = 0`.



(4) Simulation Waveform for `clock_div_thirty_three`

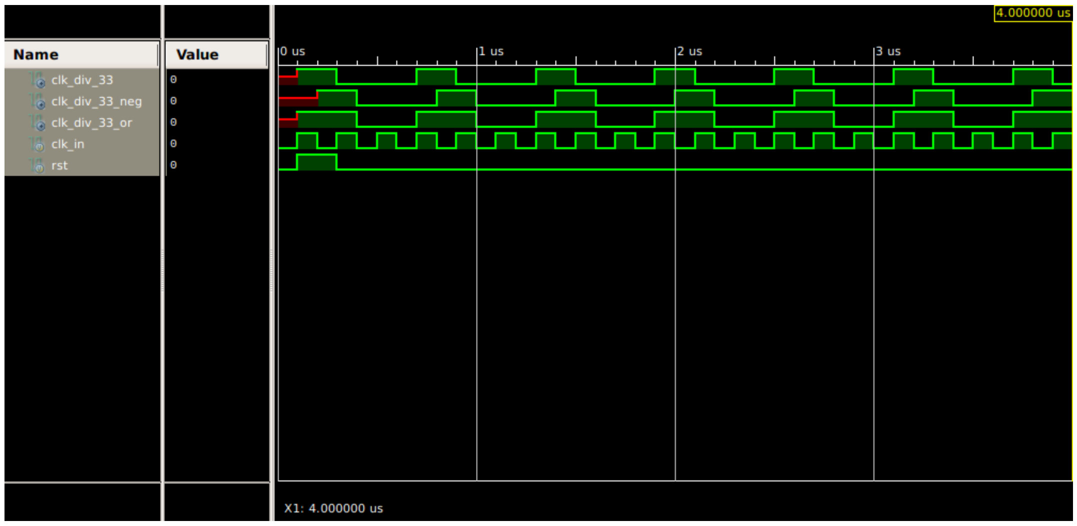
e. **(5)** Side by Side 33% Duty Cycle Clock

For this task, we build on the `clock_div_thirty_three` module, by adding an `always` block sensitive to `negedge` of the clock. Side by side waveform of the two output clock signals is shown below. We can observe that the `clk_div_33_neg` signal is half a cycle behind the `clk_div_33` as it is sensitive to the negative edge of the clock.



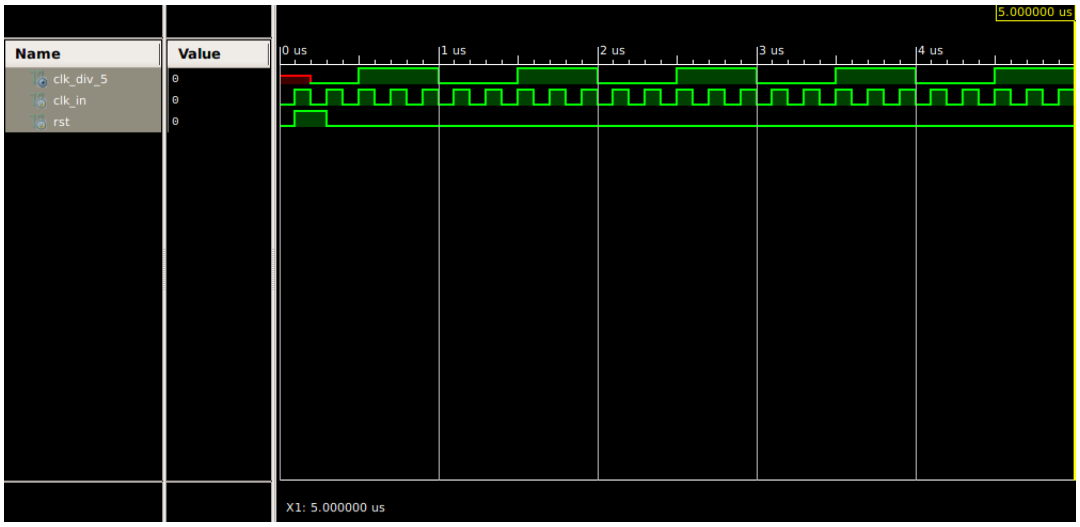
(5) Simulation Waveform for `clock_div_thirty_three` Side by Side

f. **(6)** Logical Or of two 33% Duty Cycle Clock
For this task, we build on the `clock_div_thirty_three` module with addition of an `always` block sensitive to `negedge` of the clock and logical or the two outputs `clk_div_33` and `clk_div_33_neg` to get `clk_div_33_or` signal. The resulting waveform is shown below. We can observe that the `clk_div_33_or` signal is actually a 50% duty cycle divide-by-3 clock.



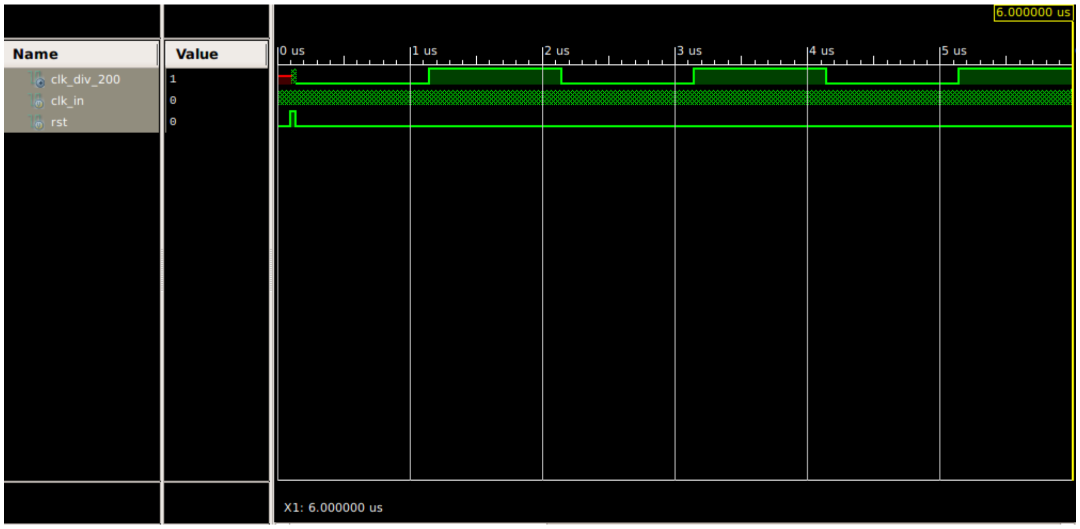
(6) Simulation Waveform for `clock_div_thirty_three` Logical Or

g. **(7)** 50% Duty Cycle Divide-By-5 Clock
For this task, we built the `clock_div_five` module. In my test bench for this module, I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We observe that for five cycles of the input clock, we have one cycle for our output `clk_div_5` clock signal, showing that our module works as expected and produces a clock that is five times slower than the input clock.



(7) Simulation Waveform for `clock_div_five`

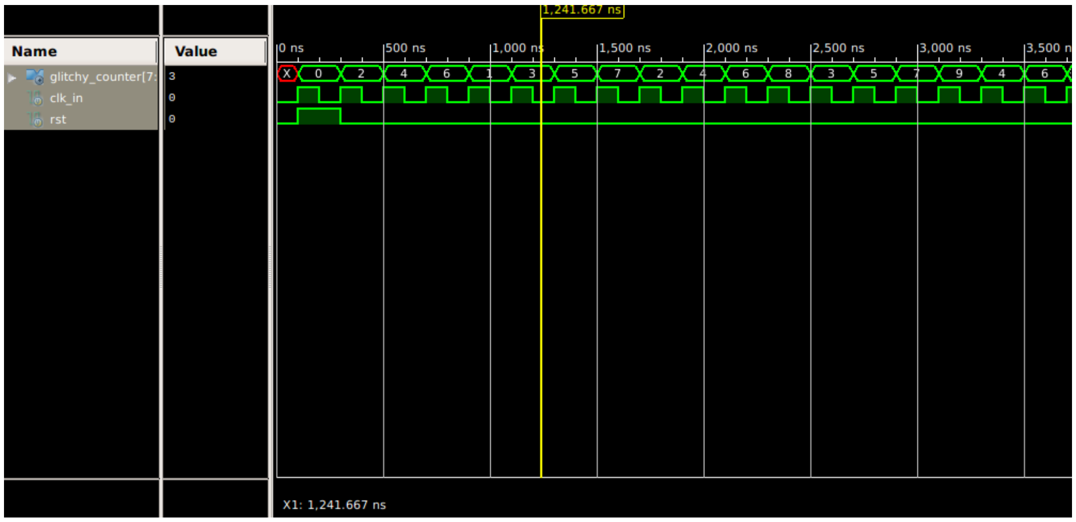
h. **(8)** 50% Duty Cycle Divide-By-200 Clock at 500kHz
For this task, we built the `clock_div_two_hundred` module. In my test bench for this module, I initially set `rst` to high and then flipped the `clk_in` signal every 5ns to create a 100Mhz system clock. The resulting waveform is shown below. We observe that our output clock signal `clk_div_200` has period of 2us. Since it has a period of 2us, this means that the frequency of our `clk_div_200` clock is $1/0.000002s = 500kHz$, which matches the description in the project manual. This also makes sense as our output clock signal should be 200x slower then the input system clock, $100mHz/200 = 500kHz$. It is also clearly a 50% duty cycle clock as half of the period, our output clock signal is 1.



(8) Simulation Waveform for `clock_div_two_hundred`

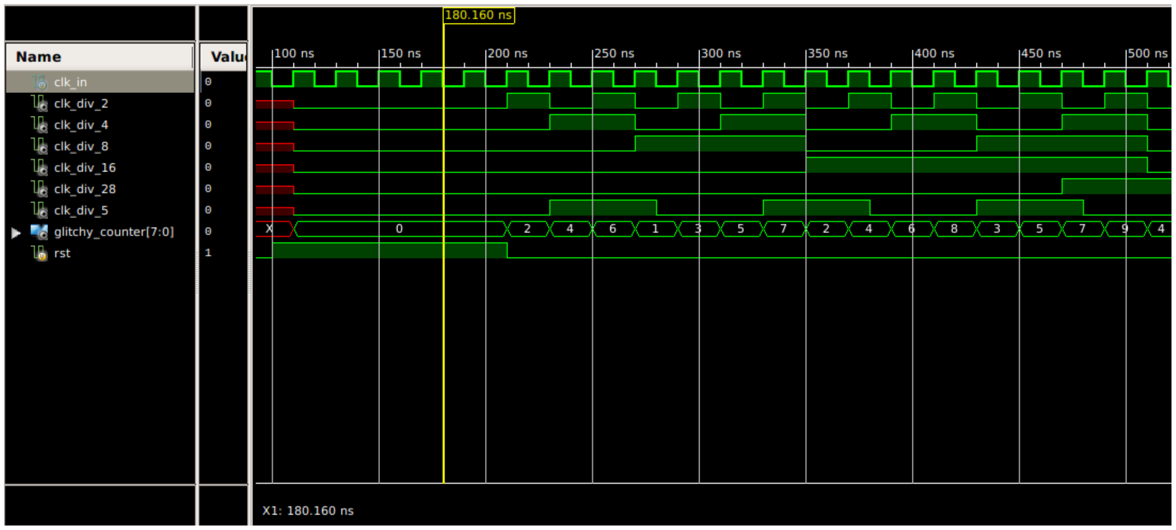
One bug I found during simulation for this module is that initially the period of `clk_div_200` clock is 8us instead of the expected 2us. Because of this, I went back to investigate my Verilog code and realize that I had calculated my input system clock's value incorrectly and was able to fix the system clock's frequency.

i. **(9)** Glitchy Counter
For this task, we built the `clock_strobe` module. In my test bench for this module, I initially set `rst` to high and then flipped the `clk_in` signal every 100ns. The resulting waveform is shown below. We observe that our glitchy counter's output correctly counts up by 2 on every positive edge of the master clock, and subtracts by 5 on every strobe.



(9) Simulation Waveform for `clock_strobe`

j. **(10)** Clock Generator Module
   The Clock Generator Module `clock_gen` combines our sub-modules together, so we can observe different types of clocks all in one wave form. In my test bench for this module, I initially set `rst` to high and then flipped the `clk_in` signal every 20ns. The resulting waveform is shown below.



(10) Simulation Waveform for `clock_gen`

4 Conclusion

In this project, I designed and implemented clocks and pulses of different frequency and duty cycles. They were designed based on ideas formulated in Project 1 and hints provided in the project manual. The even division clocks were relatively easier to implement than odd division clocks, as even division clock's implementation simply involves using counters of various bits. The main idea I used for odd division clock was rotating a binary value of certain length with varying number of 1's and 0's. For the glitchy counter module, I combined ideas from both even and odd division clock.

Some difficulties I encountered include coming up with the initial idea of how to build an odd division clock. But, with the help of the TA's lab section, I was able to come up with the correct idea, experiment with Verilog code, and eventually come up with a working implementation. Another difficulty I encountered was interpreting the RTL schematic generated by Xilinx ISE. I dealt with this by investigating closely the purpose of each component and looking through the Xilinx documentation.