Name: Chris Baker

UID: 105.180.929

CS M152: Project 3

Clock Design Methodology

Intro

Clocks play an integral role in our society and technology. They are generally effective for keeping track of time and for managing deadlines, but in technology they can play a more intricate role. One such example is the use of a system clock on a computer and the CPU which raises that clock to the speed required to perform at its given specs. Another example is the use of the clock to generate timer interrupt signals to allow the Operating System to effectively manage resources and keep a computer secure. This relates to the idea highlighted in the Lab background that clocks can allow designers, or our designs like Operating Systems, to step through and check behavior of electrical systems.

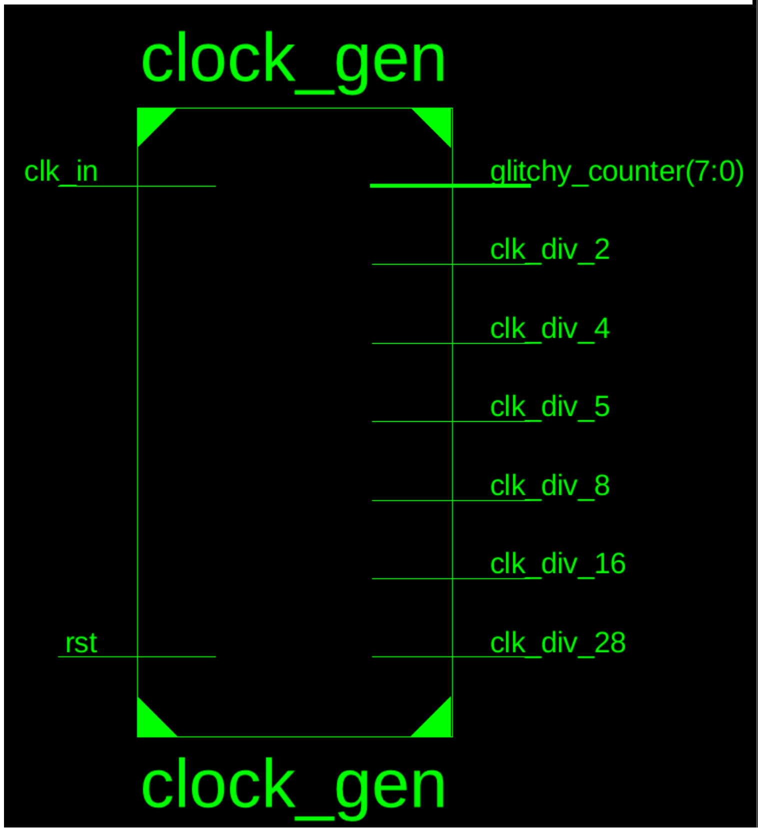
This lab focuses on four different types of clock modules we will explore. The first is a divide by power of 2 clock which will allow us to create an output clock with a proportionally even times larger period then the input clock. This will be created by focusing on the impact each bit has on identifying how many times we can slow down the original clock through extracting a bit. We do this by saving the value from the register and using a counter to count bits.

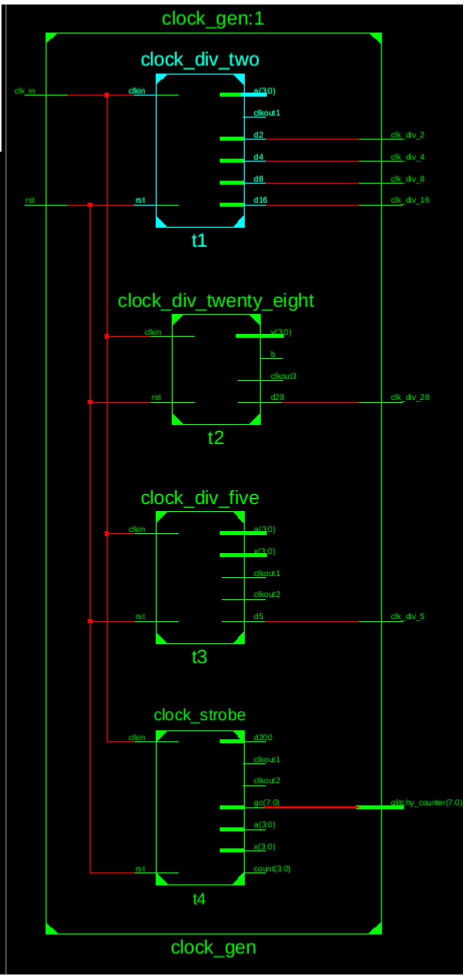
We can expand upon this capability and decide to use another counter or flip on and off clock signals to simulate another division of 2. This is shown in Part 2 Design Task (2) where we flip the clock signal at the 16th count where the 4-bit counter would overflow. We will do that to implement a divide by 32 clock.

In part 3 we will implement odd divided clocks by setting up our counters appropriately and sometimes combining waves. First, we need to remember that a duty cycles it the percentage or fraction of time that a clock signal is set during its clock period. We can achieve a target duty cycle by targeting the fraction we want and turn on our signal at one minus that fraction. We will combine this with wave addition using logical OR to achieve various duty cycles.

In part 4 we will isolate a signal to create a pulse. With this signal we can even more finely concentrate on extracting data since we can extract a value at just one point in time if we desire. We will use information from the previous parts to create a divide by 100 clock with a 1% duty cycle. This means we will keep the clock on for only 1/100th of the period and we want the original clock to complete 100 cycles before the new clock finishes one cycle. We will wrap up our Lab 3 by creating a glitchy counter that utilizes the pulse and a 8 bit counter to create a quirky counting pattern. Upon finishing this everything will be connected through the top module for testing through the nfctrL3TB main test bench file.

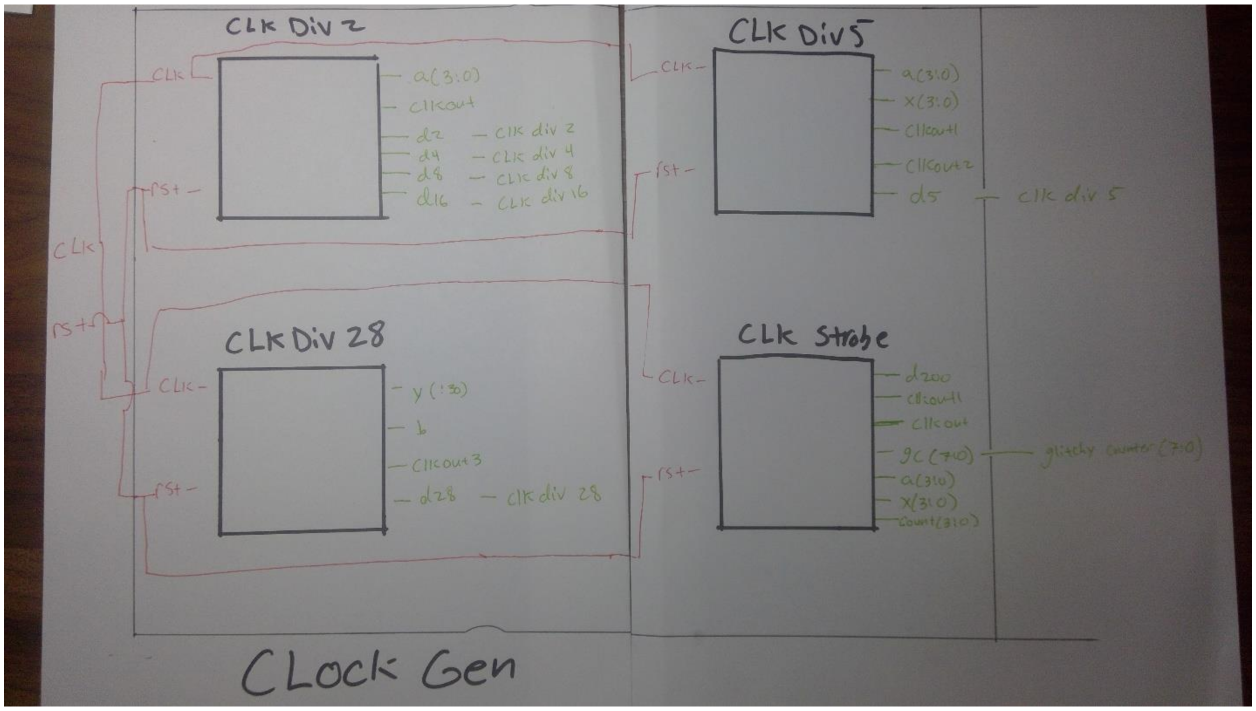
Design Description





Name: Chris Baker

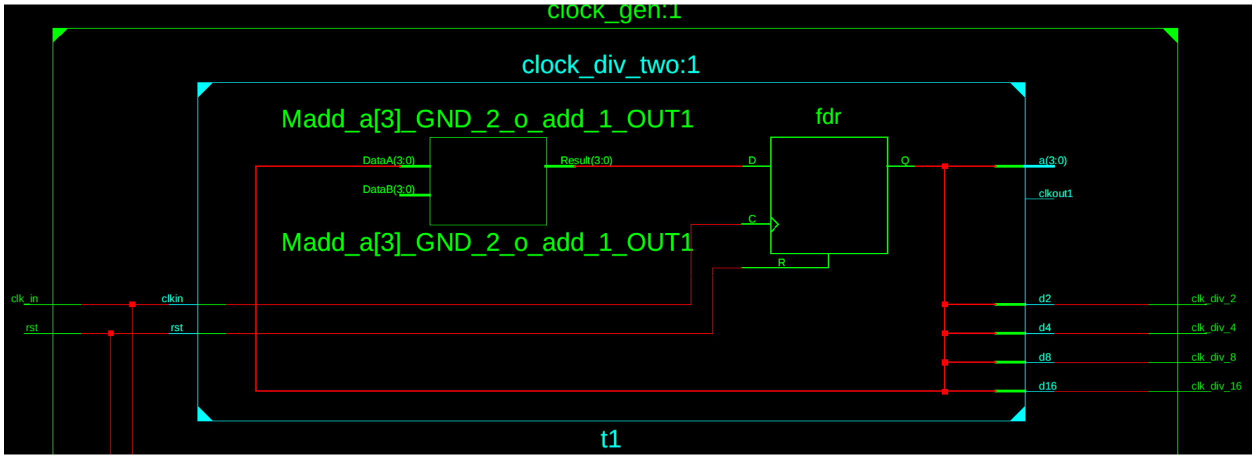
UID: 105.180.929



The top module for our clock generator is provided by the lab manual and is intended to utilize 4 submodules. The submodules include the clock 2^n divider, the even 28 divider, our odd 5 divider, and the glitchy counter. Each of these module takes a clock and reset value which are used to determine if values should be forcibly reset to 0. In our design we use the reset to also initialize our base counting values when reset is zero while encountering the first clock edge. There is also a part in the odd divider where we utilize negative edges and check for those too. The clock we used is the recommended 100MHz clock that flips every 5 ns as suggested by our TA. Among the top module are several other outputs we were instructed to provide. These include clk_div_2, clk_div_4, clk_div_8, clk_div_16, clk_div_28, clk_div_5, and the 8-bit glitchy counter.

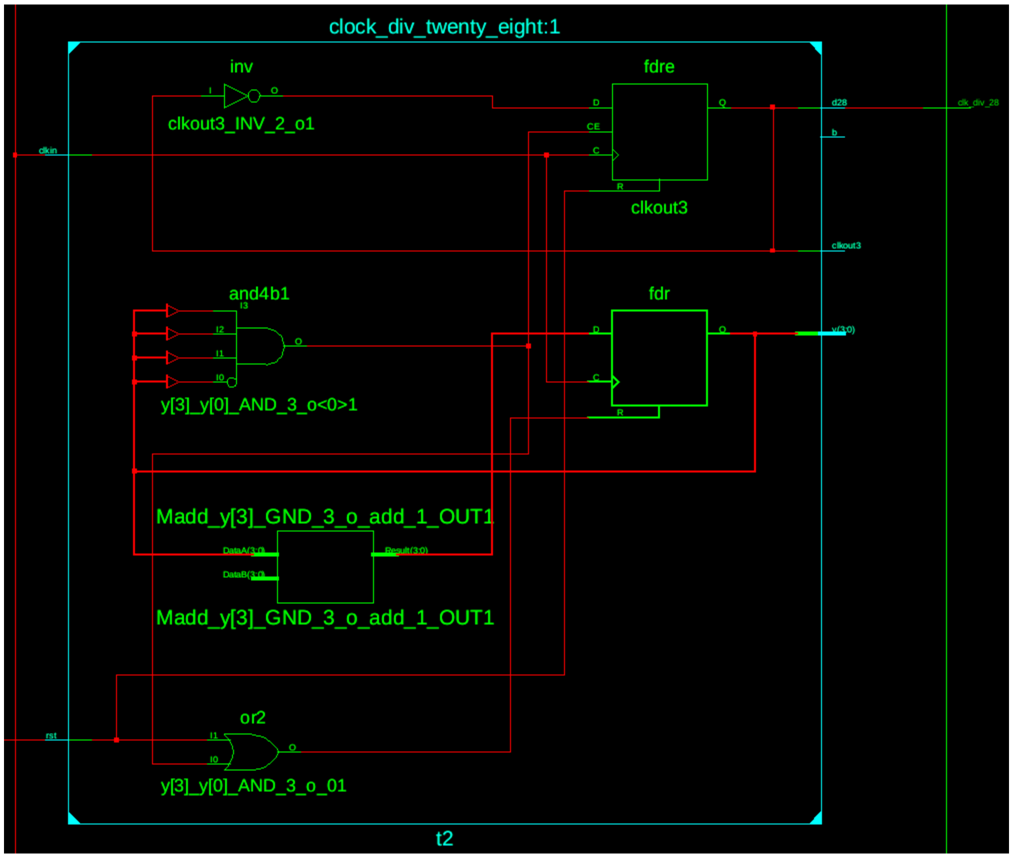
I approached this project from the bottom up design by performing one design task at a time. Although, this did make the ending synthesis more time consuming I believe it provided more depth, so we will start by exploring each submodule and my process.

Name: Chris Baker
UID: 105.180.929



The **power of 2 clock divider** was intended to show the relationship between the bits and the proportionally longer clock signal that can occur from dividing by that number in the counter. In my first lab I explored this idea briefly mentioning that the first bit is set every other time. Then next bit is set half as much as the first bit and this repeats. We saw that exact same idea exemplified in this submodule. To demonstrate this, I used near identical code from my first lab and chose to store the values of each bit into 4 different wires. This was a nice startup task given that most of our work for this was previously done and we just needed to assign the wires to extract those bits. Checking the results of these bits we noticed that the first bit created a divide by 2 output clock, the next bit made represented a divide by 4 output clock, and so on. Notice for the **RTL Schematic** above the flip flop takes the reset value as well as the clock to determine certain state conditions such as initializing values or clearing values. The Madd performs the counting. The O is the output which feeds back into the design and distributes the values to each wire for us to check our results. The clock_div_2 submodule provided a glance at how to approach dividing by other even numbers.

Name: Chris Baker
UID: 105.180.929

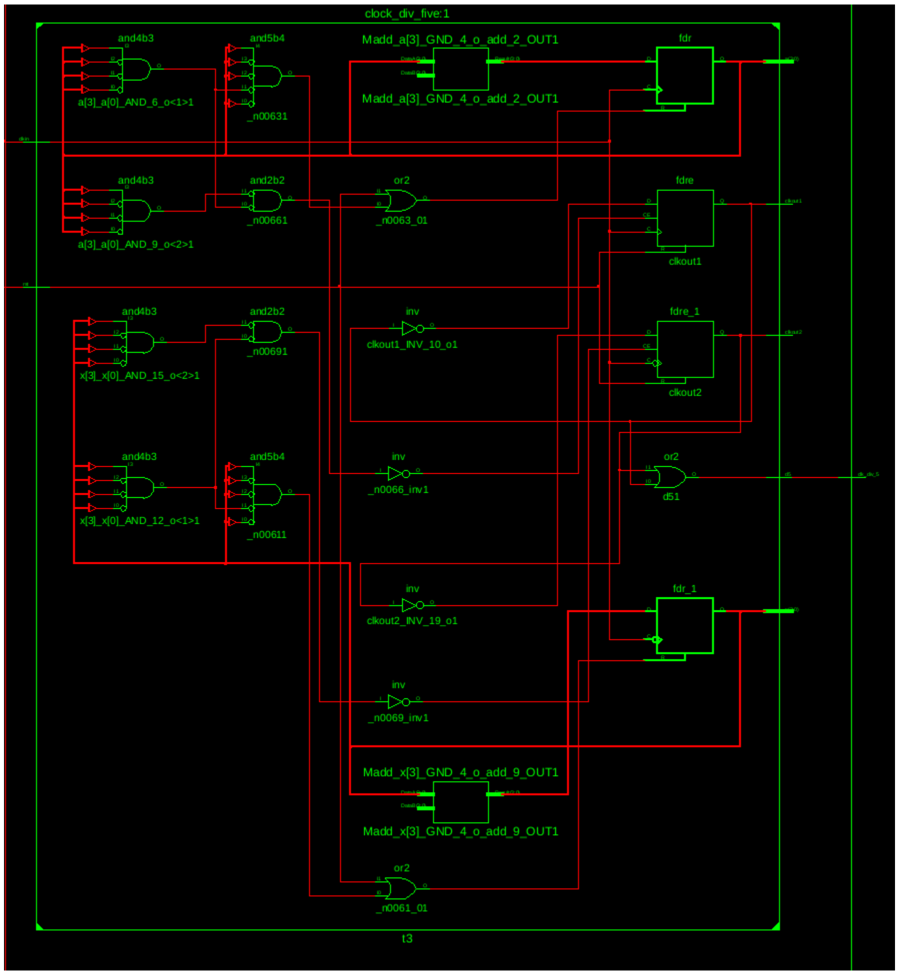


In our second submodule we needed to perform a **divide by 28**. Just prior to this we used a 4-bit counter which let us count to 16. Further, we flipped the clock on the 16th iteration which meant the clock was on for 16 original clock cycles and would then be off for 16 original clock cycles. The analysis drawn from this result is that flipping a counter once at the end of its max count will yield a clock divided by twice the max count. This allowed us to **successfully make the divide by 32 clock.**

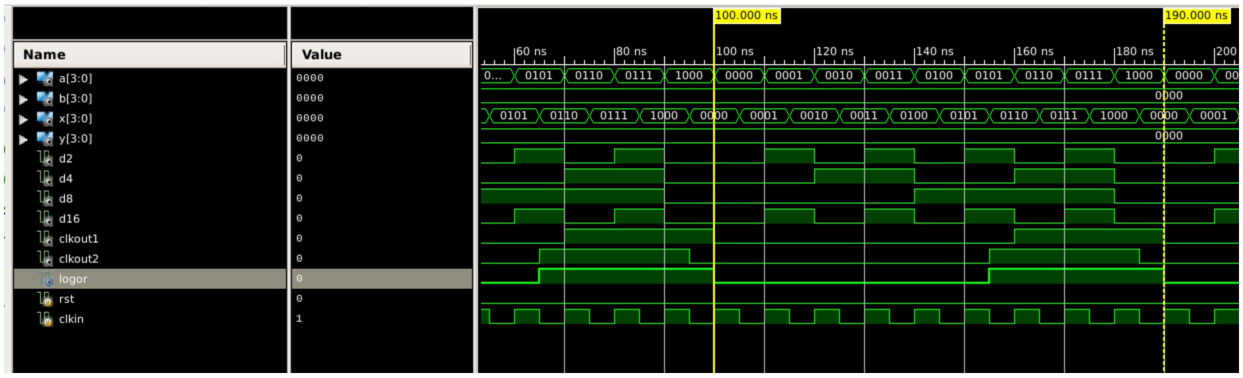
Noticing how the bits determined the clock division based on our 4-bit counter I knew I would need to implement a counter and flip the clock at some point in the counting. Initially thoughts were to flip at 14 or 28. I went to talk to my sister and brainstorm my approach with her hoping that thinking about it out loud would give me the answer. Through that conversation I realized that I would need to count to 14 and then flip. I decided to just test my idea at this point, however it was incorrect. During testing I saw 15 cycles had passed, so I understood that I wanted to flip at 14 cycles which meant I would need to flip on count at 13. **Running the RTL schematic**, we can see that two flip flops were generated. The bottom flip flop

Name: Chris Baker
UID: 105.180.929

fdr is responsible for counting and outputs 4 bits (could be values from 0 to 31). However, our design is meant to **generate a divide by 28 output clock.** This means that we use that counter and other combinatory logic shown above to check for when we achieve a count of 13. At this point we use the inverter for clockout3 to flip our clock signal. Further, once we count the next count of 13, we will then have achieved one cycle of our new clock. To output the cycle of our new clock the fdre flip flop is used where clockout3 can act as a reset. This allows us to create our divide by 28 clock by flipping the clock on after count from 0 to 13 and then resetting and turning it off after another 0 to 13.

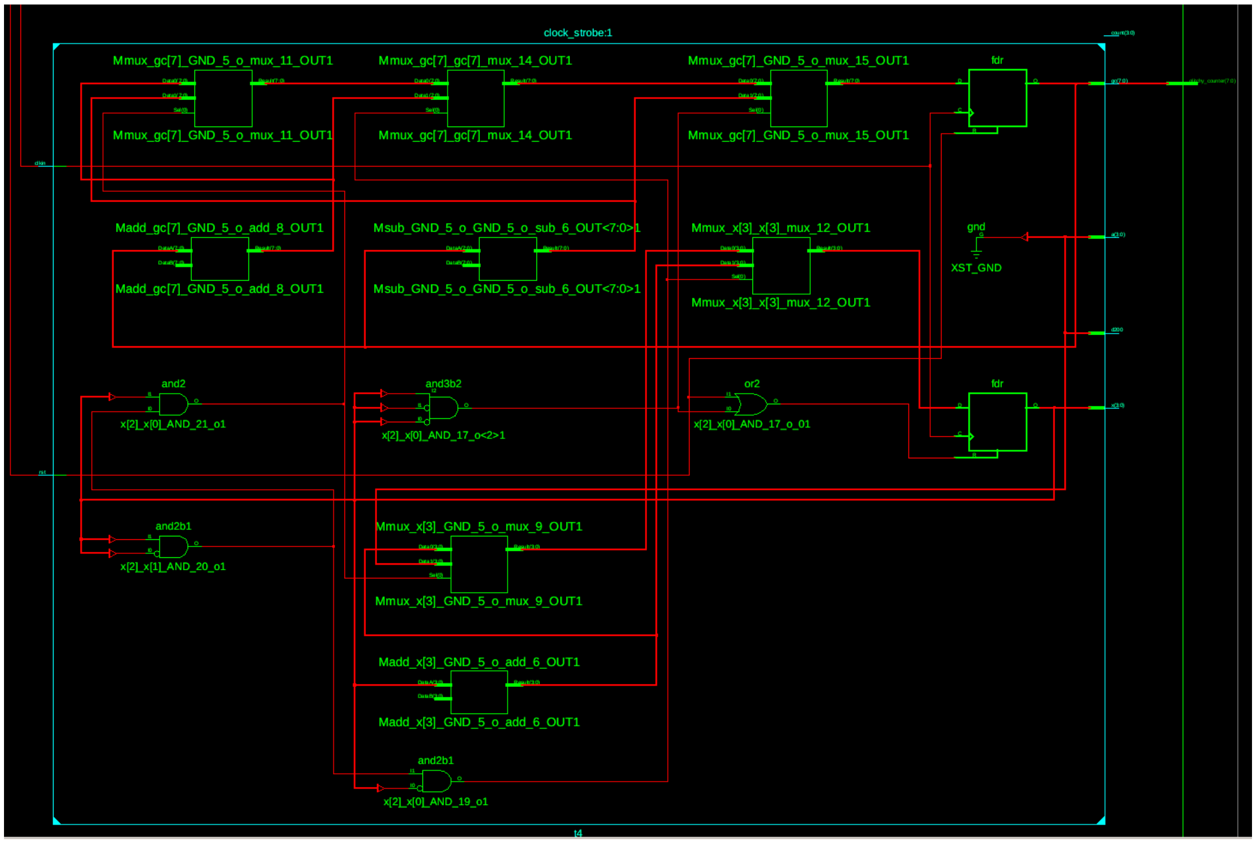


Knowing how the even counters we would apply those concepts to duty cycles and **odd dividers**. A duty cycle is the fraction or percentage of time that a clock is on/set during its period. Our first task here was to **create a 33% duty cycle**. I approached this idea by coming up with an interpretation for 33% which I chose 3/9. This means I would need 9 cycles and would have to turn on the clock and leave it on for 3 cycles. Alternatively, this means the clock would need to be off for 6 cycles. I thought this could be done by making a counter that counted from 0 to 5 and then turned on the signal. Then the clock could be left on for 6 ,7, and 8th counts. On the 8th count I would shut off the clock. I created a 4-bit counter to carry out that process with if statements and clock flips/sets and it worked. I followed the lab and repeated this idea with clock negative edges triggering the conditionals. I noticed here that I had to change test bench values for this to work properly since I made them for catching positive edges. After doing this I noticed a phase shift in when the signals were caught. This made sense since the original clock flipped every 5ns so, moving from pos edge to neg edge would change triggers by 5ns. I also thought back on old classes in physics and computer science to predict what would happen when I took the logical OR of two waves. I believed this would have a constructive interference type of effect where the wave would be set for all values that either of the two separate waves were set for. This was true as well and I was able to **extend my output clocks set signal by another 5ns. In other terms we took our duty cycle of (30ns/90ns) and added half of an original period split over our new time period (5ns/90ns) which gave us roughly 38.9% as our new duty cycle value (35ns/90ns).**



Name	Value
a[3:0]	0000
b[3:0]	0000
x[3:0]	0000
y[3:0]	0000
d2	0
d4	0
d8	0
d16	0
clkout1	0
clkout2	0
logor	0
rst	0
clkin	1

Then **to create the 50% duty cycle and divide by 5 clock** we made a counter from 0 to 4 and fiddled with values of adding the negative edge and positive edge. More specifically, I knew that setting the clock for 2/5 of the time would give me 40% duty cycle. This is equivalent to 20ns/50ns, so this led me to believe that if I could add another 5ns then I would get my 50% duty cycle. To set it for 2/5 of the time I had to turn it on at 1 – (2/5) which is 3/5. I made a counter from 0 to 4 and set clock when 2 was reached. I then would reset the values at 4. I copied this logic and implemented it with a neg edge trigger also. The results were as predicted after fixing a couple errors in between numbers I meant to use and some I accidentally typed. Like the divide by 28 clock example we have a couple of flip flops responsible for the output of 4-bit numbers which allow us to count from 0 to 8 (9 iterations). In **The RTL Schematic** The values are passed through multiple logic gates to determine when the count values that we use to turn on the clock and shut off the clock. Those values are fed into the flip **flops** which then have a feedback loop that will determine if the clock signal needs to be inverted for each clock in their respective flip flops. That same output that feeds back into those inverters also gets connected to the final or gate that takes the logical or of the outputs and returns d5 (clock_div_5).


clock_strobe:1
Mmux_gc[7]_GND_5_o_mux_11_OUT1
Mmux_gc[7]_gc[7]_mux_14_OUT1
Mmux_gc[7]_GND_5_o_mux_15_OUT1
fdr
Madd_gc[7]_GND_5_o_add_8_OUT1
Msub_GND_5_o_GND_5_o_sub_6_OUT<7:0>1
Mmux_x[3]_x[3]_mux_12_OUT1
gnd
XST_GND
and2
x[2]_x[0]_AND_21_o1
and3b2
x[2]_x[0]_AND_17_o<2>1
or2
x[2]_x[0]_AND_17_o_01
fdr
and2b1
x[2]_x[1]_AND_20_o1
Mmux_x[3]_GND_5_o_mux_9_OUT1
Madd_x[3]_GND_5_o_add_6_OUT1
and2b1
x[2]_x[0]_AND_19_o1
t4

Our last tasks revolved around using **Pulses and Strobes**. Our 7th design task was to create a **divide by 100 clock with a 1% duty cycle**. From our previous experience I knew I would count from 0 to 99 and that I would set only one original cycle in this new clock to have 1/100 set which is 1%. This meant I would need to have it off for 99/100. Further analysis made me expect to see the clock off for 990ns and on for the last 10ns if I my intentions could be accomplished. I created the counter and chose to set the clock on at 98 (99th iteration) and to clear all values after count 99 (100th iteration). This did yield the 990s ns of an unset clock and 10ns of a set clock. The next part was confusing, and I kept watching the TA video to try and get clarification. However, it was not until I started to try drawing what was meant to be communicated that I began to understand what I was supposed to do. To create a 50% duty cycle I know the output clock would need to be on for half the time and off half the time. If I **made the new clock a divide by 200 clock, then it would take 2000ns** for the period. This matches appropriately with the given frequency of 500Khz. By drawing on paper I realized that I just needed to flip an output clock whenever a pulse occurred on the first clock. I let a new variable clock (clock2) take on the value of the flipped clock and I had a new clock that would be set for 1000ns at a time or off for 1000ns and would have a period of 2000ns and 500Khz frequency. Our last task was to use a variation of the pulse with a 4 strobe to generate an 8-bit counter with an interesting pattern of adding 2 unless the strobe was encountered, then it would decrement 5. We started by going to paper again and drawing it out. I planned to add 2 three times as shown in the lab. Then I would need to subtract 5 from my counter. I created a pule that would turn on at the value 2 in a 4-bit counter. Then I would shut the pulse off and reset at the value 3. This gave me an extracting pulse to work with that could be used every 3 iterations. Then on the next iteration I would clear all values. When I cleared values, I needed to subtract 5 from the glitchy counter, and all other times I would add 2 to the glitchy counter. After drawing it on paper and then coding it I was able to get the result more smoothly then just thinking about a pulse and its position and all the logic involved. In the **RTL Schematic** many of the muxes are used to select the values that we wish to detect such as when the pulse should be set and when to add values or decrement values. These are fed to other logic gates and the flip flops which then perform counting, potential clears, and the output data such as our glitchy counter result. I had extra variables in here some of which were used for testing like count and others like d200 and 'a'. These were originally used to count cycles that occurred and were other counters. Them still being in the RTL made it messier and may have produced the ground value we see that could be setting unnecessary values to zero.

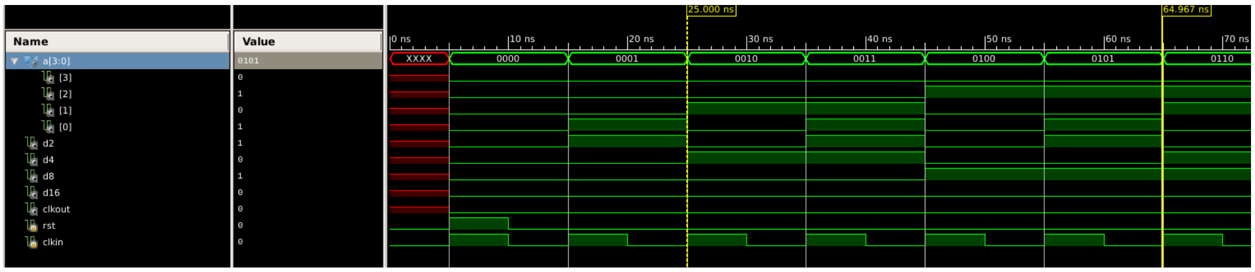
Name: Chris Baker
UID: 105.180.929

Finally, we designed our top level module by making sure we named the variable appropriately. I then performed some adjustments on linking my variables in submodules explicitly to the top-level module. I also tried to use a similar test bench for the top module that worked across most of my submodules. Lastly, I name each instance of the module in a similar manner to the example shown in the Lab Manual last page because I still feel unfamiliar with module incorporation in Verilog code.

Simulations and Tests

The following tests were set up using initialized values, flipping clock signals, and counters. The counters were all verified to restart if the counter overflowed and in most cases I set them to zero before anything like that would even happen. Thus, we have avoided edge cases in all these demonstrations. The following waveforms and explanations will focus on the visual tests shown and results. **I will also include an errors section after all the tests** since I documented them in a clean and organized manner.

Part 1: Clock Divide Powers of 2

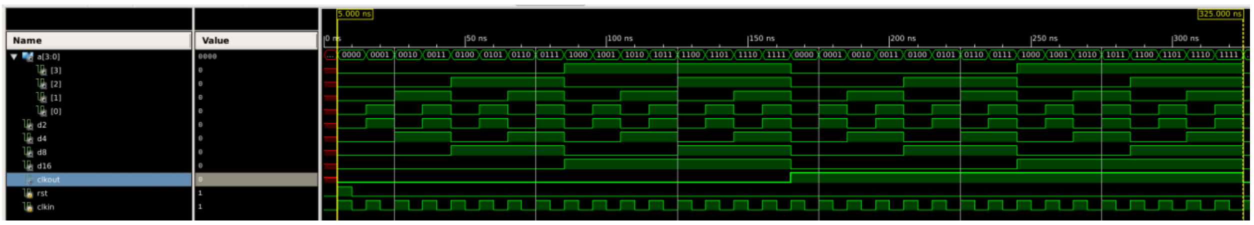


(1) In the above wave form we accomplished task 1 which was to assign four 1-bit wires to the 4-bit counter. I tested this visually for divide by clock correctness by counting how many original clock cycles it took until one of the bits had completed a period. In the above example I marked 25ns and 65ns to show one period for the divide by 4 counter. Notice the fourth signal of clkin occurs on 65ns and the divide by 4 clock completes its first period there. This test was repeated for each clock counter. Divide by 2 clock had 2 original clock signals before 1 occurred, 8 clock had 8 original signals before 1 occurred, and 16 clock had 16 original signals before 1 occurred.

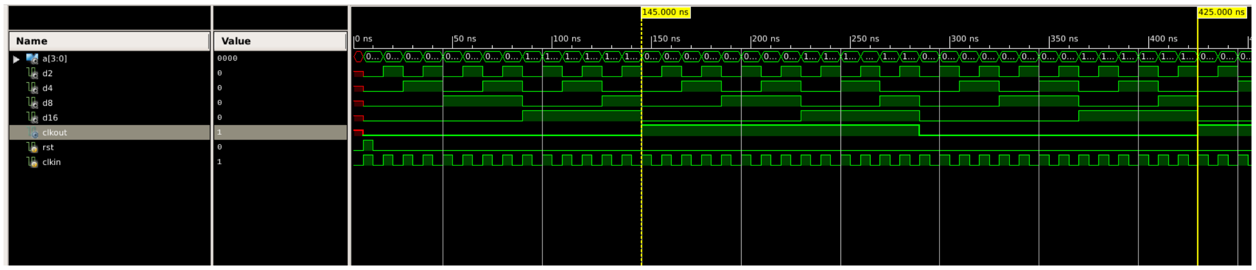
Part 2: Even Divisions

Name: Chris Baker

UID: 105.180.929

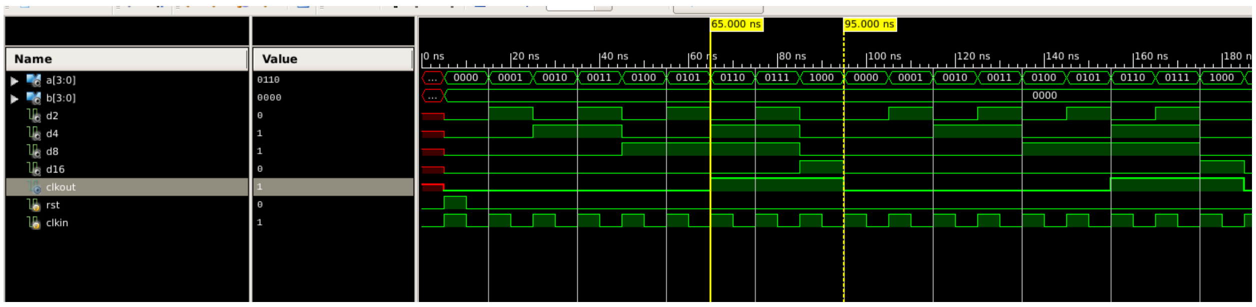


(2) Then we created a divide by 32 clock. I noticed from counting the other dividers that I could use them to quantify bigger clock dividers, but the simplest method I resolved was that we could count how many original clock signals occurred in just the off portion or on portion alone and multiply that by 2. Thus, we see that from 5ns to 165ns there have been 16 clock signals. 16 times 2 is 32 clock signals, which covers the whole period of our divider. Another method is to count the time and divide by the original clock signal periods. So 325ns – 5ns = 320ns. 320ns/ 10ns = 32 signals occurred within our 32 times longer period new clock.



(3) The example above shows our divide by 28 clock divider. We will use the time method to show the accuracy here. 425 ns – 145 ns = 280ns . 280ns / 10ns = 28 original cycles occurred in the new 28 times longer clock period. I also compared the signal by hand and counted to double check.

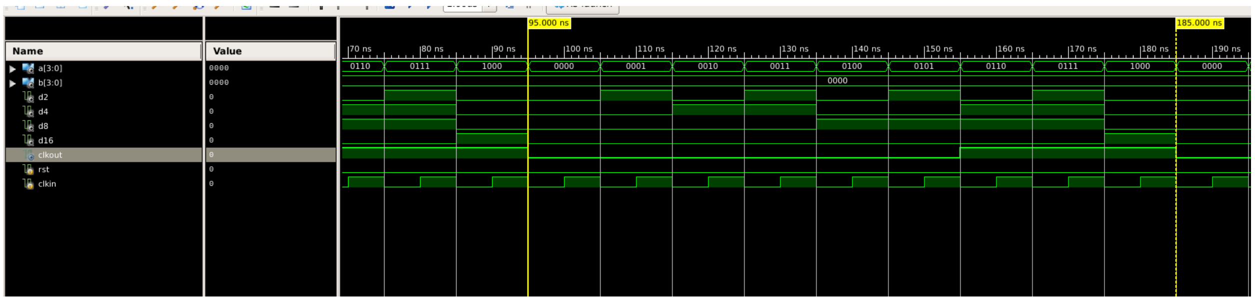
Part 3: Odd Clock Dividers

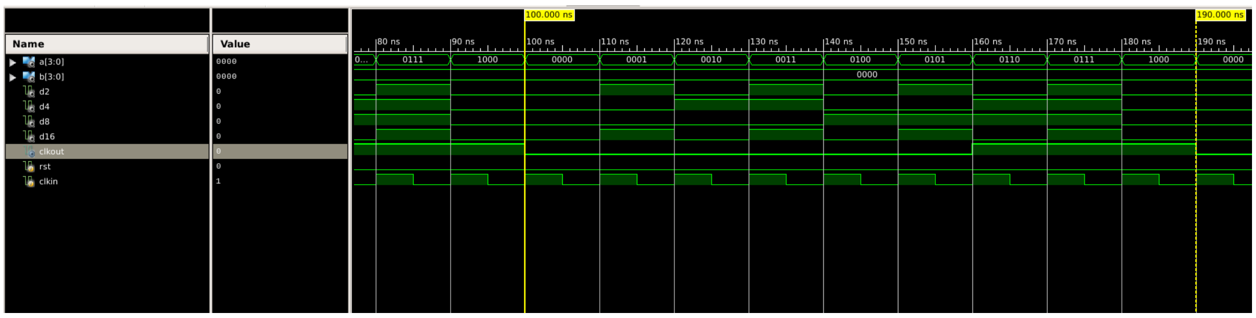


Name: Chris Baker

UID: 105.180.929

(4) The waveform above shows a 33% duty cycle using if statements and counters. I selected the 65 to 95 ns portion to show when the output clock is on. This occurred for 3 counts (0110, 0111, 1000) , but there was 9 total counts (0000 to 1000) . Thus, we have 3/9 set which is 33% duty cycle. I also verified this with counting the nanoseconds. Take away 5 ns from everything to center wave graph at 0. 90ns – 60ns = 30ns. 30ns/ 90ns = 33% duty cycle.

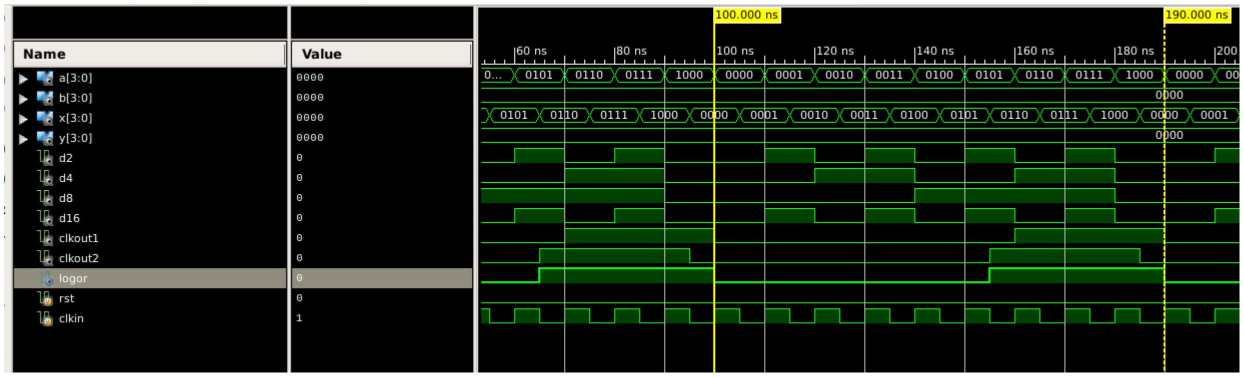




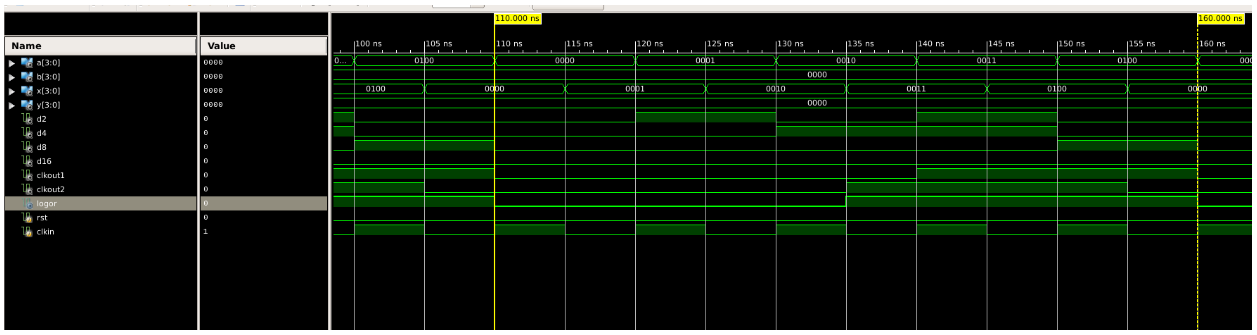
(5) Viewing the two waveforms side by side we see that there is a 5ns phase difference between the waves. This makes sense since the pos edge triggers at the setting (left side) of any given pulse while the neg edge triggers when it falls (right side) of any given pulse. Our signals flip every 5ns, so we see the 5ns difference here. Shown above from range of 95ns to 185ns vs 100ns to 190ns. The neg edge is the later time values .

Name: Chris Baker

UID: 105.180.929



(6) I predicted that assigning the Logical OR of the two waveforms would result in any value that was set for any duration would remain set. This was what happened and makes sense since the logical OR is 1 for any non-zero value in Computer Science. Our non-zero values ORed allowed us to extend our wave by another 5ns. This was another observation. That half the original clock period would be added when you incorporate the neg edge to the posedge value.



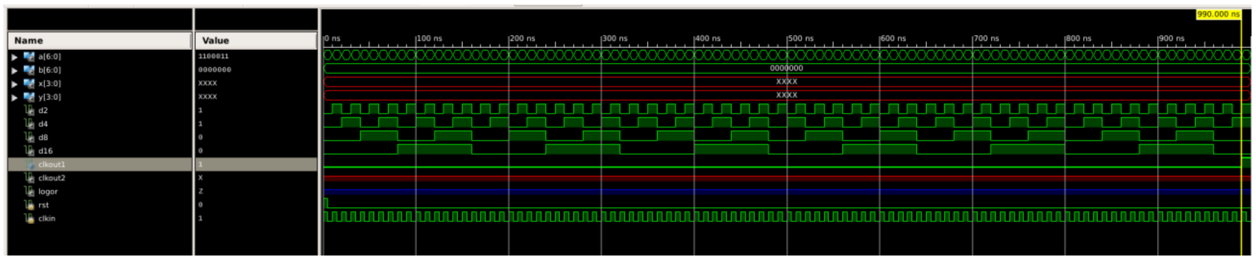
(7) The above picture shows a 50% duty cycle with a divide by 5 clock. This can be seen using the methods I described above to verify this. First, we can take the 160ns – 110ns = 50ns. Since our original clock period is 10ns then we have 5 iterations (also demonstrated by the count from 0000 to 0100 above). To prove our 50% duty cycle we take our midpoint (160ns + 110ns)/2 = 135ns which is where our signal flips back on for our divide by 5 clock between the yellow region we indicated. We accomplished this by noting our counter was going to give us 2/5 duty cycle which was 20ns/50ns. In order to get the remaining 5ns, we used the Logical OR with the negedge to grab the 5ns from the phase difference. 25ns/50ns = ½ = 50% duty cycle.

Part 4: Pule/Strobes

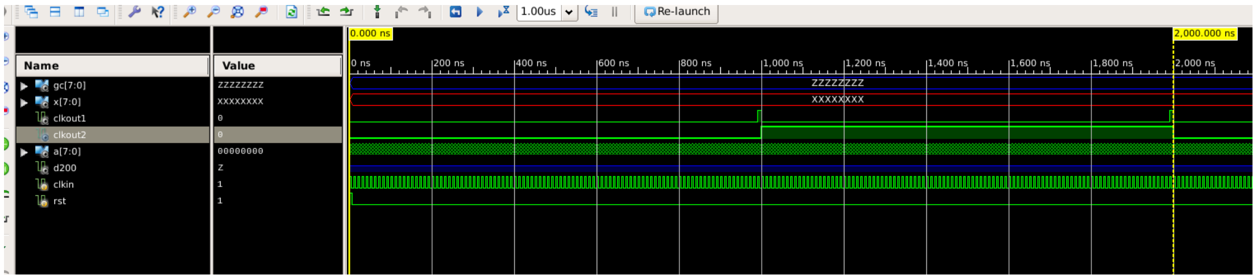
Name: Chris Baker

UID: 105.180.929

Please ignore the extra values. As stated in the intro I started from the bottom up and so I had many variables and tests for accuracy as I built a large one window program.



(8) 1. We need to create a divide by 100 clock with 1% duty cycle. So, I made a counter go up to 98, then set the clock, then turn off at count 99. This got my clock signal on at 990ns and it turned off at 1000ns. (1 – 990/1000) = 1% duty cycle. Or inspect it and see there only one tiny original clock signal at the bottom that matches the clkout1 signal turning on just at the end on the yellow line. I also verified these tests by using the green arrow button that allows you to skip value transitions on the waveform graph just above the middle. This helped me guarantee when values changed from 0 to 1 and 1 to 0.



2. The second parted was to then turn that clock into a 50% duty cycle divide by 200 clock. I did this by flipping the clock signal at the phase shift. We see above the one tiny pulse in the middle and right below it our clock turns on.

To verify correctness, we check several things.

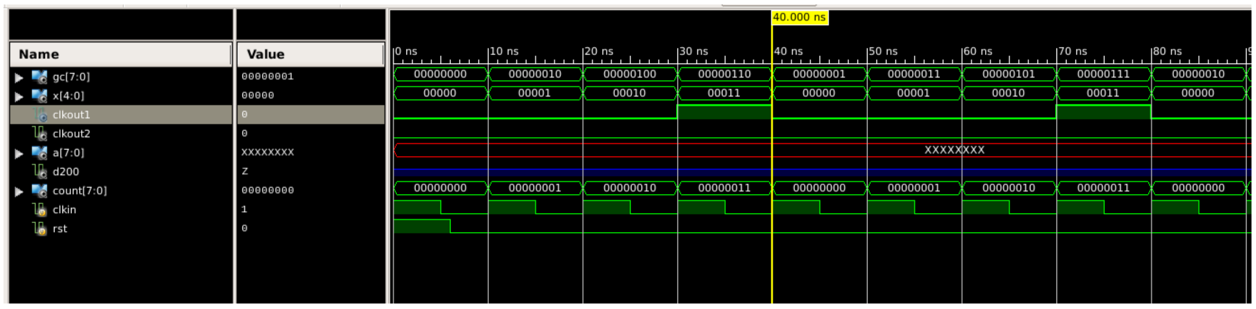
Period: 2000ns – 0ns, our period is 2000ns which is twice as long as the divide by 100 clock 1000ns period.

Name: Chris Baker

UID: 105.180.929

Duty Cycle: The clkout2 turns on at 1000ns and stays on till 2000ns. That is ½ the period. Thus, 50% duty cycle.

I even counted. With a counter. To triple check.



Name	Value
gc[7:0]	00000001
x[4:0]	00000
clkout1	0
clkout2	0
a[7:0]	XXXXXXXX
d200	Z
count[7:0]	00000000
clkin	1
rst	0

(9) For the glitchy counter we utilized all tools and ideas from previous modules to set up a pulse that occurred on the count 3 of a 4 iteration (3 bit) counter. We increment values of 2 everytime except when we finish the pulse, then we subtract 5 from our whacky counter. We visually inspected this and can reaffirm the pattern.

00,10,100,110,001,011,101,111,010 …

0 , 2, 4 , 6 , 1 3, 5 , 7 , 2 ….

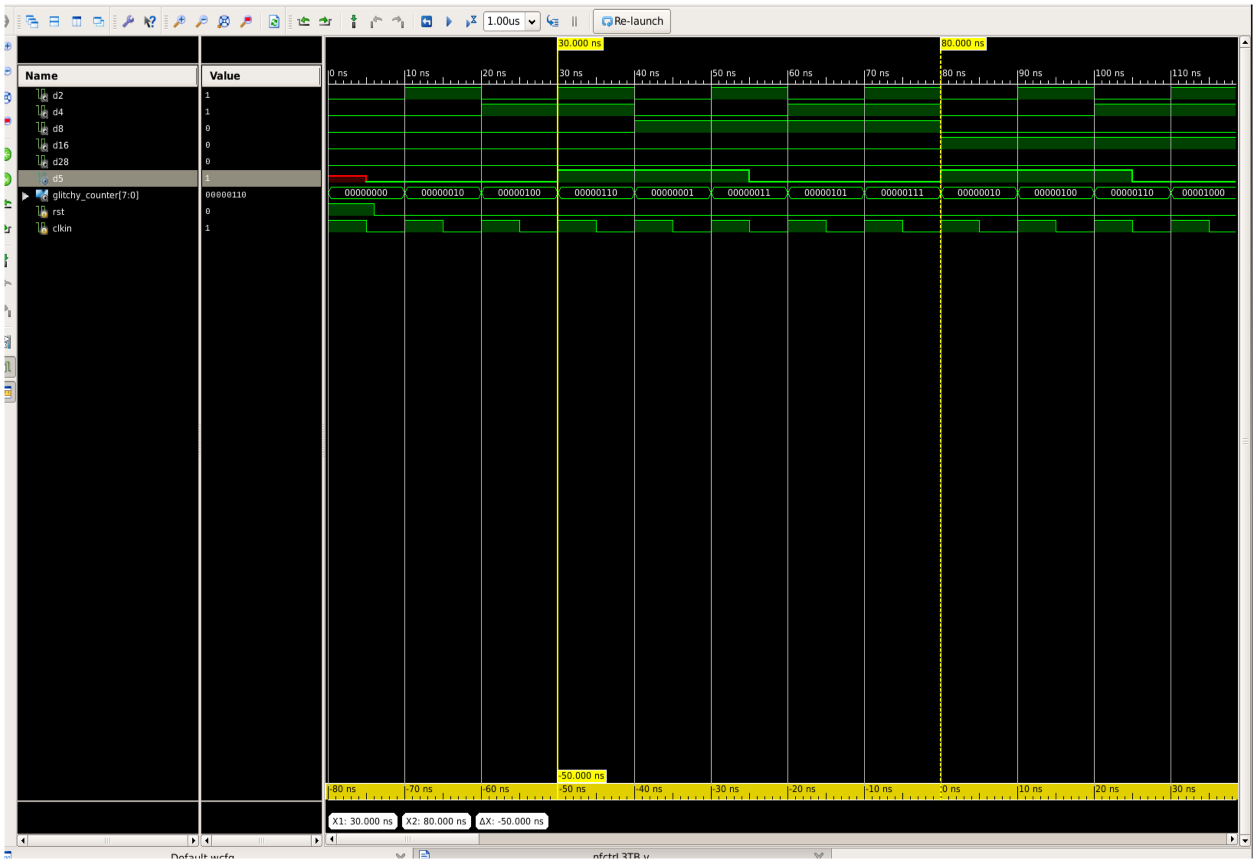
```
//
////////////////////////////////////////////////////////////////////////////////////

module clock_gen(clk_in, rst, clk_div_2, clk_div_4, clk_div_8, clk_div_16, clk_div_28, clk_div_5, glitchy_counter
    );
    //, clkout1, clkout2, a, b, x, y
    //clkout1, clkout2, a, b, x, y, logor, clk_in
    //Set up our output wires, and 4 bit reg. Input: clk and rst
    output wire clk_div_2; output wire clk_div_4; output wire clk_div_8; output wire clk_div_16;
    output wire clk_div_28; output wire clk_div_5;
    input rst; input clk_in;
   wire clkout1; wire clkout2; wire clkout3; wire logor;
   wire [3:0] a; wire b;
   wire [3:0] x; wire [3:0] y;
   output wire [7:0] glitchy_counter ;
```

(10) The above shows our module with the ports as specified in Lab manual , and extra test values.

Name: Chris Baker

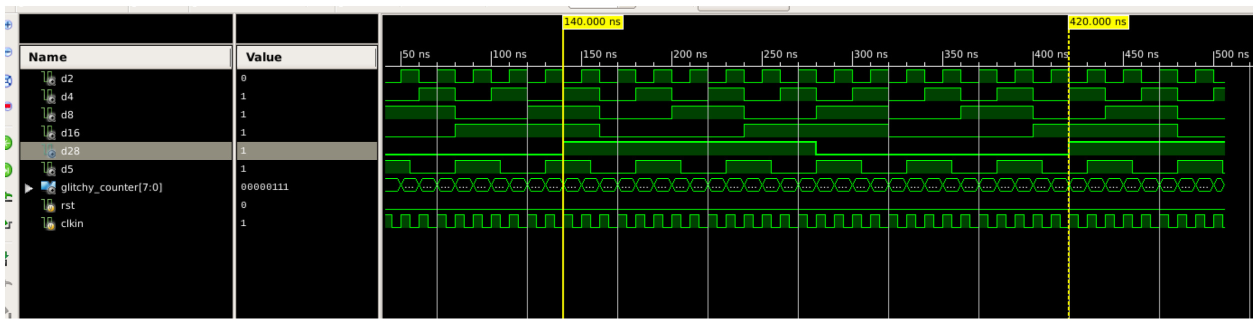
UID: 105.180.929



The above demonstrates a snip of the overall testbench top level module. I demonstrate here that the 5 clock divider works as before in my submodule. It takes 5 original clock signals until the new longer 5 clock divider signal repeats. Notice the 2, 4, 8, and 16 clock dividers work as intended also. There are 2 clock signals for every one 2 clock divider signal repeat , 4 clock signals for every one 4 clock divider signal repeat ….

Also note the glitchy counter pattern is the same:

0, 2, 4, 6, 1 , 3 , 5 , 7 , 2 , 4, 6 , 8 …



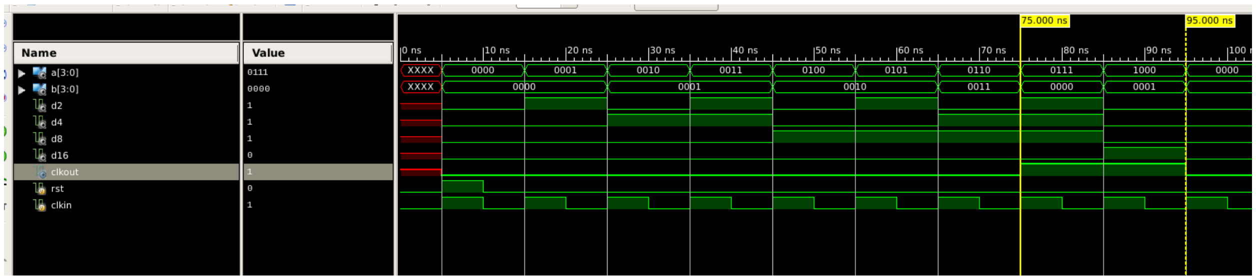
Name: Chris Baker

UID: 105.180.929

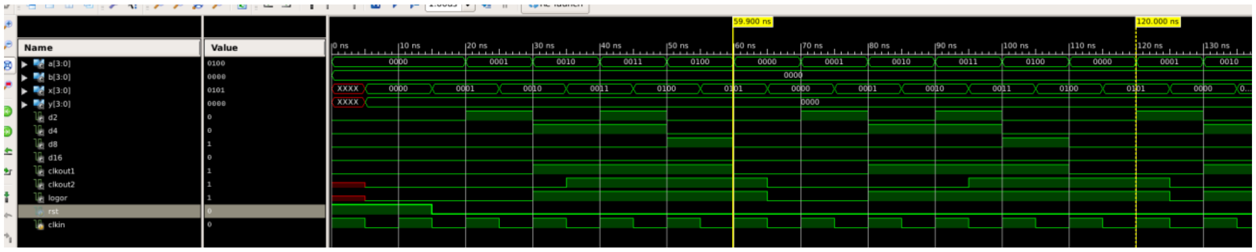
Above is the focus on the 28 divider . Using our time count method we take 420ns – 140 ns = 280ns . 280ns / 10ns = 28 clock signals that occur for one new divide by 28 clock signal to repeat.

Thus, all submodules and top level modules are tested to work as intended.

Errors: (Parenthesis encapsulate the part where the error occurred and what it was)



Clock output stayed on in above when it was suppose to reset, this was a logic error fixed in conditionals



Notice above when trying to add the waveforms I knew something was wrong because the posedge was on for much longer then 5ns. I set the counter incorrectly when rewriting some code so one edge triggered much earlier then the other. I fixed this appropriately.

See More errors and which part they occurred in below:

(1) Assigning wires twice leads to X (DC) values. I may not initialize them to 0.

assign d2 = 1'b0; assign d4 = 1'b0; assign d8 = 0; assign d16 = 0;

driving out

(2) Did not like using a <= 4'b1111 as comparison, flipped every cycle

(3) My above guess was incorrect. At 155ns clkout was set to 1. Next set to 1 was at 305ns.

So 150ns/5ns = 30. We had 30 cycles. Because we counted from 0 to 14, so that is 15 for each clk flip

(4): I rearranged conditions to try and catch higher priorities first

I also had to change the detection of when to turn on the clock:

b[1] & b[0] would detect counter at 5, one iteration after. So i only had clock set for 2/9th of the time

I adjusted for this error by catching a[0] being set twice with b[1] & !b[0] with a[0] to catch the 5 immediately

(5) I copied an extra end, assigns, and end module, Also Test Bench had to be updated to initialize neg edge conditions

(6) I was using the same variables/registers across both always blocks, because the Lab Manual prompted us to duplicate our code. I was received a permanent log or when it was set. This was because both code pieces adjusted same variables. Fixed this by using differ variables in them.

Conclusion

In designing the clock generator, we incorporated 4 sub modules. Each module performed some type of clock division to demonstrate how clock dividers can be created to slow down a signal proportionately over a larger period. In practicing the modules, we learned how to implement clock dividers in powers of 2, other even powers using counters, odd powers using counters with negative edges, and pulses that isolated a signal. Our top module contained several output values to capture and represent these frequency and period of each clocks. Those outputs consisted of clock dividers 2, 4, 8, 16, 28, 5, and even a 8 bit glitchy counter.

Analyzing our top module you can see that the overall main test bench distributes our inputs of system input clock and reset signal to each sub module. In our RTL Schematics you can see that the flip flops performed counting and the inversions necessary to determine and flip clock signals. Multiplexers, logic gates, and other combo logic made up the tests that would determine when to isolate a pulse, flip a signal, or set the clear or reset on a clock. Any final combinational logic like those used to OR the pos edge and negedge wave forms for manipulating duty cycles and odd dividers would also be used at the end before some outputs. These outputs were all generated on the same top level module from the main top level test bench, but their output values were generally independent of each other.

Challenges in this lab was trying to reorganize everything after I spent so much time building from the bottom up. I had every sub module tested and working great, but when trying to connect everything to the top module I had incorrect pairings and had many small errors. The most notable error came after I thought everything would work well. I ran the test bench and saw that my divide by 28 and divide by 5 were either never set or were on high impedance. I ran through code for a couple hours and looked at documentation of errors. Eventually I checked the main test bench and saw that I explicitly called those ports incorrectly and divider 28 was trying to drive both divide 28 and divide 5. So, when I fixed this and reran the test both of those issues cleared up. The other issue was trying to create the top-level module all together. There were numerous errors I had in connecting with so many test variables. I eventually discovered I could declare only the necessary ports in the module and leave the other variables I use as wires outside the main module. Then I would be able to connect those in the submodule instances and use them in my programs without VHDL improper port declaration errors. To deal with this I tried to let things be clean and simple and focused on the lab manual directions and look at documents on writing Verilog modules. My other errors and challenges are well documented above, but most of those logic issues were solved by reason and writing thoughts out clearly on paper with clock and value mapping.

To improve this lab, I would recommend a read through for clarity of having a fellow TA or professor look to make nice clear edits of the paper. Implementing a couple of these parts felt difficult to understand, especially part 4. These topics can look different to less experienced individuals compared with professors or master’s students in engineering. I would also very much appreciate a better tutorial on modules and Verilog then those available on the web. If there was a TA or professor who was very well experienced at UCLA who could design more material to help students understand, then that would save me hours on this project and help me better understand what I am doing when I write code in Verilog compared with a familiar well documented language like C. Further, these labs take a lot of time for a single individual to perform. If there was anyway to reduce this or give more time, then that would be helpful. I know credits do not reflect work well in engineering majors, but the workload in this class is more than other 3 and 4 credit courses I have taken.