Joshua Aymett

COM SCI M152A-6

02/7/2021

Project 2: Clock Design Methodology

Project Summary

Clock dividers are circuits that enable the division of clock frequencies by a numeric factor. They are particularly useful for synchronization between devices and can be used in circuits that require some sort of division of clock frequency. Further adjustments to clock frequency can be accomplished by modifying the duty cycle, or ratio of time on to time in the clock period. In this lab, the following types of clock dividers are implemented:

-Divide-by-2^n – Divides by some power of 2 (e.g. 2, 4, 8, 16, or 32)

-Divide by even factor – Divides by some even factor (e.g. 28)

-Divide by odd factor – Divides by some odd factor (e.g. 5)

-Strobe clock divider – Divides by some unspecified factor with a small duty cycle

Each of these clocks is initialized using a reset flag which sets all counting-related variables to 0. They differ by counting method which is described in each module's section. The types of clock dividers discussed above were used to implement the following tasks:

(1) Design Task: (Divide-by-2^n counter) Assign 4 1-bit wires to each of the bits from the 4-bit counter

(2) Generate the divide by 32 clocks by flipping the output clock on every counter overflow.

(3) Design Task: Generate a clock that is 28 times smaller by modifying when the counter resets to 0.

(4) Generate a 33% duty cycle clock using if statement and counters and verify the waveform.

(5) Duplicate the design in another always block that triggers on the falling edge instead. View the two-waveform side by side.

*(6) (Answer the question) "What happens if you assign a wire that takes the logical or of the two 33% clocks?"***

(7) Design Task: Generate a 50% duty cycle divide-by-5 clock.

(8) Create a divide-by-100 clock with only 1% duty cycle using the counter methods previously introduced in parts 2 and 3. Create a second always block that runs on the system clock (100Mhz) and switch the output clock every time the divide-by-100 pulse is active with an if statement. Verify that the output clock is 50% duty cycle divide by 200 clock running at 500Khz.

(9) Design task: Use the master clock and a divide-by-4 strobe to generate an 8-bit counter that counts up by 2 on every positive edge of the master clock, but subtracts by 5 on every strobe. 0→2→ 4→6→1→3→5→7→2→4→6→8→3→5→7→9→4→…..

***See odd divider section for answer**

clock_gen Project Status (02/07/2021 - 20:46:06)			
Project File:	M152_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:**	6 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)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	20	18,224	1%	
Number used as Flip Flops	20			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	14	9,112	1%	
Number used as logic	14	9,112	1%	
Number using O6 output only	7			
Number using O5 output only	0			
Number using O5 and O6	7			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number of occupied Slices	4	2,278	1%	
Number of MUXCYs used	0	4,556	0%	
Number of LUT Flip Flop pairs used	14			
Number with an unused Flip Flop	1	14	7%	
Number with an unused LUT	0	14	0%	
Number of fully used LUT-FF pairs	13	14	92%	
Number of unique control sets	1			
Number of slice register sites lost to control set restrictions	4	18,224	1%	
Number of bonded IOBs	16	232	6%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	1	16	6%	
Number used as BUFGs	1			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	248	0%	
Number of OLOGIC2/OSERDES2s	0	248	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	32	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	4.26			

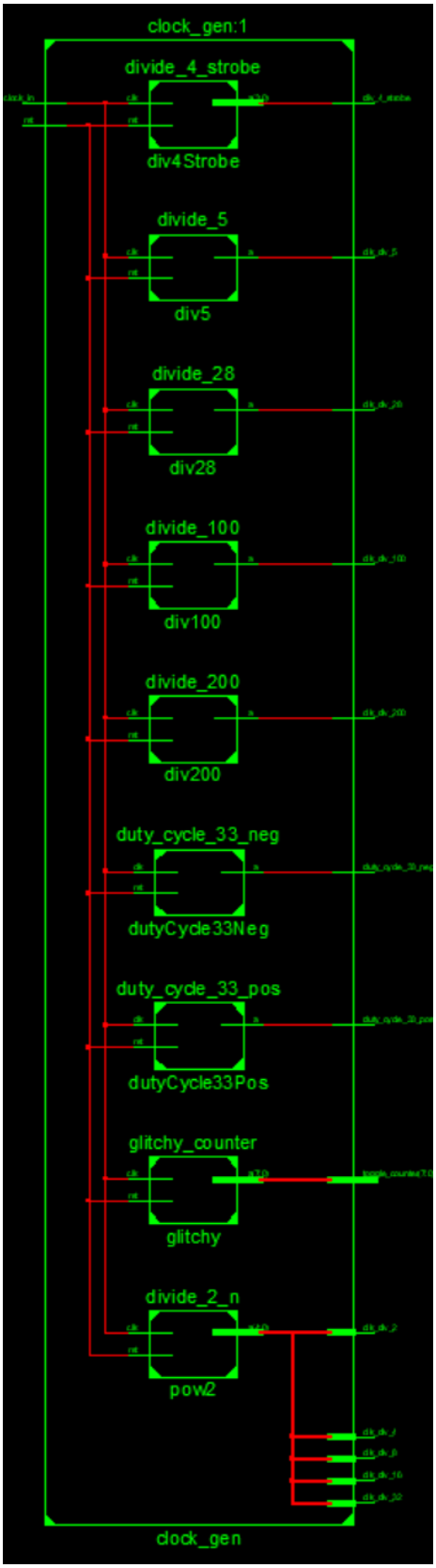
Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	
Timing Constraints:	All Constraints Met			

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Feb 7 20:45:21 2021	0	6 Warnings (0 new)	4 Infos (1 new)
Translation Report	Current	Sun Feb 7 20:45:32 2021	0	0	0
Map Report	Current	Sun Feb 7 20:45:45 2021	0	0	6 Infos (0 new)
Place and Route Report	Current	Sun Feb 7 20:45:55 2021	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Sun Feb 7 20:46:03 2021	0	0	4 Infos (0 new)
Bitgen Report					

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
ISIM Simulator Log	Out of Date	Sun Feb 7 20:42:23 2021	

Date Generated: 02/07/2021 - 20:46:06

Design summary of clock_gen module. The presence of slice registers indicates that the module is a sequential circuit.



High level design of clock_gen. Displays how each submodel connects to parent module output.

Design and verification task waveforms together (Design tasks at top, verification tasks below)

From top to bottom:

Design Tasks:

1. Clock input

2. Divide- by-2

3. Divide-by-4

4. Divide-by-8

5. Divide-by-16

6. Divide-by-5

7. Glitchy Counter

8. Reset Signal

Verification Tasks:

9. Divide-by-32

10. Divide-by-3 33% duty cycle positive edge

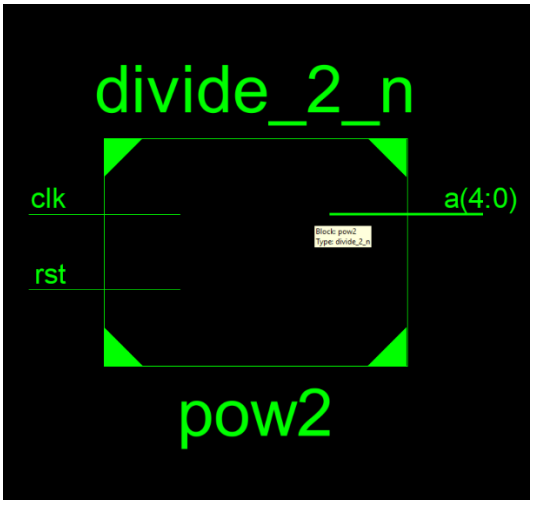
11. Divide-by-3 33% duty cycle negative edge

12. Divide-by-100 strobe 1% duty cycle

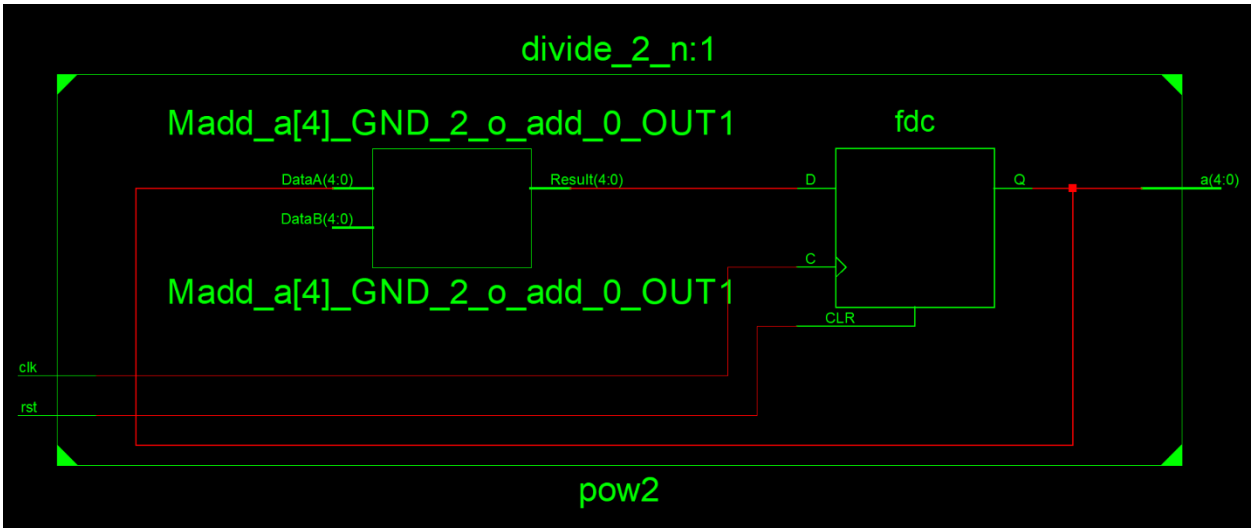
13. Divide-by-200 50% duty cycle based on divide-by 100-strobe

14. Divide-by-4 strobe

2^n Clock Dividers:

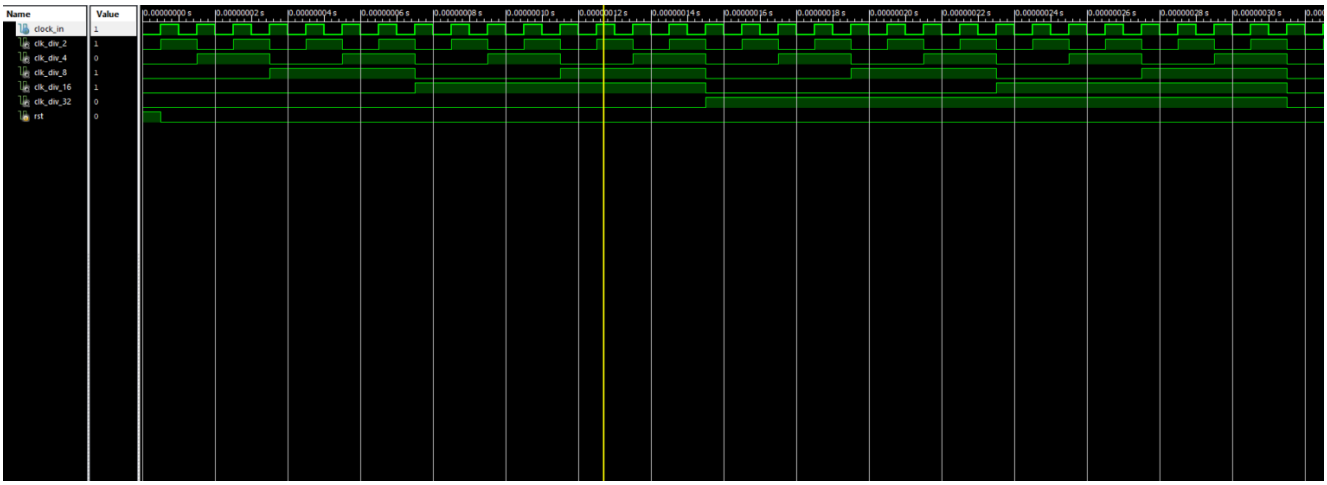


Black box of divide_2_n listing inputs and outputs



Schematic of divide_2_n expanded 5-bit version. Displays an adder connected to a D flip flop with async clear

---

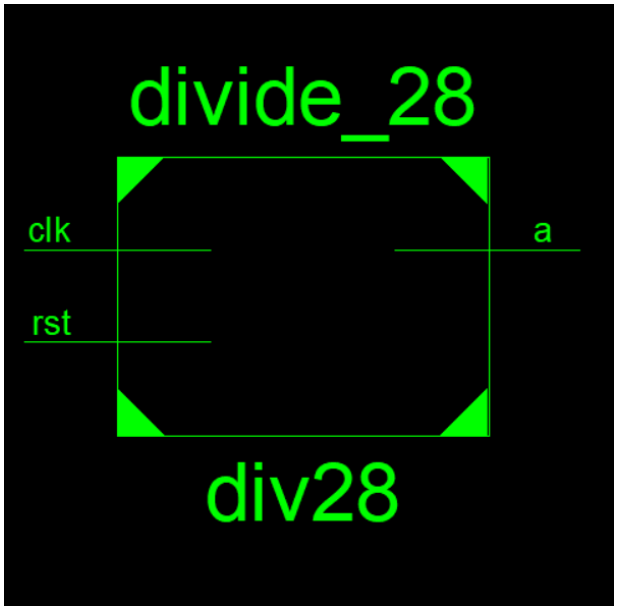


Simulation output of divide-by-2^n with the bottom-most waveform being the reset flag. Notice how each waveform takes 2 cycles to complete for each single cycle of the preceding (above) clock.

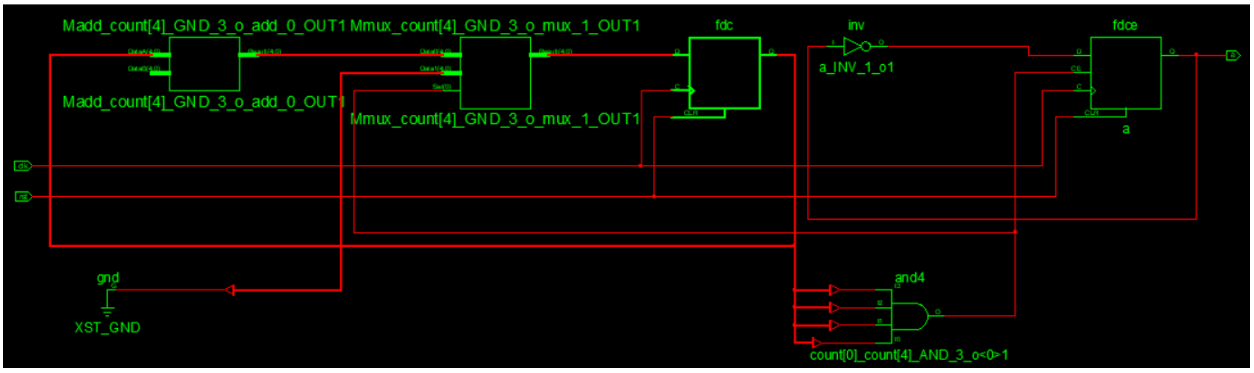
The 2^n clock divider was used to implement divide-by 2, 4, 8, 16, and 32 clock dividers. The divider was initially designed to divide-by 2, 4, 8, and 16 using a 4-bit counter. The counter value was assigned to a bus output and the clock_gen outputs for clk_div_2 – clk_div_16 were assigned to the first through fourth bits respectively. This worked to divide the clock by powers of 2 because the output would be incremented on every positive edge of the main clock which meant that the nth bit (starting from 1) completed a cycle every 2^n cycles of the main clock. This design was expanded to implement a divide-by-32 divider by expanding the counter to 5 bits and setting clk_div_32 to equal the most significant bit of the output. Further expansions of the counter would lead to divide-by-64, divide-by-128, or any larger divide-by-2^n divider.

```
// Basic 5-bit counter used to implement divide-by-2^n
module divide_2_n (
        input clk,
        input rst,
        output reg [4:0] a
        );
        always @ (posedge clk or posedge rst) begin
                if(rst)
                        a <= 5'b00000;
                else
                        a <= a +1'b1;
        end
endmodule
```

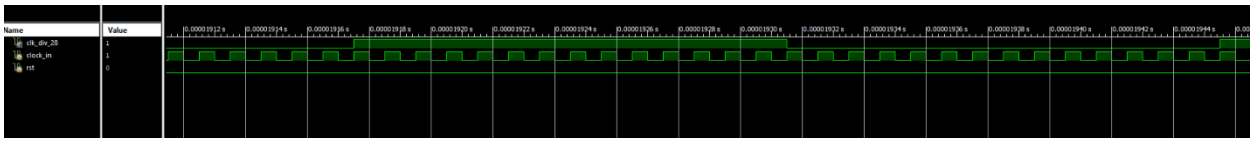
Even Division Clock Divider:



Black box of divide_28 listing inputs and outputs



Schematic of divide_28. Shows an adder, a mux, two flip flops and other primitive gates which are components of the module.

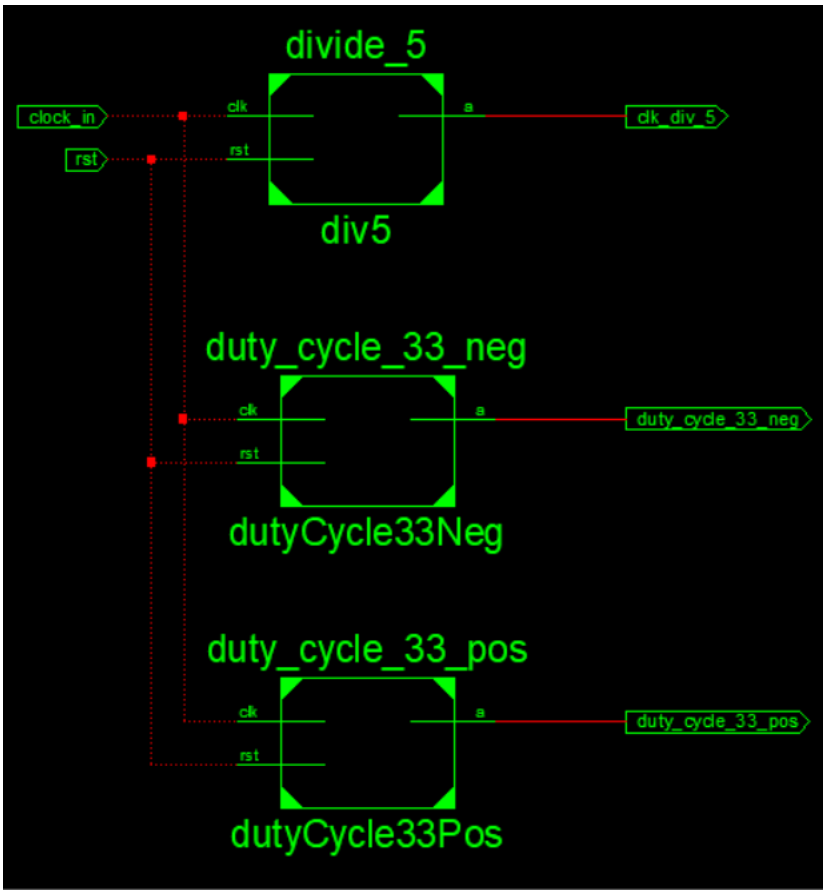


Waveform of divide_28

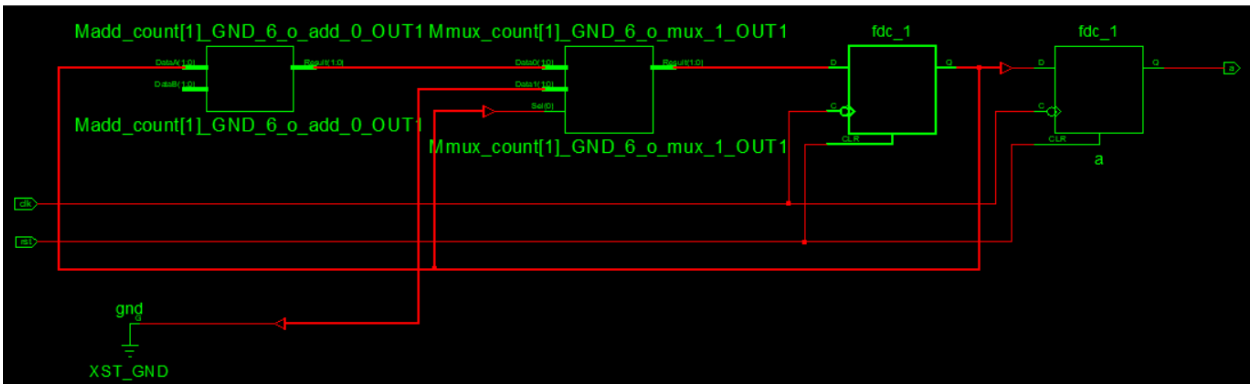
The even division clock divider logic was used to implement a divide-by-28 divider with a duty cycle of 50%. As can be seen from the waveform above, the clock is active for 14 cycles and off for 14 cycles of the main clock.

```
module divide_28 (
        input clk,
        input rst,
        output reg a
        );
        reg [3:0] count;
        always @ (posedge clk or posedge rst) begin
                if(rst) begin
                        count <= 4'b0000;
                        a <= 1'b0;
                end
                else begin
                        count <= count +1'b1;
                        if (count[0] && count[2] && count[3]) begin
                                a <= ~a;
                                count <= 4'b0000;
                        end
                end
        end
endmodule
```

Odd Division Clock Using Counters:

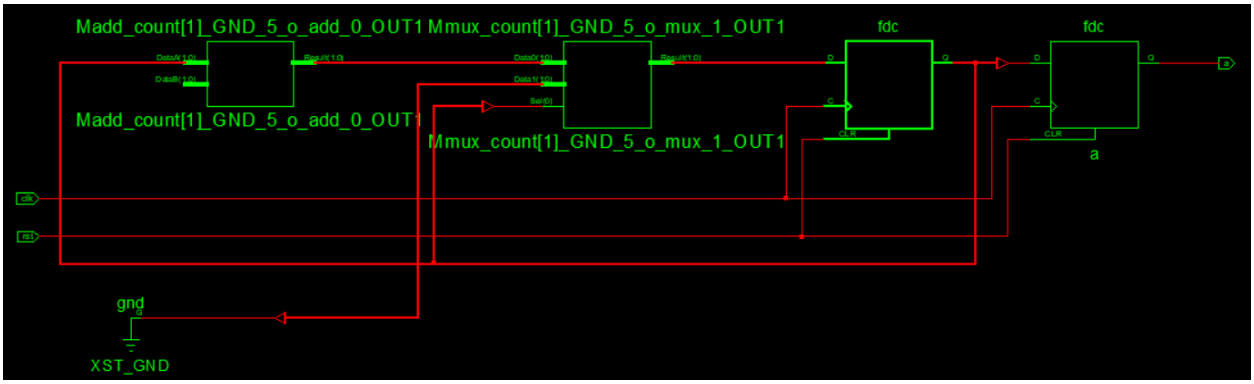


Black box of odd dividers listing inputs and outputs

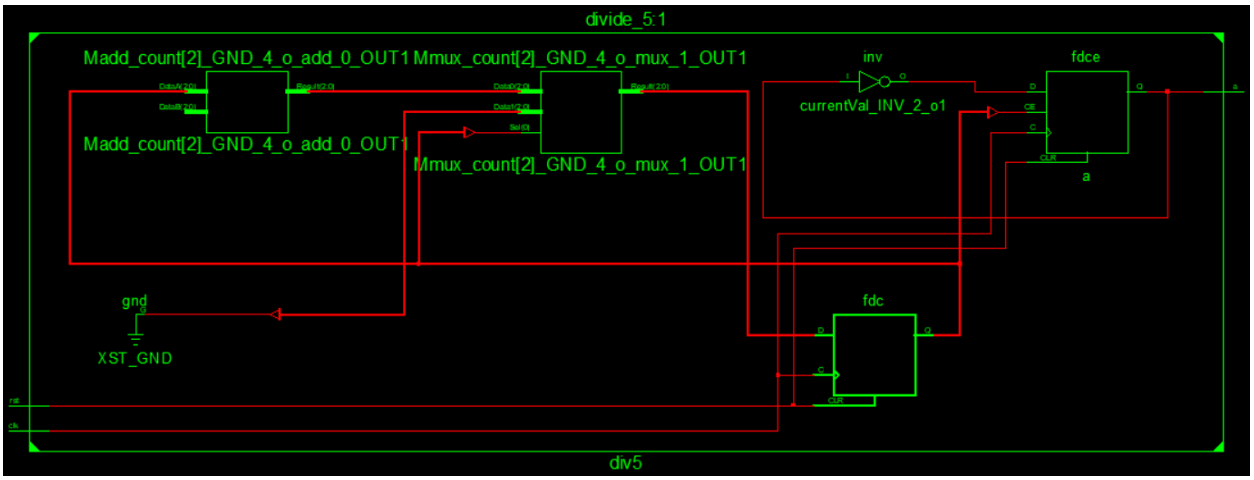


Schematic of divide-by-3 duty_cycle_33_neg. 2 multiplexers and 2 D flip-flops with async clear are used

---



Schematic of divide-by-3 duty_cycle_33_pos. The same components as the negative divide-by-3 divider are used; however, the change is triggered on the positive edge of the main clock.



Schematic of divide_5. An adder, a mux, a D flip-flop with async clear, and a D flip-flop with async clear and clock enable are used to build this module.

Waveforms for all three odd clock dividers.

The odd division logic was used to implement three dividers: two divide-by-3 33% duty cycle clocks triggered on opposite edges and a 50% duty cycle divide-by-5 clock. By taking the logical or of the two 33% duty cycle clocks, a 50% duty cycle divide-by-2 divider could be created. Because the duty cycles are different between the divide-by-3 and divide-by-5 clocks, the logical approach is different. The divide-by-3 clocks use a counter that counts to 2, the third possible state from 0, which turns on the clock for 1 cycle and then turns it off. Each of the two 33% duty cycle clocks is triggered by a different edge which makes them half a clock cycle out of sync. Because the logic only differs by which edge of the clock triggers the count, only the positive module's code is included for brevity.

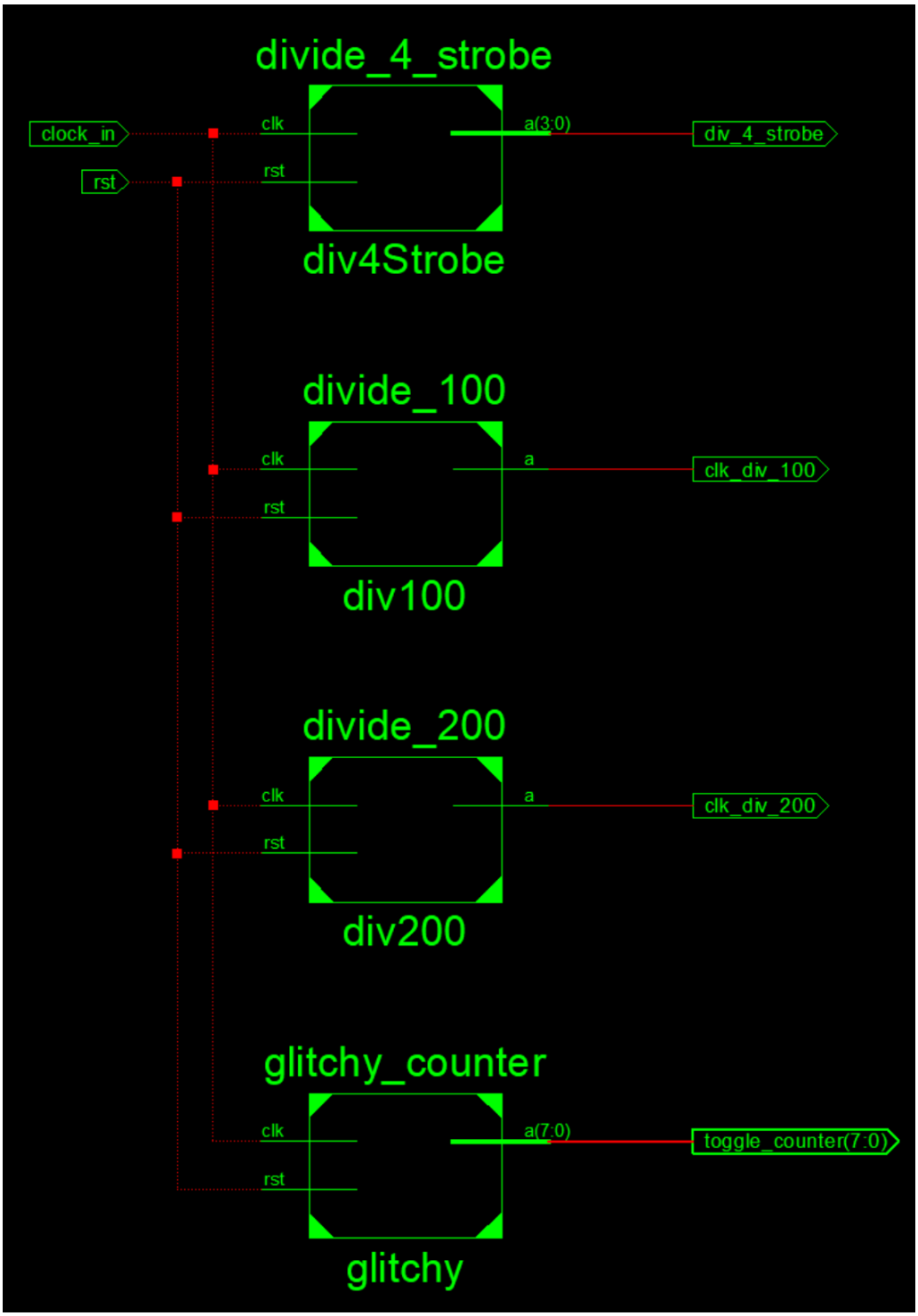
```
module duty_cycle_33_pos (
                input clk,
                input rst,
                output reg a
                );
                reg [1:0] count;
                reg currentVal;
                always @ (posedge clk or posedge rst) begin
                        if(rst) begin
                                a <= 1'b0;
                                count <= 2'b00;
                        end
                        else begin
                                count <= count + 1'b1;
                                if (count[1]) begin
                                        a <= 1;
                                        count <= 2'b00;
                                end
                                else begin
                                        a <= 0;
                                end
                        end
                end
endmodule
```

In addition to the 33% duty cycle divide-by-3 clocks described above, a 50% duty cycle divide-by-5 clock was also implemented. Similarly to before, the clock uses a counter to count up to a certain state, in this

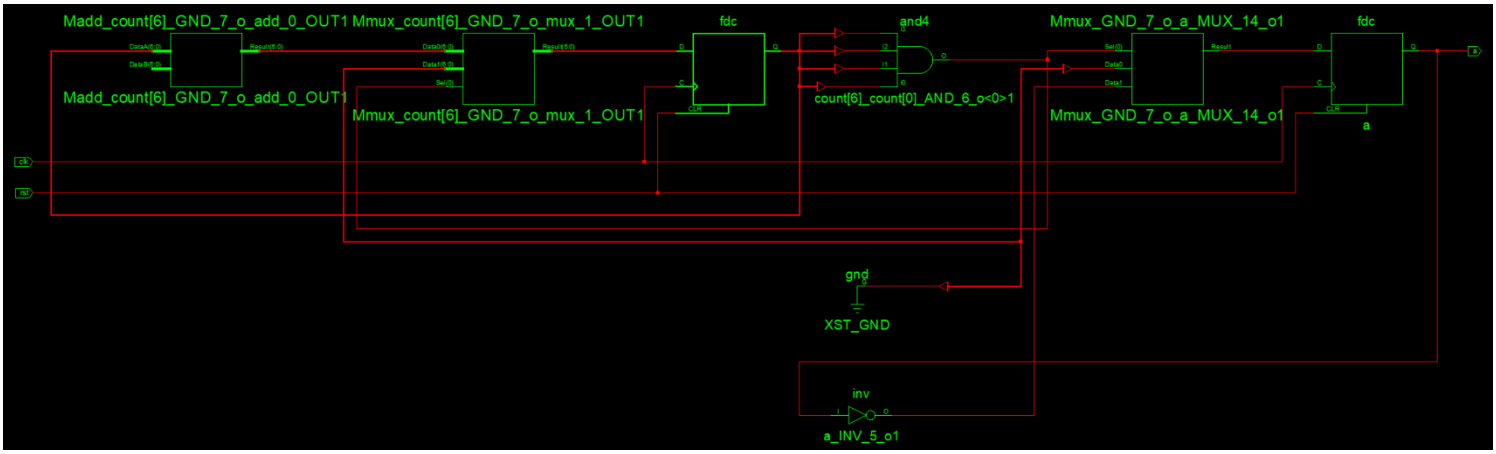
case the fifth state, and then resets back to 0 for another 5 state changes. These state changes are triggered on both the rising and falling edges to enable odd division. This results in a 50% duty cycle divide-by-5 clock divider.

```
module divide_5 (
        input clk,
        input rst,
        output reg a
        );
        reg [2:0] count;
        always @ (posedge clk or negedge clk or posedge rst) begin
                if(rst) begin
                        count <= 0;
                        a <= 0;
                end
                else begin
                        count <= count +1'b1;
                        if (count[2]) begin
                                a <= ~a;
                                count <= 0;
                        end
                end
        end
endmodule
```

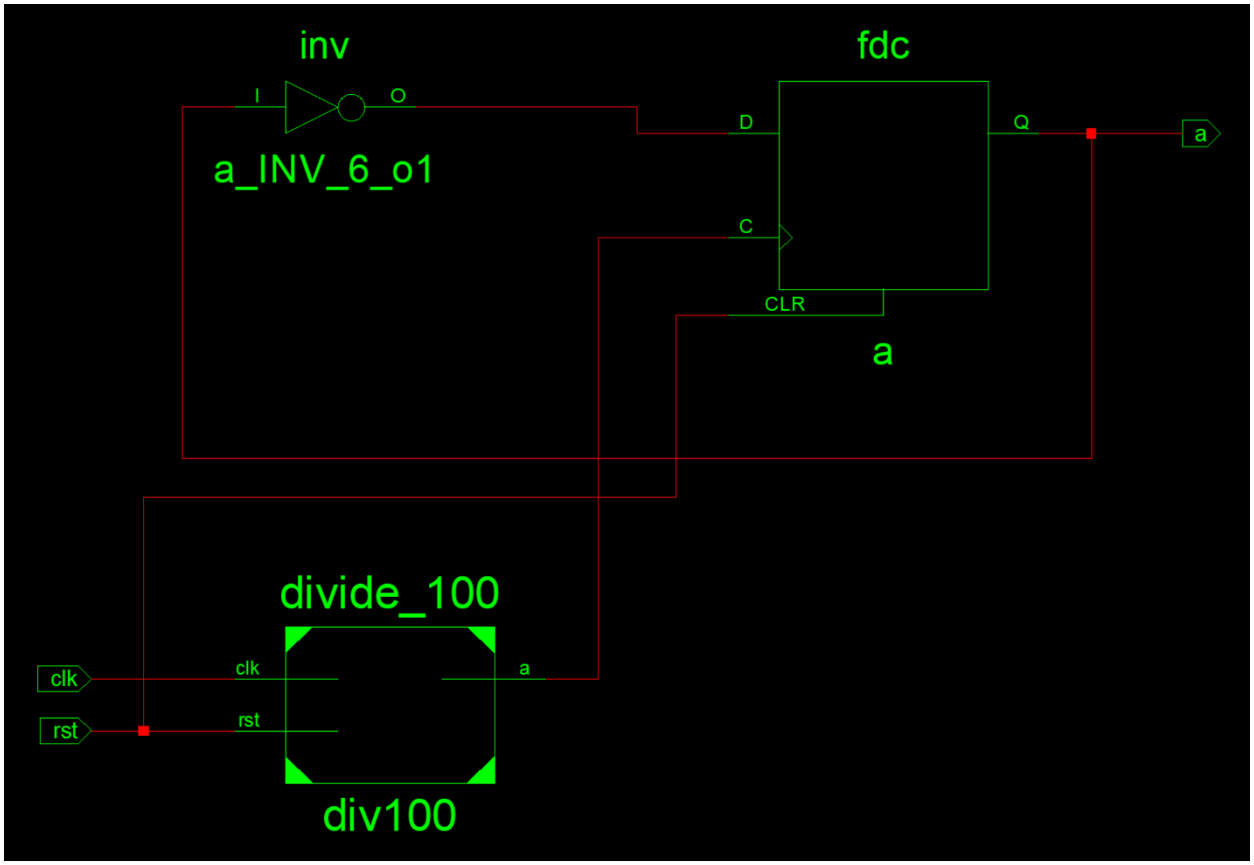
Pulse/Strobes:



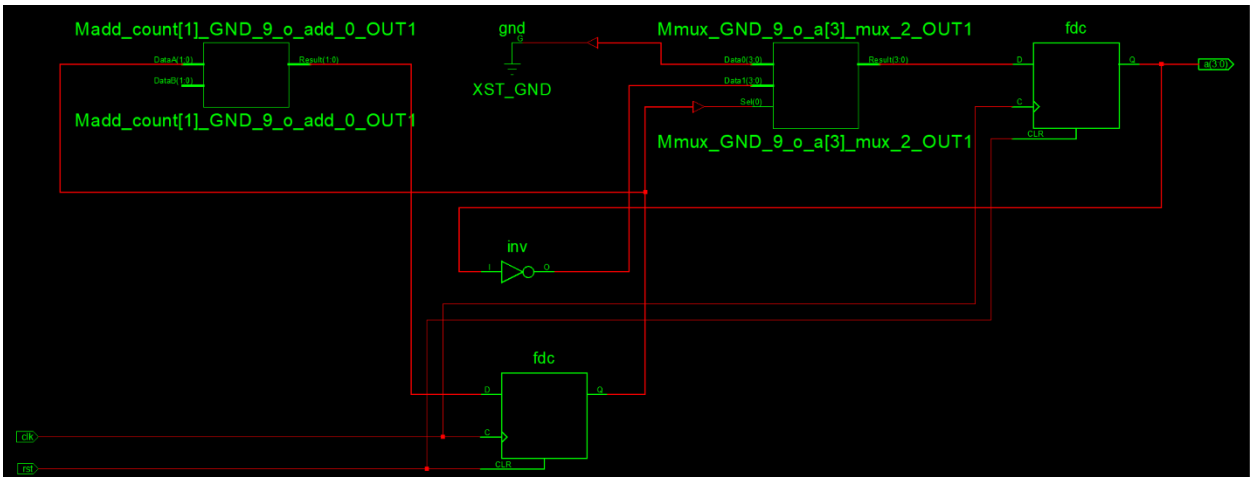
Black box schematics of all counters implemented using strobe.



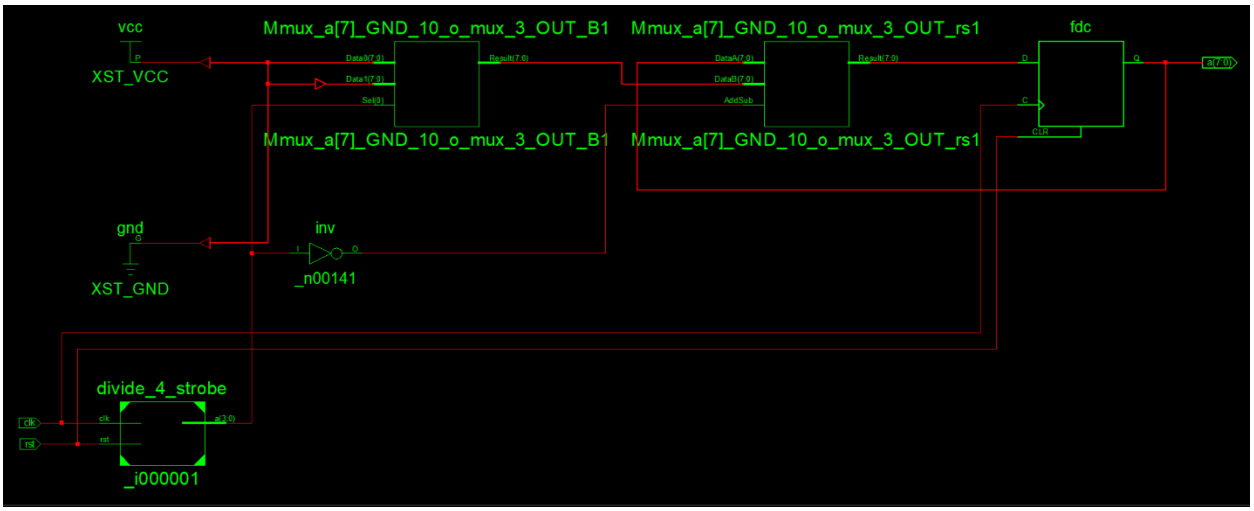
Schematic of divide_100 strobe. Three multiplexers and two D flip-flops with async clear are used to construct this module.



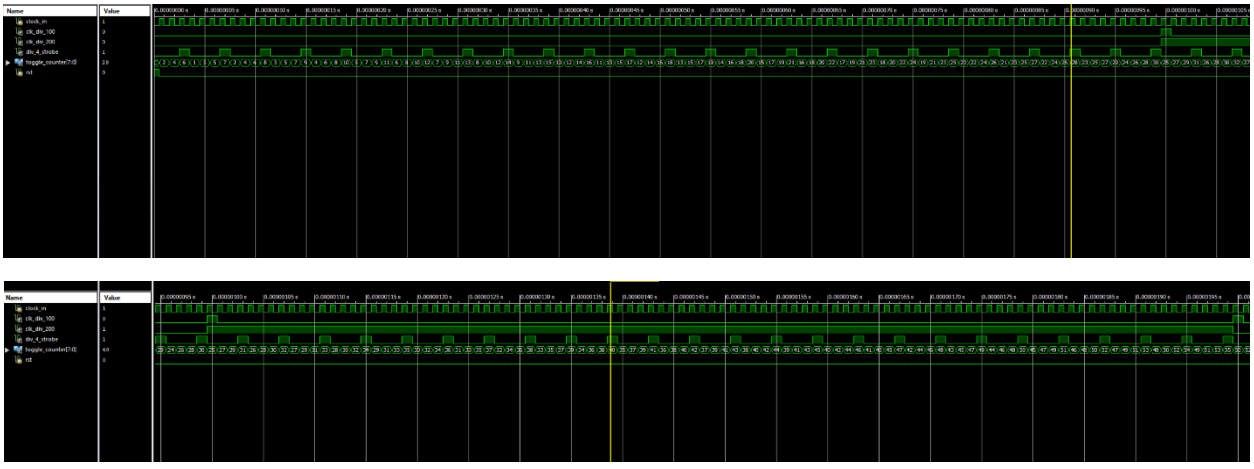
Schematic of divide_200. The divide-by-100 strobe is used as a clock for the D flip-flop with async clear.



Schematic of divide_4 strobe. Two multiplexers and two D flip-flops with async clear are used to construct this module.



Schematic of glitchy counter. The divide-by-4 strobe is used to trigger the necessary subtraction for the module



Wave forms for counters implemented using strobe logic

Strobe logic was used to design four dividers: a divide-by-100, divide-by-200, divide-by-4, and glitchy divider. The divide-by-100 strobe counts to 99, the hundredth state, and then sets the output clock, “a”, to equal 1 for exactly one clock cycle. This results in a divider that is on for one cycle and off for 99 cycles of the master clock. Using this module, a divide-by-200 divider was implemented using the even factor division logic. To implement this, the output was complemented each pulse of the divide-by-100 strobe. This resulted in a 50% duty cycle divide-by-200 clock operating at 500khz as 100mhz/200 = 500khz.

```
module divide_100 (
                input clk,
                input rst,
                output reg a
                );
                reg [6:0] count;
                always @ (posedge clk or posedge rst) begin
                        if(rst) begin
                                count <= 7'b0000;
                                a <= 1'b0;
                        end
                        else begin
                                count <= count +1'b1;
                                if (count[6] && count[5] && count[1] && count[0]) begin
                                        a <= ~a;
                                        count <= 7'b0000;
                                end
                                else begin
                                        a <= 0;
                                end
                        end
                end
endmodule
```

```
module divide_200 (
                input clk,
                input rst,
                output reg a
                );
                wire div100out;
                divide_100 div100(.clk(clk), .rst(rst), .a(div100out));
                always @ (posedge div100out or posedge rst) begin
                        if(rst) begin
                                a <= 1'b0;
                        end
                        else begin
                                a <= ~a;
                        end
                end
        endmodule
```

Similarly to the implementation of the divide-by-200 divider, a divide-by-4 strobe divider was used to implement a glitchy counter which increments by 2 each clock cycle except on the fourth cycle where it decrements by 5. *0→2→ 4→6→1→3→5→7→2→4→6→8→3→5→7→9→4→…..*

This was accomplished by nesting combining the 2^n divider logic with the 33% duty cycle logic where instead of counting to the third state, the counter sets the clock to 1 on the fourth state and then immediately sets the clock to 0 at the beginning of the following cycle. The divide-by-4 strobe was then nested inside of the glitchy counter and used to decrement by 5 each time it was active.

```
module divide_4_strobe (
                input clk,
                input rst,
                output reg [3:0] a
                );
                reg [1:0] count;
                always @ (posedge clk or posedge rst) begin
                        if(rst) begin
                                count <= 2'b00;
                                a <= 1'b0;
                        end
                        else begin
                                count <= count +1'b1;
                                if (count[1]) begin
                                        a <= ~a;
                                end
                                else begin
                                        a <= 0;
                                end
                        end
                end
endmodule
```

```
module glitchy_counter (
		input clk,
		input rst,
		output reg [7:0] a
		);
		wire div_4_strobe_out;
		divide_4_strobe(.clk(clk), .rst(rst), .a(div_4_strobe_out));
		always @ (posedge clk or posedge rst) begin
			if(rst)
				a <= 8'b00000;
			else
				if (div_4_strobe_out) begin
					a <= a - 5;
				end
				else begin
					a <= a + 2'b10;
				end
			end
endmodule
```

Test Bench

The test bench is a simple block of code that triggers a reset to initialize counters and then complements the current clock every 5 nanoseconds to create a 100mhz clock frequency. To verify the correct output, the number of cycles from the main clock can be compared to the specified factor for each sub-module.

```
initial begin
	// Initialize Inputs
	clock_in = 0;
	rst = 1;
	// Wait 100 ns for global reset to finish
	#5;
	rst = 0;
end
always begin
	#5 clock_in=~clock_in;
end
```

Post-Project Evaluation

My methodology for solving the given tasks involved the creation of individual clock dividers with the specified factors. I used additional outputs to verify each divider's functionality, including dividers whose outputs were not required by the assignment. Using this approach allowed me to visualize each step of my design and ensure that my logical design worked before applying it to additional specifications. This approach saved time by catching logical errors before they were copied to other parts of the code. For example, when designing the strobe, I mistakenly created a .00990099% duty cycle clock divider by incrementing my counter one time too many. By catching this error early, I was able to prevent the bug from being introduced into my divide-by-200 clock divider. Additionally, I was able to catch a mistake in the glitchy counter where the reset caused the divide-by-4 strobe to be off by one cycle, causing the count to reach 8 before decrementing by 5. This was fixed by adjusting the divide-by-4 strobe incrementation logic.

The project was of reasonable difficulty and was both practical and relevant to the class. To improve the project, I would change the description of the tasks and remove the design and verification categories. They added unnecessary confusion to the assignment and would be better as separate questions at the end. All things considered, I enjoyed designing the counters and felt that I gained valuable knowledge and experience relating to clock dividers.