

Sriram Balachandran (UID: 805167463)

Mohit Garg

CS M152A: Introductory Digital Design Laboratory

Section 2

November 8th 2020

Laboratory 2 Report

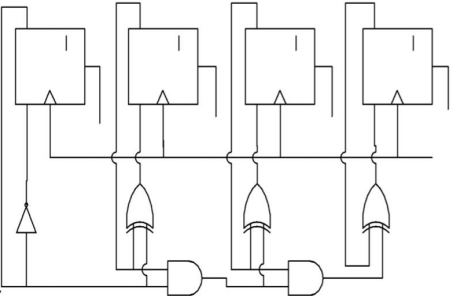
Introduction & Background

The purpose of this lab was to design and test various clock waveforms of different frequencies and duty cycles, and to show how they can be utilized. We were tasked with creating clocks that divide by powers of 2, even numbers, odd numbers, and finally creating a counter that utilized aforementioned techniques to follow a certain numeric sequence.

Clocking is crucial for synchronous data transmission, something that is extremely important as Internet of Things (IoT) devices grow increasingly commonplace. Clocks are also the basis of timer systems in a variety of embedded devices that dominate our daily lives, such as traffic lights, phones, and digital stopwatches. Additionally, clocks can also allow designers to debug designs by stepping through clock signals in a waveform simulation.

Design and Verification Requirements

Design Task 1: In order to create the power of 2 divider clock, we can simply implement a 4 bit counter, assigning each of the bits of the counter to one of the output wires. The least significant bit of the counter is a direct division of 2, and the preceding bit is a direct division of 4, and so on and so forth. The corresponding logic gate diagram for the counter is as follows.

	<pre> reg [3:0] a; always @ (posedge clk) if (rst) a <= 4'b0000; else a <= a + 1'b1; </pre>
<p>Figure 1: 4-bit Counter Logic Circuit</p>	<p>Code 1: 4-bit Counter Verilog Implementation</p>

Verification Task 2: From the previous example, we can see that in order to generate a clock that divides into some number 2^n , where n is an even integer, we need to toggle the output clock signal when the counter has counted n times. Since counting starts from 0, this would be when the counter's value is equal to $\text{binary}(n-1)$. Since we were tasked with creating a clock that divides by 32, we simply modify the code from the previous assignment in the following way. Instead of directly assigning the output clock to one of the bits in the counter, we assign the output clock to a separate output state register. Then, whenever we increment the counter, we check if the counter's value has reached our desired clock state toggle value. Using the formula mentioned earlier, this would be binary 15, 1111. If the toggle value has been reached, we reset the counter, then we “flip” the output state register's value by assigning it to the negation of its current value.

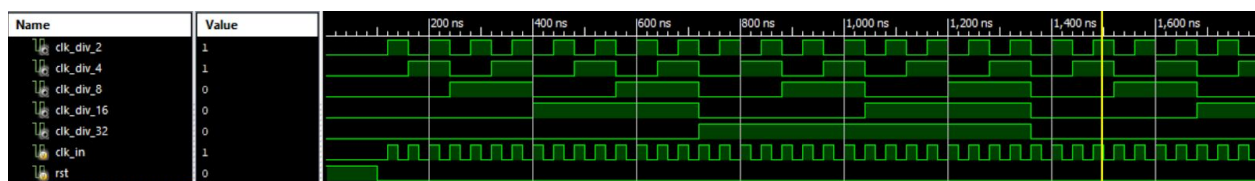


Figure 2: *clk_div_32* showcases a 50% duty cycle divide by 32 clock

```

reg [3:0] counter = 4'b0000;
// toggle after 32/2 positive edges, when counter = 16-1
reg [3:0] toggle_val = 4'b1111;

reg clk_out_state = 0;
assign clk_out = clk_out_state;

always @ (posedge clk_in or posedge rst) begin
    if (rst) begin
        counter <= 4'b0000;
        clk_out_state <= 0;
    end else begin
        counter <= counter + 1;
        if (counter == toggle_val) begin
            counter <= 4'b0000;
            clk_out_state <= ~clk_out_state;
        end
    end
end
end

```

Code 2: Implementation of 50% duty cycle divide by 32 clock

Design Task 3: Using the same logic from the previous verification task, we simply take the code from the divide by 32 clock, but set the toggle value to binary($28/2 - 1$) = binary(13). The waveform for this and all design tasks will be showcased at the end of this section.

Verification Task 4: We are tasked with creating a 33% duty cycle clock. This means that it should remain high for $\frac{1}{3}$ of its clock period. The way we achieve this is by first picking some integer n such that $2n\%3 == 0$, that is, double n is divisible by 3. Then we modify our code from verification task 2. Now, what we previously called our toggle value will be renamed to reset value, as it is the value when our actual counter will rest. We will then introduce a toggle value with a new meaning: if the counter is ever equal to this toggle value, we simply toggle the output clock state without resetting the counter. As for the value of this toggle value, it can be determined through a bit of logic. We'd like for the clock we're creating to be high for 33% of the period. We know that when the reset value is reached, the output state will be toggled. Therefore we should aim to toggle the output state $n/3$ cycles before the reset value. Therefore, to generate a 33% duty cycle clock, we would want the toggle value to be (reset value - $n/3$). In order to generate this particular 33% duty cycle clock, I picked the value 3 for n .

In order to generate a $1/d$ duty cycle clock, the following values are needed:
Reset Value: nk , where both n and k are integers
Toggle Value: $nk - nk/d$

Table 1: Formula to calculate reset value and toggle value for an arbitrary duty cycle clock

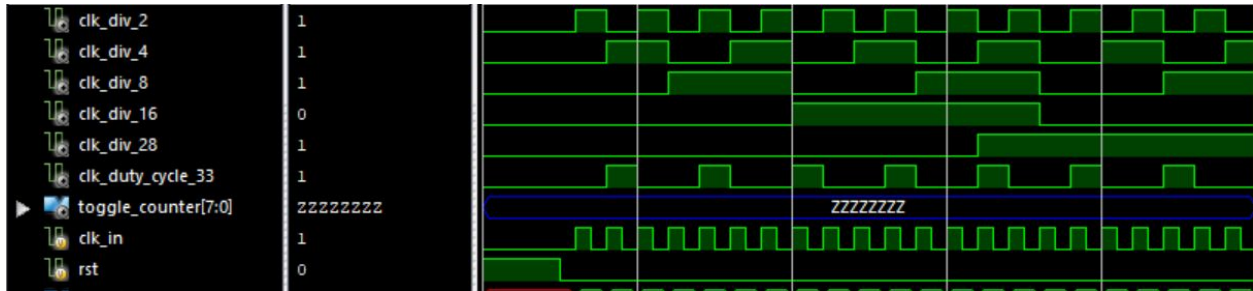


Figure 3: `clk_duty_cycle_33` showcases a 33% duty cycle clock

```

reg [1:0] reset_val = 2'b10;
reg [1:0] toggle_val = reset_val - (reset_val/3);

reg clk_out_state = 0;
assign clk_out = clk_out_state;

always @ (posedge clk_in or posedge rst) begin
    if (rst) begin
        counter <= 2'b00;
        clk_out_state <= 0;
    end else begin
        counter <= counter + 1;
        if (counter == toggle_val) begin
            clk_out_state = ~clk_out_state;
        end else if (counter == reset_val) begin
            clk_out_state = ~clk_out_state;
            counter <= 2'b00;
        end
    end
end
end

```

Code 3: Implementation of a 33% duty cycle clock

Verification Task 5: Duplicate code and logic to verification task 4, simply move the code such that it is implemented within an always block that is triggered on the falling edge of the clock input.

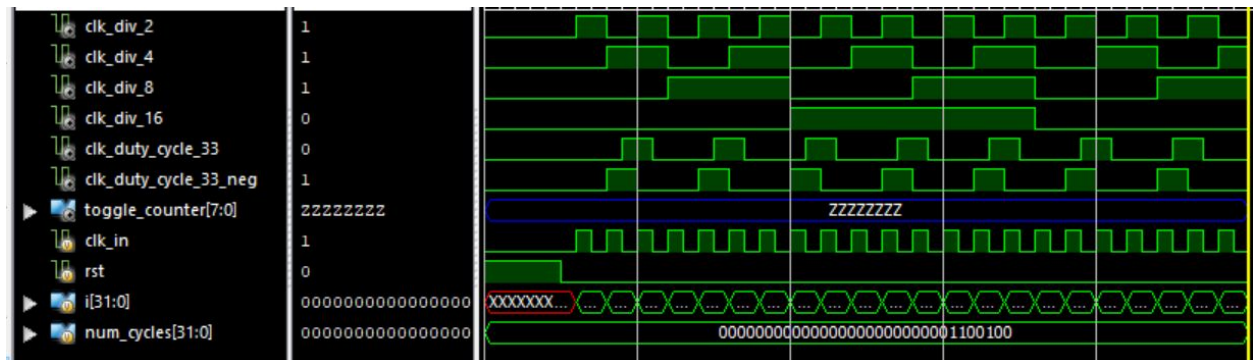


Figure 4: `clk_duty_cycle_33_neg` showcases a 33% duty cycle clock triggered on the falling edge of the clock signal, as opposed to the rising edge like `clk_duty_cycle_33`.

Verification Task 6: The implementation of this task is fairly trivial. However the conclusions are much more powerful. What we can see is that we can generate a 50% duty divide by 3 clock by taking the logical or of 2 33% duty cycle clocks that are offset by half a clock cycle. This

principle can be abstracted to create any 50% duty divide by odd number clock using the following principle:

In order to generate a 50% duty cycle divide by $2n+1$ clock, where n is a positive integer:
Create two $(n/2n+1)$ duty cycle clocks, one that counts on the rising clock edge, and one that counts on the falling clock edge.
50% duty divide by $2n+1$ clock = Logical OR of both $(n/2n+1)$ duty cycle clocks

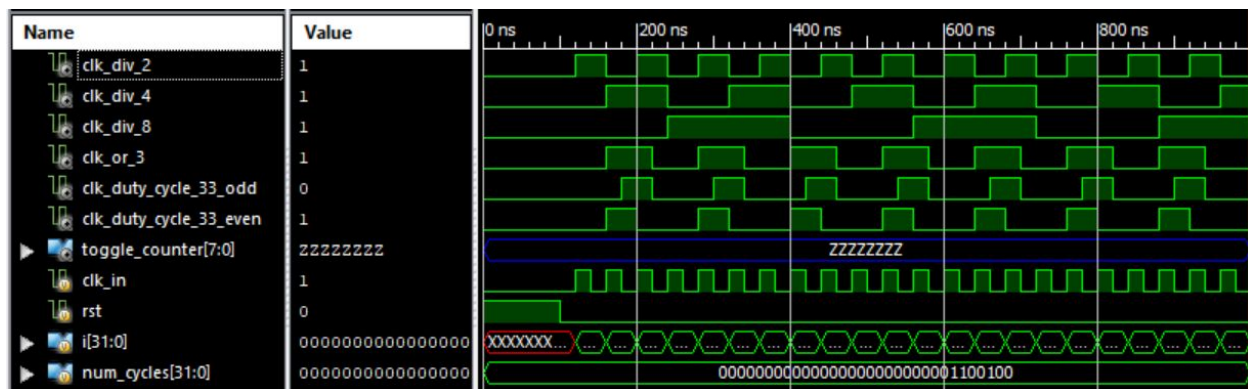
Table 2: Procedure to create any 50% duty cycle divide by odd number clock

Figure 5: *clk* or 33 showcases a 50% duty cycle divide by 3 clock

Design Task 7: Utilizing the conclusions from verification task 6, we create a submodule that runs two % duty cycle clocks, one on the rising clock edge, and another on the falling clock edge. Then we simply assign the clock output to the logical OR of the two % duty cycle clocks. The waveform for this and all design tasks will be showcased at the end of this section.

Verification Task 8: The implementation of this task is relatively simple. Utilizing the previous counter-clock model configured with a reset value and a toggle value, we simply take the reset value to be `binary(99)`, which will translate to 100 cycles. Since we're attempting to create a 1% duty cycle clock, utilizing the formula in Table 1, letting $n = 100$ and $d = 100$, we get a toggle value of `binary(98)`. However, we don't directly assign this 1% duty cycle clock state to the output clock. We then add a second always block where we toggle the output clock state value whenever the 1% duty cycle clock state is high.

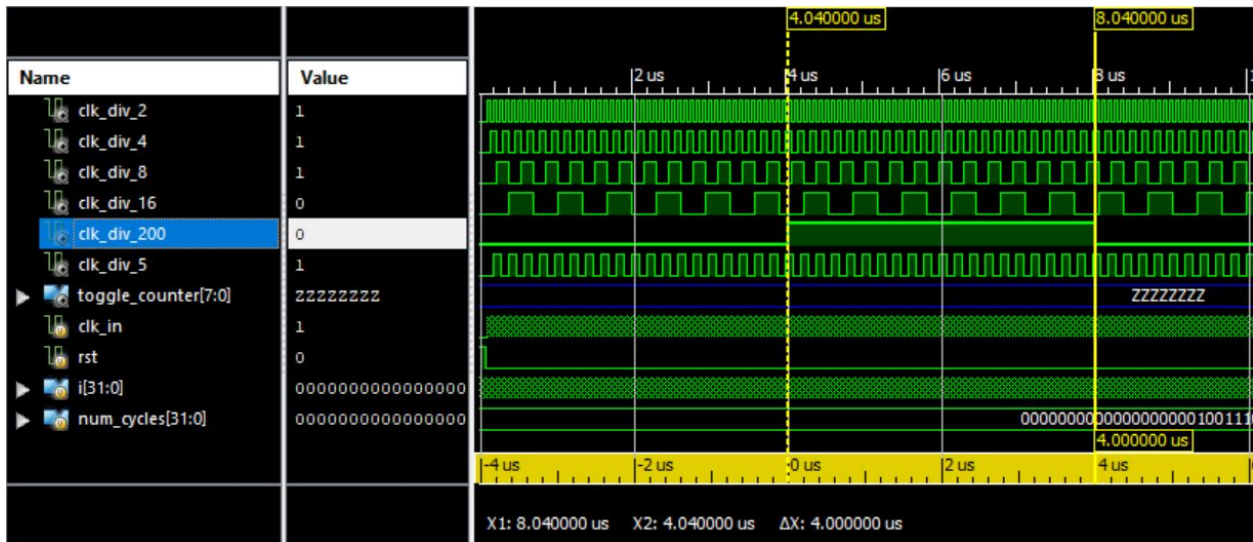


Figure 6: *clk_div_200* showcases a 50% duty cycle divide by 200 clock. The system clock, *clk_in*, has a period of 40ns. $40\text{ns}/\text{cycle} \times 100\text{cycle} = 4\mu\text{s}$.

Design Task 9: Utilizing a similar ideology to verification task 8, we create a 25% duty cycle clock, and then have the counter incrementation take place in a separate always block. If the 25% duty cycle clock signal is high, then we decrement the register assigned to the output counter by 5, and otherwise we increment by 2. Below is the verification for all 4 design tasks.

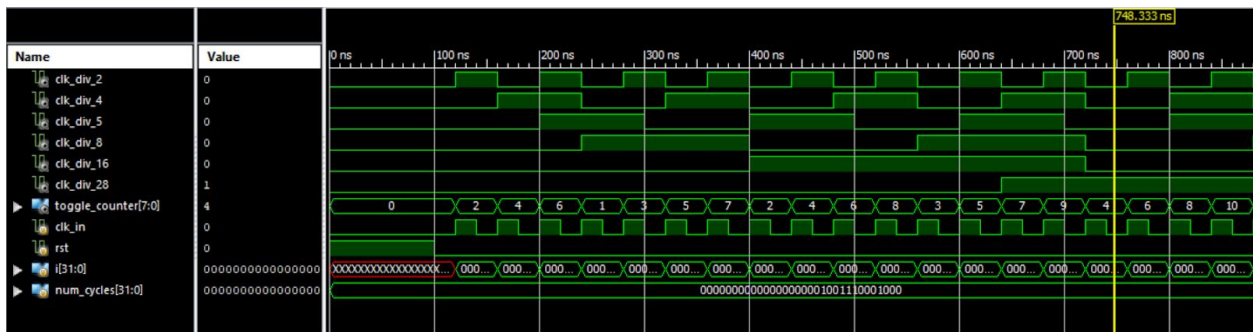


Figure 7: Final design task verification.

Design Description

The implementation details were discussed in the previous section, but as for the final structure of the code, it is as follows. There is a top level *clock_gen* module, which handles the implementation of the divide by 2^n clocks. The implementation of the divide by 28 is handled in a submodule, since it would be difficult to read the code if it was implemented directly alongside the divide by 2^n clocks. For similar reasons, the implementation of the divide by 5 is handled in a submodule for similar reasons, with code blocks being run in always blocks on the rising and falling edges. However, since the code inside these blocks is almost identical, it could be further

abstracted to another submodule. This would be a point of improvement for future iterations of this project. The glitchy counter is also implemented in a submodule, for the same reasons as the abstraction of the divide by 28 submodule.

Simulation Documentation

My testbench utilized a for loop which would toggle the clock input signal, wait for 20ns, then toggle the clock input signal again. Each run of the loop simulated one full clock cycle, and by altering the stopping condition of the for loop I was able to easily run my simulation for more cycles. By looking at the loop counter variable, it was also easy to verify whether or not the clock signals were running as expected. However, with some of the larger non- 2^n clocks, it was a little more difficult to verify by looking at the waveforms and the loop counter variables alone, and I actually had an off by 1 error in my verification task 8, which I almost didn't catch. What helped there was utilizing the Xilinx Waveform Simulation window to look at the timestamps of when clock signals went high and low, and then factoring in the period of the system clock to figure out how many cycles had passed.

Initially I also faced some troubles synthesizing my code. This mainly came from my `clk_div_5` module. I had two always blocks, one for a rising edge clock and another for a falling edge clock. However, I tried to only handle the reset signal in one of the always blocks, where I reset the counters for both the rising and falling clocks. This was not synthesizable, and I realized that variables used in an always block should only be modified in that always block itself, or else the code would not be synthesizable.

Conclusion

I was able to successfully implement all of the design and verification tasks, and by progressing through the lab in order of the design/verification tasks, I was able to come to various conclusions, such as those in Table 1 and Table 2, that I was able to leverage in subsequent design tasks. The structure of this laboratory is very student/learner friendly, and naturally guides one towards conclusions that can be built upon to complete the more complex design tasks that might have otherwise been difficult to finish. At this point in time, I do not have any further improvements to suggest for this lab.