Stephanie Doan (604981556)
Rohan Surve
CS M152A Lab 6
24 January 2021

Lab 2: Clock Design Methodology

I. Introduction

In this lab, I explore sequential circuits through exercises focusing on counters and clocks based on the system clock. Each of the four clocks are implemented a distinct sub-module called by a top-level module `clock_gen`, which takes the input clock and reset wires and outputs all the registers of the clocks controlled by the four submodules that accomplish the following:

- Divide by 2^n clock, counting by 2, 4, 8, and 16
- Even division clock by 28
- Odd division clock by 5
- Glitchy counter controlled by a strobe counter

II. Design Tasks

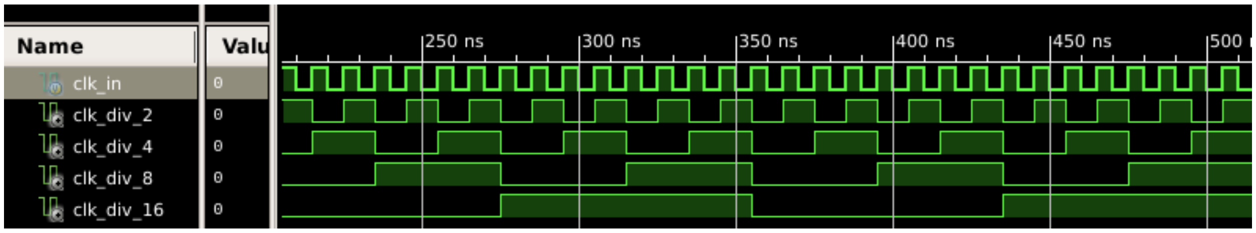
1. **Design Task:** [divide by 2^n clock] Assign 4 1-bit wire to each bit from 4-bit counter

In the `clock_div_two` module, I implemented the 4-bit counter as a 4-bit register, with the least significant bit corresponding to a divide-by-2 clock, all the way to the most significant bit corresponding to a divide-by-16 clock. On every positive clock edge, 1 is aded to this counter, which resets to `4'b0000` whenever the counter reaches its maximum value. This is essentially a divide by 2^n clock because it acts only on every positive clock edge, taking a whole input clock cycle to flip the least significant bit, then twice as long to flip the second least significant bit, and onwards. On every change to this register, the value of each bit is extracted and stored in output registers `clk_div_2`, `clk_div_4`, `clk_div_8`, and, `clk_div_16`. The figure below shows this 4-bit counter as well as the output register each bit corresponds to.

count[3]	**count[2]**	**count[1]**	**count[0]**
Divide by 16 clock Flips bit at 1/16 frequency of clk_in `clk_div_16`	Divide by 8 clock Flips bit at 1/8 frequency of clk_in `clk_div_8`	Divide by 4 clock Flips bit at 1/4 frequency of clk_in `clk_div_4`	Divide by 2 clock Flips bit at 1/2 frequency of clk_in `clk_div_2`

Four-bit register reg[3:0] count as divide by 2^n clock

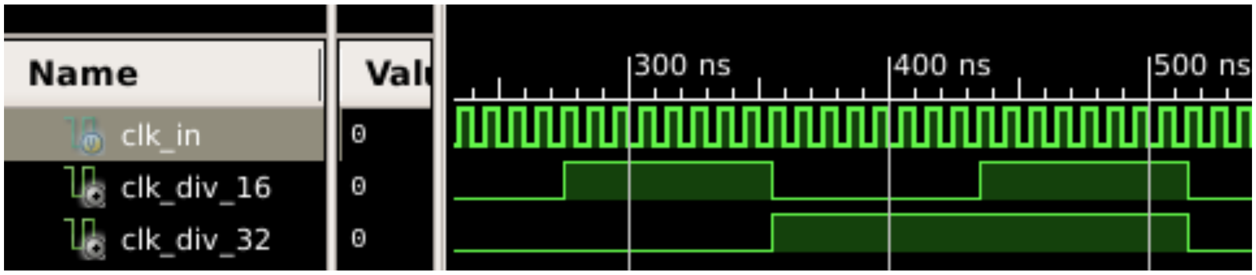
To verify functionality of this 4-bit counter, I visualized the waveforms below showing the values of each output register corresponding to the bits of the counter, as well as the `clk_in` (clock in). Notice that changes to each output register happens only at the positive edges of `clk_in`.



Waveforms of `clock_div_two` module

2. **Verification Task:** [even division clock] Divide by 32 clocks by flipping output clock on every counter overflow

In order to create a divide-by-32 clock, I extend the 4-bit counter in the last task to a 5-bit counter, so that when the least significant 4 bits overflow, the most significant bit flips, which is essentially dividing by 32. Similar to the implementation of the first task, every change to this counter (via an always block) had `clk_div_32` and `clk_div_16` read and store the first and second most significant bits respectively. The waveforms of this are visualized below, with `clk_div_32` flipping at half the frequency of the `clk_div_16` register as expected.



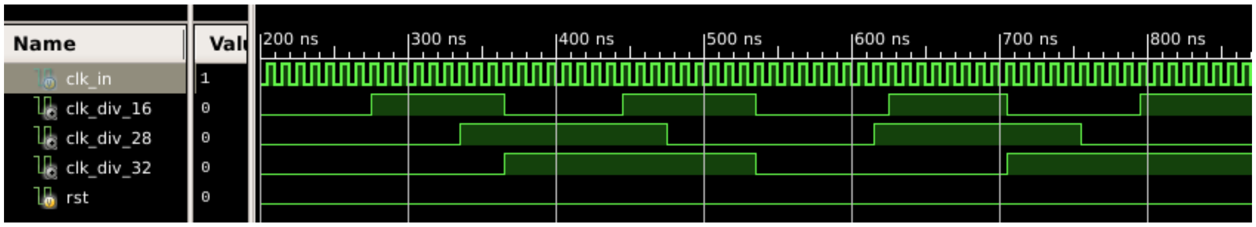
Waveforms showing functionality of divide-by-32 clock

3. **Design Task:** [even division clock] Generate clock 28 times smaller by modifying when counter resets to 0

To extend my module written for the previous task, I created a new 4-bit counter that would reset to 0 whenever it held the value `4’b1101` = 13 in decimal. This is essentially resetting every 14 positive clock edges, equivalent to 28 total clock edges. I introduced more conditional logic to my code, explicitly checking the counter value to reset to 0 and flip the `clk_div_28` register.

```
else if (count_28 == 4’b1101)
begin
    count_28 <= 4'b0000;
    clk_div_28 <= ~clk_div_28;
end
```

As seen in the waveform below, `clk_div_8` flips at a frequency in between `clk_div_16` and `clk_div_28` as expected.



Divide-by-28 clock in comparison to divide-by-16 and divide-by-32

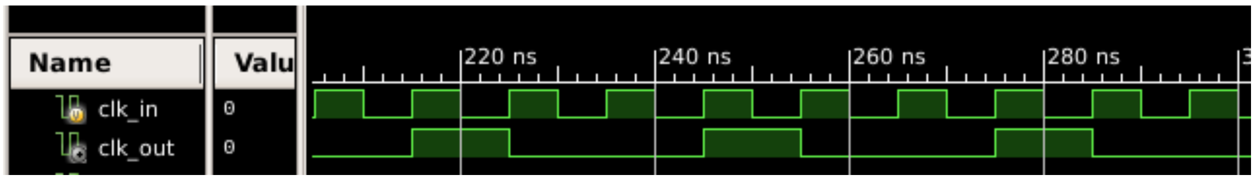
4. Verification Task: [odd division clock] Verify waveform of 33% duty cycle clock

Here I create a clock whose signal is active for 1/3 of the time. Using if statements and counters as specified, I create a 2-bit counter that is incremented at every positive clock edge (though this could also be any edge), and checks if the value held is `2'b10` = 2 in decimal, essentially resetting to 0 every 3 positive clock edges. Every change to this counter makes the output register `clk_out` read from the most significant bit of the counter, which will read 1 exactly 1/3 of the time, since the counter cycles between holding the values `2'b00`, `2'b01`, and `2'b10`.

```
module clock_33_duty(
    input wire clk_in,
    input wire rst,
    output reg clk_out
);
    reg[1:0] count = 2'b00;
    always @ (posedge clk_in)
    begin
        if (rst)
            count <= 2'b00;
        else if (count == 2'b10)
            count <= 2'b00;
        else
            count <= count + 1'b1;
    end

    always @ (count)
        clk_out <= count[1];
endmodule
```

The waveform is shown below with clk_out active for 33% of the time.



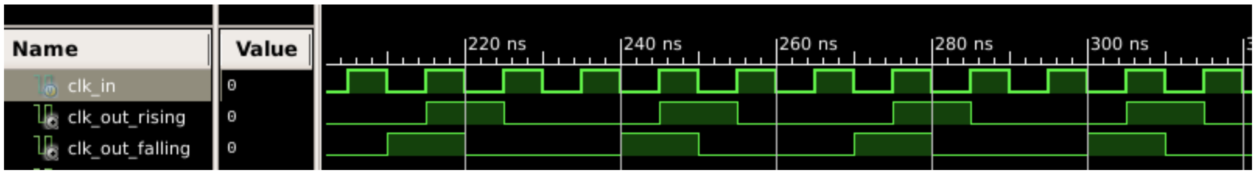
Waveform of 33% duty cycle clock

5. Verification Task: [odd division clock] Rising edge vs falling edge triggered clock

In order to have the output signal change on the rising edge of the input clock, I essentially duplicated my code from the previous task only replacing the always block's sensitivity list

```
always @ (negddge clk_in)
```

so that updates to the counter happens on the rising edge instead. Below are the side by side waveforms of the clock from Task 4 (`clk_out_rising`) and the new one (`clk_out_falling`).



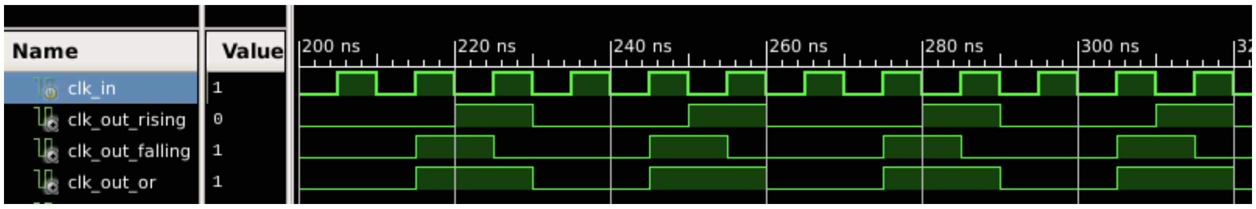
33% duty cycle clock: rising edge vs falling edge

6. Verification Task: [odd division clock] Assign wire that takes logical OR of two 33% clocks

In order to take the logical OR of the two clocks from the pervious tasks, I added a line to the always block triggered by changes to the counters, to assign to output clk_out_or the logical or of the values given to clk_out_rising and clk_out_falling.

```
clk_out_rising <= count_rising[1];
clk_out_falling <= count_falling[1];
clk_out_or <= count_rising[1] | count_falling[1];
```

The waveforms are shown below, with `clk_out_or` active wherever `clk_out_rising` or `clk_out_falling` is active.



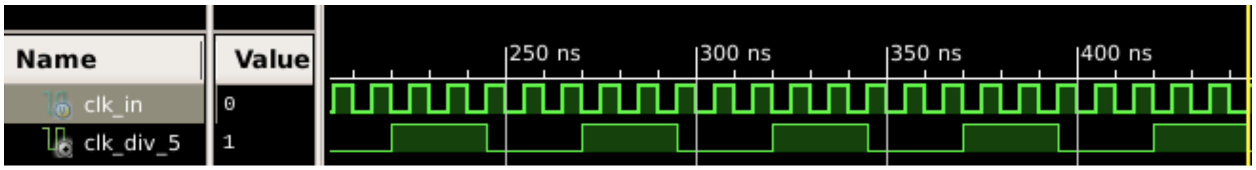
`clk_out_or` as logical or of two 33% clocks

7. **Design Task:** [odd division clock] 50% duty cycle divide-by-5 clock

To implement a divide-by-5 clock, I modify the code written for task 4 to have the always block's sensitivity list include both the positive and negative edge, as the counter needs to be incremented on every edge, as we are now counting by 1s, not 2s. I declare a 3-bit counter `reg[2:0]` to hold 3 bits that count up to `3'b100` = 4 in decimal before being reset to 0, essentially counting 5 clock edges.

```
reg[2:0] count = 3'b000;
always @ (posedge clk_in or negedge clk_in)
begin
     if (rst)
     begin
          count <= 3'b000;
          clk_div_5 <= 1'b0;
     end
     else if (count == 3'b100)
     begin
          count <= 3'b000;
          clk_div_5 <= ~clk_div_5;
     end
     else
          count <= count + 1'b1;
end
```

The waveforms are shown below, with `clk_div_5` being flipped every 2.5 clock cycles of `clk_in`. The clock is indeed 50% duty as the output clock signal spends exactly half its time active.

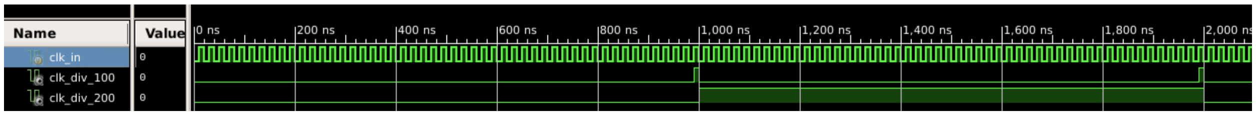


Divide-by-5 clock

8. **Verification Task:** [glitchy counter] Divide-by-100 clock with 1% duty cycle, 50% duty divide-by-200 clock

To implement the divide-by-100 clock, I initialized a 7-bit counter to store up to `0b'1100011` = 99 in decimal, as we would like to count up to 100 clock edges. In an always block that runs on the system clock's positive and negative clock edges, the counter is incremented by 1 until the value held is 99, in which the output `clk_div_100`, originally initialized to 0, is flipped to 1, then flipped back to 0 in the very next clock edge. As shown in the waveforms below, this clock has 1% duty cycle because it spends 10 ns out of 1000 ns active.

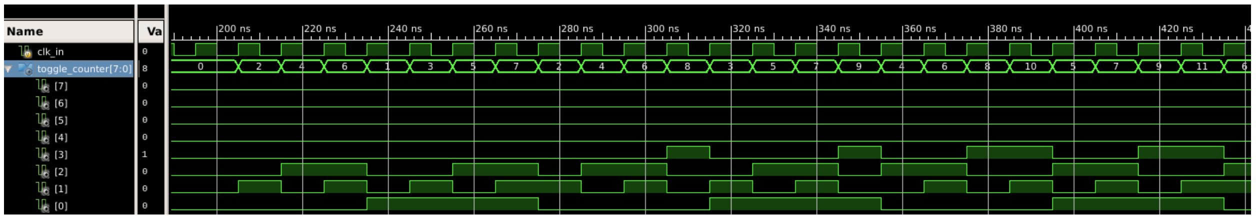
To implement the second clock, I created a second always block also running on the positive and negative edges of the system clock, flipping its value whenever `clk_div_100` is 1 via an if statement. As seen in the waveform below, this output clock, `clk_div_200`, is 50% duty cycle because it spends 1000 ns of 2000 ns active. The clock is at 500KHz because each cycle of it takes 200 clock edges = 100 clock cycles. Since my test bench sets clk_in to be flipped every 10 ns, clk_in has a 20 ns clock cycle. Therefore, 100 cycles * (20 ns / cycle) = 2000 ns per clock cycle, and 1 cycle / 2000 ns = 500000 Hz = 500 KHz.



Waveforms of divide-by-100 and divide-by-200 clocks

9. Design Task: [glitchy counter] 8-bit counter that goes up by 2 on every positive edge and subtracts 5 by every strobe of divide-by-4 strobe

To implement this counter, I initialized a 2-bit register `reg[2:0]` strobe to count up to `2'b11` = 3 in decimal for 4 strobes. In an always block that acts on every positive edge of the system clock, I add `2'b10` to toggle_counter, effectively counting up by 2, and add `1'b1` to strobe. If strobe holds `2'b11`, then `3'b101` = 5 in decimal is subtracted from toggle_counter and strobe is reset to 0. As shown in the waveform below, the sequence of values held by the counter is 0→2→ 4→6→1→3→5→7→2→4→6→8→3→5→7→9→4→… as expected.



Waveform for `toggle_counter`

III. Clock Generator Module

With the functionalities and lessons learned from these tasks, I wrote the top level module `clock_gen` to call each of the four submodules, taking `clk_in` and `rst` as inputs set by the written testbench module that is later explained. These are passed to the submodules along with output registers specific to each.

- `clock_div_two`

This submodule has an always block running on the positive edge of `clk_in`, adding 1 to the 4-bit register that is the main counter essentially incremented every 2 edges, or 1 clock cycle. In another always block running on any changes to this counter, the counter's bits are written to the 4 output registers representing the divide-by-[2 | 4 | 8 | 16] clock.

- `clock_div_twenty_eight`

This submodule has an always block also running on the positive edge of of `clk_in`, adding 1 to the 4-bit counter, which is reset to 0 when it holds 13, equivalently counting 14 clock cycles or 28 clock edges. Each reset is accompanied by the flipping of the register `clk_div_28`.

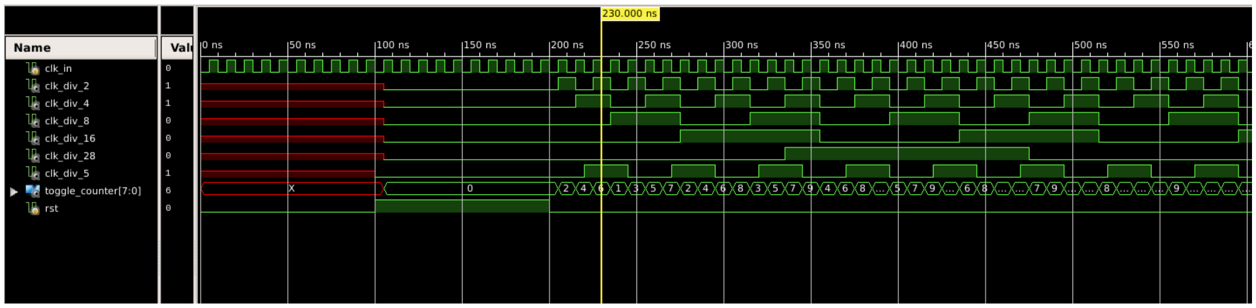
- `clock_div_five`

This submodule has an always block running on both positive and negative edges of `clk_in`, adding 1 to the 3-bit register that is reset to 0 when it holds 4, equivalently counting 5 clock edges as we are now counting by 1 instead of 2s. Output `clk_div_5` is also flipped on each reset.

- `clock_strobe`

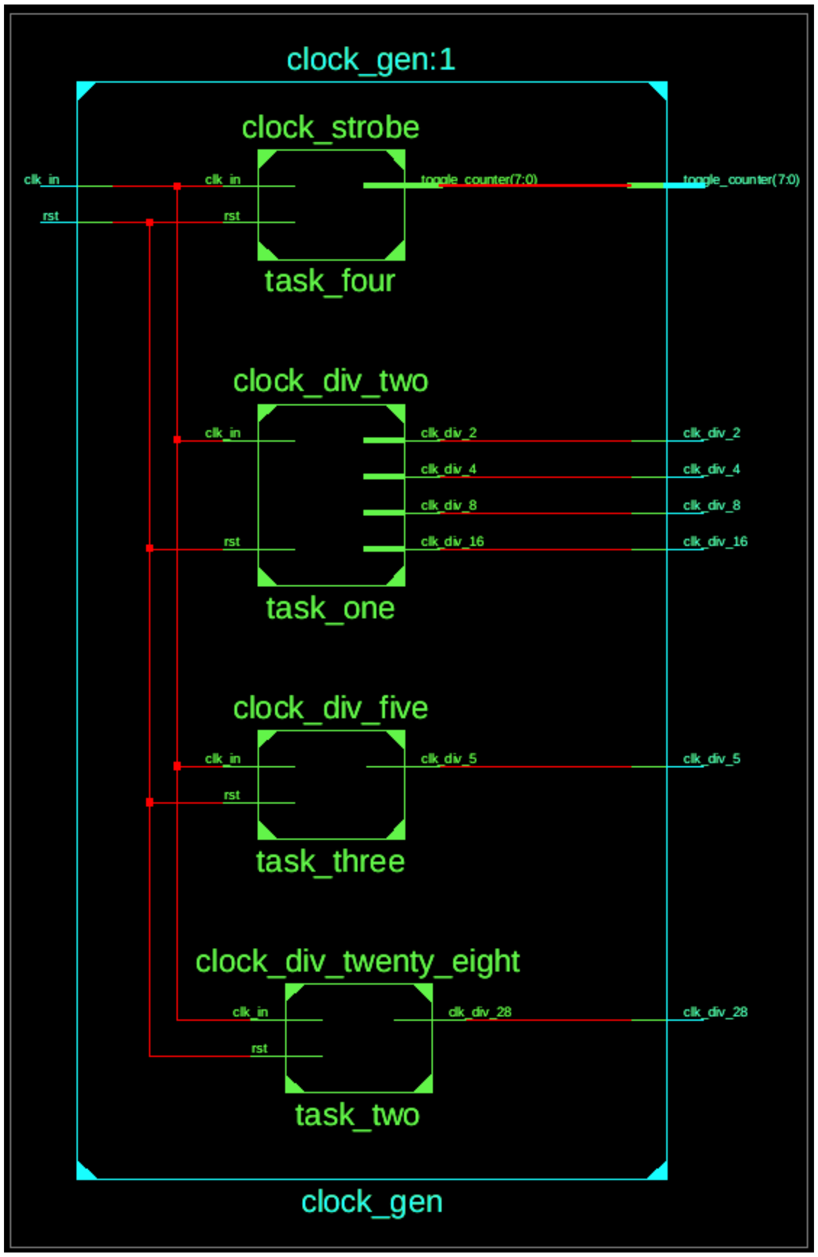
This submodule has an always block running on positive clock edges, adding 2 directly to the `toggle_counter` output register, as well as incrementing the 2-bit register representing the strobe. The `toggle_counter` is decremented by 5 whenever `strobe` reaches 3 which occurs every 4 clock cycles.

The waveforms from the top module is shown below:



Final clock generator

In addition, the synthesized RTL schematic is shown below.



RTL Schematic for `clock_gen`

IV. Testing

The testing procedure was simple, with a short testbench module that stayed constant throughout the development process. It essentially flipped the value of input register `clk_in` every 5 ns, while passing the necessary inputs and outputs to the top level `clock_gen` module.