Lab 2 - Clock Design Methodology

Introduction and Requirments

This lab focusing on creating clocks and pulses of different frequency and duty cycles. Except for the design, testing the clock waveforms on the digital system is also required. The design will be focusing on comparing the different waveform generated by the design.

What is Clock?

From the spec: Clocks are often used as the basis of timer systems used in numerous embedded devices such as traffic light, monitor screens, digital stopwatches, phones, etc.

Modules

There are mainly 4 main modules need to be explored.

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

There are total 9 taskes and submoduels we need to design and implement in the lab, in this lab, what I have done is to use the template in the spec, and design based on the 9 submodules.

Design Description

Design Task 1 [Clock Divider by Power of 2s]

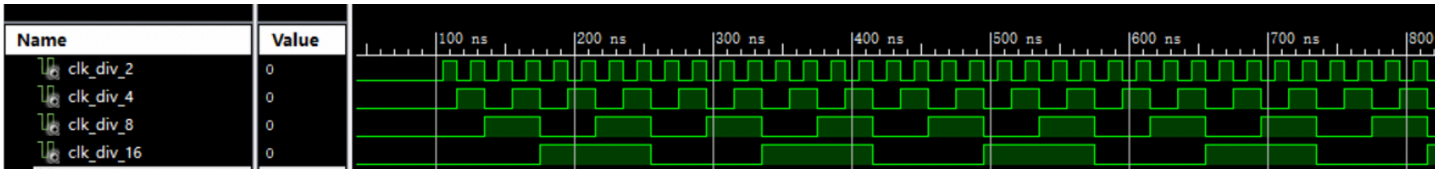
> Assign 4 1-bit wires to each of the bits from the 4-bit counter.

In the `clock_div_two` module, I used a 4-bit register `a` to implment the 4-bit counter, I used the given code of 40-bit counter from the spec. Note that the 4-bit counter has a range (0~15 in decimal). The lease significant bit of the 4-bit counter flips every second clock cycle, and then 2nd LSB filps once for every 4 clock cycles, 3nd LSB filps for every 8 cycles, and the 4nd LSB (i.e., MSB) fiples for every 16 clock cycles.

Thus, assign the output register `clk_div_2` , `clk_div_4` , `clk_div_8` , and, `clk_div_16` to the corresponding bit of the counter `a` , the code is as the bottom of the below.

```
reg [3:0] a;
always@(posedge clk_in )
begin
  if(rst)
    a <= 4'b0000;
  else
    a <= a + 1'b1;
end
assign clk_div_2 = a[0];
assign clk_div_4 = a[1];
assign clk_div_8 = a[2];
assign clk_div_16 = a[3];
```

Waveforms of `clock_div_two` module:



Clearly from the simulation, the output registers flips like what is stated ahead.

Design Task 2 [Even Division Clock]

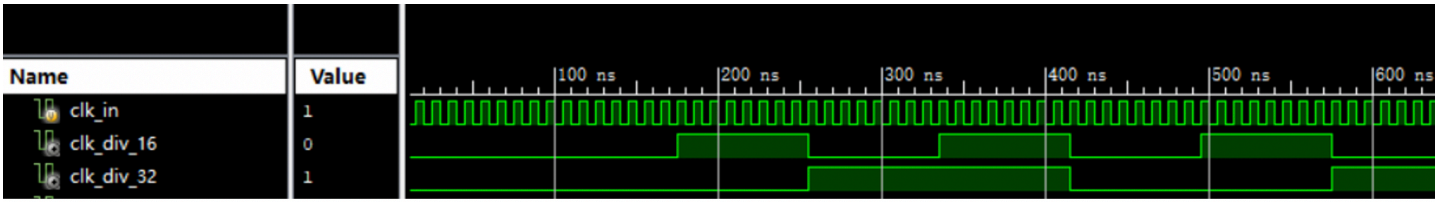
> In continuation of the 4-bit counter design, generate the divide by 32 clocks by flipping the output clock on every counter overflow.

The key of designing the task is to make use of the 4-bit counter. My design is to consider adding a extra bit to left of the most significant bit, transform the 4-bit counter as a virtual 5-bit counter. For a 5-bit counter, the range is 0~31 in decimal. That is, like what we did in task1, when the least significant 4-bits overflow (4b'1111), flip the virtual bit. i.e., the 1-bit output register `clk_div_32` .

```
reg [3:0] a;
always@(posedge clk_in ) begin
  if(rst)
    begin
      a <= 4'b0000;
      clk_div_32<=1'b0;
    end
  else
   begin
      a <= a + 1'b1;
      if(a == 4'b1111) // when the counter is 15 we flip the divide by 32 clock
        clk_div_32 <= ~clk_div_32;  //flip
```

```
13        end
14    end
```

Waveforms showing functionality of `divide-by-32` clock as below.



From the waveforms, we can see `clk_div_32` flipping at half the frequency of the `clk_div_16` register as expected.

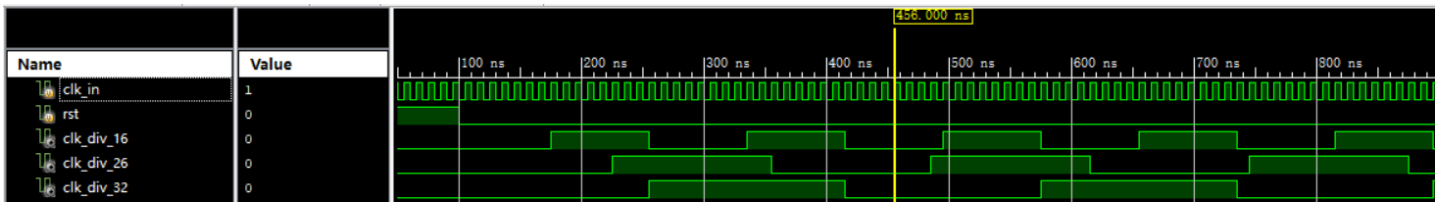
Design Task 3 [Even Division Clock]

> Generate a clock that is 26 times smaller by modifying when the counter resets to 0.

In this task, similar as in task 1 and task 2. My design is to create a new 4-bit counter. Everytime when it held the value 4'b1100 = 12 in decimal, the counter reset to 0, and the output resgister `clk_div_26` filps. This results in the output clock flipping every 13 counts of the system clock, and the output clock completes one clock cycle every 26 cycles of the input/system clock.

```
1   reg [3:0] a;
2   always@(posedge clk_in)
3   begin
4   if(rst)
5     begin
6       a <= 4'b0000;
7       clk_div_26<=1'b0;
8     end
9   else
10    begin
11      if(a == 4'b1100)  // when the counter is 12 we flip the divide by 26 clock
12        begin
13          a <= 4b'0000;
14          clk_div_26 <= ~clk_div_26;
15        end
16      else
17          a <= a + 1'b1;
18      end
19  end
20
```

Waveforms as below:



Design Task 4 [Odd Division Clock]

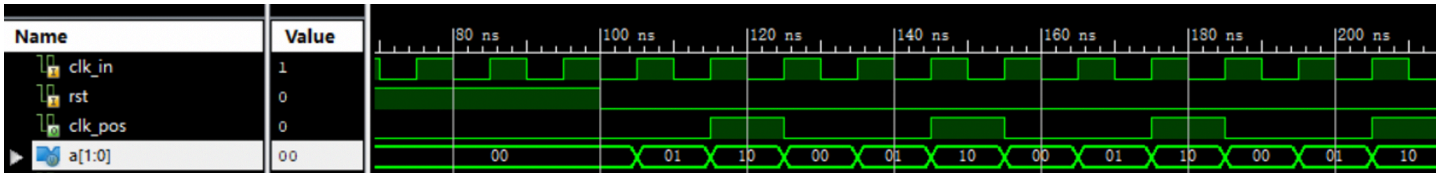
> Generate a 33% duty cycle clock using if statement and counters and verify the waveform.

To generate a 33% duty cycle clock named `clk_pos`. I use a 2-bit counter. When a = 2'b01 or a = 2'b10, the `clk_pos` is flipped on positive clock edges. And when `a` counter reach 2'b10 = 2 in decimal, it is reset to 2'b00 for every 3 positive clock edges. The counter `a` cycles between the values 0, 1, 2 in decimal.

In prediction, the output signal shoud active for 1/3 of the time.

```
reg [1:0] a;
always@(posedge clk_in) //positive
begin
if(rst)
  begin
    a <= 2'b00;
    clk_pos <= 1'b0;
  end
else
  if(a == 2'b01) //1
    begin
      clk_pos<= ~clk_pos;
      a <= a + 1'b1;
    end
  else if(a == 2'b10)  //counter to 2'b10 = 2 ,and next to 2'b00
    begin
      clk_pos<= ~clk_pos;
      a <= 2'b00;
    end
  else
    begin
      a <= a + 1'b1;
      clk_pos <= clk_pos;
    end
end
```

Waveforms for `clk_pos`:



As expected, the output active for 33% of the time.

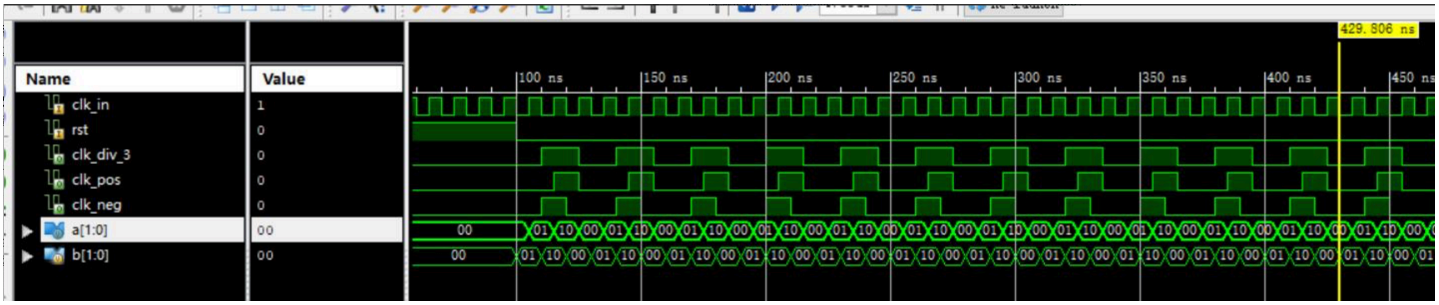
Design Task 5 [Odd Division Clock]

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

This task is basically the same as in the task 4. Duplicate the code and logic, and just change the always block to the one which is triggered on the falling edge of the clock input.

```
1   reg [1:0] b;
2   always@(negedge clk_in) //negedge
3     begin
4       if(rst)
5         begin
6           b <= 2'b00;
7           clk_neg <= 1'b0;
8         end
9       else if(b == 2'b01)
10        begin
11          clk_neg<= ~clk_neg;
12          b <= b + 1'b1;
13        end
14      else if(b == 2'b10)  //counter to 2'b10 ,and next to 2'b00
15        begin
16          clk_neg<= ~clk_neg;
17          b <= 2'b00;
18        end
19      else
20        begin
21          b <= b + 1'b1;
22          clk_neg <= clk_neg;
23        end
24    end
```

Waveforms `clk_neg` and comparison to `clk_pos`:



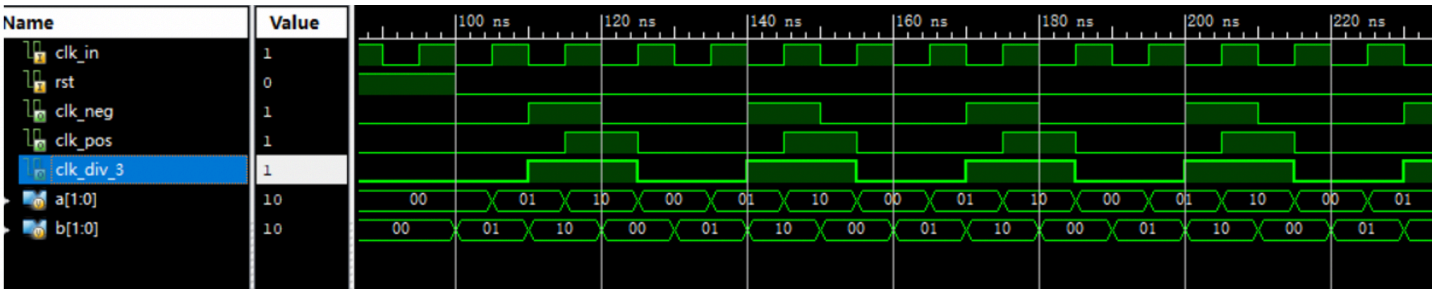
Design Task 6 [Odd Division Clock]

> Assign a wire that takes the logical or of the two 33% clocks

This one is simple, to take the logical OR of the two clocks from the pervious tasks, just use Combinational logic statement to connect the two output register of task4 and task5 to achieve it.

```
1    assign clk_div_3 = clk_neg | clk_pos; //logic or
```

Waveforms `clk_div_3` :



As expected, the `clk_div_3` active wherever `clk_pos` and `clk_neg` are active. This is what happened if assign a wire that takes the logical or of the two 33% clocks. And half of the clock cycle is a system clock period of 1.5 cycles in this case.

Design Task 7 [Odd Division Clock]

> Generate a 50% duty cycle divide-by-5 clock

In this task, I duplicated the design and code in task 4,5,6, inlclude the always block's sensitivity list include both the positive and negative edge. I created a 3-bit counter for this task. It could hold 3 bits that counts up to 3'b100. It is counting 5 clock edges essentially. So the counter `a` cycles between the values 0, 1, 2, 3, 4 in decimal. When `a` = 3'b010 or `a` = 3'b100, the `clk_pos` is flipped on positive clock edges. And when the input counter `a` reach 3'b100 = 4 in decimal, and the counter is reseted to 3'b000 = 0. Simillarly did it in the falling edgee case. Finally, use OR operator to generates a output register `clk_div_5` .

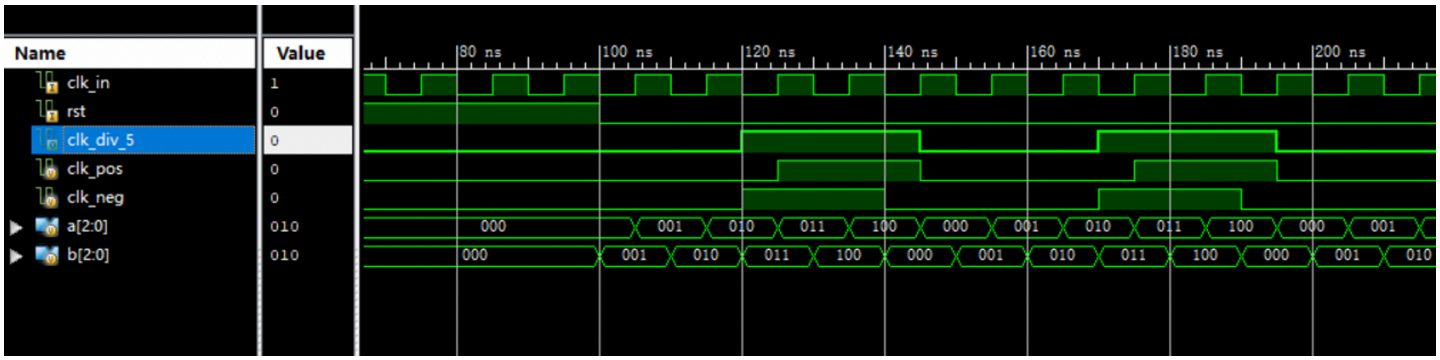
```
1    reg clk_pos;
2    reg [2:0] a;
3    //pos
```

```
always@(posedge clk_in)  //posedge
begin
  if(rst)
    begin
      clk_pos <= 0;
      a <= 3'b000;
    end
  else if(a == 3'b010)
    begin
      clk_pos <= ~clk_pos;
      a <= a + 1'b1;
    end
  else if(a == 3'b100)  //counter to 3'b100 ,and next to 3'b000
    begin
      clk_pos <= ~clk_pos;
      a <= 3'b000;
    end
  else
    begin
      clk_pos <= clk_pos;
      a <= a + 1'b1;
    end
end
// neg
reg       clk_neg;
reg [2:0] b;

always@(negedge clk_in)  //negedge
  begin
    if(rst)
      begin
        clk_neg <= 0;
        b <= 3'b000;
      end
    else if(b == 3'b010)
      begin
        clk_neg <= ~clk_neg;
        b <= b + 1'b1;
      end
    else if(b == 3'b100)  //counter to 3'b100 ,and next to 3'b000
      begin
        clk_neg <= ~clk_neg;
        b <= 3'b000;
      end
  else
      begin
        clk_neg <= clk_neg;
        b <= b + 1'b1;
```

```
52          end
53        end
54      //OR
55      assign clk_div_5 = clk_neg | clk_pos;
```

Waveforms `clk_div_5`:



As shown, with `clk_div_5` being flipped every 2.5 clock cycles of `clk_in` as expected. This proved that the designed clock is 50% duty as the output clock signal spends exactly half its time active.

Design Task 8 [Pulse/Strobes - 500kHz Clock]

> 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.**
>
> i.e., Divide-by-100 clock with 1% duty cycle, 50% duty divide-by-200 clock

In this lab, I initialized a 7-bit counter to hold the value up to 7b'110_0011 = 99 in decimal, which can be translated to 100 clock edges. The logic and design is similar as in task 4. We need to flip the bit only once when the counter reaches 100, and then to reset the counter back to 0. Here we need a duty cycle of 1%, we can get a toggle value, 98 in decimal. Flipped the output bit as well when reach this toggle value.

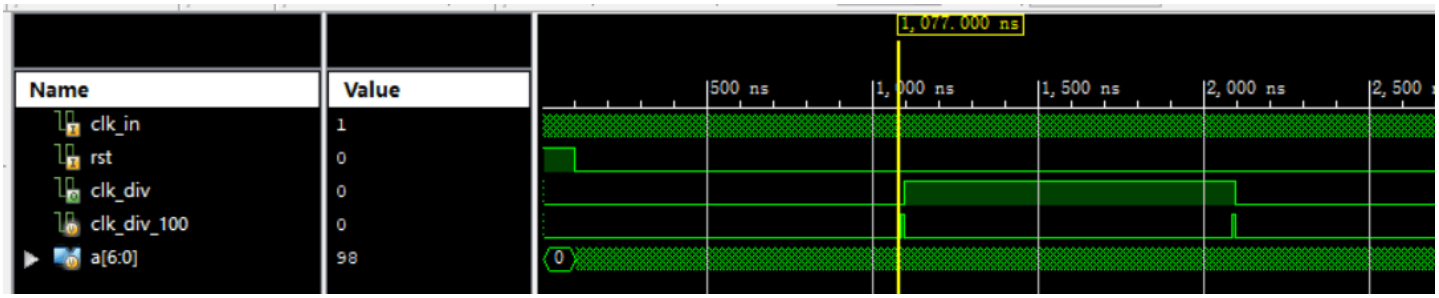
```
1   always@(posedge clk_in)
2     begin
3       if(rst)
4         begin
5           a <= 7'b000_0000;
6           clk_div_100 <= 1'b0;
7         end
8       else if(a == 7'd98)
9         begin
10          a <= a + 1'b1;
11          clk_div_100 <= ~clk_div_100;
12        end
13      else if(a == 7'd99) //counter to 7'd99 ,and next to 7'd0
```

```
        begin
          a <= 7'd0;
          clk_div_100 <= ~clk_div_100;
        end
      else
        begin
          a <= a + 1'b1;
          clk_div_100 <= clk_div_100;
        end
    end
```

Then I created 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. If `clk_div_100` is 1, the output register filps. Then it should be able to result a output clock with 50% duty cycle that is running at 500Khz, and is divide-by-200.

```
always@(posedge clk_in)
  begin
    if(rst)
      begin
        clk_div <= 0;
      end
    else if(clk_div_100)
      begin
        clk_div <= ~clk_div; //clk_div 500Khz
      end
    else
      clk_div <= clk_div;
  end
```

Waveforms of divide-by-100 and divide-by-200 clocks: `clk_div_100` and `clk_div` (divide by 200)



As expected, the output clock `clk_div`, which referes to divide-by-200 clock here. It is 50% duty cycle divide by 200 clock running at 500Khz. Reason: It spends 1000ns of 2000ns active. And each cycle of it takes 100 clock edges, (i.e., 200 clock cycles). In my testbench, i set `clk_in` to be filpped every 5 ns, so it has 10 ns per clocks cycle.

200 cycles x (10 ns/cycle) = 2000ns per clock cycle. And 1s/ 2000ns = 500000 Hz = 500K Hz.

Design Task 9 [Pulse/Strobes - Glitchy Counter]

> Use the master clock and a divide-by-4 **strobe** to generate an 8-bit counter that counts up by 3 on every positive edge of the master clock, but subtracts by 5 on every strobe. The sequence generated should be as shown below:
> 0→3→ 6→9→4→7→>10 -> 13→ 8 -> 11 -> 14 -> 17 -> 12 ...

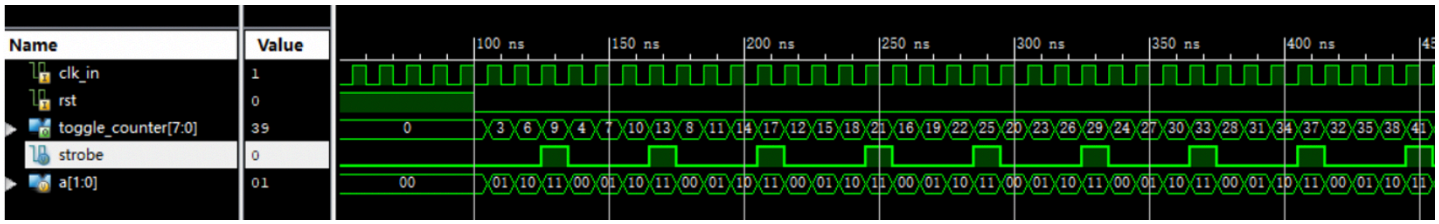
In this task, I initialized a 2-bit register reg[2:0] strobe to count up to 2'b11= 3 in decimal for strobe.

In the always block, acts on every positive edge of the system,

- When strobe is high, 3'b101 = 5 in decimal is subtracted from `toggle_counter` and strobe is flipped.
- When strobe is low, add 2'b11 = 3 to `toggle_counter` , effectively counting up by 2.

```
reg strobe;
reg [1:0] a;
always @ (posedge clk_in)
  begin
    if (rst)
      begin
        a <= 2'b00;
        strobe <= 1'b0;
        toggle_counter <= 8'b0000_0000;
      end
    else
      begin
        a <= a + 1'b1;
        strobe <= a[1]
        if (strobe)
          begin
            toggle_counter <= toggle_counter - 3'b101;// subtract by 5
            strobe <= ~strobe;
          end
        else
          toggle_counter <= toggle_counter + 2'b11;    // add 3
      end
  end
```

Waveform for `toggle_counter` :



Clearly, as except, the sequence of values held by the counter is

0→3→ 6→9→4→7→>10 -> 13→ 8 -> 11 -> 14 -> 17 -> 12 ...

Simulation Documentation

TestBench

The testing procedure was simple, it doesn't need to have too much code.

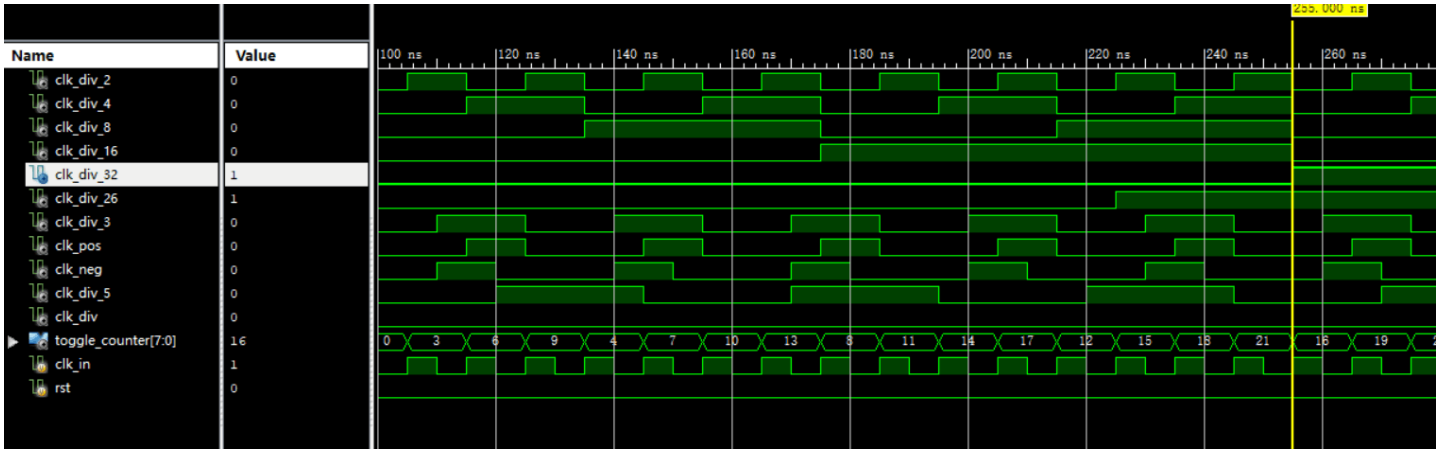
The testing procedure was simple, with a short testbench module that stayed constant throughout the development process. It essentially flipped the value of input register `clk_in` every 5 ns, while passing the necessary inputs and outputs to the top level clock_gen module.

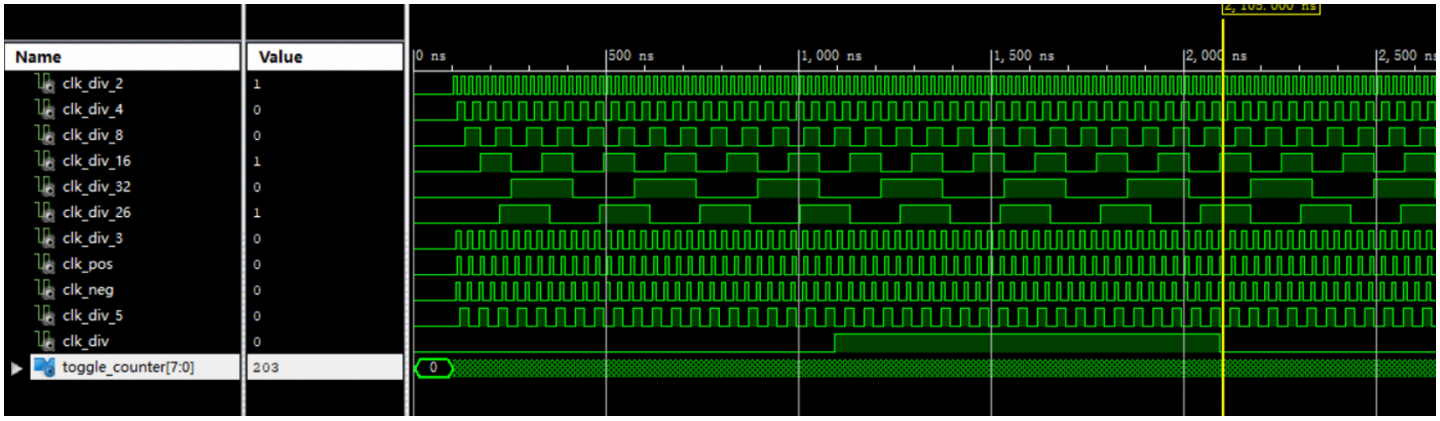
```
1   always #5 clk_in = ~clk_in;
```

Simulation Waveforms

A *testbench_UID.v* is created for testing. Switch to the simulation view and run "Simulation Behavioral Model", the simulation waveforms output is display as below figure.

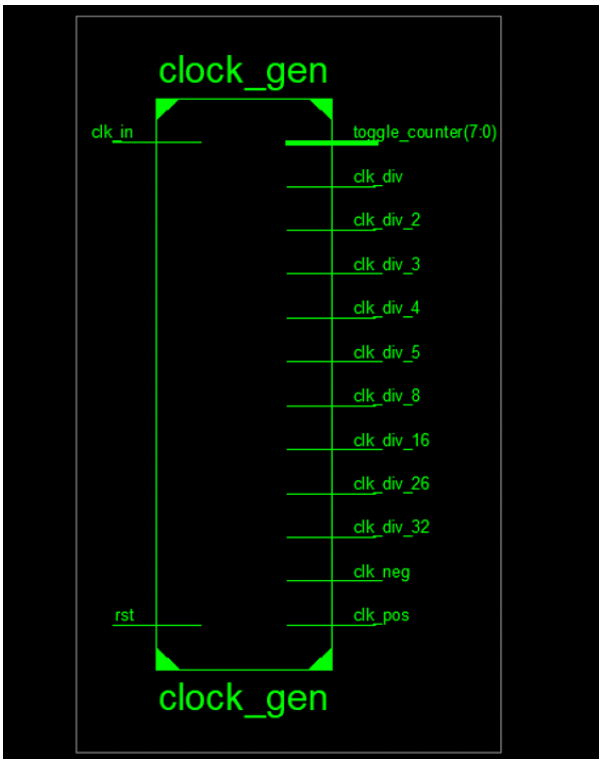
All 14 waveforms for the top-level module as shown below, the signals are ordered from top to bottom as requiremnet in the spec.

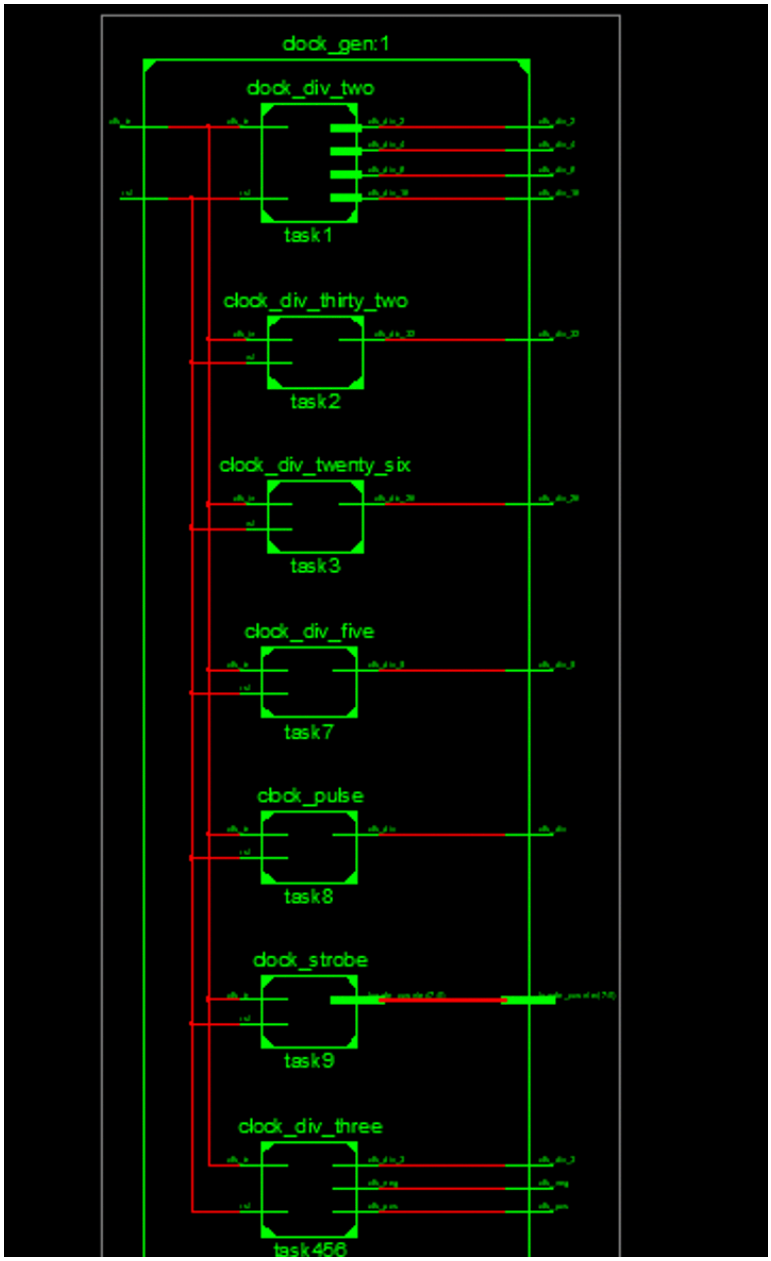




Rtl and Design summary report

RTL Schematic for clock_gen





Design summary report

It is a great lab for getting know the clock concept in digital system. When to file the bit is important.

The below reports display the general design overview summary.

clock_gen Project Status (05/02/2021 - 07:27:16)			
Project File:	Project2.xise	**Parser Errors:**	No Errors
Module Name:	clock_gen	**Implementation State:**	Synthesized
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	No Warnings
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	35	11,440	1%	
Number used as Flip Flops	35			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	29	5,720	1%	
Number used as logic	29	5,720	1%	
Number using O6 output only	16			
Number using O5 output only	0			
Number using O5 and O6	13			
Number used as ROM	0			
Number used as Memory	0	1,440	0%	
Number of occupied Slices	11	1,430	1%	
Number of MUXCYs used	0	2,860	0%	
Number of LUT Flip Flop pairs used	30			
Number with an unused Flip Flop	7	30	23%	
Number with an unused LUT	1	30	3%	

clock_gen.v clock_gen (RTL1)

Date Generated: 05/02/2021 - 07:42:58

Difficulties

Basiclly I am following the templates in the spec and expand. This lab took me most of time to understand, especially in terms of the mechanics for creating the clocks. However, once that was understood, the actual programming of the clocks was not all too difficult. The main problem for me was the conceptual understanding of everything that was going on.

The task 1-7 is not that hard, it could be easily implemented as long as you follow the hints in the spec. But task 8 and 9 is kind of trickly, they may need to review some formula about the period and frequency in advance.