# Lab 2 Report

CS M152A, Spring 2021

Caleb Lee

---

## Introduction

Lab 2 was about designing and testing various clock waveforms on digital systems. The lab was done through simulation only. There are a total of 9 tasks to do in the lab.

Task 1 tests the example code given from the instruction by assigning each bit of counter to output. The least significant bit of the counter is the division of 2, then it doubles as more bits from the left are taken. We are instructed to create a clock divider by 2, 4, 8, and 16 times slower.

Task 2 instructs us to generate a clock divide by 32 using the information given from Task 1. Task 3 instructs to generate a clock that is 26 times slower through changing the counter.

Task 4 is about verifying if the waveform created is a 33% duty cycle, which means the check waveform should be active only 1 out of 3 clock pulses. Task 5 replicates task 4 but with a negative edge instead of a positive edge. Task 6 asks to combine results from tasks 4 and 5. Task 7 requires a 50% duty cycle divide by 5-clock

Task 8's instruction was verifying if the output clock is a 50% duty cycle device by 200. Lastly, we are instructed to use the master clock and divide it by 4 to generate an 8 bit counter blue. The lab was written in Verilog to complete the above tasks.

## Design

The following figures (**Figure1** and **2**) are module declarations for lab 2.

```
module clock_gen(
    input clk_in,
    input rst,
    output clk_div_2,
    output clk_div_4,
    output clk_div_8,
    output clk_div_16,
    output clk_div_32,
    output clk_div_26,
    output clk_div_3,
    output clk_pos,
    output clk_neg,
    output clk_div_5,
    output clk_div,
    output [7:0] toggle_counter
     );

clock_div_two task1(
    .clk_in (clk_in),
    .rst (rst),
    .clk_div_2(clk_div_2),
    .clk_div_4(clk_div_4),
    .clk_div_8(clk_div_8),
    .clk_div_16(clk_div_16)
);

clock_div_thirty_two task2 (
    .clk_in(clk_in),
    .rst(rst),
    .clk_div_32(clk_div_32)
);

clock_div_twenty_six task3(
    .clk_in (clk_in),
    .rst (rst),
    .clk_div_26 (clk_div_26)
);
```

**Figure 1:** Module declaration 1

```
clock_div_three task456(
    .clk_in(clk_in),
    .rst(rst),
    .clk_div_3(clk_div_3),
    .clk_pos(clk_pos),
    .clk_neg(clk_neg)
);

clock_div_five task7(
    .clk_in (clk_in),
    .rst (rst),
    .clk_div_5 (clk_div_5)
);
clock_pulse task8(
    .clk_in(clk_in),
    .rst(rst),
    .clk_div(clk_div)
);
clock_strobe task9(
    .clk_in (clk_in),
    .rst (rst),
    .toggle_counter (toggle_counter)
);
endmodule
```

**Figure 2:** Module declaration 2

For task 1, I used the code that was provided in the instruction. From there, I added assigned statements to make the clock divided by 2, 4, 8, 16. **Figure 3** below is the module implementation of task 1.

```
// task 1
module clock_div_two(clk_in, rst, clk_div_2, clk_div_4,
clk_div_8, clk_div_16);
input clk_in, rst;
output clk_div_2, clk_div_4, clk_div_8, clk_div_16;
   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];
endmodule
```

**Figure 3:** Task 1 Module

Below the figure, **Figure 4**, represents the module for task 2. I implemented code from task 1 and switched flip-flop when the counter reaches 15. Since the counter goes from zero to 15, it will be divided by 32 for the full cycle.

```
// task 2
module clock_div_thirty_two(clk_in, rst, clk_div_32);
input clk_in, rst;
output reg clk_div_32;
reg [3:0] b;
   always @ (posedge clk_in)
   begin
      if (rst) begin
         b <= 4'b0000;
         clk_div_32 <= 1'b0;
      end
      else begin
         if(b == 4'b1111)
            clk_div_32 <= ~clk_div_32;
         b <= b + 1'b1;
      end
   end

endmodule
```

**Figure 4:** Task 2 Module

The below figure, **Figure 5**, represents the module for task 3. I implemented code from task 1 again. The code shown in **Figure 6** is the given code for task 1, and it will be my skeleton code for most of the tasks. For task 3, my counter resets when it reaches 12 (0 to 12) and switches flip-flop when that occurs. The full cycle will be 26 in the result.

```
// task 3
module clock_div_twenty_six(clk_in, rst, clk_div_26);
input clk_in, rst;
output reg clk_div_26;
reg [3:0] b;
   always @ (posedge clk_in)
   begin
      if (rst) begin
         b <= 4'b0000;
         clk_div_26 <= 1'b0;
      end
      else if (b == 4'b1100) begin
         clk_div_26 <= ~clk_div_26;
         b <= 4'b0000;
      end
      else
         b <= b + 1'b1;
   end
endmodule
```

**Figure 5:** Task 3 Module

```
// Example Verilog code for the counter

reg [3:0] a;

always @ (posedge clk)

  if (rst)

    a <= 4'b0000;

  else

    a <= a + 1'b1;
```

**Figure 6:** Skeleton Code

Below **Figure 7** and **8** is all in one module that performs task 4, 5, and 6. In **Figure 7**, task 4 is performed. Inside the always block of positive edge, code tries to switch flip-flop on when the counter is one and then two to turn it right off. The counter condition has to be one because the counter goes from 1 to 2 to 0 to 1 again. At 1, flip-flop would be 1, then at 2 it will be 0, and it will be 0 at 0 again, then 1 at 1. This allows two clock pulses to be in between every rise of clk_pos, which means it is one cycle for 3 cycles (satisfying 33%). **Figure 8** implements the same code as task 4 for task 5 but instead of a positive edge, it is the negative edge that drives always block this time. For task 6, task 4 and task 5 get OR'ed to combine two logic circuits.

```
// task 4, 5, 6
module clock_div_three(clk_in, rst, clk_div_3, clk_pos,
clk_neg);
input clk_in, rst;
output clk_div_3;
output reg clk_pos, clk_neg;
reg[3:0] b;
   //task 4
   always @ (posedge clk_in)
   begin
      if (rst) begin
         b <= 4'b0000;
         clk_pos <= 1'b0;
      end
      else if (b == 4'b0001) begin
         b <= b + 1'b1;
         clk_pos <= ~clk_pos;
      end
      else if (b == 4'b0010) begin
         b <= 4'b0000;
         clk_pos <= ~clk_pos;
      end
      else
         b <= b + 1'b1;
   end
```

**Figure 7:**Task 4 Module

```
reg[3:0] a;
   //task 5
   always @ (negedge clk_in)
   begin
      if (rst) begin
         a <= 4'b0000;
         clk_neg <= 1'b0;
      end
      else if (a == 4'b0001) begin
         a <= a + 1'b1;
         clk_neg <= ~clk_neg;
      end
      else if (a == 4'b0010) begin
         a <= 4'b0000;
         clk_neg <= ~clk_neg;
      end
      else
         a <= a + 1'b1;
   end

   //task 6
   assign clk_div_3 = clk_neg || clk_pos;
endmodule
```

**Figure 8:** Task 5,6 Module

```
// task 7
module clock_div_five(clk_in, rst, clk_div_5);
input clk_in, rst;
output clk_div_5;
reg clk_pos,clk_neg;
reg[3:0] b;
   //right
   always @ (posedge clk_in)
   begin
      if (rst) begin
         b <= 4'b0000;
         clk_pos <= 1'b0;
      end
      else if (b == 4'b0010) begin
         b <= b + 1'b1;
         clk_pos <= ~clk_pos;
      end
      else if (b == 4'b0100) begin
         b <= 4'b0000;
         clk_pos <= ~clk_pos;
      end
      else
         b <= b + 1'b1;
   end
```

**Figure 9:** Task 7 Module 1

```
reg[3:0] a;
   //left
   always @ (negedge clk_in)
   begin
      if (rst) begin
         a <= 4'b0000;
         clk_neg <= 1'b0;
      end
      else if (a == 4'b0010) begin
         a <= a + 1'b1;
         clk_neg <= ~clk_neg;
      end
      else if (a == 4'b0100) begin
         a <= 4'b0000;
         clk_neg <= ~clk_neg;
      end
      else
         a <= a + 1'b1;
   end

   //div 5
   assign clk_div_5 = clk_neg || clk_pos;
endmodule
```

**Figure 10:** Task 7 Module 2

The above **Figures 9 and 10** are a module for task 7, divide by 5 with 50% duty cycle. The code implementation is exactly the same as **Figures 7 and 8**. I have positive edge-driven always block and a negative edge-driven block, which I combine by using OR at the end to get the cycle divide by an odd number. Unlike even numbers, dividing by odd numbers require two always block. The only difference for **Figures 9 and 10** is that the counter

resetting point is adjusted, and the cycle length is also adjusted.

```
// task 8
module clock_pulse(clk_in, rst, clk_div);
input clk_in, rst;
output reg clk_div;
reg [6:0] b;
always @ (posedge clk_in)
begin
   if (rst) begin
      b <= 7'b110_0001;
   end
   else if (b == 7'b110_0011) begin
      b <= 7'b000_0000;
   end
   else
      b <= b + 1'b1;
end
always @ (posedge clk_in)
begin
   if (rst)
      clk_div <= 0;
   if(b == 7'b110_0011)
      clk_div <= ~clk_div;
end
endmodule
```

**Figure 11:** Task 8 Module

**Figure 11** shows the implementation of the task 8 module. The first always block is driven by the positive edge. It is a divide-by-100 clock with a 1% duty cycle. Register b also became 7 bit to be able to go all the way up to 100. Every time once counter hits 100, it resets to 0. The initial value of the counter might be odd because it is not 0, but it is necessary t be that high to start the clock pulse as early as the graph in the instructions. The second always blocks basically switches flip-flop every time counter hits 100, so it occurs once every hundred.

```
// task 9
module clock_strobe(clk_in, rst, toggle_counter);
input clk_in, rst;
output reg [7:0] toggle_counter;
reg [3:0] b;
   always @ (posedge clk_in)
   begin
      if (rst) begin
         b <= 4'b0000;
         toggle_counter <= 8'b0000_0000;
      end
      else if (b == 4'b0011) begin
         toggle_counter <= toggle_counter - 8'b0000_0101;
         b <= 4'b0000;
      end
      else begin
         b <= b + 1'b1;
         toggle_counter <= toggle_counter + 8'b0000_0011;
      end
   end
endmodule
```

**Figure 12:** Task 9 Module

**Figure 12** is the last module; it is the module for task 9. In clock_strobe, I basically subtract 8'b0000_0101 when the counter is at 3 since it happens every fourth time. Otherwise, I increment the counter by one while adding 8'b0000_0011 to the toggle_counter. I cannot do -5 and +3 because I have to represent them as a binary expression to avoid any errors.

## Simulation

The following codes are loading up modules in the testbench and running the clock on 100MHz. Clock pulse switches every 5 ns. In **Figure 14**, I wait for #4000 to see the divide-by-100 clock. I followed the guide in the lab session to set up the code.

```
module clock_gen_tb;

   // Outputs
   reg clk_in;
   reg rst;
   wire clk_div_2;
   wire clk_div_4;
   wire clk_div_8;
   wire clk_div_16;
   wire clk_div_32;
   wire clk_div_26;
   wire clk_div_3;
   wire clk_pos;
   wire clk_neg;
   wire clk_div_5;
   wire clk_div;
   wire [7:0] toggle_counter;

   // Instantiate the Unit Under Test (UUT)
   clock_gen uut (
      .clk_in(clk_in),
      .rst(rst),
      .clk_div_2(clk_div_2),
      .clk_div_4(clk_div_4),
      .clk_div_8(clk_div_8),
      .clk_div_16(clk_div_16),
      .clk_div_32(clk_div_32),
      .clk_div_26(clk_div_26),
      .clk_div_3(clk_div_3),
      .clk_pos(clk_pos),
      .clk_neg(clk_neg),
      .clk_div_5(clk_div_5),
      .clk_div(clk_div),
      .toggle_counter(toggle_counter)
   );
```

**Figure 13:** Testbench code 1

```
    initial begin
        // Initialize Inputs
        clk_in = 1'b0;
        rst = 1'b1;
        // Wait 100 ns for global reset to finish
        #100;
        clk_in = 1;
        rst = 1'b0;
        // Add stimulus here
        #4000;
        $stop;
    end
        always clk_in = #5 ~clk_in;
endmodule
```

**Figure 14:** Testbench code 2

(sorry for the format from here)

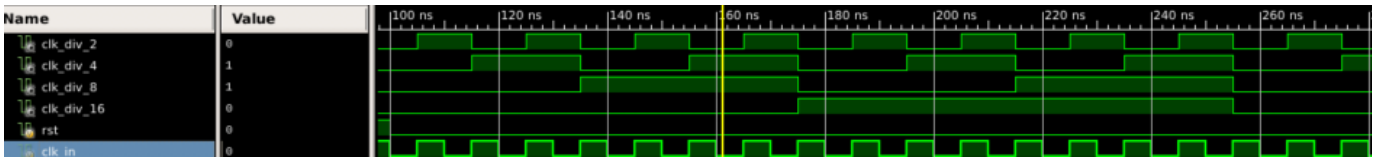


**Figure 15:** Waveform for task 1

**Figure 15** verified that the code works fine. The clock pulse is the last waveform. The first waveform is clk_div_2 and its flip-flop rises every other clock pulse. The second waveform is clk_div_4 and its flip-flop rises every other pulse compared to clk_div_2, making it div 4. There are 4 clock pulses for one cycle for div_4. We can observe that then clk_div_8 and clk_div_16 are also right because their cycle requires two cycles of their previous waves. For example, 4 needs two 2, 8 needs two 4, 16 needs two 8.

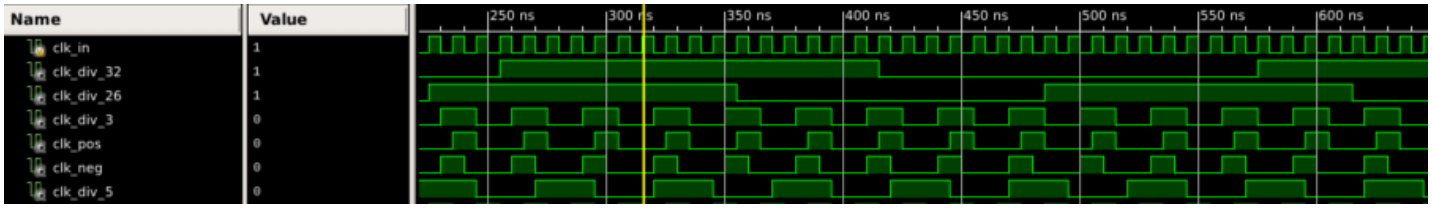


**Figure 16:** Waveform for task 2 - 7

**Figure 16** confirmed the correctness of tasks 2 to 7. The clock pulse is the first waveform. Task 2 is clk_div_32. If one counts the number of clock pulses for one cycle of clk_div_32, there are a total of 32 clock cycles. The same goes for clk_div_26 which is task 3. There is a total of 26 clock cycles for one clk_div_26 cycle. Task 4 is the clk_pos, and task 5 is the clk_neg. One clk_pos and one

clk_neg cycle are both 3 clock cycles. The only thing that differs is when they start because clk_pos works on positive edges and clk_neg activate on negative edges. Clk_div_3 is task 6 and is supposed to be the combined logic of tasks 4 and 5. As you can see in the waveform, the length of clk_div_3 is the overlap of clk_pos and clk_neg, so we know it works. Task 7 is a 50% duty cycle for divide-by-5. If you look at the waveform for clk_div_5 and clock pulse, there are a total of 5 cycles for one cycle for clk_div_5. This signifies that the duty cycle is 50% and also that it is divide by 5.

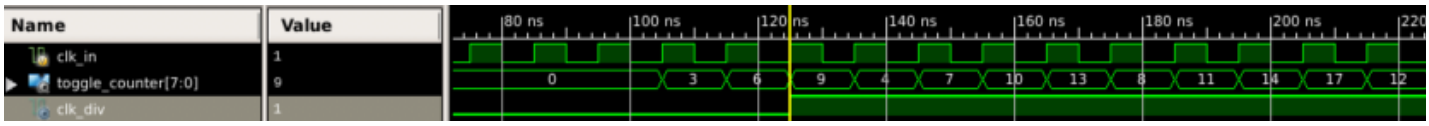


**Figure 17:** Waveform for task 9

**Figure 17** shows task 9 in unsigned decimal form, and a preview of task 8, which is clk_div. Every clock pulse, toggle_counter is increment by 3 and gets decremented by 5 every fourth clock pulse. So task 9 works fine. The preview of task 8 is there to show where the start of the clock pulse is. Task 8 clock pulse starts at 125 ns. Below **Figure 18** will show the entire task 8.

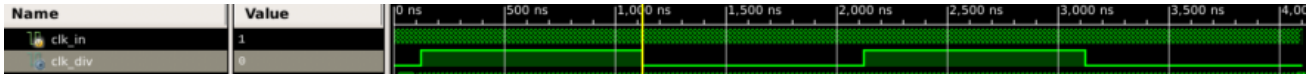


**Figure 18:** Waveform for task 8

Hard to see in **Figure 18**, but the termination of the first clock rise for clk_div is 1125, which is exactly 1000ns. The next clock rise for clk_dic occurs at 2125 ns, which makes a total of 2000ns (or 500Khz) per cycle and 50% duty cycle. So we have confirmed the waveform. **Figure 19** below is all waveforms combined.

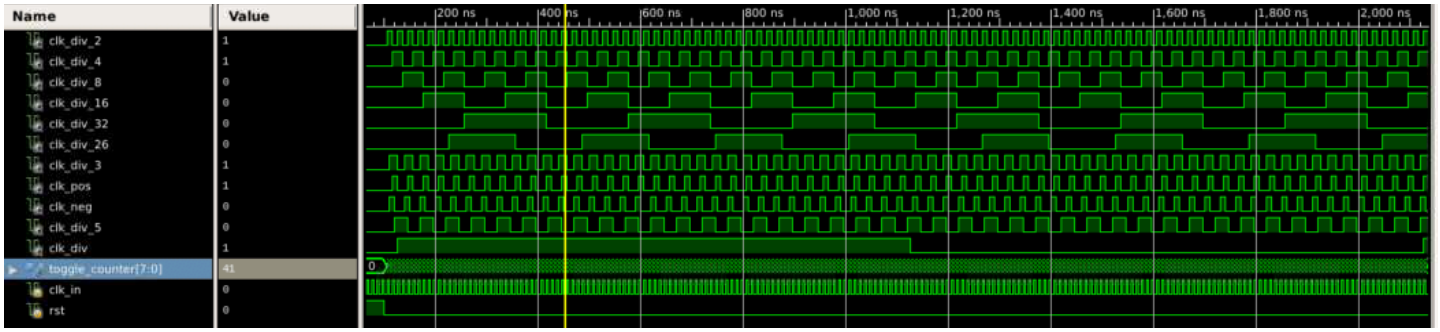


**Figure 19:** Waveform for task 1-9

**Table 1:** Design Summary

| clock_gen Project Status (05/03/2021 - 05:01:52) | | | |
|---|---|---|---|
| **Project File:** | Proj2.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:** | 5 Warnings (0 new) |
| **Design Goal:** | Balanced | **• Routing Results:** | |
| **Design Strategy:** | Xilinx Default (unlocked) | **• Timing Constraints:** | |
| **Environment:** | System Settings | **• Final Timing Score:** | |

| Device Utilization Summary (estimated values) | | | |
|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slice Registers | 41 | 18224 | 0% |
| Number of Slice LUTs | 62 | 9112 | 0% |
| Number of fully used LUT-FF pairs | 41 | 62 | 66% |
| Number of bonded IOBs | 21 | 232 | 9% |
| Number of BUFG/BUFGCTRLs | 1 | 16 | 6% |

| Detailed Reports | | | | | |
|---|---|---|---|---|---|
| **Report Name** | **Status** | **Generated** | **Errors** | **Warnings** | **Infos** |
| Synthesis Report | Current | Mon May 3 05:01:51 2021 | 0 | 5 Warnings (0 new) | 5 Infos (0 new) |
| Translation Report | Out of Date | Mon May 3 02:35:40 2021 | 0 | 0 | 0 |
| Map Report | Out of Date | Mon May 3 02:35:43 2021 | X 1 Error (1 new) | 0 | 0 |
| Place and Route Report | | | | | |
| Power Report | | | | | |
| Post-PAR Static Timing Report | | | | | |
| Bitgen Report | | | | | |

| Secondary Reports | | |
|---|---|---|
| **Report Name** | **Status** | **Generated** |
| ISIM Simulator Log | Out of Date | Mon May 3 05:01:17 2021 |

Above **Table 1** is the design summary of the code. The 5 warnings in the synthesis report state about truncating the extra bit when my counter overflows; however I do not have to worry about the extra bit that overflows, so it is fine. Map Report error is about licensing issue, but it did not give me any problem. I hit the map report by accident too.

## Conclusion

The design takes a clock pulse and uses counters to either divide the clock pulse by even or odd. The total 9 tasks all resulted in success when they were tested in waveforms. The difficulty I had during this lab was finding out what to do for the odd division clock. Syntax of Verilog felt a lot smoother due to practice from lab 1. Lab 2 had more math and concepts behind it. I think task 8 should be more detailed or clarified or explained a lot more thoroughly in class. Other than that, the lab was a fair game.