Danning Yu, 305087992
CS M152A, Lab 2
TA: Logan Kuo

# Project 3 Report

## Introduction

This purpose of this lab is to create a variety of clock waveforms with varying frequencies and duty cycles from a 100 MHz system clock by using various clock divider techniques that incorporate sequential and combinational logic. The main types of clock manipulation that will take place are division of a clock signal by powers of 2 (specifically, division by 2, 4, 8, and 16), division by an even number (specifically, division by 28), division by an odd number (specifically, division by 5), and a clock that overrides the system clock at preset intervals (specifically, every 4 clock cycles) to create a strobed counter. After designing the module, test bench code will be written to confirm the output clock frequencies and duty cycles in Xilinx Isim, and then synthesis and implementation will be carried out to view the physical resource usage if these clock manipulators were implemented on an actual FPGA board. The motivation for this lab is that on a real FPGA board, there is only one master system clock, and any other clock signals needed must be generated from this master clock.

Note: All clock signals in this report have a 50% duty cycle unless otherwise specified.

## Design

### Clock Generator (Top Level Module)

This is the top level module whose function is to include the divide by powers of 2, divide by even number, divide by odd number, and strobed counter modules. It consists of these 4 modules, passing in to each module the same 100 MHz clock signal `clk_in` and reset signal `rst`. It captures all the output signals from the 4 modules and outputs them. It was implemented in Verilog as a module named `clock_gen`. Its inputs and outputs are given in the table below.

| Input | Output |
|---|---|
| `clk_in` | Input: master or system clock signal, frequency of 100 MHz |
| `rst` | Input: reset signal to initialize the circuit |
| `clk_div_2` | Output: clock signal that divides the input clock signal by 2 |
| `clk_div_4` | Output: clock signal that divides the input clock signal by 4 |
| `clk_div_8` | Output: clock signal that divides the input clock signal by 8 |
| `clk_div_16` | Output: clock signal that divides the input clock signal by 16 |
| `clk_div_28` | Output: clock signal that divides the input clock signal by 28 |
| `clk_div_5` | Output: clock signal that divides the input clock signal by 5 |
| `glitchy_counter[7:0]` | Output: 8 bit counter that counts up by 2 on each clock cycle, except on strobe signals when it counts down by 5 |

Table 1: A summary of the inputs and outputs of the top level `clock`_gen module.

Using Table 1, along with the input and output specifications for each module, the following Verilog code to create the module `clock_gen`. The figure to the right it shows the RTL schematic for this module. Because this module only contained other submodules, the schematic consists solely of submodules and no gates or registers. This satisfies task (10).

```
---Clock Generator Module Code---
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_28,
    output clk_div_5,
    output [7:0] glitchy_counter
);
clock_div_two task_one(
      .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_twenty_eight task_two(
      .clk_in(clk_in),
      .rst(rst),
      .clk_div_28(clk_div_28)
);
clock_div_five task_three(
      .clk_in(clk_in),
      .rst(rst),
      .clk_div_5(clk_div_5)
);
clock_strobe task_four(
      .clk_in(clk_in),
      .rst(rst),
      .glitchy_counter
       (glitchy_counter)
);
endmodule
--------------------------------
```

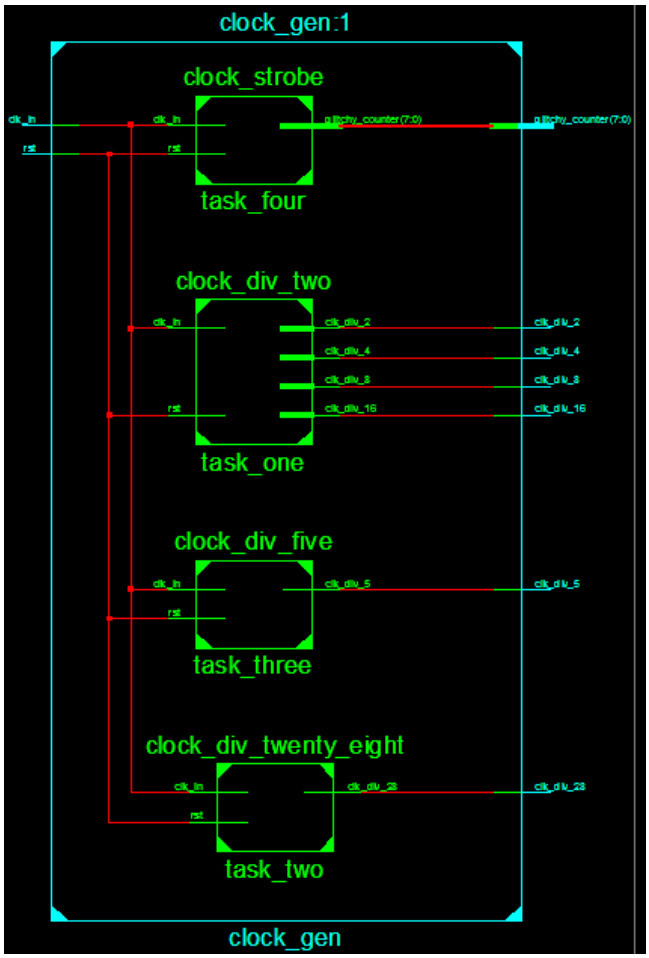


Figure 1: RTL synthesis diagram of the top level `clock_gen` module after synthesis. It is simply comprised of 4 submodules, which will be discussed in detail below. Two input signals, `clk_in` and `rst`, are connected to all the submodules, and all the submodules' outputs are passed on as the top level module's output, in accordance with Table 1.

**Divide Clock by $2^n$ Module**

The purpose of this module is to divide a 100 MHz input clock by powers of 2, specifically divide by 2, 4, 8, and 16. This is achieved by simply connecting each bit of a 4 bit counter to an output wire, with the least significant bit having a clock period twice that of the input clock, the second lowest bit having a clock period of 4 times of the input clock, and so on, until the most significant bit has a clock period that is 16 times longer than the input clock. This is was implemented in the Verilog code below, where the inputs `clk_in` and `rst` are the input clock and synchronous reset signal respectively. Using an internal 4 bit counter out, each divide by $2^n$ module output (`clk_div_[2, 4, 8, 16]`) was connected to each bit of `out`. Due to the properties of binary arithmetic, when the counter reaches 15, on the next clock cycle when 1 is added to it, it automatically goes back to 0, so no special logic is needed to control the maximum value of out. This satisfies task (1). The RTL synthesis schematic is given after the code.

```
--------------------Divide Clock by 2ⁿ Module Code--------------------
reg [3:0] out;
assign clk_div_2 = out[0];
assign clk_div_4 = out[1];
assign clk_div_8 = out[2];
assign clk_div_16 = out[3];

always @(posedge clk_in) begin
  if(rst) begin
    out <= 4'b0;
  end
  else begin
    out <= out + 4'b1;
  end
end
----------------------------------------------------------------------
```

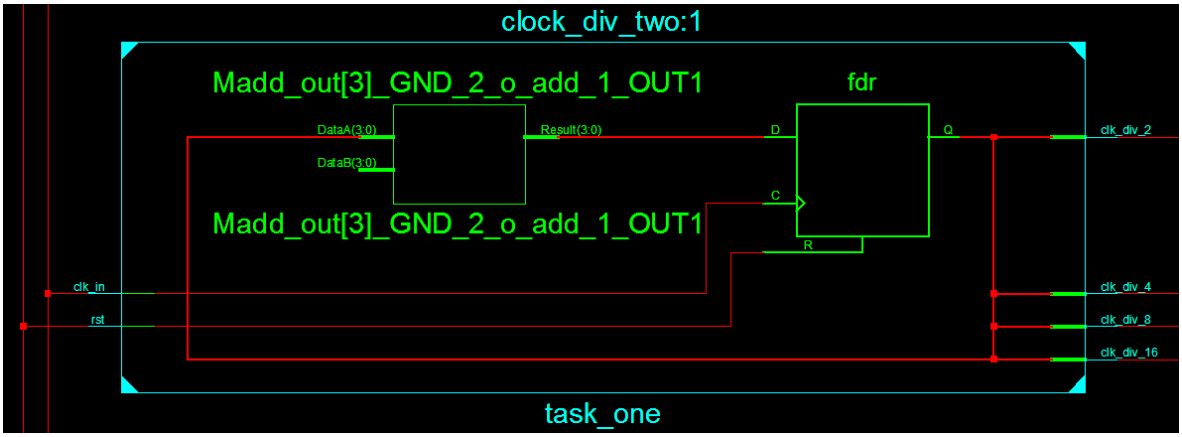


Figure 2: RTL synthesis of divide clock by $2^n$ module after synthesis. The design is very simple, with two inputs `clk_in` and `rst` on the right for the clock and rest signals, a 4 bit counter with a 4 bit register to store the result, and then the value inside the 4 bit register split into 4 individual wires to output, namely `clk_div_[2, 4, 8, 16]`.

**Even Division of Clock Module**

The function of this module is to divide the 100 MHz input clock by an even number, specifically 28. To prepare for this, first, a divide by 32 clock using a 4 bit counter was designed. This was done by creating a signal that inverts on every overflow of the 4 bit counter, which means every 16 clock cycles, thus leading to a divide by 32 counter. The Verilog code to do this is shown below, where `clk_in` and `rst` are inputs and `out2` is an internal 4 bit counter. At every positive clock edge, whenever the current value of `out2` is 15, it means it is about to be incremented to 16, which overflows to 0, and thus the value of `clk_div_32` should be complemented. This satisfies task (2).

```
------------------------Divide Clock by 32 Code-----------------------
reg [3:0] out2;
reg clk_div_32;
always @(posedge clk_in) begin
  if(rst) begin
    out2 <= 4'b0;
    clk_div_32 <= 1'b0;
  end
  else begin
    if(out2 == 4'b1111) begin
      clk_div_32 <= ~clk_div_32;
    end
    out2 <= out2 + 1'b1;
  end
end
----------------------------------------------------------------------
```

To create a divide by 28 counter, the same logic is applied, but now, the counter is changed as to when it "overflows": when its value becomes 14, it should go to 0, and when this overflow occurs, the divide clock by 28 output signal should be inverted. This design is shown in the Verilog code on the next page. It is very similar to the divide by 32 counter, but now, the divide by 28 output signal, called `clk_div_28`, is inverted whenever the counter out2 is 13, as it means the counter is about to get incremented to 14, which overflows to 0. After the code, the RTL synthesis schematic is given. This satisfies task (3).

```
------------------------Divide Clock by 28 Code-----------------------
reg [3:0] out2;

always @(posedge clk_in) begin
  if(rst) begin
    out2 <= 4'b0;
    clk_div_28 <= 1'b0;
  end
  else begin
    if(out2 == 4'b1101) begin
      clk_div_28 <= ~clk_div_28;
      out2 <= 4'b0;
    end
    else begin
      out2 <= out2 + 4'b1;
    end
  end
end
----------------------------------------------------------------------
```

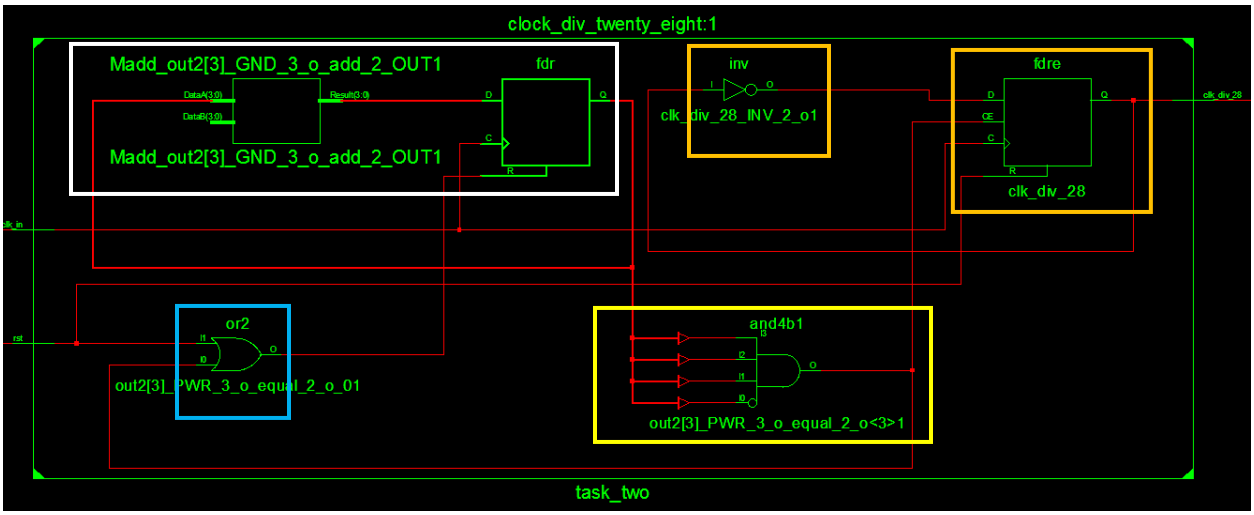


Figure 2: RTL schematic of the divide by 28 module. It uses a 4 bit adder (white) and stores the value in a 4 bit register (white), and then an AND gate (yellow) is used to check if the value in the 4 bit register is equal to 13 (4'b1101). If so, it enables the 1 bit register that stores the value of `clk_div_28` so that its value can be inverted (orange), and also resets the value in the 4 bit register, which can also be achieved using the `rst` signal through the use of an OR gate (blue).

**Odd Division of Clock Module**

The function of this module is to divide the 100 MHz input clock by an odd number, specifically 5. To prepare for this, a divide by 3 clock was designed. Because the clock is divided by an odd number, complementing the signal when the counter reaches half of the divide by value, as was done with the previous module, will not work. Doing this with an odd number such as 3 would mean flipping the output signal when the counter is 1.5, which is not possible.

Thus, to achieve a divide by odd number clock, one must create 2 clock signals: one that triggers on the positive edge and one that triggers on the negative edge, and then combine them together. The negative edge triggered signal serves provide the "0.5" of a clock signal. In addition, the duty cycles of the individual clocks must also be set so that when they are combined together, the resulting clock has a 50% duty cycle. For a divide by 3 clock, 2 33% duty cycle clocks are used. A 33% duty cycle clock is implemented by setting the output to 1 when the counter is 0 and setting the output to 0 when the counter is 1 or 2. Then, the counter is turned into a mod-3 counter so that after reaching 2, it resets to 0 so that the output clock turns on again, thus creating a 33% duty cycle. This is done on both the rising and falling edges of the input clock, and when the two resulting 33% duty cycle clocks are OR'd together, the result is a 50% duty cycle divide by 3 clock. The Verilog code to do this is shown below, where `out3_[pos, neg]` are registers to store the results of the 2 counters, `clk_33_[pos, neg]` are the two clocks, and `pos_neg_or` is the signal created by ORing the two clocks. `clk_in` is the master clock input and `rst` is a synchronous reset input. This accomplishes tasks (4) to (6).

----------Divide Clock by 3 Code----------

```
reg [1:0] out3_pos;
reg [1:0] out3_neg;
reg clk_33_pos;
reg clk_33_neg;
wire pos_neg_or;
assign pos_neg_or
  = clk_33_pos | clk_33_neg;

always @(posedge clk_in) begin
  if(rst) begin
    out3_pos <= 2'b0;
    clk_33_pos <= 1'b0;
  end
  else begin
    if(out3_pos == 2'b10) begin
      clk_33_pos <= 1'b1;
      out3_pos <= 2'b0;
    end
    else if(out3_pos == 2'b00)
    begin
     clk_33_pos <= 1'b0;
     out3_pos <= out3_pos + 2'b1;
    end
    else begin
      out3_pos <= out3_pos + 2'b1;
    end
  end
end
```

```
always @(negedge clk_in) begin
  if(rst) begin
  out3_neg <= 2'b0;
  clk_33_neg <= 1'b0;
  end
  else begin
    if(out3_neg == 2'b10) begin
       clk_33_neg <= 1'b1;
        out3_neg <= 2'b0;
    end
    else if(out3_neg == 2'b00)
    begin
      clk_33_neg <= 1'b0;
      out3_neg <= out3_neg + 2'b1;
    end
    else begin
     out3_neg <= out3_neg + 2'b1;
    end
  end
end
```

---

Using this divide by 3 clock as a model, a divide by 5 clock can be created by creating 2 divide by 5 clocks with a 40% duty cycle that trigger on the positive and negative edges and then ORing them together to get a 50% duty cycle divide by 5 clock. A 40% duty cycle was determined using the picture shown below. The idea is essentially to use the positive edge to turn on the output clock signal and then use the negative edge to turn off the signal, thus getting the half of a clock signal needed for division by an odd clock. This is shown in the picture below.

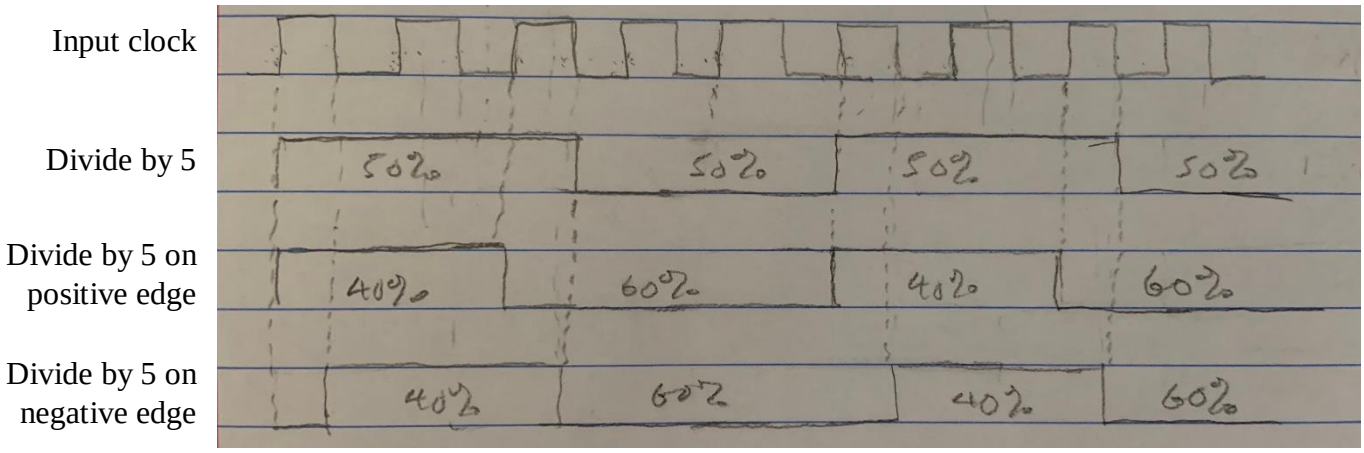


Figure 3: Waveforms for designing a 50% duty cycle divide by 5 clock. From an input clock (top waveform), two 40% duty cycle divide by 5 clocks are created (bottom 2 waveforms), one triggering on the positive edge and the other on the negative edge. When combined together, they form the desired divide by 5 clock (second waveform from the top).

Implementing this in Verilog resulted in the following code on the next page, where `clk_in` and `rst` are input signals and `clock_div_5` is the output. The variables `out3_[pos, neg]` are used to store the values of the 3 bit counters, and `clock_40_[pos, neg]` are used to store the 40% duty cycle divide by 5 clock. When synthesized, the counters consist of adder and register pair, both 3 bits. This satisfies task (7).

```
------------------------Divide Clock by 5 Code------------------------
reg [2:0] out3_pos;
reg [2:0] out3_neg;
reg clk_40_pos;
reg clk_40_neg;
assign clk_div_5
  = clk_40_pos | clk_40_neg;

always @(posedge clk_in) begin
  if(rst) begin
      out3_pos <= 3'b0;
      clk_40_pos <= 1'b0;
  end
  else begin
    if(out3_pos == 3'b100) begin
      clk_40_pos <= 1'b1;
      out3_pos <= 3'b0;
    end
    else if(out3_pos == 3'b001)
    begin
      clk_40_pos <= 1'b0;
      out3_pos <= out3_pos + 3'b1;
    end
    else begin
      out3_pos <= out3_pos + 3'b1;
    end
  end
end
```

```
always @(negedge clk_in) begin
  if(rst) begin
    out3_neg <= 3'b0;
    clk_40_neg <= 1'b0;
  end
  else begin
    if(out3_neg == 3'b100) begin
      clk_40_neg <= 1'b1;
      out3_neg <= 3'b0;
    end
    else if(out3_neg == 3'b001)
    begin
      clk_40_neg <= 1'b0;
      out3_neg <= out3_neg + 3'b1;
    end
    else begin
      out3_neg <= out3_neg+ 3'b1;
     end
  end
end
```

```
------------------------------------------------------------------------
```

The following page gives the RTL schematic generated from this code after synthesis..

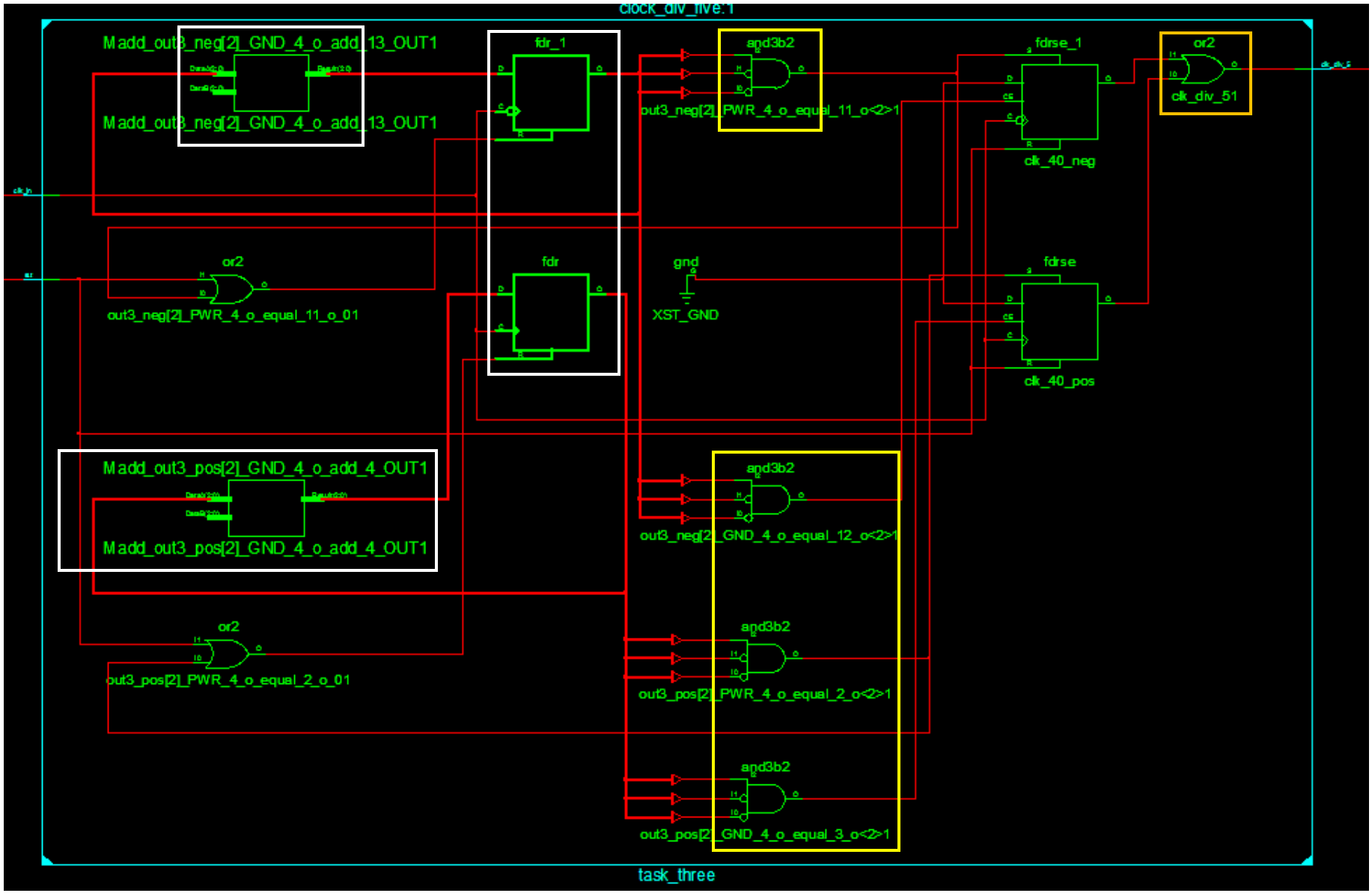


Figure 3: RTL schematic of a divide clock 5 module after synthesis. There are essentially two sets of identical hardware, one for the positive edge triggered divide by 5 clock and another for the negative edge triggered version. Their outputs are then OR'd (orange) to get the final divide by 5 clock. Like the divide by even number module, AND gates (yellow) are used to set and reset the 40% duty cycle clocks at various values of the counters (comprised of a 3 bit adder-register pair, in white).

**Strobed Counter Module**

The function of this module is to implement a counter that counts up by 2 on every 100 MHz master clock signal, except for every 4th clock signal, where it subtracts 5 instead. To prepare for this, first a 1% duty cycle divide by 100 clock was created, which was then used to create a divide by 200 clock by inverting its signal whenever the divide by 100 clock turns on. The 1% duty cycle divide by 100 clock was implemented using a mod-100 7 bit counter that turned on the divide clock by 100 signal whenever the counter became 0 and turned it off when the counter became 1. Then, a divide by 200 clock was created by inverting its signal whenever the divide by 100 clock was on. The code below implements this, where `clk_in` and `rst` are inputs and `divide_by_100` and `divide_by_200` are the clocks that divide by 100 and 200, respectively. A 7 bit register `counter` stores the current counter value. This satisfies task (8).

```
--------------------Divide Clock by 100 & 200 Code--------------------
reg divide_by_100;
reg divide_by_200;
reg [6:0] counter;

always @(posedge clk_in) begin
  if(rst) begin
    divide_by_100 <= 1'b0;
    counter <= 7'b0;
  end
  else begin
    if(counter == 7'b0) begin
      divide_by_100 <= 1'b0;
      counter <= counter + 7'b1;
    end
    else if(counter == 99) begin
      divide_by_100 <= 1'b1;
      counter <= 7'b0;
    end
    else begin
      counter <= counter + 7'b1;
    end
  end
end
```

```
always @(posedge clk_in) begin
  if(rst) begin
    divide_by_200 <= 1'b0;
  end
  else begin
    if(divide_by_100 == 1'b1)
    begin
      divide_by_200
        <= ~divide_by_200;
    end
  end
end
```

```
----------------------------------------------------------------------
```

Then, the strobed or glitchy counter that counts up by 2 on every 100 MHz master clock signal but subtracts 5 every 4th cycle was implemented by using the master clock input and a 2 bit counter that creates a divide by 4 clock from the master clock. In the code below, `clk_in` and `rst` are inputs, `counter` is an internal 2 bit counter, and `glitchy_counter` is an 8 bit output register. This satisfies task (9).

```
------------------------Strobed Counter Code-------------------------
reg [1:0] counter;
always @(posedge clk_in) begin
  if(rst) begin
    counter <= 2'b0;
    glitchy_counter <= 8'b0;
  end
  else begin
    if(counter == 2'b11) begin
      glitchy_counter <= glitchy_counter - 8'd5;
      counter <= counter + 1'b1;
    end
    else begin
      glitchy_counter <= glitchy_counter + 8'd2;
      counter <= counter + 1'b1;
    end
  end
end
---------------------------------------------------------------------
```

After synthesis, the following RTL schematic was generated, shown on the next page.

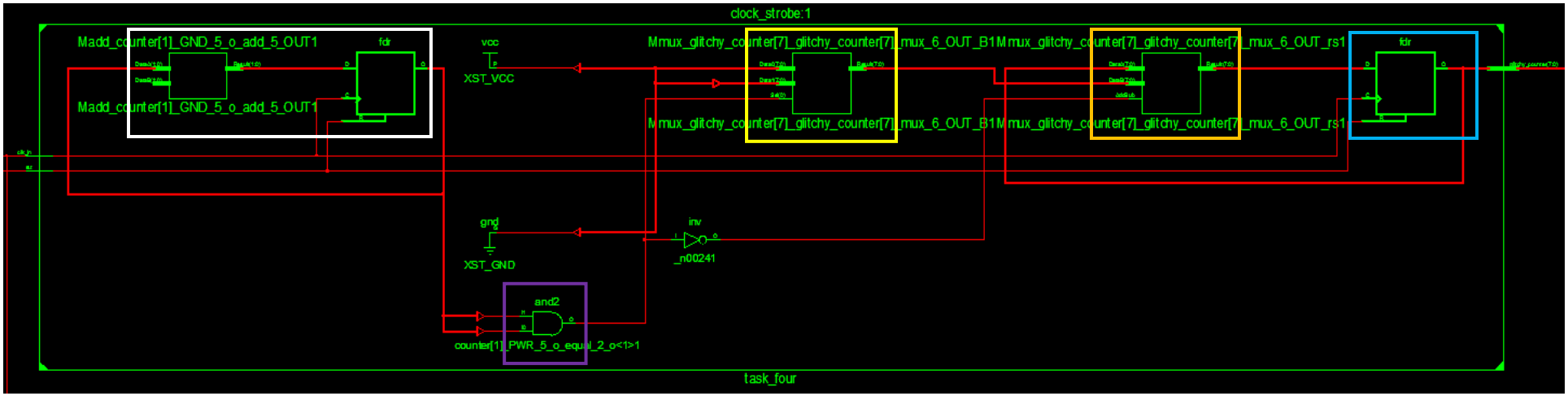


Figure 5: RTL schematic of the strobed counter module containing a glitchy counter. It uses a 2 bit adder and register (white) to form a 2 bit counter used to create a divide by 4 clock. An 8 bit 2 to 1 multiplexer (yellow) is used to select between the values 2 and 5, which are the magnitudes of how much the value stored in the glitchy counter can change. The selection bit is taken from the 2 bit counter, where whenever the valued stored in the 2 bit register is 2'b11 (violet), the multiplexer passes through 5; otherwise, it outputs 2. Then, an addition-subtraction module (orange) takes in an input from the 8 bit register storing the current value of the glitchy counter (blue), as well as the output of the 5 bit 2 to 1 multiplexer, and uses the same selection bit used for the multiplexer to determine whether to add or subtract the two values, thus producing the expected behavior of the module.

Note: the adder/subtractor shows up as a multiplexer ("mmux") most likely because it must select between two operations, making it kind of similar to an arithmetic logic unit

## Simulation Documentation

### Clock Generator Module (Top Level Module)

The requirement for this top level module is to output clocks that divide the input 100 MHz clock by 2, 4, 5, 8, 16, and 28, as well as a strobed counter that counts up to by 2 every master clock cycle, except for every 4th clock cycle, where it decreases by 5. It does this by including 4 submodules (described below), which contain the actual logic, providing to each module an input clock `clk_in` to divide and reset signal `rst` and then capturing their outputs.

The test bench code below initializes the reset signal `rst` to 1 to initialize all the registers of the submodules, and then inverts the clock signal `clk_in` every 5 ns, creating a 100 MHz clock. Examining the results shown in Figure 6 shows that the output signals `clk_div_[2, 4, 5, 8, 16, 28]` divide the input clock `clk_in` correctly. Also, the values in the 8 bit `glitchy_counter` fit the expected behavior. This satisfies task (10).

```
------------------Test Bench Code for Clock Divider-----------------
reg clk_in;                              .clk_div_28(clk_div_28),
reg rst;                                 .clk_div_5(clk_div_5),
wire clk_div_2;                          .glitchy_counter(glitchy_counter)
wire clk_div_4;                        );
wire clk_div_8;
wire clk_div_16;                       initial begin
wire clk_div_28;                         clk_in = 0;
wire clk_div_5;                          rst = 1;
wire [7:0] glitchy_counter;              #10;
                                         rst = 0;
clock_gen uut (                        end
  .clk_in(clk_in), .rst(rst),
  .clk_div_2(clk_div_2),               always begin
  .clk_div_4(clk_div_4),                 clk_in = ~clk_in;
  .clk_div_8(clk_div_8),                 #5; // 100 MHz clock
  .clk_div_16(clk_div_16),             end
-----------------------------------------------------------------------
```

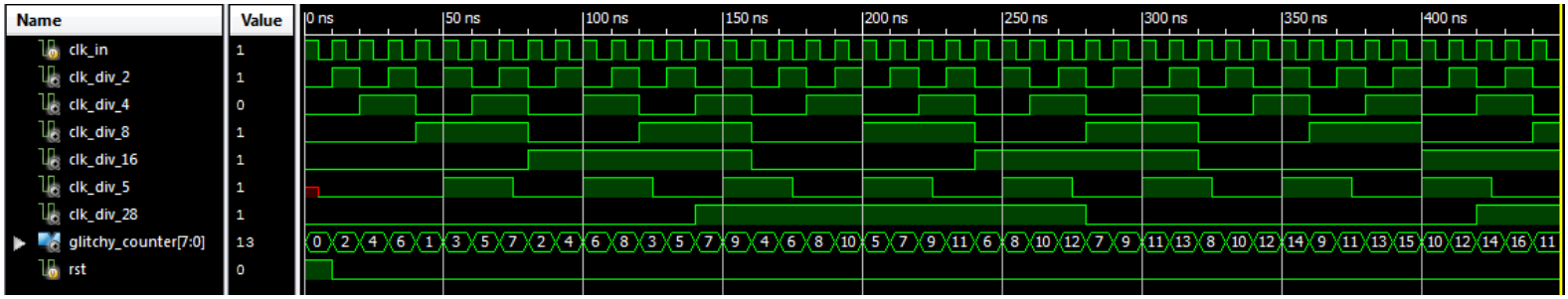


Figure 6: Waveform simulation results for the top level module. There are two inputs, a clock `clk_in` with a frequency of 100 MHz and a reset signal `rst`. The outputs are the 6 clock dividers that divide `clk_in` by 2, 4, 5, 8, 16, and 28, as well as an 8 bit `glitchy_counter` that updates on every rising edge of `clk_in`. The output clocks have the expected frequency and the value inside the strobed counter updates as expected, thus verifying the module's functionality.

### Divide Clock by $2^n$ Module

The requirement for this module is that it divides the input clock signal of 100 MHz by 2, 4, 8, and 16. By running the test bench code given above, this module was tested, as it is included in the top level clock generator module. Figure 7 below shows that the input clock `clk_in` is divided correctly, verifying the functionality of this module. This satisfies task (1).

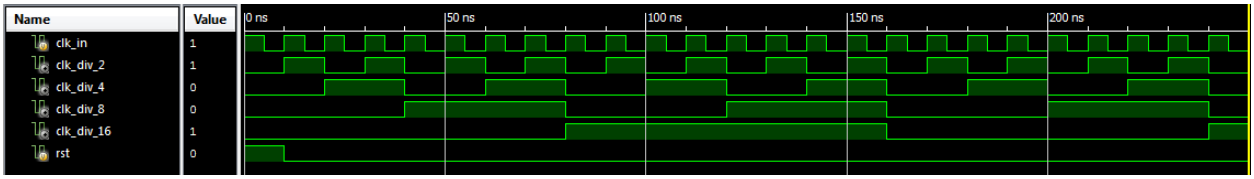


Figure 7: Waveform simulation results of the divide clock by $2^n$ module via testing the top level clock generator module. The divide clock by 2, 4, 8, and 16 output signals have the correct timing relative to the input clock `clk_in`, increasing the clock period by the appropriate factor.

### Even Division of Clock Module

The requirement for this module is that it divides the input clock signal of 100 MHz by a factor of 28. To prepare for this, first, a divide clock by 32 signal was generated from the input clock. Examining the waveform in Figure 8, where the input clock is `clk_in` and the output signal is `clk_div_32`, shows that the output clock signal has a period of 32 times that of the input clock signal, thus verifying the design. This satisfies task (2).

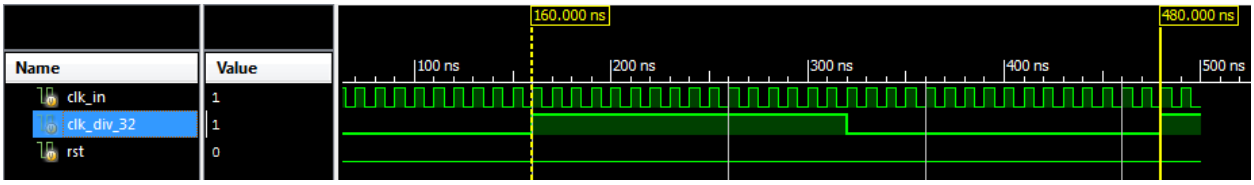


Figure 8: Waveform simulation results of a module to generate a divide clock by 32 output signal. The 2 time cursors, located at 2 successive rising edges of the `clk_div_32` signal at 160 ns and 480 ns, are located 320 ns apart, thus verifying the module's functionality.

After generating a divide clock by 32 signal, the same strategy was applied to generate a divide by 28 signal. By running the previously given test bench code, this module was tested, as it is included in the top level clock generator module. Examining Figure 9, where the input clock is `clk_in` and the output signal is `clk_div_28`, shows that the output clock signal has a period of 28 times that of the input clock signal, thus verifying the design. This satisfies task (2).

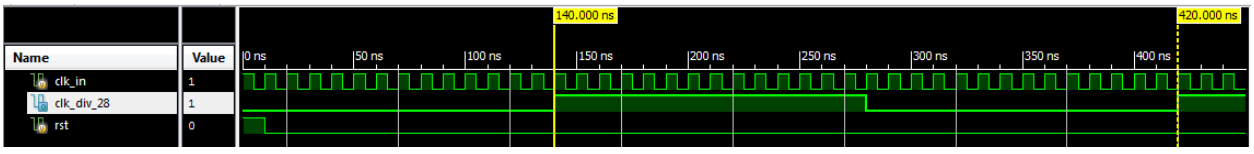


Figure 9: Waveform simulation results of the module to generate a divide clock by 28 output signal. The 2 time cursors, located at 2 successive rising edges of the `clk_div_28` signal at 140 ns and 420 ns, are located 280 ns apart, thus verifying the module's functionality.

**Odd Division of Clock Module**

The requirement for this module is that it divides the input clock signal of 100 MHz by a factor of 5. To prepare for this, first, a divide clock by 3 signal was generated from the input clock. This was done by creating 2 33% duty cycle divide by 3 clocks, one that updates on the rising edge and another on the falling edge, which were then OR'd together to create a 50% duty divide by 3 clock. Examining the waveform in Figure 10, where the input clock is `clk_in`, the 2 33% duty cycle divide by 3 clocks are `clk_33_pos` and `clk_33_pos`, and the 50% duty cycle divide by 3 clock is `pos_neg_or`, we see the signal `pos_neg_or` has a period of 3 times that of the input clock signal with a 50% duty cycle, thus verifying the design. This satisfies tasks (4-6).

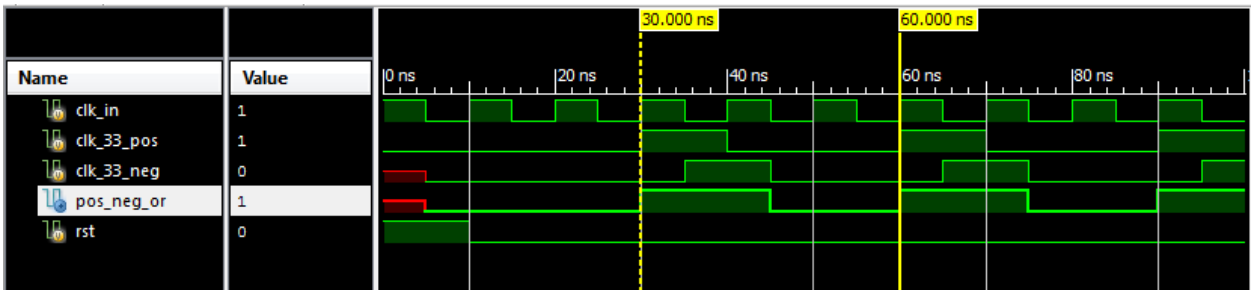


Figure 10: Waveform simulation results of a module to generate a divide clock by 3 output signal. Both `clk_33_pos` and `clk_33_neg` are only on for 33% of their period, and one triggers on the rising edge of `clk_in`, while the other triggers on the falling edge. The 2 time cursors, located at 2 successive rising edges of the `pos_neg_or` signal at 30 ns and 60 ns, are located 30 ns apart, thus verifying the module's functionality.

After generating a divide clock by 3 signal, the same strategy was applied to generate a divide by 5 signal from a 100 MHz input clock. By running the previously given test bench code, this module was tested, as it is included in the top level clock generator module. Examining the waveform in Figure 11, where the input clock is `clk_in` and the output signal is `clk_div_5`, shows that the output clock signal has a period of 5 times that of the input clock signal, thus verifying the design. This satisfies task (7).

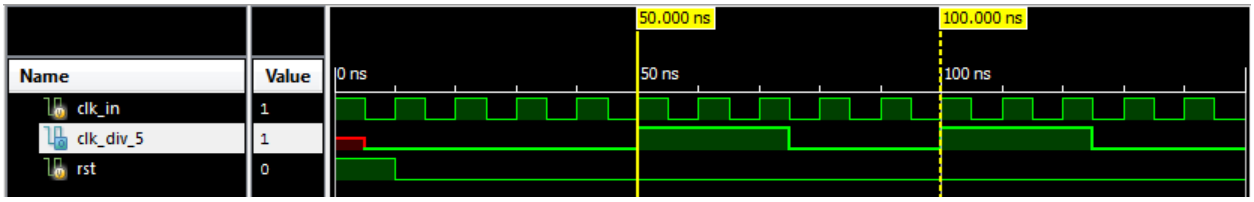


Figure 11: Waveform simulation results of the module to generate a divide clock by 5 output signal. The 2 time cursors, located at 2 successive rising edges of the `clk_div_5` signal at the 50 ns and 100 ns positions, are located 50 ns apart, thus verifying the module's functionality.

**Strobed Counter Module**

The requirement for this module is that it initializes an 8 bit value to 0, and then increments it by 2 on every rising edge of the 100 MHz input clock, except for every 4th cycle of the input clock, where it decrements the 8 bit value by 5 instead. To prepare for this, a 1% duty cycle divide by 100 clock signal was generated and then used to create a divide by 200 clock signal. Examining the waveform in Figure 12, where the input clock is `clk_in`, the divide by 100 clock signal is `clk_div_100`, and the divide by 200 clock signal is `clk_div_200`, shows that `clk_div_100` signal has a frequency of 1 MHz and a 1% duty cycle, and the `clk_div_200` has a frequency of 500 kHz, thus verifying the design. This satisfies task (8).

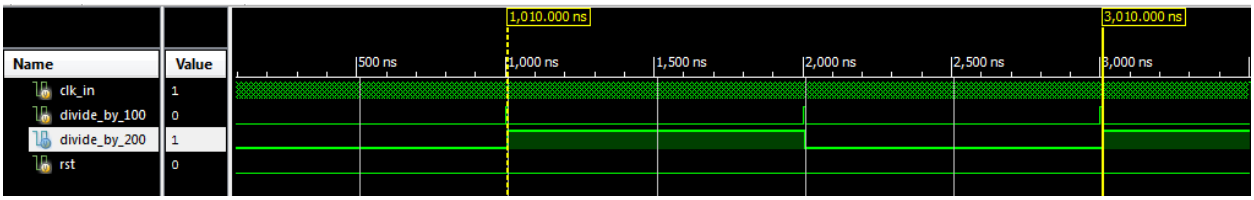


Figure 12: Waveform simulation results of a module to generate a 1% duty cycle divide clock by 100 signal and a divide clock by 200 signal. The divide clock by 100 signal, `divide_by_100`, has two blips located at 1 µs and 2 µs, showing that it has a desired frequency of 1 MHz. The 2 time cursors for the `divide_by_200` signal are located at 1.01 µs and 3.01 µs and thus are 3 µs apart, and the input clock `clk_in` has a period of 10 ns, verifying the module's functionality.

Next, the strobed or "glitchy" counter was tested. By running the previously given test bench code, this module containing the strobed counter was tested, as it is included in the top level clock generator module. Examining the waveform in Figure 13, where the input clock is `clk_in` and the output signal is `glitchy_counter` with its values shown in unsigned decimal, shows that the counter updates its values in accordance with the design specification, verifying the design and satisfying task (9).

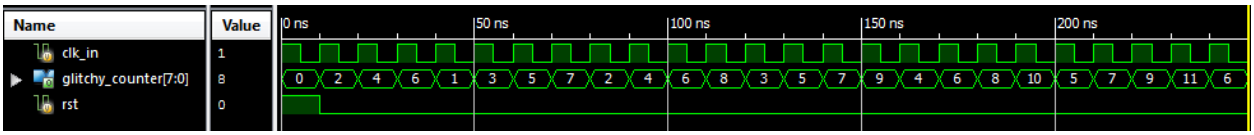


Figure 13: Waveform simulation results of the strobed counter module. Inspecting the values of `glitchy_counter`, which updates on every rising edge of the input clock `clk_in`, shows that it is updating with the correct behavior of incrementing by 2 and decrementing by 5 every 4th clock cycle, thus verifying the module's functionality.

After synthesis, the text report indicated that 23 of 18224 slice registers were used, as well as 27 of 9112 slice LUTs. Out of the 27 slice LUTs used, 4 had an unused flip flop, while 23 had a fully used LUT flip flop pair. 16 of 232 bonded IOBs were used, which corresponds to 2 1 bit inputs (clock and reset signals), 6 1 bit output clocks, and the 8 bit strobed counter output. After advanced HDL synthesis, 1 2 bit up counter was used (for the strobed counter module to have different behavior every 4th clock cycle), 2 3 bit up counters were used, (to create the divide by 5 clock) 2 4 bit up counters were used (for the divide by 2, 4, 8, 16, and 28 clocks), and

1 8 bit updown accumulator was used (to store the value of the strobed counter). Finally, an 8 bit 2 to 1 multiplexer was used to choose between the values 2 and 5 for the strobed counter, as described in the figure description in Figure 5. Detailed extracts from the synthesis report are given in Appendix A.

After implementation, the mapping report showed that 23 slice registers were used of 18224 total. The number of slice LUTs was reduced down to 18 of 9112. Of the 19 LUT flip flop pairs used, 2 had an unused flip flop, 1 had an unused LUT, and 16 had fully used LUT-FF pairs. The number of bonded IOBs remained the same at 16 out of 232. Detailed extracts from the mapping report are given in Appendix B. The place & route report has the exact same information concerning slice registers and slice LUTs, so it is not included in this report.

Overall, the LUT and register usage is low for both the synthesis and mapping reports, indicating that this is a feasible design to implement on the actual board.

## Conclusion

In this lab, clock dividers that divided the input 100 MHz clock by 2, 4, 5, 8, 16, and 28 were created, as well as a strobed counter that incremented by 2 on every input clock rising edge, except for every 4th clock cycle, where it decremented by 5 instead. The even clock dividers were all implemented by toggling the output signal when an internal counter reached certain values on the rising edge, and the divide by 5 clock was implemented using a combination of signals that updated on the rising and falling edges of the input clock. The strobed counter used an internal counter to send a signal every 4 cycles to cause the strobed counter to decrement by 5 instead of incrementing by 2, which is the default behavior. The clock dividers and strobed counters were implemented in 4 different modules that was then brought together into a top level module that was tested as one whole unit in a test bench. The resulting output waveforms had their clock frequency verified (and for the strobed counter, its value), confirming the correctness of the design. From this lab, I learned a variety techniques for dividing a clock by various amounts and changing a clock's duty cycle from its default 50%.

During this lab, the main difficulty I encountered was with figuring out what duty cycle to use for the 2 individual divide by 5 clock signals triggering on the positive and negative edge of the input clock that would be OR'd together to create a divide by 5 clock. However, after carefully sketching out the expected behavior (Figure 3), I was able to figure out that a 40% duty cycle was needed. As a whole, the lab was straightforward and the specification was clear, so I don't have any suggestions for improving the lab.

# Appendix

Note: Only sections pertaining to slice registers, slice LUTs, gates, and IO are included. Duplicate information within the same report may be omitted.

## Appendix A: Synthesis Report

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================
Synthesizing Unit <clock_gen>.
    Related source file is "C:\Danning\CS M152A\Project3\clock_gen.v".
    Summary:
      no macro.
Unit <clock_gen> synthesized.

Synthesizing Unit <clock_div_two>.
    Related source file is "C:\Danning\CS M152A\Project3\clock_gen.v".
    Found 4-bit register for signal <out>.
    Found 4-bit adder for signal <out[3]_GND_2_o_add_1_OUT> created at
line 76.
    Summary:
      inferred   1 Adder/Subtractor(s).
      inferred   4 D-type flip-flop(s).
Unit <clock_div_two> synthesized.

Synthesizing Unit <clock_div_twenty_eight>.
    Related source file is "C:\Danning\CS M152A\Project3\clock_gen.v".
    Found 1-bit register for signal <clk_div_28>.
    Found 4-bit register for signal <out2>.
    Found 4-bit adder for signal <out2[3]_GND_3_o_add_2_OUT> created
at line 101.
    Summary:
      inferred   1 Adder/Subtractor(s).
      inferred   5 D-type flip-flop(s).
Unit <clock_div_twenty_eight> synthesized.

Synthesizing Unit <clock_div_five>.
    Related source file is "C:\Danning\CS M152A\Project3\clock_gen.v".
    Found 1-bit register for signal <clk_40_pos>.
    Found 3-bit register for signal <out3_neg>.
    Found 1-bit register for signal <clk_40_neg>.
    Found 3-bit register for signal <out3_pos>.
    Found 3-bit adder for signal <out3_pos[2]_GND_4_o_add_4_OUT>
created at line 206.
    Found 3-bit adder for signal <out3_neg[2]_GND_4_o_add_13_OUT>
created at line 227.
```

```
Summary:
      inferred   2 Adder/Subtractor(s).
      inferred   8 D-type flip-flop(s).
Unit <clock_div_five> synthesized.

Synthesizing Unit <clock_strobe>.
    Related source file is "C:\Danning\CS M152A\Project3\clock_gen.v".
    Found 8-bit register for signal <glitchy_counter>.
    Found 2-bit register for signal <counter>.
    Found 8-bit subtractor for signal
<glitchy_counter[7]_GND_5_o_sub_3_OUT> created at line 285.
    Found 8-bit adder for signal
<glitchy_counter[7]_GND_5_o_add_4_OUT> created at line 289.
    Found 2-bit adder for signal <counter[1]_GND_5_o_add_5_OUT>
created at line 290.
    Summary:
      inferred   2 Adder/Subtractor(s).
      inferred  10 D-type flip-flop(s).
      inferred   1 Multiplexer(s).
Unit <clock_strobe> synthesized.
=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 6
 2-bit adder                                           : 1
 3-bit adder                                           : 2
 4-bit adder                                           : 2
 8-bit addsub                                          : 1
# Registers                                            : 9
 1-bit register                                        : 3
 2-bit register                                        : 1
 3-bit register                                        : 2
 4-bit register                                        : 2
 8-bit register                                        : 1
# Multiplexers                                         : 1
 8-bit 2-to-1 multiplexer                              : 1
=========================================================================
INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that
some arithmetic operations in this design can share the same physical
resources for reduced device utilization. For improved clock frequency
you may try to disable resource sharing.
```

```
=========================================================================
*                        Advanced HDL Synthesis                         *
=========================================================================
Synthesizing (advanced) Unit <clock_div_five>.
The following registers are absorbed into counter <out3_neg>: 1
register on signal <out3_neg>.
The following registers are absorbed into counter <out3_pos>: 1
register on signal <out3_pos>.
Unit <clock_div_five> synthesized (advanced).

Synthesizing (advanced) Unit <clock_div_twenty_eight>.
The following registers are absorbed into counter <out2>: 1 register
on signal <out2>.
Unit <clock_div_twenty_eight> synthesized (advanced).

Synthesizing (advanced) Unit <clock_div_two>.
The following registers are absorbed into counter <out>: 1 register on
signal <out>.
Unit <clock_div_two> synthesized (advanced).

Synthesizing (advanced) Unit <clock_strobe>.
The following registers are absorbed into accumulator
<glitchy_counter>: 1 register on signal <glitchy_counter>.
The following registers are absorbed into counter <counter>: 1
register on signal <counter>.
Unit <clock_strobe> synthesized (advanced).
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Counters                                             : 5
 2-bit up counter                                      : 1
 3-bit up counter                                      : 2
 4-bit up counter                                      : 2
# Accumulators                                         : 1
 8-bit updown accumulator                              : 1
# Registers                                            : 3
 Flip-Flops                                            : 3
```

```
=========================================================================
*                        Low Level Synthesis                            *
=========================================================================
INFO:Xst:2146 - In block <clock_gen>, Counter <task_one/out>
<task_four/counter> are equivalent, XST will keep only <task_one/out>.

Optimizing unit <clock_gen> ...
INFO:Xst:2261 - The FF/Latch <task_one/out_0> in Unit <clock_gen> is
equivalent to the following FF/Latch, which will be removed :
<task_two/out2_0>
INFO:Xst:2261 - The FF/Latch <task_one/out_2> in Unit <clock_gen> is
equivalent to the following FF/Latch, which will be removed :
<task_four/glitchy_counter_0>

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block clock_gen, actual
ratio is 0.

Final Macro Processing ...
=========================================================================
Final Register Report

Macro Statistics
# Registers                                          : 23
 Flip-Flops                                          : 23

=========================================================================
*                            Design Summary                             *
=========================================================================

Top Level Output File Name         : clock_gen.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 28
#      INV                         : 2
#      LUT2                        : 2
#      LUT3                        : 3
#      LUT4                        : 9
#      LUT5                        : 6
#      LUT6                        : 5
#      MUXF7                       : 1
# FlipFlops/Latches                : 23
#      FD                          : 9
#      FDR                         : 13
#      FDR_1                       : 1
```

```
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 15
#      IBUF                        : 1
#      OBUF                        : 14

Device utilization summary:
---------------------------
Selected Device : 6slx16csg324-3

Slice Logic Utilization:
 Number of Slice Registers:             23  out of  18224     0%
 Number of Slice LUTs:                  27  out of   9112     0%
    Number used as Logic:               27  out of   9112     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    27
   Number with an unused Flip Flop:      4  out of     27    14%
   Number with an unused LUT:            0  out of     27     0%
   Number of fully used LUT-FF pairs:   23  out of     27    85%
   Number of unique control sets:        3

IO Utilization:
 Number of IOs:                         16
 Number of bonded IOBs:                 16  out of    232     6%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:              1  out of     16     6%
```

## Appendix B: Mapping Report

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                23 out of  18,224    1%
    Number used as Flip Flops:              23
    Number used as Latches:                  0
    Number used as Latch-thrus:              0
    Number used as AND/OR logics:            0
  Number of Slice LUTs:                     18 out of   9,112    1%
    Number used as logic:                   18 out of   9,112    1%
      Number using O6 output only:          10
      Number using O5 output only:           0
      Number using O5 and O6:                8
      Number used as ROM:                    0
    Number used as Memory:                   0 out of   2,176    0%
```

```
Slice Logic Distribution:
  Number of occupied Slices:                  9 out of   2,278    1%
  Number of MUXCYs used:                      0 out of   4,556    0%
  Number of LUT Flip Flop pairs used:        19
    Number with an unused Flip Flop:          2 out of      19   10%
    Number with an unused LUT:                1 out of      19    5%
    Number of fully used LUT-FF pairs:       16 out of      19   84%
    Number of unique control sets:            4
    Number of slice register sites lost
      to control set restrictions:           17 out of  18,224    1%

IO Utilization:
  Number of bonded IOBs:                     16 out of     232    6%
```