DIGITAL SYSTEM DESIGN APPLICATION

EHB436E CRN: 11280

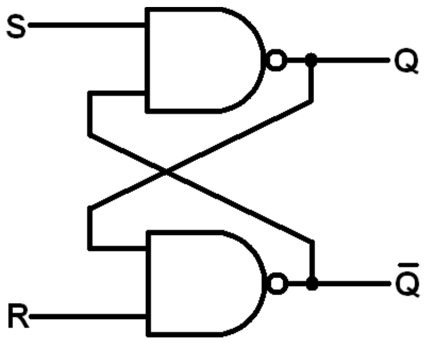
Salih Ömer Ongün
040220780

Experiment 5

Salih Ömer Ongün
040220780

D FLIP -FLOP

SR NAND LATCH

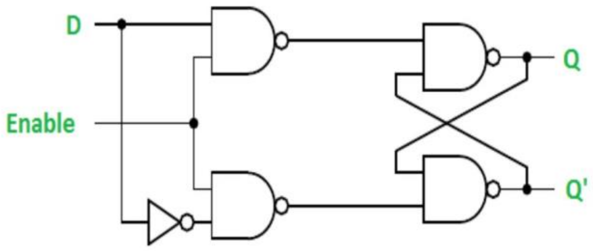


S	R	Q	Q’
0	1	1	0
1	1	No Change	
1	0	0	1
0	0	Forbidden	

Characteristic Function: Q(t+1) = R’ + S’ * Q(t)

When the S = 0, the output Q = 1. This is called the "Set" state. When the R = 1, the output Q = 0. This is called the "Reset" state. When S = 1 and R = 1, the latch retains its previous state. The outputs Q and Q' do not change. When S = 0 and R = 0, the outputs Q and Q' both become 1. Q and Q’ have the same logic value. This is an invalid or forbidden state and should be avoided.

GATED D LATCH

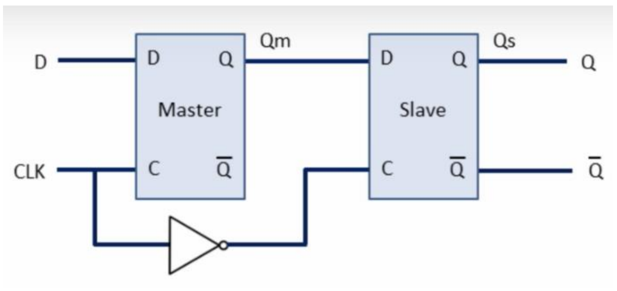


E	D	Q(n+1)
0	x	No change (Q(n))
1	0	0
1	1	1

When the E =0, output cannot change independent of D. When E = 1 and D =0, output is logic 0. This is called “Reset” state. When E = 1 and D =1, output is logic 1. This is called “Set” state.

Characteristic Function: Q(n+1) = E.D + E’.Q(n) ***[1]***

Edge Trigged D Flip Flop



This is Master-Slave falling edge D flip flop. It consists of two latches. If CLK = 1, Master load with new data and transfer to Qm, Slave cannot transfer Qm to output. If CLK =0, Slave transfer to Qm value to Q output but Master lacth cannot transfer new data to Qm.

Flip flops transfer data to the output when CLK transitions from 1 to 0 or from 0 to 1. This is called Edge Sensitivity. Flip flops transfer data to output when CLK is 1 or 0. This is called Level Sensitivity.

DFF With Synchronous Reset

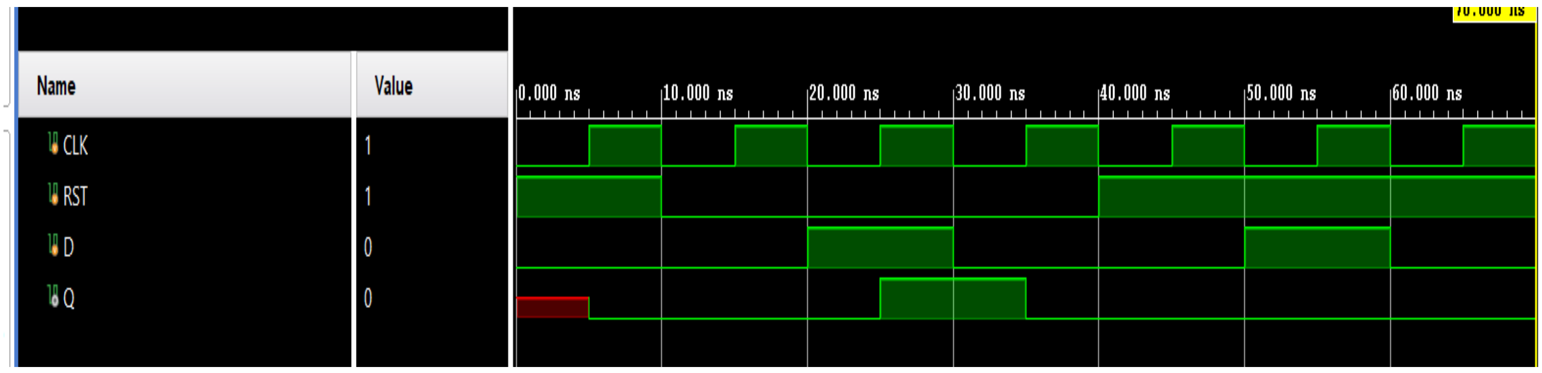
Design Source

```
module DFF_sync
(
    input clk,
    input rst,
    input D,
    output Q
);

    reg Q_reg;

    always @(posedge clk)
begin
        if(rst==1'b1) begin
            Q_reg <= 1'b0;
        end
        else begin
            Q_reg <= D;
        end
    end
    assign Q = Q_reg;
```

Simulation Source

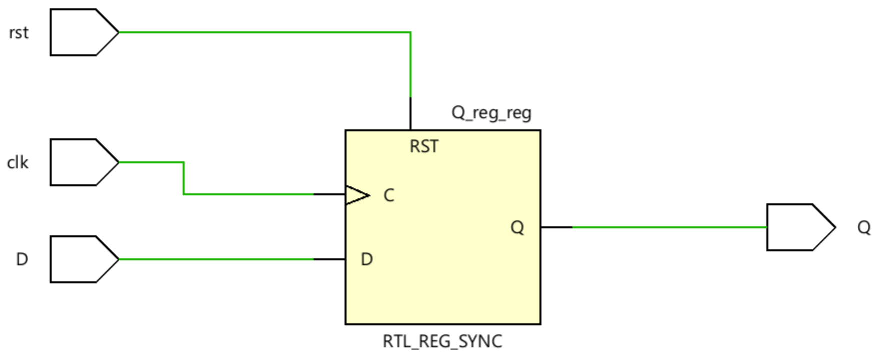
```
module DFF_sync_tb();

    reg CLK= 1'b0;
    reg RST= 1'b0;
    reg D= 1'b0;
    wire Q;

    DFF_sync uut (
        .clk(CLK),
        .rst(RST),
        .D(D),
        .Q(Q)
    );

    always begin
        #5 CLK = ~CLK;
    end


    initial
    begin
        RST = 1;
        #10;
        RST = 0;
        #10;
        D = 1;
        #10;
        D = 0;
        #10;
        RST = 1;
        #10;
        D = 1;
        #10;
        D = 0;
        #10;
        $finish;
    end
```

Simulation Wave

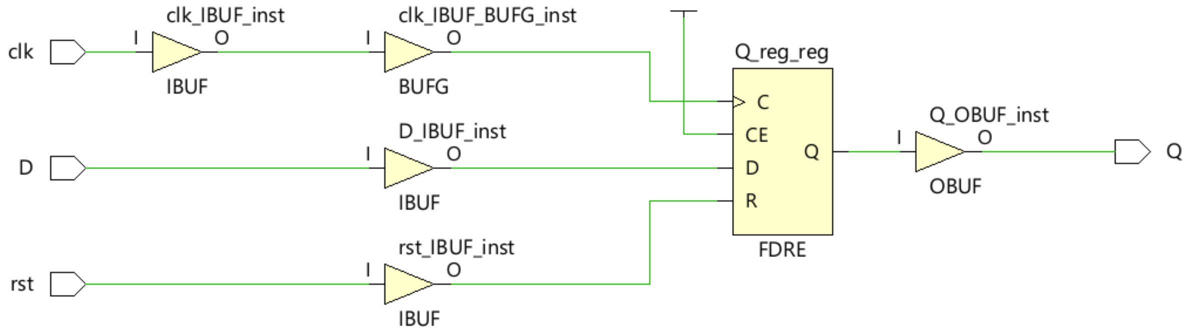


As seen in the simulation, the module works as it should. Reset signal depend on the rising edge of clock signal. If the reset signal is logical, Q will be logical 0. Otherwise, Q will be loaded with the value D, which is the rising edge of the clock signal.

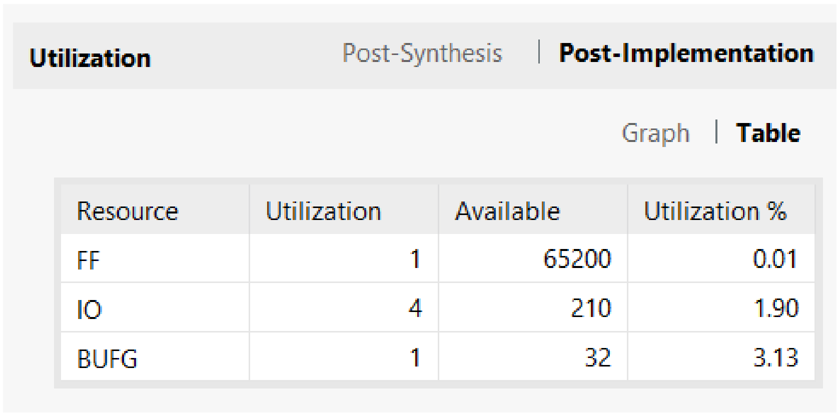
RTL Schematic



Technology Schematic



Utilization Report



Resource	Utilization	Available	Utilization %
FF	1	65200	0.01
IO	4	210	1.90
BUFG	1	32	3.13

DFF With Active – Low Asynchronous Reset

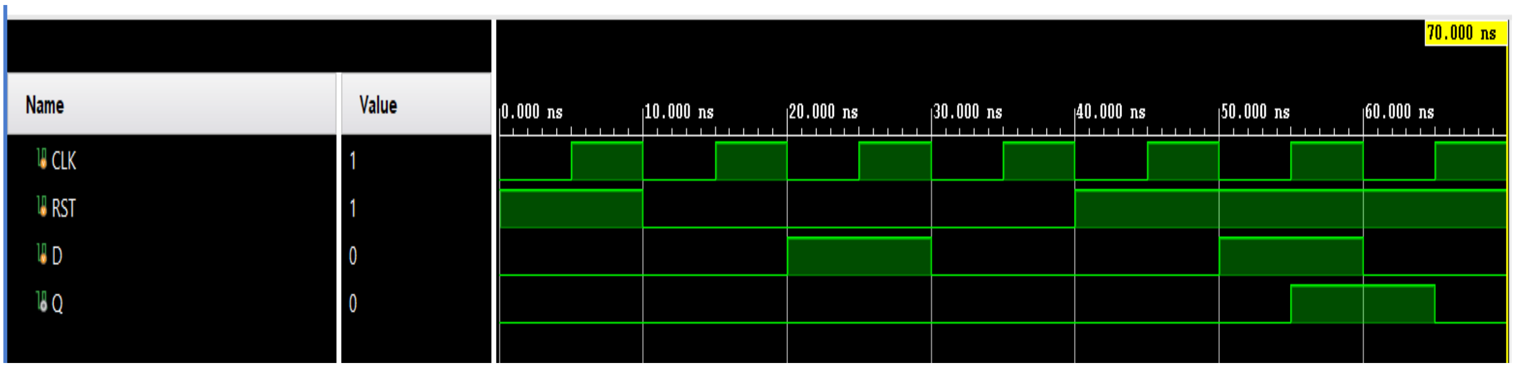
Design Source

```
module DFF_async
(
    input clk,
    input rst,
    input D,
    output Q
);

    reg Q_reg;

    always @(posedge clk, negedge
rst) begin
        if(rst==1'b0) begin
            Q_reg <= 1'b0;
        end
        else begin
            Q_reg <= D;
        end
    end
    assign Q = Q_reg;
endmodule
```

Simulation Source

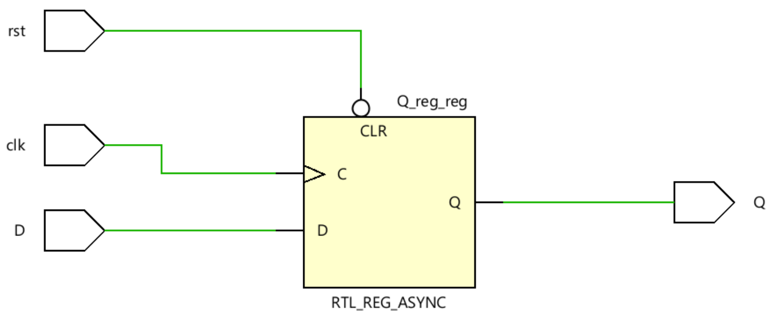
```
module DFF_async_tb();

    reg CLK= 1'b0;
    reg RST= 1'b0;
    reg D= 1'b0;
    wire Q;

    DFF_async uut (
        .clk(CLK),
        .rst(RST),
        .D(D),
        .Q(Q)
    );

    always begin
        #5 CLK = ~CLK;
    end

    initial
    begin
        RST = 1;
        #10;
        RST = 0;
        #10;
        D = 1;
        #10;
        D = 0;
        #10;
        RST = 1;
        #10;
        D = 1;
        #10;
        D = 0;
        #10;
        $finish;
    end
endmodule
```

Simulation Wave

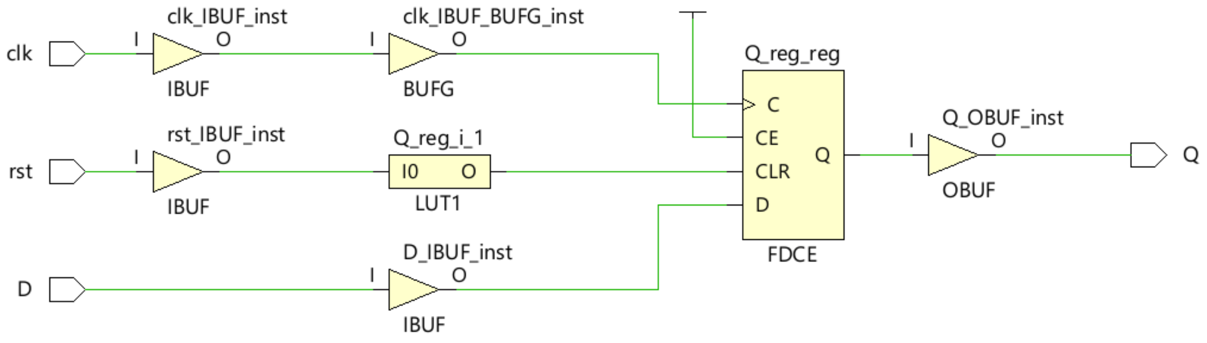


As seen in the simulation, the module works as it should. The reset signal is independent of the rising edge of the clock signal. Output Q can be switched when there is a rising edge of the clock signal and a falling edge of the reset signal. If the reset signal is logic 0, Q will be logic 0. Otherwise, Q will be loaded with input D value.

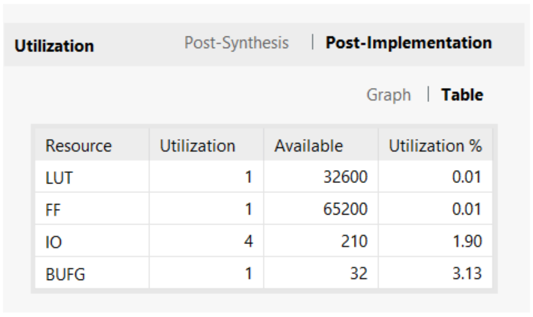
RTL Schematic



Technology Schematic



Utilization Report



Utilization | Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	1	32600	0.01
FF	1	65200	0.01
IO	4	210	1.90
BUFG	1	32	3.13

Both synchronous and asynchronous circuits have the same resource usage with one exception. Asynchronous reset is an active low reset so it uses an extra LUT.

8 – BIT SHIFT REGISTER

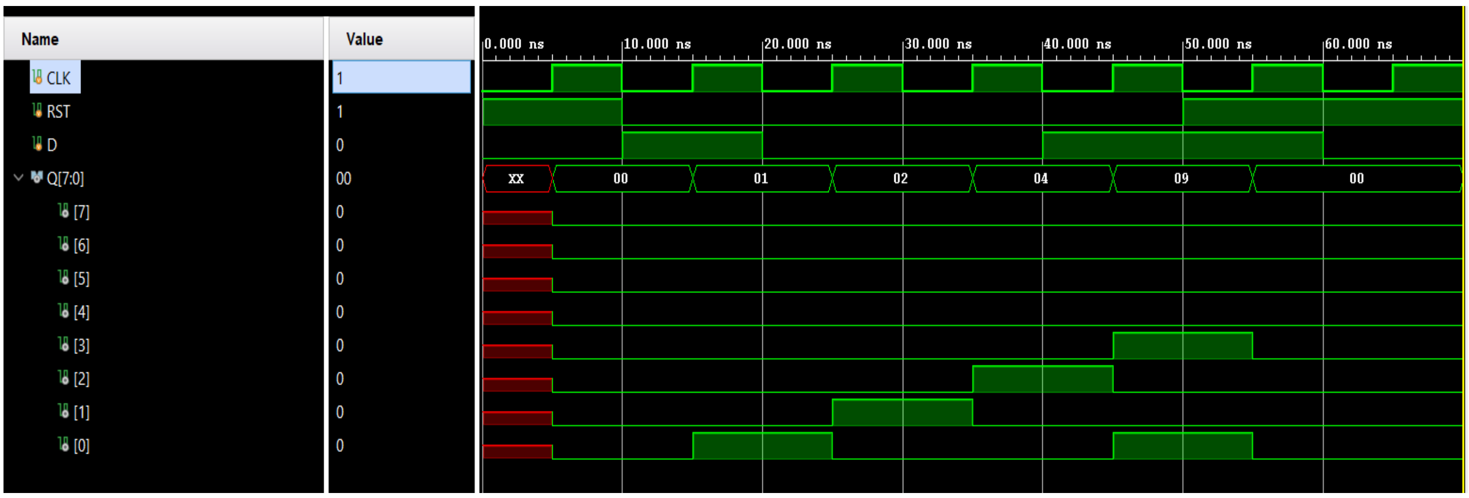
Design Source

```
module shift8
(
    input clk,
    input rst,
    input D,
    output [7:0] Q
);

    reg [7:0] Q_reg;

    always @(posedge clk) begin
        if(rst==1'b1) begin
            Q_reg <= 8'b0;
        end
        else begin
            Q_reg <=
{Q_reg[6:0],D};
        end
    end
    assign Q = Q_reg;
endmodule
```

Simulation Source

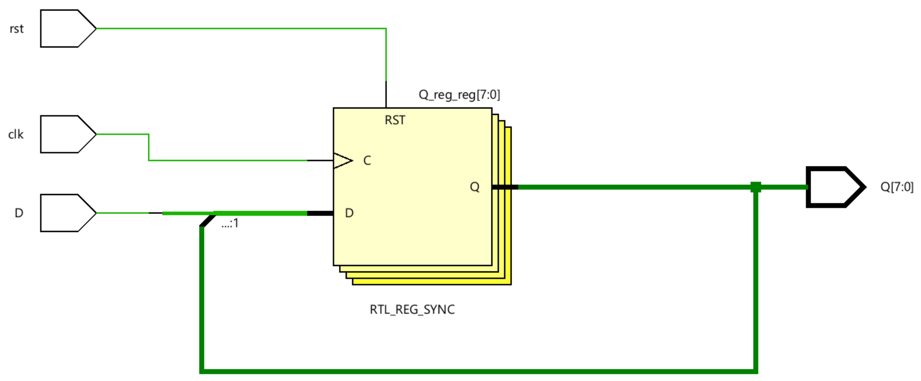
```
module shift8_tb();

        reg CLK=1'b0;
        reg RST=1'b0;
        reg D=1'b0;
        wire [7:0] Q;


        shift8 uut (
                .clk(CLK),
                .rst(RST),
                .D(D),
                .Q(Q)
        );


        always begin
                #5 CLK = ~CLK;
        end


        initial
        begin
                RST = 1;
                #10 RST = 0;

                D = 1;
                #10;

                D = 0;
                #10;

                D = 0;
                #10;

                D = 1;
                #10;

                RST = 1;

                D = 1;
                #10;

                D = 0;
                #10;
                $finish;
        end
endmodule
```

Simulation Wave

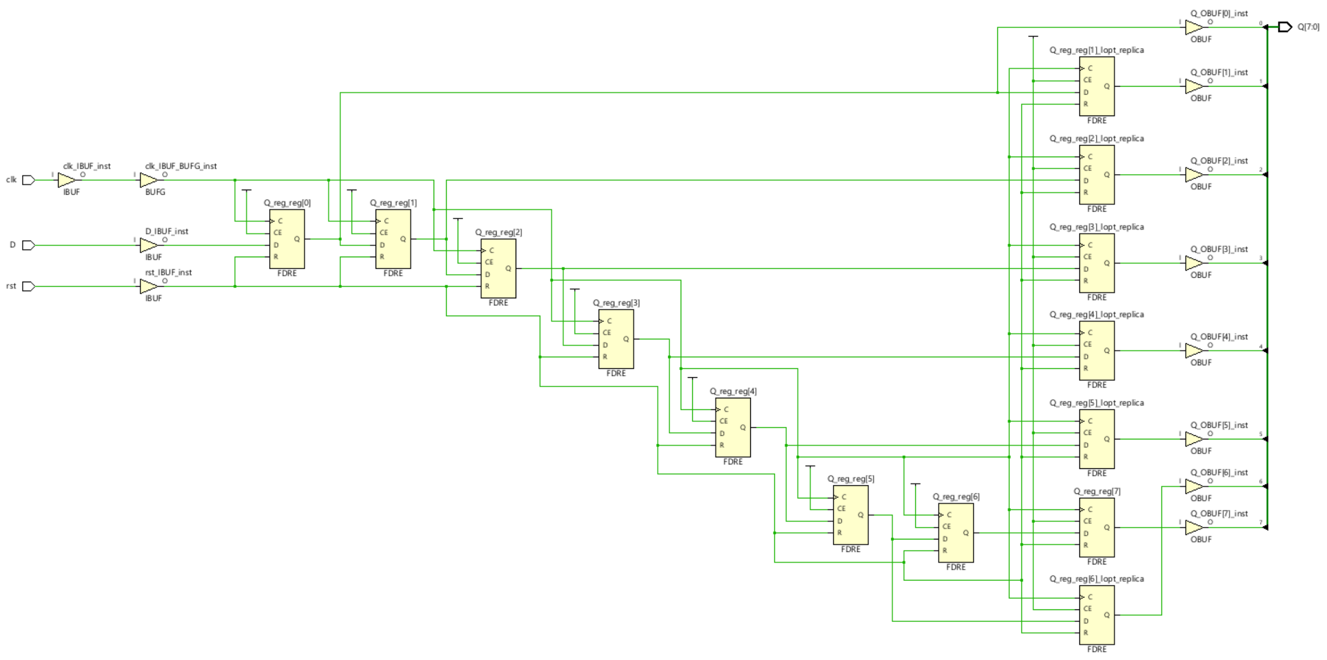


As seen in the simulation, the module works as it should. When a clock signal is given, the least significant bit of Q is loaded via input D and the outputs are shifted one bit towards the most significant bit. When there is a rising edge of the clock signal, reset is logic 1, Q outputs have logic 0 value.

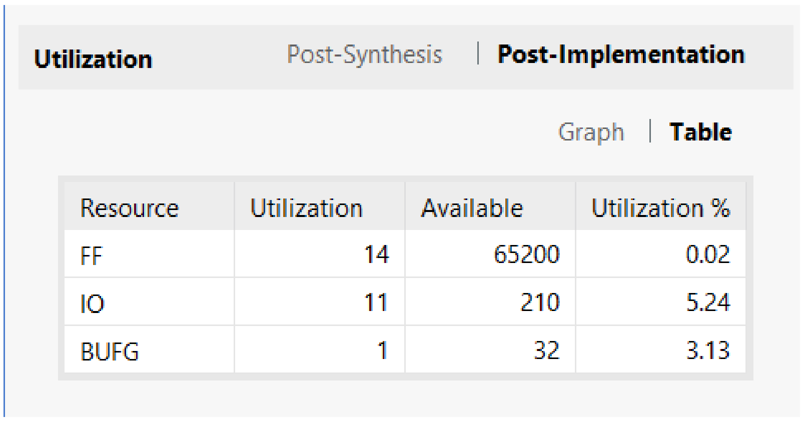
RTL Schematic



Technology Schematic



Utilization Report



Resource	Utilization	Available	Utilization %
FF	14	65200	0.02
IO	11	210	5.24
BUFG	1	32	3.13

CLOCK DIVIDER

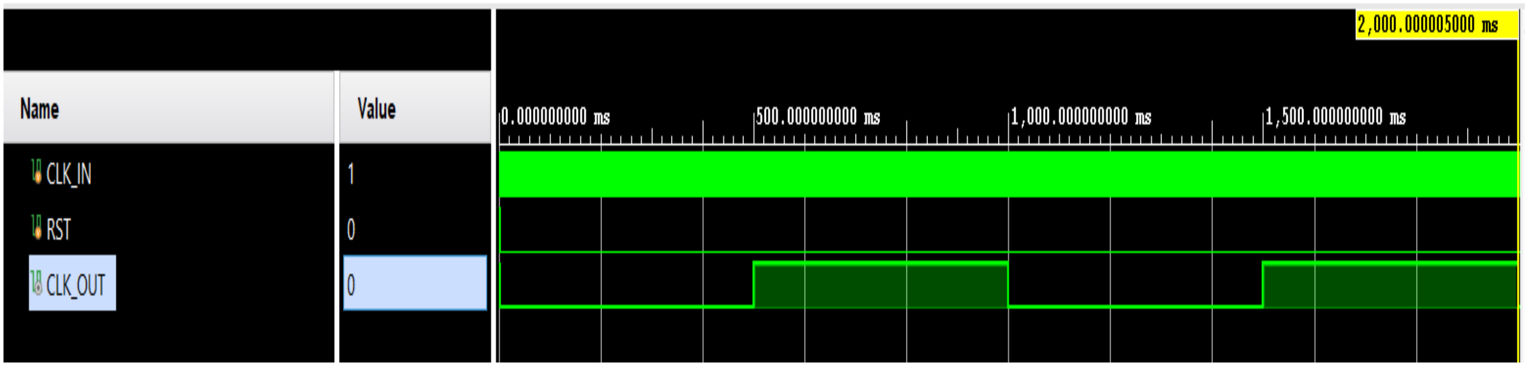
Design Source

```
module clk_divider #(parameter [27:0] CLK_DIV = 100)
(
    input clk_in,
    input rst,
    output clk_out
);

    reg clk_out_reg;
    reg [27:0] counter;

    always @(posedge clk_in) begin
        if(rst==1'b1) begin
            clk_out_reg <= 1'b0;
            counter <= 28'b0;
        end
        else begin
            if(counter ==((CLK_DIV/2)-1))begin
                clk_out_reg <= ~clk_out;
                counter <= 28'b0;
            end
            else begin
                counter <= counter + 1;
            end
        end
    end
    assign clk_out = clk_out_reg;
endmodule
```

Simulation Source and Simulation Wave

Simulation for 1Hz clock

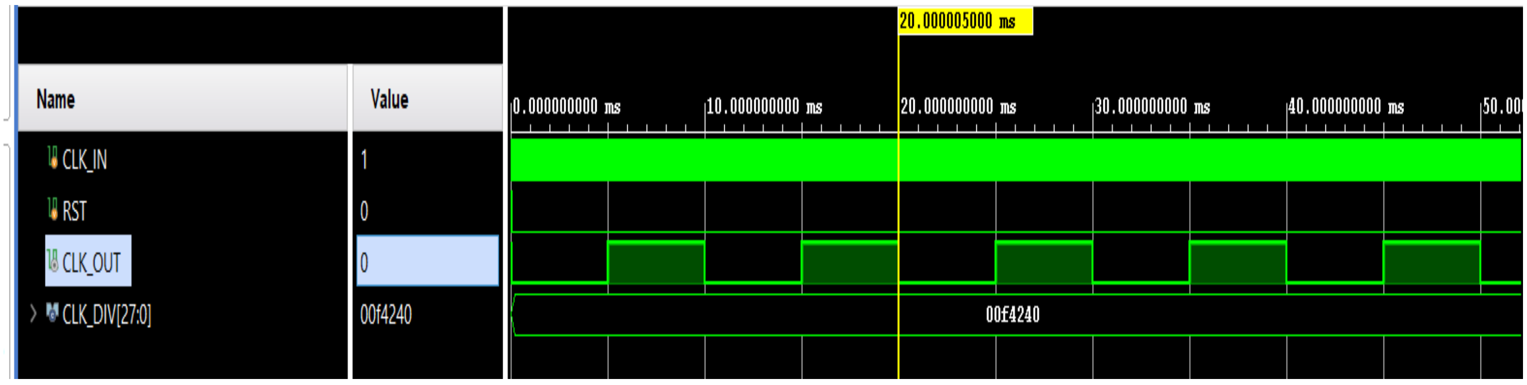
```
module clk_divider_tb();


    parameter [27:0] CLK_DIV = 100000000; //clk_out = 1Hz
    reg CLK_IN=1'b0;
    reg RST=1'b0;
    wire CLK_OUT;

    clk_divider #(.CLK_DIV(CLK_DIV)) uut
    (
        .clk_in(CLK_IN),
        .rst(RST),
        .clk_out(CLK_OUT)
    );

    always begin
    #5 CLK_IN = ~CLK_IN;
    end

    initial begin


        RST = 1;
        #10;
        RST = 0;

        #2000;
        $finish;

    end
endmodule
```



We have a 100 MHz (10^8 Hz) internal clock frequency. If we want to have 1 Hz clock frequency, we have to divide the internal clock frequency by 10^8. The period of a 1Hz signal is 1 second. We have a simulation interval of 2000 ms(2 s). Therefore, there are two periods of 1Hz clock period in the wave.

Simulation for 100 Hz clock

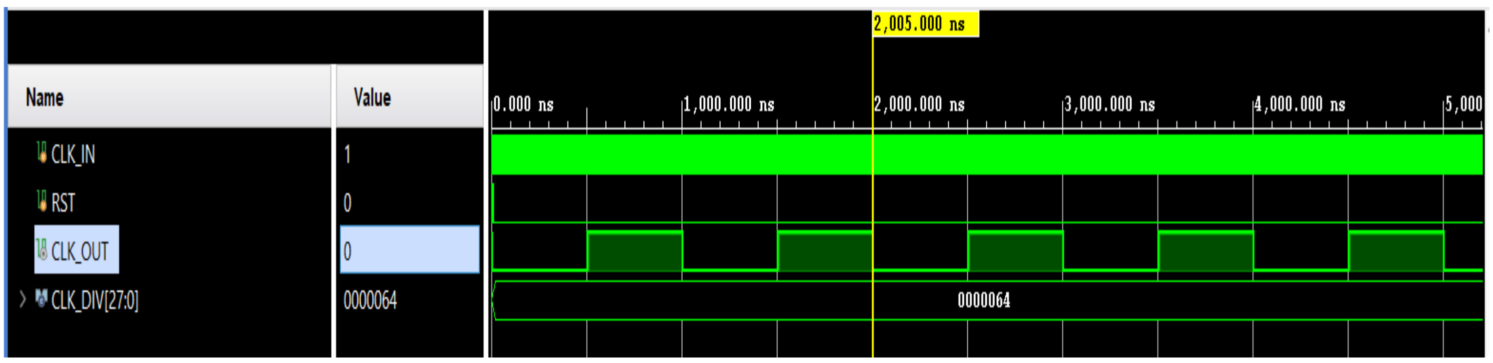
```
module clk_divider_tb();

    parameter [27:0] CLK_DIV = 1000000; //clk_out = 100Hz
    reg CLK_IN=1'b0;
    reg RST=1'b0;
    wire CLK_OUT;

    clk_divider #(.CLK_DIV(CLK_DIV)) uut
    (
        .clk_in(CLK_IN),
        .rst(RST),
        .clk_out(CLK_OUT)
    );

    always begin
    #5 CLK_IN = ~CLK_IN;
    end

    initial begin

        RST = 1;
        #10;
        RST = 0;

        #2000;
        $finish;
    end
endmodule
```



We have a 100 MHz (10^8 Hz) internal clock frequency. If we want to have 100 Hz clock frequency, we have to divide the internal clock frequency by 10^6. The period of a 100Hz signal is 0.01(10 ms) second. We have a simulation interval of 20 ms. Therefore, there are two periods of 100Hz clock period in the wave.

Simulation for 1MHz clock

```
module clk_divider_tb();

    parameter [27:0] CLK_DIV = 100; //clk_out = 1MHz
    reg CLK_IN=1'b0;
    reg RST=1'b0;
    wire CLK_OUT;

    clk_divider #(.CLK_DIV(CLK_DIV)) uut
    (
        .clk_in(CLK_IN),
        .rst(RST),
        .clk_out(CLK_OUT)
    );

    always begin
    #5 CLK_IN = ~CLK_IN;
    end

    initial begin

        RST = 1;
        #10;
        RST = 0;

        #2000;
        $finish;
    end
endmodule
```



We have a 100 MHz (10^8 Hz) internal clock frequency. If we want to have 1 MHz clock frequency, we have to divide the internal clock frequency by 10^2. The period of a 1MHz signal is 10^-6(100 us = 1000 ns) second. We have a simulation interval of 2000 ns. Therefore, there are two periods of 1MHz clock period in the wave.

STOP_WATCH

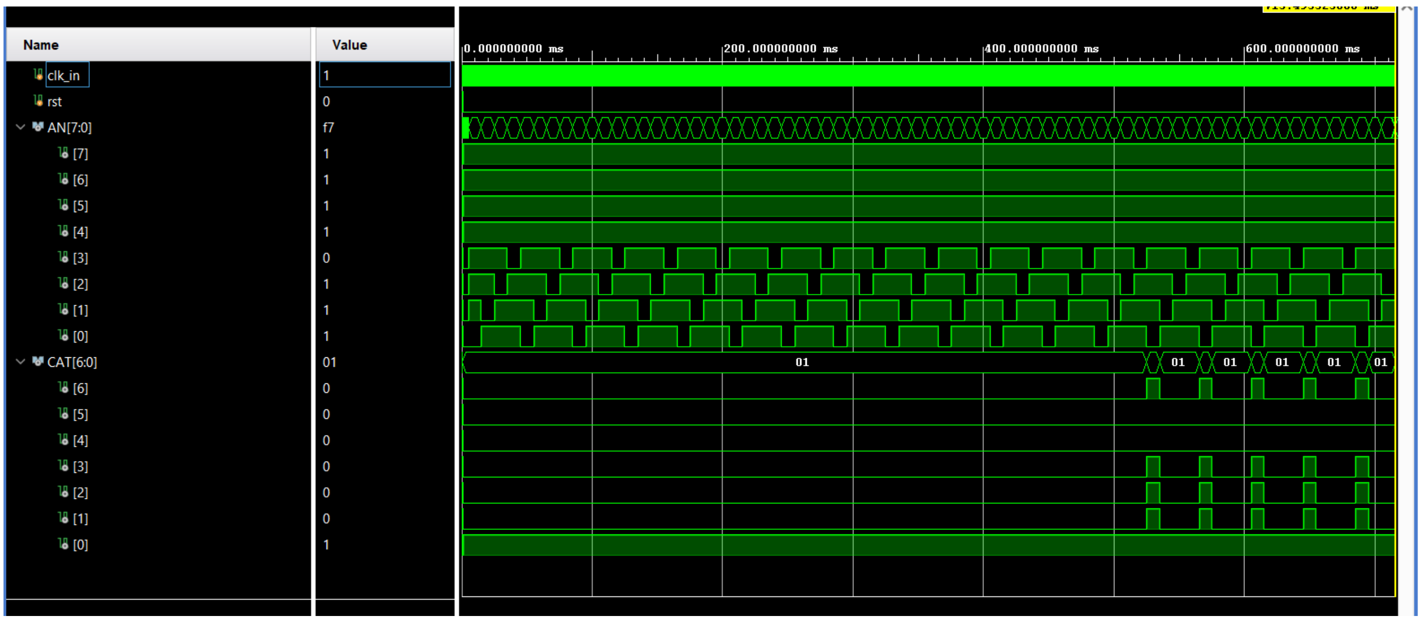
Design Source

```
module stopwatch(
    input               clk_in,
    input               rst,
    output reg [7:0]    AN,         //
Anode   : select 7-segment displays
    output reg [6:0]    CAT         //
Catode : select 7-segment LEDs
);

    wire         clk_out1;
// 1sec clk
    wire         clk_out100;
// 10msec clk (100Hz)

    reg   [13:0] cnt;
// 1sec timer
    wire  [15:0] cnt_bcd;
// BCD converted
    reg   [3:0]  display;
// display number
    reg   [1:0]  refresh_cnt;
// refresh counter



//////////////////////////////////
/////////////
    ////**** INSTANTIATE YOUR
CLK_DIVIDERs ****////

//////////////////////////////////
/////////////

    // CLK DIVIDER 1Hz
    clk_divider #(
.CLK_DIV(100000000) )
    CLKDIV1(
        .clk_in (clk_in),
        .rst    (rst),
        .clk_out(clk_out1)
    );

       // CLK DIVIDER 100Hz
    clk_divider #(
.CLK_DIV(1000000) )
    CLKDIV100(
        .clk_in (clk_in),
        .rst    (rst),
        .clk_out(clk_out100)
    );


//////////////////////////////////
//////////////

//////////////////////////////////
//////////////

    // Binary to Decimal
Conversion
    bin2bcd BIN2BCD(
        .bin(cnt),
        .bcd(cnt_bcd)
    );
```

```
    // Timer 1sec
    always @(posedge clk_out1, posedge rst)
    begin
        if( rst )
        begin
            cnt     <= 0;
        end
        else
        begin
            if( cnt == 9999 )
                cnt <= 0;
            else
                cnt <= cnt + 1;
        end
    end

    // Refresh 7-segment displays
    always @(posedge clk_out100, posedge rst)
    begin
        if( rst )
        begin
            AN          <= 8'b1111_0000;
            display     <= 4'b0000;
            refresh_cnt <= 0;
        end
        else
        begin
            case( refresh_cnt )
                2'b00: begin
                    AN          <=
8'b1111_1110;
                    display     <=
cnt_bcd[3:0];
                    refresh_cnt <=
refresh_cnt + 1;
                end
                2'b01: begin
                    AN = 8'b1111_1101;
                    display     <=
cnt_bcd[7:4];
                    refresh_cnt <=
refresh_cnt + 1;
                end
                2'b10: begin
                    AN = 8'b1111_1011;
                    display     <=
cnt_bcd[11:8];
                    refresh_cnt <=
refresh_cnt + 1;
                end
                2'b11: begin
                    AN = 8'b1111_0111;
                    display     <=
cnt_bcd[15:12];
                    refresh_cnt <=
refresh_cnt + 1;
                end
            endcase
          end
    end

    // Select 7-segment LEDs
    always @(*)
    begin
         case(display)
             4'b0000: CAT = 7'b0000001; //
"0"
             4'b0001: CAT = 7'b1001111; //
"1"
             4'b0010: CAT = 7'b0010010; //
"2"
             4'b0011: CAT = 7'b0000110; //
"3"
             4'b0100: CAT = 7'b1001100; //
"4"
             4'b0101: CAT = 7'b0100100; //
"5"
             4'b0110: CAT = 7'b0100000; //
"6"
             4'b0111: CAT = 7'b0001111; //
"7"
             4'b1000: CAT = 7'b0000000; //
"8"
             4'b1001: CAT = 7'b0000100; //
"9"
             default: CAT = 7'b0000001; //
"0"
         endcase
    end
endmodule
```

Simulation Source

```
module stopwatch_tb();

    reg  clk_in = 0;
    reg  rst    = 0;
    wire [7:0]  AN;         // Anode   : select 7-segment displays
    wire [6:0]  CAT;        // Catode  : select 7-segment LEDs

    stopwatch STW(
        .clk_in(clk_in),
        .rst(rst),
        .AN(AN),
        .CAT(CAT)
    );

    always #5 clk_in = ~clk_in;

    initial
    begin
        repeat(100) @(posedge clk_in);

        rst = 1;
        repeat(100) @(posedge clk_in);
        rst = 0;
        repeat(100) @(posedge clk_in);
    end

endmodule
```



As seen in the simulation, the module works as it should. Cathode and anode signals change respectively.

SLIDING LEDs

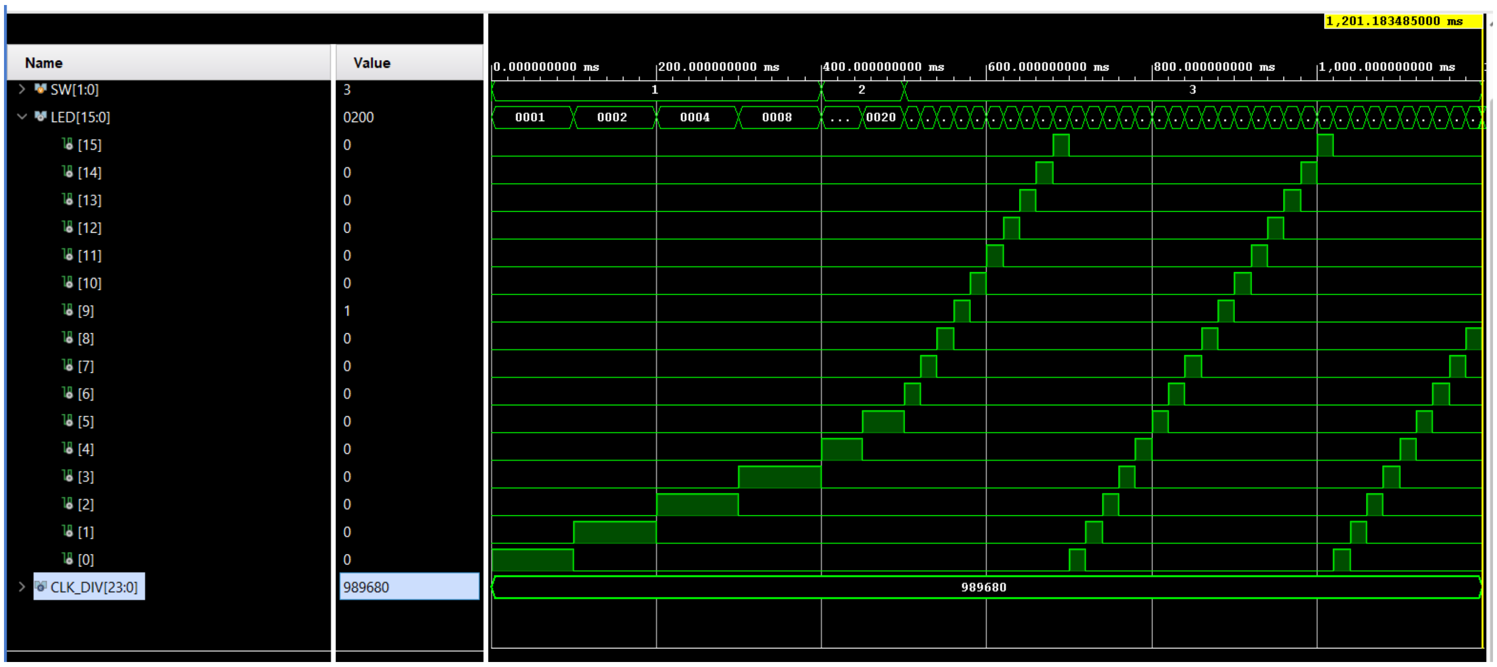
Design Source

```
module sliding_leds #(parameter [23:0] CLK_DIV = 10000000)
(
    input clk,
    input rst,
    input [1:0] SW,
    output [15:0] LED
);

    reg [23:0] counter;
    reg [15:0] led_reg;

    always @(posedge clk, posedge rst) begin
        if(rst==1'b1) begin
            counter <= 24'b0;
            led_reg <= 16'b0000000000000001;
        end
        else begin
            case(SW)
                2'b00:  led_reg <= led_reg;

                2'b01: begin
                    if(counter == CLK_DIV -1) begin
                        counter <= 24'b0;
                        if(led_reg == 16'b1000000000000000) begin
                            led_reg <= 16'b0000000000000001;
                        end
                        else begin
                            led_reg <= led_reg << 1;
                        end
                    end
                    else begin
                        counter <= counter + 1;
                    end
                end

                2'b10: begin
                    if(counter == (CLK_DIV/2) -1) begin
                        counter <= 24'b0;
                        if(led_reg == 16'b1000000000000000) begin
                            led_reg <= 16'b0000000000000001;
                        end
                        else begin
                            led_reg <= led_reg<<1;
                        end
                    end
                    else begin
                        counter <= counter + 1;
                    end
                end

                2'b11: begin
                    if(counter == (CLK_DIV/5) -1) begin
                        counter <= 24'b0;
                        if(led_reg == 16'b1000000000000000) begin
                            led_reg <= 16'b0000000000000001;
                        end
                        else begin
                            led_reg <= led_reg<<1;
                        end
                    end
                    else begin
                        counter <= counter + 1;
                    end
                end

                default:  led_reg <= led_reg;
            endcase
        end
    end

    assign LED = led_reg;

endmodule
```

Simulation Source

```
module sliding_leds_tb();

    parameter [23:0] CLK_DIV = 10000000;
    reg CLK = 1'b0;
    reg RST=1'b1;
    reg [1:0] SW = 2'b00;
    wire [15:0] LED;

    sliding_leds #(.CLK_DIV(CLK_DIV)) uut
    (
        .clk(CLK),
        .rst(RST),
        .SW(SW),
        .LED(LED)
    );

    always begin
        #5 CLK = ~CLK;  // 100MHz
    end

    initial
    begin
        RST = 1;
        #10;
        RST = 0;
        #10;

        SW = 2'b01;
        #400000000;
        SW = 2'b10;
        #100000000;
        SW = 2'b11;
        #100000000;
        $finish;
    end
endmodule
```

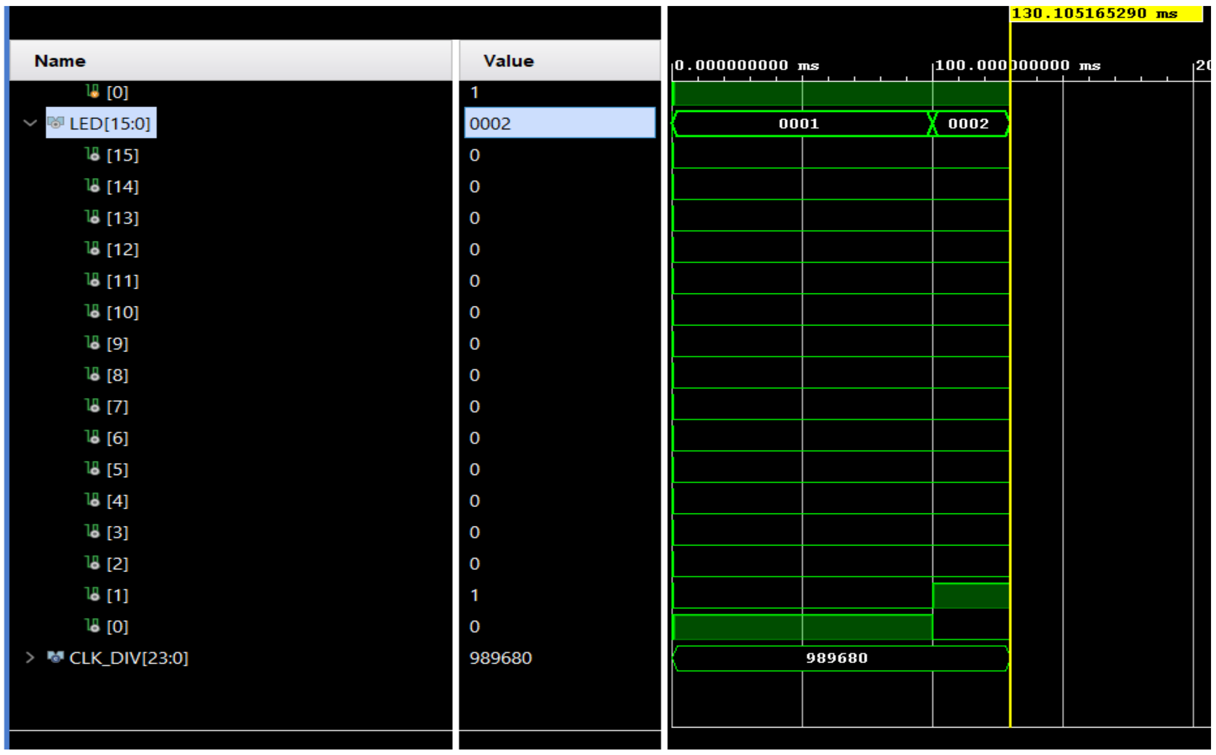
Simulation Wave



Name	Value
SW[1:0]	3
LED[15:0]	0200
[15]	0
[14]	0
[13]	0
[12]	0
[11]	0
[10]	0
[9]	1
[8]	0
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0
[0]	0
CLK_DIV[23:0]	989680

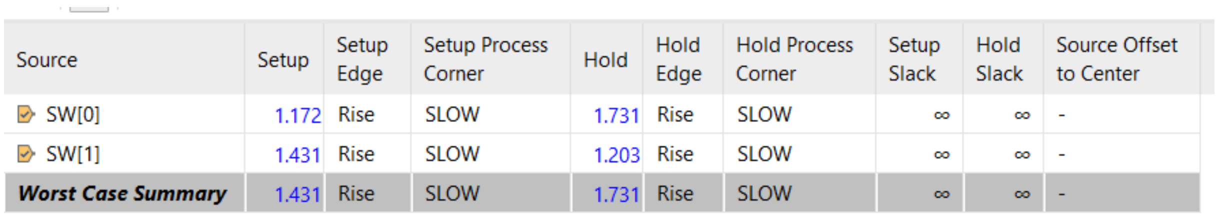
As seen in the simulation, the module works as it should. When SW = 01, we have 10 Hz frequency and 0.1 second period. Therefore, we have four logic 1 cycle which have 0.1 second period. When SW = 10, we have 20 Hz frequency and 0.05 second period. Therefore, we have two logic 1 cycle which have 0.05 second period. When SW = 11, we have 50 Hz frequency and 0.02 second period. Therefore, we have thirty four logic 1 cycle which have 0.1 second period

Post-Implementation Timing Simulation



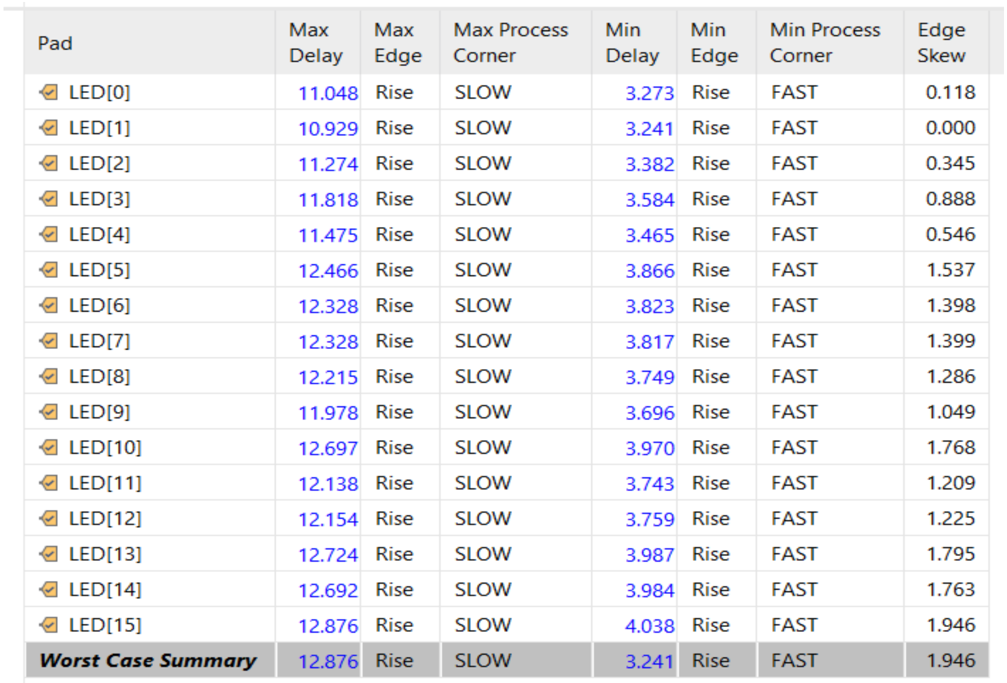
Name	Value
[0]	1
LED[15:0]	0002
[15]	0
[14]	0
[13]	0
[12]	0
[11]	0
[10]	0
[9]	0
[8]	0
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	1
[0]	0
CLK_DIV[23:0]	989680

Timing Report



Source	Setup	Setup Edge	Setup Process Corner	Hold	Hold Edge	Hold Process Corner	Setup Slack	Hold Slack	Source Offset to Center
SW[0]	1.172	Rise	SLOW	1.731	Rise	SLOW	∞	∞	-
SW[1]	1.431	Rise	SLOW	1.203	Rise	SLOW	∞	∞	-
Worst Case Summary	1.431	Rise	SLOW	1.731	Rise	SLOW	∞	∞	-

This photo shows setup and hold times for inputs. SW[1] have the worst case for setup/hold time compared with SW[0].



Pad	Max Delay	Max Edge	Max Process Corner	Min Delay	Min Edge	Min Process Corner	Edge Skew
LED[0]	11.048	Rise	SLOW	3.273	Rise	FAST	0.118
LED[1]	10.929	Rise	SLOW	3.241	Rise	FAST	0.000
LED[2]	11.274	Rise	SLOW	3.382	Rise	FAST	0.345
LED[3]	11.818	Rise	SLOW	3.584	Rise	FAST	0.888
LED[4]	11.475	Rise	SLOW	3.465	Rise	FAST	0.546
LED[5]	12.466	Rise	SLOW	3.866	Rise	FAST	1.537
LED[6]	12.328	Rise	SLOW	3.823	Rise	FAST	1.398
LED[7]	12.328	Rise	SLOW	3.817	Rise	FAST	1.399
LED[8]	12.215	Rise	SLOW	3.749	Rise	FAST	1.286
LED[9]	11.978	Rise	SLOW	3.696	Rise	FAST	1.049
LED[10]	12.697	Rise	SLOW	3.970	Rise	FAST	1.768
LED[11]	12.138	Rise	SLOW	3.743	Rise	FAST	1.209
LED[12]	12.154	Rise	SLOW	3.759	Rise	FAST	1.225
LED[13]	12.724	Rise	SLOW	3.987	Rise	FAST	1.795
LED[14]	12.692	Rise	SLOW	3.984	Rise	FAST	1.763
LED[15]	12.876	Rise	SLOW	4.038	Rise	FAST	1.946
Worst Case Summary	12.876	Rise	SLOW	3.241	Rise	FAST	1.946

This photo shows max/min delays for outputs. LED[15] have the worst case in terms of delays. Therefore, the path which is connected to LED[15] is the critical path.

RESEARCH

1)

Timing analysis is one of the most significant part of digital design. Designers can add the project timing constraints. Static Timing Analysis (STA) verify all paths meet the timing constraints and required performance in terms of delays. It calculates delays for all paths and verifies whether constraints are met.

Critical Path: The critical path is the longest path that a signal can take from one flip- flop to another in a synchronous circuit. It is the worst case in terms of delays.

Clock Periods: The duration of one clock cycle, which determines the maximum speed at which the FPGA can operate.

Input/Output Delays: The time taken for signals to propagate from input pins to internal logic and from internal logic to output pins.

Setup/Hold Times: Setup time is the minimum time before the clock edge that data must be stable, while hold time is the minimum time after the clock edge that data must remain stable. ***[2]***

$$T_{clock} \geq T_{setup} + T_{combinational} + T_{clk-comb}$$

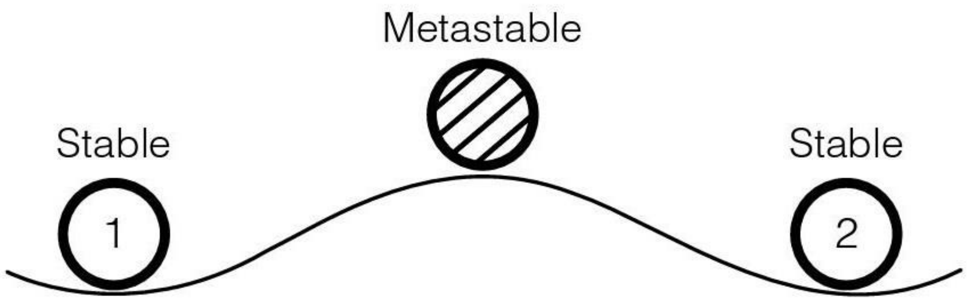
$$f_{max} = 1/T_{clock}$$

2) In digital design, flip-flops have a specified set-up time and hold time. The minimum time before the clocking activity by which the input signal must be stable is called set-up time. The minimum time after the clocking activity, during which the input signal must remain stable, is called hold time. The input should not change during the setup and hold period for stable system. On the contrary, if input changes during setup and hold time, outputs have unknown results. This is called a metastable state. Metastability is the propagation of the metastable state. ***[3]***

A synchronizer typically consists of multiple cascaded flip-flops. This structure allows the signals to be stabilized before the clock signal. This works like buffer.

Increasing setup and hold time can solve metastability problem. However, it can increase period, decrease frequency and it can affect performance of system.

Being able to provide input and clock signals reaching different paths synchronously can also solve the metastability problem. ***[4]***



[4]

REFERENCES

[1] GeeksforGeeks. (2024, May 20). *Latches in digital logic*. GeeksforGeeks. https://www.geeksforgeeks.org/latches-in-digital-logic/

[2] Harvie, L. (2024, July 31). How to perform Static Timing Analysis (STA). *RunTime Recruitment*. https://runtimerec.com/how-to-perform-static-timing-analysis/

[3] Cadence. (2024, August 12). *How to avoid metastability in digital circuits*. Advanced PCB Design Blog | Cadence. https://resources.pcb.cadence.com/blog/2022-how-to-avoid-metastability-in-digital-circuits

[4] Pandey, P. (2023, June 14). *(7) Unveiling metastability in VLSI: Taming the Unpredictable Beast! | LinkedIn*. https://www.linkedin.com/pulse/unveiling-metastability-vlsi-taming-unpredictable-beast-priya-pandey/