

**İTÜ**



**DIGITAL SYSTEM DESIGN APPLICATION**

**EHB436E      CRN: 11280**

**Salih Ömer Ongün  
040220780**

**Experiment 6**

# CLOCK GENERATION

## Design Source

```
module clock_gen
(
    input clk,
    input rst,
    output [6:0] cnt100,
    output [6:0] cnt80,
    output [6:0] cnt60
);
    wire clk80;
    wire clk60;
    reg [6:0] cnt100_reg;
    reg [6:0] cnt80_reg;
    reg [6:0] cnt60_reg;

    clk_wiz_0 instance_name
    (
        // Clock out ports
        .clk_out1(clk80),    // output clk_out1 80MHz
        .clk_out2(clk60),    // output clk_out2 60MHz
        // Status and control signals
        .reset(rst), // input reset
        .locked(locked),    // output locked
        // Clock in ports
        .clk_in1(clk)       // input clk_in1
    );

    always @(posedge clk) begin
        if(rst==1'b1) begin
            cnt100_reg <= 7'b0;
        end
        else if(locked ==1'b1) begin
            if(cnt100_reg < 100) begin
                cnt100_reg <= cnt100_reg + 1;
            end
            else begin
                cnt100_reg <= cnt100_reg;
            end
        end
    end
    assign cnt100 = cnt100_reg;

    always @(posedge clk80) begin
        if(rst==1'b1) begin
            cnt80_reg <= 7'b0;
        end
        else if(locked ==1'b1) begin
            if(cnt80_reg < 80) begin
                cnt80_reg <= cnt80_reg + 1;
            end
            else begin
                cnt80_reg <= cnt80_reg;
            end
        end
    end
    assign cnt80 = cnt80_reg;

    always @(posedge clk60) begin
        if(rst==1'b1) begin
            cnt60_reg <= 7'b0;
        end
        else if(locked ==1'b1) begin
            if(cnt60_reg < 60) begin
                cnt60_reg <= cnt60_reg + 1;
            end
            else begin
                cnt60_reg <= cnt60_reg;
            end
        end
    end
    assign cnt60 = cnt60_reg;

endmodule
```

## Simulation Source

```
module clock_gen_tb();

    reg CLK = 1'b0;
    reg RST = 1'b0;
    wire [6:0] CNT100;
    wire [6:0] CNT80;
    wire [6:0] CNT60;

    clock_gen uut
    (
        .clk(CLK),
        .rst(RST),
        .cnt100(CNT100),
        .cnt80(CNT80),
        .cnt60(CNT60)
    );

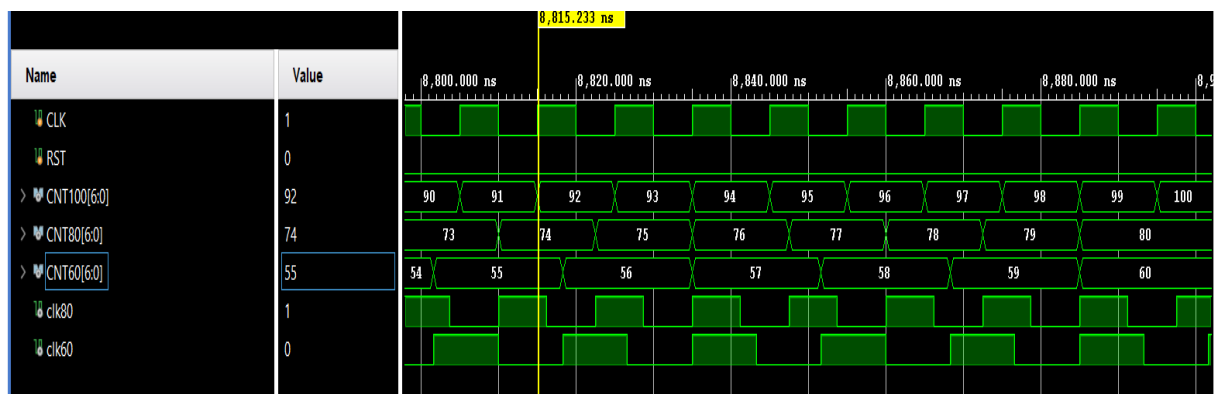
    always begin
        #5 CLK = ~CLK;
    end

    initial begin

        RST = 1;
        #50;
        RST = 0;

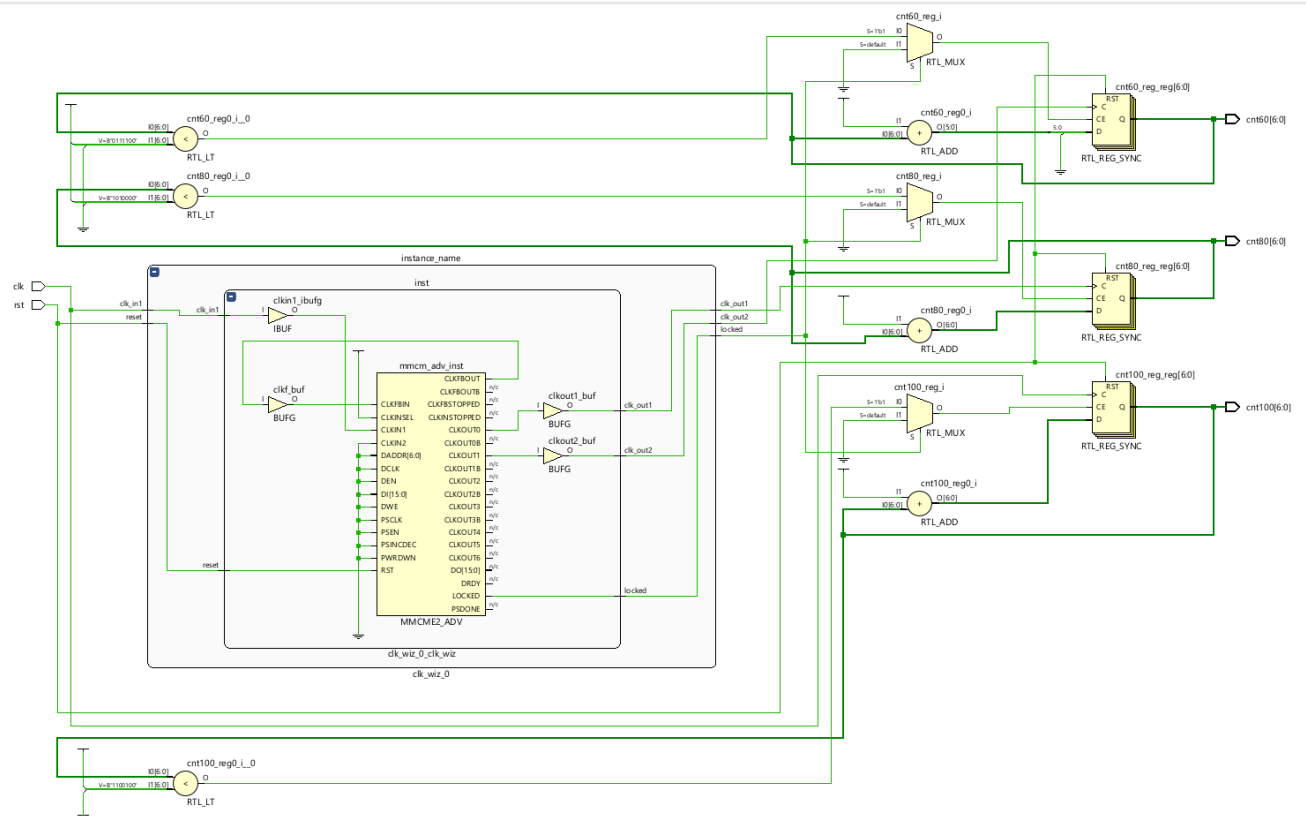
        #2000;
        $finish;
    end
endmodule
```

## Simulation Wave

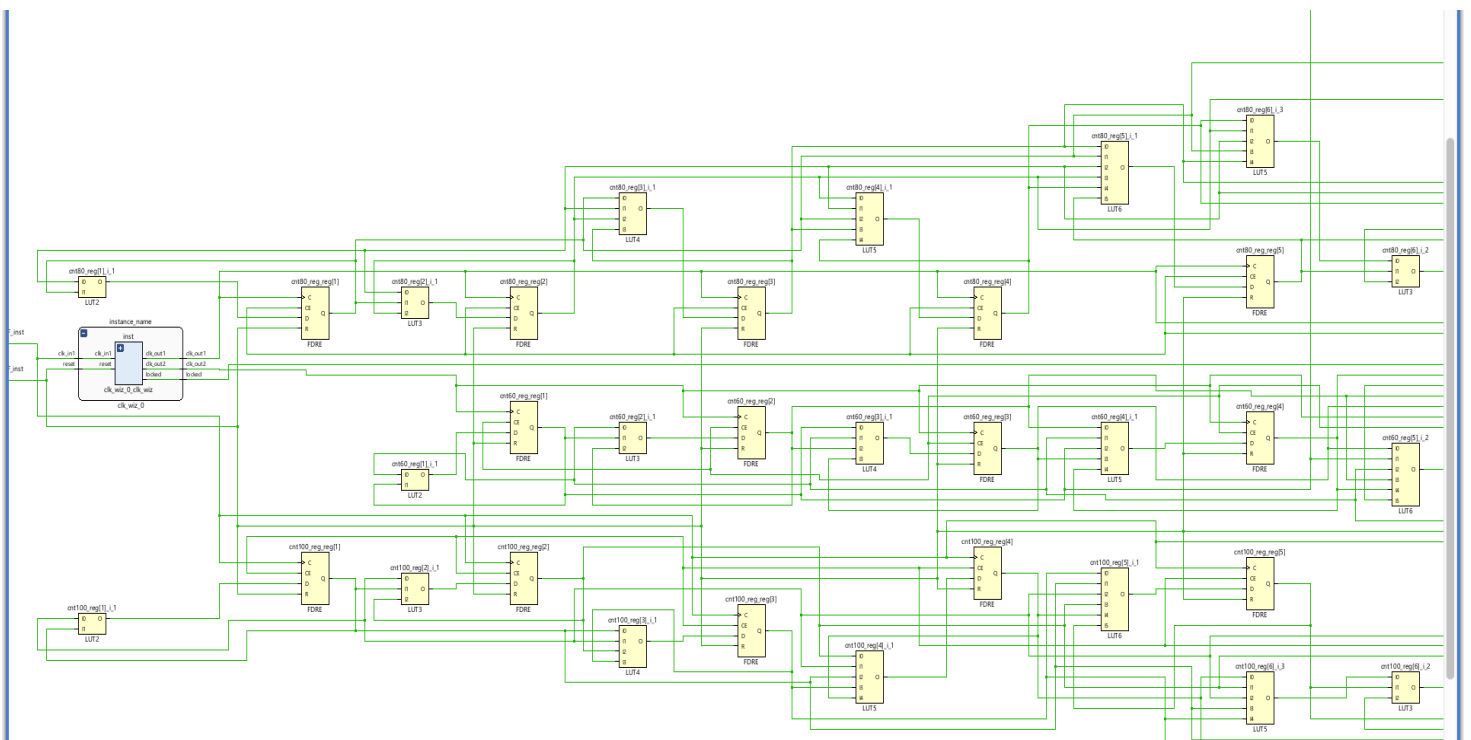


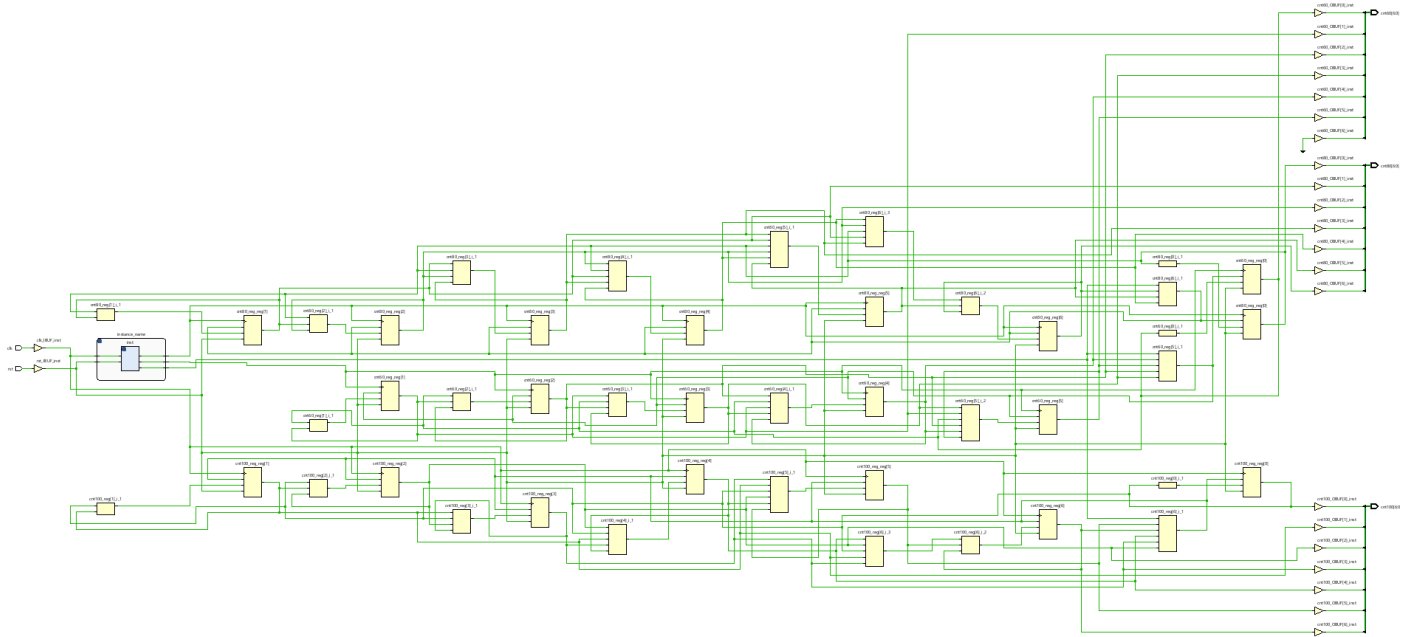
As can be seen in the image, our code is working correctly. The cnt100 counter runs on the rising edge of the clk clock. The cnt60 counter runs on the rising edge of the clk60 clock. The cnt80 counter runs on the rising edge of the clk80 clock. When cnt100 reaches 100, the counter stops. When cnt60 reaches 60, the counter stops. When cnt80 reaches 80, the counter stops.

## RTL Schematic



## Technology Schematic

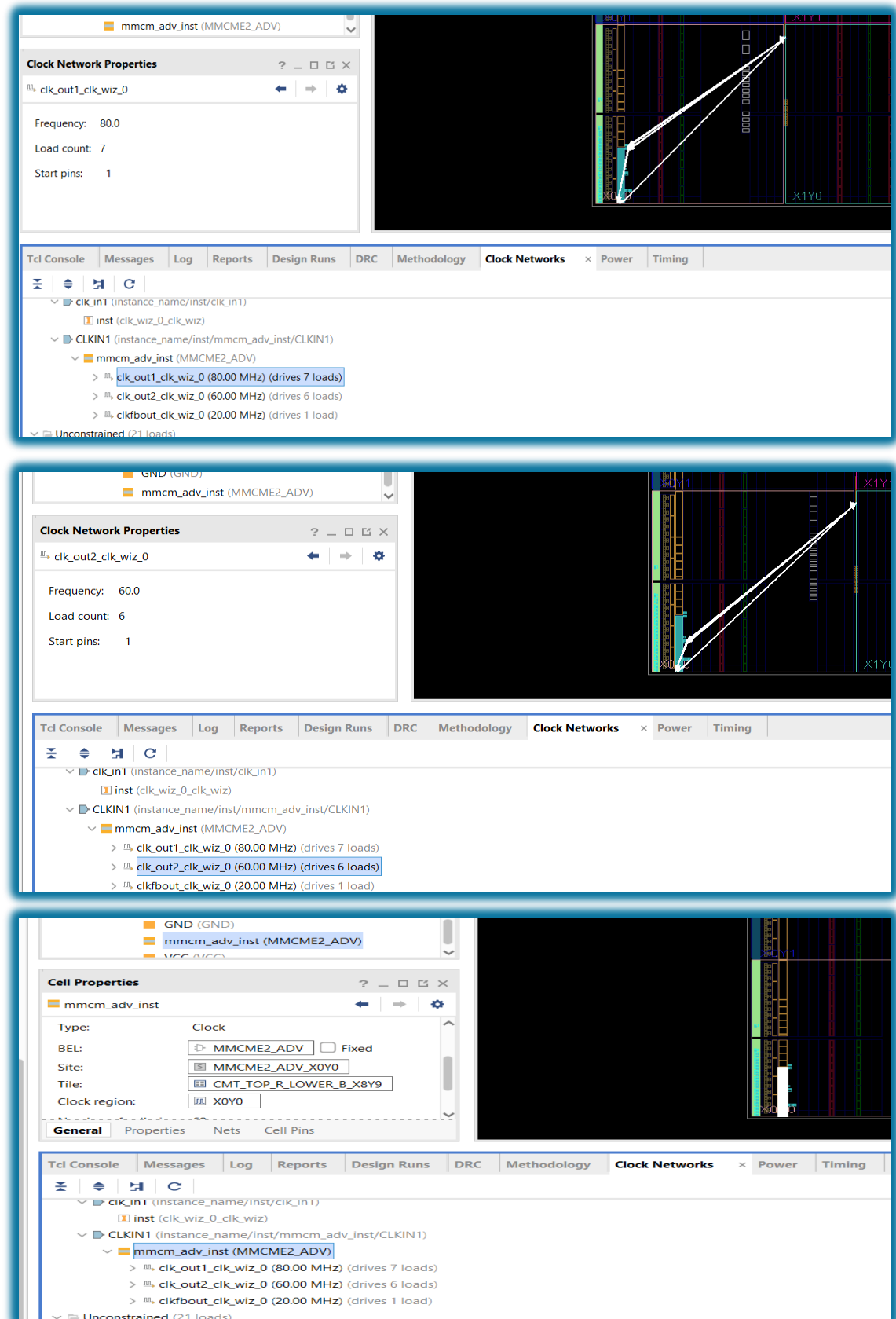




## Utilization Report

Utilization	Post-Synthesis			Post-Implementation	
	Graph			Table	
Resource	Utilization	Available	Utilization %		
LUT	17	32600	0.05		
FF	20	65200	0.03		
IO	23	210	10.95		
BUFG	3	32	9.38		
MMCM	1	5	20.00		

## Clock Network Report



We use 100MHz, 80MHz and 60MHz clocks.

# CLOCK GATING

## Design Source

```
module clock_gating
(
    input clk,
    input rst,
    output [6:0] cnt50,
    output [6:0] cnt25
);

    reg clk50_en, clk25_en;
    wire clk50, clk25;
    reg [6:0] cnt50_reg;
    reg [6:0] cnt25_reg;

    BUFR #(
        // 50MHz
        .BUFR_DIVIDE("2"),
        .SIM_DEVICE("7SERIES")
    )
    BUFR_inst50 (
        .O(clk50),
        .CE(clk50_en),
        .CLR(1'b0),
        .I(clk)
    );

    BUFR #(
        // 25MHz
        .BUFR_DIVIDE("4"),
        .SIM_DEVICE("7SERIES")
    )
    BUFR_inst25 (
        .O(clk25),
        .CE(clk25_en), // 1-bit input: Active high, clock enable (Divided modes only)
        .CLR(1'b0), // 1-bit input: Active high, asynchronous clear (Divided modes only)
        .I(clk)
    );

    always @(posedge clk50, posedge rst) begin
        if(rst==1'b1) begin
            clk50_en <= 1'b1;
            cnt50_reg <= 7'b0;
        end
        else if(clk50_en == 1'b1)begin
            if(cnt50_reg < 50) begin
                cnt50_reg <= cnt50_reg + 1;
            end
            else begin
                clk50_en <= 1'b0;
            end
        end
    end
    assign cnt50 = cnt50_reg;

    always @(posedge clk25, posedge rst) begin
        if(rst==1'b1) begin
            clk25_en <= 1'b1;
            cnt25_reg <= 7'b0;
        end
        else if(clk25_en == 1'b1)begin
            if(cnt25_reg < 25) begin
                cnt25_reg <= cnt25_reg + 1;
            end
            else begin
                clk25_en <= 1'b0;
            end
        end
    end
    assign cnt25 = cnt25_reg;
endmodule
```

## Simulation Source

```
module clock_gating_tb();

    reg CLK = 1'b0;
    reg RST = 1'b0;
    wire [6:0] CNT50;
    wire [6:0] CNT25;

    clock_gating uut
    (
        .clk(CLK),
        .rst(RST),
        .cnt50(CNT50),
        .cnt25(CNT25)
    );

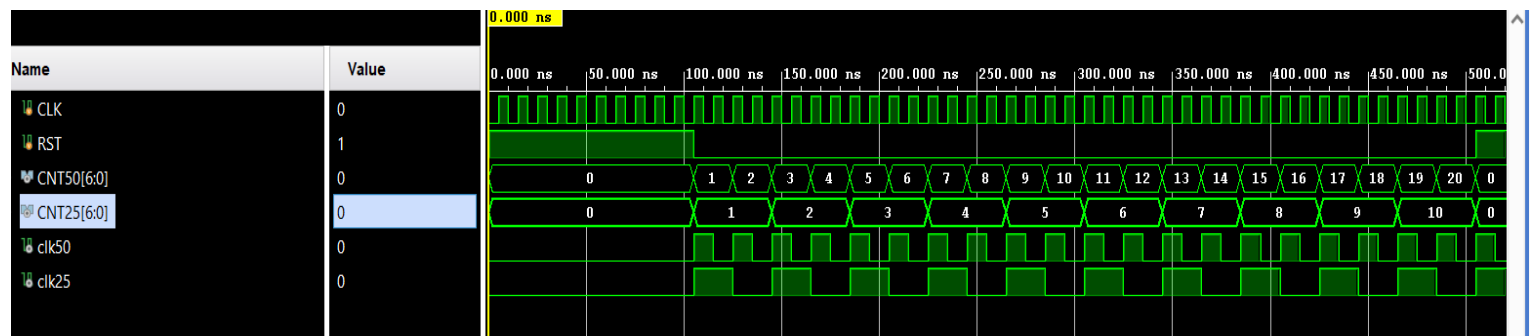
    always begin
        #5 CLK = ~CLK;
    end

    initial begin

        RST = 1;
        #105;
        RST = 0;

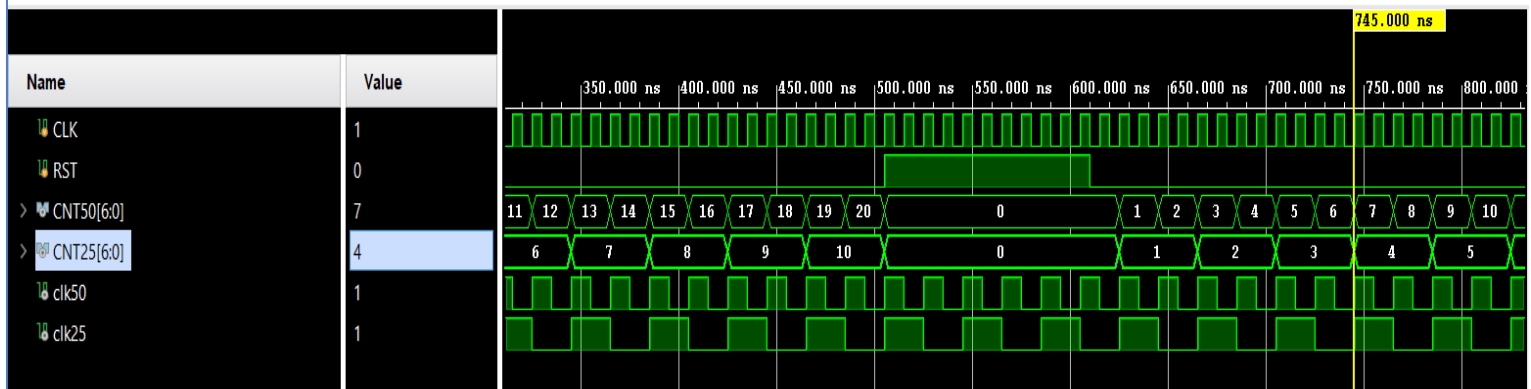
        #800;
        RST = 1;
        #105;
        RST = 0;
        $finish;
    end
endmodule
```

## Simulation Wave

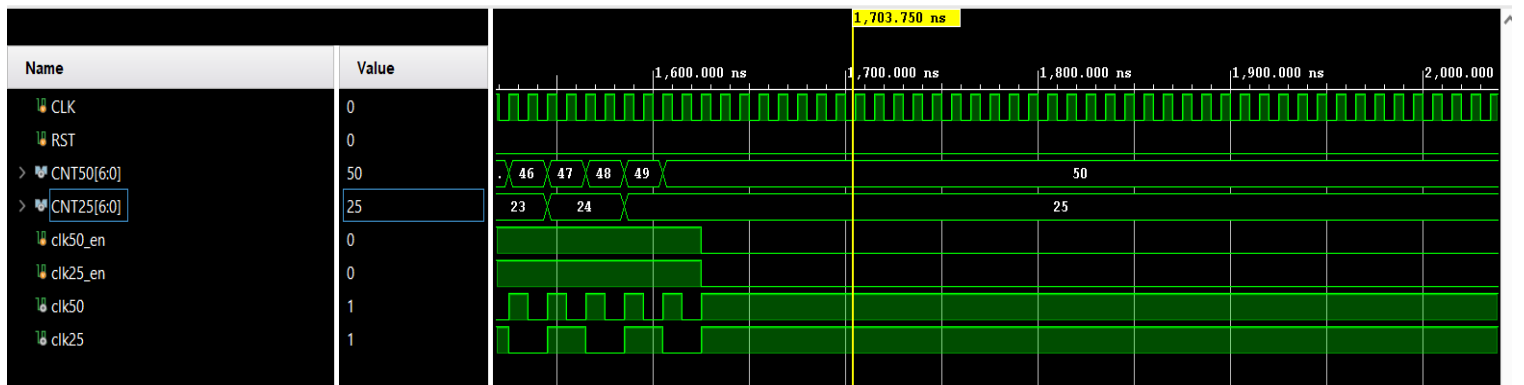


As can be seen in the image, our code is working correctly. Our internal 100MHz clock is available in clk. External clk50(50MHz) and clk25(25MHz) work correctly. The cnt50 counter runs on the rising edge of the clk50 clock. The cnt25 counter runs on the rising edge of the clk25 clock. If we give the rst signal the counters and enable signals are reset. However, the clocks continue to run.



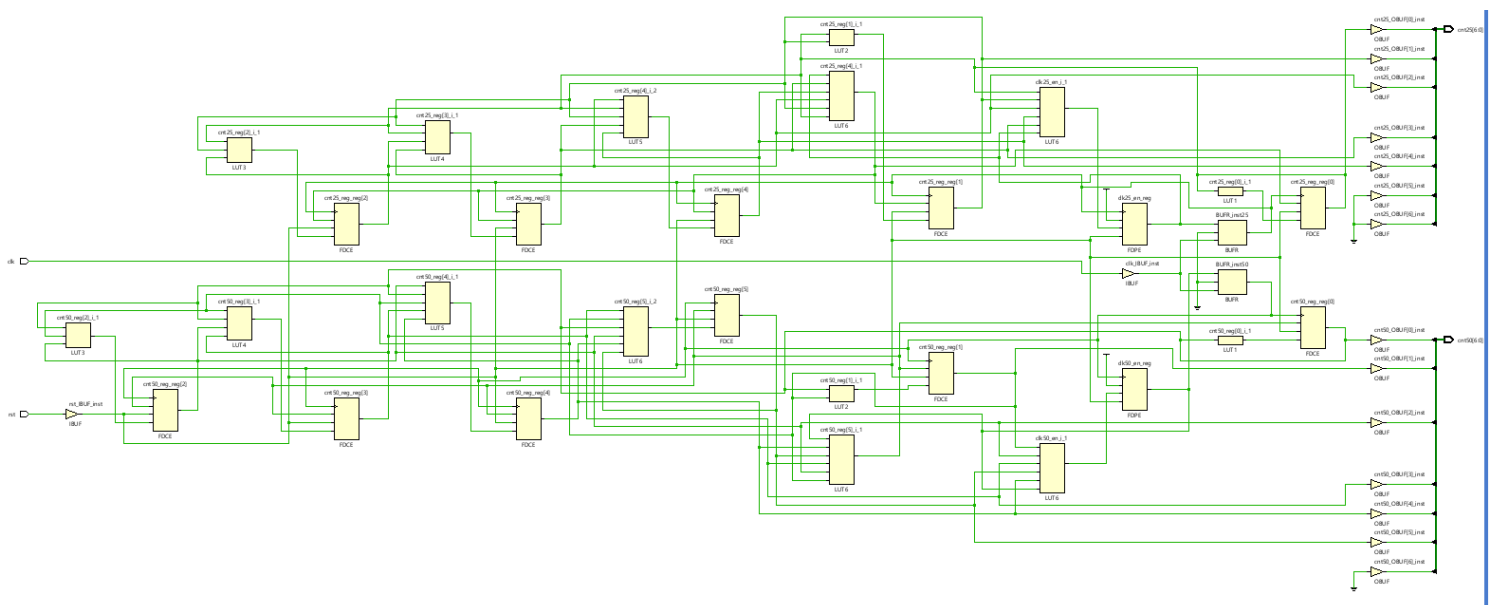


After the rst signal, counters start counting.



If counters reach their final value, enable signal of clocks have logic zero value. Therefore, counters stop counting and clocks stop.

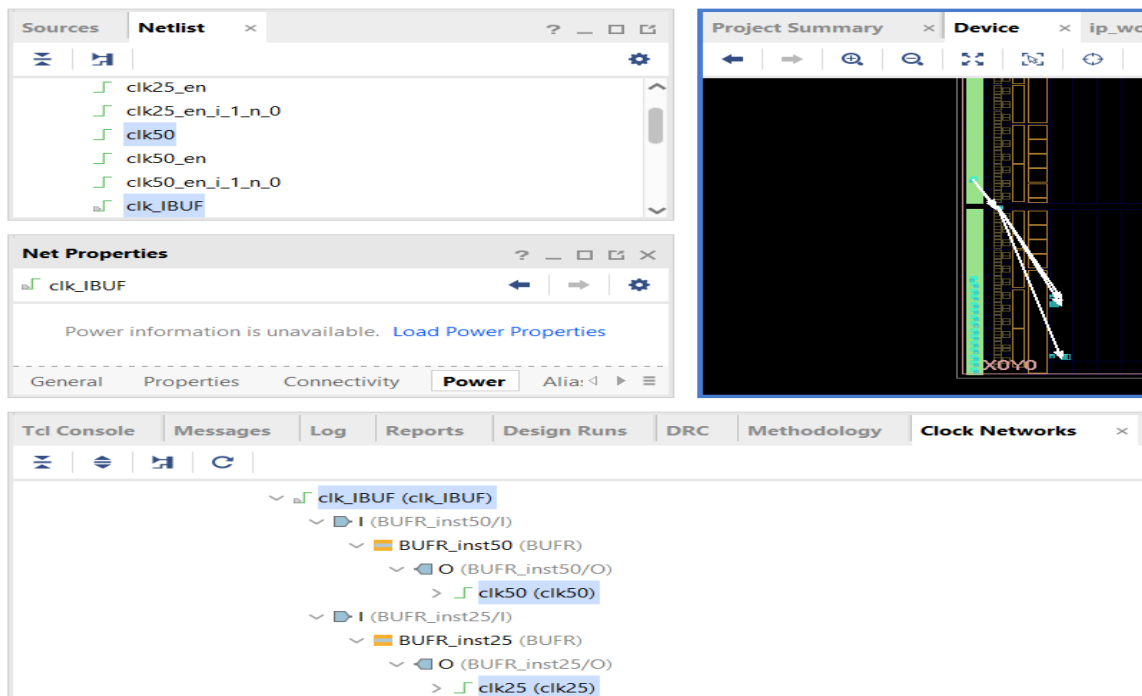
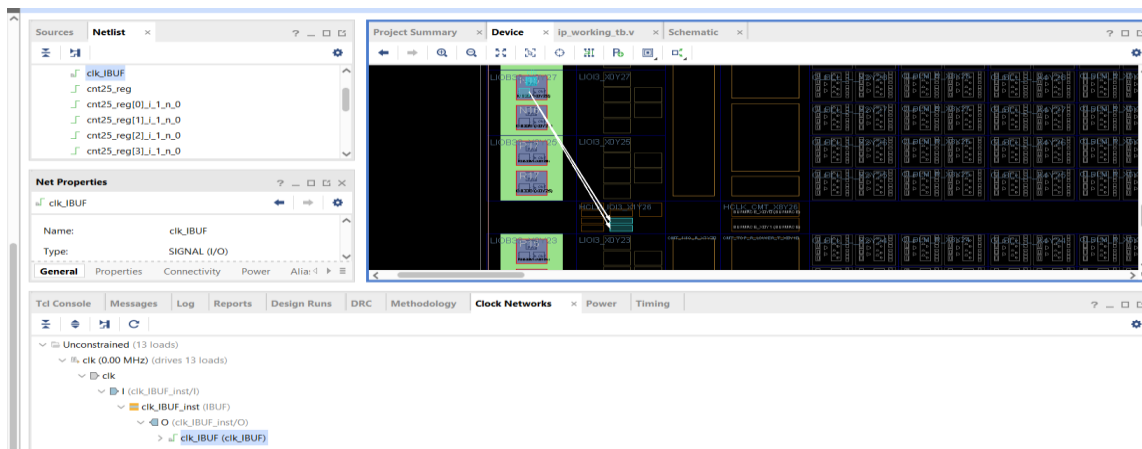
## Technology Schematic



## Utilization Reports

Utilization			
		Post-Synthesis	Post-Implementation
		Graph   Table	
Resource	Estimation	Available	Utilization %
LUT	11	32600	0.03
FF	13	65200	0.02
IO	16	210	7.62

## Clock Networks



We use 50MHz and 25MHz clocks.

## BLOCK RAM

### Design Source

```
module block_ram
(
    input clka,
    input wea,
    input [3:0] addra,
    input [7:0] dina,
    output [7:0] douta
);

    wire clk_50;
    blk_mem_gen_0
your_instance_name
    (
        .clka(clk_50),    // input
    wire clka
        .wea(wea),        // input
    wire [0 : 0] wea
        .addra(addra),    // input
    wire [3 : 0] addra
        .dina(dina),      // input
    wire [7 : 0] dina
        .douta(douta)     // output
    wire [7 : 0] douta
    );

    clk_wiz_2 instance_name
    (
        // Clock out ports
        .clk_50(clk_50),    // output
    clk_50
        // Status and control signals
        .reset(reset), // input reset
        .locked(locked),    //
    output locked
        // Clock in ports
        .clk_in1(clka)      // input
    clk_in1
    );

endmodule
```

## Simulation Source

```
module block_ram_tb();

    reg clka = 1'b0;
    reg wea = 1'b0;
    //reg ena = 1'b1;
    reg [3:0] addra = 4'b0;
    reg [7:0] dina = 8'b0;
    wire [7:0] douta;
    integer i;
    reg [7:0] vary [15:0];
    //reg [7:0] vary;
    //reg [7:0] vary2;
    block_ram uut
    (
        .clka(clka),
        .wea(wea),
        .addra(addra),
        .dina(dina),
        .douta(douta)
    );

    always begin
        #5 clka = ~clka;
    end

    initial begin
        $display("we are reading");
        #300; // ilk baslangicta LOCKED oluyor
        for(i = 0; i<16; i=i+1) begin
            addra = i;
            #50;
            $display("Address: %d, Data Read: %h", addra, douta);
        end

        #10
        $display("we are writing to array");
        for(i = 0; i<16; i=i+1) begin
            addra = i;
            #50;
            vary[15-i] = douta;
            $display("array'e yaz Address: %d, Data Output: %h,Vary: %h", addra, douta,vary[15-i]);
        end

        #10;
        $display("we are writing to address");
        wea = 1'b1;
        for(i = 0; i<16; i=i+1) begin
            addra = i;
            dina = vary[i];
            #50
            $display("Address: %d, Data Input: %h, Vary: %h, Data Output: %h", addra, dina,vary[i],douta);
        end

        $display("we are reading number backwards");
        wea = 1'b0;
        for(i = 0; i<16; i=i+1) begin
            addra = i;
            #50;
            $display("Address: %d, Data Read: %h", addra, douta);
        end
        $finish;
    end
endmodule
```

## Simulation TCL Console

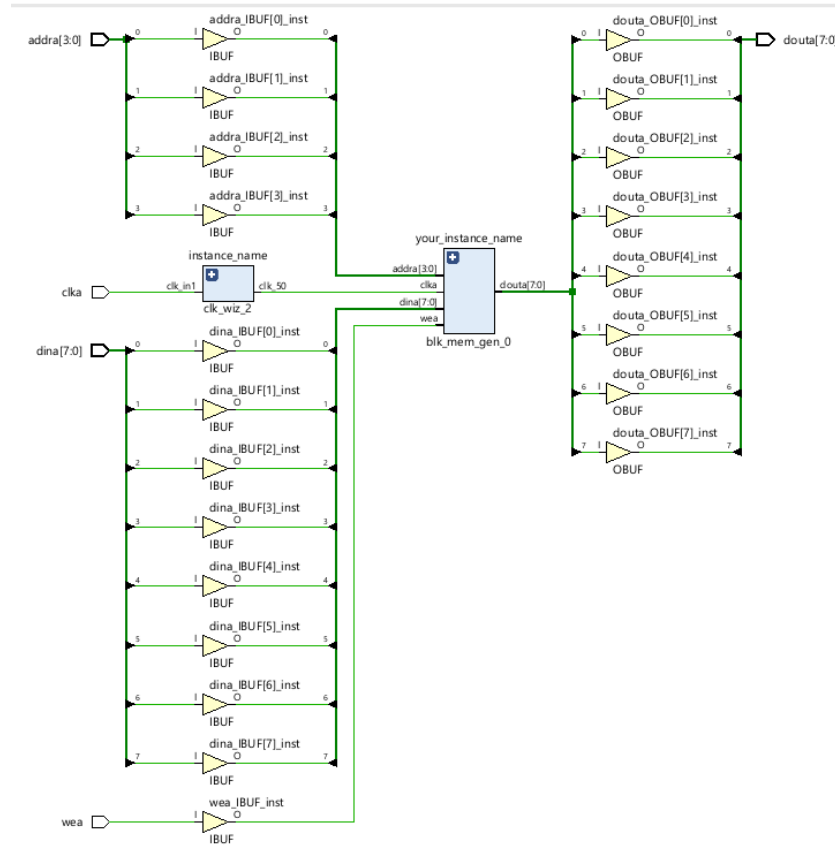
```
block_ram_tb_behav: Running...
we are reading
Address: 0, Data Read: 04
Address: 1, Data Read: 00
Address: 2, Data Read: 02
Address: 3, Data Read: 02
Address: 4, Data Read: 00
Address: 5, Data Read: 07
Address: 6, Data Read: 08
Address: 7, Data Read: 00
Address: 8, Data Read: 04
Address: 9, Data Read: 00
Address: 10, Data Read: 02
Address: 11, Data Read: 02
Address: 12, Data Read: 00
Address: 13, Data Read: 07
INFO: [USF-XSim-96] XSim completed. Design snapshot 'block_ram_tb_behav'
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:09 . Memory
run all
Address: 14, Data Read: 08
Address: 15, Data Read: 00
we are writing to array
array'e yaz Address: 0, Data Output: 04,Vary: 04
array'e yaz Address: 1, Data Output: 00,Vary: 00
array'e yaz Address: 2, Data Output: 02,Vary: 02
array'e yaz Address: 3, Data Output: 02,Vary: 02
array'e yaz Address: 4, Data Output: 00,Vary: 00
array'e yaz Address: 5, Data Output: 07,Vary: 07
array'e yaz Address: 6, Data Output: 08,Vary: 08
array'e yaz Address: 7, Data Output: 00,Vary: 00
array'e yaz Address: 8, Data Output: 04,Vary: 04
array'e yaz Address: 9, Data Output: 00,Vary: 00
array'e yaz Address: 10, Data Output: 02,Vary: 02
array'e yaz Address: 11, Data Output: 02,Vary: 02
array'e yaz Address: 12, Data Output: 00,Vary: 00
array'e yaz Address: 13, Data Output: 07,Vary: 07
array'e yaz Address: 14, Data Output: 08,Vary: 08
array'e yaz Address: 15, Data Output: 00,Vary: 00
```

First, we read the data filled from the memory.coe file. It read my numbers in the correct order. Then, I created an array and created the data that the block ram wrote to the array in reverse order. For example, the last index of the array has the first index data of the block ram.

```
we are writing to address
Address: 0, Data Input: 00, Vary: 00, Data Output: 00
Address: 1, Data Input: 08, Vary: 08, Data Output: 08
Address: 2, Data Input: 07, Vary: 07, Data Output: 07
Address: 3, Data Input: 00, Vary: 00, Data Output: 00
Address: 4, Data Input: 02, Vary: 02, Data Output: 02
Address: 5, Data Input: 02, Vary: 02, Data Output: 02
Address: 6, Data Input: 00, Vary: 00, Data Output: 00
Address: 7, Data Input: 04, Vary: 04, Data Output: 04
Address: 8, Data Input: 00, Vary: 00, Data Output: 00
Address: 9, Data Input: 08, Vary: 08, Data Output: 08
Address: 10, Data Input: 07, Vary: 07, Data Output: 07
Address: 11, Data Input: 00, Vary: 00, Data Output: 00
Address: 12, Data Input: 02, Vary: 02, Data Output: 02
Address: 13, Data Input: 02, Vary: 02, Data Output: 02
Address: 14, Data Input: 00, Vary: 00, Data Output: 00
Address: 15, Data Input: 04, Vary: 04, Data Output: 04
we are reading number backwards
Address: 0, Data Read: 00
Address: 1, Data Read: 08
Address: 2, Data Read: 07
Address: 3, Data Read: 00
Address: 4, Data Read: 02
Address: 5, Data Read: 02
Address: 6, Data Read: 00
Address: 7, Data Read: 04
Address: 8, Data Read: 00
Address: 9, Data Read: 08
Address: 10, Data Read: 07
Address: 11, Data Read: 00
Address: 12, Data Read: 02
Address: 13, Data Read: 02
Address: 14, Data Read: 00
Address: 15, Data Read: 04
Sfinish called at time : 3520 ns : File "D:/SSTU lab project/
```

Then the data in the array is written to the block ram address in order. My ID number has been written to block ram in reverse order. Finally, it read value in block ram, the values is inverse order of my student ID.

## Technology Schematic



## Utilization Reports

Utilization		Post-Synthesis		Post-Implementation
		Graph		Table
Resource	Utilization	Available	Utilization %	
BRAM	0.50	75	0.67	
IO	22	210	10.48	
BUFG	2	32	6.25	
MMCM	1	5	20.00	

## FIFO

### Design Source

```
module FIFO
(
    input clk,
    input rst,
    input wr_en,
    input rd_en,
    input [7:0] din,
    output [7:0] dout,
    output empty,
    output full,
    output overflow,
    output underflow
);

    wire clk_50,clk_25;

    fifo_generator_0 your_instance_name
    (
        .wr_clk(clk_50),          // input wire wr_clk
        .wr_rst(rst),            // input wire wr_rst
        .rd_clk(clk_25),         // input wire rd_clk
        .rd_rst(rst),            // input wire rd_rst
        .din(din),               // input wire [7 : 0] din
        .wr_en(wr_en),           // input wire wr_en
        .rd_en(rd_en),           // input wire rd_en
        .dout(dout),             // output wire [7 : 0] dout
        .full(full),             // output wire full
        .overflow(overflow),      // output wire overflow
        .empty(empty),           // output wire empty
        .underflow(underflow)    // output wire underflow
    );

    clk_wiz_3 instance_name
    (
        // Clock out ports
        .clk_50(clk_50),         // output clk_50
        .clk_25(clk_25),         // output clk_25
        // Status and control signals
        .reset(reset),           // input reset
        .locked(locked),         // output locked
        // Clock in ports
        .clk_in1(clk)            // input clk_in1
    );

endmodule
```



## Simulation Source

```
module FIFO_tb();

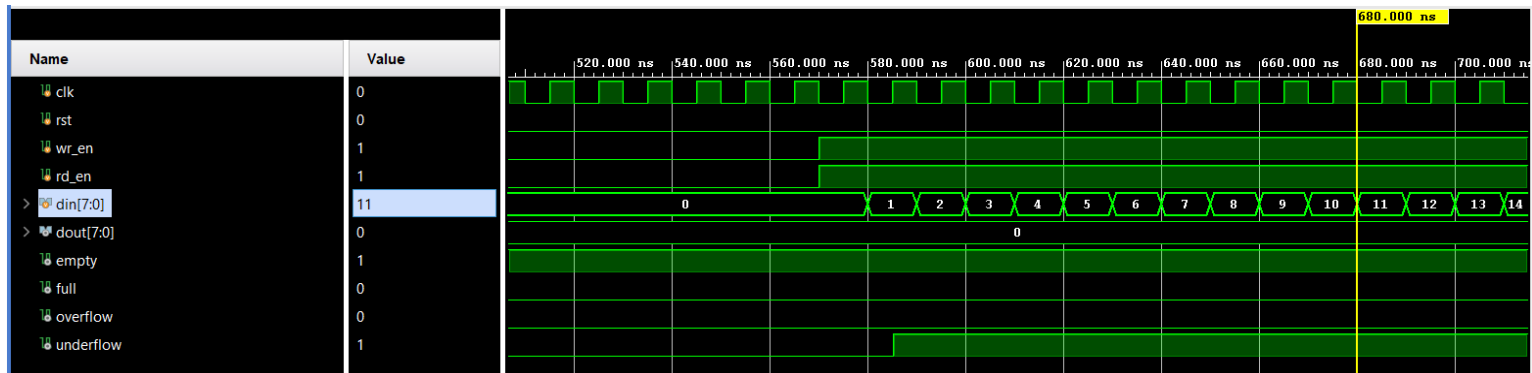
    reg clk = 1'b0;
    reg rst = 1'b0;
    reg wr_en = 1'b0;
    reg rd_en = 1'b0;
    reg [7:0] din = 1'b0;
    wire [7:0] dout;
    wire empty;
    wire full;
    wire overflow;
    wire underflow;
    integer i = 0;

    FIFO uut
    (
        .clk(clk),
        .rst(rst),
        .wr_en(wr_en),
        .rd_en(rd_en),
        .din(din),
        .dout(dout),
        .empty(empty),
        .full(full),
        .overflow(overflow),
        .underflow(underflow)
    );

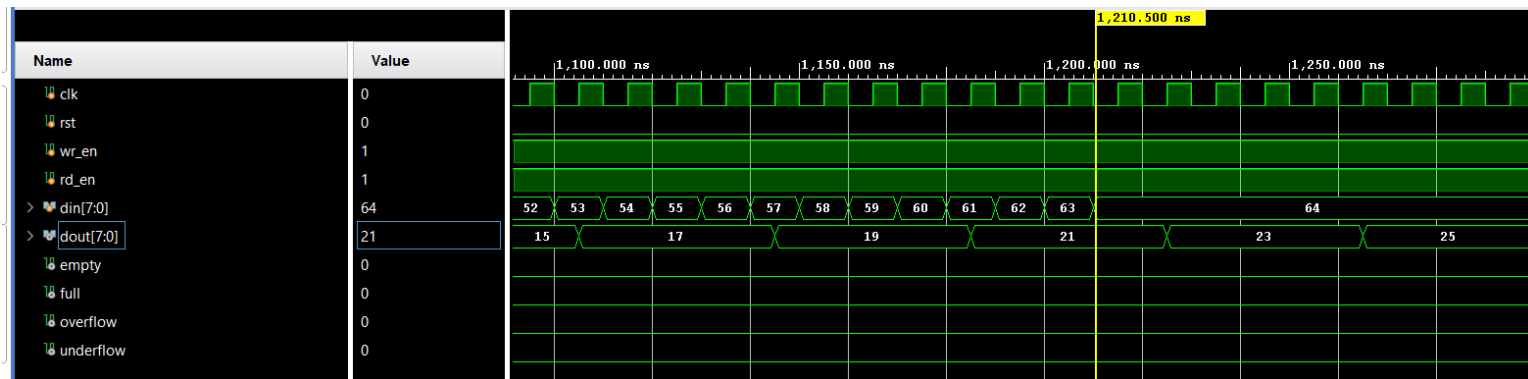
    always begin
        #5 clk = ~clk;
    end

    initial begin
        #200;
        rst = 1'b1;
        #20;
        rst = 1'b0;
        #350;
        while(i<65) begin
            wr_en = 1'b1;
            din = i;
            $display("Data Write: %d",din);
            rd_en = 1'b1;
            i=i+1;
            #10;
            $display("Data Read: %d",dout);
        end
    end
endmodule
```

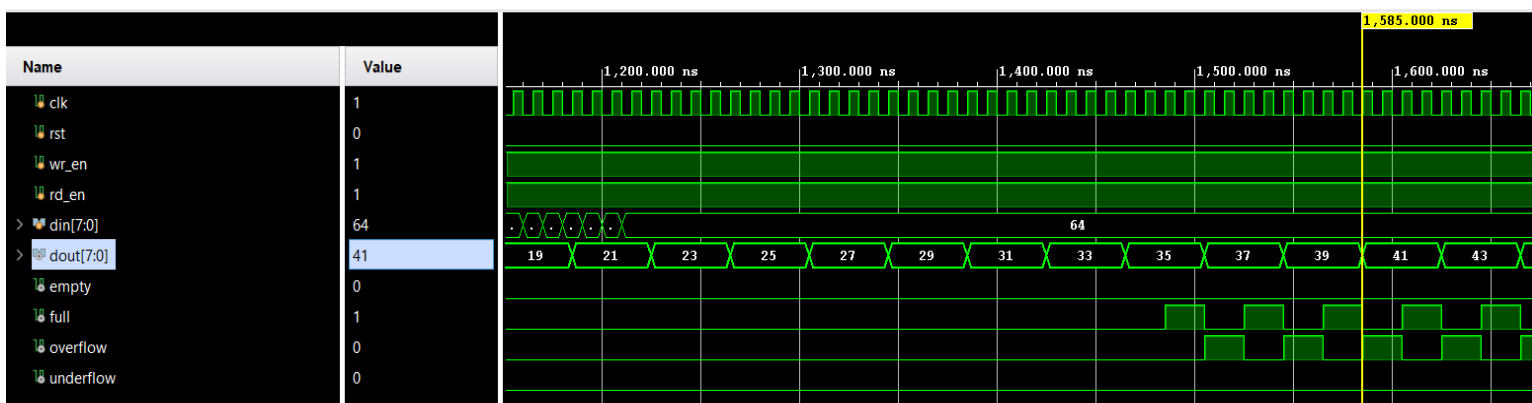
## Simulation Wave



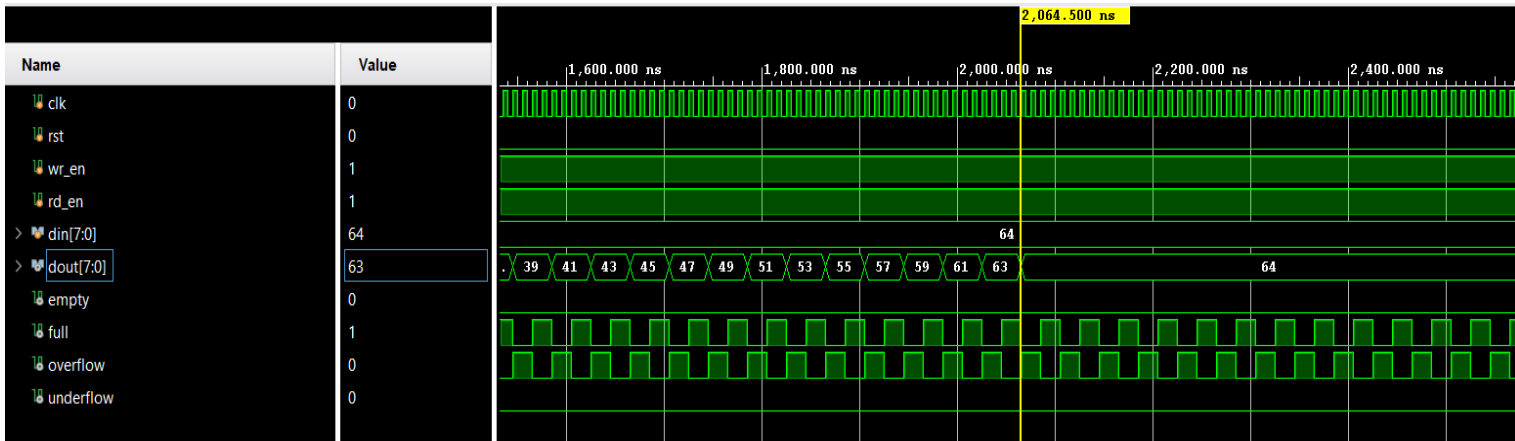
Simulation start to write data.



Write clock has 50MHz and read clock 25MHz frequency. Therefore, writing is faster than reading. Writing reach final value but reading does not.

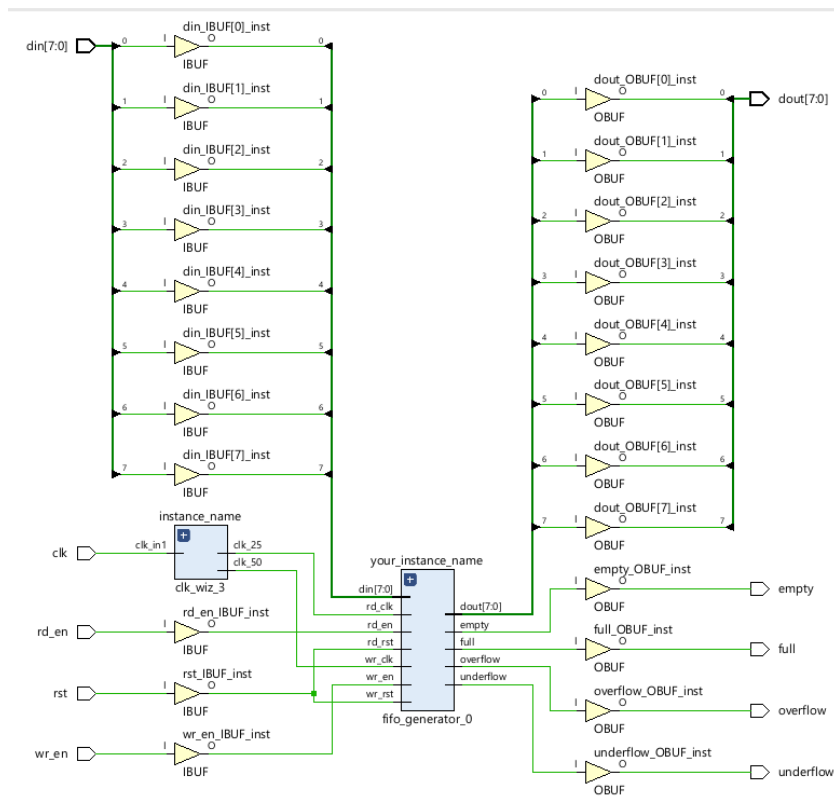


Writing reach final value but reading does not. Therefore, full flag has logic 1 value. Simulation write data to FIFO although full has logic 1 value. Therefore, overflow flag has logic 1 value.



Finally, reading reach final value but it can not read some intermediate value.

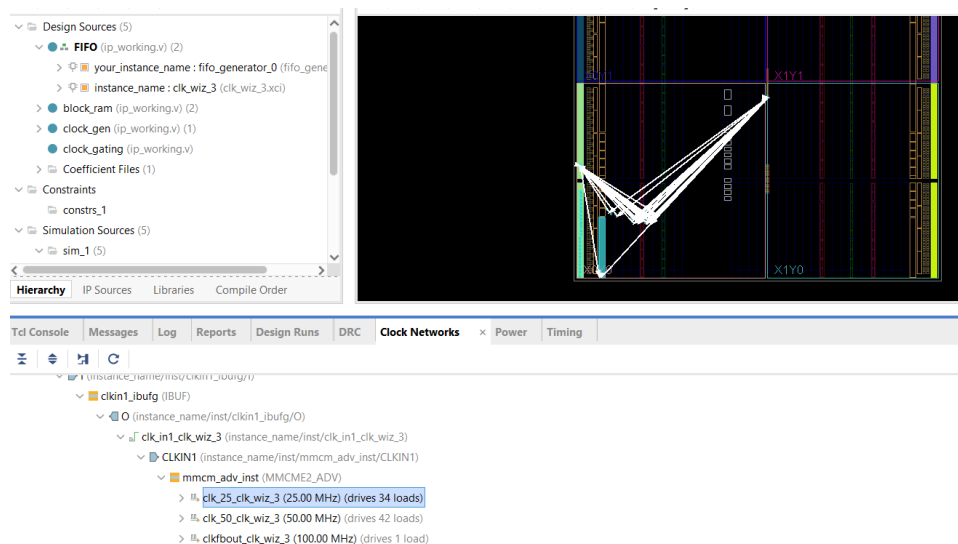
## RTL Schematic



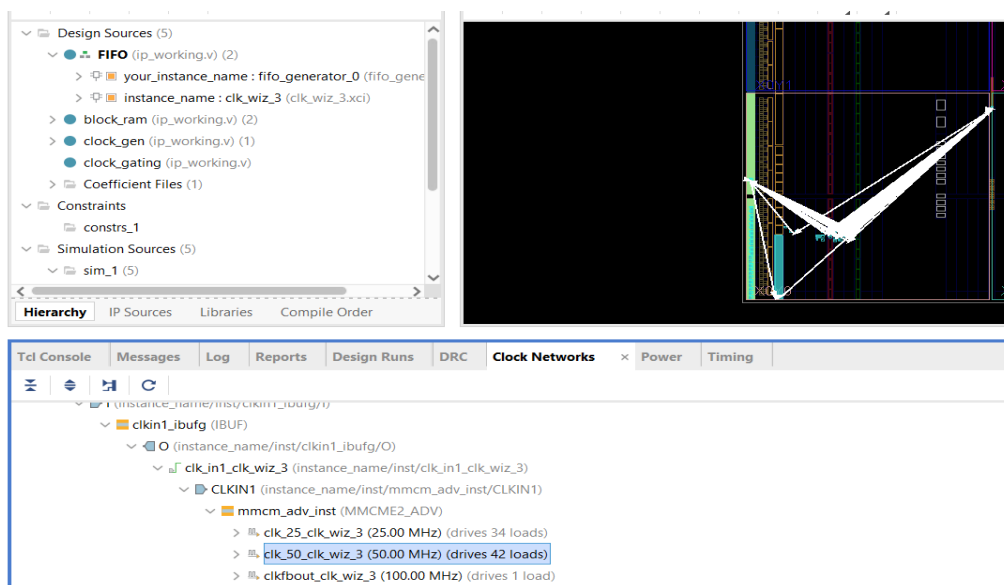
## Utilization Report

Utilization		Post-Synthesis	Post-Implementation
		Graph   Table	
Resource	Utilization	Available	Utilization %
LUT	32	32600	0.10
FF	74	65200	0.11
BRAM	0.50	75	0.67
IO	24	210	11.43
BUFG	3	32	9.38
MMCM	1	5	20.00

## Clock Network Reports



The screenshot displays the Vivado IDE interface. On the left, the Hierarchy tab shows the design structure. The main window shows the clock network diagram, which includes a clock generator (X1Y0) and a clock divider (X1Y1). The clock signal path is highlighted in red, showing the connection from the clock generator to the clock divider and then to the clock divider output.



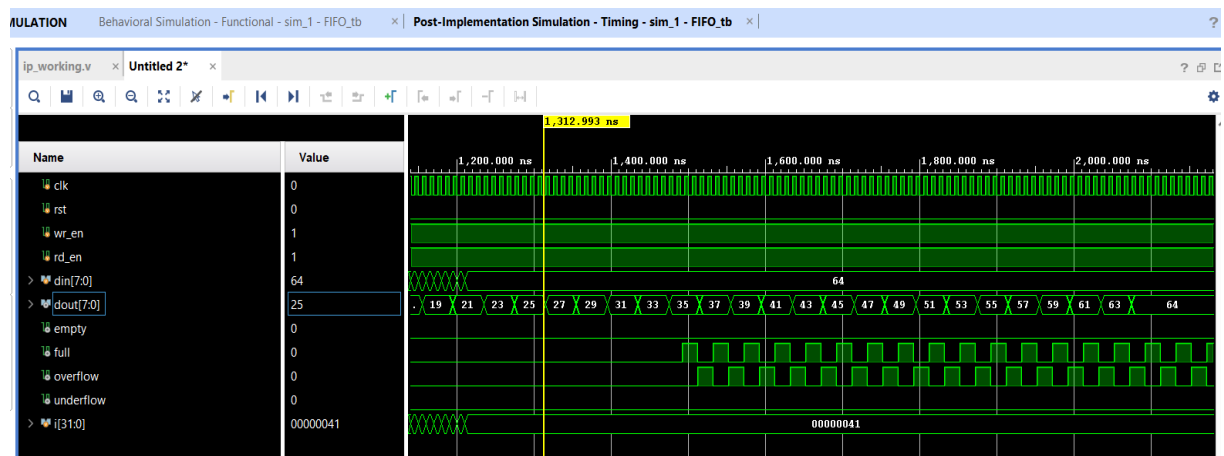
The screenshot displays the Vivado IDE interface. On the left, the Hierarchy tab shows the design structure. The main window shows the clock network diagram, which includes a clock generator (X1Y0) and a clock divider (X1Y1). The clock signal path is highlighted in red, showing the connection from the clock generator to the clock divider and then to the clock divider output.

We use 50MHz and 25MHz clocks.

## Clock Network Interaction



## Post-Implementation Timing Simulation



## RESEARCH

- 1) Clock Tree Synthesis is one of most important technique in digital desgin. It aims distribute clock signal balanced. It insert buffers/inverters along the clock routes of an ASIC design. [1]

**Clock Latency :** Clock latency refers to the arrival time of the clock signal from the clock source to flip flops. The clock latency from the port up to the clock pin is referred to as the network latency.

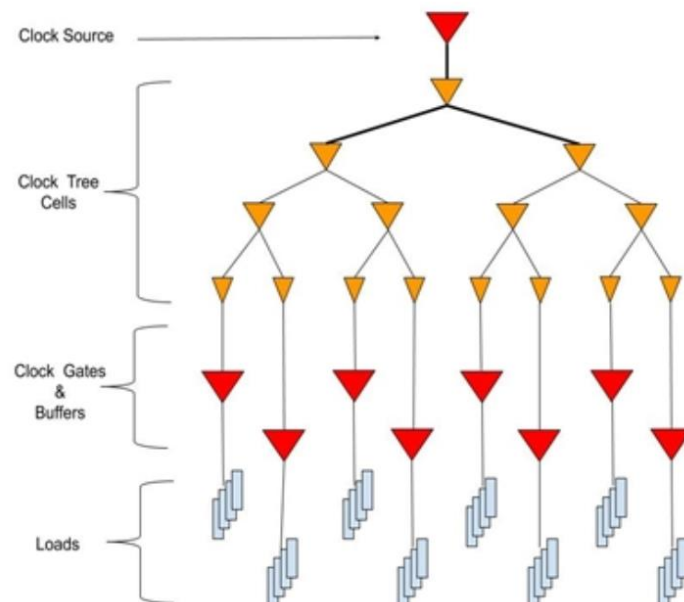
**Clock Skew :** Clock Skew refers to the difference in the clock arrival time between two registers.

**Clock Slew (or transition time):** The time that a given signal takes to rise from a level of 10% of the rail voltage to the level of 90% of the rail voltage is referred to as rise slew. Similarly, the time that a given signal takes to fall from a level of 90% of the rail voltage to the level of 10% of the rail voltage is referred to as fall slew.[2]

**Clock Jitter:** Normally, clocks have specific period. However, noise, temperature and other effects cause disruption in period of clock cycle. It is called clock jitter.

Conventional CTS/Single point CTS:

Single point CTS is the default choice for most of the designers having lower frequency & lesser no of sinks. As name suggested having single clock source which distribute clock to every corner of design.



Clock Mesh Structure

As the name suggests it create a dense mesh of shorted wires which is being driven by mesh drivers to distribute clock in every corner of the design.

The output of all the mesh drivers will be shorted using a metal mesh, which will carry the clock signal across the block using horizontal and vertical metal stripes

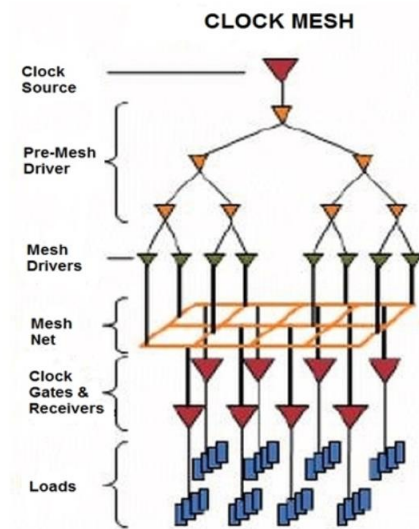
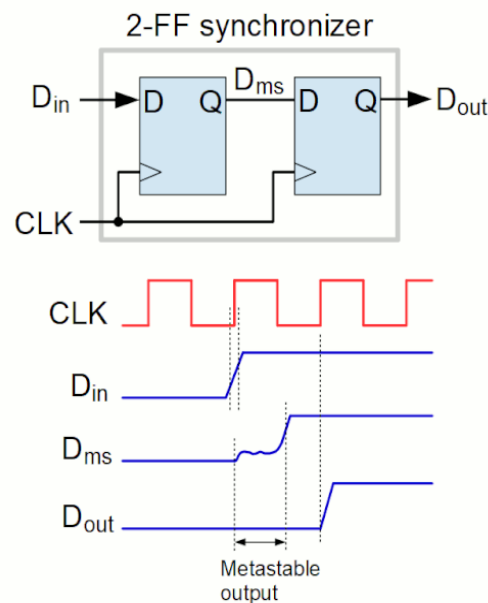


Figure 2. Mesh Structure

[3]

- 2) If the signal is not in a stable state at a certain time before and after the clock signal, the problem of metastability occurs. One of the methods to solve metastability is the Double Flop Synchroniser method. This method consists of two cascaded D flip-flops that sample the input signal with the destination clock. Flip-flops work as buffers to solve the metastability problem. Signals that cannot reach a stable state in the first flip-flop become stable due to the delay that occurs until they reach the second flip-flop. Therefore, it can solve metastability problem. [4]



[5]

## REFERENCE

- [1] B, D. M. (2024, March 27). *What is Clock Tree Synthesis?* ChipEdge VLSI Training Company. <https://chippedge.com/what-is-clock-tree-synthesis/>
- [2] Anysilicon. (2022, September 24). *Ultimate Guide: Clock Tree Synthesis - AnySilicon*. AnySilicon. <https://anysilicon.com/clock-tree-synthesis/>
- [3] Abhishek. (n.d.). *Different types of clock tree structure*. <https://vlsiconceptsforyou.blogspot.com/2020/07/different-types-of-clock-tree-structure.html?m=1>
- [4] Satheesh, M. (2024, April 3). *Double Flip-Flop Synchronizer for CDC*. Digital System Design. <https://digitalsystemdesign.in/double-flip-flop-synchronizer-for-cdc/?srsltid=AfmBOooDFmO3E6dyGGnOqOVNQIvss2JmlYAYptwuCR7QY3uE-IxriopT>
- [5] *File:2FF Synchronizer.gif - Wikimedia Commons*. (2020, January 18). [https://commons.wikimedia.org/wiki/File:2FF\\_synchronizer.gif](https://commons.wikimedia.org/wiki/File:2FF_synchronizer.gif)