DIGITAL SYSTEM DESIGN APPLICATIONS

Experiment 5

SEQUENTIAL DESIGN & MEMORY ELEMENTS

Berfin Duman
040190108

Contents

1 SR-Latch

SR latch is an electronic circuit element, also called Set-Reset latch or Set-Reset flip-flop. This circuit is used as a basic memory element and forms the basis of many other logic circuits. The SR latch is characterized by two input signals "Set" (S) and "Reset" (R).
The basic working principle of SR latch is as follows:
Reset (R): This input sets the Q output to "1" (high) level. That is, when input R is "1", output Q is "1" and Set (S) is "0".
Reset (R): This input sets the Q output to "0" (low) level. That is, when input S is "1" and R is "0", output Q is "0".
The interaction between these two basic input signals determines the operating state of the SR latch. The situation where both input elements are 0 is called invalid state, while the output maintains its previous value when both input elements are 1.

Characteristic Function (Q) of SR Latch

$$Q(t+1) = S + \overline{R} \cdot Q(t) \tag{1}$$

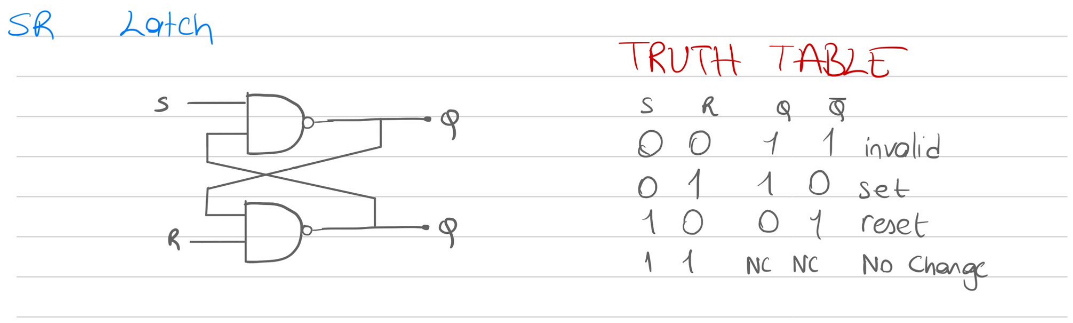
In this equation, the output $Q(t+1)$ is the result of a NAND operation between the set input (§) added to a NAND operation between the reset input (complement of (R)) and $Q(t)$.

Non-characteristic Function (Q) of SR Latch

$$Qn(t+1) = R \cdot (\overline{S} + \overline{Qn(t)}) \tag{2}$$

S	R	Q	Q'
0	0	1	1
0	1	1	0
1	0	0	1
1	1	NC	NC

Table 1: SR-Latch Truth Table



TRUTH TABLE

S	R	Q	$\overline{Q}$	
0	0	1	1	invalid
0	1	1	0	set
1	0	0	1	reset
1	1	NC	NC	No Change

Figure 1: Realize SR LATCH by hand-written

After researching Sr latch and learning its working principle, I wrote its module and teshbench. In order to see the values I expected in the truth table, I created a simulation in which I could observe every input and output.

Listing 1: SR_latch module

```
`timescale 1ns / 1ps

module SR_latch(
    input S, R, output Q, Qn
    );
NAND nand1(.O(Q),.I1(S),.I2(Qn));
NAND nand2(.O(Qn),.I1(R),.I2(Q));
endmodule
```

Listing 2: SR_latch module testbench

```
`timescale 1ns / 1ps
module sr_latch_tb();
  reg S;
  reg R;
  wire Q;
  wire Qn;
  SR_latch uut(.S(S),.R(R),.Q(Q),.Qn(Qn));
    initial
    begin
    S=0; R=0;
    #10 S=0; R=1;
    #10 S=1; R=1;
    #10 S=1; R=0;
    #10 S=1; R=1;
    #10 S=0; R=0;
    #10 S=1; R=0;
    #10
    $finish;
    end
endmodule
```

When both input elements are 0, the Outputs become 1. In cases where Reset is only 1, Q becomes 1. Then, between 20-30 ns when both are 1, the latch maintains the previous state (1) as it should be. Later, when we set R to 0 and keep S the same, Qn becomes 1 Q 0, and similarly, when we give 1- 1 to the inputs, we see that this value is also preserved. At the end of the simulation, the inputs are set to 0 and the outputs show 1- 1.

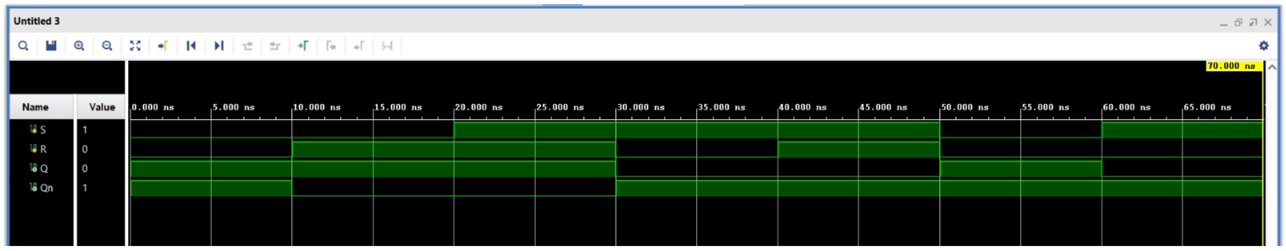


Figure 2: Behavioral Simulation of SR LATCH

In the RTL schematic I observe two nand gates as I did in my Sr Latch hand drawing. Likewise, the technology schematic can be seen using circuit 1 LUT2 1 LUT3.

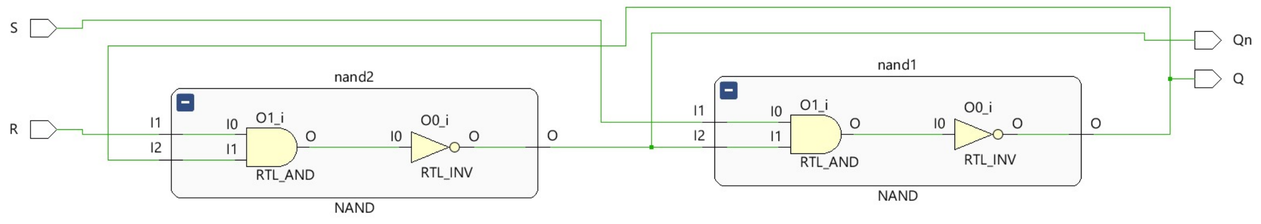


Figure 3: RTL Schematic of SR LATCH

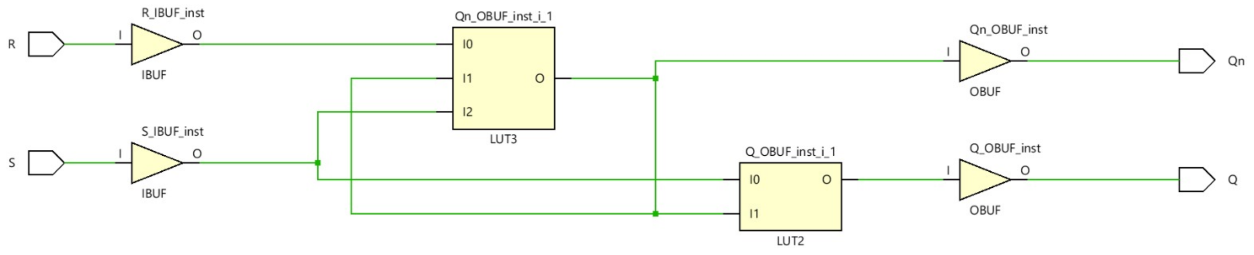
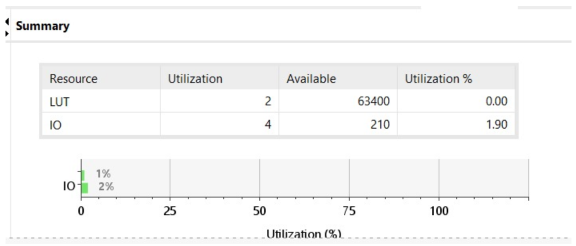


Figure 4: Technology Schematic of SR LATCH

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { S }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { R}];
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Q }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Qn }];
set_property ALLOW_COMBINATORIAL_LOOPS true

set_property SEVERITY {Warning}  [get_drc_checks LUTLP-1]

set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
```

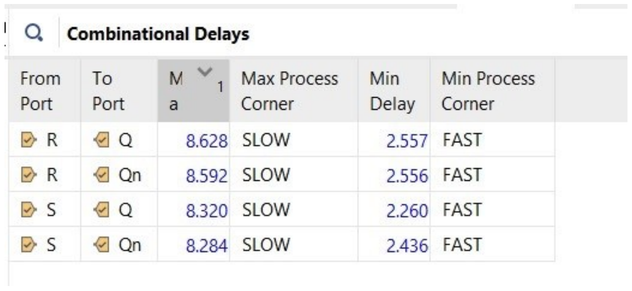
I created the constrain file by connecting the inputs and outputs of the module I created to the switches and LEDs, and continued with the synthesis, implementation and bit generation section.


Summary

Resource	Utilization	Available	Utilization %
LUT	2	63400	0.00
IO	4	210	1.90

Figure 5: Utilization Summary of SR LATCH

The utility report contained 2 and 4 io ports, as we observed in the technology schematic.


Combinational Delays

From Port	To Port	Ma	Max Process Corner	Min Delay	Min Process Corner
R	Q	8.628	SLOW	2.557	FAST
R	Qn	8.592	SLOW	2.556	FAST
S	Q	8.320	SLOW	2.260	FAST
S	Qn	8.284	SLOW	2.436	FAST

Figure 6: Timing Summary of SR LATCH

When I looked at the timing report, I observed that the biggest delay was from the R input to the Q output.

2 D Flip Flop

D flip-flop is a type of data storage element used in digital circuits. Basically, the D flip-flop has a D (data) input, a Clock (clock) input, and usually one or two outputs. The D flip-flop operates under a given clock signal and usually has negative edge (negative clock rise) or positive edge (positive clock rise) triggering capabilities.
The basic function of the D flip-flop is to receive data at the D input at a certain edge of the clock signal and pass this data to the output.

Characteristic and Non-Characteristic Functions (Q) of D Latch

$$Q(t+1) = CLK \cdot D + \overline{CLK} \cdot Q(t) \tag{3}$$

$$Qn(t+1) = (\overline{CLK} + \overline{D}) + (CLK * \overline{Q(t)}) \tag{4}$$

CLK	D	Q	Q’
0	x	NC	NC
1	0	0	1
1	1	1	0

Table 2: D-Latch Truth Table

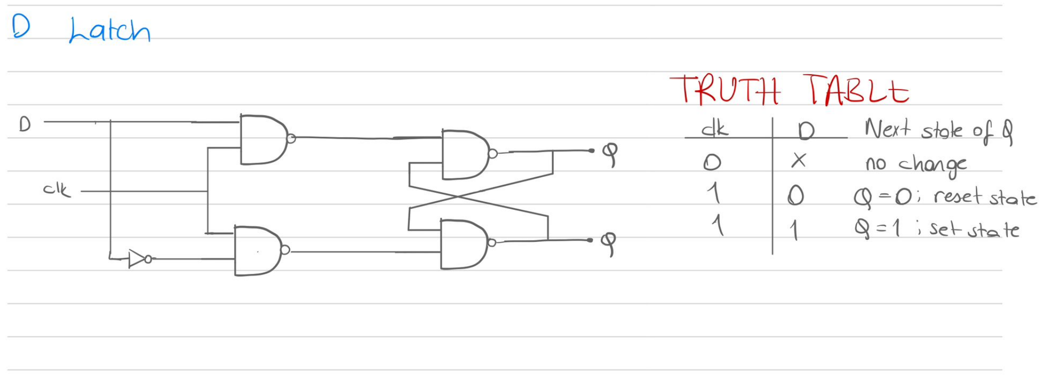


Figure 7: Realize D LATCH by hand-written

I did the same research on the D flip flop and wrote the module and testbench after drawing the required circuit and creating the truth table. To test my module, I simulated the module.

Listing 3: D_latch module

```
‘timescale 1ns / 1ps
module D_latch(
    input CLK, D, output Q, Qn
    );
wire D_not;
wire S;
wire R;
NOT not1(.O(D_not), .I(D));
NAND nand1(.O(R),.I1(D_not),.I2(CLK));
NAND nand2(.O(S),.I1(D),.I2(CLK));
NAND nand3(.O(Q),.I1(S),.I2(Qn));
NAND nand4(.O(Qn),.I1(R),.I2(Q));
endmodule
```

Listing 4: D_latch module testbench

```
‘timescale 1ns / 1ps
module d_latch_tb();
  reg D;
  reg CLK;
  wire Q;
  wire Qn;
  //D_latch uut(.D(D),.CLK( CLK),.Q(Q),.Qn(Qn));
  behavioral_dff uut(.D(D),.CLK( CLK),.Q(Q),.Qn(Qn));
    initial
    begin
    D=0;  CLK=1;
    #10 D=0;  CLK=1;
    #10 D=1;  CLK=1;
    #10 D=1;  CLK=0;
    #10 D=1;  CLK=1;
    #10 D=0;  CLK=0;
    #10 D=1;  CLK=0;
    #10 D=0;  CLK=1;
    #10
    $finish;
    end
endmodule
```

When we look at the simulation, we see that the clock maintains its value when it is not active. It has been observed that when the clock is 1, the output is observed as the D input. Initially, when D was 0 and CLK was 1, Q became 0 and Qn became 1. Then, when D increases to 1, the value in Q becomes equal to 1, which is the D input, since CLK is active. When we reduce CLK to 0, the output maintains its previous state. While this continues until 50 ns, when CLK is 1 and D is 0, Q drops to 0, which is the value of D.

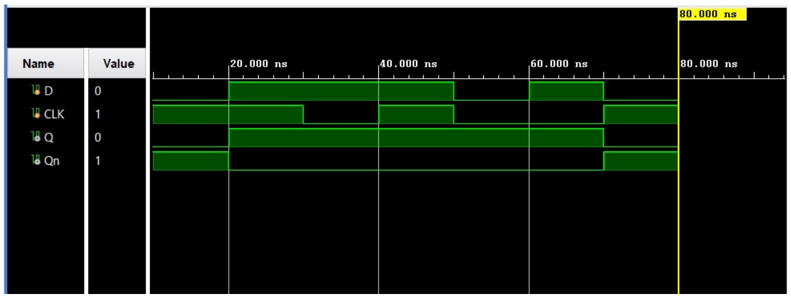


Figure 8: Behavioral Simulation of D LATCH

As I showed in the drawing, D latch consists of 4 nand gates, and at the same time, D's input enters one of the first nand gates. In the RTL schematic, 4 NAND and 1 NOT gates meet the result we expect.

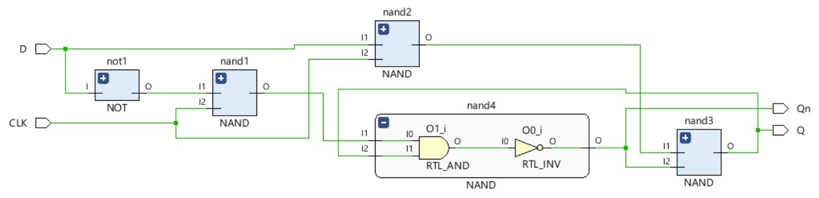


Figure 9: RTL Schematic of D LATCH

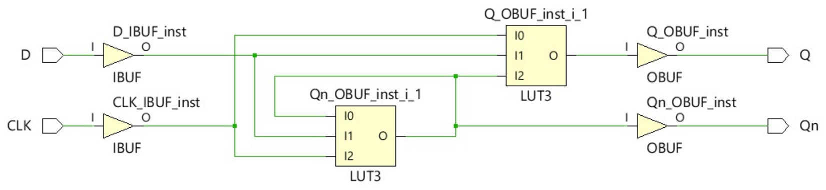
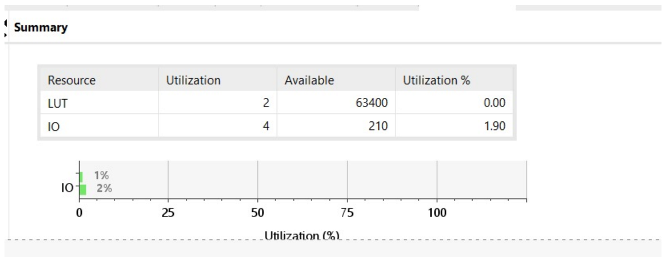


Figure 10: Technology Schematic of D LATCH

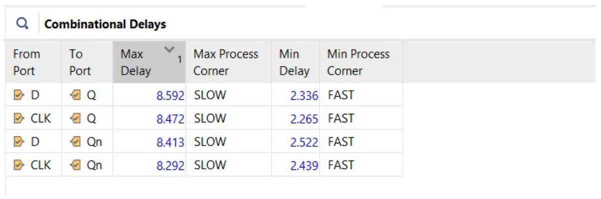
It is observed in D Latch's technology schematic that it uses 2 LUT3s. After organizing the constraint file according to the module, it was seen that 2 lots were used in the post-implementation utilization report we received. The results are consistent.

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { D }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { CLK}];
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Q }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Qn }];
set_property ALLOW_COMBINATORIAL_LOOPS true

set_property SEVERITY {Warning}  [get_drc_checks LUTLP-1]

set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
```


Summary

Resource	Utilization	Available	Utilization %
LUT	2	63400	0.00
IO	4	210	1.90

Figure 11: Utilization Summary of D LATCH



Combinational Delays

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
D	Q	8.592	SLOW	2.336	FAST
CLK	Q	8.472	SLOW	2.265	FAST
D	Qn	8.413	SLOW	2.522	FAST
CLK	Qn	8.292	SLOW	2.439	FAST

Figure 12: Timing Summary of D LATCH

The maximum delay in my circuit was 8.592 ns between D and Q. Based on this value, it is concluded that the maximum clock frequency should be 116.38MHz.
1/8.592 ns = 116.38MHZ

3 Master- Slave D- Flip Flop

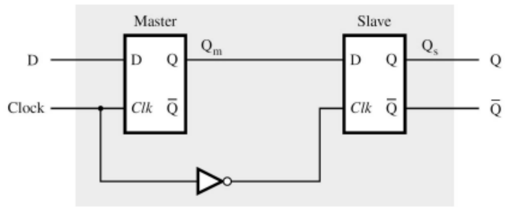


Fig.3: Master-Slave D-FF

Figure 13: Circuit Diagram

This structure consists of two D flip-flops connected in series. The first flip-flop is called the master and the second flip-flop is called the auxiliary flip-flop.
Whatever the change in CLK value for Qm, it takes the value at input D (Qm=D). Whenever our CLK value for Qs decreases from 1 to 0, when it receives a negedge signal, it moves the current D value to Qs. In addition, the CLK 1 signal is needed for the Qm signal to be active. Negedge signal is required for Q's to be active.
Negative edge of the clock signal effects the output.

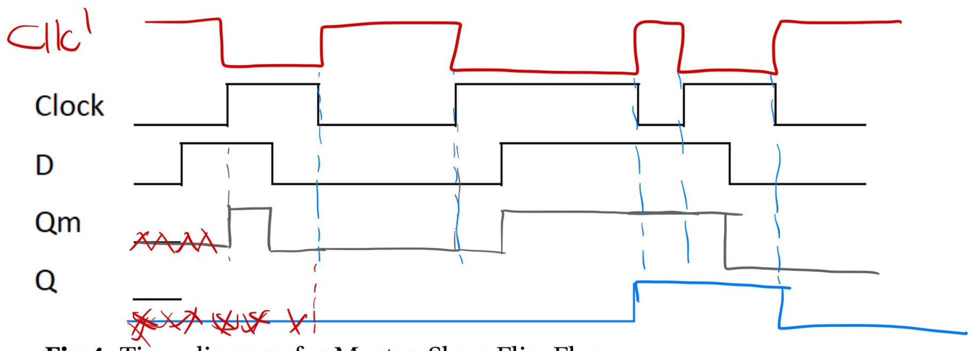


Fig.4: Time diagram for Master-Slave Flip-Flop.

Figure 14: Time diagram of M-S FF

Listing 5: MS-DFF module

```
`timescale 1ns / 1ps
module master_slaveDFF(input CLK, D, output Q, Qn);
wire wire1;
wire Q_m;
wire CLK_not;
NOT not1(.O(CLK_not), .I(CLK));
D_latch dlatch1(.CLK(CLK),.D(D),.Q(Q_m),.Qn(wire1));
D_latch dlatch2(.CLK(CLK_not),.D(Q_m),.Q(Q),.Qn(Qn));
endmodule
```

Listing 6: MS-DFF module testbench

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module ms_dff_tb();
  reg D;
  reg CLK;
  wire Q;
  wire Qn;
  master_slaveDFF uut(.D(D),.CLK( CLK),.Q(Q),.Qn(Qn));
    initial
    begin

    D=0;  CLK=1;
    #10 D=1;  CLK=1;
    #10 D=1;  CLK=0;
    #10 D=1;  CLK=1;
    #10 D=0;  CLK=0;
    #10 D=1;  CLK=0;
    #10 D=0;  CLK=1;
    #10 D=0;  CLK=0;
    #10
    $finish;
    end
endmodule
```

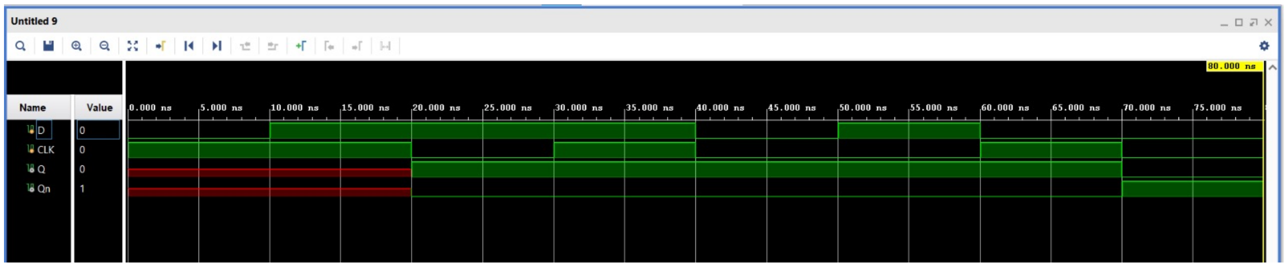


Figure 15: Behavioral Simulation of Master-Slave FF

Now, as I said above, in this circuit, CLK triggers the circuit at the negedge, giving Q as a result of D. Therefore, since CLK does not trigger D before 20 ns, we see X values in the first 20 ns of output. As CLK drops to 0, the circuit triggered by negedge processes the Q output to 1, which is the current value of Qm. Then, when CLK reaches 1, the circuit maintains its previous value as CLK will be 0 on the slave. In this process, Qm increases to 1, and when clk decreases to 0, the Qm value is transferred to Qs, that is, 0. When CLK is 0, the Qm value remains constant because the master is inactive and therefore Qs maintains its previous value. At 60 ns, CLK is 1 again and Qm switches to 0, the value of D, between 60 and 65 ns. When CLK goes to 0 with the negative edge, Qm moves to Qs and takes the value Q 0 Qm 1.

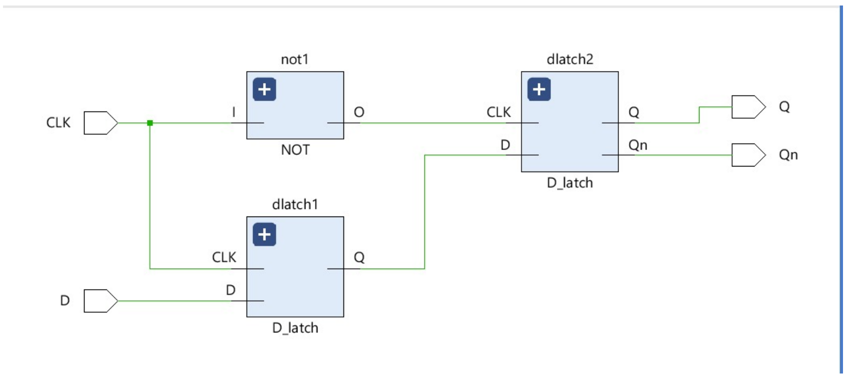


Figure 16: RTL Schematic of Master-Slave FF

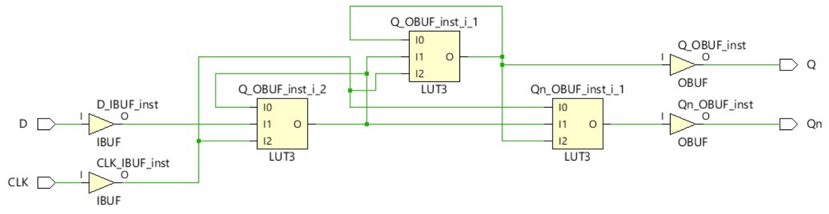
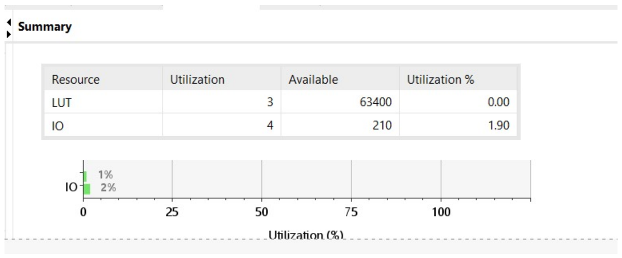


Figure 17: Technology Schematic of Master-Slave FF

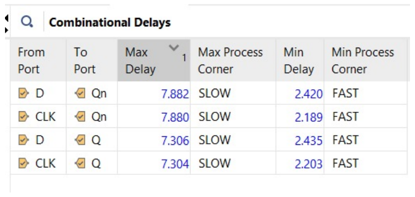
When we looked at the technology schematic of the module, I observed that it was created from two latches, as I wrote, and this was shown as 3 lut3 in the technology schematic. When I set the constraint file and look at the implementation result, it appears that 3 out of 4 io ports are used, as seen in the technology schematic. The longest delay is 7.882 ns between D input and Qn output.

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { D }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { CLK}];
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Q }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Qn }];
set_property ALLOW_COMBINATORIAL_LOOPS true

set_property SEVERITY {Warning}  [get_drc_checks LUTLP-1]

set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
```


Summary

Resource	Utilization	Available	Utilization %
LUT	3	63400	0.00
IO	4	210	1.90

Figure 18: Utilization Summary of Master-Slave FF


Combinational Delays

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
D	Qn	7.882	SLOW	2.420	FAST
CLK	Qn	7.880	SLOW	2.189	FAST
D	Q	7.306	SLOW	2.435	FAST
CLK	Q	7.304	SLOW	2.203	FAST

Figure 19: Timing Summary of Master-Slave FFH

4 D Flip-Flop Behavioral Design

This time I created the D Flip flop with the always block using the FF reg as requested.

Listing 7: D Flip-Flop module

```
‘timescale 1ns / 1ps
module behavioral_dff(
    input CLK, D, output Q, Qn);
    reg FF;
    always @(posedge CLK)
    begin
         FF<=D;
    end
    assign Q=FF;
    assign Qn=~FF;
endmodule
```

Listing 8: D Flip-Flop module testbench

```
‘timescale 1ns / 1ps
‘timescale 1ns / 1ps
module d_latch_tb();
  reg D;
  reg CLK;
  wire Q;
  wire Qn;
  //D_latch uut(.D(D),.CLK( CLK),.Q(Q),.Qn(Qn));
  behavioral_dff uut(.D(D),.CLK( CLK),.Q(Q),.Qn(Qn));
    initial
    begin
    D=0;  CLK=1;
    #10 D=0;  CLK=1;
    #10 D=1;  CLK=1;
    #10 D=1;  CLK=0;
    #10 D=1;  CLK=1;
    #10 D=0;  CLK=0;
    #10 D=1;  CLK=0;
    #10 D=0;  CLK=1;
    #10
    $finish;
    end
endmodule
```

To check whether d-latch gives the value we want in both modules, I simulated both modules with the same testbench.
The biggest reason why I see different results even though I use the same test bench is that

the clock of the D flip flop is triggered by the positive edge. At the positive edge, the clock assigns the value in D to Q and assigns its note to Qn. When we look at the simulation, when the clock beats the positive edge at the beginning, the D value was 0, so the Q value was 0 and the Qm value was 1. This continues until the 40th nanosecond, which is the next positive edge of CLK. Since the D value is 1 in the 40th ns, Q 1 is updated as Qm 0. continues until the next positive edge 70 ns. Then, similarly, Q becomes 0 and Qm becomes 1 depending on the D value.

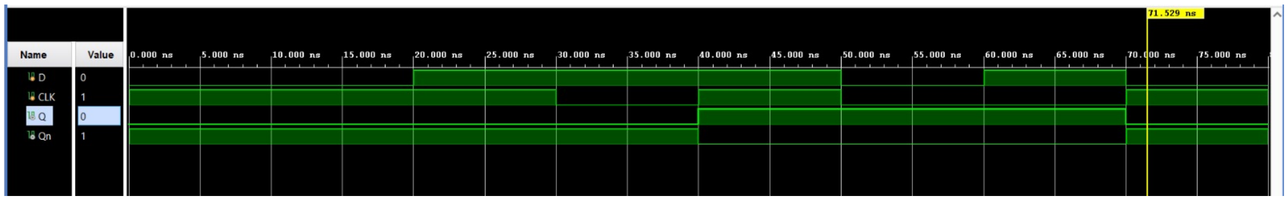


Figure 20: Behavioral Simulation of D LATCH

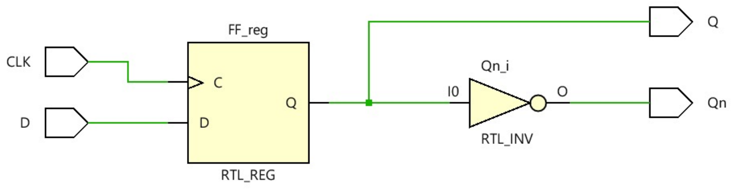


Figure 21: RTL Schematic of D LATCH

FDRE: The FDRE design element is a single D-type flip-flop with clock enable and synchronous reset features.
Clock Enable (CE) and Synchronous Reset (R): Data at the data input (D) of the FDRE is transferred to the corresponding data output (Q) during the clock (C) transition. This transfer occurs when clock enable (CE) is high and synchronous reset (R) is not performed.
When Reset (R) is Active: When reset (R) is active, it overrides all other inputs and resets the data output (Q) low at the next clock transition.
When CE is Low: Clock transitions are ignored when clock enable (CE) is low.
Asynchronous Startup: This flip-flop is started asynchronously when power is applied. When global set/reset (GSR) is active at power up or GSR is enabled via the initialization block, the value of the INIT property is placed at the output of the register.

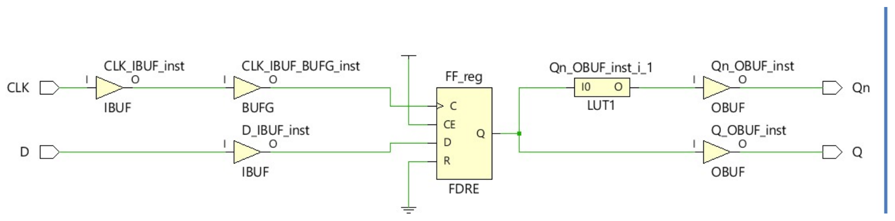
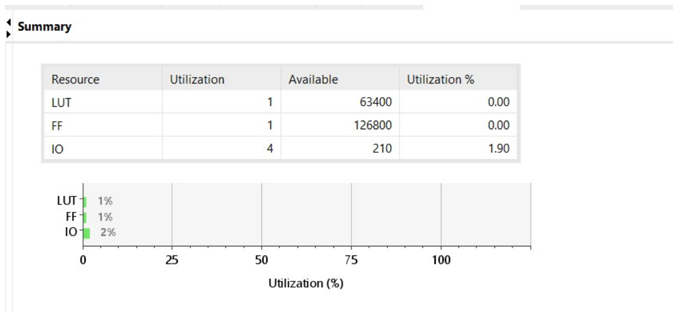


Figure 22: Technology Schematic of D LATCH

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { D }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { CLK}];
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Q }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Qn }];
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets CLK_IBUF];
```

When I just wrote the inputs and outputs to the constrain file and ran it, it gave an error and asked me to set the Clock_Dedicated_Route parameter in the last line. When I edited and ran it, I was able to implement it without any problems.


Summary

Resource	Utilization	Available	Utilization %
LUT	1	63400	0.00
FF	1	126800	0.00
IO	4	210	1.90

Figure 23: Utilization Summary of D LATCH

As we observed in the technology schematic, I observed that there was a FF (FDRE) and a LUT (Qn= Q') in the utilization report and there were 4 i/o ports as I expected.

5 8-bit Register

First, I created the module as requested and triggered the always block according to positive edge CLK or CLEAR being 1. If CLEAR is 1, it is output directly. Otherwise, it throws the value in out. I wrote a testbench to observe this and simulated it.

Listing 9: 8-bit Register module

```
‘timescale 1ns / 1ps
module register_8(input CLK, CLEAR, input [7:0] IN, output reg
    [7:0] OUT);
    always @(posedge CLK or posedge CLEAR)
    begin
        if (CLEAR)
            OUT <= 8'b0;  // Clear the register to zero when CLEAR is asserted
        else
            OUT <= IN;    // Assign input to output on the rising edge of the clock
    end
endmodule
```

Listing 10: 8-bit Register module testbench

```
`timescale 1ns / 1ps
module register_tb();
    reg CLK;
    reg CLEAR;
    reg [7:0] IN;
    wire [7:0] OUT;
    register_8 uut(.CLK(CLK),.CLEAR(CLEAR),.IN(IN),.OUT(OUT));
initial
begin
    IN=5; CLK=0; CLEAR= 0; #10
    IN=8; CLK=1; CLEAR=0; #10
    IN=9; CLK =1; CLEAR=1; #10
    IN =37; CLK=0; #10
    IN=66; CLK=1; CLEAR= 0; #10
    #10
    $finish;
end
endmodule
```

As can be seen, when CLEAR is 0 and CLK is 0, OUT becomes XX (no assignment can be made), if CLK is 1 and CLEAR is 0, IN takes the same value, but when CLEAR is 1, OUT is 0 for 30-40 ns. When CLK becomes 1 again (CLEAR 0), the IN value is written to OUT again.

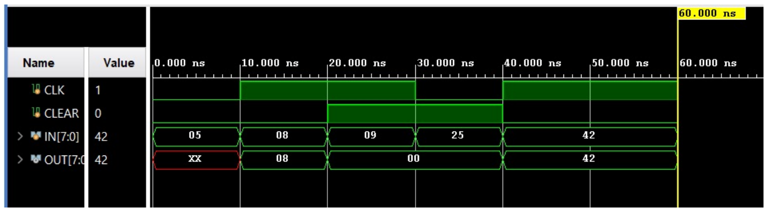


Figure 24: Behavioral Simulation of 8-bit Register

When we look at the RTL and technology schematics, we can see that what we actually did is that we connected 8 D flip flops to the same clear and clock, and the value of each in was written to the out from the relevant d-ff. In the technology schematic, we connected our CLear input to the Clear port of FDCE. In our Utilization result, we observed that there were 8 FF and 18i/o ports, as we saw in the technology schematic.

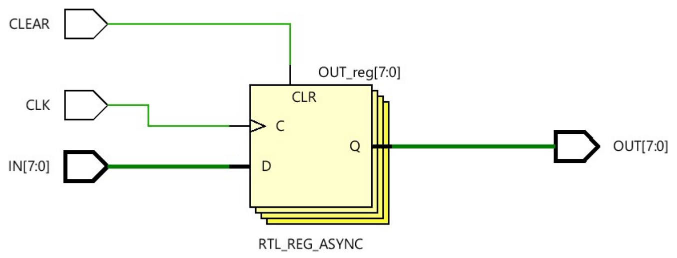


Figure 25: RTL Schematic of 8-bit Register

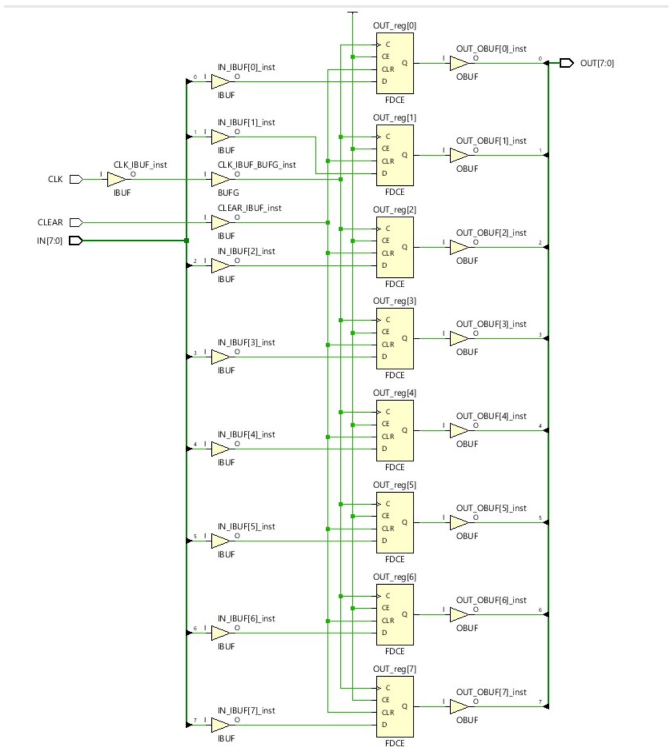
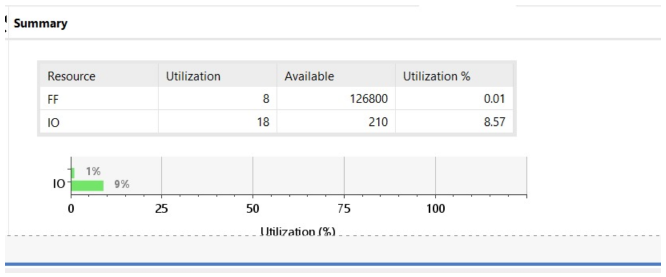


Figure 26: Technology Schematic of 8-bit Register


Summary

Resource	Utilization	Available	Utilization %
FF	8	126800	0.01
IO	18	210	8.57

Figure 27: Utilization Summary of 8-bit Register

The last thing asked in the report about the 8-bit Register is to explain how to implement a 4*8-bit register. We can achieve this change by editing the code as follows.

Listing 11: 4*8-bit Register module

```
module register_array_4x8 (
    input [7:0] IN [3:0],
    input CLK,
    input CLEAR,
    output reg [7:0] OUT [3:0]
);
genvar i;
always @(posedge CLK or posedge CLEAR) begin
    if (CLEAR) begin

        for (i = 0; i < 4; i = i + 1) begin
            OUT[i] <= 8'b0;
        end
    end else begin
        for (i = 0; i < 4; i = i + 1) begin
            OUT[i] <= IN[i];
        end
    end
end
endmodule
```

6 Block RAM

I created the BRAM as requested in the report, wrote the top_module verilog code to connect the BRAM. While editing the constraint file according to the module, I defined clk as shown in the slide and connected a clock to 50MHZ. I filled the memory.coe file with a binary representation of my own number, written twice in a row. I hoped to write a testbench simulation where I could observe this and wait for my school number in the behavioral simulation.

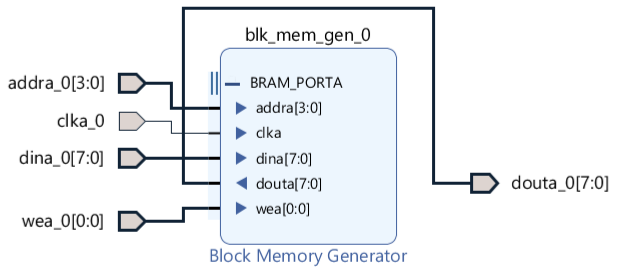


Figure 28: BRAM

Listing 12: Block RAM module

```verilog
`timescale 1ns / 1ps
module top_bram(
  input wea,
  input clk,
  input [3:0] addr,
  output [7:0] dout
);

  wire [7:0] dina;

  BRAM bram_inst(
    .clka_0(clk),
    .wea_0(wea),
    .addra_0(addr),
    .dina_0(dina),
    .douta_0(dout)
  );
endmodule
```

Listing 13: Block RAM module testbench

```verilog
`timescale 1ns / 1ps
module bram_tb;
wire [7:0] DOUT;
reg [3:0] ADDR=0;
reg CLK;
reg ENB=0;
integer count=0;
top_bram uut(.wea(ENB),
  .clk(CLK),
  .addr(ADDR),
  .dout(DOUT));
initial
```

```
begin
    CLK=0;
    forever #10 CLK=~CLK;
end
initial
$monitor ($time, "ADDR=%b , DOUT=%b", ADDR, DOUT);
initial
begin
    while(count<16)
    begin
        ADDR=count;
        #20 count=count+1;
    end
    #20 $finish;
end
endmodule
```

Memory Coe File

memory.coe file is a file type used in FPGA (Field-Programmable Gate Array) based design projects. FPGAs are programmable integrated circuits that allow users to design and implement custom digital logic circuits. In such projects, memory.coe files are used to initialize memory elements in the FPGA. This file is a text-based file that specifies memory content and is often used by FPGA design tools such as Xilinx's Vivado or Altera's Quartus. The word COE stands for "Memory Initialization File".

```
MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
00000100,
00000000,
00000001,
00001001,
00000000,
00000001,
00000000,
00001000,
00000100,
00000000,
00000001,
00001001,
00000000,
00000001,
00000000,
00001000;
```

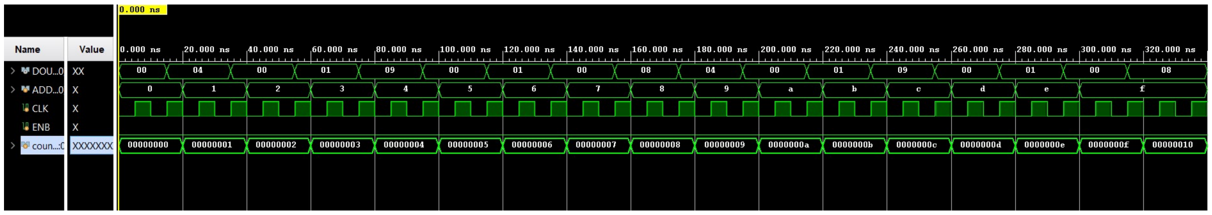

0.000 ns
Name
Value
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
300.000 ns
320.000 ns
DOU...0 XX 00 04 00 01 09 00 01 00 08 04 00 01 09 00 01 00 08
ADD...0 X 0 1 2 3 4 5 6 7 8 9 a b c d e f
CLK X
ENB X
coun...:0 XXXXXXX 00000000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000a 0000000b 0000000c 0000000d 0000000e 0000000f 00000010

Figure 29: Behavioral Simulation of Block RAM

According to the working principle of BRAM and the Verilog code I wrote, the number corresponding to the Address value on each positive edge of the clock should be output in DOUT. We have not given any Religion, we will just observe Dout according to the address. In this context, our simulation gave the correct result as we wanted. On the positive edge of CLK, we can see the value 040190108, which is my number according to the addresd value, twice side by side. Trying to read the data in the 2nd clock posedge signal.

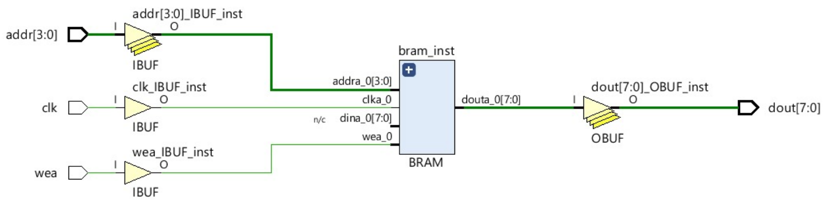


Figure 30: RTL Schematic of Block RAM

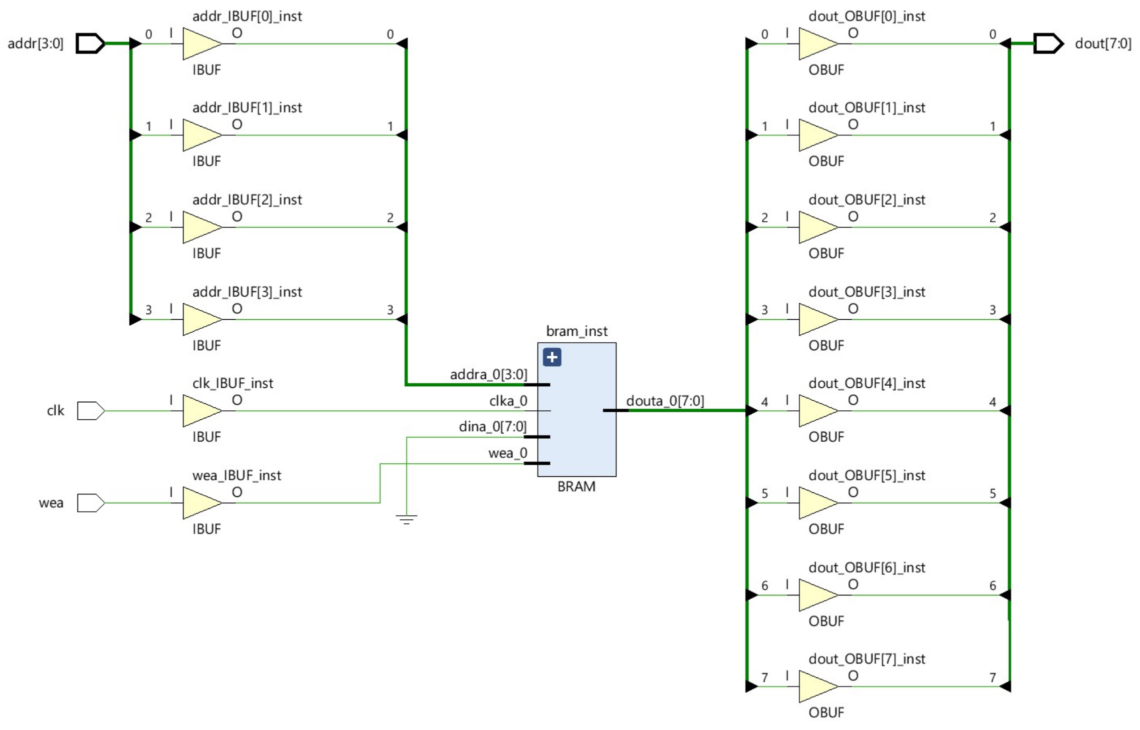
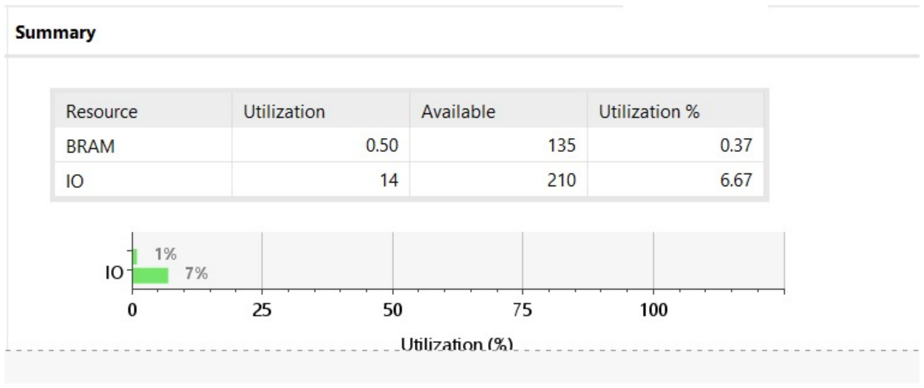


Figure 31: Technology Schematic of Block RAM

While editing the constraint file before the synthesis, the period had to be 20 ns in order to operate at 50 MHz. I arranged the constraint as a basis and gave the appropriate inputs and outputs to my module. Since I created the BRAM module specifically with IP catolog, the circuit was directly output in RTL and technology. According to the utilization report, I observed that I used 0.5 BRAM for this study.

```
##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { wea }];
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { addr[0] }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { addr[1] }];
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { addr[2] }];
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { addr[3] }];
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { dout[0] }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { dout[1] }];
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { dout[2] }];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { dout[3] }];
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { dout[4] }];
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { dout[5] }];
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { dout[6] }];
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { dout[7] }];
set_property -dict { PACKAGE_PIN E3  IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin -period 20.00  -waveform {0 10} [get_ports { clk }];
```


Summary

Resource	Utilization	Available	Utilization %
BRAM	0.50	135	0.37
IO	14	210	6.67

Figure 32: Utilization Summary of Block RAM

7 BONUS: Sliding LEDs

I went step by step as mentioned in the report. First, I opened a new project folder and created the `sliding_leds.v` Verilog design file. I added the testbench code and constraint file given to us in Nineveh to the project.

I created the `sliding_led` module in the `sliding_leds.v` file. I gave `clk`, `rst`, and `SW` inputs as inputs and connected a 16-bit LED to the output. To run the module with inputs in this way, I edited the constraint file as mentioned.

Then I made calculations to create a frequency divider:

The period of a clock operating at a frequency of 100 MHz is 10 ns.

The period of a clock operating at a frequency of 10 Hz is 0.1 s.

The period of a clock operating at a frequency of 20 Hz is 0.05 s.

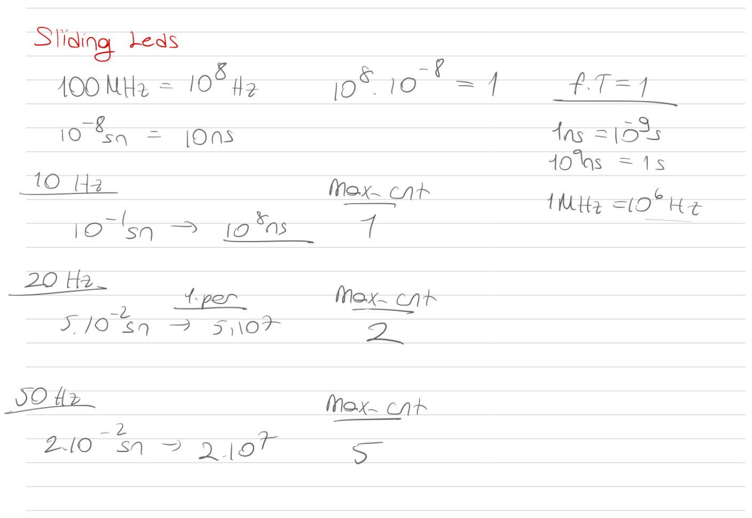
The period of a clock operating at a frequency of 50 Hz is 0.02 s.

As a result of these calculations, the board has completed a 10 Hz period for the first time after 10^8 repetitions of the period, while the board must complete the period 2×10^7 times for the 20 Hz one and 5×10^7 times for 50 Hz to complete a period. In this case, `MAX_CNT` Destination appears as 10^8. Then I started creating the module according to the working principle of the function:

I defined 2 reg 1 wire in the module. The reason why I defined count as reg while defining count_dest as wire was that I needed to define count this way because I used it in always. I set up the working principle of the function in my mind as follows:

The counter reg will increase with every positive clock up to the parameter I set. When it is equal to the parameter, the flag will increase to 1. After stopping for 1 clock, the counter will reset itself and the counter will continue to increase again. In addition, this always button will be triggered asynchronously by active-1 reset, and when rst is set to 1, the flag and counter will be reduced to zero.

The count_dest wire will determine the frequency of the function by changing depending on the Switch status. Since we determined the parameter according to 10 Hz, we need to assign one less than 2 of the parameter value when working at 20 Hz and 1 less divided



Sliding Leds

$100\,MHz = 10^8\,Hz$ $\quad 10^8 \cdot 10^{-8} = 1$ $\quad f \cdot T = 1$

$10^{-8}\,sn = 10ns$ $\quad 1ns = 10^{-9}s$

$10^9 ns = 1s$

10 Hz $\quad$ Max_cnt $\quad 1MHz = 10^6\,Hz$

$10^{-1}\,sn \rightarrow 10^8 ns$ $\quad$ 1

20 Hz $\quad$ 1.per $\quad$ Max_cnt

$5.10^{-2}\,sn \rightarrow 5.10^7$ $\quad$ 2

50 Hz $\quad$ Max_cnt

$2.10^{-2}\,sn \rightarrow 2.10^7$ $\quad$ 5

Figure 33: Calculation Max_Count

by two when working at 50 Hz. Count_dest, in a way, calculates how many hours it will take for the function to run at the relevant hertz to complete its 1 period, according to the 10 nanosecond clock. Here, I set count_dest to 0 in case of 00. I will write an algorithm accordingly in the next always block.
So far, we have counted the counter in the first always block and assigned count_dest according to the value of the switches. In the last always block, we will set the status of the LEDs: In the working principle of the function, if the Sw variable was set to 00, it wanted the LED to maintain its current state; I also set count_dest to 0 when the SW value was 00 above. In the last always block, I created it to be triggered according to positive clock and reset as usual. If count_dest is 0, the Led will remain in its current state; otherwise, each counter count_dest and its multiples will remove the Led by one shift. When the 15th bit of the LED variable becomes 1, the next shift will switch to bit 0 again and continue. I designed this blog based on this idea.

Listing 14: sliding_leds.v module

```
`timescale 1ns / 1ps
module sliding_leds #(parameter MAX_CNT_DEST=100000000)
(input clk,
input rst,
input [1:0] SW,
output reg [15:0]  LED);
wire [$clog2(MAX_CNT_DEST-1)-1:0] count_dest;
reg [$clog2(MAX_CNT_DEST-1)-1:0] counter;
reg flag;

//edge counter
always @(posedge clk or posedge rst)
begin
```

```
    if (rst)
        begin
            flag=0;
            counter=0;
        end
    else
        begin
            if (!flag)
            begin
                if (counter==MAX_CNT_DEST)

                    flag=1;
                else
                    counter=counter + 1;
            end
            else
                begin
                    flag=0;
                    counter=0;
                end
        end

end

assign count_dest = (SW == 2'b01) ? (MAX_CNT_DEST-1) :
                    (SW == 2'b10) ? (MAX_CNT_DEST-1) / 2:
                    (SW == 2'b11) ? (MAX_CNT_DEST-1) /5: 0 ;
always @(posedge clk or posedge rst)
    begin
        if (rst)
            LED<=4'b1;
        else
        begin
            if (count_dest==0)
                begin
                    LED <= LED;
                end
            else
                begin
                 if (counter % count_dest==0 && counter!=0)
                        begin
                        if (LED[15])
                            begin
                                LED <= 1;
                            end
```

```
                                        else
                                            LED <= LED << 1;
                                        end
                                end
                        end
                end
endmodule
```

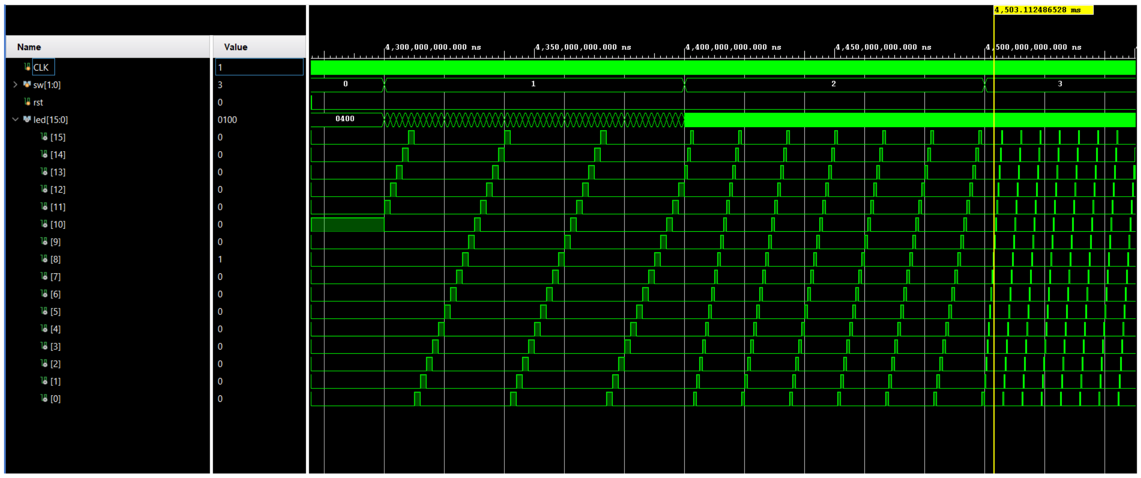
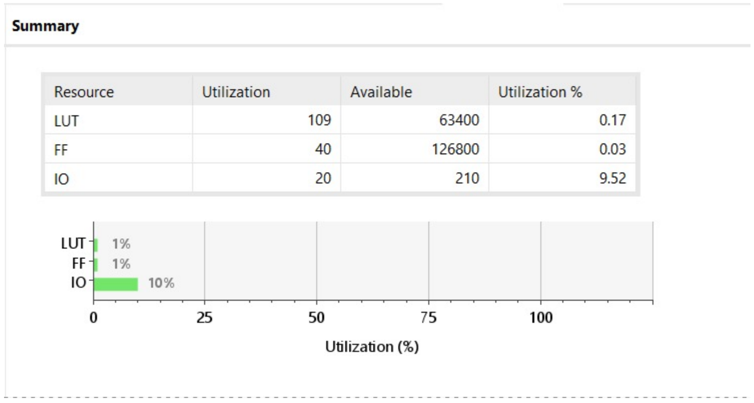


Figure 34: Behavioral Simulations of Sliding Leds

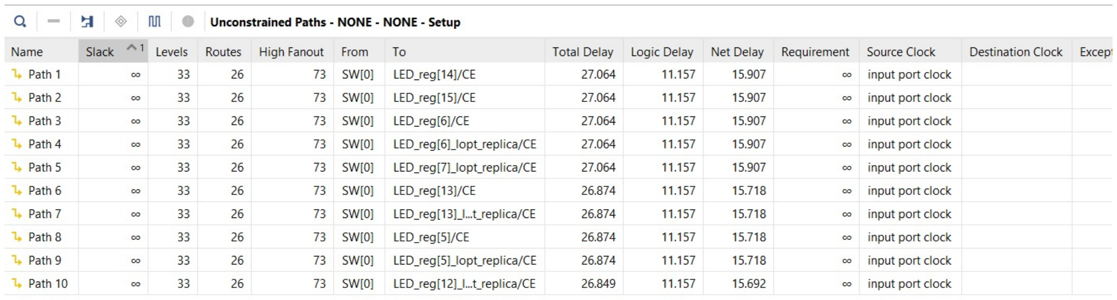
I reduced MAX_CNT_DEST to 10^5 so that we could observe better in the simulation and left the simulation as given. As you can see, when the switch is on switch 1, the LEDs have the longest burning time in the index, followed by 2 and 3. When it is at 00, the LED remains in the same state.
I reduced the parameter value to 500 and received implementation result reports. According to the report, 40 FF, 109 LUT and 20 IO ports were used to perform the function.



Summary

Resource	Utilization	Available	Utilization %
LUT	109	63400	0.17
FF	40	126800	0.03
IO	20	210	9.52

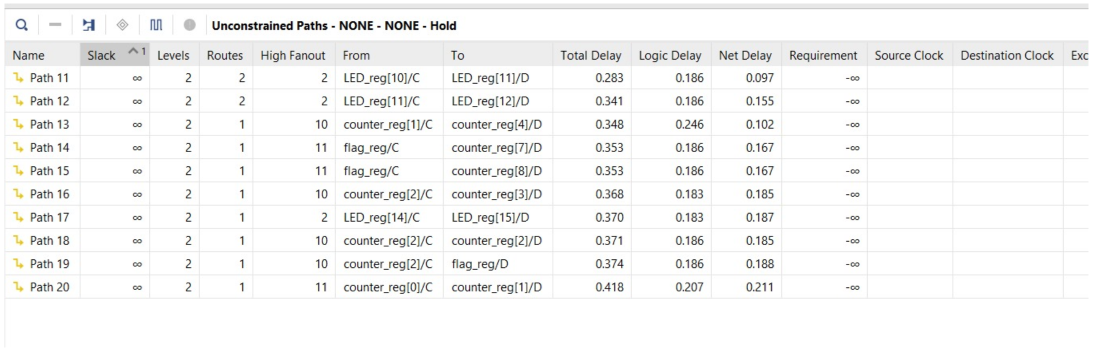
Figure 35: Utilization Report of Sliding Leds



Unconstrained Paths - NONE - NONE - Setup

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Excep
Path 1	∞	33	26	73	SW[0]	LED_reg[14]/CE	27.064	11.157	15.907	∞	input port clock		
Path 2	∞	33	26	73	SW[0]	LED_reg[15]/CE	27.064	11.157	15.907	∞	input port clock		
Path 3	∞	33	26	73	SW[0]	LED_reg[6]/CE	27.064	11.157	15.907	∞	input port clock		
Path 4	∞	33	26	73	SW[0]	LED_reg[6]_lopt_replica/CE	27.064	11.157	15.907	∞	input port clock		
Path 5	∞	33	26	73	SW[0]	LED_reg[7]_lopt_replica/CE	27.064	11.157	15.907	∞	input port clock		
Path 6	∞	33	26	73	SW[0]	LED_reg[13]/CE	26.874	11.157	15.718	∞	input port clock		
Path 7	∞	33	26	73	SW[0]	LED_reg[13]_l...t_replica/CE	26.874	11.157	15.718	∞	input port clock		
Path 8	∞	33	26	73	SW[0]	LED_reg[5]/CE	26.874	11.157	15.718	∞	input port clock		
Path 9	∞	33	26	73	SW[0]	LED_reg[5]_lopt_replica/CE	26.874	11.157	15.718	∞	input port clock		
Path 10	∞	33	26	73	SW[0]	LED_reg[12]_l...t_replica/CE	26.849	11.157	15.692	∞	input port clock		

Figure 36: Critical Path Setup Time of Sliding Leds



Unconstrained Paths - NONE - NONE - Hold

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exc
Path 11	∞	2	2	2	LED_reg[10]/C	LED_reg[11]/D	0.283	0.186	0.097	-∞			
Path 12	∞	2	2	2	LED_reg[11]/C	LED_reg[12]/D	0.341	0.186	0.155	-∞			
Path 13	∞	2	1	10	counter_reg[1]/C	counter_reg[4]/D	0.348	0.246	0.102	-∞			
Path 14	∞	2	1	11	flag_reg/C	counter_reg[7]/D	0.353	0.186	0.167	-∞			
Path 15	∞	2	1	11	flag_reg/C	counter_reg[8]/D	0.353	0.186	0.167	-∞			
Path 16	∞	2	1	10	counter_reg[2]/C	counter_reg[3]/D	0.368	0.183	0.185	-∞			
Path 17	∞	2	1	2	LED_reg[14]/C	LED_reg[15]/D	0.370	0.183	0.187	-∞			
Path 18	∞	2	1	10	counter_reg[2]/C	counter_reg[2]/D	0.371	0.186	0.185	-∞			
Path 19	∞	2	1	10	counter_reg[2]/C	flag_reg/D	0.374	0.186	0.188	-∞			
Path 20	∞	2	1	11	counter_reg[0]/C	counter_reg[1]/D	0.418	0.207	0.211	-∞			

Figure 37: Critical Path Hold Time of Sliding Leds

8 Appendix: Bonus Part Test Bench

I added test bench given in homework below:

Listing 15: Bonus part testbench

```
`timescale 1ns / 1ps

module sliding_led_tb();
    reg CLK;
    reg [1:0]sw;
    reg rst;

    wire [15:0] led;

    initial
    begin
    CLK=1;
    rst=0;#1;rst=1;#1;rst=0;
    end
    always #10 CLK=~CLK;

    initial
    begin
    sw[0]=0;
    sw[1]=0;
    end
    always
    begin
    #10000000 sw[0]=~sw[0]; //10
    #10000000 sw[0]=~sw[0]; sw[1]=~sw[1]; //01
    #10000000 sw[0]=~sw[0]; //10
    #10000000 sw[0]=~sw[0];  //00

    end

    sliding_leds clkd(

    .clk(CLK),
    .SW(sw),
    .rst(rst),
      .LED(led)
  );

endmodule
```