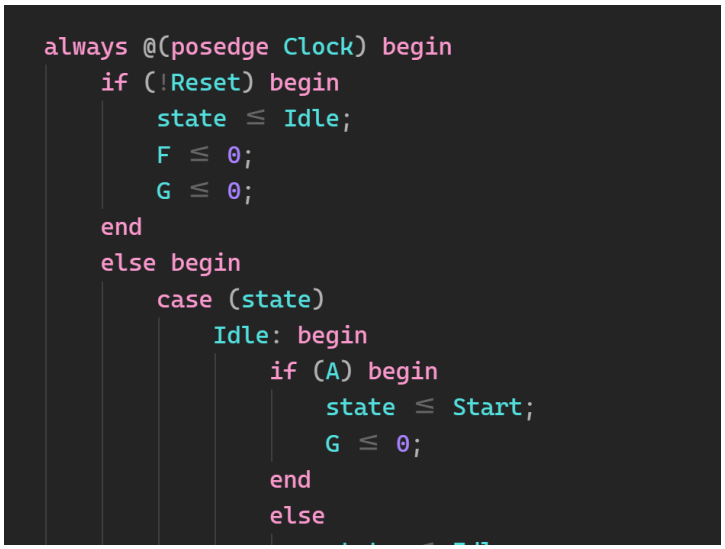
RTL Design Portfolio

Core Digital Logic Modules Implemented in Verilog



```
always @(posedge Clock) begin
    if (!Reset) begin
        state <= Idle;
        F <= 0;
        G <= 0;
    end
    else begin
        case (state)
            Idle: begin
                if (A) begin
                    state <= Start;
                    G <= 0;
                end
                else
```

Compiled and completed by

Chirag Gupta

Connect with me on:

- LinkedIn
- Github

Table of Contents

Simulation of 8:1 MUX using 2:1 MUX

Aim:

To write a Verilog HDL program to simulate 8 by 1 multiplexer using 2 by 1 multiplexer and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

2-to-1 Multiplexer (2:1 MUX)

A 2-to-1 multiplexer is a basic digital device used to select one of two input signals and pass it to a single output line based on the value of a control or select signal. It has two data inputs, I_0 and I_1, one select line S, and a single output Y.

- When the select line S is 0, the output Y will be equal to I_0 (i.e., $Y=I_0$).
- When the select line S is 1, the output Y will be equal to I_1 (i.e., $Y=I_1$).

The logical expression for the output Y can be written as: $Y=S\cdot I_1+\bar{S}\cdot I_0$

In this equation, $S\cdot I_1$ selects input I_1 when S=1, and $\bar{S}\cdot I_0$ selects input I_0 when S=0. The 2-to-1 MUX is used in a variety of applications such as data routing, conditional operations, and in building larger multiplexers.

8-to-1 Multiplexer (8:1 MUX)

An 8-to-1 multiplexer is a digital circuit that selects one of eight input signals and passes it to a single output line. It has eight data inputs I_0 to I_7, three select lines S_0, S_1, and S_2, and a single output Y.

- The three select lines S_0, S_1, and S_2 are used to determine which of the eight inputs will be connected to the output.
- Depending on the binary value of the select lines, one of the inputs I_0 to I_7 is passed to the output.

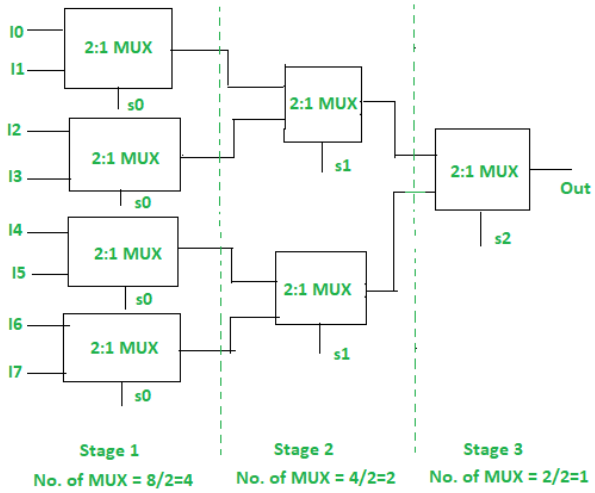
For example:

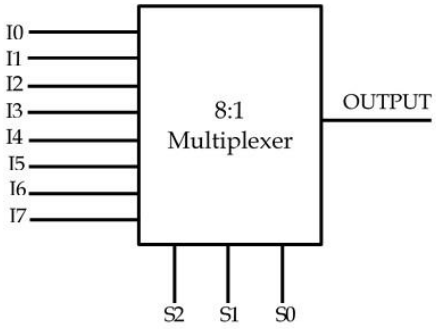
- If $S_2S_1S_0=000$, the output $Y=I_0$,
- If $S_2S_1S_0=001$, the output $Y=I_1$, and so on.

The logical expression for the output Y is:

$Y=\bar{S}_2\cdot\bar{S}_1\cdot\bar{S}_0\cdot I_0+\bar{S}_2\cdot\bar{S}_1\cdot S_0\cdot I_1+\cdots+S_2\cdot S_1\cdot S_0\cdot I_7$

The 8-to-1 MUX is commonly used in applications where multiple data sources need to be routed to a single output, such as in digital systems for data selection, communication systems, and arithmetic circuits. It can also be built using smaller multiplexers, like cascading 2:1 MUXes or 4:1 MUXes to achieve the 8-to-1 functionality.





Truth table:

Input	Input	Input	Output
S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Code:

mux8by1using2by1.v × mux8by1using2by1_tb.v ×

C:/Users/Chirag Gupta/combinational_circuits/combinational_circuits.srcs/sources_1/new/mux8by1using2by1.v

```
1  `timescale 1ns / 1ps
2  module mux8bylusing2byl(y,s,i);
3  input [7:0]i;
4  input [2:0]s;
5  output y;
6  wire [5:0]x;
7
8  //stage 1
9  assign x[0]=s[0]?i[1]:i[0];
10 assign x[1]=s[0]?i[3]:i[2];
11 assign x[2]=s[0]?i[5]:i[4];
12 assign x[3]=s[0]?i[7]:i[6];
13 //stage 2
14 assign x[4]=s[1]?x[1]:x[0];
15 assign x[5]=s[1]?x[3]:x[2];
16 //stage 3
17 assign y=s[2]?x[5]:x[4];
18
19 endmodule
20
```

Testbench:

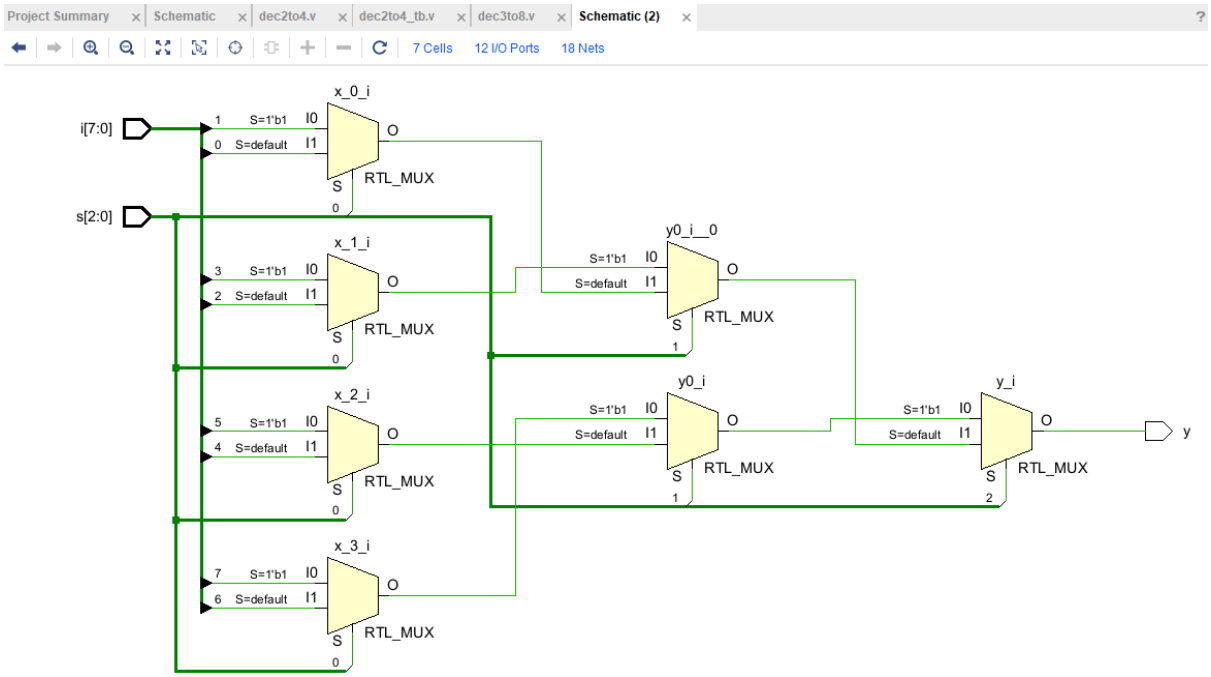
mux8by1using2by1.v × mux8by1using2by1_tb.v ×

C:/Users/Chirag Gupta/combinational_circuits/combinational_circuits.srcs/sim_1/new/mux8by1using2by1_tb.v

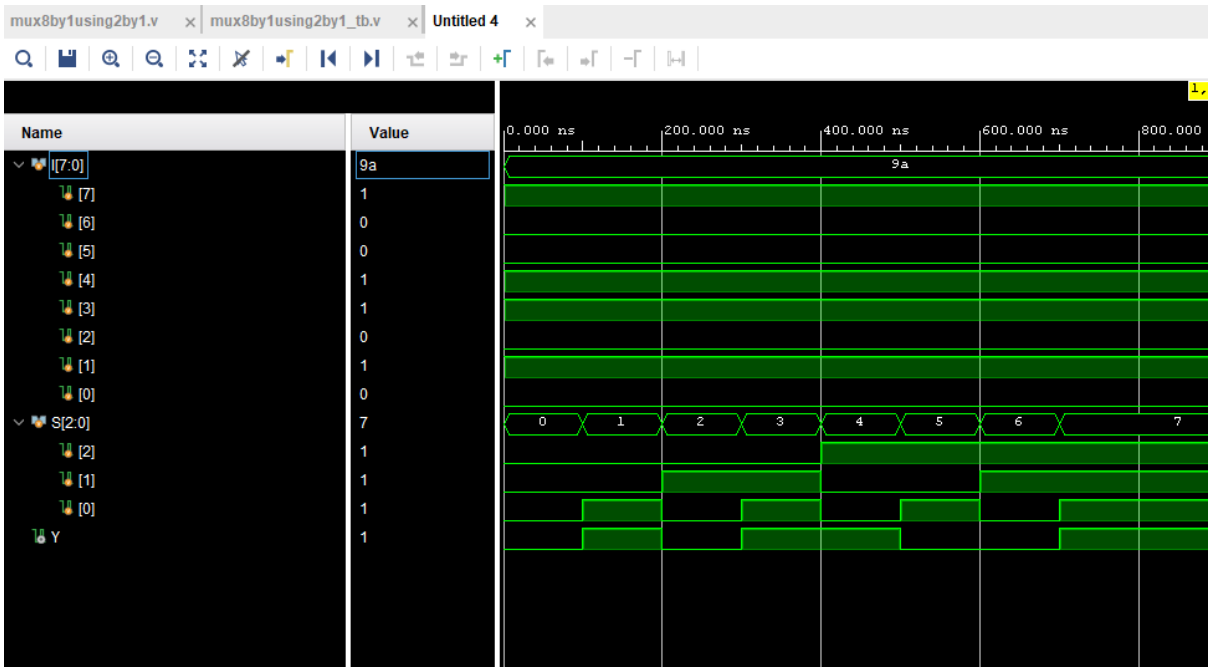
```
1  `timescale 1ns / 1ps
2  module mux8bylusing2byl_tb;
3  reg [7:0]I;
4  reg [2:0]S;
5  wire Y;
6
7  mux8bylusing2byl uut(.i(I),.s(S),.y(Y));
8  initial
9  begin
10 I=8'b10011010;
11 S=0;
12 #100;
13 S=1;
14 #100;
15 S=2;
16 #100;
17 S=3;
18 #100;
19 S=4;
20 #100;
21 S=5;
22 #100;
23 S=6;
24 #100;
25 S=7;
26 #100;
27 end
28 endmodule
29
```

Observation:

RTL Analysis:



Waveform:



Result:

8:1 MUX was simulated using Xilinx simulator using 2:1 MUX and its functionality has been verified.

Simulation and Verification of 3 to 8 Decoder using 2 to 4 Decoder

Aim:

To write a Verilog HDL program to simulate 3 to 8 decoder using 2 to 4 decoder and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

2-to-4 Decoder

A 2-to-4 decoder is a digital circuit that takes two binary inputs and decodes them into four unique outputs. It has two input lines, typically labelled as A1 and A0, and four outputs Y0 to Y3. The function of the decoder is to generate one active output (logic 1 or high) for each unique combination of the inputs, while all other outputs remain inactive (logic 0 or low).

The relationship between inputs and outputs is as follows:

- When A_1A_0=00, the output Y_0=1, and Y_1, Y_2, Y_3=0
- When A_1A_0=01, the output Y_1=1, and Y_0, Y_2, Y_3=0
- When A_1A_0=10, the output Y_2=1, and Y_0, Y_1, Y_3=0
- When A_1A_0=11, the output Y_3=1, and Y_0, Y_1, Y_2=0

The truth table for a 2-to-4 decoder is:

Input	Input	Output	Output	Output	Output
A_0	A_1	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The 2-to-4 decoder can be used in applications like memory address decoding, where specific memory locations are selected based on input combinations, and in enabling specific functions within a digital system.

3-to-8 Decoder

A 3-to-8 decoder is a digital circuit that decodes three binary inputs into one of eight outputs. It has three input lines, usually labelled as A_2, A_1, and A_0, and eight outputs Y_0 to Y_7. For each unique combination of the three input bits, exactly one output will be active (logic 1), while all other outputs will be inactive (logic 0).

The input-to-output mapping is as follows:

- When $A_2A_1A_0$=000, the output Y_0=1, and Y_1 to Y_7=0
- When $A_2A_1A_0$=001, the output Y_1=1, and Y_0, Y_2, …, Y_7=0
- When $A_2A_1A_0$=010, the output Y_2=1, and Y_0, Y_1, Y_3, …, Y_7=0

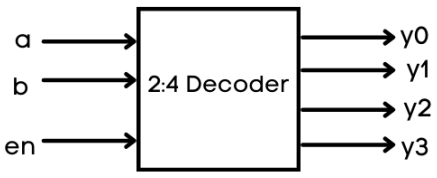
And so on, until:

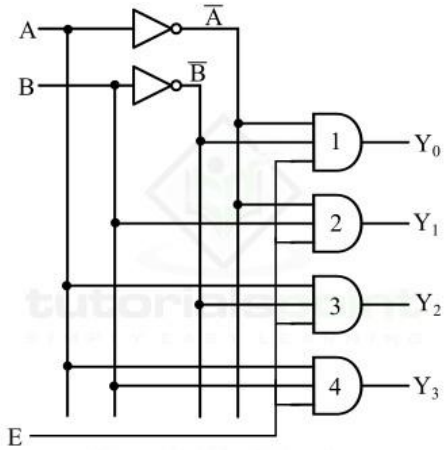
- When $A_2A_1A_0$=111, the output Y_7=1, and Y_0, Y_1, …, Y_6=0

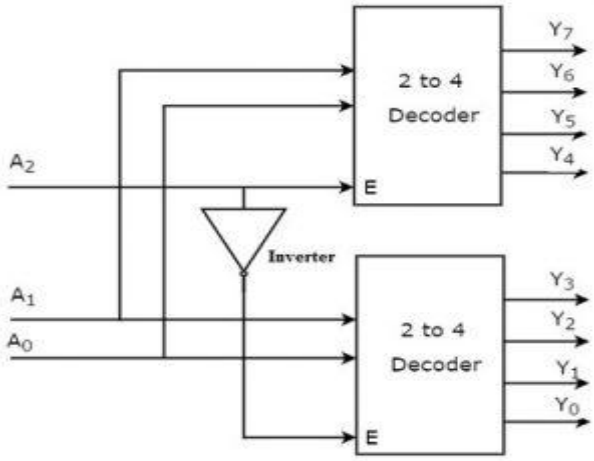
The truth table for a 3-to-8 decoder is:

Input	Input	Input	Output	Output	Output	Output	Output	Output	Output	Output
A_2	A_1	A_0	Y_0	Y_1	Y_2	Y_3	T_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

A 3-to-8 decoder is commonly used in applications such as memory selection, address decoding in computers, and other systems where a specific output needs to be activated based on a combination of binary inputs. It is a useful tool for expanding the number of controlled outputs in digital systems.







Code:

Code for 2 to 4 decoder:

dec2to4.v × dec2to4_tb.v ×

C:/Users/Chirag Gupta/combinational_circuits/combinational_circuits.srcs/sources_1/new/dec2to4.v

```
1  `timescale 1ns / 1ps
2  module dec2to4(y,a,e);
3  input [1:0]a;
4  input e;
5  output [3:0]y;
6  assign y[0]=(~a[1])&(~a[0])&e;
7  assign y[1]=(~a[1])&a[0]&e;
8  assign y[2]=a[1]&(~a[0])&e;
9  assign y[3]=a[1]&a[0]&e;
10 endmodule
11
```

Testbench for 2 to 4 decoder:

```
1  `timescale 1ns / 1ps
2  module dec2to4_tb;
3  reg [1:0]A;
4  reg E;
5  wire [3:0]Y;
6
7  dec2to4 uut (.a(A),.e(E),.y(Y));
8  initial
9  begin
10 E=1;
11 A=2'b00;
12 #100;
13 A=2'b01;
14 #100;
15 A=2'b10;
16 #100;
17 A=2'b11;
18 #100;
19 end
20 endmodule
```

Code for 3 to 8 decoder:

dec2to4.v | dec2to4_tb.v | **dec3to8.v** | dec3to8_tb.v

C:/Users/Chirag Gupta/combinational_circuits/combinational_circuits.srcs/sources_1/new/dec3to8.v

```
1  `timescale 1ns / 1ps
2  module dec3to8(y,e,a);
3  input [1:0]a;
4  input e;
5  output [7:0]y;
6
7  dec2to4 d1(y[3:0],a[1:0],(~e));
8  dec2to4 d2(y[7:4],a[1:0],e);
9  endmodule
10
```

Testbench for 3 to 8 decoder:

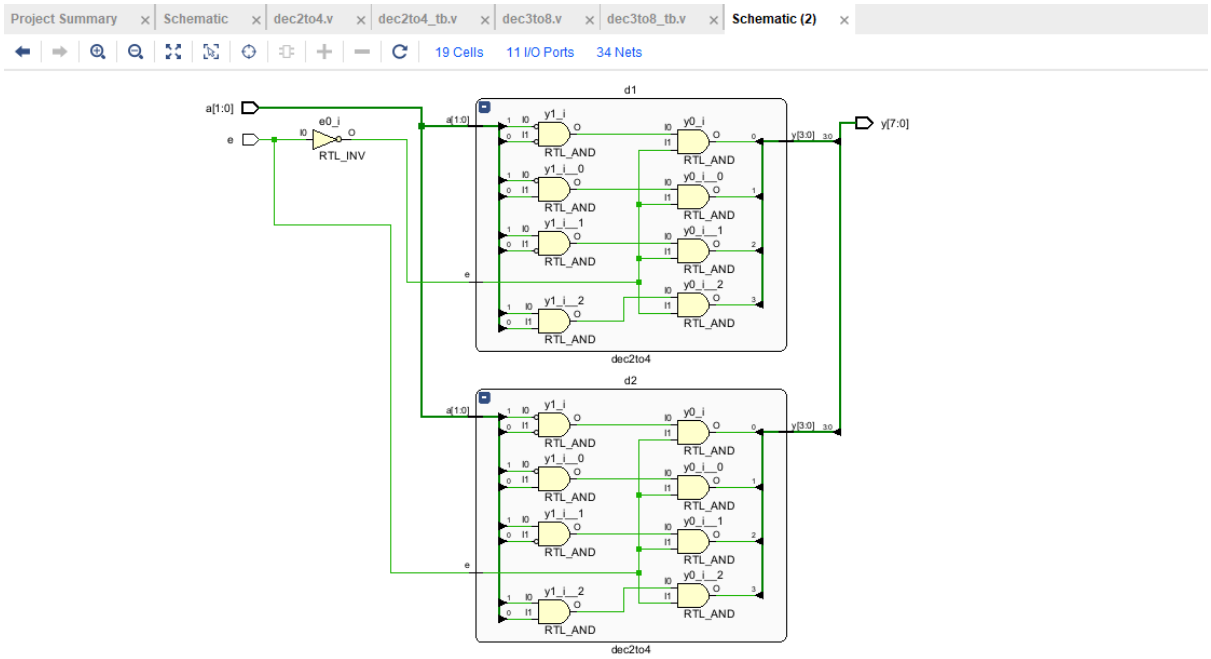
dec2to4.v | dec2to4_tb.v | dec3to8.v | **dec3to8_tb.v**

C:/Users/Chirag Gupta/combinational_circuits/combinational_circuits.srcs/sim_1/new/dec3to8_tb.v

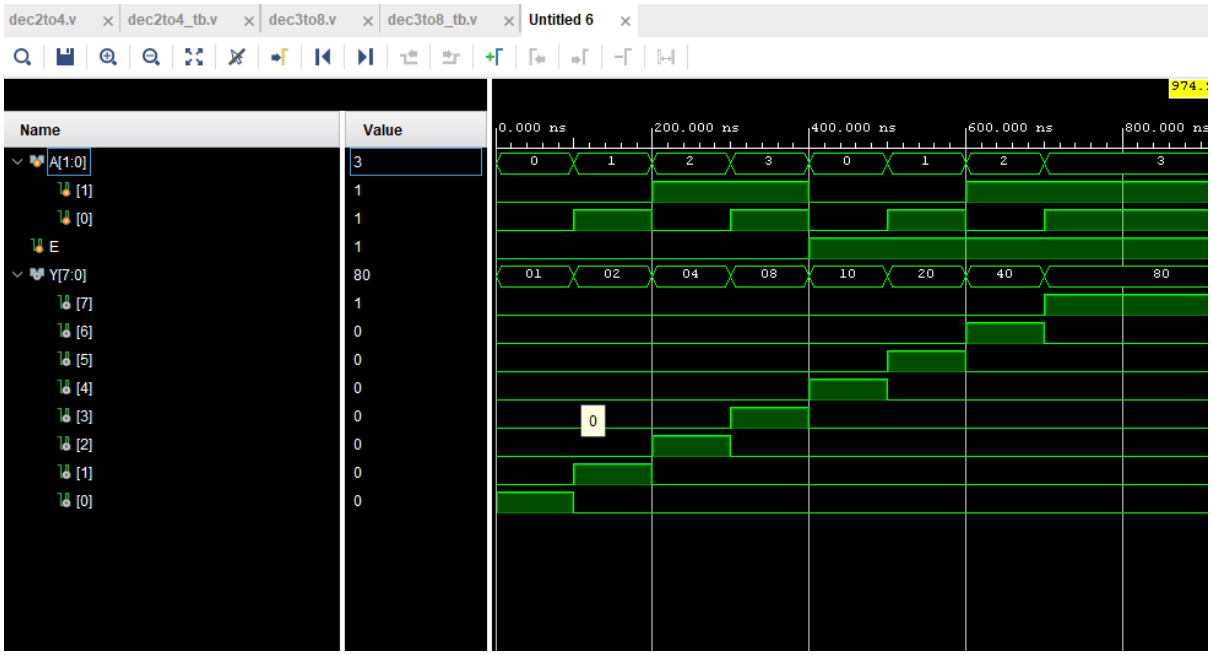
```
1  `timescale 1ns / 1ps
2  module dec3to8_tb;
3  reg [1:0]A;
4  reg E;
5  wire [7:0]Y;
6
7  dec3to8 uut(.a(A),.y(Y),.e(E));
8  initial
9  begin
10 E=0;
11 A=0;
12 #100;
13 A=1;
14 #100;
15 A=2;
16 #100;
17 A=3;
18 #100;
19 E=1;
20 A=0;
21 #100;
22 A=1;
23 #100;
24 A=2;
25 #100;
26 A=3;
27 #100;
28 end
29 endmodule
30
```

Observation:

RTL Analysis:



Waveform:



Result:

3 to 8 decoder was simulated using Xilinx simulator using 2 to 4 decoder and its functionality has been verified.

Conversion of Flip Flop (JK Flip Flop to T Flip Flop)

Aim:

To write a Verilog HDL program to convert a JK flip flop to a T flip flop and simulate D flip flop and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A **JK flip-flop** can be converted into a **T flip-flop** by appropriately configuring its inputs (**J** and **K**) so that it exhibits the behaviour of a T flip-flop. A T flip-flop (Toggle flip-flop) toggles its output on each clock pulse if the **T** input is HIGH, and holds its state if the **T** input is LOW.

Characteristic Table:

CLK	T	Q_n	Q_{n+1}
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0

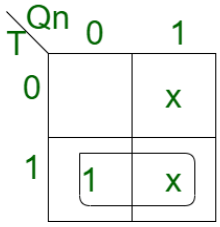
Excitation Table for JK flip flop

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

T	Q_n	Q_{n+1}	J	K
0	0	0	0	X
0	1	1	X	0
1	0	1	1	X
1	1	0	X	1

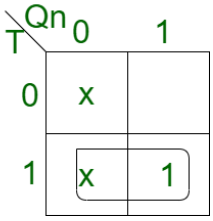
K-maps for finding the Boolean expression of J and K:

For J (T, Q_n),



J=T

For K (T, Q_n),



K=T

To mimic a T flip-flop:

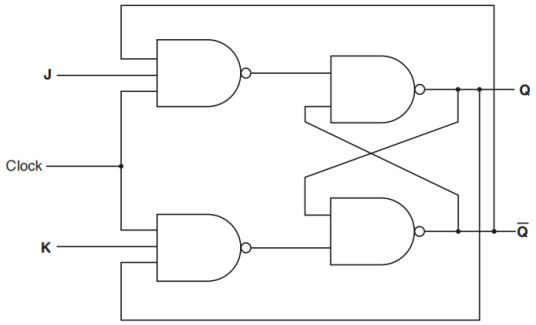
1. For toggling:
    - The JK flip-flop must have **J = 1** and **K = 1** to toggle the output state.
2. For holding:
    - The JK flip-flop must have **J = 0** and **K = 0** to retain its current state.

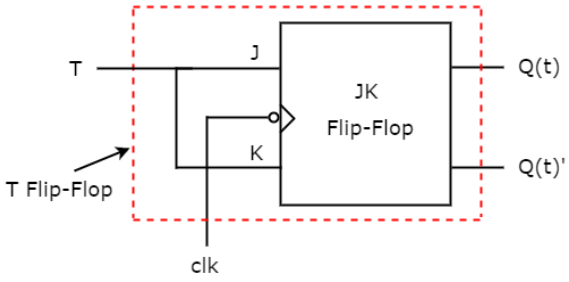
This can be achieved by connecting both the **J** and **K** inputs of the JK flip-flop to the **T** input.

Circuit Implementation:

- **J = T**
- **K = T**

When **T = 0**, the JK flip-flop holds its state (**J = 0, K = 0**). When **T = 1**, the JK flip-flop toggles its state (**J = 1, K = 1**).





Code:

```
`timescale 1ns / 1ps
module jk_flipflop (clk, rst, j, k, qn, qnbar);
input wire clk, rst, j, k;
output reg qn, qnbar;

always @(posedge clk or ~rst)
begin
    if (rst) begin
        qn <= 0;
        qnbar <= 0;
    end else begin
        case ({j, k})
            2'b00: begin
                qn <= qn;
                qnbar<=qnbar;
            end 2'b01: begin
                qn <= 0;
                qnbar<=1;
            end 2'b10: begin
                qn <= 1;
                qnbar<=0;
            end 2'b11: begin
                qn <= ~qn;
                qnbar <= ~qnbar;
            end default: begin
                qn <= qn;
                qnbar<=qnbar;
            end
        endcase
    end
end
endmodule
```

```
`timescale 1ns / 1ps
module t_flipflop(t, clk, rst, qn, qnbar);
input wire t, clk, rst;
output reg qn, qnbar;

jk_flipflop jkff(clk, rst, t, t, qn, qnbar);
endmodule
```

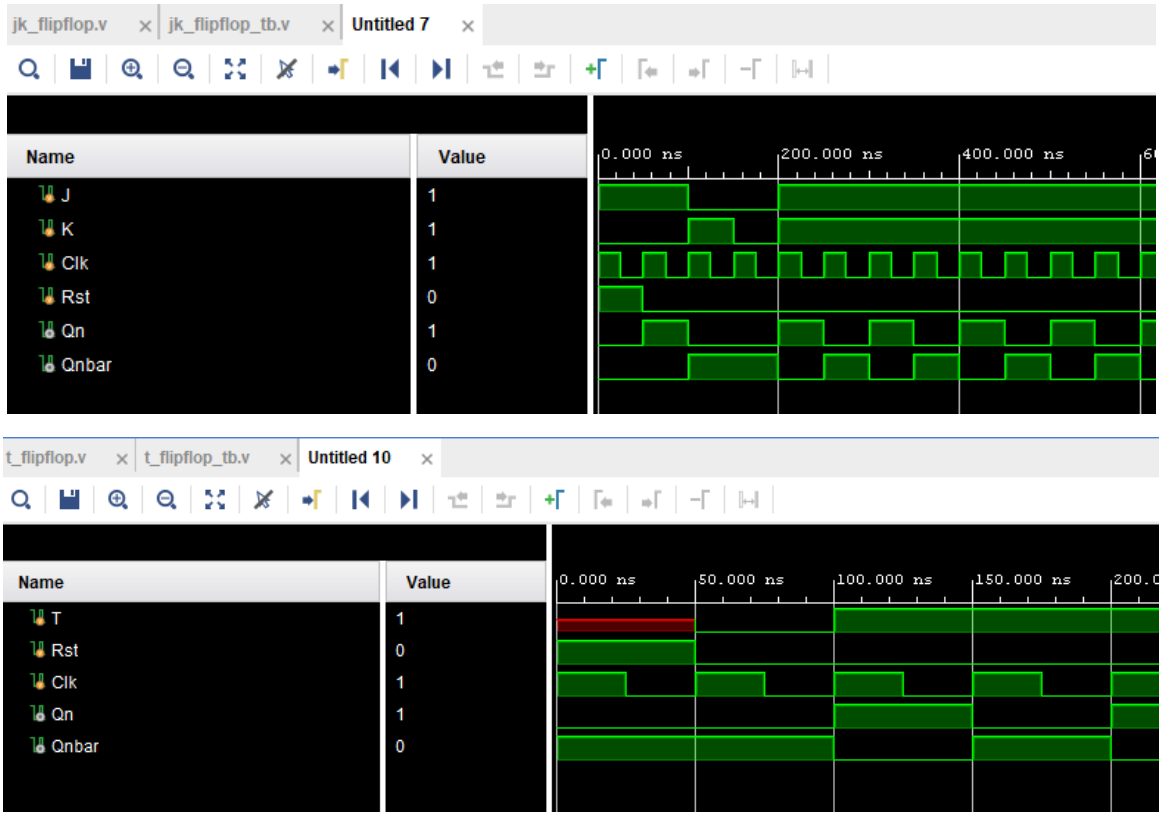
Testbench:

```
`timescale 1ns / 1ps

module jk_flipflop_tb;
reg J,K;
reg Clk, Rst;
wire Qn, Qnbar;

jk_flipflop uut (.j(J), .k(K), .clk(Clk), .rst(Rst), .qn(Qn), .qnbar(Qnbar));
initial
begin
    J=1;
    K=0;
    Clk=1;
    Rst=1;
    #50;
    Rst=0;
    #50;
    J=0;
    K=1;
    #50;
    J=0;
    K=0;
    #50;
    J=1;
    K=1;
end

always
begin
#25;
Clk=~Clk;
end
endmodule
```

```
`timescale 1ns / 1ps
module t_flipflop_tb;
reg T, Rst, Clk;
wire Qn, Qnbar;

t_flipflop uut(.t(T), .clk(Clk), .rst(Rst), .qn(Qn), .qnbar(Qnbar));
initial
begin
T=0;
Clk=1;
Rst=1;
#10;
Rst=0;
#50;
T=1;
#50;
end

always
begin
#25;
Clk=~Clk;
end
endmodule
```

Observation:

Output Waveforms:



Result:

JK flip flop was converted to T flip flop and it was simulated using Xilinx simulator and its functionality has been verified.

Simulation of SR Flip-Flop

Aim:

To write a Verilog HDL program to simulate SR flip flop and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

An **SR flip-flop** (Set-Reset flip-flop) is a basic bistable circuit widely used in digital electronics for data storage, signal synchronization, and control logic. It is categorized as a **sequential circuit** because its output depends not only on the current inputs but also on the previous state of the system.

The SR flip-flop has:

- **Two inputs:** Set (S) and Reset (R).
- **Two outputs:** Q (normal output) and Q' (complementary output), which are always inverses of each other.

It is designed to store a single bit of binary information, with its operation governed by the logic levels of the S and R inputs.

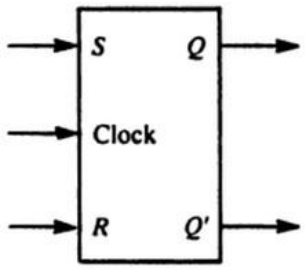
Truth Table:

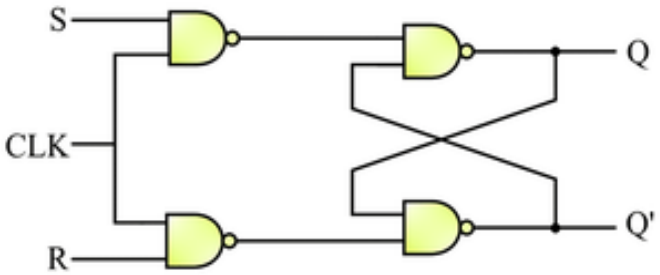
S	R	Q (Next State)	Q' (Next State)	Description
0	0	No change	No change	Memory State
0	1	0	0	Reset State
1	0	1	0	Set state
1	1	Invalid	Invalid	Undefined State (Ambiguous)

1. **Set (S=1, R=0):** When the **Set** input is high, the flip-flop forces the output **Q=1** and **Q'=0**, regardless of its previous state. This is known as the **Set state**.
2. **Reset (S=0, R=1):** When the **Reset** input is high, the flip-flop forces the output **Q=0** and **Q'=1**, clearing the stored value. This is the **Reset state**.
3. **Memory State (S=0, R=0):** When both inputs are low, the flip-flop retains its previous state, maintaining its stored value. This is the **hold or memory state**.
4. **Invalid State (S=1, R=1):** When both inputs are high simultaneously, the outputs **Q** and **Q'** become logically inconsistent, as both are forced to be 1. This creates an undefined or invalid state, which must be avoided.

Implementation:

1. **Using NOR Gates:**
    - Two NOR gates are cross-coupled such that the output of each gate is connected to an input of the other.
    - Active-high inputs (S=1 and R=1) produce an undefined state.
2. **Using NAND Gates:**
    - Two NAND gates are cross-coupled similarly.
    - Active-low inputs are used, meaning the logic is inverted compared to NOR-based SR flip-flops.





Code:

Project Summary | sr_flipflop.v | sr_flipflop_tb.v

C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/sr_flipflop.v

```
1  `timescale 1ns / 1ps
2  module sr_flipflop(qn,qnbar,s,r,clk);
3  output reg qn, qnbar;
4  input wire s, r, clk;
5
6  always @(posedge clk)
7  begin
8       if (s && r)
9       begin
10         qn<=1'bx;
11         qnbar<=1'bx;
12      end
13      else if (s && ~r)
14      begin
15         qn<=1;
16         qnbar<=0;
17      end
18      else if (~s && r)
19      begin
20         qn<=0;
21         qnbar<=1;
22      end
23      else if (~s && ~r)
24      begin
25         qn<=qn;
26         qnbar<=qnbar;
27      end
28 end
29 endmodule
```

Testbench:

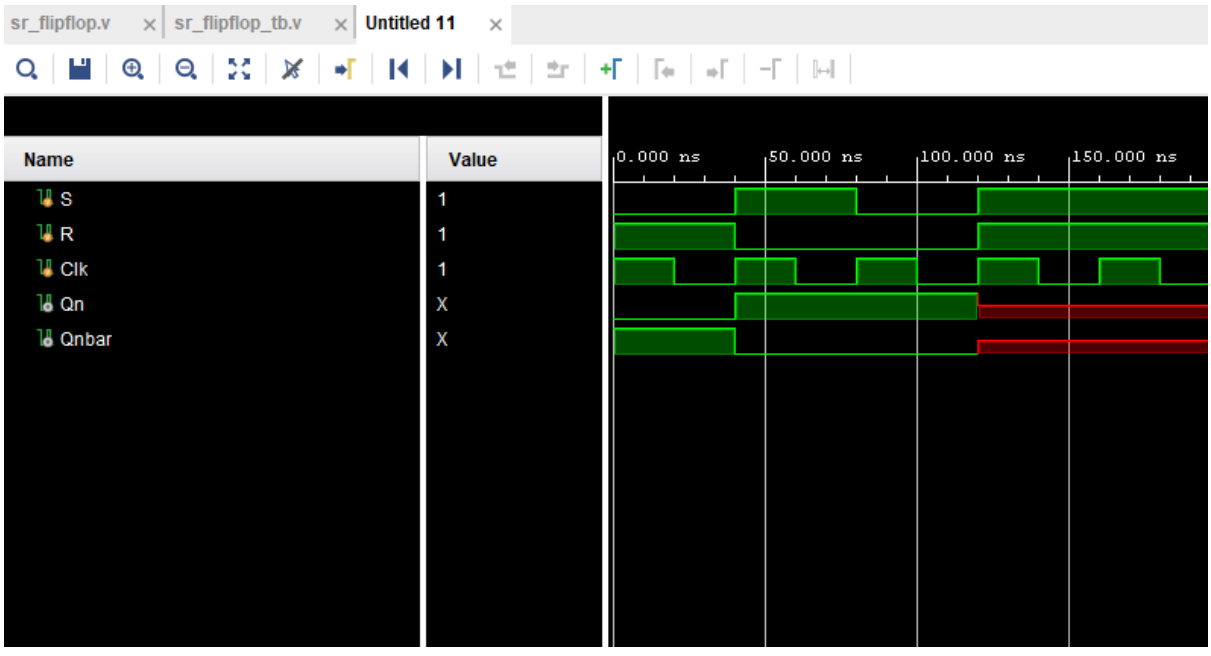
Project Summary | sr_flipflop.v | sr_flipflop_tb.v

C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sim_1/new/sr_flipflop_tb.v

```
1  `timescale 1ns / 1ps
2  module sr_flipflop_tb;
3  reg S, R , Clk;
4  wire Qn, Qnbar;
5
6  sr_flipflop uut(.s(S), .r(R), .clk(Clk), .qn(Qn), .qnbar(Qnbar));
7  initial begin
8  Clk=1;
9  S=0;
10 R=1;
11 #40;
12 S=1;
13 R=0;
14 #40;
15 S=0;
16 R=0;
17 #40;
18 S=1;
19 R=1;
20 end
21
22 always
23 begin
24 #20;
25 Clk=~Clk;
26 end
27 endmodule
```

Observation:

Output Waveforms:



Result:

SR flip flop was simulated using Xilinx simulator and its functionality has been verified.

Simulation of D Flip Flop

Aim:

To write a Verilog HDL program to simulate D flip flop and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A **D flip-flop** (Data or Delay flip-flop) is a sequential logic circuit that captures the value of the data input (**D**) on a specific clock edge and stores it until the next clock cycle. It is one of the simplest and most widely used flip-flops in digital electronics due to its predictable behaviour and elimination of invalid states.

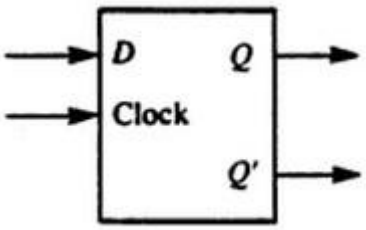
The D flip-flop has:

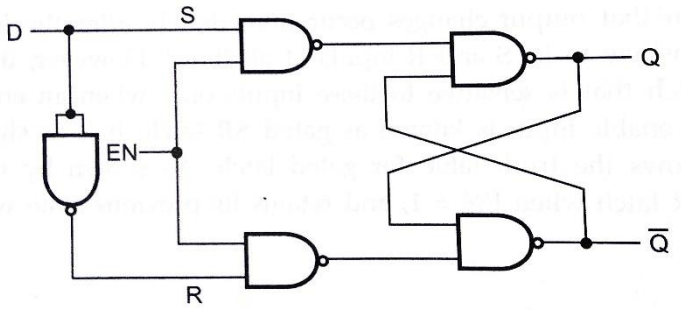
- **One data input (D):** Determines the value to be stored.
- **One clock input (CLK):** Controls when the data is stored.
- **Two outputs (Q and Q'):** The stored value and its complement.

Truth Table:

D	Q (Next State)	Q' (Next State)	Description
0	0	1	Stores 0
1	1	0	Stores 1

1. **On Clock Edge:**
    - When a rising edge (positive edge-triggered) or falling edge (negative edge-triggered) of the clock occurs, the value of **D** is captured and stored in **Q**.
    - The output remains constant until the next clock edge.
2. **Transparent Latch Behavior (Optional):**
    - If implemented with an enable signal instead of a clock, the D flip-flop acts as a transparent latch, where **Q** follows **D** as long as the enable is active.





Code:

d_flipflop.v × d_flipflop_tb.v ×

C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/d_flipflop.v

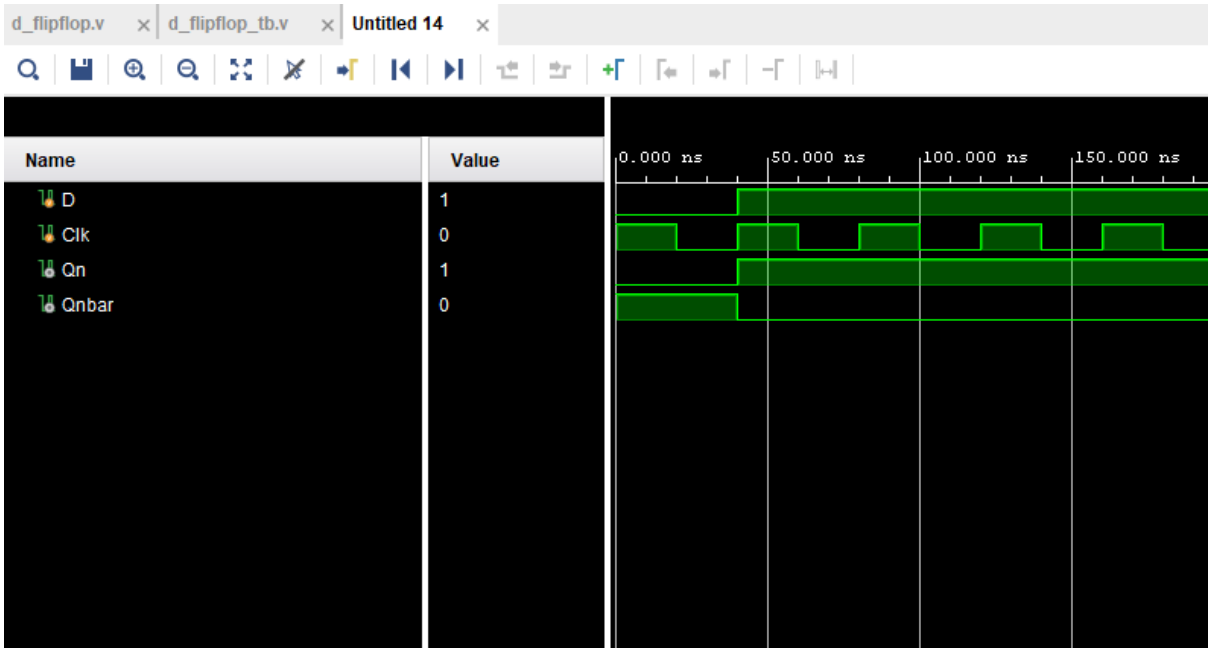
```
1  `timescale 1ns / 1ps
2  module d_flipflop(d, clk, qn, qnbar);
3  input wire d, clk;
4  output reg qn, qnbar;
5
6  always @(posedge clk)
7  begin
8      if (d)
9      begin
10         qn<=1;
11         qnbar<=0;
12     end
13     else if (~d)
14     begin
15         qn<=0;
16         qnbar<=1;
17     end
18 end
19 endmodule
20
```

Testbench:

```
1  `timescale 1ns / 1ps
2  module d_flipflop_tb;
3  reg D, Clk;
4  wire Qn, Qnbar;
5
6  d_flipflop uut(.d(D), .clk(Clk), .qn(Qn), .qnbar(Qnbar));
7  initial
8  begin
9      Clk=1;
10     D=0;
11     #40;
12     D=1;
13     #40;
14 end
15
16 always
17 begin
18     #20;
19     Clk=~Clk;
20 end
21 endmodule
```

Observation:

Output Waveforms:



Result:

D flip flop was simulated using Xilinx simulator and its functionality has been verified.

Simulation of counters (Up counter, Down counter)

Aim:

To write a Verilog HDL program to simulate different types of counters and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A **counter** is a sequential circuit that goes through a predetermined sequence of states upon receiving clock pulses. Counters are broadly classified into two categories based on their counting behaviour:

1. **Up Counter**: Counts incrementally (e.g., 0 → 1 → 2 → 3).
2. **Down Counter**: Counts decrementally (e.g., 3 → 2 → 1 → 0).

Up Counter (Asynchronous)

An **up counter** increments the count with each clock pulse, transitioning through binary states in ascending order.

Working Principle:

1. The LSB flip-flop toggles its state with every incoming clock pulse.
2. Each subsequent flip-flop toggles its state when the preceding flip-flop transitions from 1 to 0 (falling edge of the output).
3. The counter's states follow a binary counting sequence, starting from 0 and increasing until the maximum value is reached, after which it resets to 0.

State Transition:

For a 3-bit asynchronous up counter:

- States: 000 → 001 → 010 → 011 → 100 → 101 → 110 → 111 → 000.

Circuit Configuration:

- Each flip-flop is connected in series.
- The clock input is applied only to the first flip-flop.
- The output of each flip-flop serves as the clock input for the next flip-flop.

Down Counter (Asynchronous)

A **down counter** decrements the count with each clock pulse, transitioning through binary states in descending order.

Working Principle:

1. The LSB flip-flop toggles its state with every clock pulse.
2. Each subsequent flip-flop toggles its state when the preceding flip-flop transitions from 0 to 1 (rising edge of the output).
3. The counter's states follow a binary counting sequence in reverse, starting from the maximum value and decreasing until 0, after which it wraps back to the maximum value.

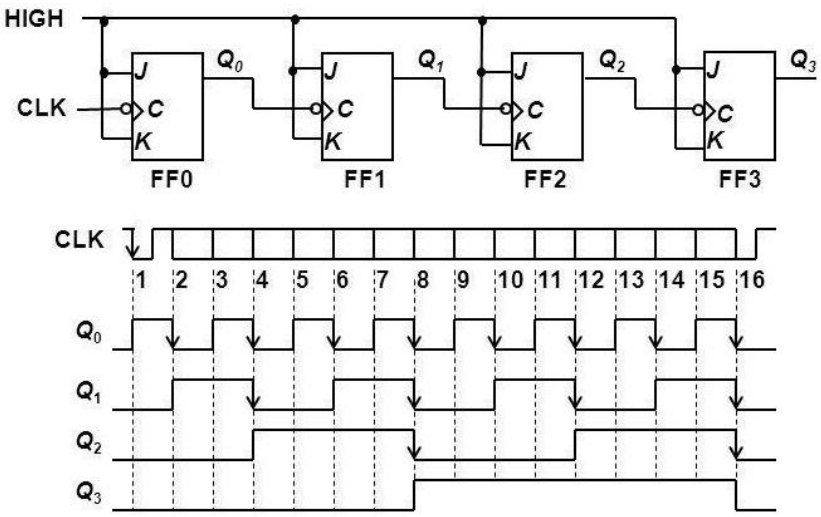
State Transition:

For a 3-bit asynchronous down counter:

- States: 111 → 110 → 101 → 100 → 011 → 010 → 001 → 000 → 111.

Circuit Configuration:

- Similar to the up counter, but the clock triggering logic is reversed.
- Flip-flops are configured to decrement the count instead of incrementing.



Code:

Up Counter

```
`timescale 1ns / 1ps
module upcounter(count, clk, rst);
output [2:0]count;
input clk, rst;
reg [2:0]qn;

always @(posedge clk or rst) begin
    if (rst) begin
        qn[0]<=0;
    end else begin
        qn[0]=~qn[0];
    end
end

always @(negedge qn[0] or rst) begin
    if (rst) begin
        qn[1]<=0;
    end else begin
        qn[1]<=~qn[1];
    end
end

always @(negedge qn[1] or rst) begin
    if (rst) begin
        qn[2]<=0;
    end else begin
        qn[2]<=~qn[2];
    end
end

assign count=qn;
endmodule
```

Down Counter

```
`timescale 1ns / 1ps
module downcounter(count, clk, rst);
output [2:0]count;
input clk, rst;
reg [2:0]qn;

always @(posedge clk or rst) begin
    if (rst) begin
        qn[0]<=0;
    end else begin
        qn[0]=~qn[0];
    end
end

always @(posedge qn[0] or rst) begin
    if (rst) begin
        qn[1]<=0;
    end else begin
        qn[1]<=~qn[1];
    end
end

always @(posedge qn[1] or rst) begin
    if (rst) begin
        qn[2]<=0;
    end else begin
        qn[2]<=~qn[2];
    end
end

assign count=qn;
endmodule
```

Testbench:

Up Counter

```
`timescale 1ns / 1ps
module upcounter_tb;
reg Clk, Rst;
wire [2:0]Count;

upcounter uut (.clk(Clk), .rst(Rst), .count(Count));
initial
begin
    Clk=1;
    Rst=1;
    #10;
    Rst=0;
end

always
begin
    #20;
    Clk=~Clk;
end
endmodule
```

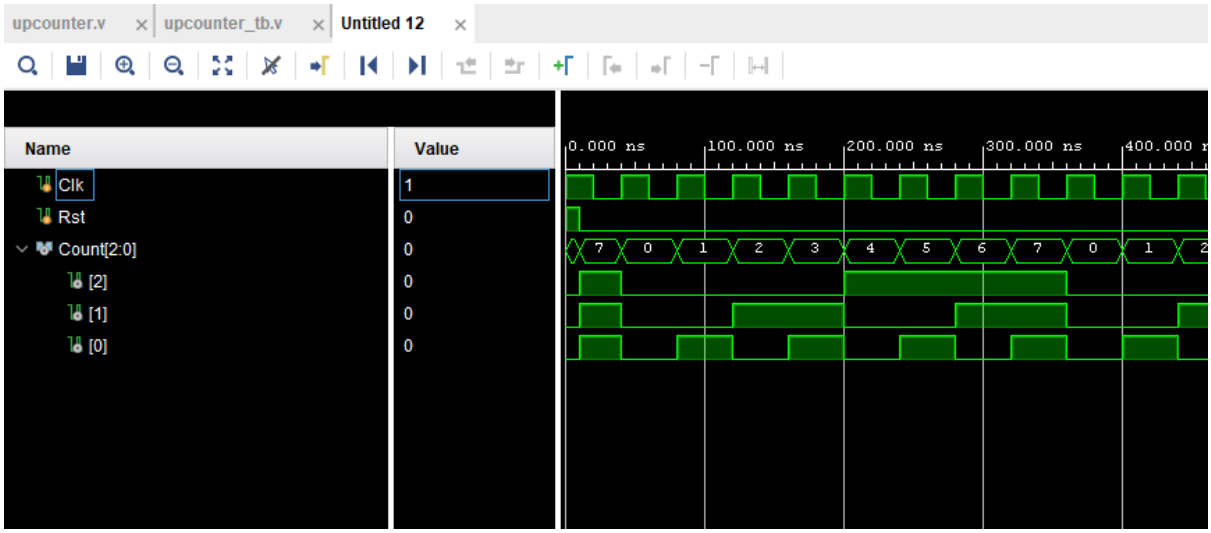
Down Counter

```
`timescale 1ns / 1ps
module downncounter_tb;
reg Clk, Rst;
wire [2:0]Count;

downcounter uut (.clk(Clk), .rst(Rst), .count(Count));
initial
begin
    Clk=1;
    Rst=1;
    #10;
    Rst=0;
end

always
begin
    #20;
    Clk=~Clk;
end
endmodule
```

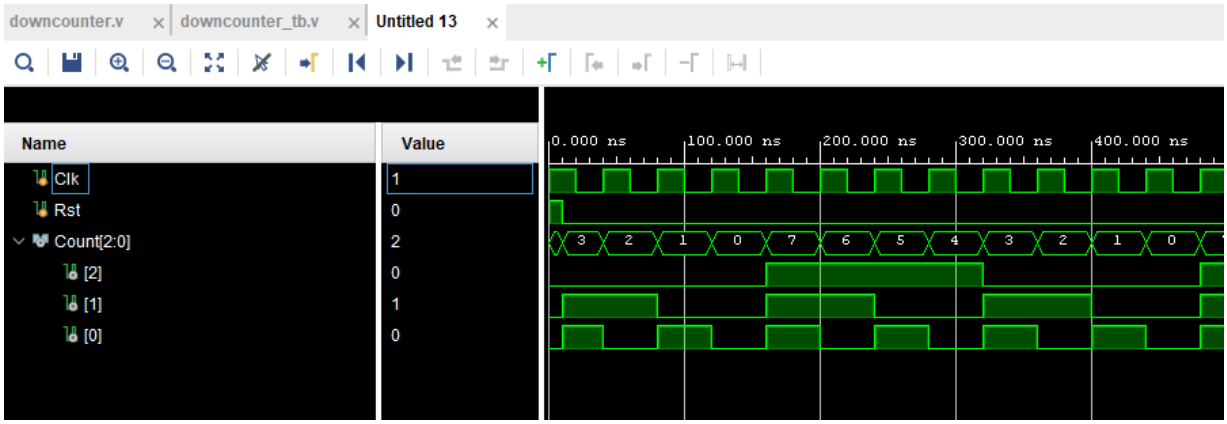
Observation:

Output Waveforms:

Up Counter



Down Counter:



Result:

Up counter and down counter were simulated using Xilinx simulator and its functionality has been verified.

Simulation of Decade Counter

Aim:

To write a Verilog HDL program to simulate decade counter and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A decade counter is a type of counter that counts from 0 to 9 (ten states in total, hence the name "decade"). After reaching 9 (binary 1001), it resets back to 0 on the next clock pulse. Decade counters are commonly used in applications that require counting in decimal form, such as digital clocks, frequency dividers, and event counters.

Decade counters can be implemented as either synchronous or asynchronous counters. Here, we will focus on the synchronous decade counter.

In a synchronous counter, all flip-flops are clocked simultaneously by the same clock signal. This ensures that the state transitions occur without the ripple effect seen in asynchronous counters.

1. A decade counter counts from 0000 (0) to 1001 (9) in binary.
2. When the count reaches 1010 (binary 10 or decimal 10), it resets back to 0000 on the next clock pulse.
3. This reset mechanism is implemented using logic gates that detect the 1010 state and force the counter back to 0000.

Designing a 4-bit Synchronous Decade Counter

To design a synchronous decade counter using JK flip-flops or T flip-flops, follow these steps:

Step 1: Determine the Required Flip-Flops

- A decade counter needs to count from 0 to 9, which requires 4 bits (since 2^4 = 16 possible states).
- Use 4 flip-flops labelled as FF_3, FF_2, FF_1, and FF_0 (MSB to LSB).

Step 2: Create the Truth Table

Count	Present State				Next State			
	Q_3	**Q_2**	**Q_1**	**Q_0**	**Q_3***	**Q_2***	**Q_1***	**Q_0***
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	0	0	0	0

Step 3: Determine the Flip-Flop Inputs

For JK flip-flops, the inputs J and K determine the state transition:

Count	FF – 3 ($Q_3 \rightarrow Q_3$*)		FF – 2 ($Q_2 \rightarrow Q_2$*)		FF – 1 ($Q_1 \rightarrow Q_1$*)		FF – 0 ($Q_0 \rightarrow Q_0$*)	
	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	X	0	X	0	X	1	X
1	0	X	0	X	1	X	X	1
2	0	X	0	X	X	0	1	X
3	0	X	1	X	X	1	X	1
4	0	X	X	0	0	X	1	X
5	0	X	X	0	1	X	X	1
6	0	X	X	0	X	0	1	X
7	1	X	X	1	X	1	X	1
8	X	0	0	X	0	X	1	X
9	X	1	0	X	0	X	X	1

Step 4: Implement the Reset Logic

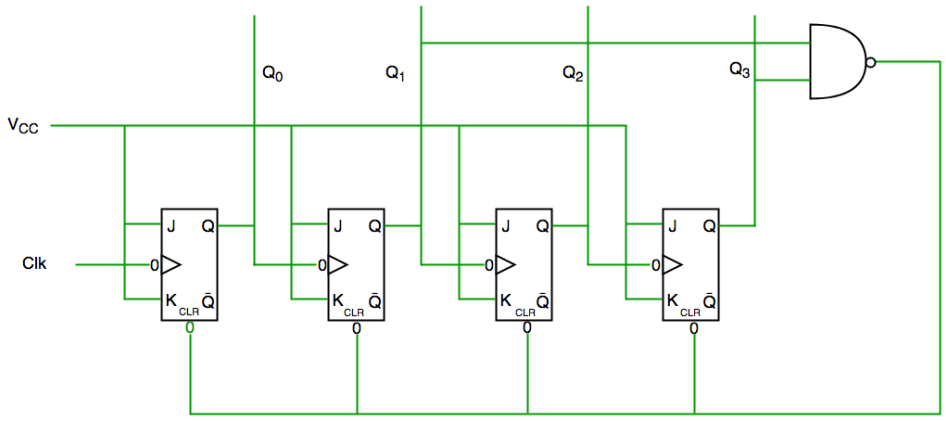
To reset when the count reaches 1010:

1. Use an AND gate to detect when Q3 = 1 and Q1 = 1.
2. Connect the AND gate output to the clear (reset) input of all flip-flops.

Reset Condition: Reset = Q3 · Q1

Logic Diagram

1. Use four JK flip-flops connected in a synchronous manner.
2. Connect the AND gate to reset the counter when the state reaches 1010.



Code:

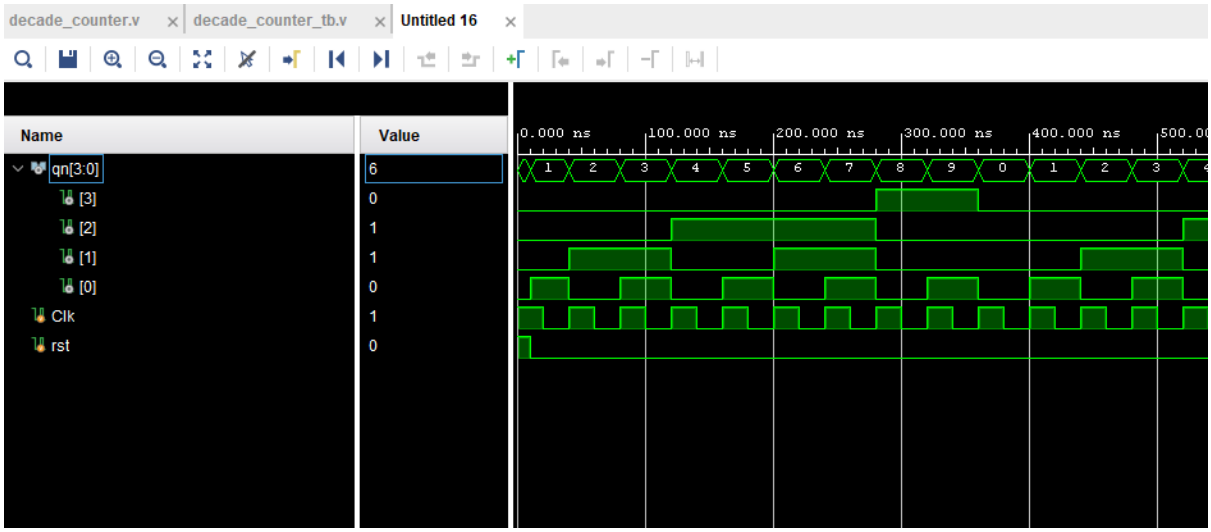
```
`timescale 1ns / 1ps
module decade_counter(clk, reset, q);
input wire clk, reset;
output reg [3:0]q;

always @(posedge clk or reset)
begin
    if  (reset) begin
        q<=4'b0000;
    end
    else begin
        if (q == 4'b1001)   q <= 4'b0000;
        else    q<= q+1;
    end
end
endmodule
```

Testbench:

```
`timescale 1ns / 1ps
module decade_counter_tb;
wire [3:0]qn;
reg Clk, rst;

decade_counter uut (.q(qn), .clk(Clk), .reset(rst));
initial begin
rst=1;
Clk=1;
#10;
rst=0;
end

always
begin
    #20;
    Clk = ~Clk;
end
endmodule
```

Observations:

Output Waveform



Result:

Decade counter was simulated using Xilinx simulator and its functionality has been verified.

Simulation of Registers

Aim:

To write a Verilog HDL program to simulate decade counter and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A **register** is a group of flip-flops used to store and transfer data. Each flip-flop stores a single bit, so a register with n flip-flops can store n bits of data. Registers play a critical role in digital systems, acting as temporary storage elements and enabling data manipulation, transfer, and synchronization.

Registers are primarily categorized based on how data is loaded into and shifted out of them. The shifting direction and data flow (serial or parallel) distinguish different types of registers.

Types of Registers

1. Parallel-In Parallel-Out (PIPO) Register

- **Operation**: Data is loaded into the register simultaneously (parallel load) and all bits are output simultaneously (parallel output).
- **Use Case**: Temporary storage or buffering of data.

Working:

- All bits are written into the flip-flops simultaneously using parallel data input lines.
- All bits are read simultaneously via parallel output lines.

Example: Storing 4-bit binary data in one clock cycle.

2. Parallel-In Serial-Out (PISO) Register

- **Operation**: Data is loaded into the register in parallel but is shifted out one bit at a time serially.
- **Use Case**: Converting parallel data into serial form for transmission.

Working:

1. Data is loaded into the register in parallel.

2. On each clock pulse, one bit is shifted out serially, starting from the MSB or LSB depending on the design.

Example: Sending 8-bit data serially over a single wire.

3. Serial-In Parallel-Out (SIPO) Register

- **Operation**: Data is loaded serially, one bit at a time, and is output in parallel.
- **Use Case**: Converting serial data into parallel form for processing.

Working:

1. Bits are shifted into the register serially on each clock pulse.
2. Once all bits are loaded, they are available simultaneously at the parallel output.

Example: Receiving data from a serial communication line and converting it to parallel form for further use.

4. Serial-In Serial-Out (SISO) Register

- **Operation**: Data is loaded serially and is also shifted out serially, one bit at a time.
- **Use Case**: Temporary storage and transfer of data in serial form.

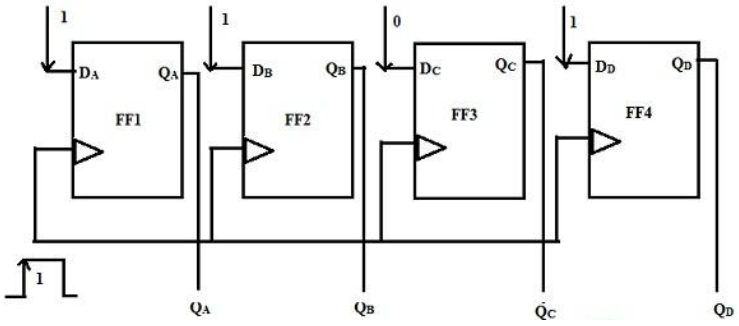
Working:

1. Each bit is shifted into the register on a clock pulse.
2. The stored bits are shifted out serially on subsequent clock pulses.

Example: Transferring data within serial communication systems.

Type	Input Mode	Output Mode	Purpose
PIPO	Parallel	Parallel	Temporary storage, high-speed buffering
PISO	Parallel	Serial	Parallel-to-serial data conversion
SIPO	Serial	Parallel	Serial-to-parallel data conversion
SISO	Serial	Serial	Serial data shifting/storage

PIPO (Parallel In Parallel Out)



Code:

reg_pipo.v × reg_pipo_tb.v ×

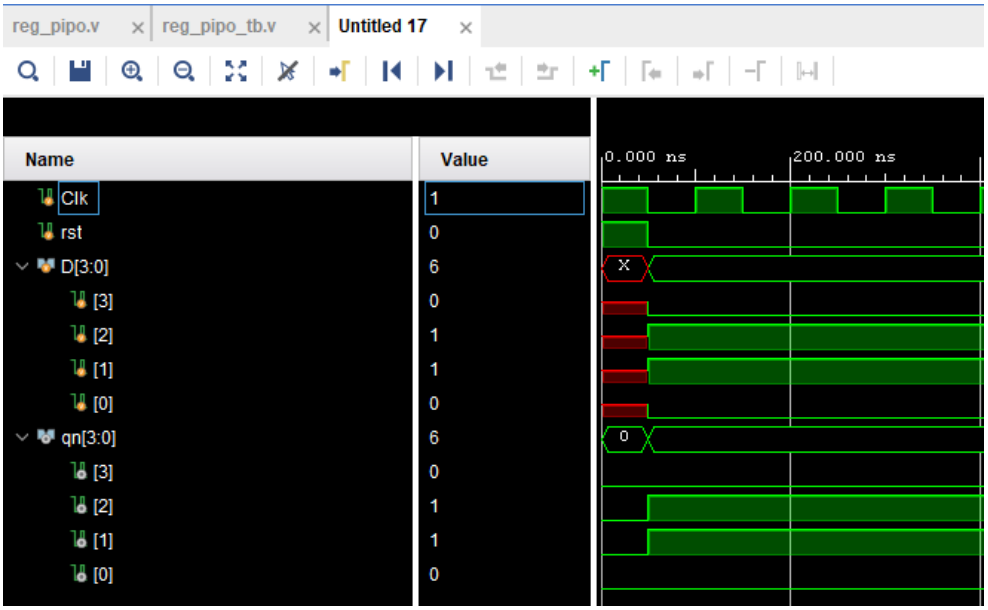
C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/reg_pipo.v

```
1  `timescale 1ns / 1ps
2  module reg_pipo(clk, reset, q, d);
3  input wire clk, reset;
4  input wire [3:0]d;
5  output reg [3:0]q;
6
7  always @(posedge clk or reset)
8  begin
9      if (reset)  q<=4'b0000;
10     else    q <= d;
11 end
12 endmodule
```

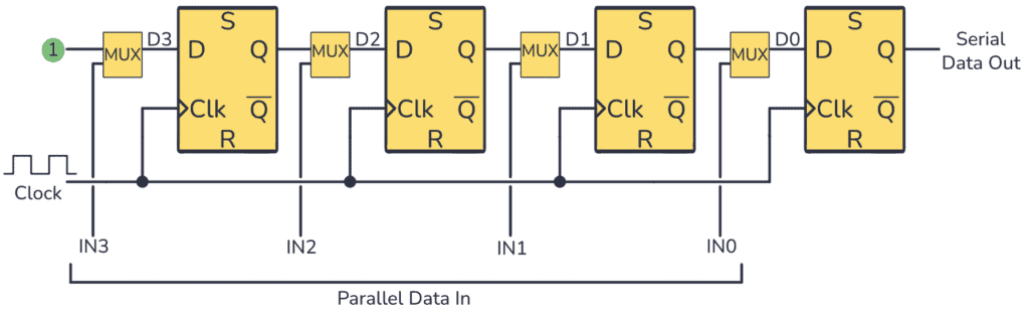
Testbench:

```
1  `timescale 1ns / 1ps
2  module reg_pipo_tb;
3  reg Clk, rst;
4  reg [3:0]D;
5  wire [3:0]qn;
6
7  reg_pipo uut (.clk(Clk), .reset(rst), .d(D), .q(qn));
8  initial begin
9      Clk=1;
10     rst=1;
11     #50;
12     rst=0;
13     D=4'b0110;
14 end
15
16 always
17 begin
18     #50;
19     Clk = ~Clk;
20 end
21 endmodule
```

Output waveform:



PISO (Parallel In Serial Out)



Code:

reg_piso.v × reg_piso_tb.v ×

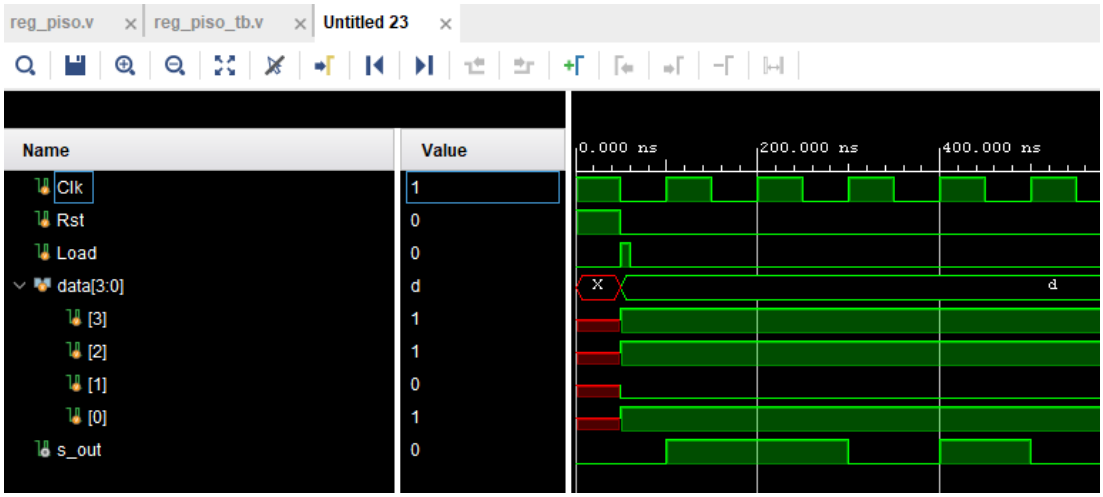
C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/reg_piso.v

```
1  `timescale 1ns / 1ps
2  module reg_piso(clk, rst, load, d_in, s_out);
3  input wire clk, rst, load;
4  input wire [3:0] d_in;
5  output reg s_out;
6  reg [3:0] shift_reg;
7
8  always @(posedge clk or rst) begin
9      if (rst) begin
10         shift_reg <= 4'b0000;
11         s_out <= 1'b0;
12     end else if (load) begin
13        shift_reg <= d_in;
14     end else begin
15         s_out <= shift_reg[3];
16         shift_reg <= shift_reg << 1;
17     end
18 end
19 endmodule
20
```

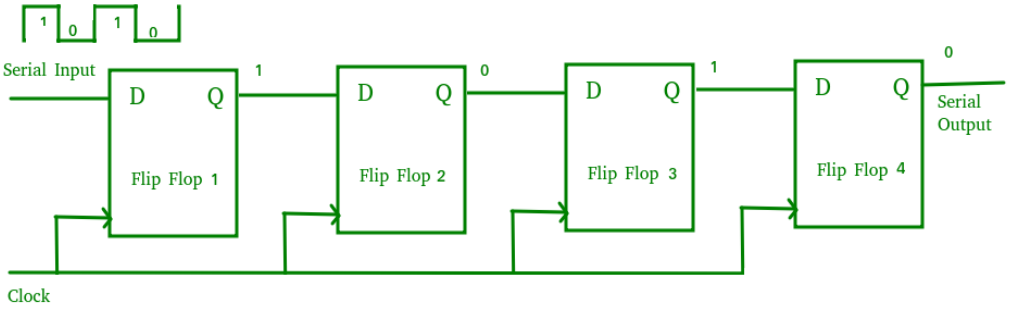
Testbench:

```
1  `timescale 1ns / 1ps
2  module reg_piso_tb;
3  reg Clk, Rst, Load;
4  reg [3:0]data;
5  wire s_out;
6
7  reg_piso uut(.clk(Clk), .rst(Rst), .load(Load), .d_in(data), .s_out(s_out));
8  initial begin
9  Clk = 1;
10 Load = 0;
11 Rst = 1;
12 #50;
13 Rst = 0;
14 data = 4'b1101;
15 Load = 1;
16 #10;
17 Load = 0;
18 end
19
20 always
21 begin
22 #50;
23 Clk = ~Clk;
24 end
25 endmodule
```

Output Waveform:



SIPO (Serial In Parallel Out)



Code:

reg_sipo.v | reg_sipo_tb.v

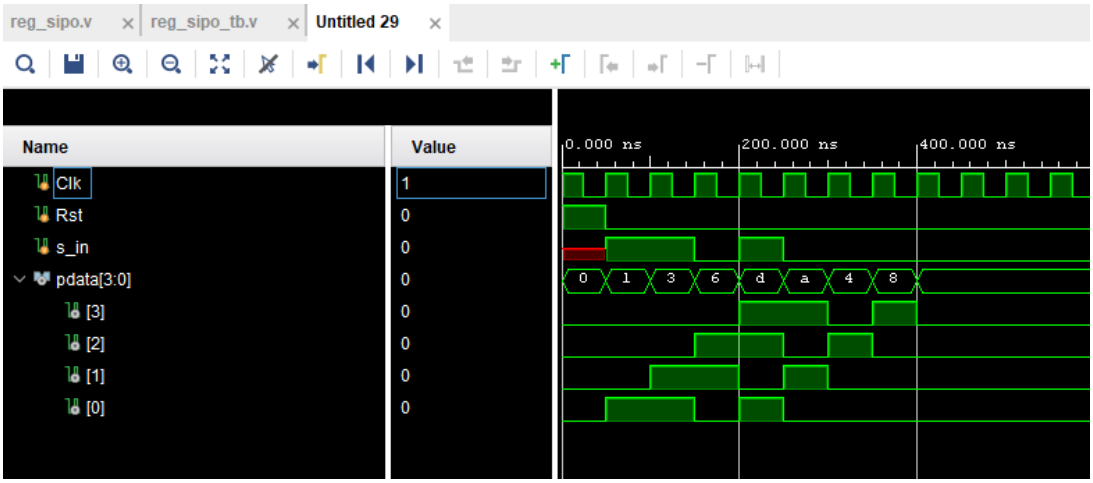
C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/reg_sipo.v

```
1  `timescale 1ns / 1ps
2  module reg_sipo(clk, rst, sin, pdata);
3
4  input wire clk, rst;
5  input wire sin;
6  output reg [3:0] pdata;
7
8  always @(posedge clk or rst) begin
9      if (rst) begin
10         pdata <= 4'b0000;
11     end else begin
12         pdata <= {pdata[2:0], sin};
13     end
14 end
15 endmodule
```

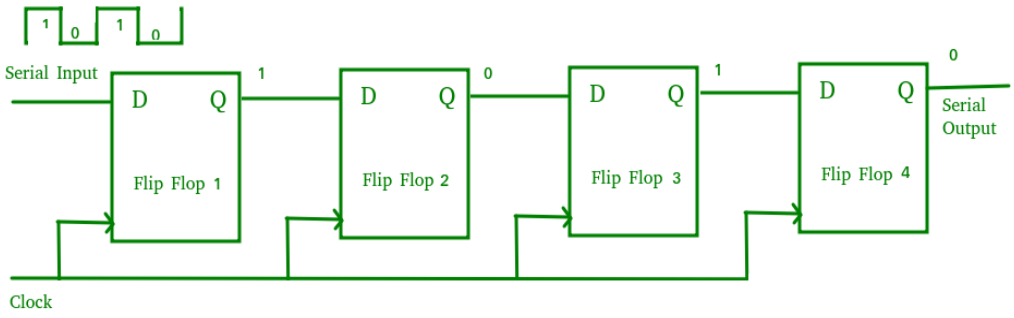
Testbench:

```
`timescale 1ns / 1ps
module reg_sipo_tb;
reg Clk, Rst;
reg s_in;
wire [3:0] pdata;

reg_sipo uut (.clk(Clk), .rst(Rst), .sin(s_in), .pdata(pdata));
initial begin
Clk=1;
Rst=1;
#50;
Rst=0;
s_in = 1'b1;
#50;
s_in = 1'b1;
#50;
s_in = 1'b0;
#50;
s_in = 1'b1;
#50;
s_in = 1'b0;
end

always begin
#25;
Clk = ~Clk;
end
endmodule
```

Output Waveform:



SISO (Serial In Serial Out)



Code:

reg_siso.v × reg_siso_tb.v ×

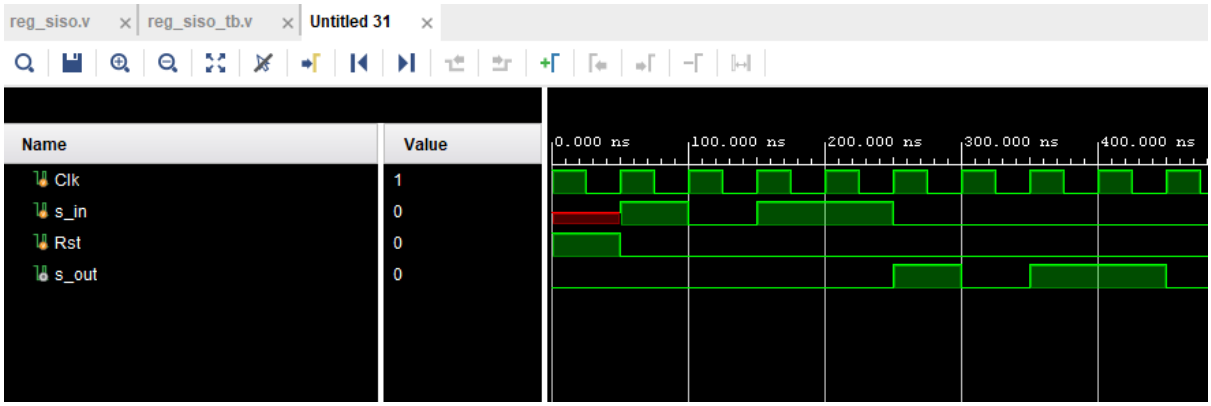
C:/Users/Chirag Gupta/lab_mannual/lab_mannual.srcs/sources_1/new/reg_siso.v

```
1  `timescale 1ns / 1ps
2  module reg_siso(clk, rst, sin, sout);
3  input wire clk, sin, rst;
4  output reg sout;
5  reg [3:0]shift_reg;
6
7  always @(posedge clk or rst) begin
8      if (rst) begin
9          shift_reg <= 4'b0000;
10         sout <= 1'b0;
11     end else begin
12         sout <= shift_reg[3];
13         shift_reg <= {shift_reg[2:0], sin};
14     end
15 end
16 endmodule
```

Testbench:

```
1  `timescale 1ns / 1ps
2  module reg_siso_tb;
3  reg Clk, s_in, Rst;
4  wire s_out;
5
6  reg_siso uut (.clk(Clk), .rst(Rst), .sin(s_in), .sout(s_out));
7  initial
8  begin
9  Clk=1;
10 Rst=1;
11 #50;
12 Rst=0;
13 s_in = 1'b1;
14 #50;
15 s_in = 1'b0;
16 #50;
17 s_in = 1'b1;
18 #50;
19 s_in = 1'b1;
20 #50;
21 s_in = 1'b0;
22 end
23
24 always begin
25 #25;
26 Clk = ~Clk;
27 end
28 endmodule
29
```

Output Waveforms:



Result:

Different types of registers were simulated using Xilinx simulator and its functionality has been verified.

Simulation of Basic 4-bit ALU

Aim:

To write a Verilog HDL program to simulate a basic 4-bit ALU and verify the outputs using simulator.

Apparatus:

Xilinx Vivado
Personal Computer

Theory:

A 4-bit Arithmetic Logic Unit (ALU) is a combinational digital circuit designed to perform arithmetic and logic operations on two 4-bit binary inputs. This ALU uses Verilog's built-in operators (+, -, &, |, ^, ~, <<, >>) for simplicity and synthesis efficiency.

Inputs:

- A[3:0]: 4-bit input operand A
- B[3:0]: 4-bit input operand B
- opcode[2:0]: 3-bit selector to choose the operation

Outputs:

- res[3:0]: 4-bit result output
- Zero: High if the result is 0000
- CarryOut: High if addition generates carry out (optional)

Supported Operations:

Opcode	Operation	Description
000	Y = A + B	4-bit addition
001	Y = A - B	4-bit subtraction
010	Y = A & B	Bitwise AND
011	Y = A | B	Bitwise OR
100	Y = A ^ B	Bitwise XOR
101	Y = ~A	Bitwise NOT of A
110	Y = A << 1	Logical left shift of A
111	Y = A >> 1	Logical right shift of A

Implementation Notes

- Arithmetic and logic operations are implemented using Verilog operators for ease and clarity.
- The Zero flag checks if output Y is 0000.
- CarryOut can be optionally derived by checking the MSB in case of addition.

This 4-bit ALU provides a compact and functional implementation of key arithmetic and logic functions. It serves as a foundational module for larger processor designs and demonstrates the use of synthesizable Verilog constructs to implement hardware functionality efficiently.

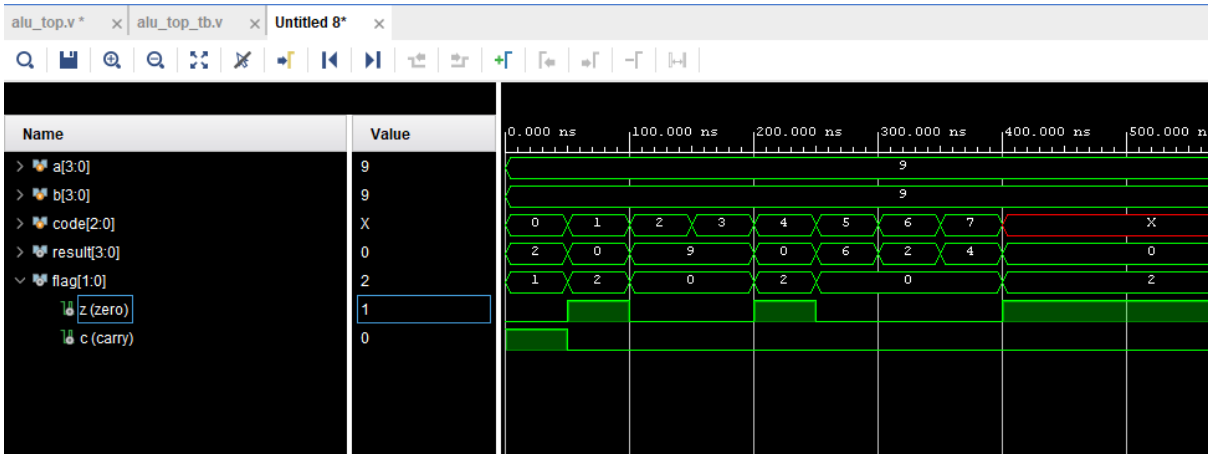
Code:

```
`timescale 1ns / 1ps
module alu_top(A,B,opcode,res,flag);

    input wire [3:0] A, B;
    input wire [2:0] opcode;
    output reg [3:0] res;
    output reg [1:0]flag;
    reg [4:0]add;
    reg c,z;

    always @(*)
    begin
        c=1'b0;
        flag = 2'b00;
        case (opcode)
            3'b000:
                begin
                    add = A + B;
                    res = add[3:0];
                    c = add[4];
                end
            3'b001: res = A - B;
            3'b010: res = A & B;
            3'b011: res = A | B;
            3'b100: res = A ^ B;
            3'b101: res = ~ A;
            3'b110: res = A << 1;
            3'b111: res = A >> 1;
            default: res = 4'b0000;
        endcase

        z = (res == 4'b0000)? 1'b1 : 1'b0;
        flag = {z, c};
    end
endmodule
```

Testbench:

```
`timescale 1ns / 1ps
module alu_top_tb;
    reg [3:0]a,b;
    reg [2:0]code;
    wire [3:0]result;
    wire [1:0]flag;

    alu_top uut (.A(a), .B (b), .opcode(code), .res(result), .flag(flag));

    initial
    begin
        $monitor ("Opcode = %b, A = (%b %d), B = (%b %d), Result = (%b %d)", code, a, a, b, b, result, result);
        a = 4'b1001;
        b = 4'b1001;
        code = 3'b000;
        #50 code = 3'b001;
        #50 code = 3'b010;
        #50 code = 3'b011;
        #50 code = 3'b100;
        #50 code = 3'b101;
        #50 code = 3'b110;
        #50 code = 3'b111;
        #50 code = 3'bx;
    end
endmodule

```

Output Waveform:



Result:

4-bit basic ALU has been simulated with the help of Xilinx Simulator and its functionality has been verified.