DIGITAL DESIGN USING VERILOG HDL-RTL
Coding and Synthesis

COURSE REPORT SUBMITTED TO

SURE TRUST

By

ARTI TYAGI

UNDER THE GUIDANCE OF

MR. NAGA SITARAM M.

SKILL UPGRADATION FOR RURAL -YOUTH EMPOWERMENT(SURE-TRUST)
Sreeguru Tower Second Floor Gopuram Road, Opp. Union bank of India, Puttaparthi. 515134

DECLARATION

I am Arti Tyagi hereby declare that the work which is being presented in this course report "DIGITAL DESIGN USING VERILOG HDL-RTL Coding and Synthesis" by me,in partial fulfillment of the requirements for the award of "Advance VLSI Design and Verification" Trainee at
SURE TRUST.

(Arti Tyagi)

CONTENTS

COMBINATIONAL LOGIC DESIGN - I

ARITHMATIC CIRCUITS

Arithmetic operations such as addition and subtraction has an important role in the efficient design of processor logic. Arithmetic logic unit(ALU) of any processor can be designed to perform the addition,subtraction,increment,decrements operations. These circuits can be operated with binary values 0 and 1.

Binary Adder

The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.

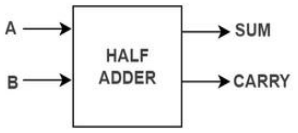
1.1 Adder

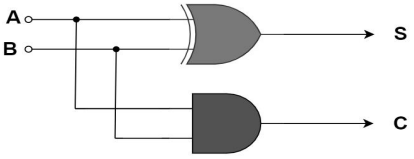
Adder are use to perform the binary addition of two binary numbers.Adders are used for signed or unsigned addition operations.

1.1.1 HALF ADDER :

Half adder is a combinational circuit, which performs the addition of two **bits A and B are of single bit numbers. It produces two outputs sum & carry as** Outputs.

BLOCK DIAGRAM :





TRUTH TABLE :

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

BOOLEAN EXPRESSIONS :

SUM = A ^ B = AB`+A`B

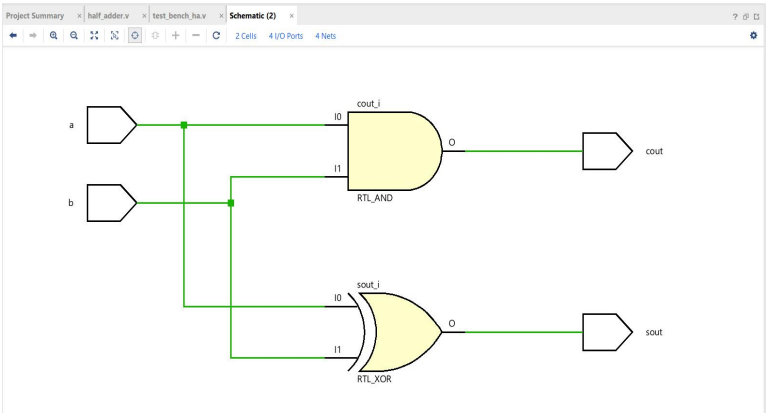
CARRY = A & B = AB

VERILOG CODE AND TESTBENCH CODE:

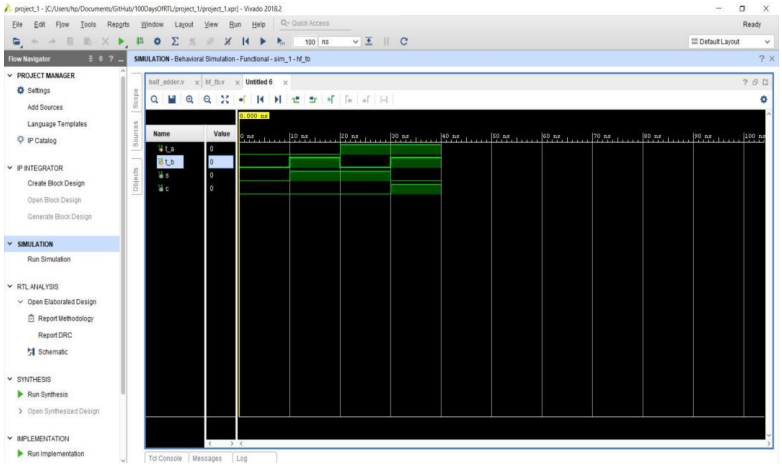
```
module ha_bhv(a,b,sum,carry);
input a,b;
output reg sum,carry;

      always@(a or b)begin
        if(a==0 && b ==0) begin
                sum = 0;
                carry = 0;
              end

       else if(a==1 && b==1) begin
                sum = 0;
               carry = 1;
                  end

      else begin
a==1 && b==0) or (a==0 && b==1)
                sum = 1;
               carry = 0;
            end
    end
endmodule
```

```
module hf_tb();
reg t_a,t_b;
wire s,c;
half_adder
dut(.a(t_a), .b(t_b), .s(s), .c(c));

initial begin
t_a=0; t_b=0;
#10
t_a=0;t_b=1;
#10
t_a=1;t_b=0;
#10
t_a=1;t_b=1;
#10
$stop;
end
endmodule
```

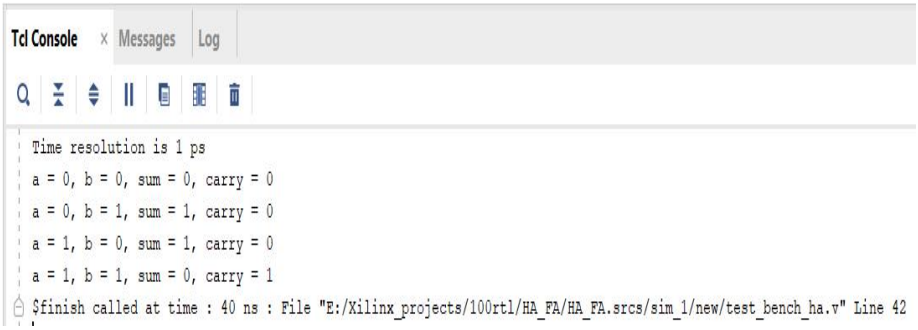
RTL IMPLEMENTATION :



SIMULATION RESULTS:



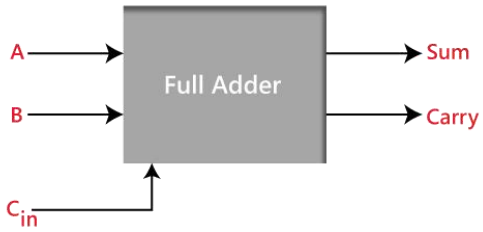
TCL CONSOLE :



1.2 FULL ADDER

A full adder is a combinational logic circuit that forms the arithmetic sum of three bits. it consists of three inputs and two outputs. which performs the addition of three **bits A ,B and Carrry and It produces sum & carry as** Outputs.
Half adder have no scope of adding the carry bit ,to overcome this drawback,full adder comes into play.

BLOCK DIAGRAM :



TRUTH TABLE FOR FULL ADDER:

Inputs			Outputs	
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

BOOLEAN EXPRESSION:

SUM = A ^ B ^ C
CARRY = (A & B) | (B & C) | (A & C)

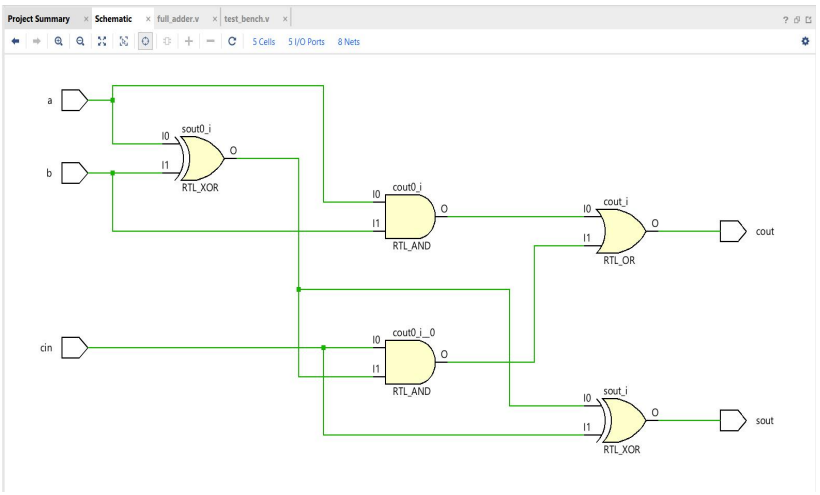
VERILOG RTL CODE AND TESTBENCH CODE FOR FULL ADDER :

```
module
FA_gate(a,b,c,sout,cout);
input a,b,c;
output sout,cout;
wire w1,c1,c2,c3,out1;

xor x1(w1,a,b);
xor x2(sout,a,b);

and a1(c1,a,b);
and a2(c2,b,c);
and a3(c3,a,c);

or  o1(out1,c1,c2);
or  o2(cout,out1,c3);

endmodule
```

```
module full_adder_data(
   input a, b, cin,
   output sout, cout
   );

   assign sout = a ^ b ^ cin;
   assign cout = (a&b) | cin & (a^b);
endmodule
```

```
module Tb_g;
reg a,b,c;
wire sout,cout;

FA_gate FA(a,b,c,sout,cout);

initial begin
a=1'b0;
b=1'b0;
c=1'b0;
#10;

a=1'b0;
b=1'b0;
c=1'b1;
#10;

a=1'b0;
b=1'b1;
c=1'b0;
#10;

a=1'b0;
b=1'b1;
c=1'b1;
#10;

a=1'b1;
b=1'b0;
c=1'b0;
#10;

a=1'b1;
b=1'b0;
c=1'b1;
#10;

a=1'b1;
b=1'b1;
c=1'b0;
#10;

a=1'b1;
b=1'b01;
c=1'b1;
#10;

$finish;
end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION RESULT :

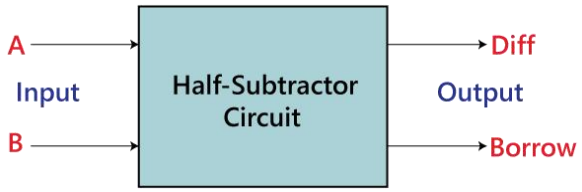
1.2 SUBTRACTOR

Subtractors are used to perform the binary subtraction of two binary numbers.this section describes about
the half and full subtractors.

A 1.2.1 HALF SUBTRACTOR

half subtractor is a digital logic circuit that performs binary subtraction of two single-
bit binary numbers. It has two inputs, A and B, and two outputs, DIFFERENCE and
BORROW. Subtracts B from A & produceds Difference & Borrow as Outputs.

BLOCK DIAGRAM :



TRUTH TABLE :

Inputs		Outputs	
A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

BOOLEAN EXPRESSIONS :

Diff= A'B+AB'
Borrow = A'B

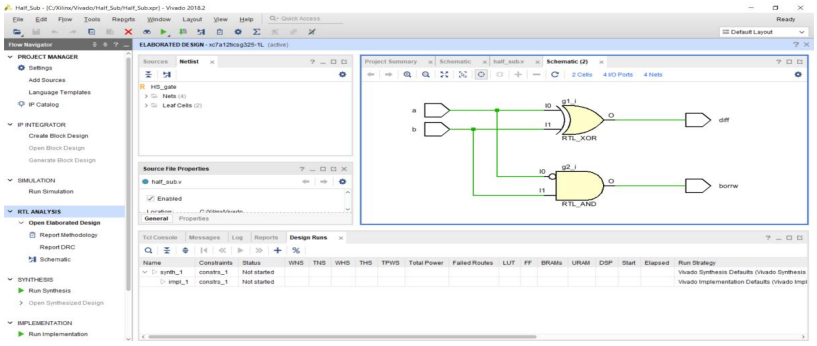
VERILOG RTL CODE AND TESTBENCH CODE FOR FULL ADDER :

```
modul HS_bhav(a,b,diff,borrw);
input a,b;
output diff,borrw;
reg diff,borrw;

always@(a or b)
begin
diff = a^b;
borrw = (~a) & b;
end
endmodule
```

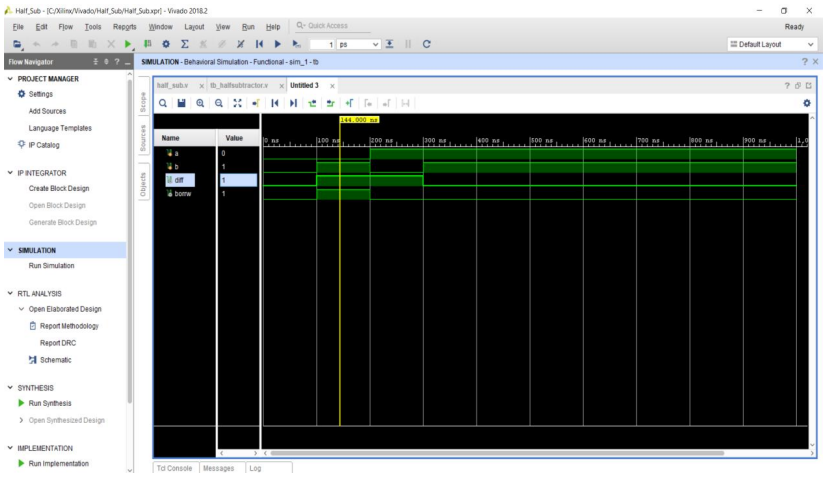
```
module tb;
reg a,b;
wire diff,borrw;

HS_gate
dut(.a(a), .b(b), .diff(diff), .borrw(bo
rrw));
initial begin
a=0; b=0;
#100 a=0; b=1;
#100 a=1; b=0;
#100 a=1; b=1;
$display($time, "a=%b, b=%b,
diff=%b, borrw=%b",
a,b,diff,borrw);
end
endmodule
```

```
module HS_gate(a,b,diff,borrw);
input a,b;
output diff,borrw;
wire x;

xor g1(diff,a,b);
not(x,a);
and g2(borrw,x,b);
endmodule
```

RTL IMPLEMENTATION :



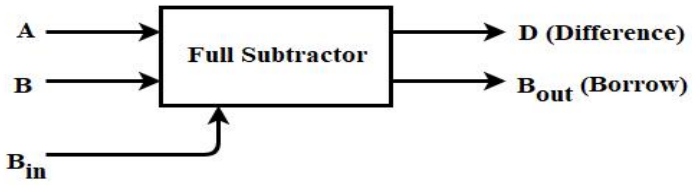
SIMULATION RESULT :



1.1.1 FULL SUBTRACTOR

A full subtractor is a **combinational circuit** that performs subtraction of two bits, A full subtractor has three
inputs A,B and C and two outputs difference and Borrow.

BLOCK DIAGRAM :



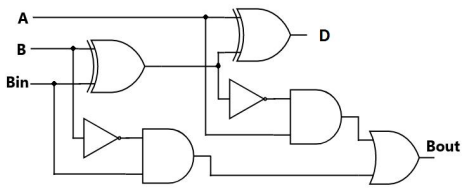
TRUTH TABLE:

INPUT			OUTPUT	
A	B	B_{in}	D (Difference)	B_{out} (Borrow)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

BOOLEAN EXPRESSION :

$$\text{Difference} = A \oplus B \oplus Bin$$
$$Bout = A'\,Bin + A'\,B + B\,Bin$$

CIRCUIT DIAGRAM OF FULL SUBTRACTOR :

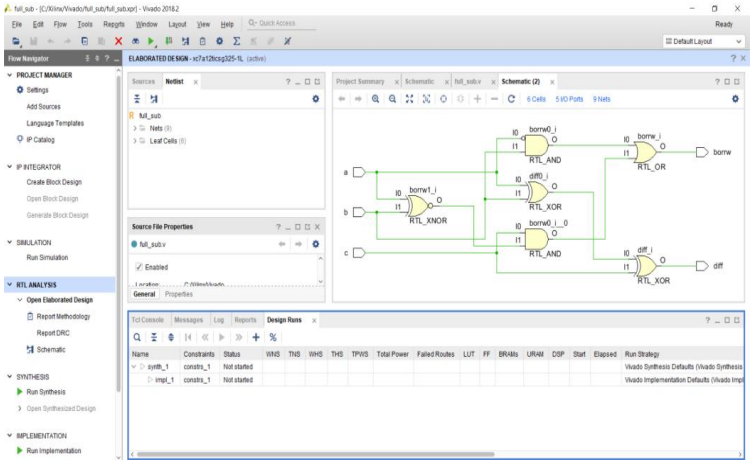


VERILOG RTL CODE AND TESTBENCH CODE FOR FULL SUBTRACTOR :

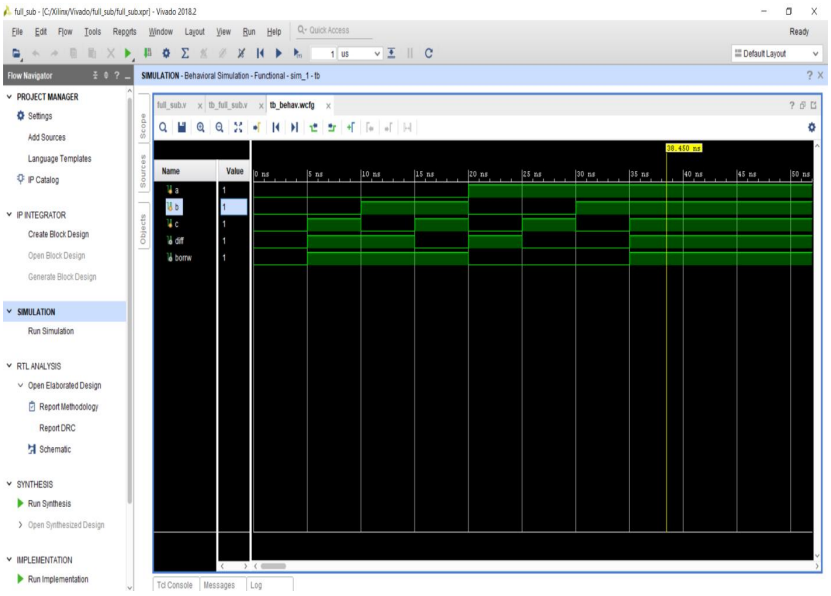
```
Module full_sub(a,b,bin,diff,borrw);
input a,b,bin;
output diff,borrw;

assign diff = a^b^bin;
assign borrw =(~a & b)|(bin&(a~^b));

endmodule
```

```
module testbench;
reg a,b,c;
wire diff,borrw;

full_sub dut(.a(a), .b(b), .c(c), .diff(diff), .borrw(borrw));

initial begin
a=0; b=0; c=0;
#5 a=0; b=0; c=1;
#5 a=0; b=1; c=0;
#5 a=0; b=1; c=1;
#5 a=1; b=0; c=0;
#5 a=1; b=0; c=1;
#5 a=1; b=1; c=0;
#5 a=1; b=1; c=1;
$display($time, "a=%b, b=%b, c=%b, diff=%diff, borrw=%borrw",
a,b,c,diff,borrw);
end
endmodule
```

RTL IMPLEMENTATION:



SIMULATION RESULT :



1.4 MULTI-BIT ADDERS

Multi-bit adders and subtractors are used in the design of arithmatic units for the processors.
The logic density depends upon the number of input bits of adder or subtractor.

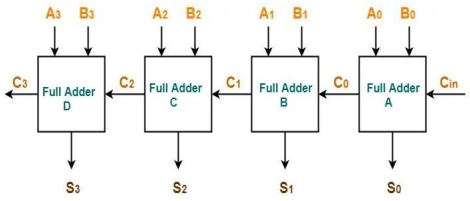
1.4.1 FOUR-BIT FULL ADDER

Many practical use multi-bit adder and subtractors.it is the industrial practical to use basic component as full adder perform the addition operation. for example , if designer needs to implement the four-bit design logic of an adder,then four full adders are required.

RIPPLE CARRY ADDER :

Ripple carry adder is a combinational logic circuit.
Its used for the purpose of adding two n-bit binary numbers.
Its requires n full adders in its circuit for adding two n-bit binary numbers.
Its also known as n-bit parallel adder.

BLOCK DIAGRAM OF RIPPLE CARRY ADDER :



4-bit Ripple Carry Adder

Ripple Carry Adder works in different stages .each full adder takes the carry-in as input and produces carry-out and sum bit as output.

WORKING OF 4-BIT RIPPLE CARRY ADDER :

Lets take an example:

- The two bit numbers are 0101(A3A2A1A0) and 1010(B3B2B1B0).These numbers are to be added using a 4-bit ripple carry adder.
- When Cin is fed as input to the full adder A,it activates the full adder A.Then full adder A is A0 = 1, B0 = 0 , Cin =0.
- Sum and carry will be generated as per the sum and carry equations of this adder the output equation for the Sum = A1 ⊕ B1 ⊕ Cin and Carry = A1B1 ⊕ B1Cin ⊕ CinA
- As per equation, for 1st full adder A 1 = 1 and carry output C1 = 0.
- Same like for next input bits A2 and B2 output S2 =1 and C2 = 0.Here the important points is the second stage full adder gets input carry.C1 which is output carry of initial stage full adder.
- Like this we will get the final output seq (S4 S3 S2 S1) = (1 1 1 1) and output carry C4 = 0.

BOOLEAN EQUATION:

Sum = A1 ⊕ B1 ⊕ Cin and Carry = A1B1 ⊕ B1Cin ⊕ CinA

TRUTH TABLE FOR RIPPLE CARRY ADDER :

A1	A2	A3	A4	B4	B3	B2	B1	S4	S3	S2	S1	carry
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	1
1	0	1	0	1	0	1	0	0	1	0	0	1
1	1	0	0	1	1	0	0	1	0	0	0	1
1	1	1	0	1	1	1	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1

RIPPLE CARRY ADDER APPLICATION :

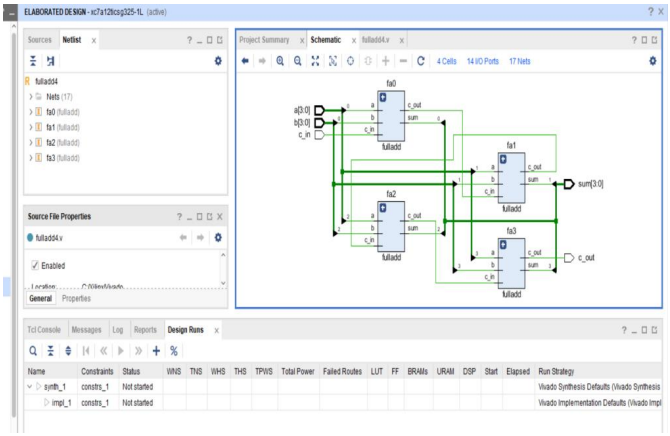
- These carry adders are used mostly in addition to n-bit input sequences.
- These carry adders are applicable in the digital signal processing and microprocessors.

VERILOG RTL CODE AND TESTBENCH CODE FOR RIPPLE CARRY ADDER :

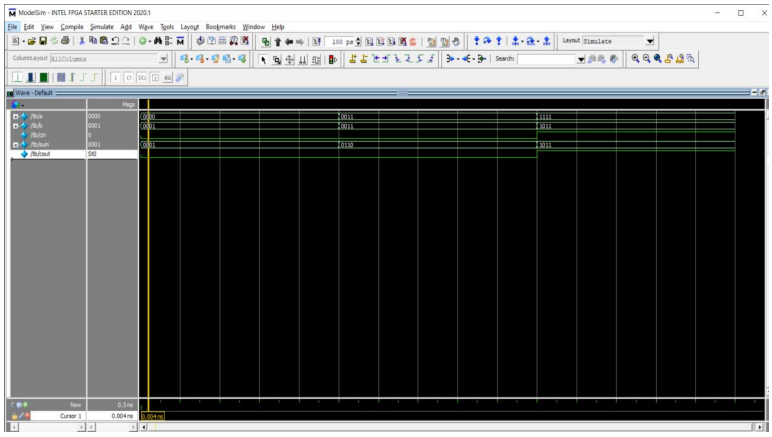
```
//Implementaion of RIPPLE CARRY ADDER by using Structural Modelling..

module FA(a,b,cin,sum,cout);
input a,b,cin;
output sum,cout;
assign sum=a^b^cin;
assign cout=(a^b)&cin|a&b;
endmodule

module Parallel_adder(a,b,cin,sum,cout);
input[3:0]a,b;
input cin;
output[3:0]sum;
output cout;
wire c1,c2,c3;
//Specify  the function of full adder...
FA f0(a[0],b[0],cin,sum[0],c1);
FA f1(a[1],b[1],c1,sum[1],c2);
FA f2(a[2],b[2],c2,sum[2],c3);
FA f3(a[3],b[3],c3,sum[3],cout);
endmodule
```

```
module testbench;

reg [3:0] a,b;
reg cin;
//output value
wire[3:0]sum;
wire cout;

// Instantiate the design Under Test (dUT)
Parallel_adder dut(a,b,cin,sum,cout);
initial begin
a=4'b0000; b=4'b0000; cin=1'b0;
a=4'b0001; b=4'b0010; cin=1'b1;
#10;
a=4'b0001; b=4'b0110; cin=1'b0;
#10;
a=4'b0101; b=4'b0011; cin=1'b0;
#10;
$display("a=%d, b=%d, cin=%d -> sum=%d cout=%b" , a,b,cin,sum,cout);
end
endmodule
```

RTL IMPLEMENTATION OF RIPPLE CARRY ADDER :



SIMULATION RESULT OF RIPPLE CARRY ADDER :



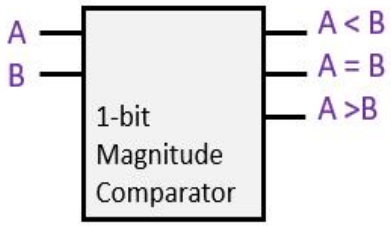
1.5 MAGNITUDE COMPARATORS

In most of the practical scenarios; Magnitude Comparators are used to compare the equality of two binary numbers.

MAGNITUDE COMPARATORS :

The comparison of two binary numbers is an operation that determines whether one number is greater than or lesser than or equal to the other number.A magnitude comparator is a combinational circuit that compares two numbers A and B and determines their relative magnitudes. We will have two inputs one for A and the other for B and have three output terminals one for A > B condition, one for A = B condition and A < B condition.

BLOCK DIAGRAM COMPARATOR :



TRUTH TABLE :

a	b	a>b	a<b	a=b
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

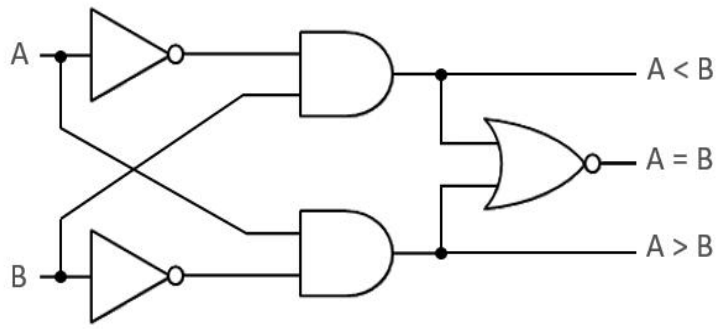
BOOLEAN EXPRESSIONS FOR MAGNITUDE COMPARATOR :

A>B: AB'

A<B: A'B

A=B: A'B' + AB

1-BIT MAGNITUDE COMPARATOR LOGIC GATES DIAGRAM :



VERILOG RTL CODE AND TESTBENCH CODE FOR MAGNITUDE COMPARATOR :

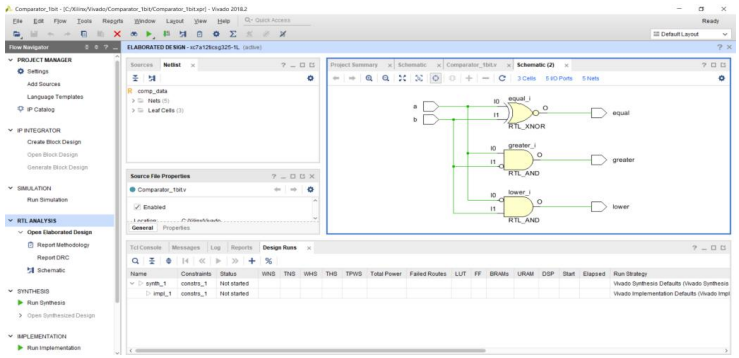
```
//1 bit_comparator using data flow
modeling

module
comp_data(a,b,equal,greater,lower);
output equal;
output greater;
output lower;
input a;
input b;
assign equal = a~^b;
assign lower = (~a)&b;
assign greater = a&(~b);
endmodule
```

```
//1 bit_comparator using Bhaviour
modeling

module
comp_bhv(a,b,equal,greater,lower
);
input a;
input b;
output reg equal;
output reg greater;
output reg lower;

always@(*) begin
equal=0; greater=0; lower=0;
   if(a>b)
            begin
              greater = 1'b1;
            end
            else if(a<b)
            begin
              lower = 1'b1;
            end
            else(a=b);
              equal = 1'b1;
end

endmodule
```

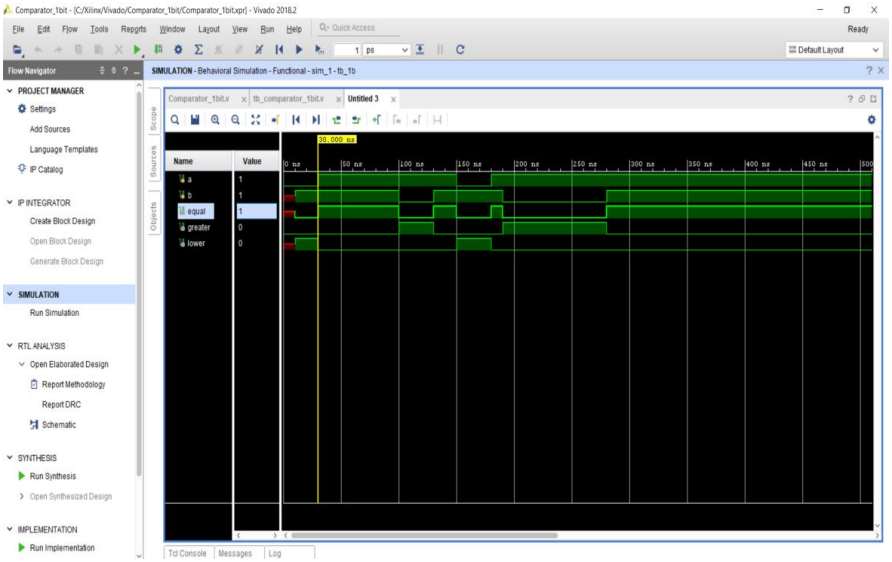
```
//1 bit_comparator using gate modeling

module comp_g(a,b,equal,greater,lower);
output equal;
output greater;
output lower;
input a;
input b;
wire w1,w2;

not g1(w1,a);
not g2(w2,b)
and g3(w1,a,b);
and g4(w2,a,b);
xnor g5(E,w1,w2)
endmodule
```

```
//test_Bench for 1bit comparator

module testbench;
reg a,b;
wire equal,greater,lower;

comp_bhv comp_1(a,b,equal,greater,lower);

initial begin
     repeat(5) begin
               a=$random;
               b=$random;
               #5;
               end
$display("a=%b,b=%b, ___ equal = %b,greater = %b,lower
= %b",a,b,equal,greater,lower);
end
endmodule
```

RTL IMPLEMENTATION :



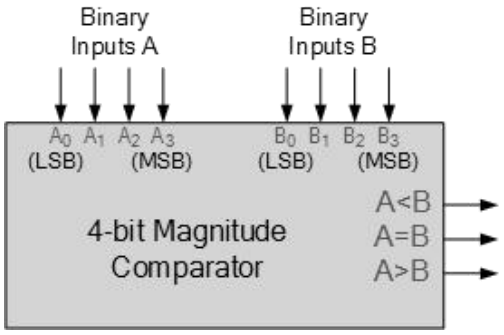
SIMULATION RESULT :



4-BIT MAGNITUDE COMPARATOR :

A comparator used to compare two binary numbers each of four bits is called a 4-bit magnitude comparator. It consists of eight inputs each for two four-bit numbers and three outputs to generate less than, equal to, and greater than between two binary numbers.

BLOCK DIAGRAM FOR 4-BIT MAGNITUDE COMPARATOR :



TRUTH TABLE :

The truth table for a 4-bit comparator would have 4^4 = 256 rows. So we will do Things a bit differently here. We will compare each bit of the two 4-bit numbers,

COMPARING INPUTS				OUTPUT		
A3, B3	**A2, B2**	**A1, B1**	**A0, B0**	**A > B**	**A < B**	**A = B**
A3 > B3	X	X	X	H	L	L
A3 < B3	X	X	X	L	H	L
A3 = B3	A2 >B2	X	X	H	L	L
A3 = B3	A2 < B2	X	X	L	H	L
A3 = B3	A2 = B2	A1 > B1	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 > B0	H	L	L

BOOLEAN EXPRESSION :

A=B :- The equation for the A=B condition was A▬B. Let's call this x. Taking a look at the

truth table above, A=B is true only when (A3=B3 and A2=B2 and A1=B1 and A0=B0). Hence

Z (A=B) = A3B3 . A2B2 . A1.B1 . A0.B0 = x3x2x1x0

A>B

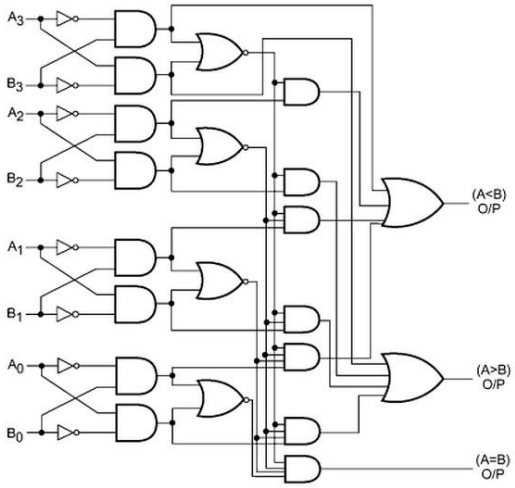
X(A>B) = A3B3′ + x3A2B2′ + x3x2A1B1′ + x3x2x1A0B0′

A<B

Employing the same principles we used above, we get the following equation

Y(A<B) = A3’B3 + X3A2’B2 + X3X2A1’B1 + X3X2X1A0’B0

4-BIT MAGNITUDE COMPARATOR LOGIC GATES DIAGRAM :



APPLICATION OF MAGNITUDE COMPARATOR

- Comparators are used in central processing units (CPUs) and microcontrollers (MCUs).
- These are used in control applications in which the binary numbers representing physical variables such as temperature, position, etc. are compared with a reference value.
- Comparators are also used as process controllers and for Servo motor control.
- Used in password verification and biometric applications.

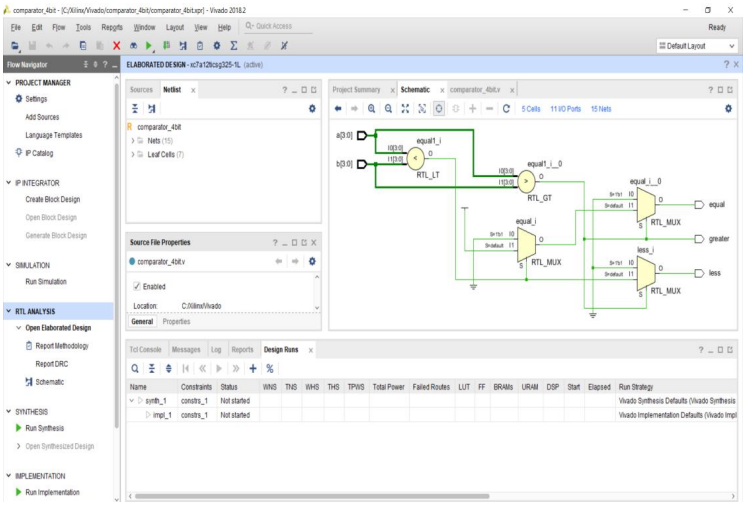
VERILOG RTL CODE AND TESTBENCH CODE FOR MAGNITUDE COMPARATOR :

```
// 4 bit_comparator using Bhaviour modeling

module comparator_4bit(a,b,equal,greater,less);
input [3:0] a;
input [3:0]b;
output reg equal;
output reg greater;
output reg less;

always @(*) begin
equal=0; greater=0; less=0;
   if(a>b)
            begin
               equal = 0;
               less = 0;
               greater = 1;
            end
            else if(a<b)
            begin
               equal = 0;
         less = 1;
         greater = 0;
            end
            else  begin
               equal = 1;
         less = 0;
         greater = 0;
   end
end
endmodule
```

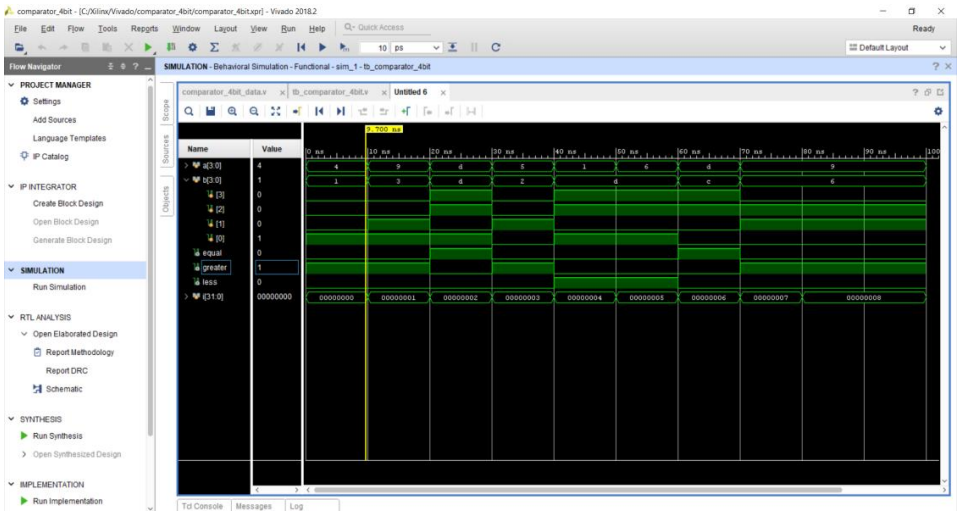
```
// 4 bit_comparator using data flow modeling

module comp_data(a,b,equal,greater,less);

output equal;
output greater;
output less;

input [3:0] a;
input [3:0] b;
assign greater = ~(a[3] & ~b[3] | ~a[3] & b[3])
& a[2] & ~b[2] | a[3] & ~b[3];

assign equal = ~(a[2] & ~b[2] | ~a[2] & b[2]) &
~(a[3] & ~b[3] | ~a[3] & b[3]);

assign less = ~(a[3] & ~b[3] | ~a[3] & b[3]) &
~a[2] & b[2] | ~a[3] & b[3];
endmodule
```

```
module testbench;
reg  [3:0] a;
reg [3:0]b;
wire equal;
wire greater;
wire less;
integer i;

comp_data
dut(.a(a), .b(b), .equal(equal), .greater(greater)
, .less(less));

initial begin
for(i=0;i<8;i=i+1)
begin
a=$random;
b=$random;
#10;
end
end

initial begin
$monitor("a=%b, b=%b, equal=%b,
greater=%b, less=%b" , a,b,equal,greater,less);
#100 $finish;
end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION RESULT :



1.6 CODE CONVERTER

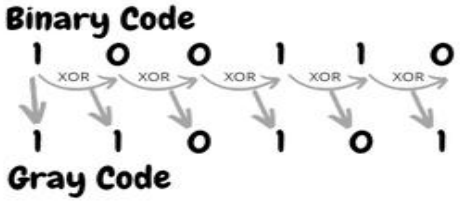
As name itself indicates the code converters are used to convert the code from one number system to another number system. In the practical scenarios, binary to gray and gray to binary converters are used.

1.6.1 BINARY TO GRAY CODE CONVERTER :

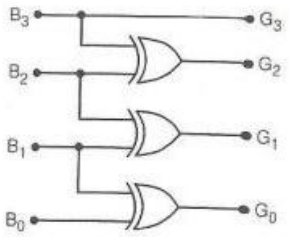
Base of binary number system is 2, for any multi-bit binary number one or more than one bit changes at a time.

BINARY TO GRAY CONVERSION :

Lets take a input string and we have to convert into gray code.



CIRCUIT DIAGRAM BINARY TO GRAY CODE :



BOOLEAN EXPRESSION BINARY TO GRAY CODE :

$$B3 = G3$$

$$B2 \oplus B3 = G2$$

$$B1 \oplus B2 = G1$$

$$B0 \oplus B1 = G0$$

WORKING : HOW TO CONVERT BINARY TO GRAY CODE:

- In the Gray code,the MSB(Most significant Bit) will always be the same as the 1'st bit of the given binary number.
- In order to perform the 2nd bit of the gray code, we perform the exclusive-or (XOR) of the 1'st and 2nd bit of the binary number. It means that if both the bits are different, the result will be one else the result will be 0.
- In order to get the 3rd bit of the gray code, we need to perform the exclusive-or (XOR) of the 2nd and 3rd bit of the binary number. The process remains the same for the 4th bit of the Gray code. Let's take an example to understand these steps.

TRUTH TABLE BINARY TO GRAY CODE:

BINARY INPUT				*GRAY CODE INPUT*			
B3	*B2*	*B1*	*B0*	*G3*	*G2*	*G1*	*G0*
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

VERILOG RTL CODE AND TESTBENCH CODE FOR BINARY TO GRAY CODE :

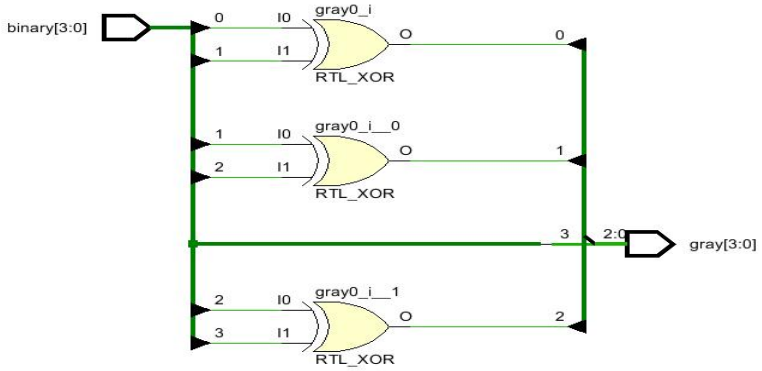
```
//verilog code for 4-bit Binary to Gray code converter using gate level modeling...

module binary_gray(b,g);
input[3:0]b;
output[3:0]g;

buf g1(b[3],g[3]);
xor g2(b[3],b[2],g[2]);
xor g3(b[2],b[1],g[1]);
xor g4(b[1],b[0],g[0]);

endmodule
```

```
module testbench;
reg[3:0]b;
wire[3:0]g;

binary_gray dut(b,g);

initial begin
repeat(8) begin
{b}=$random;

$display("b=%b, g=%b",
b,g);
#1;
end
end
endmodule
```

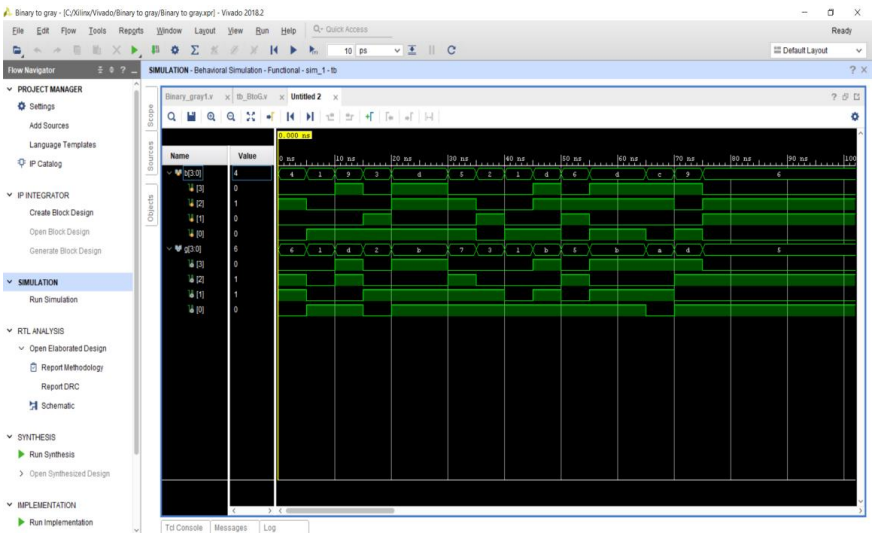
```
//verilog code for 4-bit Binary to Gray code converter using data flow modeling...

module Binary_Gray_code(bin,gray);
input[3:0]bin;
output[3:0]gray;

assign gray[3]=bin[3];
assign gray[2]=bin[3]^bin[2];
assign gray[1]=bin[2]^bin[1];
assign gray[0]=bin[1]^bin[0];

endmodule
```

```
//verilog code for 4-bit Binary to Gray code converter using bhaviour modeling...

module binary_gray(b,g);
input[3:0]b;
output reg [3:0]g;

always@(b)
begin
g[0]=b[1]^b[0];
g[1]=b[2]^b[1];
g[2]=b[3]^b[2];
g[3]=b[3];
end
endmodule
```

RTL IMPLEMENTATION :



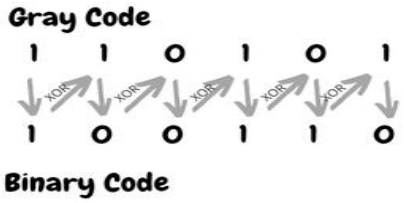
SIMULATION RESULT



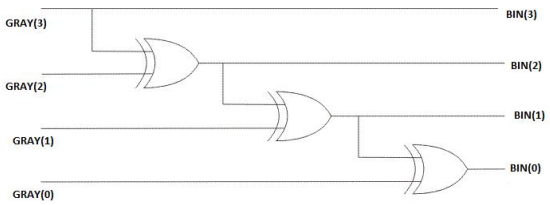
1.6.2 GRAY TO BINARY CODE CONVERSION

The Gray to Binary code converter is a logical circuit that is used to convert the gray code into its equivalent binary code.

GRAY TO BINARY CONVERSION :



CIRCUIT DIAGRAM GRAY TO BINARY CODE:



BOOLEAN EXPRESSION GRAY TO BINARY CODE :

$$b(3) = g(3)$$

$$b(2) = b(3) \oplus g(2)$$

$$b(1) = b(2) \oplus g(1)$$

$$b(0) = b(1) \oplus g(0)$$

TRUTH TABLE GRAY CODE TO BINARY :

Gray Code Input				Binary Code Output			
G3	G2	G1	G0	B3	B2	B1	B0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

APPLICATION OF GRAY CODE :

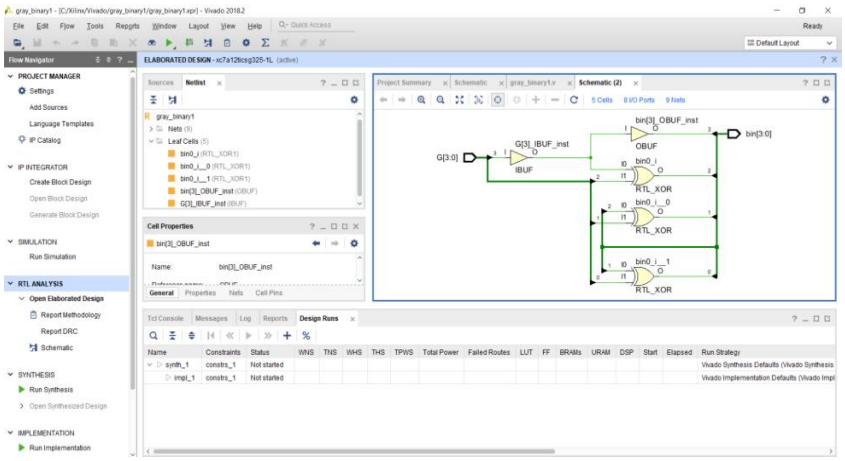
- Boolean circuit minimization
- Communication between clock domains
- Error correction
- Genetic algorithms
- Mathematical puzzles
- Position encode

VERILOG RTL CODE AND TESTBENCH CODE FOR GRAY CODE TO BINARY:

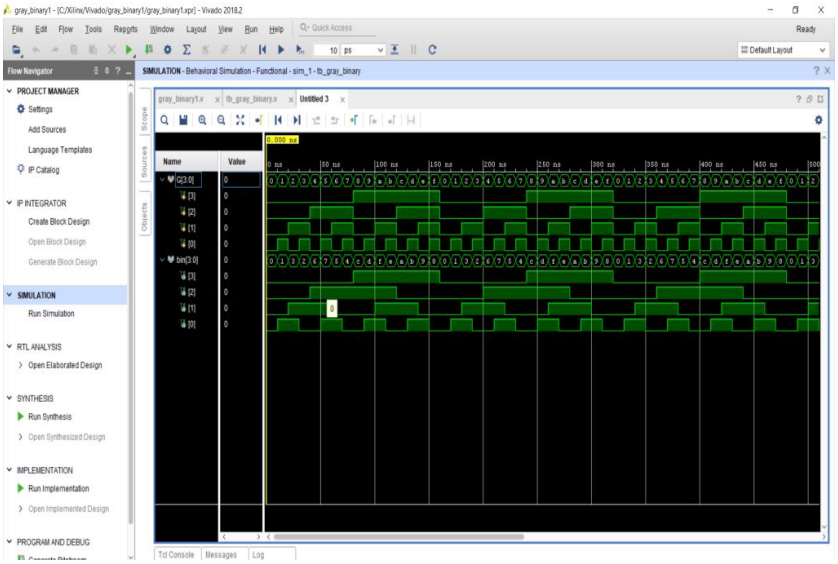
```
//verilog code for 4-bit Gray code to binary converter
using data flow modeling…

module gray_binary1(G,bin);
input [3:0]G;
output [3:0]bin;//binary input

assign bin[3] = (G[3]);
assign bin[2] = (G[3] ^ G[2]);
assign bin[1] = (G[2] ^ G[1]);
assign bin[0] = (G[1] ^ G[0]);

endmodule
```

```
module testbench_gray_binary;
reg [3:0]G;
wire [3:0]bin;

gray_binary1 dut(.G(G),  .bin(bin));

initial begin
G = 4'b0000;
#1000;
$finish;
End

always #10 G[0] = ~G[0];
always #20 G[1] = ~G[1];
always #40 G[2] = ~G[2];
always #80 G[3] = ~G[3];

endmodule
```

RTL IMPLEMENTATION



SIMULATION RESULT :



COMBINATIONAL LOGIC CIRCUITS - II

2.1 MULTIPLEXERS :

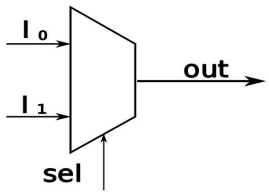
Multiplexer is a combinational circuit that has maximum of 2n data inputs, 'n' selection lines and single output line.
Multiplexers are also called as universal logic and terminology used in the practical world is MUX.
Its select input also called as data selector.
Multiplexer consumes lesser area as compared to adders and most of the time MUX is used to implement arithmetic components such as adders and subtractors.

for 'n' selection lines, there are N = 2^n input lines.

N:1 denotes it has 'N' input lines and one output line.

2:1 MULTIPLEXERS :

A 2:1 MUX has two input lines, one select line and one output line.



WORKING OF 2X1 MULTIPLEXERS :

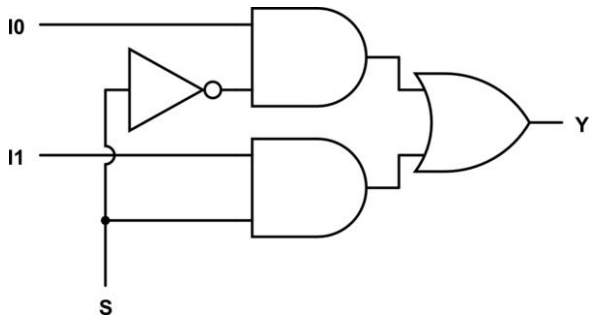
A 2-to-1 multiplexer consists of two inputs I0 and I1, one select input Sel and one output out. Depending on the select signal, the output is connected to either of the inputs. Since there are two input signals, only two ways are possible to connect the inputs to the outputs.

If the select line is low, then the output will be switched to D0 input, whereas if select line is
high, then the output will be switched to D1 Input.

BOOLEAN EXPRESSION OF 2 to 1 :

Out = sel' . i0 + sel . i1

LOGICAL CIRCUIT OF 2:1 MULTIPLEXERS :



ADVANTAGES OF MULTIPLEXERS :

ADVANTAGES :

- It reduce the no of wires
- It reduces the cost of circuit and complexity of the circuit.
- It makes transmission circuit economical and les complex

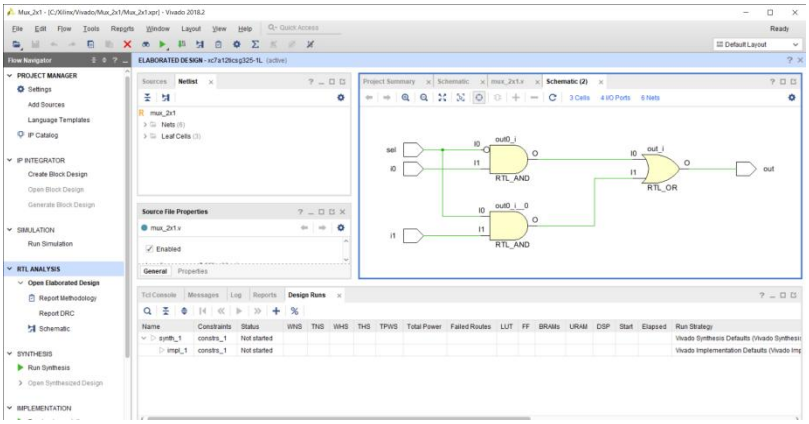
VERILOG RTL CODE AND TESTBENCH CODE FOR 2x1 MULTIPLEXERS :

```
//verilog code for mux_2x1 using data flow modeling…


module mux_2x1(input sel,i0,i1,
 output out);

assign out = (~sel&i0)|(sel&i1);
endmodule
```

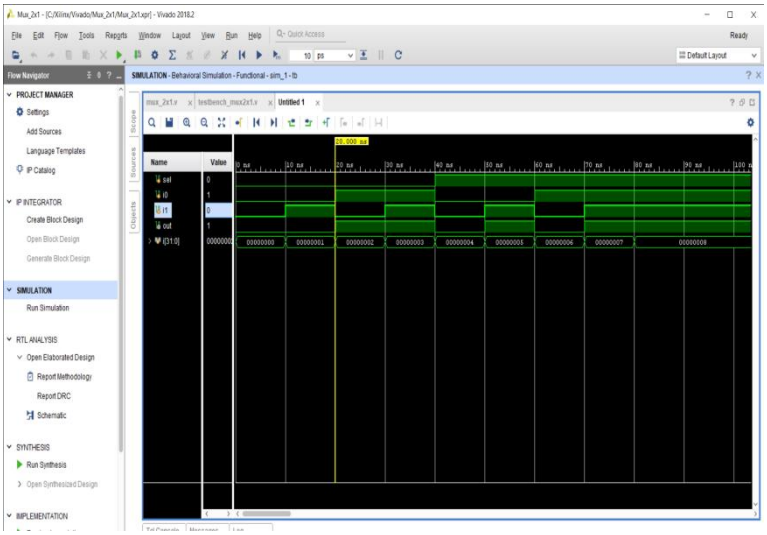
```
//verilog code for mux_2x1 using bhaviour modeling…

module mux2x1(input sel,i0,i1,
 output  out);
        always @(*) begin
        if(sel)
                out = i1;
        else
                out = i0;
        end
endmodule
```

```
module testbench;
   reg sel,i0,i1;
   wire out;
   integer i;

 mux_2x1 dut(sel,i0,i1,out);

 initial begin
   for(i=0;i<8;i=i+1)
     begin
      {sel,i0,i1} = i;
      #10;
     end
 end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION RESULT :

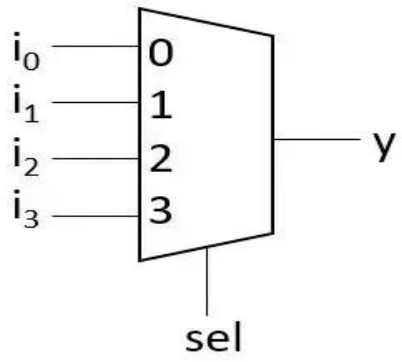


2.1.2 4x1 MULTIPLEXERS :

In the 4×1 multiplexer, there is a total of four inputs, i.e., i_0, i_1, i_2, and i_3, 2 selection lines, i.e., S_0 and S_1 and single output, i.e.Y. On the basis of the combination of inputs that are present at the selection lines S0 and S1, one of these 4 inputs are connected to the output.

BLOCK DIAGRAM 4x1 MULTIPLEXERS :

One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines.



TRUTH TABLE 4x1 MUX :

sel[0]	sel[1]	y
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

4 X 1 WORKING MULTIPLEXERS :

When both the select inputs Sel0 = 0, Sel1 = 0, the top of AND logic gate is enable and other two AND gate is disable, so the data input D_0 input line is selected and transmitted as output.

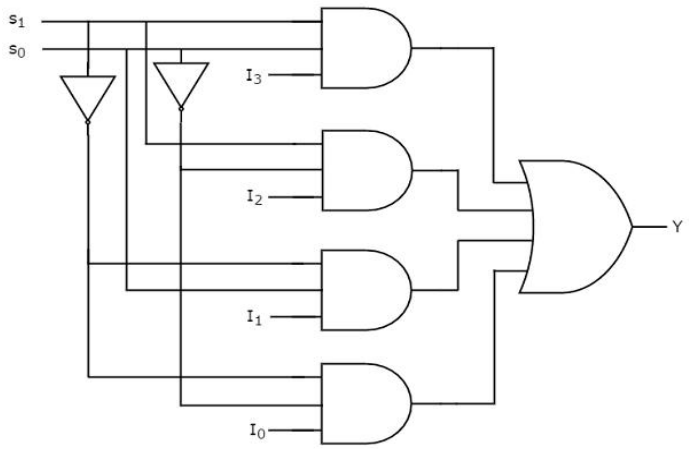
Hence, the output Y = i_0.When both the select inputs Sel0 = 1, Sel1= 1, the bottom of AND logic gate is enable and other two AND gate is disable, so the data input i_3 input line is selected and transmitted as output. Hence, the output Y = i_3. And so on.

BOOLEAN EXPRESSION OF 4x1 MULTIPLEXERS :

Y=Sel1' Sel0' i0+Sel1' Sel0 i1+Sel1 Sel0' i2+Sel1 Sel0 i3.

LOGICAL CIRCUIT 4x1 MUX :

We can implement this Boolean function using Inverters, AND gates & OR gate.
The **circuit diagram** of 4x1 multiplexer



VERILOG RTL CODE AND TESTBENCH CODE FOR 4x1 MULTIPLEXERS :

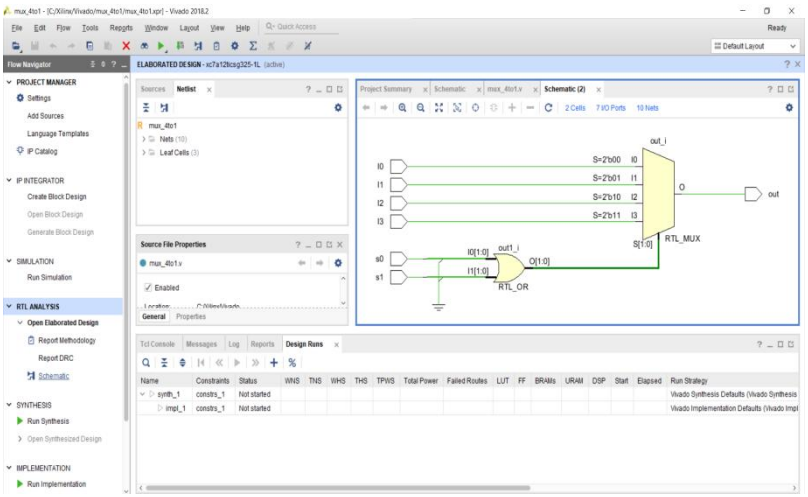
```
//verilog code for mux4x1 using gate/structure level modeling.

module mux4x1_g(out,i0,i1,i2,i3,s0,s1);
output out;
input i0,i1,i2,i3,s0,s1;
wire s0bar,s1bar,w0,w1,w2,w3;

not g1(s0bar,s0),(s1bar,s1);
and g2(w1,i0,s0bar,s1bar),(w2,i1,s0bar,s1),(w3,i2,s0,sbar),(w4,i3,s0,s1);
or  g3(out,w1,w2,w3,w4);

endmodule
```

```
input s0,s1;
output out;

assign out = (s1 ? i3: i2) : (s0 ? i1 : i0);

endmodule
```

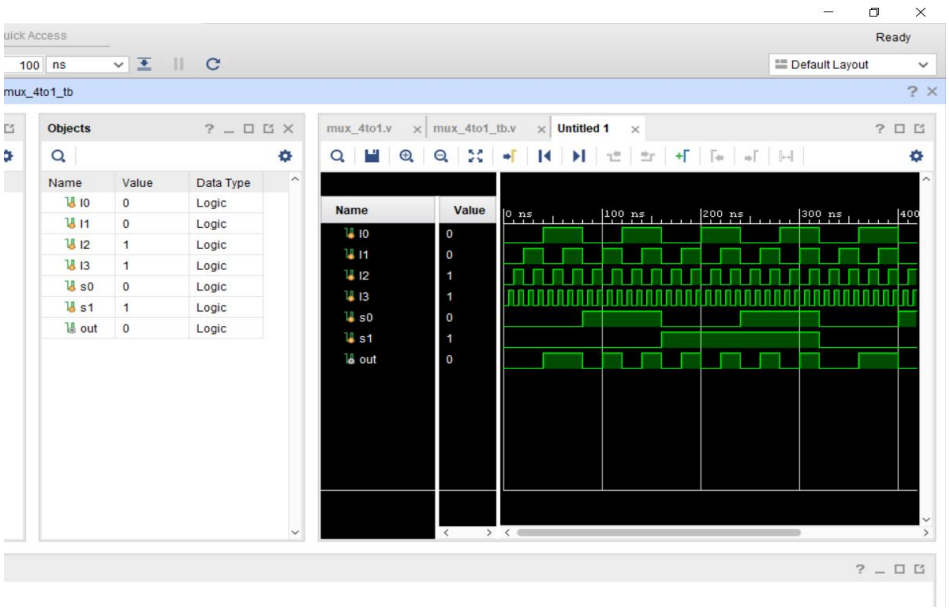
```
module testbench;
reg  I0;
reg  I1;
reg  I2;
reg  I3;
reg s0, s1;
wire out;

mux_4to1 uut(.out(out), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .s0(s0), .s1(s1));
 initial begin
 I0=1'b0; I1=1'b0; I2=1'b0; I3=1'b0;
 s0=1'b0; s1=1'b0;
 #500 $finish;
end
always #40 I0=~I0;
always #20 I1=~I1;
always #10 I2=~I2;
always #5 I3=~I3;
always #80 s0=~s0;
always #160 s1=~s1;
always@(I0,I1,I2,I3, s0 , s1)
$monitor("At time = %t, Output = %d", $time, out);
endmodule
```

```
//verilog code for mux4x1 using Bhaviour level modeling.

module mux4x1_b(out,i0,i1,i2,i3,s0,s1);
output reg out;
input s0,s1,i0,i1,i2,i3;

always @(*)
begin
case(sel)
   2'b: y = i0;
            2'b: y = i1;
            2'b: y = i2;
   2'b: y = i3;
default:2'bx;
$display("i0=%b,i1=%b,i2=%b,i3=%b,s0=%b,s1=%b,out=%
b" ,i0,i1,i2,i3,s0,s1,out);
endcase
end
endmodule
```

RTL IMPLEMENTATION :



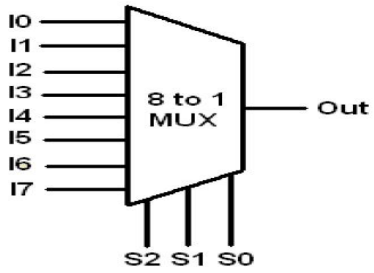
SIMULATION RESULT :



8x1 MULTIPLEXERS :

In the 8 to 1 multiplexer, there are total eight inputs, i.e., i0, i1, i2, i3, i4, i5, i6, and i7, 3 selection lines,I.e S0, S1and S2 and single output, i.e., Y. On the basis of the combination of inputs that are present at the selection lines S0, S1, and S2, of these 8 inputs are connected to the output.

BLOCK DIAGRAM 8x1 MULTIPLEXERS :



TRUTH TABLE 8x1 MULTIPLEXERS :

Inputs			Output
S_2	S_1	S_0	O
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

8 X1 MULTIPLEXER WORKING PRINCIPLE :

When all select input line are S_0=0 S_1=0, S_2=0, then the topmost AND logic gates is enable

- and all other AND logic gates are disable so the data input D_0 is selected and transmitted as output. The output is Y = i_0.
- When all select input line are S_0=1 S_1=1, S_2=, then the topmost AND logic gates is disable and all other AND logic gates are enable so the data input i_7 is selected and transmitted as output. The output is Y = i_7.

BOOLEAN EXPRESSION 8x1 MULTIPLEXERS :

Out = S0'S1'S2'I0 + S0S1'S2'I1 + S0'S1S2'I2 + S0S1S2'I3 + S0'S1'S2I4 + S0S1'S2I5 + S0'S1S2I6 + S0S1S2I7.

APPLICATIONS OF MULTIPLEXERS :

Multiplexers are used in various types of applications :

Communication System:- In communication system, the transmission data

efficiency of communication system can be increase by using multiplexer.

Such audio and video data through a single line.

Computer Memory:- In computer memory, to maintain the huge amount of memory by used multiplexer. It reduces the copper lines requirement to connect the other part of memory.

Telephone Network:- Multiple voice signals are integrated by using multiplexer technique.

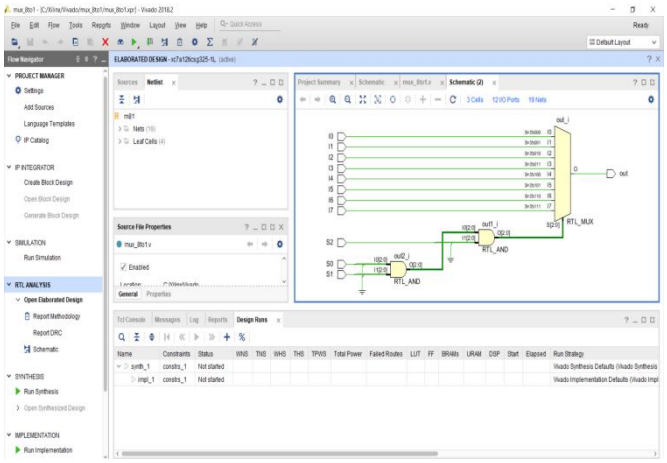
VERILOG RTL CODE AND TESTBENCH CODE FOR 8x1 MULTIPLEXERS :

```
//verilog code for mux8x1 using Bhaviour level modeling.
module mux8x1_b(i,s,out);
input[7:0]i;
input[2:0]s;
output out;
reg out;
always @(*)
begin
case(s)
  3'b000: out = i[0];
          3'b001: out = i[1];
          3'b010: out = i[2];
          3'b011: out = i[3];
          3'b100: out = i[4];
          3'b101: out = i[5];
          3'b110: out= i[6];
          3'b111: out = i[7];
          default: out = 1'bx;
endcase
end
endmodule
```

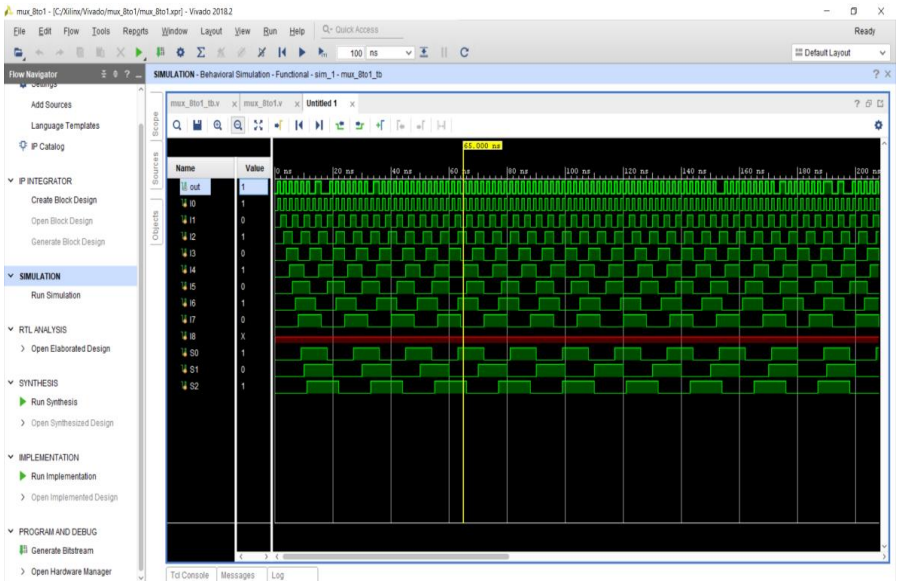
```
//verilog code for mux4x1 using gate/structure level modeling.
module mux8x1_g(out,i,s);
output out;
input[7:0]i,
input[2:0]s;
wire[7:0]w;

not g1(s0bar,s0);
not g2(s1bar,s1);
not g3(s2bar,s2);

and g4(w1,i0,s2bar,s1bar,s0bar);
and g5(w2,i1,s2bar,s1bar,s0);
and g6(w3,i2,s2bar,s1,s0bar);
and g7(w4,i3,s2bar,s1,s0);
and g8(w5,i4,s2,s1bar,s0bar);
and g9(w6,i5,s2,s1bar,s0);
and g10(w7,i6,s2,s1,s0bar);
and g11(w8,i7,s2,s1,s0);

or  g12(out,w1,w2,w3,w4,w5,w6,w7,w8);

endmodule
```

```
module testbench;
reg[7:0]i;
reg[2:0]s;
wire out;
//wire[7:0]w;

mux8x1_b mux_b(i,s,out);

initial begin
      repeat(8) begin
          {i,s}  = $random;
           #5;
      $display("time = %t, i=%b,sel = %b,out = %b",
$time,i,s,out);
     end
 end
endmodule
```

RTL IMPLEMENTATION MUX 8x1 :

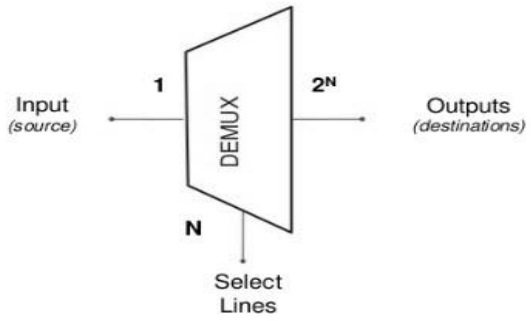


SIMULATION RESULT :



2.2 DEMULTIPLEXER

- The word “Demultiplex” means “One into many”.
- DeMultiplexing is the process of taking information from one input and transmitting
- the same over one of several outputs.
- Demultiplexer is an also a combinational circuit in which one input and more than two output.
- It has single input, ‘n’ selection lines and maximum of 2n outputs.
- The demultiplexer is a reverse process of multiplexer.De-multiplexer is also treated as De-mux.
- Since there are ‘n’ selection lines, there will be 2n possible combinations of zeros and ones.
- So, each combination can select only one output.



1x8 De-Multiplexer :

In 1 to 8 De-multiplexer, there are total of eight outputs, i.e., Y0,Y1,Y2,Y3,Y4,Y5,Y6 and Y7,

3 selection lines, i.e., S_0, $S_1$and S_2 and single input, i.e., A. On the basis of the combination of inputs which are present at the selection lines S^0, S^1 and S_2, the input will be connected to one of these outputs.

BOOLEAN EXPRESSION 1X8 DEMUX

Y0 = S2 S1 S0 D

Y1 = S2 S1 S0 D

Y2 = S2 S1 S0 D

Y3 = S2 S1 S0 D

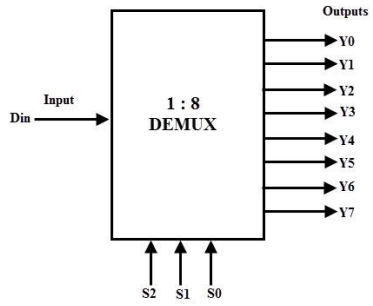
Y4 = S2 S1 S0 D

Y5 = S2 S1 S0 D

$$Y6 = S2\ S1\ S0\ D$$

$$Y7 = S2\ S1\ S0\ D$$

BLOCK DIAGRAM 1x8 DE-MULTIPLEXER :



TRUTH TABLE FOR 1-to-8 DE-MULTIPLEXER :

Data Input	Select Inputs			Outputs							
D	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	0	0	0	0	0	D
D	0	0	1	0	0	0	0	0	0	D	0
D	0	1	0	0	0	0	0	0	D	0	0
D	0	1	1	0	0	0	0	D	0	0	0
D	1	0	0	0	0	0	D	0	0	0	0
D	1	0	1	0	0	D	0	0	0	0	0
D	1	1	0	0	D	0	0	0	0	0	0
D	1	1	1	D	0	0	0	0	0	0	0

CIRCUIT DIAGRAM USING LOGIC GATE

APPLICATION OF DEMULTIPLEXER :

Since the demultiplexers are used to select or enable the one signal out of many, these are extensively used in microprocessor or computer control systems such as:

Selecting different IO devices for data transfer (Data Routing)

- Choosing different banks of memory (Memory Decoding)
- Depends on the address, enabling different rows of memory chips
- Enabling different functional units.
- A demultiplexer is used to connect a single source to multiple destinations.
- Demultiplexers are mainly used in the field of the communication system.
- DEMUX are used in Serial to Parallel Converter.
- For store the output of Arithmetic Logic Unit in multiple register.

DIFFERENCE BETWEEN MULTIPLEXER AND DEMULTIPLEXER

MULTIPLEXER

- It is a combinational circuit in which have several input and one output.
- It is a digital switching circuit.
- It is a parallel to serial converter.
- The multiplexer used in TDM.
- 8-1 MUX, 16-1 MUX, and 32-1 MUX are the different type of MUX
- In multiplexer, the selection lines are used to control the specific input.
-

DEMULTIPLEXER

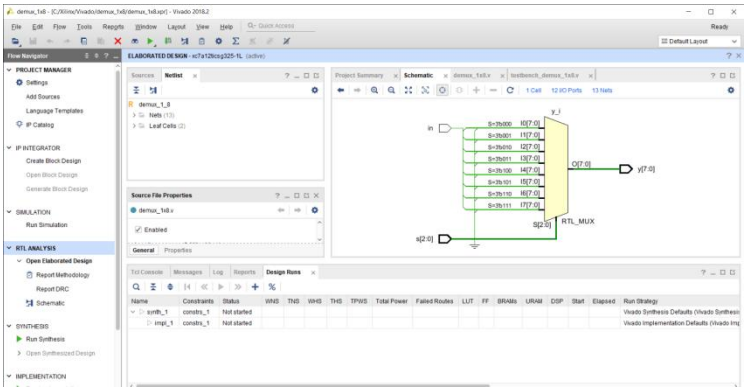
- It is a combination circuit.
- It has one input and several outputs.
- It is a data distributer.
- The serial to parallel conversion.
- It works on the principle of one-to-many.
- The various types of demultiplexers are 1-8 Demux, 1-16 Demux, 1-32 Demux.

VERILOG RTL CODE AND TESTBENCH CODE FOR 1x8 DEMULTIPLEXERS :

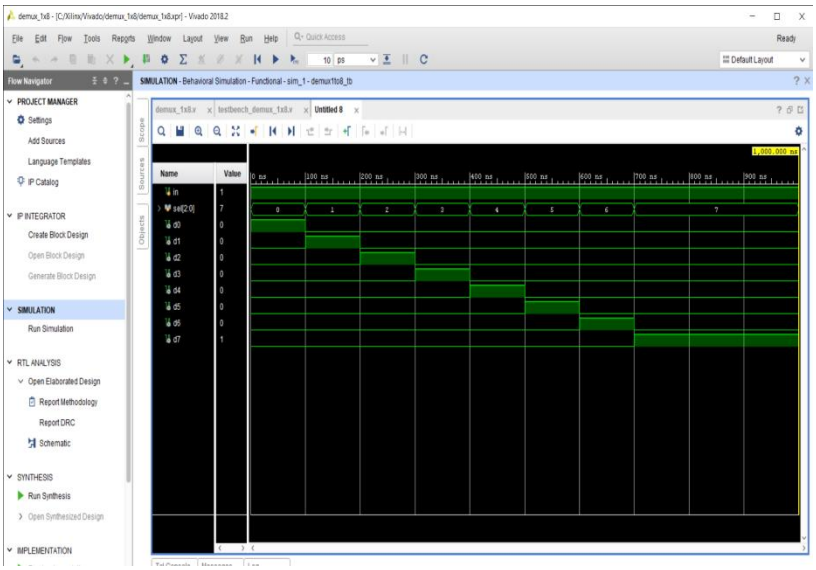
```
//verilog code for demux 1x8 using Bhaviour level
modeling.

module demux_1_8(y,s,in);
output reg [7:0]y;
input [2:0]s;
input in;
always @(*)
begin
y=0;
case(s)
3'd0: y[0]=in;
3'd1: y[1]=in;
3'd2: y[2]=in;
3'd3: y[3]=in;
3'd4: y[4]=in;
3'd5: y[5]=in;
3'd6: y[6]=in;
3'd7: y[7]=in;
endcase
end
endmodule
```

```
module testbench;
reg [2:0]S;
reg in;
wire [7:0]Y;
demux_1_8 mydemux(.y(Y), .in(in), .s(S));
initial begin
in=1;
S=3'd0;
#5;
S=3'd1;
#5;
S=3'd2;
#5;
S=3'd3;
#5;
S=3'd4;
#5;
S=3'd5;
#5;
S=3'd6;
#5;
S=3'd7;
#5;
end
endmodule
```

RTL IMPLEMENTATION :



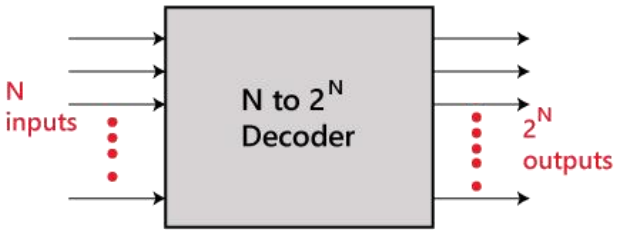
SIMULATION RESULT :



2.3 DECODER

Decoder has ‘n’ select lines or input lines and ‘m’ output lines and used to generate either active high or active low output. The relation between select lines and output lines is given by m = 2^n Depending on the logic status on ‘n’ input lines at a time one of the output line goes high or low.the **Decoder** performs the reverse operation of the **Encoder**. at a time, only one input line is activated for simplicity. The produced 2^N-bit output code is equivalent to the binary information.

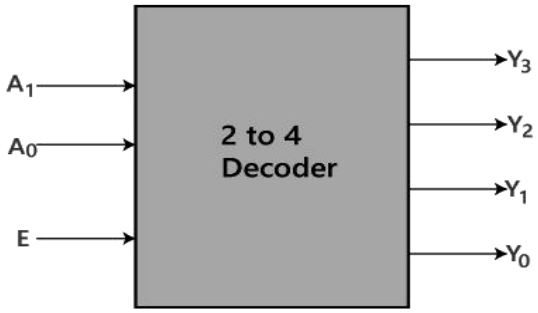
BLOCK DIAGRAM OF DECODER :



2 to 4 Decoder :

In the 2 to 4 line decoder, there is a total of three inputs, i.e., A_0, and A_1 and E and four outputs, ie., Y_0, Y_1, Y_2, and Y_3. For each combination of inputs, when the enable 'E' is set to 1, one of these four outputs will be 1.

BLOCK DIAGRAM 2x4 DECODER :



TRUTH TABLE 2x4 DECODER :

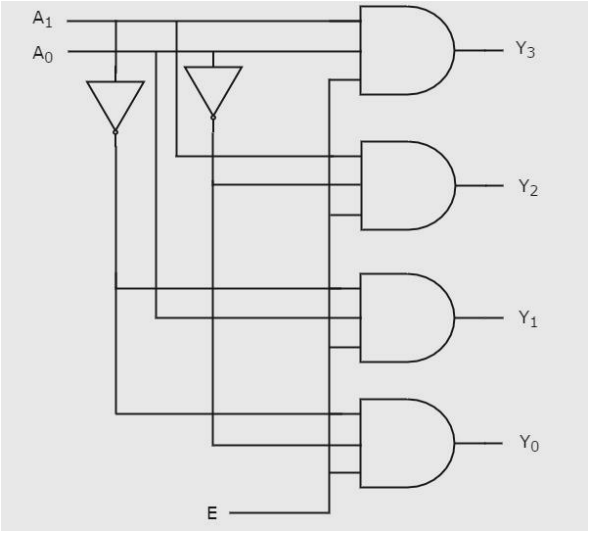
Enable	INPUTS		OUTPUTS			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

BOOLEAN EXPRESSION 2x4 DECODER :

logical expression of the term Y0, Y0, Y2, and Y3 is as follows:

$Y_3=E.A_1.A_0$
$Y_2=E.A_1.A_0'$
$Y_1=E.A_1'.A_0$
$Y0=E.A_1'.A_0'$

CIRCUIT DIAGRAM USING LOGIC GATE 2x4 DECODER :



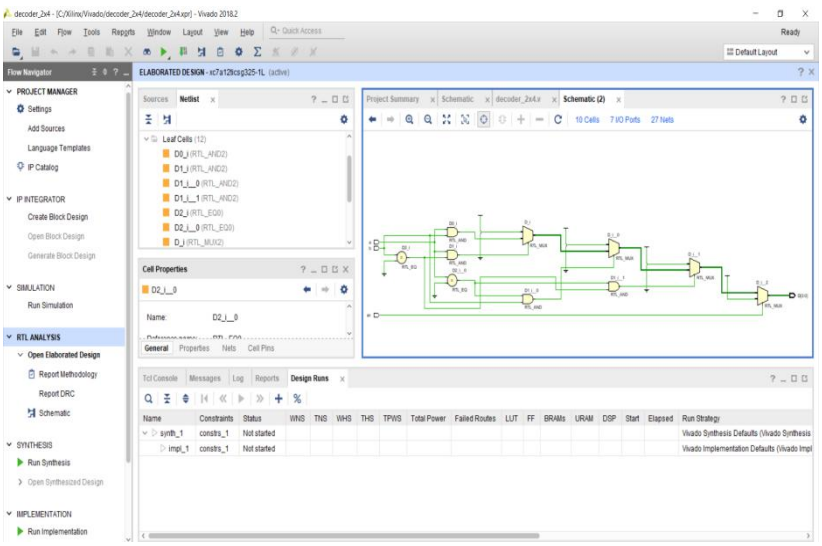
VERILOG RTL CODE AND TESTBENCH CODE FOR 2x4 DEODER :

```
//implement 2x4_decoder using Behavioral Modeling...


module decoder_2x4(en ,a ,b ,D);
input en,a,b;
output reg [3:0]D;

always@(en,a,b)
begin
if(en==0)
begin
if(a==1'b0 & b==1'b0)  D=4'b1110;
else if(a==1'b0 & b==1'b1)  D=4'b1101;
else if(a==1'b1 & b==1'b0)  D=4'b1011;
else if(a==1'b1 & b==1'b1)  D=4'b0111;
else D =4'b0000;
end
else
D=4'b1111;
end
endmodule
```

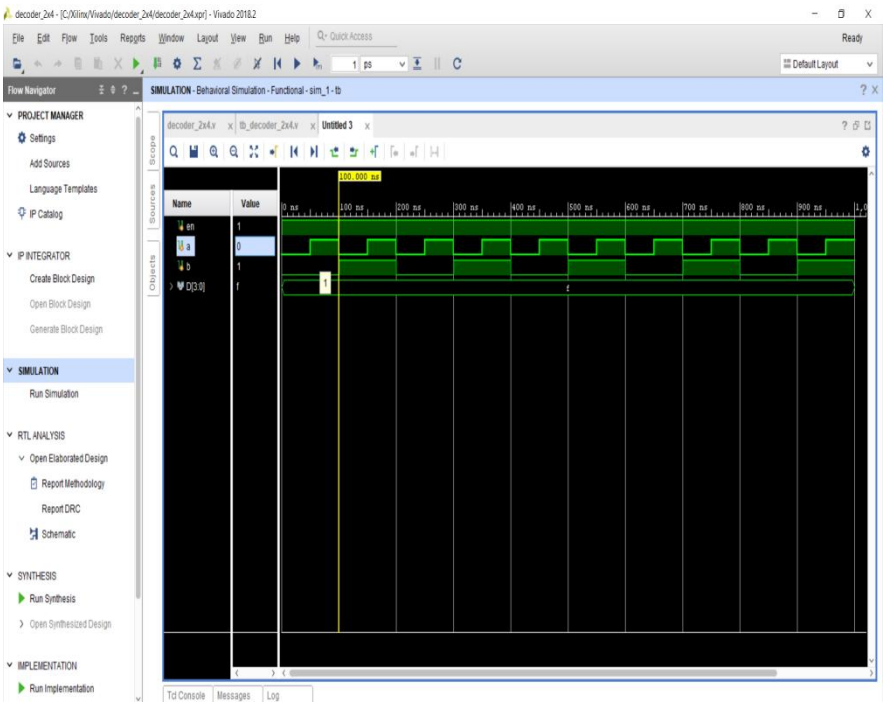
```
/implement decoder_2x4 using data_flow modeling...
//syntax: assign out = expression;

module decoder_2x4(a,b,D);
input a,b;
output[3:0]D;

//assign output value by referring to logic diagram
assign D[0] = (~a)&(~b);
assign D[1] =  (~a)&(b);
assign D[2] =  (a)&(~b);
assign D[3] =  (a)&(b);
endmodule
```

```
module testbench;
reg en;
reg a,b;
wire [3:0]D;

decoder_2x4 dut(en,a,b,D);

initial begin
$monitor("en=%b,a=%b,b=%b,D=%b", en,a,b,D);
en = 1'b0;
a=1'b0; b=1'b0;
en = 1'b1;
end
begin
always #50 a=~a;
always #100 b=~b;
end
endmodule
```

RTL IMPLEMENTATION :



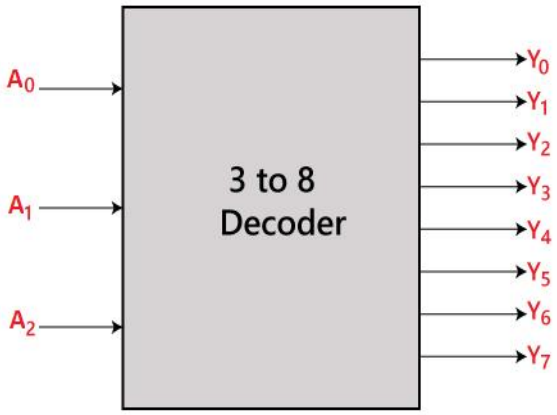
SIMULATION LOGIC :



3 to 8 line decoder:

The 3 to 8 line decoder is also known as **Binary to Octal Decoder**. In a 3 to 8 line decoder, there is a total of eight outputs, i.e., Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 and three outputs, i.e., A_0, A1, and A_2. This circuit has an enable input 'E'. Just like 2 to 4 line decoder, when enable 'E' is set to 1, one of these four outputs will be 1.

BLOCK DIAGRAM 3x8 DECODER :



TRUTH TABLE 3x8 DECODER :

Enable	INPUTS			Outputs							
E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

BOOLEAN EXPRESSION FOR 3x8 DECODER :

The logical expression of the term Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 is as follows:

$$Y_0=A_0'.A_1'.A_2'$$

$$Y_1=A_0.A_1'.A_2'$$

$$Y_2=A_0'.A_1.A_2'$$

$$Y_3=A_0.A_1.A_2'$$

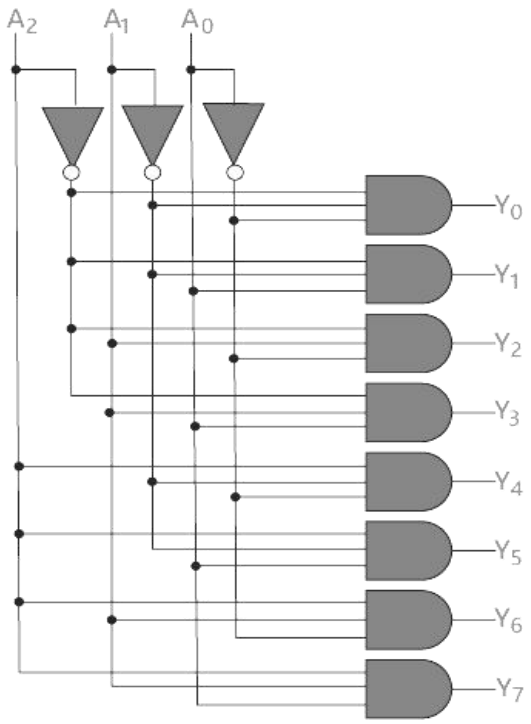
$$Y_4=A_0'.A_1'.A_2$$

$$Y_5=A_0.A_1'.A_2$$

$$Y_6=A_0'.A_1.A_2$$

$$Y_7=A_0.A_1.A_2$$

CIRCUIT DIAGRAM USING LOGIC GATE 3x8 DECODER :

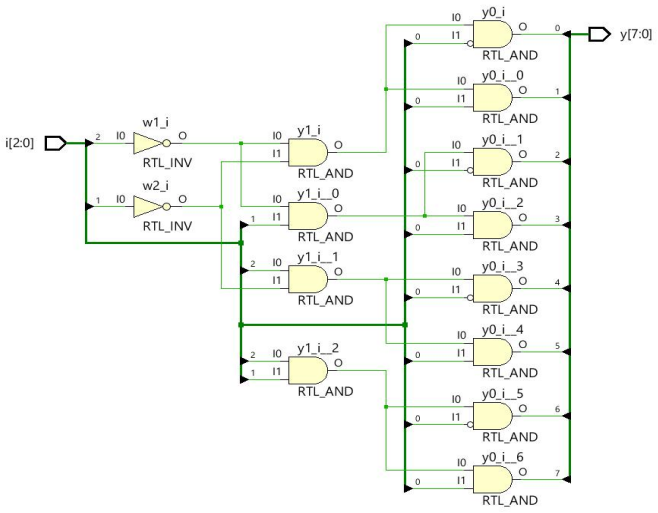


VERILOG RTL CODE AND TESTBENCH CODE FOR 3x8 DECODER :

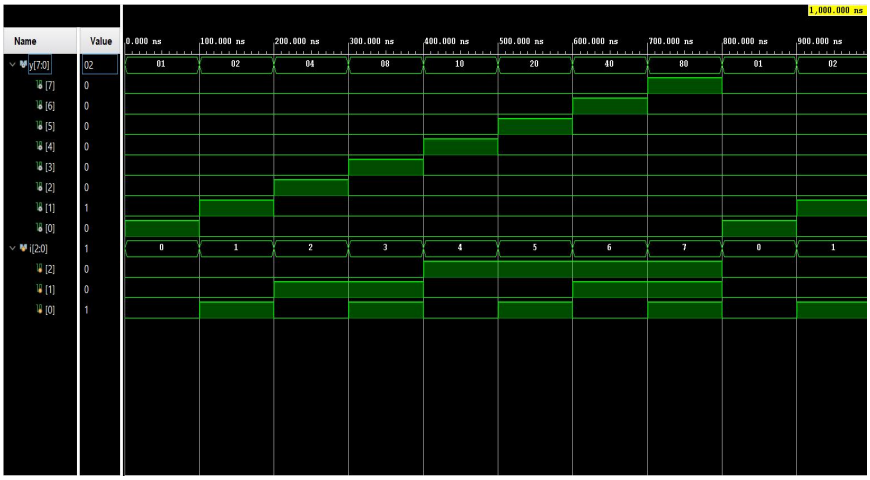
```
//
module decoder_3x8(in,En,D);
input[2:0]in;
input En;
output reg[7:0]y;

//functionality of design..

always@(in or En)
begin

if(En)
case(in)
  3'b000: y=8'b0000_0001;
  3'b001: y=8'b0000_0010;
  3'b010: y=8'b0000_0100;
  3'b011: y=8'b0000_1000;
  3'b100: y=8'b0001_0000;
  3'b101: y=8'b0010_0000;
  3'b110: y=8'b0100_0000;
  3'b111: y=8'b1000_0000;
  endcase
else
  y=8'b0000_0000;

end
endmodule
```

```
module testbench;
reg[2:0]in;
reg En;

wire[7:0]y;

decoder_3x8
dut(.in(in), .En(En), .y(y));
initial begin
 En=1'b1;
 #100
 in=3'b000;
 #100
 in=3'b001;
 #100
 in=3'b010;
 #100
 in=3'b011;
 #100
 in=3'b100;
 #100
 in=3'b101;
 #100
 in=3'b110;
 #100
 in=3'b111;
 #100;

 $monitor("in=%b,
En=%b,y=%b" ,in,En,y);

end
endmodule
```

RTL IMPLEMENTATION :

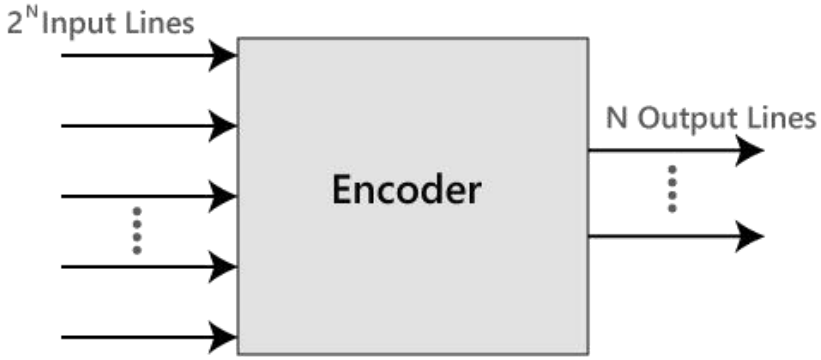


SIMULATION RESULT :



2.4 ENCODER

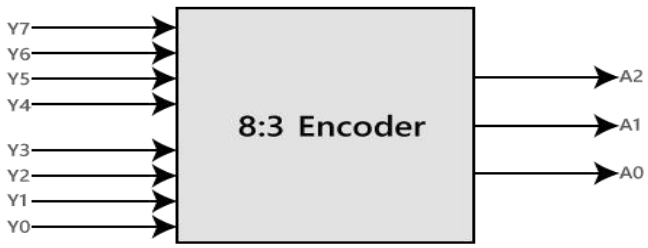
The combinational circuits that change the binary information into N output lines are known as **Encoders**. The binary information is passed in the form of 2^N input lines. The output lines define the N-bit code for the binary information. In simple words, the **Encoder** performs the reverse operation of the **Decoder**. At a time, only one input line is activated for simplicity.



8 x3 LINE ENCODER :

The 8 to 3 line Encoder is also known as **Octal to Binary Encoder**. In 8 to 3 line encoder, there is a total of eight inputs, i.e., Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 and three outputs, i.e., A_0, A1, and A_2. In 8-input lines, one input-line is set to true at a time to get the respective binary code in the output side. Below are the block diagram and the truth table of the 8 to 3 line encoder.

BLOCK DIAGRAM FOR 8x3 ENCODER:



TRUTH TABLE 8x3 ENCODER :

INPUTS								OUTPUTS		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

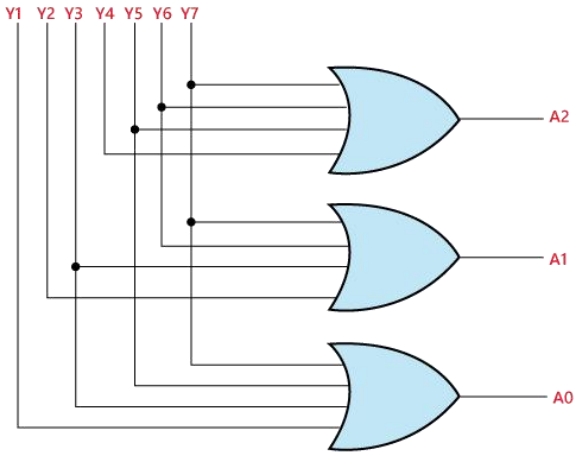
BOOLEAN EXPRESSION FOR 8X3 ENCODER :

$$A_2=Y_4+Y_5+Y_6+Y_7$$

$$A_1=Y_2+Y_3+Y_6+Y_7$$

$$A_0=Y_7+Y_5+Y_3+Y_1$$

CIRCUIT DIAGRAM USING LOGIC GATES 8x3 ENCODER :

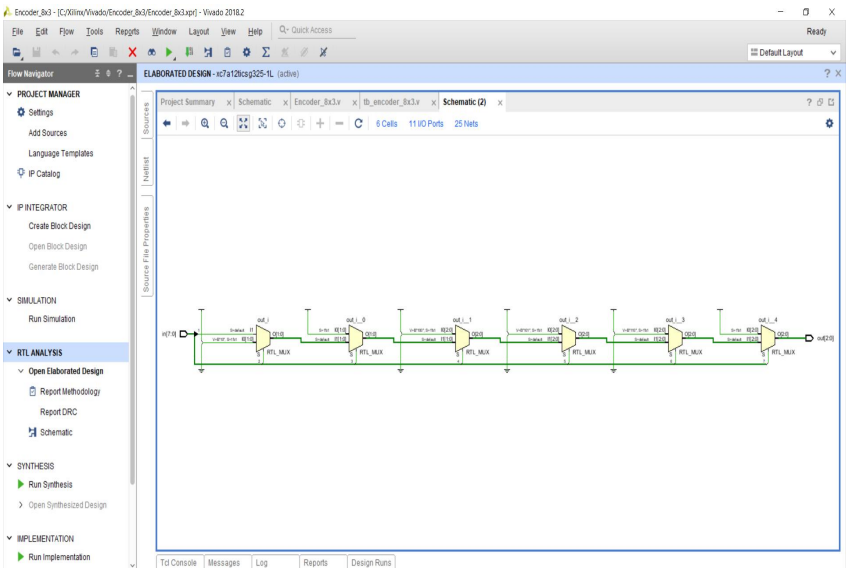


VERILOG RTL CODE AND TESTBENCH CODE FOR 8x3 ENCODER :

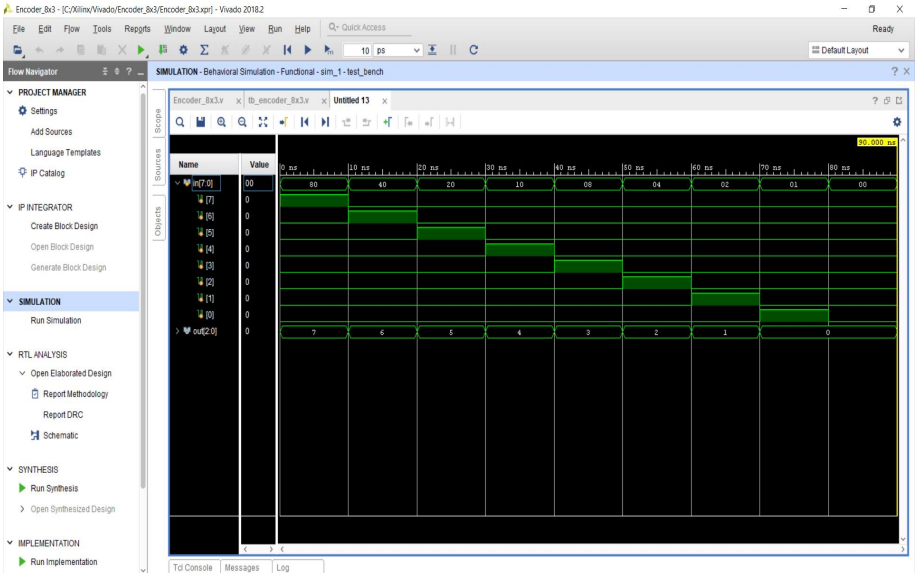
```
module encoder_8_3(
  input[7:0]in,
  output reg[2:0]out );
        always@(in)
        begin
                if(in[7]==1)
                 out=3'b111;
                else if(in[6]==1)
                 out=3'b110;
                else if(in[5]==1)
                 out=3'b101;
                else if(in[4]==1)
                 out=3'b100;
                else if(in[3]==1)
                 out=3'b011;
                else if(in[2]==1)
                 out=3'b010;
                else if(in[1]==1)
                 out=3'b001;
                else
                 out=3'b000;
        end
endmodule
```

```
module testbench;
          reg [7:0] in;
          wire [2:0] out;
         encoder_8_3
dut(.in(in), .out(out));
          Initial begin
                    #0 in=8'b10000000;
                    #10 in=8'b01000000;
                    #10 in=8'b00100000;
                    #10 in=8'b00010000;
                    #10 in=8'b00001000;
                    #10 in=8'b00000100;
                    #10 in=8'b00000010;
                    #10 in=8'b00000001;
                    #10 in=8'b00000000;
          end
  initial
  begin $monitor("in: %b   out: %b ",in,
out);
  #100 $finish;
  end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION WAVEFORM :



2.5 PRIORITY ENCODER

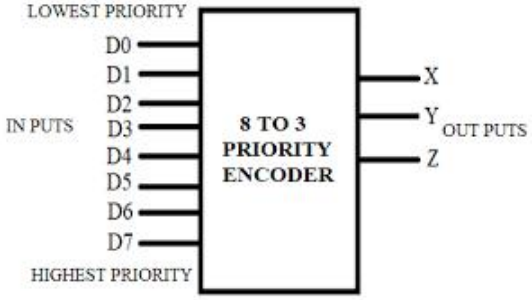
8x3 BIT PRIORITY ENCODER :

8 to 3-bit priority encoder is an encoder that consists of 8 input lines and 3 output lines.

It can also be called an Octal to a binary encoder. Each input line has a base value of

8 (octal) and each output has a base value of 2 (binary).

BLOCK DIAGRAM 8x3 PRIORITY ENCODER :



TRUTH TABLE :

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

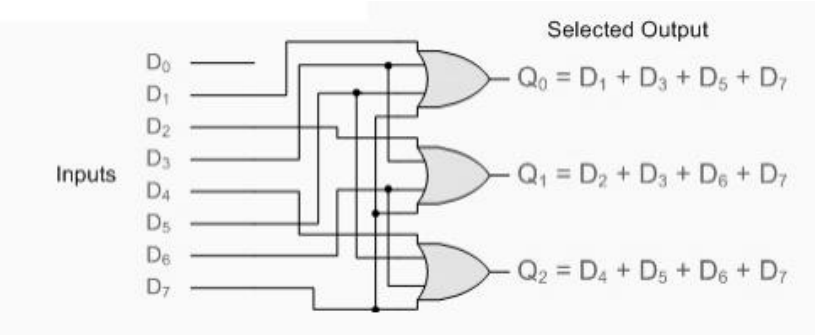
BOOLEAN EXPRESSION :

Q0 = D4+D5+D6+D7

Q1 = D2+D3+D6+D7

Q2 = D1+D3+D5+D7

CIRCUIT DIAGRAM USING LOGIC GATES :



APPLICATION :

Some of the **applications of priority encoder** are,

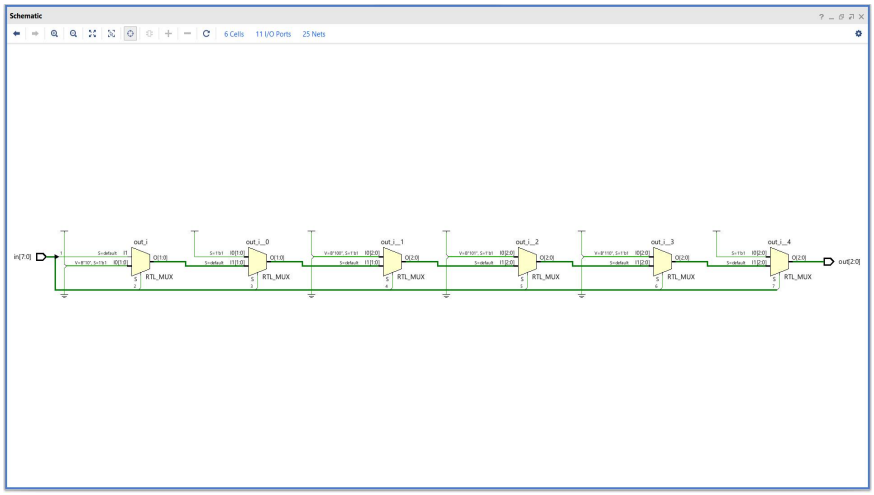
- It is used to reduce the no. of wires and connections required for electronic circuit designing
- that have multiple input lines. Example keypads and keyboards.
- Used in controlling the position in the ship's navigation and robotics arm position.
- Used in the detection of highest priority input in various applications of microprocessor
- interrupt controllers.
- Used to encode the analog to digital converter's output.
- Used in synchronization of the speed of motors in industries.
- Used robotic vehicles
- Used in applications of home automation systems with RF
- Used in hospitals for health monitoring systems
- Used in secure communication systems with RF technology to enable secret code.

VERILOG RTL CODE AND TESTBENCH CODE FOR 8x3 PRIORITY ENCODER :

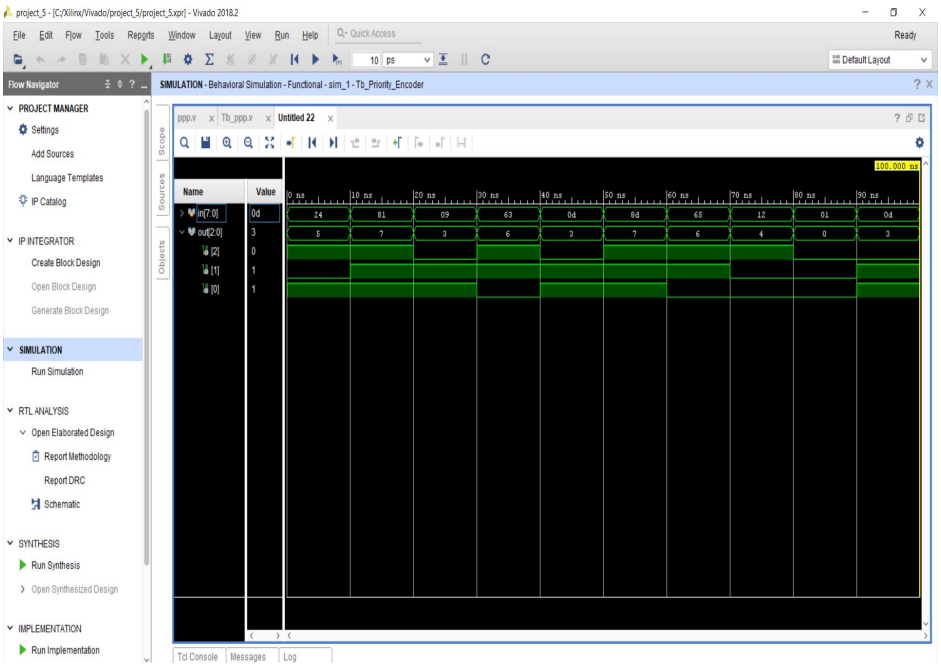
```
module Priority_Encoder(out,in);
 input [7:0]in;
 output reg[2:0]out;
 always @(in)
    begin
      if(in[7]==1) out=3'b111;
      else if(in[6]==1) out=3'b110;
      else if(in[5]==1) out=3'b101;
      else if(in[4]==1) out=3'b100;
      else if(in[3]==1) out=3'b011;
      else if(in[2]==1) out=3'b010;
      else if(in[1]==1) out=3'b001;
      else out=3'b000;
 end
endmodule
```

```
`timescale 1ns / 1ps
module Tb_Priority_Encoder;
reg [7:0]in;
wire [2:0]out;

Priority_Encoder dut(out,in);
initial begin
   repeat(10) begin
      in = $random;
      #10;
    end
end
initial begin
$monitor("in: %b  out: %b",in, out);
    #100 $finish;
    end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION WAVEFORM :



3 .SEQUENTIAL LOGIC DESIGN

- Output of sequential circuit depend on present input as well as past output.
- Sequential circuit is an example of closed loopsystem.
- Sequential circuit consist of memory element and combinational circuit

BLOCK DIAGRAM :

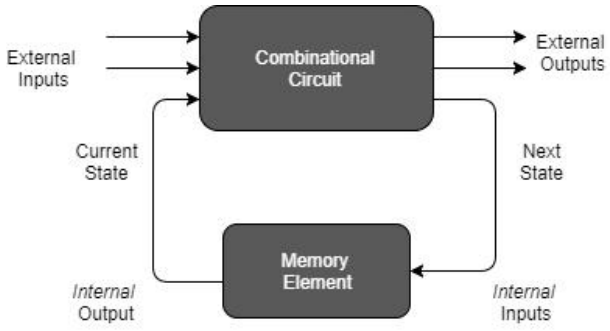


Figure: Sequential Circuit

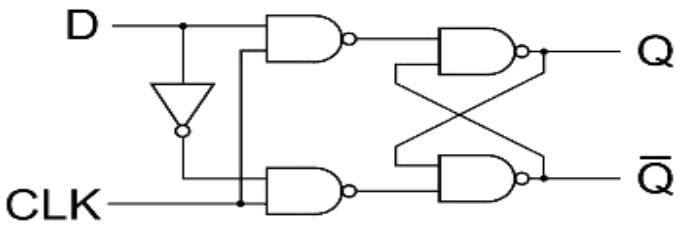
3.1CONSTRUCTION OF FLIP-FLOPS :

Flip-flop is also called bistable multivibrator or binary.it can store 1 bit of information

D-FLIP FLOP :

D Flip flop used to data Transmission.
It is reffered as transparent latch and its also reffered as data transmission flip flop.
D flip-flop is a better alternative that is very popular with digital electronics. They are commonly used for counters and shift registers and input synchronization.
D flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, D latch operates with enable signal. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal

CIRCUIT DIAGRAM D-FLIP FLOP :



In the D flip-flops, the output can only be changed at the clock edge, and if the input changes at other times, the output will be unaffected.

TRUTH TABLE FOR D-FLIP FLOP :

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

CHARACTERISTICS EQUATION FOR D FLIP FLOP :

D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as,

$$Qn+1 = D$$

APPLICATIONS :

Some of the applications of D flip flop in real-world includes:

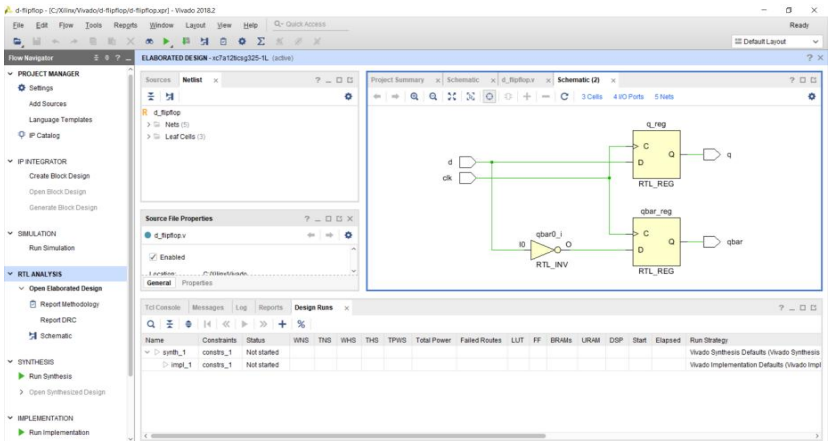
- **Shift registers:** D flip-flops can be cascaded together to create shift registers, which are used to store and shift data in digital systems. Shift registers are commonly used in serial communication protocols such as UART, SPI, and I2C.
- **State machines:** It can be used to implement state machines, which are used in digital systems to control sequences of events. State machines are commonly used in control systems, automotive applications, a nd industrial automation.
- **Counters:** It can be used in conjunction with other digital logic gates to create binary counters that can count up or down depending on the design. This makes them useful in real-time applications such as timers and clocks.
- **Data storage:** D flip-flops can be used to store temporary data in digital systems. They are often used in conjunction with other memory elements to create more complex storage systems.

VERILOG RTL CODE AND TESTBENCH CODE FOR D FLIP FLOP :

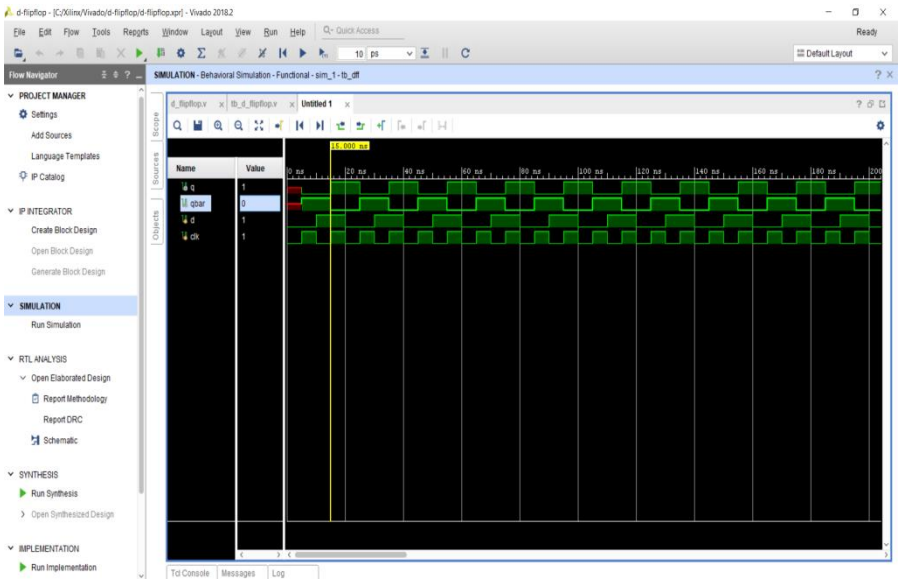
```
`timescale 1ns / 1ps

module d_flipflop(q,qbar,d,clk);
output reg q;
output reg qbar;
input d,clk;

always @(posedge clk)
begin
q <= d;
qbar <= ~d;
end
endmodule
```

```
`timescale 1ns / 1ps

module testbench_dff();
wire q;
wire qbar;
reg d,clk;

d_flipflop dut(.q(q), .qbar(qbar), .d(d), .clk(clk));
initial begin
clk=1'b0;
d=1'b0;

end

always #5 clk=~clk;
always #10 d=~d;

endmodule
```

RTL IMPLEMENTATION :



SIMULATION RESULT :

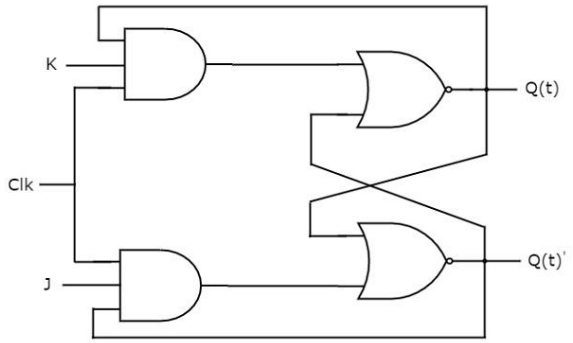


3.1.3 JK FLIP-FLOP

Due to the undefined state in the SR flip-flops, another flip-flop is required in electronics.

The JK flip-flop is an improvement on the SR flip-flop where S=R=1 is not a problem.

CIRCUIT DIAGRAM :



JK FLIP-FLOP TRUTH TABLE :

J	K	Q(n+1)	State	
0	0	Qn	No Change	
0	1	0	RESET	
1	0	1	SET	
1	1	Qn’	TOGGLE	

CHARACTERISTICS EQUATION FOR JK FLIP FLOP :

$$QN+1 = JQ'N + K'QN$$

APPLICATIONS :

Some of the applications of JK flip-flop in real-world includes:

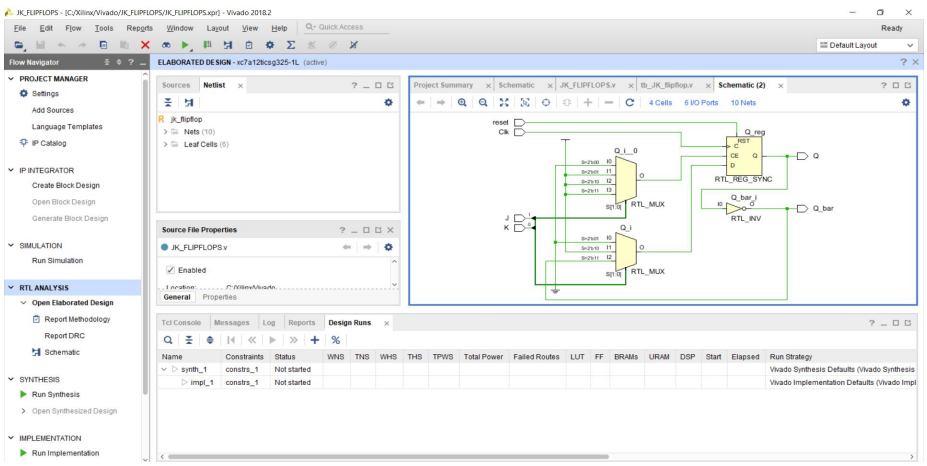
- **Counters:** The JK flip-flop can be used in conjunction with other digital logic gates to create a binary counter. This makes it useful in real-time applications such as timers and clocks.
- **Data storage:** The JK flip-flop can be used to store temporary data in digital systems.
- **Synchronization:** The JK flip-flop can be used to synchronize data signals between two digital circuits, ensuring that they are operating on the same clock cycle. This makes it useful in applications where timing is critical.

Frequency division: The JK flip-flop can be used to create a frequency divider, which is a circuit that divides the frequency of an input signal by a fixed amount. This makes it useful in real-time applications such as audio and video processing.

VERILOG RTL CODE AND TESTBENCH CODE FOR JK FLIP FLOP :

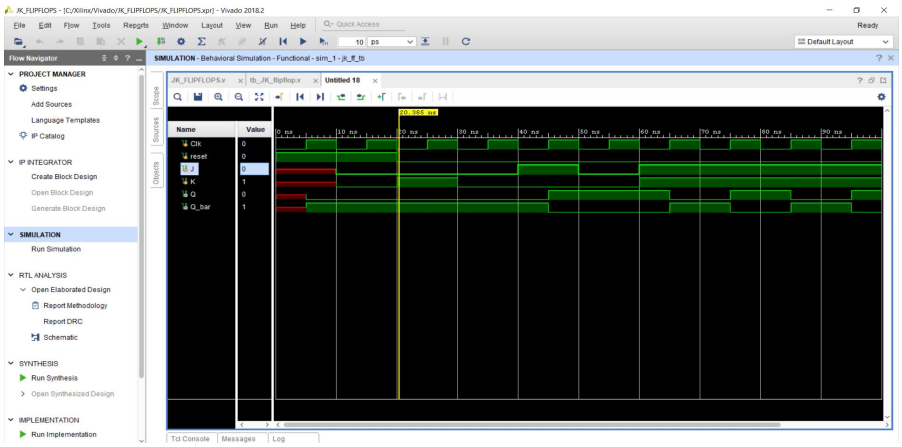
```
module jk_flipflop(Clk,reset,J,K,Q,Q_bar);
input J,K;
input Clk,reset;
output reg Q;
output Q_bar;
always@(posedge Clk)
begin
        if({reset})
            Q <= 1'b0;
        else
        begin
         case({J,K})
         2'b00: Q <= Q;
         2'b01: Q <= 1'b0;
          2'b10: Q <= 1'b1;
          2'b11: Q <= ~Q;
        endcase
        end

end
 assign Q_bar = ~Q;

endmodule
```

RTL IMPLEMENTATION :



```
// Module Name: tb_JK_flipflop
module jk_ff_tb();
reg Clk,reset,J,K;
wire Q,Q_bar;

jk_flipflop dut(Clk,reset,J,K,Q,Q_bar);

initial begin
 Clk=1'b0;
   forever #5 Clk = ~Clk;
end
initial begin
 reset = 1;
 #10;
 J = 1'b0; K = 1'b0; #10; //Initial memory
 reset = 0;
 J = 1'b0; K = 1'b1; #10; //Reset State
 J = 1'b0; K = 1'b0; #10; //Memory State

 J = 1'b1; K = 1'b0; #10; //Set State
 J = 1'b0; K = 1'b0; #10; //Memory State

 J = 1'b1; K = 1'b1; #10; //Toggle State
end
initial begin
$monitor("\t Clk = %d, J=%d,K=%d,Q=%d",Clk,J,K,Q);
#100 $finish;
end
endmodule
```

SIMULATION WAVEFORM :

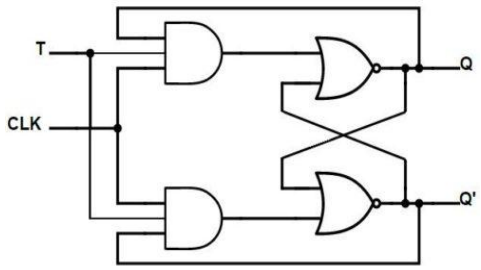


3.1.2 T FLIP-FLOP

T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions. These are basically single-input versions of JK flip-flops. This modified form of the JK is obtained by connecting inputs J and K together. It hasonly one input along with the clock input.

CIRCUIT DIAGRAM T-FLIP FLOP :

This circuit has single input T and two outputs Q & Q'. The operation of T flip-flop is same as that of JK flip-flop. Here, we considered the inputs of JK flip-flop as
J = T and
K = T in order to utilize the modified JK flip-flop for 2 combinations of inputs. So, we
eliminated the other two combinations of J & K, for which those two values are complement to each other in T flip-flop.



CHARACTERISTIC TABLE OF T-FLIP FLOP :

CLK	T	Q(n+1)	State
	0	Q	NO CHANGE
	1	Q'	TOGGLE

Inputs	Present State	Next State
T	**Q**	**Qt+1**
0	0	0
0	1	1
1	0	1
1	1	0

CHARACTERISTICS EQUATION FOR T FLIP FLOP:

$$QN+1 = Q'NT + QNT' = QN \text{ XOR } T$$

APPLICATIONS OF T FLIP FLOP :

The applications are:

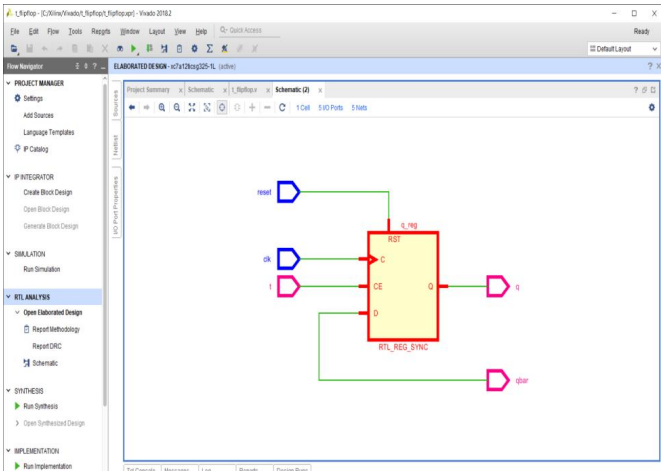
- Used as bounce elimination switch
- Used as frequency divider circuit that means divides the frequency of periodic wave forms
- Used as data storage where the Flip-Flops are connected in series in which each Flip-Flop stores one-bit information
- Used as digital counters, it counts pulses or events and it can be made by connecting series of Flip-Flops

VERILOG RTL CODE AND TESTBENCH CODE FOR T FLIP FLOP :

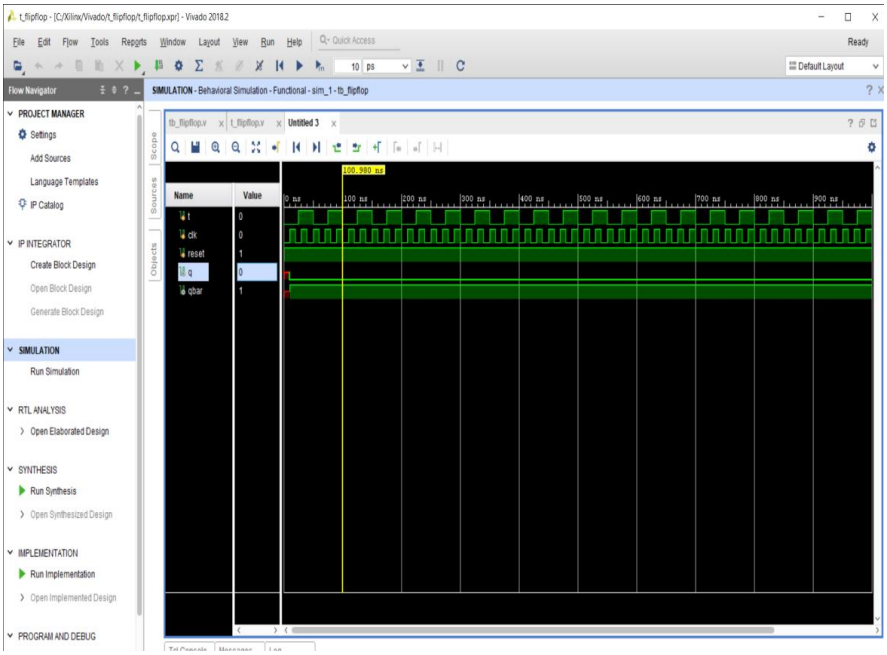
```
module t_flipflop(t,clk,reset,q,qbar);
input t,clk;
input reset;
output reg q;
output qbar;

always@(posedge clk) begin
if(reset==1) begin
q <= 1'b0;  //for initial condition
end
else
case(t)
     1'b0:q <= q;
     1'b1:q <= ~q;
endcase
end
assign qbar = ~q;
endmodule
```

```
`timescale 1ns / 1ps
module testbench_flipflop;

     reg t,clk,reset;
     wire q,qbar;

     t_flipflop dut(.t(t),.clk(clk),.reset(reset),.q(q),.qbar(qbar));
     initial
     begin
     t = 0;
     clk = 0;
     reset = 1;
     end

     always #10 clk = ~clk;
     always #25 t = ~t;

endmodule
```

RTL IMPLEMENTATION :



SIMULATION RESULT :



3.2 COUNTERS

Counters are basically used to count no. Of clock pulse applied.The counter is one of the widest applications of the flip flop. Based on the clock pulse, the output of the counter contains a predefined state. The number of the pulse can be counted using the output of the counter.

CLASSIFICATION OF COUNTERS :

Depending on the way in which the counting progresses, the synchronous or asynchronous counters are classified as follows −

- Up counters
- Down counters
- Up/Down counters

UP/DOWN COUNTER :

Up counter and down counter is combined together to obtain an UP/DOWN counter. A mode control (M) input is also provided to select either up or down mode. combinational circuit is required to be designed and used between each pair of flip-flop in order to achieve the up/down operation.

N_BIT UP/DOWN COUNTER USING PARAMETER :

An up/down counter is a digital counter which can be set to count either from 0 to max_value
or max_value to 0. The direction of the count(mode) is selected using a single bit input.

I

APPLICATION OF COUNTERS :

- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator.

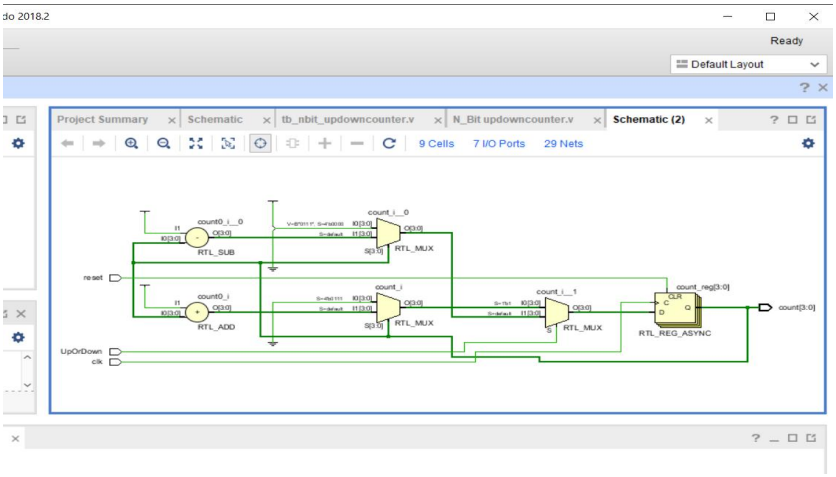
VERILOG CODE AND TESTBENCH CODE FOR N_BIT UP/DOWN COUNTER :

Verilog code for N bit up/down counter

When Up mode is selected, counter counts from 0 to 15 and then again from 0 to 15.
When Down mode is selected, counter counts from 15 to 0 and then again from 15 to 0.
Changing mode doesn't reset the Count value to zero.
You have apply high value to reset, to reset the Counter output.

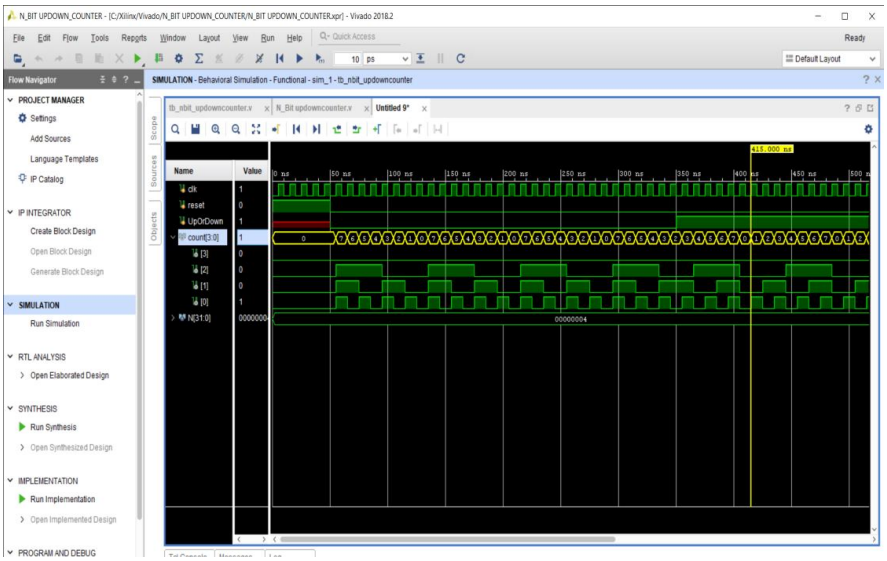
```
`timescale 1ns / 1ps
// Module Name: N_Bit updowncounter
module N_Bit_updowncounter(clk,reset ,UpOrDown,count);
    parameter N = 4;
   //input ports and their sizes
   input clk,reset,UpOrDown;
   //output ports and their size
   output reg[N-1 : 0] count;

    always @(posedge(clk) or posedge(reset))
    begin
      if(reset == 1)
         count <= 0;
      else
         if(UpOrDown == 1)   //Up mode selected
            if(count == 2*N-1)
               count <= 0;
            else
               count <= count + 1; //Incremend Counter
         else  //Down mode selected
            if(count == 0)
               count <= 2*N-1;
            else
               count <= count - 1; //Decrement counter
    end
endmodule
```

RTL IMPLEMENTATION :




```
module testbench_nbit_updowncounter;
 parameter N = 4;
   // Inputs
   reg clk;
   reg reset;
   reg UpOrDown;
   wire [N-1:0] count;
   // Instantiate the Unit Under Test (UUT)
   N_Bit_updowncounter uut (
      .clk(clk),
      .reset(reset),
      .UpOrDown(UpOrDown),
      .count(count) );
   initial clk = 0;
   always #5 clk = ~clk;
  initial begin
      reset = 1;
      #50 reset = 0;
      UpOrDown = 0;
      #300;
      UpOrDown = 1;
      #300;
      reset = 1;
      UpOrDown = 0;
      #100;
      reset = 0;
   end
initial begin
$monitor("\t\t UpOrDown =%b, Count = %b",UpOrDown,count);
#800 $finish;
end
endmodule
```

SIMULATION WAVEFORM FOR N-BIT UP/DOWNCOUNTER :



3.3 MODULUS COUNTER (MOD-N COUNTER)

The 2-bit ripple counter is called as MOD-4 counter and 3-bit ripple counter is called as MOD-8 counter. So in general, an n-bit ripple counter is called as modulo-N counter.

Where, MOD number = 2n.

MODULUS OF COUNTER :

Counter indicates the number of states through which counter passes during its operation.

- 2-Bit counter = Mod 4 counter.
- 3-BIT counter = Mod 8 counter.
- MOD-N Counter is also called as Modulo Counter.

TYPE OF MODULUS :

Therefore, a "Mod-N" counter will require the "N" number of flip-flops connected to count a single data bit while providing 2^n different output states (n is the number of bits). Note that N is always a whole integer value.

Then we can see that MOD counters have a modulus value that is an integral power of 2, that is, 2, 4, 8, 16 and so on to produce an n-bit counter depending on the number of flip-flops used, and how they are connected, determining the type and modulus of the counter.

- 2-bit up or down (MOD-4)
- 3-bit up or down (MOD-8)
- 4-bit up or down (MOD-16)

Application of counters :

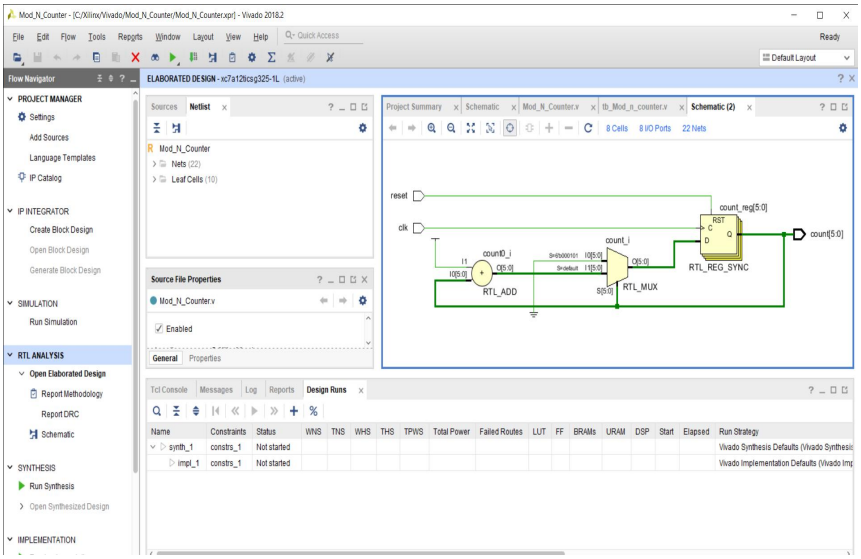
- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator.

VERILOG CODE AND TESTBENCH CODE FOR MOD-N COUNTER :

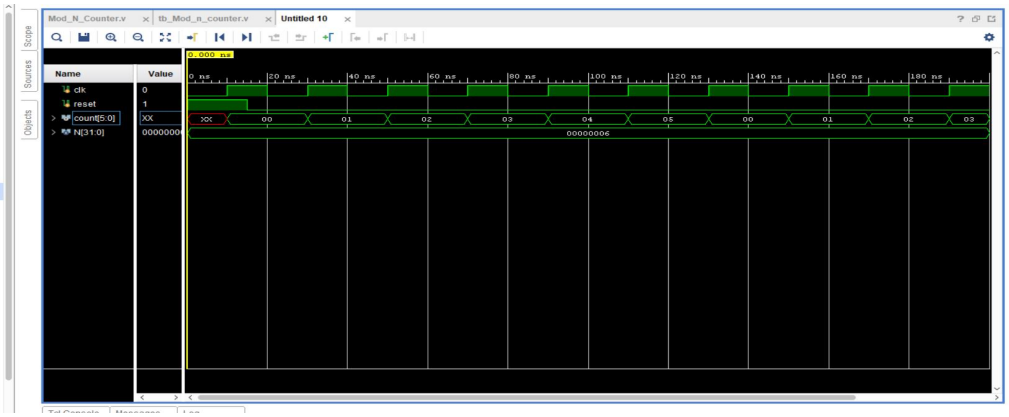
```
`timescale 1ns / 1ps
module Mod_N_Counter(clk,reset,count);
parameter N = 6;
input clk,reset;
output reg[N-1:0]count;

always @(posedge clk)
begin
    if(reset)
        count <= 0;
    else
        if(count == N-1)
           count <= 0;
        else
            count <= count+1;
end
endmodule
```

```
`timescale 1ns / 1ps
module testbench_Mod_n_counter;
parameter N = 6;
reg clk,reset;
wire [N-1:0]count;

Mod_N_Counter dut(clk,reset,count);
initial begin
clk =0;
forever #10 clk = ~clk;
end
initial begin
reset = 1;
#15;
reset = 0;
end
initial begin
$monitor("\t count= %b",count);
#200 $finish;
end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION WAVEFORM :



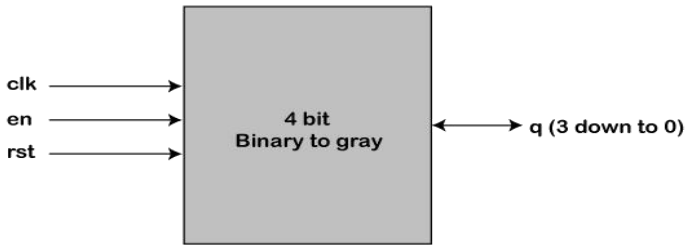
3.4 GRAY CODE COUNTER

Gray code is one kind of binary number system where only one bit will change at a time.

Today gray code is widely used in digital . This will helpful for error correction and

signal transmission. Gray counter is also useful in design and verification in VLSI domain.

BLOCK DIAGRAM GRAY CODE COUNTER :

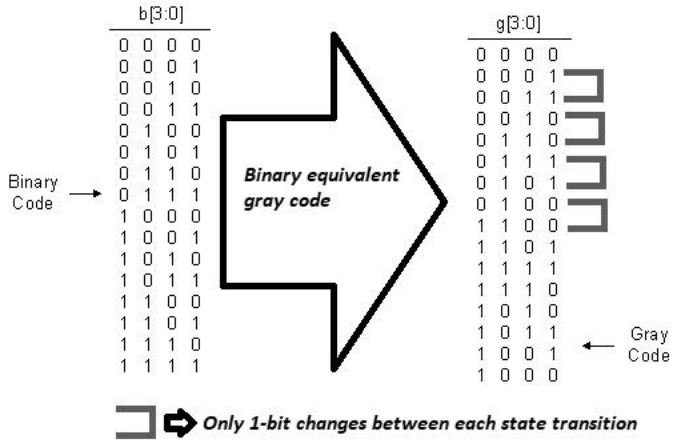


DESIGN FOR GRAY CODE COUNTER :

In a gray code, only one bit changes at one time. This design code has two inputs, clock and reset signals and one 4 bit output that will generate gray code.

First, if the reset signal is high, then the output will be zero, and as soon as reset goes low, on the rising edge of clk, the design will generate a four-bit gray code and continue to generate at every rising edge of clk signal.

This design code can be upgraded and put binary numbers as input, and this design will work as a binary to gray code converter.

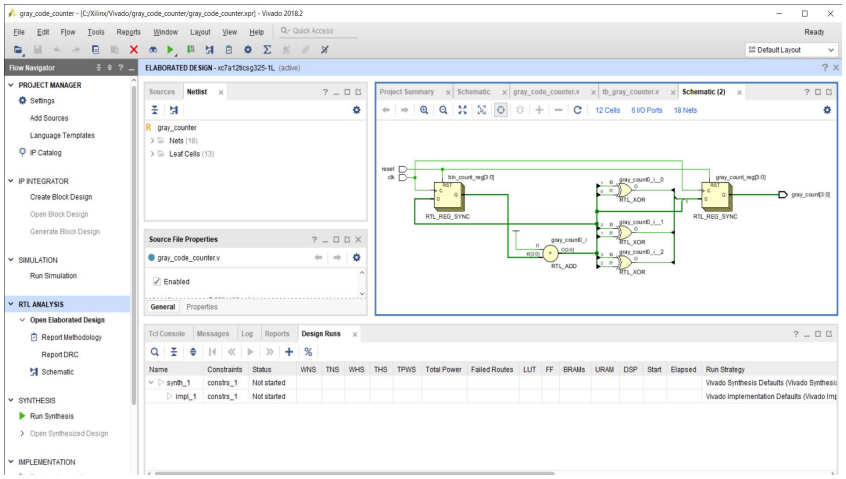


VERILOG CODE AND TESTBENCH CODE FOR GRAY CODE COUNTER :

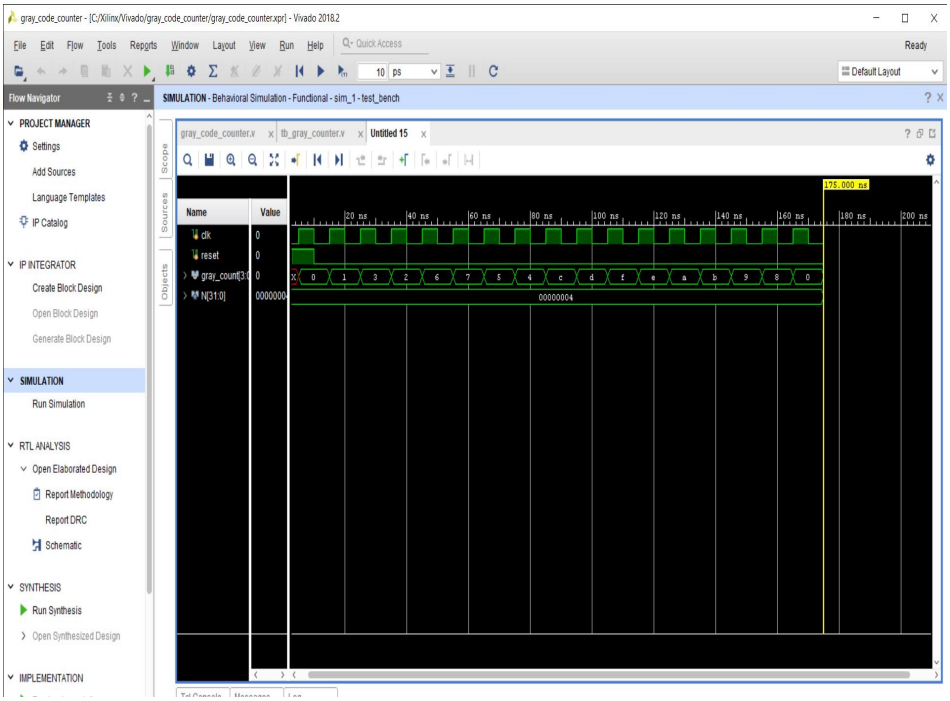
```
module gray_counter(clk,reset,gray_count);
parameter N = 4;
   input clk,reset;
   output reg [N-1:0] gray_count;
   reg [N-1:0] bin_count;

   always@(posedge clk)
   begin
      if(reset)
         begin
            gray_count=4'b0000;
            bin_count=4'b0000;
         end
      else
         begin
            bin_count = bin_count + 1;
            gray_count =
{bin_count[3],bin_count[3]^bin_count[2],bin_count[2]^bin_count[1],
bin_count[1]^bin_count[0]};
         end
   end
endmodule
```

```
module test_bench;
parameter N = 4;
reg clk,reset;
wire [N-1:0] gray_count;
gray_counter dut(clk, reset, gray_count);
initial begin
clk= 1'b0;
forever #5 clk= ~clk;
end
initial begin
reset= 1'b1;
#10;
reset= 1'b0;
end
initial begin
$monitor("\t\t counter: %d", gray_count);
#175 $finish;
end
endmodule
```

RTL IMPLEMENTATION:



SIMULATION WAVEFORM :



3.5 RING COUNTER

A ring counter is a special type of application of the **Serial IN Serial OUT** Shift register.

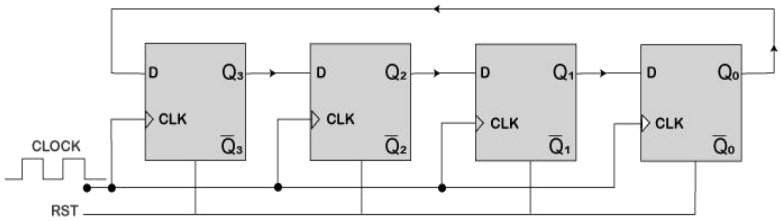
The only difference between the shift register and the ring counter is that the last flip flop outcome is taken as the output in the shift register.the output of the last flip-flop is connected to the input of the first flip-flop in the case of the ring counter but in the case of the shift register it is taken as output. Except for this, all the other things are the same.

When the circuit is reset, except one of the flipflop output,all others are made zero. For n-flipflop ring counter we have a MOD-n counter. That means the counter has n different states.

No.of states in Ring counter = No.of flip-flop used

CIRCUIT DIAGRAM FOR RING COUNTER :

At the time of reset the value of the counter is initialized to, say, 0001. It then becomes 0010 at the next clock cycle - and this keeps going on. Basically there is one bit that keeps shifting to left 1 bit at each clock cycle and then it rolls over when it reaches MSB.



TRUTH TABLE OF RING COUNTER :

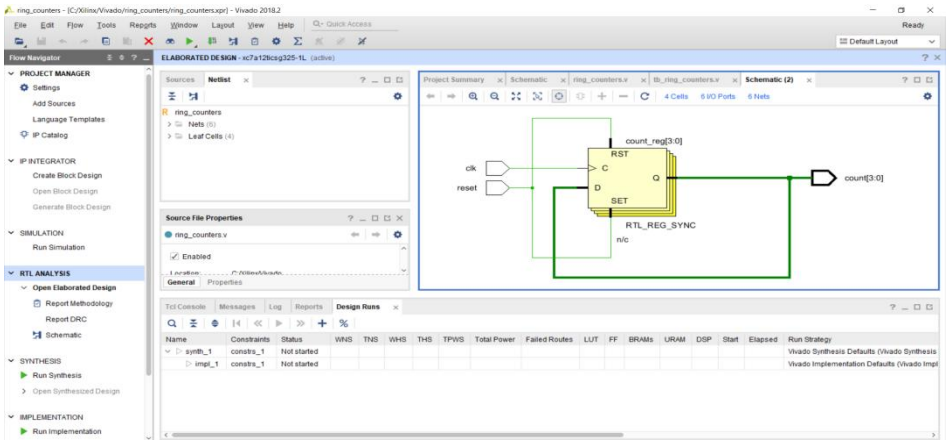
Q_0	Q_1	Q_2	Q_3
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

VERILOG CODE AND TESTBENCH CODE FOR RING COUNTER :

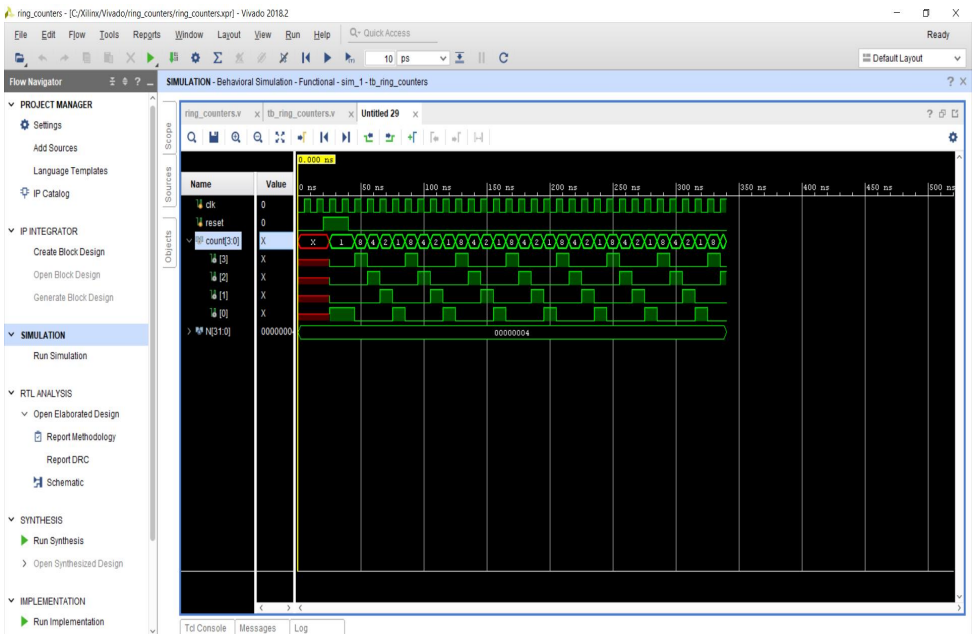
```
`timescale 1ns / 1ps
module tb_ring_counters();
parameter N = 4;
reg clk,reset;
wire [N-1:0]count;

ring_counters dut(clk,reset,count);

always #5 clk = ~clk;

initial begin
clk = 0;
reset = 0;
#20 reset = 1;
#20reset = 0;
#300 $finish;
end
endmodule
```

```
`timescale 1ns / 1ps
module ring_counters(clk,reset,count);
parameter N = 4;
input clk,reset;
output reg [N-1:0]count;

always@(posedge clk)
begin
if(reset)
        count <= 4'b0001;
    else
        count <= {count[0],count[N-1:1]};

end
endmodule
```

RTL IMPLEMENTATION RING COUNTER :



SIMULATION WAVEFORM FOR RING COUNTER :



3.5 TWISTED RING COUNTER

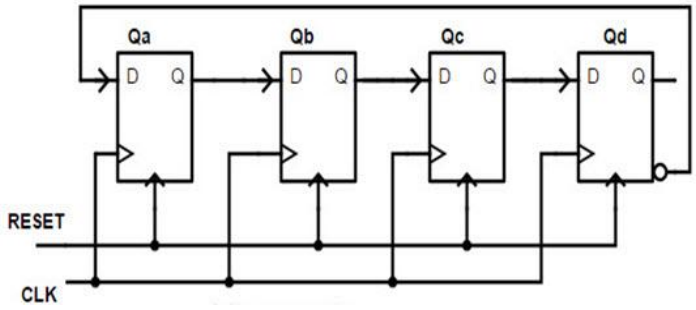
It is also known as a switch-tail ring counter, or Johnson counter.

The Johnson counter is a modification of ring counter.In this the complemented output of the
last stage flip flop is connected to the input of first flip flop. If we use n flip flops to design the Johnson counter, it is known as 2n bit Johnson counter or Mod 2n Johnson Counter.

The main difference between the 4 bit ring counter and the Johnson counter is that,in ring counter , we connect the output of last flip flop directly to the input of first flip flop. But in johnson counter, we connect the inverted output of last stage to the first stage input.

The Johnson counter is also known as Twisted Ring Counter, with a feedback. In Johnson counter the input of the first flip flop is connected from the inverted output of the last flip flop.

CIRCUIT DIAGRAM JOHNSON COUNTER :



TRUTH TABLE :

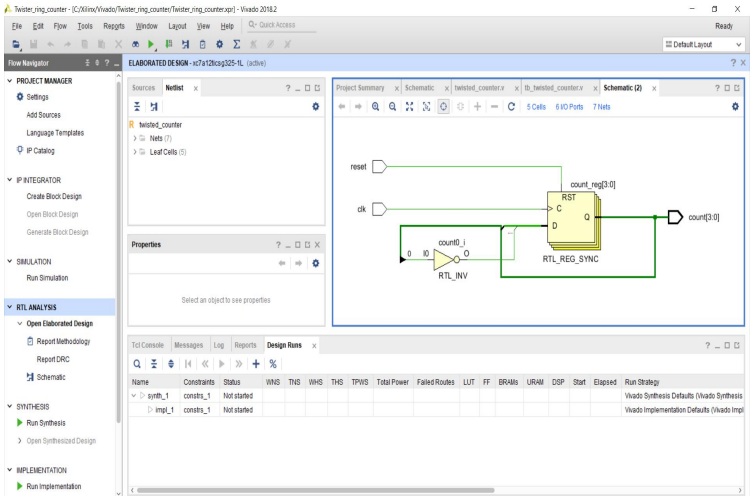
Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

VERILOG CODE AND TESTBENCH CODE FOR TWISTED COUNTER :

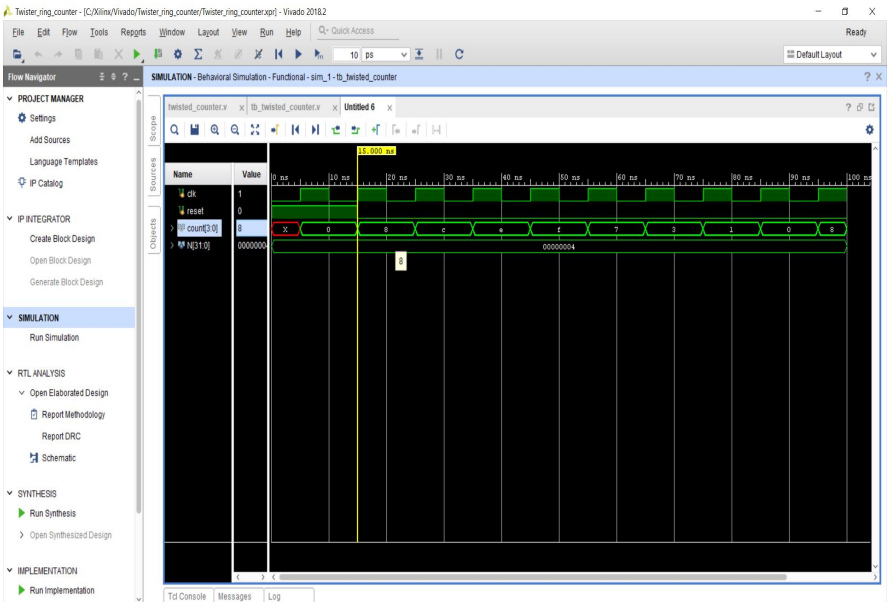
```
module twisted_counter(clk,reset,count);
parameter N = 4;
input clk,reset;
output reg[N-1:0]count;

always@(posedge clk)
begin
      if(reset)
            count <= 0;
       else
            count <= {~count[0],count[N-1:1]};
end
endmodule
```

```
`timescale 1ns / 1ps
// Module Name: tb_twisted_counter
module tb_twisted_counter;
parameter N = 4;
reg clk,reset;
wire [N-1:0]count;

twisted_counter dut(clk,reset,count);

initial begin
clk =1'b0;
forever #5 clk = ~clk;
end

initial begin
reset =1'b1;
#15;
reset = 1'b0;
end
initial begin
$monitor("\t count = %d",count);
#100 $finish;
end
endmodule
```

RTL IMPLEMENTATION :



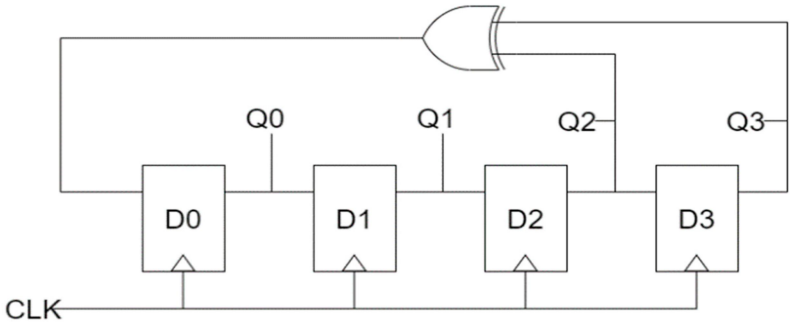
SIMULATION WAVEFORM :



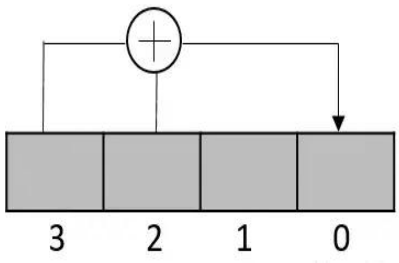
3.6 LINEAR FEEDBACK SHIFT REGISTER (LFSR)

Linear Feedback Shift Register As we have already defined an LFSR is a shift register whose input bit is a linear function of the previous bit. Therefore linear operation of single bit is exclusive-or (X-OR) operation is used. The initial value in the LFSR is called seed. Thus by changing the value of seed, the sequence at the output is also change. As register having a finite number of states, it may enter a repeating cycle. Thus LFSR having properly chosen feedback function can produce sequence of random patterns at the output of a repeating cycle. This feedback function is called a maximum length feedback polynomial.

LSFRs are used for digital counters, cryptography and circuit testing.



CIRCUIT DIAGRAM 4-BIT PSEUDO-RANDOM SEQUENCE GENERATOR :



At every step,

- Q[3] xor Q[2]
- Q = Q << 1

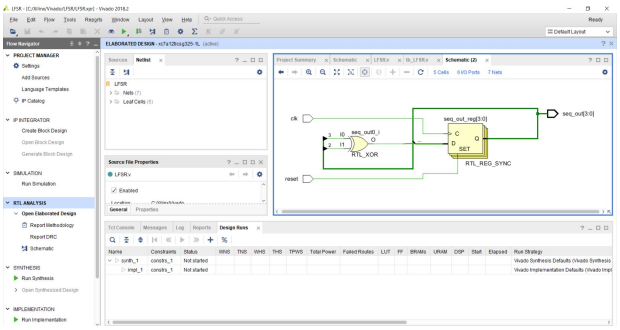
The result of the XOR operation is fed to the LSB (0th bit).

VERILOG CODE AND TESTBENCH CODE FOR LFSR :

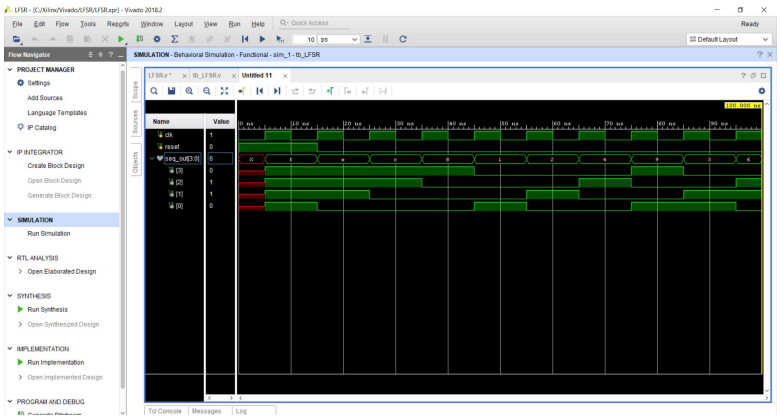
```
module LFSR(clk,reset,seq_out);
input clk,reset;
output reg [3:0] seq_out;

always@(posedge clk)begin
   if(reset)
      seq_out <= 4'hf;
    else
        seq_out <= {seq_out[2:0], seq_out[3]^seq_out[2]};

end
endmodule
```

```
module tb_LFSR;
reg clk,reset;
wire [3:0] seq_out;

LFSR  dut(clk,reset,seq_out);

always #5 clk = ~clk;
initial begin
clk = 0;
reset = 1;
#15;
reset = 0;
end
initial begin
$monitor("\t clk = %b seq_out = %d",clk, seq_out);
#100 $finish;
end
endmodule
```

RTL IMPLEMENTATION :



SIMULATION WAVEFORM FOR LFSR :



TCL CONSOLE :

```
# }
# run 1000ns
      clk = 0 seq_out =  x
      clk = 1 seq_out = 15
      clk = 0 seq_out = 15
      clk = 1 seq_out = 14
      clk = 0 seq_out = 14
      clk = 1 seq_out = 12
      clk = 0 seq_out = 12
      clk = 1 seq_out =  8
      clk = 0 seq_out =  8
      clk = 1 seq_out =  1
      clk = 0 seq_out =  1
      clk = 1 seq_out =  2
      clk = 0 seq_out =  2
      clk = 1 seq_out =  4
      clk = 0 seq_out =  4
      clk = 1 seq_out =  9
      clk = 0 seq_out =  9
      clk = 1 seq_out =  3
      clk = 0 seq_out =  3
      clk = 1 seq_out =  6
$finish called at time : 100 ns : File "C:/Xilinx/Vivado/tb_LFSR.v" Line 19
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_LFSR_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:07 . Memory (MB): peak = 1236.496 ; gain = 0.000
```