Lab Code: 20EC605

Digital Design Using Verilog HDL

Lab Manual

Department of Electronics & Communication Engineering

Bapatla Engineering College :: Bapatla

(Autonomous)

G.B.C. Road, Mahatmajipuram, Bapatla-522102, Guntur (Dist.)

Andhra Pradesh, India.

E-Mail:bec.principal@becbapatla.ac.in

Web:www.becbapatla.ac.in

Contents

Bapatla Engineering College :: Bapatla

(Autonomous)

Vision

- To build centers of excellence, impart high quality education and instill high standards of ethics and professionalism through strategic efforts of our dedicated staff, which allows the college to effectively adapt to the ever-changing aspects of education.

- To empower the faculty and students with the knowledge, skills and innovative thinking to facilitate discovery in numerous existing and yet to be discovered fields of engineering, technology and interdisciplinary endeavours.

Mission

- Our Mission is to impart the quality education at par with global standards to the students from all over India and in particular those from the local and rural areas.

- We continuously try to maintain high standards so as to make them technologically competent and ethically strong individuals who shall be able to improve the quality of life and economy of our country.

Bapatla Engineering College :: Bapatla

(Autonomous)

Department of Electronics and Communication Engineering

Vision

To produce globally competitive and socially responsible Electronics and Communication Engineering graduates to cater the ever-changing needs of the society.

Mission

- To provide quality education in the domain of Electronics and Communication Engineering with advanced pedagogical methods.
- To provide self-learning capabilities to enhance employability and entrepreneurial skills and to inculcate human values and ethics to make learners sensitive towards societal issues.
- To excel in the research and development activities related to Electronics and Communication Engineering.

Bapatla Engineering College :: Bapatla

(Autonomous)

Department of Electronics and Communication Engineering

Program Educational Objectives (PEO's)

PEO-I: Equip Graduates with a robust foundation in mathematics, science and Engineering Principles, enabling them to excel in research and higher education in Electronics and Communication Engineering and related fields.

PEO-II: Impart analytic and thinking skills in students to develop initiatives and innovative ideas for Start-ups, Industry and societal requirements.

PEO-III: Instill interpersonal skills, teamwork ability, communication skills, leadership, and a sense of social, ethical, and legal duties in order to promote lifelong learning and Professional growth of the students.

Program Outcomes (PO's)

Engineering Graduates will be able to:

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7.Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and Teamwork: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Life-long learning: Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Bapatla Engineering College :: Bapatla

(Autonomous)

Department of Electronics and Communication Engineering

Program Specific Outcomes (PSO's)

PSO1: Develop and implement modern Electronic Technologies using analytical methods to meet current as well as future industrial and societal needs.

PSO2: Analyze and develop VLSI, IoT and Embedded Systems for desired specifications to solve real world complex problems.

PSO3: Apply machine learning and deep learning techniques in communication and signal processing.

1.FULL ADDER

Aim: To design a Full Adder using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A **Full Adder** is a digital circuit that computes the sum of three binary bits. Unlike a half adder, which only adds two binary numbers, a full adder includes an additional input known as the "carry-in," allowing it to add three bits. The three inputs are:

- **A** - the first bit.
- **B** - the second bit.
- **Cin** - the carry-in from a previous addition.

The outputs of a full adder are:

- **Sum (S)**: The binary sum of the inputs.
- **Carry-out (Cout)**: The carry that gets passed on to the next stage of addition.

Boolean Expressions:

- **Sum (S)**:

  $S=A \oplus B \oplus Cin$

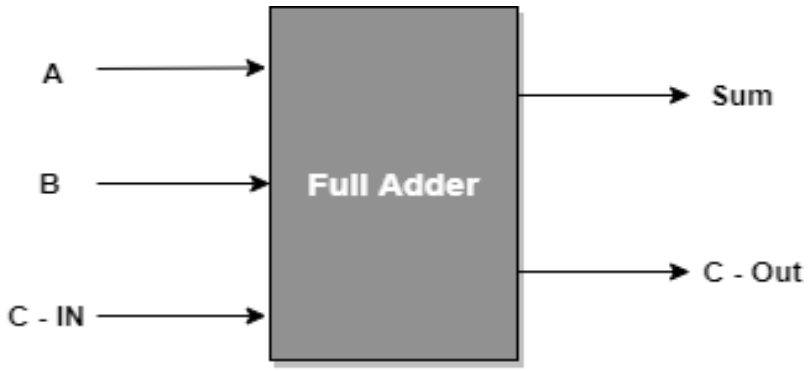
(Where $\oplus$ represents the XOR operation)

- **Carry-out (Cout)**:

  $Cout=(A \cdot B)+(Cin \cdot (A \oplus B))$

Full adders are essential in constructing arithmetic logic units (ALUs), where they can be cascaded together to add binary numbers of any length.

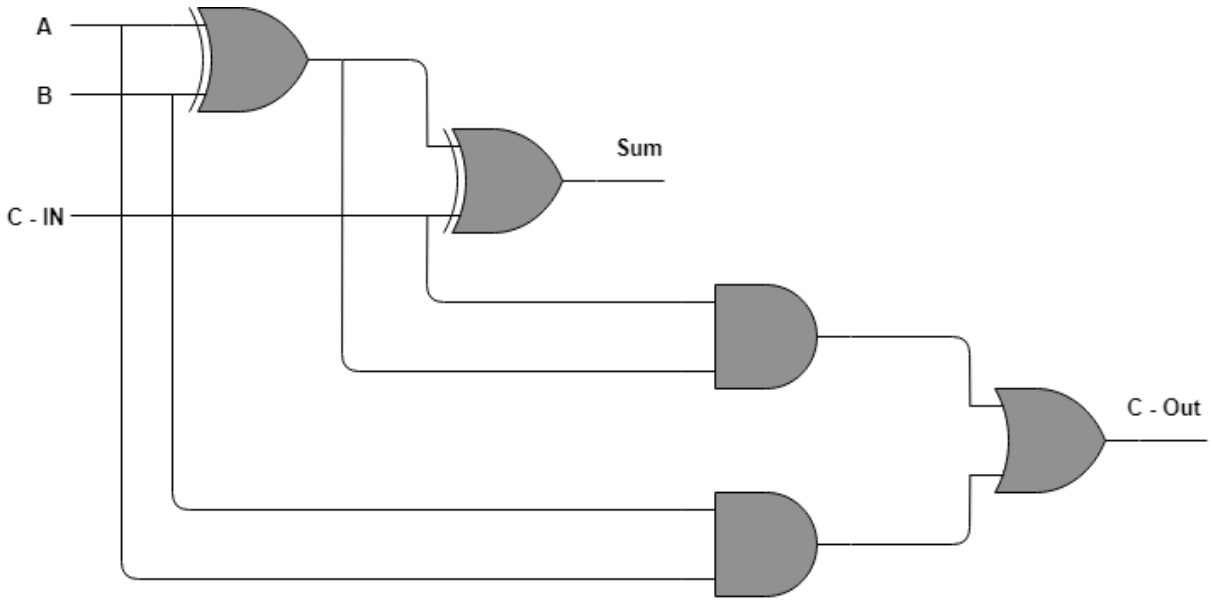
This forms the basis of ripple-carry adders, which are commonly used in digital circuits for binary addition.



Full Adder Truth Table:

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

Circuit Diagram:



Source Code:

Design Block:

```
module fa(a,b,cin,sum,cout);
        input a,b,cin;
        output sum,cout;
        assign {cout,sum}=a+b+cin;
endmodule
```

Test Bench:

```
module fa_tb();
reg a,b,cin;
wire sum,cout;
fa n1(a,b,cin,sum,cout);
        initial begin
                a = 0;b = 0;cin = 0;
                #10 a = 0;b = 0;cin = 1;
                #10 a = 0;b = 1;cin = 0;
                #10 a = 0;b = 1;cin = 1;
                #10 a = 1;b = 0;cin = 0;
                #10 a = 1;b = 0;cin = 1;
                #10 a = 1;b = 1;cin = 0;
                #10 a = 1;b = 1;cin = 1;
        end
endmodule
```

Result:

Full Adder using Verilog is designed and simulated successfully.

2. SUBTRACTORS

Aim: To design a Full Adder using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A **Full Subtractor** is a combinational circuit used to perform subtraction of three binary bits. The three inputs for a full subtractor are:

1. **A** - the minuend (the number from which another number is to be subtracted).
2. **B** - the subtrahend (the number that is to be subtracted).
3. **Bin** - the borrow-in from the previous subtraction.

The two outputs of a full subtractor are:

- **Difference (D)**: The result of the subtraction.
- **Borrow-out (Bout)**: The borrow generated for the next stage of subtraction.

Boolean Expressions:

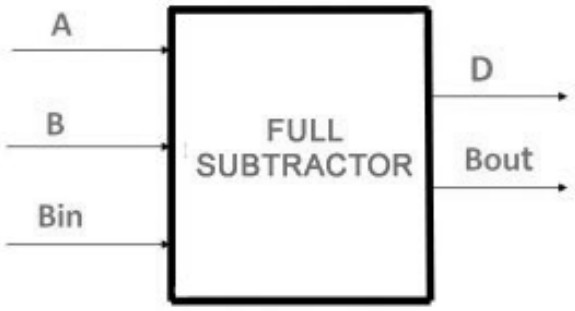
- **Difference (D)**:

$$D=A \oplus B \oplus Bin$$

Borrow-out (Bout):

$$Bout=(A \cdot B)+(B \cdot Bin)+(A \cdot Bin)$$

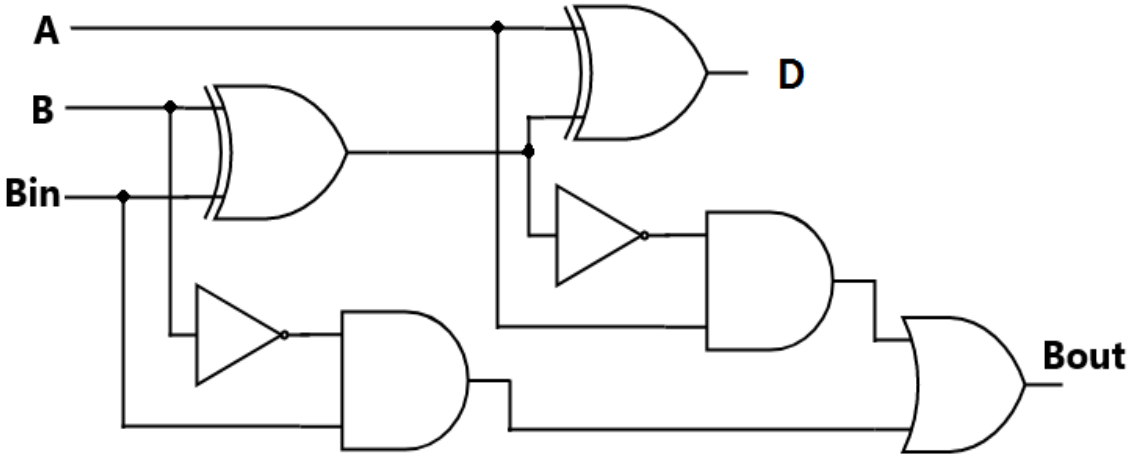
Full subtractors are important in digital circuits for arithmetic operations, particularly for subtracting multi-bit binary numbers. They are used in various computational units, such as in the design of arithmetic logic units (ALUs) and in binary subtraction operations.



Full Subtractor Truth Table:

INPUT			OUTPUT	
A	B	Bin	D	Bout
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

Circuit Diagram:



Source Code:

Design Block:

```
module Full_Subtractor_3(output D, B, input X, Y, Z);
```

```
assign D = X ^ Y ^ Z;
assign B = ~X & (Y^Z) | Y & Z;
endmodule
```

Test Bench:

```
module Full_Subtractor_3_tb;
wire D, B;
reg X, Y, Z;
Full_Subtractor_3 Instance0 (D, B, X, Y, Z);
initial begin
   X = 0; Y = 0; Z = 0;
#1  X = 0; Y = 0; Z = 1;
#1  X = 0; Y = 1; Z = 0;
#1  X = 0; Y = 1; Z = 1;
#1  X = 1; Y = 0; Z = 0;
#1  X = 1; Y = 0; Z = 1;
#1  X = 1; Y = 1; Z = 0;
#1  X = 1; Y = 1; Z = 1;
end
initial begin
   $monitor ("%t, X = %d| Y = %d| Z = %d| B = %d| D = %d", $time, X, Y, Z
, B, D);
   $dumpfile("dump.vcd");
   $dumpvars();
end
endmodule
```

Result:

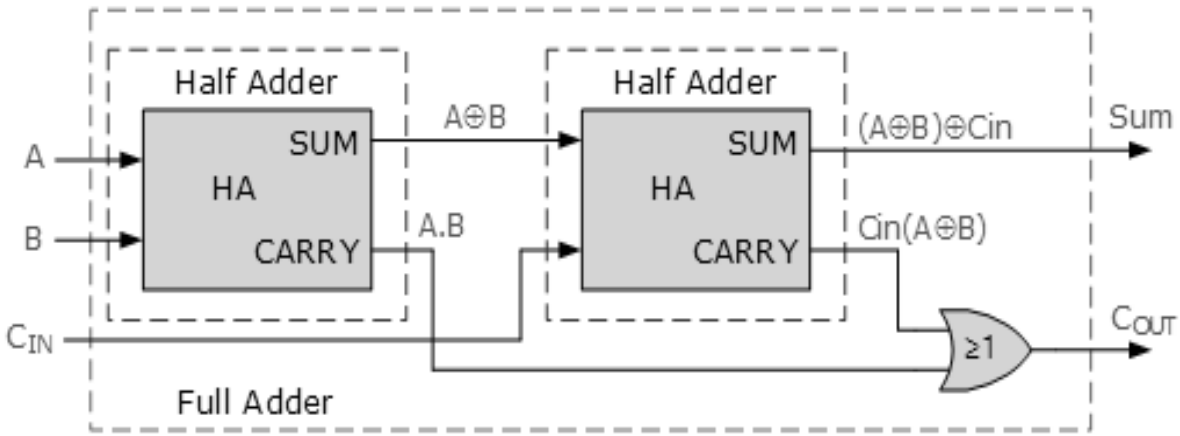
Full Subtractor using Verilog is designed and simulated successfully

3. Full adder using Half Adder

Aim: To design a Full Adder using half adder using Verilog HDL.

Software Used: Vivado 2016.4

Circuit Diagram:



Source Code:

Design Block:

```
module half_adder(a,b,sum,carry);
input a,b;
output sum,carry;
assign sum = a^b;
assign carry = a&b;
endmodule
```

The Verilog code for Full Adder using Half Adder and OR gate:

```
module full_adder(a_in, b_in, c_in, sum_out, carry_out);
input a_in,b_in,c_in;
 output sum_out,carry_out;
   //Declare the internal wires
   wire w1,w2,w3;
   //Instantiate the Half-Adders using port mapping
   half_adder HA1(.a(a_in),
                  .b(b_in),
```

```
                .sum(w1),
                .carry(w2));
  half_adder HA2(.a(w1),
                .b(c_in),
                .sum(sum_out),
                .carry(w3));
  //Instantiate the OR gate
  or or1(carry_out,w3,w2);
endmodule
```

Testbench:

```
module full_adder_tb();
    reg   a,b,cin;
  wire  sum,carry;
  integer i;
//   Instantiate the full adder with order based port mapping
  full_adder DUT(a,
                b,
                cin,
                sum,
                carry);
    initial
     begin
        a   = 1'b0;
        b   = 1'b0;
        cin = 1'b0;
     end

  initial
     begin
       for (i=0;i<8;i=i+1)
         begin
           {a,b,cin}=i;
```

```
        #10;
      end
    end
     //monitor the changes in the variables
  initial
    $monitor("Input a=%b, b=%b, c=%b, Output sum =%b,
carry=%b",a,b,cin,sum,carry);
     //terminate simulation after 100ns
  initial
#100 $finish;
  endmodule
```

Result:

Full Adder using half adder in Verilog is designed and simulated successfully

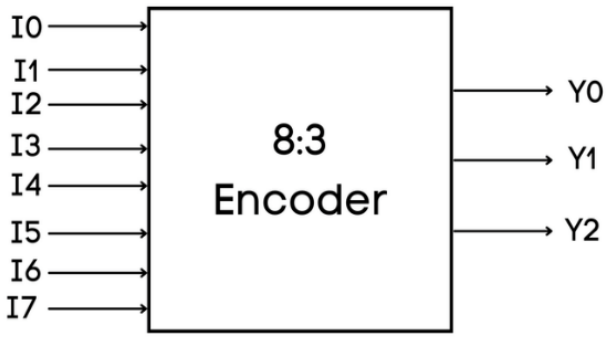
4. Priority Encoder

Aim: To design a Priority Encoder using Verilog HDL.

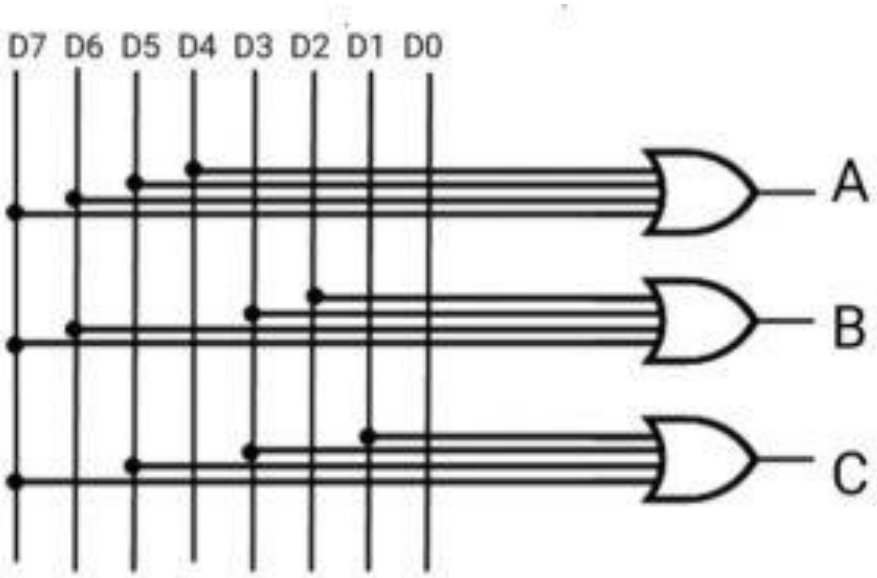
Software Used: Vivado 2016.4

Theory:

A Priority Encoder is a digital circuit which encodes the input signals according to the priority. It has several input lines; its output line shows the binary code of set active input line considered to be the highest priority. In some cases, several inputs can be active; in this case only the most important input is processed by the encoder, the other inputs are excluded. For instance, let's take a look at the 4-to-2 priority encoder: If the inputs I3 and I1 are active high, then the encoder will return the binary code for I3 since it has the highest priority. High-priority encoders are used in the cases when specific signals or tasks need to be performed with higher priority than the others.



Circuit Diagram:



The output expression are obtained as shown below,

A = D4+D5+D6+D7

B = D2+D3+D6+D7

C = D1+D3+D5+D7

Source Code:

Design Block:

```
module priorityencoder_83(en,i,y);
  // declare
  input en;
  input [7:0]i;
  // store and declare output values
  output reg [2:0]y;
  always @(en,i)
  begin
    if(en==1)
      begin
        // priority encoder
        // if condition to choose
        // output based on priority.
        if(i[7]==1) y=3'b111;
        else if(i[6]==1) y=3'b110;
        else if(i[5]==1) y=3'b101;
        else if(i[4]==1) y=3'b100;
        else if(i[3]==1) y=3'b011;
        else if(i[2]==1) y=3'b010;
        else if(i[1]==1) y=3'b001;
        else
        y=3'b000;
      end
     // if enable is zero, there is
     // an high impedance value.
    else y=3'bzzz;
```

```
  end
endmodule
```

Testbench:

```
module tb;
  reg [7:0]i;
  reg en;
  wire [2:0]y;

  // instantiate the model: creating
  // instance for block diagram
  priorityenoder_83 dut(en,i,y);
  initial
    begin
      // monitor is used to display the information.
      $monitor("en=%b i=%b y=%b",en,i,y);
      // since en and i are input values,
      // provide values to en and i.
      en=1; i=128;#5
      en=1; i=64;#5
      en=1; i=32;#5
      en=1; i=16;#5
      en=1; i=8;#5
      en=1; i=4;#5
      en=1; i=2;#5
      en=1; i=0;#5
      en=0;i=8'bx;#5
      $finish;
    end
endmodule
```

Result:

Priority Encoder in Verilog is designed and simulated successfully.

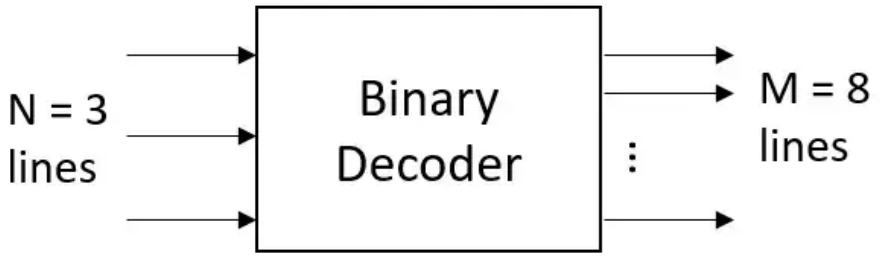
5. DECODER

Aim: To design a decoder using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A decoder is a combinational logic circuit that converts encoded inputs into a specific set of outputs. It essentially takes a binary input and generates an active output corresponding to the binary code. Decoders are widely used in applications where particular lines need to be selected or activated based on a binary input, such as memory address decoding, data routing, and instruction decoding. A decoder takes n binary inputs and generates 2^n outputs, with only one of these outputs being active at any given time, depending on the combination of the input bits. The output is typically binary, but can also be other forms, depending on the implementation. Inputs: n binary inputs. Outputs: 2^n outputs (one for each possible input combination)



Truth Table:

	Input	Output							
Decimal Number	Binary ($y_2y_1y_0$)	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	000	1	0	0	0	0	0	0	0
1	001	0	1	0	0	0	0	0	0
2	010	0	0	1	0	0	0	0	0
3	011	0	0	0	1	0	0	0	0
4	100	0	0	0	0	1	0	0	0
5	101	0	0	0	0	0	1	0	0
6	110	0	0	0	0	0	0	1	0
7	111	0	0	0	0	0	0	0	1

Source Code:

Design Block:

```
module binary_decoder(
```

```
  input [2:0] D,
  output reg [7:0] y);
    always@(D) begin
    y = 0;
    case(D)
      3'b000: y[0] = 1'b1;
      3'b001: y[1] = 1'b1;
      3'b010: y[2] = 1'b1;
      3'b011: y[3] = 1'b1;
      3'b100: y[4] = 1'b1;
      3'b101: y[5] = 1'b1;
      3'b110: y[6] = 1'b1;
      3'b111: y[7] = 1'b1;
      default: y = 0;
    endcase
  end
endmodule
```

Testbench:

```
module tb;
  reg [2:0] D;
  wire [7:0] y;
    binary_decoder bin_dec(D, y);
    initial begin
    $monitor("D = %b -> y = %0b", D, y);
    repeat(5) begin
      D=$random; #1;
    end
  end
endmodule
```

Result: Decoder in Verilog is designed and simulated successfully.

6. MULTIPLEXER

Aim: To design a multiplexer using Verilog HDL.

Software Used: Vivado 2016.4

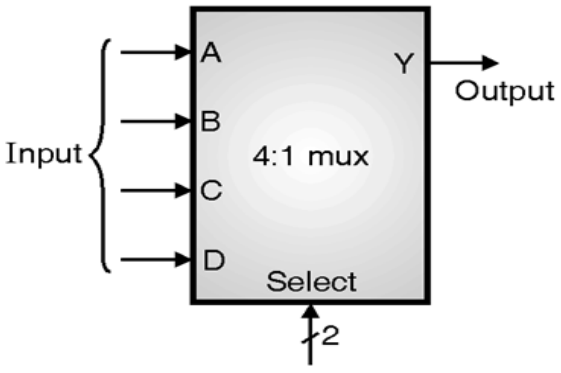
Theory:

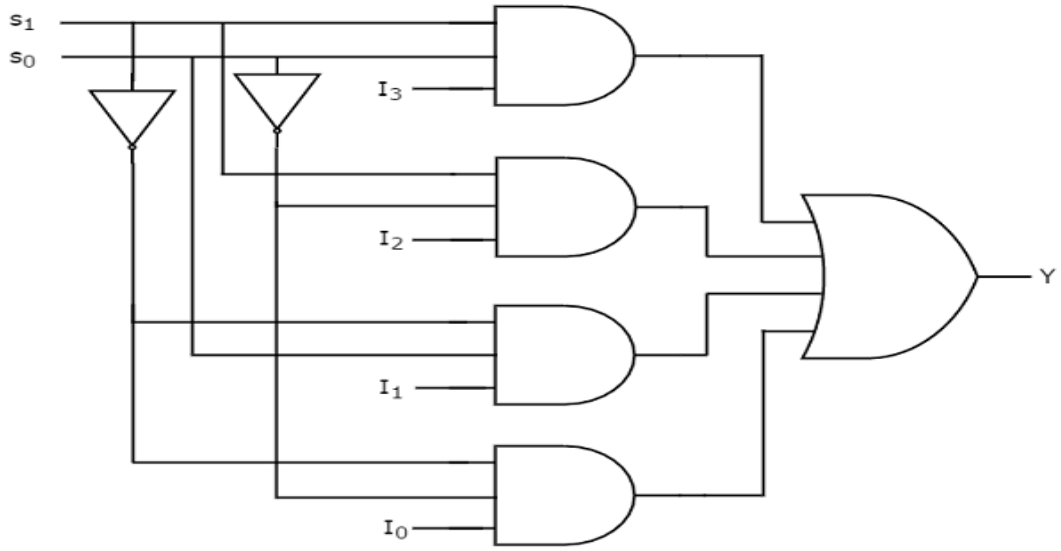
A multiplexer is a data selector device that selects one input from several input lines, depending upon the enabled, select lines, and yields one single output.A multiplexer of 2n inputs has n select lines, are used to select which input line to send to the output. There is only one output in the multiplexer, no matter what's its configuration.These devices are used extensively in the areas where the multiple data can be transferred over a single line like in the communication systems and bus architecture hardware. Visit this post for a crystal clear explanation to multiplexers.

Truth table:

Select		Output Y
S_1	S_0	
0	0	A
0	1	B
1	0	C
1	1	D

Circuit Diagram :





Source Code:

Design Block:

```
module m41 ( input a,
input b,
input c,
input d,
input s0, s1,
output out);
 assign out = s1 ? (s0 ? d : c) : (s0 ? b : a);
endmodule
```

Testbench:

```
module top;
wire  out;
reg  a;
reg  b;
reg  c;
reg  d;
reg s0, s1;
m41 name(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));
 initial
```

```
 begin
 a=1'b0; b=1'b0; c=1'b0; d=1'b0;
 s0=1'b0; s1=1'b0;
 #500 $finish;
end
always #40 a=~a;
always #20 b=~b;
always #10 c=~c;
always #5 d=~d;
always #80 s0=~s0;
always #160 s1=~s1;
always@(a or b or c or d or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out)
endmodule
```

Result: Multiplexer in Verilog is designed and simulated successfully.

6. MULTIPLIER

Aim: To design a multiplier using Verilog HDL.

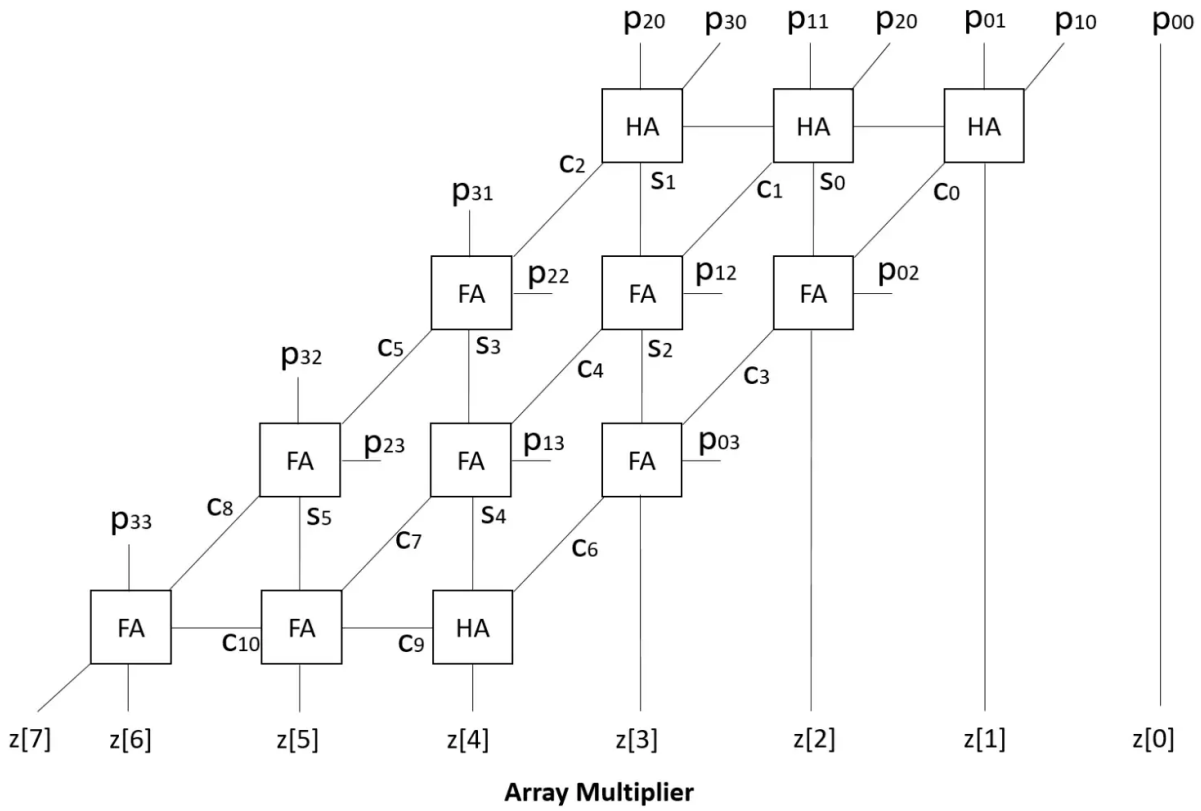
Software Used: Vivado 2016.4

Theory:

A multiplier is a digital combinational circuit designed to perform multiplication of two binary numbers. It takes two numbers (operands) as inputs and generates a product as output. Multipliers are fundamental components in many digital systems, including processors, signal processing units, and various arithmetic operations. There are several types of digital multipliers, with varying complexity and performance characteristics:

1. Array Multiplier
2. Wallace Tree Multiplier
3. Booth Multiplier
4. Sequential Multiplier

$$
\begin{array}{cccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & \\
 & & x & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & p_{30} & p_{20} & p_{10} & p_{00} & \\
 & & p_{31} & p_{21} & p_{11} & p_{01} & x & \\
 & p_{32} & p_{22} & p_{12} & p_{02} & x & x & \\
p_{33} & p_{23} & p_{13} & p_{03} & x & x & x & \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}
$$



Array Multiplier

Source Code:

Design Block:

```
module half_adder(input a, b, output s0, c0);
  assign s0 = a ^ b;
  assign c0 = a & b;
endmodule

module full_adder(input a, b, cin, output s0, c0);
  assign s0 = a ^ b ^ cin;
  assign c0 = (a & b) | (b & cin) | (a & cin);
endmodule

module array_multiplier(input [3:0] A, B, output [7:0] z);
  reg signed p[4][4];
  wire [10:0] c; // c represents carry of HA/FA
  wire [5:0] s;  // s represents sum of HA/FA
```

```
  // For ease and readability, two diffent name s and c are used instead of single wire name.
  genvar g;

  generate
    for(g = 0; g<4; g++) begin
      and a0(p[g][0], A[g], B[0]);
      and a1(p[g][1], A[g], B[1]);
      and a2(p[g][2], A[g], B[2]);
      and a3(p[g][3], A[g], B[3]);
    end
  endgenerate
  assign z[0] = p[0][0];

  //row 0
  half_adder h0(p[0][1], p[1][0], z[1], c[0]);
  half_adder h1(p[1][1], p[2][0], s[0], c[1]);
  half_adder h2(p[2][1], p[3][0], s[1], c[2]);

  //row1
  full_adder f0(p[0][2], c[0], s[0], z[2], c[3]);
  full_adder f1(p[1][2], c[1], s[1], s[2], c[4]);
  full_adder f2(p[2][2], c[2], p[3][1], s[3], c[5]);

  //row2
  full_adder f3(p[0][3], c[3], s[2], z[3], c[6]);
  full_adder f4(p[1][3], c[4], s[3], s[4], c[7]);
  full_adder f5(p[2][3], c[5], p[3][2], s[5], c[8]);

  //row3
  half_adder h3(c[6], s[4], z[4], c[9]);
  full_adder f6(c[9], c[7], s[5], z[5], c[10]);
  full_adder f7(c[10], c[8], p[3][3], z[6], z[7]);
```

```
endmodule
```

Testbench:

```
module TB;
  reg [3:0] A, B;
  wire [7:0] P;
    array_multiplier am(A,B,P);
    initial begin
    $monitor("A = %b: B = %b --> P = %b, P(dec) = %0d", A, B, P, P);
    A = 1; B = 0; #3;
    A = 7; B = 5; #3;
    A = 8; B = 9; #3;
    A = 4'hf; B = 4'hf;
  end
endmodule
```

Result: Multiplier in Verilog is designed and simulated successfully.

7. ARITHMETIC LOGIC UNIT

Aim: To design a Arithmetic Logic Unit using Verilog HDL.

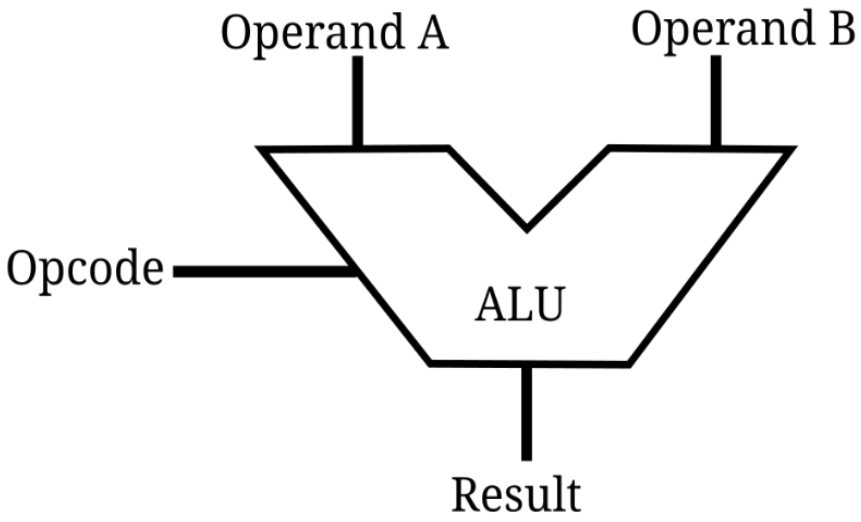
Software Used: Vivado 2016.4

Theory:

The Arithmetic Logic Unit (ALU) is a fundamental digital circuit in a computer's central processing unit (CPU) and is responsible for performing arithmetic and logical operations on binary data. It is a crucial component that determines the computing power of a processor.

Key Roles of the ALU:

1. Arithmetic Operations: Performs basic arithmetic like addition, subtraction, multiplication, and division.
2. Logical Operations: Performs logical operations like AND, OR, XOR, and NOT.
3. Shift Operations: Supports shift and rotate operations on binary data.
4. Comparison Operations: Compares numbers and generates flags (e.g., zero, carry, overflow, etc.) used in decision-making.



Source Code:

Design Block:

```
module alu(
        input [7:0] A,B,  // ALU 8-bit Inputs
        input [3:0] ALU_Sel,// ALU Selection
        output [7:0] ALU_Out, // ALU 8-bit Output
        output CarryOut // Carry Out Flag    );
   reg [7:0] ALU_Result;
   wire [8:0] tmp;
   assign ALU_Out = ALU_Result; // ALU out
   assign tmp = {1'b0,A} + {1'b0,B};
   assign CarryOut = tmp[8]; // Carryout flag
   always @(*)
   begin
      case(ALU_Sel)
      4'b0000: // Addition
        ALU_Result = A + B ;
      4'b0001: // Subtraction
        ALU_Result = A - B ;
      4'b0010: // Multiplication
        ALU_Result = A * B;
      4'b0011: // Division
        ALU_Result = A/B;
      4'b0100: // Logical shift left
        ALU_Result = A<<1;
       4'b0101: // Logical shift right
        ALU_Result = A>>1;
       4'b0110: // Rotate left
        ALU_Result = {A[6:0],A[7]};
       4'b0111: // Rotate right
        ALU_Result = {A[0],A[7:1]};
        4'b1000: //  Logical and
        ALU_Result = A & B;
```

```
        4'b1001: //  Logical or
         ALU_Result = A | B;
        4'b1010: //  Logical xor
         ALU_Result = A ^ B;
        4'b1011: //  Logical nor
         ALU_Result = ~(A | B);
        4'b1100: // Logical nand
         ALU_Result = ~(A & B);
        4'b1101: // Logical xnor
         ALU_Result = ~(A ^ B);
        4'b1110: // Greater comparison
         ALU_Result = (A>B)?8'd1:8'd0 ;
        4'b1111: // Equal comparison
          ALU_Result = (A==B)?8'd1:8'd0 ;
        default: ALU_Result = A + B ;
      endcase
  end
endmodule
```

Testbench:

```
module tb_alu;
//Inputs
 reg[7:0] A,B;
 reg[3:0] ALU_Sel;
//Outputs
 wire[7:0] ALU_Out;
 wire CarryOut;
  integer i;
 alu test_unit(
          A,B,  // ALU 8-bit Inputs
          ALU_Sel,// ALU Selection
          ALU_Out, // ALU 8-bit Output
```

```
        CarryOut // Carry Out Flag
    );
  initial begin
  // hold reset state for 100 ns.
    A = 8'h0A;
    B = 4'h02;
    ALU_Sel = 4'h0;

    for (i=0;i<=15;i=i+1)
    begin
     ALU_Sel = ALU_Sel + 8'h01;
     #10;
    end;

    A = 8'hF6;
    B = 8'h0A;

  end
endmodule
```

Result:

Arithmetic Logic Unit in Verilog is designed and simulated successfully.

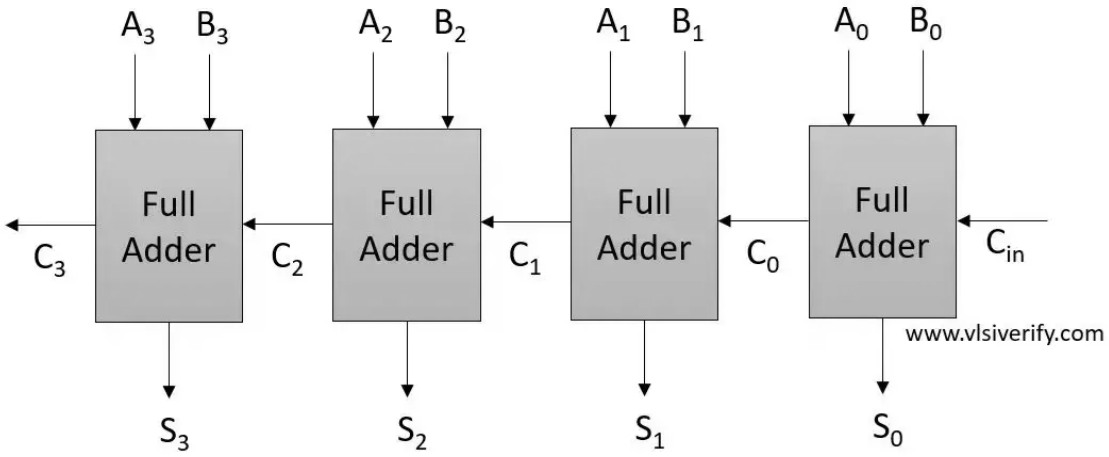
8.FAST ADDERS

Aim: To design a Fast Adders using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

a) Ripple Carry Adder



4-Bit Ripple Carry Adder

Source Code:

Design Block:

```
module full_adder(
  input a, b, cin,
  output sum, cout);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module ripple_carry_adder #(parameter SIZE = 4) (
  input [SIZE-1:0] A, B,
  input Cin,
  output [SIZE-1:0] S, Cout);
```

```
  genvar g;

  full_adder fa0(A[0], B[0], Cin, S[0], Cout[0]);
  generate  // This will instantiate full_adder SIZE-1 times
    for(g = 1; g<SIZE; g++) begin
      full_adder fa(A[g], B[g], Cout[g-1], S[g], Cout[g]);
    end
  endgenerate
endmodule
```

Testbench:

```
module RCA_TB;
  wire [3:0] S, Cout;
  reg [3:0] A, B;
  reg Cin;
  wire[4:0] add;

  ripple_carry_adder rca(A, B, Cin, S, Cout);
  assign add = {Cout[3], S};

  initial begin
    $monitor("A = %b: B = %b, Cin = %b --> S = %b, Cout[3] = %b, Addition =
%0d", A, B, Cin, S, Cout[3], add);
    A = 1; B = 0; Cin = 0; #3;
    A = 2; B = 4; Cin = 1; #3;
    A = 4'hb; B = 4'h6; Cin = 0; #3;
    A = 5; B = 3; Cin = 1; #3;
    $finish;
  end

  initial begin
```

```
    $dumpfile("waves.vcd");
    $dumpvars;
  end
endmodule
```

b) Carry Look Ahead Adder

Theory:

The drawback of 'Ripple carry adder' is that it has a carry propagation delay that introduces slow computation. Since adders are used in designs like multipliers and divisions, it causes slowness in their computation. To tackle this issue, a carry look-ahead adder (CLA) can be used that reduces propagation delay with additional hardware complexity. LA has introduced some functions like 'carry generate (G)' and 'carry propagate (P)' to boost the speed. Carry Generate (G): This function denotes how the carry is generated for single-bit two inputs regardless of any input carry. As we have seen in the full adder, carry is generated using the equation as A.B.

Hence, G = A·B (similar to how carry is generated by full adder) Carry Propagate (P): This function denotes when the carry is propagated to the next stage with an addition whenever there is an input carry. Let's consider single bit two inputs A and B.

A	B	Carry In	Description
0	0	1	Carry is not propagated (0)
0	1	1	Carry is propagated (1)
1	0	1	Carry is propagated (1)
1	1	1	Carry is propagated (1)

Thus, P = A+B

Carry computation function for next stage

$C_{j+1} = G_j + (P_j \cdot C_j)$

$= A_j \cdot B_j + (A_j + B_j) \cdot C_j$

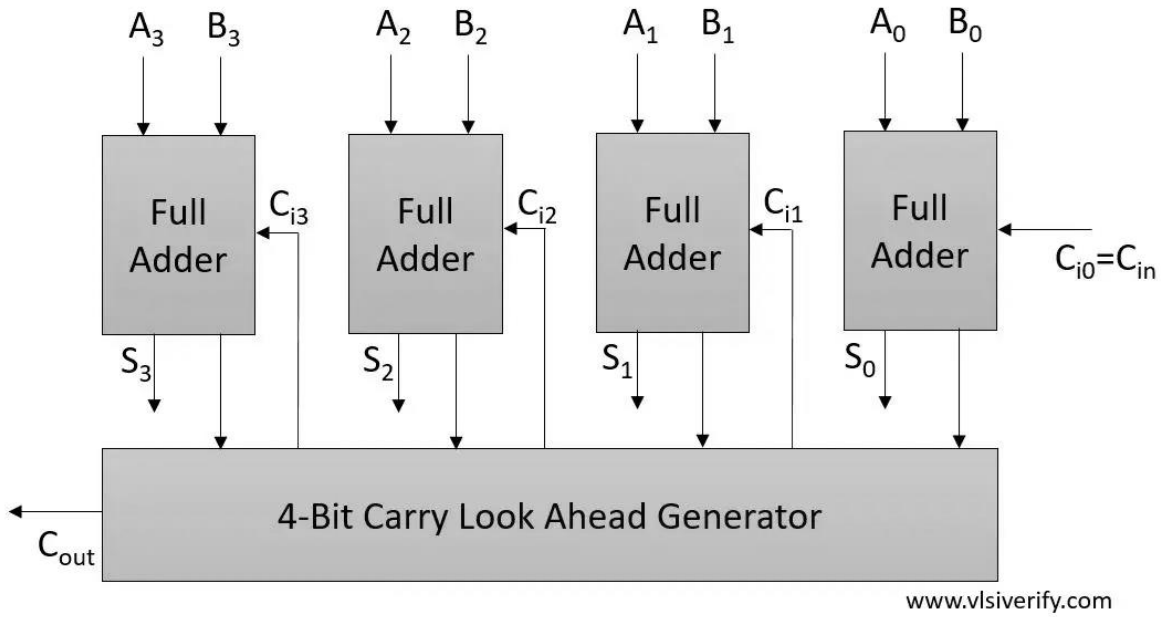
However, if you notice clearly whenever Aj=1 and Bj=1

Cj+1 = 1 (always) as Aj·Bj component nullifies effect of [(Aj + Bj)·Cj] part. Thus, it does not matter even if you use the equation propagate function as P = A (+) B as mentioned in other literature.

G = A·B

P = A + B or A (+) B

Block Diagram



4-Bit Carry Look Ahead Adder

Source Code:

Design Block:

```
module CarryLookAheadAdder(
  input [3:0]A, B,
  input Cin,
  output [3:0] S,
  output Cout);
  wire [3:0] Ci; // Carry intermediate for intermediate computation
    assign Ci[0] = Cin;
  assign Ci[1] = (A[0] & B[0]) | ((A[0]^B[0]) & Ci[0]);
  //assign Ci[2] = (A[1] & B[1]) | ((A[1]^B[1]) & Ci[1]); expands to
```

```
  assign Ci[2] = (A[1] & B[1]) | ((A[1]^B[1]) & ((A[0] & B[0]) | ((A[0]^B[0]) & Ci[0])));
  //assign Ci[3] = (A[2] & B[2]) | ((A[2]^B[2]) & Ci[2]); expands to
  assign Ci[3] = (A[2] & B[2]) | ((A[2]^B[2]) & ((A[1] & B[1]) | ((A[1]^B[1]) & ((A[0] & B[0]) | ((A[0]^B[0]) & Ci[0])))));
  //assign Cout  = (A[3] & B[3]) | ((A[3]^B[3]) & Ci[3]); expands to
  assign Cout  = (A[3] & B[3]) | ((A[3]^B[3]) & ((A[2] & B[2]) | ((A[2]^B[2]) & ((A[1] & B[1]) | ((A[1]^B[1]) & ((A[0] & B[0]) | ((A[0]^B[0]) & Ci[0])))))));

  assign S = A^B^Ci;
endmodule
```

Testbench:

```
module TB;
  reg [3:0]A, B;
  reg Cin;
  wire [3:0] S;
  wire Cout;
  wire[4:0] add;
    CarryLookAheadAdder cla(A, B, Cin, S, Cout);
    assign add = {Cout, S};
  initial begin
    $monitor("A = %b: B = %b, Cin = %b --> S = %b, Cout = %b, Addition = %0d", A, B, Cin, S, Cout, add);
    A = 1; B = 0; Cin = 0; #3;
    A = 2; B = 4; Cin = 1; #3;
    A = 4'hb; B = 4'h6; Cin = 0; #3;
    A = 5; B = 3; Cin = 1;
  end
endmodule
```

Result: Fast Adders in Verilog is designed and simulated successfully.

9.PARITY GENERATOR

Aim: To design a parity generator using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

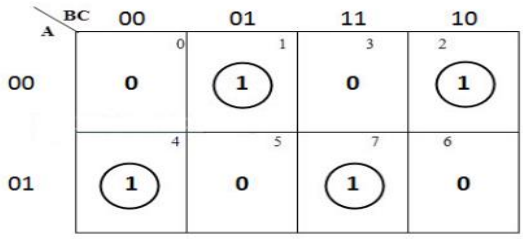
It is combinational circuit that accepts an n-1 bit stream data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is termed as a parity bit.In **even parity** bit scheme, the parity bit is '**0**' if there are **even number of 1s** in the data stream and the parity bit is '**1**' if there are **odd number of 1s** in the data stream.In **odd parity** bit scheme, the parity bit is '**1**' if there are **even number of 1**s in the data stream and the parity bit is '**0**' if there are **odd number of 1s** in the data stream. Let us discuss both even and odd parity generators.

Even Parity Generator

A 3-bit message is to be transmitted with an even parity bit. Let the three inputs A, B and C are applied to the circuits and output bit is the parity bit P. The total number of 1s must be even, to generate the even parity bit P.The figure below shows the truth table of even parity generator in which 1 is placed as parity bit in order to make all 1s as even when the number of 1s in the truth table is odd.

3-bit message			Even parity bit generator (P)
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The K-map simplification for 3-bit message even parity generator is



From the above truth table, the simplified expression of the parity bit can be written as

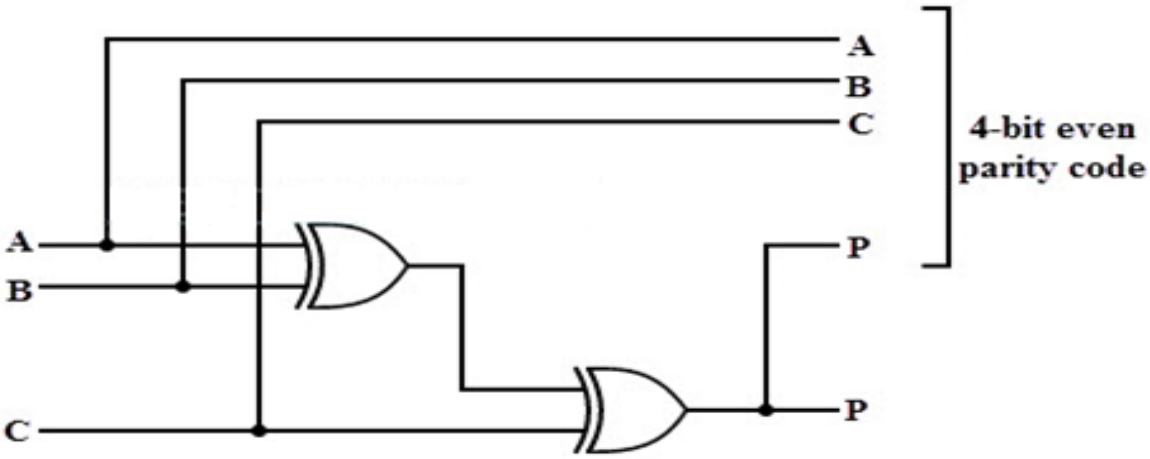
$$P = \overline{A}\ \overline{B}\ C + \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + A\ B\ C$$

$$= \overline{A}\ (\overline{B}\ C + B\ \overline{C}) + A\ (\overline{B}\ \overline{C} + B\ C)$$

$$= \overline{A}\ (B \oplus C) + A\ (\overline{B \oplus C})$$

$$P = A \oplus B \oplus C$$

The above expression can be implemented by using two Ex-OR gates. The logic diagram of even parity generator with two Ex – OR gates is shown below. The three bit message along with the parity generated by this circuit which is transmitted to the receiving end where parity checker circuit checks whether any error is present or not.To generate the even parity bit for a 4-bit data, three Ex-OR gates are required to add the 4-bits and their sum will be the parity bit.



Source Code:

Design Block:

```
module parity_generator(data_in , parity_out);
output  parity_out ;
input [3:0]data_in ;
assign parity_out = (^ data_in);
endmodule
```

Test Bench:

```
module parity_tb();
reg [3:0]data_in;
wire parity_out;
parity_generator n1(data_in,parity_out);
initial
begin
data_in=0000;
#10 data_in=0001;
#10 data_in=1010;
#10 data_in=1011;
#10 data_in=0100;
#10 data_in=0101;
#10 data_in=1110;
#10 data_in=1111;
#10 data_in=1000;
```

```
#10 data_in=1001;

#10 data_in=0010;

#10 data_in=0011;

#10 data_in=1100;

#10 data_in=1101;

#10 data_in=0110;

#10 data_in=0111;

end

endmodule
```

ODD Parity Generator:

Theory:

The 3-bit data is to be transmitted with an odd parity bit. The three inputs are A, B and C and P is the output parity bit. The total number of bits must be odd in order to generate the odd parity bit.In the given truth table below, 1 is placed in the parity bit in order to make the total number of bits odd when the total number of 1s in the truth table is even.

3-bit message			Odd parity bit generator (P)
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

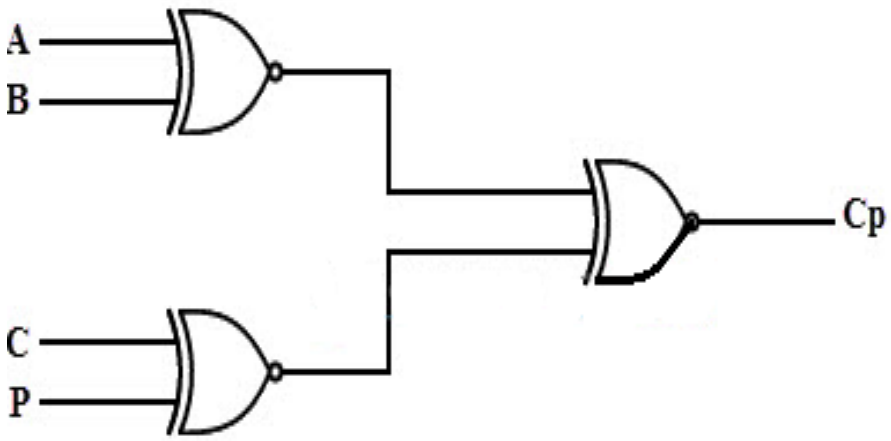
The truth table of the odd parity generator can be simplified by using K-map as

A \ BC	00	01	11	10
00	1 (0)	0 (1)	1 (3)	0 (2)
01	0 (4)	1 (5)	0 (7)	1 (6)

The output parity bit expression for this generator circuit is obtained as

$$P = A \oplus B \text{ Ex-NOR } C$$

The above Boolean expression can be implemented by using one Ex-OR gate and one Ex-NOR gate in order to design a 3-bit odd parity generator.The logic circuit of this generator is shown in below figure , in which . two inputs are applied at one Ex-OR gate, and this Ex-OR output and third input is applied to the Ex-NOR gate , to produce the odd parity bit. It is also possible to design this circuit by using two Ex-OR gates and one NOT gate.



Source Code:

Design Block:

```
module odd_parity(data_in , parity_out);
output  parity_out ;
input [3:0]data_in ;
assign parity_out = ~(^ data_in);
endmodule
```

Test Bench:

```
module odd_parity_tb();
reg [3:0]data_in;
wire parity_out;
odd_parity n1(data_in,parity_out);
initial
begin
data_in=0000;
#10 data_in=0001;
#10 data_in=1010;
#10 data_in=1011;
#10 data_in=0100;
#10 data_in=0101;
#10 data_in=1110;
#10 data_in=1111;
#10 data_in=1000;
#10 data_in=1001;
#10 data_in=0010;
#10 data_in=0011;
#10 data_in=1100;
#10 data_in=1101;
#10 data_in=0110;
#10 data_in=0111;
```

```
end
endmodule
```

Result: parity generator in Verilog is designed and simulated successfully.

10. COMPARATOR

Aim: To design a Fast Adders using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A magnitude digital Comparator is a combinational circuit that **compares two digital or binary numbers** to find out whether one binary number is equal, less than or greater than the other binary number. We logically design a circuit for which we will have two inputs one for A and other for B and have three output terminals, one for A > B condition, one for A = B condition and one for A < B condition.

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.

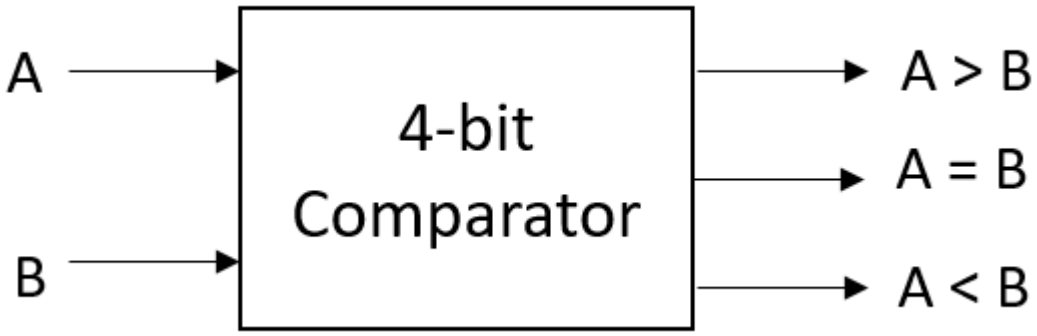
In a 4-bit comparator the condition of A>B can be possible in the following four cases:

1. If A3 = 1 and B3 = 0
2. If A3 = B3 and A2 = 1 and B2 = 0
3. If A3 = B3, A2 = B2 and A1 = 1 and B1 = 0
4. If A3 = B3, A2 = B2, A1 = B1 and A0 = 1 and B0 = 0

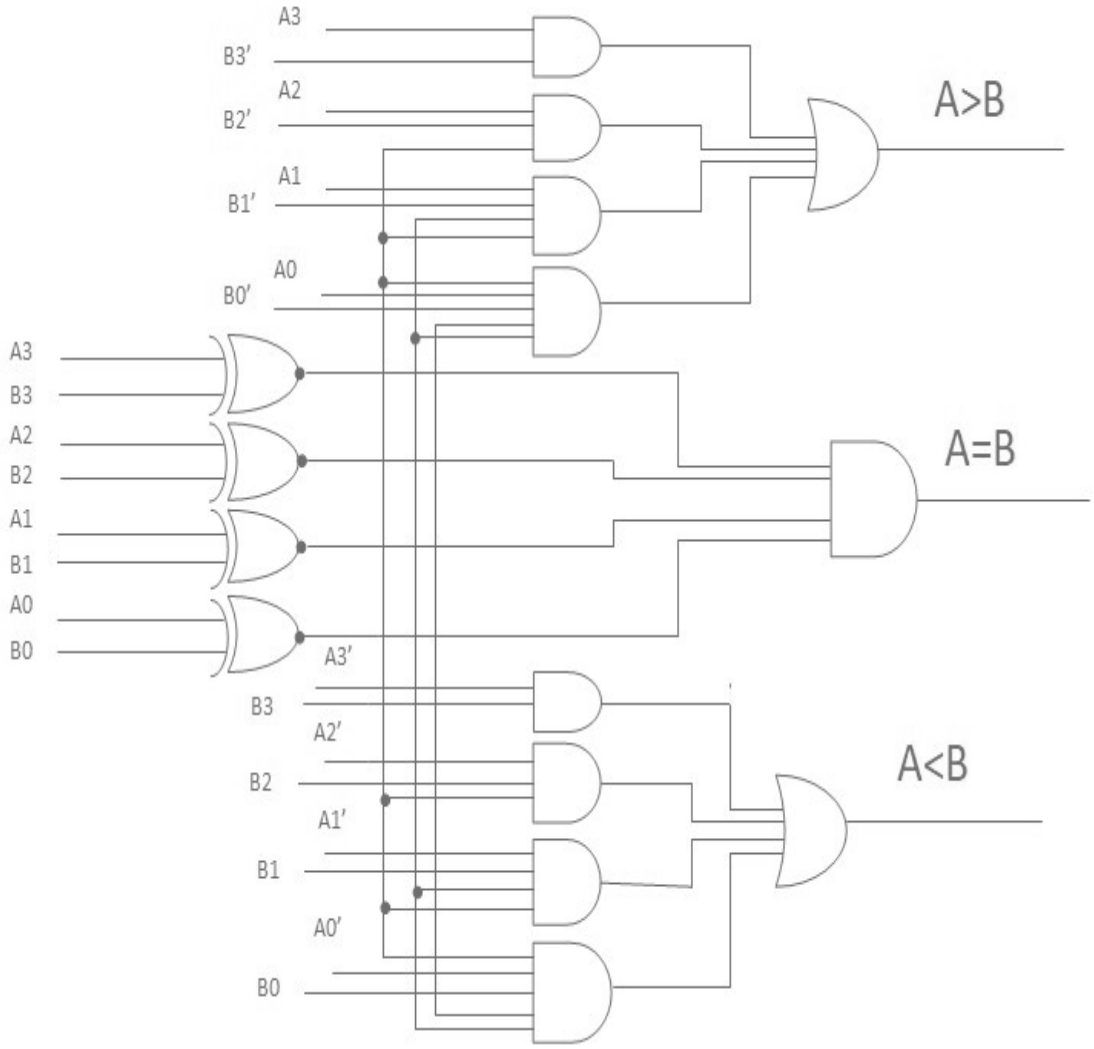
Similarly, the condition for A<B can be possible in the following four cases:

1. If A3 = 0 and B3 = 1
2. If A3 = B3 and A2 = 0 and B2 = 1
3. If A3 = B3, A2 = B2 and A1 = 0 and B1 = 1
4. If A3 = B3, A2 = B2, A1 = B1 and A0 = 0 and B0 = 1

The condition of A=B is possible only when all the individual bits of one number exactly coincide with corresponding bits of another number.



Circuit Diagram:



Source Code:

Design Block:

```
module comparator_4bit_bh(
  output reg EQ,
  output reg GT,
```

```
  output reg LT,
  input [3:0] a,
  input [3:0] b
);
  always @(*) begin
    LT = (a < b);
    EQ = (a == b);
    GT = (a > b);
  end
 endmodule
```

Testbench:

```
module comparator_tb;
  // Declare wires for outputs
  wire  EQ;
  wire  GT;
  wire  LT;
  // Declare registers for inputs
  reg [3:0] a;
  reg [3:0] b;
  // Instantiate the comparator module
  comparator_4bit_bh dut(EQ,GT,LT,a,b);

  initial begin
    // Test scenario for greater than
    a = 5;
    b = 3;
    #10; // Wait for 10 time units
    // Display the inputs and outputs
    $display("a = %d, b = %d, EQ = %b, GT = %b, LT = %b", a, b, EQ, GT, LT);

    // Test scenario for less than
    a = 3;
```

```
    b = 5;
    #10; // Wait for 10 time units
    // Display the inputs and outputs
    $display("a = %d, b = %d, EQ = %b, GT = %b, LT = %b", a, b, EQ, GT, LT);

    // Test scenario for equal to
    a = 4;
    b = 4;
    #10; // Wait for 10 time units
    // Display the inputs and outputs
    $display("a = %d, b = %d, EQ = %b, GT = %b, LT = %b", a, b, EQ, GT, LT);

    // Finish the simulation
    $finish;
  end
endmodule
```

Result: Comparator in Verilog is designed and simulated successfully.

11. FLIPFLOPS

Aim: To design a SR, D, JK,T flipflops using Verilog HDL.

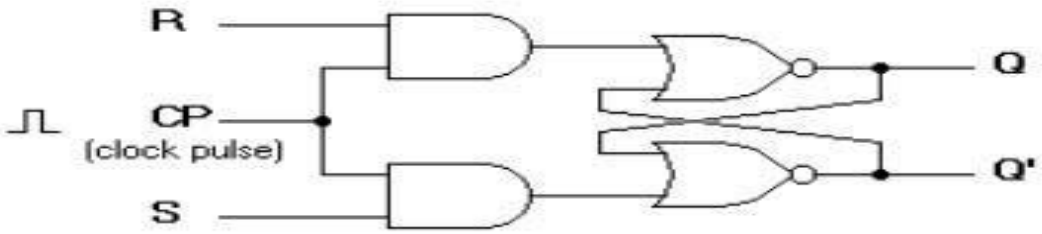
Software Used: Vivado 2016.4

Theory:

SR FLIPFLOP:

The SET-RESET flip flop is designed with the help of two NOR gates and also two NAND gates. These flip flops are also called S-R Latch.It is also called a Gated S-R flip flop.

The problems with S-R flip flops using NOR and NAND gate is the invalid state. This problem can be overcome by using a bistable SR flip-flop that can change outputs when certain invalid states are met, regardless of the condition of either the Set or the Reset inputs. For this, a clocked S-R flip flop is designed by adding two AND gates to a basic NOR Gate flip flop. The circuit diagram and truth table is shown below.



(a) Logic diagram

Q	S	R	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	indeterminate

(b) Truth table

Clocked SR flip-flop

A clock pulse [CP] is given to the inputs of the AND Gate. When the value of the clock pulse is '0', the outputs of both the AND Gates remain '0'. As soon as a pulse is given the value of CP turns '1'. This makes the values at S and R to pass through the NOR Gate flip flop. But when the values of both S and R values turn '1', the HIGH value of CP causes both of them to turn to '0' for a short moment. As soon as the pulse is removed, the flip flop state becomes intermediate. Thus either of the two states may be caused, and it depends on whether the set or reset input of the flip-flop remains a '1' longer than the transition to '0' at the end of the pulse. Thus the invalid states can be eliminated.

Source Code:

Design Block:

```
module sr_ff(s,r,clk,rst, q,qb);
input s,r,clk,rst;
output q,qb;
wire s,r,clk,rst,qb;
 reg q;
 always@(posedge clk)
 begin
 if(rst)
 q<=1'b0;
  else if(s==1'b0&&r==1'b0) q<=q;
  else if(s==1'b0&&r==1'b1) q<=1'b0;
  else if(s==1'b1&&r==1'b0) q<=1'b1;
  else if(s==1'b1&&r==1'b1) q<=1'bx;
  end
```

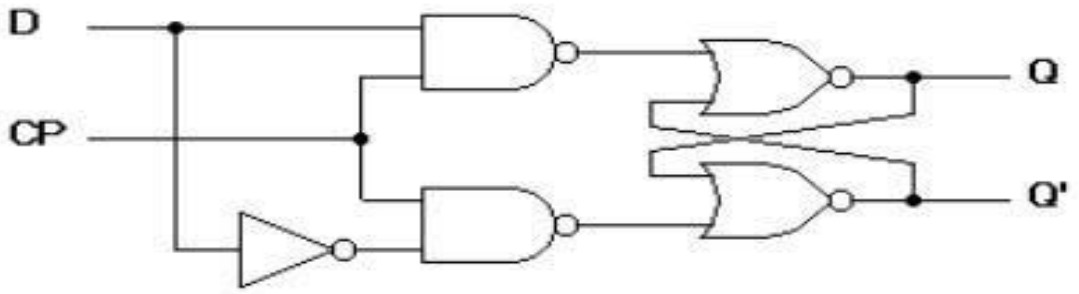
```
  assign qb=~q;
 endmodule
```

Test Bench:

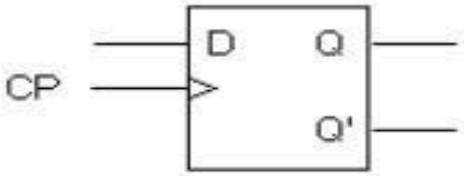
```
module srff_tb();
reg s,r,clk,rst;
wire q,qb;
sr_ff p(s,r,clk,rst,q,qb);
initial
begin
clk=0;
s=0; r=0;
#5 rst=1;
#30 rst=0;
$monitor($time,"clk=%b,rst=%b,s=%b,r=%b,q=%b,qb=%b",clk,rst,s,r,q,qb);
#100 $finish;
end
always #5 clk=~clk;
always #30 s=~s;
always #40 r=~r;
endmodule
```

D FLIPFLOP:

The circuit diagram and truth table is given below.



(a) Logic diagram with NAND gates



(b) Graphical symbol

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

(c) Transition table

Clocked D flip-flop

D flip flop is actually a slight modification of the above explained clocked SR flip-flop. From the figure you can see that the D input is connected to the S input and the complement of the D input is connected to the R input. The D input is passed on to the flip flop when the value of CP is '1'. When CP is HIGH, the flip flop moves to the SET state. If it is '0', the flip flop switches to the CLEAR state.

Source Code:

Design Block:

```
module dff(D,clk,reset,Q);
input D,clk,reset;
output Q;
reg Q;
always @(posedge clk)
begin
 Q <= D;
end
endmodule
```

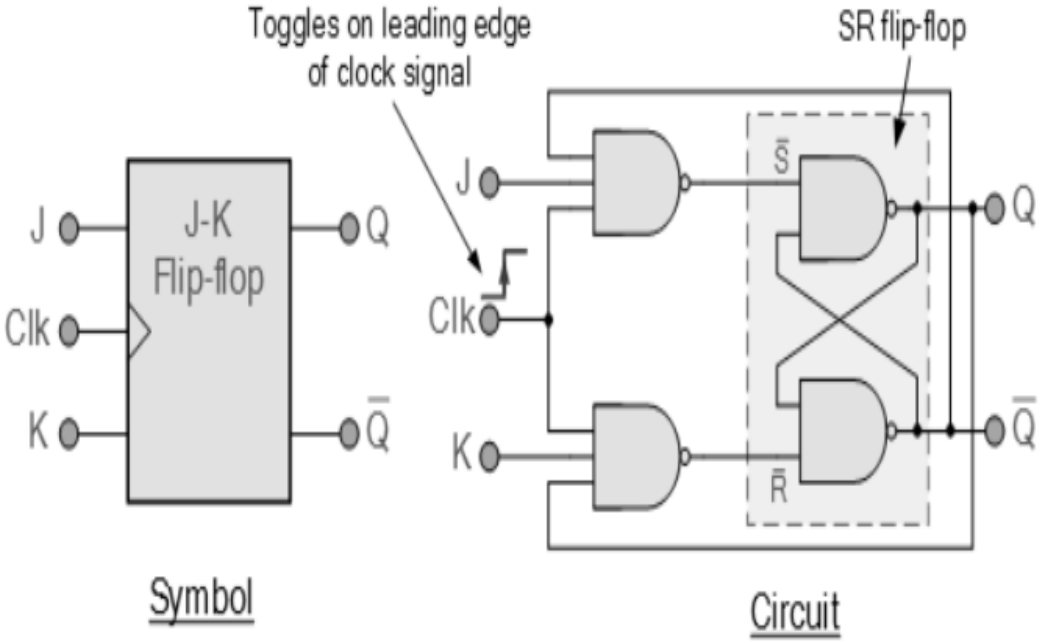
Test Bench:

```
module dff_tb();
reg D;
reg clk;
reg reset;
wire Q;
dff d1(D,clk,reset,Q);
initial begin
  clk=0;
     forever #10 clk = ~clk;
end
initial begin
 reset=1;
 D <= 0;
 #100;
 reset=0;
 D <= 1;
```

```
 #100;
 D <= 0;
 #100;
 D <= 1;
end
endmodule
```

JK FLIPFLOP:

Flip-flops are fundamental building blocks of sequential circuits. A flip flop can store one bit of data. Hence, it is known as a memory cell. Since they work on the application of a clock signal, they come under the category of synchronous circuits.The J-K flip-flop is the most versatile of the basic flip flops. The JK flip flop is a gated SR flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic 1. Due to this additional clocked input, a JK flip-flop has four possible input combinations, “logic 1”, “logic 0”, “no change” and “toggle”.



Truth table:

J	K	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	$\overline{Q_n}$(Toggles)

Source Code:

Design Block:

```
module JK_flipflop (
  input clk, rst_n,
  input j,k,
  output reg q,
  output q_bar
  );
  // always@(posedge clk or negedge rst_n) // for asynchronous reset
  always@(posedge clk) begin // for synchronous reset
    if(!rst_n) q <= 0;
    else begin
      case({j,k})
        2'b00: q <= q;     // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= ~q;    // Toggle
      endcase
    end
```

```
  end
  assign q_bar = ~q;
endmodule
```

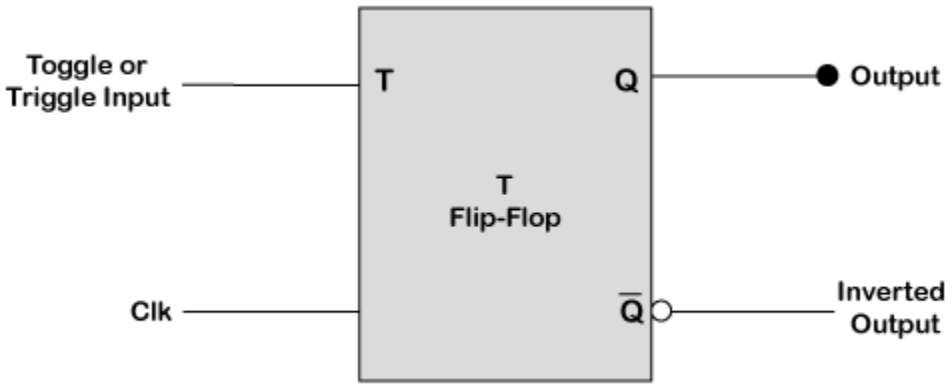
Testbench:

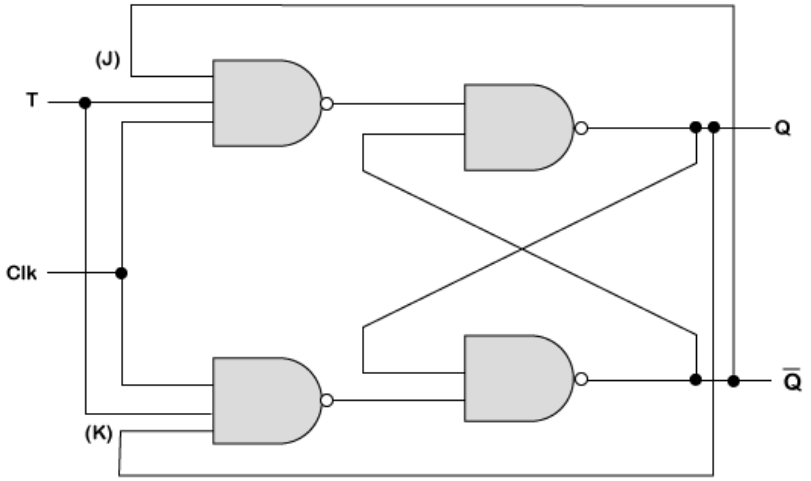
```
module tb;
  reg clk, rst_n;
  reg j, k;
  wire q, q_bar;
    JK_flipflop dff(clk, rst_n, j, k, q, q_bar);

  always #2 clk = ~clk;
  initial begin
    clk = 0; rst_n = 0;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);
    #3 rst_n = 1;
    $display("Reset=%b --> q=%b, q_bar=%b", rst_n, q, q_bar);
    drive(2'b00);
    drive(2'b01);
    drive(2'b10);
    drive(2'b11); // Toggles previous output
    drive(2'b11); // Toggles previous output
    #5;
```

```
    $finish;
  end
  task drive(bit [1:0] ip);
    @(posedge clk);
    {j,k} = ip;
    #1 $display("j=%b, k=%b --> q=%b, q_bar=%b",j, k, q, q_bar);
  endtask
    initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
endmodule
```

T FLIPFLOP:

T stands for ("toggle") flip-flop to avoid an intermediate state in SR flip-flop. We should provide only one input to the flip-flop called Trigger input Toggle input to avoid an intermediate state occurrence. Then the flip - flop acts as a Toggle switch. The next output state is changed with the complement of the present state output. This process is known as Toggling. We can construct the T flip-flop by making changes in the JK flip-flop. The T flip-flop has only one input, which is constructed by connecting the input of JK flip-flop. This single input is called T.





Source Code:

Design Block:

```
module tff ( input clk, input rstn, input t, output reg q);
   always @ (posedge clk) begin
   if (!rstn)
     q <= 0;
   else
      if (t)
         q <= ~q;
      else
         q <= q;
 end
```

```
endmodule
```

Testbench:

```
module tb;
  reg clk;
  reg rstn;
  reg t;

  tff u0 (  .clk(clk),
            .rstn(rstn),
            .t(t),
           .q(q));
    always #5 clk = ~clk;
   initial begin
    {rstn, clk, t} <= 0;
      $monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time, rstn, t, q);
    repeat(2) @(posedge clk);
    rstn <= 1;
      for (integer i = 0; i < 20; i = i+1) begin
      reg [4:0] dly = $random;
      #(dly) t <= $random;
    end
    #20 $finish;
  end
endmodule
```

Result: Flipflops in Verilog are designed and simulated successfully.

12. UNIVERSAL SHIFT REGISTER

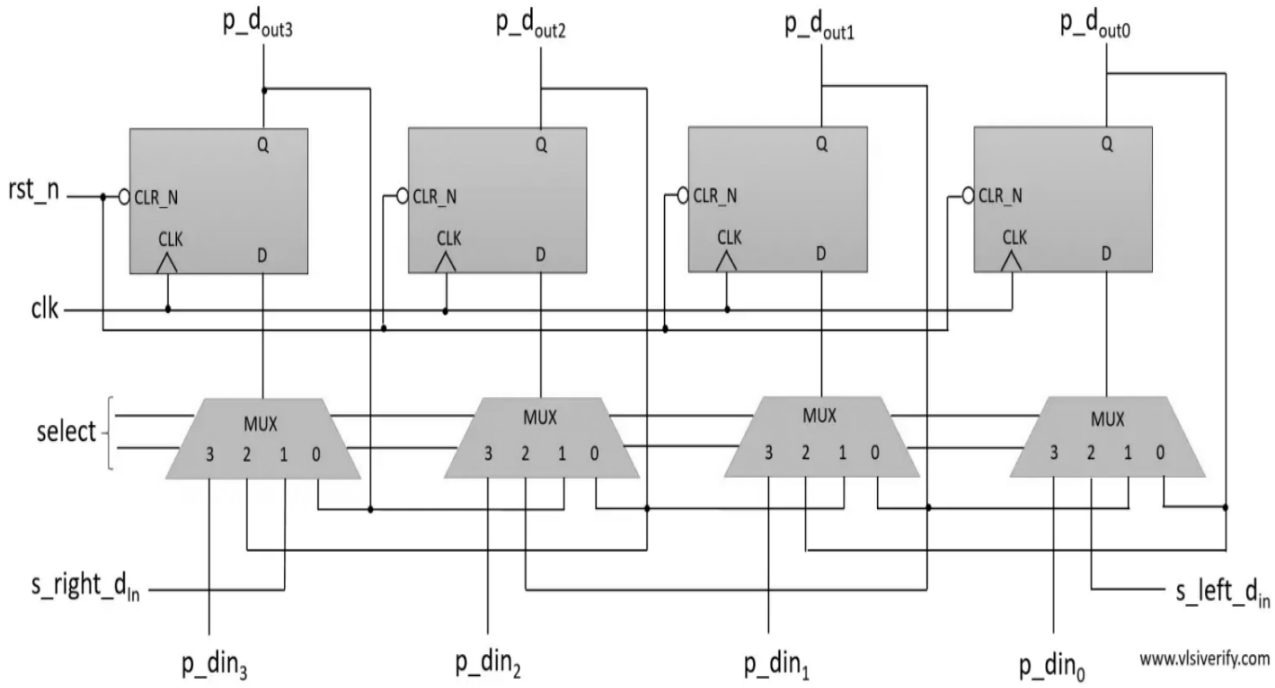
Aim: To design a Universal shift register using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A universal shift register is a sequential logic that can store data within and on every clock pulse it transfers data to the output port.The universal shift register can be used as:

- Parallel In Parallel Out shift register
- Parallel In Serial Out shift register
- Serial In Parallel Out shift register
- Serial In Serial Out shift register



Mode control (select)		Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Source Code:

Design Block:

```
module universal_shift_reg(
  input clk, rst_n,
  input [1:0] select, // select operation
  input [3:0] p_din,  // parallel data in
  input s_left_din,   // serial left data in
  input s_right_din,  // serial right data in
  output reg [3:0] p_dout, //parallel data out
  output s_left_dout, // serial left data out
  output s_right_dout // serial right data out
);
  always@(posedge clk) begin
    if(!rst_n) p_dout <= 0;
    else begin
      case(select)
        2'h1: p_dout <= {s_right_din,p_dout[3:1]}; // Right Shift
        2'h2: p_dout <= {p_dout[2:0],s_left_din};  // Left Shift
        2'h3: p_dout <= p_din; // Parallel in - Parallel out
        default: p_dout <= p_dout; // Do nothing
      endcase
    end
  end
  assign s_left_dout = p_dout[0];
  assign s_right_dout = p_dout[3];
endmodule
```

Testbench:

```
module TB;
  reg clk, rst_n;
  reg [1:0] select;
  reg [3:0] p_din;
  reg s_left_din, s_right_din;
```

```
  wire [3:0] p_dout; //parallel data out
  wire s_left_dout, s_right_dout;
    universal_shift_reg usr(clk, rst_n, select, p_din, s_left_din, s_right_din, p_dout, s_left_dout, s_right_dout);
    always #2 clk = ~clk;
  initial begin
    $monitor("select=%b, p_din=%b, s_left_din=%b, s_right_din=%b --> p_dout = %b, s_left_dout = %b, s_right_dout = %b",select, p_din, s_left_din, s_right_din, p_dout, s_left_dout, s_right_dout);
    clk = 0; rst_n = 0;
    #3 rst_n = 1;
    p_din = 4'b1101;
    s_left_din = 1'b1;
    s_right_din = 1'b0;

    select = 2'h3; #10;
    select = 2'h1; #20;
    p_din = 4'b1101;
    select = 2'h3; #10;
    select = 2'h2; #20;
    select = 2'h0; #20;
    $finish;
  end
  // To enable waveform
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
   endmodule
```

Result: Universal shift register in Verilog is designed and simulated successfully.

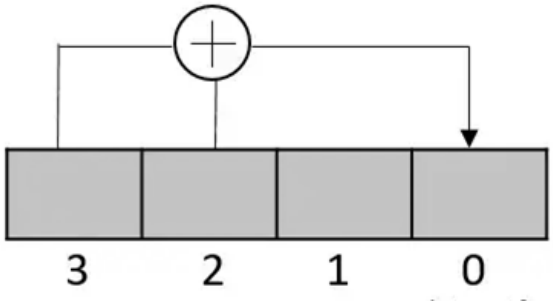
13.LINEAR FEEDBACK SHIFT REGISTER

Aim: To design a Linear Feedback Shift Register using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Linear Feedback Shift Register (LFSR) is a sequential shift register that generates pseudo-random binary sequences. It operates by shifting bits and feeding back a linear combination of its previous values. An LFSR consists of a shift register and a feedback function. The register contains a series of flip-flops, each storing a single bit. The feedback is typically an XOR operation applied to selected bits of the register, and the result is fed back into the input.Shift Register: It is a series of flip-flops where each clock cycle shifts the bits one position to the right.Feedback: A linear feedback function (usually an XOR of certain bits of the register) determines the value of the bit that gets shifted into the register.Feedback Taps: These are the positions in the register whose values are XORed together to form the feedback. Periodicity: LFSRs are periodic; after a certain number of shifts, the sequence repeats. The maximum period for an LFSR of length nnn is 2n−12^n - 12n−1, which happens when the feedback taps are selected correctly (this is called a *maximal-length* LFSR).Pseudorandomness: The sequence generated by an LFSR appears random, but it is deterministic and repeats after the period.Applications: LFSRs are widely used in communication systems, cryptography, error detection and correction (CRC), random number generation, and digital signal processing.Fibonacci LFSR: The feedback is applied to the input directly from the XOR of selected bits from the current state.Galois LFSR: The feedback affects the state during each clock cycle as the bits are shifted, improving efficiency in hardware implementations.



Source Code:

Design Block:

```
module LFSR(input clk, rst, output reg [3:0] op);
  always@(posedge clk) begin
    if(rst) op <= 4'hf;
    else op = {op[2:0],(op[3]^op[2])};
  end
endmodule
```

Testbench:

```
module TB;
  reg clk, rst;
  wire [3:0]op;
    LFSR lfsr1(clk, rst, op);
    initial begin
    $monitor("op=%b",op);
    clk = 0; rst = 1;
    #5 rst = 0;
    #50; $finish;
  end
    always #2 clk=~clk;
    initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Result: Linear Feedback Shift Register in Verilog is designed and simulated successfully.

13.COUNTERS

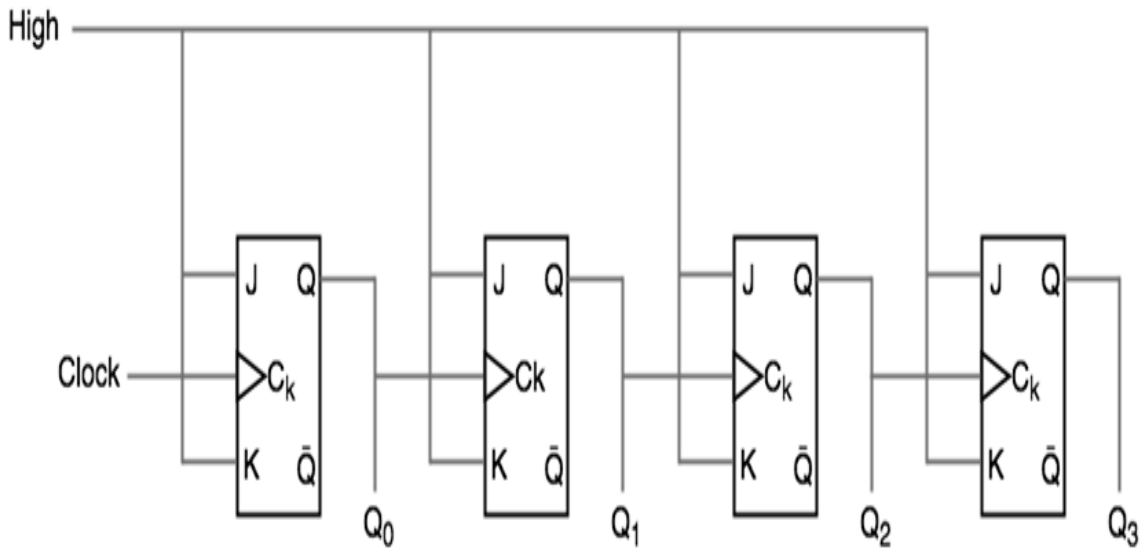
Aim: To design a counter using Verilog HDL.

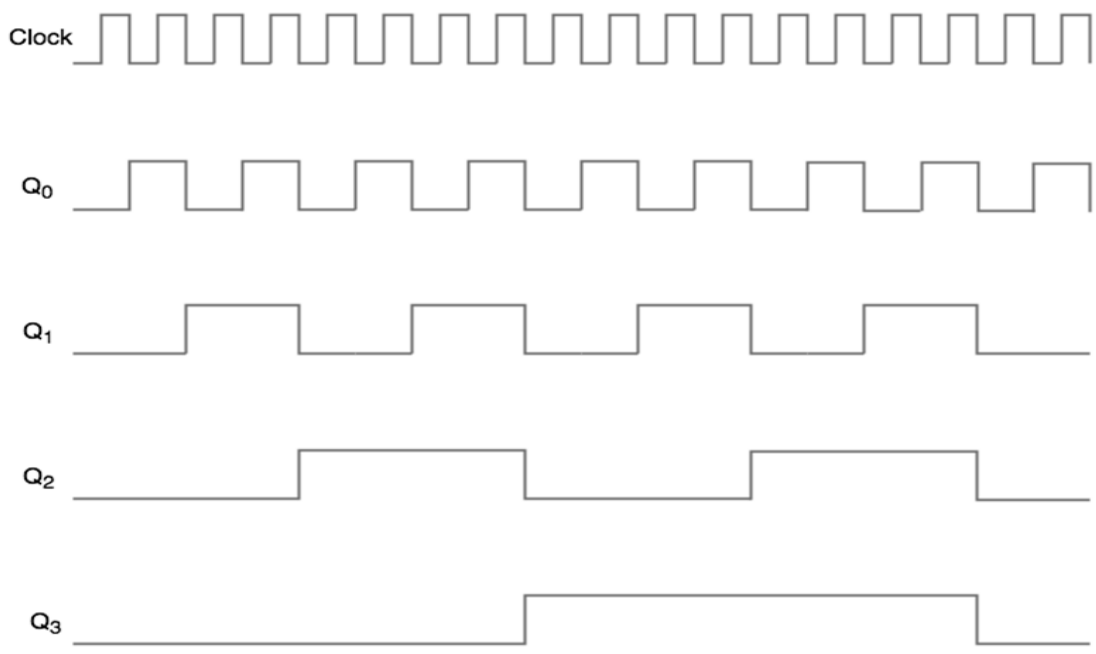
Software Used: Vivado 2016.4

Theory:

UP COUNTER:

An up-counter counts in ascending order from 0 up to a predefined maximum value (e.g., 1111 for a 4-bit counter, which is 15 in decimal) and then rolls back to 0. Each time a clock pulse is received, the counter increases by one. An up counter consists of the following components: Flip-Flops: Typically, T flip-flops or JK flip-flops are used. For a counter with n-bits, n-flip-flops are required. Clock Input: All flip-flops receive the same clock signal (in synchronous counters) or different clock signals (in asynchronous counters). Binary Output: The output is a binary number, representing the current count. Binary Up Counter: Counts in binary from 0 to the maximum value. Decade Counter (BCD Counter): Counts from 0 to 9 and then resets to 0. This is a type of up counter that generates a Binary-Coded Decimal (BCD) output. Modulus Counter (MOD Counter): A counter that counts from 0 to a predefined value (less than the maximum) and then resets. Up counters are used in various applications: Timers, Event Counters, Frequency Dividers, Digital Instruments





Source Code:

Design Block:

```
module up_counter(input clk, reset, output[3:0] counter  );
reg [3:0] counter_up;
// up counter
always @(posedge clk or posedge reset)
begin
if(reset)
 counter_up <= 4'd0;
else
 counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

Testbench:

```
module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;
up_counter dut(clk, reset, counter);
```

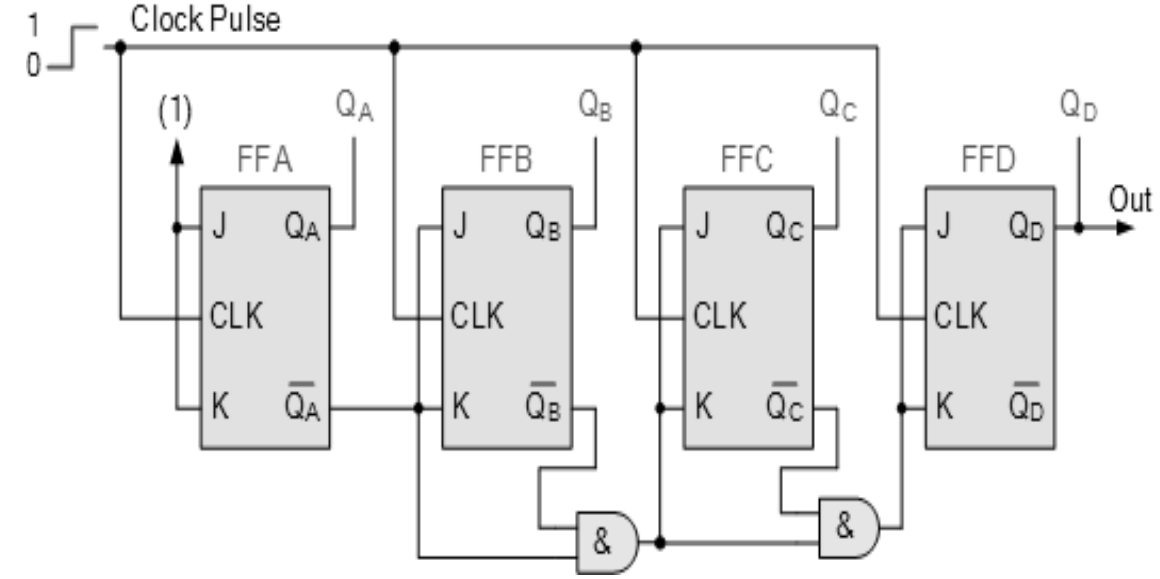
```
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

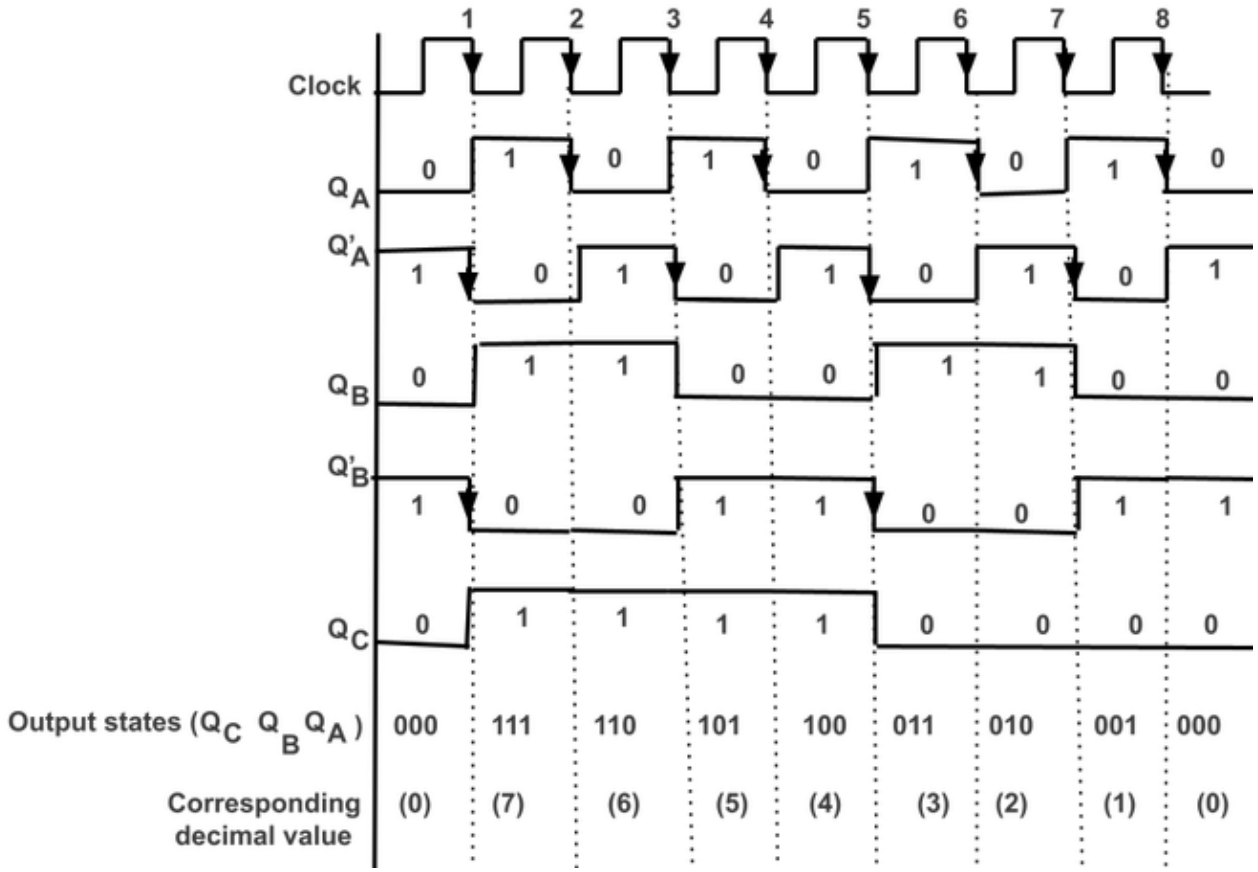
DOWN COUNTER:

THEORY:

A Down Counter is a type of digital sequential circuit that counts downward from a preset value to zero. Like up counters, down counters are made using flip-flops, and they are widely used in timing applications, event counting, frequency division, and other digital systems. The main difference from an up counter is that the count decreases on each clock pulse instead of increasing. A down counter starts from a predefined maximum value and decreases by one with each clock pulse until it reaches zero, after which it either stops or rolls over back to the maximum value (depending on the design). The output is in binary form and decrements with each clock cycle.A typical down counter consists of:Flip-Flops: Each flip-flop represents a single bit in the binary count. For a counter with nnn bits, nnn flip-flops are required.Clock Input: Like up counters, down counters can either be synchronous (where all flip-flops share the same clock signal) or asynchronous (where each flip-flop is triggered by the output of the preceding flip-flop).Binary Output: The counter's output is a binary number representing the current count.In a down counter, the binary number decreases on each clock pulse.If it is a continuous down counter, after reaching zero, the counter rolls over to its maximum value and continues counting down. If it's a terminal down counter, it stops when

the count reaches zero.Binary Down Counter: Counts downward in binary, from the maximum value down to zero.Decade (BCD) Down Counter: Counts from 9 down to 0 in Binary Coded Decimal (BCD) format.Modulus (MOD) Down Counter: A MOD-N counter counts down from N-1 to 0. For example, a MOD-10 counter counts down from 9 to 0.





Source Code:

Design Block:

```
module down_counter(input clk, reset, output [3:0] counter  );
reg [3:0] counter_down;
// down counter
always @(posedge clk or posedge reset)
begin
if(reset)
 counter_down <= 4'hf;
else
 counter_down <= counter_down - 4'd1;
end
assign counter = counter_down;
endmodule
```

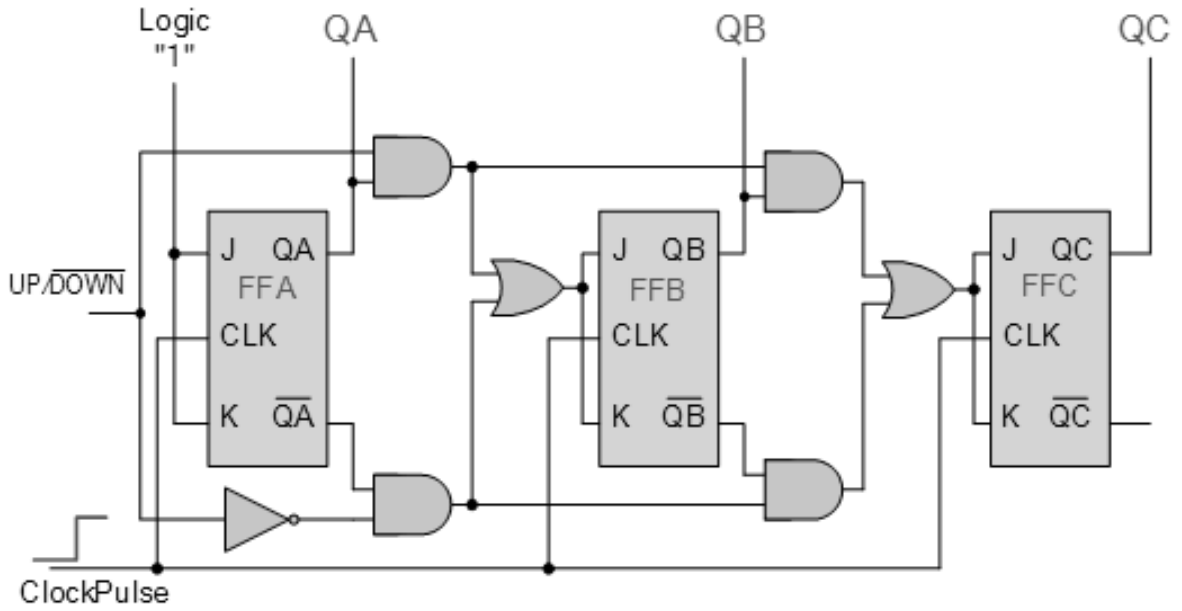
Testbench:

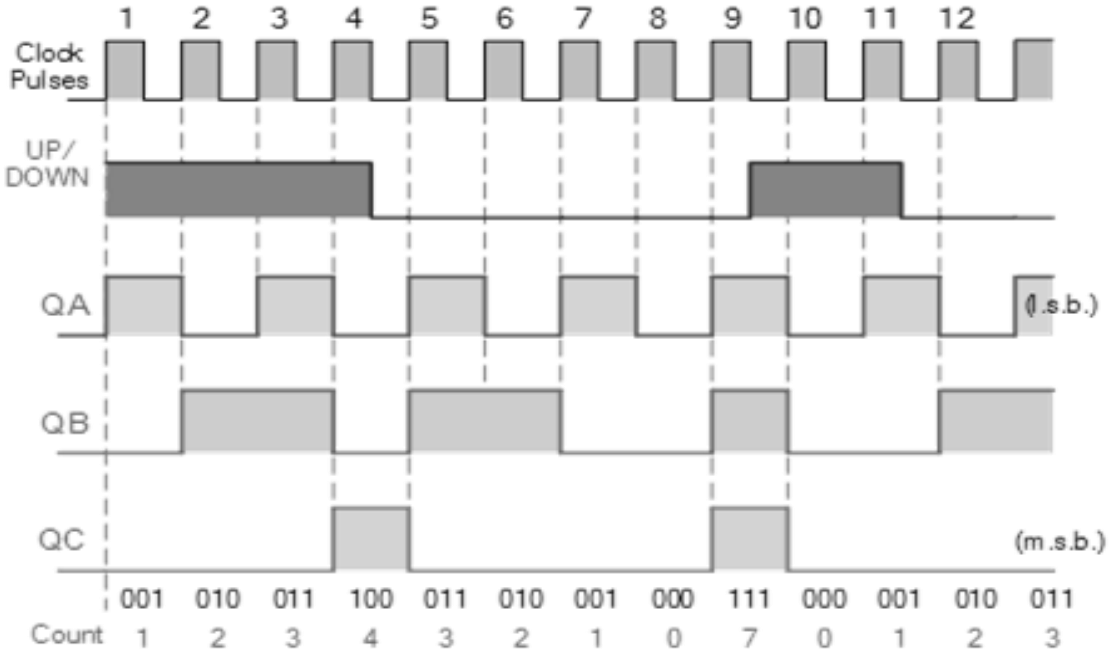
```
module downcounter_testbench();
reg clk, reset;
wire [3:0] counter;
down_counter dut(clk, reset, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

UP-DOWN COUNTER:

THEORY:

An Up-Down Counter is a type of digital sequential circuit that can count in both directions: upward (incrementing) and downward (decrementing), depending on a control signal. This feature makes it versatile in applications where bidirectional counting is needed, such as in position tracking, reversible timers, or event counters. An up-down counter operates similarly to a regular counter but includes a control signal that determines the counting direction: When the control signal is set to 1 (or HIGH), the counter behaves as an up counter, incrementing its value on each clock pulse. When the control signal is set to 0 (or LOW), the counter behaves as a down counter, decrementing its value on each clock pulse. An up-down counter consists of Flip-Flops: Typically, T flip-flops or JK flip-flops are used. For an n-bit counter, n- flip-flops are needed. Control Signal: Determines whether the counter counts upward or downward. Clock Input: Provides the timing signal to drive the counter's state transitions. Binary Output: The counter produces a binary output representing the current count. The up-down counter uses the control signal to dictate the direction of counting. Each clock pulse results in either an increment (up) or a decrement (down) of the counter's value based on the control input. Up Counting Mode: The counter increments on every clock pulse, moving from a low value (e.g., 0000) to the maximum value (e.g., 1111 for a 4-bit counter).Down Counting Mode: The counter decrements on every clock pulse, moving from the maximum value (e.g., 1111) to the minimum value (e.g., 0000).Up-down counters are used in a wide variety of digital systems, especially where counting in both directions is required as Digital Position Trackers, Reversible Timers, Digital Instruments, Frequency Counters and Dividers.





Source Code:

Design Block:

```
module up_down_counter(input clk, reset,up_down, output[3:0]  counter );
reg [3:0] counter_up_down;
// down counter
always @(posedge clk or posedge reset)
begin
```

```
if(reset)
 counter_up_down <= 4'h0;
else if(~up_down)
 counter_up_down <= counter_up_down + 4'd1;
else
 counter_up_down <= counter_up_down - 4'd1;
end
assign counter = counter_up_down;
endmodule
```

Testbench:

```
module updowncounter_testbench();
reg clk, reset,up_down;
wire [3:0] counter;
up_down_counter dut(clk, reset,up_down, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
up_down=0;
#20;
reset=0;
#200;
up_down=1;
end
endmodule
```

Result: counters in Verilog is designed and simulated successfully.

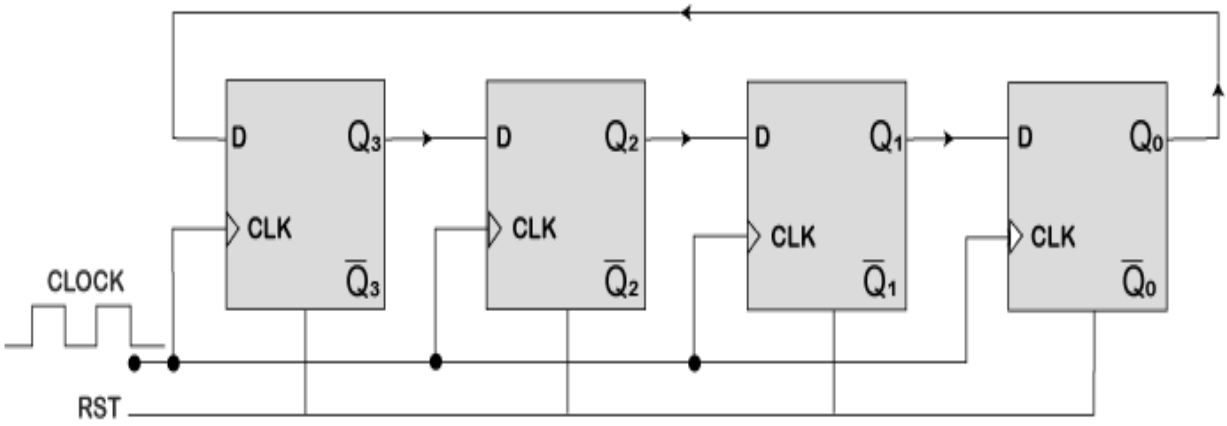
14.RING COUNTER

Aim: To design a Ring Counter using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A Ring Counter is a type of digital sequential circuit that cycles through a fixed sequence of binary states. It is made by shifting a single "1" or "0" through a series of flip-flops arranged in a ring. Ring counters are commonly used in timing circuits, pattern generation, and state machines. A ring counter is a shift register where the output of the last flip-flop is fed back to the input of the first flip-flop. The counter "rings" or circulates a single '1' or '0' bit through the register stages. For a 4-bit ring counter: It starts with a single '1' at the first flip-flop, e.g., 1000.With each clock pulse, the '1' shifts to the next flip-flop, and eventually returns to the first flip-flop after four cycles, making it a cyclic counter. Basic Ring Counter: A single '1' (or '0') circulates through the flip-flops. Johnson Ring Counter (Twisted Ring Counter): A variation where the inverted output of the last flip-flop is fed into the input of the first flip-flop, creating a count sequence twice the length of a standard ring counter. A ring counter consists of Flip-flops: Typically, D flip-flops are used, where each flip-flop represents one bit of the counter. For an n-bit ring counter, n-flip-flops are required. Clock Input: Drives the state transitions of the flip-flops. Feedback Path: The output of the last flip-flop is fed back to the input of the first flip-flop to create the ring.



Source Code:

Design Block:

```
module ring_ctr   (
   input clk,
   input rstn,
   output reg [WIDTH-1:0] out    );

  always @ (posedge clk) begin
     if (!rstn)
        out <= 1;
     else begin
       out[WIDTH-1] <= out[0];
       for (int i = 0; i < WIDTH-1; i=i+1) begin
         out[i] <= out[i+1];
       end
     end
  end
endmodule
```

Testbench:

```
module tb;
  parameter WIDTH = 4;
   reg clk;
  reg rstn;
  wire [WIDTH-1:0] out;

  ring_ctr  u0 (.clk (clk),
                .rstn (rstn),
                .out (out));

  always #10 clk = ~clk;
```

```
  initial begin
    {clk, rstn} <= 0;

    $monitor ("T=%0t out=%b", $time, out);
    repeat (2) @(posedge clk);
    rstn <= 1;
    repeat (15) @(posedge clk);
    $finish;
  end
endmodule
```

Result: Ring counters in Verilog is designed and simulated successfully.

15.Bus Transceiver

Aim: To design a Bus Transceiver using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

A bus transceiver is a bidirectional communication device that enables data transfer between two buses. It allows for the flow of data in both directions, either from bus A to bus B or from bus B to bus A, depending on the control signal. These are commonly used in systems where different parts need to share data but operate at different times. Bus: A bus is a communication pathway shared by multiple devices, typically used to transfer data, control signals, and power. It reduces the number of physical connections by allowing multiple devices to use the same connection lines. Transceiver: A transceiver (Transmitter + Receiver) is a device that can send and receive data. In the case of bus transceivers, it allows communication between two buses by driving signals in both directions. Bidirectional Data Flow: A bus transceiver is capable of handling bidirectional data transmission. Control signals decide the direction of the data flow—either from Bus A to Bus B or from Bus B to Bus A. Control Signals: There are typically two control signals—one to enable the transceiver and one to control the direction of data flow. Enable (EN): When this signal is active, the transceiver is enabled, allowing data to pass through. Direction (DIR): This signal controls the direction of data flow. If it is set in one state (e.g., high), data flows from Bus A to Bus B; if set to the opposite state (e.g., low), data flows from Bus B to Bus A. Bidirectional Buffers: The transceiver consists of buffers that can be driven in either direction, depending on the control signals. Isolation: When the transceiver is not enabled, it can isolate the two buses, preventing them from interacting.

Source Code:

Design Block:

```
module bus_transceiver_74LS245 (
    input [7:0] A,        // 8-bit bus A
    input [7:0] B,        // 8-bit bus B
    input DIR,            // Direction control: 1 -> A to B, 0 -> B to A
```

```
    input G,              // Enable: Active low (0 -> enabled, 1 -> disabled)
    output reg [7:0] bus_A, // Output for bus A
    output reg [7:0] bus_B  // Output for bus B
);

// Transceiver behavior
always @(*) begin
    if (!G) begin // When G is low, transceiver is enabled
        if (DIR) begin
            // A to B direction
            bus_B = A;
            bus_A = 8'bz; // Bus A in high impedance (disconnected)
        end else begin
            // B to A direction
            bus_A = B;
            bus_B = 8'bz; // Bus B in high impedance (disconnected)
        end
    end else begin
        // When G is high, both buses are in high impedance (disabled)
        bus_A = 8'bz;
        bus_B = 8'bz;
    end
end
endmodule
```

Testbench:

```
module tb_bus_transceiver_74LS245;
    reg [7:0] A;         // 8-bit bus A
    reg [7:0] B;         // 8-bit bus B
    reg DIR;             // Direction control
    reg G;               // Enable signal

    wire [7:0] bus_A;    // Output for bus A
```

```
    wire [7:0] bus_B;    // Output for bus B
    // Instantiate the bus transceiver
    bus_transceiver_74LS245 uut (
        .A(A),
        .B(B),
        .DIR(DIR),
        .G(G),
        .bus_A(bus_A),
        .bus_B(bus_B)
    );
    // Test procedure
    initial begin
        // Initialize inputs
        A = 8'b00000000;
        B = 8'b00000000;
        DIR = 1'b0; // Initial direction
        G = 1'b1;   // Disable

        // Display initial state
        $monitor("Time: %0t | G: %b | DIR: %b | A: %b | B: %b | bus_A: %b |
bus_B: %b",
                 $time, G, DIR, A, B, bus_A, bus_B);

        // Test 1: Enable and transfer from A to B
        #10;
        G = 1'b0; // Enable
        A = 8'b10101010; // Test data for A
        DIR = 1'b1; // Set direction A to B
        #10;

        // Test 2: Transfer from B to A
        B = 8'b11001100; // Test data for B
        DIR = 1'b0; // Set direction B to A
```

```
        #10;

        // Test 3: Disable transceiver
        G = 1'b1; // Disable
        #10;

        // Test 4: Check outputs when disabled
        A = 8'b11111111; // New test data for A
        B = 8'b00000000; // New test data for B
        #10;

        // End simulation
        $finish;
    end

endmodule
```

Result: Bus Transceiver in Verilog is designed and simulated successfully.

15. Simulation/Study of Static/Dynamic electrical behavior

Aim: To design a Simulation/Study of Static/Dynamic electrical behavior using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Simulating and studying the static and dynamic electrical behavior of digital circuits in Verilog involves understanding and analyzing how the circuit behaves under different conditions, focusing on both its steady-state (static) and transient (dynamic) characteristics.

1. **Static Behavior**:
    - **Static Characteristics** refer to the steady-state response of a circuit. This includes voltage levels for logic '0' and logic '1', noise margins, and power consumption at rest.
    - Common static behavior metrics include:
        - **Logic Levels**: Voltage levels for logical '0' and '1'.
        - **Noise Margins**: The difference between the minimum input voltage recognized as a logical '0' and the maximum input voltage recognized as a logical '1'.
        - **Power Consumption**: The power consumed by the circuit when in a stable state.
2. **Dynamic Behavior**:
    - **Dynamic Characteristics** refer to how a circuit responds to changes over time, including delay times, rise and fall times, and power consumption during transitions.
    - Common dynamic behavior metrics include:
        - **Propagation Delay**: Time taken for a change in input to affect the output.
        - **Transition Times**: Time taken for the output to switch from low to high or vice versa.
        - **Power Consumption During Transitions**: Power used when the circuit changes states.

Source Code:

Design Block:

```
module nand_gate (
    input A,
    input B,
    output Y
);
    assign Y = ~(A & B);
endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_nand_gate;

    // Inputs
    reg A;
    reg B;

    // Outputs
    wire Y;

    // Instantiate the NAND gate
    nand_gate uut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    // Test procedure
    initial begin
        // Monitor output
        $monitor("A = %b, B = %b, Y = %b", A, B, Y);
```

```
        // Test all combinations of inputs
        A = 0; B = 0; #10;  // 00 -> Y = 1
        A = 0; B = 1; #10;  // 01 -> Y = 1
        A = 1; B = 0; #10;  // 10 -> Y = 1
        A = 1; B = 1; #10;  // 11 -> Y = 0

        // End simulation
        $finish;
    end
endmodule
```

Result: Simulation/Study of Static/Dynamic electrical behavior in Verilog is designed and simulated successfully.

16. Simulation/Study of CMOS logic families, Low voltage CMOS interfacing

Aim: To design a Simulation/Study of CMOS logic families, Low voltage CMOS interfacing using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

When studying CMOS (Complementary Metal-Oxide-Semiconductor) logic families and low-voltage CMOS interfacing, Verilog can be used to model the behavior of CMOS circuits, allowing for the simulation of various logic families and their interaction with low-voltage environments.

CMOS Logic Families

1. Standard CMOS: This family uses complementary PMOS and NMOS transistors to implement logic functions. It is characterized by low power consumption and high noise margins.
2. Low-Voltage CMOS: This variant of standard CMOS operates at lower supply voltages (typically below 1.8V), making it suitable for battery-operated devices. It requires careful design to ensure adequate noise margins and switching speeds.
3. Other CMOS Variants:
    - Dual-Supply CMOS: Utilizes different supply voltages for different parts of the circuit.
    - Dynamic CMOS: Reduces power consumption by using a clock signal to control the charge and discharge of nodes.

Source Code:

Design Block:

```
module cmos_inverter (
   input A,
   output Y);
      assign Y = ~A;
endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_cmos_inverter;
    reg A;
    wire Y;
 cmos_inverter uut (
        .A(A),
        .Y(Y)    );
    initial begin
        // Monitor output
        $monitor("Time: %0t | A = %b, Y = %b", $time, A, Y);

        A = 0; #10;  // Input 0
        A = 1; #10;  // Input 1

        $finish;
    end
endmodule
```

Result: Simulation/Study of CMOS logic families, Low voltage CMOS interfacing in Verilog is designed and simulated successfully.