Lab Code:18ECL52
HDL Lab
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 endeavors.

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 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.

HDL Lab

II B.Tech – II Semester (Code: 18ECL42)

Lectures	4	Tutorial	1	Practical	0	Credits	1
Continuous Internal Assessment			50	Semester End Examination (3 Hours)			50

Prerequisites: Digital Electronics

Course objectives: Students will

- Describe the importance of modern programmable logic devices
- Demonstrate different styles of writing HDL code
- Use vivado tools in digital circuits modeling, simulation, functional verification in Verilog
- Validate and synthesize a digital circuit to FPGA board using Verilog HDL

Course outcomes: After studying this course, the students will be able to

CO 1	Apply EDA tools for simulation, verification and synthesis of digital design
CO 2	Develop Verilog RTL code for combinational digital circuits.
CO 3	Develop Verilog RTL code for sequential digital circuits.
CO 4	Implement digital systems by programmable devices, such as FPGA

Mapping of Course Outcomes with Program Outcomes & Program															
	PO's												**PSO's**		
CO	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
CO1	3	2	2	2	3				2				3	3	3
CO2	3	2	2	2					2				3	3	3
CO3	3	2	2	2					2				3	3	3
CO4	3	2	3	2	3				3				3	3	3
AVG	3	2	2.2	2	3				2.2				3	3	3

List of Programs:

Implement the following in Verilog HDL

1. Logic Gates.
2. Multiplexers/ De-Multiplexers.
3. Encoders/ Decoders.
4. Comparators.

5. Adders/ Subtractors.
6. Multipliers.
7. Parity Generators.
8. Design of ALU.
9. Latches.
10. Flip-Flops.
11. Synchronous Counters.
12. Asynchronous Counters.
13. Shift Registers.
14. Memories.
15. CMOS Circuits.

NOTE: A minimum of 10 (Ten) programs are to be executed and recorded to attain eligibility for the Semester End Examination.

1. Logic Gates.

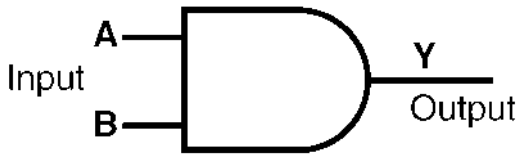
Aim: To design a Logic Gates using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

AND Gate

Symbol:



Operation: The output of the AND gate is 1 only when all its inputs are 1.

Truth Table:

A	B	Output (A AND B)
0	0	0
0	1	0
1	0	0
1	1	1

Source Code:

Design Block:

```
module AndGate (
    input wire a,
    input wire b,
    output wire and_out);
```

```
    assign and_out = a & b;  // AND gate operation
endmodule
```

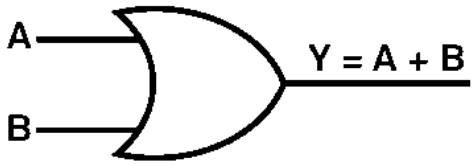
Testbench:

```
module tb_AndGate();
    reg a;
    reg b;
    wire and_out;
    // Instantiate the AndGate module
    AndGate uut (
        .a(a),
        .b(b),
        .and_out(and_out)
    );
    initial begin
        $display("Time\t a b | AND");
        $display("-------------------");
        $monitor("%0d\t %b %b |  %b", $time, a, b, and_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
endmodule
```

Theory:

OR Gate:

Symbol:



Operation: The output of the OR gate is 1 if at least one of its inputs is 1.

Truth Table:

A	B	Output (A OR B)
0	0	0
0	1	1
1	0	1
1	1	1

Source Code:

Design Block:

```
module OrGate (
    input wire a,
    input wire b,
    output wire or_out);
    assign or_out = a | b;  // OR gate operation
endmodule
```

Testbench:

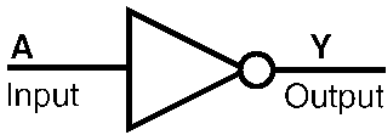
```
module tb_OrGate();
    reg a;
    reg b;
    wire or_out;
```

```
    // Instantiate the OrGate module
    OrGate uut (
        .a(a),
        .b(b),
        .or_out(or_out)
    );
    initial begin
        $display("Time\t a b | OR");
        $display("------------------");
        $monitor("%0d\t %b %b |  %b", $time, a, b, or_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
endmodule
```

Theory:

NOT Gate:

Symbol:



Operation: The NOT gate inverts the input signal. If the input is 1, the output is 0, and vice versa.

Truth Table:

A	Output (NOT A)
0	1
1	0

Source Code:

Design Block:

```
module NotGate (
    input wire a,
    output wire not_out);
    assign not_out = ~a;  // NOT gate operation
endmodule
```

Testbench:

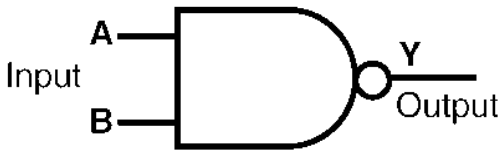
```
module tb_NotGate();
    reg a;
    wire not_out;
    // Instantiate the NotGate module
    NotGate uut (
        .a(a),
        .not_out(not_out)
    );
    initial begin
        $display("Time\t a | NOT");
        $display("----------------");
        $monitor("%0d\t %b |  %b", $time, a, not_out);
        // Test patterns
        a = 0; #10;
        a = 1; #10;
```

```
        $stop;
    end
endmodule
```

Theory:

NAND Gate:

symbol:



Operation: The output of the NAND gate is 0 only when all its inputs are 1. It is the inverse of the AND gate.

Truth Table:

A	B	Output (A NAND B)
0	0	1
0	1	1
1	1	0

Source Code:

Design Block:

```
module NandGate (
    input wire a,
    input wire b,
    output wire nand_out);
    assign nand_out = ~(a & b);  // NAND gate operation
endmodule
```

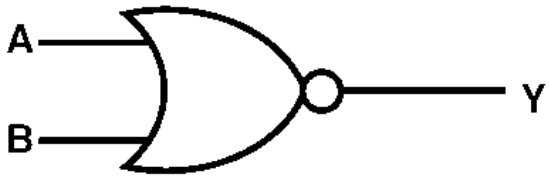
Testbench:

```
module tb_NandGate();
    reg a;
    reg b;
    wire nand_out;
    // Instantiate the NandGate module
    NandGate uut (
        .a(a),
        .b(b),
        .nand_out(nand_out)
    );
    initial begin
        $display("Time\t a b | NAND");
        $display("-------------------");
        $monitor("%0d\t %b %b |  %b", $time, a, b, nand_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
endmodule
```

Theory:

NOR Gate:

Symbol:



Operation: The output of the NOR gate is 1 only when all its inputs are 0. It is the inverse of the OR gate.

Truth Table:

A	B	Output (A NOR B)
0	0	1
0	1	0
1	0	0
1	1	0

Source Code:

Design Block:

```
module NorGate (
    input wire a,
    input wire b,
    output wire nor_out);
    assign nor_out = ~(a | b);  // NOR gate operation
endmodule
```

Testbench:

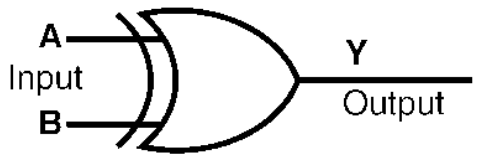
```
module tb_NorGate();
    reg a;
    reg b;
    wire nor_out;
```

```
    // Instantiate the NorGate module
    NorGate uut (
        .a(a),
        .b(b),
        .nor_out(nor_out)
    );
    initial begin
        $display("Time\t a b | NOR");
        $display("-------------------");
        $monitor("%0d\t %b %b |  %b", $time, a, b, nor_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
endmodule
```

Theory:

XOR Gate:

Symbol:



Operation: The output of the XOR gate is 1 if the inputs are different. If the inputs are the same, the output is 0.

Truth Table:

A	B	Output (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

Source Code:

Design Block:

```
module XorGate (
    input wire a,
    input wire b,
    output wire xor_out);
    assign xor_out = a ^ b;  // XOR gate operation
endmodule
```

Testbench:

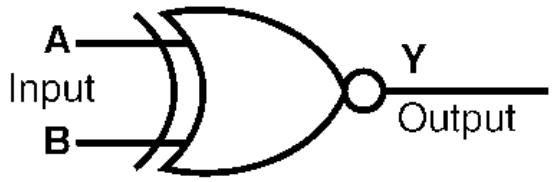
```
module tb_XorGate();
    reg a;
    reg b;
    wire xor_out;
    // Instantiate the XorGate module
    XorGate uut (
        .a(a),
        .b(b),
        .xor_out(xor_out)
    );
    initial begin
        $display("Time\t a b | XOR");
        $display("------------------");
```

```
        $monitor("%0d\t %b %b |  %b", $time, a, b, xor_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
endmodule
```

Theory:

XNOR Gate:

Symbol:



Operation: The output of the XNOR gate is 1 if the inputs are the same. If the inputs are different, the output is 0. It is the inverse of the XOR gate.

Truth Table:

A	B	Output (A XNOR B)
0	0	1
0	1	0
1	0	0
1	1	1

Source Code:

Design Block:

```
module XnorGate (
    input wire a,
    input wire b,
    output wire xnor_out);
    assign xnor_out = ~(a ^ b);  // XNOR gate operation
endmodule
```

Testbench:

```
module tb_XnorGate();
    reg a;
    reg b;
    wire xnor_out;
    // Instantiate the XnorGate module
    XnorGate uut (
        .a(a),
        .b(b),
        .xnor_out(xnor_out)
    );
    initial begin
        $display("Time\t a b | XNOR");
        $display("------------------");
        $monitor("%0d\t %b %b |  %b", $time, a, b, xnor_out);
        // Test patterns
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $stop;
    end
```

```
endmodule
```

Result: Logic Gates in Verilog is designed and simulated successfully.

2. Multiplexers/ De-Multiplexers.

Aim: To design a multiplexer using Verilog HDL.

Software Used: Vivado 2016.4

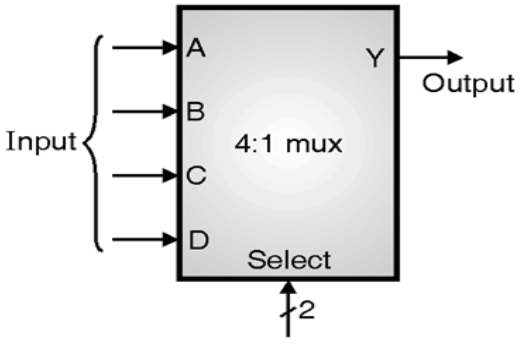
Theory:

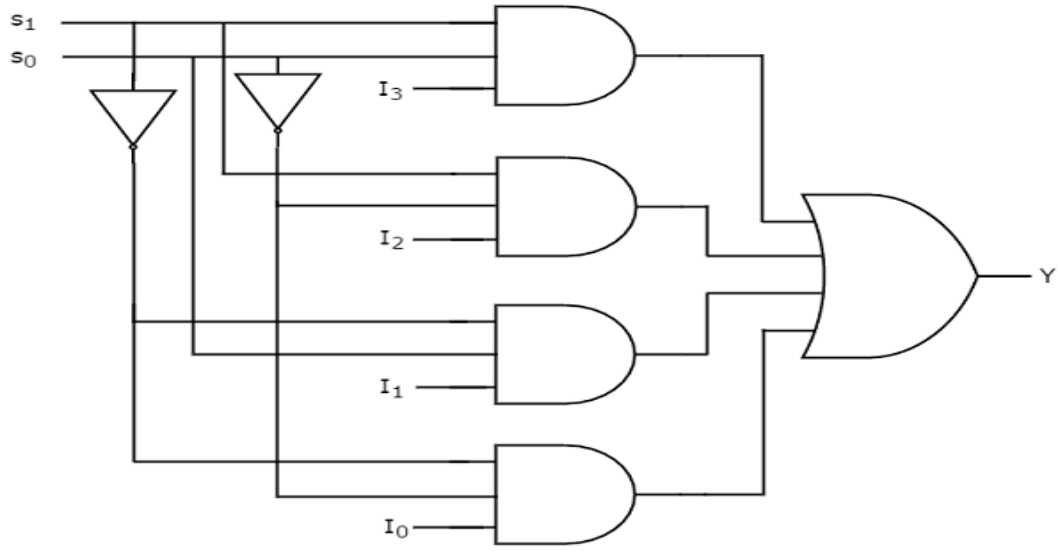
A multiplexer is a data selector device that selects one input from several input lines, depending upon the enabled, select lines, and yields one single output.A multiplexer of 2n inputs has n select lines, are used to select which input line to send to the output. There is only one output in the multiplexer, 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.

b) Demultiplexer:

Aim:

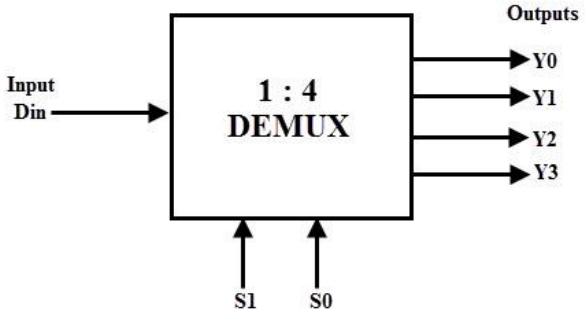
To design a 1x4 demultiplexer using Verilog HDL.

Software Used: Vivado 2016.4

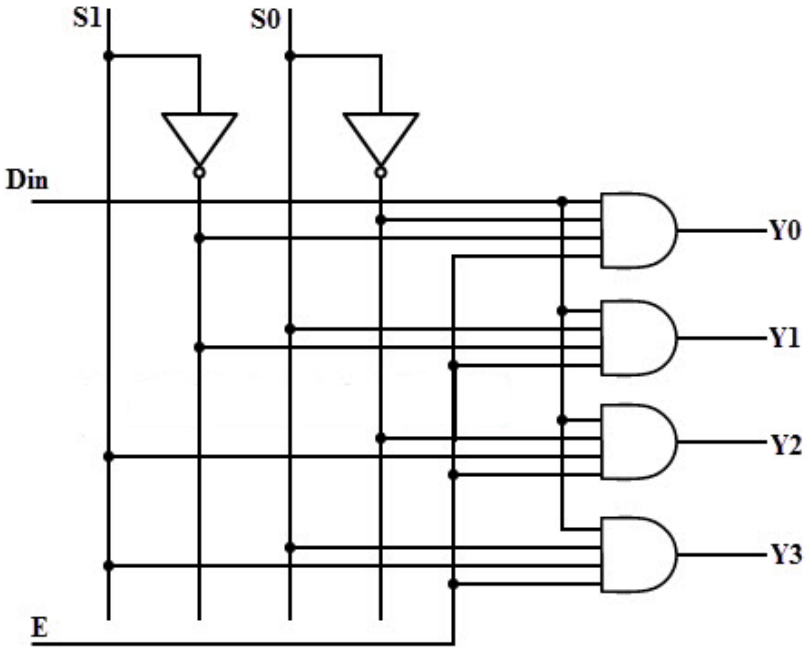
Theory:

A demultiplexer (or demux) is a device that takes a single input line and routes it to one of several digital output lines. A demultiplexer of 2^n outputs has n select lines, which are used to select which output line to send the input. A demultiplexer is also called a data distributor.Demultiplexers can be used to implement general purpose logic. By setting the input to true, the demux behaves as a decoder.A 1-to-4 demultiplexer has a single input (D), two

selection lines (S1 and S0) and four outputs (Y0 to Y3). The input data goes to any one of the four outputs at a given time for a particular combination of select lines.This demultiplexer is also called as a 2-to-4 demultiplexer which means that two select lines and 4 output lines.



The truth table of this type of demultiplexer is given below. From the truth table it is clear that, when S1=0 and S0= 0, the data input is connected to output Y0 and when S1= 0 and s0=1, then the data input is connected to output Y1.



Source Code:

Design Block:

```
module demux1x4(y0,y1,y2,y3,s0,s1,i);
output y0,y1,y2,y3;
input i;
input s0,s1;
assign y0=i&(~s0)&(~s1);
assign y1=i&(~s0)&(s1);
assign y2=i&(s0)&(~s1);
assign y3=i&(s0)&(s1);
endmodule
```

Test Bench:

```
module demux1x4_tb();
reg y0,y1,y2,y3;
reg s0,s1;
reg i;
demux1x4 n1(y0,y1,y2,y3,s0,s1,i);
initial
begin
    i=1;
    $display("i=%b",i);
end
initial
begin
s1=0;s0=0;
#5 $display("s1=%b,s0=%b,y0=%b",s1,s0,y0);
```

```
s1=0;s0=1;
#5 $display("s1=%b,s0=%b,y1=%b",s1,s0,y1);
s1=1;s0=0;
#5 $display("s1=%b,s0=%b,y2=%b",s1,s0,y2);
s1=1;s0=1;
#5 $display("s1=%b,s0=%b,y3=%b",s1,s0,y3);
end
endmodule
```

Result: Demultiplexer in Verilog is designed and simulated successfully.

3. Encoders/ Decoders.

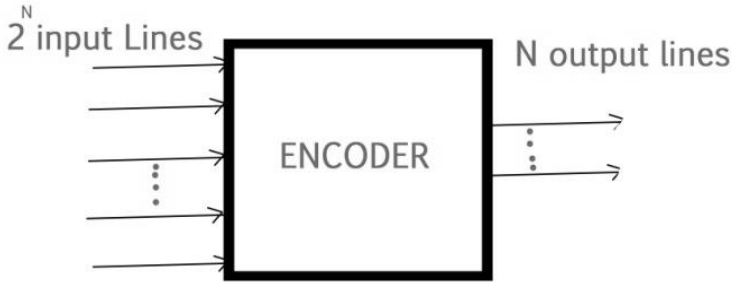
Aim:

To design a 8x3 encoder using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

An Encoder is a **combinational circuit** that performs the reverse operation of Decoder.It has maximum of **2^n input lines** and **'n' output lines**, hence it encodes the information from 2^n inputs into an n-bit code. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits.



8 : 3 Encoder (Octal to Binary) –

The 8 to 3 Encoder or octal to Binary encoder consists of **8 inputs** : Y7 to Y0 and **3 outputs** : A2, A1 & A0. Each input line corresponds to each octal digit and three outputs generate corresponding binary code.

The truth table for 8 to 3 encoder is as follows :

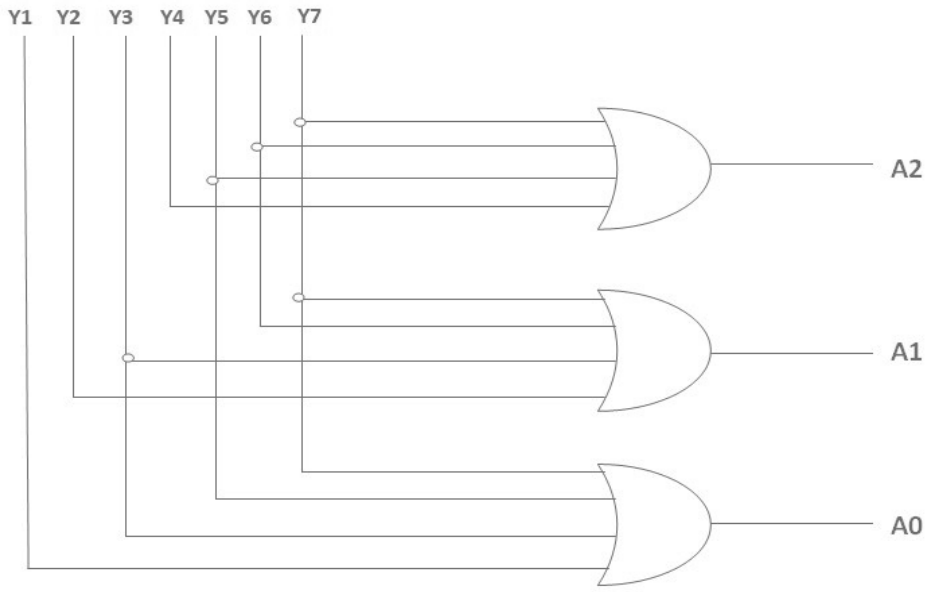
INPUTS								OUTPUTS		
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Logical expression for A2, A1 and A0 :

A2 = Y7 + Y6 + Y5 + Y4
A1 = Y7 + Y6 + Y3 + Y2
A0 = Y7 + Y5 + Y3 + Y1

Circuit Diagram:

The above two Boolean functions A2, A1 and A0 can be implemented using four Input OR gates.



Source Code:

Design Block:

```
module encoder(d0,d1,d2,d3,d4,d5,d6,d7,a,b,c);
input d0,d1,d2,d3,d4,d5,d6,d7;
output a,b,c;
   or(a,d4,d5,d6,d7);
   or(b,d2,d3,d6,d7);
   or(c,d1,d3,d5,d7);
endmodule
```

Test Bench:

```
module entb();
        reg d0,d1,d2,d3,d4,d5,d6,d7;
        wire a,b,c;
encoder n1(d0,d1,d2,d3,d4,d5,d6,d7,a,b,c);
initial
   begin
        d0 = 1;d1 = 0;d2 = 0;d3 = 0;d4 = 0;d5 = 0;d6 = 0;d7 = 0;
        #5 d0 = 0;d1 = 1;d2 = 0;d3 = 0;d4 = 0;d5 = 0;d6 =0;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 1;d3 = 0;d4 = 0;d5 = 0;d6 = 0;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 0;d3 = 1;d4 = 0;d5 = 0;d6 = 0;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 0;d3 = 0;d4 = 1;d5 = 0;d6 = 0;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 0;d3 = 0;d4 = 0;d5 = 1;d6 = 0;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 0;d3 = 0;d4 = 0;d5 = 0;d6 = 1;d7 = 0;
       #5 d0 = 0;d1 = 0;d2 = 0;d3 = 0;d4 = 0;d5 = 0;d6 = 0;d7 = 1;
        end
endmodule
```

Result:

8x3 Encoder using Verilog is designed and is simulated successfully.

b)Decoder:

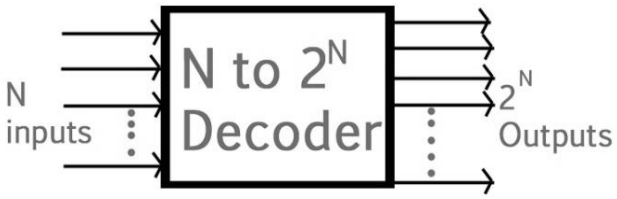
Aim:

To design 3x8 decoder using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

In Digital Electronics, discrete quantities of information are represented by binary codes. A binary code of **n bits** is capable of representing up to **2^n distinct elements** of coded information. The name **"Decoder"** means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output. A **decoder** is a **combinational circuit** that converts binary information from **n input lines** to a maximum of **2^n unique output lines**.

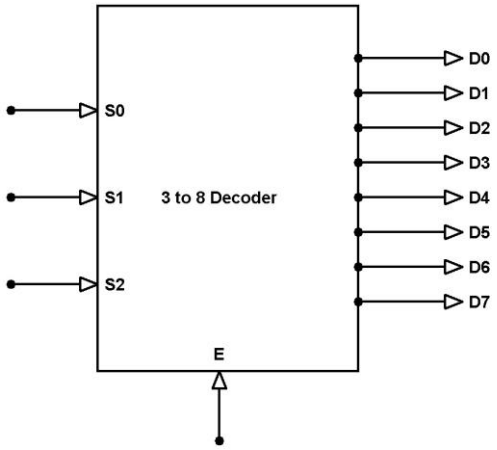


3 Line to 8 Line Decoder

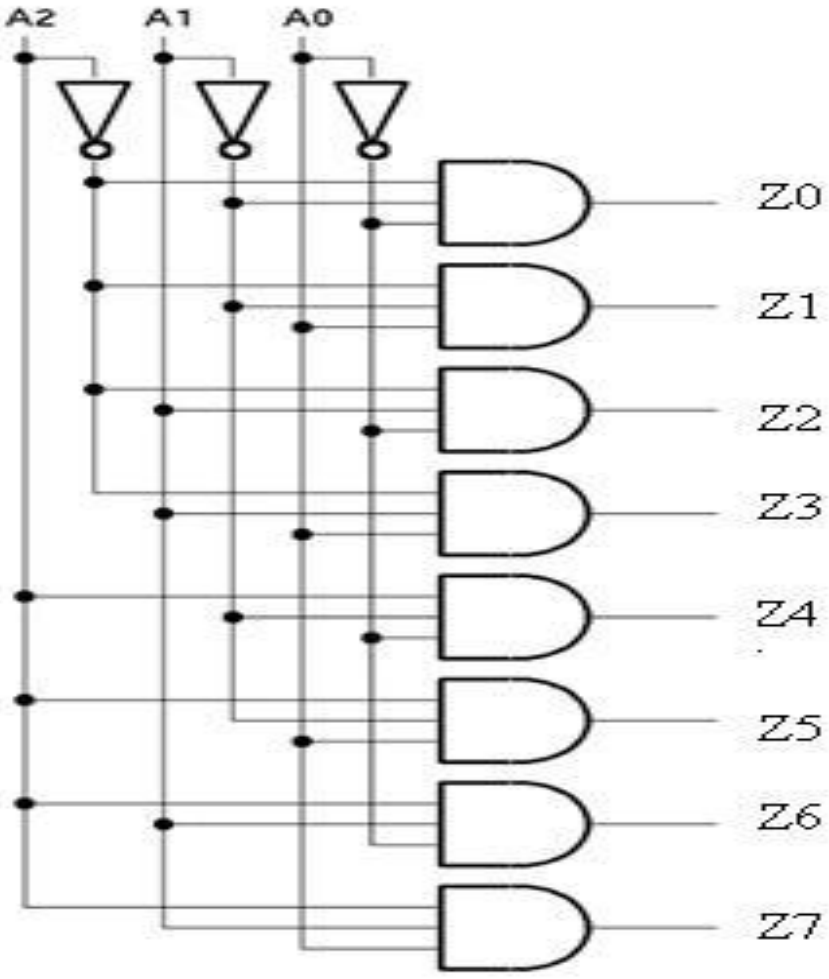
This decoder circuit gives 8 logic outputs for 3 inputs and has a enable pin. The circuit is designed with AND and NAND logic gates. It takes 3 binary inputs and activates one of the eight outputs. 3 to 8 line decoder circuit is also called as binary to an octal decoder.

3 to 8 Line Decoder Block Diagram

The decoder circuit works only when the Enable pin (E) is high. S0, S1 and S2 are three different inputs and D0, D1, D2, D3. D4. D5. D6. D7 are the eight outputs.



Circuit Diagram:



3 to 8 Line Decoder Truth Table

Inputs				Outputs							
EN	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	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

Source Code:

Design Block:

```
module decoder(a,b,c,d0,d1,d2,d3,d4,d5,d6,d7);
        input a,b,c;
        output d0,d1,d2,d3,d4,d5,d6,d7;
        assign d0=(~a&~b&~c),
         d1=(~a&~b&c),
                        d2=(~a&b&~c),
                        d3=(~a&b&c),
                        d4=(a&~b&~c),
                        d5=(a&~b&c),
                        d6=(a&b&~c),
                        d7=(a&b&c);
endmodule
```

Test Bench:

```
module decoder_tb();
reg a,b,c;
wire d0,d1,d2,d3,d4,d5,d6,d7;
```

```
decoder n1(a,b,c,d0,d1,d2,d3,d4,d5,d6,d7);
initial begin
                a = 0;b = 0;c = 0;
        #10 a = 0;b = 0;c = 1;
        #10 a = 0;b = 1;c = 0;
        #10 a = 0;b = 1;c = 1;
        #10 a = 1;b = 0;c = 0;
        #10 a = 1;b = 0;c = 1;
        #10 a = 1;b = 1;c = 0;
        #10 a = 1;b = 1;c = 1;
         end
endmodule
```

Result:

3x8 decoder using Verilog is designed and is simulated successfully

4. Comparators.

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.

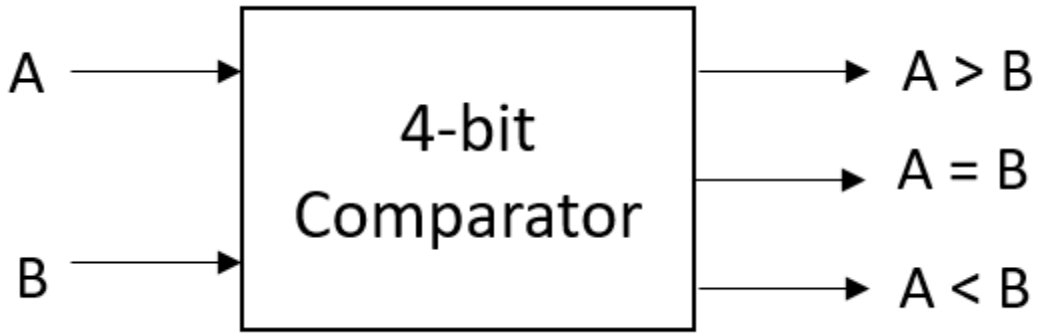
In a 4-bit comparator the condition of A>B can be possible in the following four cases:

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

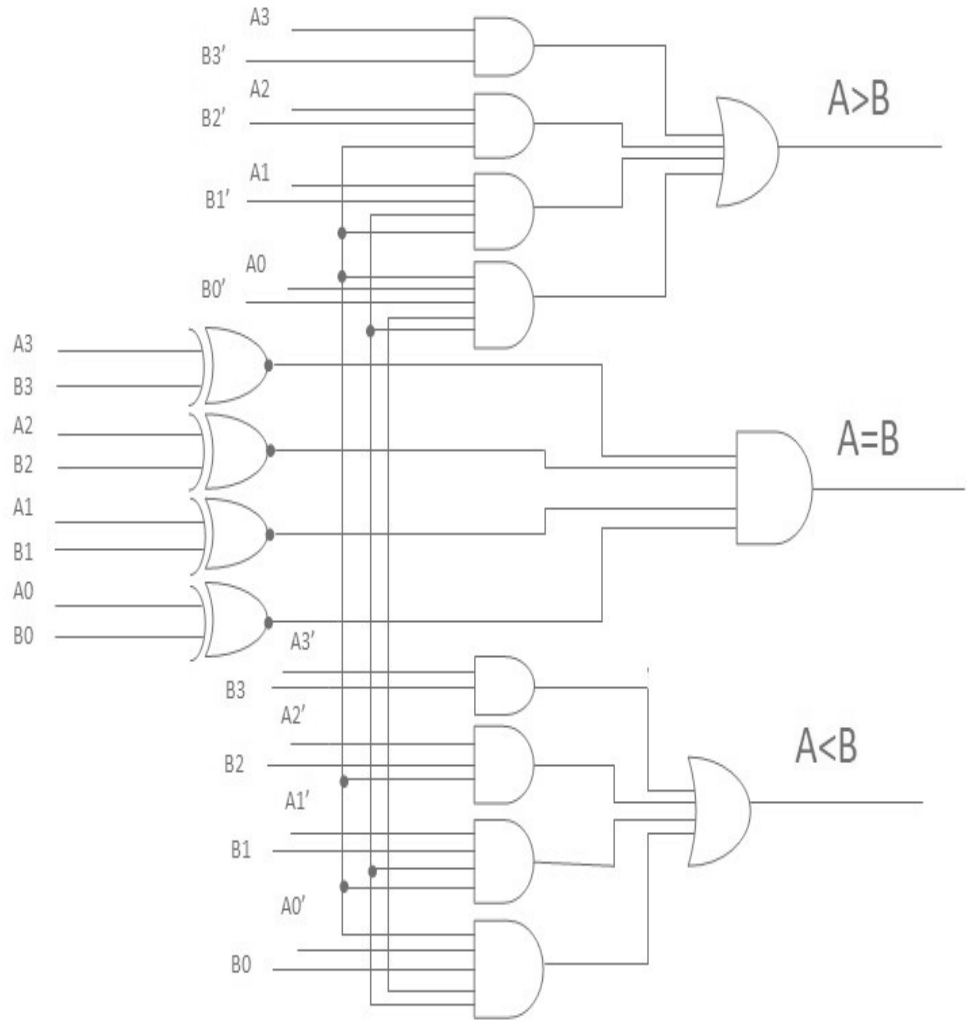
Similarly, the condition for A<B can be possible in the following four cases:

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 the 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.

5. Adders/ Subtractors.

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)**:

  $S=A \oplus B \oplus Cin$

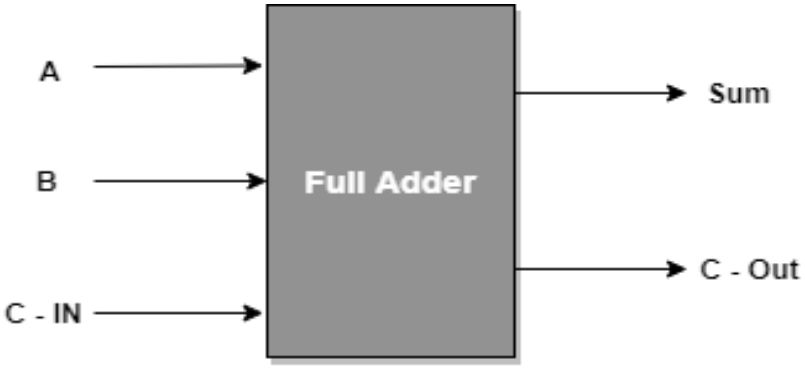
(Where $\oplus$ represents the XOR operation)

- **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

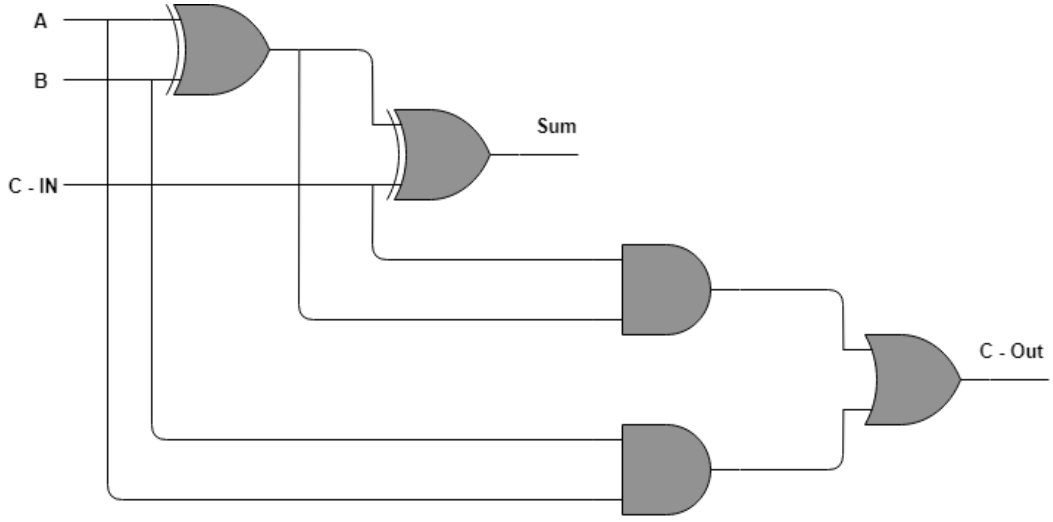
forms the basis of ripple-carry adders, which are commonly used in digital circuits for binary addition.



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.

b. 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:

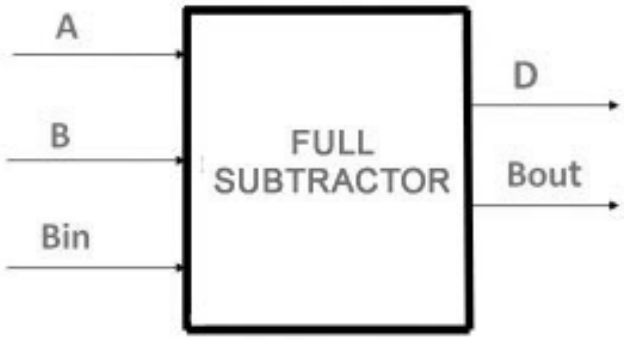
- **Difference (D)**:

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

Borrow-out (Bout):

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

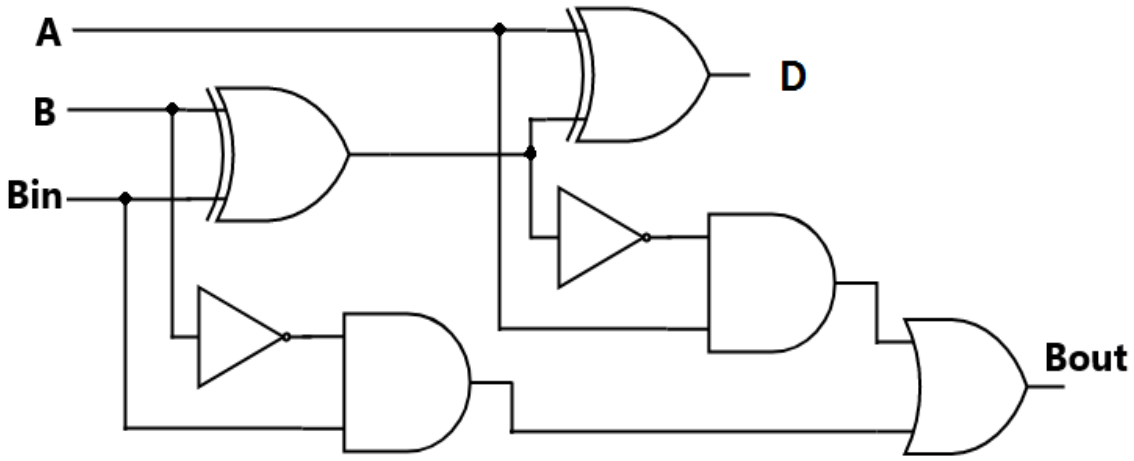
Full subtractors are important in digital circuits for arithmetic operations, particularly for subtracting multi-bit binary numbers. They are used in various computational units, such as in the design of arithmetic logic units (ALUs) and in binary subtraction operations.



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);
end
endmodule
```

Result:

Full Subtractor using Verilog is designed and simulated successfully

6. Multipliers.

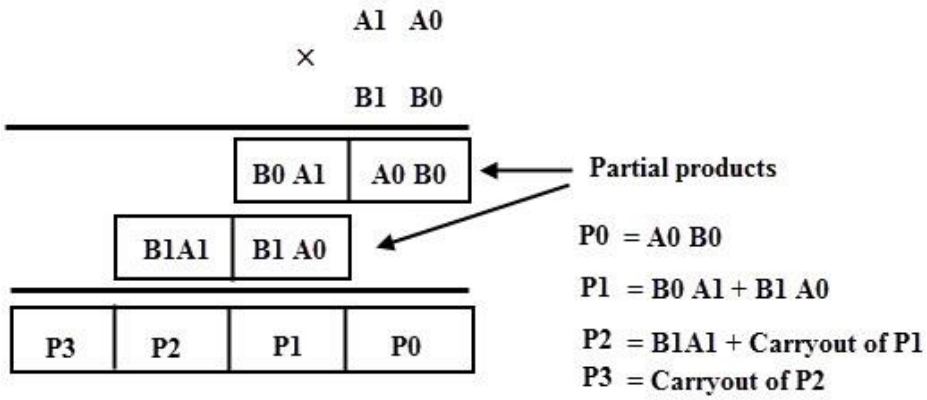
Aim:

To design multiplier using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Let us consider two unsigned 2 bit binary numbers A and B to generalize the multiplication process. The multiplicand A is equal to A1A0 and the multiplier B is equal to B1B0. The figure below shows the multiplication process of two 2 bit binary numbers.



This process involves the multiplication of two digits and the addition of digits with or without carry. After the multiplication of the each bit to the multiplicand, partial products are generated, and then these products are added to produce the total sum which represents the binary multiplication value.The first partial product is obtained by the AND gate which is nothing but a least significant bit of the multiplication result. Since the second partial product is shifted to the left position, the first partial second term and second partial product first term is added by half adder and produce the sum output along with the carry out.This carry out is added at the next half adder as an input as shown in figure. Likewise, it produces the multiplication result of two binary numbers by using the simple circuit configuration. The multiplication of the two 2 bit number results a 4-bit binary number.

Source Code:

Design Block:

```
module multiplier(a,b,q);
output [3:0]q;
input [1:0]a;
input [1:0]b;
wire [3:0]temp;
wire [1:0]x;
assign temp[0]=a[0]&b[0];
assign temp[1]=a[1]&b[0];
assign temp[2]=a[0]&b[1];
assign temp[3]=a[1]&b[1];
assign q[0]=temp[0];
half n1(temp[1],temp[2],q[1],x[0]);
half n2(temp[3],x[0],q[2],x[1]);
assign q[3]=x[1];
endmodule
module half(a,b,s,c);
input a,b;
output s,c;
assign s=a+b;
assign c=a&b;
endmodule
```

Test Bench:

```
module mul_tb;
wire [3:0]q;
reg [1:0]a;
reg [1:0]b;
```

```
multiplier h1(a,b,q);
initial
begin
a=3;b=3;
end
endmodule
```

Result:

Multiplier using Verilog is designed and is simulated successfully.

7. Parity Generators.

Aim: To design a parity generator using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

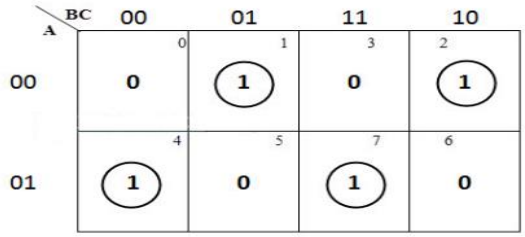
It is combinational circuit that accepts an n-1 bit stream data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is termed as a parity bit.In **even parity** bit scheme, the parity bit is '**0**' if there are **even number of 1s** in the data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

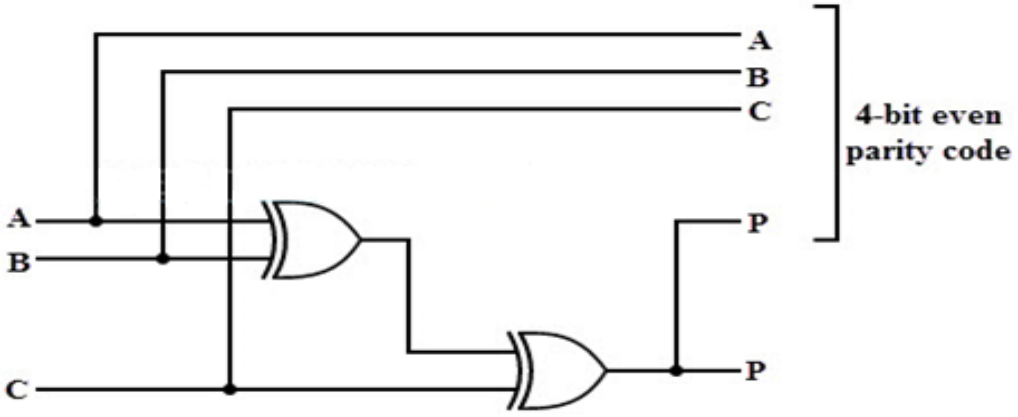
$$P = \bar{A}\,\bar{B}\,C + \bar{A}\,B\,\bar{C} + A\,\bar{B}\,\bar{C} + A\,B\,C$$

$$= \bar{A}\,(\bar{B}\,C + B\,\bar{C}) + A\,(\bar{B}\,\bar{C} + B\,C)$$

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

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

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
```

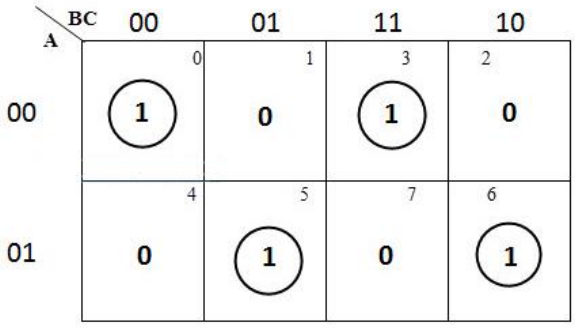
ODD Parity Generator:

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

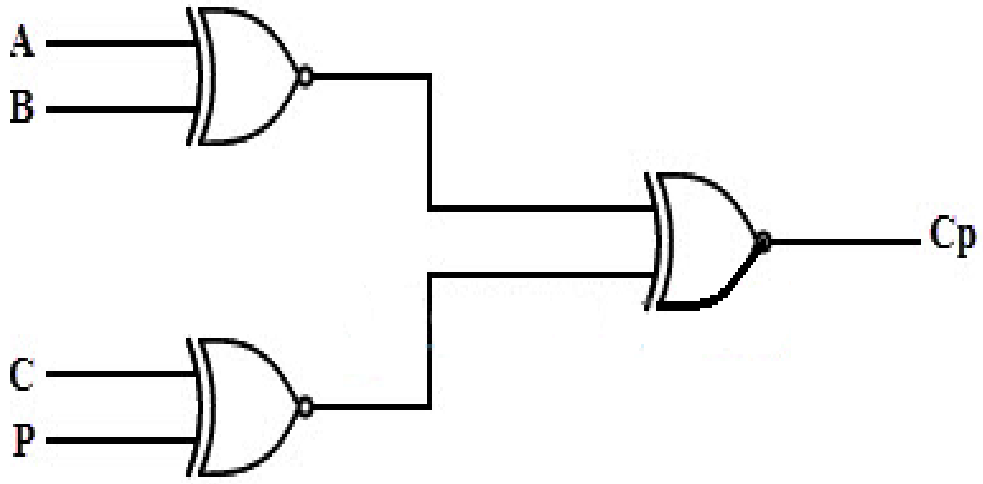
The truth table of the odd parity generator can be simplified by using K-map as



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.

8. Design of ALU.

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

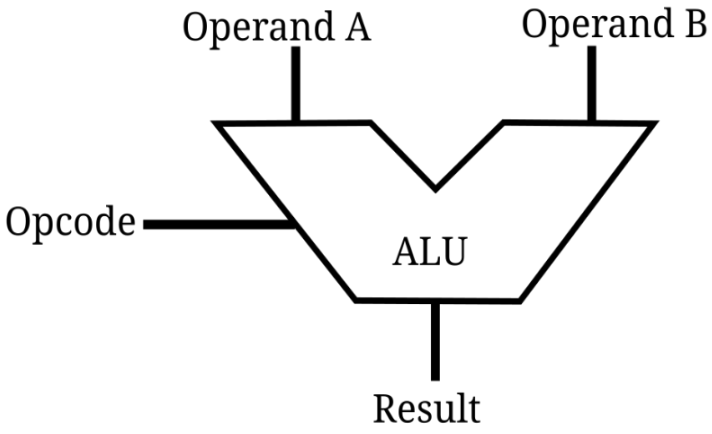
Software Used: Vivado 2016.4

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.

9. Latches.

Aim: To design a Latches using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Latches are basic memory devices used in digital electronics to store a single bit of information. They are fundamental building blocks in sequential logic circuits and serve as the basis for flip-flops, which are used in memory and storage devices. Latches are level-sensitive devices, meaning their output can change when the input changes as long as a control signal (like an enable signal) is active.

1. SR Latch (Set-Reset Latch)

The SR latch is one of the simplest types of latches, consisting of two cross-coupled NOR gates or NAND gates. It has two inputs:

- **S (Set)**: Used to set the latch output to 1.
- **R (Reset)**: Used to reset the latch output to 0.

Truth Table for SR Latch (NOR-based)

S (Set)	R (Reset)	Q (Output)	Description
0	0	Q (Previous)	No change (holds state)
0	1	0	Reset state (Q = 0)
1	0	1	Set state (Q = 1)
1	1	Undefined	Invalid condition

Characteristics:

- When both **S** and **R** are 0, the latch holds its previous state.

- The condition **S = 1** and **R = 1** is considered undefined or invalid because it causes a race condition.

2. D Latch (Data or Delay Latch)

The D latch, also known as the Data or Delay latch, is a modification of the SR latch that removes the invalid condition by using a single data input (D). It also has an enable input (EN) that controls when the data can be latched.

Working Principle

- **D (Data Input)**: The value at this input determines the output state when the latch is enabled.
- **EN (Enable Input)**: When EN is high, the latch is transparent, meaning the output follows the input D. When EN is low, the latch holds its previous state.

Truth Table for D Latch

D (Data)	EN (Enable)	Q (Output)	Description
0	0	Q (Previous)	Hold previous state
1	0	Q (Previous)	Hold previous state
0	1	0	Reset state (Q = 0)
1	1	1	Set state (Q = 1)

Characteristics:

- When **EN = 1**, the output directly follows the input D.
- When **EN = 0**, the output Q remains unchanged, making it a memory state.

3. JK Latch

The JK latch is an extension of the SR latch that eliminates the problem of the undefined state. It has two inputs, **J** and **K**, along

with an enable signal (EN). The JK latch can toggle its state based on the input conditions.

Truth Table for JK Latch

J	K	EN	Q (Output)	Description
0	0	1	Q (Previous)	No change (holds state)
0	1	1	0	Reset state (Q = 0)
1	0	1	1	Set state (Q = 1)
1	1	1	Toggle Q	Toggle state (invert Q)

Characteristics:

- When **J = K = 1**, the output toggles its state on each enable pulse.
- When **EN = 0**, the output remains in its previous state regardless of the inputs.

Source Code:

SR Latch

Source Code:

Design Block:

```
module sr_latch (
   input S,       // Set input
   input R,       // Reset input
   output Q,      // Output
   output Qn      // Complement of Q
);

   nor (Q, R, Qn);   // NOR gate to generate Q
   nor (Qn, S, Q);   // NOR gate to generate Q'
```

```
endmodule
```

Testbench:

```
module tb_sr_latch;

    reg S, R;            // Inputs to the SR latch
    wire Q, Qn;          // Outputs from the SR latch

    // Instantiate the SR latch module
    sr_latch uut (
        .S(S),
        .R(R),
        .Q(Q),
        .Qn(Qn)
    );

    initial begin
        $display("S R | Q Qn");
        S = 0; R = 0; #10;    // Hold state
        S = 1; R = 0; #10;    // Set state
        S = 0; R = 1; #10;    // Reset state
        S = 1; R = 1; #10;    // Invalid state
        $stop;
    end

endmodule
```

D Latch

Design Block:

```
module d_latch (
    input D,       // Data input
```

```
    input EN,      // Enable signal
    output Q,      // Output
    output Qn      // Complement of Q
);

    wire S, R;     // Internal signals for SR latch conversion

    assign S = D & EN;      // Set condition when D is 1 and EN is active
    assign R = ~D & EN;     // Reset condition when D is 0 and EN is active

    sr_latch sr_inst (      // Instantiating SR latch using the Set and Reset logic
        .S(S),
        .R(R),
        .Q(Q),
        .Qn(Qn)
    );

endmodule
```

Testbench:

```
module tb_d_latch;

    reg D, EN;          // Inputs to the D latch
    wire Q, Qn;         // Outputs from the D latch

    // Instantiate the D latch module
    d_latch uut (
        .D(D),
        .EN(EN),
        .Q(Q),
        .Qn(Qn)
```

```
    );

    initial begin
        $display("D EN | Q Qn");
        D = 0; EN = 0; #10;    // Hold state
        D = 0; EN = 1; #10;    // Reset state
        D = 1; EN = 1; #10;    // Set state
        D = 1; EN = 0; #10;    // Hold state
        $stop;
    end

endmodule
```

JK Latch

Design Block:

```
module jk_latch (
    input J,        // J input
    input K,        // K input
    input EN,       // Enable signal
    output reg Q    // Output
);

    always @ (J or K or EN) begin
        if (EN) begin
            case ({J, K})
                2'b00: Q = Q;       // No change
                2'b01: Q = 0;       // Reset
                2'b10: Q = 1;       // Set
```

```
            2'b11: Q = ~Q;      // Toggle
        endcase
    end
end


endmodule
```

Testbench:

```
module tb_jk_latch;

    reg J, K, EN;          // Inputs to the JK latch
    wire Q;               // Output from the JK latch

    // Instantiate the JK latch module
    jk_latch uut (
        .J(J),
        .K(K),
        .EN(EN),
        .Q(Q)
    );

    initial begin
        $display("J K EN | Q");
        J = 0; K = 0; EN = 1; #10;    // Hold state
        J = 0; K = 1; EN = 1; #10;    // Reset state
        J = 1; K = 0; EN = 1; #10;    // Set state
        J = 1; K = 1; EN = 1; #10;    // Toggle state
        EN = 0; #10;                  // Hold state
        $stop;
    end

endmodule
```

Result:

Latches in Verilog is designed and simulated successfully.

10. Flip-Flops.

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

Software Used: Vivado 2016.4

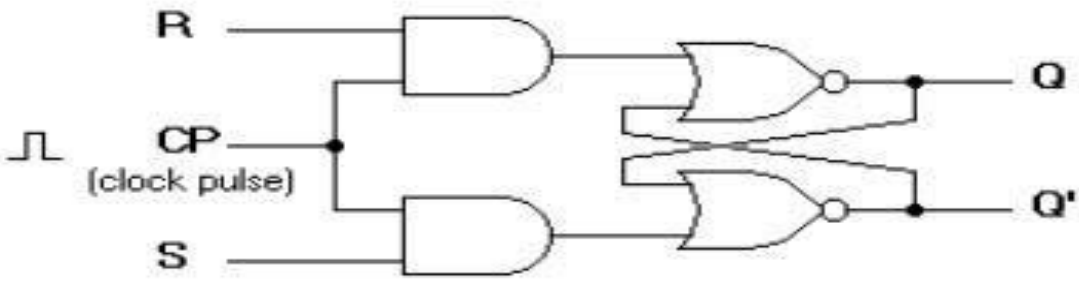
Theory:

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 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.



(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

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;
```

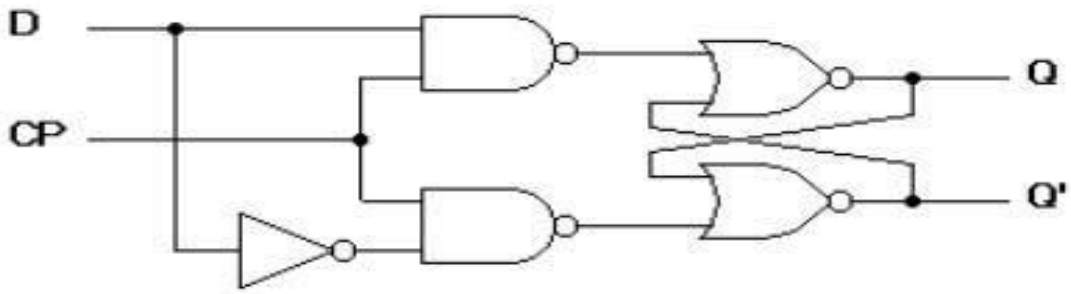
```
  else if(s==1'b0&&r==1'b0) q<=q;
  else if(s==1'b0&&r==1'b1) q<=1'b0;
  else if(s==1'b1&&r==1'b0) q<=1'b1;
  else if(s==1'b1&&r==1'b1) q<=1'bx;
  end
  assign qb=~q;
 endmodule
```

Test Bench:

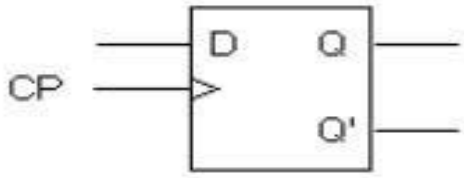
```
module srff_tb();
reg s,r,clk,rst;
wire q,qb;
sr_ff p(s,r,clk,rst,q,qb);
initial
begin
clk=0;
s=0; r=0;
#5 rst=1;
#30 rst=0;
$monitor($time,"clk=%b,rst=%b,s=%b,r=%b,q=%b,qb=%b",clk,rst,s,r,q,qb);
#100 $finish;
end
always #5 clk=~clk;
always #30 s=~s;
always #40 r=~r;
endmodule
```

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
```

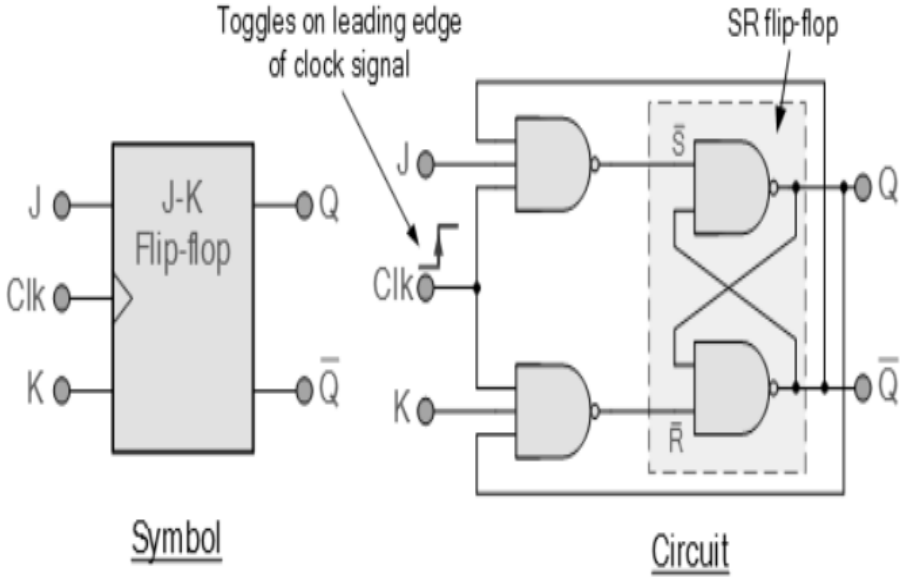
Test Bench:

```
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:

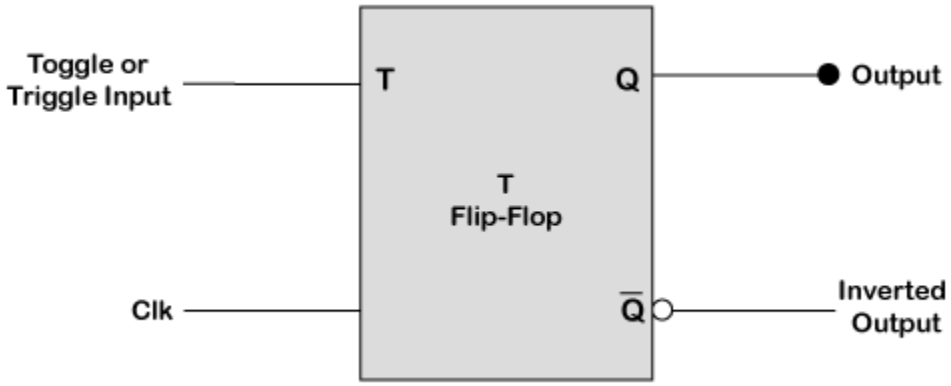
```
module JK_flipflop (
  input clk, rst_n,
  input j,k,
  output reg q,
  output q_bar
  );
  // always@(posedge clk or negedge rst_n) // for asynchronous reset
  always@(posedge clk) begin // for synchronous reset
    if(!rst_n) q <= 0;
    else begin
      case({j,k})
        2'b00: q <= q;    // 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
```

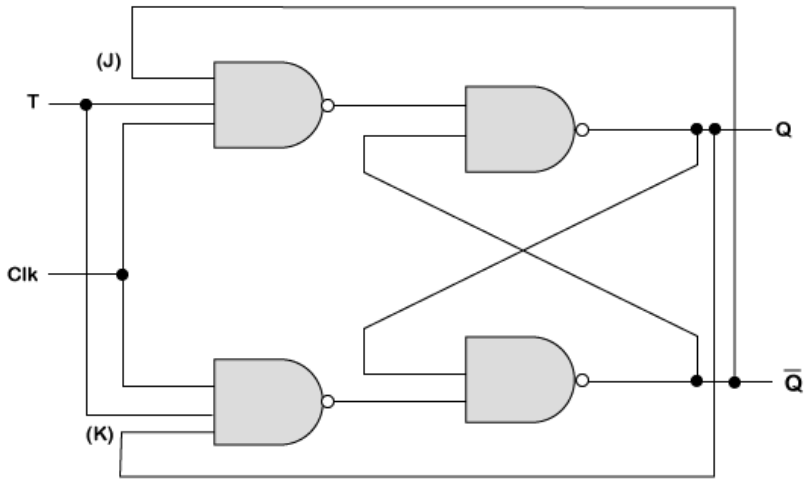
Testbench:

```
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

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.

11. Synchronous Counters.

Aim: To design a Synchronous Counters using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Synchronous counters are digital circuits in which all the flip-flops are clocked simultaneously by a common clock signal. Unlike asynchronous (ripple) counters, where the clock signal is applied to only the first flip-flop, synchronous counters ensure that all flip-flops are triggered at the same time, eliminating propagation delay issues.

Key Concepts of Synchronous Counters

1. **Clock Signal**: In a synchronous counter, the clock input is connected to all the flip-flops simultaneously. This ensures that all flip-flops are updated at the same time, leading to faster and more predictable operations.
2. **Flip-Flops**: Synchronous counters use flip-flops (usually JK or D-type) as their basic storage elements. The state of these flip-flops changes based on the input conditions.
3. **Enable Logic**: The output of each flip-flop in a synchronous counter depends not just on the current state but also on some combination of previous states. Enable logic is used to determine when each flip-flop should toggle.
4. **Count Sequence**: Synchronous counters can be designed to count in a specific sequence, such as binary, Gray code, or other custom

sequences. The count sequence determines how the states of the flip-flops change in response to the clock pulses.

5. **Modulus of the Counter**: The modulus (or mod) of a counter is the number of unique states it cycles through before returning to its starting state. For example, a mod-4 counter cycles through four states (0 to 3).

Applications of Synchronous Counters

- **Digital Clocks**: Used to keep track of time in a digital format.
- **Frequency Dividers**: Employed in circuits where a lower frequency clock signal is needed.
- **Counters in Digital Systems**: Used in various digital applications like measuring time intervals, event counting, and data synchronization.

Source Code:

Design Block:

```
module synchronous_counter #(parameter SIZE=4)(
  input clk, rst_n,
  input up,
  output reg [3:0] cnt);
  always@(posedge clk) begin
    if(!rst_n) begin
      cnt <= 4'h0;
    end
    else begin
      if(up) cnt <= cnt + 1'b1;
      else cnt <= cnt - 1'b1;
```

```
    end
  end
endmodule
```

Testbench:

```
module tb;
  reg clk, rst_n;
  reg up;
  wire [3:0] cnt;
  synchronous_counter(clk, rst_n, up, cnt);

  initial begin
    clk = 0; rst_n = 0;
    up = 1;
    #4; rst_n = 1;
    #80;
    rst_n = 0;
    #4; rst_n = 1; up = 0;
    #50;
    $finish;
  end
  always #2 clk = ~clk;
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule
```

Result:

Synchronous Counters in Verilog is designed and simulated successfully

12. Asynchronous Counters.

Aim: To design a Asynchronous Counters using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

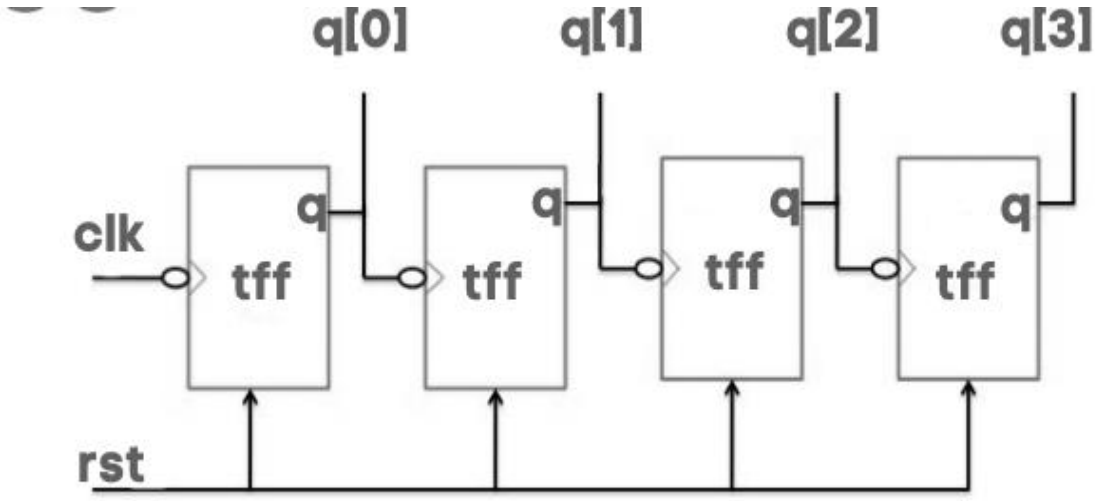
Asynchronous counters, also known as ripple counters, are digital circuits where flip-flops are connected in such a way that the output of one flip-flop acts as the clock input for the next flip-flop. In these counters, the flip-flops do not change states simultaneously, leading to a ripple effect through the circuit as each flip-flop toggles.

Key Concepts of Asynchronous Counters

1. **Clock Signal**: In an asynchronous counter, only the first flip-flop is directly connected to the clock signal. Subsequent flip-flops receive their clock signals from the output of the preceding flip-flop.
2. **Ripple Effect**: The term "ripple" refers to the way the signal propagates through the flip-flops. When the first flip-flop toggles, it triggers the next flip-flop in sequence, and so on, resulting in a small delay as the change moves through the flip-flops.
3. **Propagation Delay**: The sequential triggering of flip-flops causes a delay, known as propagation delay, which can affect the timing and accuracy of the counter. The delay accumulates with each additional flip-flop.
4. **Modulus of the Counter**: The modulus (or mod) of a counter is the number of unique states it cycles through before repeating. For example, a mod-8 counter counts from 0 to 7.

Applications of Asynchronous Counters

- **Simple Counting Tasks**: Suitable for low-speed applications like event counting, frequency division, and basic timing circuits.
- **Digital Clocks**: Used in situations where the propagation delay does not significantly affect the performance.
- **Frequency Division**: Used to generate lower frequency signals from a high-frequency clock.



Source Code:

Design Block:

```
module ripplecounter(clk,rst,q);
  input clk,rst;
  output [3:0]q;
  // initiate 4 T-FF to update the count
  tff tf1(q[0],clk,rst);
  tff tf2(q[1],q[0],rst);
  tff tf3(q[2],q[1],rst);
  tff tf4(q[3],q[2],rst);
```

```
endmodule

module tff(q,clk,rst);
  // tff takes clk and reset as input
  // q is output
  input clk,rst;
  output q;
  wire d;
  // by referring the diagram of tff,
  // instantiate d flip flop and not gate
  dff df1(q,d,clk,rst);
  not n1(d,q);
endmodule

module dff(q,d,clk,rst);
  input d,clk,rst;
  output q;
  reg q; // store the output value
  always @(posedge clk or posedge rst)
   begin
    // refer the truth table to provide
    // values to q based on reset.
    if(rst) q=1'b0;
    else q=d;
   endripplecounter
endmodule
```

Testbench:

```
module tb;
    // input to be stored in reg and output as net(wire)
    reg clk;
```

```
    reg rst;
    wire [3:0]q;
    // instantiate the ripplecounter design block
    ripplecounter dut(clk,rst,q);
    // generate clock pulse
    // initially provide 0
    // then inside always block toggle
    // clock every 5 time units
    initial
      clk = 0;
    always
      #5 clk = ~clk;
    // provide reset values as the input
    initial
       begin
         rst = 1;
         #15 rst = 0;
         #180 rst = 1;
         #10 rst = 1;
         #20 $finish;
       end
       initial
        $monitor("time=%g,rst=%b,clk=%b,q=%d",$time,rst,clk,q);
endmodule
```

Result:

Asynchronous Counters in Verilog is designed and simulated successfully

13. Shift Registers.

Aim: To design a Shift Registers using Verilog HDL.

Software Used: Vivado 2016.4

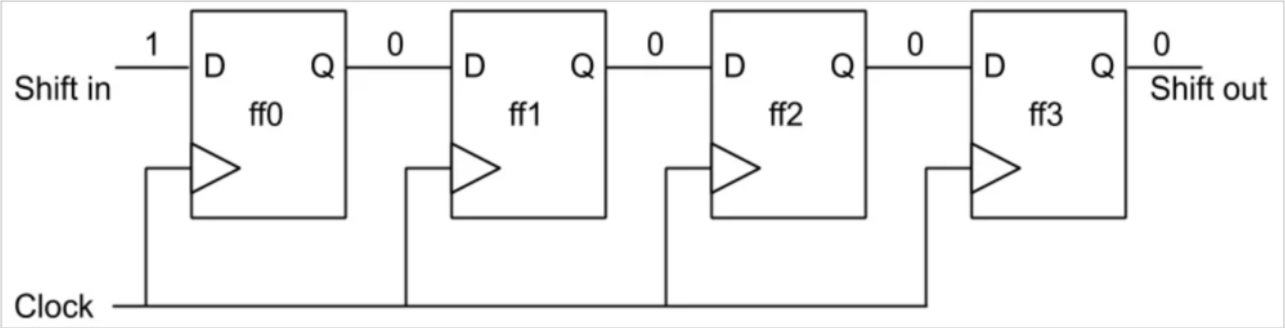
Theory:

A shift register is a type of digital circuit that can store and shift data in a sequential manner. It is a cascade of flip-flops, where the output of one flip-flop is connected to the input of the next flip-flop. The data is shifted from one flip-flop to the next on every clock cycle, allowing the circuit to delay or store the data. Please find below the Verilog code of a shift register:

Shift registers can be used in a variety of applications, such as:

1. **Serial-to-parallel conversion:** Shift registers can be used to convert a serial data stream into a parallel data stream. By loading the serial data into the shift register and shifting it out in parallel, the data can be processed more quickly and efficiently.
2. **Data storage:** Shift registers can be used to store data in a sequential manner. By loading the data into the shift register and clocking it through, the data can be retained for a period of time before it is shifted out.
3. **Arithmetic operations:** Shift registers can be used in arithmetic operations, such as multiplication and division. By shifting the data through the circuit and performing logical operations on the bits, the arithmetic operations can be performed in a more efficient manner.

4. **Frequency division:** Shift registers can be used to divide the frequency of a clock signal. By tapping into different points in the shift register, multiple clock signals can be generated with different frequencies.



Source Code:

Design Block:

```
module ShiftRegister4Bit (
    input wire clk,          // Clock signal
    input wire reset,        // Asynchronous reset signal
    input wire shift_in,     // Serial input
    output reg [3:0] q       // 4-bit parallel output
);
// Shift register operation
always @(posedge clk or posedge reset) begin
    if (reset) begin
        q <= 4'b0000;        // Reset the register to 0
    end else begin
        q <= {q[2:0], shift_in};  // Shift right and insert shift_in
    end
end
endmodule
```

Testbench:

```
module tb_ShiftRegister4Bit();
    reg clk;              // Test clock signal
```

```
reg reset;            // Test reset signal
reg shift_in;         // Test serial input
wire [3:0] q;         // Test output

// Instantiate the ShiftRegister4Bit module
ShiftRegister4Bit uut (
    .clk(clk),
    .reset(reset),
    .shift_in(shift_in),
    .q(q)
);
// Clock generation: 10 ns period
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
// Test stimulus
initial begin
    // Initialize inputs
    reset = 1;       // Activate reset
    shift_in = 0;    // Initial serial input value

    #10 reset = 0;   // Deactivate reset
    // Apply test pattern
    #10 shift_in = 1;
    #10 shift_in = 0;
    #10 shift_in = 1;
    #10 shift_in = 1;
    #10 shift_in = 0;

    // Wait and then apply reset
```

```
        #20 reset = 1;
        #10 reset = 0;
        // End of test
        #30 $stop;
    end
    // Monitor outputs
    initial begin
        $monitor("Time=%0d | reset=%b | shift_in=%b | q=%b", $time, reset,
shift_in, q);
    end
endmodule
```

Result: Shift Registers in Verilog is designed and simulated successfully.

14. Memories.

Aim: To design a Memories using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

Memory is a fundamental component of digital systems used for data storage and retrieval. In digital electronics, memory stores binary information (0s and 1s) and is classified into different types based on its functionality and architecture. Understanding the types of memories and their operations is crucial for designing digital systems, such as computers, microcontrollers, and other embedded systems.

Types of Memories

The two primary types of memories in digital systems are **Read-Only Memory (ROM)** and **Random Access Memory (RAM)**. Each has its specific characteristics and uses.

1. Read-Only Memory (ROM)

ROM is a type of non-volatile memory that stores data permanently. It retains its data even when the power is turned off. The information stored in ROM is written during manufacturing and cannot be easily altered or erased.

- **Characteristics of ROM**:
    - **Non-volatile**: Retains data even without power.
    - **Pre-programmed**: Data is usually programmed during the manufacturing process.

- **Used for firmware**: Commonly used to store firmware or software that controls the hardware of a device.

Random Access Memory (RAM)

RAM is a type of volatile memory used to store data temporarily while a computer or other digital device is powered on. RAM allows data to be read from and written to any memory location in a random order, providing fast data access.

- **Characteristics of RAM**:
  - **Volatile**: Loses its data when the power is turned off.
  - **Read/Write memory**: Allows both reading and writing of data.
  - **Fast access time**: Provides quick access to data compared to other storage types.

Source Code:

Design Block:

ROM (Read-Only Memory):

```
module SimpleROM (
    input wire [3:0] address,     // 4-bit address input
    output reg [7:0] data         // 8-bit data output
);
    // ROM memory definition (16 locations of 8 bits each)
    reg [7:0] memory [0:15];
    // Initialize ROM contents
    initial begin
        memory[0] = 8'b00000001;
        memory[1] = 8'b00000010;
        memory[2] = 8'b00000100;
        memory[3] = 8'b00001000;
```

```
        memory[4] = 8'b00010000;
        memory[5] = 8'b00100000;
        memory[6] = 8'b01000000;
        memory[7] = 8'b10000000;
        memory[8] = 8'b11111111;
        memory[9] = 8'b11000011;
        memory[10] = 8'b10101010;
        memory[11] = 8'b10011001;
        memory[12] = 8'b01100110;
        memory[13] = 8'b01010101;
        memory[14] = 8'b00110011;
        memory[15] = 8'b00001111;
    end
    // Data output based on address input
    always @(*) begin
        data = memory[address];
    end
endmodule
```

Testbench:

```
module tb_SimpleROM();
    reg [3:0] address;
    wire [7:0] data;
    // Instantiate the SimpleROM module
    SimpleROM uut (
        .address(address),
        .data(data)
    );
    initial begin
        $display("Time\t Address | Data");
        $display("-------------------------");
```

```
        $monitor("%0d\t %b | %b", $time, address, data);
        // Apply test patterns
        address = 4'b0000; #10;
        address = 4'b0001; #10;
        address = 4'b0010; #10;
        address = 4'b0011; #10;
        address = 4'b0100; #10;
        address = 4'b0101; #10;
        address = 4'b0110; #10;
        address = 4'b0111; #10;
        address = 4'b1000; #10;
        address = 4'b1111; #10;
        $stop;
    end
endmodule
```

RAM (Random Access Memory):

Design Block:

```
module SimpleRAM (
    input wire clk,                 // Clock input
    input wire write_enable,        // Write enable signal
    input wire [3:0] address,       // 4-bit address input
    input wire [7:0] write_data,    // 8-bit data input for writing
    output reg [7:0] read_data      // 8-bit data output for reading
);
    // RAM memory definition (16 locations of 8 bits each)
    reg [7:0] memory [0:15];
    // Read and write operations
    always @(posedge clk) begin
        if (write_enable) begin
            memory[address] <= write_data;  // Write data to memory
```

```
        end
        read_data <= memory[address];       // Read data from memory
    end
endmodule
```

Testbench:

```
module tb_SimpleRAM();
    reg clk;
    reg write_enable;
    reg [3:0] address;
    reg [7:0] write_data;
    wire [7:0] read_data;
    // Instantiate the SimpleRAM module
    SimpleRAM uut (
        .clk(clk),
        .write_enable(write_enable),
        .address(address),
        .write_data(write_data),
        .read_data(read_data)
    );
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // Clock period of 10 time units
    end
    initial begin
        $display("Time\t WE Address | WriteData | ReadData");
        $display("-----------------------------------------");
        $monitor("%0d\t %b  %b    | %b | %b", $time, write_enable, address,
write_data, read_data);
```

```
        // Test write operation
        write_enable = 1;
        address = 4'b0000; write_data = 8'b10101010; #10;
        address = 4'b0001; write_data = 8'b01010101; #10;
        address = 4'b0010; write_data = 8'b11110000; #10;

        // Test read operation
        write_enable = 0;
        address = 4'b0000; #10;
        address = 4'b0001; #10;
        address = 4'b0010; #10;

        $stop;
    end

endmodule
```

Result: Memories in Verilog is designed and simulated successfully.

15. CMOS Circuits

Aim: To design a CMOS Circuits using Verilog HDL.

Software Used: Vivado 2016.4

Theory:

CMOS (Complementary Metal-Oxide-Semiconductor) technology is the most widely used technology in the fabrication of integrated circuits (ICs), including microprocessors, microcontrollers, memory chips, and other digital logic circuits. It is known for its low power consumption, high noise immunity, and scalability, making it suitable for a wide range of applications in modern electronics.

Basics of CMOS Technology

CMOS circuits are built using two types of metal-oxide-semiconductor field-effect transistors (MOSFETs):

1. **PMOS (P-type MOSFET) Transistors**: These conduct when the gate voltage is low (0 or ground) and turn off when the gate voltage is high (Vdd or positive voltage).
2. **NMOS (N-type MOSFET) Transistors**: These conduct when the gate voltage is high (Vdd) and turn off when the gate voltage is low (0 or ground).

In CMOS technology, these two transistors are arranged in a complementary manner to create logic gates and digital circuits. This complementary setup is what gives CMOS circuits their low power consumption and high switching speed.

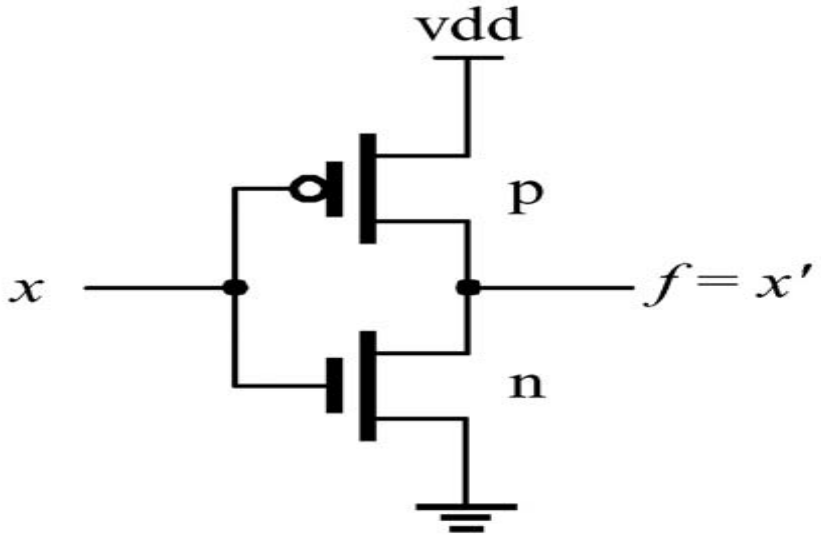
The fundamental principle of CMOS technology is that there are always two paths available for the current flow:

- When the circuit is in one logic state (e.g., logic '0'), one type of transistor (PMOS or NMOS) is turned on while the other is off.
- When the circuit switches to the other logic state (e.g., logic '1'), the roles of the transistors are reversed.

Source Code:

CIRCUIT DIAGRAM

A) CMOS INVERTER



Design Block:

```
module ine233(
   input x,
   output f    );
   supply1 vdd;
   supply0 gnd;
   // NOT gate body
   pmos p1(f,vdd,x);
```

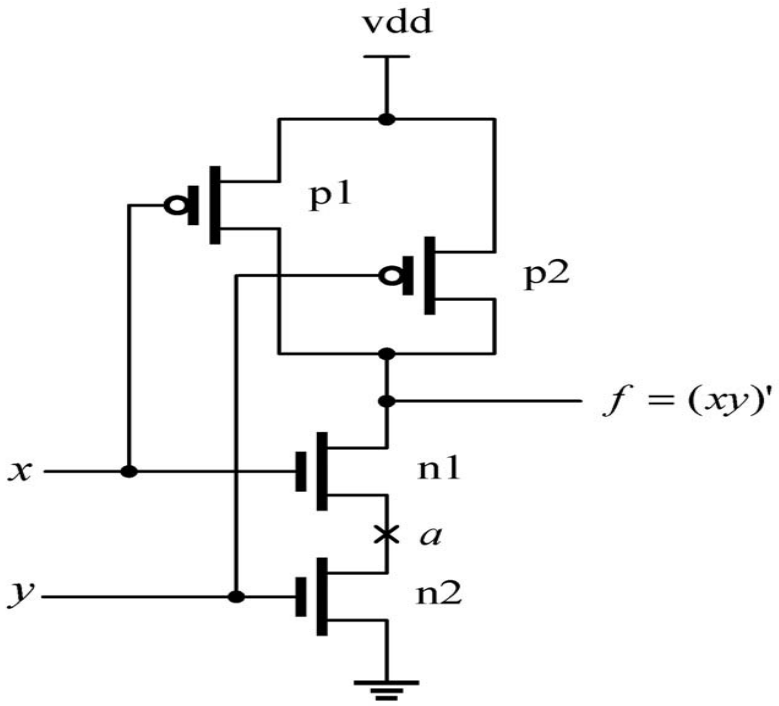
```
    nmos n1(f,gnd,x);
endmodule
```

Testbench:

```
module tb_ine233;

    reg x;          // Declare input as a register
    wire f;         // Declare output as a wire

    // Instantiate the CMOS inverter module
    ine233 uut (
        .x(x),
        .f(f)
    );

    initial begin
        // Display the header for the output table
        $display("Time\t x | f");
        $display("-------------");

        // Monitor the inputs and outputs
        $monitor("%0d\t %b | %b", $time, x, f);

        // Apply test patterns to the input
        x = 0; #10;    // Set input to 0, wait for 10 time units
        x = 1; #10;    // Set input to 1, wait for 10 time units

        $stop;         // Stop the simulation
    end

endmodule
```

B) CMOS WITH NAND

CIRCUIT DIAGRAM



Design Block:

```
module cmosnand2(
   input x,
   input y,
   output f
   );
   supply1 vdd;
   supply0 gnd;
   wire a;
   // NAND gate body
   pmos p1 (f, vdd, x);
   pmos p2 (f, vdd, y);
   nmos n1 (f, a, x);
   nmos n2 (a, gnd, y);
   endmodule
```

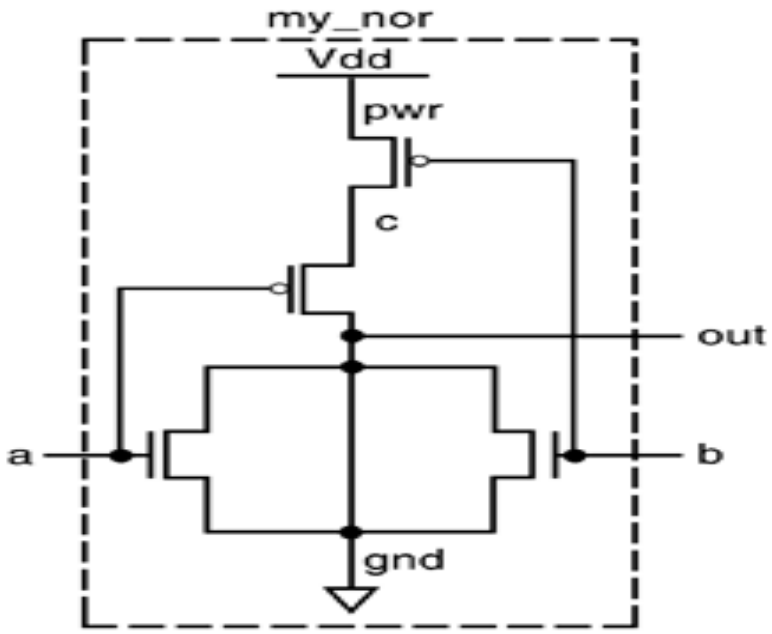
Testbench:

```
module tb_cmosnand2;

    reg x, y;            // Declare inputs as registers
    wire f;              // Declare output as a wire

    // Instantiate the CMOS NAND gate module
    cmosnand2 uut (
        .x(x),
        .y(y),
        .f(f)
    );

    initial begin
        // Display the header for the output table
        $display("Time\t x y | f");
        $display("-----------------");

        // Monitor the inputs and outputs
        $monitor("%0d\t %b %b | %b", $time, x, y, f);

        // Apply test patterns to the inputs
        x = 0; y = 0; #10;    // Inputs 00, wait for 10 time units
        x = 0; y = 1; #10;    // Inputs 01, wait for 10 time units
        x = 1; y = 0; #10;    // Inputs 10, wait for 10 time units
        x = 1; y = 1; #10;    // Inputs 11, wait for 10 time units
```

```
        $stop;              // Stop the simulation
    end

endmodule
```

C) CMOS WITH NOR

CIRCUIT DIAGRAM



Design Block:

```
module cmosnor11(
    input x,
    input y,
    output f
    );
    supply1 vdd;
    supply0 gnd;
    wire a;
    // NOR gate body
```

```
    pmos p1 (a, vdd, y);
    pmos p2 (f, a, x);
    nmos n1 (f, gnd, x);
    nmos n2 (f, gnd, y);
endmodule
```

Testbench:

```
module tb_cmosnor11;

    reg x, y;            // Declare inputs as registers
    wire f;              // Declare output as a wire

    // Instantiate the CMOS NOR gate module
    cmosnor11 uut (
        .x(x),
        .y(y),
        .f(f)
    );

    initial begin
        // Display the header for the output table
        $display("Time\t x y | f");
        $display("-----------------");

        // Monitor the inputs and outputs
        $monitor("%0d\t %b %b | %b", $time, x, y, f);
```

```
    // Apply test patterns to the inputs
    x = 0; y = 0; #10;    // Inputs 00, wait for 10 time units
    x = 0; y = 1; #10;    // Inputs 01, wait for 10 time units
    x = 1; y = 0; #10;    // Inputs 10, wait for 10 time units
    x = 1; y = 1; #10;    // Inputs 11, wait for 10 time units

    $stop;                // Stop the simulation
  end

endmodule
```

Result: CMOS Circuits in Verilog is designed and simulated successfully.