Lab Code:20ECL603
VLSI DESIGN
Lab Manual

PREPARED BY
MAHABOOB SUBHANI SHAIK

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 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.

BAPATLA ENGINEERING COLLEGE:: BAPATLA

(Autonomous)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

VLSI DESIGN LAB

III B.Tech. VI Semester (Code: 20ECL603)

Lectures :	0 Hours/Week	Tutorial :	0 Hours/Week	Practical :	3 Hours/Week
CIE Marks :	30	SEE Marks :	70	Credits :	1.5

Pre-Requisite: None.

Course Objectives: Students will be able to	
➢	Discuss basic language features of verilog HDL and the role of HDL in digital logic design.
➢	Describe the steps involved in synthesis and simulation of verilog HDL code.
➢	Design combinational circuits using HDL Programming Language.
➢	Design sequential circuits using HDL Programming Language

Course Outcomes: At the end of the course, student will be able to	
CO1	Demonstrate the basics of Hardware Description Languages, Program structure and basic language elements of Verilog.
CO2	Simulate various Combinational circuits in Dataflow, Behavioral and Gate level Abstractions.
CO3	Design sequential circuits like flip flops and counters in Behavioral description and obtain simulation waveforms.
CO4	Synthesize Combinational and Sequential circuits using verilog HDL.

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes

	PO's												**PSO's**		
CO	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
CO1	3	3			3				3	3				3	
CO2	3	3			3				3	3				3	
CO3	3	3	3		3				3	3				3	
CO4	3	3			3				3	3				3	
AVG	3	3	3		3				3	3				3	

LIST OF EXPERIMENTS 36 Hours

Write the Code using VERILOG, Simulate and synthesize the following:

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) experiments have to be performed and recorded by the candidate to attain eligibility for Semester End Lab Examination.*

1. LOGIC GATES

Aim: Write Verilog HDL code for the Verification of LOGIC GATES

Theory:

Logic gates are basic building blocks of digital circuits that perform logical operations on one or more binary inputs to produce a single binary output. Each gate implements a specific function, such as AND, OR, or NOT, which determines the output based on the combination of inputs.

AND Operation:

The binary AND operation has two inputs and one output. It is like the ADD operation, which takes two arguments (two inputs) and produces one result (one output). The inputs to a binary AND operation can only be 0 or 1 and the result can only be 0 or 1

The binary AND operation (also known as the binary AND function) will always produce a 1 output if both of its inputs are 1 and will produce a 0 output if one or both of its inputs are 0.

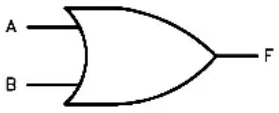
INPUT		OUTPUT
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Fig.1.1 AND GATE LOGIC SYMBOL & TRUTH TABLE

OR Operation:

The binary OR operation has two inputs and one output. The inputs to a binary OR operation can only be 0 or 1 and the result can only be 0 or 1

The binary OR operation (also known as the binary OR function) will always produce a 1 output if either of its inputs are 1 and will produce a 0 output if both of its inputs are 0.



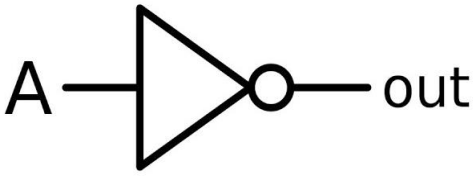
INPUT		OUTPUT
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Fig.1.2 OR gate logic symbol & truth table

NOT Operation:

The binary NOT operation has one input and one output. It is like the NEGATIVE operation which takes one argument (one input) and produces one result (one output). The input to a binary NOT operation can only be 0 or 1 and the result can only be 0 or 1

The binary NOT operation (also known as the binary NOT function or complement function or bit invert function) will always produce a 1 output if its input is 0 and will produce a 0 output if its input is 1.



Input	Output
A	Y
0	1
1	0

Fig.1.3 NOT gate logic symbol & truth table

NAND Operation:

The binary NAND operation has two inputs and one output. It is like the NEGATIVE operation which performs negation of the AND operationand produces one result (one output). The input to a binary NAND operation can only be 0 or 1 and the result can only be 0 or 1.

The binary NAND operation (also known as the binary NAND function or complement AND functionwill always produce complement of AND function.

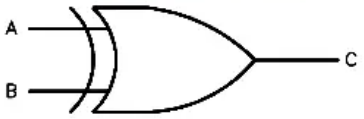
INPUT		OUTPUT
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Fig 1.4: NAND gate logic symbol & truth table

XOR Operation:

The binary XOR (exclusive OR) operation has two inputs and one output. It is like the comparison operation, which takes two arguments (two inputs) and produces one result (one output). The inputs to a binary XOR operation can only be 0 or 1, and the result can only be 0 or 1.

The binary XOR operation (also known as the binary XOR function) will always produce a 1 output if the two inputs are different and will produce a 0 output if both inputs are the same.



INPUT		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Fig 1.5: XOR gate logic symbol & truth table

XNOR Operation:

The binary XNOR (exclusive NOR) operation has two inputs and one output. It is like the comparison operation, which takes two arguments (two inputs) and produces one result (one output). The inputs to a binary XNOR operation can only be 0 or 1, and the result can only be 0 or 1.

The binary XNOR operation (also known as the binary XNOR function) will always produce a 1 output if both of its inputs are the same (either both 0 or both 1) and will produce a 0 output if the inputs are different.

INPUT		OUTPUT
A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Fig 1.6: XOR gate logic symbol & truth table

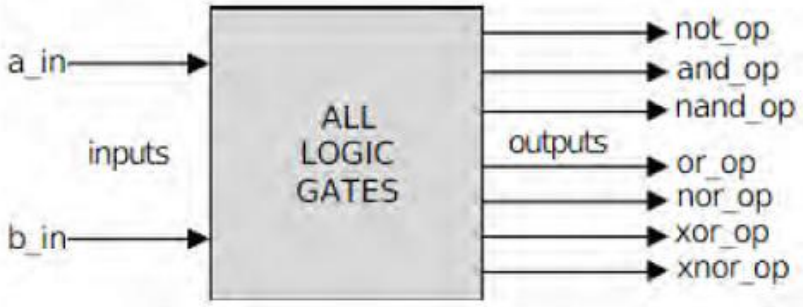


Fig 1.7: Block diagram

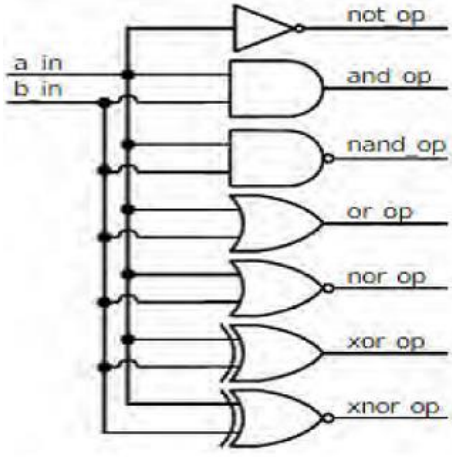


Fig 1.8: Logic diagram

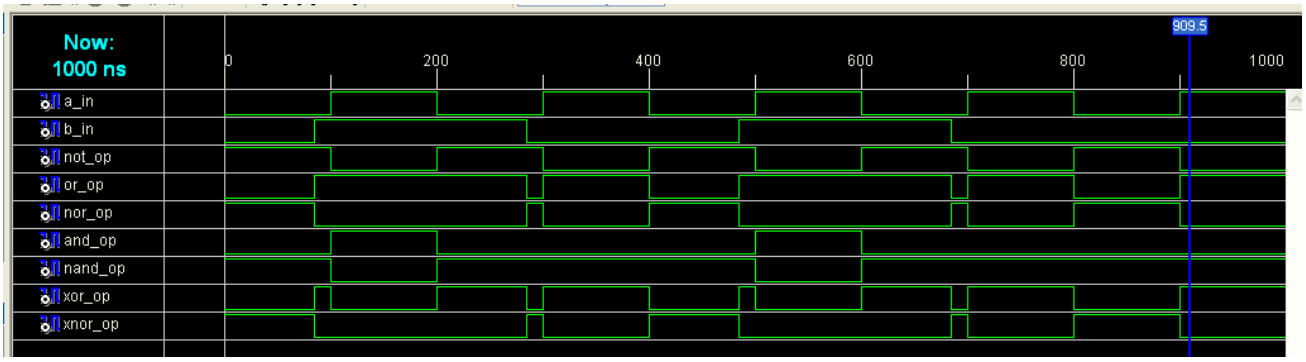
Table 1.1 Truth table For all Logic gates:

Input		Output						
a_in	b_in	not_op	and_op	nand_op	or_op	nor_op	xor_op	xnor_op
0	0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	1

VERILOG CODE :

```
module gates(a_in, b_in,
not_op,and_op,nand_op,or_op,nor_op,xor_op,xnor_op);
input a_in,b_in;
output
not_op,and_op,nand_op,or_op,nor_op,xor_op,xnor_op;
assign not_op= ~a_in;
assign and_op=a_in&b_in;
assign nand_op=~(a_in&b_in);
assign or_op=a_in | b_in;
assign nor_op=~(a_in | b_in);
assign xor_op=a_in^b_in;
assign xnor_op=~(a_in^b_in);
endmodule
```

OUTPUT:



2. MULTIPLEXER/DEMULTIPLEXER

Aim: Write Verilog HDL code for the Verification of multiplexer/demultiplexer.

Theory:

MULTIPLEXER

It is a combinational circuit which have many data inputs and single output depending on control or select inputs. For N input lines, log n (base2) selection lines, or we can say that for 2^n input lines, n selection lines are required. Multiplexers are also known as "Data n selector, parallel to serial convertor, many to one circuit, universal logic circuit". Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.

DEMULTIPLEXER

A demultiplexer (or demux) is a digital circuit that takes a single input signal and routes it to one of several output lines, based on control signals. It essentially performs the reverse operation of a multiplexer, distributing data from one input to multiple outputs.

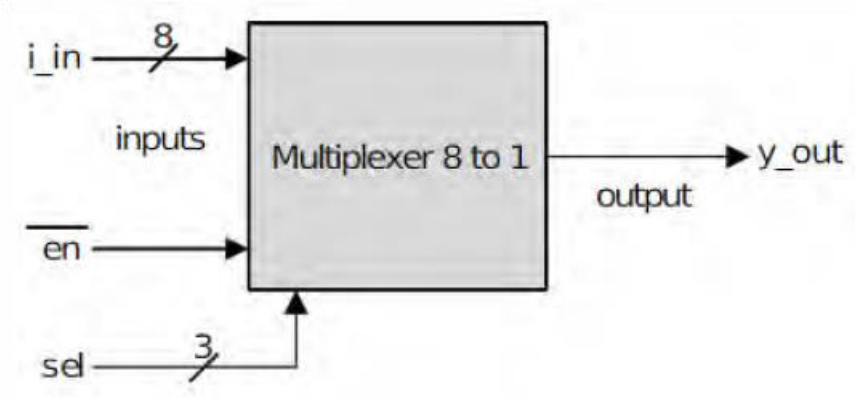


Fig 2.1: 8x1 MUX

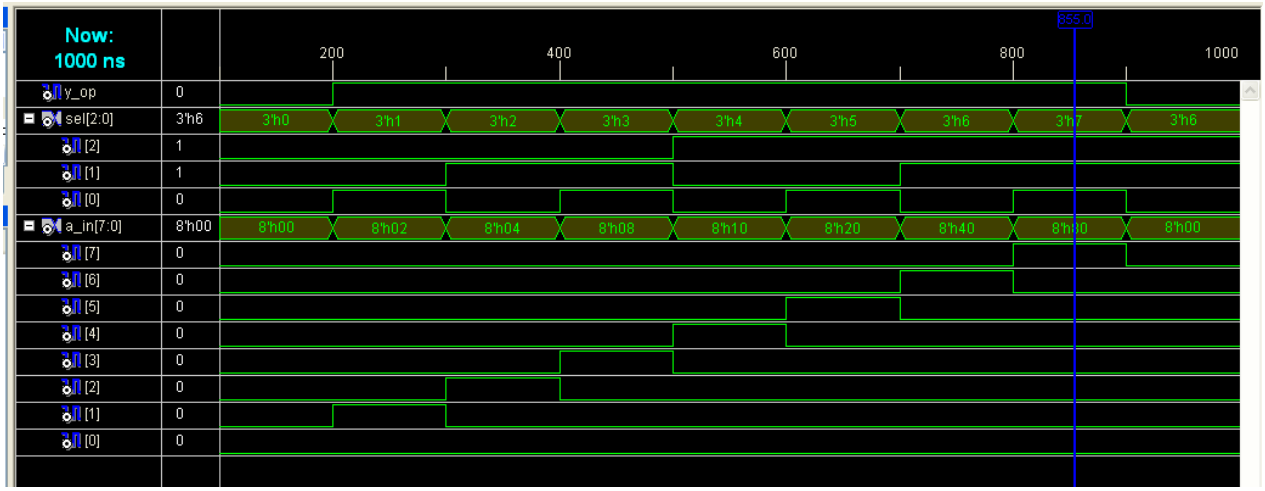
Table 2.1: 8x1 MUX TRUTH TABLE

Inputs											**Output**
Sel(2)	Sel(1)	Sel(0)	i_in(7)	i_in(6)	i_in(5)	i_in(4)	i_in(3)	i_in(2)	i_in(1)	i_in(0)	Y_out
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	1

VERILOG CODE:

```
module mux8_1(i_in, sel, y_out);
input [7:0] a_in;
input [2:0] sel;
output y_out;
reg y_out;
always@ (i_in,sel)
begin
case (sel)
      3'b000:y_out=i_in[0];
      3'b001: y_out=i_in[1];
      3'b010: y_out=i_in[2];
      3'b011: y_out=i_in[3];
      3'b100: y_out=i_in[4];
      3'b101: y_out=i_in[5];
      3'b110: y_out=i_in[6];
      3'b111: y_out=i_in[7];
      default: y_out =3'b000;
endcase
end
endmodule
```

8x1 MUX OUTPUT:



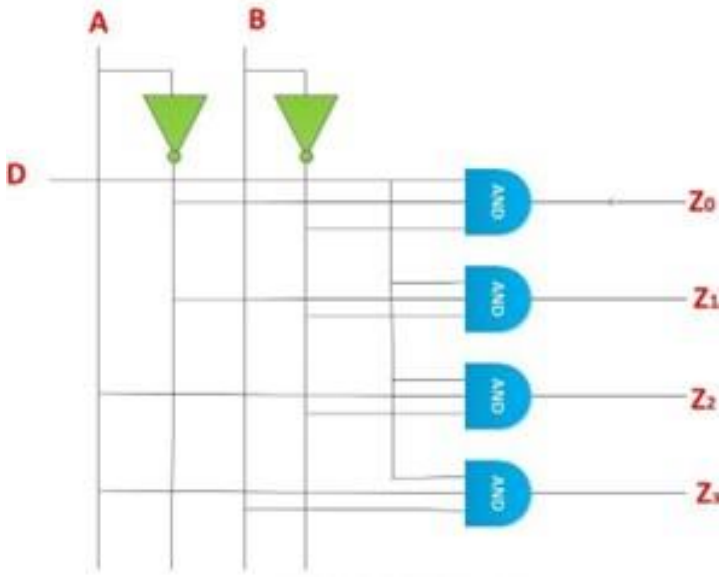


Fig 2.2: 1X4 DEMUX

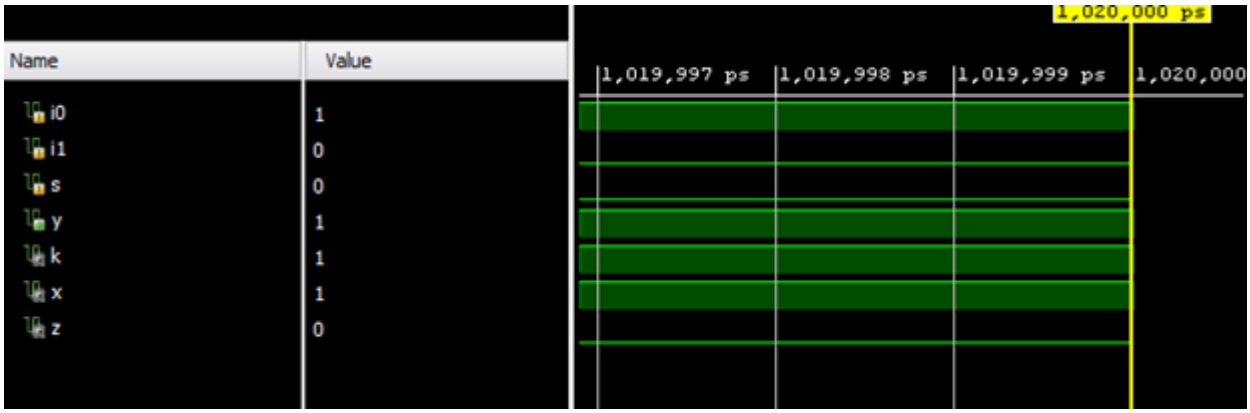
Table 2.2: 1X4 Demux Truth Table

Inputs			outputs			
A	B	D	Z0	Z1	Z2	Z3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

VERILOG CODE:

```
module demux_41(
        input i,
        input s0,
        input s1,
        output z0,
        output z1,
        output z2,
        output z3
        );
        wire s1bar,s0bar;
        not(s0bar,s0);
        not(s1bar,s1);
        and n1(z0,s0bar,s1bar);
        and n2(z1,s1bar,s0);
        and n3(z2,s1,s0bar);
        and n4(z3,s1,s0);
endmodule
```

1X4 Demux Output:


Name
Value
i0
1
i1
0
s
0
y
1
k
1
x
1
z
0
1,020,000 ps
1,019,997 ps
1,019,998 ps
1,019,999 ps
1,020,000

3. ENCODER/DECODER

Aim: Write Verilog HDL code for the Verification of 8:3 Encoder & 2:4Decoder.

Theory:

Encoder

In VLSI (Very Large Scale Integration), an encoder is a combinational circuit that converts information from one format or code into another, typically by reducing the number of input lines. For example, an encoder takes multiple input lines and encodes them into a smaller number of output lines, representing the active input in a binary form. This is useful in digital systems to simplify and optimize signal processing and data handling.

A common example is the priority encoder, which outputs the binary code of the highest-priority active input.

Decoder

In VLSI (Very Large Scale Integration), adecoder is a combinational circuit that converts encoded binary data from its input into a unique output pattern. It performs the reverse operation of an encoder, taking a small number of inputs (usually in binary form) and activating one specific output line based on the input value.

For example, a 2-to-4 decoder takes 2 binary input lines and decodes them to activate one of 4 output lines. Decoders are commonly used in memory address decoding and other digital systems where specific control signals need to be activated based on input conditions.

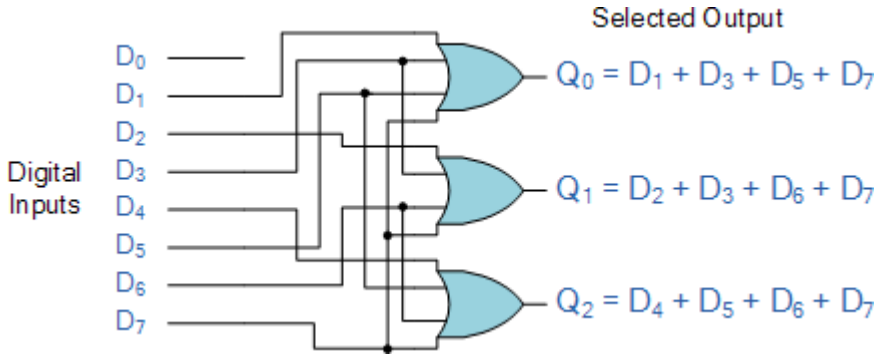


Fig 3.1: 8:3 Encoder

Table 3.1: 8:3 Encoder truth table:

Digital Inputs								Binary outputs		
D7	D6	D5	D4	D3	D2	D1	D0	Y2	Y1	Y0
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	00	00	0	0	0	0	1	1	1

VERILOG CODE:

```
module encoder_8_3(
        input d7,
        input d6,
        input d5,
        input d4,
        input d3,
        input d2,
        input d1,
        input d0,
        output y2,
        output y1,
        output y0
        );
        or n1(y0,d3,d2,d1,d0);
        or n2(y1,d5,d4,d1,d0);
        or n3(y2,d6,d4,d2,d0);
endmodule
```

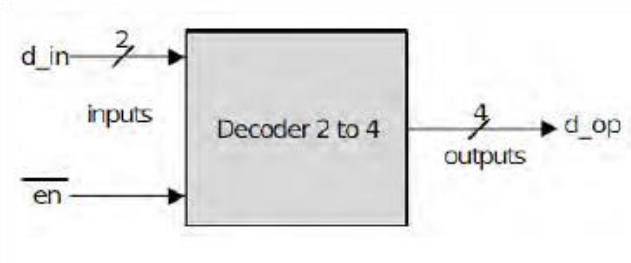


Fig3.2: 2:4 Decoder

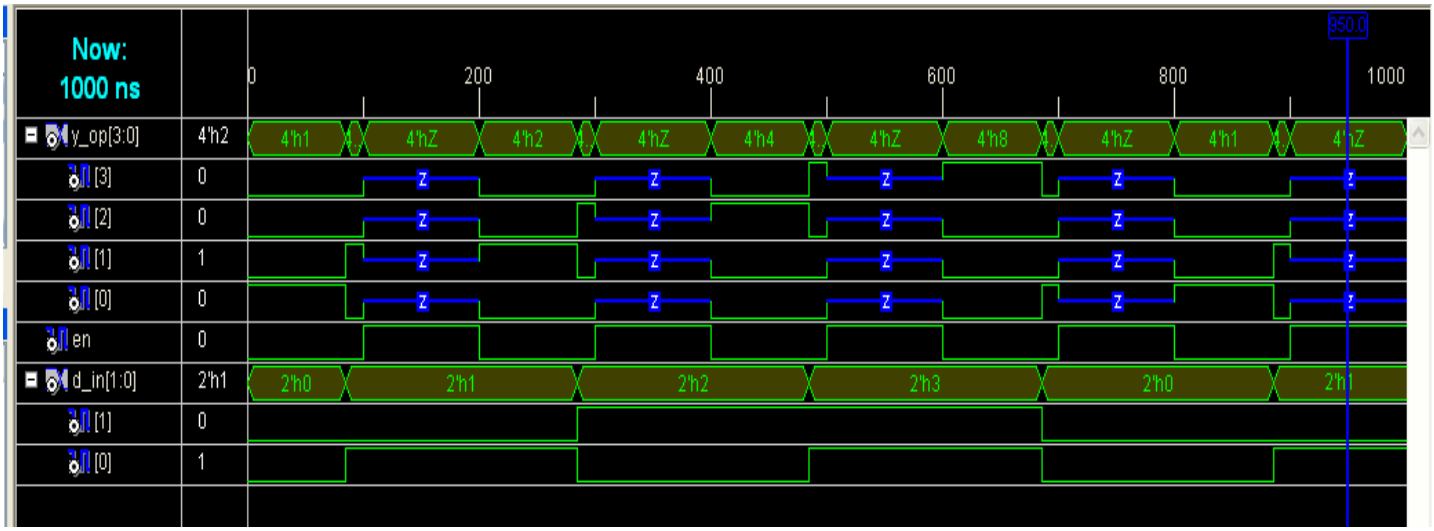
Table 3.2: 2:4 Decoder Truth table:

Inputs			outputs			
en	d_in(1)	d_in(0)	d_op(3)	d_op(2)	d_op1()	d_op0()
1	X	X	Z	Z	Z	Z
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0

VERILOG CODE:

```
module decoder (d_in,en,d_op);
input [1:0] d_in;
input en;
output [3:0] d_op;
reg [3:0] d_op;
always @(d_in,en)
begin
        if (en==1)
                d_op=4'bzzzz;
        else
                case (d_in)
                        2'b00: d_op = 4'b0001;
                        2'b01: d_op= 4'b0010;
                        2'b10:d_op= 4'b0100;
                        2'b11: d_op= 4'b1000;
                        default: d_op = 4'bxxxx;
                endcase
end
endmodule
```

2:4 Decoder Output:



4. COMPARATOR

Aim: Write Verilog HDL code for the Verification of 1-BIT & 4-BIT COMPARATOR.

Theory:

The Digital Comparator is another very useful combinational logic circuit used to compare the value of two binary digit.

Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals present at their input terminals and produce an output depending upon the condition of those inputs.

The purpose of **a** Digital Comparator is to compare a set of variables or unknown numbers, for example A (A1, A2, A3, …. An, etc) against that of a constant or unknown value such as B (B1, B2, B3, …. Bn, etc) and produce an output condition or flag depending upon the result of the comparison. For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

Which means: A is greater than B, A is equal to B, or A is less than B

This is useful if we want to compare two variables and want to produce an output when any of the above three conditions are achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

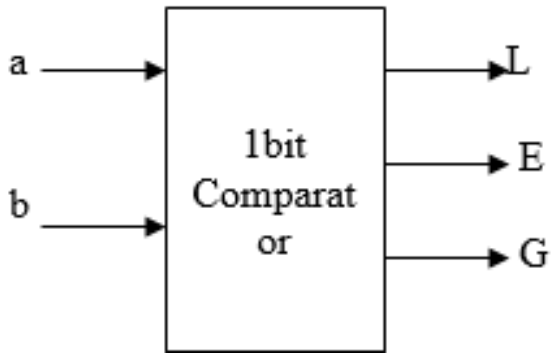


Fig4.1:1-bit Comparator

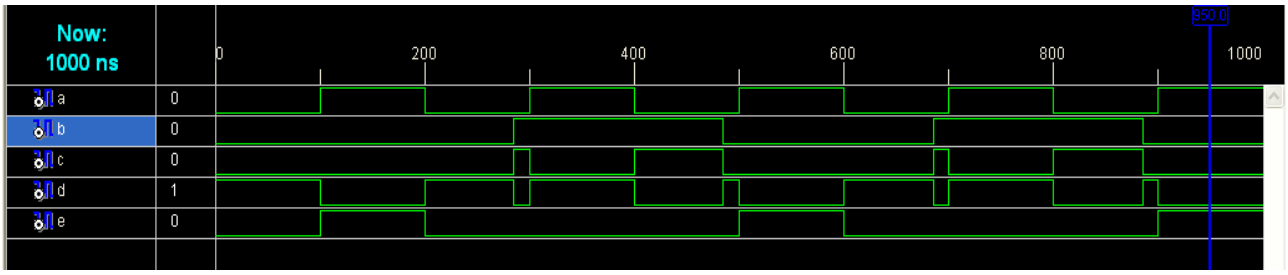
Table 4.1: 1-bit Comparator Truth Table:

A	B	C(lesser)	D(equal)	E(greater)
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

VERILOG CODE:

```
module bcomp (a, b, c, d, e)
input a, b;
output c, d, e;
assign c= (~a) & b;
assign d= ~(a ^ b);
assign e= a & (~b);
end module
```

1-bit Comparator Output:



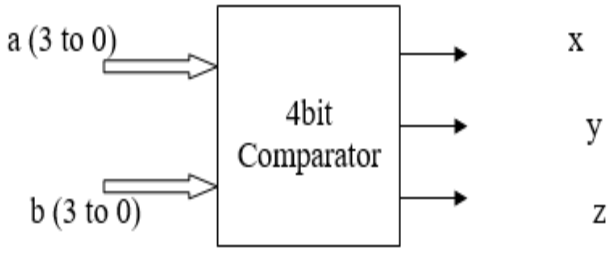


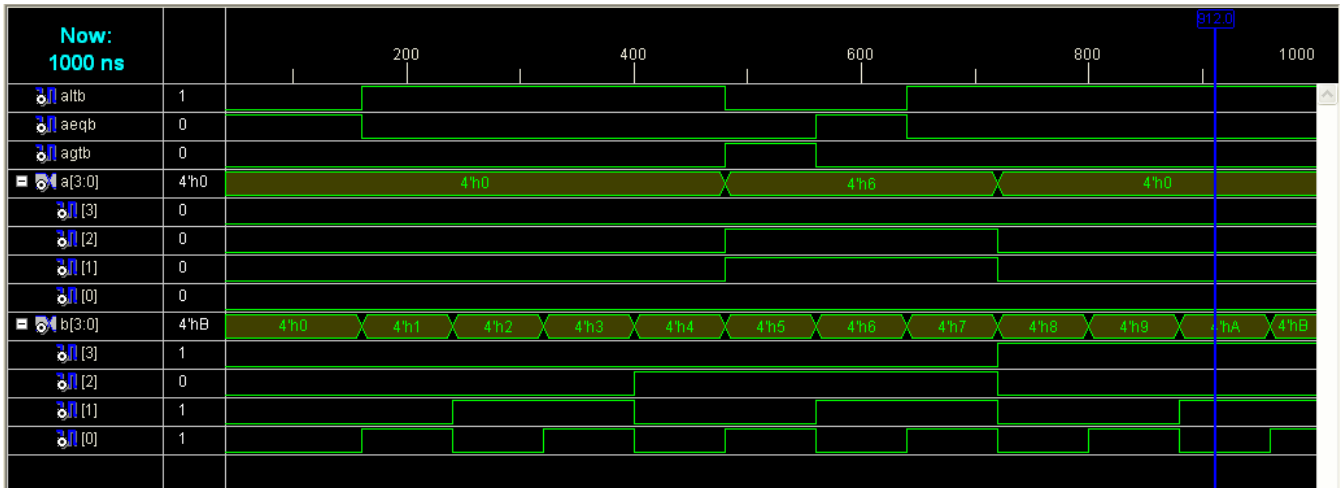
Fig4.2: 4-bit Comparator

VERILOG CODE:

```
module comp (a, b,aeqb,agtb,altb);
   input [3:0] a,b;
   output aeqb,agtb,altb;
   reg aeqb,agtb,altb;
```

```
                always @(a or b)
    begin
    aeqb=0;   agtb=0;     altb=0;
    if(a==b)
    aeqb=1;
    else if (a>b)
    agtb=1;
    else
    altb=1;
    end
endmodule
```

4-bit Comparator Output:



5. ADDER/SUBTRACTOR

AIM: Write an HDL code to describe the functions of half adder, full adders, half subtractor&full subtractor.

Theory:

HALF ADDER:

A half adder has two inputs for the two bits to be added and two outputs one from the sum ' S' and other from the carry ' c' into the higher adder position. Above circuit is called as a carry signal from the addition of the less significant bits sum from the X-OR Gate the carry out from the AND gate.

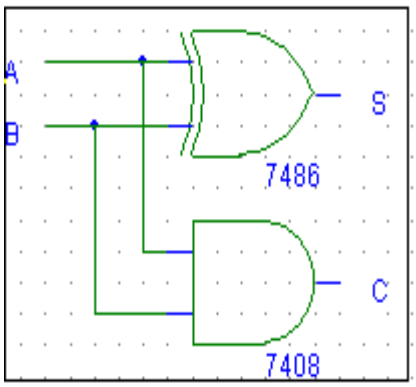


Fig 5.1: Half adder logic diagram

Table 5.1: Truth table:

INPUTS		OUTPUTS	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean expressions:

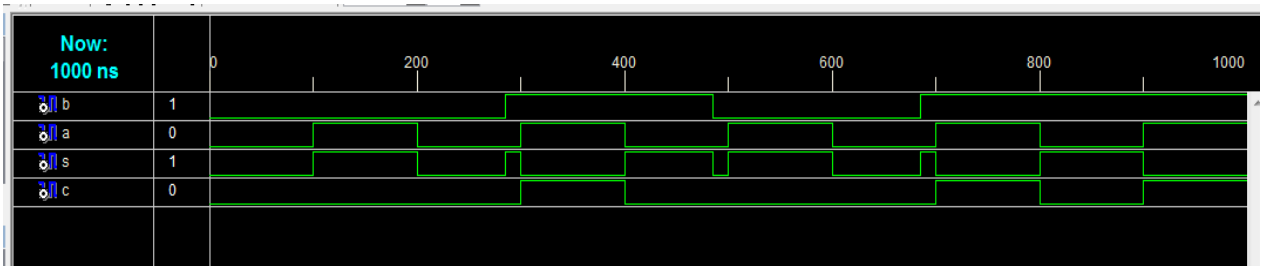
S=a ⊕b

C=a b

VERILOG CODE:

```
module ha ( a, b, s, c)
input a, b;
output s, c;
assign s= a ^ b;
assign c= a & b;
end module
```

OUTPUT:



FULL ADDER:

A full adder is a combinational circuit that forms the arithmetic sum of input; it consists of three inputs and two outputs. A full adder is useful to add three bits at a time but a half adder cannot do so. In full adder sum output will be taken from X-OR Gate, carry output will be taken from OR Gate.

Fig 5.2: Full adder Block diagram

Expression:

sum = a_in ⊕ b_in ⊕ c_in;
carry= (a_in.b_in)+(b_in.c_in)+(a_in.b_in);

Table 5.2: Full adder Truth table

Inputs			outputs	
a_in	b_in	c_in	sum	carry
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

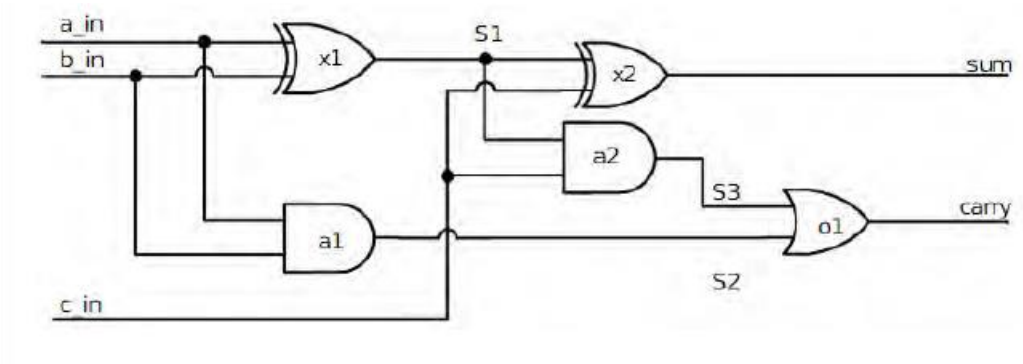
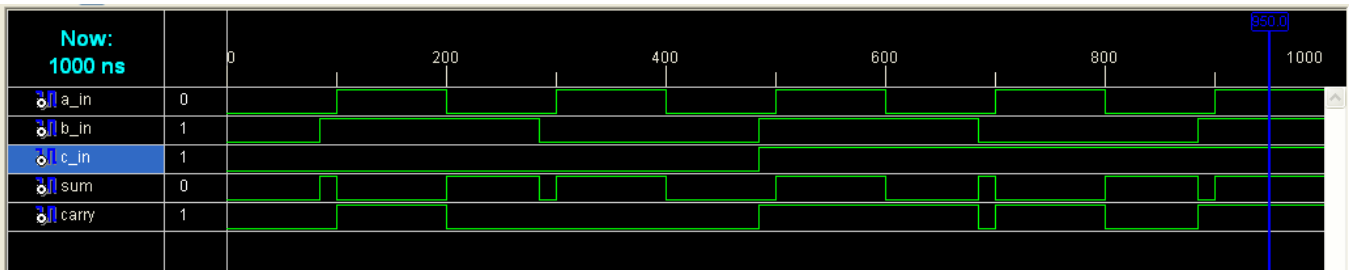


Fig 5.3: Full adder Logic diagram

VERILOG CODE:

```
module fulladder(a_in, b_in, c_in, sum, carry);

input a_in, b_in,c_in;

output sum, carry;

        assign sum = a_in^b_in^c_in;

        assign carry = (a_in & b_in) | (b_in & c_in) | (a_in & c_in);

endmodule
```

Full adder output:



Half Subtractor:

A half subtractor is a combinational circuit that performs the subtraction of two single-bit binary numbers. It has two inputs, the minuend and subtrahend, and two outputs: one for the difference ('D') and one for the

borrow ('B'). The difference is generated using an XOR gate, while the borrow signal is produced using an AND gate, followed by a NOT gate.

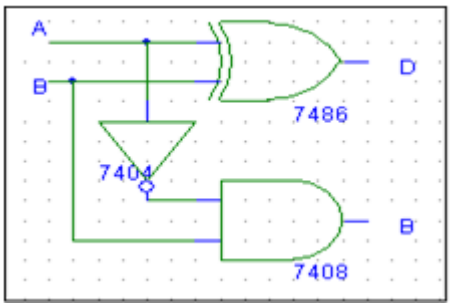


Fig 5.4: Half subtractor Logic diagram

Table 5.3: Half subtractor Truth table

INPUTS		OUTPUTS	
A	B	D	Br
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

BOOLEAN EXPRESSIONS:

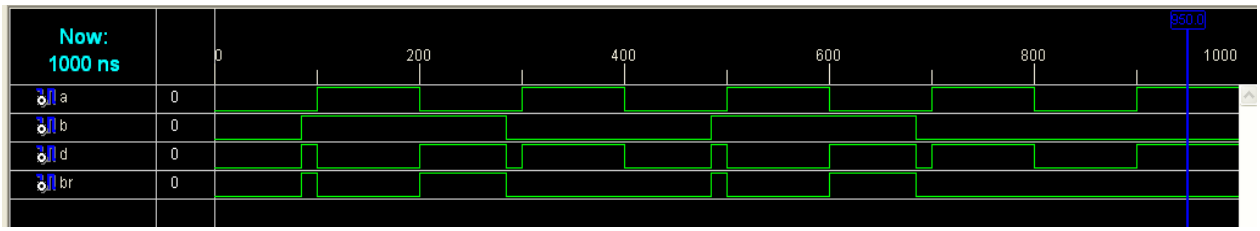
$$D = A \oplus B$$

$$Br = \bar{A}B$$

VERILOG CODE:

```
module hs ( a, b, d, br)
 input a, b;
 output d, br;
 assign d= a ^ b;
 assign br= ~a & b;
endmodule
```

Half Subtractor Output:



Full Subtractor:

A full subtractoris a combinational circuit that subtracts three binary bits: the minuend (A), subtrahend (B), and a borrow-in (Bin). It has two outputs: the difference(D) and the borrow-out(Bout). The difference is calculated using XOR gates, as \(D = A \oplus B \oplus Bin \), while the borrow-out indicates if a borrow is needed for the next higher bit, determined by \(Bout = (Bin \cdot (A \oplus B)) + (\overline{A} \cdot B) \). This allows the full subtractor to handle multi-bit subtractions efficiently.

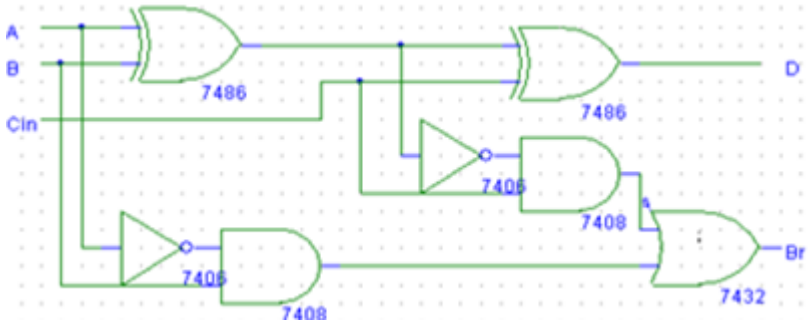


Fig 5.5: Full subtractor Logic diagram

BOOLEAN EXPRESSIONS:

$$D = A \oplus B \oplus C$$

$$Br = \bar{A}\,B + B\,Cin + \bar{A}\,Cin$$

Table 5.4: Full subtractor Truth table

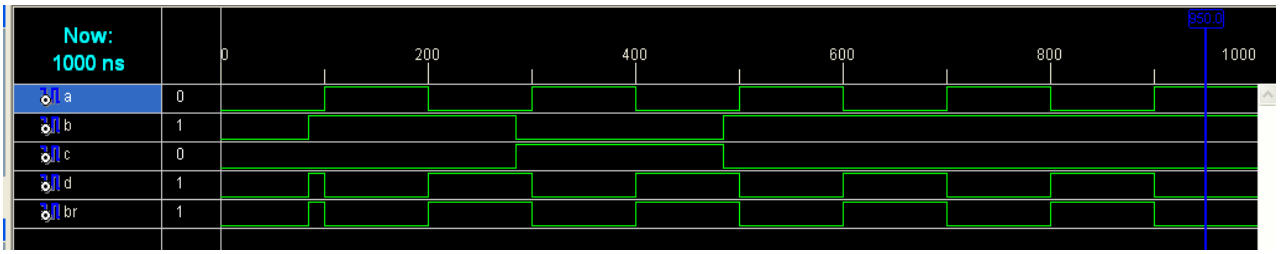
Inputs			outputs	
A	B	Cin	D	BR
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

VERILOG CODE:

```
module fs ( a, b, c, d, br)

 input a, b, c;

 output d, br;
```

```
assign d= a ^ b ^ c;

assign br=(( ~a)& (b ^ c)) | (b & c);

endmodule
```

Full Subtractor Output:



6.MULTIPLIER

AIM: Write an HDL code to describe the functions of a 4-bit Multiplier using full adders.

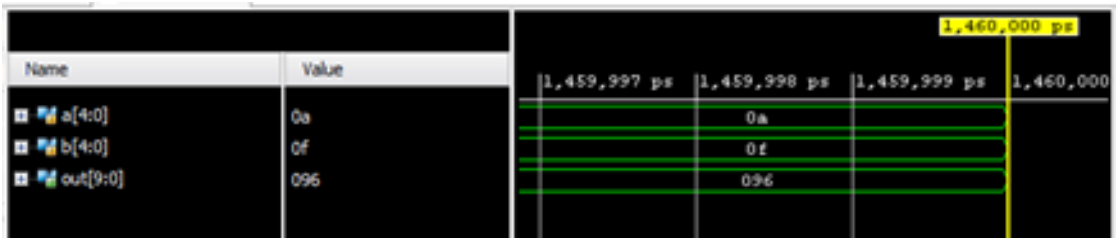
Theory:

A **multiplier** is a combinational circuit designed to perform multiplication of two binary numbers. It takes two sets of binary inputs (multiplicand and multiplier) and produces an output representing their product. Multipliers are commonly used in arithmetic units, digital signal processing, and processors, and their design focuses on optimizing speed, area, and power consumption for efficient operation.

VERILOG CODE:

```
module mul(a,b,out);
      input [4:0]a;
      input [4:0]b;
      output [9:0]out;
      assign out=(a*b);
endmodule
```

Multiplier Output:



7.PARITY GENERATOR

AIM: Write an HDL code to describe the functions of a parity generator.

Theory:

A parity generator is a digital circuit that computes a parity bit based on a set of input bits. It adds an extra bit, known as the parity bit, to ensure that the total number of 1s in the combined set of input bits (including the parity bit) is either even (even parity) or odd (odd parity). Parity generators are commonly used in error detection schemes in digital communication and storage systems to help identify transmission errors.

Even Parity Generator:

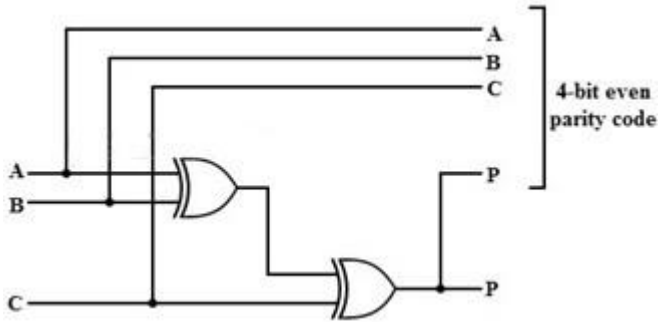


Fig 7.1: Even parity generator Circuit diagram

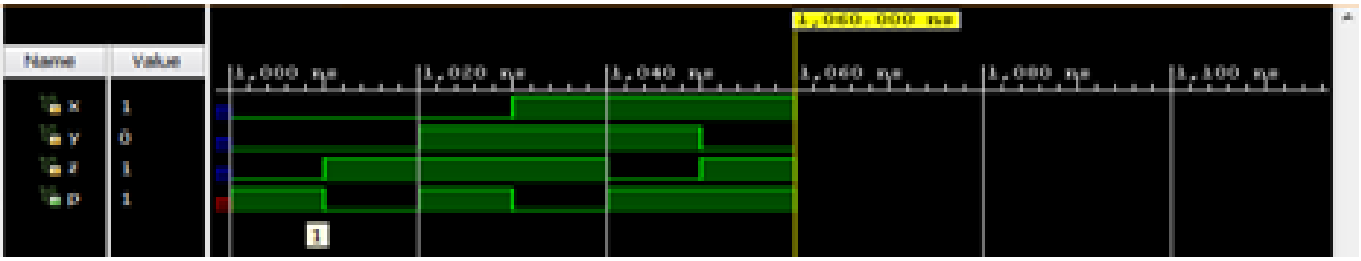
Table 7.1: Even parity generator Truth table:

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

VERILOG CODE:

```
module even_parity_generator(input A,input B,input C,output p); reg p;
always @(A,B,C)
begin case({A,B,C})
        3'b000:{p}=1'b0;
        3'b001:{p}=1'b0;
        3'b010:{p}=1'b0;
        3'b011:{p}=1'b1;
        3'b100:{p}=1'b0;
        3'b101:{p}=1'b1;
        3'b110:{p}=1'b1;
        3'b111:{p}=1'b0;
        default:{p}=1'bx;
endcase
end endmodule
```

Even parity generator Output:



Odd Parity Generator:

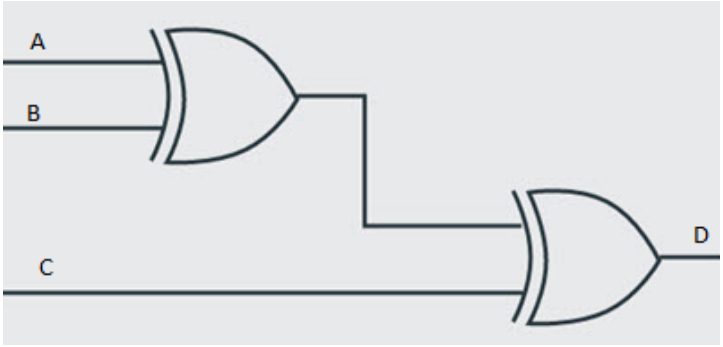


Fig 7.2: Odd parity generator Circuit diagram

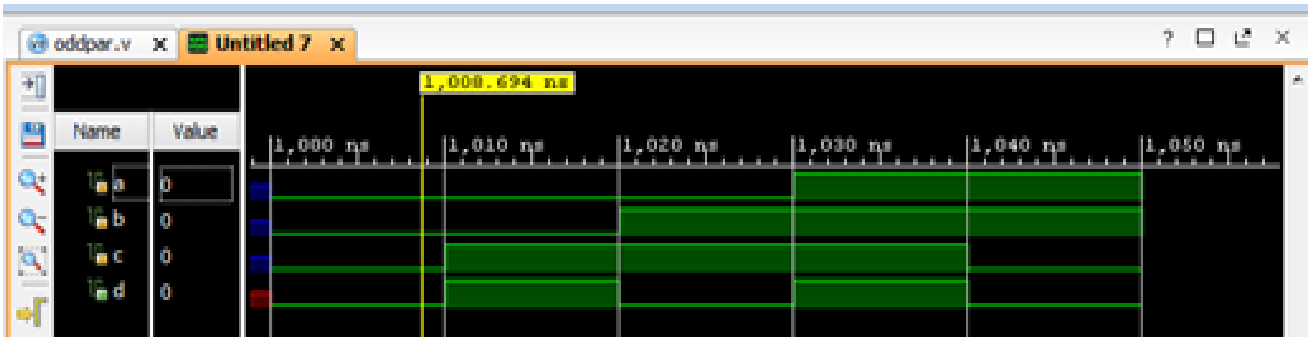
Table7.2: Odd parity generator Truth table:

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

VERILOG CODE:

```
module oddparity(input a, input b, input c, output y);
reg d;
always @ (a,b,c)
begin case({a,b,c})
        3'b000:{d}=1'b0;
        3'b001:{d}=1'b1;
        3'b010:{d}=1'b1;
        3'b011:{d}=1'b0;
        3'b100:{d}=1'b1;
        3'b101:{d}=1'b0;
        3'b110:{d}=1'b0;
        3'b111:{d}=1'b1;
endcase end
endmodule
```

Odd parity generator Output:



8. DESIGN OF ALU

AIM: Write an HDL code to design ALI(Arithmetic Logic Unit).

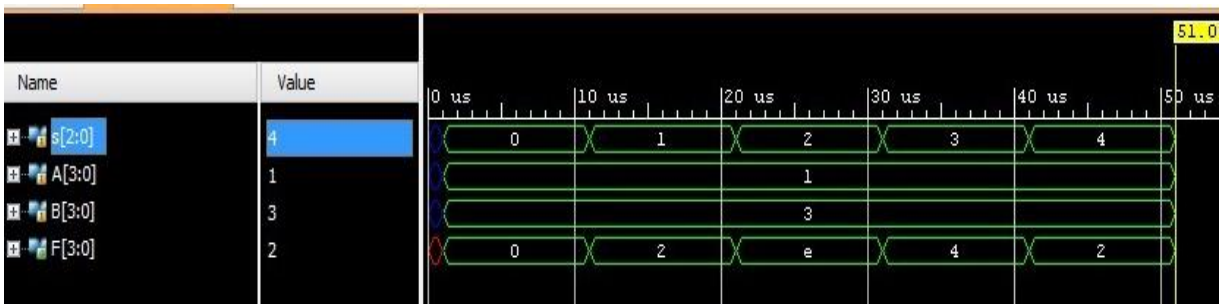
Theory:

An ALU (Arithmetic Logic Unit) is a fundamental digital circuit that performs arithmetic operations (like addition, subtraction, and multiplication) and logical operations (such as AND, OR, XOR) on binary data. It is a key component of a processor, responsible for executing the core mathematical and logical computations required by the CPU. The ALU is designed to be efficient in terms of speed, area, and power consumption in VLSI circuits.

VERILOG CODE:

```
module ALU(s, A, B, F);
   input [2:0] s;

   input [3:0] A, B;

   output[3:0] F;

   reg [3:0] F;

   always @(s or A or B)

   case (s)

      0: F =4'b0000;

      1: F= B - A;

      2: F= A- B;

      3: F= A + B;

      4: F= A^ B;

      5: F= A|B;

      6: F= A & B;

      7: F= 4'b1111;

   endcase

endmodule
```

OUTPUT


Name
Value
s[2:0]
A[3:0]
B[3:0]
F[3:0]
4
1
3
2
0 us
10 us
20 us
30 us
40 us
50 us
51.0
0
1
2
3
4
1
3
0
2
e
4
2

9.LATCHES

AIM: Write an HDL code to verifythe working of latches.

Theory:

A latch is a type of digital circuit that stores and maintains a binary state (either 0 or 1). It is a basic building block of memory in electronics, often used to hold a value until it is changed by an input signal. Latches are typically controlled by signals like "enable" or "clock" to determine when the stored value should be updated.

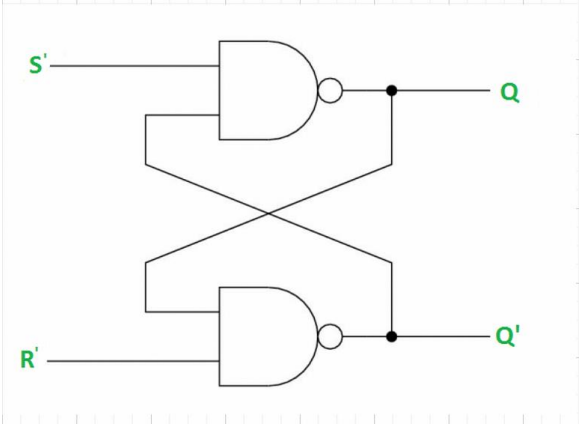


Fig 9.1: SR latch

Table 9.1: Truth table

S	R	Qn+1
0	0	0 (Hold)
0	1	0 (Reset)
1	0	1 (Set)
1	1	Forbidden

VERILOG CODE:

```
module latchsr(
        input s,
        input r,
        output q,
        output qbar
        );
        nand n1(q,s,qbar);
        nand n2(qbar,r,q);
endmodule
```

SR latch Output:

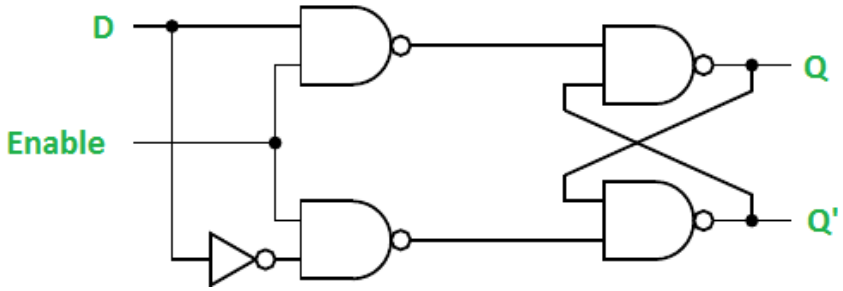


Fig 9.2: D latch

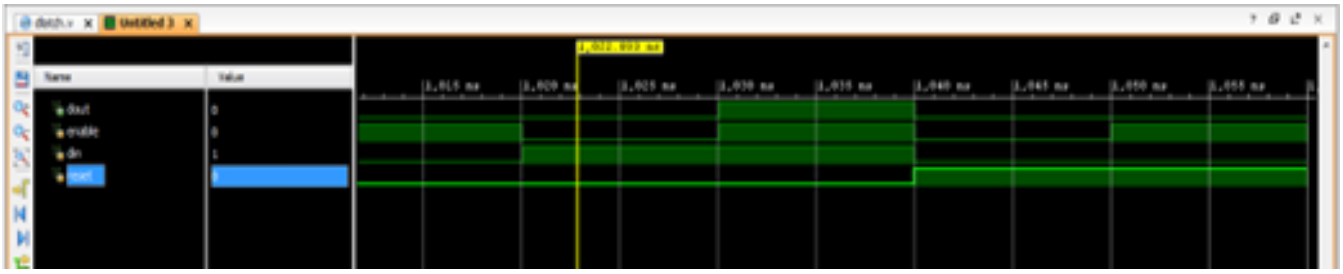
Table 9.2: D latch Truth table

E	D	Q
0	x	Q (Hold)
1	0	0 (Reset)
1	1	1 (Set)

VERILOG CODE:

```
module D_latch ( enable ,din ,reset ,dout );
output dout ;
reg dout ;
input enable ;
wire enable ;
input din ;
wire din ;
input reset ;
wire reset ;
always @ (enable or din or reset) begin
if (reset)
        dout = 0;
else begin if (enable)
        dout = din; end
end endmodule
```

D latch Output



10. FLIP-FLOPS

AIM: Write an HDL code to verify the working of Flip- flops.

Theory:

A flip-flopin VLSI is a memory element that stores a single bit of data and is used for synchronization in digital circuits. It is edge-triggered, meaning it changes its output only on the rising or falling edge of a clock signal.

There are different types of flip-flops (e.g., D flip-flop, JK flip-flop, T flip-flop), but they all share the basic property of changing their output only when the clock edge occurs, making them more suitable for synchronous circuits compared to latches.

Flip-flops are key building blocks in registers, counters, and other sequential circuits.

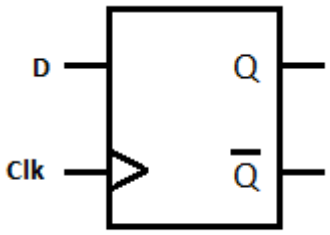


Fig 10.1: D flip-flop

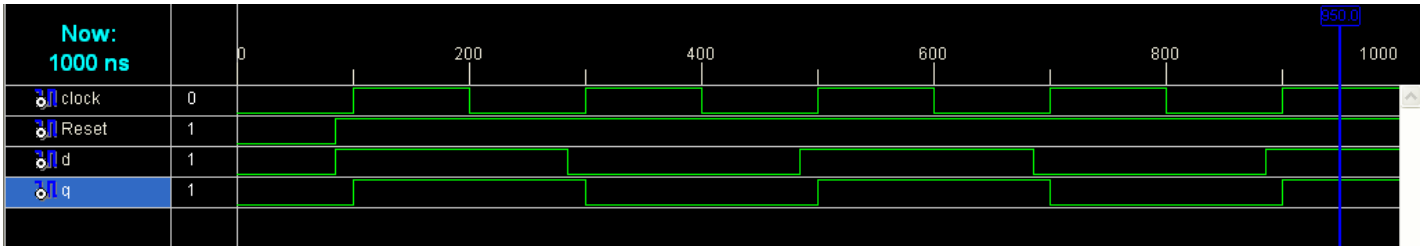
Table 10.1: D flip-flop Truth table:

CLK	D	Q	QB	OPERATION
0	X	Q	QB	NO CHANGE
1	0	0	1	RESET
1	1	1	0	SET

VERILOG CODE:

```
module DFF2(clock, Reset, d, q);
    input clock;
    input Reset;
    input d;
    output q;
reg q;
always@(posedge clock or negedge Reset)
if (~Reset)
        q=1'b0;
else
        q=d;
endmodule
```

D flip-flop Output:



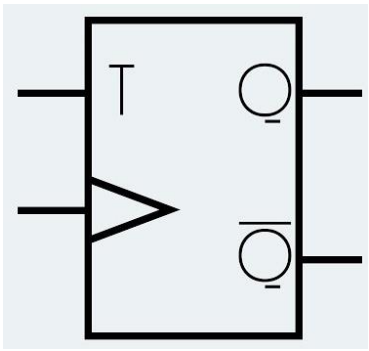


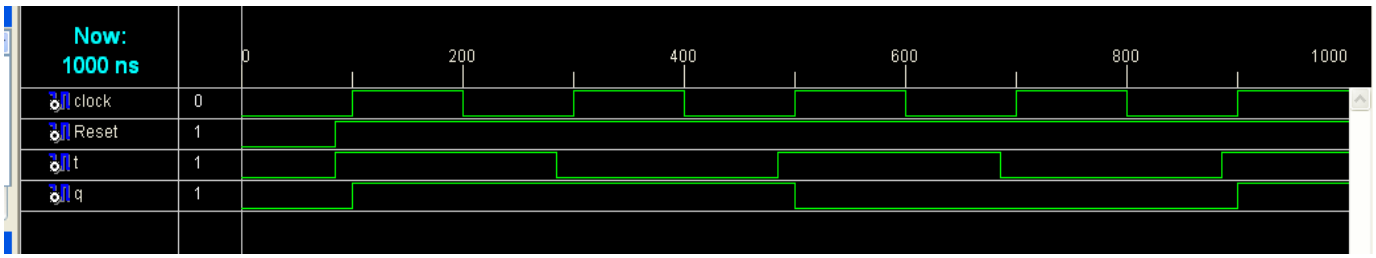
Fig 10.2: T flip-flop

Table 10.2: T flip-flop Truth table:

CLK	T	Q	Qb	OPERATION
0	x	Q	Qb	No Change
1	0	Q	Qb	No Change
1	1	Qb	Q	Toggle

VERILOG CODE:

```
module TFF(clock,  Reset,  t,  q);
   input clock;
   input  Reset;
   input  t;
   output  q;
          reg q;
          always@(posedge clock , negedge Reset)
                    if(~Reset) q=0;
                    else if (t) q=~q;
                    else q=q;
      endmodule
```



T flip-flop Output:

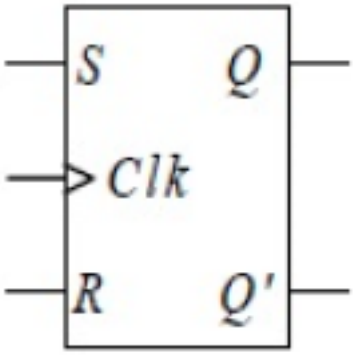


Fig 10.3: SR flip-flop

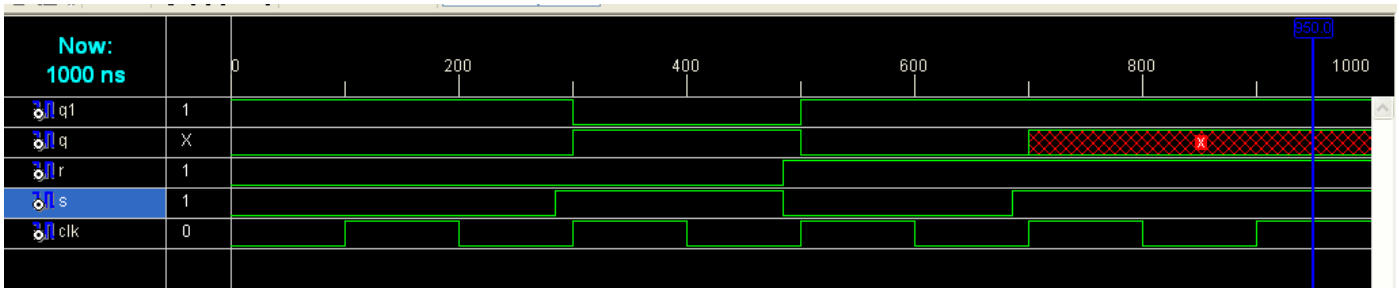
Table 10.3: SR flip-flop Truth table

CLK	s	r	Q	Qb
0	x	x	Q	Qb
1	0	1	0	1
1	1	0	1	0
1	1	1	Not defined	
1	0	0	Q	Qb

VERILOG CODE:

```
module srff(q,q1,r,s,clk);
        output q,q1;
        input r,s,clk;
        reg q,q1;
        initial
        begin
                q=1'b0;
                q1=1'b1;
        end
        always @(posedge clk)
        begin
        case({s,r})
                {1'b0,1'b0}:begin q=q; q1=q1;end
                {1'b0,1'b1}:begin q=1'b0; q1=1'b1;end
                {1'b1,1'b0}:begin q=1'b1; q1=1'b0;end
                {1'b1,1'b1}:begin q=1'bx; q=1'bx;end
        endcase
        end
endmodule
```

SR flip-flop Output:



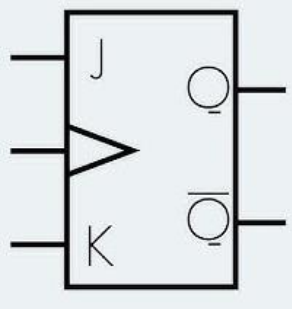


Fig 10.4: JK flip-flop

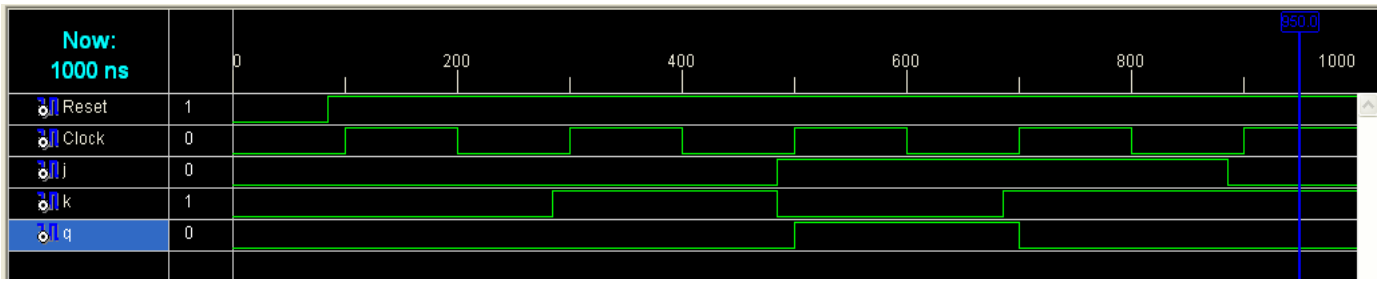
Table 10.4: JK flip-flop Truth table

CLK	J	K	Q	Qb
0	x	x	Q	Qb
1	0	1	0	1
1	1	0	1	0
1	1	1	Qb	Q
1	0	0	Q	Qb

VERILOG CODE:

```
module JKFF(Clock, Reset, j, k, q);
input Clock;
input Reset;
input j;
input k;
output q;
reg q;
always@(posedge Clock, negedge Reset)
if(~Reset)
        q=0;
else
        begin case({j,k})
                2'b00: q=q;
                2'b01: q=0;
                2'b10: q=1;
                2'b11: q=~q;
        endcase
end
endmodule
```

JK flip-flop Output:



11.SYNCHRONOUS UP/DOWN COUNTER(4BIT)

AIM: Write an HDL code to verify the SYNCHRONOUS UP/DOWN COUNTER.

Theory:

A synchronous up/down counter is a digital circuit that can count in both increasing (up) and decreasing (down) order based on a control signal, with all operations occurring simultaneously with the clock pulse. In the "up" mode, the counter increments its value, and in the "down" mode, it decrements with each clock cycle. Since all flip-flops are triggered by the same clock signal, the counter updates its output synchronously, ensuring reliable and precise timing. This type of counter is commonly used in applications like digital clocks, position tracking, and reversible counting systems.

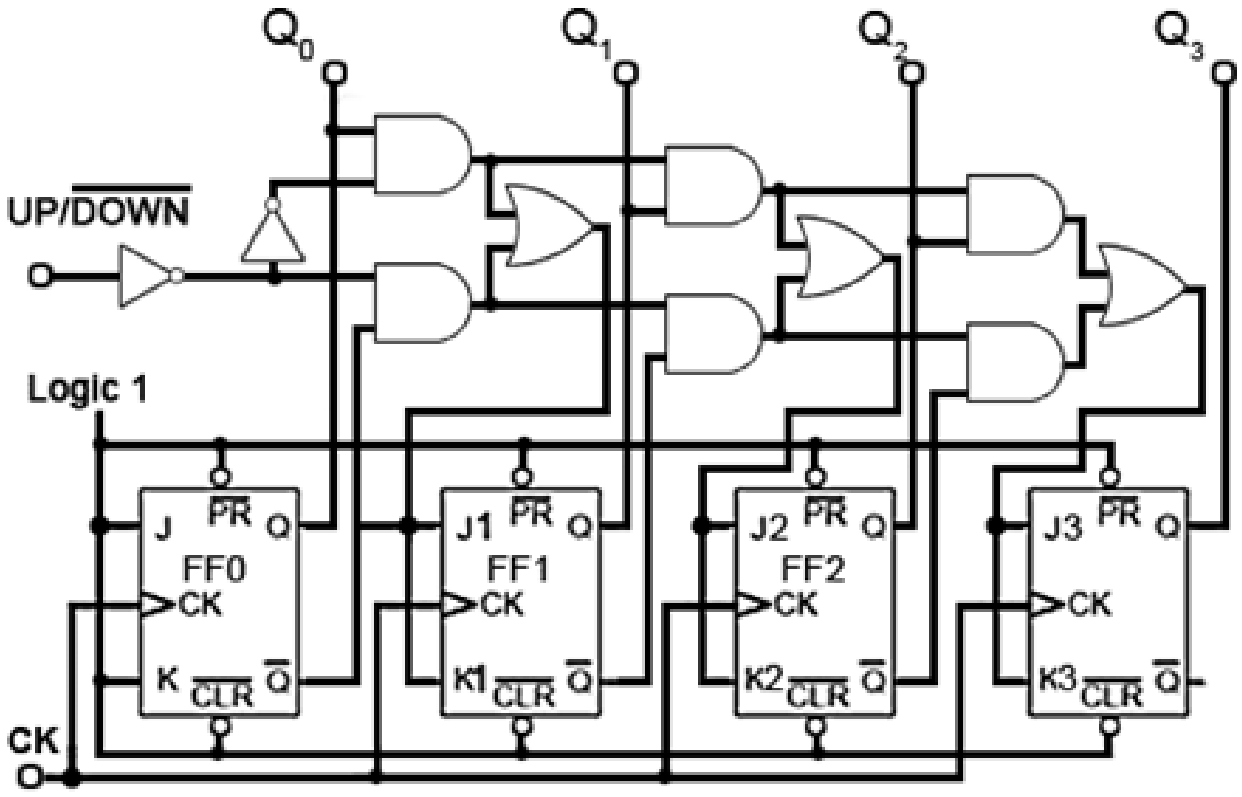


Fig 11.1: Synchronous up/down counter(4bit)

Table 11.1: Synchronous up/down counter (4bit) Truth table

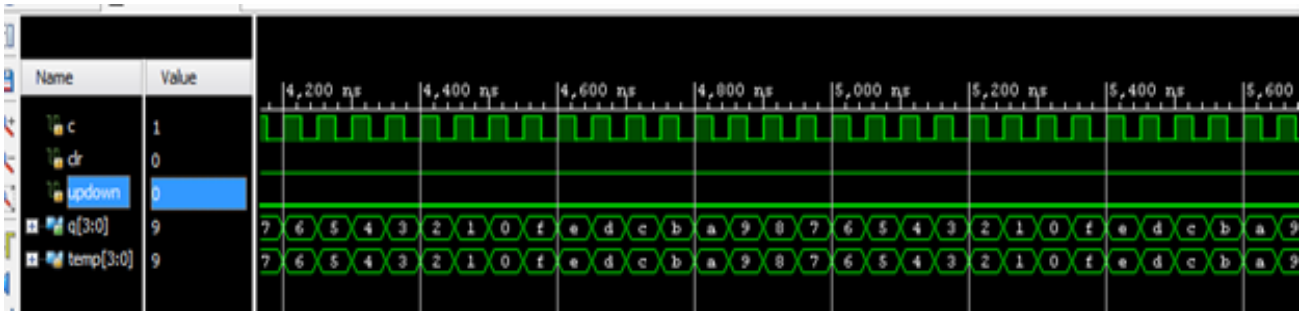
CK	Q3	Q2	Q1	Q0	Q3'	Q2'	Q1'	Q0'
0	0	0	0	0	1	1	1	1
1	0	0	0	1	1	1	1	0
2	0	0	1	0	1	1	0	1
3	0	0	1	1	1	1	0	0
4	0	1	0	0	1	0	1	1
5	0	1	0	1	1	0	1	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	0	1	1	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	0	1
11	1	0	1	1	0	1	0	0
12	1	1	0	0	0	0	1	1
13	1	1	0	1	0	0	1	0
14	1	1	1	0	0	0	0	1
15	1	1	1	1	0	0	0	0

VERILOG CODE:

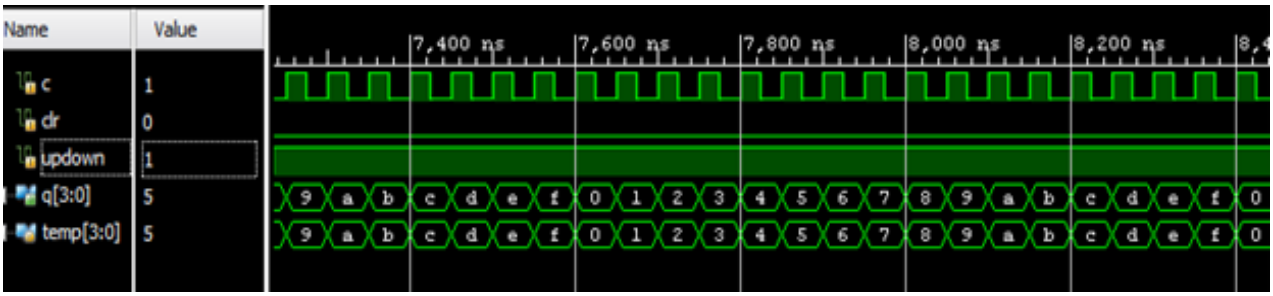
```
module syncup( input c,inputclr, input updown, output [3:0] q);
reg [3:0]temp,q;
always @(posedge c or posedgeclr) begin
        if(clr)
                temp=4'b0000;
        else if(updown)
                temp=temp+1'b1;
        else
                temp=temp-1'b1;
        assign q=temp; end
endmodule
```

Synchronous up/down counter(4bit) Output:

UPDOWN=0



UPDOWN=1


Name
Value
c
1
clr
0
updown
1
q[3:0]
5
temp[3:0]
5
7,400 ns
7,600 ns
7,800 ns
8,000 ns
8,200 ns
9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f 0
9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f 0

12. ASYNCHRONOUS COUNTER

AIM: Write an HDL code to verify the ASYNCHRONOUS COUNTER.

Theory:

An asynchronous counter also known as a ripple counter, is a type of digital counter where the flip-flops are not all triggered by the same clock signal. Instead, the output of one flip-flop serves as the clock input for the next flip-flop in the chain. This causes a delay, as each flip-flop responds to the change in the previous one, creating a "ripple" effect through the counter.

Asynchronous counters are simpler in design but slower compared to synchronous counters due to this ripple delay. They are commonly used in applications where high speed is not a critical factor, such as simple timers or low-frequency counting tasks.

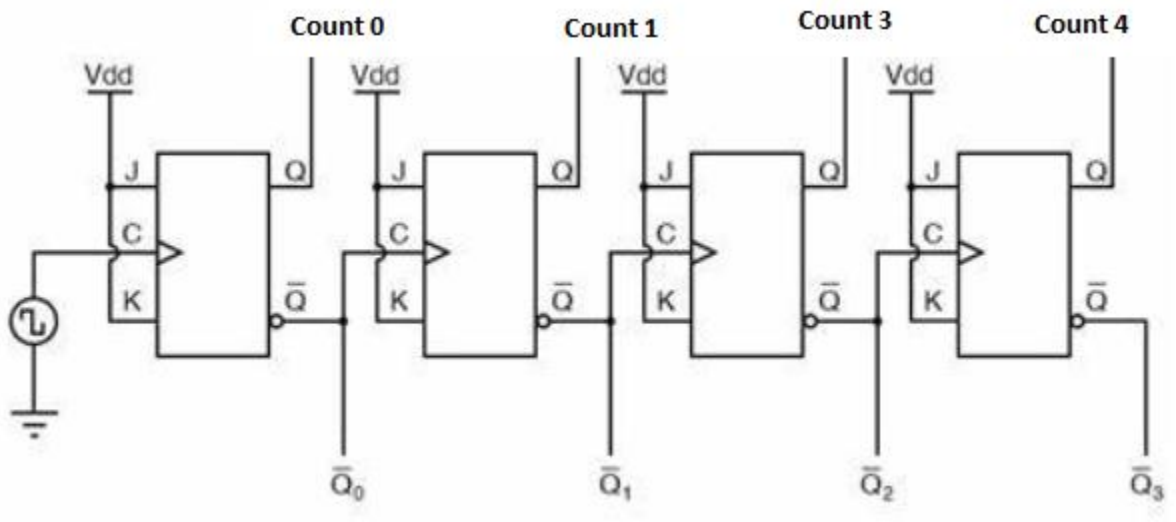


Fig 12.1: Asynchronous counter

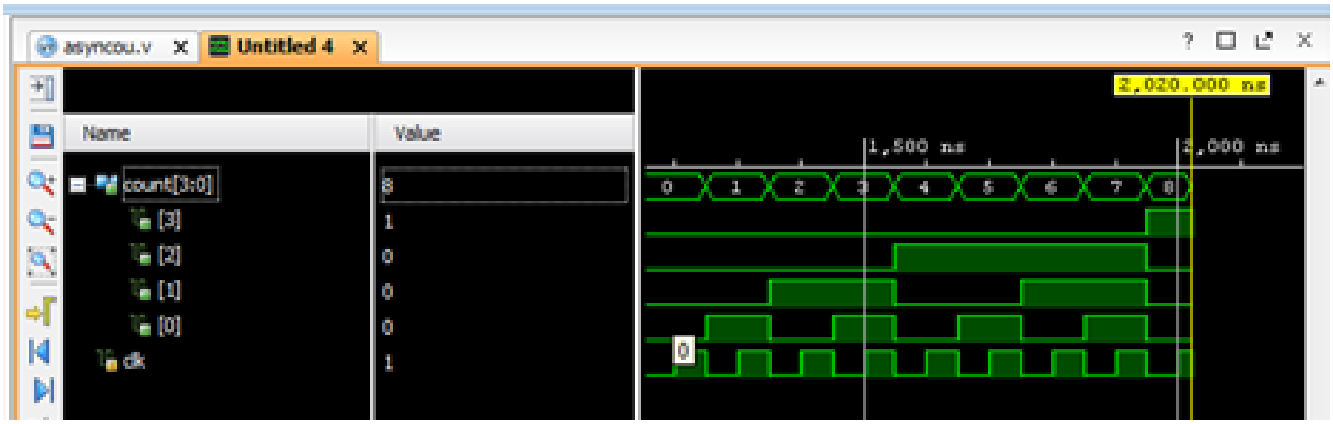
Table 12.1: Asynchronous counter Truth table:

Clk	Count 3	Count 2	Count 1	Count 0
1	0	0	0	0
1	0	0	0	1
1	1	0	1	0
1	1	0	1	1
1	1	0	1	0
1	0	1	0	1
1	0	1	1	0
1	0	1	1	1
1	1	0	0	0
1	1	0	0	1
1	1	0	1	0
1	1	0	1	1
1	1	1	0	0
1	1	1	0	1
1	1	1	1	0
1	1	1	0	1

VERILOG CODE:

```
module asyncou(clk,count);
input clk;
output[3:0] count;
reg[3:0] count;
wire clk;
initial
count =4'b0;
always @(negedge clk) count[0]<=~count[0];
always @(negedge count[0]) count[1]<=~count[1];
always @(negedge count[1]) count[2]<=~count[2];
always @(negedge count[2]) count[3]<=~count[3];
endmodule
```

Asynchronous counter Output:



13.SHIFT REGISTERS

AIM: Write an HDL code for verification of SHIFT REGISTERS.

Theory:

A **register** is a small, high-speed storage unit within a CPU or digital circuit that temporarily holds data, instructions, or memory addresses for quick access during processing. Registers are faster than regular memory (RAM) and are crucial for the CPU to perform operations efficiently. There are different types of registers based on their function.

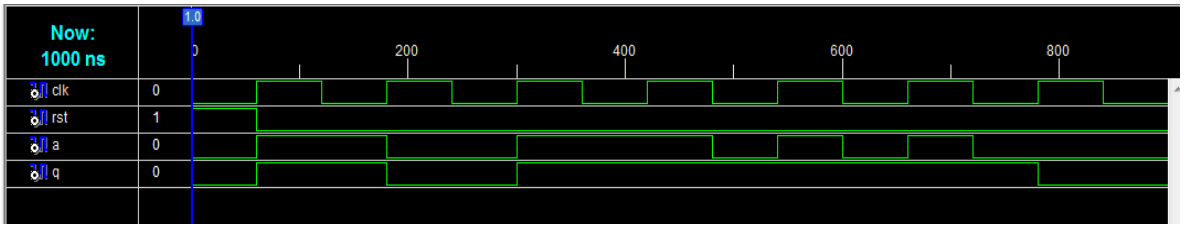
SERIAL – IN SERIAL – OUT SHIFT REGISTER

Serial-in to Serial-out (SISO) - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction under clock control.

VERILOG CODE:

```
module siso(clk,rst,a,q);
input a;
input clk,rst;
output q;
reg q;
always@(posedgeclk,posedgerst)
begin
	if(rst==1'b1)
		q<=1'b0;
	else
		q<=a;
	end
endmodule
```

Serial – in serial – out shift register Output:



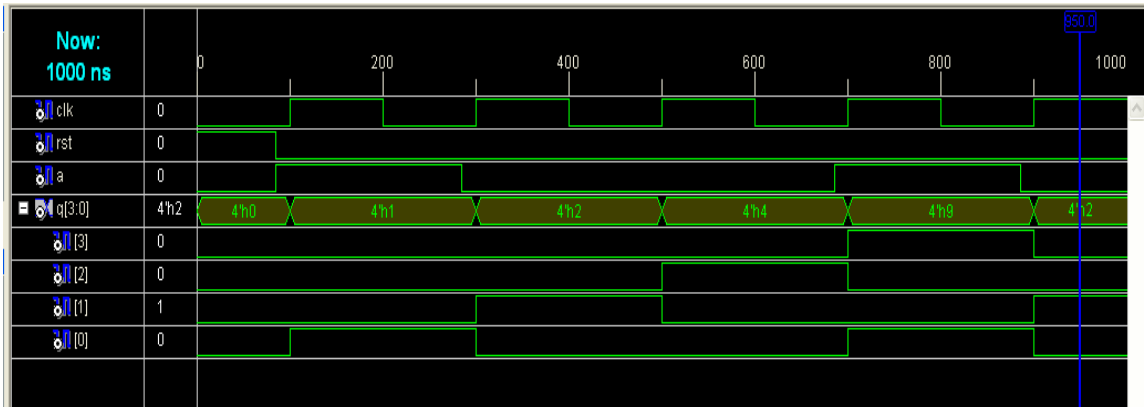
SERIAL – IN PARALLEL – OUT SHIFT REGISTER

Serial-in to Parallel-out (SIPO) - the register is loaded with serial data, one bit at a time, with the stored data being available at the output in parallel form.

VERILOG CODE:

```
module SIPO(a, clk, rst, q);
 input clk,rst;
input a;
output [3:0]q;
wire [3:0]q;
reg [3:0]temp;
always@(posedgeclk,posedgerst)
begin
        if(rst==1'b1)
                temp<=4'b0000;
        else
                begin
                temp<=temp<<1'b1;
                temp[0]<=a;
        end
end
assign q=temp;
endmodule
```

Serial – in parallel – out shift register Output:



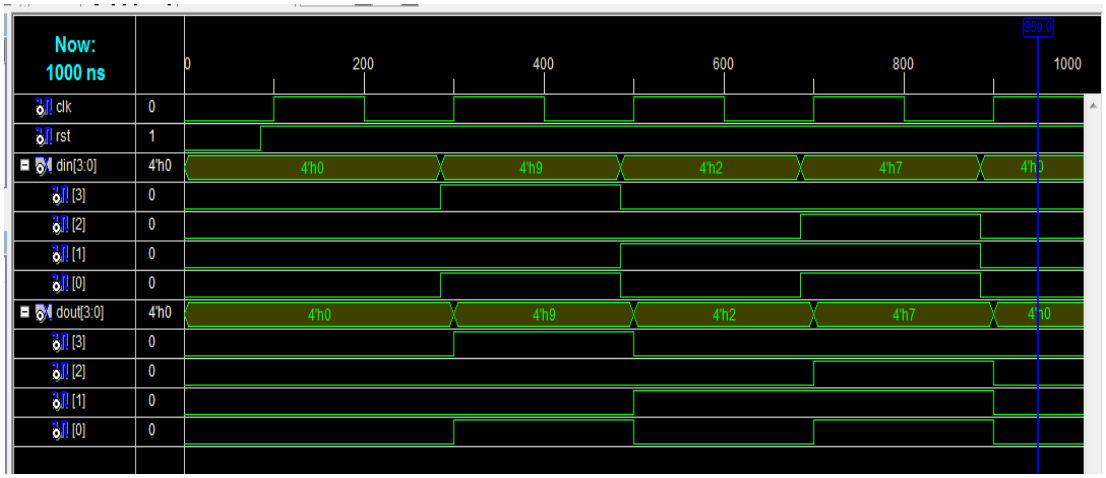
PARALLEL – IN PARALLEL – OUT SHIFT REGISTER

Parallel-in to Parallel-out (PIPO) - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse.

VERILOG CODE:

```
module pipo(din,clk,rst,dout);
input [3:0] din;
input clk,rst;
output [3:0] dout;
wire [3:0] din;
wire clk,rst;
reg [3:0] dout;
always @(posedge clk or negedgerst)
begin
if(!rst)
        begin
                dout<= 4'b0;
        end
else
        begin
                dout<= din;
        end
end
endmodule
```

Parallel – in parallel – out shift register Output:



14. MEMORIES

AIM: Write an VHDL code for verify the working of memories.

Theory

memories refer to storage components used to store and retrieve digital data in a VLSI circuit. These are crucial for designing and simulating integrated circuits that require data handling, such as microprocessors, signal processors, and other digital systems.

RAM

(Random Access Memory): A type of volatile memory that allows data to be read and written at any address location randomly. It loses its contents when power is turned off. RAM is used for temporary data storage during processing.

VERILOG CODE

```
module RAM (
   input wire
clk,
   input wire
we,
   input wire
[3:0] addr,
   input wire
[7:0] din,
   output reg
[7:0] dout
);

reg [7:0] mem
[0:15];

always
@(posedge clk)
begin
   if (we) begin
      mem[addr]
<= din;
   end else
begin
      dout <=
mem[addr];
   end
```

```
end

endmodule
```

ROM (Read-Only Memory): A non-volatile memory where data is permanently written during the manufacturing process or once during initialization. The contents cannot be modified during normal operation, and it is mainly used for storing fixed data, like firmware.

VERILOG CODE

```
module ROM (
   input wire [3:0] addr,// 4-bit address input (16
   locations)output reg [7:0] dout  // 8-bit data output
);
   // ROM initialized with 16 8-bit
   values reg [7:0] mem [15:0];
   initial begin
      // Preload some values into the ROM
      mem[0] = 8'hAA;
      mem[1] = 8'hBB;
      mem[2] = 8'hCC;
      mem[3] = 8'hDD;
      mem[4] = 8'h11;
      mem[5] = 8'h22;
      mem[6] = 8'h33;
      mem[7] = 8'h44;
      mem[8] = 8'h55;
      mem[9] = 8'h66;
      mem[10] =
      8'h77; mem[11]
      = 8'h88;
      mem[12] =
      8'h99; mem[13]
      = 8'hAA;
      mem[14] =
      8'hBB;mem[15]
      = 8'hCC;
   end

   always @(*) begin
      dout = mem[addr]; // Read operation
   (asynchronous)end
endmodule
```

15. CMOS CIRCUITS

AIM: Write a VHDL code to design CMOS CIRCUITS.

Theory:

A CMOS circuit (Complementary Metal-Oxide-Semiconductor) is an electronic circuit built using both PMOS (P-type Metal-Oxide- Semiconductor) and NMOS (N-type Metal-Oxide-Semiconductor) transistors. CMOS technology is widely used for constructing integrated circuits (ICs), including microprocessors, microcontrollers, and memory chips, due to its low power consumption, high noise immunity, and scalability.

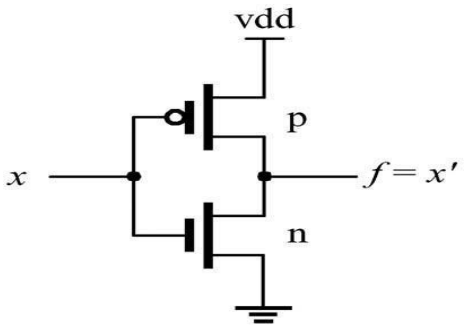
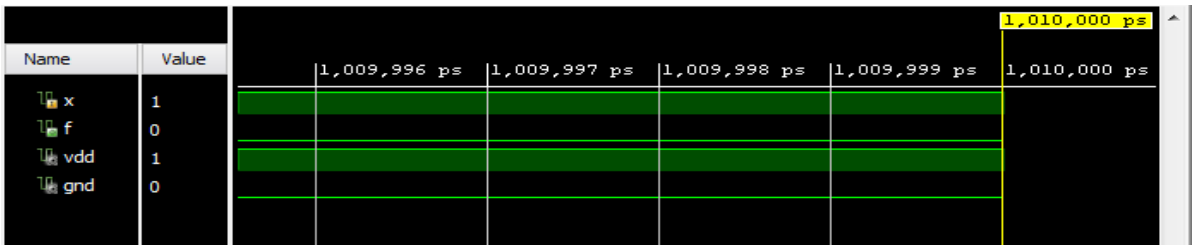


Fig 15.1: CMOS inverter circuit

VERILOG CODE:

```
moduleine233( input x, output f
);
supply1vdd;
supply0gnd;
body pmos
p1(f,vdd,x);
nmosn1(f,gnd,x);
endmodule
```

OUTPUT



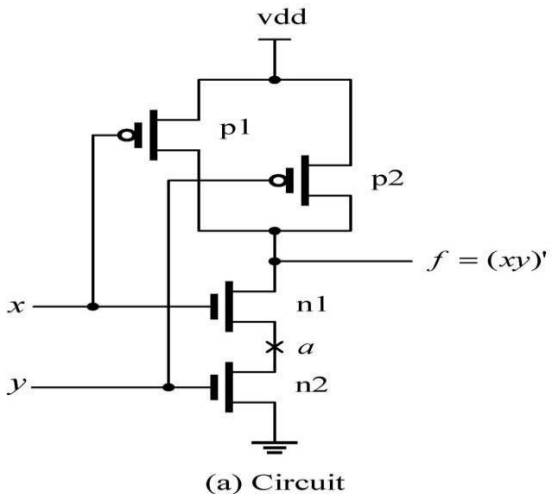
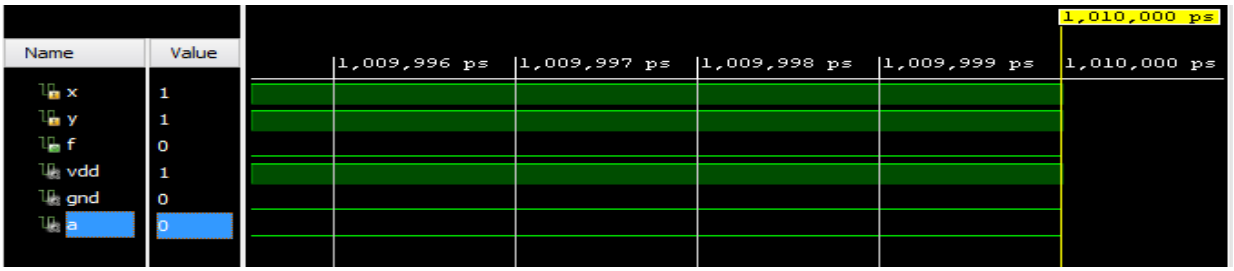


Fig 15.2: CMOS NAND

VERILOG CODE

```
modulecmosnand2(
input x,inputy, output f
);
supply1vdd; supply0gnd;
wire a;pmosp1 (f,vdd, x);
pmosp2 (f,vdd,y);
nmosn1(f,a, x);
nmosn2(a,gnd,y);
endmodule
modulecmosnand2(
input x,inputy, output f
);
supply1vdd; supply0gnd;
wire a;pmosp1 (f,vdd, x);
pmosp2 (f,vdd,y);
nmosn1(f,a, x);
nmosn2(a,gnd,y);
endmodule
```

OUTPUT



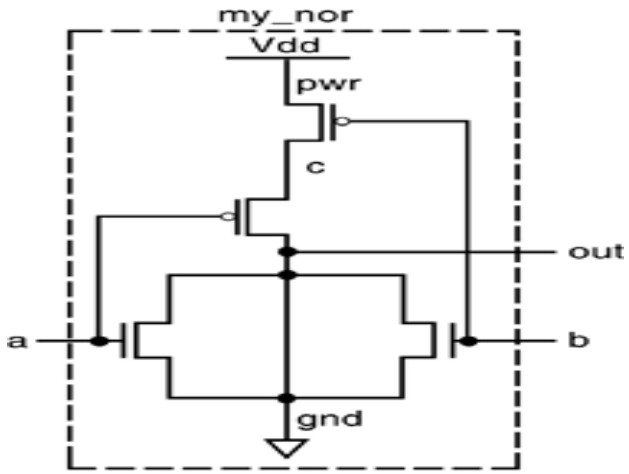
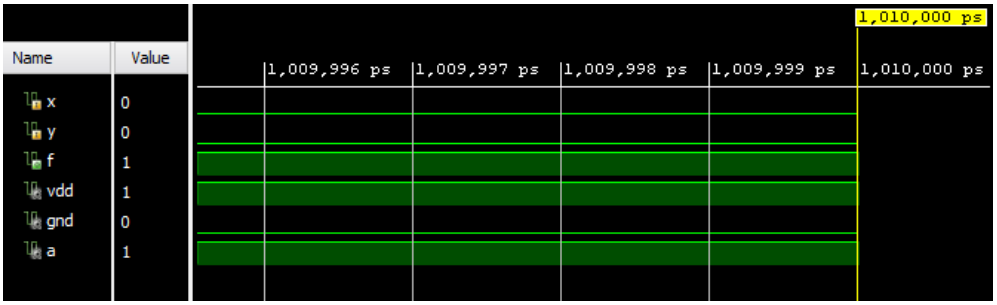


Fig 15.3:CMOS NOR

VERILOG CODE

```
modulecmosnor11( input x,
inputy, output f
);
supply1vdd; supply0gnd; wire
a;pmosp1(a,vdd,y);
pmosp2(f,a, x);
nmosn1(f,gnd,x)
;
nmosn2(f,gnd,y);
\endmodule
```

OUTPUT


Name
Value
x
0
y
0
f
1
vdd
1
gnd
0
a
1
1,009,996 ps
1,009,997 ps
1,009,998 ps
1,009,999 ps
1,010,000 ps
1,010,000 ps