Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

Department of Artificial Intelligence and Data Science

LAB MANUAL

Digital Design and Computer Organization Lab

SUBJECT CODE: BCS302

III Semester

Course coordinator
Nagapushpa B M
Associate Professor
Dept. Of AD

HOD
Dr.Gavisiddappa
Prof & Head
Dept. Of AD

PRACTICAL COMPONENT OF IPCC

Sl.NO	Experiments Simulation packages preferred: Multisim, Modelsim, PSpice or any other relevant
1.	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.
2.	Design a 4 bit full adder and subtractor and simulate the same using basic gates.
3.	Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.
4.	Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.
5.	Design Verilog HDL to implement Decimal adder.
6.	Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.
7.	Design Verilog program to implement types of De-Multiplexer.
8.	Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

Course outcomes (Course Skill Set):
At the end of the course, the student will be able to:

CO1: Apply the K–Map techniques to simplify various Boolean expressions.
CO2: Design different types of combinational and sequential circuits along with Verilog programs.
CO3: Describe the fundamentals of machine instructions, addressing modes and Processor performance.
CO4: Explain the approaches involved in achieving communication between processor and I/O devices.
CO5: Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance.

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.

● On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
● The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.
● The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.
● Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
● The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

Laboratory Session-1. Logic design components

The digital logic design area covers the digital building blocks, tools, and techniques in the design of computers and other digital systems. Digital logic design is one of the topic areas that differentiate computer engineers from electrical engineers and computer scientists. It covers a variety of basic topics, including switching theory, combinational and sequential logic circuits, and memory elements.

Circuit that takes the logical decision and the process are called logic gates. Each gate has one or moreinput and only one output.

OR, AND and NOT are basic gates. NAND, NOR are known as universal gates. Basic gates form these gates.

AND GATE:

The AND gate performs a logical multiplication commonly known as AND function. The output is high when both the inputs are high. The output is low level when any one of the inputs is low.

OR GATE:

The OR gate performs a logical addition commonly known as OR function. The output is high whenany one of the inputs is high. The output is low level when both the inputs are low.

NOT GATE:

The NOT gate is called an inverter. The output is high when the input is low. The output is low when the input is high.

NAND GATE:

The NAND gate is a contraction of AND-NOT. The output is high when both inputs are low and any one of the input is low .The output is low level when both inputs are high.

NOR GATE:

The NOR gate is a contraction of OR-NOT. The output is high when both inputs are low. The output is low when one or both inputs are high.

X-OR GATE:

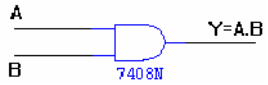
The output is high when any one of the inputs is high. The output is low when both the inputs are low and both the inputs are high.

PROCEDURE:

(i) Connections are given as per circuit diagram.
(ii) Logical inputs are given as per circuit diagram.
(iii) Observe the output and verify the truth table.

AND GATE:

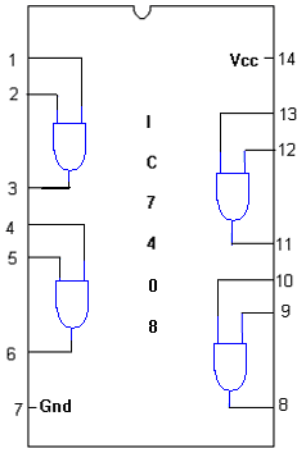
SYMBOL:



TRUTH TABLE

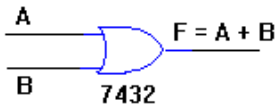
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

PIN DIAGRAM:



OR GATE:

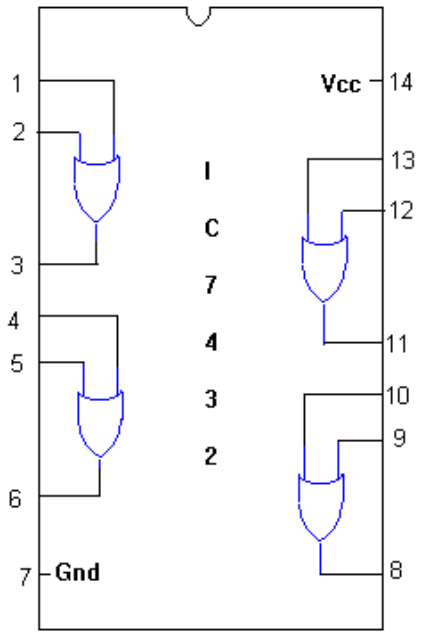
SYMBOL :



TRUTH TABLE

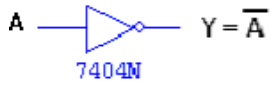
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

PIN DIAGRAM :

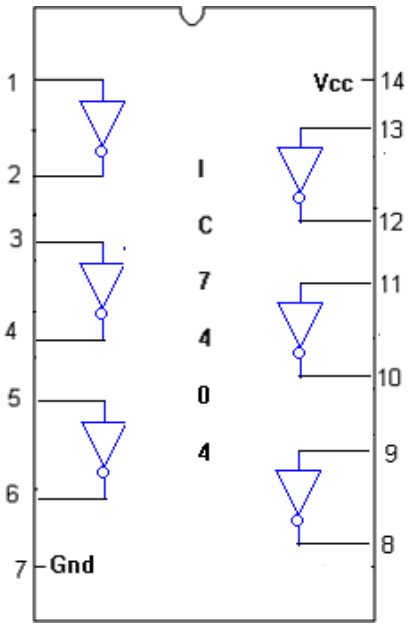


NOT GATE:

SYMBOL:



PIN DIAGRAM:

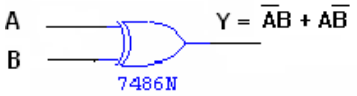


TRUTH TABLE :

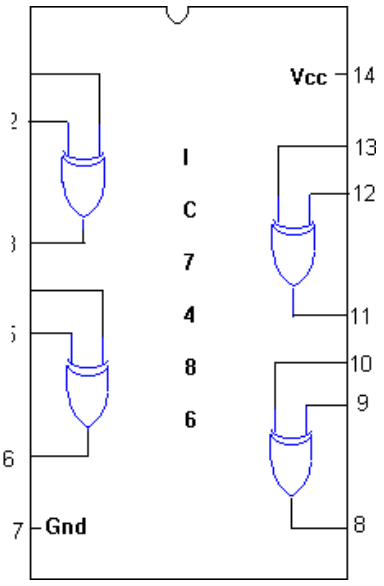
A	$\overline{A}$
0	1
1	0

X-OR GATE:

SYMBOL:



PIN DIAGRAM:

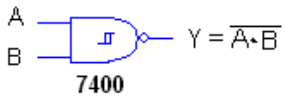


TRUTH TABLE :

A	B	$\overline{A}$B + A$\overline{B}$
0	0	0
0	1	1
1	0	1
1	1	0

2- INPUT NAND GATE:

SYMBOL:

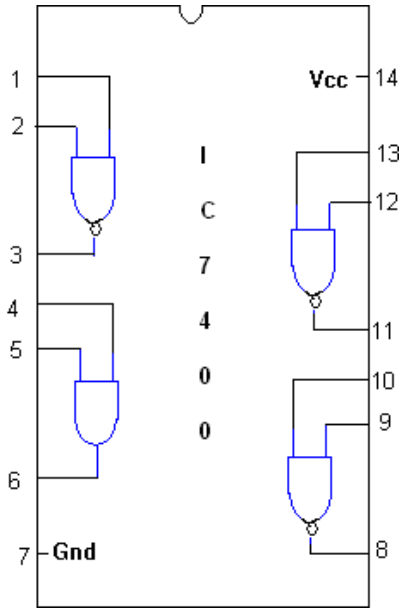


7400

TRUTH TABLE

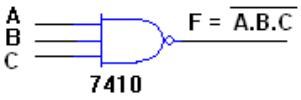
A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

PIN DIAGRAM:



3- INPUT NAND GATE:

SYMBOL :

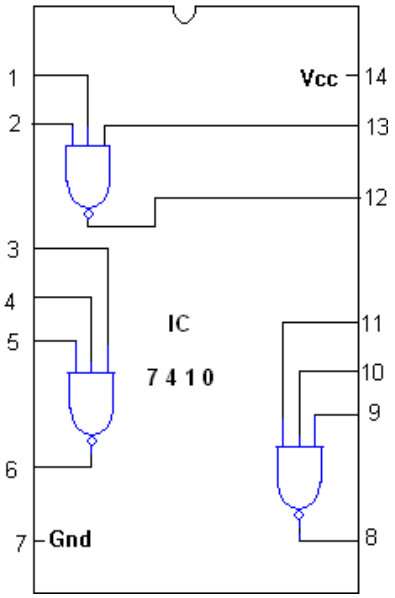


7410

TRUTH TABLE

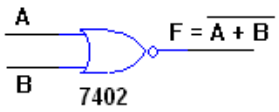
A	B	C	A.B.C
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

PIN DIAGRAM :

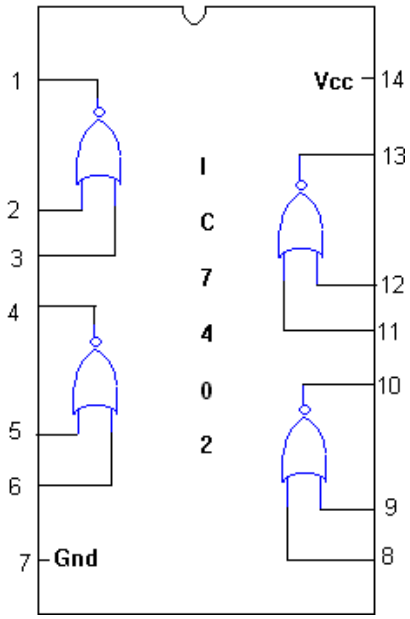


NOR GATE:

SYMBOL :



PIN DIAGRAM :



TRUTH TABLE

A	B	$\overline{A+B}$
0	0	1
0	1	1
1	0	1
1	1	0

INTRODUCTION TO XILINX

Xilinx ISE is a software tool produced by Xilinx for synthesis and analysis of HDL designs, which enables the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

ModelSim is a verification and simulation tool for VHDL, Verilog, System Verilog, and mixed language designs.

INTRODUCTION TO VHDL

HDL (Hardware Description Language) based design has established itself as the modern approach to design of digital systems, with VHDL (VHSIC Hardware Description Language) and Verilog HDL being the two dominant HDLs. Numerous universities thus introduce their students to VHDL (or Verilog). The prob-lem is that VHDL is complex due to its generality. Introducing students to the language first, and then showing them how to design digital systems with the language, tends to confuse students. The language issues tend to distract them from the understanding of digital components. And the synthesis subset issues of the language add to the confusion.

VHDL stands for **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. In the mid-1980's the U.S. Department of Defense and the IEEE sponsored the development of this hard-ware description language with the goal to develop very high-speed integrated circuit. It has become now one of industry's standard languages used to describe digital systems. The other widely used hard- ware description language is Verilog. Both are powerful languages that allow you to describe and simu- late complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language)which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry. This tutorial deals with VHDL, as described by the IEEE standard 1076-1993.

Although these languages look similar as conventional programming languages, there are some important differences. A hardware description language is inherently parallel, i.e. commands, which cor-respond to logic gates, are executed (computed) in parallel, as soon as a new input arrives. A HDL pro-gram mimics the behavior of a physical, usually digital, system. It also allows incorporation of timing specifications (gate delays) as well as to describe a system as an interconnection of different compo- nents

Sample Programs:

1. Write the Verilog /VHDL code for a 2 Input AND gate. Simulate and verify its working.

```
Entity    And1  is
        Port (A, B: IN STD_LOGIC;
                C: OUT STD_LOGIC);
End     And1;
Architecture Behavioral of And1 is
        Begin
                C <= A AND B;
End     Behavioral;
```

NOTE: write the VHDL code for remaining basic and universal gates. Simulate and realize the output.

2. Write the Verilog /VHDL code for a half adder. Simulate and verify its working.

```
Entity      Half_adder  is
        Port (A, B: IN STD_LOGIC;
              SUM, CARRY: OUT STD_LOGIC);
End     Half_adder;
Architecture  Behavioral  of  Half_adder  is
        Begin
              SUM  <=  A  XOR  B;
              CARRY  <=  A  AND
              B;
End     Behavioral;
```

EXPERIMENT NO. 1

Given a 4-variable logic expression, simplify it using appropriate technique and implement the same using basic gates. Consider f(a,b,c,d)= Σm(1,3,7,11,15)+ Σd(0,2,5).

AIM: To simplify the given expression using appropriate technique and implement using basic gates.

COMPONENTS REQUIRED: Digital IC trainer kit, patch chards, IC7408, IC7432,IC7404.

THEORY: digital electronic circuits operate with voltages of two logic levels namely logic low and logic high. The range of voltages corresponding to logic low is represented with '0' & similarly the range of voltages corresponding to logic high is represented with '1'.
The basic digital electronic circuit that has one or more inputs and single output is known as logic gate.
We know that Boolean functions can be represented either in sum of products form or in in product of sums form based on the requirement.so we can implement the functions by using basic gates. the basic gates are AND, OR & NOT GATES.

PROCEDURE:
1. Verify all components and patch cords for their good working condition.
2. Make the connection as shown in the circuit diagram.
3. Give supply to the trainer kit.
4. Provide input data to circuit via switches and verify the function table and truth table.

TRUTH TABLE

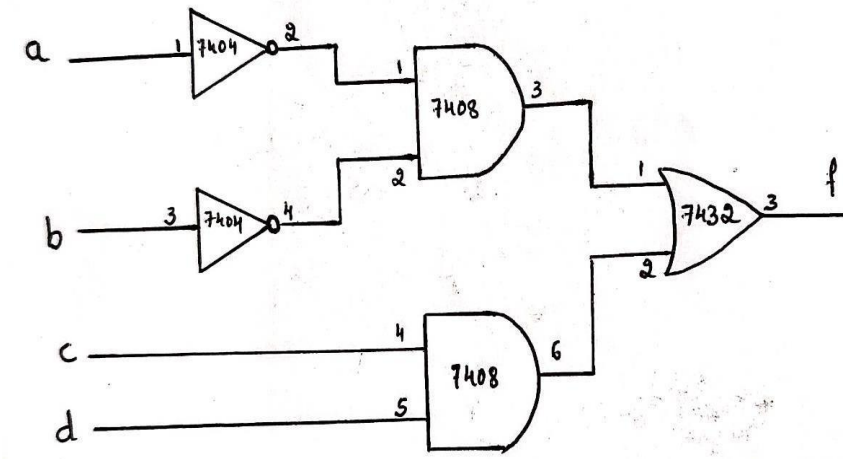
Decimal	inputs				outputs
	a	b	c	d	F
0	0	0	0	0	X
1	0	0	0	1	1
2	0	0	1	0	X
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	X
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

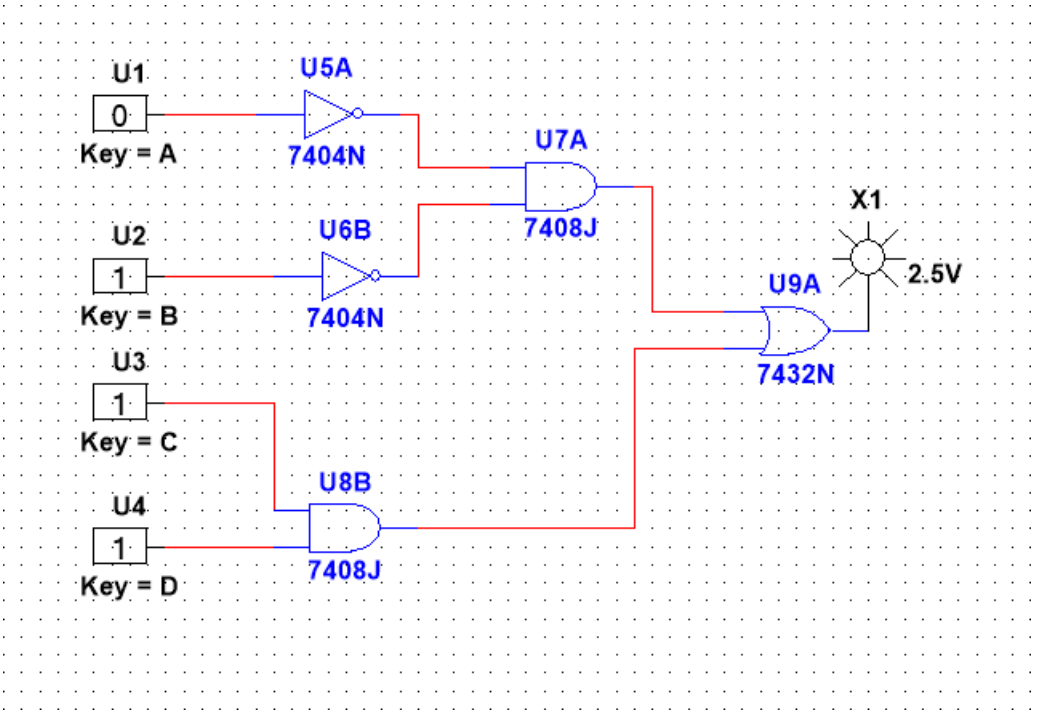
K-MAP simplification:

ab \ cd	00	01	11	10
00	X			
01	1			
11	1	1	1	1
10	X			

f = cd+a'b'

CIRCUIT DIAGRAM:





RESULT: The given 4-variable logic expression is implemented using basic gates and the corresponding truth table has been verified.

EXPERIMENT NO.2

Design a 4-bit full adder and subtractor and simulate the same using basic gates.

AIM: To Design a 4-bit full adder and subtractor and simulate the same using basic gates.
COMPONENTS REQUIRED: Digital IC trainer kit, patch chards, IC7408,7432,7486.
THEORY:
Full Adder is a combinational circuit that performs the addition of three bits (two significant bits and a previous carry) is a full adder.
A 4-bit full adder is a digital circuit that performs the arithmetic operation of addition on four bits at a time. It consists of four 1-bit full adder circuits connected in series, with the carry output of each adder connected to the carry input of the next adder. This allows the 4-bit full adder to handle both the addition of the four bits and the carrying of any overflow bits from one bit position to the next.
A full subtractor is a **combinational circuit** that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit.

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

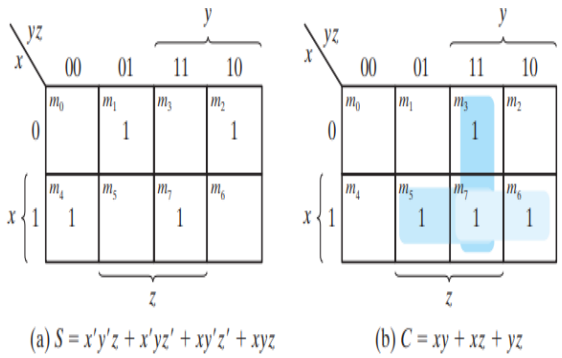


FIGURE 4.6
K-Maps for full adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$

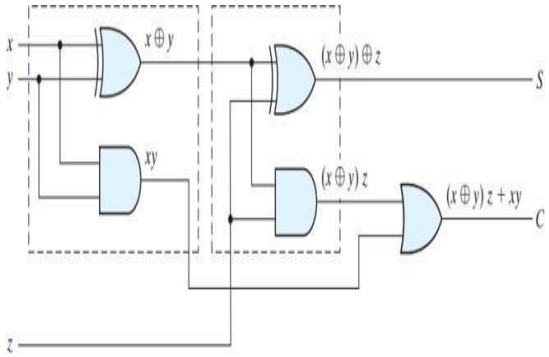
$$= z'(xy' + x'y) + z(xy + x'y')$$

$$= z'(xy' + x'y) + z(xy' + x'y)'$$

$$S = z \oplus (x \oplus y)$$

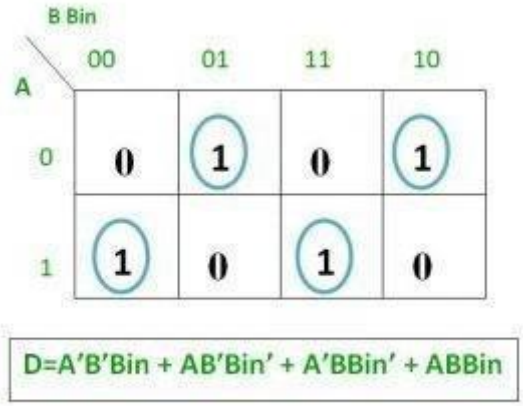
$$C = xy + xz + yz$$

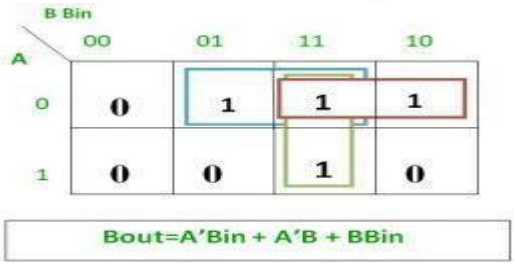
$$C = z(xy' + x'y) + xy$$



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





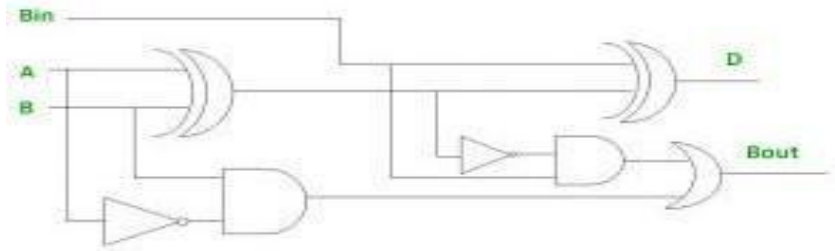
Logical expression for difference –

```
D    = A'B'Bin + A'BBin' + AB'Bin' + ABBin
     = Bin(A'B' + AB)  + Bin'(AB' + A'B)
     = Bin( A XNOR B) + Bin'(A XOR B)
     = Bin (A XOR B)'  +  Bin'(A XOR B)
     = Bin XOR (A XOR B)
     = (A XOR B) XOR Bin
```

Logical expression for borrow –

```
Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = A'B'Bin +A'BBin' + A'BBin + A'BBin + A'BBin + ABBin
     = A'Bin(B + B') + A'B(Bin + Bin') + BBin(A + A')
     = A'Bin + A'B + BBin

OR

Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = Bin(AB + A'B') + A'B(Bin + Bin')
     = Bin( A XNOR B) + A'B
     = Bin (A XOR B)' + A'B
```

Logic Diagram



4- BIT FULL ADDER/SUBTRACTOR CIRCUIT DIAGRAM:

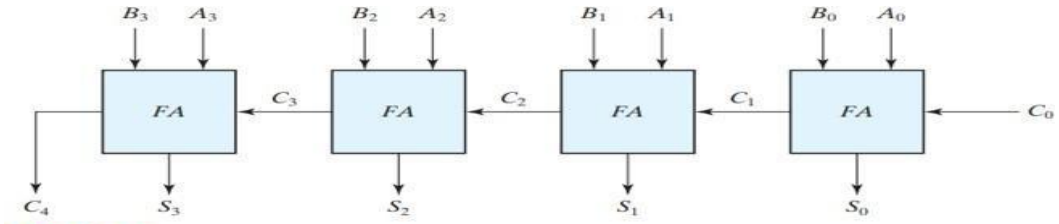
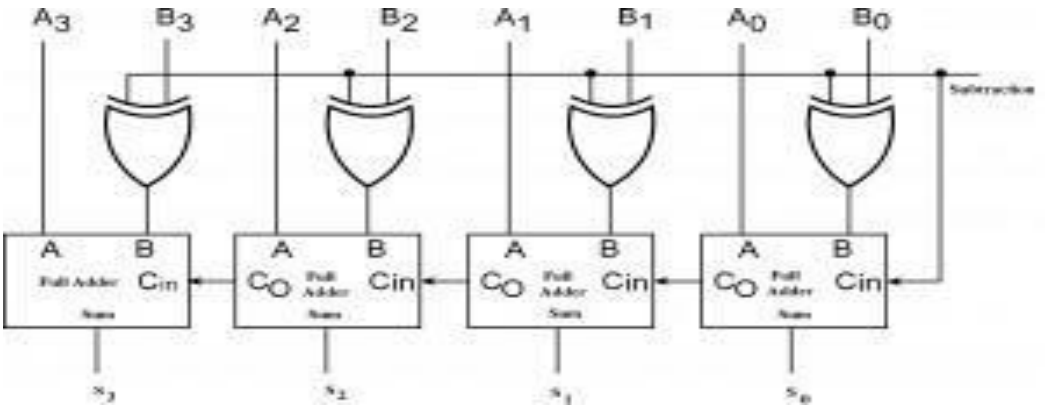
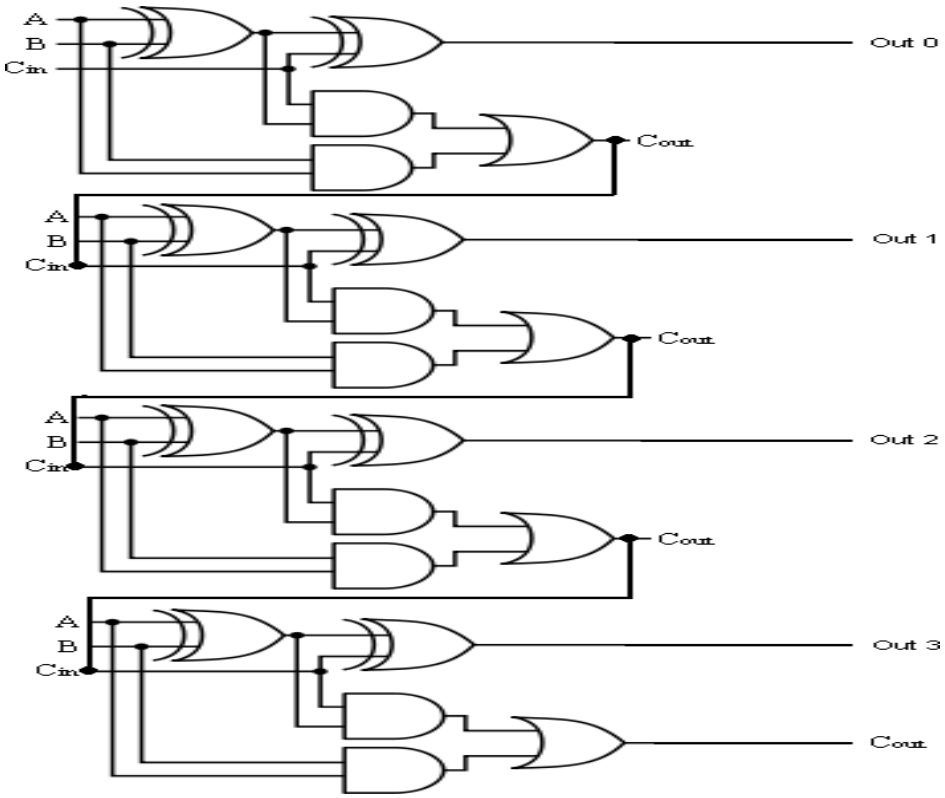


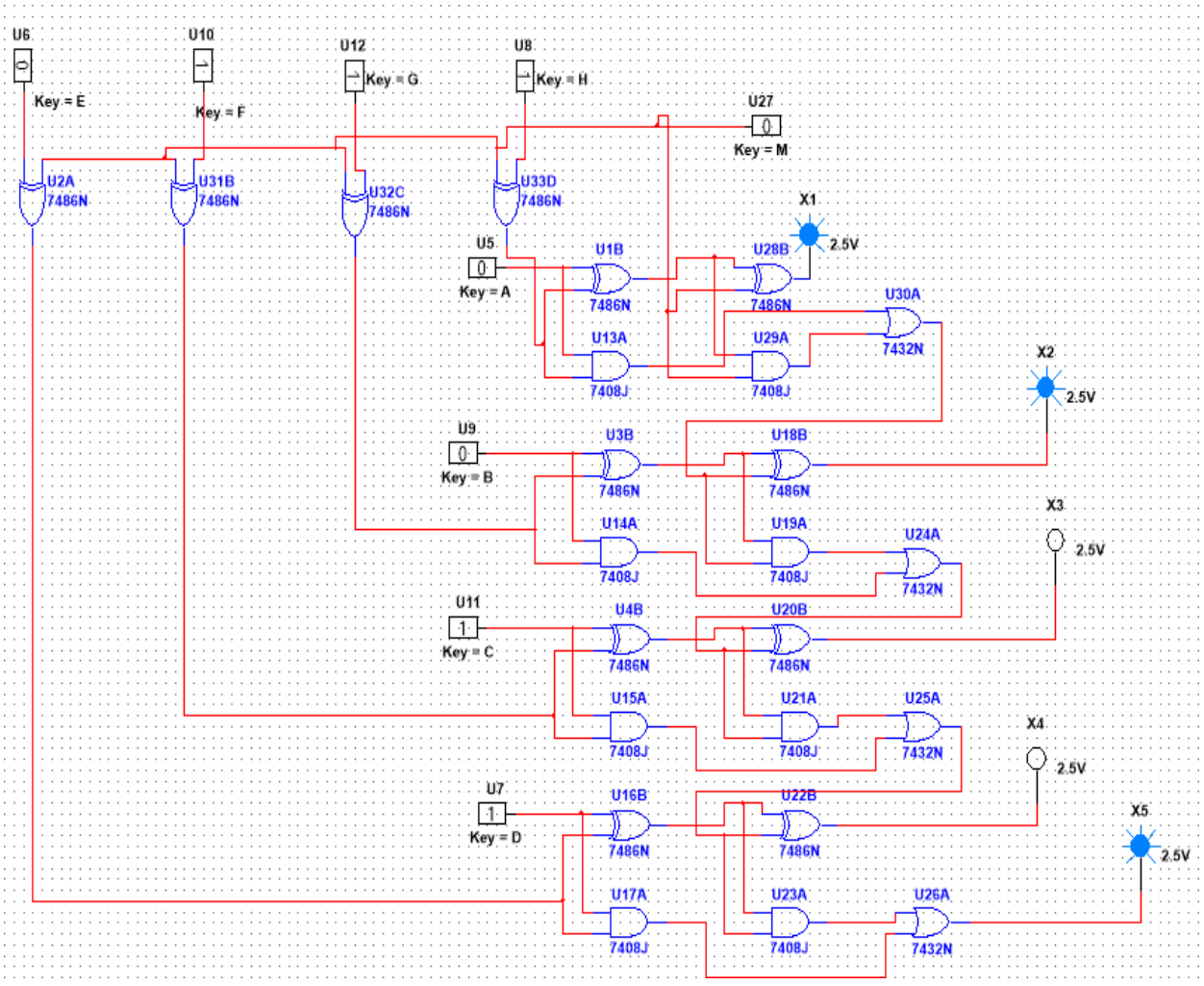
FIGURE 4.9
Four-bit adder





4-BIT FULL ADDER/SUBTRACTOR TRUTH TABLE:

A3 A2 A1 A0	**12**	**1 1 0 0**	**12**	**1 1 0 0**
B3 B2 B1 B0	**+7**	**0 1 1 1**	**-7**	**0 1 1 1**
C3 S3 S2 S1 S0	**19**	**1 0 0 1 1**	**2'S Complement (7)**	**1 0 0 1**
			5	**1 0 1 0 1**



RESULT: The 4-bit full adder and subtractor is designed and simulated using basic gates and its truth table has been verified.

EXPERIMENT NO. 3:

Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.

Aim: To design Verilog HDL to implement simple 2:1 multiplexer circuit using structural, Data flow and Behavioural model.

Theory: For any sequential or combination circuit, modelling aspects come to mind.

1. Schematic
2. Truth table
3. And logical expression

Similarly, when it is about Verilog HDL, three modelling aspects:

1. Structural
2. Behavioural
3. Data flow

The **structural modelling style** is the lowest level of abstraction obtained using logic gates. Similar to schematic or circuit diagrams of the digital circuit, Verilog uses primitive gates to compile and synthesize the program.

Of course, this abstraction can't be understood by humans. Machines, however, have the definite capability of compiling and logically synthesizing the code.

The language supports multiple gates such as ***and***, ***or***, ***nand***, ***xor***, ***nor***, and ***xnor***. You can also use tri-state gates and multiple-output gates such as ***bufif1***, ***bufif0***, ***notif1***, ***notif0***, ***not***, and ***buf***.

Here's the syntax of these gates:

- **and** | **or** | **nand** | **xor** | **nor** | **xnor** [instance name] (output, input1, …., input n);
- **not | buf** [instance name] (output 1, output2, …., output n, input);
- **bufif1** | **bufif0** | **notif1** | **notif0** [instance name] (output, input, control);

Behavioural modelling is the highest of level abstraction that completely depends on the circuit behaviour or on the truth table.

If you know how your circuit will behave, you can design it. In fact, you can design the module without knowing the components of the hardware.

However, even though it is the closest in terms of natural language understanding of the circuit functionality, this modelling type is hardest to implement and synthesize. Hence, it is utilized for complex circuits such as pure combinational or sequential circuits.

A module developed using behavioural modelling contains ***initial*** or ***always*** statements, which are executed concurrently (according to the parallelism of the model). The procedural statements in the module are executed sequentially.

At time=0, both the ***initial*** and ***always*** will execute and then, ***always*** statements run for the remaining time.

Here's the syntax:

always [timing control] procedural_statements;

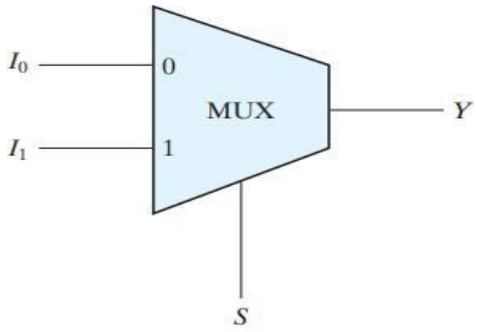
initial [timing control] procedural_statements;

The data flow is a medium level abstraction, which is achieved by defining the data flow of the module. You can design the module by defining and expressing input signals which are assigned to the output, very much similar to logical expressions.

For most of the modules, data flow modelling is simple to implement and can be easily translated to structure such as in the case of combinational circuits.

The combinational circuits use continuous assignments, where value is defined for a data ***net***.

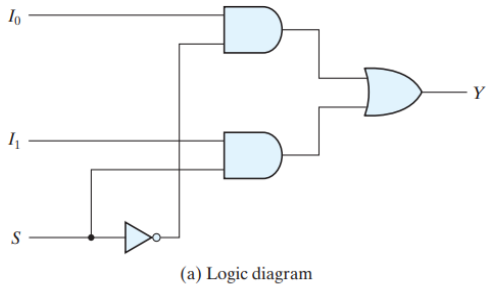
assign[delay] LHS_net = RHS_expression



(b) Block diagram

input	output
S	Y
0	I_0
1	I_1

Y=S' $I0$ + S $I1$



(a) Logic diagram

FIGURE 4.24
Two-to-one-line multiplexer

STRUCTURAL MODELLING

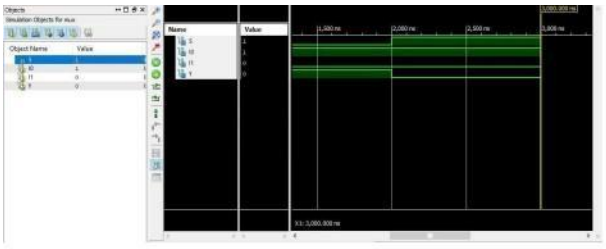
```
module mux_2to1_structural (S I0, I1, Y);
 input S I0, I1;
output Y;

wire A1, B1, S1;

and (A1, S1, I0);
and (B1, S, I1);
not (s1, s);
or (Y, A1, B1);
endmodule
```

BEHAVIOURAL MODELLING

```
module mux_2to1_behavioral (S I0, I1, Y);
input S I0, I1;
output Y;
reg Y;
always @(S or I0 or I1)
   if (S == 0)
      Y = I0;
   else
      Y = I1;
endmodule
```

DATA FLOW MODELLING

```
module mux_2to1_dataflow (S I0, I1, Y);
   input S, I0, I1;
   output Y;
assign Y = ((!S) && I0) ||(S&&I1);
endmodule
```



Result: A Verilog HDL to implement simple 2:1 mux using structural, Data flow and Behavioural model has been simulated and its truth table has been verified.

EXPERIMENT NO.4

Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.

AIM: To Design a Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.

Theory:

We have different types of digital devices like computers, calculators that can perform a variety of processing functions like addition, subtraction, multiplication, division, etc. The most basic arithmetic operation that the ALU (arithmetic logic unit) of a computer performs is the **addition** of two or more binary numbers. To perform the operation of addition, a combinational logic circuit, named **Adder** is used.

Adders are classified into two types namely −

- Half Adder
- Full Adder

A **half-adder** is a combinational logic circuit that performs the addition of only two bits (binary digits). Whereas, a **full-adder** is a combination circuit that performs three bits (binary digits), where two are the significant bits and one is a carry from previous execution.

The subtraction of two binary numbers can be performed by taking the 1's or 2's complement of the inputs. By this method, the subtraction operation becomes an addition operation, and thus can be performed by using the adder circuits. This results in the reduction of hardware and cost.

In the subtraction operation, each subtrahend bit (B) of the number is subtracted from its corresponding significant minuend bit (A) to obtain a difference bit.

Similar to adders, subtractors are also of two types −

- Half Subtractor
- Full Subtractor

half-subtractor is a combinational logic circuit that have two inputs and two outputs (i.e. difference and borrow).

A **full-subtractor** is a combinational circuit that has three inputs A, B, bin and two outputs d and b.

Truth Table of Half Adder

Inputs		Outputs	
A	**B**	**S (Sum)**	**C (Carry)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

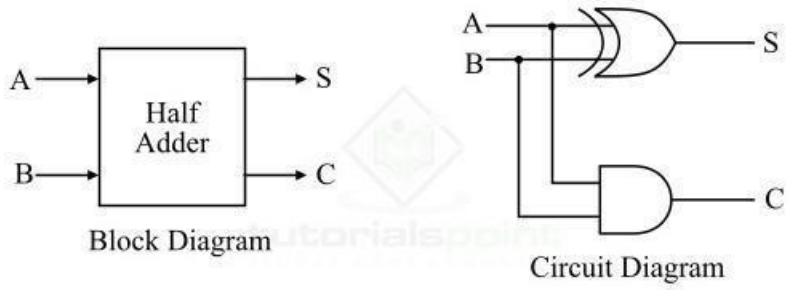


Figure 1 - Half Adder

S=$A'B + AB'$ C=AB

Verilog Code

```
module halfadder(A,B,S,C);
input A,B;
output S,C;
assign S= (!A && B) || (A && (!B));
assign C= (A&&B);
endmodule
```

Truth Table of Half Subtractor

Inputs		Outputs	
A	**B**	**d (Difference)**	**b (Borrow)**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

d=A′B+AB′

b=A′B

Verilog Code

```
module halfsub(A,B,d,b);
input A, B;
output d, b;
assign d= (!A && B) || (A && (!B));
assign b= ((!A) && B);
endmodule
```

FULL ADDER

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

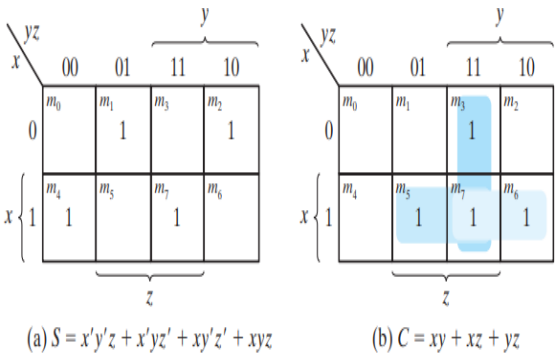


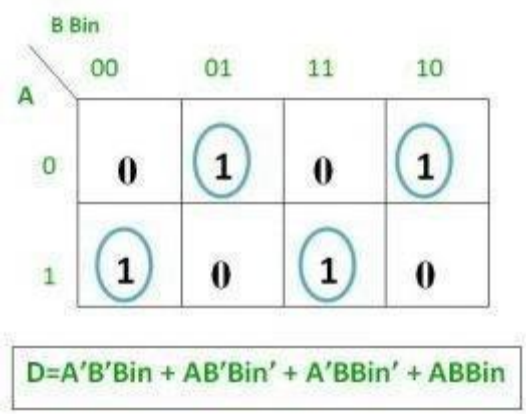
FIGURE 4.6
K-Maps for full adder

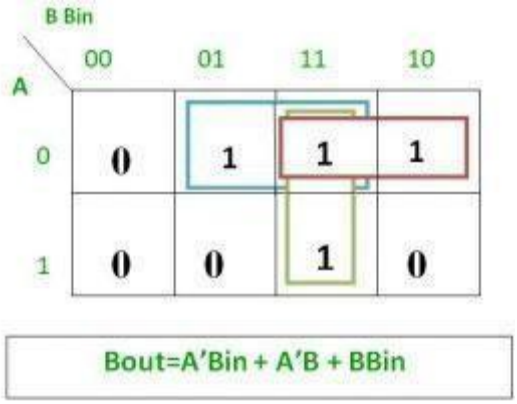
Verilog Code

```
module fulladder(x,y,z,c,s);
input x,y,z;
output c,s;
assign c = (x && y) || (x && z) || (y && z);
assign s = ((! x) && (! y) && z) || ((! x) && y (! z)) || (x && (! y) && (! z))|| (x && y && z);
endmodule
```

Full Subtractor

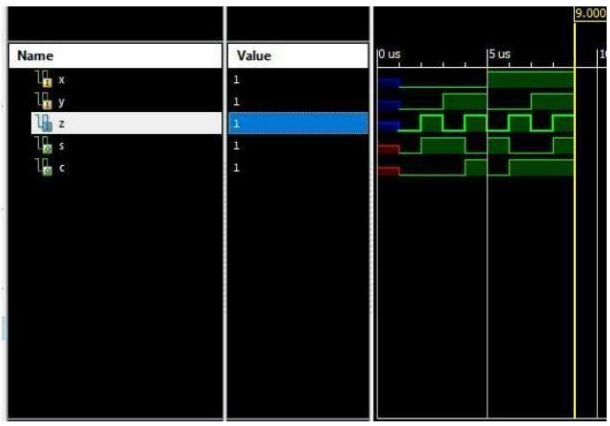
INPUT			OUTPUT	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

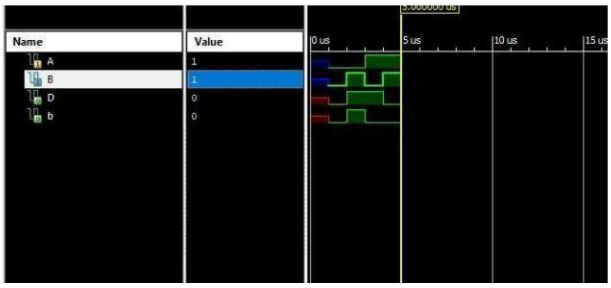


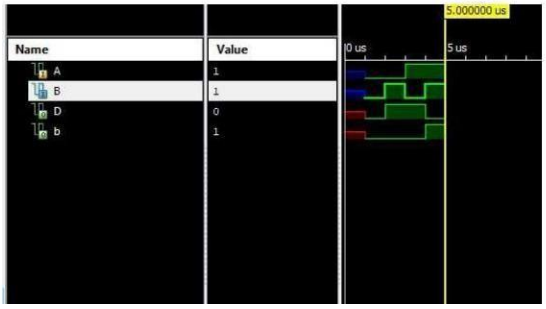


Verilog Code

```
module fullsub(a,b,bin,D,Bout);
input a,b,bin;
output D,Bout;
assign c = ((!a) && bin) || ((!a) && b) || (b && bin);
assign s = ((! a) && (! b) && bin) || ((x) && !b (! bin)) || ((!a) && (b) && (! bin)) || (a && b && bin);
endmodule
```







Result: The Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor has been simulated and its truth table has been verified.

EXPERIMENT NO.5

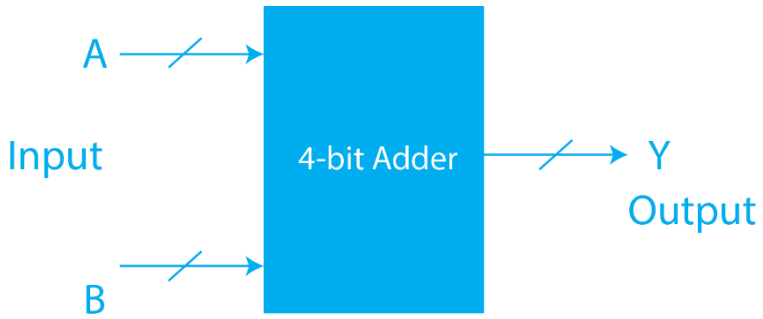
Design Verilog HDL to implement Decimal adder.

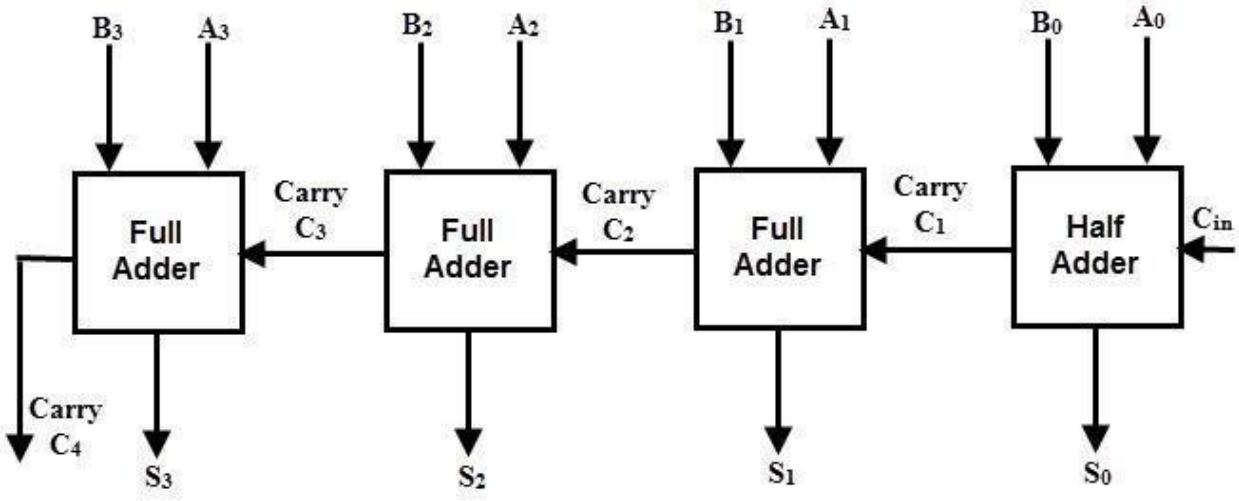
Aim: To design Verilog HDL to implement Decimal adder.

Theory:

A BCD adder, also known as a Binary-Coded Decimal adder, is a digital circuit that performs addition operations on Binary-Coded Decimal numbers. BCD is a numerical representation that uses a four-bit binary code to represent each decimal digit from 0 to 9.

Circuit Diagram:





Truth table:

A3 A2 A1 A0	**12**	**1**	**1**	**0**	**0**
B3 B2 B1 B0	**+7**	**0**	**1**	**1**	**1**
C3 S3 S2 S1 S0	**19**	**1 0**	**0**	**1**	**1**

Verilog Code:

```
module four_bit_adder(A, B, sum, carry);
input [0:3] A;
input [0:3] B;
output [0:3] sum;
output carry;
assign {carry,sum}= A+B;
endmodule
```

Result: Verilog HDL to implement Decimal adder has been simulated and verified.

EXPERIMENT NO.6

Design Verilog program to implement Different types of multiplexers like 2:1, 4:1 and 8:1.

Aim: To design Verilog program to implement Different types of multiplexers like 2:1, 4:1 and 8:1.

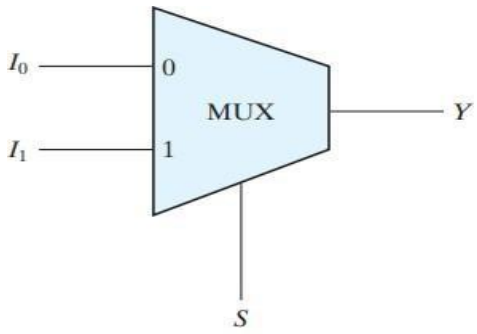
THEORY: A multiplexer or simply "mux" is a device that selects between a number of input signals. In its simplest form, a multiplexer will have two input signals, 1 control input, and 1 output. The number of inputs is generally a multiple of 2 (2, 4, 8, 16, etc), the number of outputs is 1, and n control inputs are used to select one of the data inputs.

The multiplexer output value is same as that of the selected data input. In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input.

Multiplexers are used in building digital semiconductors such as CPUs and graphics controllers. They are also used in communications; the telephone network is an example of a very large virtual mux built from many smaller discrete ones.

Circuit Diagram:

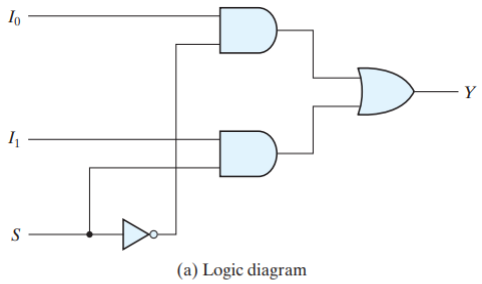
2:1 mux



(b) Block diagram

input	output
S	Y
0	I_0
1	I_1

Y=$S' I0 + S I1$



(a) Logic diagram

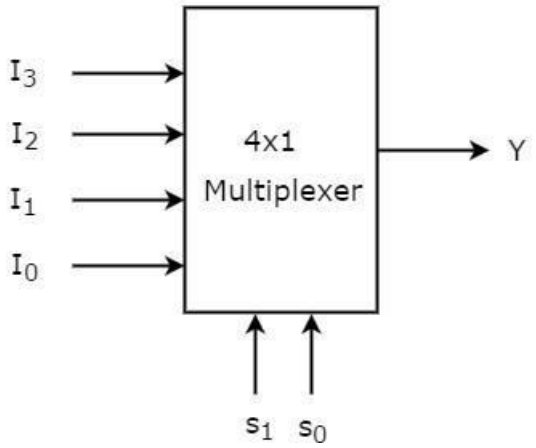
FIGURE 4.24
Two-to-one-line multiplexer

Verilog Code for 2:1 multiplexer

```
module mux_2to1_behavioral (S I0, I1, Y);
input S I0, I1;
output Y;
reg Y;
always @(S or I0 or I1)
   if (S == 0)
      Y = I0;
   else
      Y = I1;
Endmodule
```

Circuit Diagram:

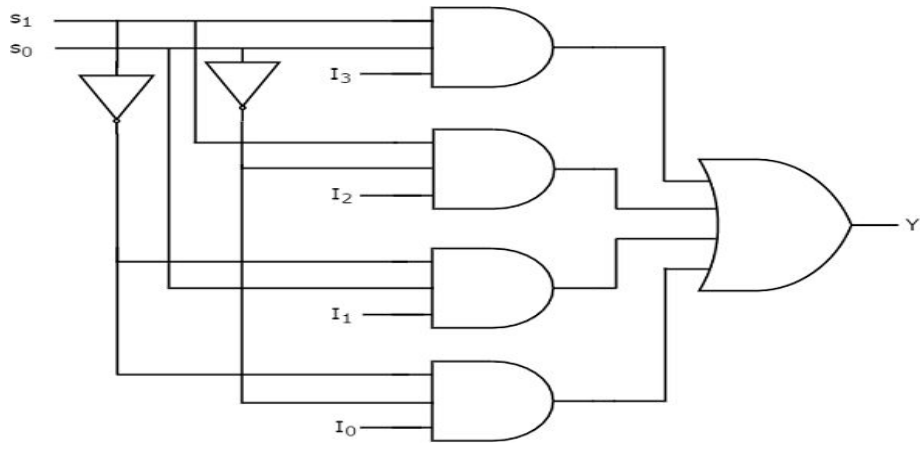
4:1 mux



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

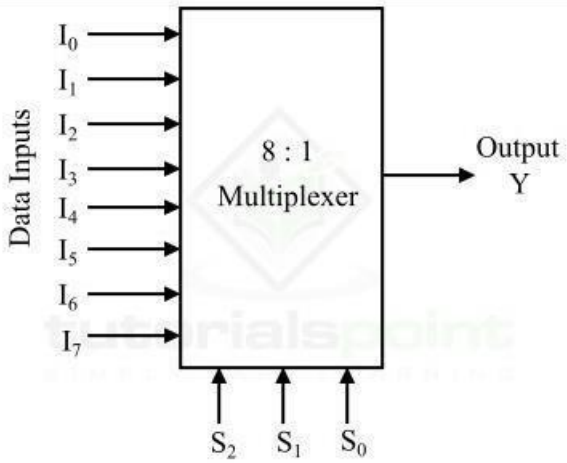


Verilog code for 4:1 Mux

```
module mux4x1(I, S,Y);
input [0:3] I;
input [0:1] S;
output reg Y;
always@(I or S)
case (S)
2'b00 : Y=I[0];
2'b01 : Y=I[1];
2'b10 : Y=I[2];
2'b11 : Y=I[3];
endcase
endmodule
```

Circuit Diagram:

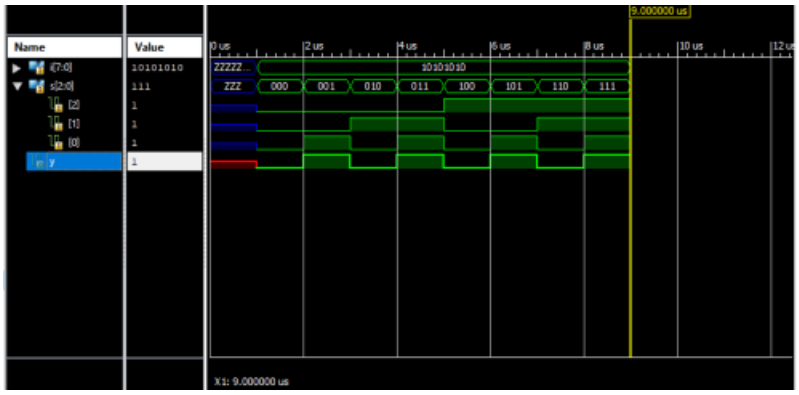
8:1 mux

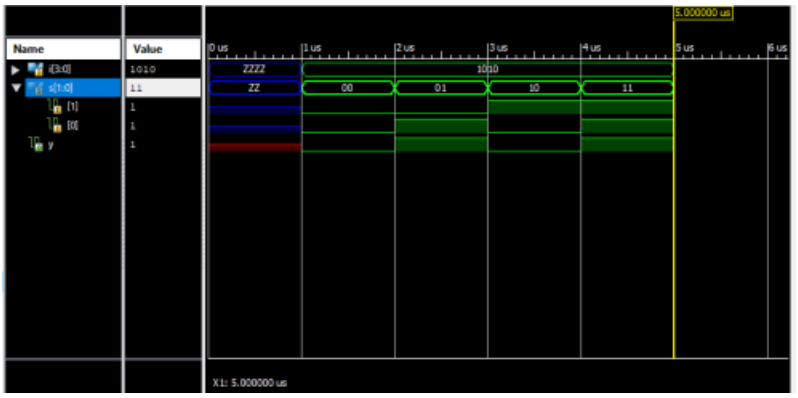


S2	S1	S0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Verilog code for 8:1 Mux

```
module mux8x1(I, S,Y);
input [0:7] I;
input [0:2] S;
output reg Y;
always@(I or S)
case (S)
3'b000 : Y=I[0];
3'b001 : Y=I[1];
3'b010 : Y=I[2];
3'b011 : Y=I[3];
3'b100:  Y=I[4];
3'b101 : Y=I[5];
3'b110 : Y=I[6];
3'b111 : Y=I[7];
endcase
endmodule
```





Result: Verilog program to implement Different types of multiplexers like 2:1, 4:1 and 8:1has been simulated and verified**.**

EXPERIMENT NO.7

Design Verilog program to implement types of De-Multiplexer.

Aim: To design Verilog program to implement Different types of de-multiplexers like 1:2, 1:4 and 1:8.

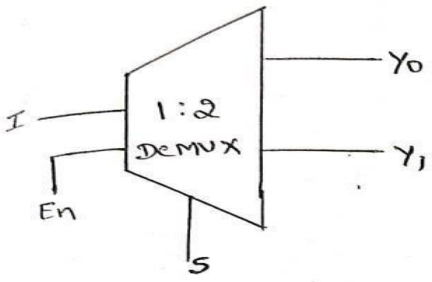
Theory: A demultiplexer (DEMUX) is a combinational circuit that works exactly opposite to a multiplexer. A DEMUX has a single input line that connects to any one of the output lines based on its control input signal (or selection lines)

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

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

Circuit Diagram:

1:2 Demux

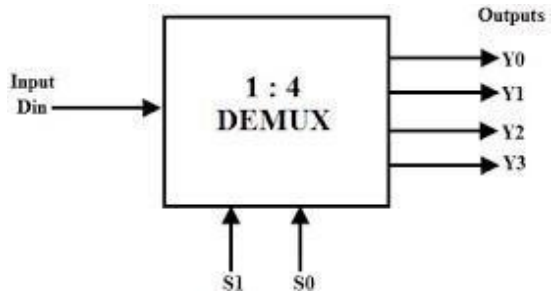


Input			output	
En	S	I	Y_0	Y_1
0	x	x	0	0
1	0	I	I	0
1	1	I	0	I

$Y_0 = ES'I$ & $Y_1 = ESI$

```
module demux2to1(en,s,i,y);
input en,s,i;
output [0:1] y;
assign y[0]= e && (!s) && i;
assign y[1]= e && s && i;
endmodule
```

1:4 Demux

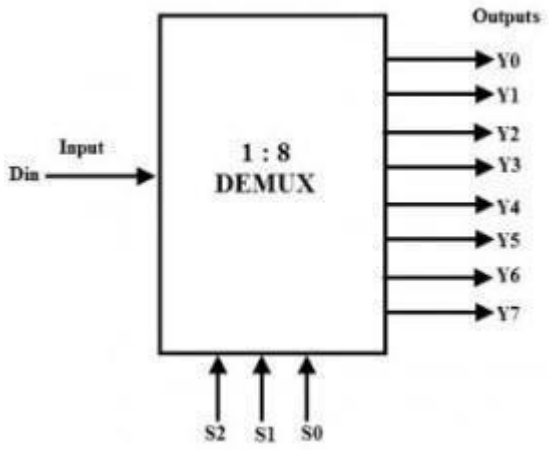


Din	S1	S0	Y0	Y1	Y2	Y3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Verilog code

```
module demux_1_4(y,s,din);
output reg [3:0]y;
input [1:0]s;
input din;
always @(din or s)
case(s)
2'b00: y[0]=din;
2'b01: y[1]=din;
2'b10: y[2]=din;
2'b11: y[3]=din;
endcase
endmodule
```

1:8 Demux



din	S2	S1	S0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

```
module demux_1_8(y,s,din);
output reg [7:0]y;
input [2:0]s;
input a;

always @(din or s)
case(s)
3'b000: y[0]=din;
3'b001: y[1]=din;
3'b010: y[2]=din;
3'b011: y[3]=din;
3'b100: y[4]=din;
3'b101: y[5]=din;
3'b110: y[6]=din;
3'b111: y[7]=din;
endcase
endmodule
```

Result: Verilog program to implement Different types of demultiplexers like 1:2, 1:4 and 1:8 has been simulated and verified**.**

EXPERIMENT NO.8
Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

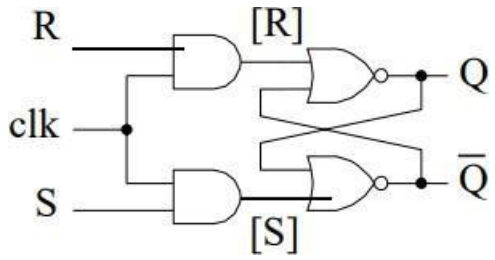
AIM: To Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

Theory: A flip flop is an electronic circuit with two stable states that can be used to store binary data. The stored data can be changed by applying varying inputs. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems.

1. R-S flip flop
2. D flip flop
3. J-K flip flop
4. T flip flop

Circuit Diagram:

SR-flipflop

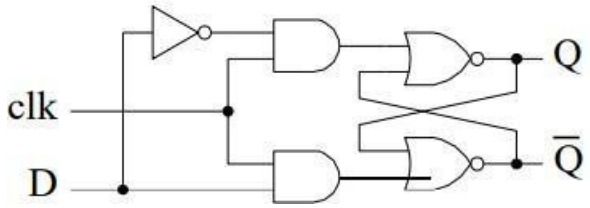


clk	S	R	Q_{n+1}
1	0	0	Q_n
1	0	1	0
1	1	0	1
1	1	1	X
0	x	x	Q_n

Verilog code

```
module srff(sr,clk,q,qb);
input[1:0] sr;
input clk;
output reg q,qb;
always@(poseedge clk)
case(sr)
2'b00:q=q;
2'b01:q=0;
2'b10:q=1;
2'b11:q=1'bz;
endcase
assign qb=(~q);
endmodule
```

D-flipflop

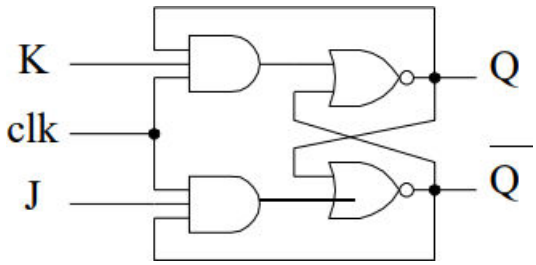


clk	D	Q_{n+1}
0	x	Q_n
1	0	0
1	1	1

Verilog code

```
module dff(d,clk,q,qb);
input d,clk;
output reg q,qb;
always@(posedge clk)
begin
q<=d;
end
assign qb=~q;
endmodule
```

JK-flipflop

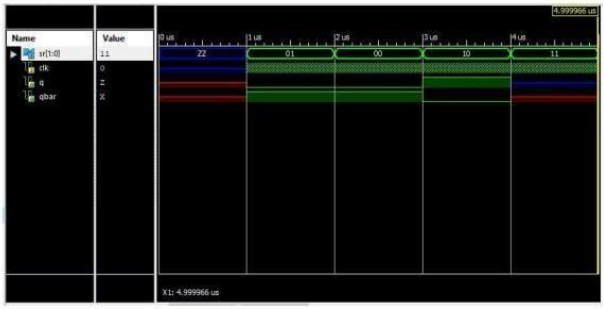


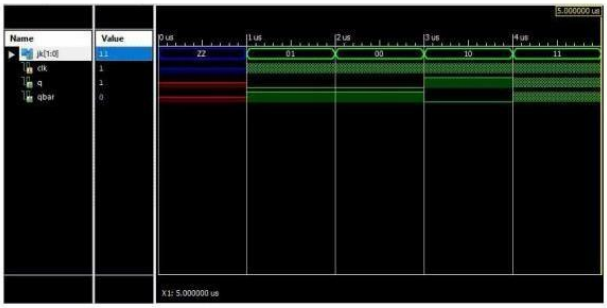
clk	J	K	Q_{n+1}
1	0	0	Q_n (no-change)
1	0	1	0(reset)
1	1	0	1(set)
1	1	1	Q_n'(toggle)
0	x	x	Q_n

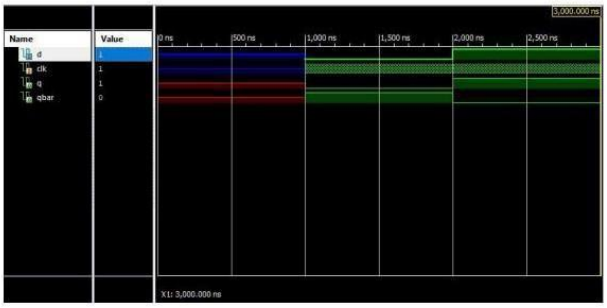
Verilog code

```
module jkff(jk,clk,q,qb);
input[1:0] jk;
input clk;
output reg q,qb;
always@(poseedge clk)
case(jk)
2'b00:q=q;
```

2'b01:q=0;
2'b10:q=1;
2'b11:q= ~q;
endcase
assign qb=(~q);
endmodule







Result: Verilog program to implement various types of Flip-Flops such as SR, JK and D been simulated and verified**.**