ವಿಶ್ವೇಶ್ವರಯ್ಯ ತಾಂತ್ರಿಕ ವಿಶ್ವವಿದ್ಯಾಲಯ, ಬೆಳಗಾವಿ

VISVESVARAYA TECHNOLOGICAL UNIVERSITY - BELAGAVI

BANGALORE INSTITUTE OF TECHNOLOGY

K. R. ROAD, V. V. PURA, BENGALURU-560 004

Department of Artificial Intelligence & Machine Learning

BCS302

Digital Design and Computer Organization

Laboratory Manual

III SEMESTER

Prepared by:

Prof. Mr. Manoj Kumar H

Bangalore Institute of Technology

K. R. Road, V. V. Pura, Bengaluru 560004

Department of Artificial Intelligence & Machine Learning

Digital Design Computer Organization

Course Code: BCS302

Module	Sl. No.	Name of Experiment
1	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.
2	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 multiplexers 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.

Bangalore Institute of Technology

K. R. Road, V. V. Pura, Bengaluru 560004

Department of Artificial Intelligence & Machine Learning

Digital Design and Computer Organization

Course Code: BCS302

Lesson Planning / Schedule of Experiments

Sl. No.	Name of Experiment	To be completed
1.	Understand the different pin diagrams, Demonstrating the usage of the software PSpice Student and realize simple circuits with hard ware components.	Week 1
2.	Demonstrating the usage of the software Xilinx in order to realize the basic gates, logical circuits. And also realize with trainer kit.	
3.	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same in HDL simulator.	Week 2
4.	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.	Week 3
5.	Design a 4 bit full adder and subtractor and simulate the same using basic gates.	Week 4
6.	Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.	Week 5
7.	Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.	Week 6
8.	Design Verilog HDL to implement Decimal adder	Week 7
9.	Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.	Week 8
10.	Design Verilog program to implement types of De-Multiplexer.	Week 9
11.	Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.	Week 10
LAB TEST 1		

PSpice

Working Steps of PSpice for simple circuit/gates:

1. Go to Draw →Select Get New Part.
2. Type the Part Name →Click on Place & Close.
3. In Toolbar →Select Draw Wire → Connect it to Component → Select Voltage Level Marker→ Place it on the wire you have connected.
4. Double click on the Voltage Marker→ Name the input & output Connections.
5. Go to Draw →Select Get New Part→ Select the Digital Clock pulse (Digclk) →Connect it with the component→ Double click on the Digital Clock pulse→ Set On Time &Off Time Values with Milliseconds (ms).
6. Go to Setup Analysis →Select Transient→ Select Final Time & Set the number of clock pulses you want in output.
7. Now save your File→ Click on Schematics in Toolbar.
8. Output Window will be displayed.

Verilog HDL Language Overview:

Introduction:

Verilog HDL is a **Hardware Description Language (HDL)**. A Hardware Description Language is a language used to describe a digital system, for example, a computer or a component of a computer. One may describe a digital system at several levels. For example, an HDL might describe the layout of the wires, resistors and transistors on an **Integrated Circuit (IC)** chip, i. e., and the **switch level**. Or, it might describe the logical gates and flip flops in a digital system,i. e., the **gate level**.

An even higher level describes the registers and the transfers of vectors of information between registers. This is called the **Register TransferLevel (RTL)**. Verilog supports all of these levels. However, this handout focuses on only the portions of Verilog which support the RTL level.

What is Verilog?

Verilog is one of the two major Hardware Description Languages (HDL) used by hardware.

Verilog was introduced in 1985 by Gateway Design System Corporation, now a part ofCadence Design Systems, Inc.'s Systems Division. Until May, 1990, with the formation ofOpen Verilog International (OVI), Verilog HDL was a proprietary language of Cadence.Cadence was motivated to open the language to the Public Domain with the expectation thatthe market for Verilog HDL-related software products would grow more rapidly with broaderacceptance of the language. Cadence realized that Verilog HDL users wanted other softwareand service companies to embrace the language and develop Verilog-supported design tools.

Verilog HDL allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioural level as well as the lower implementation levels (i. e. , gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDLs is the simulation of designs before the designer must commit to fabrication. This handout does not cover all of Verilog HDL but focuses on the use of Verilog HDL at the architectural or behavioural levels. The handout emphasizes design at the Register Transfer Level (RTL).

Use Verilog HDL?

Digital systems are highly complex. At their most detailed level, they may consists of millions of elements, i. e., transistors or logic gates. Therefore, for large digital systems, gate-level design is dead. For many decades, logic schematics served as the *lingua franca* of logic design, but not anymore. Today, hardware complexity has grown to such a degree that a schematic with logic gates is almost useless as it shows only a web of connectivity and not thefunctionality of design. Since the 1970s, Computer engineers and electrical engineers have moved toward hardware description languages (HDLs).

The most prominent modern HDLs in industry are Verilog and VHDL. Verilog is the top HDL used by over 10,000 designers at such hardware vendors as Sun Microsystems, AppleComputer and Motorola. Industrial designers like Verilog. It works.

The Verilog language provides the digital designer with a means of describing a digital systemat a wide range of levels of abstraction, and, at the same time, provides access to computer- aided design tools to aid in the design process at these levels.

Verilog allows hardware designers to express their design with **behavioral constructs**,deterring the details of implementation to a later stage of design in the design. An abstract representation helps the designer explore architectural alternatives through **simulations** and todetect design bottlenecks before detailed design begins.

Though the behavioral level of Verilog is a high level description of a digital system, itis still a precise notation. Computer-aided-design tools, i. e., programs, exist which will“compile” programs in the Verilog notation to the level of circuits consisting of logic gates andflip flops. One could then go to the lab and wire up the logical circuits and have a functioningsystem. And, other tools can “compile” programs in Verilog notation to a description of the integrated circuit masks for **very large scale integration** (VLSI). Therefore, with the properautomated tools, one can create a VLSI description of a design in Verilog and send the VLSIdescription via electronic mail to a **silicon foundry** in California and receive the integratedchip in a few weeks by way of snail mail.

Verilog also allows the designer to specific designsat the logical gate level using **gate constructs** and the transistor level using **switch constructs**.Our goal in the course is not to create VLSI chips but to use Verilog to precisely describe the *functionality* of *any* digital system, for example, a computer. However, a VLSIchip designed by way of Verilog’s behavioural constructs will be rather slow and be wastefulof chip area. The lower levels in

Verilog allow engineers to optimize the logical circuits and VLSI layouts to maximize speed and minimize area of the VLSI chip.

About Verilog Language

There is no attempt in this handout to describe the complete Verilog language. It describes only the portions of the language needed to allow students to explore the architecturalaspects of computers. In fact, this handout covers only a small fraction of the language. For thecomplete description of the Verilog HDL, consult the references at the end of the handout. Webegin our study of the Verilog language by looking at a simple Verilog program. Looking at the assignment statements, we notice that the language is very C-like. Comments have a C++ flavor, i.e., they are shown by "//" to the end of the line. The Verilog language describes a digital system as a set of **modules**, but here we have only a single module called "simple".

Program Structure

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules to describe how they are interconnected. Usually weplace one module per file but that is not a requirement. The modules may run concurrently, butusually we have one top level module which specifies a closed system containing both test dataand hardware models. The top level module invokes instances of other modules.

Modules can represent pieces of hardware ranging from simple gates to complete systems, e. g., a microprocessor. Modules can either be specified behaviorally or structurally (or a combination of the two). A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs, e. g., **if**s, assignment statements. A **structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. At the bottom of the hierarchy the componentsmust be primitives or specified behaviorally. Verilog primitives include gates, e. g., nand, as well as pass transistors (switches).

The structure of a module is the following:

```
module <module name> (<port list>);
        <declares>
        <module items>
endmodule
```

The **<module name>** is an identifier that uniquely names the module. The **<port list>** is a list of input, inout and output ports which are used to connect to other modules. The **<module items>** may be **initial** constructs, **always** constructs, continuous assignments or instances of modules.

SimulationSteps:

1. Start The Xilunx Project Navigator By Using The Desktop Shortcut or By using theStart →Programs →Xilinx ISE (14.7).
2. Create a New project Select File menu and Then Select New Project.
3. Specify the project Name and Location in pop up Window and click next.
4. To Create New **Verilog** file Right click on the device name and Select NEW SOURCE.Select **Verilog** module in New Source Wizard and Give Suitable name for the Project Click NEXT for the Define Module Window.
5. Write Behavioral Verilog code in Verilog Editor.

Session 1

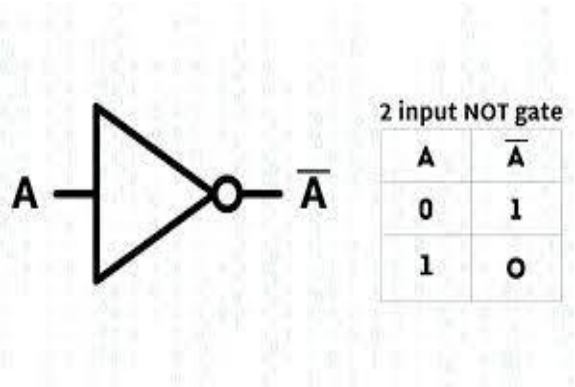
a. Write the Pin diagram, logic symbol and truth table of basic gates, 2 i/p and 3 i/p NAND gates.

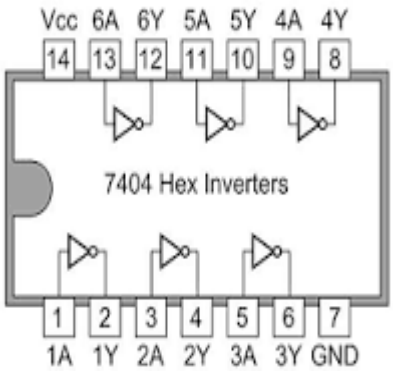
b. Write the Pin diagrams of 8:1 Mux (74151), JK flip-flop IC (7476).

c. Simulate the simple electronic circuit using PSpice along with procedure.

NOT GATE (7404):

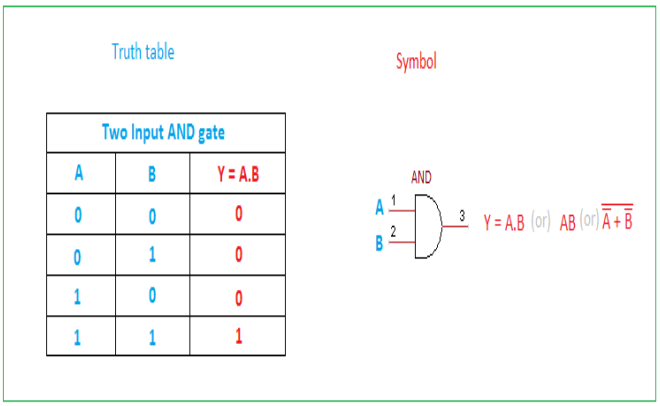
The **NOT gate** is a single input single output gate. This gate is also known as Inverter because it performs the inversion of the applied binary signal, i.e., it converts 0 into 1 or 1 into 0.



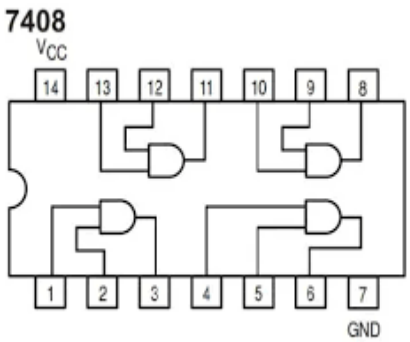


AND GATE (7408):

An AND gate is a digital logic gate with two or more inputs and one output that performs logical conjunction. The output of an AND gate is true only when all of the inputs are true. If one or more of an AND gate's inputs are false, then the output of the AND gate is false.



Two Input AND gate		
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

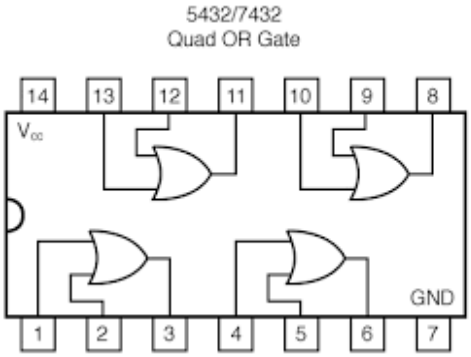


OR GATE (7432):

Any OR gate can be constructed with two or more inputs. It outputs a 1 if any of these inputs are 1, or outputs a 0 only if all inputs are 0.

2 input OR gate

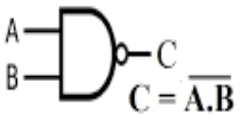
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1



NAND (7400):

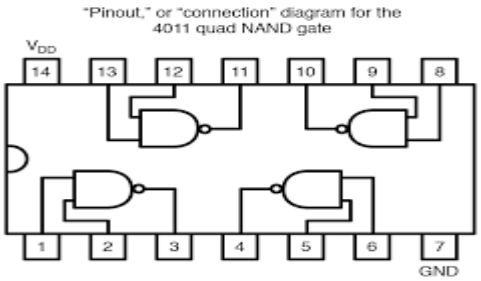
NAND gate is a Boolean operator which gives the value zero if and only if all the operands have a value of one, and otherwise has a value of one (equivalent to NOT AND)

NAND GATE

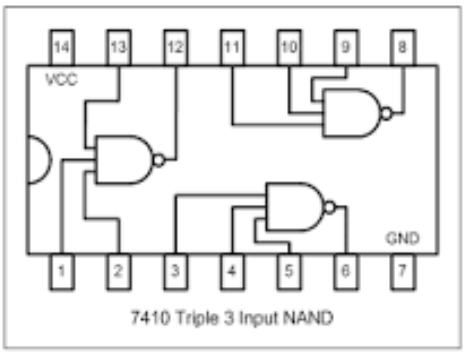


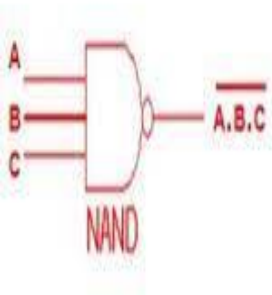
Truth Table

INPUT		OUTPUT
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0



NAND (7410) 3- input :



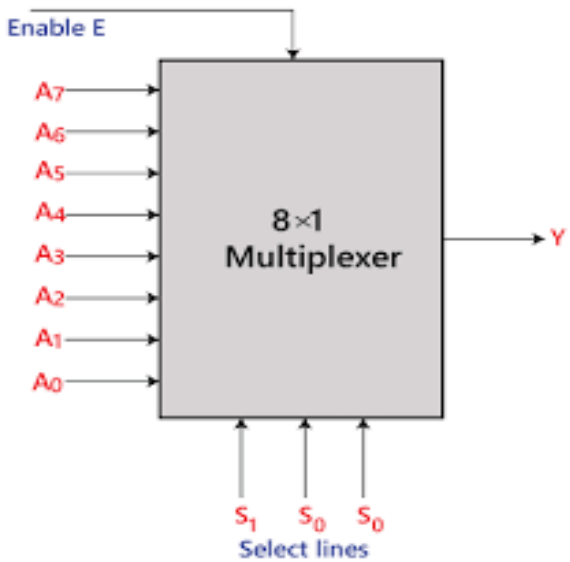


3 input NAND gate

A	B	C	A.B.C	$\overline{A.B.C}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

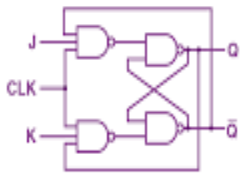
8:1MUX (74151):

In 8-to-1 multiplexer consists of eight data inputs D0 through D7, three input select lines S0 through S2 and a single output line Y. Depending on the select lines combinations, multiplexer selects the inputs.



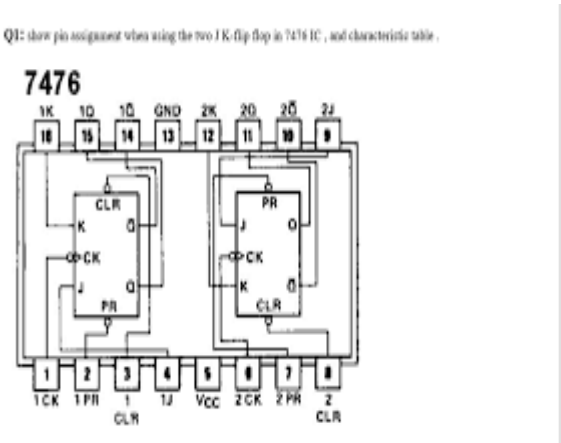
J Flip-flop (7476):

It is one kind of sequential logic circuit which stores binary information in bitwise manner. It consists of two inputs and two outputs. Inputs are Set(J) & Reset(K) and their corresponding outputs are Q and Q'.



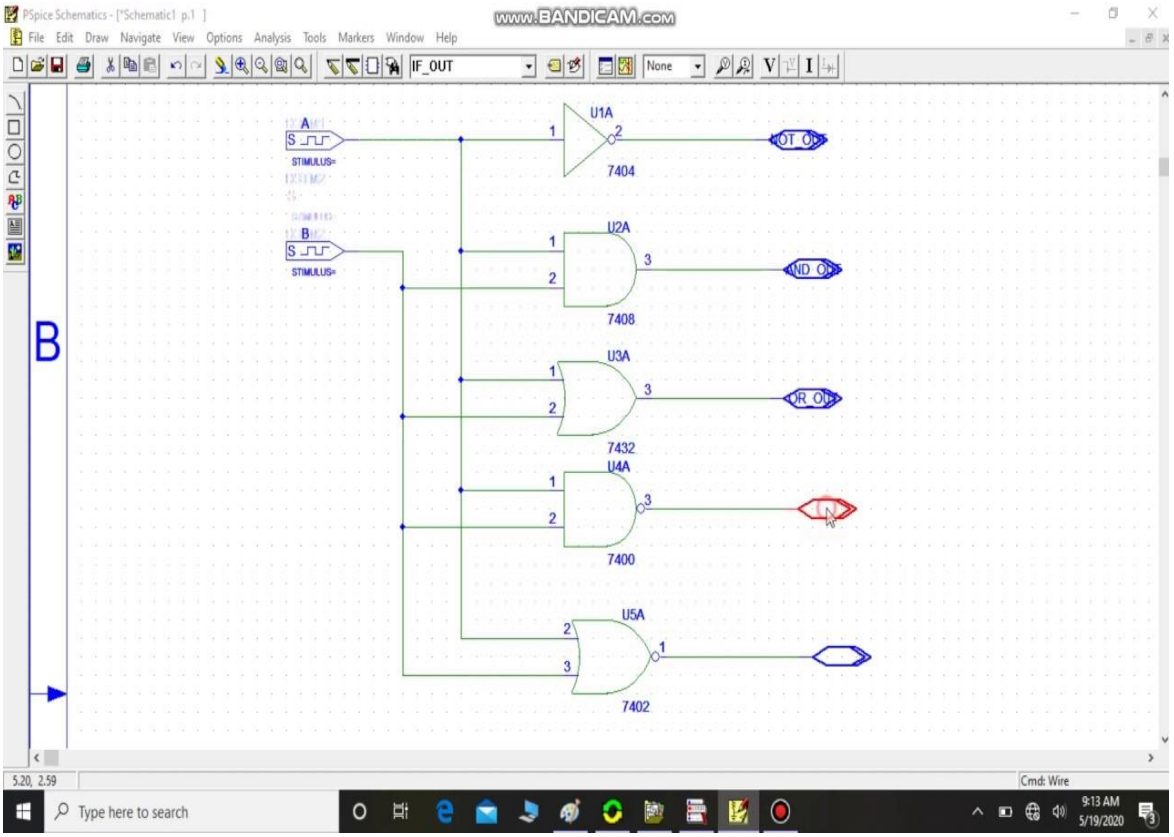
Truth Table

J	K	Q_N	Q_{N+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



c. Working Steps of PSpice for simple circuit/gates:

Circuit:


PSpice Schematics - [*Schematic1 p.1]
File Edit Draw Navigate View Options Analysis Tools Markers Window Help
IF_OUT
None
A
STIMULUS=
B
STIMULUS=
U1A
7404
U2A
7408
U3A
7432
U4A
7400
U5A
7402
NOT_OUT
AND_OUT
OR_OUT
5.20, 2.59
Cmd: Wire
Type here to search
9:13 AM
5/19/2020

Session 2

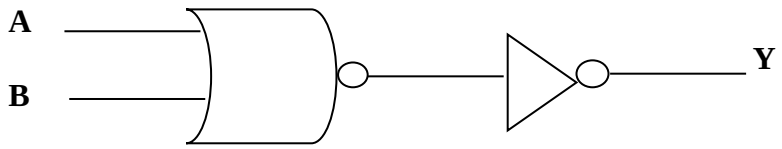
Realize the following expressions and simulate the same, document both hardware and software with the procedure. Simulated with a Verilog code.

Expression:

i. $Y=\overline{A}B + B^{\overline{\ }}$

ii. C=A or (not B)

iii. Write the Verilog code for the circuit given below.



<u>Write the Verilog code for all expression with test bench</u>

Program 1

Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.

Aim:

$$Y = \sum m(1,2,5,6,7,8,9,10,11)$$

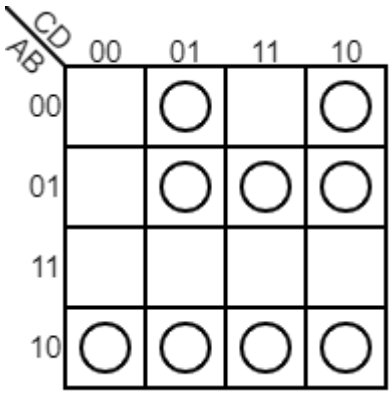
Theory

The given expression in minterm and don't care condition.Simplify the expression using K-Map method:

Truth Table

A	B	C	D	Output
0	0	0	0	F
0	0	0	1	T
0	0	1	0	T
0	0	1	1	F
0	1	0	0	F
0	1	0	1	T
0	1	1	0	T
0	1	1	1	T
1	0	0	0	T
1	0	0	1	T
1	0	1	0	T
1	0	1	1	T
1	1	0	0	F
1	1	0	1	F
1	1	1	0	F
1	1	1	1	F

K-Map:

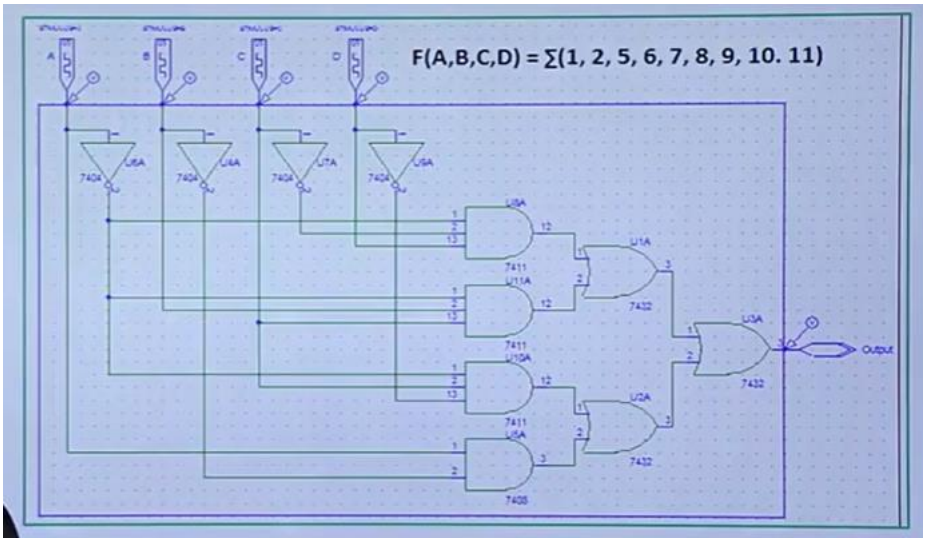


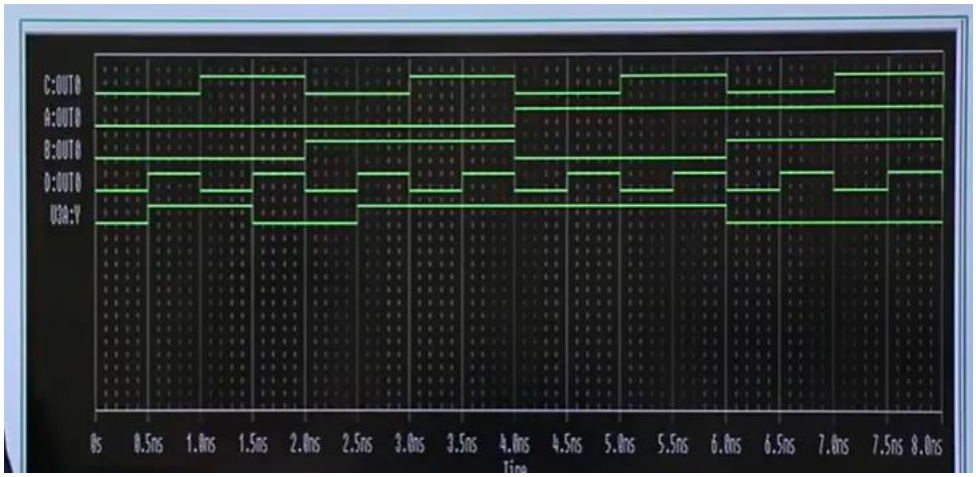
Simplied Boolean expression using K Map $\bar{A}\bar{C}\,D + \bar{A}\,BC + \bar{A}\,C\,\bar{D} + A\bar{B}$

Application:

- There are several applications of K-map, from logic simplification to interpretable AI, where engineers predict the logical relationship between the variables, and it helps detect errors.
- ML engineers can reduce the number of nodes in their neural network by simplifying the logical expression.

Pspice Ckt

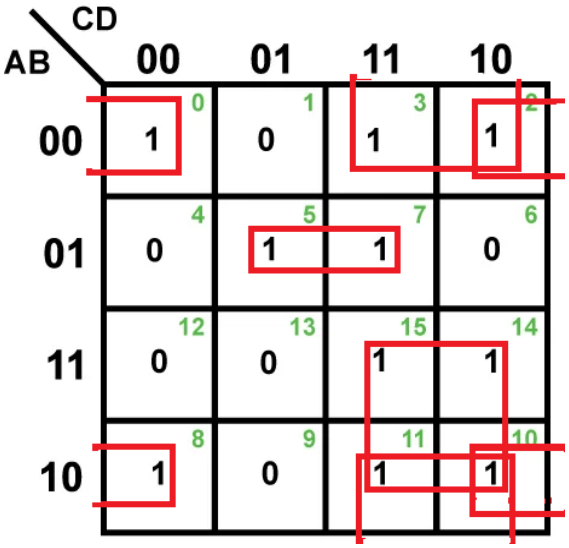




Example 2: Simplify the following Boolean function using four-variable maps:

AB'C + B'C'D' + BCD + ACD' + A'B'C + A'BC'D

K - map



Simplified function: F = AC + B'C + B'D' + A'BD

Program:

```
module 4variable(input A, B, C, D, ouput F);
        wire o1, o2, o3, o4;
        assign o1 = A & C;
        assign o2 = !B & C;
        assign o3 = !B & !D;
        assign o4 = !A & B & D;
        assign F = o1 | o2 | o3 | o4;
endmodule
```

Program 2

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

Aim

To Add/ subtractor 4 bit binary number A_3, A_2, A_1 A_0 and B_3, B_2, B_1, B_0, using full adder.

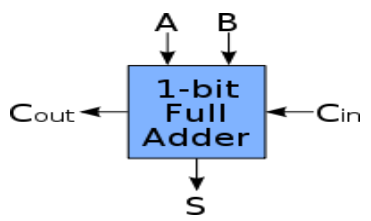
Components Required

Sl.no	Name of Component	No.'s
1	Xor gate	8
2	And	8
3	OR	4

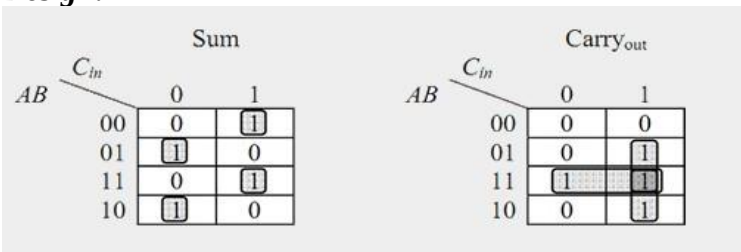
Full Adder

Truth Table

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



Design:



K-Map

From the K – Map:
S = A'B'Cin + A'BCin' + AB'Cin' + ABCin
S = Cin'(A'B + AB') + Cin(A'B' + AB)
S = Cin'(A Θ B) + Cin(A Θ B)'
S = Cin Θ (A Θ B)

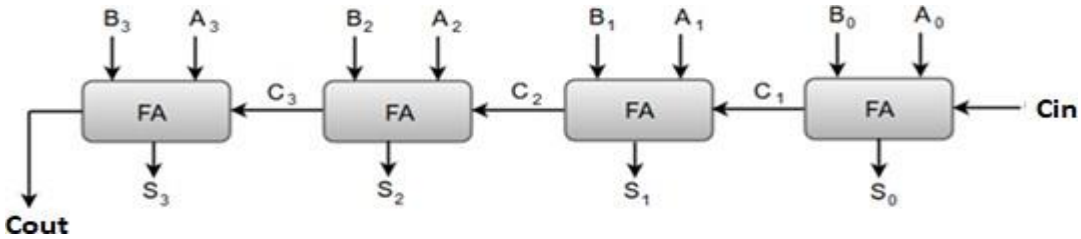
C = AB + BCin + ACin

Theory

For 4 bit Binary adder:

$Si = Ai \oplus Bi \oplus Cin$
$Ci+1 = (Ai \cdot Bi) + C_{in} \cdot (Ai \oplus Bi)$

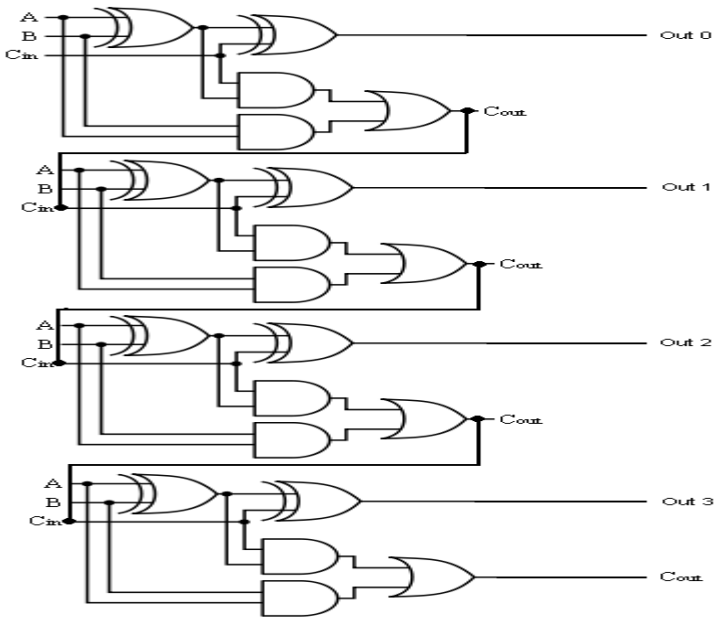
Design



Applications:

Full adders are used in calculators. Full adders also help in carrying out multiplication of binary numbers. Full adders are also used to realize critic digital circuits like multiplexers.

Simulation circuit(pspice/verilog):

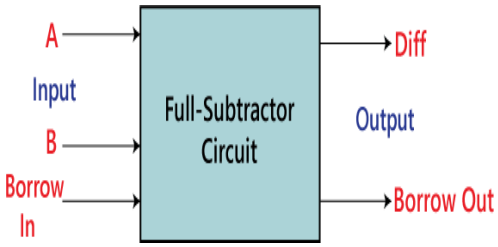


Input and output Full adder

A3 A2 A1 A0	B3 B2 B1 B0	Cin	S3 S2 S1 S0	Cout
1 0 1 1	0 1 0 1	0	0 0 0 0	1
0 1 1 0	1 0 1 1	0	0 0 0 1	1
1 0 1 1	0 1 1 1	0	0 0 1 0	1
1 1 0 1	0 0 1 0	0	1 1 1 1	0

Full Subtractor

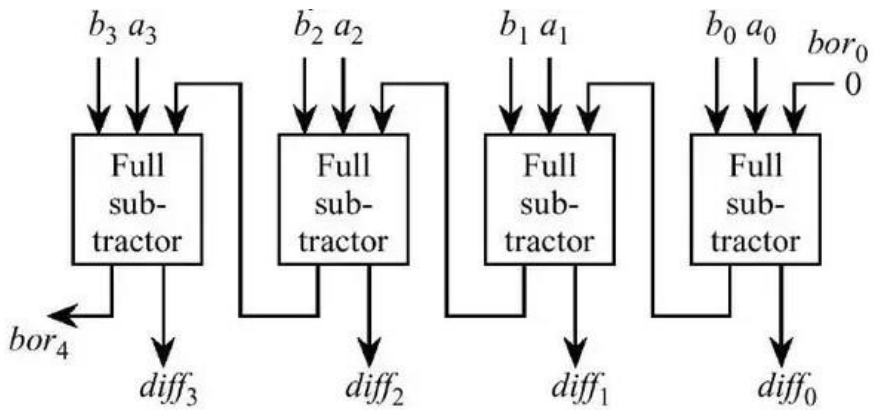
Theory



Design (Truth Table):

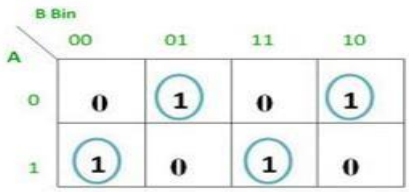
Inputs			Outputs	
A	B	$Borrow_{in}$	Diff	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

4 bit adder and subtractor



k-map simplification

For Difference



D = A'B'Bin + A'BBin' + AB'Bin' + ABBin

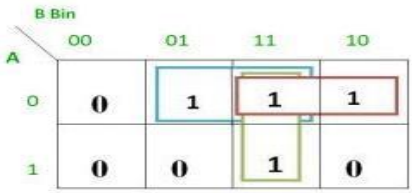
D = Bin(A'B' + AB) + Bin'(AB' + A'B)

D = Bin(A ⊙ B) + Bin'(A ⊙ B)

D = Bin (A ⊙ B)' + Bin'(A ⊙ B)

D = Bin ⊙ (A ⊙ B)

For Borrow,



Bout = A'B + A'Bin + BBin

Input and Output

A3 A2 A1 A0	B3 B2 B1 B0	Bin	D3 D2 D1 D0	Bout
1 1 0 0	0 1 0 1	0	0 1 1 1	0
1 1 1 0	1 0 1 1	0	0 0 1 1	0
0 0 1 1	0 1 0 1	0	1 1 1 0	1
0 1 0 1	0 1 1 0	0	1 1 1 1	1

Program 3

Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same in HDL simulator.

Aim

To implement simple circuits (like mux) using Structural, Data flow and Behavioral model.

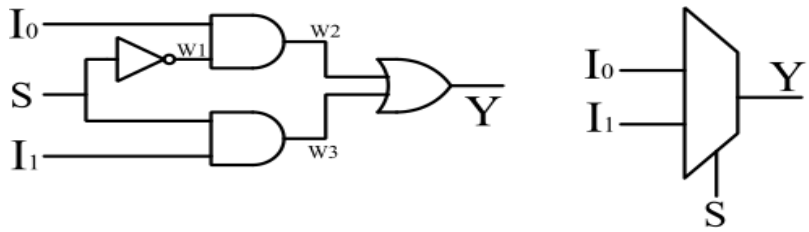
Structural Model:

Let us describe a 2 x 1 MUX using structural modeling.

Boolean expression

$$Y = S'I_0 + SI_1$$

Circuit diagram



Verilog Program

```
module mux2x1( input I0,I1,S,output Y);
   wire w1,w2,w3;not(w1,S);
   and(w2,I0,w1);
   and(w3,I1,S);
   or(Y,w2,w3);
endmodule
```

Test bench

```
        initial begin
                I0=0;
                I1 = 1;
                S=0;

        end
        always #100 I0 = ~I0;
        always #100  I1=~I1
        always #20 s=~s;
endmodule
```

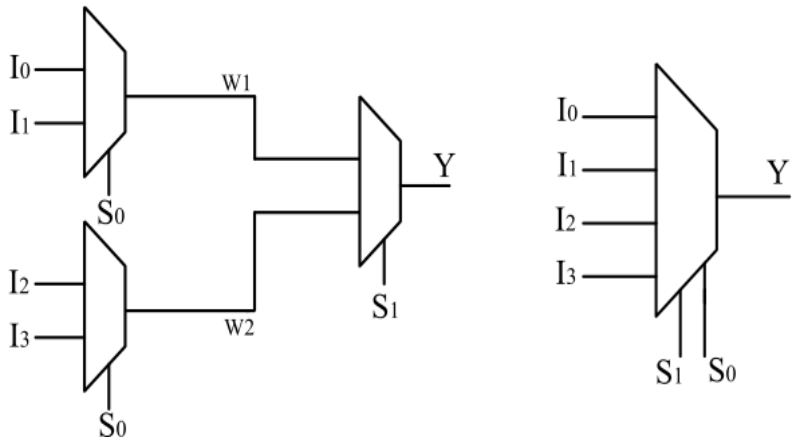
Data Flow Model

Let us describe a 4x1 MUX using data flow modeling. Let's make 4x1 MUX using 2x1 MUX

Boolean expression

$$Y = S1'S0'I0 + S1'S0I1 + S1S0'I2 + S1S0I3$$

Circuit diagram



Verilog code

```
module mux4x1(input I0,I1,I2,I3,S0,S1, output Y);
        wire w1,w2;
        assign w1 = (~S0 & I0) | (S0 & I1);
        assign w2 = (~S0 & I2) | (S0 & I3);
        assign Y = (~S1 & w1) | (S1 & w2);
      endmodule
```

Test bench

```
      initial begin
            I0=0;          I1 = 1;
            I2 = 1;        I3 =0;
            S0=0;   S1=0;

      end
        always #100 I0 = ~I0;
       always #100  I1=~I1
      always #100 I2 = ~I2;
      always #100  I3=~I3
      always #20 S0=~S0;
      always #10  s1 = ~s1
endmodule
```

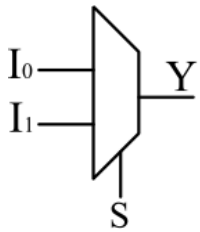
Behavioural Model.

Let us describe a 2x1 MUX using Behavioral modeling.

Boolean expression

i) $Y = S'I_0 + SI_1$

Circuit diagram



Program

```
module mux2x1( input I0, I1, S,output reg Y);
        always @ (*) beginif (S)
            Y = I1;
          else
            Y = I0;
          end
      endmodule
```

Verilog program for Structural Model - simulation:

Test bench

```
      initial begin
            I0=0;
            I1 = 1;
            S=0;

      end
      always #100 I0 = ~I0;
      always #100  I1=~I1
      always #20 s=~s;
endmodule
```

Program 4

Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.

Aim

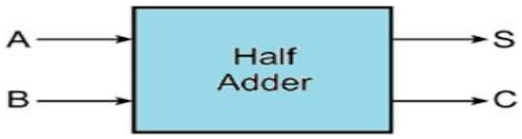
To design and implement Binary Adder-Subtractor, Half and Full Adder, Half and FullSubtractor

1. Half Adder

Theory

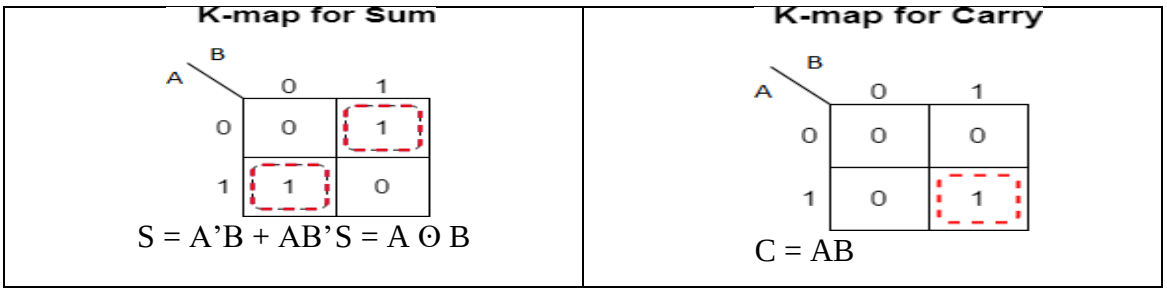
A half adder is a digital logic circuit that performs binary addition of two single-bit binary numbers. It has two inputs, A and B, and two outputs, SUM and CARRY.

Design (Truth Table)

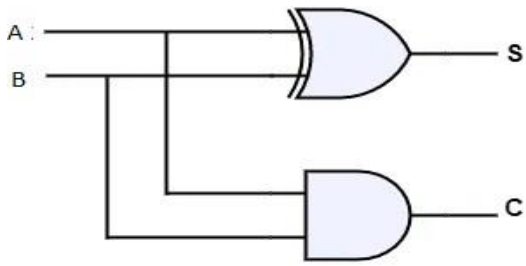


Simplification

INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Circuit



Verilog code for Half Adder

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

Verilog code for Half Adder – simulation- Test Bench

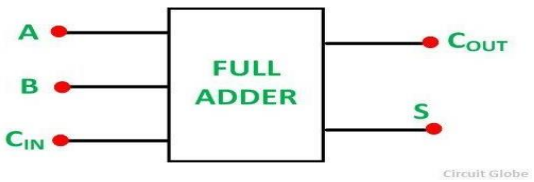
```
initial begin
        #100 a =0 ;        b= 0;
        #100 a = 0;        b = 1;
        #100 a =1 ;        b= 0;
        #100 a = 1;        b = 1;

    end
endmodule
```

2. Full adder

Theory

Full Adder is the adder that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.

Design (truth table)



INPUTS			OUTPUTS	
A	B	C_{in}	SUM	$CARRY_{OUT}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Simplification

For Sum

A \ BC_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

S = A'B'Cin + A'BCin' + AB'Cin' + ABCin

S = Cin(A'B' + AB) + Cin'(A'B + AB')

S = Cin(A Θ B)' + Cin'(A Θ B)

S = Cin Θ (A Θ B)

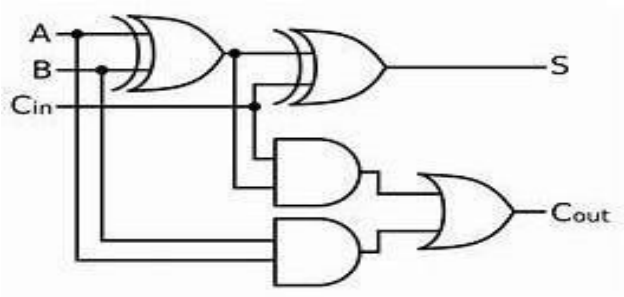
For Carry (C_{out})

A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Cout = AB + BCin + ACin

Circuit

Verilog code for Full Adder




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

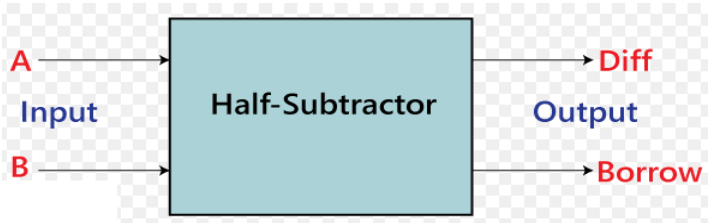
Verilog code for full Adder - simulation

```
initial begin
        a =0 ;
         b= 0;
         c = 0;
end
        always #40 a = a+1'b1;
        always #20 b = b+1'b1;
        always #10 c = c+1'b1;
endmodule
```

3. Half subtractor

Theory

A half subtractor is a digital logic circuit that performs binary subtraction of two single-bit binary numbers. It has two inputs, A and B, and two outputs, DIFFERENCE and BORROW. The DIFFERENCE output is the difference between the two input bits, while the borrow output indicates whether borrowing was necessary during the subtraction.

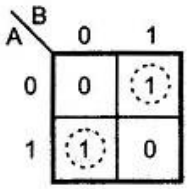


Design (truth table)

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Simplification

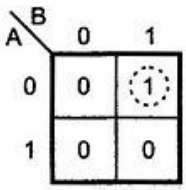
For Difference



D = A'B + AB'

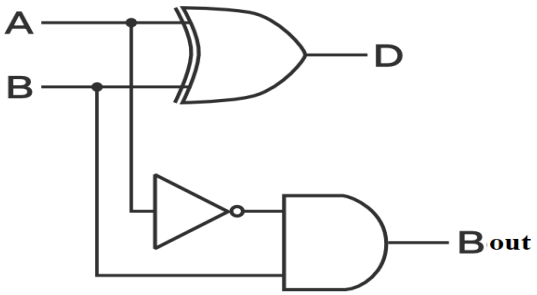
D = A Θ B

For Borrow



Bout = A'B

Circuit



Verilog code for Half Subtractor:

```
module half_subtractor(input a, b,output difference, borrow);
        assign difference = a ^ b;
        assign borrow = ~a & b;
endmodule
```

<u>Verilog code for Half Subtraction- simulation</u>

```
initial begin
        #100 a =0 ;         b= 0;
        #100 a = 0;         b = 1;
        #100 a =1 ;         b= 0;
        #100 a = 1;         b = 1;
    end
endmodule
```

4. Full subtractor

Theory

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. This circuit **has three inputs and two outputs**.

Design (truth table):

A	B	B_{in}	D	B_{out}
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

Simplification

For D

A \ BB_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

D = A'B'Bin + A'BBin' + AB'Bin' + ABBin

D = Bin'(A'B + AB') + Bin(A'B' + AB)

D = Bin'(A Θ B) + Bin(A Θ B)'

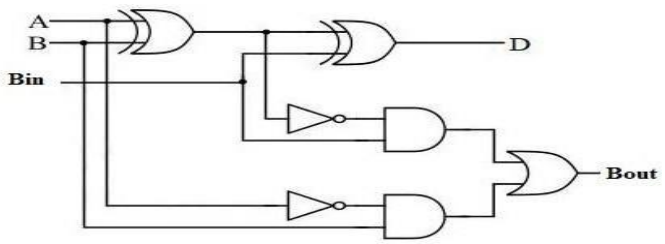
D = Bin Θ (A Θ B)

For B_{out}

A \ BB_{in}	00	01	11	10
0	0	1	1	1
1	0	0	1	0

Bout = A'B + BBin + A'Bin

Circuit:



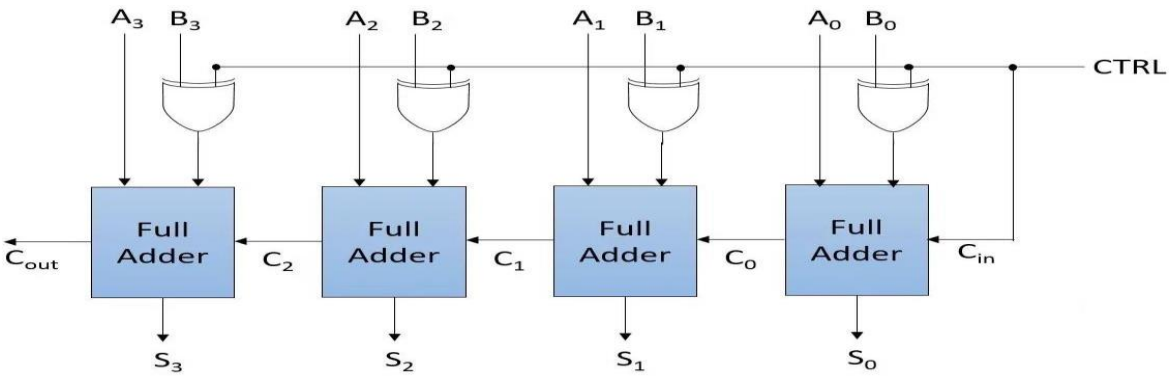
<u>Verilog code for Full_Subtractor</u>

```
module full_subtractor( input a, b, bin,output difference, borrow);
        assign difference = a ^ b ^ bin;
        assign borrow = (~a & b) | (b & bin) | (~a & bin) ;
endmodule
```

Bianry Adder - Subtraction

Theory

In Digital Circuits, A **Binary Adder-Subtractor** is capable of both the addition and subtraction of binary numbers in one circuit itself. The operation is performed depending on the binary value the control signal holds. It is one of the components of the ALU (Arithmetic Logic Unit).



Verilog code for Binary addition and subtraction

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

module tripple_adder_subs(input[3:0] A,B, input ctrl,output [3:0] S, CO,output C);
        full_adder fa0(A[0],B[0]^ctrl , ctrl , S[0] , CO[0]);
        full_adder fa1(A[1],B[1]^ctrl , CO[0] , S[1] , CO[1]);
        full_adder fa2(A[2],B[2]^ctrl , CO[1] , S[2] , CO[2]);
        full_adder fa3(A[3],B[3]^ctrl , CO[2] , S[3] , CO[3]);
        assign C = CO[3];
endmodule
```

Verilog code for Binary Adder -Subtraction- simulation

Test bench

```
        initial begin
                A = 4'b0000;
                B=  4'b0000;
                ctrl = 0;

        end
    always #10 A = A+1'b1;
   always #20 B = B+1'b1;
endmodule
```

Program 5

Design Verilog HDL to implement Decimal adder

Aim:

To design and implement a Decimal adder.

Theory:

BCD stands for binary coded decimal. It is used to perform the addition of BCD numbers. A BCD digit can have any of ten possible four-bit representations. Suppose, we have two 4-bit numbers A and B. The value of A and B can vary from 0(0000 in binary) to 9(1001 in binary) because we are considering decimal numbers.

We are adding A(=7) and B(=8).

The value of binary sum will be 1111(=15).

But, the BCD sum will be 1 0101,

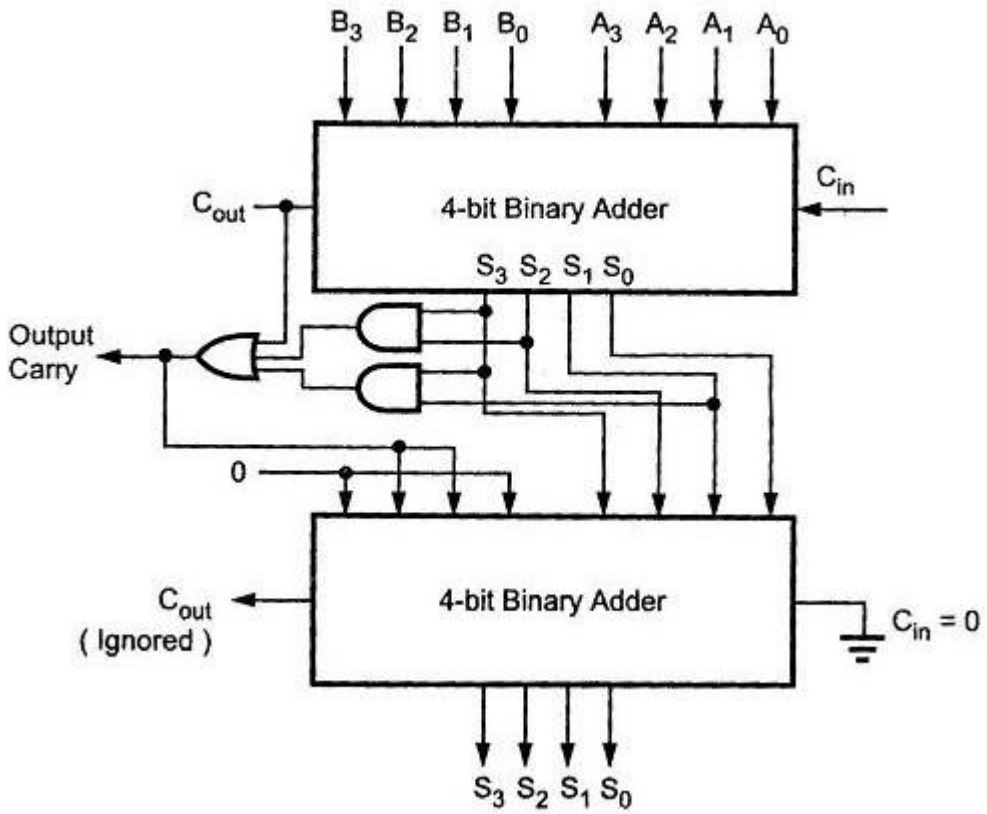
where 1 is 0001 in binary and 5 is 0101 in binary.

Truth Table

Sum bits of adder-1 →

INPUTS				OUTPUT
S3	S2	S1	S0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
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	1	1	1

Circuit:



Verilog Code for Decimal Adder:

```
module bcd_adder ( input [3:0] a, b, input carry_in, output reg [3:0] sum, output reg
carry);
// Internal variables
reg [3:0] sum_temp;

// Always block for doing the addition
always @(a, b, carry_in)
        begin
                sum_temp = a + b + carry_in; // Add all the inputs
                if (sum_temp > 9) begin
                        sum_temp = sum_temp + 6; // Add 6 if the result is more than 9.
                        carry = 1; // Set the carry output
                                end
                else begin
                        carry = 0;
                end
                sum = sum_temp[3:0];
        end

endmodule
```

Verilog code for Decimal Adder – Test bench

```
initial begin
        // Initialize Inputs
        a = 0;
        b = 0;
        carry_in = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

    end
     always #10 b = b+1;
     always #20 a = a+1;
endmodule
```

Program 6

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

Aim

To design and implement 2:1, 4:1 and 8:1 Multiplexers

Theory

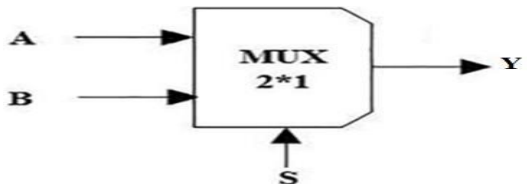
It is a combinational circuit which have many data inputs and single output depending on control or select inputs.

1. Design 2:1 mux using verilog code:

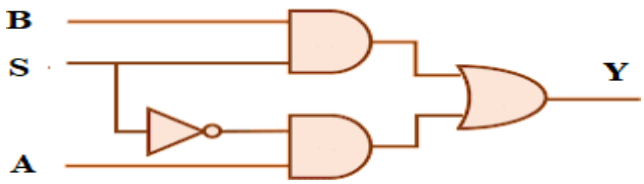
Truth table:

S	Y
0	A
1	A

Block Diagram



Circuit Using Basic Gate



Verilog code:

```
module mux_2to1(input S,input A,input B,output Y);
        assign Y = (S & ~B) | (~S & A);
endmodule
```

Test bench

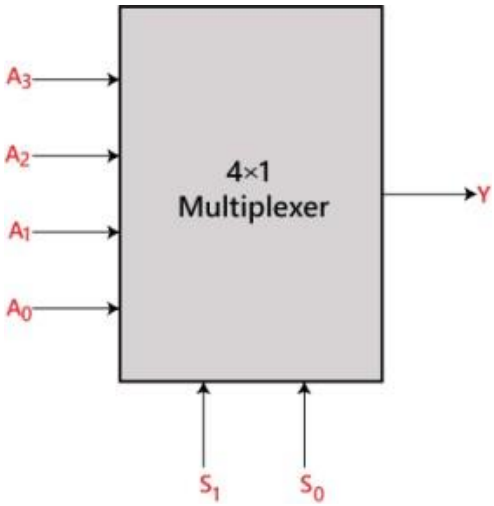
```
initial begin
        I0=0;
        I1 = 1;
        S=0;
                end
always          #100            I0 = ~I0;
always  #100            I1=~I1
always  #20             s =~s;
endmodule
```

2. Design 4:1 Mux Using Verilog Code:

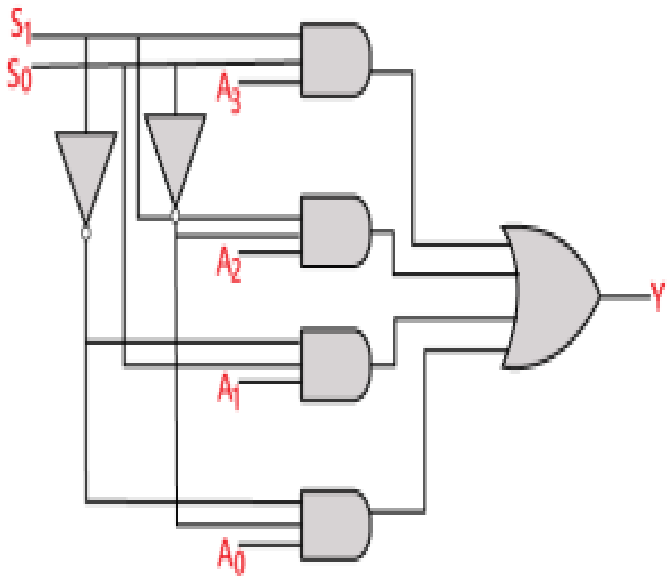
Truth table:

S1	S0	Y
0	0	A0
0	1	A1
1	0	A2
1	1	A3

Block Diagram



Circuit Using Basic Gate



Verilog Code

```
module m41 ( input I3,  I2,  I1,  I0, input s0, s1,output y);
        assign y = s1 ? (s0 ?  I0 :  I1) : (s0 ?  I2 :  I3);
endmodule
```

Test bench

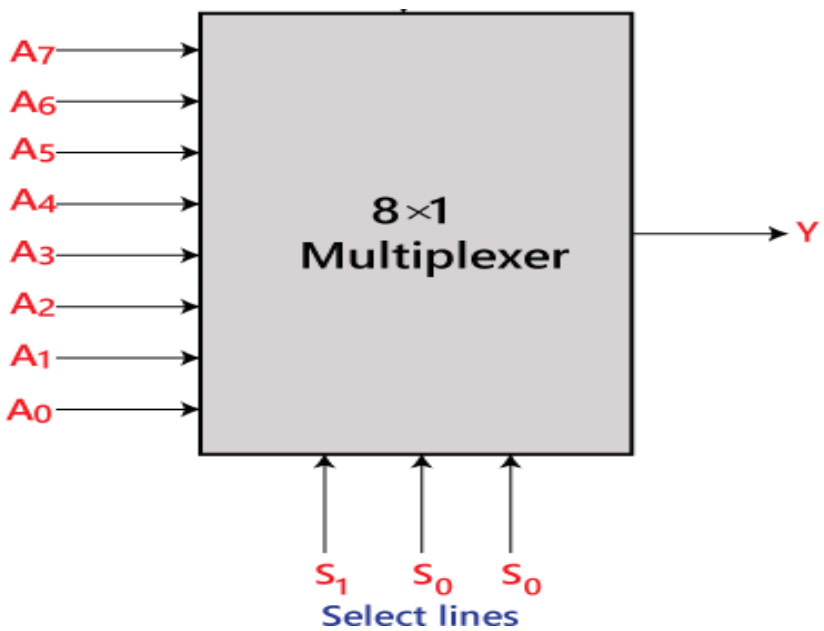
```
        initial begin
                I0=0;           I1 = 1;
                I2 = 1;         I3 =0;
                S0=0;   S1=0;

        end
                always #100 I0 = ~I0;
                always #100  I1=~I1
                always #100 I2 = ~I2;
                always #100  I3=~I3
                always #20 S0=~S0;
                always #10  s1 = ~s1
endmodule
```

3. Design 8:1 mux using verilog code:

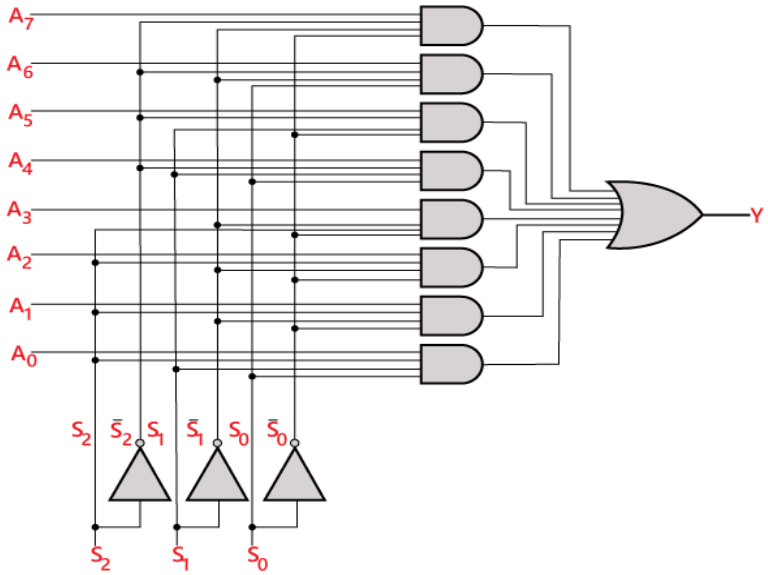
Truth Table:

INPUTS			Output
S_2	S_1	S_0	Y
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

Block diagram



Circuit



Y=S0'.S1'.S2'.A0 + S0.S1'.S2'.A1 + S0'.S1.S2'.A2 + S0.S1.S2'.A3 + S0'.S1'.S2A4 + S0.S1'.S2 A5 + S0'.S1.S2.A6 + S0.S1.S3.A7

Verilog code:

```
module eighttoone (input  [7:0]  I, input [2:0] sel, output reg out);
        always @ (a, b, c, sel)
         begin
                case(sel)
                        3'b000 :        out = I[0];
                        3'b001 :        out = I[1];
                        3'b010 :        out = I[2];
                        3'b011 :        out = I[3];
                        3'b100 :        out = I[4];
                        3'b101 :        out = I[5];
                        3'b110 :        out = I[6];
                        3'b111 :        out = I[7];
                        default         :       out =1'bx;
                endcase
        end
endmodule
```

Test bench

```
        initial begin
                I = 8'b00110111;
                S=3'b000;
        end
        always  #20 s = s+1;
endmodule
```

Program 7

Design Verilog program to implement types of De-Multiplexer.

Aim

To design and implement 1:2, 1:4 and 1:8 De-Multiplexers

Theory

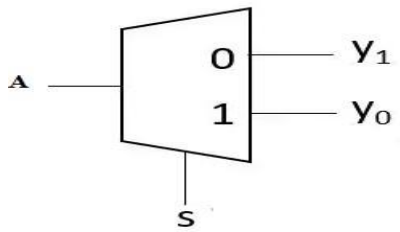
A De-multiplexer is a combinational circuit that has only 1 input line and 2^N output lines. Simply, the multiplexer is a single-input and multi-output combinational circuit. The information is received from the single input lines and directed to the output line. On the basis of the values of the selection lines, the input will be connected to one of these outputs. De-multiplexer is opposite to the multiplexer.

1. Design 1:2 De-Mux Using Verilog Code

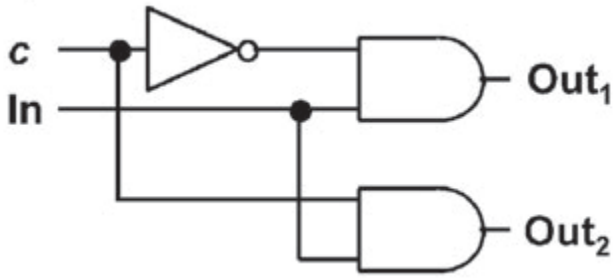
Truth table

INPUTS	Output	
S_0	Y_1	Y_0
0	0	A
1	A	0

Block Diagram :



Circuit Using Basic Gate



$Out_1 = c'.in$
$Out_2 = c.\ in$

Verilog Code:

```
module demux_2_1(input s0, a, output y0, y1);
        assign y0 =~s0 & a;
        assign y1 =s0 & a;
endmodule
```

Test bench

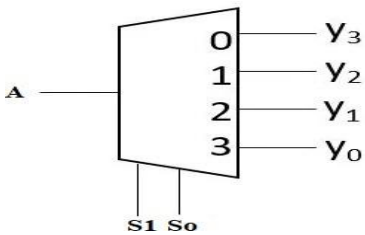
```
initial begin
                s0=0 ; a =1;
   #50          s0=0 ; a =0;
   #50          s0=1 ; a =1;
   #50          s0=1 ; a =0;
end
endmodule
```

2. Design 1:4 de- mux using verilog code

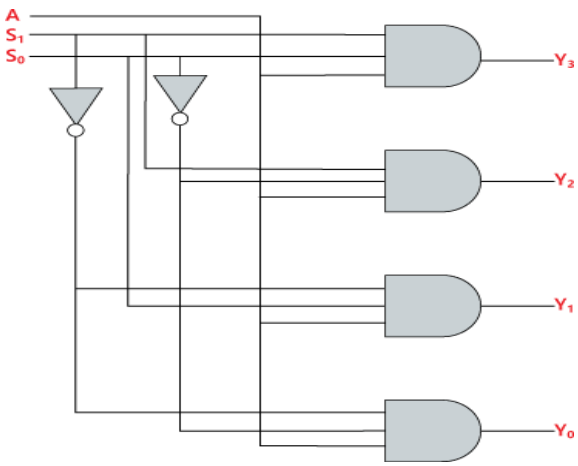
Truth Table

INPUTS		Output			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0

BLOCK DIAGRAM



Circuit



Verilog Code

```
module demux_1_4(input [1:0] sel,input  a,output reg [3:1] y);
 always @(*)
begin
        assign y[0] = ~s[1] & ~s[0] & a;
        assign y[1] = ~s[1] & s[0] & a;
        assign y[2] = s[1] & ~s[0] & a;
        assign y[3] = s[1] & s[0] & a;
end
  endmodule
```

Test bench

```
        initial begin
                a = 0;
                s= 2'b00;
        end
                always #10 s = s+1;
                always #40  a= ~a;
endmodule
```

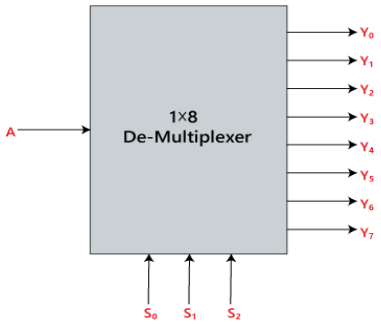
3. Design 1:8 De- Mux Using Verilog Code

Truth Table

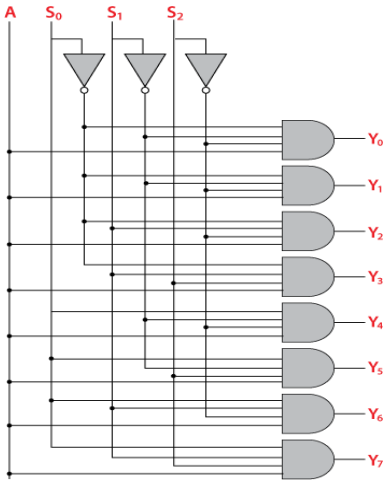
INPUTS			Output							
S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	A
0	0	1	0	0	0	0	0	0	A	0
0	1	0	0	0	0	0	0	A	0	0
0	1	1	0	0	0	0	A	0	0	0
1	0	0	0	0	0	A	0	0	0	0
1	0	1	0	0	A	0	0	0	0	0
1	1	0	0	A	0	0	0	0	0	0
1	1	1	A	0	0	0	0	0	0	0

$Y_0=S_0'.S_1'.S_2'.A$
$Y_1=S_0.S_1'.S_2'.A$
$Y_2=S_0'.S_1.S_2'.A$
$Y_3=S_0.S_1.S_2'.A$
$Y_4=S_0'.S_1'.S_2\ A$
$Y_5=S_0.S_1'.S_2\ A$
$Y_6=S_0'.S_1.S_2\ A$
$Y_7=S_0.S_1.S_3.A$

Block Diagram



Circuit



Verilog Code

```
module Demultiplexer(input a, s0,s1,s2,output reg[7:0] y);
        always @(*)
        begin
                y[0]=(a & ~s2 & ~s1 &~s0);
                y[1]=(a & ~s2 & ~s1 &s0);
                y[2]=(a & ~s2 & s1 &~s0);
                y[3]=(a & ~s2 & s1 &s0;
                y[4]=(a & s2 & ~s1 &~s0);
                y[5]=(a & s2 & ~s1 &s0);
                y[6]=(a & s2 & s1 &~s0);
                y[7]=(a & s2 & s1 &s0);
        end
endmodule
```

Test bench

```
        initial begin
                a= 0;
                s= 3'b000;
        end
                        always          #10     s = s+1;
                        always          #80     in= ~in;
endmodule
```

Program 8

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

Aim

To design and implement SR, JK and D Flip Flops

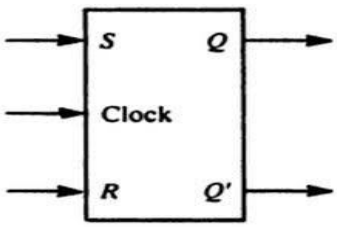
Theory

Flip flop is a term which comes under digital electronics, and it is an electronic component which is used to store one single bit of the information.

Since Flip Flop is a sequential circuit so its input is based upon two parameters, one is the current input and other is the output from previous state.It has two outputs, both are complement of each other. It may be in one of two stable states, either 0 or 1.

SR Flip Flop

It is a Flip Flop with two inputs, one is S and other is R. S here stands for Set and R here stands for Reset. Set basically indicates set the flip flop which means output 1 and reset indicates resetting the flip flop which means output 0. Here clock pulse is supplied to operate this flip flop, hence it is clocked flip flop.



Truth Table for sr Flip Flop :

S	R	Q(n+1)	State
0	0	Qn	No change
0	1	0	RESET
1	0	1	SET
1	1	x	INVALID

Verilog code for SR Flip Flop:

```
module SR_flipflop (input clk, s,r, output reg q,qb);
        q <= 0;
        q<=1;
        always@(posedge clk)
        begin
                case({s,r})
                        2'b00: q <= q;  qb<=qb;         // No change
                        2'b01: q <= 0; qb<= 1;// reset
                        2'b10: q <= 1'b1; qb<=0;// set
                        2'b11: q <= 1'bx;       qb<= 1'bx// invalid
                endcase
        end
endmodule
```

Test bench

```
        initial begin
        clk = 0;
                s= 0;           r=0;
            #100        s= 0;           r=1;
            #100        s= 1;           r=0;
            #100        s= 1;           r=1;
        end
                always #10 clk  =  ~clk;
endmodule
```

Applications

Register: SR Flip Flop used to create register. Designer can create any size of register by combining SR Flip Flops.

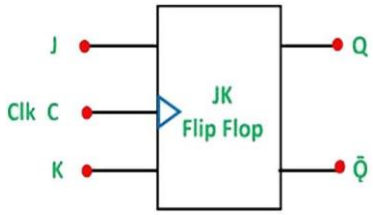
Counters: SR Flip Flops used in counters. Counters counts the number of events that occurs in a digital system.

Memory: SR Flip Flops used to create memory which are used to store data, when the power is turned off.

Synchronous System: SR Flip Flop are used in synchronous system which are used to synchronise the operation of different component.

JK Flip flop

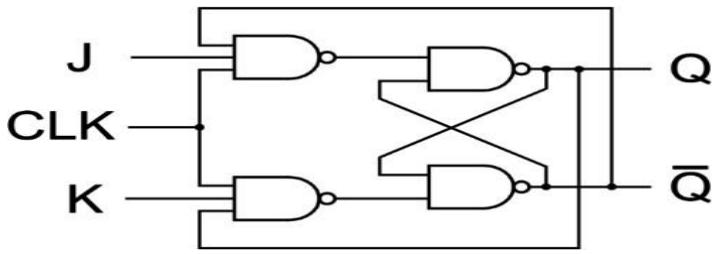
Due to the undefined state in the SR flip-flops, another flip-flop is required in electronics. The JK flip-flop is an improvement on the SR flip-flop where S=R=1 is not a problem.



Truth Table For Jk Flip Flop:

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

Circuit



Verilog code for JK Flip Flop:

```
module jk_flipflop (input clk, j,k, output reg q,qb);
q <= 0;
q<=1;
always@(posedge clk)
begin // for synchronous reset
                case({j,k})
                2'b00: q <= q;  qb<=qb;         // No change
                2'b01: q <= 0; qb<= 1;// reset
                2'b10: q <= 1'b1; qb<=0;// set
                2'b11: q <= 1’bx;       qb<= 1’bx// toggle
        endcase
        end
 endmodule
```

Test bench

```
        initial begin
        clk = 0;
                        j= 0;           k=0;
            #100        j= 0;           k=1;
            #100        j= 1;           k=0;
            #100        j= 1;           k=1;
        end
                always #10 clk  =  ~clk;
endmodule
```

Application Memory Devices of JK flip flop

Memory Devices, Counters, Shift Registers and Control Systems:

Control systems : They are used in control systems for controlling the sequence of operation.

Advantages:

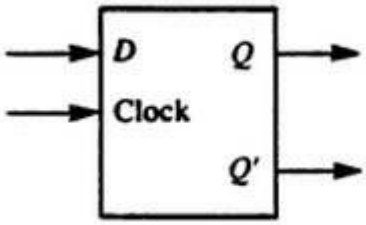
- Due to their toggling feature, JK Flip-Flops can change states without requiring any specific change in the input signals.
- They are versatile and can perform all the operations of SR and D Flip-Flops.

Disadvantages:

- The main drawback of JK Flip-Flops is the occurrence of the race around condition, which happens when the output repeatedly toggles between states within one clock cycle.
- They are more complex and have a higher gate count than other types of flip-flops, leading to increased power consumption.

D Flip Flop

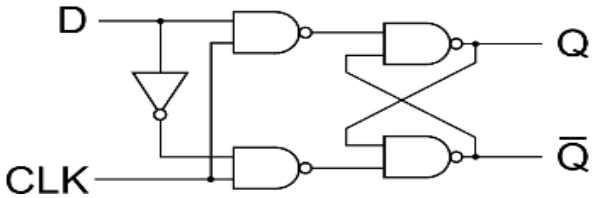
Flip flop is an electronic devices that is known as “delay flip flop” or “data flip flop” which is used to store single bit of data



Truth Table for D Flip Flop:

Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

Circuit



Verilog code for D Flip Flop:

```
module d_flipflop (input clk, d output reg q, qb);
        q <= 0;
        q<=1;
        always@(posedge clk)
                begin // for synchronous reset
                        q<=d ;
                         q<=~d;
                end
        endmodule
```

Test bench

```
        initial begin
                        clk = 0;
                        D= 0;
                #20     D = 1
        end
                always #10 clk  =  ~clk;
endmodule
```

Advantages

Single input: The D flip-flop has a single data input, which makes it simpler to use and easier to interface with other digital circuits.

No feedback loop: The D flip-flop does not have a feedback loop, which eliminates the possibility of a race condition and makes it more stable than other types of flip-flops.

No invalid states: It does not have any weak states, which helps to avoid unpredictable behavior in digital systems.

Reduced power consumption: The D flip-flop consumes less power than other types of flip-flops, making it more energy-efficient.

Bi-stable operation: Like other flip-flops, the D flip-flop has a bi-stable operation, which means that it can hold a state indefinitely until it is changed by an input signal.