

Digital Design and Computer Organization		Semester	3
Course Code	BCS302	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 20 Hours of Practicals	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	Theory		

Sl No.	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 outcome (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.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are 25 marks and that for the practical component is 25 marks.
- 25 marks for the theory component are split into 15 marks for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and 10 marks for other assessment methods mentioned in 22OB4.2. The first test shall be held at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for 25 marks).
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

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.

CONTENT

Exp.No.	Experiment Title	Page No.
	Digital Electronics	1-11
1.	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.	12-13
2.	Design a 4 bit full adder and subtractor and simulate the same using basic gates.	14-15
3.	Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioral model.	16-17
4.	Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.	18-21
5.	Design Verilog HDL to implement Decimal adder.	22
6.	Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.	23-25
7.	Design Verilog program to implement types of De-Multiplexer.	26-27
8.	Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.	28-30

DIGITAL ELECTRONICS

Digital electronics, digital technology or digital (electronic) circuits are electronics that operate on digital signals. In contrast, analog circuits manipulate analog signals whose performance is more subject to manufacturing tolerance, signal attenuation and noise. Digital techniques are helpful because it is a lot easier to get an electronic device to switch into one of a number of known states than to accurately reproduce a continuous range of values.


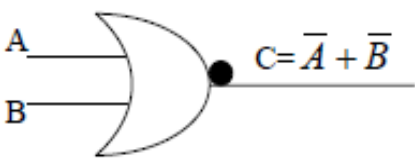
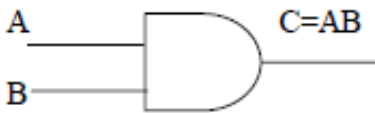
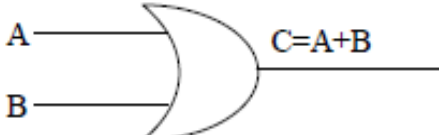
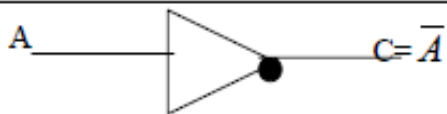
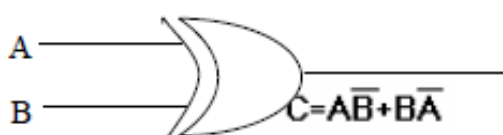
Digital electronic circuits are usually made from large assemblies of logic gates (often printed on integrated circuits), simple electronic representations of Boolean logic functions.

An advantage of digital circuits when compared to analog circuits is that signals represented digitally can be transmitted without degradation caused by noise. For example, a continuous audio signal transmitted as a sequence of 1s and 0s, can be reconstructed without error, provided the noise picked up in transmission is not enough to prevent identification of the 1s and 0s. In a digital system, a more precise representation of a signal can be obtained by using more binary digits to represent it. While this requires more digital circuits to process the signals, each digit is handled by the same kind of hardware, resulting in an easily scalable system. In an analog system, additional resolution requires fundamental improvements in the linearity and noise characteristics of each step of the signal chain.

With computer-controlled digital systems, new functions to be added through software revision and no hardware changes. Often this can be done outside of the factory by updating the product's software. So, the product's design errors can be corrected after the product is in a customer's hands. Information storage can be easier in digital systems than in analog ones. The noise immunity of digital systems permits data to be stored and retrieved without degradation. In an analog system, noise from aging and wear degrade the information stored. In a digital system, as long as the total noise is below a certain level, the information can be recovered perfectly. Even when more significant noise is present, the use of redundancy permits the recovery of the original data provided too many errors do not occur.

In some cases, digital circuits use more energy than analog circuits to accomplish the same tasks, thus producing more heat which increases the complexity of the circuits such as the inclusion of heat sinks. In portable or battery-powered systems this can limit use of digital systems. For example, battery powered cellular telephones often use a low-power analog front-end to amplify and tune in the radio signals from the base station. However, a base station has grid power and can use power hungry, but very flexible software radios. Such base stations can be easily reprogrammed to process the signals used in new cellular standards. In some systems, if a single piece of digital data is lost or misinterpreted, the meaning of large blocks of related data can completely change.

LOGIC GATES

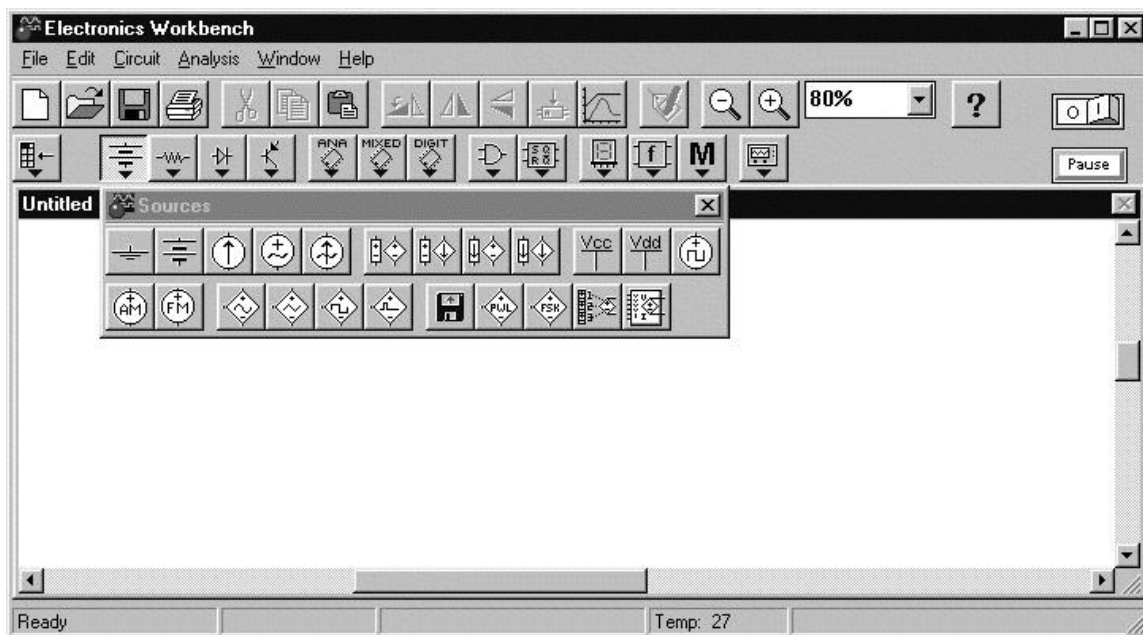
S.NO	GATE	SYMBOL	INPUTS		OUTPUT
			A	B	C
1.	NAND IC 7400	 $C = \overline{A \cdot B}$	0	0	1
			0	1	1
			1	0	1
			1	1	0
2.	NOR IC 7402	 $C = \overline{A + B}$	0	0	1
			0	1	0
			1	0	0
			1	1	0
3.	AND IC 7408	 $C = A \cdot B$	0	0	0
			0	1	0
			1	0	0
			1	1	1
4.	OR IC 7432	 $C = A + B$	0	0	0
			0	1	1
			1	0	1
			1	1	1
5.	NOT IC 7404	 $C = \overline{A}$	1	-	0
			0	-	1
6.	EX-OR IC 7486	 $C = A\overline{B} + \overline{A}B$	0	0	0
			0	1	1
			1	0	1
			1	1	0

The basic logic gates are the building blocks of more complex logic circuits. These logic gates perform the basic Boolean functions, such as AND, OR, NAND, NOR, Inversion, Exclusive-OR, Exclusive-NOR. Fig. below shows the circuit symbol, Boolean function, and truth. It is seen from the Fig that each gate has one or two binary inputs, A and B, and one binary output, C. The small circle on the output of the circuit symbols designates the logic complement. The AND, OR, NAND, and NOR gates can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

Introduction to Electronics Workbench

Electronics Workbench is an electronics and digital logic lab inside a computer, modeled after a real electronics workbench. It is a design tool that provides you with components & instruments to create “virtual” board-level designs:

- No actual breadboards, components, or instruments needed.
- Click-and-drag schematic editing.
- It offers mixed analog & digital simulation and graphical waveform analysis.
- Circuit behavior simulated realistically.
- Results displayed on multimeter, oscilloscope, bode plotter, logic analyzer, etc.



The main GUI interface of EWB

Using Electronics Workbench for Design

You may use EWB to:

- Explore ideas and test preliminary circuits.
- Refine circuits to full layout (If circuit requires parts of a previous design)
- Export files in format used by PCB (Printed Circuit Board) layout packages as move from design to production.

General EWB Functions

Selecting

- To move a component or instrument need to select it selected item highlights: components red, wires thicken
- Clicking to Select
To select single item, click on it.
To select additional items, press CTRL+ click.
- Selecting All
Choose Edit/Select All.
- Dragging to Select
Place pointer above & to side of group of items. Press & hold mouse button & drag downward diagonally. Release mouse button when rectangle encloses everything desired.
- Deselecting
To deselect single item, press CTRL+click.
To deselect all selected items, click on empty spot in window.

Setting Labels, Wiring

Setting Labels, Values, Models & Reference IDs,

- To set labels, values (for simple components) & models (for complex components), select component and choose Circuit/Component Properties, choose desired tab, make any changes, and click OK.
- Can also invoke Circuit/Component Properties box by double-clicking on component.

** Notes:*

The Circuit/Component Properties box contains a number of tabs; depending on which component is selected an analog component has either a value or a model, not both.

Wiring Components

Point to a component's terminal so it highlights; press & hold mouse button, and drag so a wire appears drag wire to a terminal on another component or to an instrument connection, when terminal on second component or instrument highlights, release mouse button

Inserting, Connecting, Editing

Inserting Components

To insert component into existing circuit, place it on top of wire; it will automatically be inserted if there is room.

Connecting Wires

- If drag a wire from a component's terminal to another wire, a **connector** is automatically created when you release mouse button.
- Note: a connector button also appears in the Basic toolbar (to insert connectors into an existing circuit).

Deleting Wires

- To delete a wire, select it & choose Edit/Delete
- Alternatively, disconnect wire by selecting one end of it & moving it to an open spot on circuit window.

Changing Wire Color

- To change a wire's color, double-click it & choose Schematic Options tab; click the Color button & choose a new color.

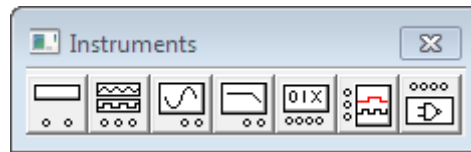
Straightening a Wire

- Move wire itself.
- Move component to which wire is attached.
- Press ALT and move component to which wire is attached.
- Select component and press appropriate arrow key to align it.
- If two wires cross in a way that makes them hard to follow, select one & drag it to new location

**Note:*

- The way a wire is routed sometimes depends on terminal from which wire was dragged; try disconnecting routed wire & then rewire from the opposite terminal.

Instruments



- Using an Instrument Icon

To display the Instruments toolbar, click the Instruments button on the Parts Bin toolbar.

To place an instrument on the circuit window, drag the desired button from the Instruments toolbar to the window. To attach an instrument to a circuit, point to a terminal on its icon so it highlights and drag a wire to a component. To remove an instrument icon, select it & choose Edit/Delete

- Opening an Instrument

Double-click the instrument's icon to see its controls

- To selection options, click buttons on the controls
- To change values or units, click the up/down arrows.

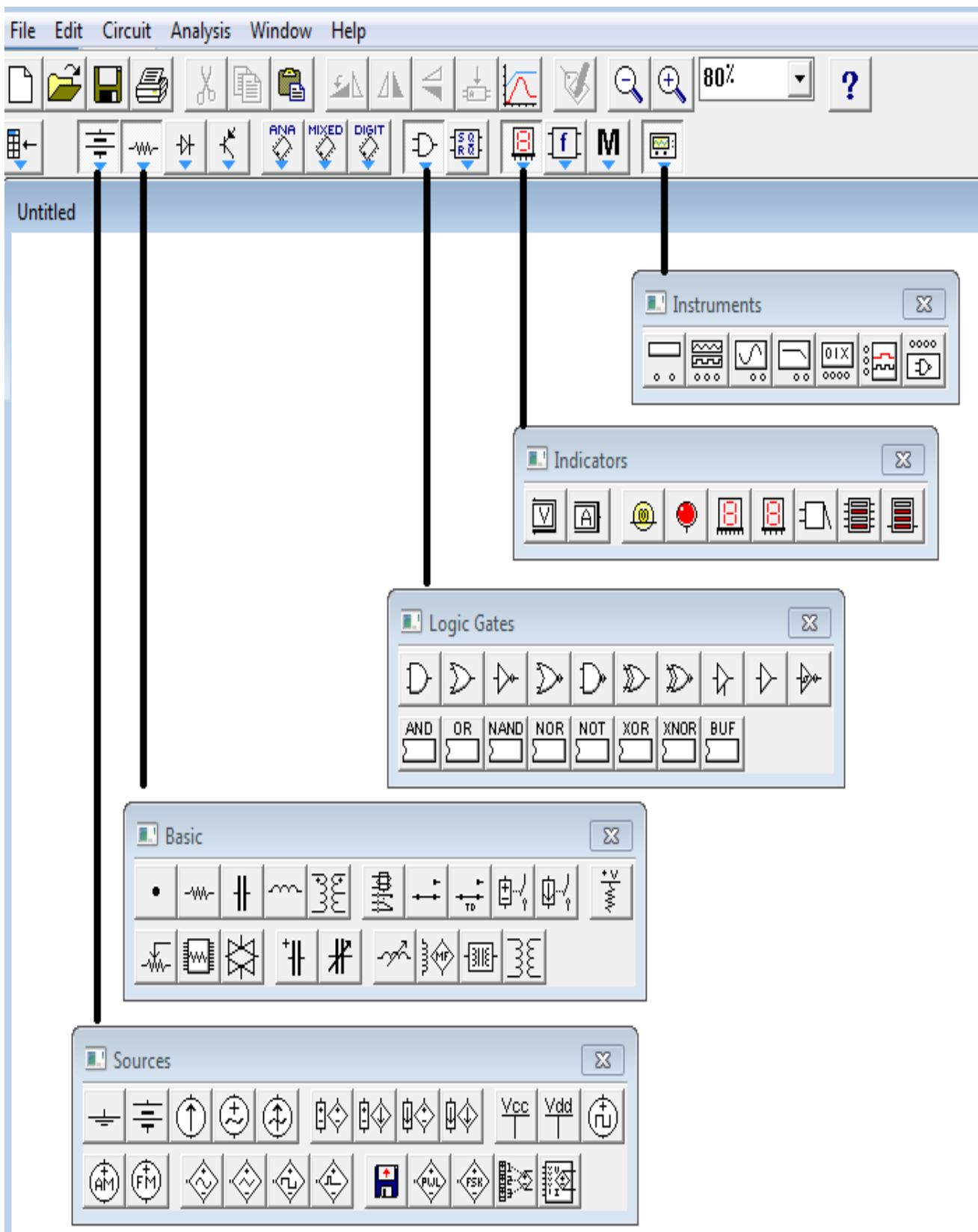
- Turning on Power

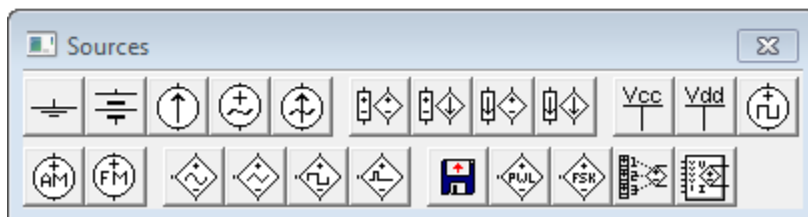
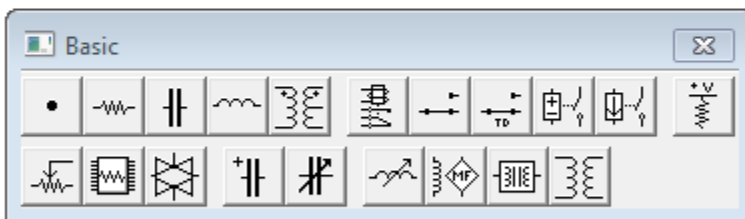
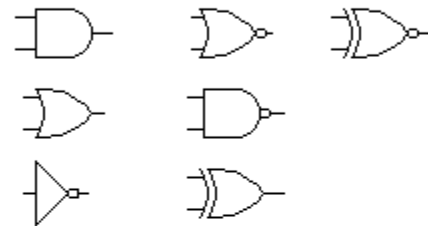
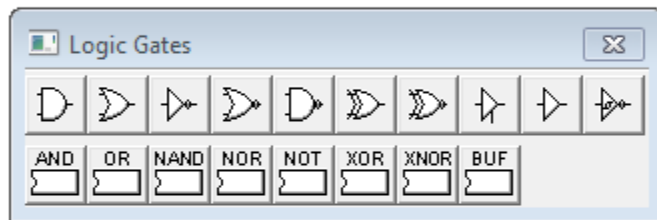
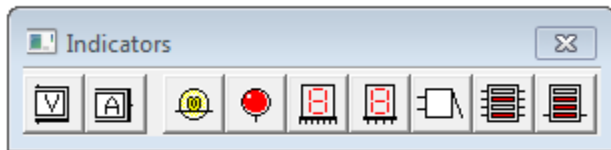
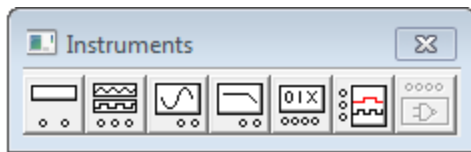
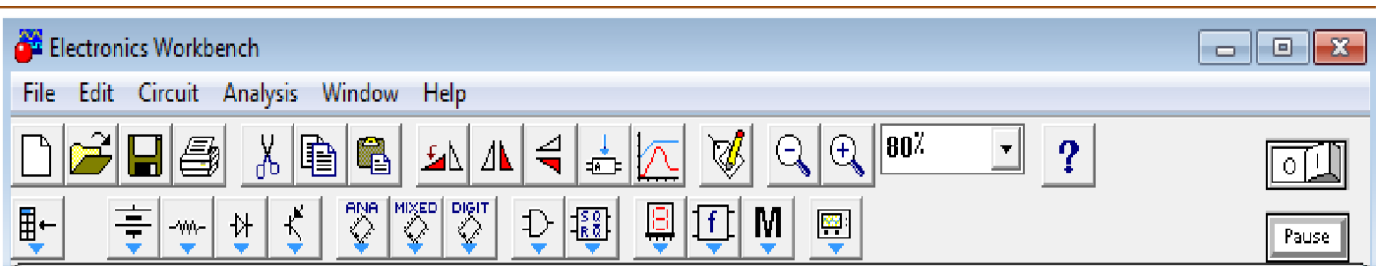


Click the power switch to turn power on. Click switch again to turn power off. (Note: Turning off power erases data & instrument traces.

Lab Tasks

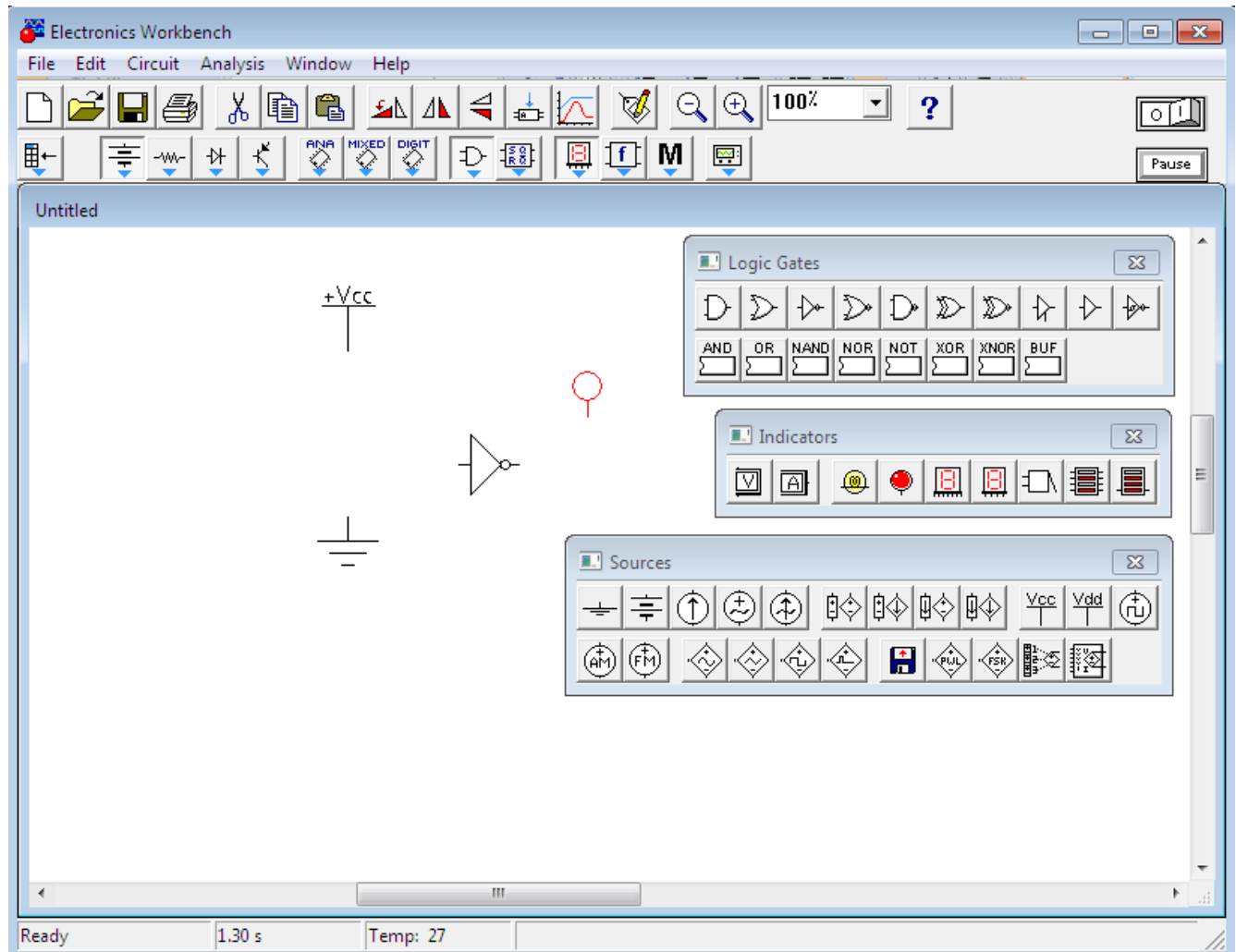
Task 1: Name the basic toolboxes of EWB



Task 2: Basic buttons in EWB toolboxes**Task 3 EWB Toolbar**

Task 4: Simple circuit; playing with EWB

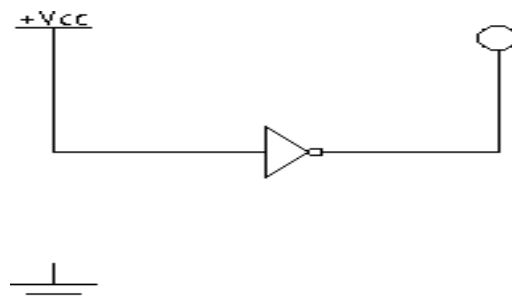
In the following circuit



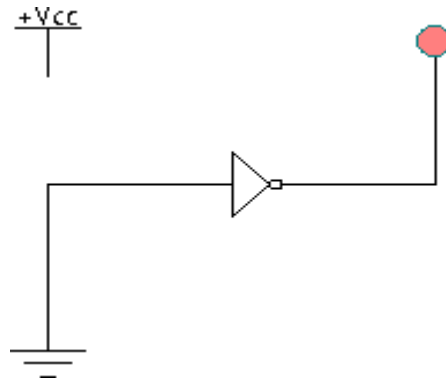
Draw the following circuit. After that make the following changes

- Connect the output of the converter to the red probe
- Connect the Vcc line to the input of the inverter
- Start simulating the circuit
- State your observation down:

Observation:



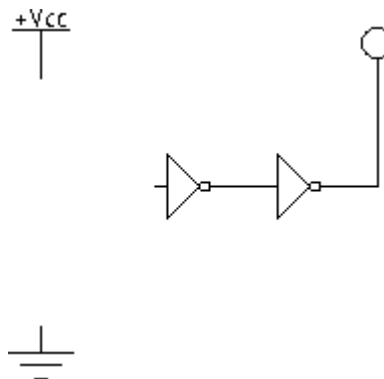
- In the same circuit above, stop the simulation and connect the ground to the input of the inverter. State your observation down:



Observation:

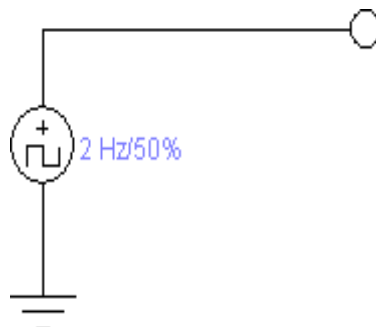
Task 5: Simple circuit; two inverters connected serially

Repeat Task 2 of this report and state down your observations.

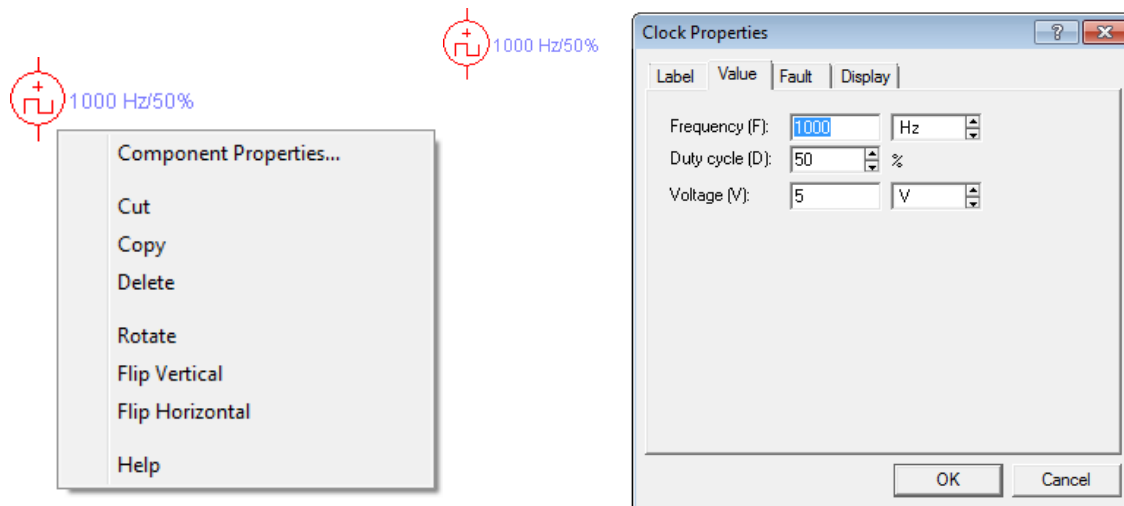


Task 6: Simple circuit; a clock source with a red probe

Draw the following circuit and simulate it. Write down your observations. Notice that the **clock** (from Sources toolbox) frequency is 2 Hz.

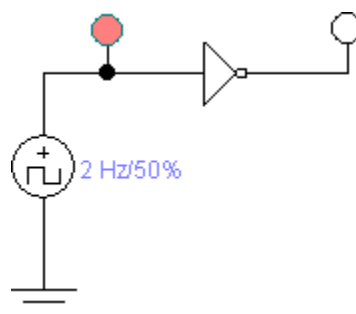


Note: You can change the default values of the clock by doing mouse right clicking on the clock and click on the “Component Properties ...” as shown below:



Task 7: Simple circuit; a clock source with two red probes

Draw the following circuit and simulate it. Write down your observations. Notice that the clock frequency is 2 Hz.



Task 8: EWB Menu

Name the following icons and state down their functions.



Experiment 1

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

Realization of Boolean Expression:

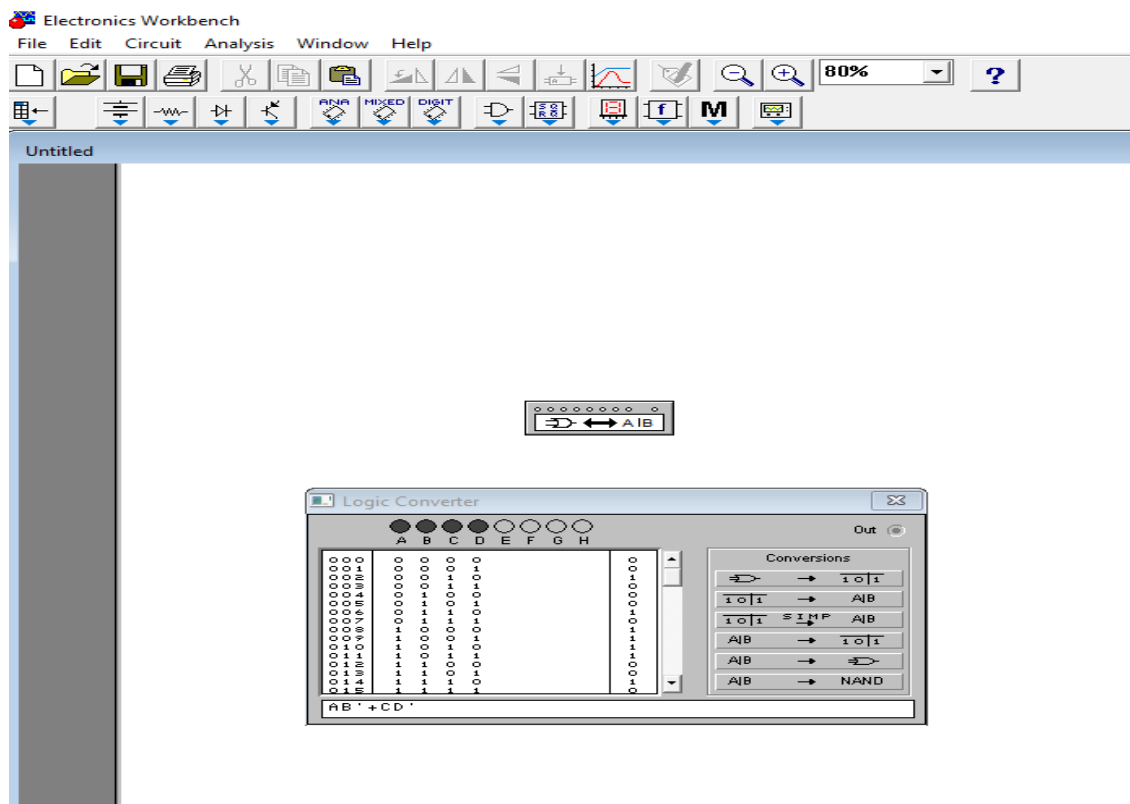
$$1) \quad Y = \bar{A}\bar{B}C\bar{D} + \bar{A}BC\bar{D} + ABC\bar{D} + A\bar{B}C\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}CD$$

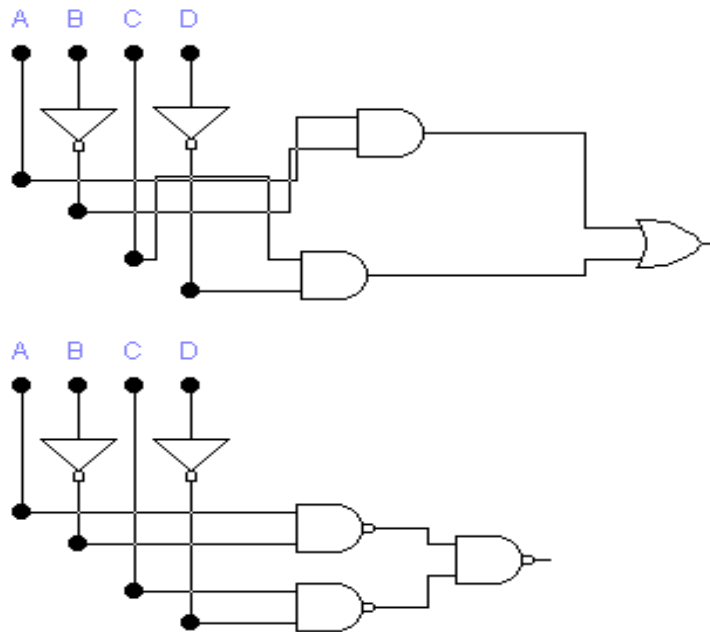
Simplification using K-Map:

	CD	C'D	C'D'	CD'
AB	0	1	3	2
A'B	4	5	7	6
A'B'	12	13	15	14
AB'	8	9	11	10

After Simplifying we get the Boolean expression $Y = A'B + CD'$

Realization using Basic Gates





Circuit Simulation

Verify the given truth table for the realized circuit.

TRUTH TABLE

INPUTS				OUTPUT
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Exercise Questions:

1. Simplify the same Boolean expression using Boolean theorems and verify the truth table.
2. Realize the given Boolean expression using minimum number of basic gates.

$$Y = AB (C+C') + ABCD + A'B'CD + AB (D+1)$$

Experiment 2

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

4-bit binary Addition of two numbers:

Example 1:

$$X = 1010$$

$$Y = 0111$$

$$Z = 11001$$

Carry out

Sum

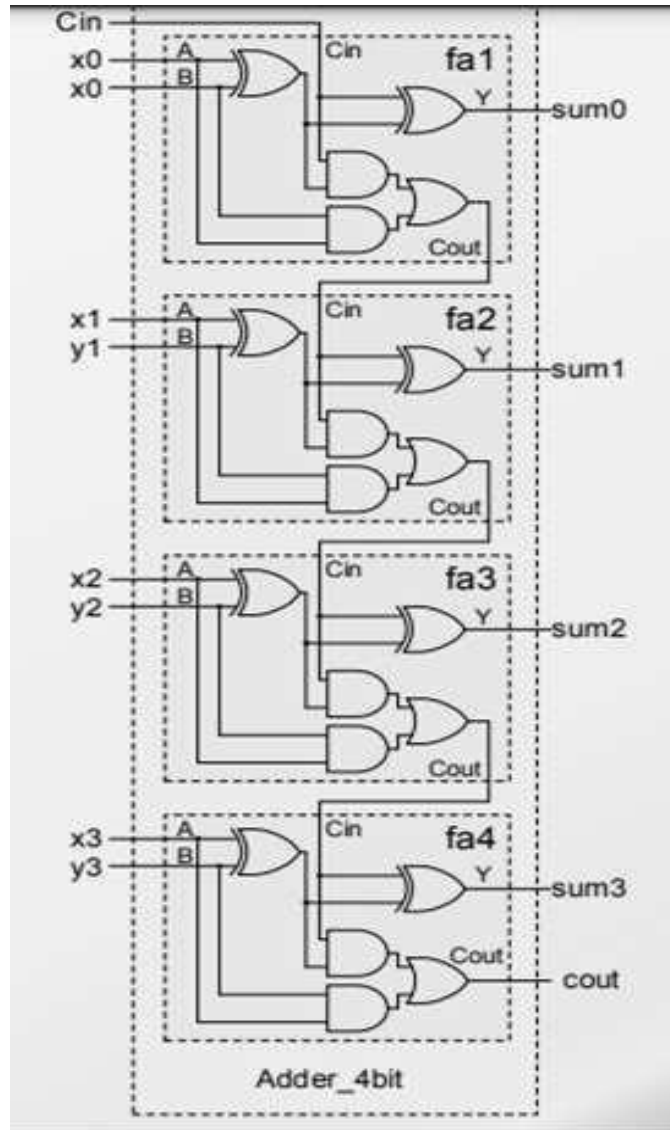
Example 2:

$$X = 0110$$

$$Y = 0100$$

$$Z = 1010$$

Sum



4 bit Adder logic circuit realized using basic gates

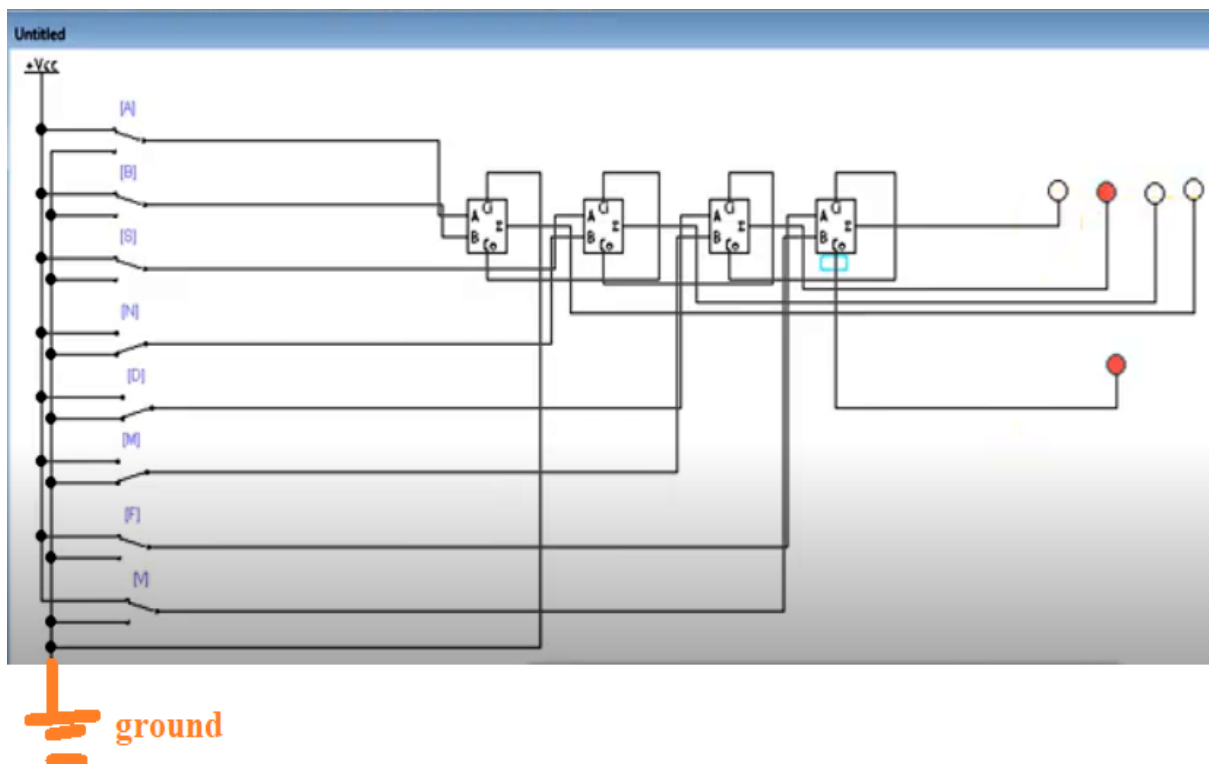


Figure: Logic diagram Implementation on EWB

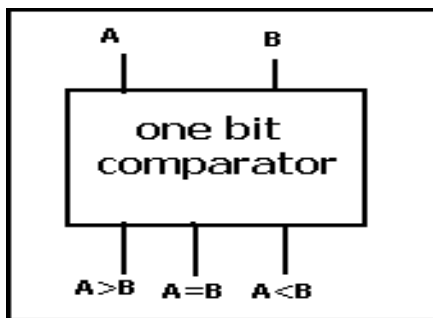
Cin	A				B				Sum				Carry
	A3	A2	A1	A0	B3	B2	B1	B0	S3	S2	S1	S0	Cout
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	1	0	0	1	0	0
0	0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	0	1	1	0	0	1	1	0	1	1	0	0
0	0	1	0	0	0	1	0	0	1	0	0	0	0
0	0	1	0	1	0	1	0	1	1	0	1	0	0
0	0	1	1	0	0	1	1	0	1	1	0	0	0
0	0	1	1	1	0	1	1	1	1	1	1	0	0
0	1	0	0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

Result: 4 bit Full adder and Subtractor is verified with examples

Experiment 3

Aim: Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioral model.

To verify a VHDL / Verilog code for 1-bit Comparator



Logic symbol

INPUT		OUTPUT		
A	B	A > B	A = B	A < B
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

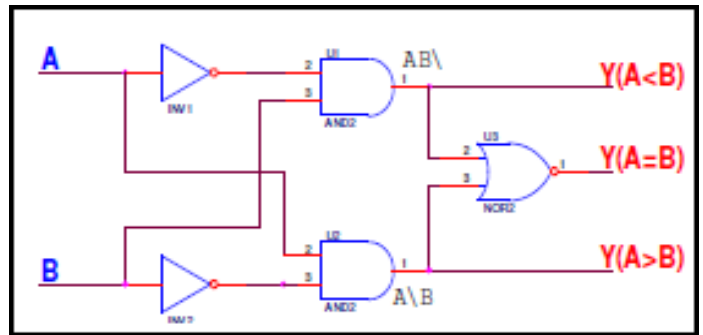
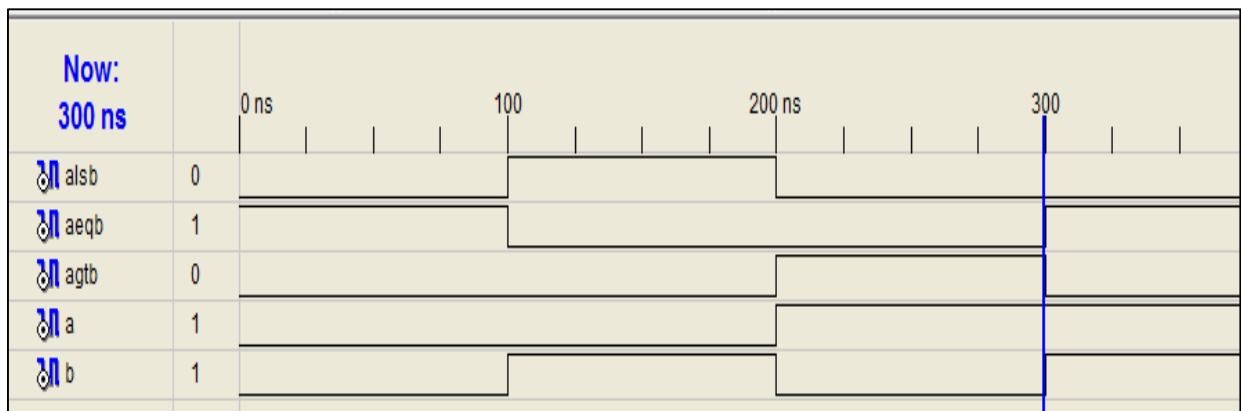
Truth table

VERILOG CODE:

Dataflow Description	Behavioral Description	Structural Description
<pre> module comp (a,b, agtb, aeqb, alsb); input a; input b; output agtb; output aeqb; output alsb; assign agtb = a&(~b); assign alsb = (~a)&b; assign aeqb = ~(a^b); endmodule </pre>	<pre> module comparator (a,b,y); input a; input b; output [2:0]y; reg[2:0]y; wire[1:0]sel; assign sel={a,b}; always @(sel)begin case(sel) 2'd0: y = 3'd2; 2'd1: y = 3'd1; 2'd2: y = 3'd4; 2'd3: y = 3'd2; endcase end endmodule </pre>	<pre> module comparator (a,b,agtb,aeqb,alsb); input a; input b; output agtb; output aeqb; output alsb; wire x,y; not u1(x,a); not u2(y,b); and u3(agtb,a,y); xnor u4(aeqb,a,b); and u5(alsb,x,b); endmodule </pre>

Expressions and Logic Diagram:

Condition	$A > B$	$A = B$	$A < B$
Expression	AB'	$A'B' + AB$	$A'B$

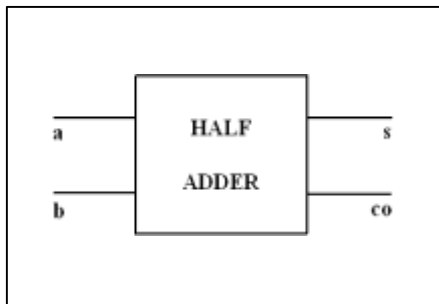
**Simulation waveforms:****Result:**

Experiment 4

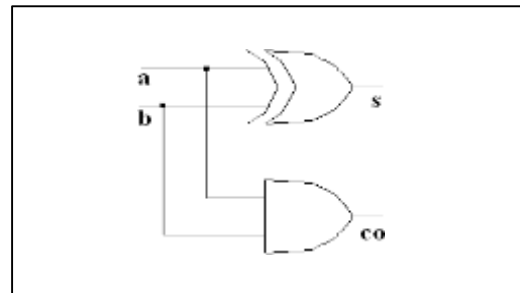
Aim: Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.

4. a HALF ADDER:

Logic symbol



Logic diagram:



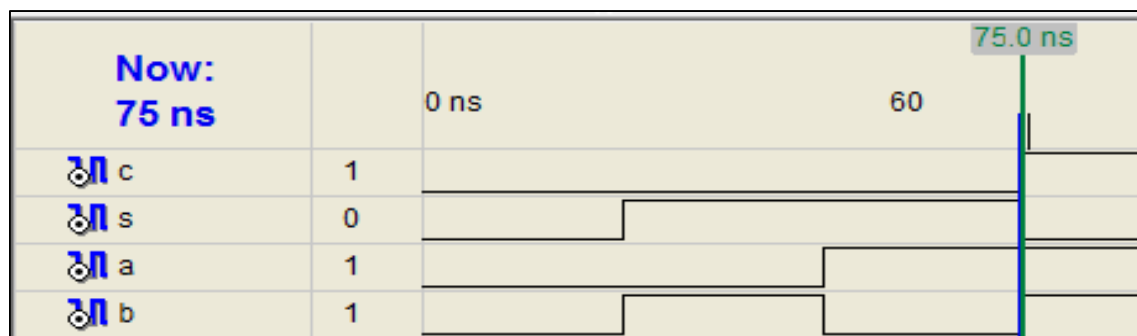
VERILOG CODE

Data Flow Description
<pre> module half (a,b,s,c); input a,b; output s,c; assign s= a ^ b; assign c= a & b; endmodule </pre>

Truth table

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

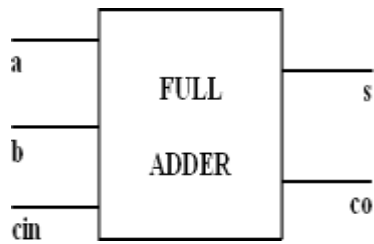
Simulation waveforms:



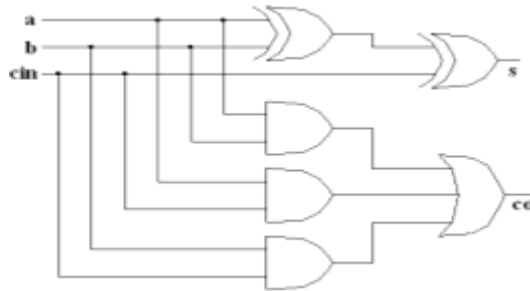
Result:

4.b FULL ADDER

Logic symbol



Logic diagram



VERILOG CODE

Data Flow Description

```

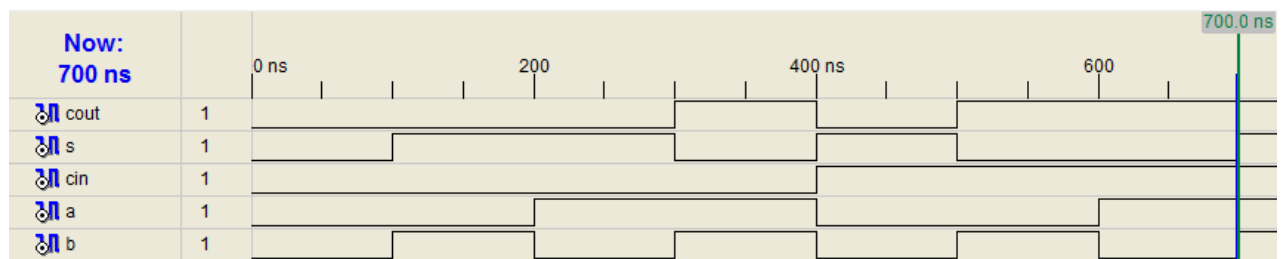
module full_adder (cin,a,b,s,cout);
input cin, a, b;
output s, cout;
reg s, cout;
always @(cin or a or b) begin
s = a ^ b ^ cin;
cout = (a & b) | (b & cin) | (cin & a);
end
endmodule

```

Truth table:

INPUT			OUTPUT	
A	B	Cin	S	CO
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

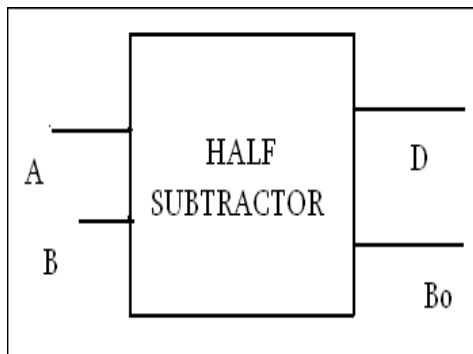
Simulation waveforms:



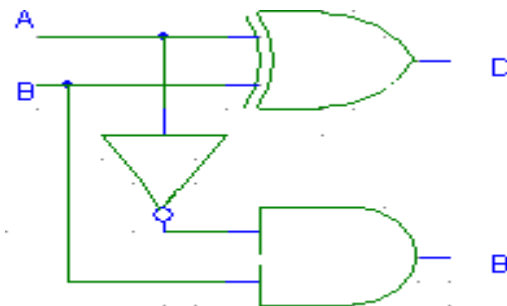
Result:

4.c HALF SUBTRACTOR

Logic symbol



Logic diagram



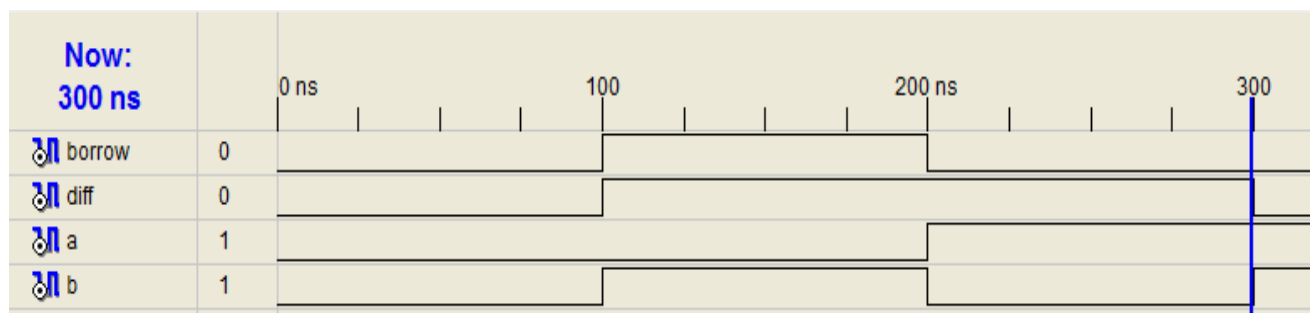
VERILOG CODE

Data Flow Description
<pre> module half_sub(input a, input b, output diff, output borrow); assign diff=a^b; assign borrow=(~a)&b; endmodule </pre>

Truth table:

INPUT		OUTPUT	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

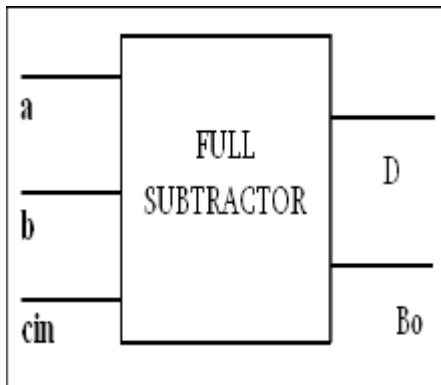
Simulation waveforms:



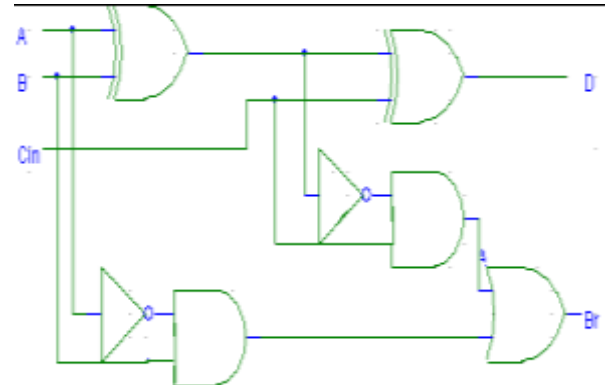
Result:

4.d FULL SUBTRACTOR

Logic symbol



Logic diagram



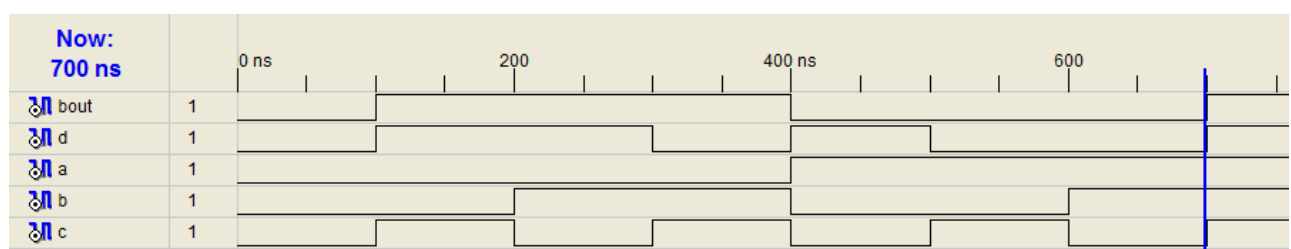
VERILOG CODE

Truth table:

Data Flow Description
<pre> module fullsub(a, b, c, d, bout); input a, b, c; output d, bout; assign d = a^b^c; assign bout= (~a & c) (~a & b) (b & c); endmodule </pre>

INPUT			OUTPUT	
A	B	Cin	Difference	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

Simulation waveforms:



Result:

Experiment 5

Aim: Design Verilog HDL to implement Decimal adder.

Binary Coded Decimal Adder:

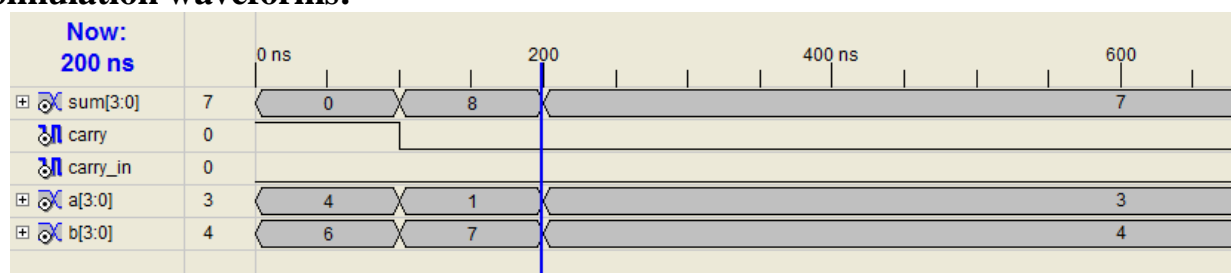
VERILOG CODE

Data Flow Description
<pre> module bcd_adder (a,b,carry_in,sum,carry); input [3:0] a,b; input carry_in; output [3:0] sum; output carry; //Internal variables reg [4:0] sum_temp; reg [3:0] sum; reg carry; always @(a,b,carry_in) begin sum_temp = a+b+carry_in; if(sum_temp > 9) begin sum_temp = sum_temp+6; carry = 1; sum = sum_temp[3:0]; end else begin carry = 0; sum = sum_temp[3:0]; end end endmodule </pre>

Truth table

Decimal	Binay (BCD)			
	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Simulation waveforms:



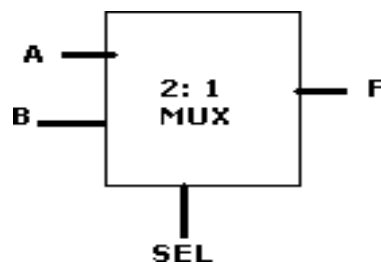
Result:

Experiment 6

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

6a. 2:1 Multiplexer:

Logic symbol



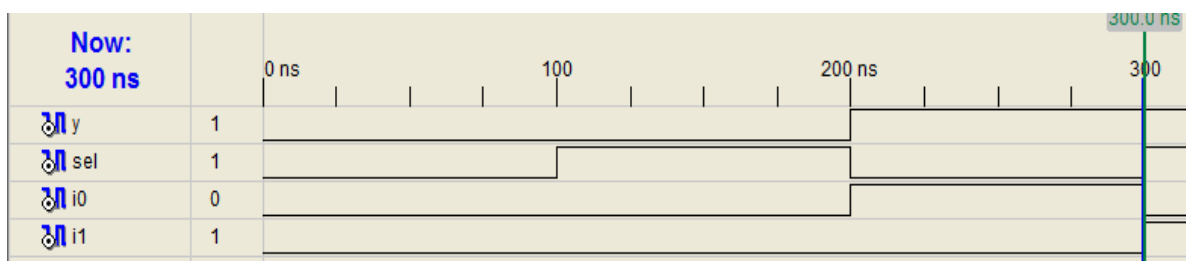
VERILOG CODE

Data Flow Description
<pre> module mux_2_1(input sel, input i0, i1, output y); assign y = sel ? i1 : i0; endmodule </pre>

Truth table

Input	Output
Sel	Y
0	i ₀
1	i ₁

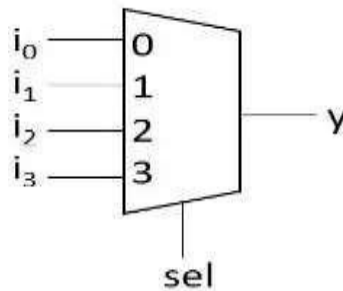
Simulation waveforms:



Result:

6.b 4:1 Multiplexer:

Logic symbol



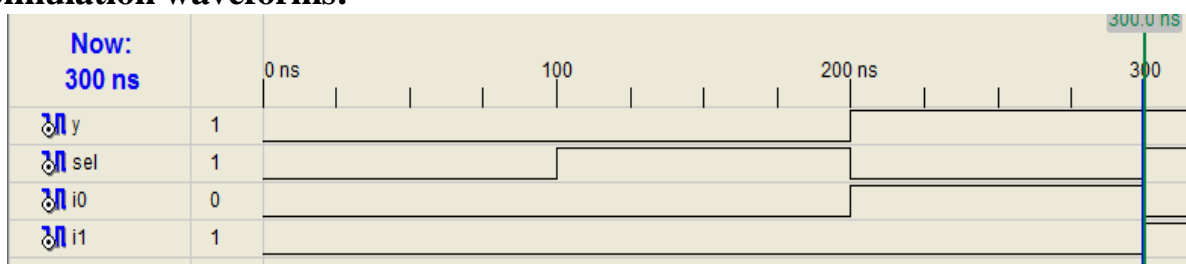
VERILOG CODE

Data Flow Description
<pre> module mux_4:1(input [1:0] sel, input i0,i1,i2,i3, output reg y); always @(*) begin case(sel) 2'h0: y = i0; 2'h1: y = i1; 2'h2: y = i2; 2'h3: y = i3; default: \$display("Invalid sel input"); end case end endmodule </pre>

Truth table

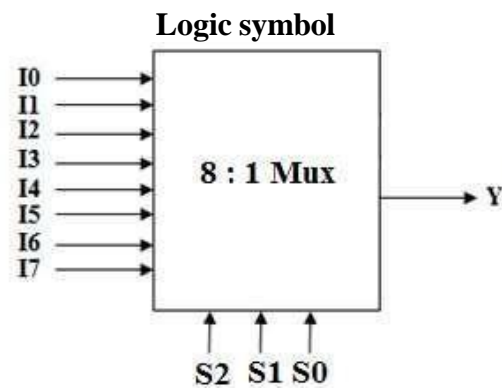
sel[0]	sel[1]	y
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

Simulation waveforms:



Result:

6.c 8:1 Multiplexer:



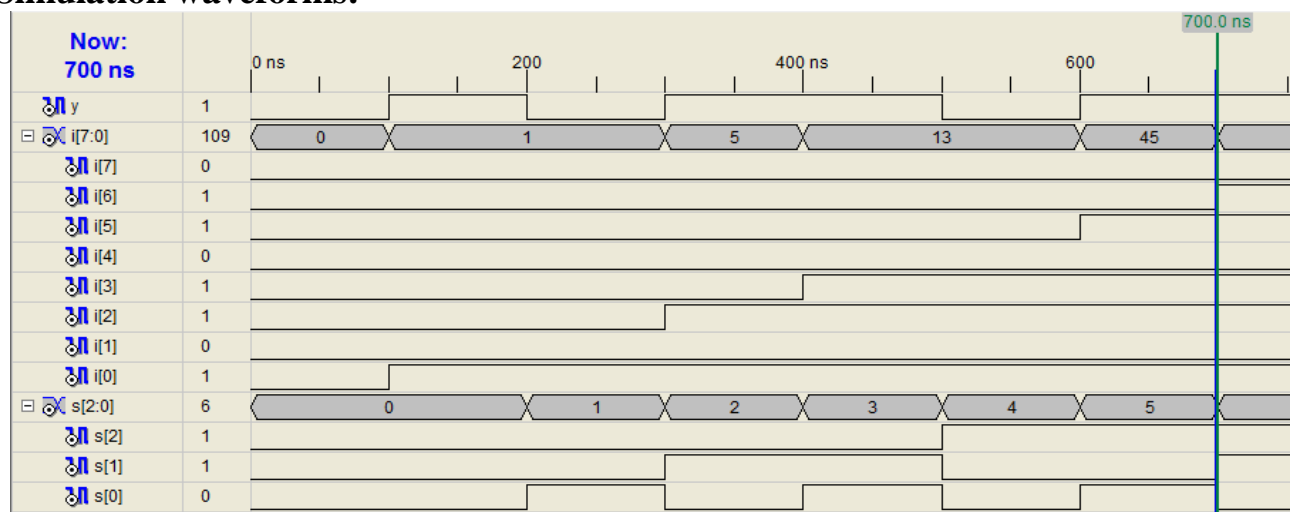
VERILOG CODE

Data Flow Description
<pre> module mux8to1(s, i, y); input [2:0]s; input [7:0]i; output y; reg y; always @ (i, s) begin case(s) 3'd0:y=i[0]; 3'd1:y=i[1]; 3'd2:y=i[2]; 3'd3:y=i[3]; 3'd4:y=i[4]; 3'd5:y=i[5]; 3'd6:y=i[6]; default :y=i[7]; endcase end endmodule </pre>

Truth table

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

Simulation waveforms:



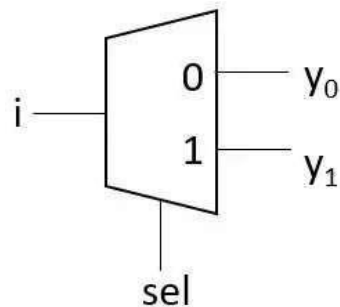
Result:

Experiment 7

Aim: Design Verilog program to implement types of De-Multiplexer.

7.a 1:2 DE-MULTIPLEXER

Logic symbol



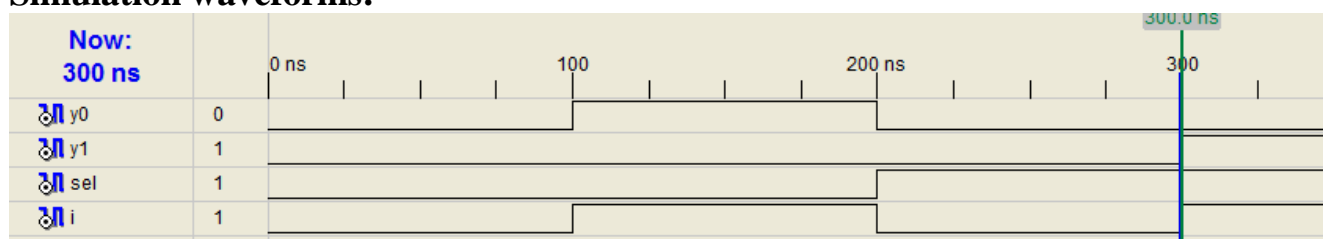
VERILOG CODE

Data Flow Description
<pre> module demux_2_1(input sel, input i, output y0, y1); assign {y0,y1} = sel?{1'b0 , i} : { i, 1'b0}; endmodule </pre>

Truth table

sel	y ₀	y ₁
0	i	0
1	0	i

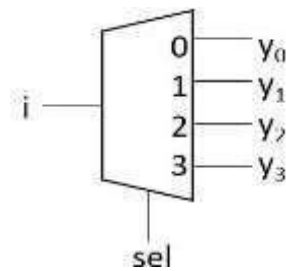
Simulation waveforms:



Result:

7.b 1:4 DEMULTIPLEXER

Logic symbol



VERILOG CODE

```

Data Flow Description

module demux_1_4( input [1:0] sel,
  input i, output reg y0,y1,y2,y3);

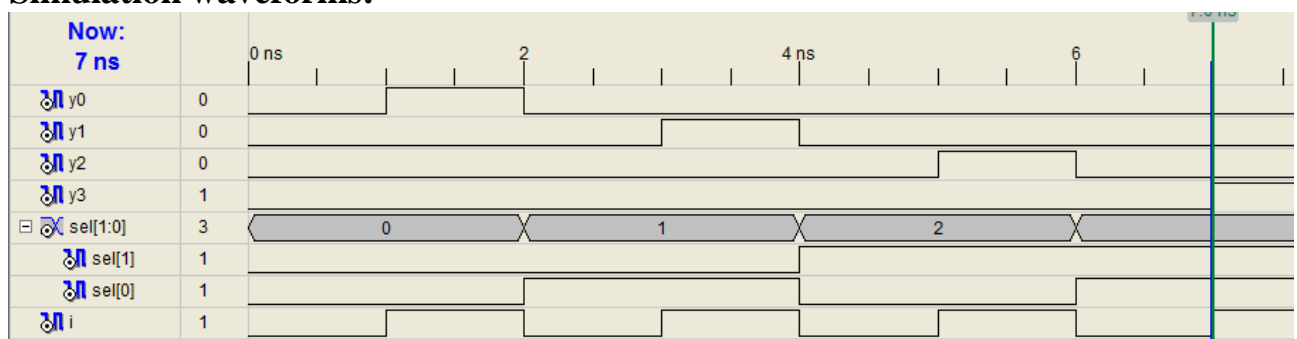
  always @(*) begin
    case(sel)
      2'h0: {y0,y1,y2,y3} = {i,3'b0};
      2'h1: {y0,y1,y2,y3} = {1'b0,i,2'b0};
      2'h2: {y0,y1,y2,y3} = {2'b0,i,1'b0};
      2'h3: {y0,y1,y2,y3} = {3'b0,i};
      default: $display("Invalid sel input");
    endcase
  end
endmodule

```

Truth table

sel[0]	sel[1]	y ₀	y ₁	y ₂	y ₃
0	0	i	0	0	0
0	1	0	i	0	0
1	0	0	0	i	0
1	1	0	0	0	i

Simulation waveforms:



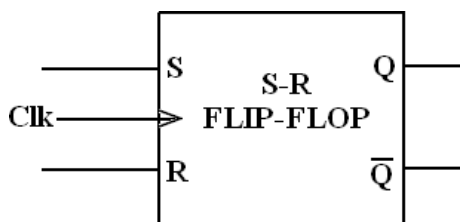
Result:

Experiment 8

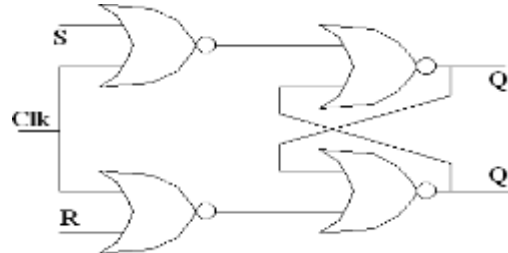
Aim: Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

8.a S R Flip Flop

Logic symbol



Logic diagram:



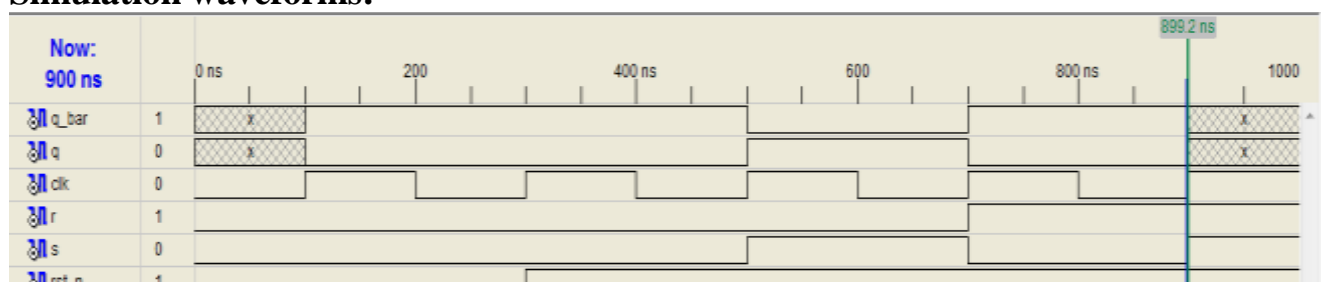
VERILOG CODE

Behavioral Description
<pre> module SR_flipflop (input clk, rst_n, input s,r, output reg q, output q_bar); always@(posedge clk) begin if(!rst_n) q <= 0; else begin case({s,r}) 2'b00: q <= q; 2'b01: q <= 1'b0; 2'b10: q <= 1'b1; 2'b11: q <= 1'bx; inputs endcase end end assign q_bar = ~q; endmodule </pre>

Truth table

INPUT			OUTPUT	
S	R	CLK	Q	Q'
0	0	↑	LAST Q	LAST Q'
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	D	D

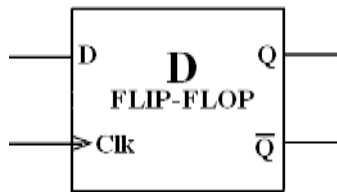
Simulation waveforms:



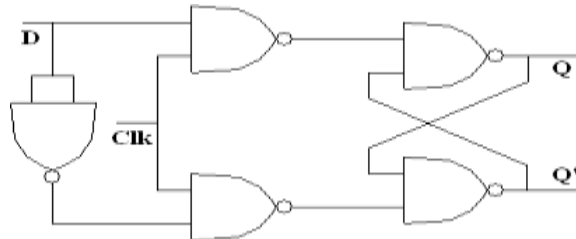
Result:

8.b D Flip Flop

Logic symbol



Logic diagram:



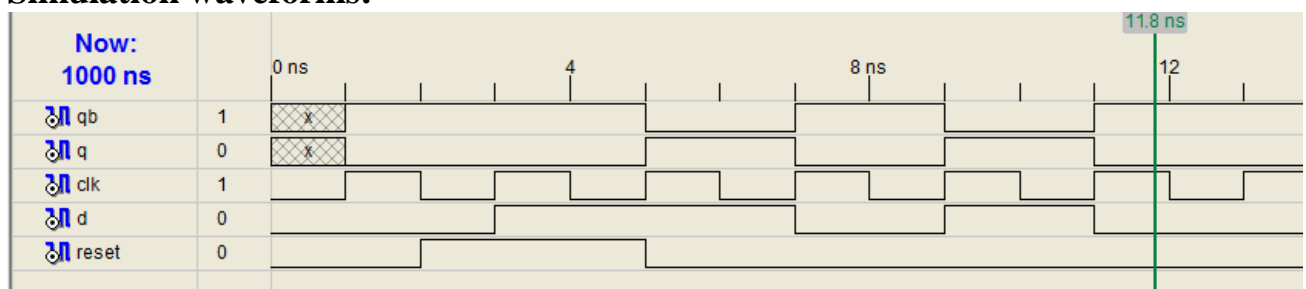
VERILOG CODE

Behavioral Description
<pre> module D_ff(d,clk,reset,q,qb); input d; input clk; input reset; output q; output qb; reg q, qb; always@(posedge clk or posedge reset) begin if(reset==1'b1) begin q<=1'b0; qb<=1'b1; end else begin q<=d; qb<=~d; end end endmodule </pre>

Truth table

INPUT		OUTPUT
D	CLK	Q
X	↓	NO CHANGE
0	↑	0
1	↑	1

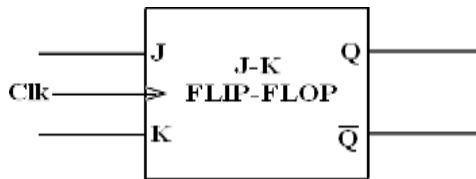
Simulation waveforms:



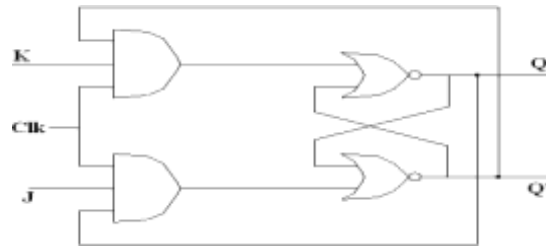
Result:

8.c JK Flip Flop

Logic symbol



Logic diagram:



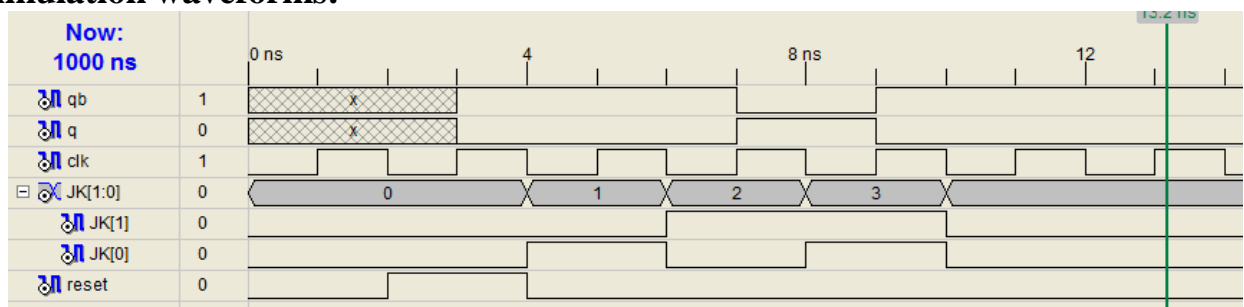
VERILOG CODE

Behavioral Description
<pre> module JK_f_f(JK,clk,reset,q,qb); input [1:0] JK; input clk; input reset; output q,qb; reg q,qb; always@(posedge clk) begin if (reset==1'b1) q = 1'd0 ; else begin case(JK) 2'b00:q= q ; 2'b01: q= 1'd0 ; 2'b10: q= 1'd1 ; 2'b11: q= ~q; endcase end qb = ~q; end endmodule </pre>

Truth table

INPUT			OUTPUT	
J	K	CLK	Q	Q'
0	0	↑	No Change	No Change
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	TOGGLE	TOGGLE

Simulation waveforms:



Result: