-SIR M. VISVESVARAYA INSTITUTE OF TECHNOLOGY

(Affiliated to VTU, Recognized by AICTE and Accredited by NBA, NAAC and an ISO 9001-2008 Certified Institution)

Bengaluru – 562157

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DIGITAL DESIGN AND COMPUTER ORANIZATION

CHOICE BASED CREDIT SYSTEM

BC302 - III Semester B.E

(*Academic Year 2023-24)*

Compiled and Prepared by:
Mrs. Rekha B N, Associate Professor,
Dept. of CSE

Under the Guidance of:
Dr. T N Anitha, Prof & Head
Professor. & Head Dept. of CSE

Department Vision and Mission

VISION

To create a center for imparting quality technical education and conducting research to meet the current and future challenges in the domain of computer science and engineering.

MISSION

- The computer science and engineering department strives for excellence in teaching, applying, promoting and imparting knowledge through comprehensive academic programs.
- Train students to effectively apply the knowledge to solve real-world problems, thus enhance their potential for life-long high-quality career and give them a competitive advantage in the ever-changing and fast paced computing world.
- Prepare students to demonstrate a sense of societal and ethical responsibilities in their professional endeavors.
- Creating amongst students and faculty a collaborative environment open to the free exchange of ideas, where research, creativity, innovation and entrepreneurship can flourish.

PROGRAM OUTCOMES

PO's	PO Description
PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

PSO's	PSO Description
PSO1	An ability to apply the knowledge of mathematics, science, engineering fundamentals and specialization to analyze, design and implement complex computer based systems
PSO2	An ability to investigate a problem by analysis, interpretation of data, design of experiments, implementation through appropriate techniques, skills and tools to provide valid conclusions
PSO3	Be able to imply the engineering and management principles to function effectively as an individual, member or leader in diverse teams and multidisciplinary settings and also to communicate effectively on technical subject matters
PSO4	An understanding of professional, ethical, legal, security issues, societal responsibilities and indulge in life-long learning

Digital Design & Computer Organization

Hours/Week: 02 **Exam Hours: 2**

CIE Marks:25 Total Hours: 24

Semester: 3rd

Course Outcomes:

The student should be able to:

CO1:Apply the K- map Techniques to simplify the Boolean Expression and build the same in Verilog Language.

CO2: Design Different types of combinational and sequential circuits along with Verilog Programs.

Usage of the Tool:

Verilog is a Hardware Description Language (HDL). It is a language used for describing a digital system such as a network switch, a microprocessor, a memory, or a flip-flop. We can describe any digital hardware by using HDL at any level. Designs described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog was developed to simplify the process and make the HDL more robust and flexible. Today, Verilog is the most popular HDL used and practiced throughout the semiconductor industry.

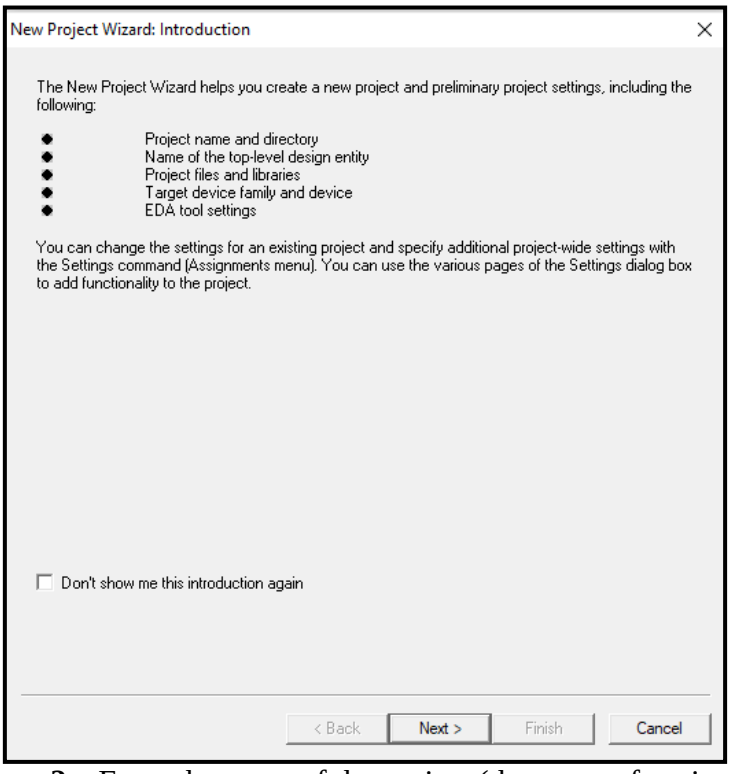
HDL was developed to enhance the design process by allowing engineers to describe the desired hardware's functionality and let automation tools convert that behavior into actual hardware elements like combinational gates and sequential logic.

Verilog is like any other hardware description language. It permits the designers to design the designs in either Bottom-up or Top-down methodology.

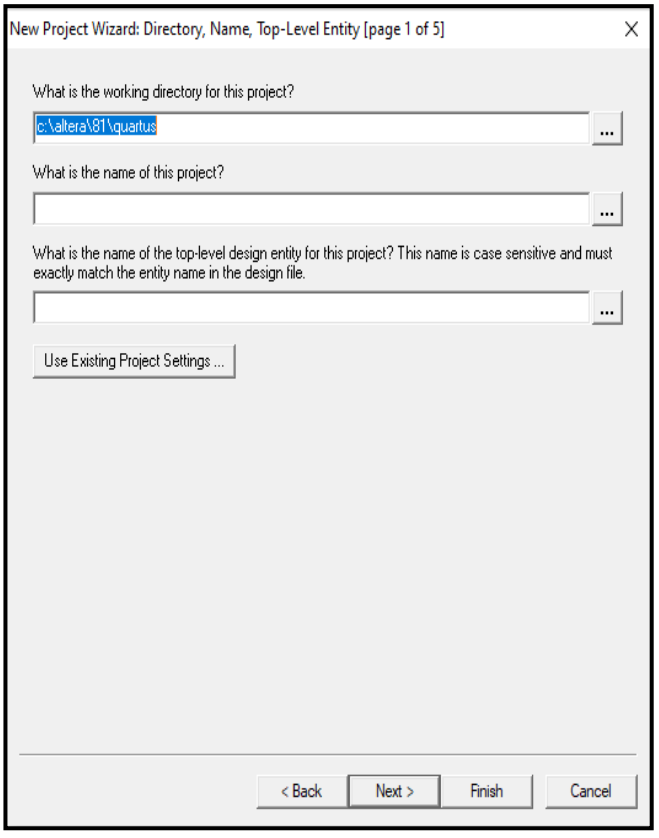
- **Bottom-Up Design:** The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standards gates. This design gives a way to design new structural, hierarchical design methods.
- **Top-Down Design:** It allows early testing, easy change of different technologies, and structured system design and offers many other benefits.

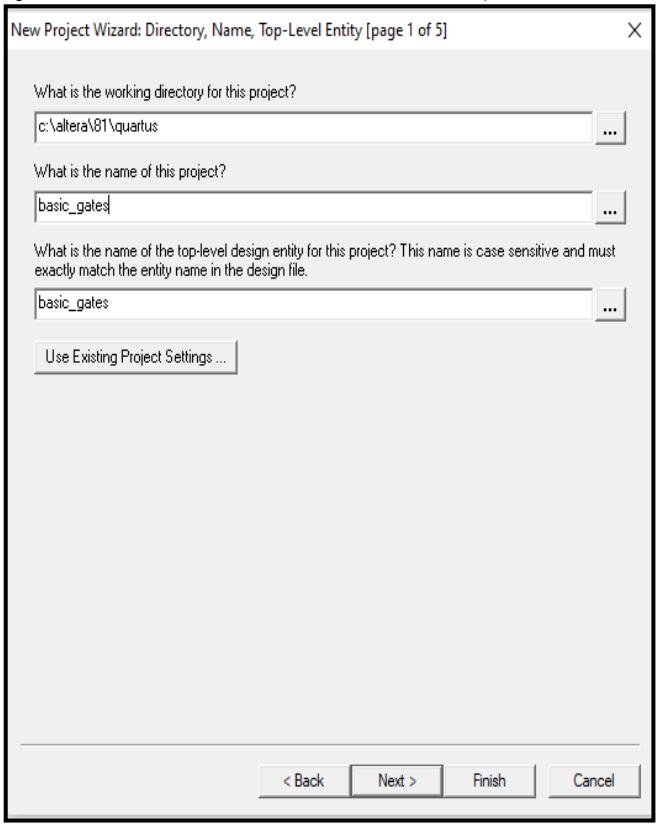
The Quartus II Analysis and Synthesis stage of the compilation flow runs Quartus II integrated synthesis, which fully supports Verilog HDL, VHDL, and Altera-specific languages, and supports major features of the SystemVerilog language. This tool includes many steps. To make user feel comfortable with the tool the steps are given below:-

1. Open the software and select **"Create new project"** then new project wizard will get opened **.**Click **Next.**



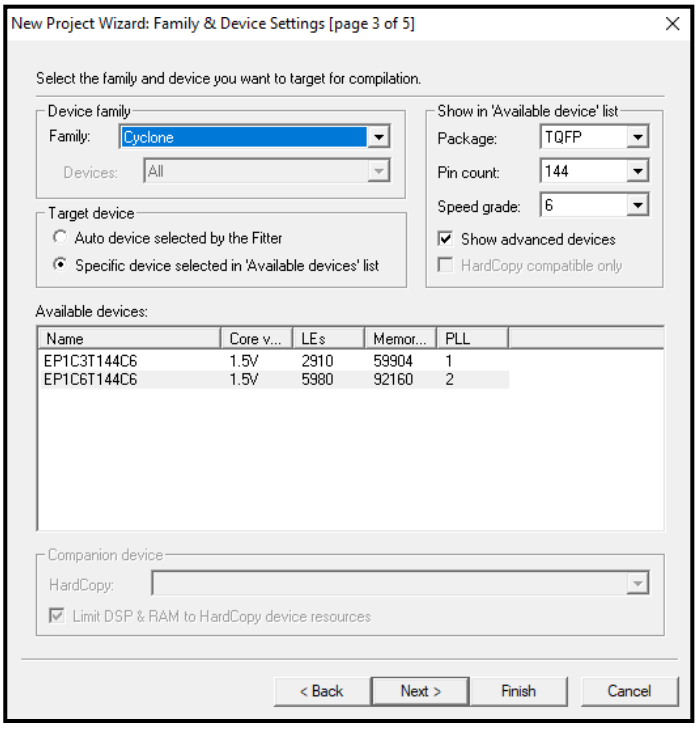
2. Enter the name of the project (the name of project should be same as module name).



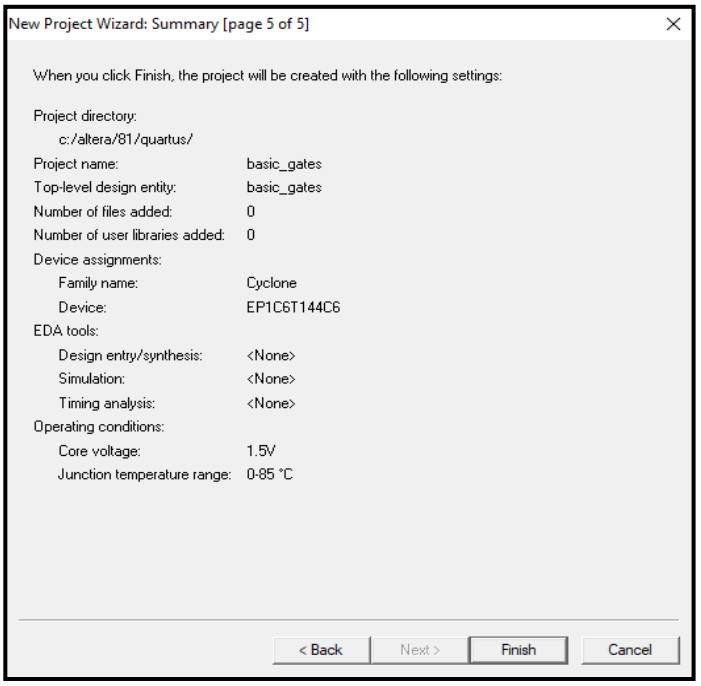


3. Select next ,will get the options to enter the following details as shown below:

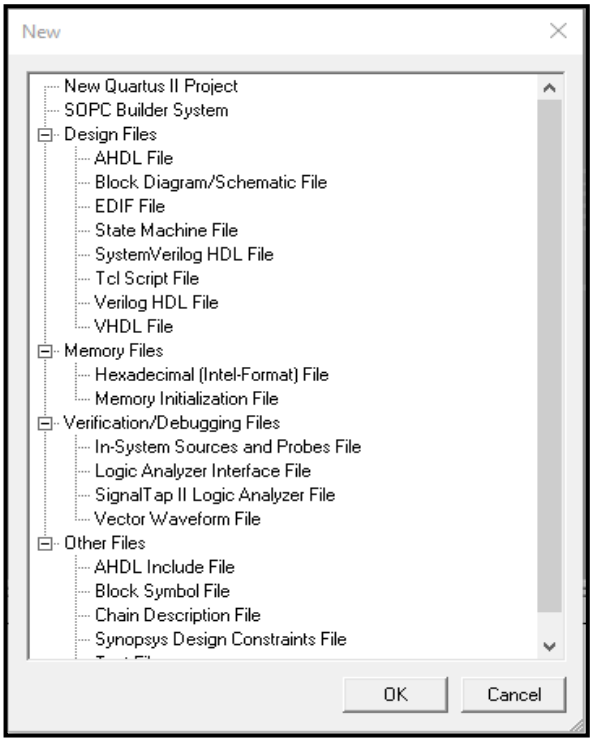
Device family : cyclone
Device : EP1C6T144C6
Package : TQFP
Pincount : 144
Speed grade :6



4. Click Next and once we get this window select Finish.



5. Select NEW in FILE MENU**.**Select Verilog HDL File. Then type the Verilog code and save in folder.



1. Select Processing Drop down menu, Choose Start compilation

2. If there are errors go back to the Verilog code and correct it. Once the compilation is successful.

3. Under the processing drop down box select simulator tool select the simulator mode to functional and click on generate functional simulation netlist. We get the success message.

4. Under simulator tool click on open. In the empty location right click . Click on insert on NODEor BUS. Then click on NODE finder. In the window opened select pins to unassigned. Click on List. It will list all the Net list select all and click ok.

5. The input and output appears give appropriate input.

6. On the simulator tool click on start simulation. Simulation success message will be prompted.

7. On simulator tool click on report to see the output.

EXPERIMENT: 1

AIM: Given a four variable logic expression, simplify it using appropriate technique andsimulate the same using basic gates.

DESCRIPTION:

A gate has one or more inputs but only one output. The basic logic gates are the building blocks of complex logic circuits. The most basic gates are -the NOT gate (inverter), the OR gate and the AND gate. Simplification of Boolean function reduces the gate count required to implement the circuit, the circuit works faster and circuit require less power consumption.

The various Boolean expression simplification techniques are

1) Algebraic techniques
2) Karnaugh Map/K-Map Method
3) Quine McCluskey Method
4) Entered Variable Map/ MEV/EMV Method

A Karnaugh map provides a systematic method for simplifying Boolean expressions. The Karnaugh map is an array of cells in which each cell represents a binary value of the input variables. The cells are arranged in a way so that simplification of a given expression is simply a matter of properly grouping the cells. Karnaugh maps can be used for expressions with two, three, four. and five variables.

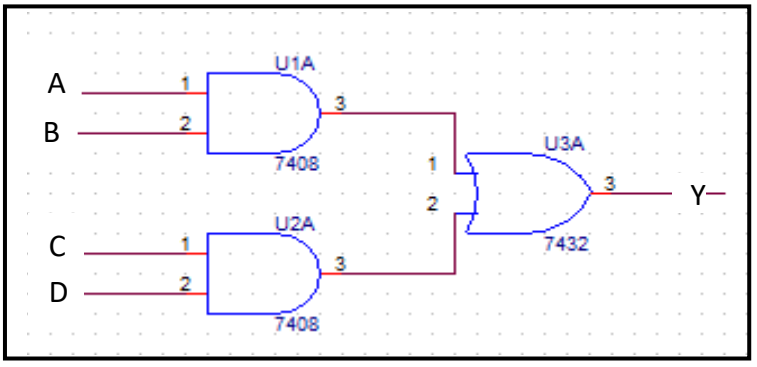
Simplify following function using K-map technique f(A,B,C,D)=∑m(3,7,11,12,13,14,15)

K-MAP:

ab \ cd	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	0

Y = ab + cd

CIRCUIT DIAGRAM:

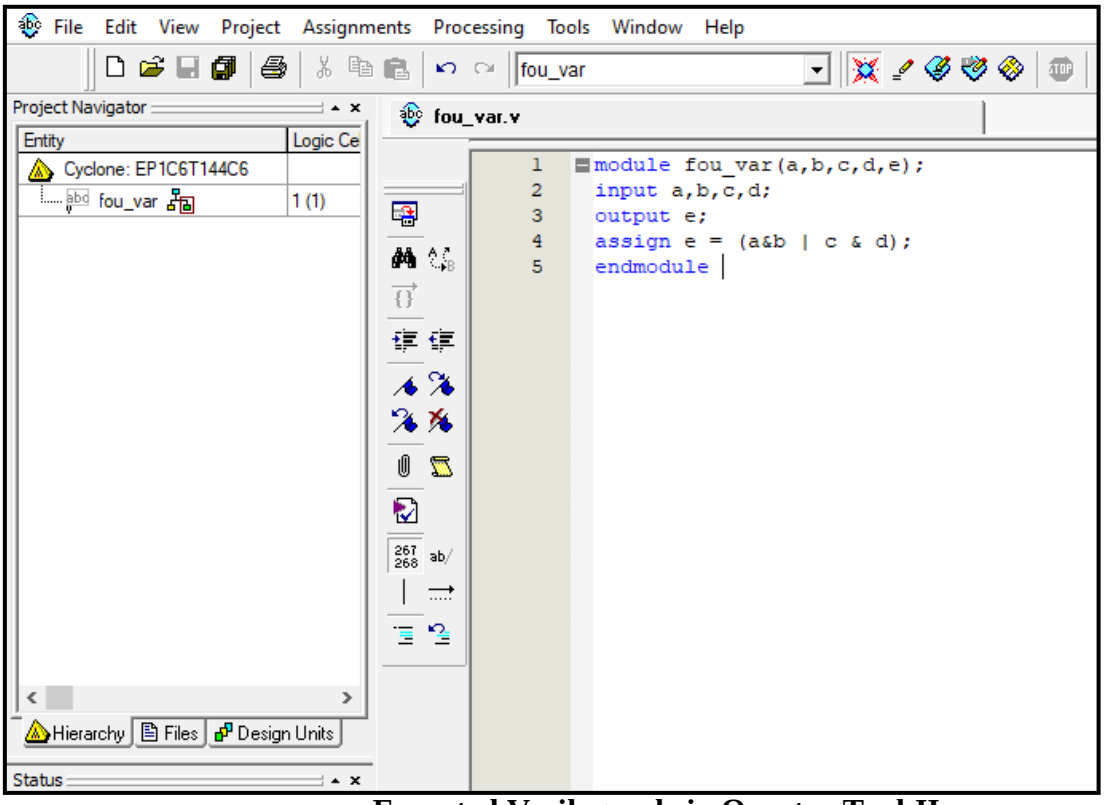


TRUTH TABLE:

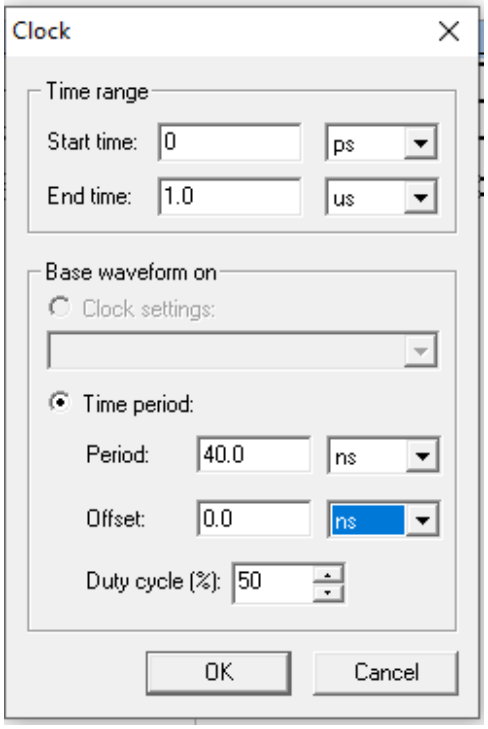
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

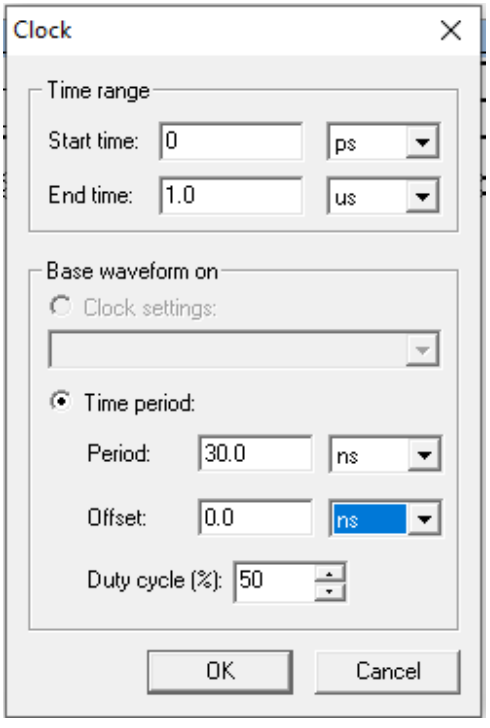
VERILOG CODE:

```
module four_var(a,b,c,d,e) ;
input a,b,c,d ;
output e ;
assign e = ( a&b | c & d ) ;
endmodule
```

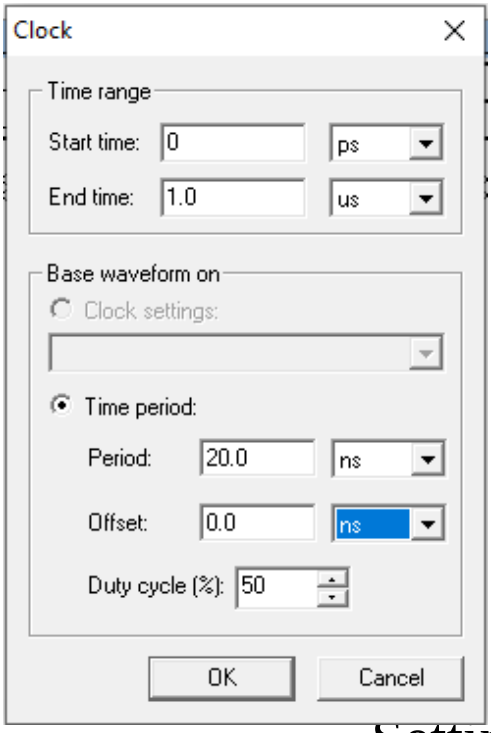


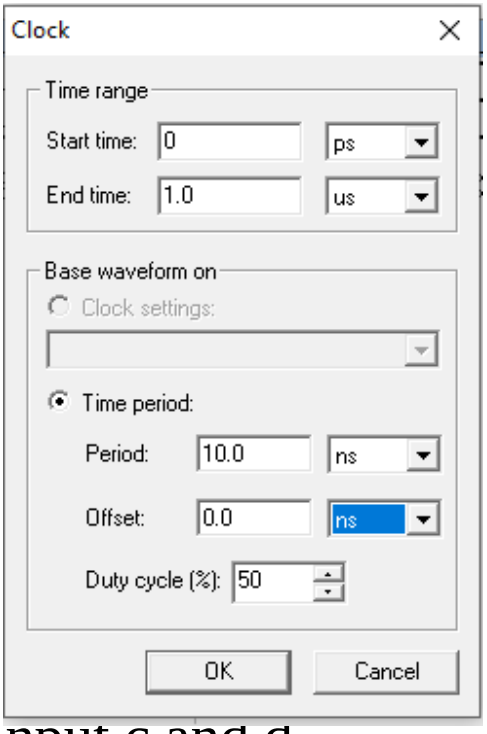
Executed Verilog code in Quartus Tool-II





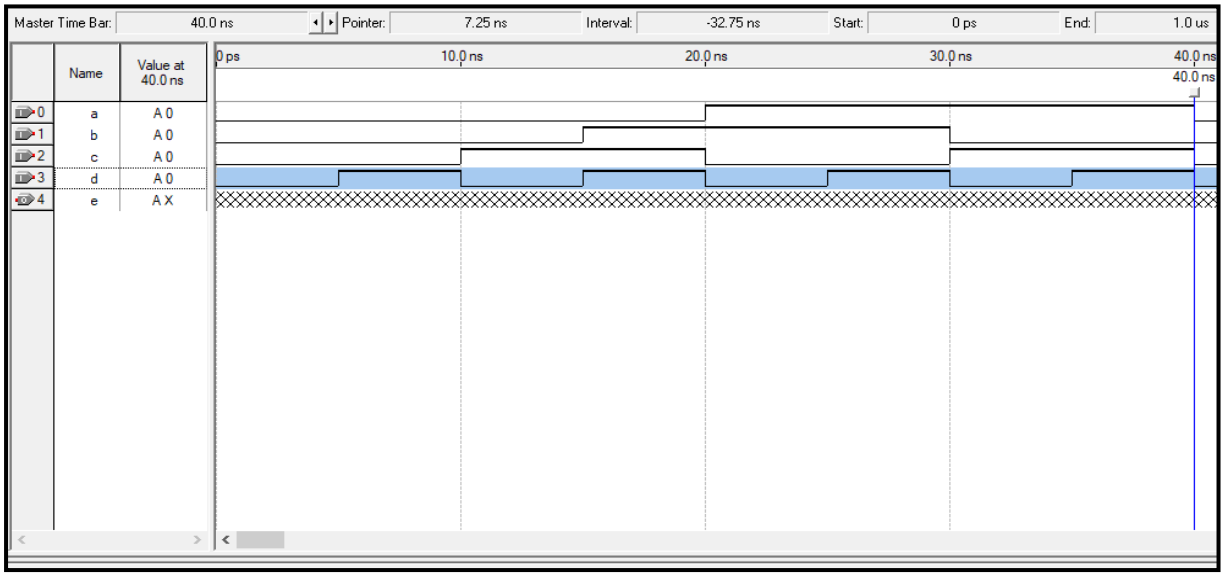
Settings of input a and b



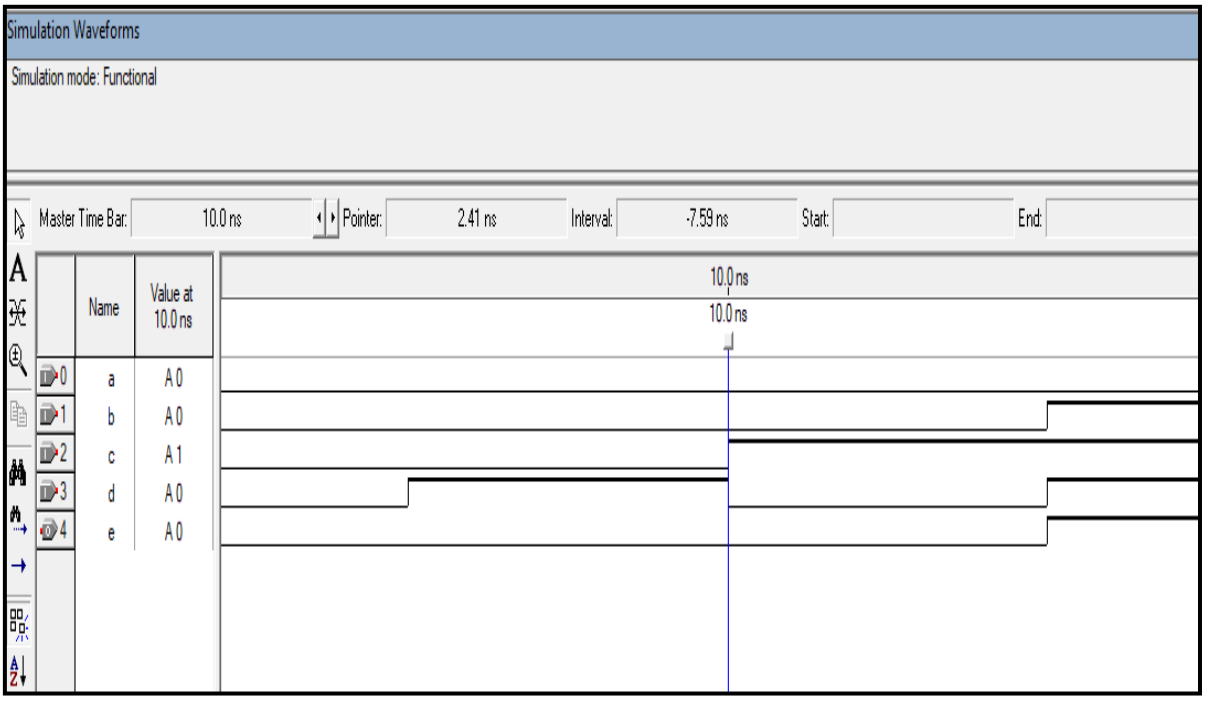


Settings of input c and d

SIMULATION WAVEFORM



Inputs to variables a,b.c and d



Output of four variable logic expression

RESULT: The four variable logic expression is simplified using K-map and simulated the same using basic gates.

EXPERIMENT: 2

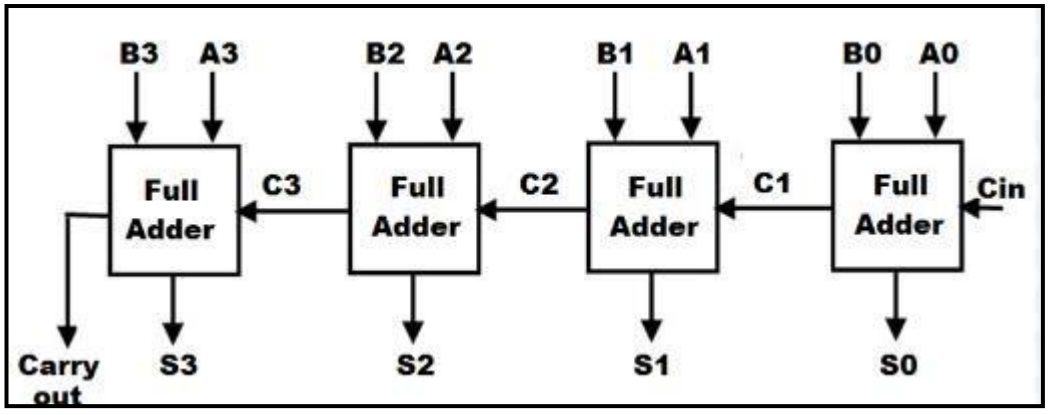
AIM: Design a 4 bit full adder and subtractor and simulate the same using basic gates.

DESCRIPTION:

A structure of multiple full adders is cascaded in a manner to gives the results of the addition of an n bit binary sequence. This adder includes cascaded full adders in its structure so, the carry will be generated at every full adder stage in a ripple-carry adder circuit. These carry output at each full adder stage is forwarded to its next full adder and there applied as a carry input to it. This process continues up to its last full adder stage. So, each carry output bit is rippled to the next stage of a full adder.

The below diagram represents the 4-bit ripple-carry adder. In this adder, four full adders are connected in cascade. Cin is the carry input bit and it is zero always. When this input carry 'Cin' is applied to the two input sequences $A_0A_1A_2$ A_3 and B_0 B_1 B_2 B_3 then output represented with S_0 S_1 S_2 S_3 and output carry C0UT.

CIRCUIT DIAGRAM:



VERILOG CODE:

```
module ripple_adder(a, b, cin, sum, cout);
input a, b, cin;
output sum, cout;
assign sum = a ^ b ^ cin;
assign cout = ((a ^ b) & cin) | (a & b);
endmodule
module fourbit_fulladder(a, b, sum, cout);
input [3:0] a;
input [3:0] b;
output [3:0] sum;
output cout;

wire c1, c2, c3;
ripple_adder fa0(a[0], b[0], 0, sum[0], c1);
ripple_adder fa1(a[1], b[1], c1, sum[1], c2);
```

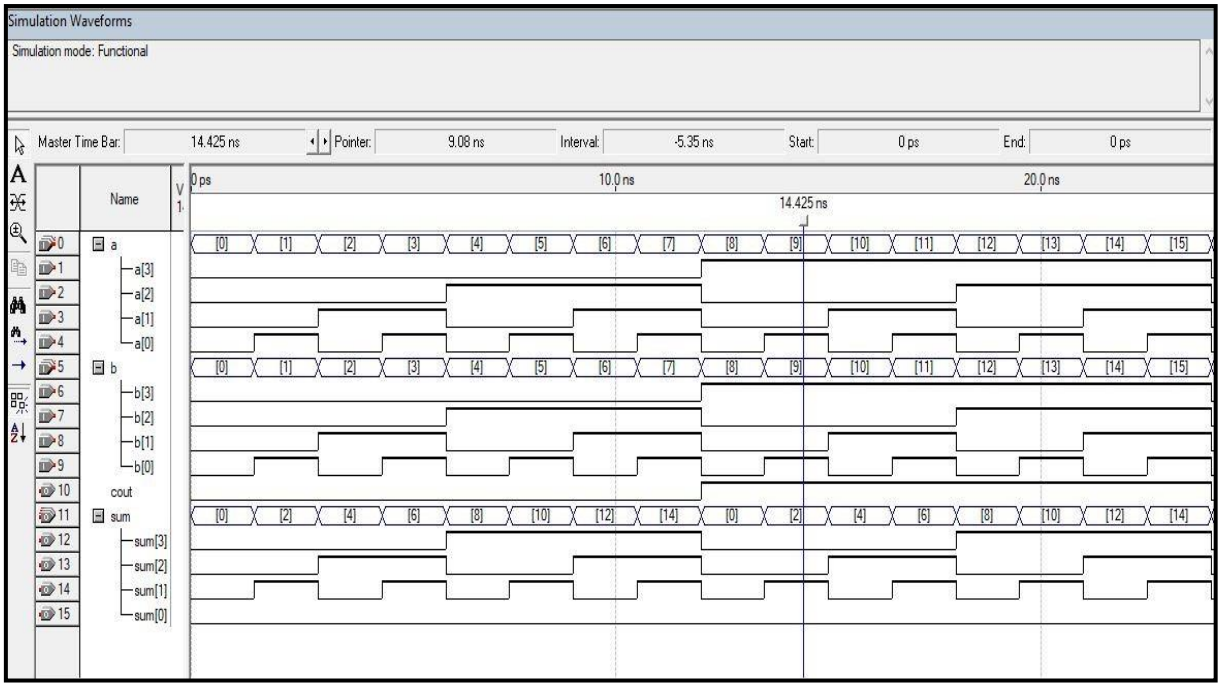
```
ripple_adder fa2(a[2], b[2], c2, sum[2], c3);
ripple_adder fa3(a[3], b[3], c3, sum[3], cout);

endmodule
```

Simulator Tool | fourbit_fulladder.v | Compilation Report - Flow Summary | fourbit_fulladder.vwf

```
1  module ripple_adder(a, b, cin, sum, cout);
2      input a, b, cin;
3      output sum, cout;
4
5      assign sum = a ^ b ^ cin;
6      assign cout = ((a ^ b) & cin) | (a & b);
7  endmodule
8  module fourbit_fulladder(a, b, sum, cout);
9      input [3:0] a;
10     input [3:0] b;
11     output [3:0] sum;
12     output cout;
13
14     wire c1, c2, c3;
15     ripple_adder fa0(a[0], b[0], 0, sum[0], c1);
16     ripple_adder fa1(a[1], b[1], c1, sum[1], c2);
17     ripple_adder fa2(a[2], b[2], c2, sum[2], c3);
18     ripple_adder fa3(a[3], b[3], c3, sum[3], cout);
19
20 endmodule
21
```

Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM



Output of 4 bit full adder and subtractor

RESULT: The four bit full adder and subtractor is designed and simulated the same using basic gates.

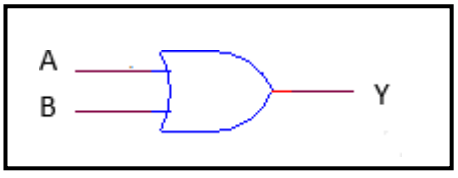
EXPERIMENT: 3

AIM: Design Verilog HDL to implement simple circuits using structural, Dataflow and Behavioral model.

DESCRIPTION:

- **Behavioral modeling** describes a system's behavior or function in an algorithmic fashion. It is the most abstract style and consists of one or more process statements. Each process statement is a single concurrent statement that itself contains one or more sequential statements. Sequential statements are executed sequentially by a simulator, the same as the execution of sequential statements in a conventional programming language.
- **Dataflow modeling** describes a system in terms of how data flows through the system. Data dependencies in the description match those in a typical hardware implementation. A dataflow description directly implies a corresponding gate-level implementation. Dataflow descriptions consist of one or more concurrent signal assignment statements.
- **Structural modeling** describes a system in terms of its structure and interconnections between components. It uses component instantiation statements to describe how components are connected together to form the system.

Example1: OR Gate



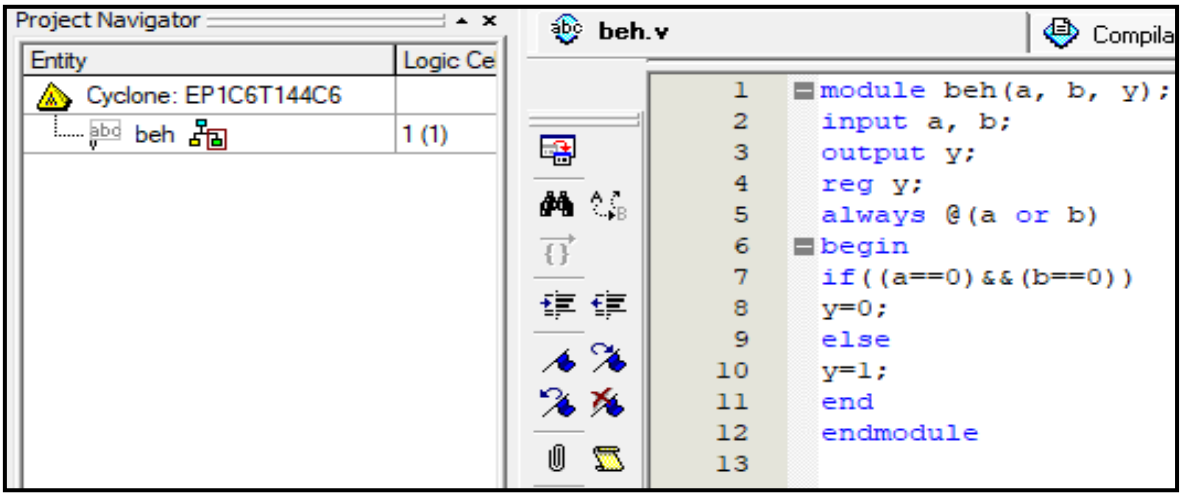
TRUTH TABLE:

A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

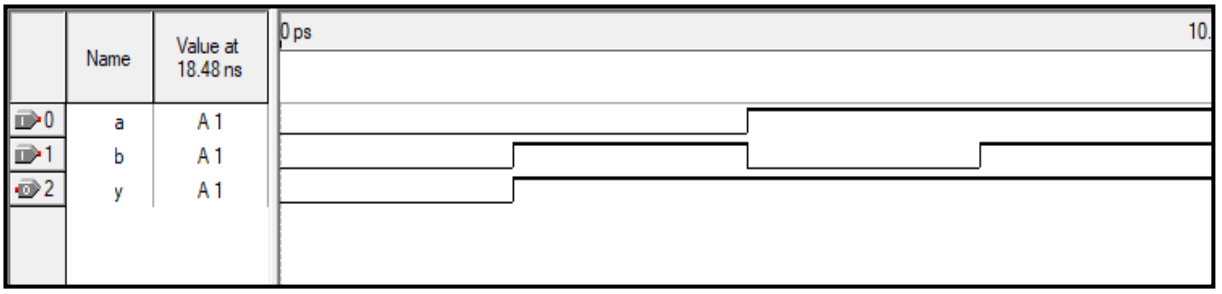
Behavioral modeling: VERILOG CODE:

```
module beh (a, b, y);
input a, b;
output y;
reg y;
always @ (a or b)
```

```
begin
if ((a = = 0) && (b = = 0))
y = 0;
else
y = 1;
end
endm
odule
```



Executed Verilog code in Quartus Tool-II

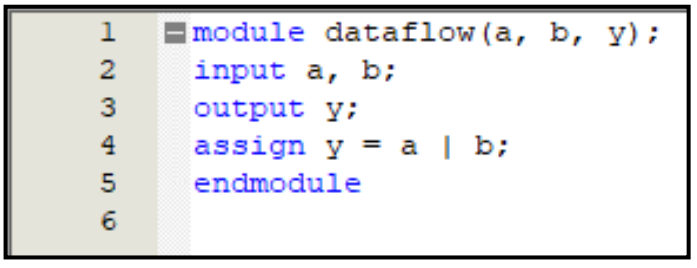


Output of Behavioral model

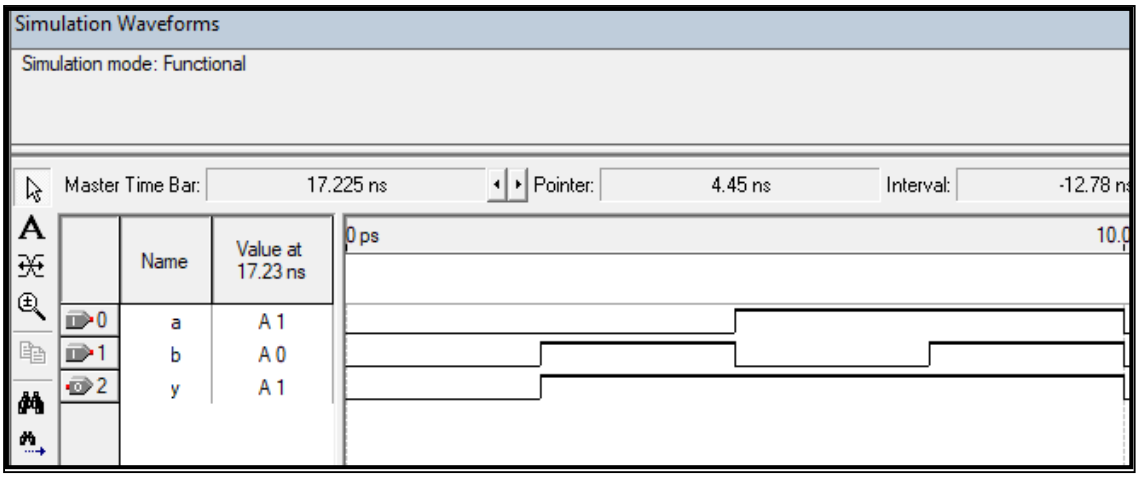
Dataflow modeling

VERILOG CODE:

module dataflow (a, b, y);
input a, b;
output y;
assign y = a | b;
endmodule



Executed Verilog code in Quartus Tool-II



Output of Dataflow model

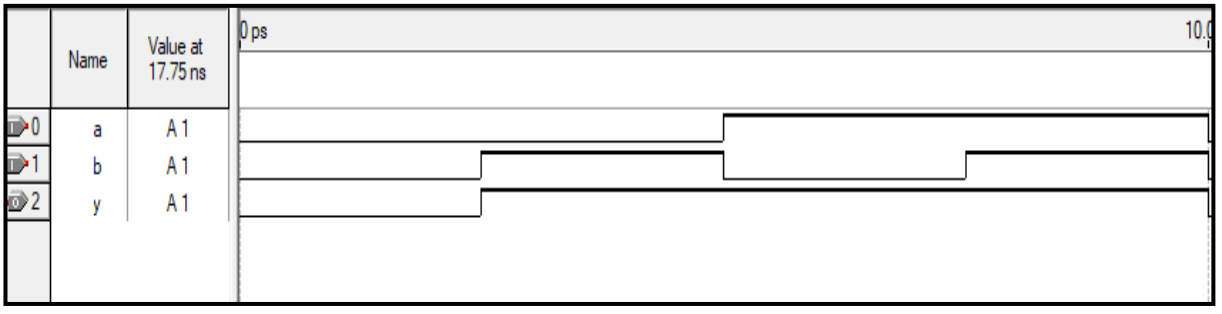
Structural modeling :

VERILOG CODE:

```
module structural(a, b, y);
input a, b;
output y;
or g1(y, a, b);
endmodule
```

```
1  module structural(a, b, y);
2    input a, b;
3    output y;
4    or g1(y,a,b);
5    endmodule
6
```

Executed Verilog code in Quartus Tool-II



Output of Structural model

RESULT: The simple circuits are implemented using structural, Dataflow and Behavioral model.

Experiment No.4:

AIM: Design Verilog HDL to implement Binary Adder- Subtractor – Half and Full Adder & Half and Full subtractor.

Description

Half-Adder: A combinational logic circuit that performs the addition of two data bits, A and B, is called a half-adder. Addition will result in two output bits; one of which is the sum bit, S, and the other is the carry bit, C. The Boolean functions describing the half-adder are:

Sum = $A \oplus B$ Cout = A B

Full-Adder: The half-adder does not take the carry bit from its previous stage into account. This carry bit from its previous stage is called carry-in bit. A combinational logic circuit that adds two data bits, A and B, and a carry-in bit, Cin , is called a full-adder. The Boolean functions describing the full-adder are:

Sum = $A \oplus B \oplus Cin$ Cout = AB + Cin ($A \oplus B$)

Half Subtractor: Subtracting a single-bit binary value B from another A (i.e. A – B) produces a difference bit D and a borrow out bit B-out. This operation is called half subtraction and the circuit to realize it is called a half subtractor. The Boolean functions describing the half- Subtractor are:

D = $A \oplus B$ Bout = A B

Full Subtractor: Subtracting two single-bit binary values, B, Cin from a single-bit value A produces a difference bit D and a borrow out Br bit. This is called full subtraction. The Boolean functions describing the full-subtracter are:

D = $A \oplus B \oplus Cin$ Br= A'B + A'(Cin) + B(Cin)

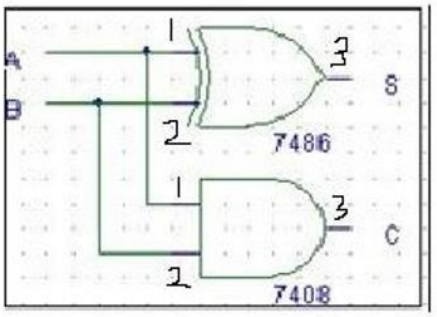
TRUTH TABLE FOR HALF ADDER

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

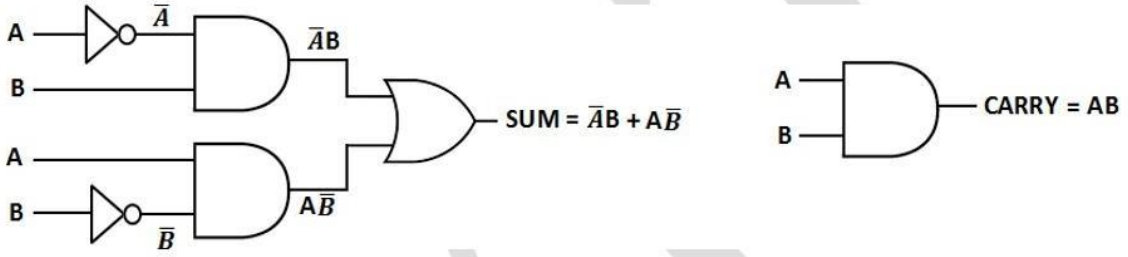
BOOLEAN EXPRESSIONS:

$S=\bar{A}B+A\bar{B}=A \oplus B$
$C=A\ B$

Circuit diagram:



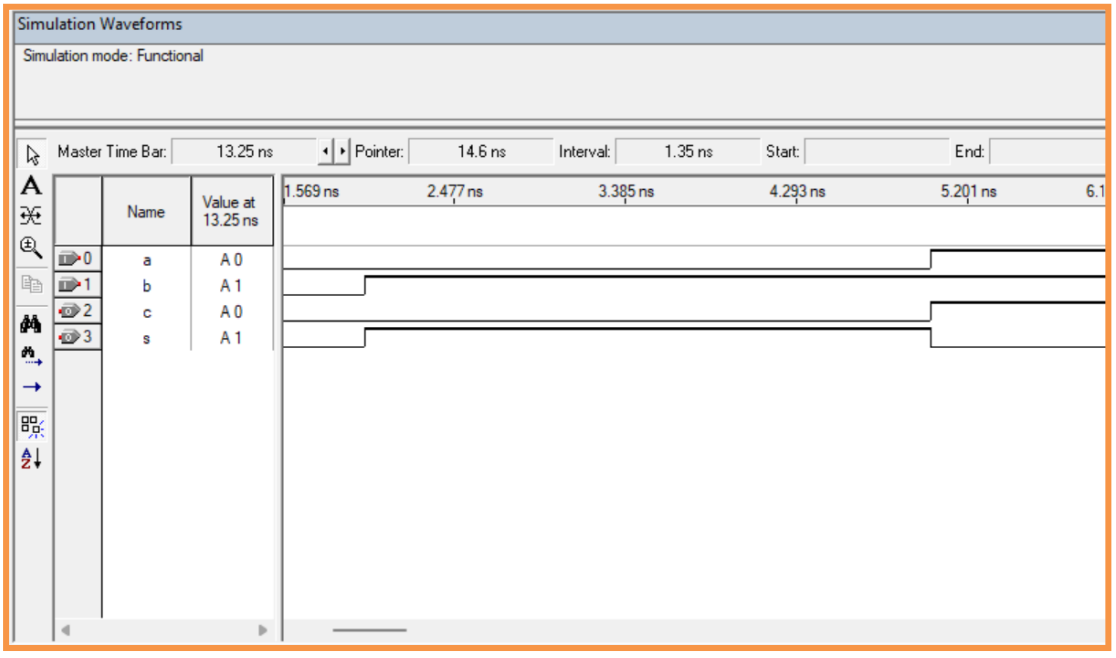
Using AND/OR/NOT:



Verilog Code

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

Simulation Results:



FULL ADDER TRUTH TABLE

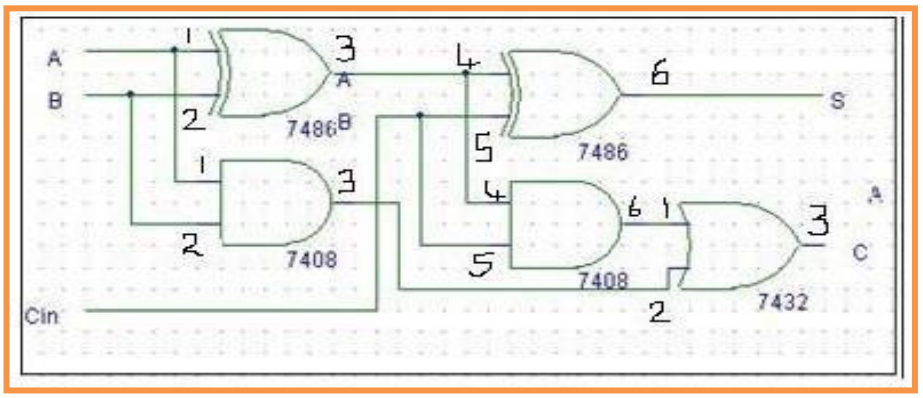
INPUTS			OUTPUTS	
A	**B**	**Cin**	**S**	**C**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

BOOLEAN EXPRESSIONS:

S= A ⊕ B ⊕ Cin

C= Cin (A ⊕ B) + AB

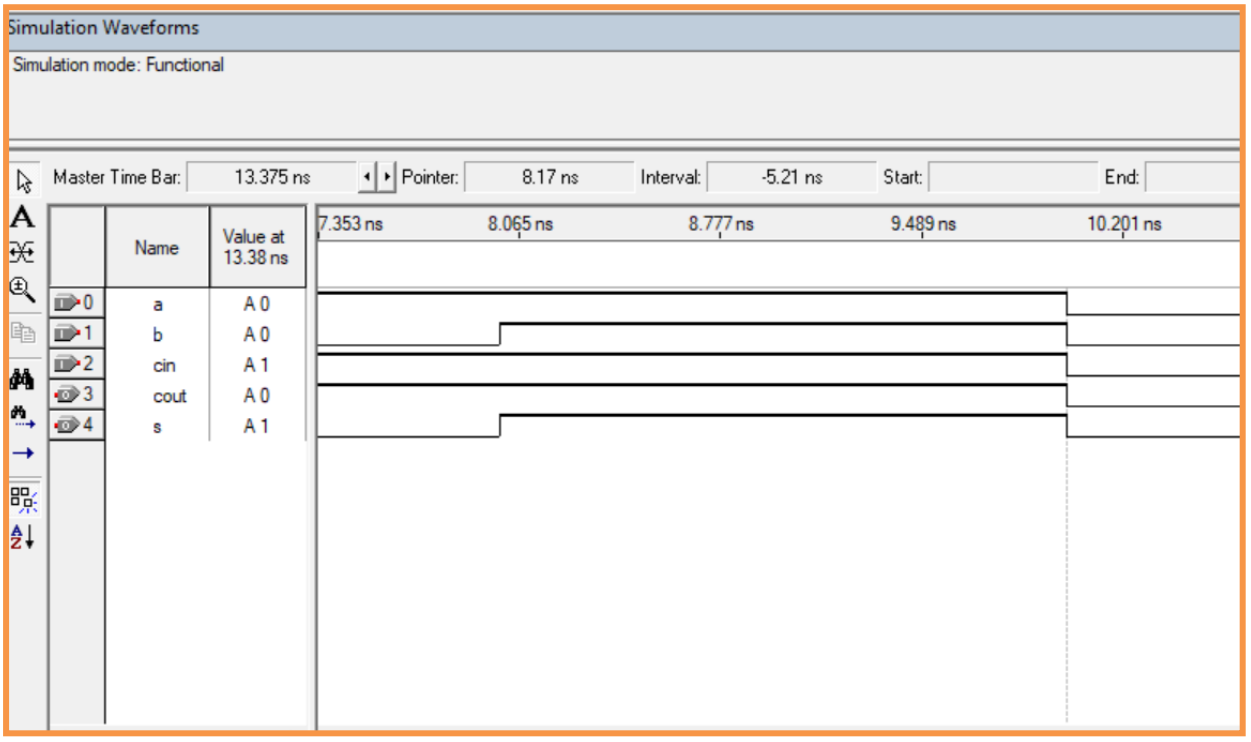
Circuit Diagram:



Verilog Code:

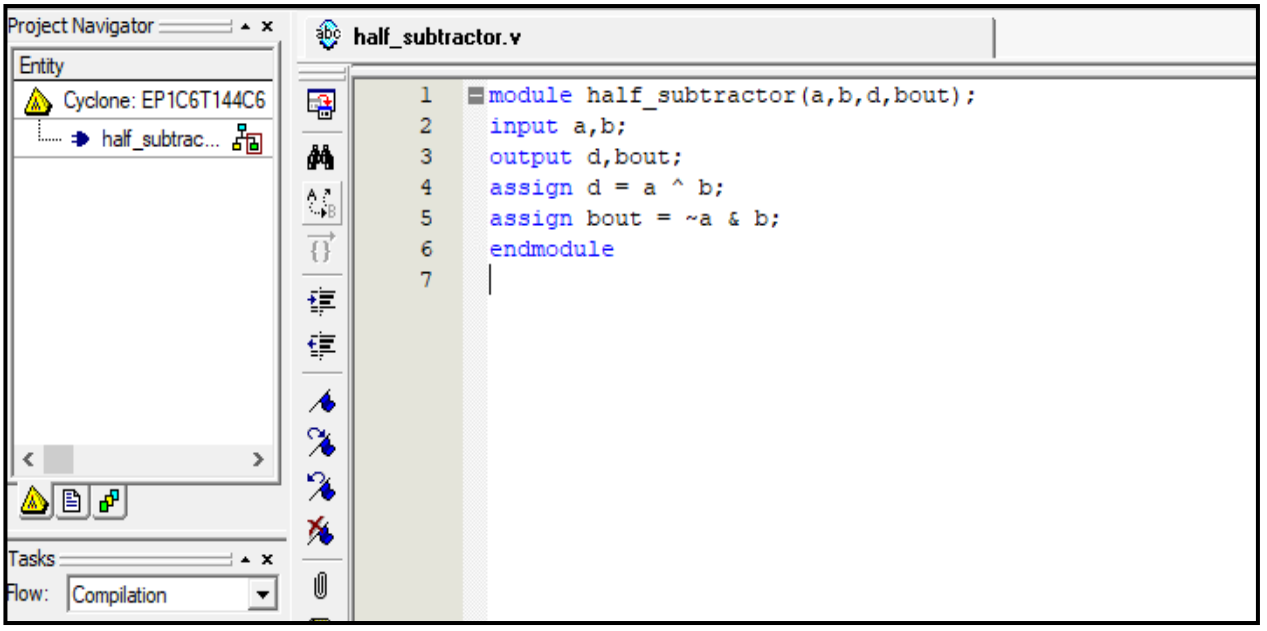
```
module full_adder(a,b,cin,s,cout);
input a,b,cin;
output s,cout;
assign s=a^b^cin;
assign cout= (a & b) | (b & cin) | (a & cin);
endmodule
```

Simulation Results:



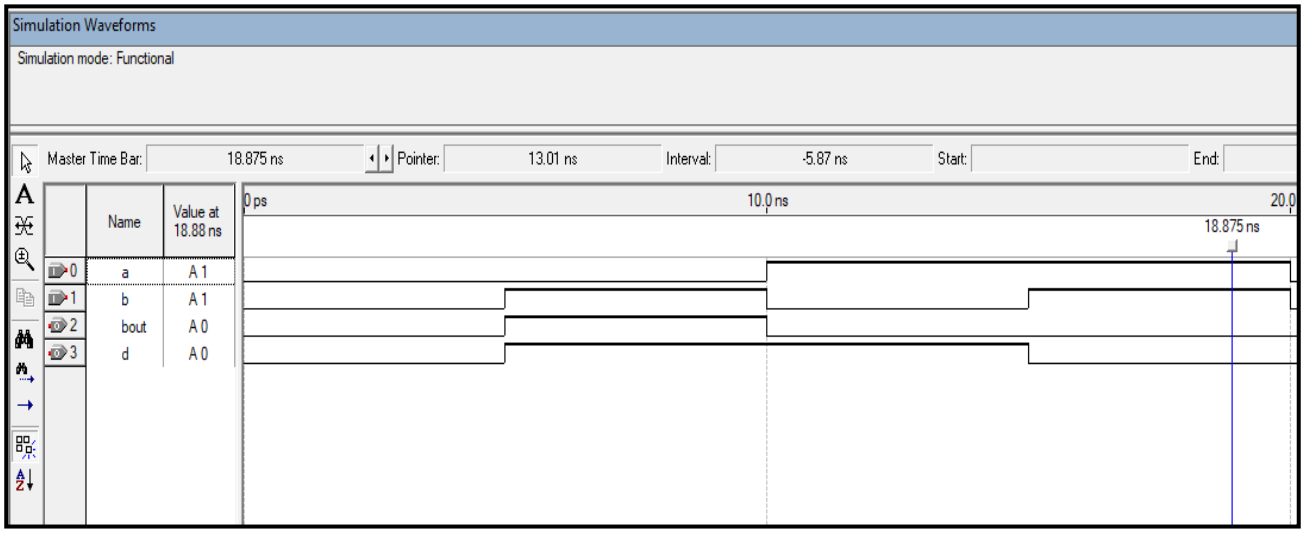
VERILOG CODE:

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



Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM



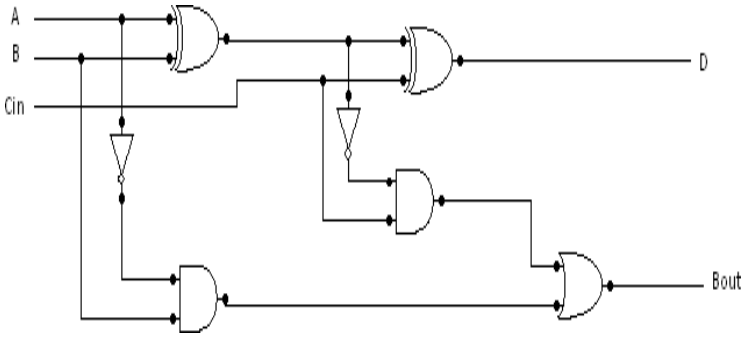
Output of Half Subtractor

(d) Full subtractor

Truth table

A	B	Bin	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

Full subtractor using two half subtractor:



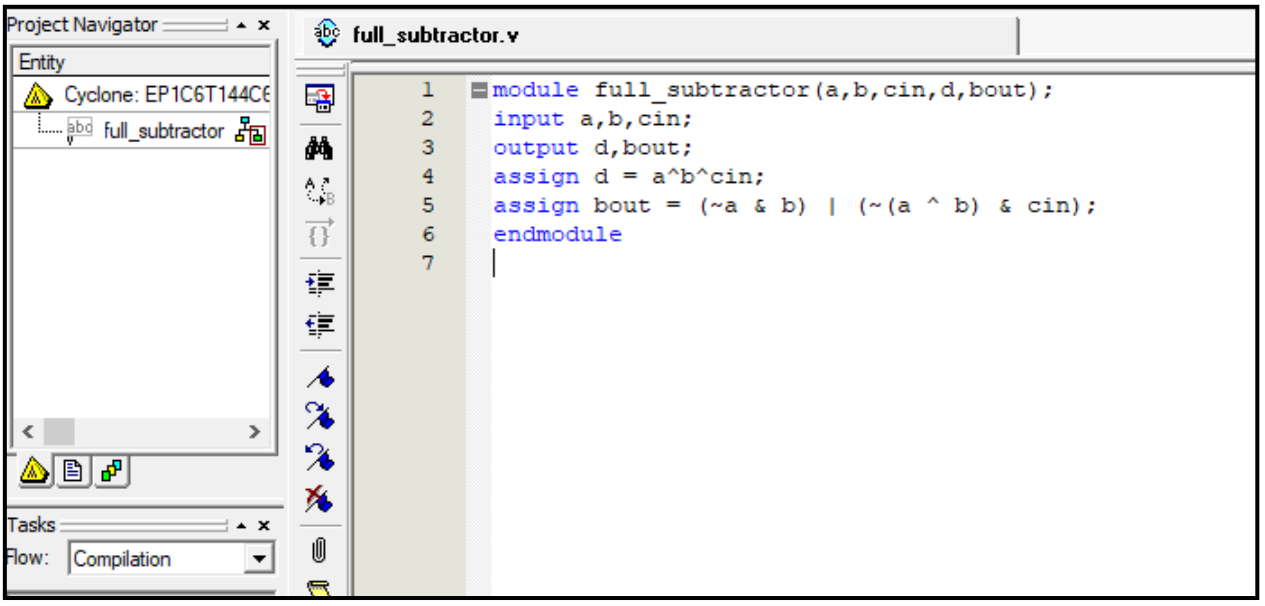
SOP expression from the above truth table

$$D = \bar{A}\bar{B}Bin + \bar{A}B\bar{B}in + A\bar{B}\bar{B}in + ABBin$$
$$= Bin(\bar{A}\bar{B} + AB) + Bin((\bar{A}B + A\bar{B})$$
$$= Bin(\overline{A \oplus B}) + \bar{B}in(A \oplus B)$$

$$Bout = \bar{A}\bar{B}Bin + \bar{A}B\bar{B}in + \bar{A}BBin + ABBin$$
$$= \bar{A}B(\bar{B}in + Bin) + Bin(\bar{A}\bar{B} + AB)$$
$$= \bar{A}B + Bin(\overline{A \oplus B})$$

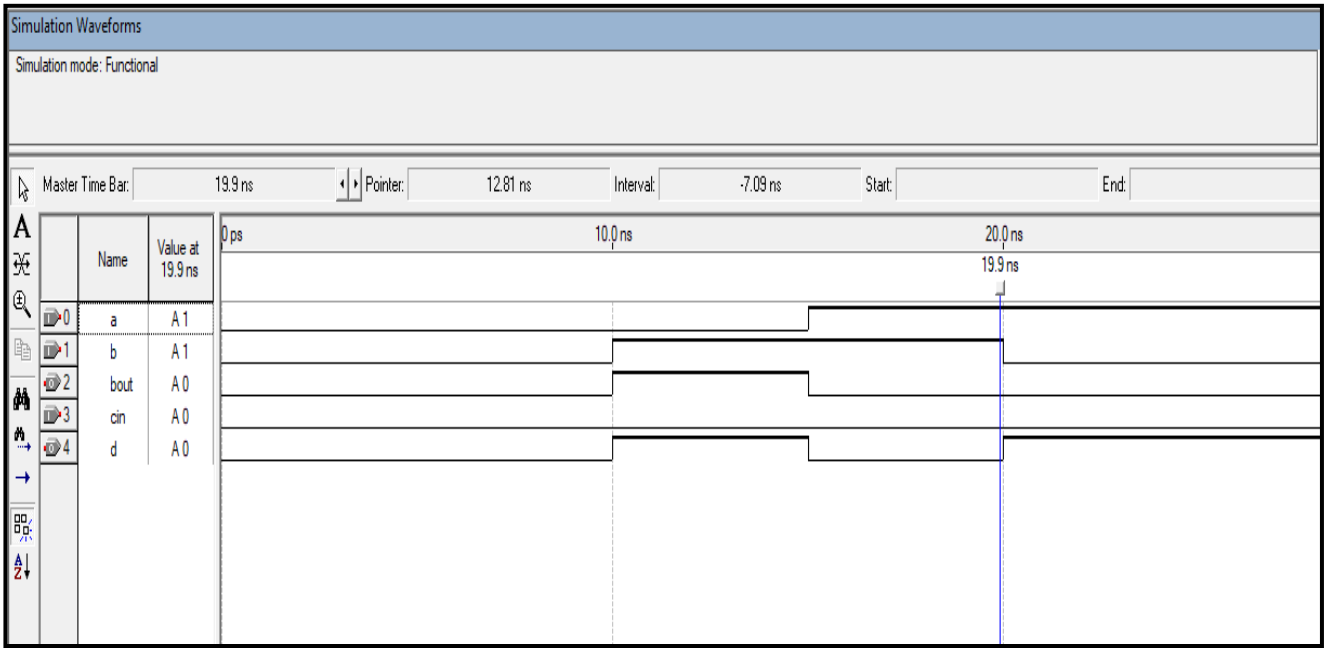
VERILOG CODE:

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



Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM

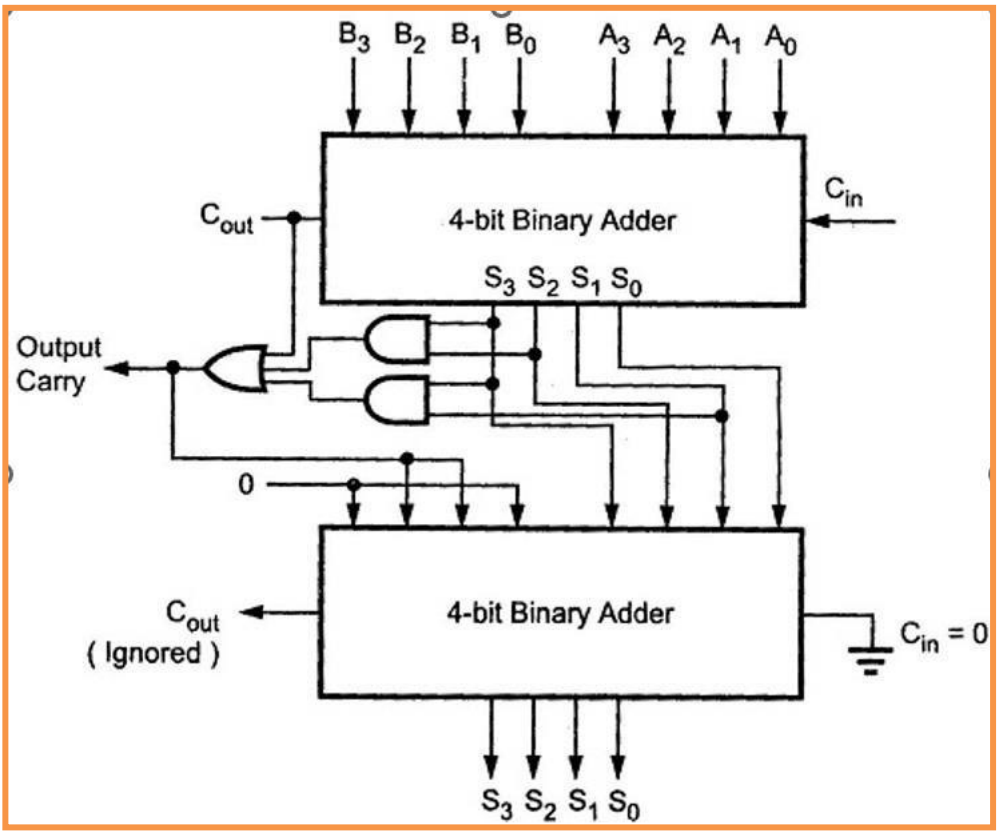


Output of Full Subtractor

RESULT: The truth table of half /full adder and half/full subtractor are verified and simulated using basic gates.

Experiment No.5:

AIM: Design Verilog HDL to implement Decimal Adder.



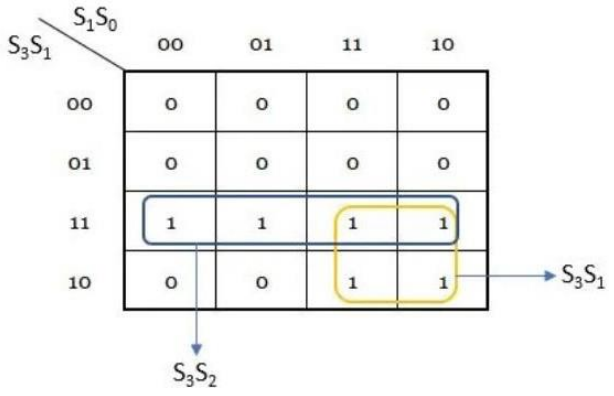
- A BCD Adder adds two BCD digits and produces a BCD digit, A BCD cannot be greater than 9.
- The two given BCD numbers are to be added using the rules of binary addition.
- If the sum is less than or equal to 9 and carry = 0, then no correction is necessary. The sum is correct and in the true BCD form.
- But if the sum is invalid BCD or carry = 1, then the result is wrong and needs correction.
- The wrong result can be corrected by adding six (0110) to it.

From the above point which we have discussed, we understand that the 4 bit BCD adder should consist of the following blocks.

1. A 4 bit binary adder to add the given numbers A and B.
2. A combinational circuit to check if the sum is greater than 9 or carry = 1.
3. One or more 4 bit binary adder to add six (0110) to the incorrect sum if sum > 9 or carry1.

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



From the above K-map, we can write the Boolean expression as:

$$\mathbf{Y = S_3S_2 + S_3S_1}$$

Case 1: Sum <= 9 and Carry = 0

- The output of combinational circuit Y' = 0. Hence $B_3B_2B_1B_0$ = 0000 for adder-2.
- Hence the output of adder-2 is the same as that of adder-2.

Case 2: Sum > 9 and Carry = 0

- If $S_3S_2S_1S_0$ of adder-1 is greater than 9, then output Y' of the combinational circuit becomes 1.
- Therefore $B_3B_2B_1B_0$ = 0110 (adder-2)
- hence six (0110) will be added to the sum output of adder-1.
- We get the corrected BCD result at the output of adder-2

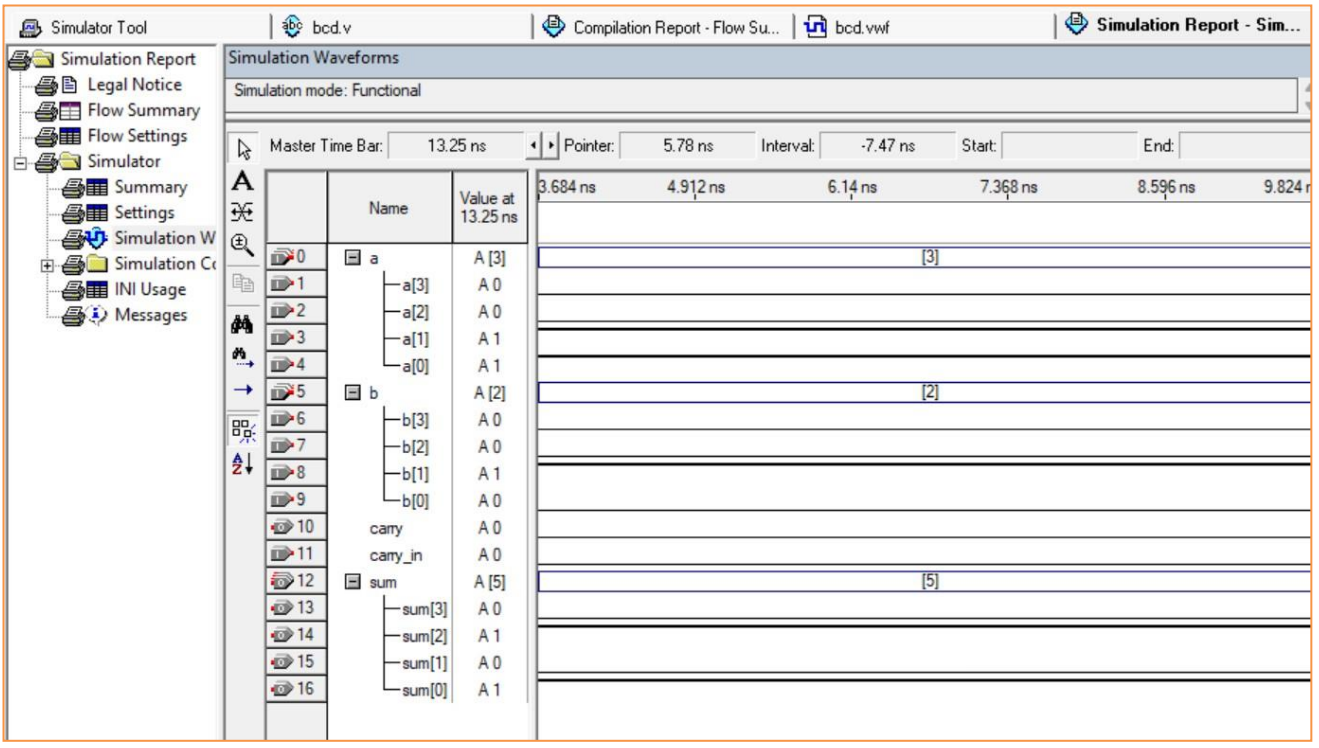
Case 3: Sum <= 9 and Carry = 1

- As the carry output of adder-1 is high, Y' = 1.
- Therefore $B_3B_2B_1B_0$ = 0110 (adder-2)
- So, 0110 will be added to the sum output of adder-1.

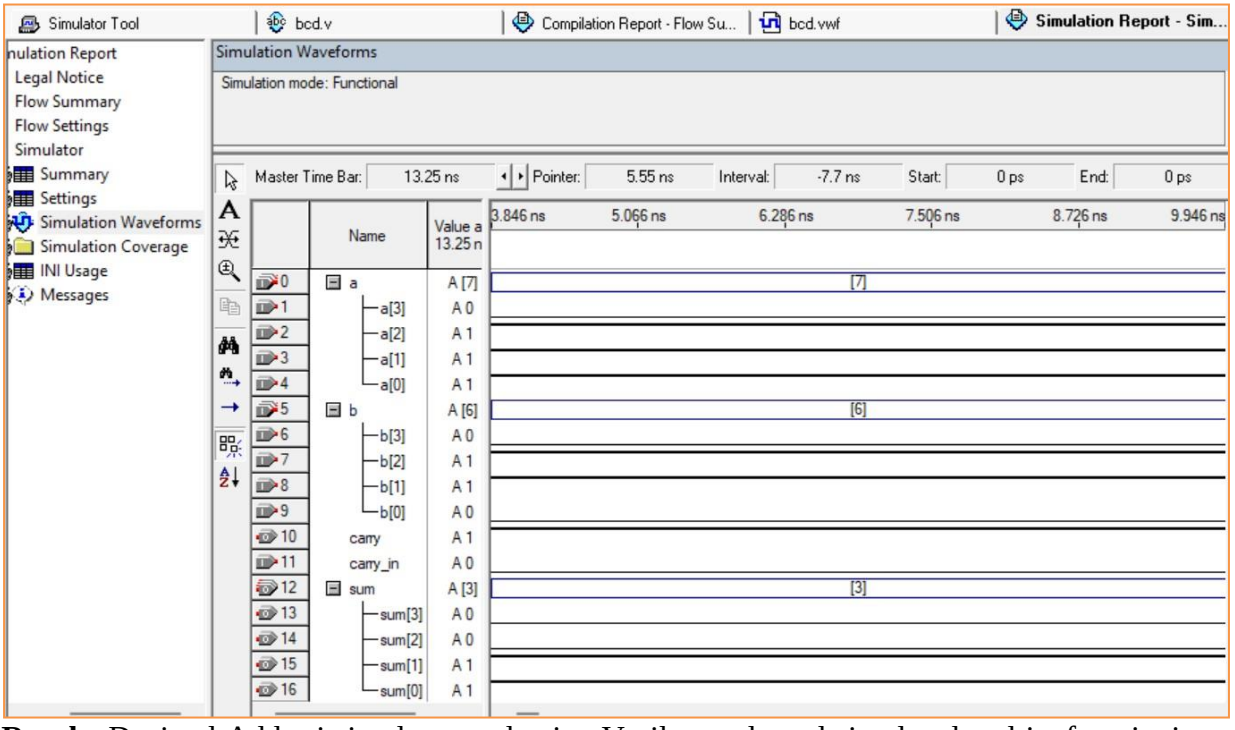
Verilog Code

```
module bcd(a,b,carry_in,sum,carry);

//declare the inputs and outputs of the module with their sizes.
   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 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 result is more than 9.
         carry = 1;  //set the carry output
         sum = sum_temp[3:0];    end
      else    begin
         carry = 0;
         sum = sum_temp[3:0];
      end
   end

endmodule
```

Simulation results for the case where there is no requirement of adding "6" as the result is less than or equal to 9 after adding a[3:0], b[3:0] and carry_in.



Simulation results for the case where there is a requirement of adding "6" as the result is greater than 9 after adding a[3:0], b[3:0] and carry_in.



Result: Decimal Adder is implemented using Verilog code and simulated and its functioning correctly for all possible values.

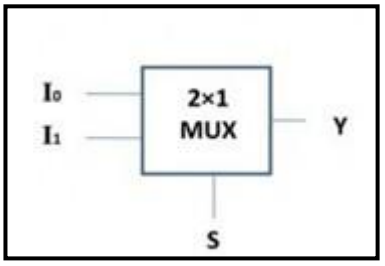
EXPERIMENT: 6

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

DESCRIPTION:

Multiplex means many to one. Digital multiplexers provide the digital equivalent of an analog selector switch. A digital multiplexer connects one of 'n' inputs to a single output line, so that the logical value of the input selected is transferred to the output. One of the 'n' input selection is determined by 'm' select lines where $n=2^m$.Thus a 4:1 mux requires 2 select lines. Two output levels exists, active high output Y and active low output.

a) MUX 2:1

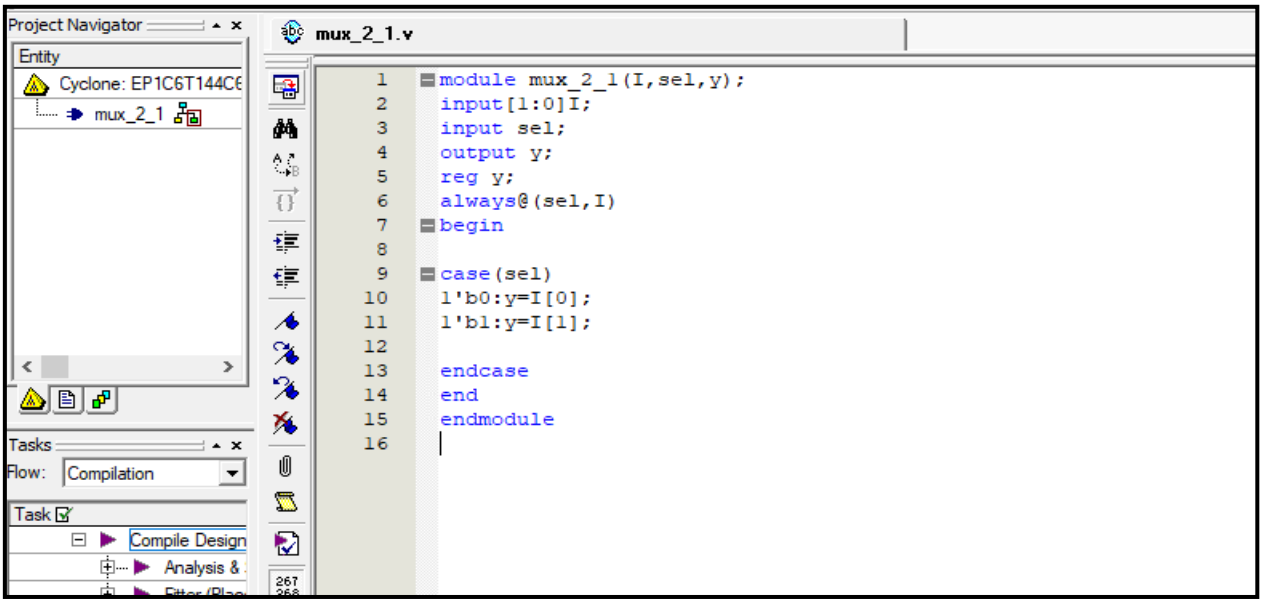


Truth table

S	Y
0	I_0
1	I_1

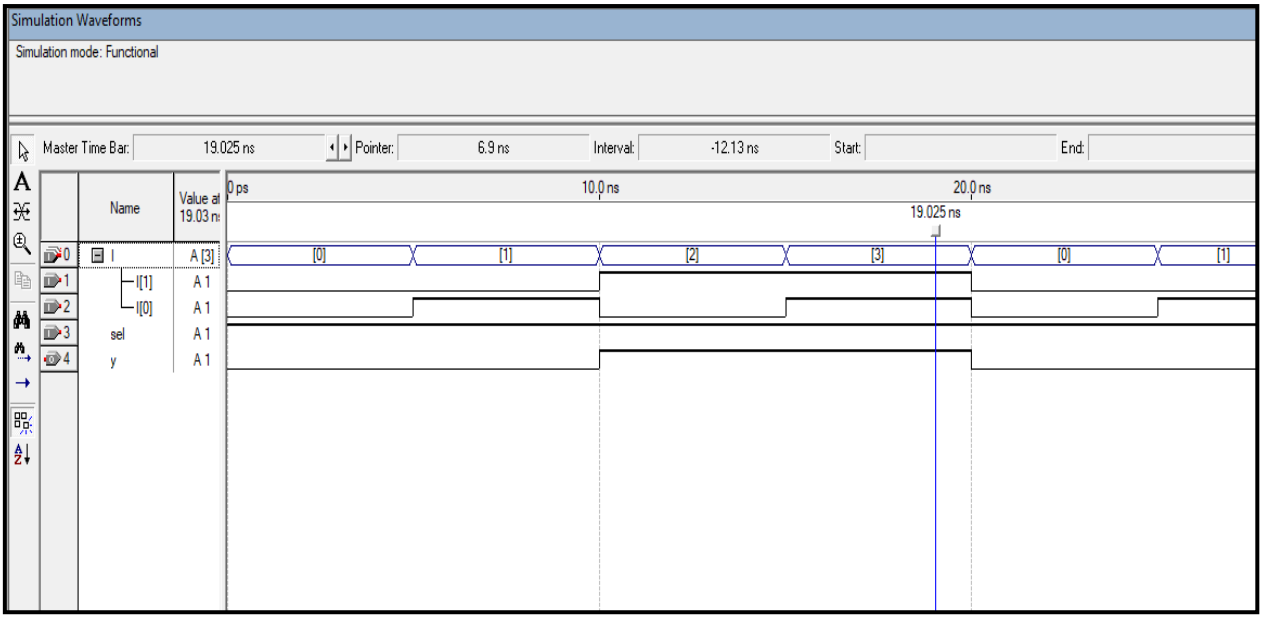
VERILOG CODE:

```
module mux_2_1( I, sel, y);
input [1:0] I ;
input sel ;
output y ;
reg y;
always@ (sel , I)
    begin

case (sel)
1'b0: y = I [0]  ;
1'b1: y = I [1]  ;
    end
    cae
    end
    end
    mo
    dul
    e
```



Executed Verilog code in Quartus Tool-II

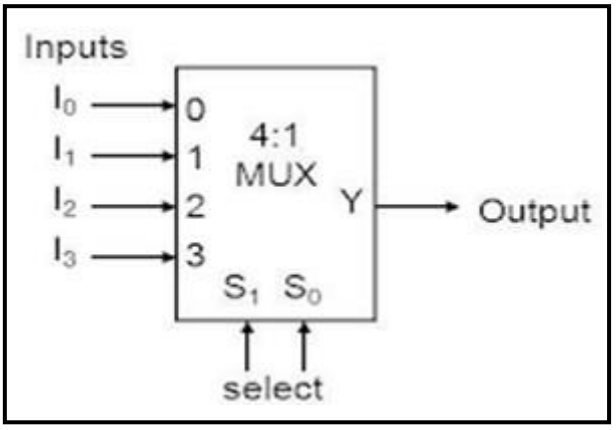
SIMULATION WAVEFORM



Output of 2:1 mux

b) MUX 4:1

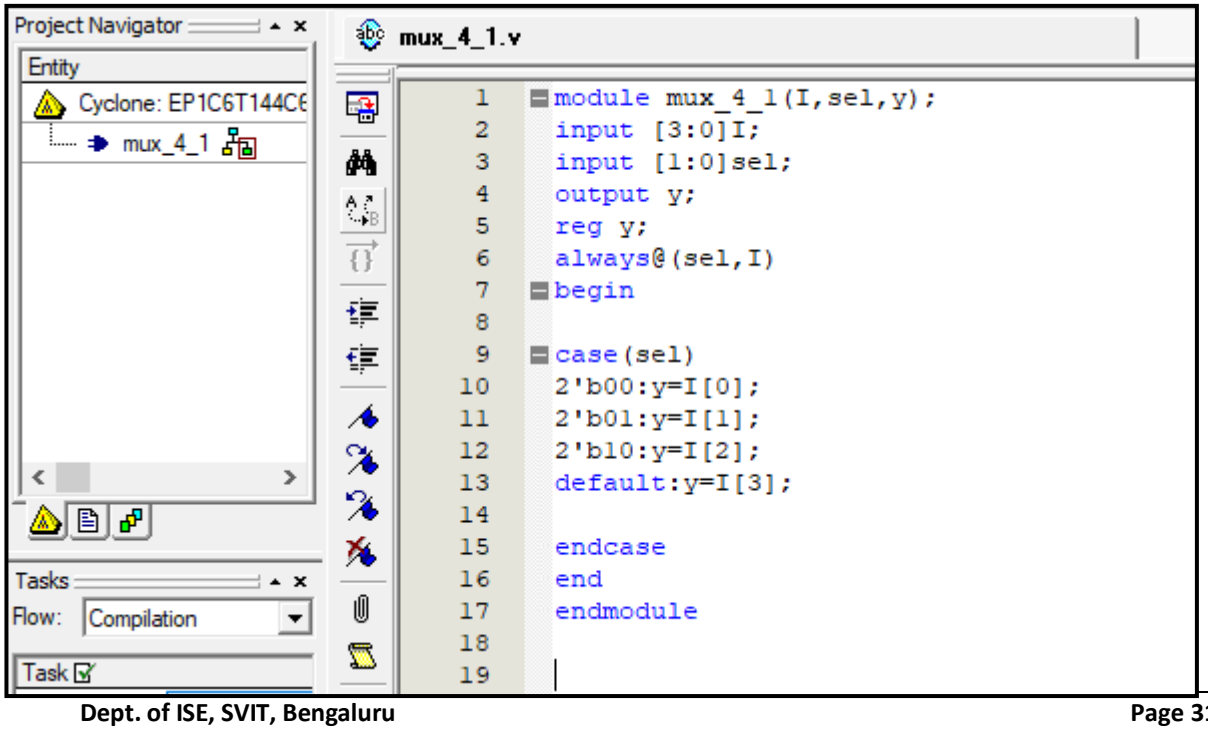
Truth table



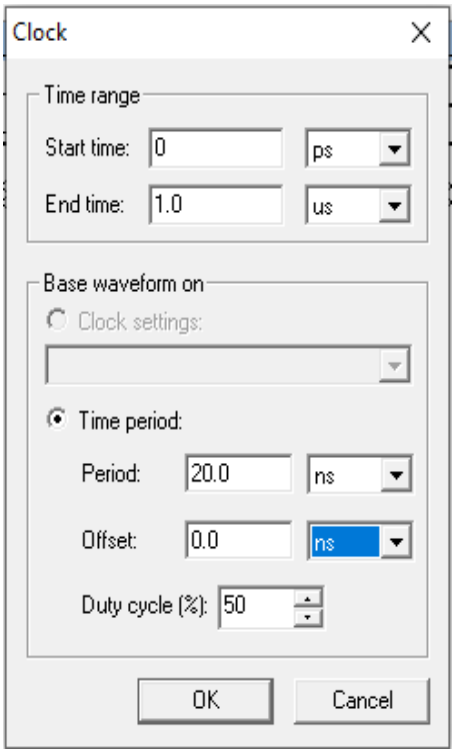
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

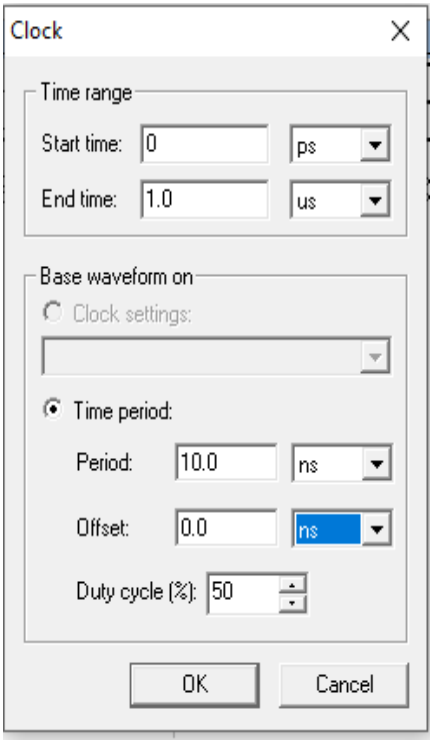
VERILOG CODE:

```
module mux_4_1 (I, sel, y);
input [3:0] I;
input  [1:0] sel;
output y;
reg y;
always@ (sel, I)
    begin

case  (sel)
2'b00: y = I [0];
2'b01: y = I [1];
2'b10: y = I [2];
default: y = I [3];
    end
    cas
    e
    end
    end
    mo
    dul
    e
```



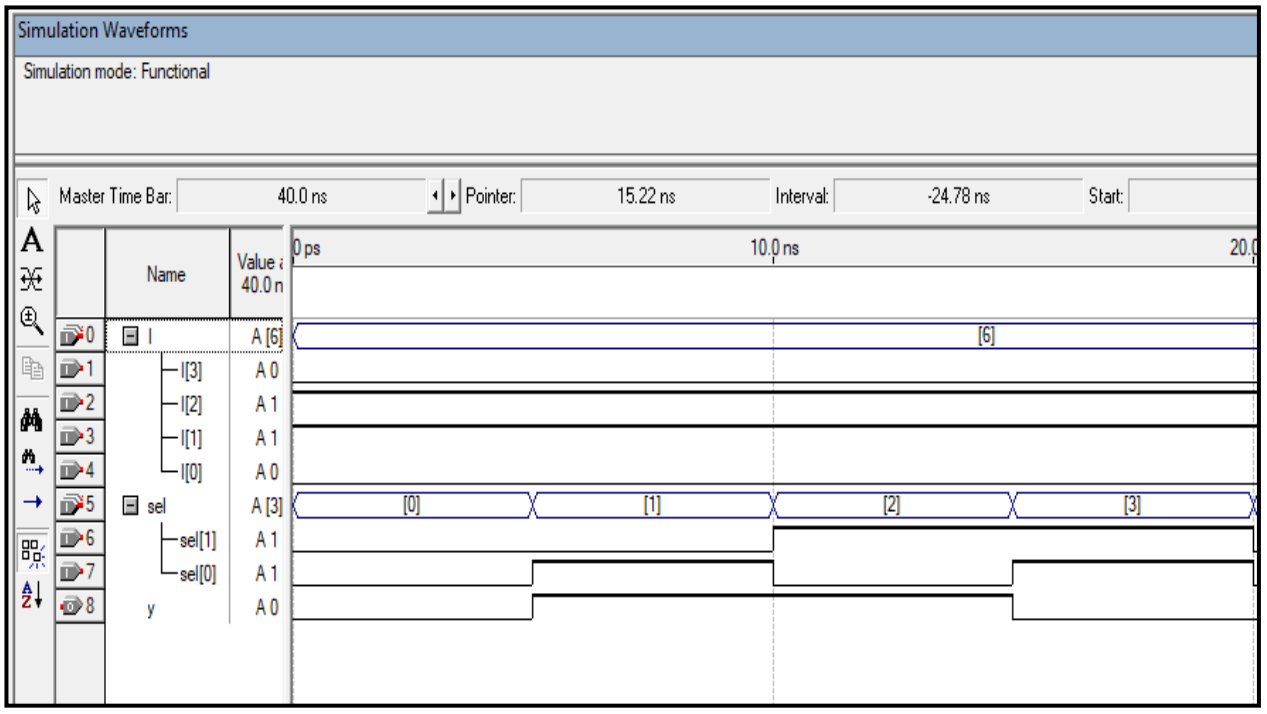
Executed Verilog code in Quartus Tool-II





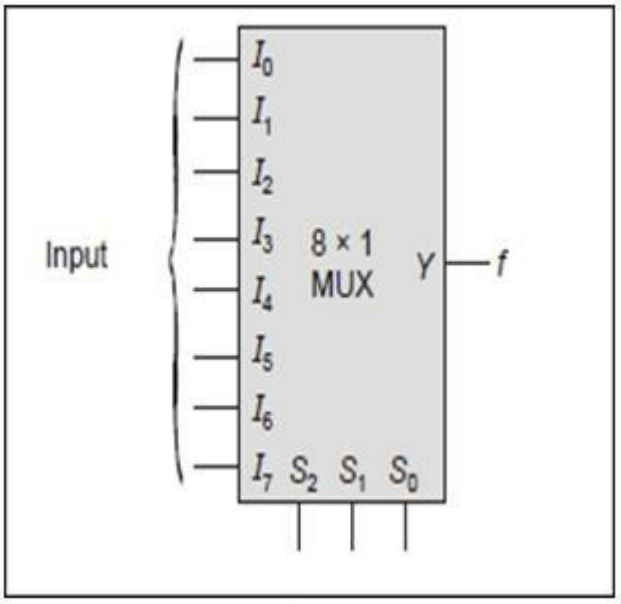
Settings of select lines S1 and S0

SIMULATION WAVEFORM



Output of 4:1 mux

c) MUX 8:1



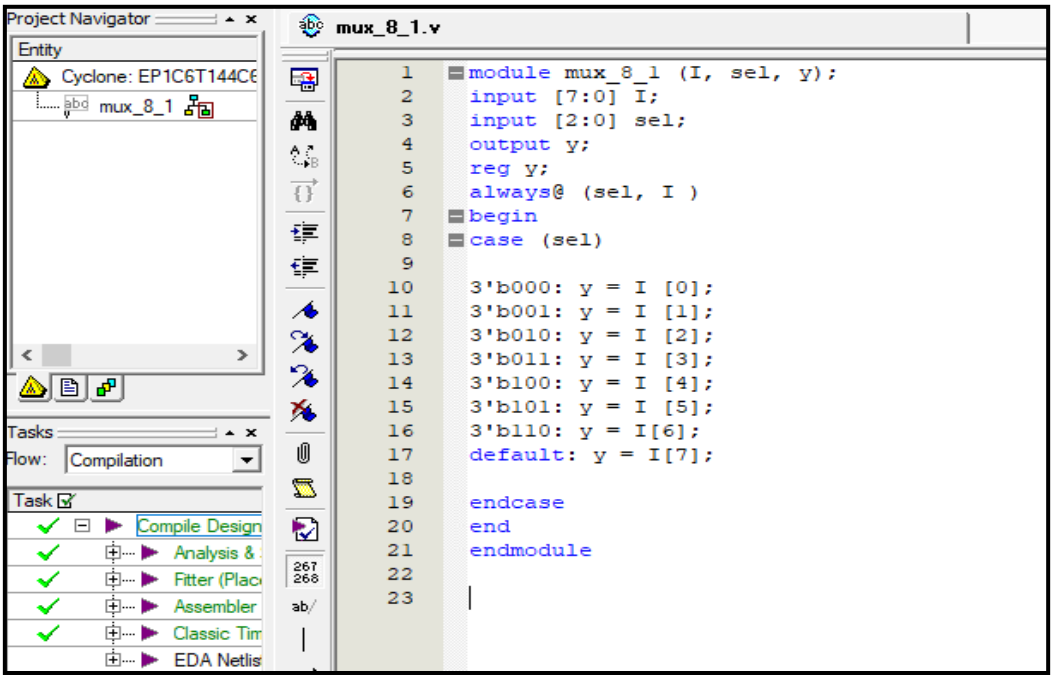
Truth table

S_2	S_1	S_0	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

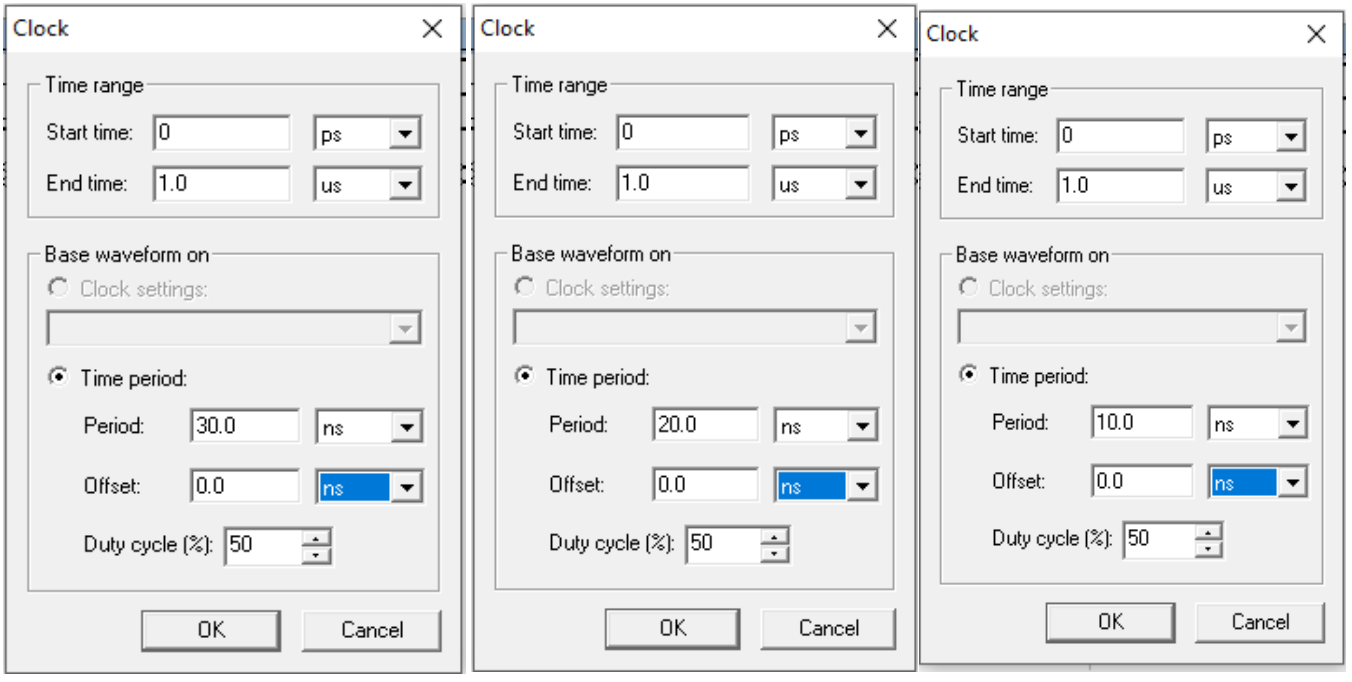
VERILOG CODE:

```
module mux_8_1 (I, sel, y);
input [7:0] I;
input [2:0] sel;
output y;
reg y;
always@ (sel, I )
    be
    gi
    n
    ca
    se
    (s
    el)

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];
default: y = I [7];
```

```
end
cas
e
end
end
mo
dul
e
```

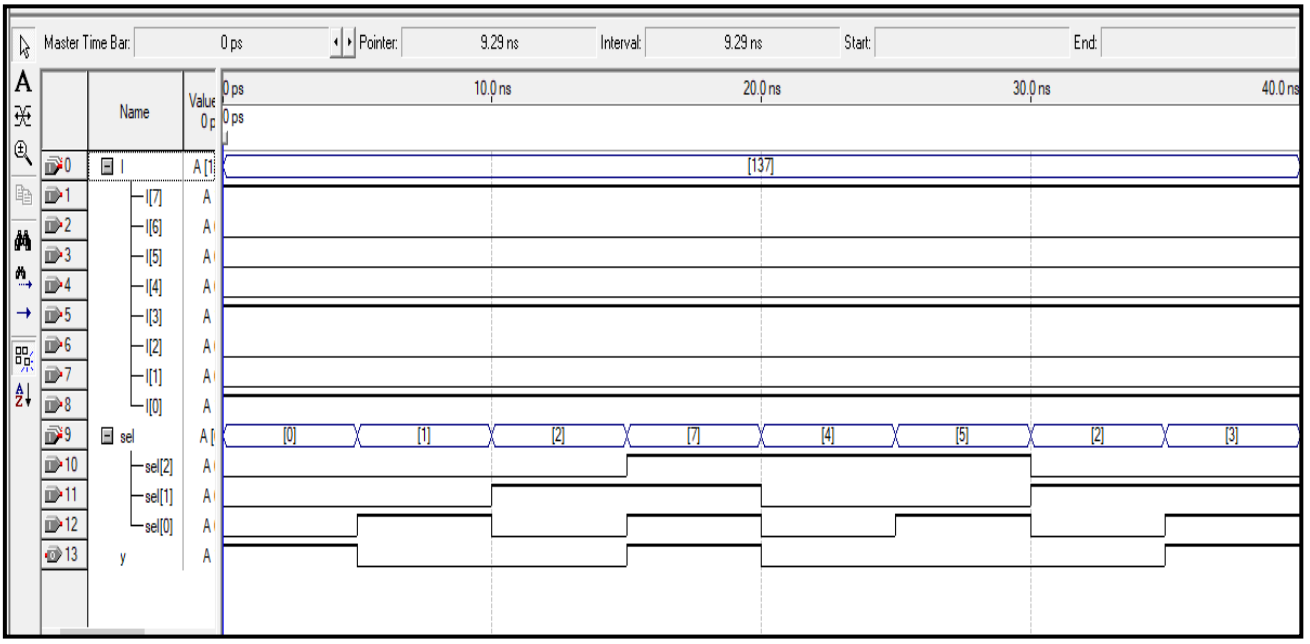


Executed Verilog code in Quartus Tool-II



Settings of select lines S2 , S1 and S0

SIMULATION WAVEFORM



Output of 8:1 mux

RESULT: The truth table of different types of mux 2:1, 4:1 and 8:1 are verified and simulated.

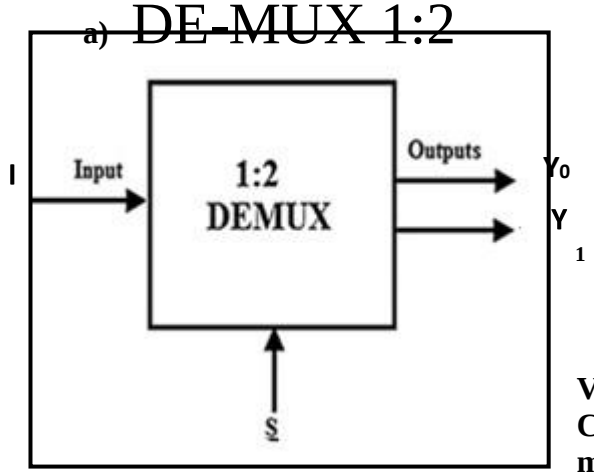
EXPERIMENT: 7

AIM: Design Verilog program to implement types of De-multiplexer.

DESCRIPTION:

Demultiplex means one to many. Demultiplexing is a process of taking information from one input and transmitting over one of several outputs. Demultiplexer is a circuit that performs reverse operation of a multiplexer. A demux has 1 input line and 2^n output lines, where n represents the number of select lines.

a) DE-MUX 1:2

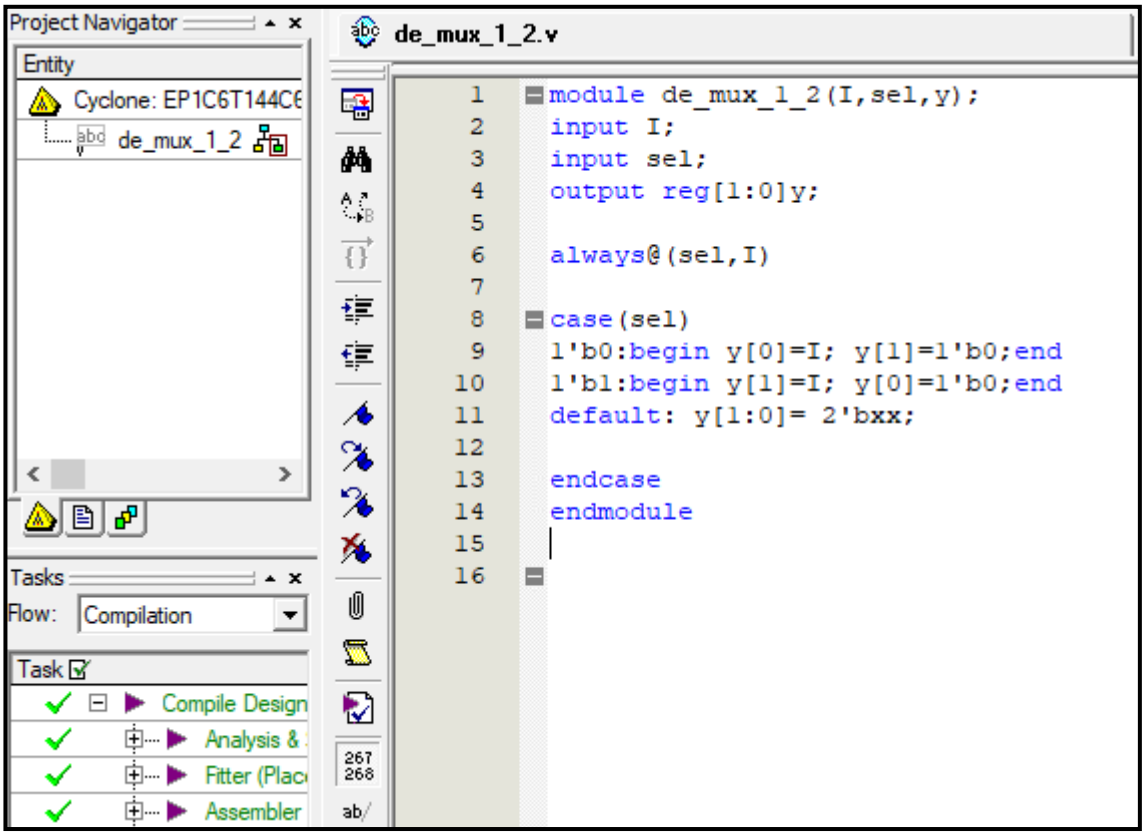


Truth table

I	S	Y_0	Y_1
I	0	I	0
I	1	0	I

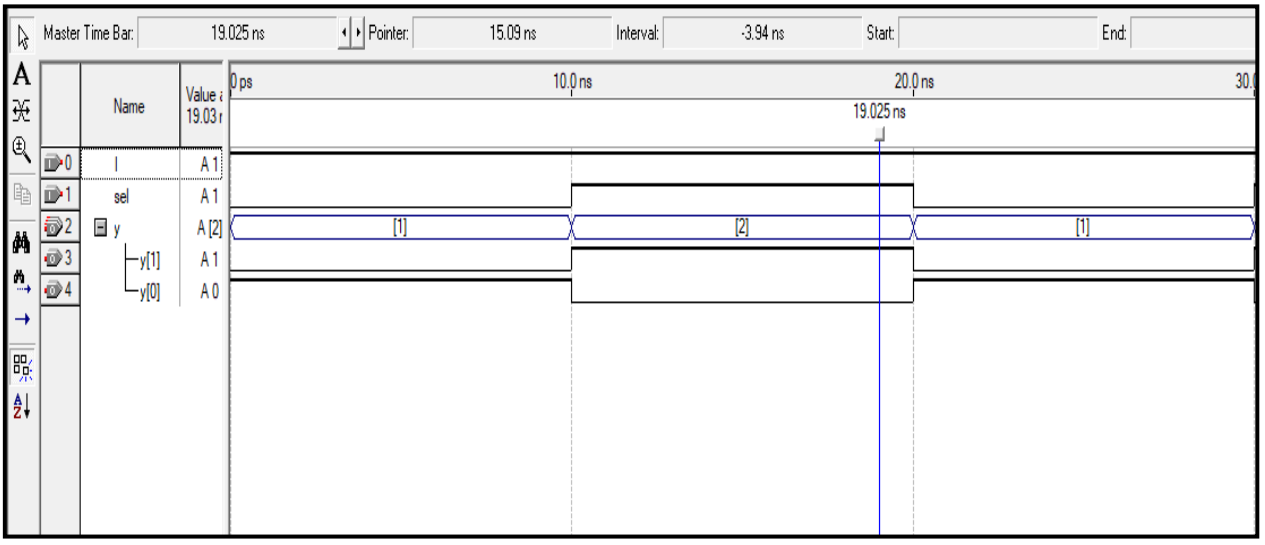
VERILOG CODE:

```
module de_mux_1_2 (I, sel, y);
input I;
input sel;
output reg [1:0] y;
always@ (sel, I)
begin
case (sel)
1'b0: begin y [0] = I;  y [1] = 1'b0; end
1'b1: begin  y [0] = 1'b0 ; y [1] = I;  end
end
cas
e
end
end
mo
dul
e
```



Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM



Output of 1:2 de-mux

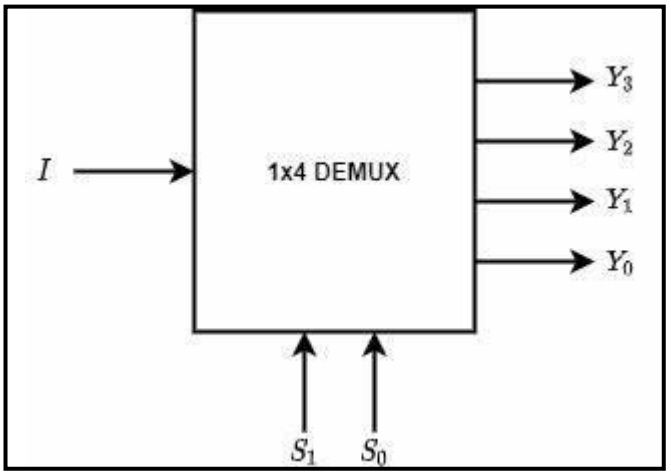
b) DE-MUX 1:4

Truth table

Input	Select lines		Output lines			
I	S_1	S_0	Y_0	Y_1	Y_2	Y_3

I	0	0	I	0	0	0
I	0	1	0	I	0	0

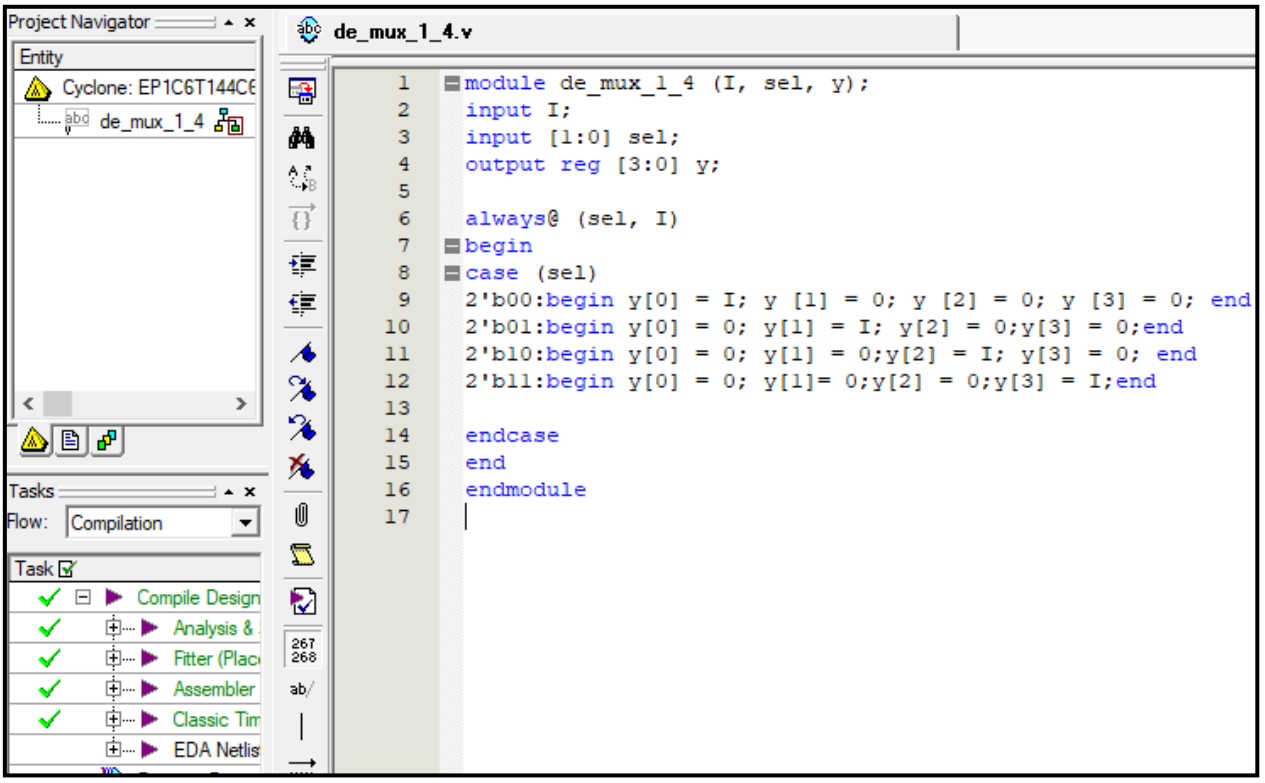
I	1	0	0	0	I	0
I	1	1	0	0	0	I



VERILOG CODE:

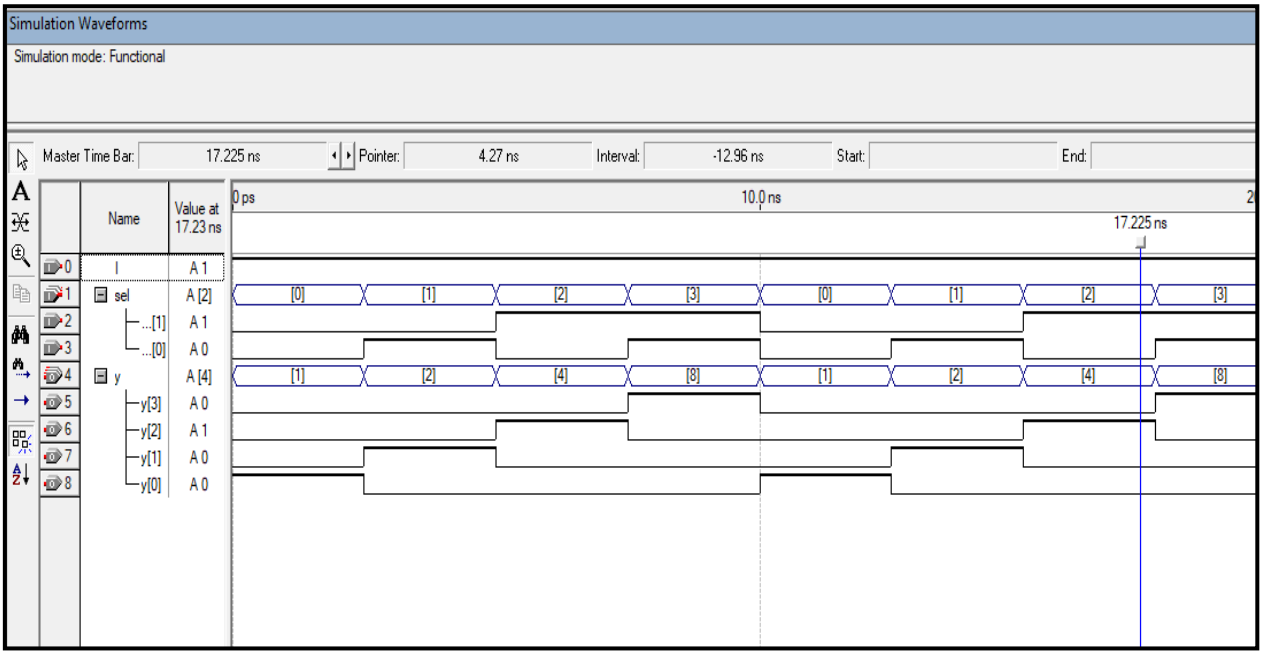
```
module de_mux_1_4 (I, sel, y);
input I;
input [1:0] sel;
output reg [3:0] y;

always@ (sel, I)
begin
case (sel)
2'b00: begin y [0] = I; y [1] = 0; y [2] = 0; y [3] = 0; end
2'b01: begin y [0] = 0; y [1] = I; y [2] = 0; y [3] = 0; end
2'b10: begin y [0] = 0; y [1] = 0; y [2] = I; y [3] = 0; end
2'b11: begin y [0] = 0; y [1] = 0; y [2] = 0; y [3] = I; end
```

```
end
cas
e
end
end
mo
dul
e
```



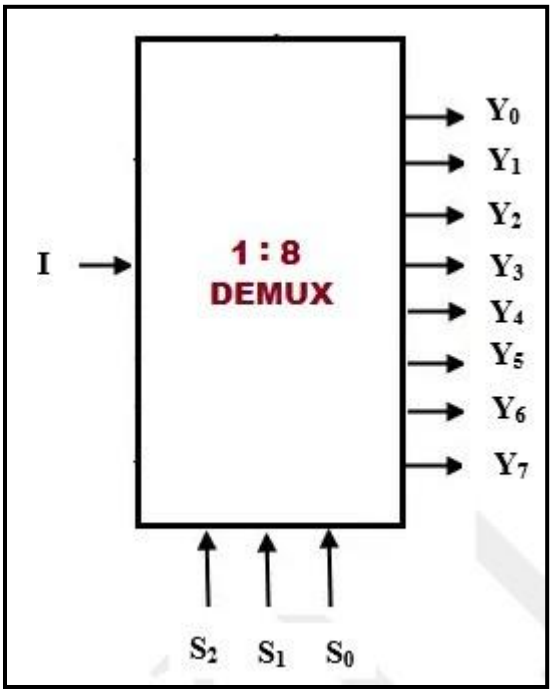
Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM



Output of 1:4 de-mux

c) DE-MUX 1:8



Truth table

Input	Select lines			Output lines							
I	**S_2**	**S_1**	**S_0**	**Y_0**	**Y_1**	**Y_2**	**Y_3**	**Y_4**	**Y_5**	**Y_6**	**Y_7**
I	0	0	0	I	0	0	0	0	0	0	0
I	0	0	1	0	I	0	0	0	0	0	0
I	0	1	0	0	0	I	0	0	0	0	0
I	0	1	1	0	0	0	I	0	0	0	0
I	1	0	0	0	0	0	0	I	0	0	0
I	1	0	1	0	0	0	0	0	I	0	0
I	1	1	0	0	0	0	0	0	0	I	0
I	1	1	1	0	0	0	0	0	0	0	I

VERILOG CODE:

```
module de_mux_1_8 (I, sel, y);
input I;
input [2:0] sel;
output reg [7:0] y;
always@(sel ,I)

case (sel)
3'b000:begin y[0] = I; y[1] = 0 ;y[2] = 0;y[3] = 0 ;y[4] = 0;y[5] = 0;y[6] = 0;y[7] = 0;end
3'b001:begin y[0]  = 0; y[1] = I; y[2] = 0;y[3] = 0;y[4] = 0;y[5] = 0;y[6] = 0;y[7] = 0;end
3'b010:begin y[0] = 0; y[1] = 0;y[2] = I; y[3] = 0;y[4] = 0;  y[5] = 0;y[6] = 0;y[7] = 0;end
```

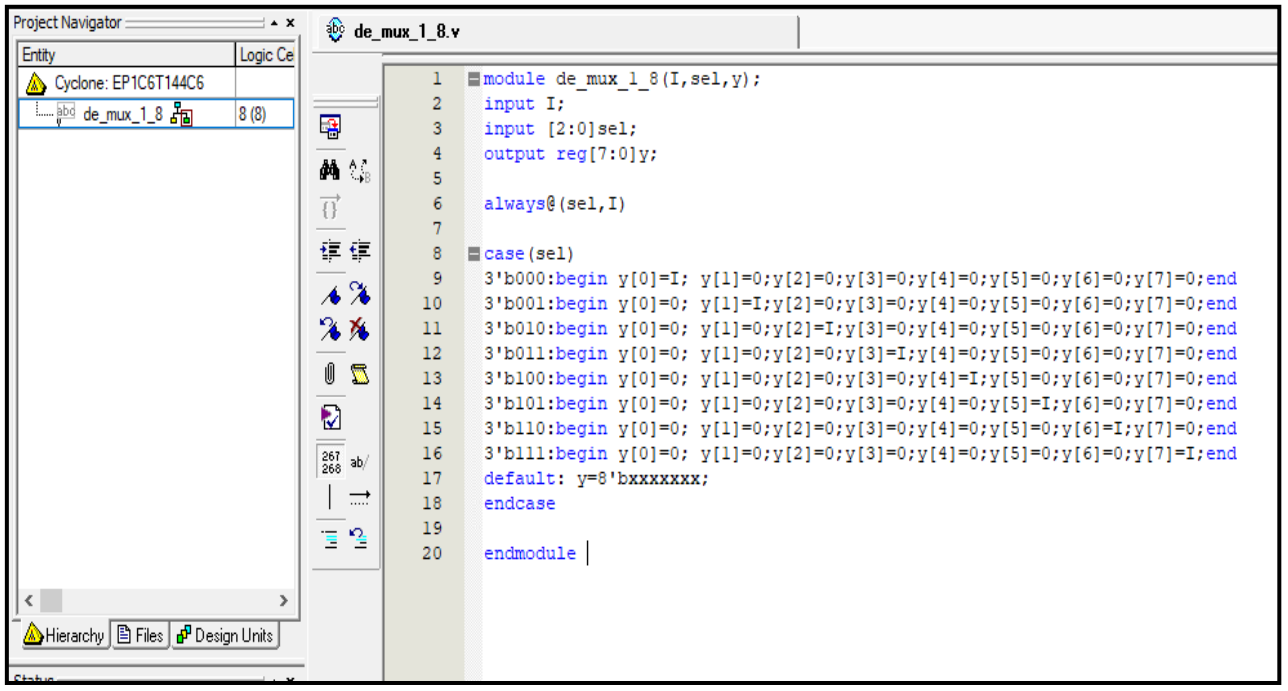
3'b011:begin y[0] = 0;y [1] = 0; y[2] = 0;y[3] = I ;y[4]= 0;y[5] = 0 ;y[6] = 0;y[7] = 0;

end

3'b100:begin y[0] = 0; y[1] = 0;y[2] = 0;y[3] = 0;y[4] = I; y[5] = 0;y[6] = 0;y[7] = 0; **end**
3'b101:begin y[0] = 0; y[1] = 0;y[2] = 0;y[3] = 0;y[4] = 0;y[5] = I; y[6] = 0;y[7] = 0; **end**
3'b110:begin y[0] = 0; y[1] = 0;y[2] = 0;y[3] = 0;y[4] = 0;y[5] = 0;y[6] = I; y[7] = 0; **end**
3'b111:begin y[0] = 0; y[1] = 0;y[2] = 0;y[3] = 0;y[4] = 0;y[5] = 0;y[6] = 0;y[7] = I; **end**

default: y
=
8'bxxxxxxx
x; endcase

endmodule



Executed Verilog code in Quartus Tool-II

SIMULATION WAVEFORM

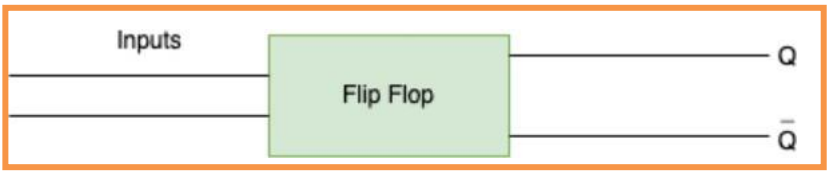
Output of 1:8 de-mux

RESULT: The truth table of different types of de-mux 2:1, 4:1 and 8:1 are verified and simulated.

Experiment 8:

AIM: Design Verilog Program for implementing various types of Flip Flops such as SR, JK and D

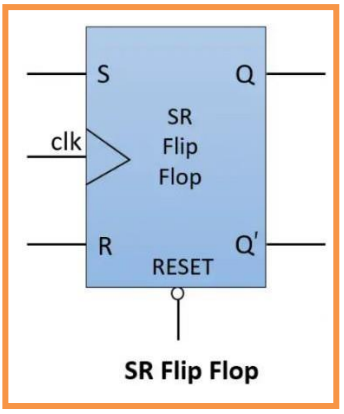
Flip flop is a term which comes under digital electronics, and it is an electronic component which is used to store one single bit of the information.



SR Flip Flop:

The SR flip flop has two inputs SET 'S' and RESET 'R'. As the name suggests, when S = 1, output Q becomes 1, and when R = 1, output Q becomes 0. The output Q' is the complement of Q.

Block Diagram:



Truth Table:

S	R	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	x

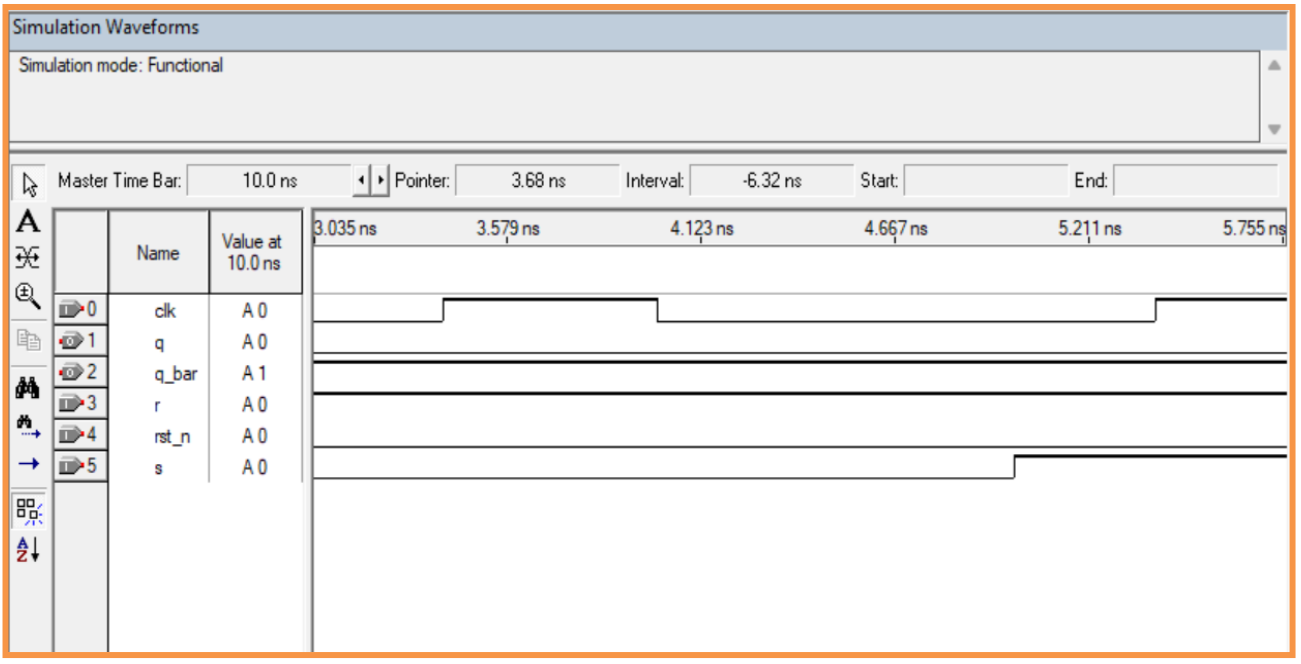
Verilog Code:

```
module sr(
  input clk, rst_n,
  input  s,r,
  output reg q,
  output q_bar
  );
```

```
// always@(posedge clk or negedge rst_n) // for asynchronous reset
always@(posedge clk) begin // for synchronous reset
  if(!rst_n) q <= 0;
```

```
    else begin
      case({s,r})
        2'b00: q <= q;    // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= 1'bx; // Invalid inputs
      endcase
    end
  end
  assign q_bar = ~q;
endmodule
```

Simulation Results:

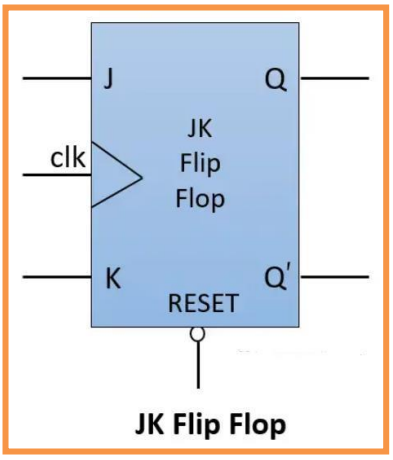


JK Flip Flop:

The JK flip flop has two inputs ‘J’ and ‘K’. It behaves the same as SR flip flop except that it eliminates undefined output state (Q = x for S=1, R=1)

For J=1, K=1, output Q toggles from its previous output state.

Block Diagram:



JK Flip Flop

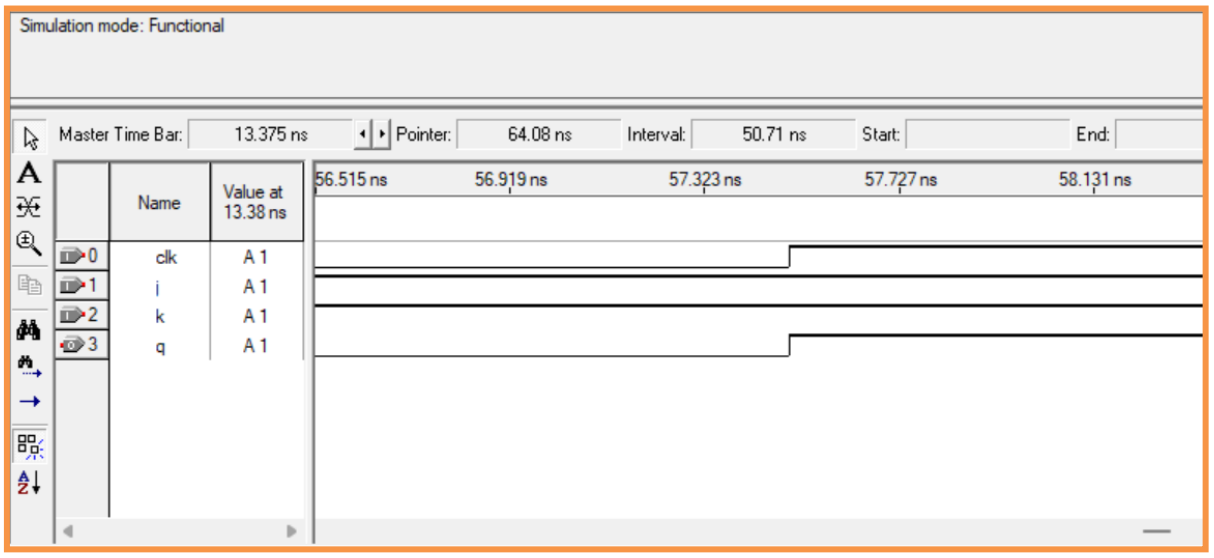
Truth Table:

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

Verilog code:

```
module jk( input j, input k, input clk, output reg q);


  always @ (posedge clk)
    case ({j,k})
      2'b00 :  q <= q;
      2'b01 :  q <= 0;
      2'b10 :  q <= 1;
      2'b11 : q <= ~q;
    endcase
endmodule
```

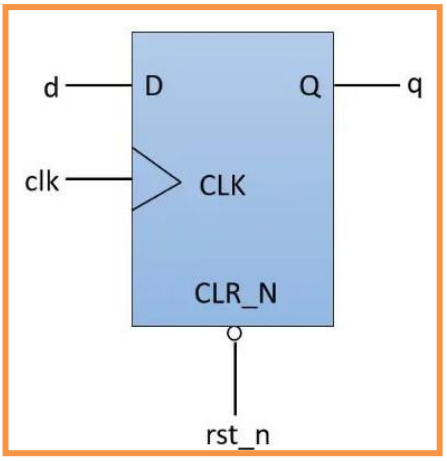
Simulation Results:



D Flip Flop:

The D flip flop is a basic sequential element that has data input 'd' being driven to output 'q' as per clock edge. Also, the D flip-flop held the output value till the next clock cycle. Hence, it is called an edge-triggered memory element that stores a single bit.

Block Diagram:

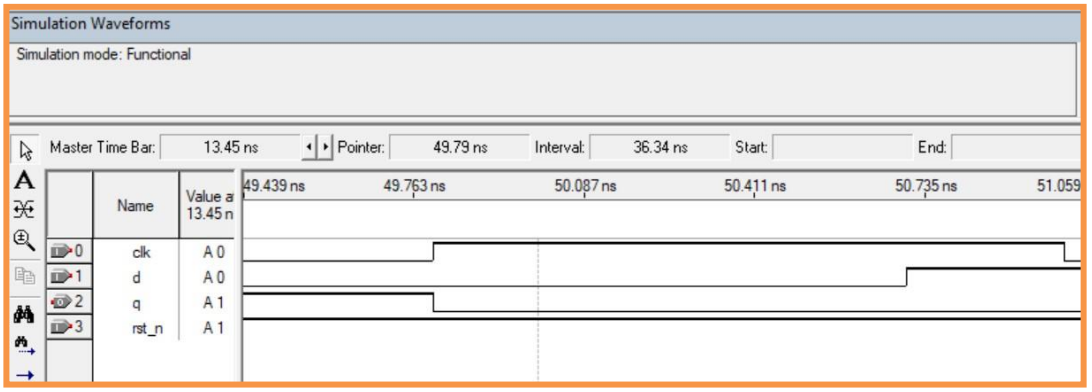


Verilog code:

```
module dd
 (
 input clk, rst_n,
 input d,
 output reg q
 );

 always@(posedge clk or negedge rst_n) begin
  if(!rst_n) q <= 0;
```

```
  else      q <= d;
end

endmodule
```

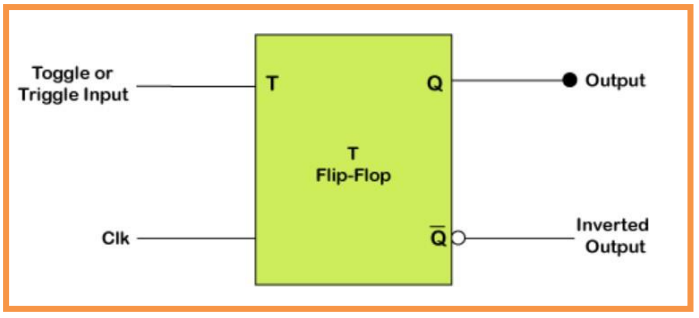
Simulation results:



T Flipflop (Content beyond syllabus)

The T flip flop has single input as a 'T'. Whenever input T=1, the output toggles from its previous state else the output remains the same as its previous state.

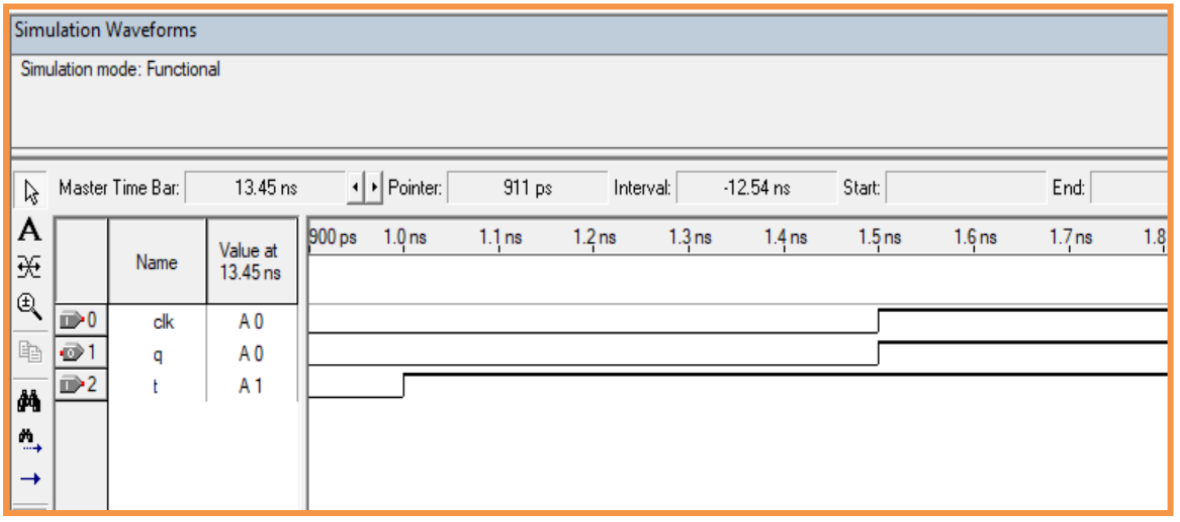
Block Diagram:



Verilog code:

```
module t_ff(clk,t,q);
input clk,t;
output reg q;

always @ (posedge clk)begin
   if(t == 0)
      q <= q;
   else
      q = ~q;
end

endmodule
```

Simulation results:



Result: SR, JK , D & T Flip Flop is implemented using Verilog code and simulated.

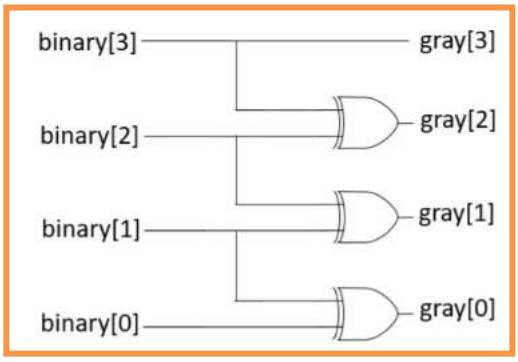
Binary to gray code converter (Content beyond syllabus)

Gray Code is similar to binary code except its successive number differs by only a single bit. Hence, it has importance in communication systems as it minimizes error occurrence. They are also useful in rotary, optical encoders, data acquisition systems, etc.

Truth table:

Decimal Number	4-bit Binary Code ($A_3A_2A_1A_0$)	4-bit Gray Code ($G_3G_2G_1G_0$)
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Circuit Diagram:



Verilog Code:

```
module bg
    (input [3:0] bin, //binary input output
     [3:0] G //gray code output
    );

//xor gates.
assign G[3] = bin[3];
assign G[2] = bin[3] ^ bin[2];
assign G[1] = bin[2] ^ bin[1];
assign G[0] = bin[1] ^ bin[0];

endmodule
```

Simulation Results: