# PART B

**Experiment 9. Realization of Logic Gates and Familiarization of Verilog**
(a) Familiarization of the basic syntax of Verilog
(b) Development of Verilog modules for basic gates and to verify truth tables.
(c) Design and simulate the HDL code to realize three and four-variable Boolean functions

**Experiment 10: Half adder and full adder**
(a) Development of Verilog modules for half adder in 3 modeling styles (dataflow/ structuralbehaviorall).
(b) Development of Verilog modules for full adder in structural modeling using half adder.

**Experiment 11: Mux and Demux in Verilog**
(a) Development of Verilog modules for a 4x1 MUX.
(b) Development of Verilog modules for a 1x4 DEMUX.

**Experiment 12: Flipflops and shiftregisters**
(a) Development of Verilog modules for SR, JK, T and D flip flops.
(b) Development of Verilog modules for a Johnson/Ring counter

## EXPERIMENT 9
## REALIZATION OF LOGIC GATES AND FAMILIARIZATION OF VERILOG

**AIM OF THE EXPERIMENT:**

a) To familiarize the basic syntax of Verilog
b) To develop Verilog modules for basic gates and to verify truth tables.
c) To design and simulate the HDL code to realize three and four-variable Boolean functions

**LEARNING OBJECTIVE:**

After completing this experiment, the student will be able to:
- Implement basic gates using Verilog modules

**BRIEF DESCRIPTION:**

**<u>INTRODUCTION IN VERILOG HDL</u>**

Verilog is a hardware description language (HDL) used to model electronic systems. The language supports the design, verification, and implementation of analog, digital, and mixed-signal circuits at various levels of abstraction. The language is case- sensitive, has a preprocessor like C, and the major control flow keywords, such as "if" and "while", are similar. The formatting mechanism in the printing routines and language operators and their precedence are also similar. The language differs in some fundamental ways. Verilog uses Begin/End instead of curly braces to define a block of code. The concept of time, so important to an HDL won't be found in C. The language differs from a conventional programming language in that the execution of statements is not strictly sequential.

A Verilog design consists of a hierarchy of modules defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behaviour of the module by defining the relationships between the ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all concurrent statements and all begin/end blocks in the design are executed in parallel, qualifying Verilog as a Dataflow language. A module can also contain one or more instances of another module to define sub-behavior. A subset of statements in the language is synthesizable. If the modules in a design contain a netlist that describes the basic components and connections to be implemented in hardware only synthesizable statements, the software can be used to transform or synthesize the design into the netlist may then be transformed into, for example, a form describing the standard cells of an integrated circuit (e.g., ASIC) or a bit stream for a programmable logic device (e.g., FPGA).

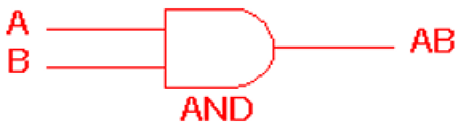
## INTRODUCTION TO EDA PLAYGROUND

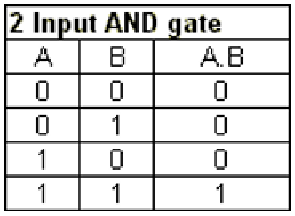
EDA Playground gives engineers immediate hands-on exposure to simulating and synthesizing System Verilog, Verilog, VHDL, C++/System C, and other HDLs.

## STEPS

1. Log into EDA Playground
2. Login: Click the Login button (top right) Then either
   - I. Click on Google or Facebook or
   - II. Register by clicking on ‘Register for a full account’ (which enables all the simulators on EDA Playground)
3. Select “Icarus Verilog 0.9.7” in “Tools & Simulators” and select “Open EPWave after run” in Run Options.
4. In either the Design window pane, type the code, and in the Testbench window pane, type the testbench.
5. Click on ‘Save’ to save the design and ‘Run’ to run the design.
6. Th simulation will run on EDA Playground and load the resulting waves in EPWave

## AND Gate





| 2 Input AND gate | | |
|---|---|---|
| A | B | A.B |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Structural Model**

```
module
and_gate(x,y,z);
  input x,y;
  output z;
  and g1(z,x,y);
endmodule
```

**Data Flow Model**

```
module
and_gate(x,y,z);
  input x,y;
  output z;
  assign z=(x&y);
endmodule
```

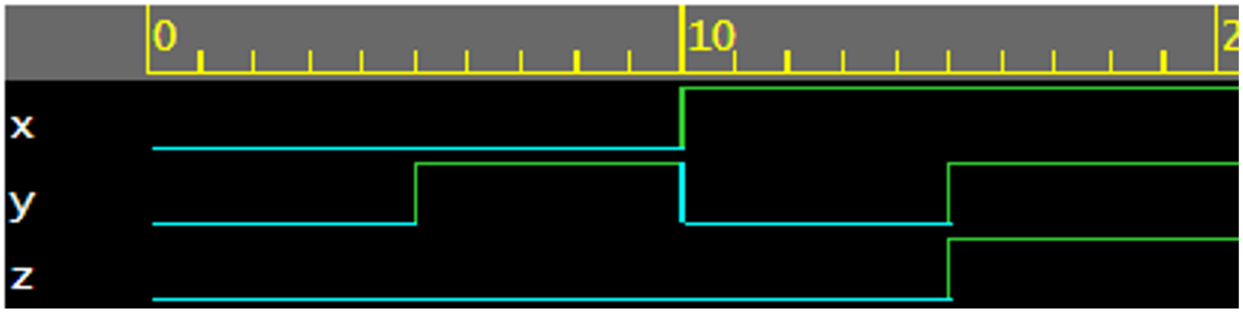
**Behavioural Model**

```
module
and_gate(x,y,z);
  input x,y;
  output z;
  reg z;
  always@(x,y)z=x&y;
endmodule
```

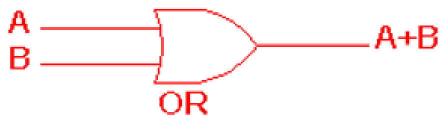
**TESTBENCH:**

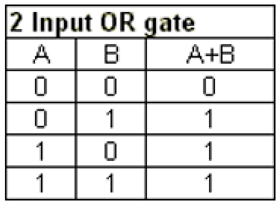
```
module andstr_tb;
        reg x,y;
        wire z;
  and_gate inst(.x(x),
.y(y), .z(z));

initial begin

$dumpfile("dump.vcd");
    $dumpvars;
    #30 $finish;
end

initial
  begin
    $monitor(x,y,z);
        x<=0;  y<=0;  #5;
        x<=0;  y<=1;  #5;
        x<=1;  y<=0;  #5;
        x<=1;  y<=1;  #5;
  end
endmodule
```

### OUTPUT WAVEFORMS



**OR Gate**





| 2 Input OR gate | | |
|---|---|---|
| A | B | A+B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Structural Model**

```
module
or_gate(x,y,z);
  input x,y;
  output z;
  or g1(z,x,y);
endmodule
```

**Data Flow Model**

```
module
or_gate(x,y,z);
input x,y;
output z;
assign z=(x|y);
endmodule
```

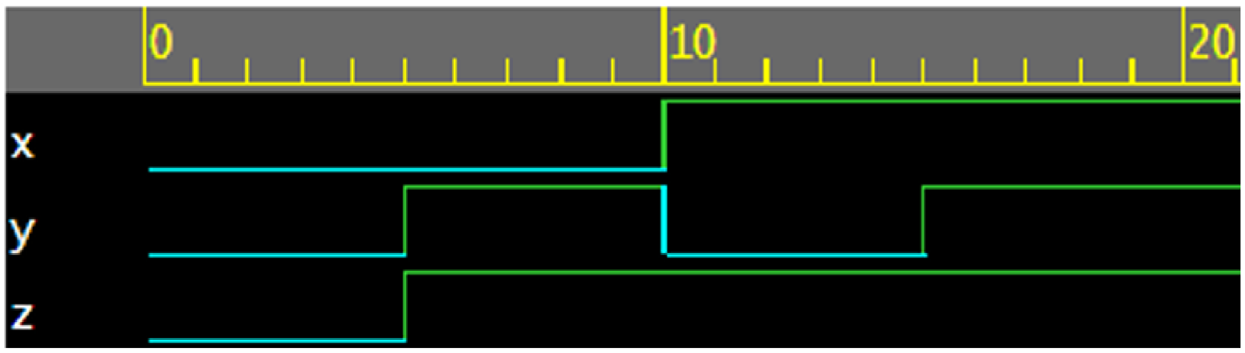
**Behavioural Model**

```
module or_gate(x,y,z);
   input x,y;
   output z;
   reg z;
   always@(x,y)z=x|y;
 endmodule
```

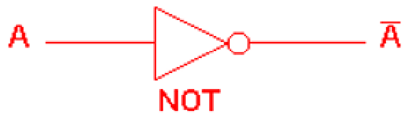
**TEST BENCH:**

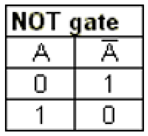
```
module or_tb;
           reg x,y;
           wire z;
  or_gate inst(.x(x), .y(y),
.z(z));

initial begin

$dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end

initial
  begin
    $monitor(x,y,z);
           x<=0;  y<=0;  #5;
           x<=0;  y<=1;  #5;
           x<=1;  y<=0;  #5;
           x<=1;  y<=1;  #5;
  end
endmodule
```

**OUTPUT WAVEFORMS**



**NOT Gate**

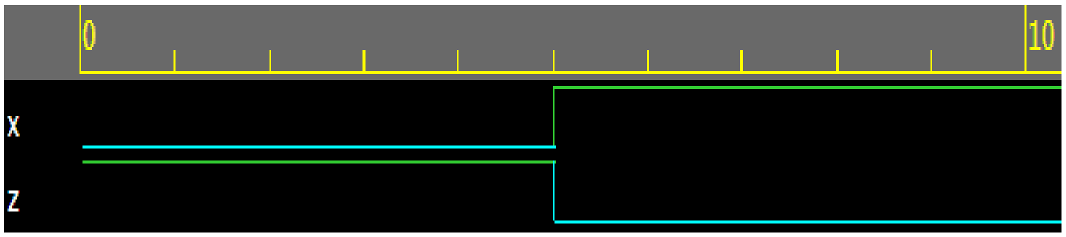




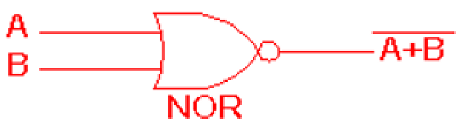
| NOT gate | |
|---|---|
| A | $\overline{A}$ |
| 0 | 1 |
| 1 | 0 |

| **Structural Model** | **Data Flow Model** | **BehaviouralModel** | **TESTBENCH:** |
|---|---|---|---|
| module not_gate(x,z); | module not_gate(x,z); | module not_gate(x,z); | module notdf_tb; |
| input x; | input x; | input x; | reg x; |
| output z; | output z; | output z; | wire z; |
| not g1(z,x); | assign z=!x; | reg z; | not_gate |
| endmodule | endmodule | always@(x)z=!x; | inst(.x(x),.z(z)); |
| | | endmodule | initial begin |
| | | | x=0;#5; |
| | | | x=1;#5; |
| | | | end |
| | | | endmodule |

**OUTPUT WAVEFORMS**



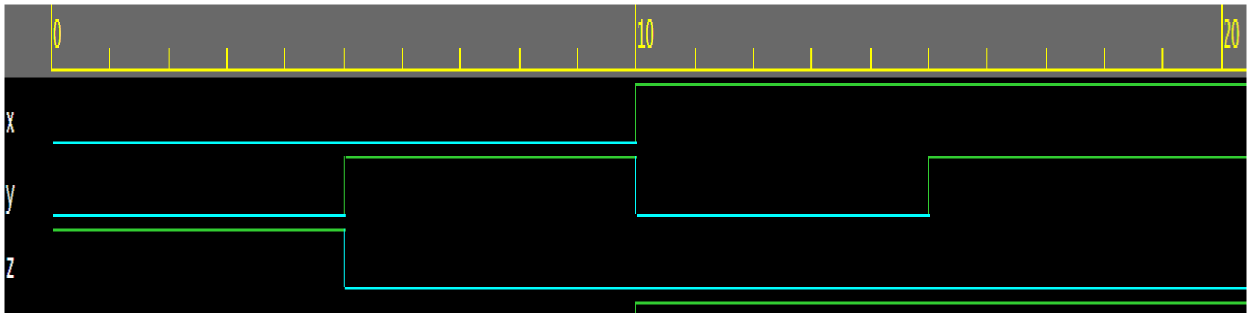
**NOR Gate**



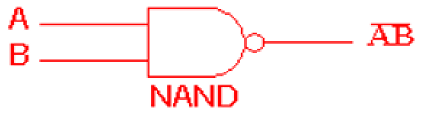
| Input | | Output |
|---|---|---|
| A | B | $\overline{A+B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| Structural Model | Data Flow Model | Behavioral Model | TESTBENCH: |
|---|---|---|---|
| <pre>module<br>nor_gate(x,y,z);<br>  input x,y;<br>  output z;<br>  nor g1(z,x,y);<br>endmodule</pre> | <pre>module<br>nor_gate(x,y,z);<br>  input x,y;<br>  output z;<br>  assign z=! (x\|y);<br>endmodule</pre> | <pre>module<br>nor_gate(x,y,z);<br>  input x,y;<br>  output z;<br>  reg z;<br>always@(x,y)z=!(x\|y);<br>endmodule</pre> | <pre>module nor_tb;<br>           reg x,y;<br>           wire z;<br>  nor_gate inst(.x(x), .y(y),<br>.z(z));<br>initial begin<br>    $dumpfile("dump.vcd");<br>    $dumpvars;<br>    #30 $finish;<br>end<br>initial<br>  begin<br>    $monitor(x,y,z);<br>           x<=0;  y<=0;  #5;<br>           x<=0;  y<=1;  #5;<br>           x<=1;  y<=0;  #5;<br>           x<=1;  y<=1;  #5;<br>  end<br>endmodule</pre> |

**OUTPUT WAVEFORMS**



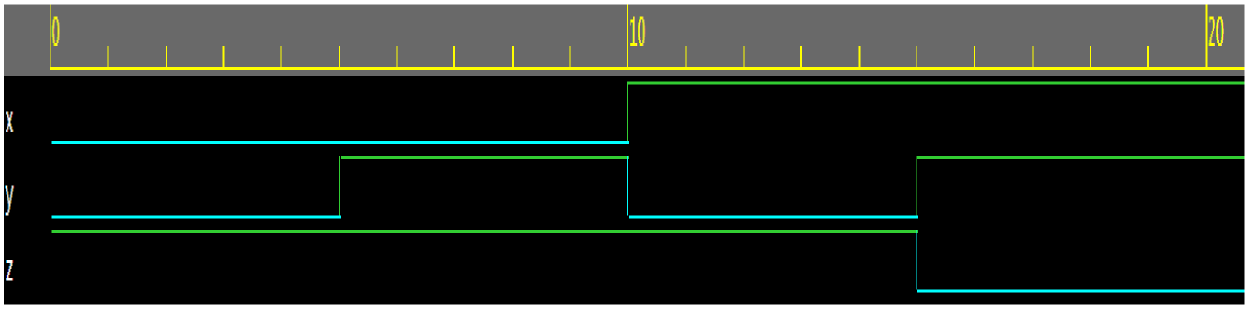
**NAND Gate**



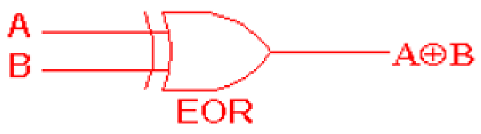
| Input | | Output |
|---|---|---|
| A | B | Y= $\overline{A.B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

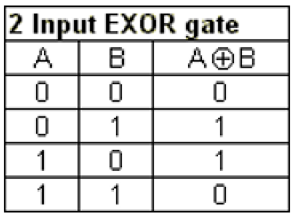
| Structural Model | Data Flow Model | Behavioural Model | TESTBENCH: |
|---|---|---|---|
| module nand_gate(x,y,z);<br>input x,y;<br>output z;<br>nand g1(z,x,y);<br>endmodule | module nand_gate(x,y,z);<br>input x,y;<br>output z;<br>assign z=!(x&y);<br>endmodule | module nand_gate(x,y,z);<br>input x,y;<br>output z;<br>reg z;<br>always@(x,y)z=!(x&y);<br>endmodule | module nand_tb;<br>reg x,y;<br>wire z;<br>nand_gate inst(.x(x), .y(y), .z(z));<br>initial begin<br>$dumpfile("dump.vcd");<br>$dumpvars;<br>#30 $finish;<br>end<br>initial<br>begin<br>$monitor(x,y,z);<br>x<=0; y<=0; #5;<br>x<=0; y<=1; #5;<br>x<=1; y<=0; #5;<br>x<=1; y<=1; #5;<br>end<br>endmodule |

**OUTPUT WAVEFORMS**



**XOR Gate**





| 2 Input EXOR gate | | |
|---|---|---|
| A | B | A⊕B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Structural Model**

```
module
xor_gate(x,y,z);
  input x,y;
  output z;
  xor g1(z,x,y);
endmodule
```

**Data Flow Model**

```
module
xor_gate(x,y,z);
  input x,y;
  output z;
  assign z=(x^y);
endmodule
```

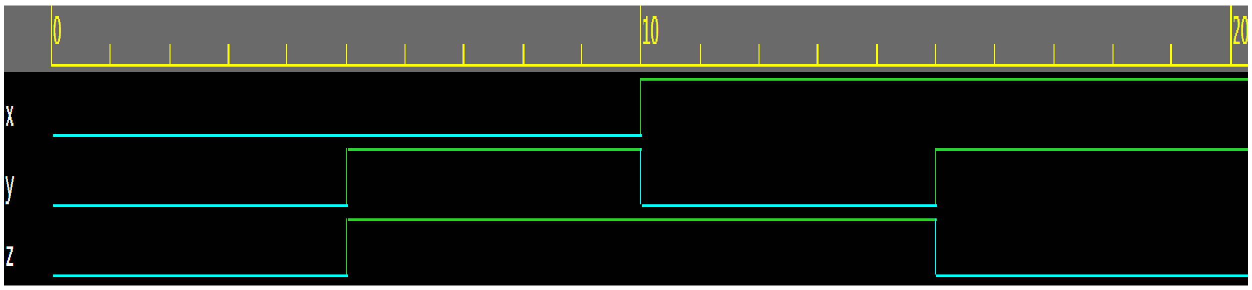
**Behavioural Model**

```
module xor_gate(x,y,z);
  input x,y;
  output z;
  reg z;
always@(x,y)z=x^y;
endmodule
```

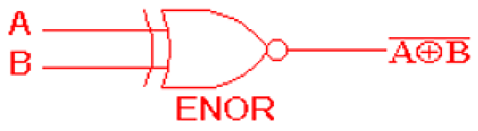
**TESTBENCH:**

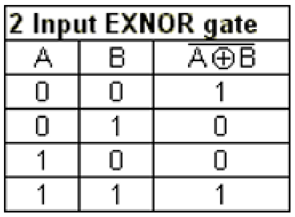
```
module xordf_tb;
reg x,y;
wire z;
xor_gate (.x(x),.y(y),.z(z));
initial begin
x=0; y=0;  #5;
x=0; y=1;#5;
x=1; y=0;#5;
x=1;y=1;#5;
end
endmodule
```

**OUTPUT WAVEFORMS**



**XNOR Gate**





| 2 Input EXNOR gate | | |
|---|---|---|
| A | B | $\overline{A \oplus B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Structural Model**

```
module
xnor_gate(x,y,z);
  input x,y;
  output z;
  xnor g1(z,x,y);
endmodule
```

**Data Flow Model**

```
module
xnor_gate(x,y,z);
  input x,y;
  output z;
  assign z=!(x^y);
endmodule
```

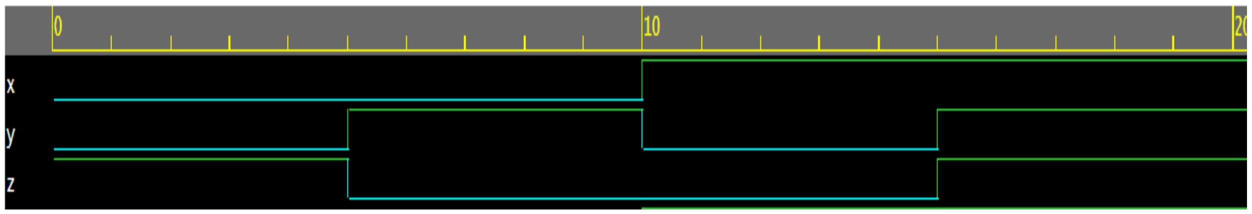
**Behavioural Model**

```
module
xnor_gate(x,y,z);
  input x,y;
  output z;
  reg z;
always@(x,y)z=!(x^y);
endmodule
```

**TESTBENCH:**

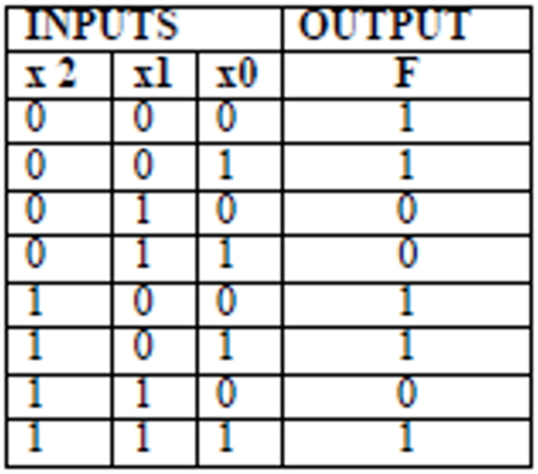
```
module xnor_tb;
          reg x,y;
          wire z;
  xnor_gate inst(.x(x), .y(y),
.z(z));
initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #30 $finish;
end
initial
  begin
    $monitor(x,y,z);
          x<=0;  y<=0;  #5;
          x<=0;  y<=1;  #5;
          x<=1;  y<=0;  #5;
          x<=1;  y<=1;  #5;
  end
endmodule
```

**OUTPUT WAVEFORMS**

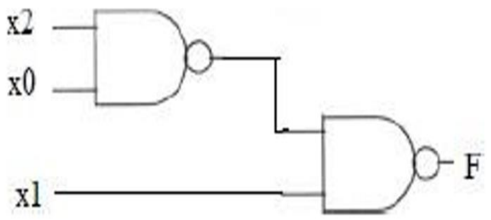


c. To design and simulate the HDL code to realize three and four-variable Boolean functions

i. F1(x2, x1,x0) = $x_2'x_1'x_0' + x_2'x_1'x_0 + x_2x_1'x_0' + x_2x_1'x_0 + x_2x_1x_0$



| INPUTS | | | OUTPUT |
|---|---|---|---|
| x 2 | x1 | x0 | F |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



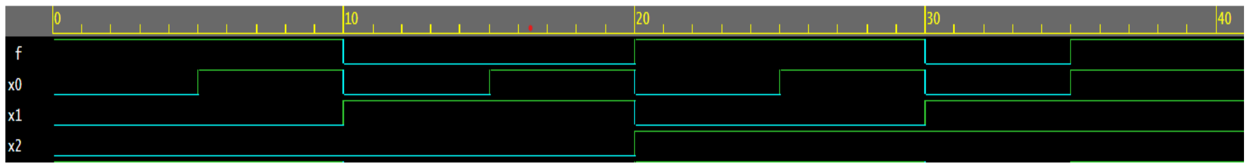
**Verilog code**

```
module f1(x2,x1,x0,f);
input x2,x1,x0;
output f;
wire x;
  nand u1(x,x2,x0);
  nand u2(f,x,x1);
endmodule
```

**Test Bench**

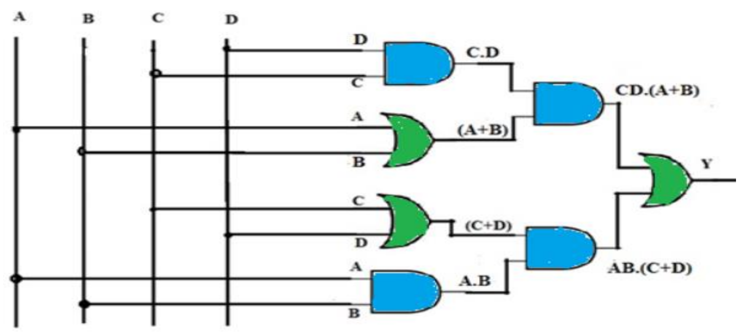
```
module f1_tb;
        reg x2,x1,x0;
        wire f;
  f1 inst(.x2(x2),.x1(x1),.x0(x0),.f(f));
initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #50 $finish;
end
initial
  begin
    $monitor(x2,x1,x0,f);
        x2<=0;   x1<=0;   x0<=0;      #5;
        x2<=0;   x1<=0;   x0<=1;      #5;
        x2<=0;   x1<=1;   x0<=0;      #5;
        x2<=0;   x1<=1;   x0<=1;      #5;
        x2<=1;   x1<=0;   x0<=0;      #5;
        x2<=1;   x1<=0;   x0<=1;      #5;
        x2<=1;   x1<=1;   x0<=0;      #5;
        x2<=1;   x1<=1;   x0<=1;      #5;
  end
endmodule
```

**OUTPUT WAVEFORMS**



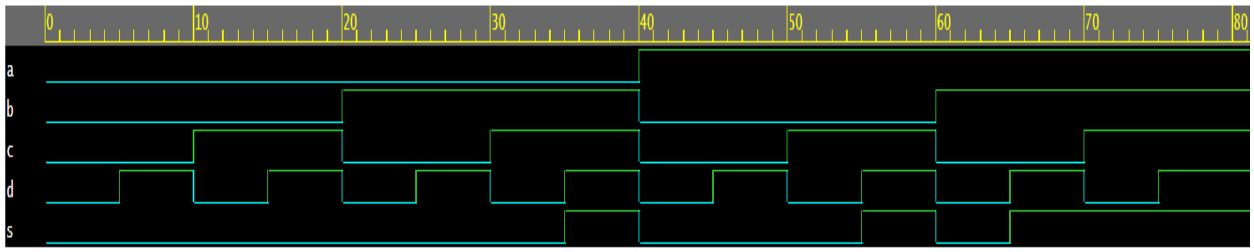
ii. Y = AB’CD+ABCD+A’BCD+ABCD’+ABC’D = ACD + BCD +ABC +ABD
= CD(A+B) +AB(C+D)

| INPUTS | | | | OUTPUT |
|---|---|---|---|---|
| A | B | C | D | S |
| 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 | 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 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



| **Verilog code** | **Test Bench** |
|---|---|
| module f1(a,b,c,d,s); | module f1_tb; |
| input a,b,c,d; | reg a,b,c,d; |
| output s; | wire s; |
| wire x1,x2,x3,x4,x5,x6; | f1 inst(.a(a),.b(b),.c(c),.d(d),.s(s)); |
| and u1(x1,c,d); | initial begin |
| or u2(x2,a,b); | $dumpfile("dump.vcd"); |
| and u3(x3,a,b); | $dumpvars; |
| or u4(x4,c,d); | #100 $finish; |
| and u5(x5,x1,x2); | end |
| and u6(x6,x3,x4); | initial |
| or u7(s,x5,x6); | begin |
| endmodule | $monitor(a,b,c,d,s); |
| | a<= 0; b<=0; c<=0; d<=0; #5; |
| | a<= 0; b<=0; c<=0; d<=1; #5; |
| | a<= 0; b<=0; c<=1; d<=0; #5; |
| | a<= 0; b<=0; c<=1; d<=1; #5; |
| | a<= 0; b<=1; c<=0; d<=0; #5; |
| | a<= 0; b<=1; c<=0; d<=1; #5; |
| | a<= 0; b<=1; c<=1; d<=0; #5; |
| | a<= 0; b<=1; c<=1; d<=1; #5; |
| | a<= 1; b<=0; c<=0; d<=0; #5; |
| | a<= 1; b<=0; c<=0; d<=1; #5; |
| | a<= 1; b<=0; c<=1; d<=0; #5; |
| | a<= 1; b<=0; c<=1; d<=1; #5; |
| | a<= 1; b<=1; c<=0; d<=0; #5; |
| | a<= 1; b<=1; c<=0; d<=1; #5; |
| | a<= 1; b<=1; c<=1; d<=0; #5; |
| | a<= 1; b<=1; c<=1; d<=1; #5; |
| | end |
| | endmodule |

**OUTPUT WAVEFORMS**



**INFERENCE:**
Familiarized with the basic syntax of Verilog and modeled AND, OR, NOT, NAND, NOR, XOR, and XNOR gates and three and four-variable Boolean functions in Verilog HDL, and their truth tables were verified.

## EXPERIMENT 10
## HALF ADDER AND FULL ADDER

**AIM OF THE EXPERIMENT:**

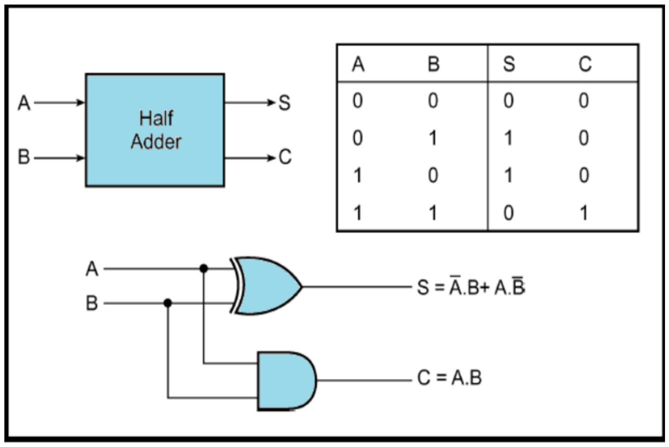
(a) To develop Verilog modules for half adder in 3 modeling styles (dataflow/ structural/ behavioral).

(b) To develop Verilog modules for full adder in structural modeling using half adder.

**BRIEF DESCRIPTION:**

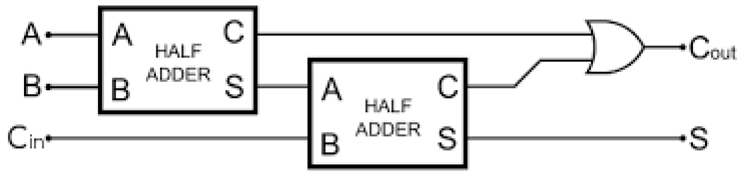
### HALF ADDER

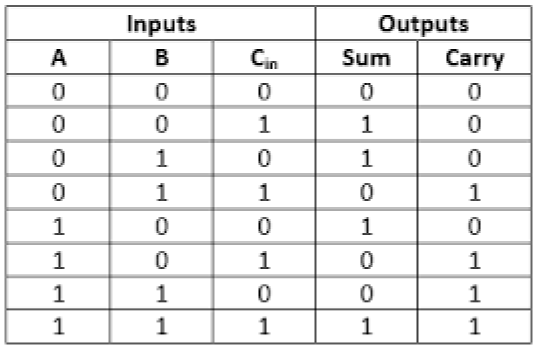
Half adders perform a simple binary addition of 2 bits producing 2 outputs, the sum bit (S) and carry bit (C). The half adder is shown in the block diagram in the following figure



| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

### FULL ADDER USING HALF ADDER





| Inputs | | | Outputs | |
|---|---|---|---|---|
| A | B | $C_{in}$ | Sum | Carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**a.1) Half Adder: Dataflow style modelling**

**Verilog code**

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

**Test Bench**

```
module HA_tb;
            reg a,b;
            wire s,c;
  HA_D inst(.a(a), .b(b), .s(s), .c(c));
 initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end
initial
  begin
    $monitor(a,b,s,c);
            a<=0;   b<=0;  #5;
            a<=0;   b<=1;  #5;
            a<=1;   b<=0;  #5;
            a<=1;   b<=1;  #5;
  end
endmodule
```

**a.2) Half adder: Structural style modelling**

**Verilog code**

```
module HA_D(a,b,s,c);
  input a,b;
  output s,c;
  xor x1(s,a,b);
  and x2(c,a,b);
endmodule
```

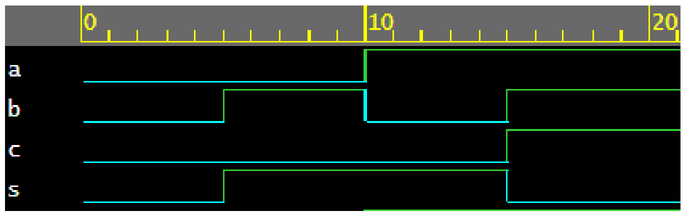
**Test Bench**

```
module HA_tb;
            reg a,b;
            wire s,c;
  HA_D inst(.a(a), .b(b), .s(s), .c(c));
 initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end
initial
  begin
    $monitor(a,b,s,c);
            a<=0;   b<=0;  #5;
            a<=0;   b<=1;  #5;
            a<=1;   b<=0;  #5;
            a<=1;   b<=1;  #5;
  end
endmodule
```

**a.3) Half adder: Behavioural style modelling**

| Verilog code | Test Bench |
|---|---|
| <pre>module HA_D(a,b,s,c);<br>  input a,b;<br>  output reg s,c;<br>  always @(a,b)<br>begin<br> s = a^b;<br> c = a & b;<br>end<br>endmodule</pre> | <pre>module HA_tb;<br>            reg a,b;<br>            wire s,c;<br>  HA_D inst(.a(a), .b(b), .s(s), .c(c));<br> initial begin<br>    $dumpfile("dump.vcd");<br>    $dumpvars;<br>    #100 $finish;<br>end<br>initial<br>  begin<br>    $monitor(a,b,s,c);<br>            a<=0;  b<=0;  #5;<br>            a<=0;  b<=1;  #5;<br>            a<=1;  b<=0;  #5;<br>            a<=1;  b<=1;  #5;<br>  end<br>endmodule</pre> |

**OUTPUT WAVEFORMS**



**b.Full adder in structural style modelling using half adder**

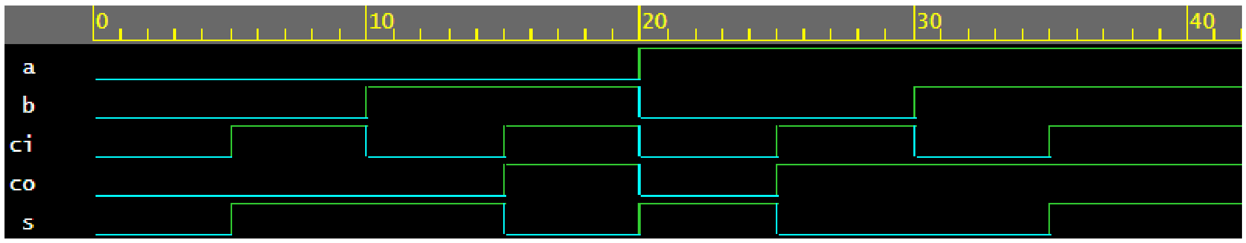
**Verilog code**

```
module HA_D(a,b,s,c);
  input a,b;
  output s,c;
  xor x1(s,a,b);
  and x2(c,a,b);
endmodule
module FA(a,b,ci,s,co);
input a,b,ci;
output s,co;
wire s1,c1,c2;
HA_D u1(a,b,s1,c1);
HA_D u2(s1,ci,sum,c2);
or us(co,c1,c2);
endmodule
```

**Test Bench**

```
module FA_tb;
            reg a,b,ci;
            wire s,co;
  FA inst(.a(a), .b(b), .ci(ci), .s(s), .co(co));
initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end
initial
  begin
    $monitor(a,b,s,ci,co);
            a<=0;  b<=0;  ci<=0;  #5;
            a<=0;  b<=0;  ci<=1;  #5;
            a<=0;  b<=1;  ci<=0;  #5;
            a<=0;  b<=1;  ci<=1;  #5;
            a<=1;  b<=0;  ci<=0;  #5;
            a<=1;  b<=0;  ci<=1;  #5;
            a<=1;  b<=1;  ci<=0;  #5;
            a<=1;  b<=1;  ci<=1;  #5;
  end
endmodule
```

**OUTPUT WAVEFORMS**



**INFERENCE**
Modelled Half Adder in three modelling styles and full adder using half adder in Verilog HDL and their truth-tables were verified.

## EXPERIMENT NO. 11
## MUX AND DEMUX IN VERILOG

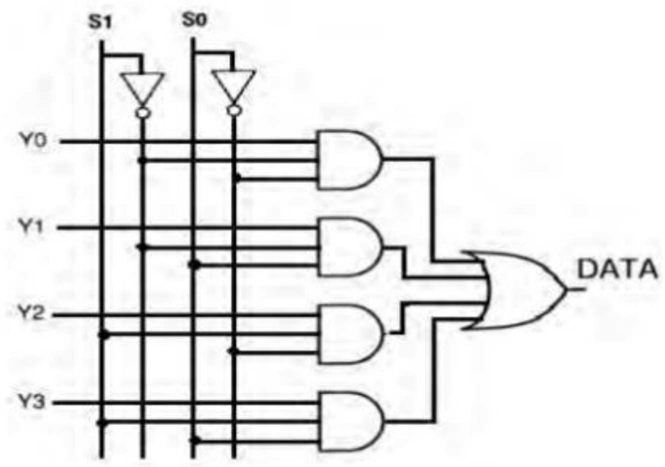
**AIM OF THE EXPERIMENT:**

(a) To develop Verilog modules for a 4x1 MUX.
(b) To develop Verilog modules for a 1x4 DEMUX.

**BRIEF DESCRIPTION**

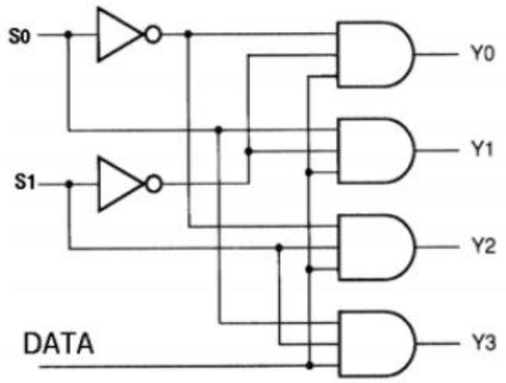
A multiplexer (MUX) is a device that selects one of several analog or digital input signals and forwards the selected input into a single line. A multiplexer of 2 m inputs has m select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector.
Conversely, a demultiplexer (DEMUX) is a device taking a single input signal and selecting one of many data-output lines, which is connected to the single input. A multiplexer is often used with a complementary demultiplexer on the receiving end. The conversion from one code to another is common in digital systems.

**4X1 MULTIPLEXER - Logic Diagram of 4X1 multiplexer**



| TRUTH TABLE | | |
|---|---|---|
| **S1** | **S0** | **DATA** |
| 0 | 0 | Y0 |
| 0 | 1 | Y1 |
| 1 | 0 | Y2 |
| 1 | 1 | Y3 |

**1X4 DEMULTIPLEXER - Logic Diagram of 1X4 de-multiplexer**



| TRUTH TABLE | | | | | | |
|---|---|---|---|---|---|---|
| INPUTS | | | OUTPUTS | | | |
| DATA | S1 | S0 | Y3 | Y2 | Y1 | Y0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

## VERILOG CODE FOR 4X1 MULTIPLEXER

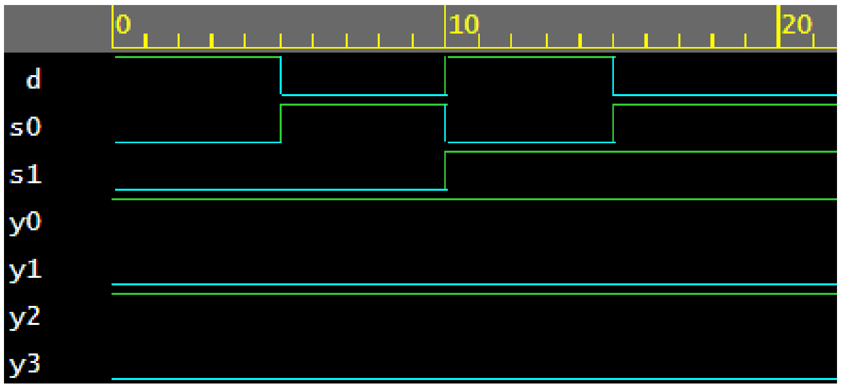
**Verilog code**

```
module mux(y0,y1,y2,y3,s1,s0,d);
input y0,y1,y2,y3,s1,s0;
output d;
assign d =
s1?(s0?y3:y2):(s0?y1:y0);
endmodule
```

**Test Bench**

```
module mux4x1_tb;
  reg y0,y1,y2,y3,s1,s0;
  wire d;
  mux4x1 u0(y0,y1,y2,y3,s1,s0,d);
initial
begin
    $dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end
initial
  begin
    $monitor(y0,y1,y2,y3,s1,s0,d);
y0<=1; y1<=1; y2<=1; y3<=1;
s1<=0; s0<=0; #5;
s1<=0; s0<=1; #5;
s1<=1; s0<=0; #5;
s1<=1; s0<=1; #5;
  end
endmodule
```

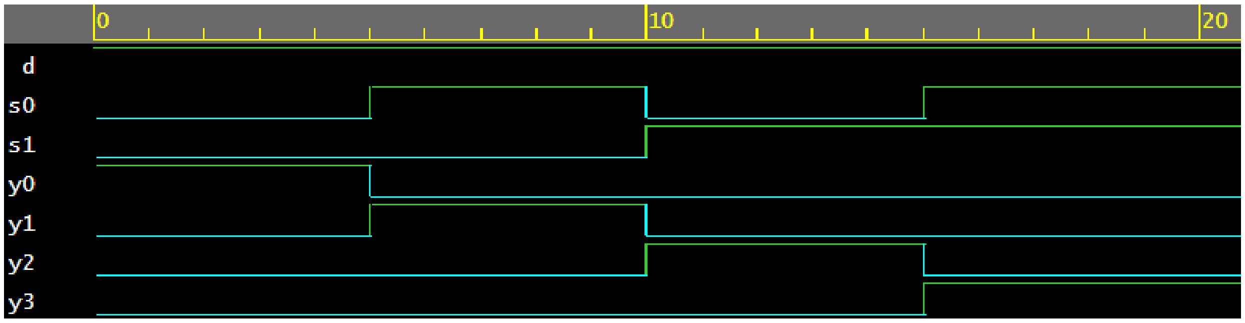
## OUTPUT WAVEFORMS
## 4X1 MULTIPLEXER



**VERILOG CODE FOR 1X4 DEMULTIPLEXER**

| Verilog code | Test Bench |
|---|---|
| module demux(d,s1,s0,y0,y1,y2,y3); | module demux_tb; |
| input d,s1,s0; | reg d,s1,s0; |
| output y0,y1,y2,y3; | wire y0,y1,y2,y3; |
| assign y0=d&(~s1)&(~s0); | demux u0(d,s1,s0,y0,y1,y2,y3); |
| assign y0=d&(~s1)&(s0); | initial |
| assign y0=d&(s1)&(~s0); | begin |
| assign y0=d&(s1)&(s0); | $dumpfile("dump.vcd"); |
| endmodule | $dumpvars; |
| | #50 $finish; |
| | end |
| | initial |
| | begin |
| | $monitor(y0,y1,y2,y3,s1,s0,d); |
| | d<=1; |
| | s1<=0; s0<=0; #5; |
| | s1<=0; s0<=1; #5; |
| | s1<=1; s0<=0; #5; |
| | s1<=1; s0<=1; #5; |
| | end |
| | endmodule |

**OUTPUT WAVEFORMS**
**1x4 DEMULTIPLEXER**



**INFERENCE**
Modelled multiplexer and demultiplexer in Verilog HDL and their truth-tables were verified

## EXPERIMENT 12:
## FLIPFLOPS AND SHIFTREGISTERS

**AIM OF THE EXPERIMENT:**

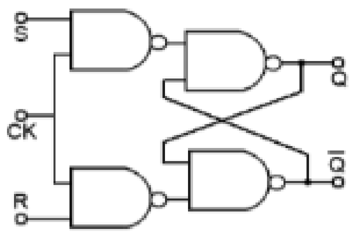
(a) To develop Verilog modules for SR, JK, T and D flip flops.

(b) To develop Verilog modules for a Johnson/Ring counter

**BRIEF DESCRIPTION**

**a) Circuit diagram and truth table of SR, JK and D flipflops**

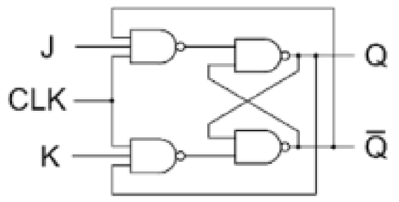
### S R Flip Flop



TRUTH TABLE

| S | R | $Q_N$ | $Q_{N+1}$ |
|---|---|---|---|
| 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 | - |

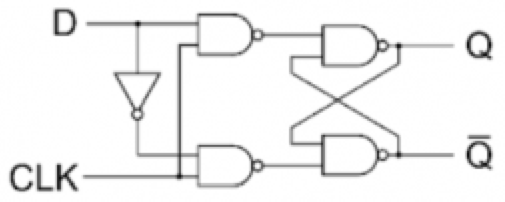
### J K Flip Flop



TRUTH TABLE

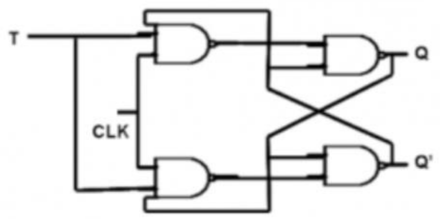
| J | K | $Q_N$ | $Q_{N+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

### D Flip Flop



| Q | D | Q(t+1) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

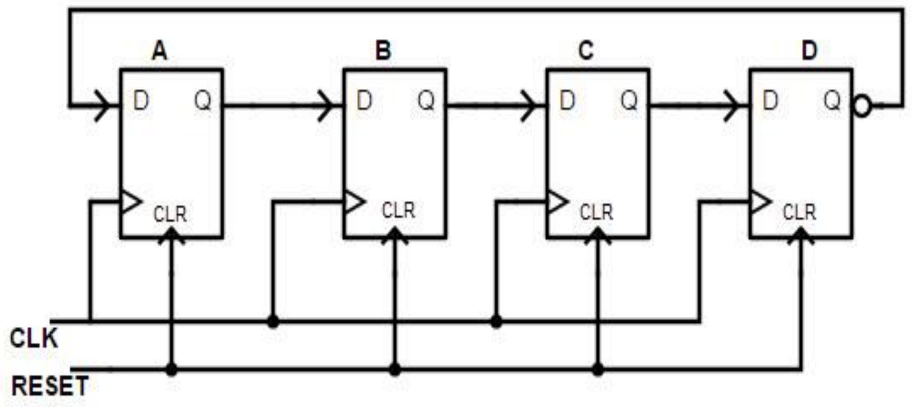
### T Flip Flop



| $T$ | $Q_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**b) Circuit diagram** of Johnson/Ring counter

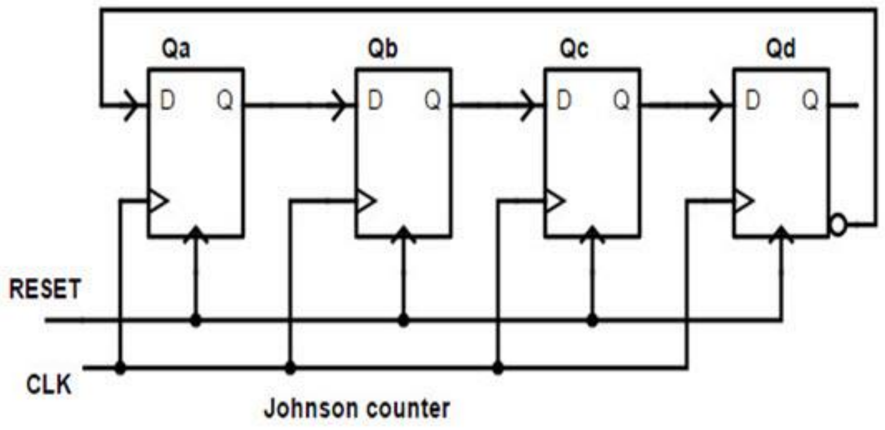
**RING COUNTER**



| $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

**JOHNSON COUNTER**

| $Q_A$ | $Q_B$ | $Q_C$ | $Q_D$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| repeat | | | |



## VERILOG CODE FOR SR FLIPFLOP

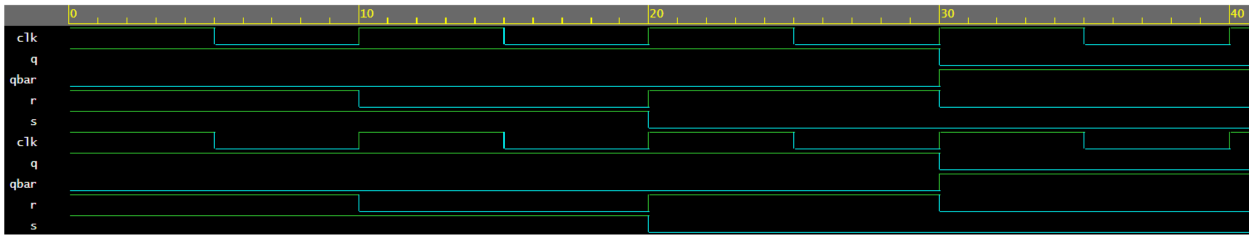
**Verilog code**

```
module srff(s,r,clk,q,qbar);
        input s,r,clk;
        output reg q, qbar;
always@(posedge clk)
begin
        if(s ==1)
        begin
                q=1;
                qbar=0;
        end
        else if(r==1)
        begin
                q=0;
                qbar=1;
        end
        else if(s==0 & r==0)
        begin
                q<=q;
                qbar<=qbar;
        end
end
endmodule
```

**Test Bench**

```
module srff_tb;
reg s,r,clk;
wire q,qbar;
srff u1(s,r,clk,q,qbar);
always #5 clk = ~clk;
initial begin
$dumpfile("dump.vcd");
   $dumpvars;
   #50 $finish;
end
initial
  begin
   $monitor(s,r,clk,q,qbar);
clk <=1;
s<=1; r<=1;    #10;
s<=1; r<=0;    #10;
s<=0; r<=1;    #10;
s<=0; r<=0;    #10;
end
endmodule
```

## OUTPUT WAVEFORMS



## VERILOG CODE FOR JK FLIPFLOP

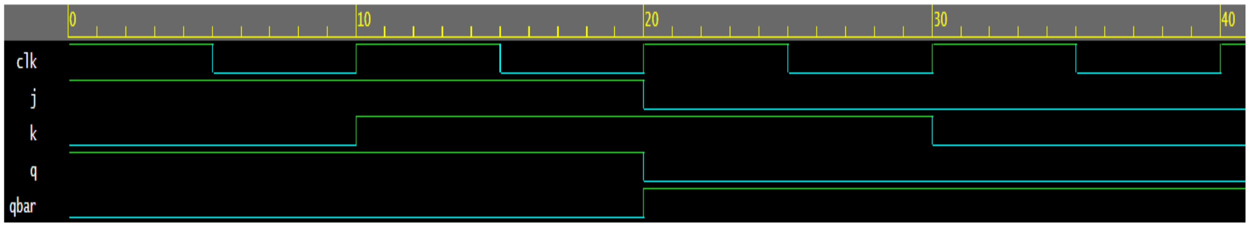
**Verilog code**

```
module jkff(j,k,clk,q,qbar);
            input j,k,clk;
            output reg q, qbar;
always@(posedge clk)
begin
  if(j==1 & k==1)
            begin
      q<=~q;
      qbar<=~qbar;
    end
  else if (j==1 & k==0)
    begin
            q=1;
                        qbar=0;
            end
  else if(j==0 & k==1)
            begin
                        q=0;
                        qbar=1;
            end
  else if(j==0 & k==0)
            begin
                        q<=q;
                        qbar<=qbar;
            end
end
endmodule
```

**Test Bench**

```
module jkff_tb;
reg j,k,clk;
wire q,qbar;
  jkff u1(j,k,clk,q,qbar);
always #5 clk = ~clk;
initial begin
$dumpfile("dump.vcd");
    $dumpvars;
    #50 $finish;
end
initial
  begin
    $monitor(j,k,clk,q,qbar);
clk <=1;
j<=1; k<=0;     #10;
j<=1; k<=1;     #10;
j<=0; k<=1;     #10;
j<=0; k<=0;     #10;
end
endmodule
```

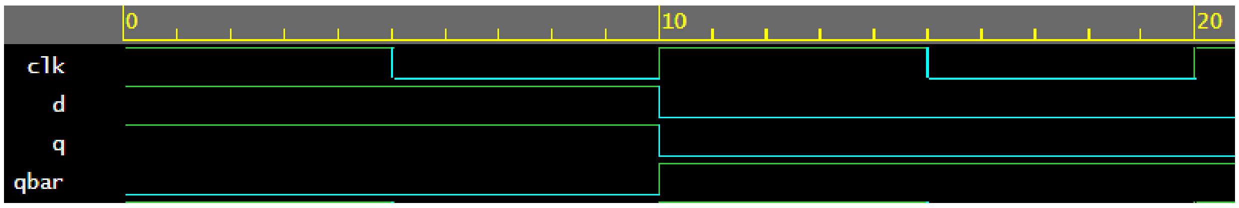
## OUTPUT WAVEFORMS



## VERILOG CODE FOR D FLIPFLOP

| Verilog code | Test Bench |
|---|---|
| <pre>module dff(d,clk,q,qbar);<br>input d,clk;<br>output q,qbar;<br>assign q=clk?d:q;<br>assign qbar=~q;<br>endmodule</pre> | <pre>module dff_tb;<br>reg d,clk;<br>wire q,qbar;<br>  dff u1(d,clk,q,qbar);<br>always #5 clk = ~clk;<br>initial begin<br>$dumpfile("dump.vcd");<br>    $dumpvars;<br>    #50 $finish;<br>end<br>initial<br>  begin<br>    $monitor(j,k,clk,q,qbar);<br>clk <=1;<br>d<=1;  #10;<br>d<=0;  #10;<br>d<=1;  #10;<br>d<=0;  #10;<br>end<br>endmodule</pre> |

## OUTPUT WAVEFORMS



## VERILOG CODE FOR T FLIPFLOP

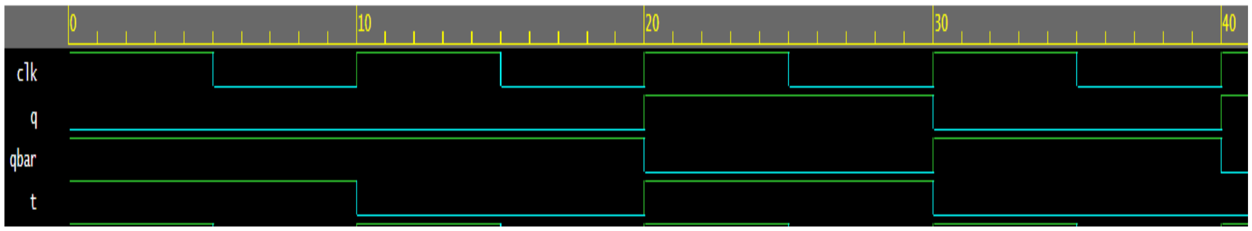
**Verilog code**

```
module tff(t,clk,q,qbar);
input t,clk;
output reg q,qbar;
always@(posedge clk)
begin
  q=1;
  qbar=0;
  if(t ==0)
            begin
                        q=q;
                        qbar=qbar;
            end
            else
    begin
                        q<=~q;
                        qbar<=~qbar;
            end
end
endmodule
```

**Test Bench**

```
module tff_tb;
reg t,clk;
wire q,qbar;
  tff u1(t,clk,q,qbar);
always #5 clk = ~clk;
initial begin
$dumpfile("dump.vcd");
    $dumpvars;
    #50 $finish;
end
initial
  begin
    $monitor(t,clk,q,qbar);
clk <=1;
t<=1;   #10;
t<=0;   #10;
t<=1;   #10;
t<=0;   #10;

end
endmodule
```

## OUTPUT WAVEFORMS



## VERILOG CODE FOR 4 BIT RING COUNTER

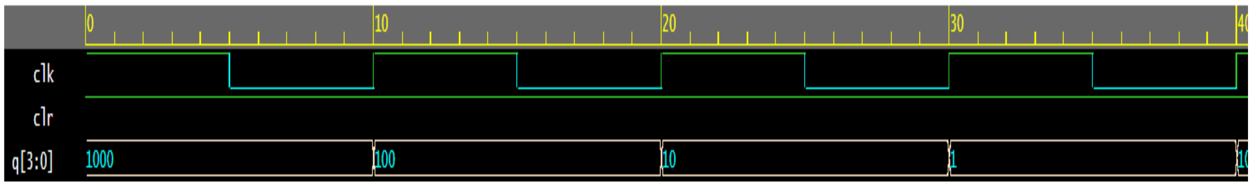
**Verilog Code**

```
module ring_c(clk,clr,q);
            input clk,clr;
            output [3:0] q;
            wire clk,clr;
  reg [3:0] q =4'b0000;
  always @(posedge clk,posedge clr)
    begin
      if(clr==0)
        begin
          q<=1;
        end
      else
      begin
        q[3]<=q[0];
        q[2]<=q[3];
        q[1]<=q[2];
        q[0]<=q[1];
              end
end
endmodule
```

**Test bench**

```
module ring_tb;
            reg clk=1,clr=0;
            wire [3:0] q;
            ring_c inst(clk,clr,q);
            always #5 clk = ~clk;
initial
  begin
            $dumpfile("dump.vcd");
    $dumpvars;
    #50 $finish;
end
initial
  begin
    $monitor(clk,clr,q);
    clr = 1;
end
endmodule
```

## OUTPUT WAVEFORMS



## VERILOG CODE FOR JOHNSON COUNTER

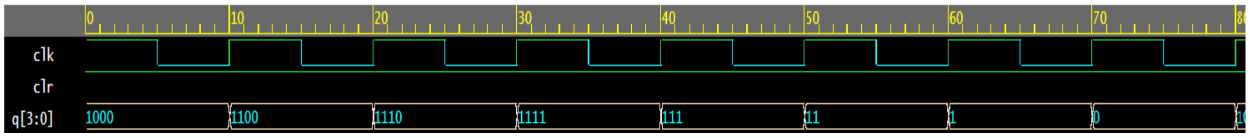
**Verilog Code**

```
module john_c(clk,clr,q);
             input clk,clr;
             output [3:0] q;
             wire clk,clr;
  reg [3:0] q =4'b0000;
  always @(posedge clk,posedge clr)
    begin
      if(clr==0)
        begin
          q<=1;
        end
      else
      begin
        q[3] <=~q[0];
        q[2]<=q[3];
        q[1]<=q[2];
        q[0]<=q[1];
               end
end
endmodule
```

**Test bench**

```
module john_tb;
             reg clk=1,clr=1;
             wire [3:0] q;
             john_c inst(clk,clr,q);
             always #5 clk = ~clk;
initial
  begin
             $dumpfile("dump.vcd");
    $dumpvars;
    #100 $finish;
end
initial
  begin
    $monitor(clk,clr,q);
#5 clr = 1;

end
endmodule
```

## OUTPUT WAVEFORMS



## INFERENCE

Modeled SR, JK, T, and D Flip Flops and Ring and Johnson counters in Verilog HDL, and their truth tables were verified.