# SRI SHAKTHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**20VD212 – TESTING OF VLSI CIRCUITS LABORATORY**

**LABORATORY RECORD**

**NAME:**__________________________**ROLLNO:** ____________________________________

**CLASS:**___________________________**BRANCH:** ____________________________________

**ACADEMIC YEAR:** ________________**BATCH:** ____________**SEMESTER:** _____________

Certified and Bonafide record of work done by ………..…………….……………………

Place:

Date:

**Staff In-Charge** **Head of the Department**

University Register Number: ………………………………………………...

Submitted for the University Practical Examination held on ……..…………

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

# INDEX

| EX.NO | DATE | NAME OF THE EXPERIMENT | MARKS | SIGN |
|---|---|---|---|---|
| 1 | 25.02.25 | Design and simulation of basic logic gates and universal gates using Xilinx ISE | | |
| 2 | 04.03.25 | Design and simulation of adder and subtractor using Xilinx ISE | | |
| 3 | 04.03.25 | Design and simulation of encoder and decoder using Xilinx ISE | | |
| 4 | 18.03.25 | Design and simulation of multiplexer and demultiplexer using Xilinx ISE | | |
| 5 | 18.03.25 | Design and simulation of 8 bit adder and multiplier using Xilinx ISE | | |
| 6 | 25.03.25 | Design and simulation of PRBS generator and accumulator using Xilinx ISE | | |
| 7 | 02.04.25 | Design and simulation of parity generator using Xilinx ISE | | |
| 8 | 02.04.25 | Design and simulation of SISO shift registers using Xilinx ISE | | |
| 9 | 13.05.25 | Design and simulation of 4 bit magnitude comparator using Xilinx ISE | | |
| 10 | 20.05.25 | Design and simulation of flip flop using Xilinx ISE | | |
| | | AVERAGE: | | |

## SIGNATURE

**Experiment No 1:**

**DESIGN AND SIMULATION OF BASIC LOGIC GATES AND UNIVERSAL GATES**

**AIM:**

To design basic logic gates and universal logic gates using Nclaunch.

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

**AND GATE:**

The AND gate is a primary logic gate where the output is equal to the product of its inputs. The output of this gate is high only if both the inputs are high else the output is low. Here's the logical representation of the AND gate.


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

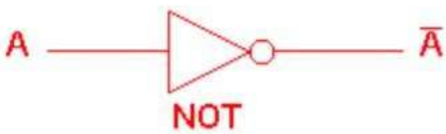
**OR GATE:**

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.


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

**NOT GATE:**

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It can also be done using NOR logic gates in the same way.




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

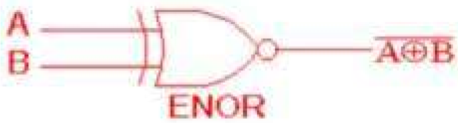
**EXOR GATE:**

The 'Exclusive-OR' gate is a circuit which will give a high output if either, but not both, of its two inputs are high. An encircled plus sign (+) is used to show the EOR operation.


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

**EXNOR GATE:**

The 'Exclusive-NOR' gate circuit does the opposite to the EOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.




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

**PROGRAMME:**

**VERILOG CODE:**

```
module basic1(a,b,YAND,YOR,YNOT,YXOR,YXNOR);
input a,b;
 output YAND,YOR,YNOT,YXOR,YXNOR;
assign YAND = a & b;
assign YOR = a | b;
assign YNOT =~ a ;
assign YXOR = a ^ b;
 assign YXNOR = ~(a ^ b);
 endmodule
```

**TEST BENCH:**

```
module basic1_tb();
wire YAND,YOR,YNOT,YXOR,YXNOR;
reg a,b;
 basic1 basic1_ins(a,b, YAND,YOR,YNOT,YXOR,YXNOR);
 initial begin
a=0; b=0;
 #5 a=0; b=1;
 #5 a=1; b=0;
#5 a=1; b=1;
 end
endmodule
```

**RESULT:**

Verilog code for the basic gates circuit and its test bench for verification is written, the waveform is observed.

**Experiment No 2:**

## DESIGN AND SIMULATION OF ADDER AND SUBTRACTOR

**AIM:**

To design adder and subtractor using Nclaunch.

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

**PROGRAM:**

**VERILOG CODE - HALF ADDER:**

```
module half_adder(
input a,b,
output sum,carry);

  assign sum = a^b;
  assign carry = a & b;

endmodule
```

**TESTBENCH:**

```
module tb_half_adder;

   reg A,B;
   wire SUM,CARRY;

   half_adder HA (.a(A) ,.b(B),.sum(SUM),.carry(CARRY))
   initial begin
      A =1'b0;
      B= 1'b0;
      #45 $finish;
   end
   always #6 A =~A;
   always #3 B =~B;
   always @(SUM,CARRY)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value SUM =%b
CARRY =%b ",$time,A,B,SUM,CARRY);
endmodule
```

**VERILOG CODE - FULL ADDER:**

**DATAFLOW:**

```
module full_adder(
input a,b,cin,
output reg  sum,cout);

  always @(*) begin
      sum = a^b^cin;
      cout = (a&b)+(b&cin)+(cin&a);
  end

endmodule
```

**TESTBENCH:**

```
module tb_full_adder;

    reg A,B,CIN;
    wire SUM,COUT;

    full_adder FA (.a(A) ,.b(B),.sum(SUM),.cin(CIN),.cout(COUT));
    initial begin
        A =1'b0;
        B= 1'b0;
        CIN = 1'b0;
        #45 $finish;
    end

    always #6 A =~A;
    always #3 B =~B;
    always #12 CIN = ~CIN;
    always @(SUM,COUT)
    $display( "time =%0t \tINPUT VALUES: \t A =%b B =%b CIN =%b \t output value
SUM=%b COUT =%b ",$time,A,B,CIN,SUM,COUT);
endmodule
```

**BEHAVIORAL – CASE STATEMENT:**

```
full_adder(input wire A, B, Cin, output reg S, output reg Cout);
 always @(A or B or Cin)
  begin

  case (A | B | Cin)
    3'b000: begin S = 0; Cout = 0; end
    3'b001: begin S = 1; Cout = 0; end
    3'b010: begin S = 1; Cout = 0; end
    3'b011: begin S = 0; Cout = 1; end
    3'b100: begin S = 1; Cout = 0; end
    3'b101: begin S = 0; Cout = 1; end
```

```
    3'b110: begin S = 0; Cout = 1; end
   3'b111: begin S = 1; Cout = 1; end
   endcase
  end
endmodule
```

**BEHAVIORAL - IF- ELSE:**

```
module full_adder( A, B, Cin, S, Cout);
input wire A, B, Cin;
output reg S, Cout;

always @(A or B or Cin)
begin
 if(A==0 && B==0 && Cin==0)
  begin
   S=0;
   Cout=0;
  end

 else if(A==0 && B==0 && Cin==1)
  begin
   S=1;
   Cout=0;
  end

 else if(A==0 && B==1 && Cin==0)
  begin
   S=1;
   Cout=0;
  end

 else if(A==0 && B==1 && Cin==1)
  begin
   S=0;
   Cout=1;
  end

 else if(A==1 && B==0 && Cin==0)
  begin
   S=1;
   Cout=0;
  end

 else if(A==1 && B==0 && Cin==1)
  begin
   S=0;
   Cout=1;
  end
```

```
 else if(A==1 && B==1 && Cin==0)
  begin
   S=0;
   Cout=1;
  end

 else if(A==1 && B==1 && Cin==1)
  begin
   S=1;
   Cout=1;
  end
end
endmodule
```

**TESTBENCH:**

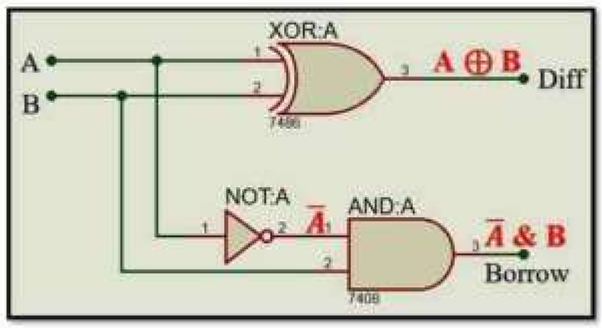
```
module top;
  reg  A_input, B_input, C_input;
  wire Sum, C_output;
  full_adder instantiation(.A(A_input), .B(B_input), .Cin(C_input), .S(Sum),
.Cout(C_output));

  initial
    begin
      A_input=0;
    B_input=0;
    C_input=0;
    #100 $finish;
    end

always #40 A_input=~A_input;
always #20 B_input=~B_input;
always #10 C_input=~C_input;

always @(A_input or B_input or C_input)
$monitor("At TIME(in ns)=%t, A=%d B=%d C=%d Sum = %d Carry = %d", $time,
A_input, B_input, C_input, Sum, C_output);
endmodule
```

**HALF SUBTARCTOR:**



**FULL SUBTRACTOR:**

```
module Full_Subtractor_3(output D, B, input X, Y, Z);
assign D = X ^ Y ^ Z;
assign B = ~X & (Y^Z) | Y & Z;
endmodule
```

**TEST BENCH:**

```
module Full_Subtractor_3_tb;
wire D, B;
reg X, Y, Z;
Full_Subtractor_3 Instance0 (D, B, X, Y, Z);
initial begin
    X = 0; Y = 0; Z = 0;
#10  X = 0; Y = 0; Z = 1;
#10  X = 0; Y = 1; Z = 0;
#10  X = 0; Y = 1; Z = 1;
#10 X = 1; Y = 0; Z = 0;
#10  X = 1; Y = 0; Z = 1;
#10  X = 1; Y = 1; Z = 0;
#10  X = 1; Y = 1; Z = 1;
end
initial begin
    $monitor ("%t, X = %d| Y = %d| Z = %d| B = %d| D = %d", $time, X, Y, Z, B, D);
end
endmodule
```

**RESULT:**

Verilog code for the adder, subtractor and its test bench for verification is written, the waveform is observed.

**Experiment No 3:**

## DESIGN AND SIMULATION OF ENCODER AND DECODER

**AIM:**

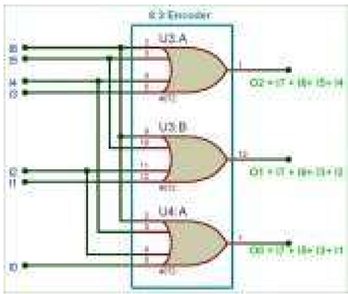
To design encoder and decoder using Nclaunch.

**APPARATUS REQUIRED:**

Nclaunch – Cadence tool

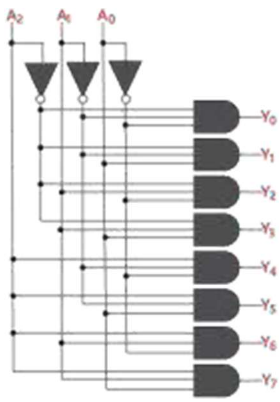
**THEORY:**

**ENCODER:**
An encoder is a combinational circuit that converts binary information in the form of a $2^N$ input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.



**DECODER:**

A decoder does the opposite job of an encoder. It is a combinational circuit that converts n lines of input into $2^n$ lines of output.




**PROGRAM:**

**8 to 3 Encoder:**

```
module encoder(
input i0, i1, i2, i3, i4, i5, i6, i7;
output y0,y1,y2;
or (y0, i4, i5, i6, i7);
or (y1, i2, i3, i6, i7);
```

```
or (y2, i1, i3, i5, i7);
endmodule
```

**TESTBENCH:**

```
module encodertb;
  reg  i0, i1, i2, i3, i4, i5, i6, i7;
  wire y0,y1,y2;

  encoder instantiation(.i0(i0), .i1(i1), .i2(i2), .i3(i3), .i4(i4), .i5(i5), .i6(i6), .i7(i7), .y0(y0),
.y1(y1), .y2(y2));

  initial
    begin
      i0=0; i1=0; i2=0; i3=0; i4=0; i5=0; i6=0; i7=0;
#10 i0=0; i1=0; i2=1; i3=0; i4=1; i5=1; i6=0; i7=1;
#10 i0=0; i1=1; i2=0; i3=1; i4=0; i5=1; i6=0; i7=1;
#10 i0=1; i1=0; i2=1; i3=0; i4=1; i5=0; i6=1; i7=0;
#10 i0=1; i1=1; i2=1; i3=1; i4=1; i5=1; i6=1; i7=1;
    end

always @(i0 or i1 or i2 or i3 or i4 or i5 or i6 or i7)
      $monitor("i0=%b, i1=%b, i2=%b,i3=%b, i4=%b, i5=%b, i6=%b, i7=%b, y0=%b, y1=%b,
y2=%b", i0, i1, i2, i3, i4, i5, i6, i7, y0, y1, y2);
endmodule
```

**VERILOG CODE - 3 to 8 Decoder:**

```
module decoder
input a,b,c,en;
output y1,y2,y3,y4,y5,y6,y7,y8;
wire w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12;
not(w1,a);
not(w2,b);
not(w3,c);
not(w4,a);
not(w5,b);
not(w6,a);
not(w7,c);
not(w8,a);
not(w9,b);
not(w10,a);
not(w11,b);
not(w12,c);
and(y1,w1,w2,w3);
and(y2,w4,w5,c);
and(y3,w6,b,w7);
and(y4,w8,b,c);
and(y5,a,w9,w10);
```

```
and(y6,a,w11,c);
and(y7,a,b,w12);
and(y8,a,b,c);
endmodule
```

**TESTBENCH:**

```
module decodertb;
  reg  a,b,c,en;
  wire y1,y2,y3, y4,y5,y6,y7,y8;

  encoder instantiation(.a(a), .b(b), .c(c), .en(en), .y1(y1), .y2(y2), .y3(y3), .y4(y4), .y5(y5),
.y6(y6), .y7(y7), .y8(y8));
  initial
    begin
       en = 0
#10 en = 1;
#10 a = 0; b = 0; c = 0;
#10 a = 0; b = 0; c = 1;
#10 a = 0; b = 1; c = 0;
#10 a = 0; b = 1; c = 1;
#10 a = 1; b = 0; c = 0;
#10 a = 1; b = 0; c = 1;
#10 a = 1; b = 1; c = 0;
#10 a = 1; b = 1; c = 1;
    end

initial begin
      $monitor("en=%b, a=%b, b=%b,c=%b, y1=%b, y2=%b, y3=%b, y4=%b, y5=%b,
y6=%b, y7=%b, y8=%b”, en, a, b, c, y1, y2, y3, y4, y5, y6, y7, y8);
endn
endmodule
```

**RESULT:**

Verilog code for the encoder decoder and its test bench for verification is written, the waveform is observed.

**Experiment No 4:**

## DESIGN AND SIMULATION OF MULTIPLEXER AND DEMULTIPLEXER

**AIM:**

To design multiplexer and demultiplexer using Nclaunch.

**APPARATUS REQUIRED:**

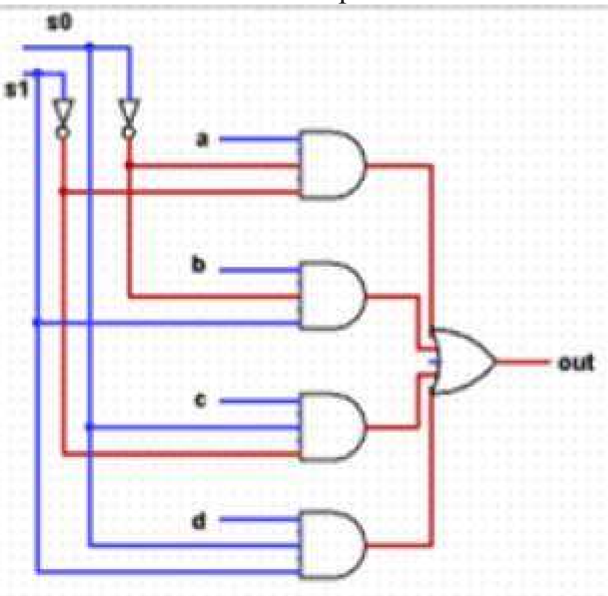
Nclaunch – Cadence tool

**THEORY:**

**MULTIPLEXER:**

In electronics, a **multiplexer** (or **mux**; spelled sometimes as **multiplexor**), also known as a **data selector**, is a device that selects between several analog or digital input signals and forwards the selected input to a single output line. The selection is directed a separate set of

digital inputs known as select lines. A multiplexer of inputs has select lines, which are used to select which input line to send to the output.




| Select Lines | | Output |
|---|---|---|
| s1 | s0 | out |
| 0 | 0 | a |
| 0 | 1 | b |
| 1 | 0 | c |
| 1 | 1 | d |

**DEMULTIPLEXER:**

A demultiplexer (or demux) is a device that takes a single input line and routes it to one of several digital output lines. A demultiplexer of $2^n$ outputs has n select lines, which are used to select which output line to send the input. A demultiplexer is also called a data distributor.

**PROGRAM:**

**VERILOG CODE:**

**GATE LEVEL MODELING:**

```
module m41(out, a, b, c, d, s0, s1);
output out;
input a, b, c, d, s0, s1;
```

```
wire sobar, s1bar, T1, T2, T3, T4;
not (s0bar, s0), (s1bar, s1);
and (T1, a, s0bar, s1bar), (T2, b, s0bar, s1),(T3, c, s0, s1bar), (T4, d, s0, s1);
or(out, T1, T2, T3, T4);
endmodule
```

**DATA FLOW MODELING:**

```
module m41 ( input a,
input b,
input c,
input d,
input s0, s1,
output out);
assign out = s1 ? (s0 ? d : c) : (s0 ? b : a);
endmodule
```

**BEHAVIORAL MODELING:**

```
module m41 ( a, b, c, d, s0, s1, out);
input wire a, b, c, d;
input wire s0, s1;
output reg out;
always @ (a or b or c or d or s0, s1)
begin
case (s0 | s1)
2'b00 : out <= a;
2'b01 : out <= b;
2'b10 : out <= c;
2'b11 : out <= d;
endcase
end
endmodule
```

**TESTBENCH:**

```
module top;
wire  out;
reg  a;
reg  b;
reg  c;
reg  d;
reg s0, s1;
m41 name(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));
 initial
 begin
 a=1'b0; b=1'b0; c=1'b0; d=1'b0;
 s0=1'b0; s1=1'b0;
 #500 $finish;
end
always #40 a=~a;
```

```
always #20 b=~b;
always #10 c=~c;
always #5 d=~d;
always #80 s0=~s0;
always #160 s1=~s1;
always@(a or b or c or d or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out);
endmodule
```

**DEMULTIPLEXER:**

**BEHAVIORAL MODELING - USING CASE STATEMENT:**

```
module Demultiplexer_1_to_4_case (output reg [3:0] Y, input [1:0] A, input din);
always @(Y, A) begin
   case (A)
      2'b00 : begin Y[0] = din; Y[3:1] = 0; end
      2'b01 : begin Y[1] = din; Y[0] = 0;  end
      2'b10 : begin Y[2] = din; Y[1:0] = 0; end
      2'b11 : begin Y[3] = din; Y[2:0] = 0; end
   endcase
end
endmodule
```

**TESTBENCH:**

```
module Demultiplexer_1_to_4_case_tb;
wire [3:0] Y;
reg [1:0] A;
reg din;
Demultiplexer_1_to_4_case I0 (Y, A, din);
initial begin
   din = 1;
   A = 2'b00;
#1  A = 2'b01;
#1  A = 2'b10;
#1  A = 2'b11;
end
initial begin
   $monitor("%t| Din = %d| A[1] = %d| A[0] = %d| Y[0] = %d| Y[1] = %d| Y[2] = %d| Y[3]
= %d",
           $time, din, A[1], A[0], Y[0], Y[1], Y[2], Y[3]);
end
endmodule
```

**DATAFLOW MODELING:**

```
module Demultiplexer_1_to_4_assign(output [3:0] Y, input [1:0] A, input din);
assign Y[0] = din & (~A[0]) & (~A[1]);
assign Y[1] = din & (~A[1]) & A[0];
assign Y[2] = din & A[1] & (~A[0]);
assign Y[3] = din & A[1] & A[0];
endmodule
```

**TEST BENCH:**

```
module Demultiplexer_1_to_4_assign_tb;
wire [3:0] Y;
reg [1:0] A;
reg din;
Demultiplexer_1_to_4_assign I0 (Y, A, din);
initial begin
din = 1;
A[1] = 0; A[0] = 0;
 #1 A[1] = 0; A[0] = 1;
 #1 A[1] = 1; A[0] = 0;
 #1 A[1] = 1; A[0] = 1;
end
initial begin
 $monitor("%t| Din = %d| A[1] = %d| A[0] = %d| Y[0] = %d| Y[1] = %d| Y[2] = %d| Y[3] =
%d", $time, din, A[1], A[0], Y[0], Y[1], Y[2], Y[3]);
end
endmodule
```

**RESULT:**

Verilog code for the mux demux and its test bench for verification is written, the waveform is observed.

**Experiment No 5:**

## DESIGN AND SIMULATION OF 8-BIT ADDER AND MULTIPLIER

**AIM:**

To design 8-bit adder and multiplier using Nclaunch.

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

**ADDER:**

An **adder** is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the arithmetic logic unit or **ALU**. They are also used in other parts of the processor, where they are used to calculate addressed, table indices, increment and decrement operators and similar operations.

**PROGRAM:**

**8- BIT ADDER:**

**VERILOG CODE:**

```
module adder(
input [7:0] a,
input [7:0] b,
input cin,
output reg [7:0] sum,
output cout
);

always @(a,b,cin)
begin
	{cout,sum}= a+b+cin;
end
endmodule
```

**TESTBENCH:**

```
module tb_adder;

   reg [7:0 ] A;
    reg [7:0 ] B;
    reg CIN;
    wire [7:0] SUM;
    wire  COUT;

    adder ADD (.a(A) ,.b(B),.sum(SUM),.cin(CIN),.cout(COUT));
    initial begin
```

```
        A =8'b0;
        B= 8'b0;
        CIN = 1'b0;

#10 A=00000001; B=00000011; C=1;
#10 A=00000011; B=00001111; C=0;
#10 A=00111001; B=01100011; C=1;
#10 A=11100001; B=00011011; C=0;
#10 A=00001111; B=01110011; C=0;
#10 A=11111111; B=00000011; C=1;
#10 A=00000011; B=11111111; C=1;
end

always @(SUM,COUT)
    $display( "time =%0t \tINPUT VALUES: \t A =%b B =%b CIN =%b \t output value
SUM=%b COUT =%b ",$time,A,B,CIN,SUM,COUT);
endmodule
```

**MULTIPLIER:**
**VERILOG CODE:**

```
module mult4bit(a,b,p);
input [3:0] a,b;
wire [3:0] m0;
wire [4:0] m1;
wire [5:0] m2;
wire [6:0] m3;

wire [7:0] s1,s2,s3;
output[7:0] p;

assign m0={4{a[0]}}&b[3:0];
assign m1={4{a[1]}}&b[3:0];
assign m2={4{a[2]}}&b[3:0];
assign m3={4{a[3]}}&b[3:0];
assign s1=m0 + (m1<<1);
assign s2=s1 + (m2<<2);
assign s3=s2 + (m3<<3);
assign p = s3;
endmodule
```

**TEST BENCH:**

```
module tb;
reg[3:0] A, B;
wire[7:0] R;
mult4bit mm1(.a(A),.b(B),.p(R));
initial begin
A=4’b0011; B=4’d3;
#10
```

```
A=7; B=1;
#10
A=4’hA; b=4’h10;
end
endmodule
```

**RESULT:**

Verilog code for the adder multiplier and its test bench for verification is written, the waveform is observed.

**Experiment No 6:**

**DESIGN AND SIMULATION OF PRBS GENERATOR AND ACCUMULATOR USING NCLAUNCH**

**AIM:**

To design the Pseudo random binary sequence generator and Acculumator in Verilog HDL Using Nclaunch

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

**PRBS:**

A pseudorandom binary sequence (PRBS) is a binary sequence that, while generated with a deterministic algorithm, is difficult to predict and exhibits statistical behaviour similar to a truly random sequence.

**ACCUMULATOR:**

In accumulator differs from a counter in the nature of the operands of the add and subtract operation:

- In a counter, the destination and first operand is a signal or variable and the other operand is a constant equal to 1:
  A <= A + 1.
- In an accumulator, the destination and first operand is a signal or variable, and the second operand is either:
  - a signal or variable: A <= A + B.
  - a constant not equal to 1: A <= A + Constant.

An inferred accumulator can be up, down or updown. For an updown accumulator, the accumulated data may differ between the up and down mode:

```
...
if updown = '1' then
  a <= a + b;
else
  a <= a - c;
...
```

**PROGRAM:**

**PRBS:**

```
module prbs (rand, clk, reset);
input clk, reset;
output rand;
wire rand;
reg [3:0] temp;
always @ (posedge reset)
```

```
begin
temp <= 4'hf;
end
always @ (posedge clk)
begin
if (~reset)
begin
temp <= {temp[0]^temp[1],temp[3],temp[2],temp[1]};
end
end
assign rand = temp[0];
endmodule
```

**TESTBENCH:**

```
module main;
reg clk, reset;
wire rand;
prbs pr (rand, clk, reset);
initial
begin
forever
begin clk <= 0;
 #5 clk <= 1;
#5 clk <= 0;
end
end
initial
begin
reset = 1;
 #12 reset = 0;
#90 reset = 1;
 #12 reset = 0;
end
endmodule
```

**ACCUMULATOR:**

```
module accumod (in, acc, clk, reset);
input [7:0] in;
input clk, reset;
output [7:0] acc;
reg [7:0] acc;
always@(clk) begin
if(reset)
acc <= 8'b00000000;
else
acc <= acc + in;
end
endmodule
```

**TEST BENCH**

```
module accumt_b;
reg [7:0] in;
reg clk;
reg reset;
wire [7:0] acc;
accumod uut ( .in(in), .acc(acc),.clk(clk),.reset(reset) );
initial begin
#5 reset<=1'b1;
#5 reset<=1'b0;
clk =1'b0;
in = 8'b00000001;
#50 in = 8'b00000010;
#50 in = 8'b00000011;
end
always #10 clk = ~clk;
initial#180 $stop;
endmodule
```

**RESULT:**

Verilog code for the PRBS Generator & Accumulator and its test bench for verification is written, the waveform is observed.

**Experiment No 7:**

**DESIGN AND SIMULATION OF PARITY GENERATOR USING NCLAUNCH**

**AIM:**

To design the parity generator in Verilog HDL Using Nclaunch

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

**PARITY GENERATOR:**

A Parity Generator is a combinational logic circuit that generates the parity bit in the transmitter. The sum of the data bits and parity bits can be even or odd. In even parity, the added parity bit will make the total number of 1s an even number, whereas in odd parity, the added parity bit will make the total number of 1s an odd number.

**PROGRAM:**

**PARITY GENERATOR:**

```
module parityGenerator(DOUT, parity, DIN);
output [4:0] DOUT;
output parity;
input [3:0] DIN;
assign parity = DIN[0] ^ DIN[1] ^ DIN[2] ^ DIN[3];
assign DOUT = { DIN, parity };
endmodule
```

**TEST BENCH:**

```
module parityGeneratorTb;
wire [4:0] DOUT;
wire parity;
reg [3:0] DIN;

parityGenerator pgrtr(DOUT, parity, DIN);
initial
begin
  $display("RSLT\tD is parity with DOUT");
  DIN = 4'b0011; #10;
  if ( (parity == 0) && (DOUT === { DIN, 1'b0 }) )
    $display("PASS\t%p is %p with %p", DIN, parity, DOUT);
  else
    $display("FAIL\t%p is %p with %p", DIN, parity, DOUT);
  DIN = 4'b1011; #10;
  if ( (parity == 1) && (DOUT === { DIN, 1'b1 }) )
```

```
    $display("PASS\t%p is %p with %p", DIN, parity, DOUT);
  else
    $display("FAIL\t%p is %p with %p", DIN, parity, DOUT);
  DIN = 4'b1111; #10;
  if ( (parity == 0) && (DOUT === { DIN, 1'b0 }) )
    $display("PASS\t%p is %p with %p", DIN, parity, DOUT);
  else
    $display("FAIL\t%p is %p with %p", DIN, parity, DOUT);
 end
endmodule
```

**RESULT:**

Verilog code for the Parity generator and its test bench for verification is written, the waveform is observed.

**Experiment No 8:**

**DESIGN AND SIMULATION OF SISO SHIFT REGISTER USING NCLAUNCH**

**AIM:**

To design the SISO Shift register in Verilog HDL Using Nclaunch

**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

**SHIFT REGISTERS:**

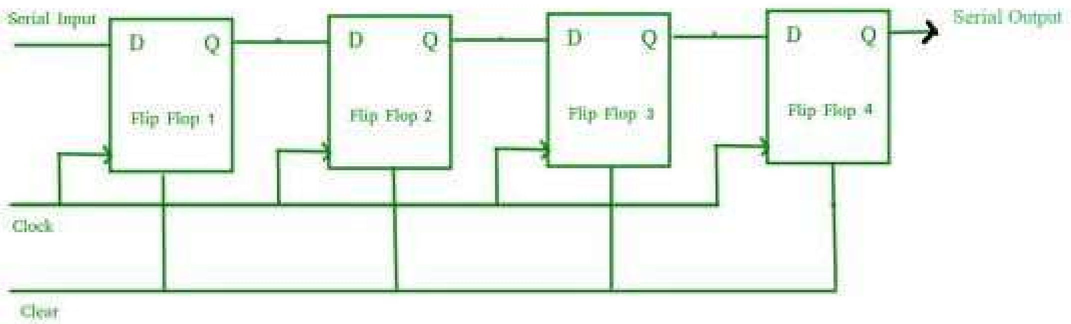
The Shift Register is another type of sequential logic circuit that can be used for the storage or the transfer of binary data. This sequential device loads the data present on its inputs and then moves or "shifts" it to its output once every clock cycle, hence the name **Shift Register**.

**Serial-in to Parallel-out (SIPO)** - the register is loaded with serial data, one bit at a time, with the stored data being available at the output in parallel form.

**Serial-in to Serial-out (SISO)** - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction under clock control.

**Parallel-in to Serial-out (PISO)** - the parallel data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.

**Parallel-in to Parallel-out (PIPO)** - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse.




**SISO:**

```
module sisomod(clk,clear,si,so);
input clk,si,clear;
output so;
reg so;
```

```
reg [3:0] tmp;
always @(posedge clk )
begin
if (clear)
tmp <= 4'b0000;
else
tmp <= tmp << 1;
tmp[0] <= si;
so = tmp[3];
end
endmodule
```

**TEST BENCH**

```
module sisot_b;
reg clk;
reg clear;
reg si;
wire so;
sisomod uut (.clk(clk), .clear(clear),.si(si),.so(so));
initial begin
clk = 0;
clear = 0;
si = 0;
#5 clear=1'b1;
#5 clear=1'b0;
#10 si=1'b1;
#10 si=1'b0;
#10 si=1'b0;
#10 si=1'b1;
#10 si=1'b0;
#10 si=1'bx;
end
always #5 clk = ~clk;
initial #150 $stop;
endmodule
```

**RESULT:**

Verilog code for the SISO Shift register and its test bench for verification is written, the waveform is observed.

**Experiment No 9:**

**DESIGN AND SIMULATION OF 4 BIT MAGNITUDE COMPARATOR USING NCLAUNCH**

**AIM:**

To design the 4 bit magnitude comparator in Verilog HDL Using Nclaunch

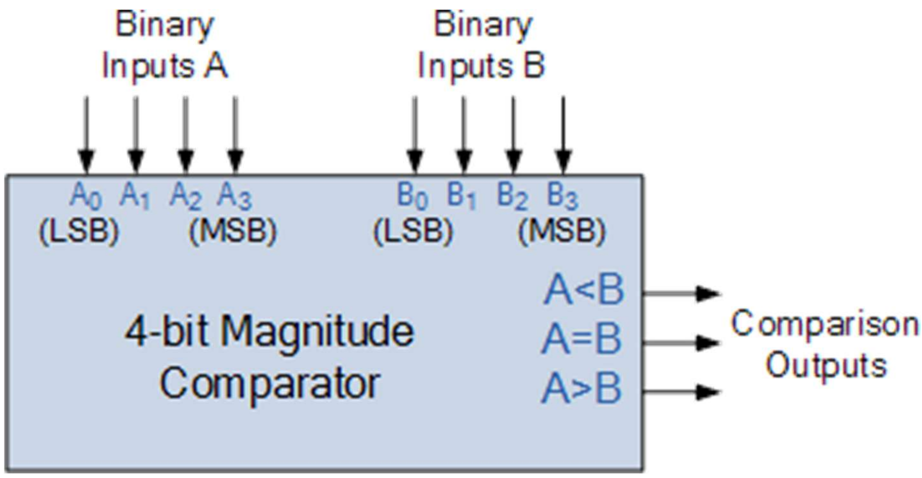
**APPARATUS REQUIRED:**

Nclaunch – Cadence Tool

**THEORY:**

A magnitude digital Comparator is a combinational circuit that **compares two digital or binary numbers** in order to find out whether one binary number is equal, less than or greater than the other binary number. We logically design a circuit for which we will have two inputs one for A and other for B and have three output terminals, one for A > B condition, one for A = B condition and one for A < B condition.

A comparator used to compare two binary numbers each of four bits is called a 4-bit magnitude comparator. It consists of eight inputs each for two four bit numbers and three outputs to generate less than, equal to and greater than between two binary numbers.




**PROGRAM:**

**4BIT MAGNITUDE COMPARATOR:**

```
module comparator(a,b,eq,lt,gt);
input [3:0] a,b;
output reg eq,lt,gt;
always @(a,b)
begin
 if (a==b)
 begin
  eq = 1'b1;
  lt = 1'b0;
```

```
 gt = 1'b0;
 end
 else if (a>b)
 begin
  eq = 1'b0;
 lt = 1'b0;
 gt = 1'b1;
 end
 else
 begin
  eq = 1'b0;
 lt = 1'b1;
 gt = 1'b0;
 end
end
endmodule
```

**TEST BENCH**

```
module comparator_tst;
reg [3:0] a,b;
wire eq,lt,gt;
comparator DUT (a,b,eq,lt,gt);
initial
begin
 a = 4'b1100;
 b = 4'b1100;
 #10;
 a = 4'b0100;
 b = 4'b1100;
 #10;
 a = 4'b1111;
 b = 4'b1100;
 #10;
 a = 4'b0000;
 b = 4'b0000;
 #10;
 $stop;
end
endmodule
```

**RESULT:**

Verilog code for the 4bit magnitude comparator and its test bench for verification is written, the waveform is observed.

**Experiment No 10:**

**DESIGN AND SIMULATION OF FLIPFLOP USING NCLAUNCH**

**AIM:**

To design the flipflop in Verilog HDL Using Nclaunch

**APPARATUS REQUIRED:**

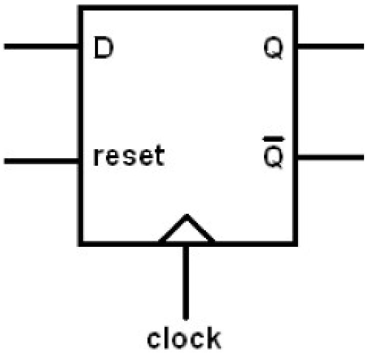
Nclaunch – Cadence Tool

**THEORY:**

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

**PROGRAM:**

**D FLIPFLOP:**



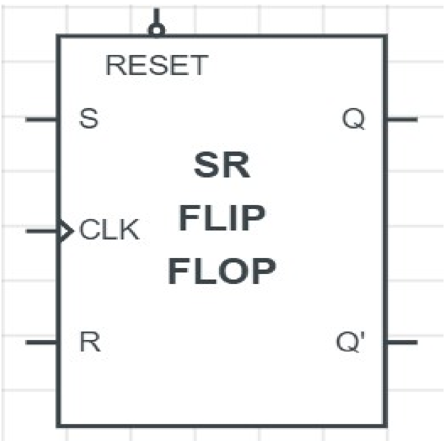

```
Module DFF( Q,Qbar,D,Clk,Reset);
output  reg  Q;
output   Qbar;
input  D,Clk, Reset;
assign Qbar = ~Q;
always @(posedge Clk)
begin
 if (Reset == 1'b1) //If not at reset
 Q = 1'b0;
 else
Q = D;
end
 endmodule
```

**TEST BENCH:**

```
module DFF_tb;
```

```
 // Inputs
 reg D;
 reg Clk;
 reg Reset;
 // Outputs
 wire Q;
 wire Qbar;
 // Instantiate the Unit Under Test (UUT)
 DFF uut (.Q(Q), .Qbar(Qbar), .D(D), .Clk(Clk),.Reset(Reset));
 initial begin
  // Initialize Inputs
  D  = 1'b0;
  Clk  = 1'b0;
  Reset   = 1'b1;
  // Wait 100 ns for global reset to finish
  #100;
  // Add stimulus here
  Reset = 1'b0;
  #20;
  forever #40 D = ~ D;
 end
   always #10 Clk = ~Clk;
endmodule
```

**SR FLIPFLOP:**



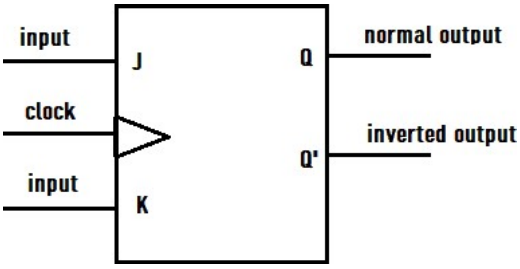

```
module SR_ff(s,r,clk,reset,q,q_bar);
input s,r,clk,reset;
output q,q_bar;
wire s,r,clk;
reg q,q_bar;
always @(posedge clk) begin
if (reset) begin
q=1'b0;
q_bar=1'b1;
end
else
```

```
begin
case({s,r})
{1'b0,1'b0}: begin q=q;q_bar=q_bar; end
{1'b0,1'b1}: begin q=1'b0;q_bar=1'b1; end
{1'b1,1'b0}: begin q=1'b1;q_bar=1'b0; end
{1'b1,1'b1}: begin q=1'bx; q_bar=1'bx; end
endcase
end
end
endmodule
```

**TEST BENCH:**

```
module SR_ff_tb;
 reg clk;
reg reset;
reg s,r;
wire q;
wire qb;
SR_ff srflipflop( .clk(clk), .reset(reset), .s(s), .r(r), .q(q), .q_bar(qb) );
 initial begin
$monitor(clk,s,r,q,qb,reset);
s = 1'b0;
r = 1'b0;
reset = 1;
clk=1;
#10
reset=0;
s=1'b1;
r=1'b0;
#100
reset=0;
s=1'b0;
r=1'b1;
#100
reset=0;
s=1'b1;
r=1'b1;
#100
reset=0;
s=1'b0;
r=1'b0;
#100
reset=1;
s=1'b1;
r=1'b0;
end
always #25 clk <= ~clk;
endmodule
```

**JK FLIPFLOP:**




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

**Testbench**

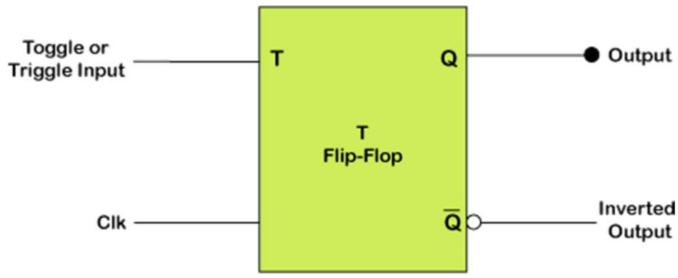
```
module tb_jk;
reg j,k,clk;
wire q;
always #5 clk = ~clk;

 jk_ff    jk0 ( .j(j),
              .k(k),
              .clk(clk),
              .q(q));

 initial begin
    j <= 0;
    k <= 0;

    #5 j <= 0;
       k <= 1;
    #20 j <= 1;
        k <= 0;
    #20 j <= 1;
        k <= 1;
    #20 $finish;
 end
```

**T FLIPFLOP:**




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

**TEST BENCH**

```
module tb;
  reg clk;
  reg rstn;
  reg t;

  tff u0 (  .clk(clk),
            .rstn(rstn),
            .t(t),
          .q(q));

  always #5 clk = ~clk;

  initial begin
    {rstn, clk, t} <= 0;

    $monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time, rstn, t, q);
    repeat(2) @(posedge clk);
    rstn <= 1;

    for (integer i = 0; i < 20; i = i+1) begin
      reg [4:0] dly = $random;
      #(dly) t <= $random;
    end
    #20 $finish;
end
endmodule
```

**RESULT:**

Verilog code for the flipflop and its test bench for verification is written, the waveform is observed.