ECE5001 – Digital System Architecture and Design using HDL

Master of Technology

In

(VLSI DESIGN)

Submitted

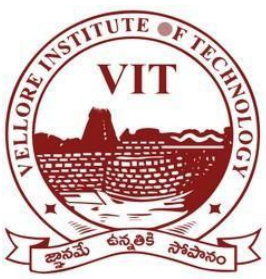
to

Dr. Nalluri Purnachand

By

Name: …….

Roll Number: ……..



VIT-AP UNIVERSITY

LIST OF EXPERIMENTS

S.No	Experiment Name
1	Adders and Subtractors
2	Decoder, Priority Encoder and Mux and De mux
3	N Bit comparator and N bit Adder
4	Flip Flops
5	Universal Counter
6	SRAM Design
7	Universal Shift Registers
8	Floating Point Adder
9	Floating Point Multiplier
10	Boot Multiplier

1. Adders and Subtractors

AIM: Implement all the Adders and Subtractors using FPGA.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

1. Half Adder

A Half Adder is a combinational logic circuit that performs the addition of two single-bit binary inputs. It produces:

- Sum (S) , Carry (C)

It does not consider any carry input from a previous stage, hence the name *half* adder.

Logic Equations

- $\text{Sum} = A \oplus B$, $\text{Carry} = A \cdot B$

Truth Table

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2.Full Adder :

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

The C-OUT is also known as the majority 1's detector, whose output goes high when more than one input is high.

A full adder logic is designed in such a manner that can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to another.

Table:-

INPUT			**OUTPUT**	
A	**B**	**C-IN**	**Sum**	**C-OUT**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**1**
1	**1**	**0**	**0**	**1**
1	**1**	**1**	**1**	**1**

Half Subtractor:

A half subtractor is a digital logic circuit that performs the binary subtraction of two single-bit binary numbers. It has two inputs, A and B, and two outputs, Difference and Borrow. The Difference output represents the result of subtracting B from A, while the Borrow output indicates whether a borrow is needed when A is smaller than B.

Truth Table:

Inputs		**Outputs**	
A	**B**	**D (Difference)**	**B (Borrow)**
0	**0**	**0**	**0**
0	**1**	**1**	**1**
1	**0**	**1**	**0**
1	**1**	**0**	**0**

Full Subtractor:

A Full Subtractor is a combinational circuit used to perform binary subtraction. It has three inputs:

- A (Minuend)
- B (Subtrahend)
- B-IN (Borrow-in from the previous stage)

It produces two outputs:

- Difference (D): The result of the subtraction.
- Borrow-out (B-OUT): Indicates if a borrow is needed for the next stage.

Truth Table of Full Subtractor

Input			Output	
A	**B**	**Bin**	**D**	**Bout**
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**

CODE :-

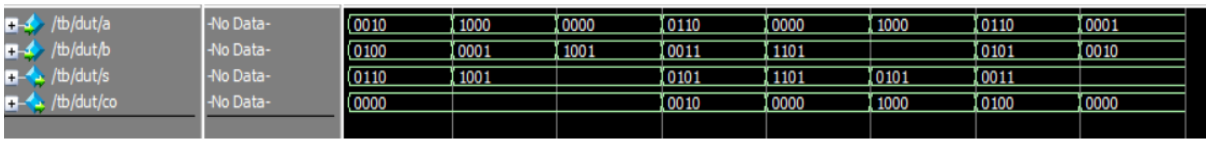
```
module half_adder_gatelevel (
    input a, b,
    output sum, carry
);
  xor (sum, a, b);
  and (carry, a, b);
endmodule
module half_adder_dataflow (
    input a, b,
    output sum, carry
);
```

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

module half_adder_behavioral (
    input a, b,
    output reg sum, carry
);
  always @(*) begin
    sum   = a ^ b;
    carry = a & b;
  end
endmodule

`timescale 1ns/1ps
module tb_half_adder();
  reg a, b;
  wire sum_g, carry_g;
  wire sum_d, carry_d;
  wire sum_b, carry_b;
  // DUT Instances
  half_adder_gatelevel   U1 (.a(a), .b(b), .sum(sum_g), .carry(carry_g));
```

```
 half_adder_dataflow    U2 (.a(a), .b(b), .sum(sum_d), .carry(carry_d));

 half_adder_behavioral  U3 (.a(a), .b(b), .sum(sum_b), .carry(carry_b));

 initial begin

  $display("Time | A B | GateLevel(Sum Carry) | Dataflow(Sum Carry) | Behavioral(Sum Carry)");

  $monitor("%4t | %b %b |       %b     %b      |       %b     %b      |       %b     %b",

          $time, a, b,

          sum_g, carry_g,

          sum_d, carry_d,

          sum_b, carry_b);

  // Apply all input combinations

  a=0; b=0; #10;

  a=0; b=1; #10;

  a=1; b=0; #10;

  a=1; b=1; #10;

  $finish;

 end

endmodule
```

OUTPUT :

```
# Time | A B | GateLevel(Sum Carry)
#    0 | 0 0 |           0   0
# 10000 | 0 1 |           1   0
# 20000 | 1 0 |           1   0
# 30000 | 1 1 |           0   1
```



Full Adder :

1. Gate-Level Full Adder

```
module full_adder_gatelevel (
    input a, b, cin,
    output sum, cout
);
  wire s1, c1, c2;
  // XOR for sum
  xor (s1, a, b);
  xor (sum, s1, cin);
  // Carry logic
  and (c1, a, b);
  and (c2, s1, cin);
  or  (cout, c1, c2);
endmodule
```

2. Dataflow Full Adder

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

3. Behavioral Full Adder

```
module full_adder_behavioral (
    input a, b, cin,
    output reg sum, cout
);
  always @(*) begin
    {cout, sum} = a + b + cin;
  end
endmodule
```

4. Parameterized + Generate Full Adder

This version creates **N-bit Full Adder** using **1-bit full adders**.

```
module full_adder_param #(
    parameter N = 4   // default 4-bit adder
```

```
)(
    input  [N-1:0] a, b,
    input          cin,
    output [N-1:0] sum,
    output         cout
);
  wire [N:0] carry;
  assign carry[0] = cin;
  genvar i;
  generate
    for (i = 0; i < N; i = i + 1) begin : FA_LOOP
      full_adder_dataflow FA (
        .a(a[i]), .b(b[i]), .cin(carry[i]),
        .sum(sum[i]), .cout(carry[i+1])
      );
    end
  endgenerate
  assign cout = carry[N];
endmodule
```

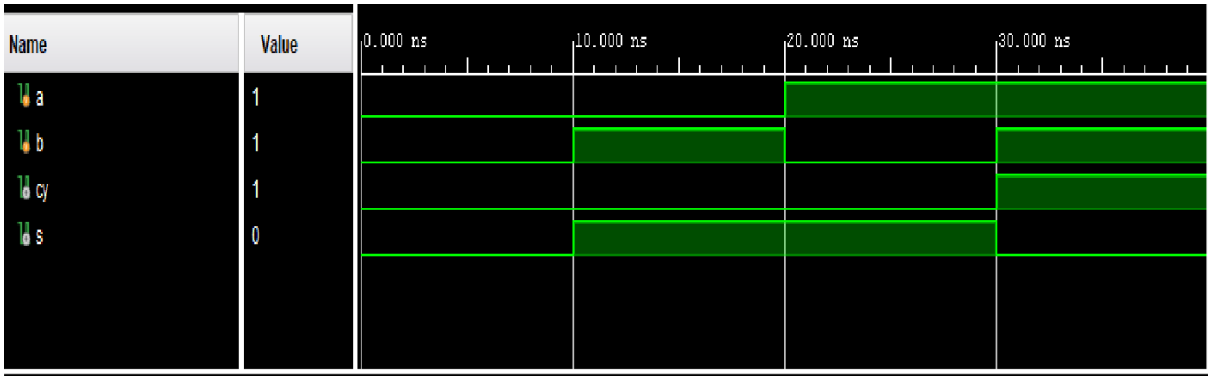
Common Testbench for All Models

```
`timescale 1ns/1ps
```

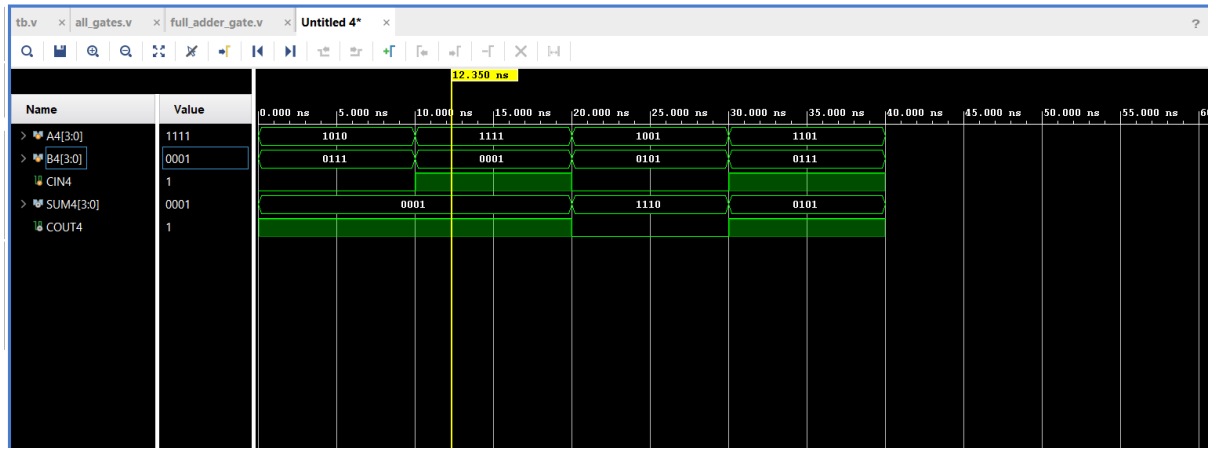
```
module tb_full_adder;
  reg a, b, cin;
  wire sum_gate, cout_gate;
  wire sum_data, cout_data;
  wire sum_behav, cout_behav;
  // DUTs
  full_adder_gatelevel   U1 (.a(a), .b(b), .cin(cin), .sum(sum_gate),  .cout(cout_gate));
  full_adder_dataflow    U2 (.a(a), .b(b), .cin(cin), .sum(sum_data),  .cout(cout_data));
  full_adder_behavioral  U3 (.a(a), .b(b), .cin(cin), .sum(sum_behav), .cout(cout_behav));
  // Parameterized 4-bit example
  reg [3:0] A4, B4;
  reg CIN4;
  wire [3:0] SUM4;
  wire COUT4;
  full_adder_param #(.N(4)) U4 (
    .a(A4), .b(B4), .cin(CIN4), .sum(SUM4), .cout(COUT4)
  );
  initial begin
    $display("Full Adder Testbench\n");
    $monitor("T=%0t | A=%b B=%b Cin=%b || Gate: S=%b Cout=%b | Data: S=%b Cout=%b | Behav:
S=%b Cout=%b",
             $time, a, b, cin,
```

```
        sum_gate, cout_gate,
        sum_data, cout_data,
        sum_behav, cout_behav);
    // 1-bit test
    a=0; b=0; cin=0; #10;
    a=0; b=1; cin=0; #10;
    a=1; b=0; cin=0; #10;
    a=1; b=1; cin=0; #10;
    a=0; b=0; cin=1; #10;
    a=0; b=1; cin=1; #10;
    a=1; b=0; cin=1; #10;
    a=1; b=1; cin=1; #10;
    // 4-bit param test
    A4=4'b1010; B4=4'b0111; CIN4=0; #10;
    $display("4-bit Param FA: A=%b B=%b CIN=%b -> SUM=%b COUT=%b", A4, B4, CIN4,
SUM4, COUT4);
    A4=4'b1111; B4=4'b0001; CIN4=1; #10;
    $display("4-bit Param FA: A=%b B=%b CIN=%b -> SUM=%b COUT=%b", A4, B4, CIN4,
SUM4, COUT4);
    $finish;
  end
endmodule
```

Output (Half Adder) :-



Output (Full Adder):-



Half Subtractor :-

1. Gate-Level Half Subtractor

```
module half_subtractor_gatelevel (
    input a, b,
    output diff, borrow
);
  xor (diff, a, b);     // Difference
  not (na, a);
  and (borrow, na, b);  // Borrow = ~a & b
```

```
endmodule
```

2. Dataflow Half Subtractor

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

3. Behavioral Half Subtractor

```
module half_subtractor_behavioral (
    input a, b,
    output reg diff, borrow
);
  always @(*) begin
    diff   = a ^ b;
    borrow = (~a) & b;
  end
endmodule
```

4. Parameterized + Generate Half Subtractor

Here we extend to **N-bit Half Subtractor** using **1-bit half subtractors**.

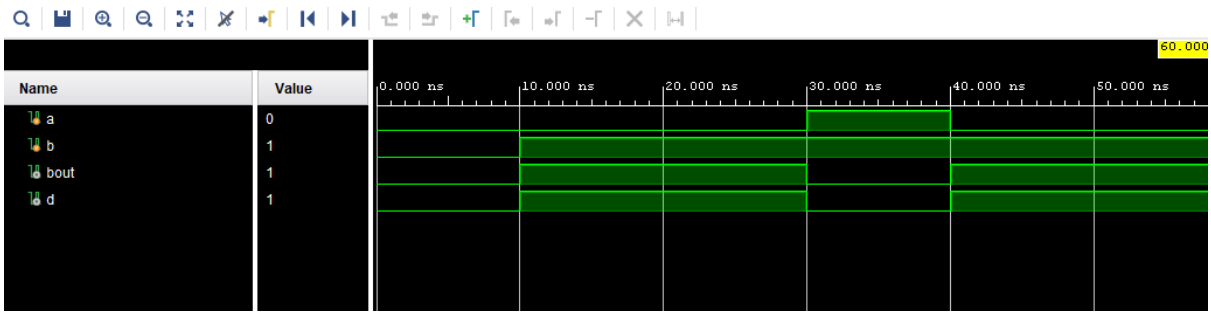
```
module half_subtractor_param #(
    parameter N = 4
)(
    input  [N-1:0] a, b,
    output [N-1:0] diff, borrow
);
  genvar i;
  generate
    for (i = 0; i < N; i = i + 1) begin : HS_LOOP
      half_subtractor_dataflow HS (
        .a(a[i]), .b(b[i]),
        .diff(diff[i]), .borrow(borrow[i])
      );
    end
  endgenerate
endmodule
```

Test Bench for All Levels :-

```
`timescale 1ns/1ps
module tb_half_subtractor_gatelevel;
```

```
  reg a, b;
  wire diff, borrow;
  // DUT instance
  half_subtractor_gatelevel UUT (
    .a(a), .b(b),
    .diff(diff),
    .borrow(borrow)
  );
  initial begin
    $display("Gate-Level Half Subtractor");
    $display("Time | A B | DIFF BORROW");
    $monitor("%5t | %b %b |  %b    %b", $time, a, b, diff, borrow);
    a=0; b=0; #10;
    a=0; b=1; #10;
    a=1; b=0; #10;
    a=1; b=1; #10;
    $finish;
  end
endmodule
```

Output:-



Full Subtractor :-

```
module full_subtractor_gate (
    input  a, b, bin,
    output diff, bout );
  wire w1, w2, w3;
  // diff = a ^ b ^ bin
  xor (w1, a, b);
  xor (diff, w1, bin);
  // bout = (~a & b) | (~(a ^ b) & bin)
  and (w2, ~a, b);
  and (w3, ~w1, bin);
  or  (bout, w2, w3);
endmodule


module full_subtractor_dataflow (
    input  a, b, bin,
    output diff, bout
);
  assign diff = a ^ b ^ bin;
  assign bout = (~a & b) | (~(a ^ b) & bin);
```

```
endmodule


module full_subtractor_behavioral (  input  a, b, bin,
    output reg diff, bout
);
 always @(*) begin
    diff = a ^ b ^ bin;
    bout = (~a & b) | (~(a ^ b) & bin);
  end
endmodule
```

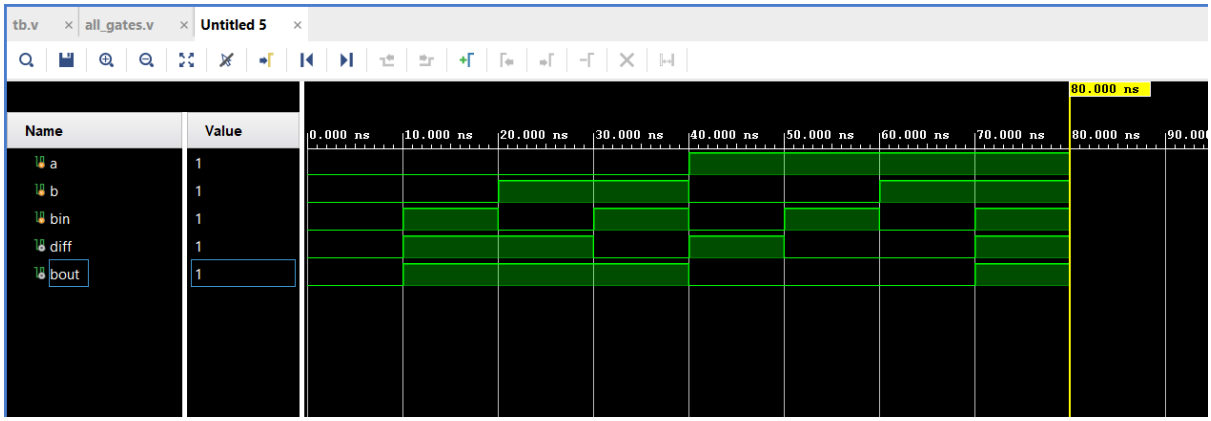
Testbench:-

```
`timescale 1ns/1ps;
module tb_full_subtractor_behavioral;
  reg a, b, bin;
  wire diff, bout;
  full_subtractor_behavioral UUT (.a(a), .b(b), .bin(bin), .diff(diff), .bout(bout));
  initial begin
    $display("Behavioral Full Subtractor");
    $display("time  a b bin | diff bout");
    $monitor("%t %b %b %b | %b %b", $time, a, b, bin, diff, bout);
    a=0; b=0; bin=0; #10;
    a=0; b=0; bin=1; #10;
    a=0; b=1; bin=0; #10;
    a=0; b=1; bin=1; #10;
    a=1; b=0; bin=0; #10;
```

```
    a=1; b=0; bin=1; #10;
    a=1; b=1; bin=0; #10;
    a=1; b=1; bin=1; #10;
    $finish;
  end
endmodule
```

Results:-

```
Behavioral Full Subtractor
time   a b bin | diff bout
                   0 0 0 0 | 0 0
               10000 0 0 1 | 1 1
               20000 0 1 0 | 1 1
               30000 0 1 1 | 0 1
               40000 1 0 0 | 1 0
               50000 1 0 1 | 0 0
               60000 1 1 0 | 0 0
               70000 1 1 1 | 1 1
```



2. Decoder, Priority Encoder and Mux and De-Mux

AIM: Implement all the Decoder, Priority Encoder and Multiplexer, De-Mux using FPGA.

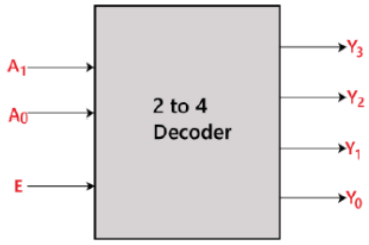
Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

1. Decoder :

The combinational circuit that change the binary information into 2N output lines is known as Decoders. The binary information is passed in the form of N input lines. The output lines define the 2N-bit code for the binary information. In simple words, the Decoder performs the reverse operation of the Encoder. At a time, only one input line is activated for simplicity. The produced 2N-bit output code is equivalent to the binary information.

Block Diagram:

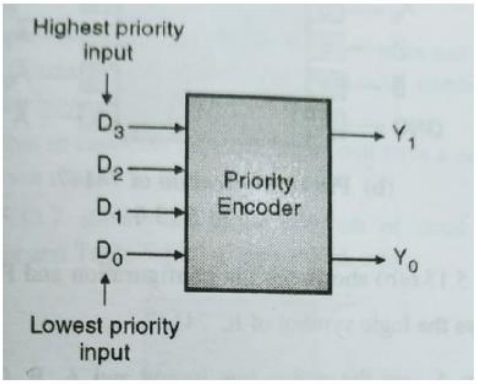


Truth Table:

Enable	INPUTS		OUTPUTS			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
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

2. Priority Encoder :

An Encoder is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2n input lines and ‘n’ output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2n input lines with ‘n’ bits. It is optional to represent the enable signal in encoders.

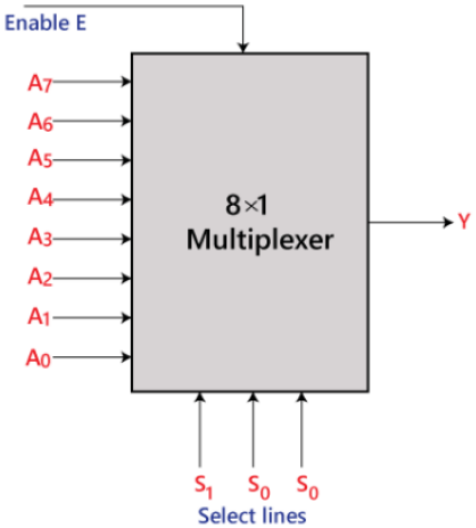


Inputs				Outputs		
I_0	I_1	I_2	I_3	X	Y	IST
1	x	x	x	0	0	1
0	1	x	x	0	1	1
0	0	1	x	1	0	1
0	0	0	1	1	1	1
0	0	0	0	x	x	0

3. Multiplexer:

A multiplexer is a combinational circuit that has 2n input lines and a single output line. Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

In the 8 to 1 multiplexer, there are total eight inputs, i.e., A0, A1, A2, A3, A4, A5, A6, and A7, 3 selection lines, i.e., S0, S1and S2 and single output, i.e., Y. On the basis of the combination of inputs that are present at the selection lines S0, S1, and S2, one of these 8 inputs are connected to the output. The block diagram and the truth table of the 8×1 multiplexer are given below.

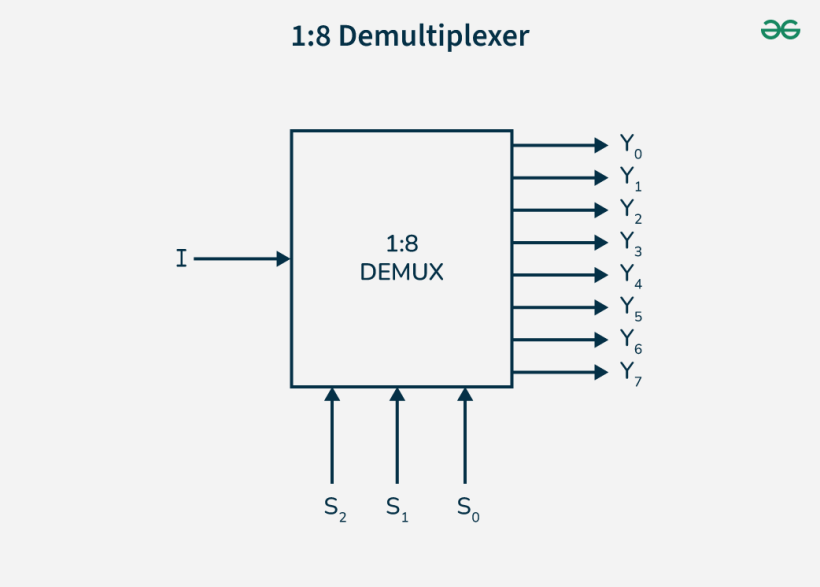


Truth Table:

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

Demultiplexer :-

A De-mux (Demultiplexer)is a digital circuit that takes a single input signal and routes it to one of several output lines, acting as a data distributor or a single-to-many switch, the reverse of a multiplexer (MUX), using control/select lines to choose the destination output. De-muxes are vital in communications and computing for directing data from one source to multiple destinations, like converting serial data to parallel or routing signals in networks .



Truth Table: -

Selection Inputs			**Outputs**							
S2	**S1**	**S0**	**Y7**	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**
0	0	0	0	0	0	0	0	0	0	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0
1	1	1	**I**	0	0	0	0	0	0	0

Code: -

```
module decoder3x8(
   input A, B, C,
   output Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7
);
   wire Abar, Bbar, Cbar;
   // Generate complements
   not (Abar, A);
   not (Bbar, B);
   not (Cbar, C);
   // AND gates for each output line
   and (Y0, Abar, Bbar, Cbar);  // 000
   and (Y1, Abar, Bbar, C);     // 001
   and (Y2, Abar, B, Cbar);     // 010
   and (Y3, Abar, B, C);        // 011
   and (Y4, A, Bbar, Cbar);     // 100
   and (Y5, A, Bbar, C);        // 101
   and (Y6, A, B, Cbar);        // 110
   and (Y7, A, B, C);           // 111
endmodule
```

Data Flow :-

```
module tb_oct_to_bin_dataflow;
   reg [7:0] d;
   wire [2:0] bin;
   // Instantiate the gate-level encoder
   oct_to_bin_dataflow uut (.d(d), .bin(bin));
```

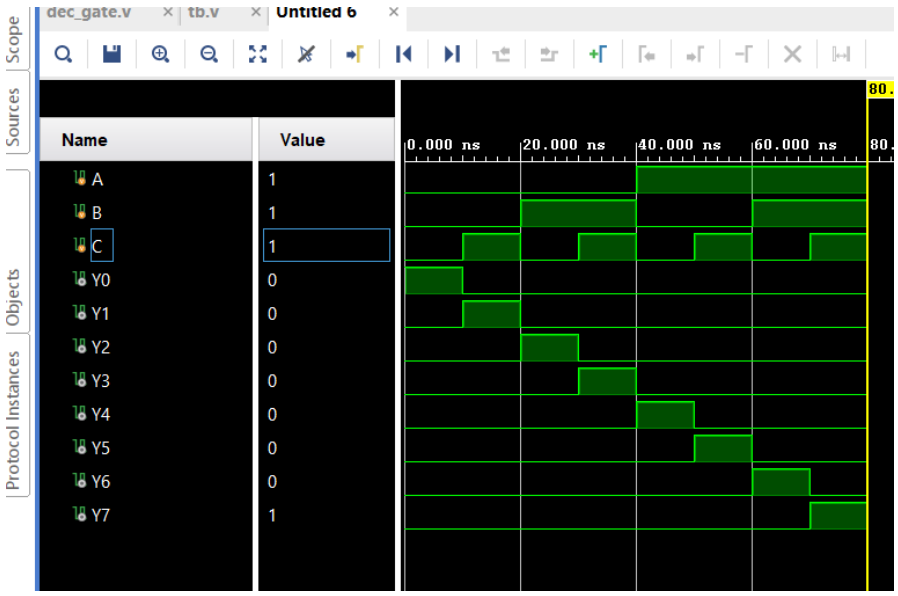
```
    // Monitor signals continuously
    initial begin
        $monitor("Time=%0t | Input d=%b | Binary Output=%b", $time, d, bin);
    end
    // Apply test vectors
    initial begin
      $display("Octal-to-Binary Data flow Level Test");
        d = 8'b00000001; #10;
        d = 8'b00000010; #10;
        d = 8'b00000100; #10;
        d = 8'b00001000; #10;
        d = 8'b00010000; #10;
        d = 8'b00100000; #10;
        d = 8'b01000000; #10;
        d = 8'b10000000; #10;
        $finish;
    end
endmodule
```

Behavioural:

```
module oct_to_bin_beh(
    input [7:0] d,
    output reg [2:0] bin
```

```
);
    always @(*) begin
        case (1'b1)
            d[0]: bin = 3'd0;
            d[1]: bin = 3'd1;
            d[2]: bin = 3'd2;
            d[3]: bin = 3'd3;
            d[4]: bin = 3'd4;
            d[5]: bin = 3'd5;
            d[6]: bin = 3'd6;
            d[7]: bin = 3'd7;
            default: bin = 3'd0;
        endcase
    end
endmodule
module tb_oct_to_bin_beh;
    reg [7:0] d;
    wire [2:0] bin;
    oct_to_bin_beh uut (.d(d), .bin(bin));
    initial begin
        $display("Octal-to-Binary Behavioral Test");
        d = 8'b00000001; #10;
        d = 8'b00000010; #10;
        d = 8'b00000100; #10;
        d = 8'b00001000; #10;
```

```
        d = 8'b10000000; #10;
        $finish;
    end
        initial begin
       #10 $monitor("Time=%0t | Input d=%b | Binary Output=%b", $time, d, bin);
    end
endmodule
```



Priority Encoder :-

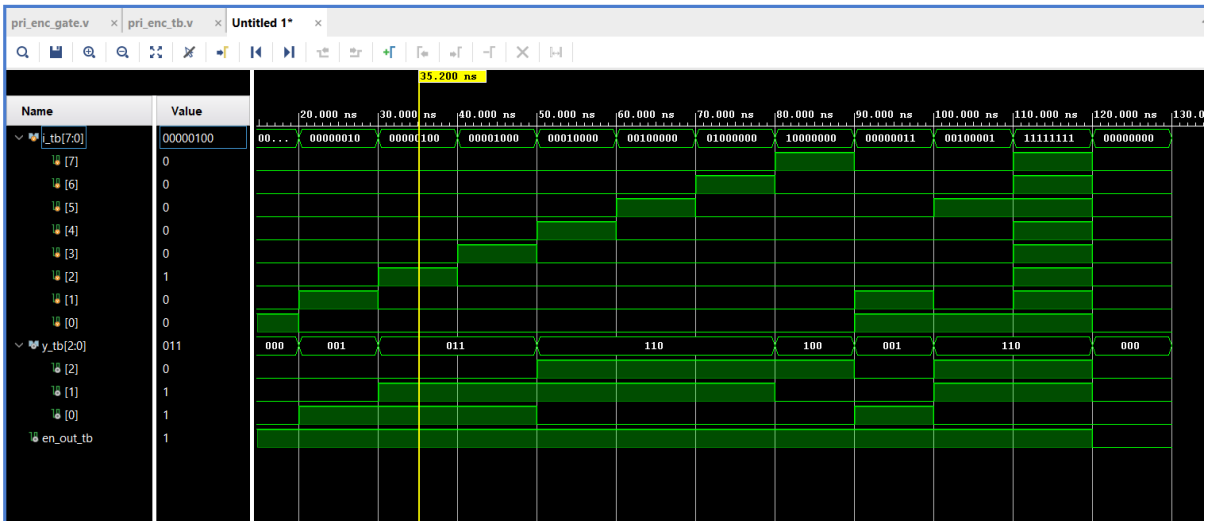
```
module enc_8X3(
 input [7:0] i,
 output reg [2:0] y);
  always @ (*) begin
  casex(i)
   8'b 1xxxxxxx: y = 3'b 111;
```

```
   8'b x1xxxxxx: y = 3'b 110;

   8'b xx1xxxxx: y = 3'b 101;

   8'b xxx1xxxx: y = 3'b 100;

   8'b xxxx1xxx: y = 3'b 011;

   8'b xxxxx1xx: y = 3'b 010;

   8'b xxxxxx1x: y = 3'b 001;

   8'b xxxxxxx1: y = 3'b 000;

   default: begin end

  endcase

  end

endmodule

module tb_priority_encoder_8x3_gate;

    reg [7:0] i_tb;

    wire [2:0] y_tb;

    wire en_out_tb;

    // Instantiate the Unit Under Test (UUT)

    priority_encoder_8x3_gate dut (

        .i(i_tb),

        .y(y_tb),

        .en_out(en_out_tb)

    );
```

```
initial begin
    // Initialize inputs
    i_tb = 8'b00000000;
    #10;
    // Test cases for priority encoder
    i_tb = 8'b00000001; // i[0] active
    #10;
    i_tb = 8'b00000010; // i[1] active
    #10;
    i_tb = 8'b00000100; // i[2] active
    #10;
    i_tb = 8'b00001000; // i[3] active
    #10;
    i_tb = 8'b00010000; // i[4] active
    #10;
    i_tb = 8'b00100000; // i[5] active
    #10;
    i_tb = 8'b01000000; // i[6] active
    #10;
    i_tb = 8'b10000000; // i[7] active
    #10;
```

```
        // Test cases with multiple inputs active (priority should be observed)
        i_tb = 8'b00000011; // i[1] has priority over i[0]
        #10;
        i_tb = 8'b00100001; // i[5] has priority over i[0]
        #10;
        i_tb = 8'b11111111; // i[7] has highest priority
        #10;
        i_tb = 8'b00000000; // No input active
       #10;
        $finish; // End simulation
    end
    initial begin
        $monitor("Time=%0t, Input i=%b, Output y=%b, Enable_out=%b", $time, i_tb, y_tb,
en_out_tb);
    end
endmodule
```

Output:



Multiplexer (8 X 1) :-

```
module mux8to1_behavioral (     input  d0, d1, d2, d3, d4, d5, d6, d7,
   input  [2:0] sel,  output reg y );
 always @(*) begin
   case (sel)
     3'b000: y = d0;
     3'b001: y = d1;
     3'b010: y = d2;
     3'b011: y = d3;
     3'b100: y = d4;
     3'b101: y = d5;
     3'b110: y = d6;
     3'b111: y = d7;
     default: y = 0;
   endcase
```

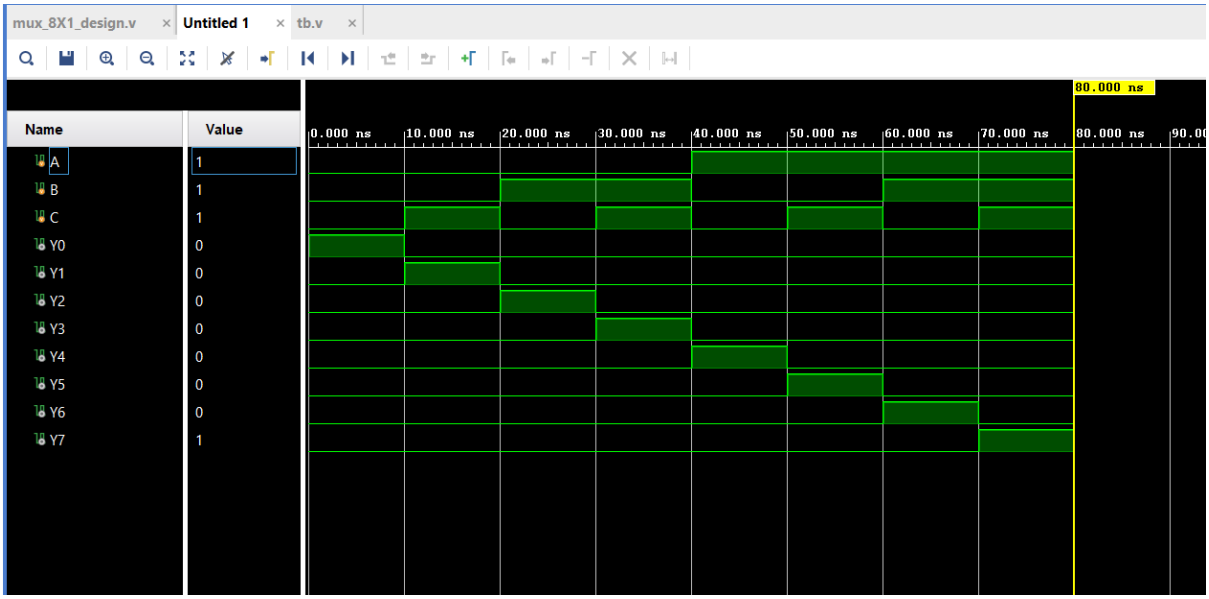
```
  end
endmodule
```

Test Bench:-

```
module tb_decoder3x8;
    reg A, B, C;
    wire Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7;
    decoder3x8 uut (A, B, C, Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7);
    initial begin
        $monitor("A=%b B=%b C=%b => Y7..Y0=%b%b%b%b%b%b%b%b",
                A, B, C, Y7, Y6, Y5, Y4, Y3, Y2, Y1, Y0);
        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; #10;
        $stop;
    end
endmodule
```

Results and Simulation:

```
# run 1000ns
A=0 B=0 C=0 => Y7..Y0=00000001
A=0 B=0 C=1 => Y7..Y0=00000010
A=0 B=1 C=0 => Y7..Y0=00000100
A=0 B=1 C=1 => Y7..Y0=00001000
A=1 B=0 C=0 => Y7..Y0=00010000
A=1 B=0 C=1 => Y7..Y0=00100000
A=1 B=1 C=0 => Y7..Y0=01000000
A=1 B=1 C=1 => Y7..Y0=10000000
```



De-Multiplexer :-

```
module demux1x8_behav (
    input d,
    input [2:0] sel,
    output reg [7:0] y
);
  always @(*) begin
    y = 8'b00000000;    // default
    if (d) begin
```

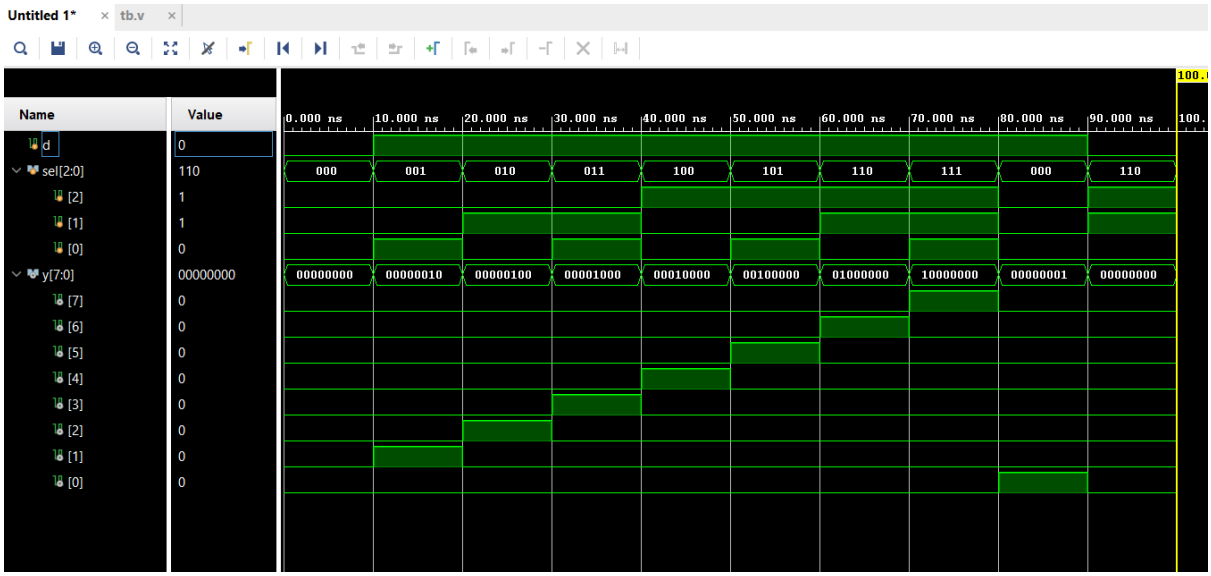
```
      case (sel)
        3'b000: y[0] = 1;
        3'b001: y[1] = 1;
        3'b010: y[2] = 1;
        3'b011: y[3] = 1;
        3'b100: y[4] = 1;
        3'b101: y[5] = 1;
        3'b110: y[6] = 1;
        3'b111: y[7] = 1;
      endcase
    end
  end
endmodule
```

Testbench:-

```
`timescale 1ns/1ps
module tb_demux1x8_behav;
    reg d;
    reg [2:0] sel;
    wire [7:0] y;
    demux1x8_behav DUT (.d(d), .sel(sel), .y(y));
    initial begin
```

```
        $monitor("Time=%0t d=%b sel=%b -> y=%b", $time, d, sel, y);
        d = 0; sel = 3'b000; #10;
        d = 1;
        repeat (8) begin
            sel = sel + 1; #10;
        end
        d = 0; sel = 3'b110; #10;
        $finish;
    end
endmodule
```

Results and simulation :-



```
Time=0 d=0 sel=000 -> y=00000000
Time=10000 d=1 sel=001 -> y=00000010
Time=20000 d=1 sel=010 -> y=00000100
Time=30000 d=1 sel=011 -> y=00001000
Time=40000 d=1 sel=100 -> y=00010000
Time=50000 d=1 sel=101 -> y=00100000
Time=60000 d=1 sel=110 -> y=01000000
Time=70000 d=1 sel=111 -> y=10000000
Time=80000 d=1 sel=000 -> y=00000001
Time=90000 d=0 sel=110 -> y=00000000
```

3. N Bit Comparator and N Bit Binary Adder

AIM: Implement the N Bit Comparator and N Bit Binary Adder using FPGA.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

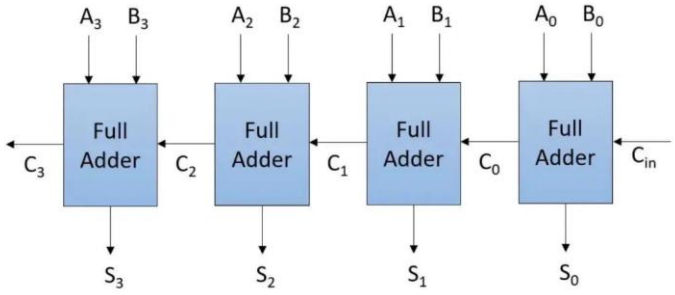
N-Bit Comparator: -

A 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 the 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	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

N-Bit Binary Adder:-

Full adder can add single bit two inputs and extra carry bit generated from its previous stage. To add multiple ‘n’ bits binary sequence, multiples cascaded full adders can be used which can generate a carry bit and be applied to the next stage full adder as an input till the last stage of full adder. This appears as carry-bit ripples to the next stage, hence it is known as “Ripple carry adder”.



4-Bit Ripple Carry Adder

Code :

```
module comp_bh #(parameter N=4)(
      input [N-1:0]a,
      input[N-1:0]b,
      output g,e,l,g_e,l_e
);
 assign{g,e,l,g_e,l_e}={(a>b),(a==b),(a<b),(a>=b),(a<=b)};
 endmodule
```

Test Bench:-

```
module comp_tb #(parameter N=4)();
   reg [N-1:0]a;
   reg [N-1:0]b;
   wire g,e,l,g_e,l_e;
   comp_bh DUT(a,b,g,e,l,g_e,l_e);

   initial begin
      $monitor("a=%b,b=%b,g=%b,e=%b,l=%b,g_e=%b,l_e=%b",a,b,g,e,l,g_e,l_e);
     #10;
     repeat(10) begin
          {a,b}=$random();
         #10;
```

```
    end
    $finish();
  end
endmodule
```

Results and Simulations: -

a=xxxx,b=xxxx,g=x,e=x,l=x,g_e=x,l_e=x

a=0010,b=0100,g=0,e=0,l=1,g_e=0,l_e=1

a=1000,b=0001,g=1,e=0,l=0,g_e=1,l_e=0

a=0000,b=1001,g=0,e=0,l=1,g_e=0,l_e=1

a=0110,b=0011,g=1,e=0,l=0,g_e=1,l_e=0

a=0000,b=1101,g=0,e=0,l=1,g_e=0,l_e=1

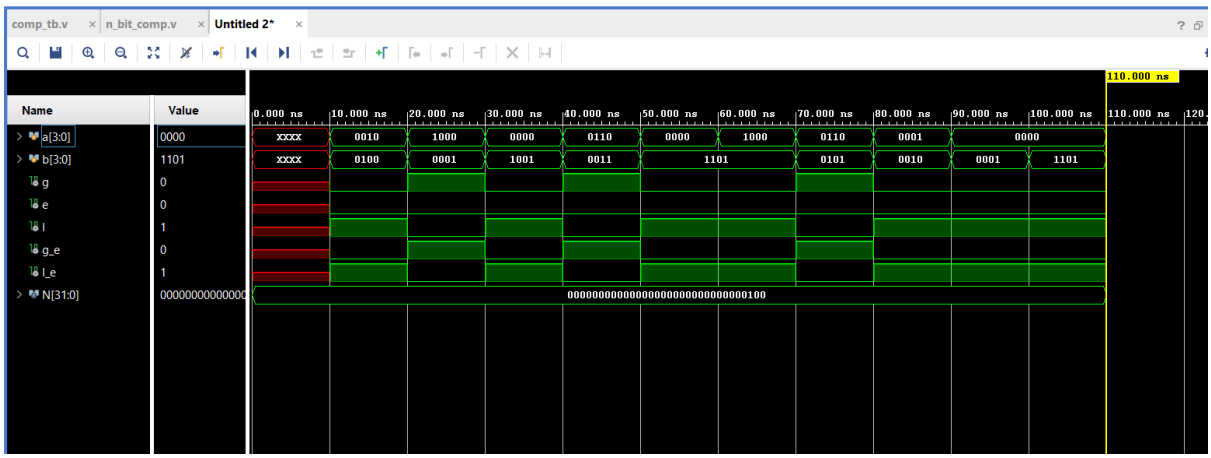
a=1000,b=1101,g=0,e=0,l=1,g_e=0,l_e=1

a=0110,b=0101,g=1,e=0,l=0,g_e=1,l_e=0

a=0001,b=0010,g=0,e=0,l=1,g_e=0,l_e=1

a=0000,b=0001,g=0,e=0,l=1,g_e=0,l_e=1

a=0000,b=1101,g=0,e=0,l=1,g_e=0,l_e=1



N_Bit Adder :-

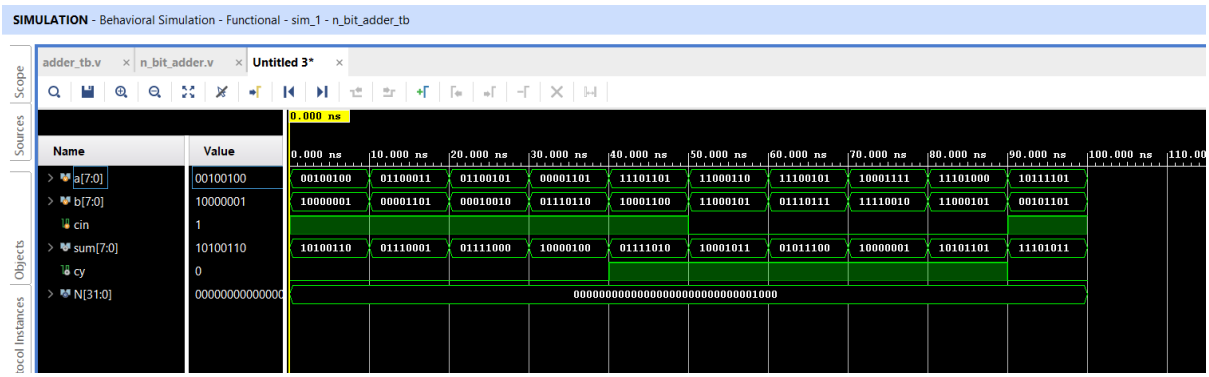
```
module n_bit_adder #(parameter N=8)(
input [N-1:0]b ,
input [N-1:0]a ,
input cin,
output [N-1:0] sum,
output cy );
 wire [N:0] carry;
assign carry[0]= cin;
assign cy = carry[N];
genvar i;
generate
  for(i=0;i<N;i=i+1) begin
   full_adder_behavioral NFA(.a(a[i]),.b(b[i]),.cin(carry[i]),.cout(carry[i+1]),.sum(sum[i]) );
  end
  endgenerate
endmodule
```

TestBench :-

```
module n_bit_adder_tb #(parameter N=8);
reg [N-1:0]a ;
reg  [N-1:0]b ;
```

```
reg cin;

wire [N-1:0] sum;

wire cy;



n_bit_adder dut (a,b,cin,sum,cy);

initial begin

$monitor("a=%b b=%b cin=%b sum=%b cy=%b",a,b,cin,sum,cy);

  repeat (10) begin

    a=$random();

    b=$random();

    cin =$random()%2;

    #10 ;

  end

    $finish;

end

endmodule
```

Results and Simulation :-



```
a=00100100 b=10000001 cin=1 sum=10100110 cy=0
a=01100011 b=00001101 cin=1 sum=01110001 cy=0
a=01100101 b=00010010 cin=1 sum=01111000 cy=0
a=00001101 b=01110110 cin=1 sum=10000100 cy=0
a=11101101 b=10001100 cin=1 sum=01111010 cy=1
a=11000110 b=11000101 cin=0 sum=10001011 cy=1
a=11100101 b=01110111 cin=0 sum=01011100 cy=1
a=10001111 b=11110010 cin=0 sum=10000001 cy=1
a=11101000 b=11000101 cin=0 sum=10101101 cy=1
a=10111101 b=00101101 cin=1 sum=11101011 cy=0
```

4. FlipFlops

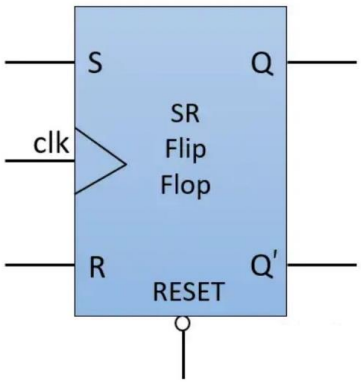
AIM: Implement all the FLIPFLOPS.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

SR FLIPFLOP:

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



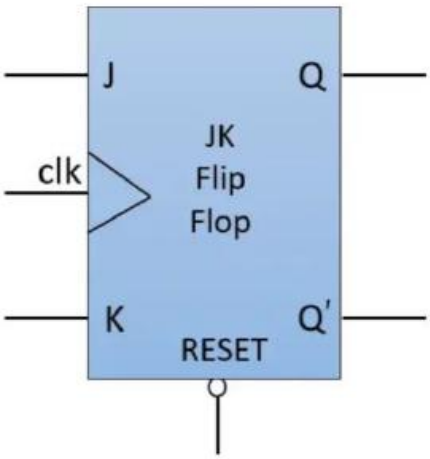
SR Flip Flop

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

JK FLIPFLOP:

- The JK flip flop is used to remove the drawback of the S-R flip flop, i.e., undefined states.
- The JK flip flop is formed by doing modification in the SR flip flop.
- When S and R input is set to true, the SR flip flop gives an inaccurate result. But in the case of JK flip flop, it gives the correct output.
- When the input J and K are different then the output Q takes the value of J at the next clock edge.
- When J and K both are low then NO change occurs at the output.

- If both J and K are high, then at the clock edge, the output will toggle from one state to the other.

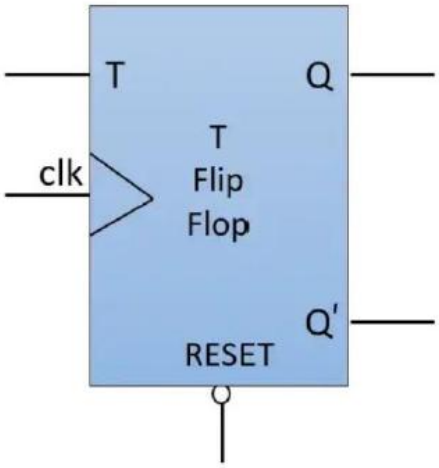


JK Flip Flop

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

T-Flip Flop: -

T flip flop or to be precise is known as Toggle Flip Flop because it can able to toggle its output depending upon on the input. There stands for Toggle. Toggle basically indicates that the bit will be flipped i.e., either from 1 to 0 or from 0 to 1. Here, a clock pulse is supplied to operate this flop, hence it is a clocked flip-flop.



T Flip Flop

Truth Table

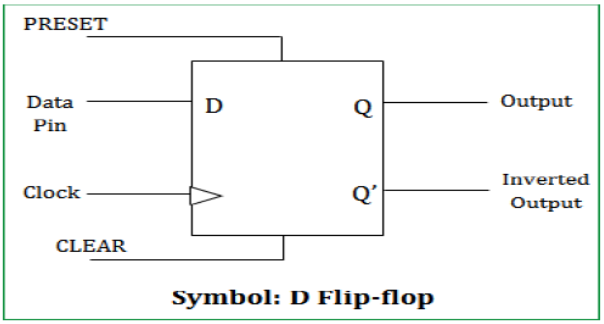
T	Q_{n+1}
0	Q_n(No Change)
1	$\overline{Q_n}$(Toggles)

D-Fllip Flop:-

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

The D flip flop has two inputs, data and clock input which controls the flip flop. when clock input is high, the data is transferred to the output of the flip flop.

when the clock input is low, the output of the flip flop is held in its previous state.



Symbol: D Flip-flop

D	CLK	$\bar{Q}$
0	Rising Edge	0
1	Rising Edge	1

Code:-

SR:

```
module sr_flipflop(
  input s,r,rst,clk,
  output reg q,q_bar);
always @ (posedge clk) begin
    if(rst) begin
    q<=0;
    q_bar<=1;
     end
   else begin
```

```
    case({s,r})
      2'b00:begin
              q<=q; q_bar<=q_bar;
            end
      2'b01:begin
              q<=0; q_bar<=1;
            end
      2'b10:begin
              q<=1; q_bar<=0;
            end
      2'b11: begin
            end
    default : begin
            end
endcase
end
end
endmodule
```

Test Bench:

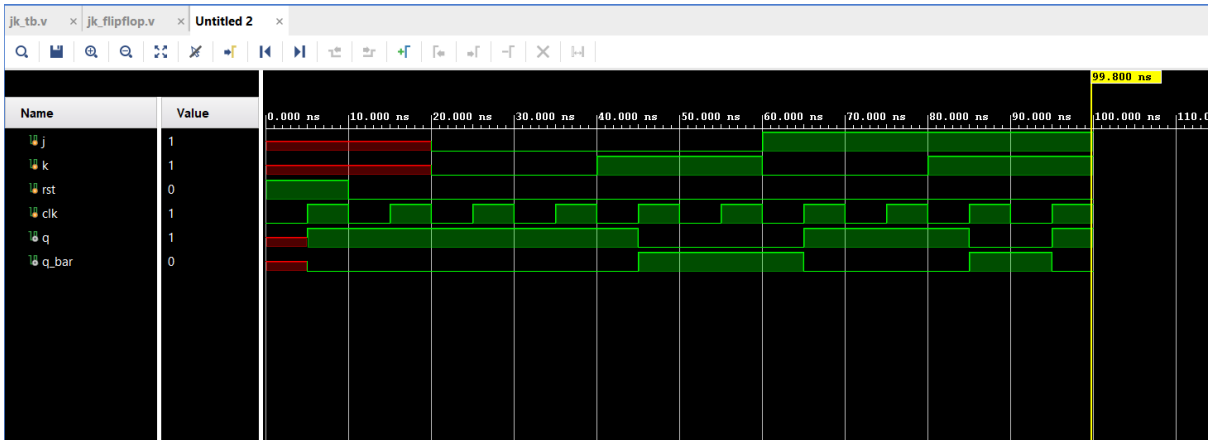
```
module sr_tb();
reg s,r,rst,clk;
```

```
wire q,q_bar;

sr_flipflop dut(s,r,rst,clk,q,q_bar);

initial clk=0;

always #5 clk=(~clk);

initial begin

rst=1;#10; rst=0;#10;

{s,r}=2'b10;#20;

{s,r}=2'b01;#20;

{s,r}=2'b00;#20;

{s,r}=2'b11;#20;

$finish();

end

endmodule
```

JK FLIPFLOP :-

```
module jk_ff(
  input j,k,rst,clk,
  output reg q,q_bar);
  always @(posedge clk) begin
   if(rst) begin
      q<=1; q_bar<=0;
   end
   else begin
   case({j,k})
        2'b00  :  {q,q_bar} <= {q,q_bar};
        2'b01  :  {q,q_bar} <= 2'b01;
        2'b10  :  {q,q_bar} <= 2'b10;
        2'b11  :  {q,q_bar} <= {q_bar,q};
        default:  begin end
endcase
end
end
endmodule


module jk_tb();
```

```
reg j,k,rst,clk;

wire q,q_bar;

jk_ff dut(j,k,rst,clk,q,q_bar);

initial clk=0;

always #5 clk = ~clk;

initial begin

    rst=1;#10;

    rst=0;#10;



 {j,k}=2'b00;#20;

 {j,k}=2'b01;#20;

 {j,k}=2'b10;#20;

 {j,k}=2'b11;#20;

 $finish();

 end

endmodule
```

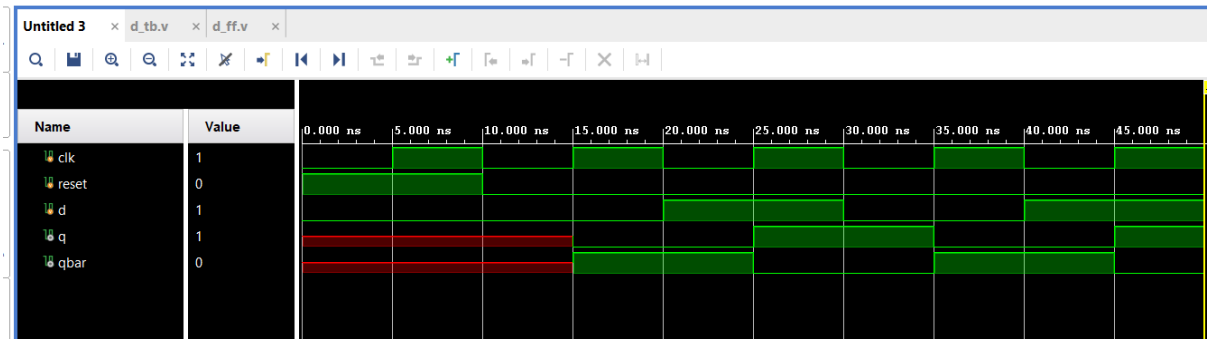


D FLIP FLOP :-

```
module dff (
    input clk,
    input reset,
    input d,
    output reg q,
    output reg qbar
);


always @(posedge clk or posedge reset) begin
    if (reset) begin
        q <= 1'bX;
        qbar <= 1'bx;
    end
    else begin
        q <= d;
```

```
        qbar <= ~d;
    end
end
endmodule

module d_tb;
reg clk, reset, d;
wire q , qbar;
dff dut ( .clk(clk), .reset(reset), .d(d),.q(q),.qbar(qbar));

initial begin
clk <= 1'b0;
reset <= 1'b1;
d <= 1'b0;
#10 reset <= 1'b0;
#10 d <= 1'b1;
#10 d <= 1'b0;
#10 d <= 1'b1;
#10 $finish;
end
always #5 clk <= ~clk;
```

```
initial begin

$monitor("clk = %b, reset = %b, d = %b, q = %b , qbar =%b", clk, reset, d, q, qbar);

end

endmodule
```

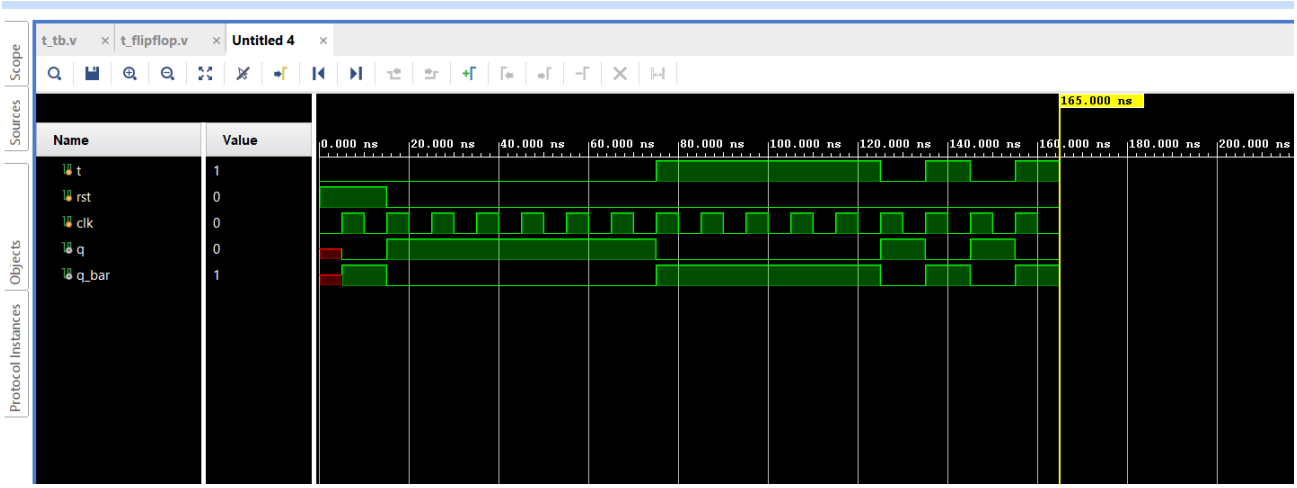


T FLIP FLOP :-

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


 always@ (posedge clk) begin
   if(rst)begin
       q<=0;
       q_bar=1;
   end
   else begin
       if(t)begin
```

```
            q<=0;
            q_bar<=1;
        end
        else begin
            q<=1;
            q_bar<=0;
        end
    end
 end
endmodule

module t_tb();

  reg t, rst, clk;
  wire q, q_bar;

  // Change port order based on your t_ff module
  t_ff dut(.clk(clk), .rst(rst), .t(t), .q(q), .q_bar(q_bar));

  // Clock generation
  initial clk = 0;
```

```
  always #5 clk = ~clk;

  initial begin

    rst = 1; t = 0; #15;

    rst = 0; #10;

    t = 0; #50;

    t = 1; #50;

    t = 0; #10;

    t = 1; #10;

    t = 0; #10;

    t = 1; #10;

    $finish;

  end

endmodule
```



5. Universal Counter

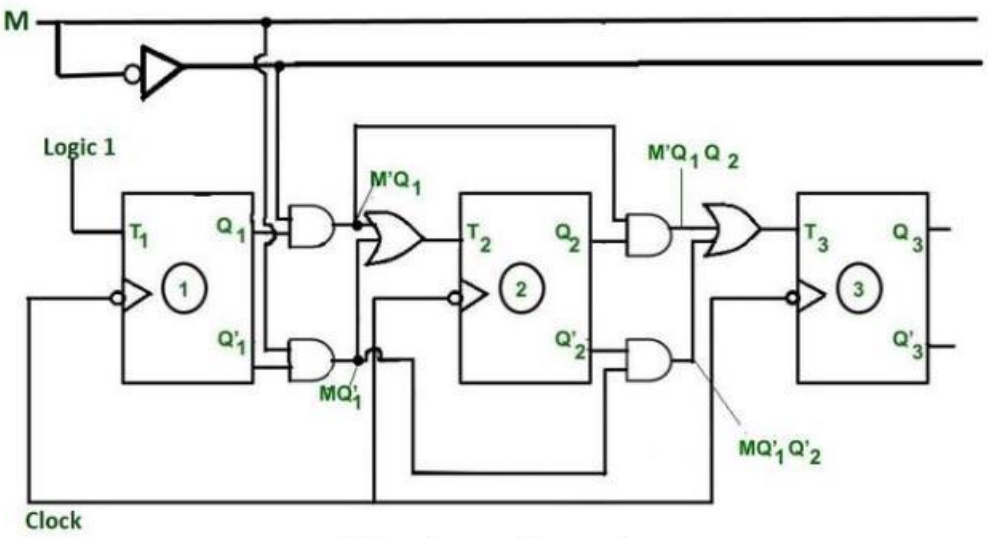
AIM: Implement the Universal Counter.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

Universal Counter:

- These types of counters fall under the category of synchronous controller counter.
- Here the mode control input is used to decide whether which sequence will be generated by the counter.
- In this case, mode control input is used to decide whether the counter will perform up counting or down counting.
- Designing of such a counter is the same as designing a synchronous counter but the extra combinational logic for mode control input is required.



3 bit synchronous up/down counter.

Code:-

```
module synchronous_counter #(parameter SIZE=4)(
   input clk, rst_n, up,
   output reg [SIZE-1:0] cnt  );
   always @(posedge clk)
      if (!rst_n) cnt <= 0;
      else cnt <= up ? cnt + 1 : cnt - 1;
endmodule


module tb;
  reg clk = 0, rst_n = 0, up = 1;
  wire [3:0] cnt;
  synchronous_counter uut (.clk(clk), .rst_n(rst_n), .up(up), .cnt(cnt));
  always #2 clk = ~clk;
  initial begin
    #4  rst_n = 1;
    #80 rst_n = 0;
    #4  rst_n = 1; up = 0;
    #50 $finish;
  end
endmodule
```


Design.v
tb.v
Untitled 4
Name
Value
clk
0
rst_n
1
up
0
cnt[3:0]
4
[3]
0
[2]
1
[1]
0
[0]
0
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 0 f e d c b a 9 8 7 6 5

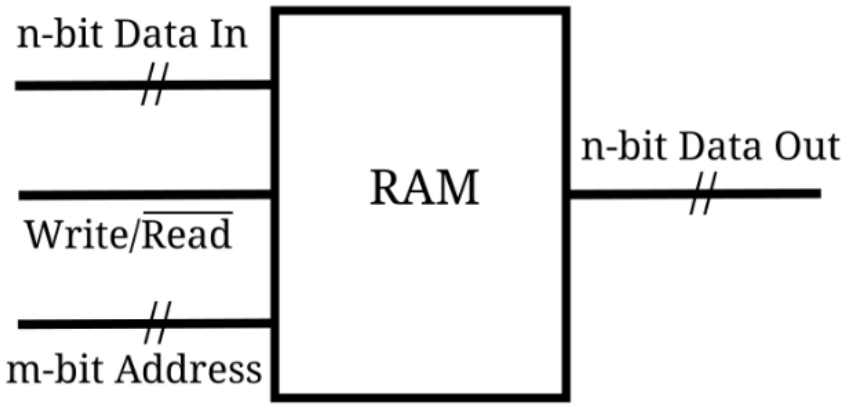
6. SRAM Design

AIM: Implement the SRAM Memory.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

Random Access Memory is the temporary memory used in a processor or the digital system which requires larger memory for storing temporary data. When designing any system on FPGA, sometimes we require a RAM block which is also called BRAM or block RAM. In this post, we'll how to describe a RAM in Verilog HDL. Most of the latest FPGAs have BRAM and if we synthesize this, it will be synthesized into a BRAM



Code:-

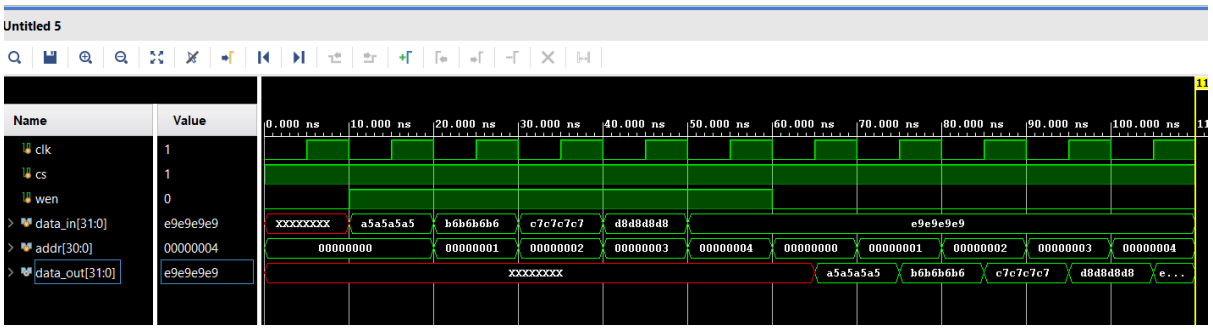
```
module memory_32b_8gb #(parameter dw=32, parameter aw=31)(
  input clk, cs, wen,
  input [dw-1:0] data_in,
  input [aw-1:0] addr,
  output reg [dw-1:0] data_out
);
```

```
  reg [dw-1:0] mem[0:(1<<aw)-1];
  always @(posedge clk) begin
    if (cs & wen) begin
      mem[addr] <= data_in;
    end
  end
  always @(posedge clk) begin
    if (cs & ~wen) begin
      data_out <= mem[addr];
    end
  end
endmodule

module tb_memory_32b_8gb();
 reg clk, cs, wen;
  reg [31:0] data_in;
  reg [30:0] addr;
  wire [31:0] data_out;
  memory_32b_8gb #(.dw(32), .aw(31)) dut (
    .clk(clk),
    .cs(cs),
```

```
  .wen(wen),
  .data_in(data_in),
  .addr(addr),
  .data_out(data_out)
);
always #5 clk = ~clk;

initial begin
  clk = 0;
end
initial
begin
  cs = 1;
  wen = 0;
  addr = 0;
  #10;
  wen = 1;
  data_in = 32'hA5A5A5A5; addr = 28'h0;
  #10;
  data_in = 32'hB6B6B6B6; addr = 28'h1;
  #10;
```

```
data_in = 32'hC7C7C7C7; addr = 28'h2;

#10;

data_in = 32'hD8D8D8D8; addr = 28'h3;

#10;

data_in = 32'hE9E9E9E9; addr = 28'h4;

#10;

wen = 0;

addr = 28'h0;

#10;

$display("Data at address 0: %h", data_out);



addr = 28'h1;

#10;

$display("Data at address 1: %h", data_out);



addr = 28'h2;

#10;

$display("Data at address 2: %h", data_out);



addr = 28'h3;

#10;
```

```
    $display("Data at address 3: %h", data_out);
     addr = 28'h4;
    #10;
    $display("Data at address 4: %h", data_out);
    $finish();
  end
endmodule
```



7. Universal Shift Register

AIM: Implement the Universal Shift Registers.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

- When we have bi directional and parallel loading then that register is known as Universal shift register.
- We are using 4x1 mux and 2 select line's are connected to all 4 registers.
- The universal shift register can store the data in parallel and can transmit the data in parallel.
- In the same manner the data can stored and transmitted by serial path through shift left and shift right operations.

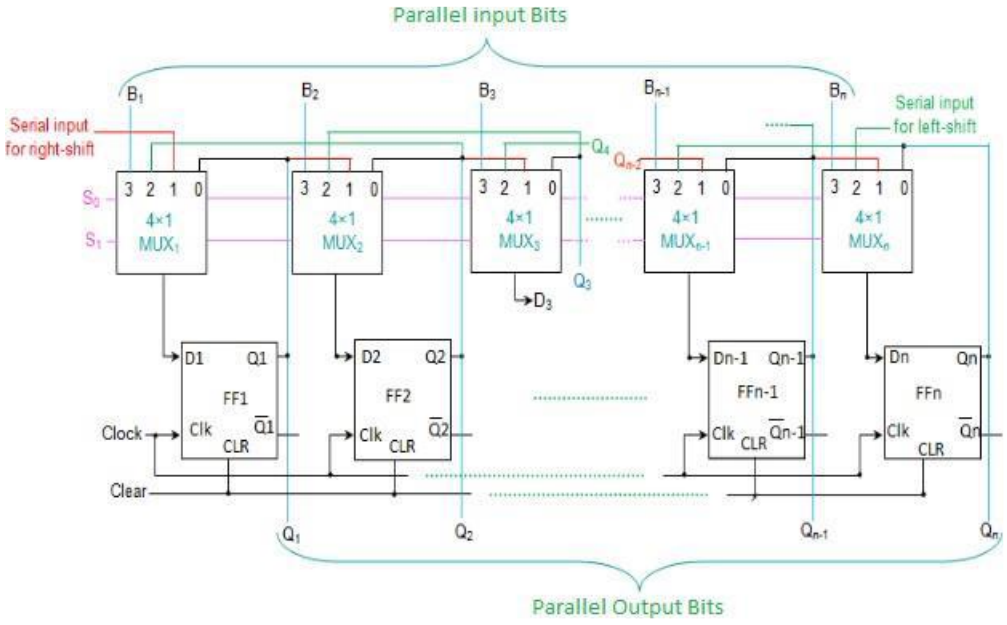


Table:-

S0	S1	Operating Mode
0	0	Locked
1	0	Shift-Right
0	1	Shift-Left
1	1	Parallel Loading

OPERATION:

When S0 = 0 and S1 = 0, then the universal shift register will be in locked state, that means no operation will be performed.

When S0 = 1 and S1 = 0, then it will shift the data to right, that is shift right operation is performed.

When S0 = 0 and S1 = 1, then it will shift the data to right, that is shift left operation is performed.

When S0 = 0 and S1 = 0, then the register results in PARALLEL LOAD operation.

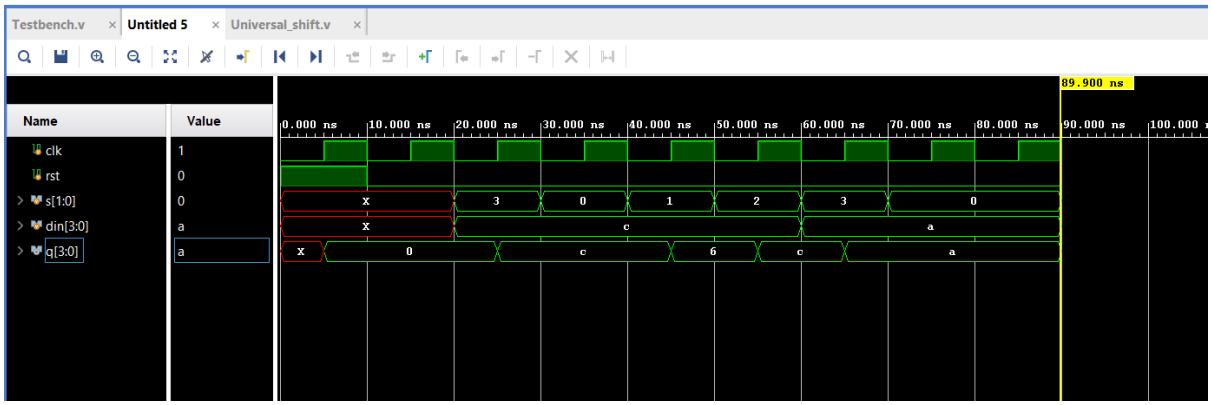
Code:-

```
module universal #(
  parameter WIDTH = 4
)(
  output reg [WIDTH-1:0] q,
  input              clk,
  input              rst,      // Active-high synchronous reset
  input       [1:0]     s,       // 00: hold, 01: shift right, 10: shift left, 11: load
  input  [WIDTH-1:0]     din
);
  always @(posedge clk) begin
    if (rst) begin
      q <= {WIDTH{1'b0}};
    end else begin
      case (s)
```

```
      2'b00: q <= q;                        // Hold
      2'b01: q <= {1'b0, q[WIDTH-1:1]};     // Shift right (zero-fill)
      2'b10: q <= {q[WIDTH-2:0], 1'b0};     // Shift left (zero-fill)
      2'b11: q <= din;                      // Parallel load
      default: q <= q;
    endcase
  end
 end
endmodule
module universal_tb;
 reg clk = 0, rst = 1;
 reg [1:0] s;
 reg [3:0] din;
 wire [3:0] q;

 universal dut (.q(q), .clk(clk), .rst(rst), .s(s), .din(din));

 always #5 clk = ~clk;   // 10 ns clock period

 initial begin
  $monitor("t=%0t | rst=%b s=%b din=%b q=%b", $time, rst, s, din, q);
```

```
    // Reset for 10 ns
    #10 rst = 0;
    // Stimulus sequence using delays
    #10 s = 2'b11; din = 4'b1100; // Load
    #10 s = 2'b00;              // Hold
    #10 s = 2'b01;              // Shift right
    #10 s = 2'b10;              // Shift left
    #10 s = 2'b11; din = 4'b1010; // Load again
    #10 s = 2'b00;              // Hold
    #20 $finish;
  end
endmodule
```

Result:



8. Floating Point Adder

AIM: Implement the Floating Point Adder.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

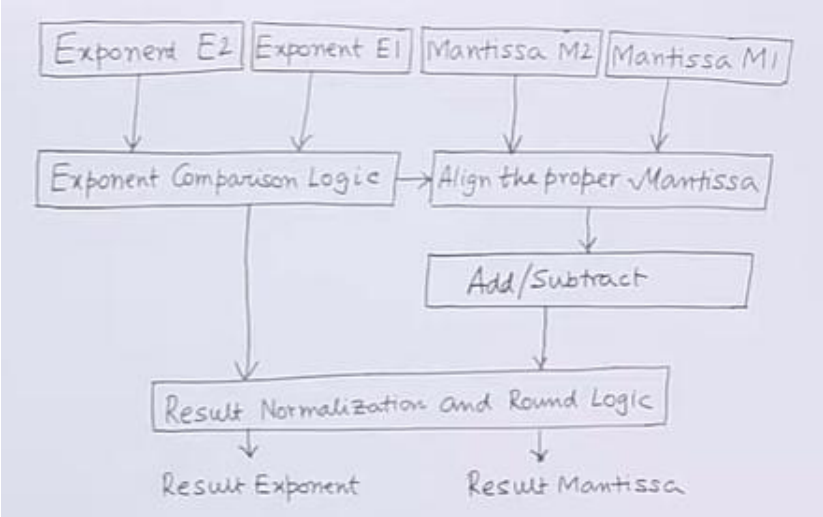
- Lets take an example of binary addition: 23.5 + 452.752 = 476.252
    - To add 23.5 and 452.752 in IEEE 754 format, the steps involve
        - converting the numbers into their IEEE 754 representation,
        - aligning the exponents,
        - performing the addition,
        - and normalizing the result.

Step	Description	23.5	452.752
1	Convert to Binary	10111.1	$111000100.1100001111010111$
2	Normalize	1.01111×2^4	$1.11000100110000111101011 \times 2^8$
3	Calculate Exponent (with bias)	$4 + 127 = 131$, Binary: 10000011	$8 + 127 = 135$, Binary: 10000111
4	Mantissa (drop leading 1, 23 bits)	01111000000000000000000	11000100110000111101011
5	IEEE 754 Representation	$0\ 10000011\ 01111000000000000000000$	$0\ 10000111\ 11000100110000111101011$
6	Align Exponents (shift smaller mantissa to match larger exponent)	Mantissa shifted: 0.000101111×2^8	Mantissa stays: $1.11000100110000111101011 \times 2^8$
7	Add Mantissas	Add: $0.000101111 + 1.11000100110000111101011 \rightarrow 1.11011100010000111101011$	

8	Normalize the Result	Already normalized: $1.11011100010000111101011 \times 2^8$	
9	Construct Final IEEE 754 Representation	Sign: 0, Exponent: 135, Mantissa: 11011100010000111101011	
10	Final IEEE 754 Binary Representation	$0\ 10000111\ 11011100010000111101011$	
11	Convert Back to Decimal	476.252	

Final Answer:

$$23.5 + 452.752 = 476.252$$



Code:-

`timescale 1ns / 1ps

module single_precision_add (

input [31:0] a, // Input IEEE 754 number 1

input [31:0] b, // Input IEEE 754 number 2

output [31:0] result // Output IEEE 754 result

);

```
// Internal Signals
reg sign_a, sign_b, sign_res;
reg [7:0] exp_a, exp_b, exp_res, exp_sum;
reg [23:0] mant_a, mant_b, mant_res;
reg [24:0] mant_sum;
reg [7:0] exp_diff;
reg [7:0] exp_temp;
reg [23:0] mant_shifted;

// Step 1: Split Inputs into Sign, Exponent, and Mantissa
always @(*)
begin : SPLIT_INPUTS
        sign_a = a[31];
        sign_b = b[31];
        exp_a  = a[30:23];
        exp_b  = b[30:23];
        mant_a = {1'b1, a[22:0]}; // Add implicit leading 1
        mant_b = {1'b1, b[22:0]};
end

// Step 2: Align Exponents
always @(*)
begin : ALIGN_EXPONENTS
        if (exp_a > exp_b)
        begin
                exp_diff = exp_a - exp_b;
                exp_sum = exp_a;
```

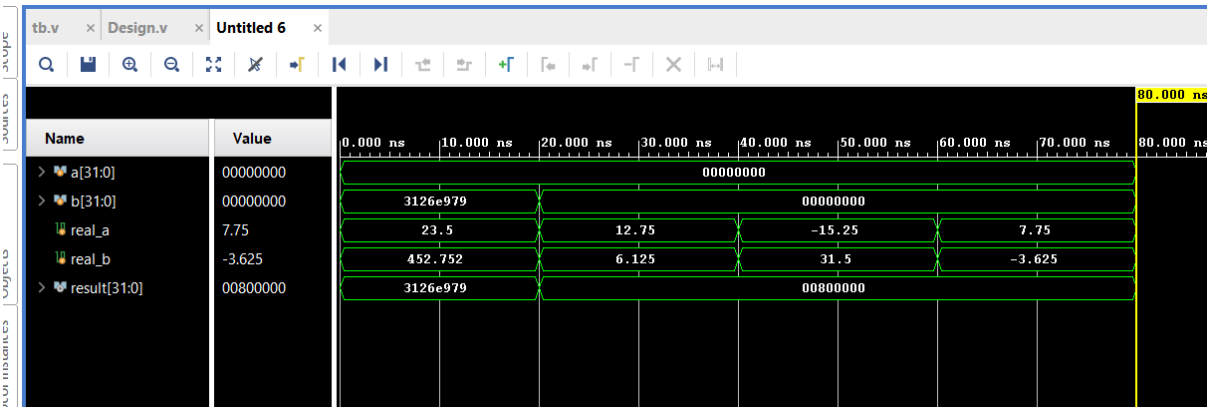
```
                mant_shifted = mant_b >> exp_diff;
                mant_sum = mant_a + mant_shifted;
        end
        else
        begin
                exp_diff = exp_b - exp_a;
                exp_sum = exp_b;
                mant_shifted = mant_a >> exp_diff;
                mant_sum = mant_b + mant_shifted;
        end
end

// Step 3: Normalize Result
always @(*)
begin : NORMALIZE
        if (mant_sum[24])
        begin
                mant_res = mant_sum[24:1]; // Normalize mantissa
                exp_res = exp_sum + 1; // Adjust exponent
        end
        else
        begin
                mant_res = mant_sum[23:0];
                exp_res = exp_sum;
        end
end

// Step 4: Combine Sign, Exponent, and Mantissa
```

```
        assign result = {sign_a, exp_res, mant_res[22:0]};

endmodule



//Testbench

`timescale 1ns / 1ps

module single_precision_add_tb;

        // Inputs
        reg [31:0] a;
        reg [31:0] b;
        real real_a;
        real real_b;

        // Output
        wire [31:0] result;

        // Instantiate the Unit Under Test (UUT)
        single_precision_add uut (
                .a(a),
                .b(b),
                .result(result)
        );

        // Task to display IEEE 754 output
```

```
task display_result;
        input [31:0] res;
        begin
                $display("Result (Binary) = %b", res);
                $display("Result (Hex)    = %h", res);
                #10;
        end
endtask

// Testbench Procedure
initial
begin : TEST_CASES
        $display("\n--- Test Case 1: 23.5 + 452.752 ---");
        real_a = 23.5;
        real_b = 452.752;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);
        $display("\n--- Test Case 2: 12.75 + 6.125 ---");
        real_a = 12.75;
        real_b = 6.125;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);

        $display("\n--- Test Case 3: -15.25 + 31.5 ---");
```

```
        real_a = -15.25;
        real_b = 31.5;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);

        $display("\n--- Test Case 4: 7.75 + -3.625 ---");
        real_a = 7.75;
        real_b = -3.625;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);

        $stop;
    end

endmodule
```

Result:-



9. Floating Point Multiplier

AIM: Implement the Floating Point Multiplier.

Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

Lets take an example of binary Multiplication: 23.5 X 452.72 = 10655.192

To multiply 23.5 and 452.752 in **IEEE 754 single-precision format**, the steps involve:

1. **Converting the numbers into their IEEE 754 representations**
2. **Adding the exponents**
3. **Multiplying the mantissas**
4. **Normalizing the result**
5. **Constructing the final IEEE 754 representation**

Step 1: Convert the Numbers to Binary and Normalize

For 23.5:

1. Convert 23.5 to binary: 10111.1.
2. Normalize: 1.01111×2^4.
3. Exponent: Add the bias 127: $4 + 127 = 131 \rightarrow 10000011$ in binary.
4. Mantissa: Drop the leading 1 → 01111000000000000000000 (23 bits).
   - Final IEEE 754 (32-bit): $0\ 10000011\ 01111000000000000000000$.

For 452.752:

1. Convert 452.752 to binary: $111000100.1100001111010111$.
2. Normalize: $1.11000100110000111101011 \times 2^8$.
3. Exponent: Add the bias 127: $8 + 127 = 135 \rightarrow 10000111$ in binary.
4. Mantissa: Drop the leading 1 → 11000100110000111101011 (23 bits).
   - Final IEEE 754 (32-bit): $0\ 10000111\ 11000100110000111101011$.

Step 2: Add the Exponents

1. Exponent of 23.5 = 131, Exponent of 452.752 = 135.
2. Add the exponents and subtract the bias:
$$131 + 135 - 127 = 139$$
3. New Exponent: 139, Binary: 10001011.

Step 3: Multiply the Mantissas

1. Mantissa of 23.5 (with implicit 1): 1.01111.
2. Mantissa of 452.752 (with implicit 1): 1.1100010011.
3. Multiply the mantissas:
$$1.01111 \times 1.1100010011 = 1.10100101111 \text{ (approx)}$$
The result is already normalized.

Step 4: Assemble the Final IEEE 754 Format

1. **Sign bit**: 0 (since the result is positive).
2. **Exponent**: 139, Binary: 10001011.
3. **Mantissa**: Keep the first 23 bits after the leading 1:
$$10100101111000000000000$$

Final IEEE 754 Representation (32-bit):

$$0\ 10001011\ 10100101111000000000000$$

Step 5: Convert Back to Decimal

1. The result in binary: $1.10100101111 \times 2^{12}$.
2. Convert to decimal:
$$1.10100101111 \times 2^{12} = 10655.192$$

Final Answer:

$$23.5 \times 452.752 = 10655.192$$

- **IEEE 754 Single-Precision Representation** (Binary):
$0\ 10001011\ 10100101111000000000000$
- **Hexadecimal**: $45A98000$.

Code:-

```
`timescale 1ns / 1ps

module single_precision_mult (

        input [31:0] a, // Input IEEE 754 number 1

        input [31:0] b, // Input IEEE 754 number 2

        output [31:0] result // Output IEEE 754 result

);
```

```
// Internal Signals

reg sign_a, sign_b, sign_res;

reg [7:0] exp_a, exp_b, exp_res;

reg [23:0] mant_a, mant_b;

reg [47:0] mant_mult;

reg [7:0] exp_sum;

reg [22:0] mant_res;

// Step 1: Split Inputs into Sign, Exponent, and Mantissa

always @(*)

begin : SPLIT_INPUTS

        sign_a = a[31];

        sign_b = b[31];

        exp_a  = a[30:23];

        exp_b  = b[30:23];

        mant_a = {1'b1, a[22:0]}; // Add implicit leading 1

        mant_b = {1'b1, b[22:0]};

end

// Step 2: Calculate Sign and Exponent

always @(*)

begin : CALCULATE_SIGN_EXP

        sign_res = sign_a ^ sign_b; // XOR of signs for result sign
```

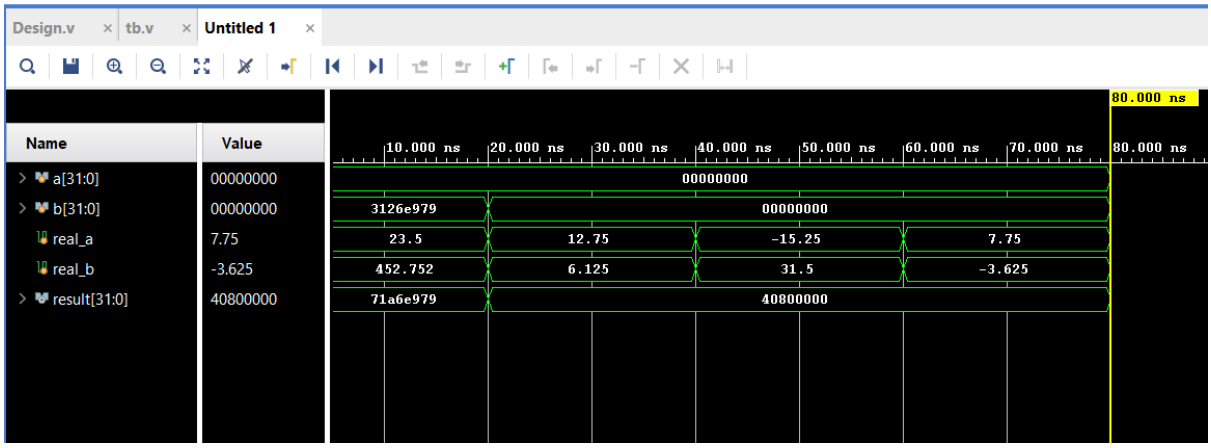
```
        exp_sum = exp_a + exp_b - 127; // Add exponents and subtract bias
end
// Step 3: Multiply Mantissas
always @(*)
begin : MULTIPLY_MANTISSAS
        mant_mult = mant_a * mant_b; // Multiply mantissas
end
// Step 4: Normalize Result
always @(*)
begin : NORMALIZE
        if (mant_mult[47])
        begin
                mant_res = mant_mult[46:24]; // Normalize mantissa (shift right by 1)
                exp_res = exp_sum + 1; // Adjust exponent
        end
        else
        begin
                mant_res = mant_mult[45:23]; // Use result directly
                exp_res = exp_sum;
        end
end
```

```
        // Step 5: Combine Sign, Exponent, and Mantissa
        assign result = {sign_res, exp_res, mant_res};
endmodule
//Testbench
`timescale 1ns / 1ps
module single_precision_mult_tb;
        // Inputs
        reg [31:0] a;
        reg [31:0] b;
        real real_a;
        real real_b;
        // Output
        wire [31:0] result;
        // Instantiate the Unit Under Test (UUT)
        single_precision_mult uut (
                .a(a),
                .b(b),
                .result(result)
        );
        // Task to display IEEE 754 output
        task display_result;
```

```
        input [31:0] res;
        begin
                $display("Result (Binary) = %b", res);
                $display("Result (Hex)    = %h", res);
                #10;
        end
endtask

// Testbench Procedure
initial
begin : TEST_CASES
        $display("\n--- Test Case 1: 23.5 * 452.752 ---");
        real_a = 23.5;
        real_b = 452.752;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);
        $display("\n--- Test Case 2: 12.75 * 6.125 ---");
        real_a = 12.75;
        real_b = 6.125;
```

```
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);
        $display("\n--- Test Case 3: -15.25 * 31.5 ---");
        real_a = -15.25;
        real_b = 31.5;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);
        $display("\n--- Test Case 4: 7.75 * -3.625 ---");
        real_a = 7.75;
        real_b = -3.625;
        a = $realtobits(real_a);
        b = $realtobits(real_b);
        #10;
        display_result(result);
        $stop;
    end
endmodule
```

Results:-


Design.v
tb.v
Untitled 1
Name
Value
a[31:0]
00000000
b[31:0]
00000000
real_a
7.75
real_b
-3.625
result[31:0]
40800000
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
00000000
3126e979
00000000
23.5
12.75
-15.25
7.75
452.752
6.125
31.5
-3.625
71a6e979
40800000

10. Boot Multiplier

AIM: Implement the Boot Multiplier.

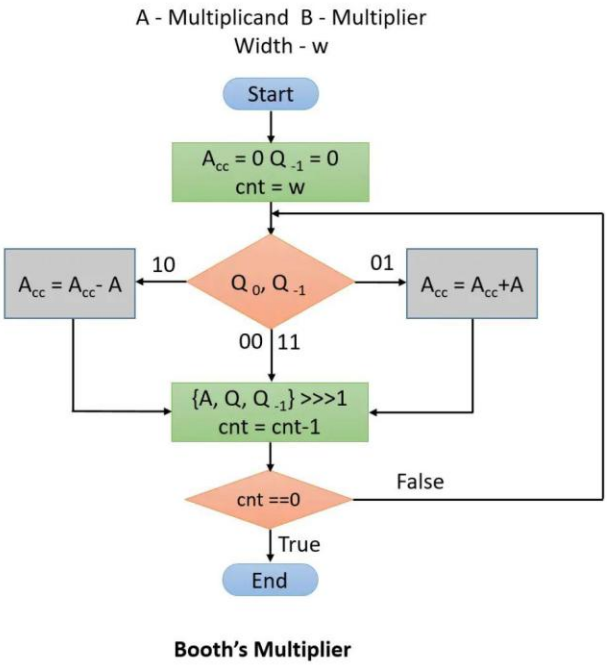
Tools: Xilinx Vivado HLx Editions 2018.2 - Z7 FPGA Board (xc7z010clg400-1)

Theory:

Booth's Multiplier

In Booth's multiplier works on Booth's Algorithm that does the multiplication of 2's complement notation of two signed binary numbers.

Flow chart of Booth's Algorithm



Booth's Multiplier

Please note of below abbreviations used:

A – holds Multiplicand

B – holds Multiplier

Q = B

Q0 – holds 0th bit (LSB) of Q register

Q-1 – 1-bit variable/register.

Acc – Accumulator holds the result of intermediate addition/subtraction.

Count = max(width of multiplicand register, width of multiplier register)

Code:-

```
`timescale 1ns / 1ps

module booth_multiplier (

    input [3:0] multiplicand,    // 4-bit multiplicand (M)
```

```
    input [3:0] multiplier,      // 4-bit multiplier (Q)
    input clk,                   // Clock signal
    input rst,                   // Active-high reset
    output reg [7:0] product,    // 8-bit product (A concatenated with Q)
    output sign,                 // Sign of the product
    output [7:0] magnitude       // Magnitude of the product
);
    // FSM state definitions using localparam
    localparam INIT              = 4'b0000;
    localparam DECISION          = 4'b0001;
    localparam ADD               = 4'b0010;
    localparam SUB               = 4'b0011;
    localparam SHIFT_RIGHT       = 4'b0100;
    localparam DEC_COUNTER       = 4'b0101;
    localparam CHECK_DONE        = 4'b0110;
    localparam RESULT_EXTRACT    = 4'b0111;
    localparam STOP              = 4'b1000;
    reg [3:0] current_state, next_state; // FSM states
    reg [3:0] A;             // Accumulator
    reg [3:0] Q;             // Multiplier
    reg q_0;                 // Extra bit (Q-1)
```

```
reg [3:0] M;            // Multiplicand

reg [2:0] n;            // Counter for bits

// Determine sign and magnitude of the product

assign sign = product[7];

assign magnitude = sign ? (~product + 8'd1) : product;

// State transition logic

always @(posedge clk)

begin

   if (rst)

      current_state <= INIT;

   else

      current_state <= next_state;

end

// Next state logic

always @(*)

begin

   next_state = current_state;

   case (current_state)

      INIT:          next_state = DECISION;

      DECISION:      case ({Q[0], q_0})

                        2'b01: next_state = ADD;
```

```
                    2'b10: next_state = SUB;
                    default: next_state = SHIFT_RIGHT;
                endcase
        ADD:            next_state = SHIFT_RIGHT;
        SUB:            next_state = SHIFT_RIGHT;
        SHIFT_RIGHT:    next_state = DEC_COUNTER;
        DEC_COUNTER:    next_state = CHECK_DONE;
        CHECK_DONE:     next_state = (n == 0) ? RESULT_EXTRACT : DECISION;
        RESULT_EXTRACT: next_state = STOP;
        STOP:           next_state = STOP;
    endcase
end
// State actions
always @(posedge clk)
begin
    if (rst)
    begin
        A <= 4'b0000;
        Q <= 4'b0000;
        q_0 <= 1'b0;
        M <= 4'b0000;
```

```
    n <= 3'd4;
    product <= 8'b00000000;
end
else
begin
    case (current_state)
        INIT: begin
            A <= 4'b0000;
            Q <= multiplier;
            q_0 <= 1'b0;
            M <= multiplicand;
            n <= 4;
        end
        ADD: begin
            A <= A + M; // Add multiplicand to accumulator
        end
        SUB: begin
            A <= A - M; // Subtract multiplicand from accumulator
        end
        SHIFT_RIGHT: begin
            q_0 <= Q[0];
```

```
            Q <= {A[0], Q[3:1]};
            A <= {A[3], A[3:1]}; // Arithmetic Shift Right
        end
        DEC_COUNTER: begin
            n <= n - 1;
        end
        RESULT_EXTRACT: begin
            product <= {A, Q}; // Combine A and Q
        end
        STOP: begin
            // Do nothing
        end
      endcase
    end
  end
endmodule
```

Testbench:-

```
`timescale 1ns / 1ps
module booth_multiplier_tb1;
  // Testbench signals
  reg [3:0] multiplicand;
```

```
reg [3:0] multiplier;

reg clk;

reg rst;

wire [7:0] product;

wire sign;

wire [7:0] magnitude;

// Instantiate the DUT (Device Under Test)

booth_multiplier dut (

    .multiplicand(multiplicand),

    .multiplier(multiplier),

    .clk(clk),

    .rst(rst),

    .product(product),

    .sign(sign),

    .magnitude(magnitude)

);

// Clock generation

always #5 clk = ~clk;

// Task to print results

task print;

    input [3:0] in_multiplicand;
```

```
        input [3:0] in_multiplier;
        input [7:0] in_product;
        input in_sign;
        input [7:0] in_magnitude;
        begin
            $display("Test Case: %d * %d, Product: %d, Sign: %b, Magnitude: %d",
                    $signed(in_multiplicand), $signed(in_multiplier), $signed(in_product), in_sign,
in_magnitude);
        end
    endtask
    // Testbench logic
    initial
    begin
        // Initialize signals
        clk = 0;
        rst = 0;
        multiplicand = 0;
        multiplier = 0;
        // Apply reset
        rst = 1;
        #20;
        rst = 0;
```

```
        // Test case: -7 x 3
        multiplicand = 4'b1001;  // -7 in 2's complement
        multiplier = 4'b0011;    // 3
        #1000;
        print(multiplicand, multiplier, product, sign, magnitude);
        #10;
         // Test case: -7 x 3
        multiplicand = 4'b0001;  // -7 in 2's complement
        multiplier = 4'b0011;    // 3
        #1000;
        print(multiplicand, multiplier, product, sign, magnitude);
        #10;
           $stop; // End simulation
    end
endmodule
```

Output:-