Experiment No: 6 Date:

4x4 MULTIPLIER

Aim: Write HDL code and implement on FPGA for 4 x 4 Multiplier

Verilog

```
 module half_adder(input a, b, output s0, c0);
assign s0 = a ^ b;
assign c0 = a & b;
endmodule

module full_adder(input a, b, cin, output s0, c0);
assign s0 = a ^ b ^ cin;
assign c0 = (a & b) | (b & cin) | (a & cin);
endmodule

module array_multiplier(input [3:0] A, B, output [7:0] z);
reg signed p[4][4];
wire [10:0] c; // c represents carry of HA/FA
wire [5:0] s;  // s represents sum of HA/FA
// For ease and readability, two diffent name s and c are used instead of single wire name.
genvar g;

generate
  for(g = 0; g<4; g++) begin
    and a0(p[g][0], A[g], B[0]);
    and a1(p[g][1], A[g], B[1]);
    and a2(p[g][2], A[g], B[2]);
    and a3(p[g][3], A[g], B[3]);
  end
  endgenerate
   assign z[0] = p[0][0];
```

```
//row 0
half_adder h0(p[0][1], p[1][0], z[1], c[0]);
half_adder h1(p[1][1], p[2][0], s[0], c[1]);
half_adder h2(p[2][1], p[3][0], s[1], c[2]);

//row1
full_adder f0(p[0][2], c[0], s[0], z[2], c[3]);
full_adder f1(p[1][2], c[1], s[1], s[2], c[4]);
full_adder f2(p[2][2], c[2], p[3][1], s[3], c[5]);

//row2
full_adder f3(p[0][3], c[3], s[2], z[3], c[6]);
full_adder f4(p[1][3], c[4], s[3], s[4], c[7]);
full_adder f5(p[2][3], c[5], p[3][2], s[5], c[8]);

//row3
half_adder h3(c[6], s[4], z[4], c[9]);
full_adder f6(c[9], c[7], s[5], z[5], c[10]);
full_adder f7(c[10], c[8], p[3][3], z[6], z[7]);
 endmodule
```

Test Bench:

Results:

Experiment No: 7 Date:

EVEN PARITY

Aim: Write a Verilog code using task and function that will check the parity of a word for even parity and implement the same on FPGA

Verilog

```
function [4:0] parity;
 input [3:0] A;
 reg temp_parity;
 begin
 temp_parity = A[0] ^ A[1] ^ A[2] ^ A[3];

parity = {A, temp_parity};
 end
endfunction

module function_test(Z);
 output reg [4:0] Z;
 reg [3:0] INP;
 initial
 begin
 INP = 4'b0101;
 Z = parity(INP);
 end

endmodule
```

Test Bench:

Results:

Experiment No: 8 Date:

LINEAR FEEDBACK SHIFT REGISTER(LFSR)

Aim: Write a Verilog code for 4 bit Linear feedback shift register(LFSR) using FPGA implementation.

Verilog:

```
module LFSR #(parameter NUM_BITS=3)
  (input i_Clk, input i_Enable,
 // Optional Seed Valueinput
 i_Seed_DV,
    input [NUM_BITS-1:0] i_Seed_Data,
    output [NUM_BITS-1:0] o_LFSR_Data,
    output o_LFSR_Done);
 reg [NUM_BITS:1] r_LFSR = 0;
 reg r_XNOR;
 reg clkdiv;
    reg[24:0] div;
  always@(posedge i_Clk)
  begin
  div=div+1'b1;
  clkdiv=div[24];
  end
  always@(posedge clkdiv)

 // Purpose: Load up LFSR with Seed if Data Valid (DV) pulse is detected.
 // Othewise just run LFSR when enabled.begin
 if (i_Enable == 1'b1)begin
 if    (i_Seed_DV    ==    1'b1)
 r_LFSR <= i_Seed_Data;else
 r_LFSR <= {r_LFSR[NUM_BITS-1:1], r_XNOR};
 end
 end
```

```
// Create Feedback Polynomials. Based on Application Note:
//

always @(*)
begin
case (NUM_BITS)3:
begin
r_XNOR = r_LFSR[3] ^~ r_LFSR[2];
end
4: begin
r_XNOR = r_LFSR[4] ^~ r_LFSR[3];
end
5: begin
r_XNOR = r_LFSR[5] ^~ r_LFSR[3];
end
6: begin
r_XNOR = r_LFSR[6] ^~ r_LFSR[5];
end
7: begin
r_XNOR = r_LFSR[7] ^~ r_LFSR[6];
end
8: begin
r_XNOR = r_LFSR[8] ^~ r_LFSR[6] ^~ r_LFSR[5] ^~ r_LFSR[4];end
9: begin
r_XNOR = r_LFSR[9] ^~ r_LFSR[5]
    end
10: begin
r_XNOR = r_LFSR[10] ^~ r_LFSR[7];
end
11: begin
r_XNOR = r_LFSR[11] ^~ r_LFSR[9];
```

```
end
12: begin
r_XNOR = r_LFSR[12] ^~ r_LFSR[6] ^~ r_LFSR[4] ^~ r_LFSR[1];end
13: begin
r_XNOR = r_LFSR[13] ^~ r_LFSR[4] ^~ r_LFSR[3] ^~ r_LFSR[1];end
14: begin
r_XNOR = r_LFSR[14] ^~ r_LFSR[5] ^~ r_LFSR[3] ^~ r_LFSR[1];end
15: begin
r_XNOR = r_LFSR[15] ^~ r_LFSR[14];
end
16: begin
r_XNOR = r_LFSR[16] ^~ r_LFSR[15] ^~ r_LFSR[13] ^~ r_LFSR[4];end
17: begin
r_XNOR = r_LFSR[17] ^~ r_LFSR[14];
end
18: begin
r_XNOR = r_LFSR[18] ^~ r_LFSR[11];
end
19: begin
r_XNOR = r_LFSR[19] ^~ r_LFSR[6] ^~ r_LFSR[2] ^~ r_LFSR[1];end
20: begin
r_XNOR = r_LFSR[20] ^~ r_LFSR[17];
end
21: begin
r_XNOR = r_LFSR[21] ^~ r_LFSR[19];
end
22: begin
r_XNOR = r_LFSR[22] ^~ r_LFSR[21];
end
23: begin
r_XNOR = r_LFSR[23] ^~ r_LFSR[18];
```

```
end
24: begin
r_XNOR = r_LFSR[24] ^~ r_LFSR[23] ^~ r_LFSR[22] ^~ r_LFSR[17];
end
25: begin
r_XNOR = r_LFSR[25] ^~ r_LFSR[22];
end
26: begin
r_XNOR = r_LFSR[26] ^~ r_LFSR[6] ^~ r_LFSR[2] ^~ r_LFSR[1];
end
27: begin
r_XNOR = r_LFSR[27] ^~ r_LFSR[5] ^~ r_LFSR[2] ^~ r_LFSR[1];end
28: begin
r_XNOR = r_LFSR[28] ^~ r_LFSR[25];
end29: begin
r_XNOR = r_LFSR[29] ^~ r_LFSR[27];
end
30: begin
r_XNOR = r_LFSR[30] ^~ r_LFSR[6] ^~ r_LFSR[4] ^~ r_LFSR[1];end
31: begin
r_XNOR = r_LFSR[31] ^~ r_LFSR[28];
end
32: begin
r_XNOR = r_LFSR[32] ^~ r_LFSR[22] ^~ r_LFSR[2] ^~ r_LFSR[1];end
endcase // case (NUM_BITS)end //
always @ (*)
assign o_LFSR_Data = r_LFSR[NUM_BITS:1];
// Conditional Assignment (?)
   assign o_LFSR_Done = (r_LFSR[NUM_BITS:1] == i_Seed_Data) ? 1'b1 : 1'b0;endmodule
 // LFSR
```

Test Bench:

Results:

Experiment No: 9 Date:

SYNCHRONOUS COUNTERS

a) SYNCHRONOUS BINARY COUNTER

Aim: Write a HDL code to design 4-bit synchronous binary counter.

Verilog

```
module syncbinary (clk, rst, q);

input clk, rst;

output [3:0]q;

reg [3:0]q;

initial q = 4'B0000;

always @ (posedge clk)

begin

if (rst == 1'B1)

q = 4'B0000;

else

q = q + 1;

end endmodule
```

Test Bench:

Hardware Implementation code:

```
module syncbinary (clk, rst, q);
input clk, rst;
output [3:0]q;
reg [3:0]q;
reg clkdiv;
reg[24:0] div;
initial q = 4'B0000;
always @ (posedge clk)
begin
div=div+1'B1;
clkdiv=div[24];
end
always@(posedge clkdiv)
begin
if (rst == 1'B1)
q = 4'B0000;
else
q = q + 1;
end endmodule
```

Results:

a) SYNCHRONOUS BCD COUNTER

Aim: Write a HDL code to design 4-bit synchronous BCD counter.

Verilog

```
module syncbcd (clk, rst, q);
input clk, rst;
output [3:0]q;
reg [3:0]q;
initial q = 4’b0000;
always @ (posedge clk)
begin
if (rst == 1’b1|q==4’b1001)
q = 4’b0000;
else q=q+1;
end endmodule
```

Test Bench:

Note: Hardware implementation code needs to be written.

Results:

Experiment No: 10 Date:

ASYNCHRONOUS COUNTERS

ASYNCHRONOUS BINARY COUNTER

Aim: Write a HDL code to design 4-bit Asynchronous Binary counter.

Verilog

```
module asyncbinary (clk, rst, q);
 input clk, rst;
output [3:0]q;
reg [3:0]q;
initial q = 4'b0000;
always @ (posedge clk or posedge rst)
begin
if (rst == 1'b1)
q = 4'b0000;
else q=q+1;
end endmodule
```

Test Bench:

Note: Hardware implementation code needs to be written.

Results:

b) ASYNCHRONOUS BCD COUNTER

Aim: Write a HDL code to design 4-bit Asynchronous BCD counter.

Verilog

```
module asyncbcd (clk, rst, q);
input clk, rst;
output [3:0]q;
reg [3:0]q;
initial q = 4’b0000;
always @ (posedge clk or posedge rst)
begin
if (rst == 1’b1)q = 4’b0000;
else if (q== 4’b1001) q = 4’b0000;
else
q=q+1;
end
endmodule
```

Test Bench:

Hardware Implementation code:

```
module asyncbcd (clk, rst, q);
input clk, rst;
output [3:0]q;
reg [3:0]q;
reg clkdiv;
 reg[22:0] div;
initial q = 4'b0000;
 always @ (posedge clk )
 begin
div=div+1'B1;
 clkdiv=div[22];
end
always@(posedge clkdiv or posedge rst)
 begin
if (rst == 1'b1)
 q = 4'b0000;
else if (q== 4'b1001)
 q = 4'b0000;
else
 q=q+1;
 end endmodule
```

Results:

Experiment No: 11 Date:

MEALY AND MOORE STATE MACHINES

a) MEALY STATE MACHINE

Aim: Write a HDL code to design Mealy state machine for the sequence 101.

Verilog

```
 //Mealy Sequence -101
  module seq_mealy(
 input x, input clk , input rst , output reg y );
  reg [1:0]cst;
 reg [1:0]nst;
 parameter s0 = 2'b00 , s1 = 2'b01 , s2 = 2'b10;
  always @(cst or x)
 begin
  case(cst)
 s0: begin
        if (x)
                begin
                nst = s1;
                 y=0;
                end
                else
                begin
                nst = cst;
                y=0;
                end
                 end
 s1: begin
        if (x==0)
         begin
        nst = s2;
        y=0;
        end
        else
        begin
        nst = cst;
         y=0;
        end
        end
s2: begin
        if (x)
        begin
         nst = s1;
         y=1;
        end
        else
         begin
        nst = s0;
         y=0;
        end
         end
        default: nst = s0 ;
endcase
 end
always@(posedge clk)
begin
if(rst)
cst <= s0;
 else
cst <= nst;
end
endmodule
```

Test Bench:

```
module seq_mealy_tb;
// Inputs
reg x; reg clk; reg rst;
// Outputs
wire y;
// Instantiate the Unit Under Test (UUT)
        seq_mealy uut (
                    .x(x),
                    .clk(clk),
                    .rst(rst),
                    .y(y)
                    );
reg [19:0]data;
        integer k;
        initial begin
                    // Initialize Inputs
                    data = 20'b10100101011101010101;
                    clk = 0;
                    k=0;
                    rst = 1;
                    #60 rst =0 ;
                    // Wait 100 ns for global reset to finish
                     #340; // Add stimulus here
                                end
                                always@(posedge clk)
                                begin
                                x = data>>k;
                                k = k+1;
                                end
                                always #20 clk=~clk;
```

```
endmodule
```

Results:

b) MOORE STATE MACHINE

Aim: Write a HDL code to design Moore state machine to detect the Sequence -101

Verilog

```
module seq_moore(input x, input clk , input rst , output reg y );
 reg [1:0]cst;
reg [1:0]nst;
parameter s0 = 2'b00 , s1 = 2'b01 , s2 = 2'b10 , s3 = 2'b11;
 always @(cst or x)
begin
case(cst)
s0: begin
y=0;
            if (x)
            nst = s1;
             else
            nst = cst;
             end
s1: begin
            y=0;
                        if (x==0)
                        nst = s2;
                        else
                        nst = cst;
                         end
s2: begin
            y=0;
                        if (x)
                         nst = s3;
                        else
                        nst = s0;
                        end
```

```
s3: begin
y=1;
        if (x)
        nst = s1;
        else
        nst = s2;
        end
        default: nst = s0 ;
        endcase
        end
        always@(posedge clk)
         begin
        if(rst)
        cst <= s0;
        else
        cst <= nst;
        end
        endmodule
```

Test Bench:

```
module seq_moore_tb;
        // Inputs
         reg x;
         reg clk;
         reg rst;
        // Outputs
         wire y;
        // Instantiate the Unit Under Test (UUT)
        seq_moore uut (
        .x(x),
        .clk(clk),
        .rst(rst),
.y(y)
);
reg [15:0]data;
integer k;
 initial begin
        // Initialize Inputs
        data = 16'b1010010101110101;
        clk = 0;
        k=0;
        rst = 1;
        #60 rst =0 ;
        // Wait 100 ns for global reset to finish
         #500;
        // Add stimulus here
        $stop;
end
always@(posedge clk)
 begin
x = data>>k;
 k = k+1;
end
always #20 clk=~clk;
endmodule
```

Results: