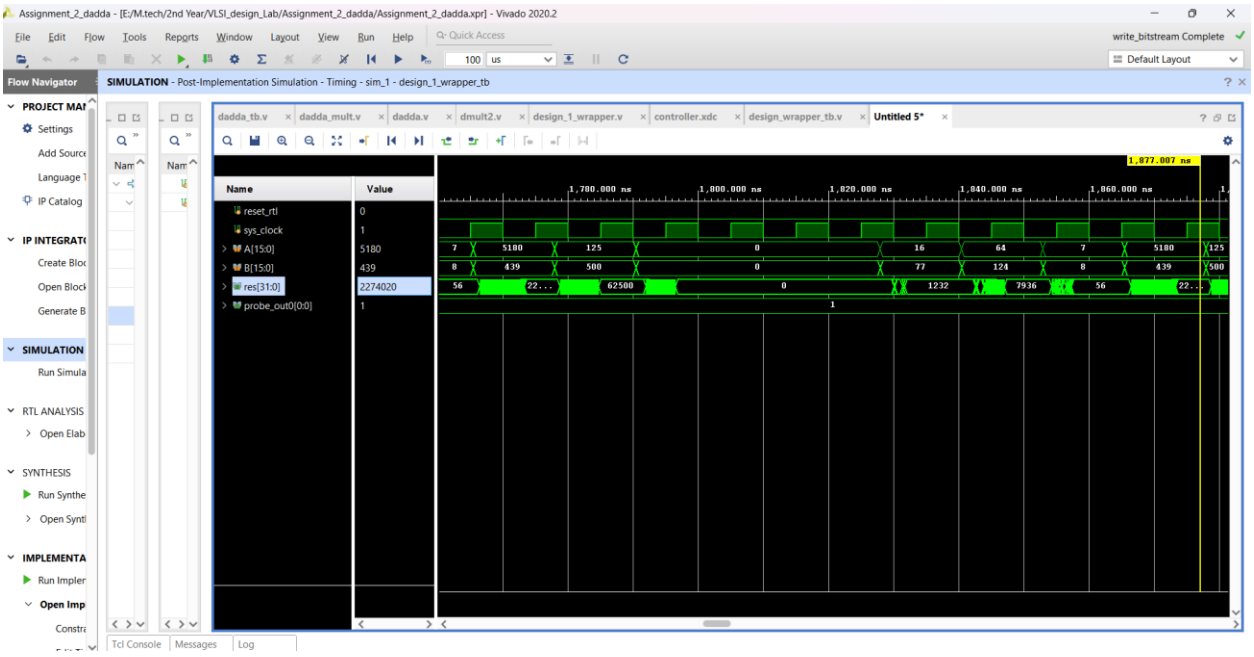
VLSI design Lab

Manish Ranjan

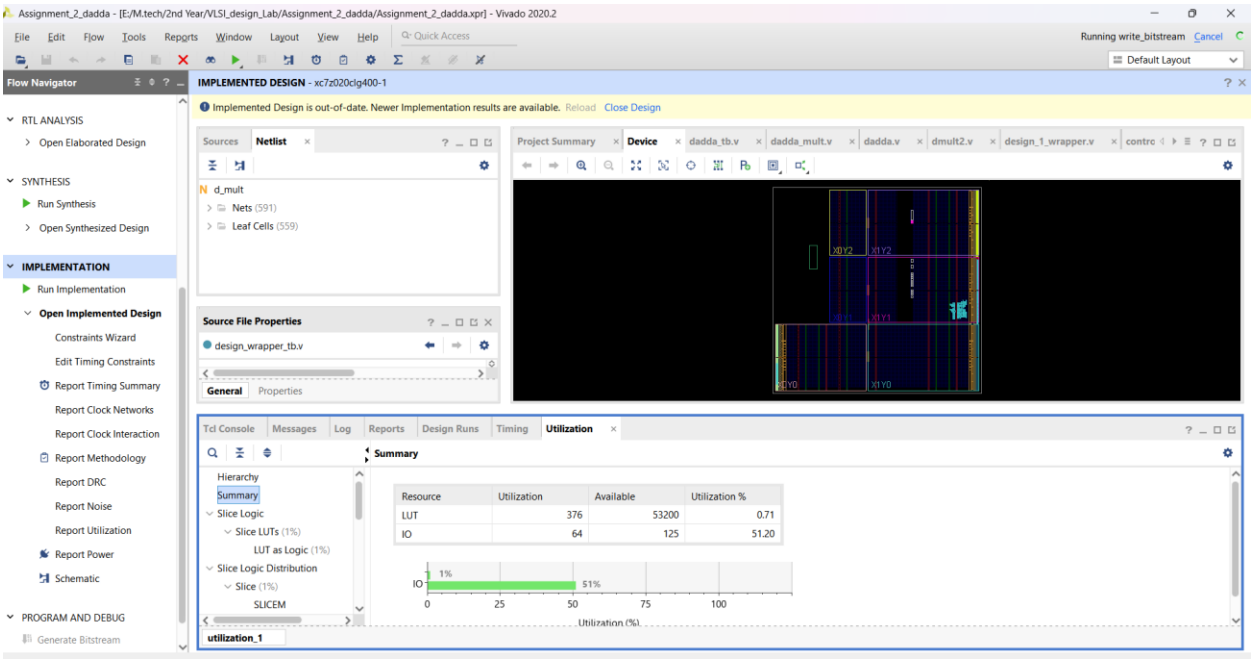
24M1176

A. Dadda Multiplier

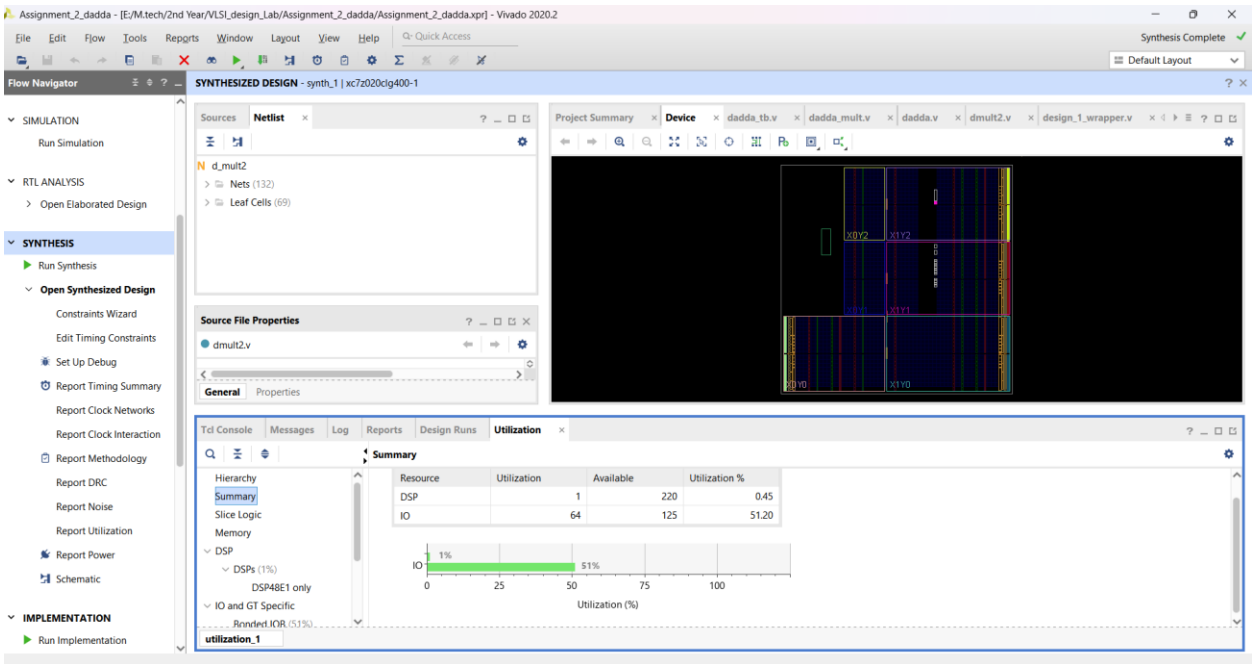
1. Post Implementation Timing Simulation



2. Dadda Multiplier Utization



3. Dadda Multiplier utilization with operator



Codes

Verilog code:

```
module d_mult(
  input [15:0] A,
  input [15:0] B,
  output [31:0] res
);

reg [30:0] pp[15:0];
integer i, j;
reg [1:0] ha_tmp;
reg [1:0] fa_tmp, fa_tmp1, fa_tmp2, fa_tmp3;
reg c_tmp, c_tmp1, c_tmp2, c_tmp3;
reg [31:0] s_temp, s_temp1;

task half_adder;
  input p, q;
  output [1:0] result;
  begin
    result[0] = p ^ q;
    result[1] = p & q;
  end
endtask
```

```
task full_adder;
 input m, n, cin;
 output [1:0] result;
 begin
  result[0] = m ^ n ^ cin;
  result[1] = (m & n) | (n & cin) | (m & cin);
 end
endtask

always @(*)
begin
  // Initialize partial products to zero
  for (i = 0; i < 16; i = i + 1)
  begin
    pp[i] = 31'd0;
  end

  // Partial products calculation
  for (i = 0; i < 16; i = i + 1)
  begin
    for (j = 0; j < 16; j = j + 1)
    begin
      pp[i][j + i] = B[i] & A[j];
    end
```

```
end

// Creating triangle structure for better visualization and reducing complexity
for (j = 16; j < 31; j = j + 1)
begin
  for (i = (j - 15); i < 16; i = i + 1)
  begin
    pp[i - (j - 15)][j] = pp[i][j];
    pp[i][j] = 0;
  end
end

// Reduction to d = 13
half_adder(pp[12][13], pp[13][13], ha_tmp);
pp[12][13] = ha_tmp[0];
c_tmp = ha_tmp[1];

full_adder(pp[12][14], pp[13][14], pp[14][14], fa_tmp);
half_adder(fa_tmp[0], c_tmp, ha_tmp);
pp[12][14] = ha_tmp[0];
c_tmp = fa_tmp[1];
c_tmp1 = ha_tmp[1];

full_adder(pp[13][15], pp[14][15], pp[15][15], fa_tmp1);
full_adder(pp[12][15], c_tmp1, c_tmp, fa_tmp2);
half_adder(fa_tmp1[0], fa_tmp2[0], ha_tmp);
```

```
pp[12][15] = ha_tmp[0];
c_tmp = fa_tmp1[1];
c_tmp1 = fa_tmp2[1];
c_tmp2 = ha_tmp[1];

full_adder(pp[12][16], pp[13][16], pp[14][16], fa_tmp1);
full_adder(c_tmp2, c_tmp1, c_tmp, fa_tmp2);
half_adder(fa_tmp1[0], fa_tmp2[0], ha_tmp);
pp[12][16] = ha_tmp[0];
c_tmp = fa_tmp1[1];
c_tmp1 = fa_tmp2[1];
c_tmp2 = ha_tmp[1];

full_adder(c_tmp2, pp[12][17], pp[13][17], fa_tmp1);
full_adder(fa_tmp1[0], c_tmp1, c_tmp, fa_tmp2);
pp[12][17] = fa_tmp2[0];
c_tmp = fa_tmp1[1];
c_tmp1 = fa_tmp2[1];

full_adder(pp[12][18], c_tmp, c_tmp1, fa_tmp1);
pp[12][18] = fa_tmp1[0];
pp[12][19] = fa_tmp1[1];

// Reduction to d = 9
half_adder(pp[8][9], pp[9][9], ha_tmp);
pp[8][9] = ha_tmp[0];
```

```
c_tmp = ha_tmp[1];

full_adder(pp[8][10], pp[9][10], pp[10][10], fa_tmp);
half_adder(fa_tmp[0], c_tmp, ha_tmp);
pp[8][10] = ha_tmp[0];
c_tmp = fa_tmp[1];
c_tmp1 = ha_tmp[1];

full_adder(pp[9][11], pp[10][11], pp[11][11], fa_tmp1);
full_adder(pp[8][11], c_tmp1, c_tmp, fa_tmp2);
half_adder(fa_tmp1[0], fa_tmp2[0], ha_tmp);
pp[8][11] = ha_tmp[0];
c_tmp = fa_tmp1[1];
c_tmp1 = fa_tmp2[1];
c_tmp2 = ha_tmp[1];

full_adder(pp[10][12], pp[11][12], pp[12][12], fa_tmp);
full_adder(pp[8][12], pp[9][12], c_tmp, fa_tmp1);
full_adder(fa_tmp[0], c_tmp1, c_tmp2, fa_tmp2);
half_adder(fa_tmp1[0], fa_tmp2[0], ha_tmp);
pp[8][12] = ha_tmp[0];
c_tmp = fa_tmp[1];
c_tmp1 = fa_tmp1[1];
c_tmp2 = fa_tmp2[1];
c_tmp3 = ha_tmp[1];
```

```
for (j = 13; j < 20; j = j + 1)
begin
   full_adder(pp[10][j], pp[11][j], pp[12][j], fa_tmp);
   full_adder(pp[8][j], pp[9][j], c_tmp, fa_tmp1);
   full_adder(fa_tmp[0], fa_tmp1[0], c_tmp1, fa_tmp2);
   full_adder(fa_tmp2[0], c_tmp2, c_tmp3, fa_tmp3);
   pp[8][j] = fa_tmp3[0];
   c_tmp = fa_tmp[1];
   c_tmp1 = fa_tmp1[1];
   c_tmp2 = fa_tmp2[1];
   c_tmp3 = fa_tmp3[1];
end

full_adder(pp[8][20], pp[9][20], pp[10][20], fa_tmp);
full_adder(fa_tmp[0], c_tmp, c_tmp1, fa_tmp1);
full_adder(fa_tmp1[0], c_tmp2, c_tmp3, fa_tmp2);
pp[8][20] = fa_tmp2[0];
c_tmp = fa_tmp[1];
c_tmp1 = fa_tmp1[1];
c_tmp2 = fa_tmp2[1];

full_adder(c_tmp, pp[8][21], pp[9][21], fa_tmp);
full_adder(fa_tmp[0], c_tmp1, c_tmp2, fa_tmp1);
pp[8][21] = fa_tmp1[0];
c_tmp = fa_tmp[1];
c_tmp1 = fa_tmp1[1];
```

```
full_adder(c_tmp, c_tmp1, pp[8][22], fa_tmp);
pp[8][22] = fa_tmp[0];
pp[8][23] = fa_tmp[1];

// Reduction to d = 6
half_adder(pp[5][6], pp[6][6], ha_tmp);
pp[5][6] = ha_tmp[0];
c_tmp = ha_tmp[1];

full_adder(pp[5][7], pp[6][7], pp[7][7], fa_tmp);
half_adder(fa_tmp[0], c_tmp, ha_tmp);
pp[5][7] = ha_tmp[0];
c_tmp = fa_tmp[1];
c_tmp1 = ha_tmp[1];

full_adder(pp[6][8], pp[7][8], pp[8][8], fa_tmp1);
full_adder(pp[5][8], c_tmp1, c_tmp, fa_tmp2);
half_adder(fa_tmp1[0], fa_tmp2[0], ha_tmp);
pp[5][8] = ha_tmp[0];
c_tmp = fa_tmp1[1];
c_tmp1 = fa_tmp2[1];
c_tmp2 = ha_tmp[1];

for (j = 9; j < 24; j = j + 1)
begin
```

```
    full_adder(pp[8][j], pp[7][j], pp[6][j], fa_tmp);
    full_adder(pp[5][j], c_tmp, c_tmp1, fa_tmp1);
    full_adder(fa_tmp[0], fa_tmp1[0], c_tmp2, fa_tmp2);
    pp[5][j] = fa_tmp2[0];
    c_tmp = fa_tmp[1];
    c_tmp1 = fa_tmp1[1];
    c_tmp2 = fa_tmp2[1];
end

full_adder(pp[6][24], pp[5][24], c_tmp, fa_tmp);
full_adder(fa_tmp[0], c_tmp1, c_tmp2, fa_tmp1);
pp[5][24] = fa_tmp1[0];
c_tmp = fa_tmp[1];
c_tmp1 = fa_tmp1[1];

full_adder(pp[5][25], c_tmp, c_tmp1, fa_tmp);
pp[5][25] = fa_tmp[0];
pp[5][26] = fa_tmp[1];

// Reduction to d = 4
half_adder(pp[3][4], pp[4][4], ha_tmp);
pp[3][4] = ha_tmp[0];
c_tmp = ha_tmp[1];

full_adder(pp[3][5], pp[4][5], pp[5][5], fa_tmp);
half_adder(fa_tmp[0], c_tmp, ha_tmp);
```

```
pp[3][5] = ha_tmp[0];
c_tmp = fa_tmp[1];
c_tmp1 = ha_tmp[1];

for (j = 6; j < 27; j = j + 1)
begin
   full_adder(pp[3][j], pp[4][j], pp[5][j], fa_tmp);
   full_adder(c_tmp, c_tmp1, fa_tmp[0], fa_tmp1);
   pp[3][j] = fa_tmp1[0];
   c_tmp = fa_tmp[1];
   c_tmp1 = fa_tmp1[1];
end

full_adder(pp[3][27], c_tmp, c_tmp1, fa_tmp);
pp[3][27] = fa_tmp[0];
pp[3][28] = fa_tmp[1];

// Reduction to d = 3
half_adder(pp[2][3], pp[3][3], ha_tmp);
pp[2][3] = ha_tmp[0];
c_tmp = ha_tmp[1];

for (j = 4; j < 29; j = j + 1)
begin
   full_adder(pp[3][j], pp[2][j], c_tmp, fa_tmp);
   pp[2][j] = fa_tmp[0];
```

```
    c_tmp = fa_tmp[1];
end
pp[2][29] = c_tmp;

// Reduction to d = 2
half_adder(pp[1][2], pp[2][2], ha_tmp);
pp[1][2] = ha_tmp[0];
c_tmp = ha_tmp[1];

for (j = 3; j < 30; j = j + 1)
begin
    full_adder(pp[1][j], pp[2][j], c_tmp, fa_tmp);
    pp[1][j] = fa_tmp[0];
    c_tmp = fa_tmp[1];
end
pp[1][30] = c_tmp;

for (j = 0; j < 31; j = j + 1)
begin
    s_temp[j] = pp[0][j];
end

for (j = 0; j < 31; j = j + 1)
begin
    s_temp1[j] = pp[1][j];
end
```

```
   s_temp[31] = 0;
   s_temp1[31] = 0;
end

wire cout;
bkadder_32 add (
   .A(s_temp),
   .B(s_temp1),
   .cin(1'b0),
   .Y(res),
   .cout(cout)
);

endmodule
```

Testbench

```
`timescale 1ns / 1ps

module dadda_multiplier_tb;
 reg [15:0] A,B;
 wire [31:0] Y;

 d_mult dut(.A(A),.B(B),.res(Y));

   initial  begin
```

```
    $monitor("%d %d %d",A,B,Y);
      A = 0;
      B = 0;
      #200 $finish;
    end

   always begin
      A = $random;
      B = $random;
      #10;
   end

Endmodule
```

Control unit:

```
`timescale 1ns / 1ps

module ControlUnit(
   input clka,
   input [31:0] douta,
   input  start_stop,
   output reg ena,
   output reg [2:0] addra,
   output reg [15:0] a,
   output reg [15:0] b
);
```

```
    always @(posedge clka) begin
        if(!start_stop)
        begin
            a <= 16'b0;
            b <= 16'b0;
            ena <= 0;
            addra <= 3'b000;
        end
        else
        begin
            a <= douta[31:16];
            b <= douta[15:0];
            ena <= 1;
            addra <= addra + 1;
        end
    end
endmodule
```

2. **Barrel Shifter**

Verilog codes :

1. Logarithmic Barrel shifter code

```
`timescale 1ns / 1ps
module Barrel_Shifter(

  input [7:0] in1,   // 8-bit input data
  input [2:0] shift,    // 3-bit shift amount (0 to 7)
  input LR,          // Direction: 0 for left, 1 for right
  output [7:0] data_out  );



  wire p1,p2,p3,p4,p5,p6,p7,p8;
  wire q1,q2,q3,q4,q5,q6,q7,q8;
  wire r1,r2,r3,r4,r5,r6,r7,r8;
  wire [7:0] out;

  // selecting Inputs based on LR

  mux2x1 m1(in1[0],in1[7],LR,p1);
  mux2x1 m2(in1[1],in1[6],LR,p2);
  mux2x1 m3(in1[2],in1[5],LR,p3);
  mux2x1 m4(in1[3],in1[4],LR,p4);
  mux2x1 m5(in1[4],in1[3],LR,p5);
  mux2x1 m6(in1[5],in1[2],LR,p6);
  mux2x1 m7(in1[6],in1[1],LR,p7);
  mux2x1 m8(in1[7],in1[0],LR,p8);
```

```
//
 mux2x1 n1(p1,p2,shift[0],q1);
 mux2x1 n2(p2,p3,shift[0],q2);
 mux2x1 n3(p3,p4,shift[0],q3);
 mux2x1 n4(p4,p5,shift[0],q4);
 mux2x1 n5(p5,p6,shift[0],q5);
 mux2x1 n6(p6,p7,shift[0],q6);
 mux2x1 n7(p7,p8,shift[0],q7);
 mux2x1 n8(p8,1'b0,shift[0],q8);

 mux2x1 o1(q1,q3,shift[1],r1);
 mux2x1 o2(q2,q4,shift[1],r2);
 mux2x1 o3(q3,q5,shift[1],r3);
 mux2x1 o4(q4,q6,shift[1],r4);
 mux2x1 o5(q5,q7,shift[1],r5);
 mux2x1 o6(q6,q8,shift[1],r6);
 mux2x1 o7(q7,1'b0,shift[1],r7);
 mux2x1 o8(q8,1'b0,shift[1],r8);

 mux2x1 k1(r1,r5,shift[2],out[7]);
 mux2x1 k2(r2,r6,shift[2],out[6]);
 mux2x1 k3(r3,r7,shift[2],out[5]);
 mux2x1 k4(r4,r8,shift[2],out[4]);
 mux2x1 k5(r5,1'b0,shift[2],out[3]);
 mux2x1 k6(r6,1'b0,shift[2],out[2]);
```

```
    mux2x1 k7(r7,1'b0,shift[2],out[1]);
    mux2x1 k8(r8,1'b0,shift[2],out[0]);

genvar i;
generate
   for (i = 0; i < 8; i = i + 1) begin : gen_block
     assign data_out[i] = LR ? out[i] : out[7-i];
 end
endgenerate




endmodule

   // 2-to-1 Multiplexer module
module mux2x1 (
   input in0,   // Input 0
   input in1,   // Input 1
   input sel,   // Select signal
   output out   // Output
);
   assign out = sel ? in1 : in0;
endmodule
```

Controller code:

```
module Control_Unit (
   input  clk,
   input  start_stop,
   input  left_right,
   input wire [10:0] douta,
   output reg [7:0] in1,
   output reg [2:0] shift,

   output reg ena,
   output reg LR,
   output reg [2:0] addra
);



  always @(posedge clk) begin
      if (!start_stop) begin
         ena <= 0;              // Disable BRAM read
         addra <= 3'b000;       // Reset BRAM address to 0
         in1 <= 8'b00000000;    // Reset input data
         shift <= 3'b000;       // Reset shift amount
      end
      else begin
         ena <= 1;
         in1 <= douta[10:3];
```

```
        shift <= douta[2:0];
        LR <= left_right ? 1 : 0;

        if (addra == 3'b111) begin
           addra <= 3'b000;
        end else begin
           addra <= addra + 1;
        end
      end
   end
endmodule
```

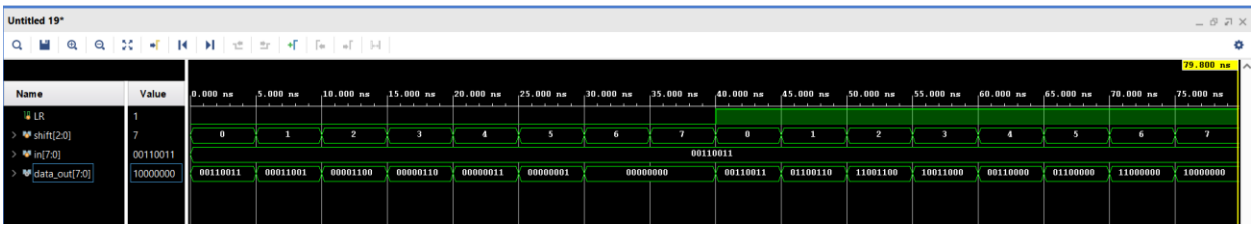
Barrel shifter Testbench

```
`timescale 1ns / 1ps

module Barrel_Shifter_tb;


   reg [7:0] in;
   reg [2:0] shift;
   reg LR;


   wire [7:0] data_out;
```

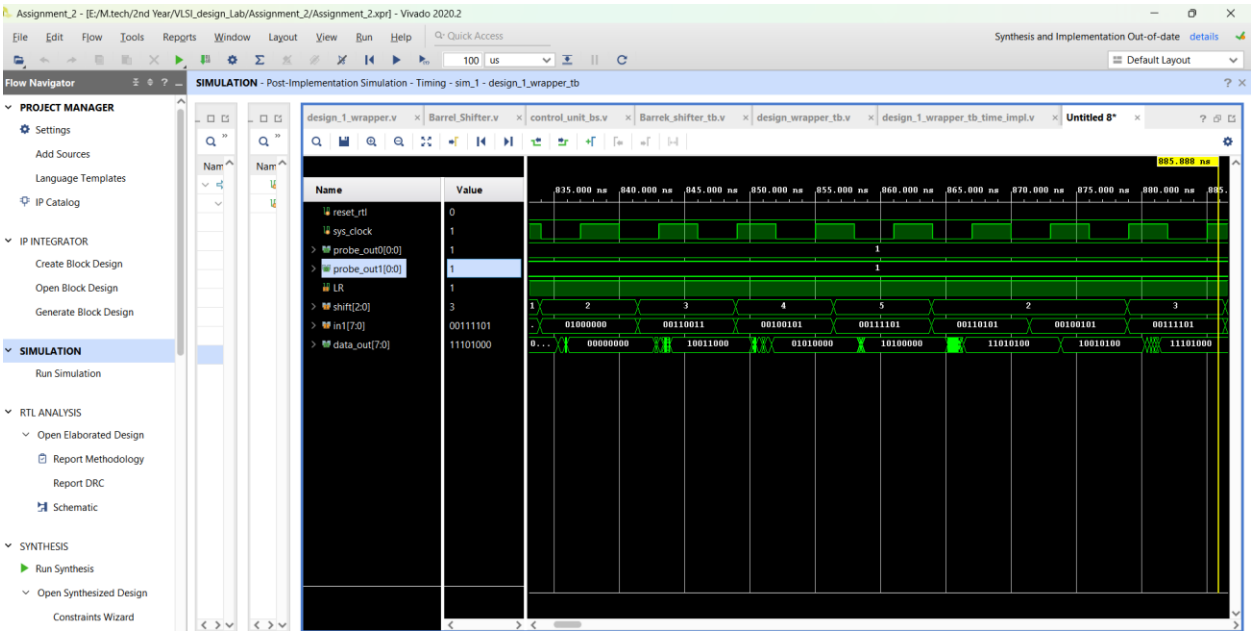
```
Barrel_Shifter_op uut (
   .in1(in),
   .shift(shift),
   .LR(LR),
   .data_out(data_out)
);



initial begin

   in = 8'b00110011;
   $display("Time\tLR\tShift\tInput\t\tOutput");



  for( LR = 0; LR<=1;LR=LR+1)
  begin
  shift = 3'b000; #5;
   $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

   shift = 3'b001; #5;
   $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

   shift = 3'b010; #5;
   $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

   shift = 3'b011; #5;
```

```
        $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

        shift = 3'b100; #5;
        $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

        shift = 3'b101; #5;
        $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

        shift = 3'b110; #5;
        $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);

        shift = 3'b111; #5;
        $display("%d\t%b\t%b\t%b\t%b", $time, LR, shift, in, data_out);
        end


        $finish;
    end
endmodule
```

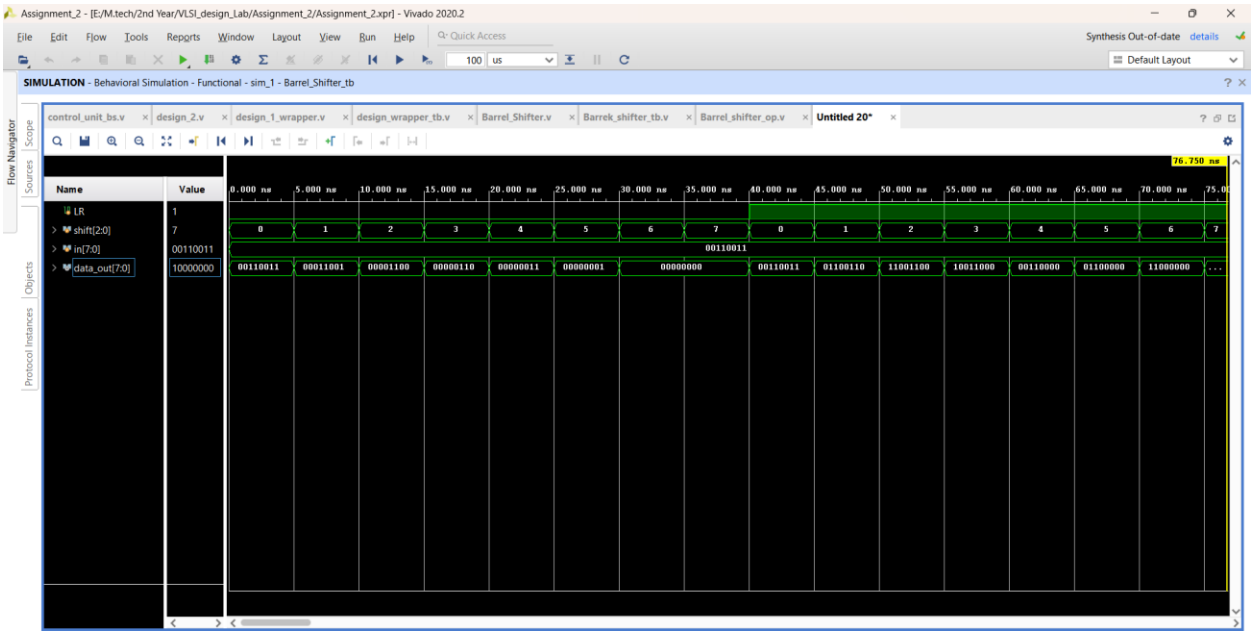
1. Behavioral simulation



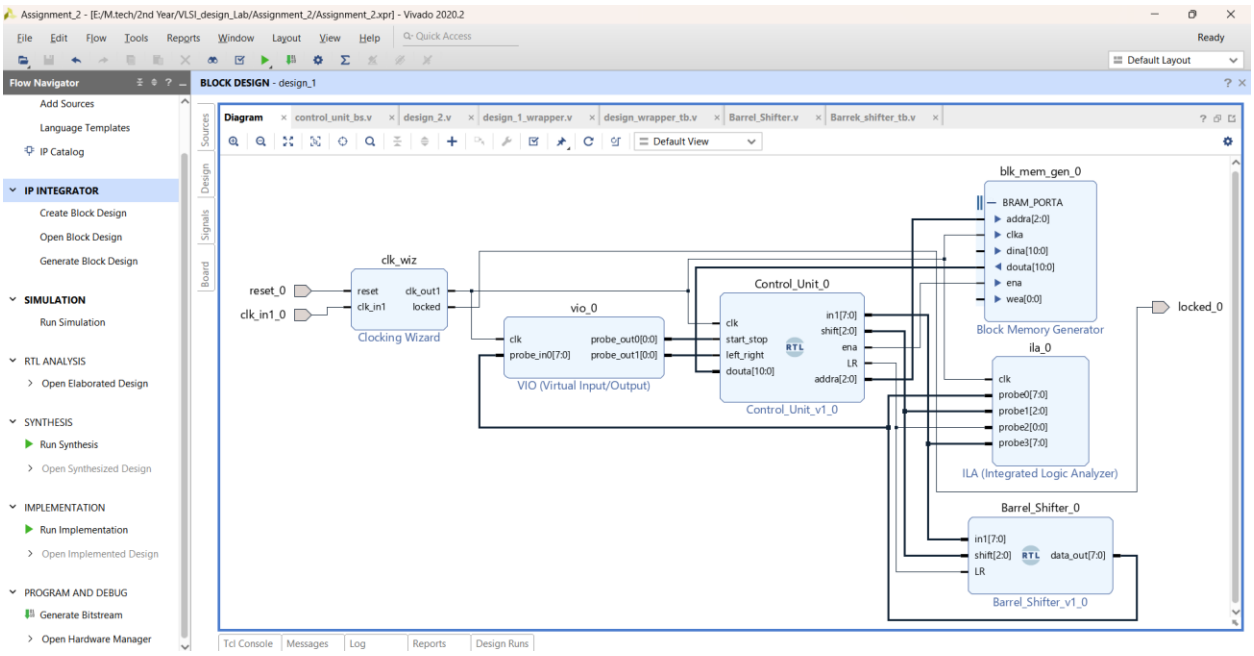
2. Post-Implemented Timing Simulation



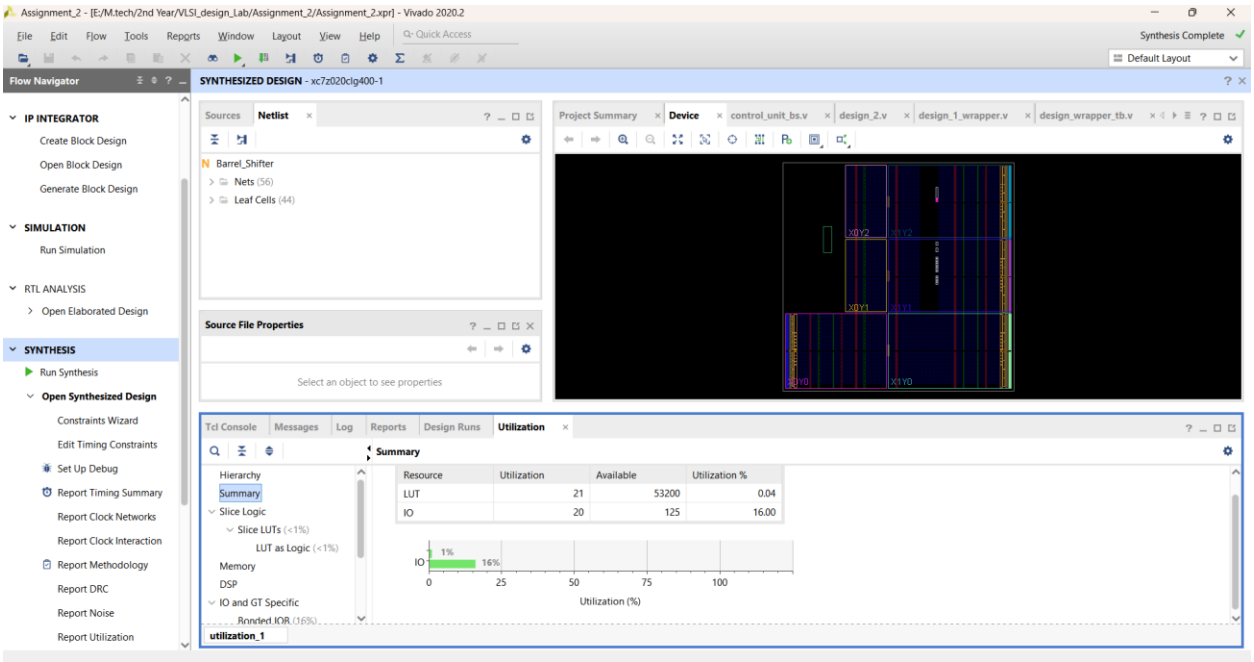
3. Barrel Shifter with Operator



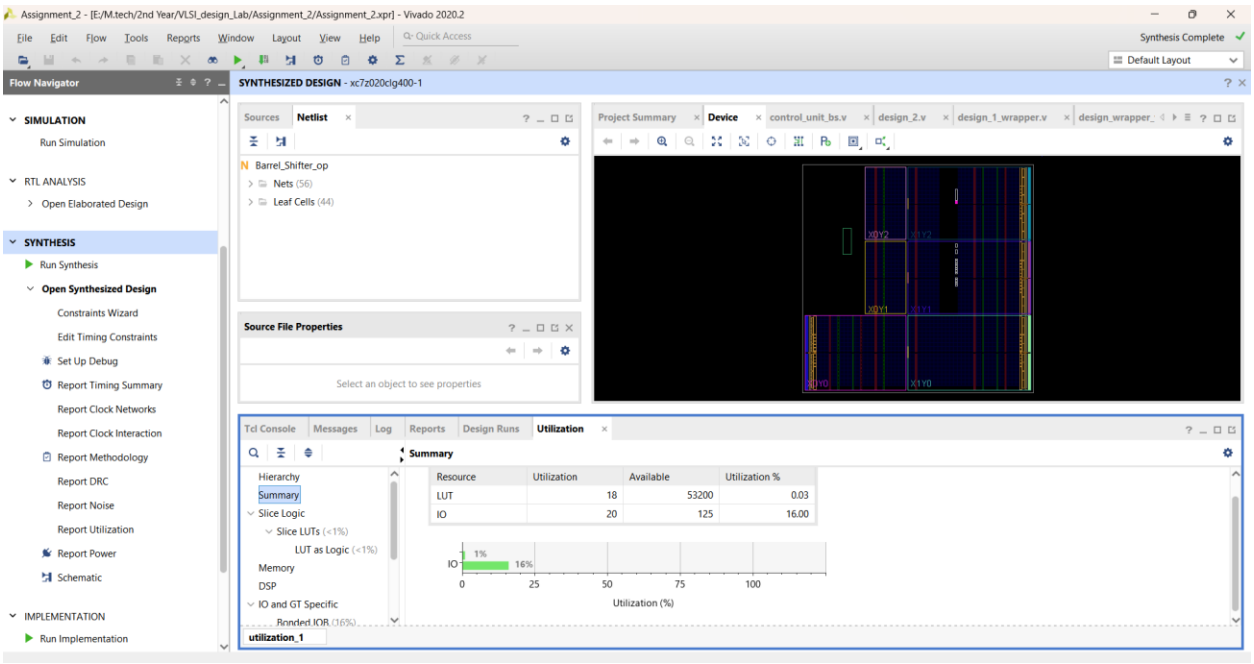
4. Block diagram



5. Logarithmic Barrel shifter utilization



6. Barrel shifter utilization with operator.



Inference : **Utilization of barrel shifter without operator is little more(3 more LUTs) then when implemented using shift operator.**