Indian Institute of Technology Bombay

EE 705 VLSI Design Lab

Assignment II

Implementation of 16 Bit Multiplier and a 8 Bit Logarithmic Barrel Shifter in FPGA

Submitted By:

Bala Murugan S

Reg No: 24M1173

Course Instructor:

Prof. Laxmeesha Somappa

Contents

List of Figures

1 TASK 1 - MULTIPLIER

To Design a 16 bit Dadda Multiplier and write a testbench and simulate the circuit.
The types of Multiplier to be designed:
(a) Using the 16 bit Dadda Multiplier
(b) Using the '*' operator only.

2 DADDA MULTIPLIER

I started designing the 16 bit Dadda Multiplier by first initializing the two dimensional array "[30:0] pp[15:0]" to zero in every indices. Then I just ANDed the respective bits of a[ ] and b[ ] and stored in the pp array at correct positions. Then I just rearranged the partial products to a cone like structure as shown in the snippet,

```
    // initialize zero
    for (i=0;i<16;i=i+1)
    begin
            pp[i]=31'd0;
    end
    // calculation  of partial products
    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
    // rearranging the partial products to a cone like structure
    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
```

The following image shows the outcome of a 12x12 multiplier. After the execution of above code this 16x16 multiplier will also look similar to this.

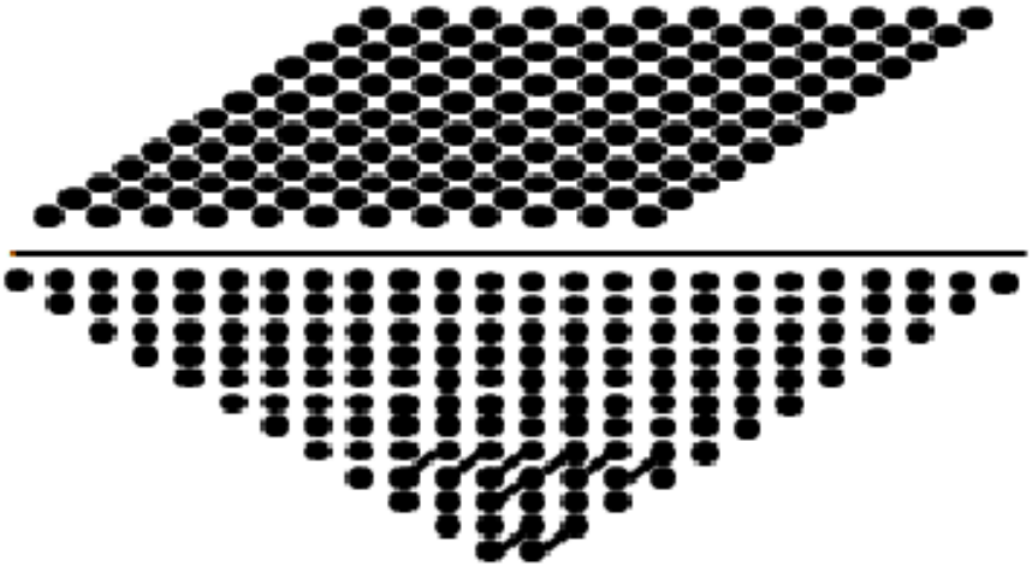

Figure 1: 12x12 Dadda Multiplier (Cone like a structure)

Then I defined two functions Half Adder and Full Adder.

```
function [1:0] half_adder;
        input a, b;
        begin
            half_adder[0] = a ^ b;
            half_adder[1] = a & b;
        end
endfunction
function [1:0] full_adder;
        input a, b, cin;
        begin
            full_adder[0] = a ^ b ^ cin;
            full_adder[1] = (a & b) | (b & cin) | (a & cin);
        end
endfunction
```

After defining those functions, I used it on the pp[ ][ ] array. The number of addends upto 13th column is just 13 bits, the value of d is 13 in the first reduction stage. So I declared a half adder in the 14th column which contains 14 bits. Here I used half adder on the last two rows of the same 14th column and and stored the result on to the 13th row of that column. The carry is forwarded to next column. Then the value of d continues to d = 9, d = 6, d = 4, d = 3, d = 2 and addition of two rows using the designed Brent Kung Adder. Here for simplicity I have attached the first reduction stage only,

```
   // starting from d = 13 first reduction stage
    half_add_res = half_adder(pp[12][13], pp[13][13]);
    pp[12][13] = half_add_res[0];
    c0 = half_add_res[1];
    full_add_res = full_adder(pp[12][14],pp[13][14], pp[14][14]);
    half_add_res = half_adder(full_add_res[0],c0);
    pp[12][14] = half_add_res[0];
    c0 = full_add_res[1];
    c1 = half_add_res[1];
    full_add_res1 = full_adder(pp[13][15],pp[14][15], pp[15][15]);
    full_add_res2 = full_adder(pp[12][15], c1, c0);
    half_add_res = half_adder(full_add_res1[0],full_add_res2[0]);
    pp[12][15] = half_add_res[0];
    c0 = full_add_res1[1];
    c1 = full_add_res2[1];
    c2 = half_add_res[1];
    full_add_res1 = full_adder(pp[12][16],pp[13][16], pp[14][16]);
    full_add_res2 = full_adder(c2, c1, c0);
    half_add_res = half_adder(full_add_res1[0],full_add_res2[0]);
    pp[12][16] = half_add_res[0];
    c0 = full_add_res1[1];
    c1 = full_add_res2[1];
    c2 = half_add_res[1];
    full_add_res1 = full_adder(c2,pp[12][17], pp[13][17]);
    full_add_res2 = full_adder(full_add_res1[0], c1, c0);
    pp[12][17] = full_add_res2[0];
    c0 = full_add_res1[1];
    c1 = full_add_res2[1];
    full_add_res1 = full_adder(pp[12][18],c0,c1);
    pp[12][18] = full_add_res1[0];
    pp[12][19] = full_add_res1[1];
 // end of first reduction stage
```

Then after completing the code of Dadda Multiplier the testbench is coded as follows,

```
`timescale 1ns / 1ps
module tb_dadda_mult;
  reg [15:0] a;
  reg [15:0] b;
  wire [31:0] result;
  dadda_mult uut (
    .a(a),
    .b(b),
    .result(result)
  );
  initial begin
    $display("Starting test cases...");
    #10 a = 16'h0010; b = 16'h004D;
    #10 $display("Test case 1: a = %h, b = %h, result = %h", a, b,
             result);
    #10 a = 16'h0040; b = 16'h007C;
```

```
    #10 $display("Test case 2: a = %h, b = %h, result = %h", a, b,
            result);
    #10 a = 16'h0007; b = 16'h0008;
    #10 $display("Test case 3: a = %h, b = %h, result = %h", a, b,
            result);
    #10 a = 16'h143C; b = 16'h01B7;
    #10 $display("Test case 4: a = %h, b = %h, result = %h", a, b,
            result);
    #10 a = 16'h007D; b = 16'h01F4;
    #10 $display("Test case 5: a = %h, b = %h, result = %h", a, b,
            result);
    $display("Test cases completed.");
    #10 $stop;
  end
endmodule
```

3 OUTPUT WAVEFORM OF BEHAVIOURAL SIMULATION

The following image shows the output waveform obtained after using the required testbench for the module "dadda_mult".

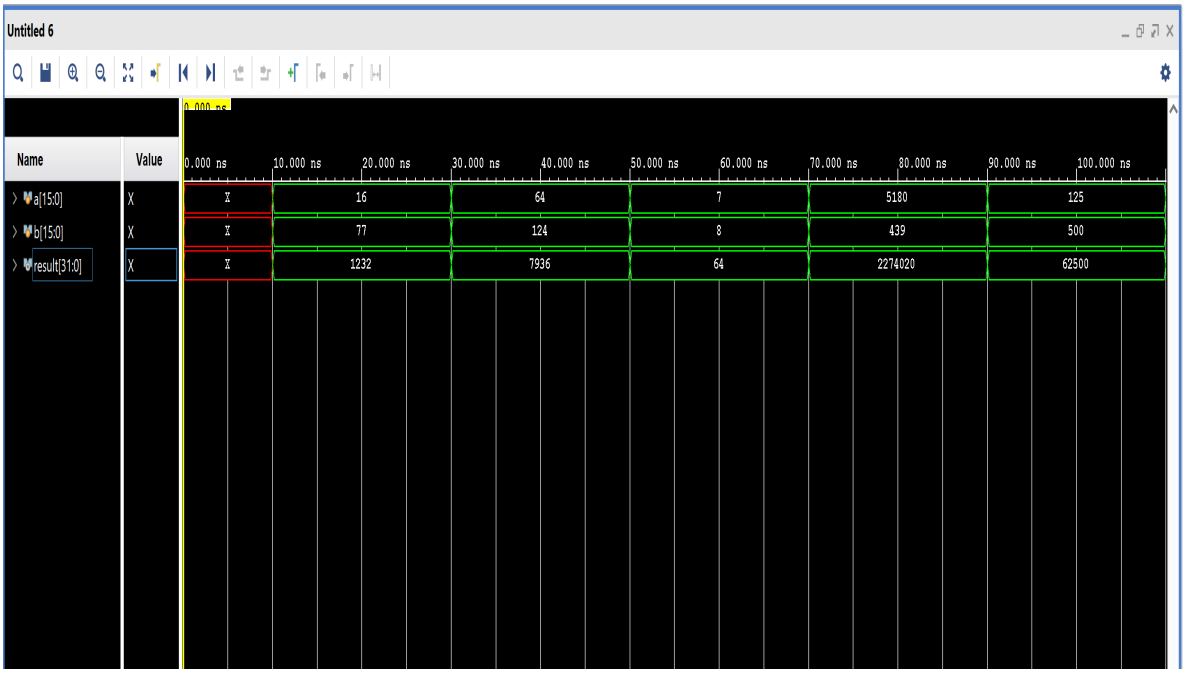

Figure 2: Output Waveform of 16-bit Dadda Multiplier

4 OUTPUT WAVEFORM OF POST IMPLEMENTATION TIMING SIMULATION

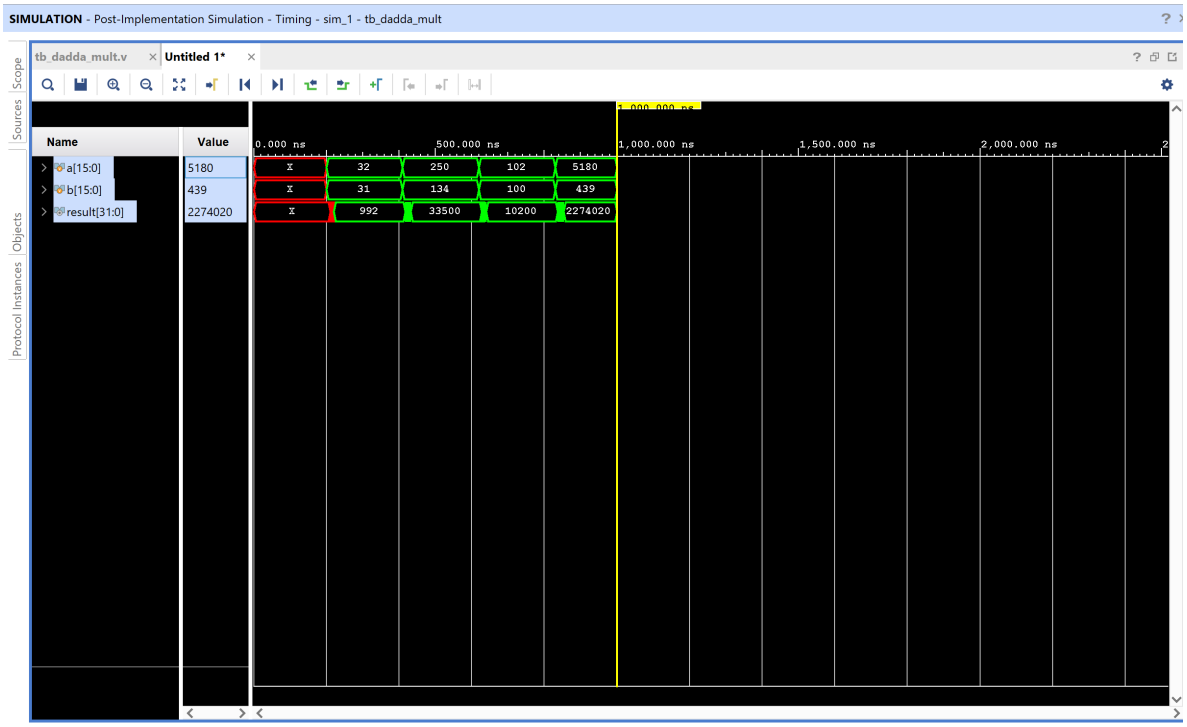

Figure 3: Post Implementation Timing Simulation

5 MULTIPLIER USING THE '*' OPERATOR

Multiplier using the asterisk operator is just assigning a result to the product of a[ ] and b[ ]. The same testbench code is used here also with some small changes while instantiating this module. The following code illustrates it,

```
module multiplier(
    input [15:0]a,
    input [15:0]b,
    output[31:0]result
);

   assign result = a*b;

endmodule
```

6 OUTPUT WAVEFORM

The following image shows the output waveform obtained after using the required test-bench for the module "multiplier".

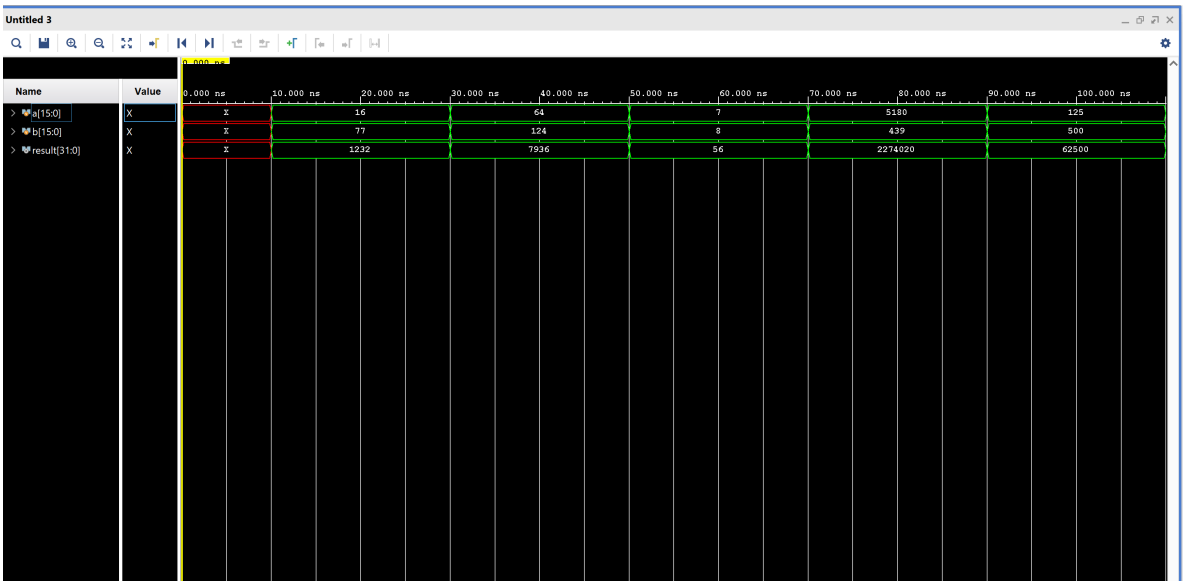

Figure 4: Output Waveform of 32 bit Multiplier using '*' Operator

7 RESOURCE UTILIZATION COMPARISON

The resource utilization of both the multipliers are shown below, the image on the top is utilization summary of multiplier with '*' Operator and the other one is Dadda Multiplier. From this, it is understood that the tool replaces the multiplication operator by some DSP blocks whereas the Dadda Multiplier is implemented with optimum number of LUT Resources. Since LUT resourcess are limited the tool cleverly uses them.

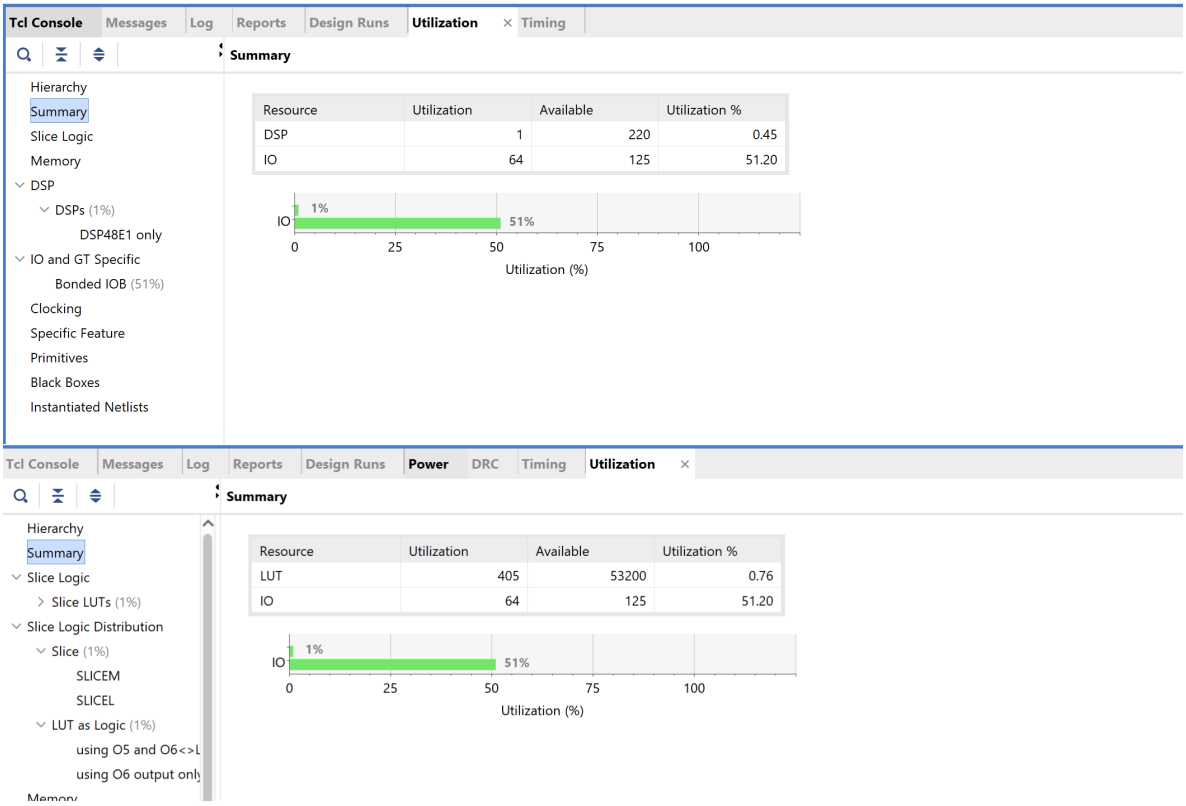

Figure 5: Utilization comparison between two Multipliers

8 TASK 2 - BLOCK DESIGN FOR THE MULTIPLIER

Now I started designing a controller which takes in `clk` and `start_stop` signals and acts accordingly. The logic of the controller works as follows: `addra` is initialized to zero. It is updated when `start_stop` is high and the posedge of `clk` is met, incrementing the address by 1. The `douta[31:0]` is split into two parts, `a[31:16]` and `b[15:0]`. The BRAM is loaded with the given .coe file. and the design is also created. The following Verilog code implements the controller logic:

```
module controller (
    input wire clka,
    input wire start_stop,
    input wire [31:0] douta,
    output reg ena,
    output reg [2:0] addra = 3'b000,
    output reg [15:0] a,
    output reg [15:0] b,
    output reg wea = 1'b0
);
```

```
    always @(posedge clka & start_stop) begin
            addra <= addra + 1;
            ena = 1'b1;
    end
    always @(douta) begin
                a <= douta[31:16];
                b <= douta[15:0];
    end
endmodule
```

9 THE FINAL DESIGN

For implementing in FPGA by generating a bitstream, the Virtual Input Output(VIO), Integrated Logic Analyzer(ILA), Clocking Wizards are connected. Here in the controller.v the block memory and mulltiplier need not be called explicitly since it is already linked by the design.The following figure shows the design,

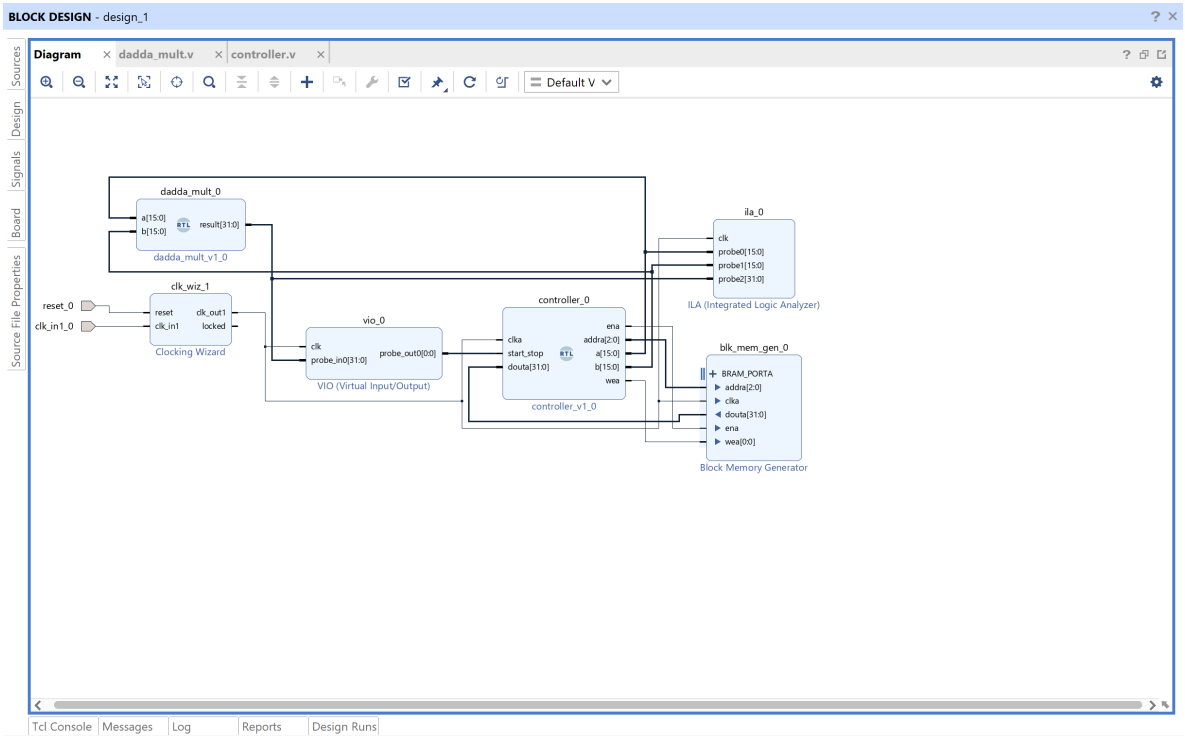

Figure 6: Design of VIO,ILA,BRAM and dadda multiplier connected with the controller

Then the constraint file is added with proper clk pin and Reset pin number on the board. Then, I generated a HDL wrapper code using the design.v and made it as top level entity. After that I synthesized and executed run implementation and the

bitstream is also generated successfully, The synthesized device image and the bitstream successful image are shown below,

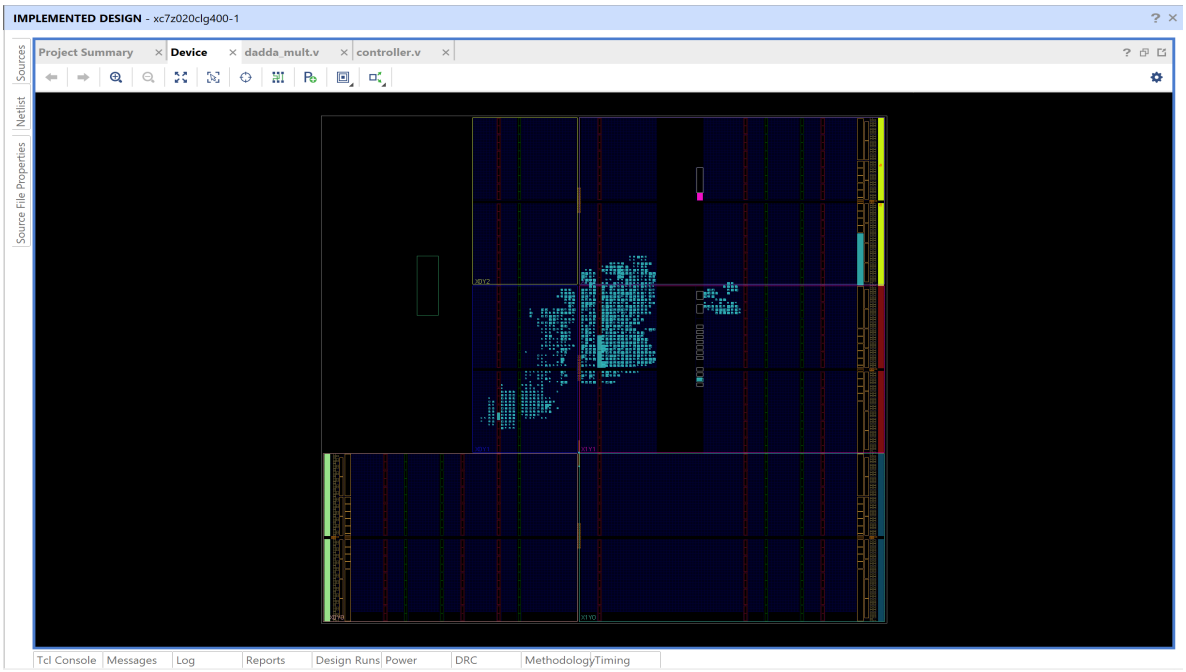

Figure 7: After the Design Synthesis

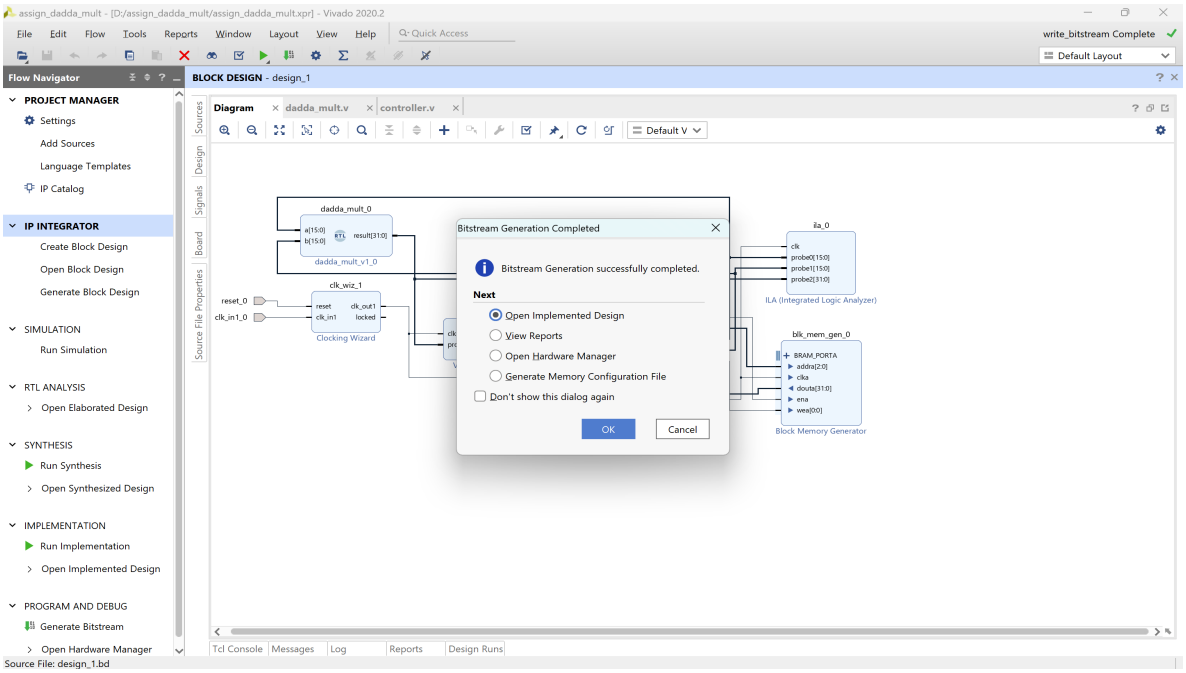

Figure 8: Bitstream Successful Dialog Box

10 TASK 1 - BARREL SHIFTER

To Design a 8 bit Barrel Shifter and write a testbench and simulate the circuit. The types of Multiplier to be designed:
(a) Using the Loigarithmic Barrel Shifter
(b) Using the "<<" and ">>" operator only.

11 LOGARITHMIC BARREL SHIFTER

I started designing the 8 bit Barrel Shifter from the right rotate structure given in the additional resources. The following architecture is first implemented by proper multiplexers,

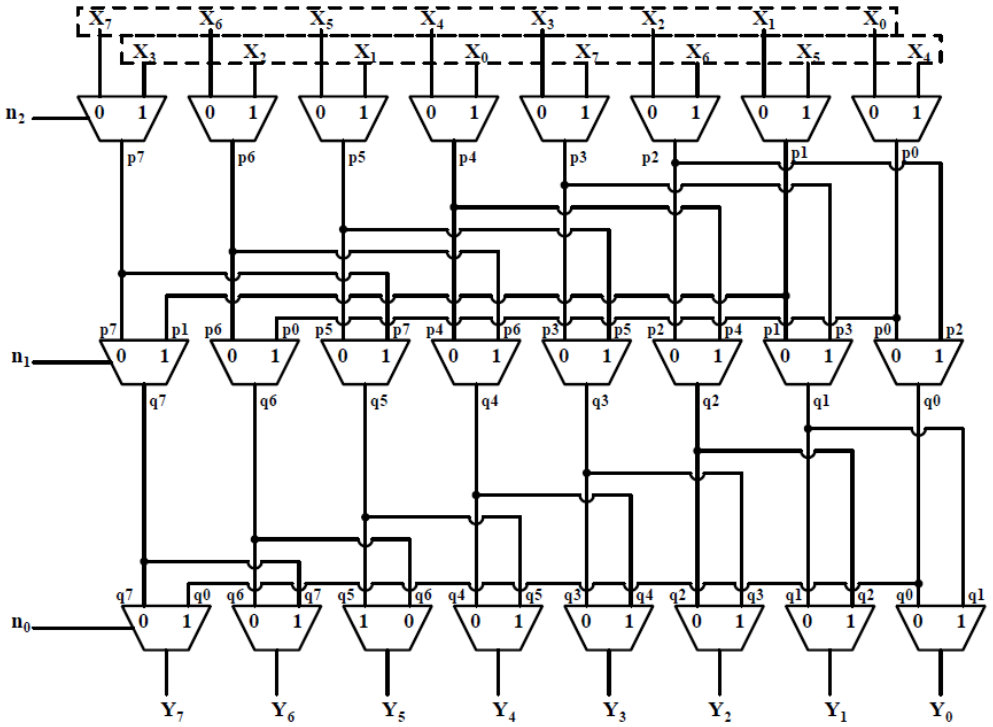

Figure 9: 8 bit Right Rotate Structure

Then I defined multiplexers as a separate module and called whenever it is needed,

```
module mux(
input a,
input b,
input sel,
output y
);
assign y = ((~sel)&a)|(sel&b);
endmodule
```

The Right Rotate Architecture is the primary circuit, after getting the rotated value the shift is determined. The same architecture can work for both left shift and right shift the difference is if it is a left shift then the right rotation should be done for 2's Complement of the shift amount given. The code for only the first stage of the architecture is shown below, since the other stages look similar. The register n[ ] contains the three bit shift amount value. After generating a right rotate structure, the number of bits needed to be shifted right are masked by zeros, for that a mask left and for a left shift mask right modules are used. The expressions are found implementing the function in a truth table.

```
    // Level 1
    mux mux_p0 (.a(x[0]), .b(x[4]), .sel(n[2]), .y(p[0]));
    mux mux_p1 (.a(x[1]), .b(x[5]), .sel(n[2]), .y(p[1]));
    mux mux_p2 (.a(x[2]), .b(x[6]), .sel(n[2]), .y(p[2]));
    mux mux_p3 (.a(x[3]), .b(x[7]), .sel(n[2]), .y(p[3]));
    mux mux_p4 (.a(x[4]), .b(x[0]), .sel(n[2]), .y(p[4]));
    mux mux_p5 (.a(x[5]), .b(x[1]), .sel(n[2]), .y(p[5]));
    mux mux_p6 (.a(x[6]), .b(x[2]), .sel(n[2]), .y(p[6]));
    mux mux_p7 (.a(x[7]), .b(x[3]), .sel(n[2]), .y(p[7]));
```

```
module maskleft(
input [2:0] shiftamt,
input [7:0] y,
output [7:0] z
);
wire [7:0]t;
wire [7:0]n;
assign n = shiftamt;
assign t[0] = 1'b1;
assign t[1] = ~(n[0] & n[1] & n[2]);
assign t[2] = ~(n[2] &  n[1]);
assign t[3] = ~(n[2] & (n[0] | n[1]));
assign t[4] = ~(n[2]);
assign t[5] = ~(n[2] | (n[1] & n[0]));
assign t[6] = ~(n[2] | n[1]);
assign t[7] = ~(n[2] | n[1] | n[3]);
assign z = y & t;
endmodule

module maskright(
input [2:0] shiftamt,
```

```
input [7:0] y,
output [7:0] z
);
wire [7:0]t;
wire [7:0]n;
assign n = shiftamt;
assign t[0] = ~(n[2] | n[1] | n[3]);
assign t[1] = ~(n[2] | n[1]);
assign t[2] = ~(n[2] | (n[1] & n[0]));
assign t[3] = ~(n[2]);
assign t[4] = ~(n[2] & (n[0] | n[1]));
assign t[5] = ~(n[2] &  n[1]);
assign t[6] = ~(n[0] & n[1] & n[2]);
assign t[7] = 1'b1;
assign z = y & t;
endmodule
```

After the design file is created, the testbench file is created with the test cases mentioned in the .coe file given. The following is the testbench code,

```
`timescale 1ns / 1ps
module barrel_shifter_tb;
    reg [7:0] x;
    reg [2:0] shiftamt;
    reg right;
    wire [7:0] z;
    barrel_shifter uut (
        .x(x),
        .shiftamt(shiftamt),
        .right(right),
        .z(z)
    );
    initial begin
        #10 {x, shiftamt} = 11'b00100101010; right = 1;
        #10 {x, shiftamt} = 11'b00111101011; right = 0;
        #10 {x, shiftamt} = 11'b00010001001; right = 1;
        #10 {x, shiftamt} = 11'b01000000010; right = 0;
        #10 {x, shiftamt} = 11'b00110011011; right = 1;
        #10 {x, shiftamt} = 11'b00100101100; right = 0;
        #10 {x, shiftamt} = 11'b00111101101; right = 1;
        #10 {x, shiftamt} = 11'b00110101010; right = 0;
        #10 $stop;
    end
endmodule
```

12 OUTPUT WAVEFORM OF BEHAVIOURAL SIMULATION

The following image shows the output waveform obtained after using the required test-bench for the module "barrel_shifter".

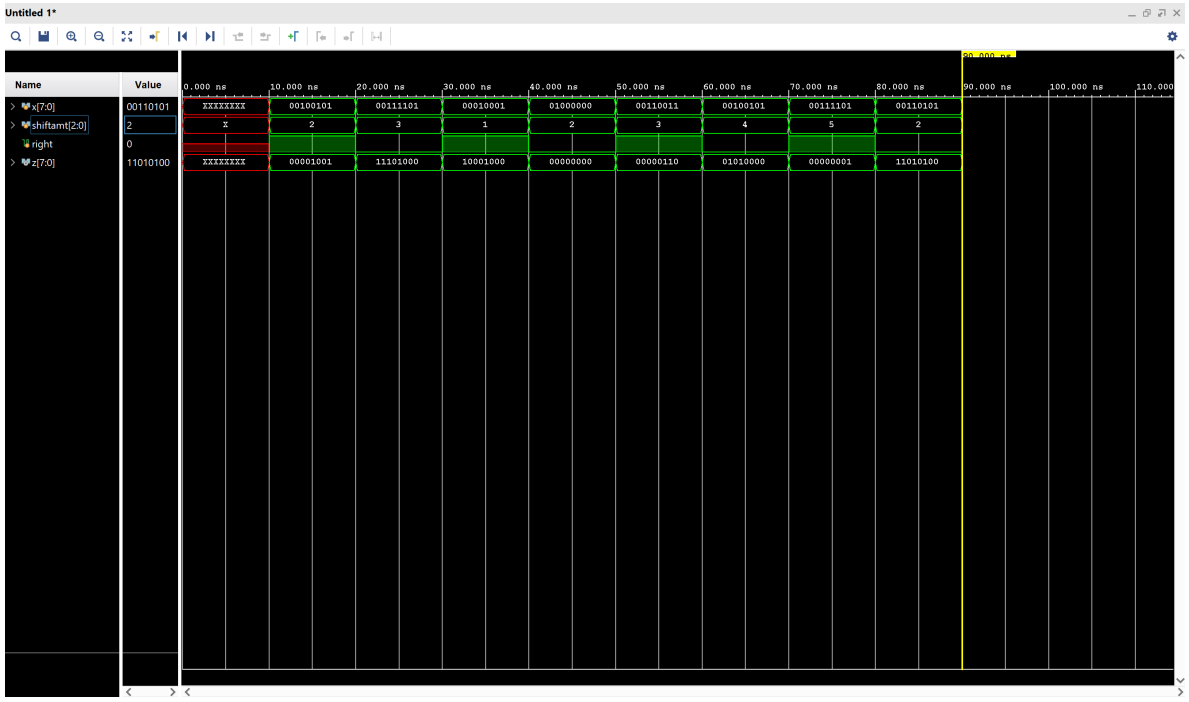

Figure 10: Output Waveform of 8 bit Barrel Shifter

13 OUTPUT WAVEFORM OF POST IMPLEMENTATION TIMING SIMULATION

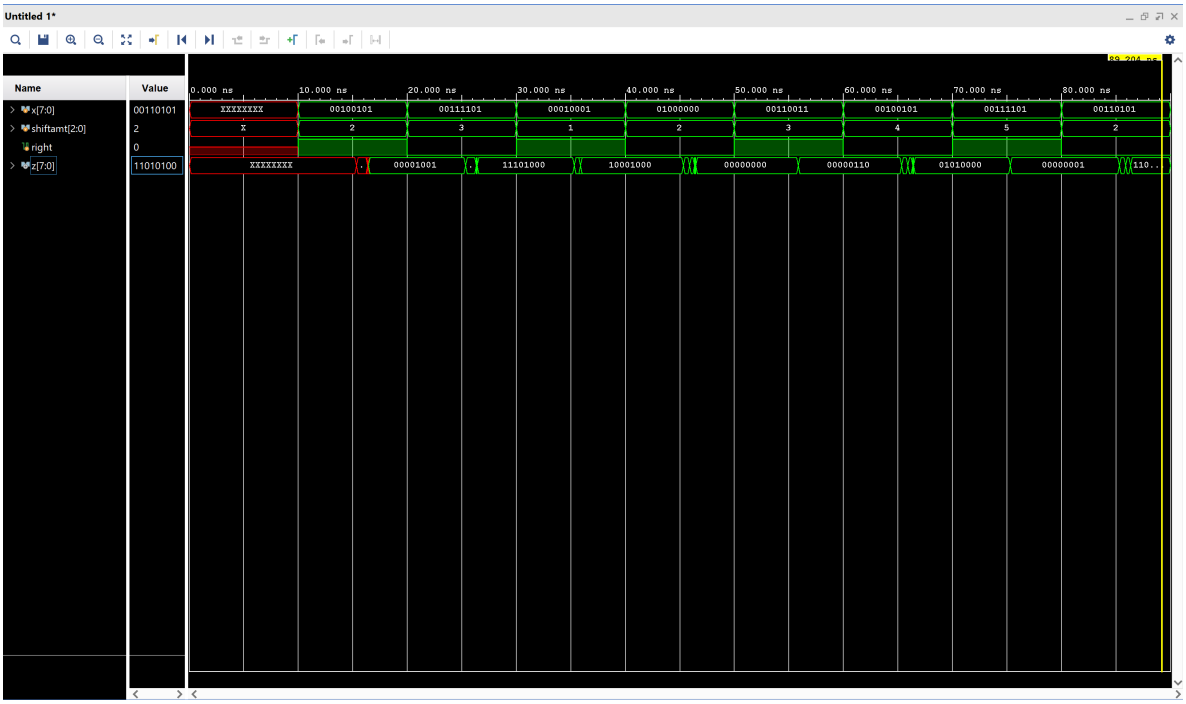

Figure 11: Post Implementation Timing Simulation

14 SHIFTER USING THE "<<" AND ">>" OPERATOR

Multiplier using the "<<" and ">>" operator is just assigning the variable z[ ] to the shifted value of x [ ]. The same testbench code is used here also with some small changes while instantiating this module. The following code illustrates it,

```
module shifter(
    input wire [7:0] x,
    input wire [2:0] shiftamt,
    input wire right,
    output reg [7:0] z
);
    always @(*) begin
        if (right)
            z = x >> shiftamt; // Right shift
        else
            z = x << shiftamt; // Left shift
    end
endmodule
```

15 OUTPUT WAVEFORM

The following image shows the output waveform obtained after using the required test-bench for the module ”shifter”.

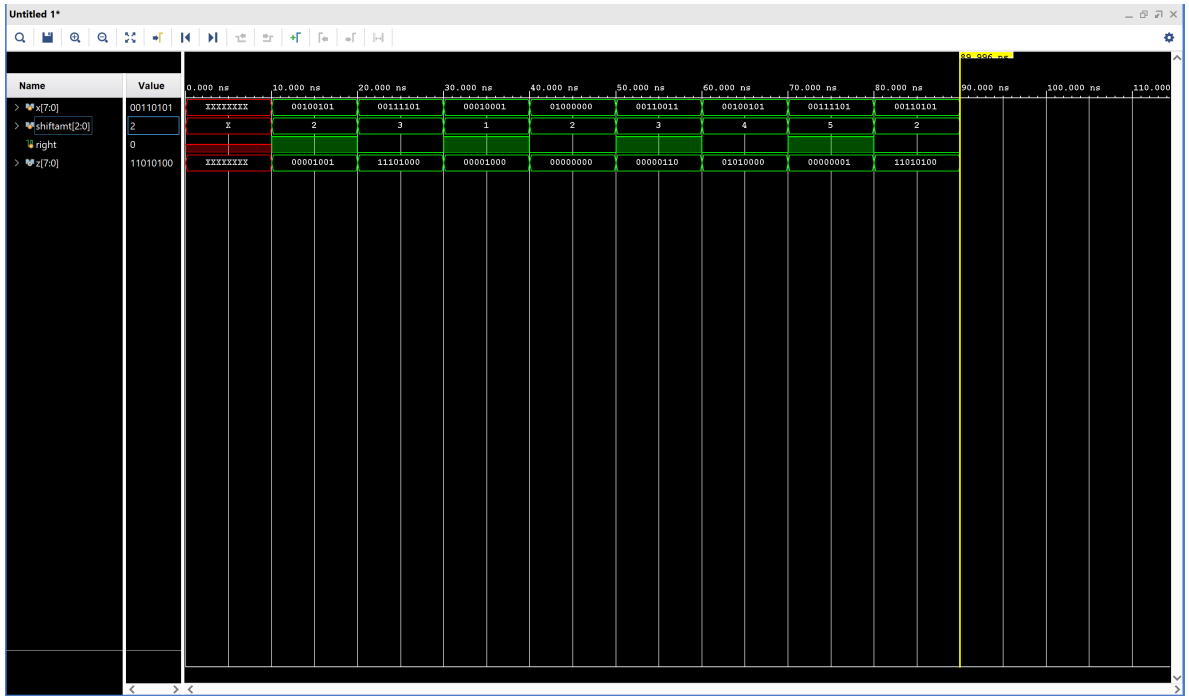

Figure 12: Output Waveform of 8 bit Shifter using ">>" and "<<" Operator

16 RESOURCE UTILIZATION COMPARISON

The resource utilization of both the shifters are shown below, the image on the top is utilization summary of shifter with "<<" and ">>" Operator and the other one is Logarithmic Barrel Shifter. From this, it is understood that the tool replaces these operators with more number of LUTs (during synthesis) when compared to the logarithmic shifters designed.

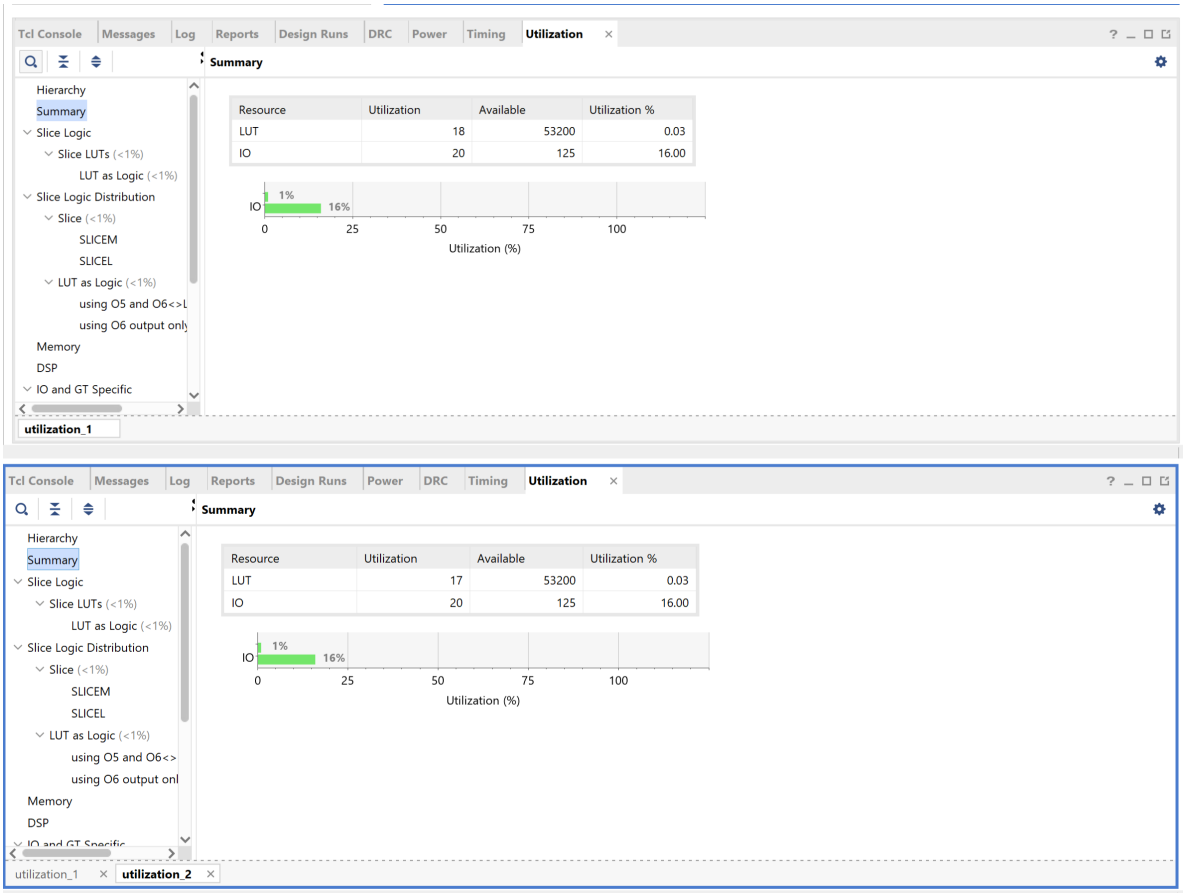

Figure 13: Utilization comparison between two Shifters

17 TASK 2 - BLOCK DESIGN FOR THE BARREL SHIFTER

Now I started designing a controller which takes in `clk` and `start_stop` and `dir` signals and acts accordingly. The logic of the controller works as follows: `addra` is initialized to zero. It is updated when `start_stop` is high and the posedge of `clk` is met, incrementing the address by 1. The value of output reg right is determined by the value of dir, if dir = 0 , it means left shift and passes right = 0 to the barrel shifter module and if dir = 1, then right = 1 is passed to the barrel shifter module. The `douta[10:0]` is split into two parts, `x[10:3]` and `shiftamt[2:0]`. The BRAM is loaded with the given .coe file. and the design is also created. The following Verilog code implements the controller logic:

```
module controller (
    input wire clka,
    input wire start_stop,
    input wire dir,
    input wire [10:0] douta,
    output reg ena,
    output reg right,
    output reg [2:0] addra = 3'b000,
    output reg [2:0] shiftamt,
    output reg [7:0]x,
    output reg wea = 1'b0
);
    always @(posedge clka & start_stop) begin
            addra <= addra + 1;
            ena = 1'b1;
    end
    always @(douta) begin
            if (dir == 1'b0) begin
                x <= douta[10:3];
                shiftamt <= douta[2:0];
                right <= 1'b0;
            end else begin
                x <= douta[10:3];
                shiftamt <= douta[2:0];
                right <= 1'b1;
            end
    end
endmodule
```

18 THE FINAL DESIGN

For implementing in FPGA by generating a bitstream, the Virtual Input Output(VIO), Integrated Logic Analyzer(ILA) , Clocking Wizards are connected. Here in the controller.v the block memory and barrel shifter module need not be called explicitly since it is already linked by the design.The following figure shows the design,

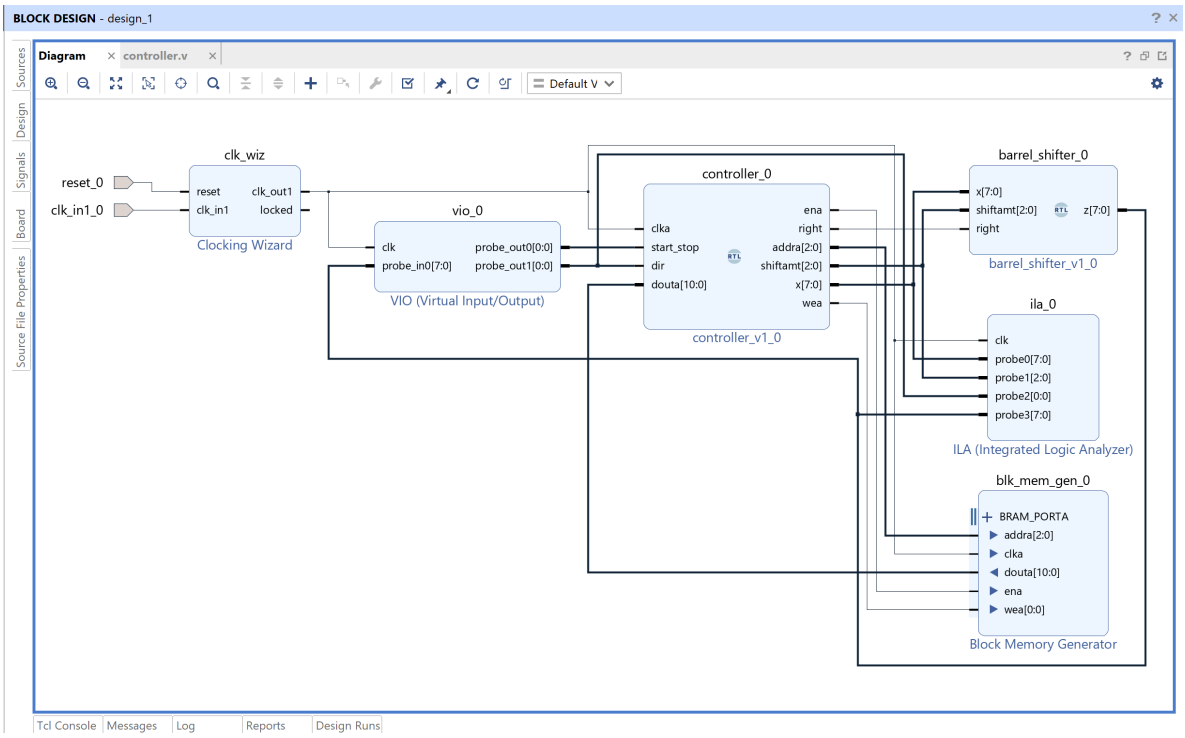

Figure 14: Design of VIO,ILA,BRAM and Barrel Shiter connected with the controller

Then the constraint file is added with proper clk pin and Reset pin number on the board. Then, I generated a HDL wrapper code using the design.v and made it as top level entity. After that I synthesized and executed run implementation and the bitstream is also generated successfully, The synthesized device image and the bitstream successful image are shown below,

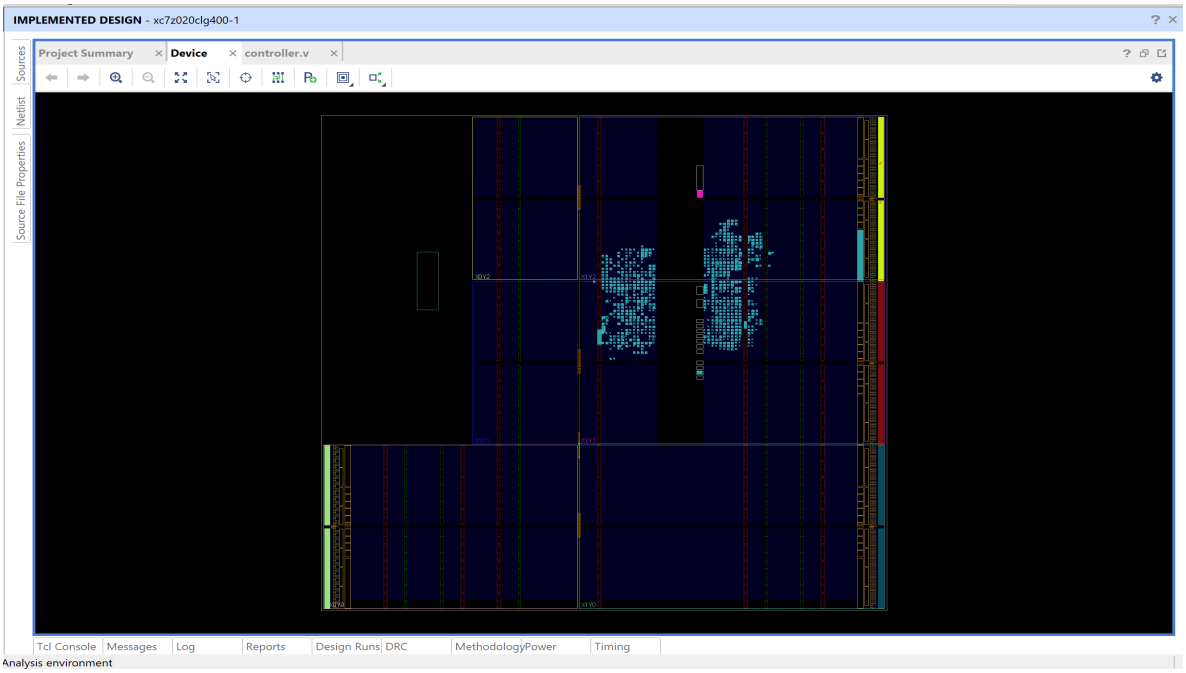

Figure 15: After the Design Synthesis

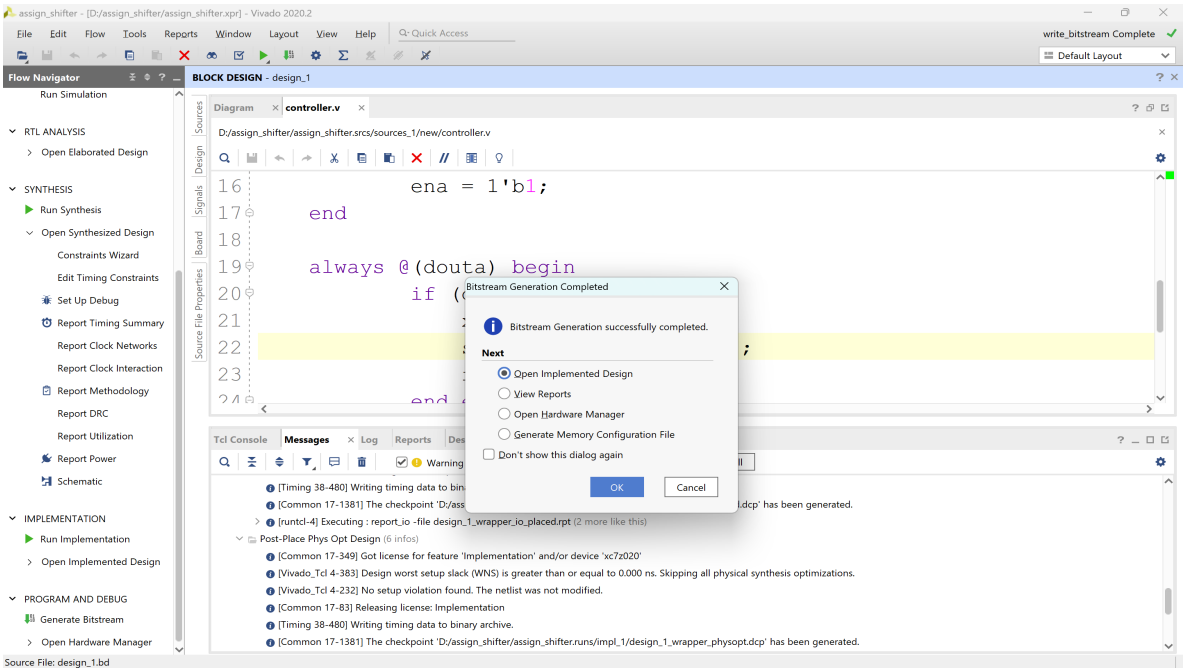

Figure 16: Bitstream Successful Dialog Box