Indian Institute of Technology Bombay

EE 705 VLSI Design Lab

Assignment I

Implementation of 32 Bit Brent Kung Adder in FPGA

Submitted By:

Bala Murugan S

Reg No: 24M1173

Course Instructor:

Prof. Laxmeesha Somappa

Contents

List of Figures

1 TASK 1

To Design a 32-bit Adder/Subtractor and write a testbench and simulate the circuit.
The types of Adder/Subtractor to be designed:
(a) Using the Brent-Kung adder
(b) Using the '+' operator only.

2 BRENT KUNG ADDER

I started designing the 32 Bit Brent-Kung Adder from the following diagram. This diagram illustrates the structure of the adder, showcasing the carry-generate and propagate logic along with the reduction tree.

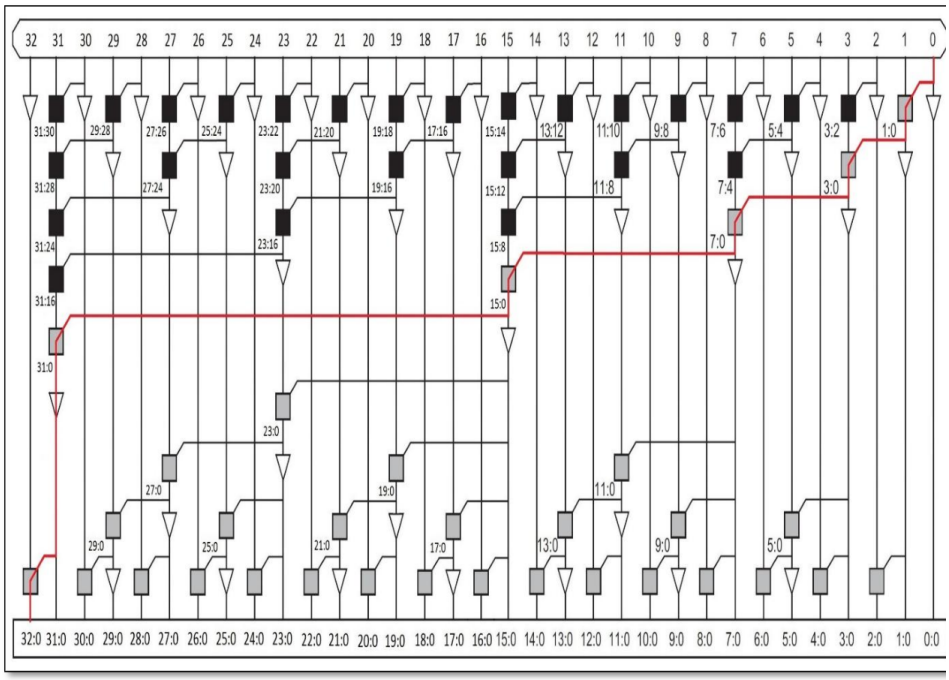

Figure 1: Tree diagram of the 32-bit Brent-Kung Adder

The gray and black cells represent the group generate and group propagate signals, respectively. The triangular elements in the diagram are buffers. Initially, the P and G signals for each input are generated using the formulas:

$$P = a_i \oplus b_i, \quad G = a_i \& b_i$$

The general formula to produce group generate signals at all stages is:

$$G_{2i+1,2i} = G_{2i+1} \mid (P_{2i+1} \& G_{2i})$$

In the first stage, G_0, P_0, G_1, P_1, and so on are computed. After one stage is complete, group generate signals from the previous stage are combined (e.g., $G_{1:0}$ and $G_{3:2}$ in stage 1 are combined to compute $G_{3:0}$), and so on. This continues through five stages, at the end of which $G_{31:0}$ and $P_{31:0}$ are generated. This reduction process allows us to generate the carry signals in $\log_2(N)$ stages.
The odd carry signals are generated using the formula:

$$C[2i+1] = G_{2i+1:0} \,|\, (P_{2i+1:0} \& C_{\text{in}})$$

Even carry signals are generated using the previously computed odd carry signals:

$$C[2i] = G[2i] \,|\, (P[2i] \& C[2i-1])$$

Finally, the sum bits are computed as:

$$\text{Sum}[i] = P[i] \oplus C[i-1]$$

Based on this logic, the Verilog code is written, and it is attached below.

```
module bkadder(
    input [31:0] a,
    input [31:0] b,
    input Cin,
    output [31:0] sum,
    output cout
    );
    wire cin;
    wire [31:0] g,p;
    wire [15:0] g1,p1;
    wire [7:0] g2,p2;
    wire [3:0] g3,p3;
    wire [1:0] g4,p4;
    wire g5,p5;
    wire [31:0] c;
    genvar i;
    assign g = a & b;
    assign p = a ^ b;
    assign cin = Cin;

    //first stage
    generate
    for (i=0;i<16;i=i+1)begin : stage1
       assign g1[i] = g[2*i+1] | (p[2*i+1]&g[2*i]);
       assign p1[i] = (p[2*i]&p[2*i+1]);
    end
    endgenerate

    //second stage
    generate
```

```
for (i=0;i<8;i=i+1)begin : stage2
   assign g2[i] = g1[2*i+1] | (p1[2*i+1]&g1[2*i]);
   assign p2[i] = (p1[2*i]&p1[2*i+1]);
end
endgenerate

//third stage
 generate
for (i=0;i<4;i=i+1)begin: stage3
   assign g3[i] = g2[2*i+1] | (p2[2*i+1]&g2[2*i]);
   assign p3[i] = (p2[2*i]&p2[2*i+1]);
end
endgenerate

//fourth stage
generate
for (i=0;i<2;i=i+1)begin: stage4
   assign g4[i] = g3[2*i+1] | (p3[2*i+1]&g3[2*i]);
   assign p4[i] = (p3[2*i]&p3[2*i+1]);
end
endgenerate

//fifth stage
assign g5 = g4[1]|(g4[0]&p4[1]);
assign p5 = (p4[1]&p4[0]);


//some intermediate g's and p's
wire g11down8 = (g1[5] | (p1[5] & g1[4]));
wire p11down8 = (p1[5] & p1[4]);
wire p11down0 = p1[5] & p1[4] & p3[0];
wire g13down8 = g1[6] | (p1[6] & g2[2]);
wire p13down8 = p1[6] & p2[2];
wire g17down0 = g1[8] | (p1[8] & g4[0]);
wire p17down0 = p1[8] & p4[0];
wire g21down16 = (g1[10] | p1[10] & g2[4]);
wire p21down16 = (p1[10] & p2[4]);
wire g25down16 = g1[12] | (p1[12] & g3[2]);
wire p25down16 = p1[12] & p3[2];
wire g25down0 = g25down16 | (p25down16 & g4[0]);
wire p25down0 = p25down16 & p4[0];
wire g27down16 = g2[6] | (p2[6] & g3[2]);
wire p27down16 = p2[6] & g3[2];
wire g27down0 = g27down16 | (p27down16 & g4[0]);
wire p27down0 = p27down16 & p4[0];
wire g29down24 = g1[14] | (p1[14] & g2[6]);
wire p29down24 = p1[14] & p2[6];
wire g29down16 = g29down24 | (p29down24 & g3[2]);
wire p29down16 = p29down24 & p3[2];
wire g29down0 = g29down16 | (p29down16 & g4[0]);
wire p29down0 = p29down16 & p4[0];
```

```
    assign c[0] = g[0] | (p[0] & cin);


    //odd carry calculations
    assign c[1] = g1[0] | (p1[0] & cin);
    assign c[3] = g2[0] | (p2[0] & cin);
    assign c[5] = g1[2] | (p1[2] & g2[0]) | (p1[2] & p2[0] & cin);
    assign c[7] = g3[0] | (p3[0] & cin);
    assign c[9] = g1[4] | (p1[4] & g3[0]) | (p1[4] & p3[0]  & cin);
    assign c[11] = g11down8 | (p11down8 & g3[0]) | (p11down0 & cin);
    assign c[13] = (g13down8 | (p13down8 & g3[0])) | (p13down8 &
                            p3[0] & cin) ;
    assign c[15] =  g4[0] | (p4[0] & cin);
    assign c[17] = g17down0 | (p17down0 & cin);
    assign c[19] = g2[4] | (p2[4] & g4[0]) | (p2[4] & p4[0] & cin);
    assign c[21] =  (g21down16 | (p21down16 & g4[0])) | (p21down16 &
                             p4[0] & cin);
    assign c[23] = g3[2] | (p3[2] &  g4[0]) | (p3[2] & p4[0] & cin);
    assign c[25] =  g25down0 | (p25down0 & cin);
    assign c[27] = g27down0 | (p27down0 & cin);
    assign c[29] = g29down0 | (p29down0 & cin);
    assign c[31] = g5 | (p5 & cin);

    //even carries
    generate
    for(i=1; i<16; i = i+1)begin: evencarry
        assign c[2*i] = g[2*i] | (p[2*i] & c[2*i-1]);
    end
    endgenerate

    assign sum[0] = p[0] ^ cin;

    generate
    for (i=1; i<32; i = i+1) begin: sumgen
        assign sum[i] = p[i] ^ c[i-1];
    end
    endgenerate

    assign cout = c[31];

endmodule
```

The following is the testbench code,

```
`timescale 1ns / 1ps
module tb_bkadder;
    reg [31:0] a, b;
    reg Cin;
    wire [31:0] sum;
    wire cout;
```

```
    bkadder uut (
        .a(a),
        .b(b),
        .Cin(Cin),
        .sum(sum),
        .cout(cout)
    );

    initial begin

        $monitor("Time: %0t | a: %d | b: %d | Cin: %d | sum: %d |
                          cout: %b", $time, a, b, Cin, sum, cout);

        a = 10; b = 5; Cin = 0;
        #10;

        a = 46; b = 32; Cin = 0;
        #10;

        a = 3244; b = 1544; Cin = 1;
        #10;

        a = 32565; b = 325778; Cin = 0;
        #10;

        a = 3422399; b = 1911079; Cin = 0;
        #10;

        a = 1570200; b = 4219577495; Cin = 0;
        #10;

        $stop;
    end

endmodule
```

3 OUTPUT WAVEFORM

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

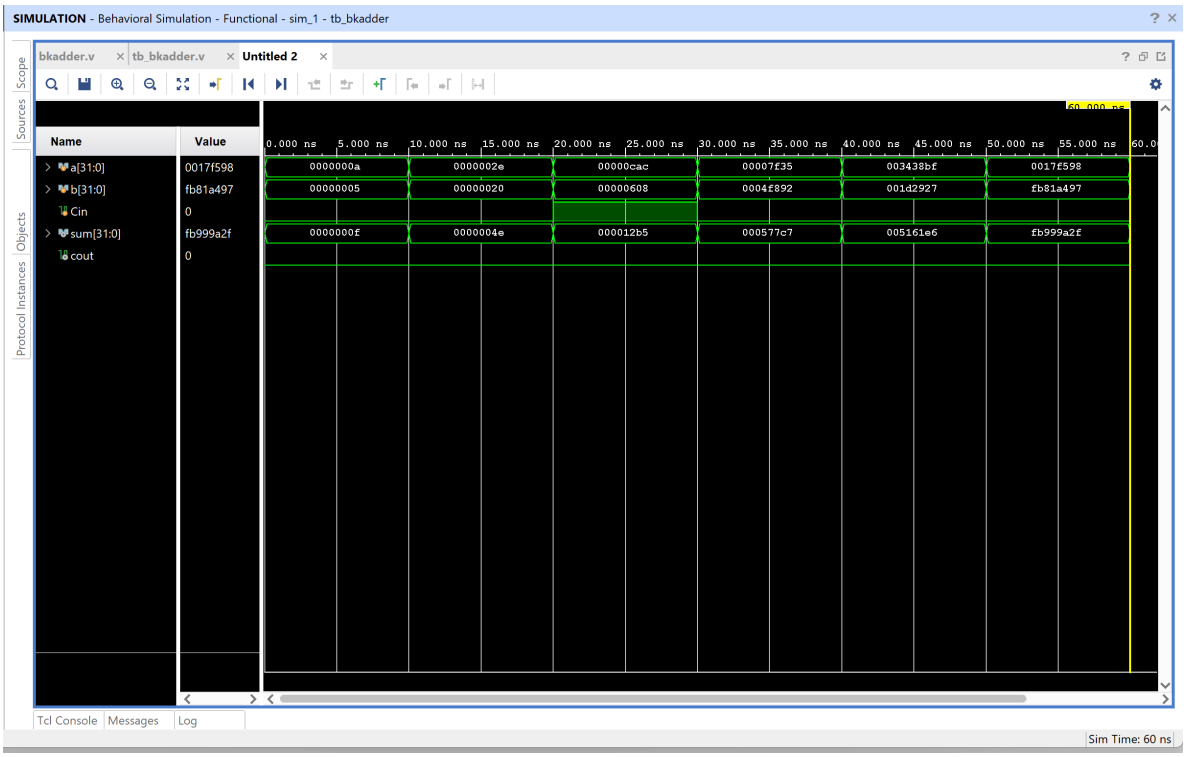

Figure 2: Output Waveform of 32 bit Brent-Kung Adder

4 ADDER USING THE '+' OPERATOR

Adder using the plus operator is just assigning a register to the sum of a,b and cin registers. The same testbench code is used here also, just the dut names are changes since the port names are also same. The following code illustrates it,

```
module adder(
    input [31:0] a,
    input [31:0] b,
    input Cin,
    output [31:0] sum,
    output cout
    );
    reg [32:0] s;

    always@(a or b) begin
         s = a + b + Cin;
    end
    assign sum = s[31:0];
    assign cout = s[32];
endmodule
```

5 OUTPUT WAVEFORM

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

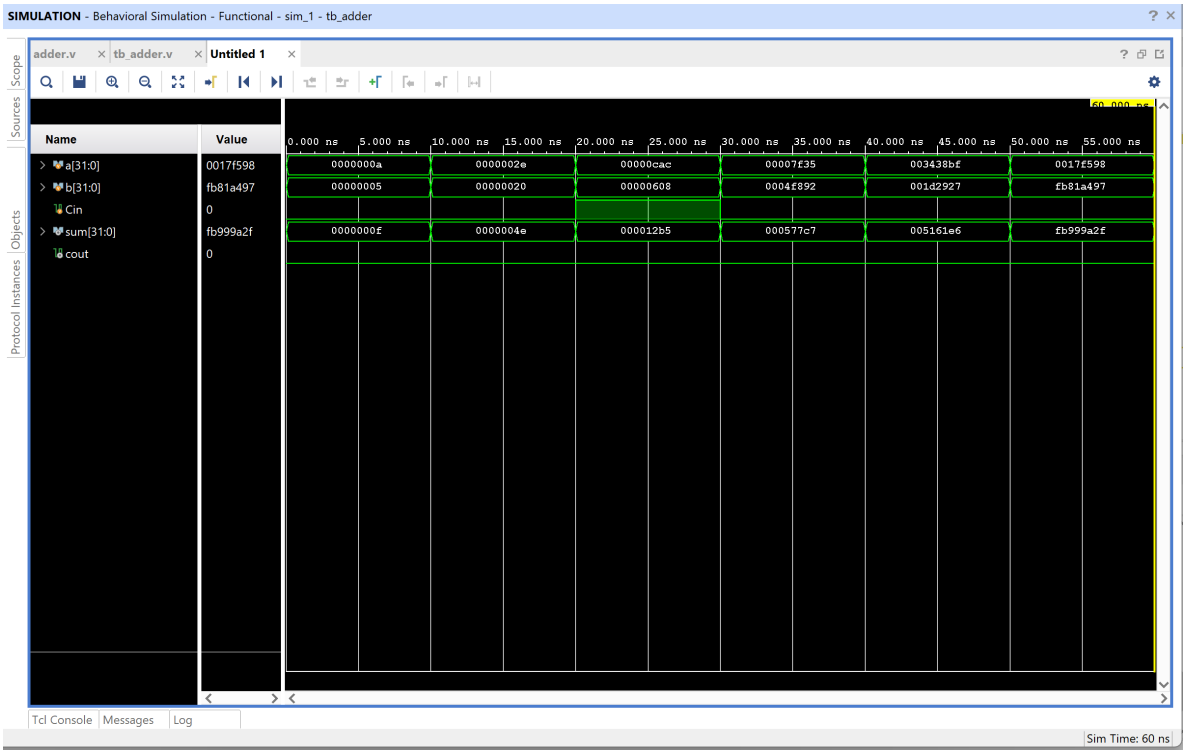

Figure 3: Output Waveform of 32 bit Adder using '+' Operator

6 RESOURCE UTILIZATION COMPARISON

The resource utilization of both the adders are shown below, the image on the top is utilization summary of adder with '+' Operator and the other one is Brent Kung Adder. From this, it is understood that the tool replaces the addition operator very efficiently compared to the Brent Kung Method, it is implemented with much minimum LUT Resources. Since LUT resourcess are limited the tool cleverly uses them.

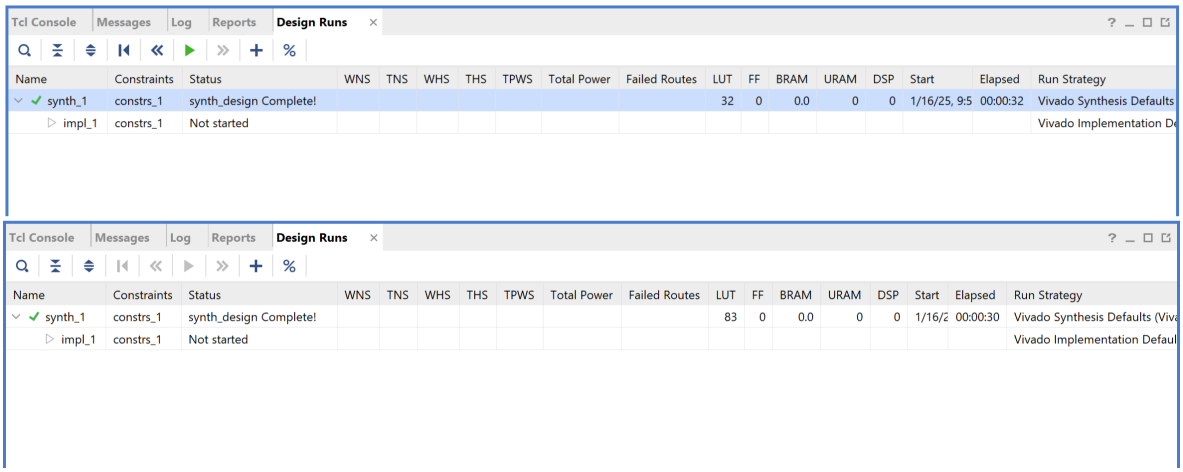

Figure 4: LUT comparison between two adders

7 TASK 2

Now I started designing a controller which takes in `clk`, `start_stop`, and `add_sub` signals and acts accordingly. To initiate and test a BRAM with a testbench, I used the IP Catalog option to create a Block RAM Memory Generator. The given test vectors are first individually (a and b) converted into hexadecimal equivalents and attached into a file called `ip.coe`, which was loaded into the Block RAM.
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. If `add_sub` is 0, the addition operation is performed. The `douta[63:0]` is split into two parts, `a[31:0]` and `b[32:63]`. The following Verilog code implements the controller logic:

```
module controller (
    input wire clka,
    input wire start_stop,
    input wire add_sub,
    input wire [63:0] douta,
    output reg [3:0] addra = 4'b0000,
    output reg [31:0] a,
    output reg [31:0] b,
    output reg Cin,
    output reg ena,
    output reg wea = 0
);

blk_mem_gen_0 mem (
  .clka(clka),    // input wire clka
  .ena(ena),      // input wire ena
  .wea(wea),      // input wire [0 : 0] wea
  .addra(addra),  // input wire [3 : 0] addra
```

```
  .dina(dina),    // input wire [63 : 0] dina
  .douta(douta)  // output wire [63 : 0] douta
);

    always @(posedge clka & start_stop) begin
            addra <= addra + 1;
            ena <= 1'b1;
    end

    always @(douta) begin
            if (add_sub == 1'b0) begin
                a <= douta[63:32];
                b <= douta[31:0];
                Cin <= 0;
            end else begin
                a <= douta[63:32];
                b <= ~douta[31:0];
                Cin <= 1;
            end
    end
bkadder add(
.a(a),
.b(b),
.Cin(Cin)
);
endmodule
```

The testbench just gives the signals `clk`, `start_stop`, and `add_sub`. When these signals are generated, the BRAM is accessed, data is fetched from it, and then the sum is computed using the `bkadder` module.

```
`timescale 1ns/1ps

module tb_controller;
reg clka = 1'b0;
reg start_stop = 1'b1;
reg add_sub = 1'b0;
wire [3:0] addra;
wire [31:0] a, b;
wire Cin;

controller dut (
    .clka(clka),
    .start_stop(start_stop),
    .add_sub(add_sub),
    .addra(addra),
    .a(a),
    .b(b),
    .Cin(Cin)
);

always #5 clka = ~clka;
```

```
initial begin
    #10 start_stop = 1;
    add_sub = 0;
    #80;
    start_stop = 0;
    #20;
    $stop;
end
endmodule
```

8 OUTPUT WAVEFORM OF ACCESSING BRAM AND SUM COMPUTATION

The following waveform shows the values of a and b displayed as in the ip.coe file sequentially and their sum and output carry in hexadecimal formats.

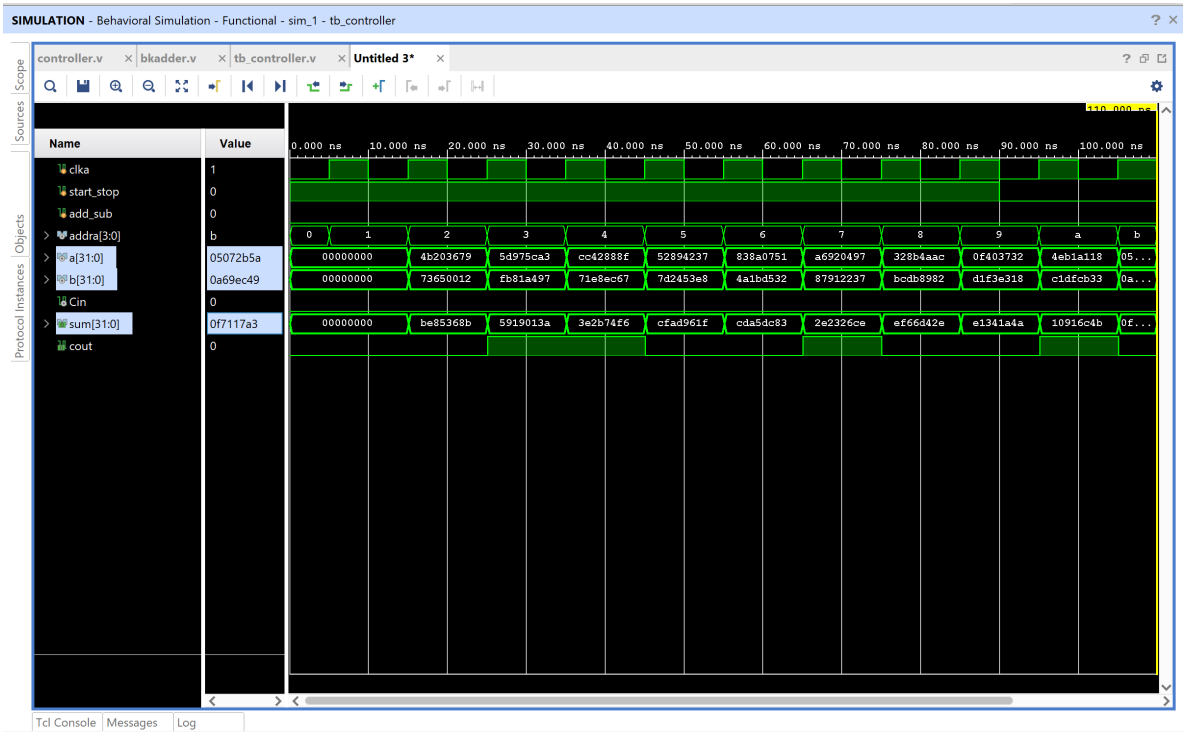

Figure 5: Values of a and b and its sum

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 adder need not be called since it is alredy linked by the design. So the same controller.v without those module calls is used. The following figure shows the design,

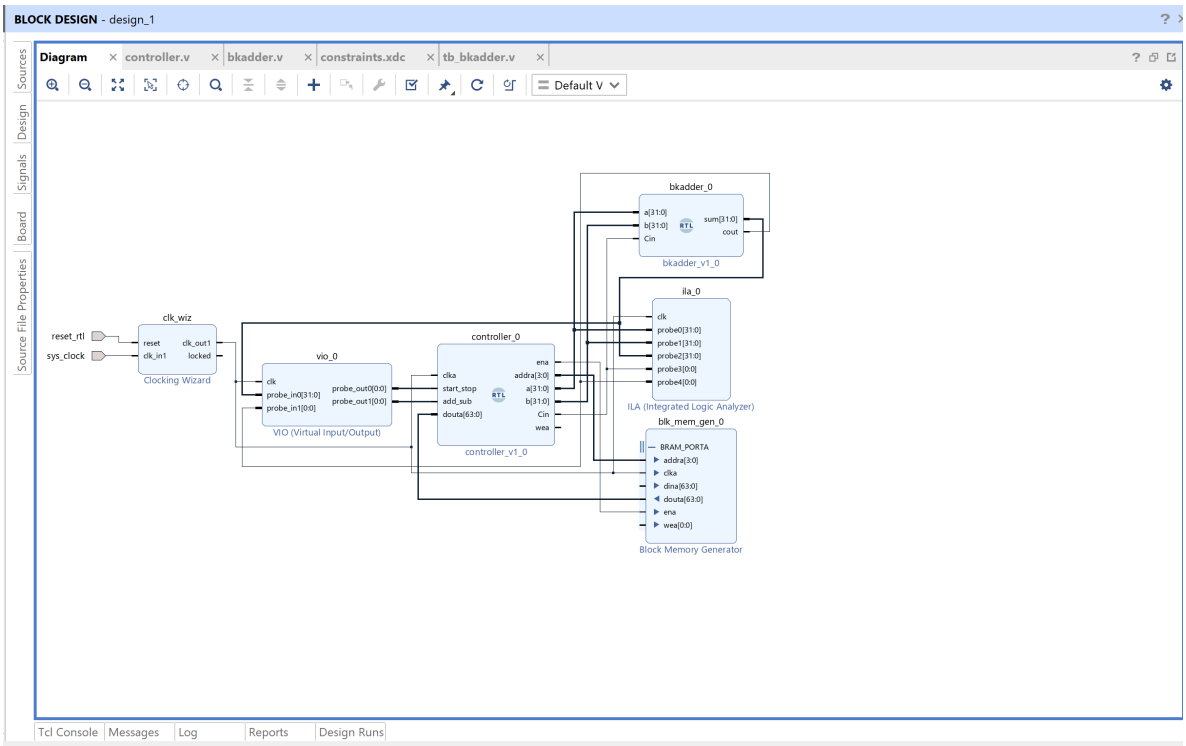

Figure 6: Design of VIO,ILA,BRAM and bkadder connected with the controller

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,

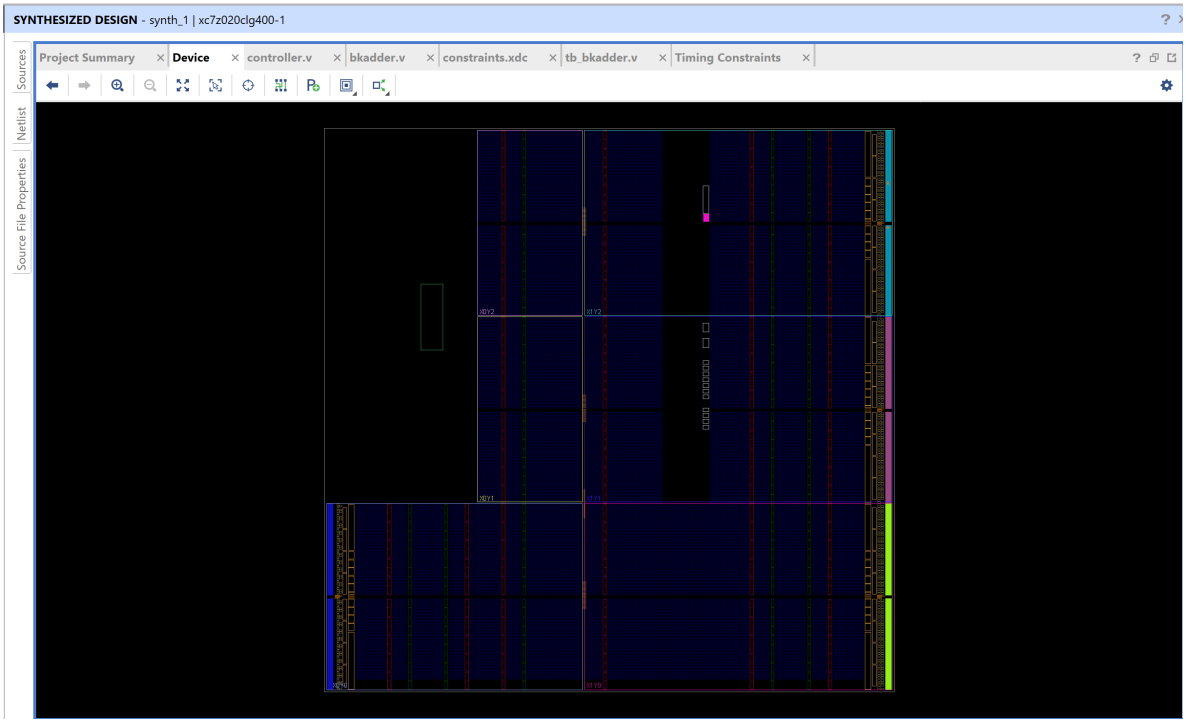

Figure 7: After the Design Synthesis

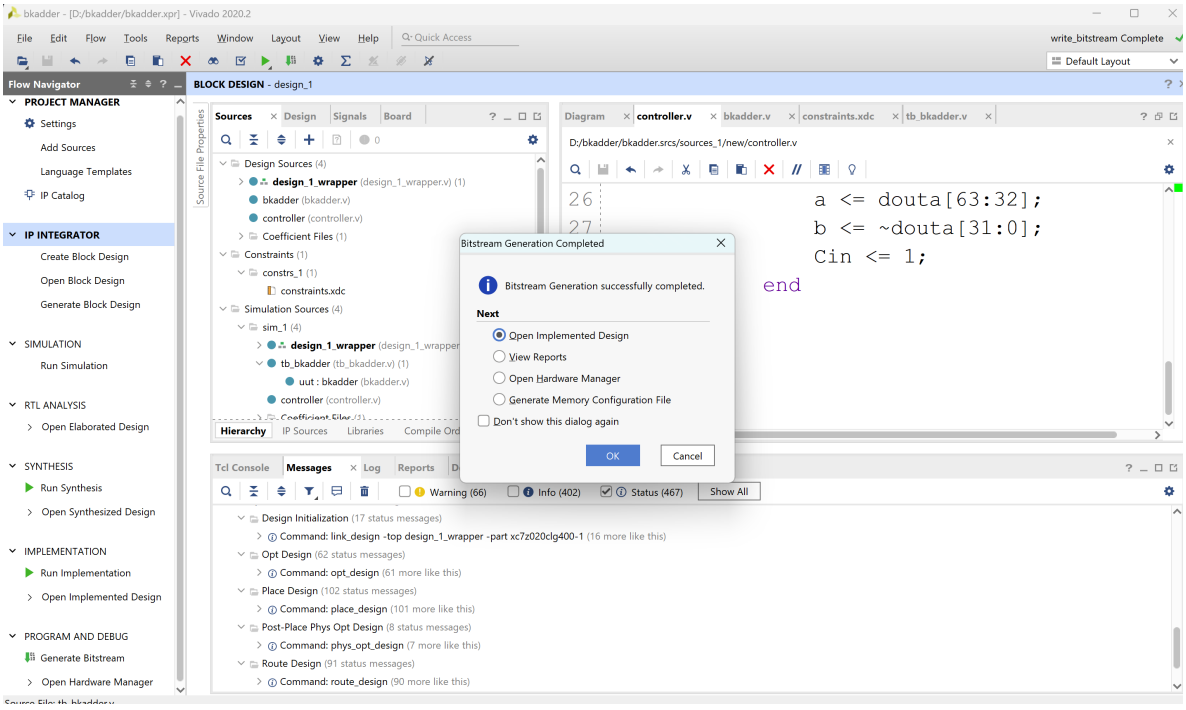

Figure 8: Bitstream Successful Dialog Box