CSEE 4280: Advanced Digital Design

Trent Jones

Abhi Boggavarapu

Assignment: Exercise 2

01 / 24 / 2025

Table of Contents

Roles and Responsibilities

Abhi	Software
Trent	Documentation

Github Code Link

The below is a link to the repository and folder within that includes the code used for this project.

https://github.com/abhibogga/CSEE4280Workspace/tree/main/2_Excercise

Build a Carry Look Ahead Adder

a.

Below we have the results for the 4 bit look ahead adder. The approach to development was to make a general 4 bit adder, model shown in figure 1.1. In this adder module, it contains 3 inputs. 2 of those inputs are 2, 4 bit arrays, called a and b. These will be the 2 variables in which you are adding. The last input is cIn, this is the bit coming into the first adder to control the first bit addition. There are 2 outputs to this module. The first is sum which is a 4 digit array. And the next is cOut, this is in the case of if the addition between inputs a and b cause an overflow error, the cOut output is there for that reason. If this project were to be even more optimized, then it would require the use of building another module such as the full adder, and then to create the 4 bit adder, would require to merge 4 adder modules together. Since we were asked to develop a simpler model, we used an approach shown in figure 2. To simulate this project we used a simple testbench and tested it inside GTKWave. We chose random test cases to test as well as the ones asked below. Our test bench worked on a clock basis in which each test case was evaluated on each change of the clock. Waveform shown in figure 1. The testbench code is shown in figure 3 while the following test cases are shown in figure 4.

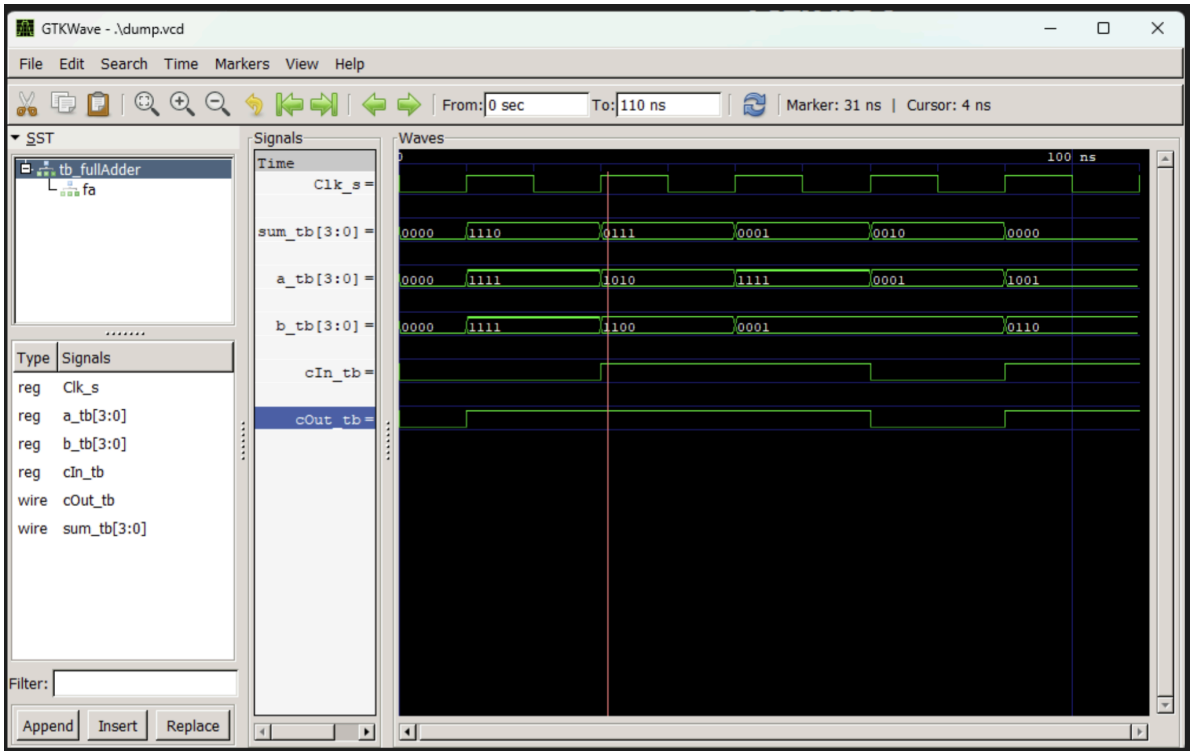

Figure 1: Waveform Generation of 4-bit Adder

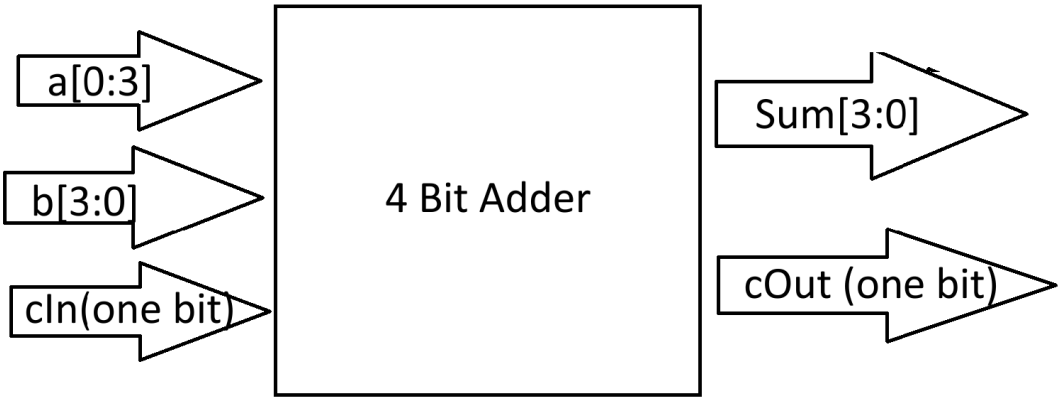

Figure 1.1: Model of 4 bit adder

```
1   module fullAdder(a, b, cIn, sum, cOut);
2
3   //Define Inputs
4   input [3:0] a, b;
5   input cIn;
6
7   //Define Outputs
8   output reg [3:0] sum; //We need this to be a reg since it is begin changed withi
9   output reg cOut;
10
11  //Define wires or midway points
12  reg [4:0] tempOutput; //We need this to be a reg since it is begin changed withi
13
14
15  always @(*) begin //The always @ (x) is for setting up a conditional, the * indi
16      //Here we want to describe what we need to happen for the full adder to work
17      tempOutput = a + b;
18      tempOutput = tempOutput + cIn;
19
20      sum = tempOutput[3:0];
21      cOut = tempOutput[4];
22  end
23
24  endmodule
```

Figure 2: Look Ahead Adder code

```
`timescale 1ns/1ns
`include "fullAdder.v"
module tb_fullAdder(); //Keep in mind that most tb will not have any sort of parameters

    //Define inputs
    reg [3:0] a_tb,b_tb; //There are no inputs and outputs in a tb file since all of that is determined by the
    reg cIn_tb;

    //Define outputs
    wire [3:0] sum_tb; //We must use wires here since they are the outputs of a module
    wire cOut_tb; //wire and reg cannot be used in the same statement, they are just meant for different purpos

    //define the clock statement
    reg Clk_s;

    //Initialize the module
    fullAdder fa (
        .a(a_tb),
        .b(b_tb),
        .cIn(cIn_tb),
        .sum(sum_tb),
        .cOut(cOut_tb)
    );
```

FIgure 3: Look Ahead Adder testbench code

```
    );
    //Clock, we want to be able to cycle out the test cases
    always begin
        Clk_s <= 0;
        #10;
        Clk_s <= 1;
        #10;
    end

    initial begin //When we are making a testbench we want to make our test cases with an intial begin, keep in mind that intial begin cannot be synthsized onto FPGA
        $dumpvars(0, tb_fullAdder);

        /* Test case 1
        a_tb = 4'b0000; b_tb = 4'b0000; cIn_tb = 1'b0;
        @(posedge Clk_s);

        // Test case 2
        a_tb = 4'b1111; b_tb = 4'b1111; cIn_tb = 1'b0;
        @(posedge Clk_s);

        // Test case 3
        a_tb = 4'b1010; b_tb = 4'b1100; cIn_tb = 1'b1;
        @(posedge Clk_s);

        // Test case 4
        a_tb = 4'b1111; b_tb = 4'b0001; cIn_tb = 1'b1;
        @(posedge Clk_s);

        // Test case 5
        a_tb = 4'b0001; b_tb = 4'b0001; cIn_tb = 1'b0;
        @(posedge Clk_s);

        // Test case 6
        a_tb = 4'b1001; b_tb = 4'b0110; cIn_tb = 1'b1;
        @(posedge Clk_s);

        */
        //Adding C to 3
        a_tb = 4'b1011; b_tb = 4'b0011; cIn_tb = 1'b0;
        @(posedge Clk_s);
        //Finish simulation
        $finish;
    end
endmodule
```

FIgure 4: Adder Test Cases

b.

Next we build the 16 bit adder. Thankfully the team can preserve a lot of code and developing time by just importing the previous adder module 4 times. Model shown in figure 7.1. There isn't much of a difference between the 4 bit adder and the 16 bit adder. The inputs: a and b, are now 16 bit arrays and cIn remains the same. The outputs are also similar but different as well. Now the sum output is 16 bits long and the cOut is now an array of 4 bits. The implementation is simple. First we initialize 4 4 bit adder modules. Each respective adder module will take care of 4 bits of the 16 bit string. Shown in figure 7. The team must be careful to make sure that all the processing done by the 4 bit adder modules are not directly passed into the output of the 16 bit adder module, but rather a temporary wire variable. Not doing this will cause the program to crash. It is important to note that the cOut for every module starts as the cIn for the next module, this way we can properly address all the calculations being done. The testbench was done in the same format as the 4 bit adder. In which we test each test case through the change of a clock cycle. Shown in figure 6.

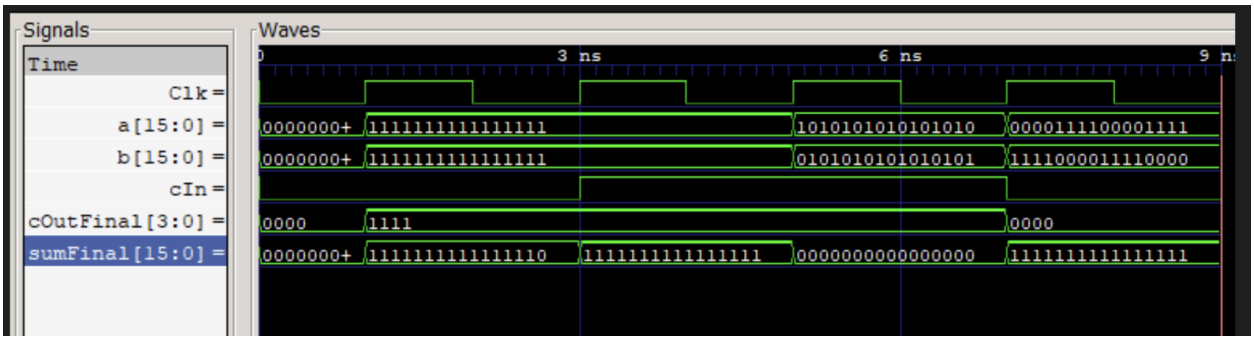

Figure 5: Waveform Generation of 16-bit Adder

```
`timescale 1ns/1ns
`include "fourFullAdder.v"
module tb_fourFullAdder();

    //Initalize the registers

  reg [15:0] a;
    reg [15:0] b;
    reg cIn;

    //Initalize the wires (outputs)
```

```
    wire [3:0] cOutFinal;
    wire [15:0] sumFinal;

    //Intialize a fourFullAdderObject
    fourFullAdder fa(
        .a(a),
        .b(b),
        .cIn(cIn),
        .cOutFinal(cOutFinal),
        .sumFinal(sumFinal)
    );

    //Define the clock
    reg Clk;

    //Get clock going
    always begin
        Clk <= 0;
        #1;
        Clk <= 1; //Here i am using the non blocking since i am in one of those blocks
        #1;
    end

   // Test cases
    initial begin
        $dumpvars(0, tb_fourFullAdder); // To capture waveform data
        /*
        // Test case 1: Add 0 + 0 with carry-in 0
        a = 16'b0000000000000000;
        b = 16'b0000000000000000; //We want blocking assingment so it is sequential logic
        cIn = 0;
        @(posedge Clk);

        // Test case 2: Add maximum values without carry-in
        a = 16'b1111111111111111;
        b = 16'b1111111111111111;
        cIn = 0;
        @(posedge Clk);

        // Test case 3: Add maximum values with carry-in
        a = 16'b1111111111111111;
        b = 16'b1111111111111111;
        cIn = 1;
        @(posedge Clk);

        // Test case 4: Add random values
        a = 16'b1010101010101010;
        b = 16'b0101010101010101;
        cIn = 1;
        @(posedge Clk);

        // Test case 5: Add alternating bits
```

```
        a = 16'b0000111100001111;
        b = 16'b1111000011110000;
        cIn = 0;
        */
        a = 16'b1100101110101001;
        b = 16'b1000011101100101;
        cIn = 0;
        @(posedge Clk);

        // Finish simulation
        $finish;
    end


endmodule
```

Figure 6: 16-bit testbench and test cases

```
module fourFullAdder(a, b, cIn, cOutFinal, sumFinal);

    //Initalize all the inputs
    input [15:0] a, b;
    input cIn;


    //Intialize all the wires since we are moving from one module to another
    wire [3:0] cOut; //These must be wire since they are the output of a fullAdder operation
    wire [15:0] sum;


    //Initailze all outputs
    output reg [3:0] cOutFinal;
    output reg [15:0] sumFinal;

    //Now we need to make sure that all the modules for the full adders are correctly made
    fullAdder zeroToThree (
        .a(a[3:0]),
        .b(b[3:0]),
        .cIn(cIn),
        .cOut(cOut[0]),
        .sum(sum[3:0])
    );

    fullAdder fourToSeven (
        .a(a[7:4]),
        .b(b[7:4]),
        .cIn(cOut[0]),
        .cOut(cOut[1]),
        .sum(sum[7:4])
    );

    fullAdder eightToEleven (
        .a(a[11:8]),
        .b(b[11:8]),
        .cIn(cOut[1]),
        .cOut(cOut[2]),
        .sum(sum[11:8])
    );


    fullAdder twelveToFifteen (
        .a(a[15:12]),
        .b(b[15:12]),
        .cIn(cOut[2]),
        .cOut(cOut[3]),
        .sum(sum[15:12])
    );
```

Figure 7: 16-bit Adder code

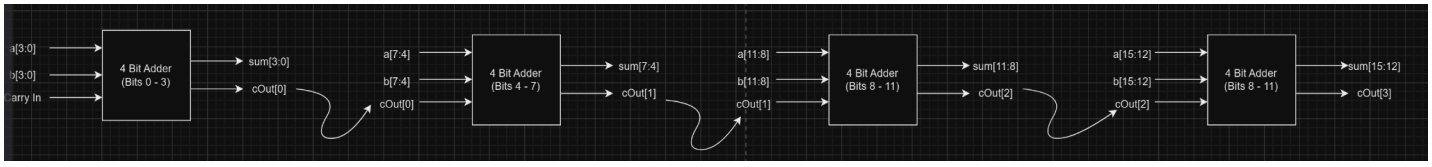

Figure 7.1: 16-bit Adder Model

C.

Below shows the output of the specified test case. In figure 8.

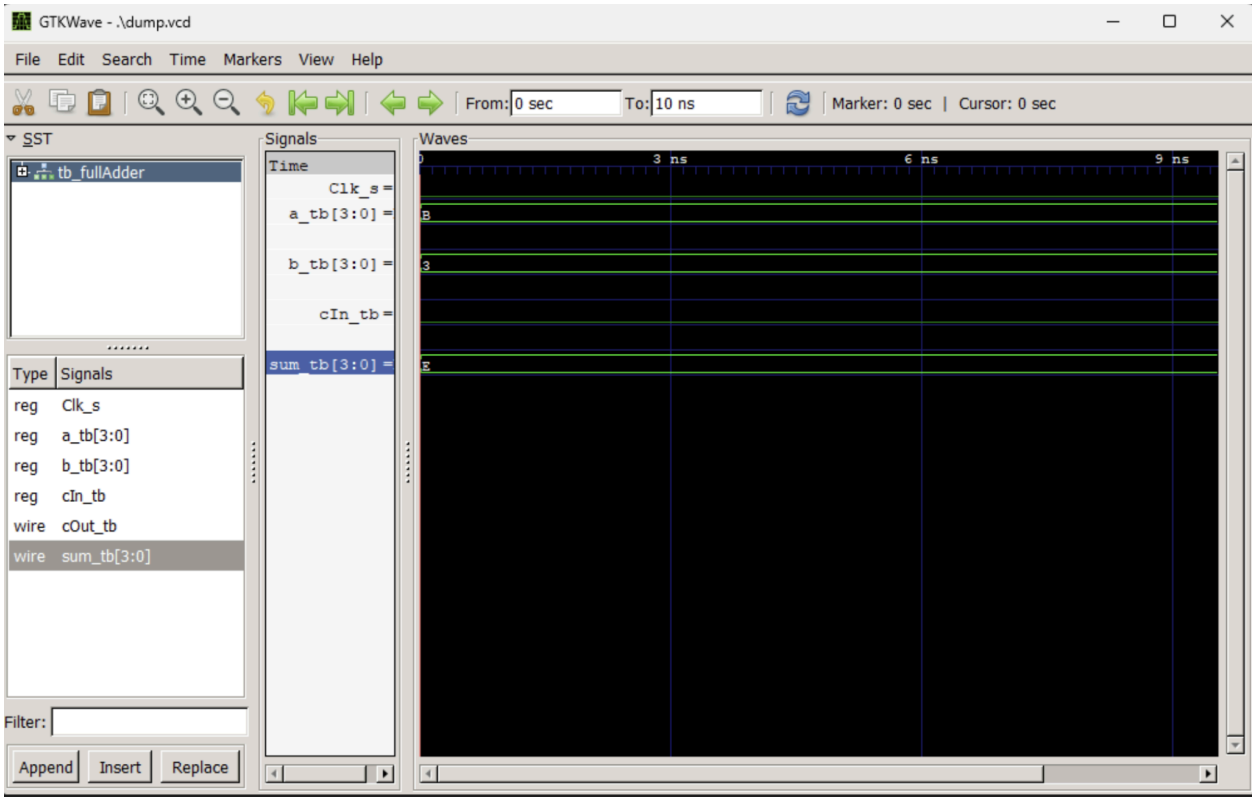

Figure 8: Waveform results for p.93

d.

Below shows the output of the specified test case. In figure 9.

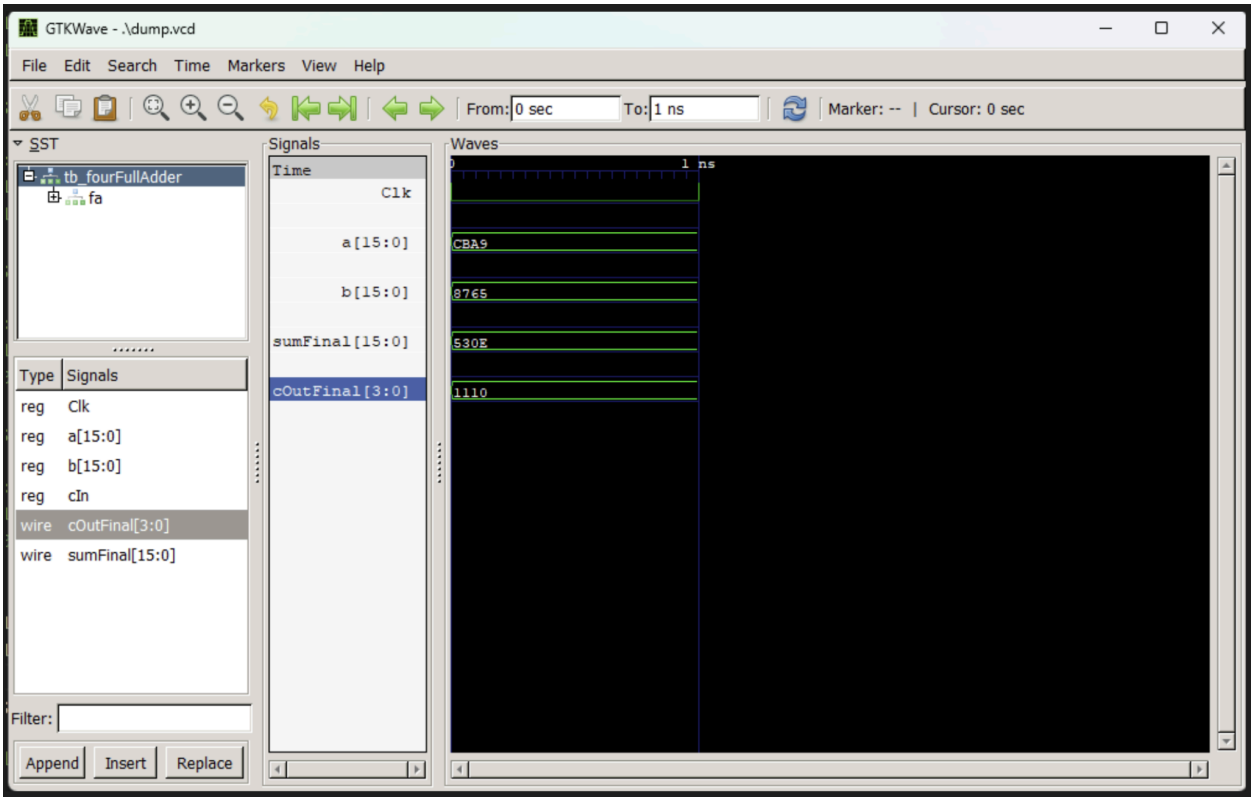

Figure 9: Waveform results for p.97

FPGA Chapter 3

Deliverables (B)

a.

Include: pg 93. What did you see on your results for the “one hot”?

During the experimentation of the provided system verilog files, the team found out that when the “one hot” code for the leading ones file was generated, the correct LED configurations would only show when only one switch was turned on. This proves the one hot encoding of the switches since it shows that each numerical value is programmed towards one and only one value only. This program was not meant to add or subtract any values.

https://drive.google.com/file/d/1MUlHuivqLqCvJI5-TIs0KrCykwykVtBQ/view?usp=drive_link

b.

Include: pg 97 Check your adder. Confirm your results match.

Here in the link below, the team can confirm the test results with the textbook. The adder will take the bottom 8 switches, configured to a 8 bit number and add them to the top 8 switches in the same configuration.

https://drive.google.com/file/d/1ABh0ZjQ7FNGfhEpxSTHAw7IRNnZqLb0N/view?usp=drive_link

c.

Include: pg 98 Check your subtractor. Try 0-1, What is your result?

We know that the system verilog code is programmed to show the subtraction results in 2’s complement, which will support correct results for subtraction operations that result in negative numbers. The team knows that -1 in 2’s complement is all ones and that is exactly what the LED configuration reflects within the teams experimental results.

https://drive.google.com/file/d/1nmxxzCwEOGRslVI7hR8HYPWysMcScP_1/view?usp=drive_link

d.

Include: pg98. Check your multiplier. Run the simulator and use the add force TCL command and is done in the book. This is a powerful debug technique. Confirm that you can match the results.

https://drive.google.com/file/d/1bTwNZ6kPPdLsdLKjXELdwpBU62r5DPQ_/view?usp=drive_link

Final Utilization (C)

Below in figure 10 shows our final utilization graph. This reflects the amount of LUT's used in the final synthesis of the code and how many IO we are using compared to how many are available on the board. This utilization graph reflects the results of the textbook.

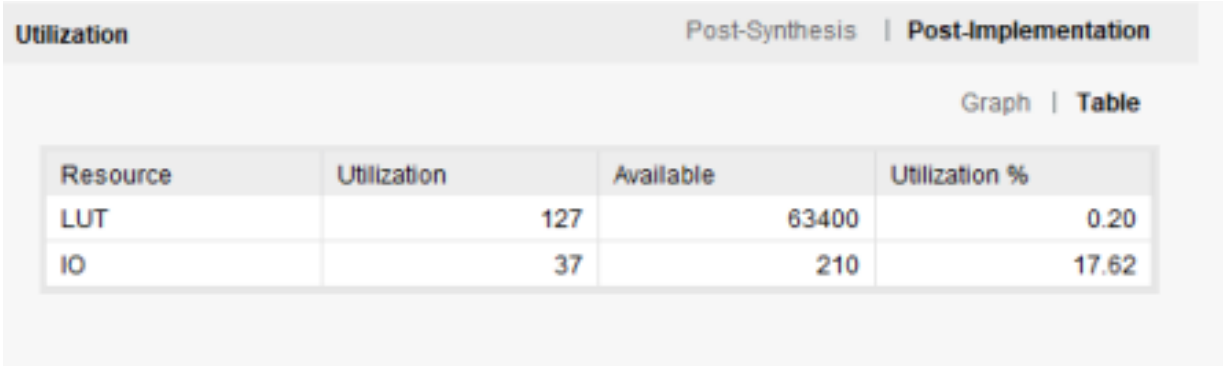

Figure 10: Utilization chart

Video Uploads (D)

Below shows our results for the following test cases provided by the write up sheet. All of the test cases passed and the team was able to gain a better understanding of how to properly use Vivado and the FPGA board.

a.

00000001,00000001

https://drive.google.com/file/d/1hOM9cWEr75yntQeZ-36YwCHXV9oghPjg/view?usp=drive_link

b.

10000000,00000001

https://drive.google.com/file/d/1WR6B59JNQupihMSM8COrPeDNSac5Q9PC/view?usp=drive_link

c.

11000000,11000000

https://drive.google.com/file/d/1ed8OnT2oI-kQTx59spuoCQ1HUWzLDJAo/view?usp=drive_link

d.

10101010,01010101

https://drive.google.com/file/d/1CFHYbvofdPDbYXuD0w9bqCTg_LnDAqMV/view?usp=drive_link

e.

11111111,11111111

https://drive.google.com/file/d/1hIZSKfTyaTDRY5lpye6GggWehuDQkalv/view?usp=drive_link

References

FPGA Handbook:
https://github.com/PacktPublishing/The-FPGAProgramming-Handbook-Second-Edition

Chip Verify Full Adder in Verilog Tutorial:

https://www.chipverify.com/verilog/verilog-full-adder