

Simulation Exercise 5

Hierarchical Design

ELEE 2640

Benyamain YACOOB

February 25, 2024

Date Performed: February 22, 2024

Partners: Ara OLADIPO
Andre PRICE

Instructor: Professor PAULIK



Contents

1 Objectives	3
2 Problem Statement	3
3 Materials	3
4 Requirements	3
5 1-Bit Full Adder	6
5.1 Results and Analysis Discussion	10
6 4-Bit Ripple Carry Adder	12
6.1 Results and Analysis Discussion	15
7 4-Bit Adder-Subtractor	16
7.1 Results and Analysis Discussion	18
8 Results and Conclusion	19
9 Group Contributions	20

1 Objectives

First Objective

Code in SystemVerilog for combinational logic design and testing.

Second Objective

Simulate and implement combinational circuits using Xilinx Vivado integrated development environment (IDE).

Third Objective

Transfer earlier discrete-IC lab work to a SystemVerilog hardware description language (HDL) implementation.

2 Problem Statement

In this exercise you will be performing many of the same tasks that you did in Exercises 2 & 3, but using the more advanced SystemVerilog HDL, the Xilinx ISE, and Digilent Basys3 board interface (introduced in Exercise 4) to implement your designs.

3 Materials

Xilinx integrated synthesis environment (ISE)

4 Requirements

Common Anode and Common Cathode Format

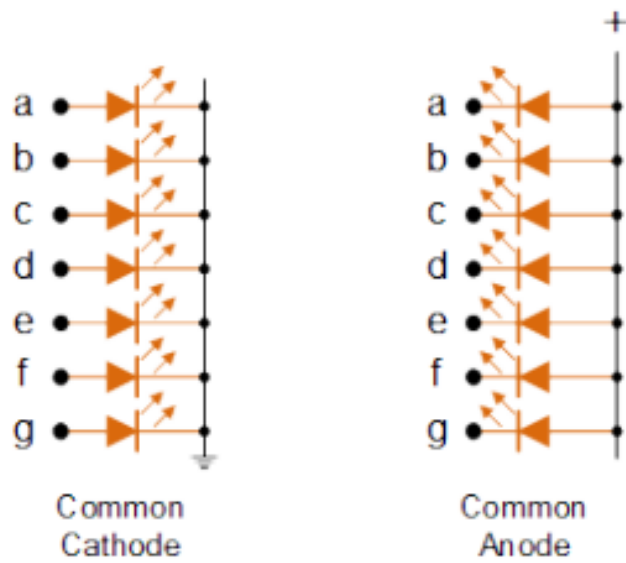


Figure 1: Common Anode and Common Cathode Wiring

Seven-Segment Display Format

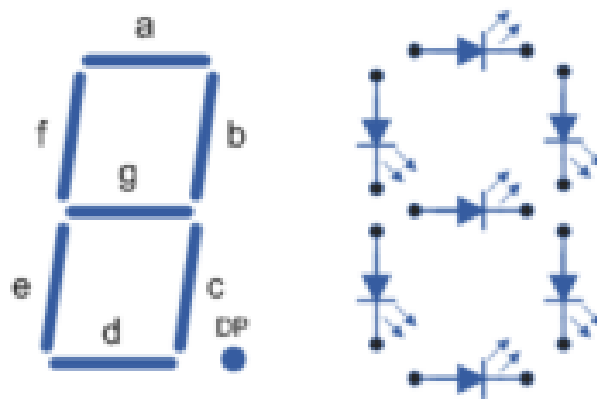


Figure 2: Seven-Segment Display Segments and Diodes

HEX Data Display on Seven-Segment Device Format



Figure 3: HEX Data Display on Seven-Segment Device

1-Bit Full Adder

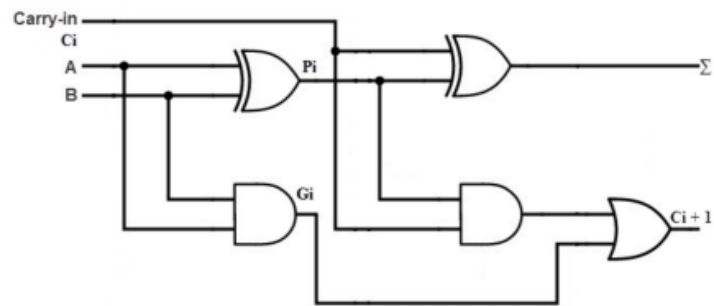


Figure 4: Circuit for Carry

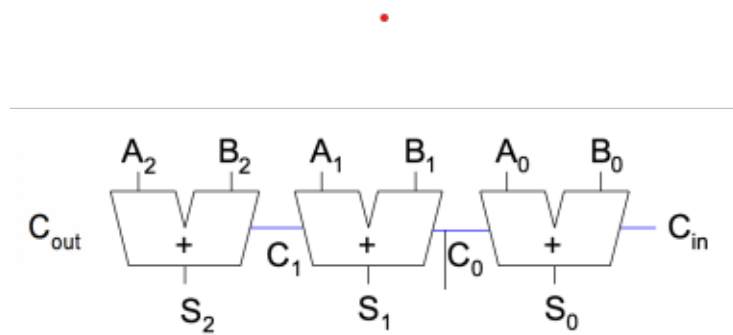


Figure 5: 3-Bit Adder

5 1-Bit Full Adder

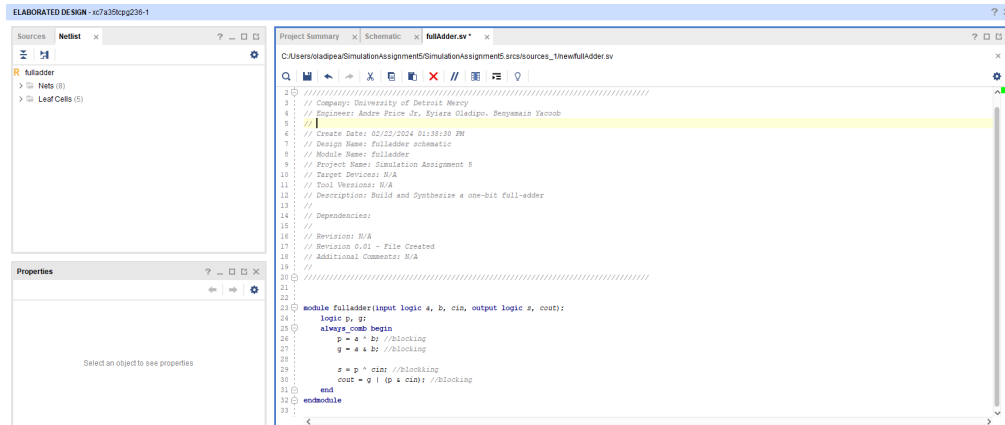


Figure 6: 1-Bit Full Adder Code Implementation One

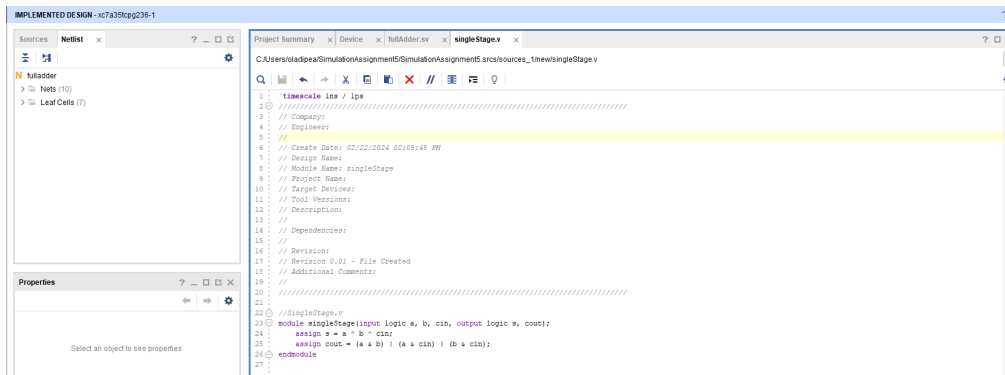


Figure 7: 1-Bit Full Adder Code Implementation Two

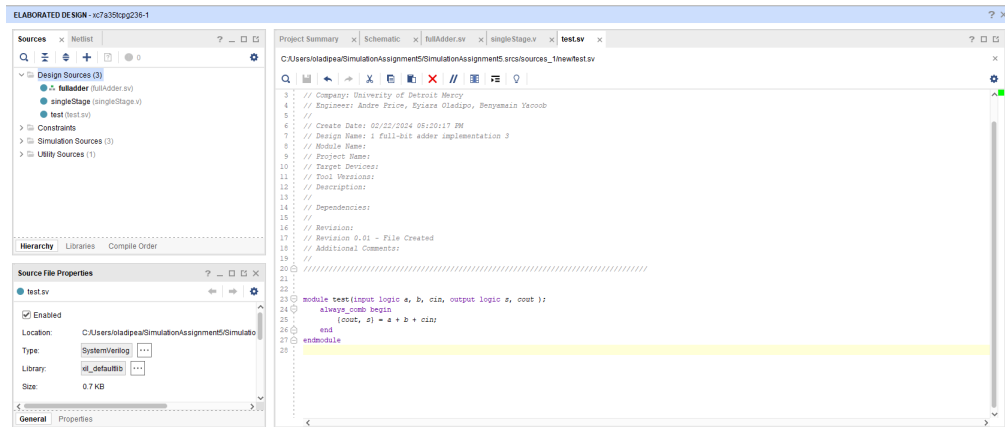


Figure 8: 1-Bit Full Adder Code Implementation Three

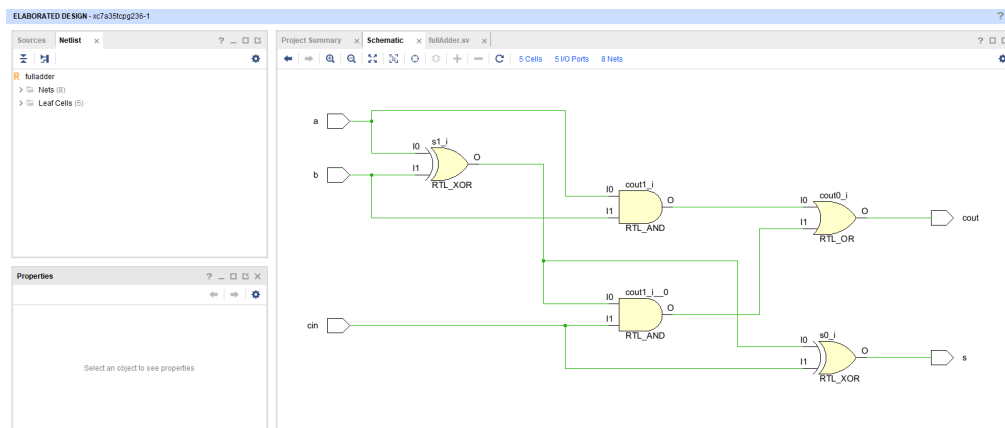


Figure 9: 1-Bit Full Adder RTL Schematic

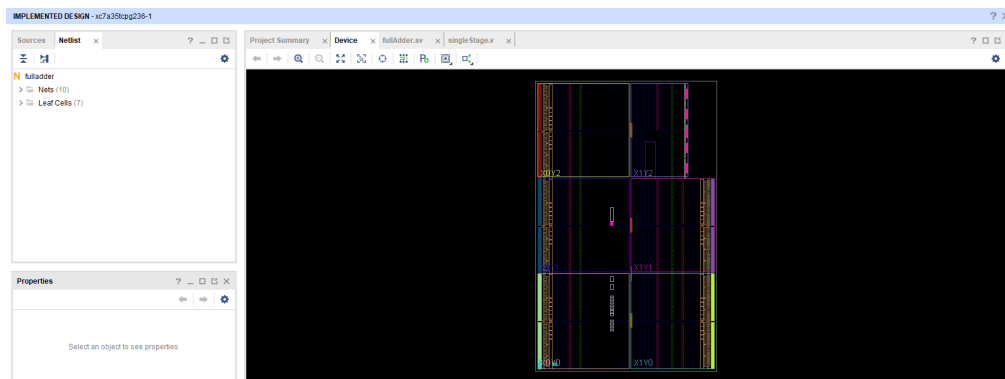


Figure 10: 1-Bit Full Adder Synthesized Design

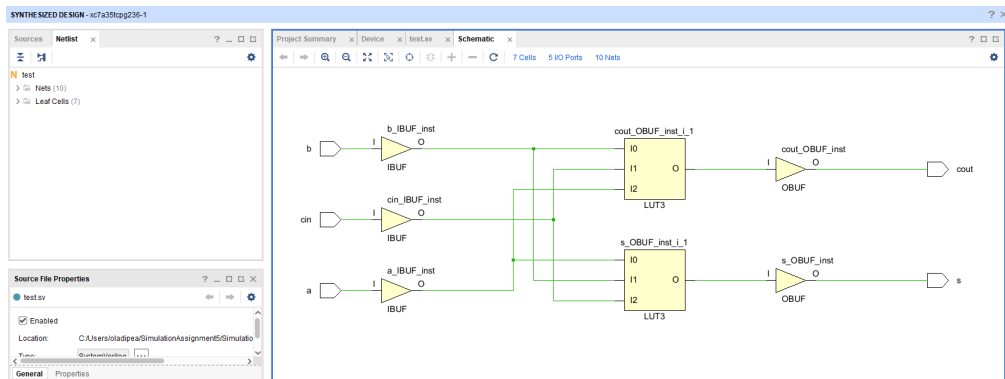


Figure 11: 1-Bit Full Adder Synthesized Schematic


```

18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module tb_1_bit_full_adder();
24     logic A, B, Cin, Sum, Cout;
25
26     fulladder myfulladder(
27         .a(A),
28         .b(B),
29         .cin(Cin),
30         .s(Sum),
31         .cout(Cout)
32     );
33
34     initial begin
35         A = 0;
36         B = 0;
37         Cin = 0;
38
39         //Waiting 20ns based on the rubric
40         #20
41
42         //Inputs changing every 20ns going from 000 to
43         #20 (A, B, Cin) = 3'b001;
44         #20 (A, B, Cin) = 3'b010;
45         #20 (A, B, Cin) = 3'b011;
46         #20 (A, B, Cin) = 3'b100;
47         #20 (A, B, Cin) = 3'b101;
48         #20 (A, B, Cin) = 3'b110;
49         #20 (A, B, Cin) = 3'b111;
50         #20 $finish;
51     end
52 endmodule
53

```

Figure 12: 1-Bit Full Adder Test Bench Code

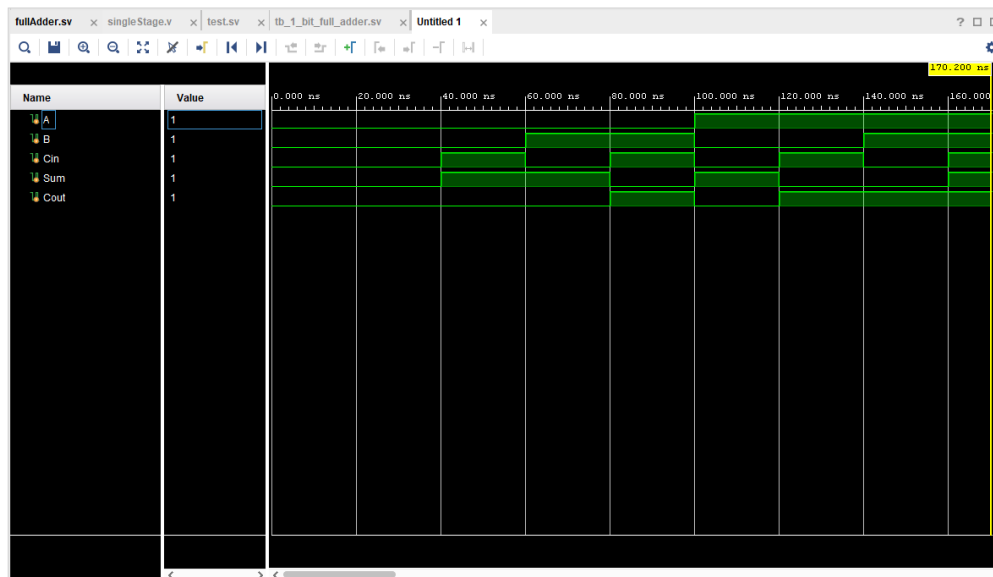


Figure 13: 1-Bit Full Adder Behavioral Simulation

5.1 Results and Analysis Discussion

Which adder coding approach do you prefer and why?

Our group preferred implementing using the code displayed in Figure 6 for the 1-bit full adder. This preference was influenced by familiarity rather than practicality. Approach one resonated with the group due to its nostalgic resemblance to other coding languages, particularly in terms of syntax. The combination of visuals and coding elements reminded the group of a mixture of C++ and SQL, making it easier to comprehend compared to alternative approaches.

Compare and contrast the resulting RTL logic circuits generated for the various coding styles.

While the underlying logic circuits remained identical, the apparent differences in the code were alternative methods of coding the same circuit. All elements of the circuits, including the gates and lines, were the same.

Based on some straightforward online searching, indicate what is the difference between a ripple carry adder and a carry propagate adder? Is one always to preferred over the other or is there a cost-benefit trade-off?

There are many differences between a ripple carry adder (RCA) and a carry propagate adder (CPA). In an RCA, each full adder stage generates its own carry bit, which ripples through the subsequent stages. This form of propagation can result in longer delay times, as each stage must wait for the carry from the previous stage to be computed. A CPA, however, ensures carry bits are computed in parallel across all stages simultaneously. This parallel carry propagation reduces the critical path delay, leading to faster operation compared to RCA. Which is preferred depends fully on the specific application and what is required. If delay must be minimized, then the CPA is the most useful. If a reduced circuit complexity is desired, then a RCA would be preferred.

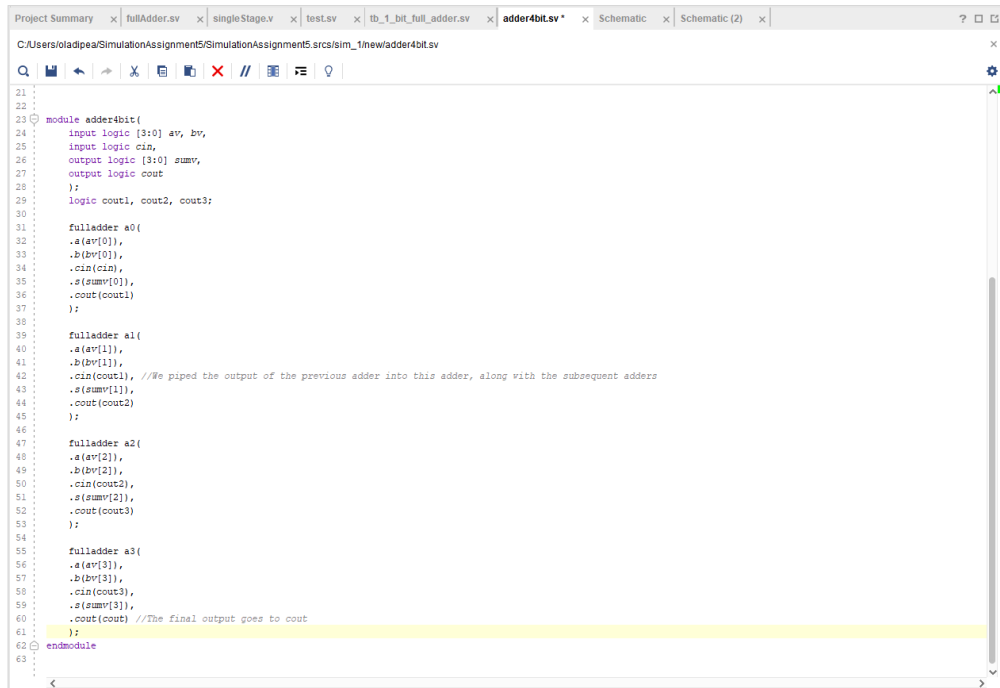
What is the delay from an input change to a stable sum and cout value when you do a post-synthesis/implementation timing simulation?

The delay from an input change to a stable *sum* and *cout* value when performing the timing simulation is about 40 nanoseconds (*ns*). We suppose that the reading of the carry out was an active high signal during the process of summing the input bits. The *sum* of bits starts occurring at about 40 *ns*, and *cout* becomes active high input signal 40 *ns* later, preemptively determining if there is an additional bit that will be produced when *sum* is calculating. Therefore, the difference in delay can be modeled from the equation below, but the final result should be 40 *ns*. We define active as when the simulation showcases the state of either being active low signal or high signal. To account for the differences in readings, we should mention that we do not expect the delay that we recorded to be exact with what is measured in the rubric.

$$Delay = C_{OUT} - SUM \quad (1)$$

While implementing the full adder's structure, we encountered minimal errors due to programmer mistakes, such as missing syntax or similar issues, which were easy to address and did not pose any significant challenges. The dataflow implementation of the full adder's structure and the implementation of the adder's truth table equations helped us understand the process. The final step of implementing the test bench code was relatively simple. The comments provided clear guidance on what needed to be done. However, the code block in this module required thorough examination due to new and subtle syntax, such as named associations for pins. These details were important for the code to function as intended. The concept of test benches aligns with professional coding practices commonly observed in software development, where most code undergoes a process known as unit testing. As we are still learning about the combined use of Vivado and SystemVerilog, it is a gradual process to understand the material fully. This also means that we may need to review the code multiple times to gain a more intuitive understanding and implement it effectively when required. These simulation assignments are conducted sequentially, helping the rapid introduction of concepts, coupled with clear instructions and visuals. This serves as a verification process for the experimenters while completing the assignment.

6 4-Bit Ripple Carry Adder



```
21
22
23 module adder4bit(
24     input logic [3:0] av, bv,
25     input logic cin,
26     output logic [3:0] sumv,
27     output logic cout
28 );
29     logic cout1, cout2, cout3;
30
31     fulladder a0(
32         .a(av[0]),
33         .b(bv[0]),
34         .cin(cin),
35         .s(sumv[0]),
36         .cout(cout1)
37     );
38
39     fulladder a1(
40         .a(av[1]),
41         .b(bv[1]),
42         .cin(cout1), //We piped the output of the previous adder into this adder, along with the subsequent adders
43         .s(sumv[1]),
44         .cout(cout2)
45     );
46
47     fulladder a2(
48         .a(av[2]),
49         .b(bv[2]),
50         .cin(cout2),
51         .s(sumv[2]),
52         .cout(cout3)
53     );
54
55     fulladder a3(
56         .a(av[3]),
57         .b(bv[3]),
58         .cin(cout3),
59         .s(sumv[3]),
60         .cout(cout) //The final output goes to cout
61     );
62 endmodule
63
```

Figure 14: 4-Bit Ripple Carry Adder Code

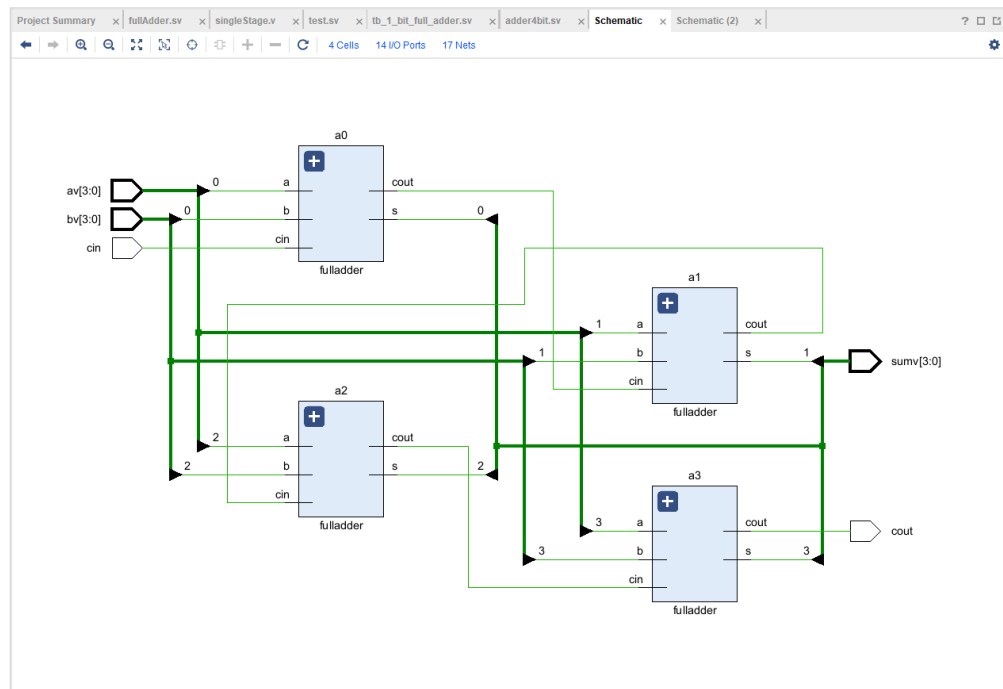


Figure 15: 4-Bit Ripple Carry Adder RTL Schematic

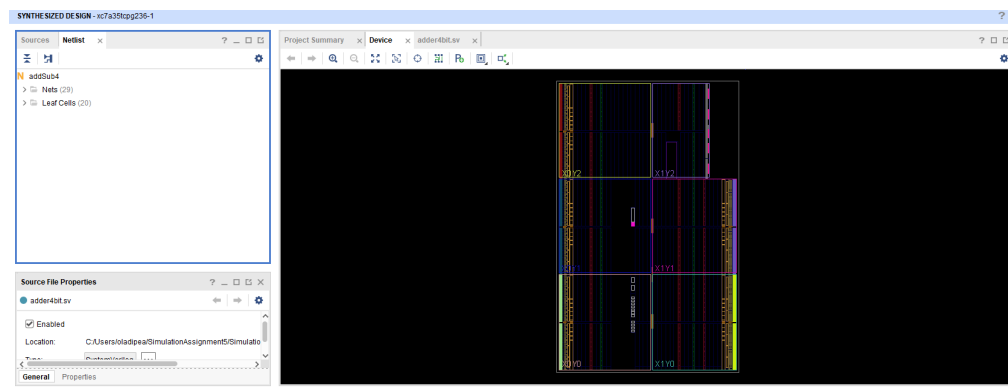


Figure 16: 4-Bit Ripple Carry Adder Synthesized Design

```

22 module tb_adder4bit();
23 // Inputs
24 logic [3:0] a; logic [3:0] b;
25
26 // Outputs
27 logic [3:0] sum; logic cin, carry; logic clk, reset;
28
29 // Test
30 integer i;
31
32 adder4bit uut(
33     .a(a),
34     .b(b),
35     .sum(sum),
36     .cin(cin),
37     .clk(clk),
38     .reset(reset)
39 );
40
41 always begin
42     clk = 1; #20; clk = 0; #20;
43 end
44
45 initial begin
46     a = 0;
47     b = 0;
48     cin = 0;
49 end
50
51 $monitor("a(b) + b(b) = carry sum(b b)", a, b, carry, sum);
52
53 always begin
54     for (i=0; i< 16; i= i + 1) begin
55         {a, b} = i;
56         #20 if(a + b != (carry + sum))
57             $display("ERROR");
58     end
59     $display("4-bit test completed", i);
60     #20 $stop;
61 end
62 endmodule
63

```

Figure 17: 4-Bit Adder Test Bench Code

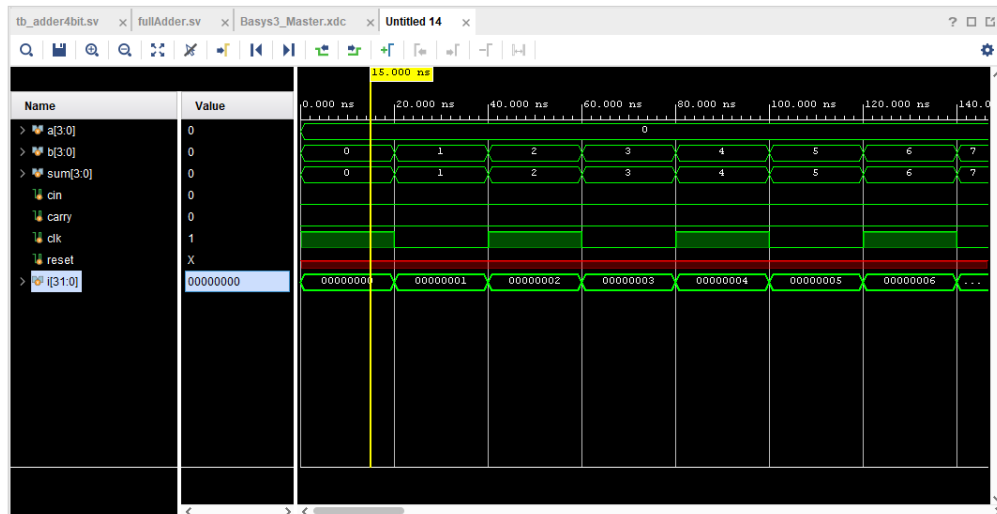


Figure 18: 4-Bit Adder Test Bench Behavioral Simulation

```

a(0010) + b(0000) = carry sum(0 0010)
a(0010) + b(0001) = carry sum(0 0011)
a(0010) + b(0010) = carry sum(0 0100)
a(0010) + b(0011) = carry sum(0 0101)
a(0010) + b(0100) = carry sum(0 0110)
a(0010) + b(0101) = carry sum(0 0111)
a(0010) + b(0110) = carry sum(0 1000)
a(0010) + b(0111) = carry sum(0 1001)
a(0010) + b(1000) = carry sum(0 1010)
a(0010) + b(1001) = carry sum(0 1011)
a(0010) + b(1010) = carry sum(0 1100)
a(0010) + b(1011) = carry sum(0 1101)
a(0010) + b(1100) = carry sum(0 1110)
a(0010) + b(1101) = carry sum(0 1111)
a(0010) + b(1110) = carry sum(1 0000)
a(0010) + b(1111) = carry sum(1 0001)
a(0011) + b(0000) = carry sum(0 0011)
a(0011) + b(0001) = carry sum(0 0100)
a(0011) + b(0010) = carry sum(0 0101)

```

Figure 19: 4-Bit Adder Test Bench Console Output

6.1 Results and Analysis Discussion

Designing the 4-bit adder was relatively straightforward, largely due to the prior implementation of a 1-bit adder from a previous section. Completing the remaining code sections presented minimal challenges, as our understanding of 4-bit adders was reinforced through previous simulation assignments. The generated RTL schematic matched the requirements for this simulation section, and a test bench was created to verify its functionality. Developing the test bench was also straightforward, allowing us to confirm the proper operation of the 4-bit adder through console outputs. Overall, the 4-bit adder implementation was a smooth and successful process.

7 4-Bit Adder-Subtractor

```

Project Summary x Schematic x tb_adder4bit.v x addSub4.v* x Schematic (2) x adder4bit.v x
C:\Users\oladipea\SimulationAssignment5\SimulationAssignment5.srcs\sources_1\new\addSub4.v

15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module addSub4(
24     input logic [3:0] av, bv,
25     input logic M,
26     output logic [3:0] resultv,
27     output logic cout
28 );
29     logic [3:0] ToBorNotToB;
30     assign ToBorNotToB[3:0] = {4{M}} ^ bv[3:0];
31
32     // We are creating our adder with the following parameters
33     // av: Input 1 (given to us in the code)
34     // bv: Input 2 (given to us in the code)
35     // sumv: The output will go into the resultv vector
36     // cout: We are piping cout into cout
37     // cin: We used M as M will stand as our cin
38     adder4bit myAdder(
39         .av(av),
40         .bv(ToBorNotToB),
41         .sumv(resultv),
42         .cout(cout),
43         .cin(M)
44     );
45 endmodule
46

```

Figure 20: 4-Bit Adder-Subtractor Code

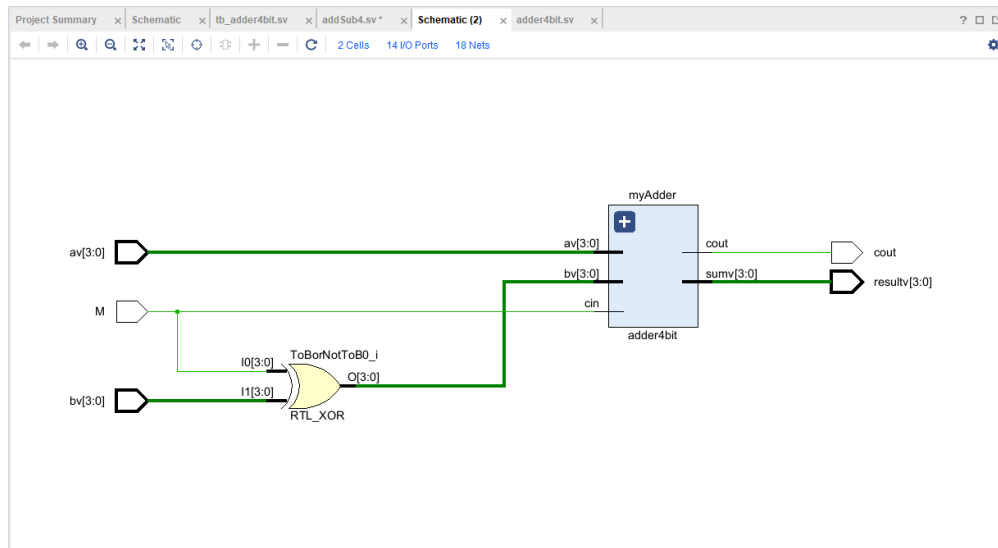


Figure 21: 4-Bit Adder-Subtractor RTL Schematic


```

module test4sub();
    logic [3:0] a; logic [3:0] b; logic Subtract;

    //Outputs
    logic [3:0] result; logic carry;

    logic clk, reset; integer i;

    addSub4 DUT(
        .av(a),
        .bv(b),
        .M(Subtract),
        .resultv(result),
        .cout(carry)
    );

    initial begin
        a = 0;
        b = 0;
        Subtract = 0;
        #5

        for(i=0; i<4; i=i+1) begin
            #1 a = a + 11; b = b + 15;
        end

        #5 Subtract = 1; a = 50; b = 10;
        for(i=0; i<4; i=i+1) begin
            #1 a = a - 10; b = b + 10;
        end

        #5 $finish;
    end
    initial begin
        $timeformat(-9, 2, " ns", 10);
        $monitor("At time %t: a=%d, b=%d, Subtract=%b, ToBorNotToB=%b, carry = %b Result=%d (%b)",
            $time, a, b, Subtract, DUT.ToBorNotToB, carry, result, result);
    end
endmodule

```

Figure 22: 4-Bit Adder-Subtractor Test Bench Code

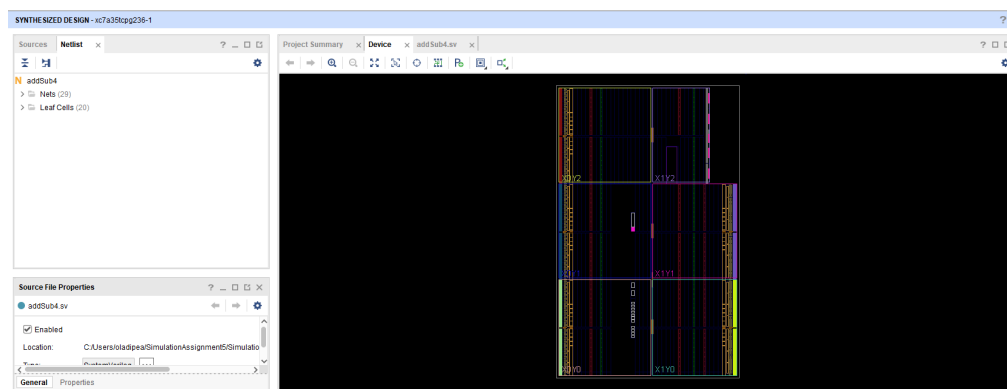
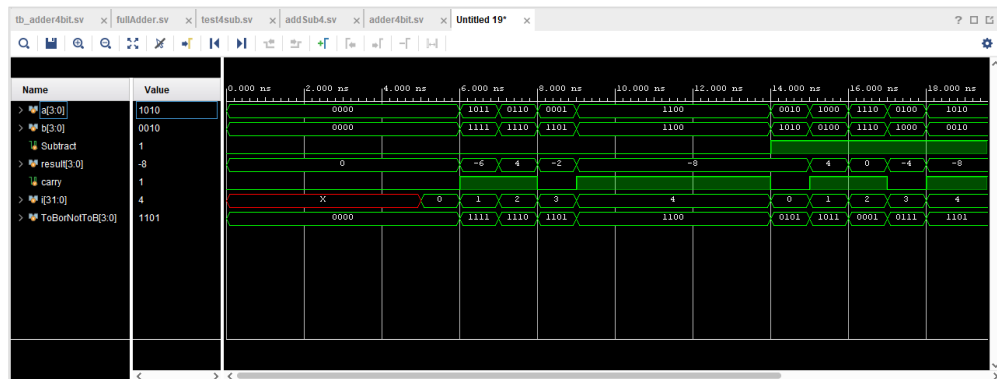
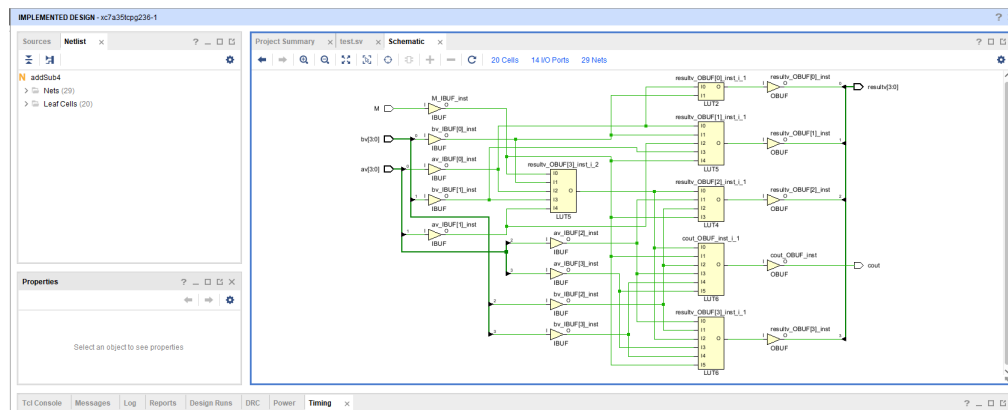


Figure 23: 4-Bit Adder-Subtractor Synthesized Design



```
Tcl Console x Messages log ?
Q 子 全 并 回 重 刷
Completed static elaboration
INFO: (USF-XSim-62) No Change in RTL. Linking previously generated obj files to create kernel
INFO: (USF-XSim-69) "elaborate" step finished in '1' seconds
Time resolution is 1 ps
At time 0.00 ns: a= 0, b= 0, Subtract=0, ToBortToB=0000, carry= 0 Result= 0 (0000)
At time 6.00 ns: a=1, b=1, Subtract=0, ToBortToB=1111, carry= 1 Result=10 (1010)
At time 7.00 ns: a= 6, b=14, Subtract=0, ToBortToB=1110, carry= 1 Result= 4 (0100)
At time 8.00 ns: a= 1, b=13, Subtract=0, ToBortToB=1101, carry= 0 Result=11 (1110)
At time 9.00 ns: a=1, b=12, Subtract=0, ToBortToB=1100, carry= 1 Result= 9 (1000)
At time 14.00 ns: a= 2, b=10, Subtract=1, ToBortToB=0101, carry= 0 Result= 8 (1000)
At time 15.00 ns: a= 6, b= 4, Subtract=1, ToBortToB=0101, carry= 1 Result= 4 (0100)
At time 16.00 ns: a=1, b=14, Subtract=1, ToBortToB=0001, carry= 1 Result= 6 (0000)
At time 17.00 ns: a= 6, b= 3, Subtract=1, ToBortToB=0111, carry= 0 Result=12 (1100)
At time 18.00 ns: a=10, b= 2, Subtract=1, ToBortToB=1101, carry= 1 Result= 8 (1000)
#Finish called at time: 23 ns : File "C:/Users/Chadape/Simulation/Assignment5/srcs/sim_3/new/cerestab.vv" File 53
<
Tcl command here
```

after a 5 *ns* wait. Only when the for loop starts does integer *i* receive an initial value.

In the initial stages of developing the 4-bit adder-subtractor, an oversight led to a challenge. Specifically, we neglected to incorporate the flag that sets the subtract variable to 1 after 14 *ns*. Recognizing this error, we fixed it, validating the code's functionality as intended. Additionally, we simulated a test bench waveform as shown in the rubric's requirements. This waveform effectively showcases the accurate operation of the adder-subtractor. We also validated the console output that came from the test bench, confirming its results.

8 Results and Conclusion

This section discusses the results of the Vivado simulation experiments. The discussion provides in-depth explanations of the results and any discrepancies between the results and theoretical expectations. It also explores potential sources of error and discusses the accuracy of the tests, the debugging process, and what was learned from these experiences.

This simulation assignment was itself to be more challenging than the previous one, although that sentiment only applies to certain aspects of it. Conceptually, the topic was easy to understand since it is something we had already completed almost a month ago. Coming into this simulation assignment, we already knew the similarities between the 4-bit subtractors and 4-bit adders. We also knew the truth tables for both, including the *SUM* output and the *CARRY* output; all in all, this was a topic that we already had familiarity with.

The problem was due to some of the coding aspects. It was safe to see that the coding segments required to complete this simulation assignment provided a new onslaught of difficulty. Earlier in this document, we mentioned as a group that some of the earlier code reminded us of C++ and SQL, which made the process of coding easier; but, that was not an aspect clear in these coding examples. It was much harder to comprehend the instantiating process because of how the code was formatted, appearing more abstract rather than concrete and easily calculable, almost acting as an oxymoron to our previous opinion on SystemVerilog. This was made abundantly clear with the multiple error messages we got throughout the coding process as it pertains to the later circuits. It took a lot of group discussion to find solutions to the many different error messages we faced, and even more backtracking to understand the apparent complexity displayed in the later circuit examples. This led the group to posit that we would have to spend more time in the future studying the different aspects of the software to fully understand its nuances.

9 Group Contributions

The work for this simulation assignment was divided among three individuals: Benyamain Yacoob, Andre Price Jr, and Eyiara Oladipo. Price and Oladipo collaborated using Vivado software and writing in SystemVerilog to wire the circuits. Although the work was done independently, all group members were present to review the progress and provide feedback. Each member focused on their respective sections of the paper, and Benyamain took the lead in organizing and proofreading the entire document before submission.