COEN 21L

# Lab 5: 4-bit Ripple Carry Adder


Dillon Kanai and Mariela Zuniga

Lab 5 Report

October 31, 2019

**Introduction:** In this lab we created a circuit to represent a 4 -bit ripple carry adder which is composed of 4 full adders connected to each other. The purpose of the lab was to add two 4-bit binary numbers and output their sum. We did this by writing the verilog code for the full adder and using that code for our ripple carry adder. Our inputs were A (4 bit binary number), B (4 bit binary number), carry in (Cin), and our outputs were V (the arithmetic overflow output), C4 (the carry out) and the Sout (the sum of A and B). We also had a segment of our lab where we debugged our code by only looking at the outputs on the FPGA.

**Schematics and Simulation:**

```
1   module myfulladd(A, B, Cin, Sout, Cout);
2
3       input A, B, Cin;
4       output Sout, Cout;
5
6       assign Sout= A ^ B ^ Cin;
7       assign Cout= (A & B)|(A & Cin)|(B & Cin);
8
9   endmodule
10
```

Figure 1: Full Adder Verilog Code

```
1   module myfulladd4(A, B, Cin, V, Sout, C4);
2       input [3:0]A,B;
3       input Cin;
4       output [3:0]Sout;
5       output C4, V;
6       wire c0, c1, c2;
7
8       assign V= (A[3] & B[3]) ^ Sout[3];
9       myfulladd FA0(A[0], B[0], Cin, Sout[0], c0);
10      myfulladd FA1(A[1], B[1], c0, c1, Sout[1]);
11      myfulladd FA2(A[2], B[2], c1, Sout[2], c2);
12      myfulladd FA3(A[3], B[3], c2, Sout[3], C4);
13
14  endmodule
15
```

Figure 2: Full Adder 4 Verilog Code

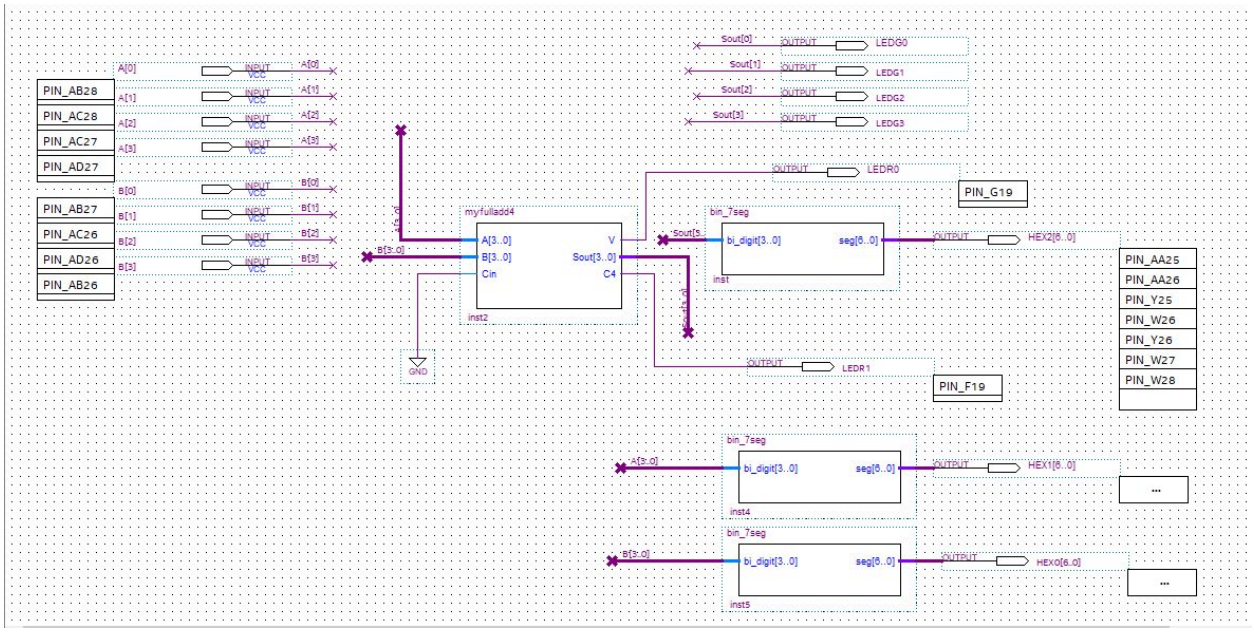



Figure 3: Priority Schematic

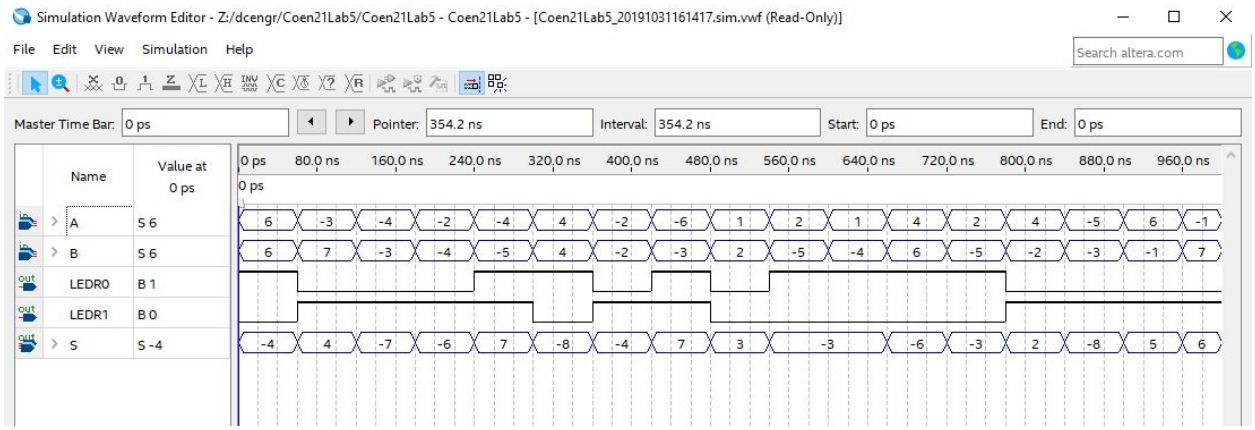



Figure 4: Signed Simulation Results

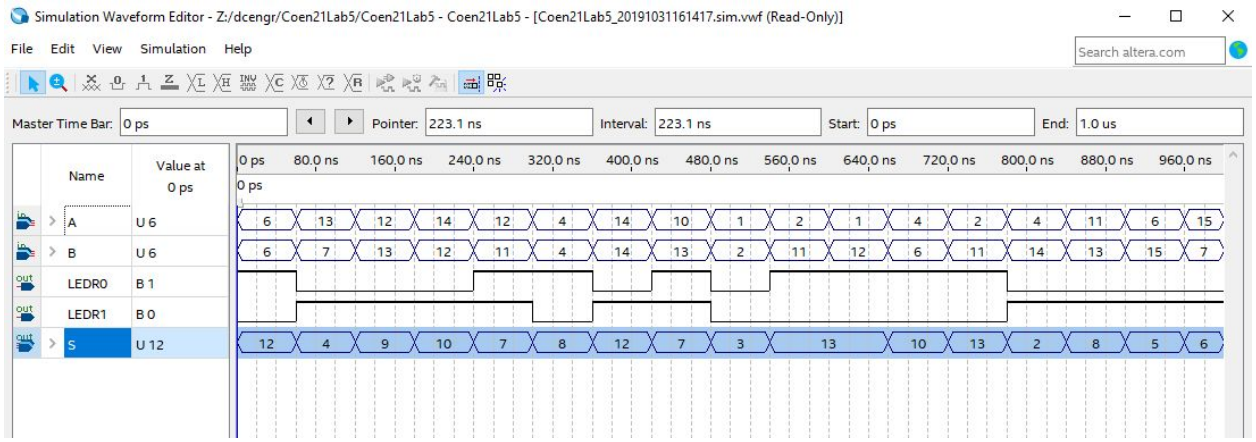



Figure 5: Unsigned Simulation Results

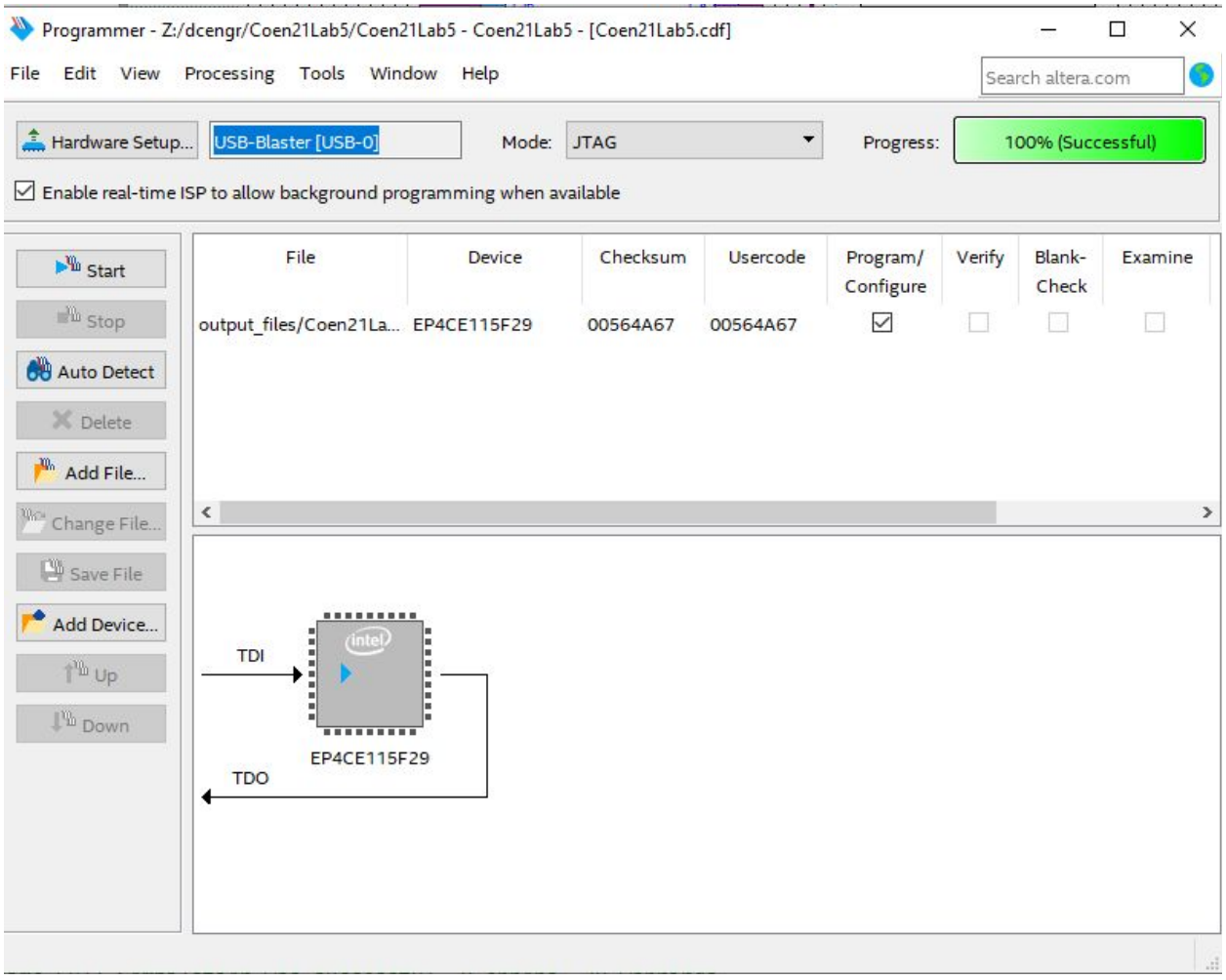



Figure 6: Proof of Successful Download

This is the Table of Values generated from our FPGA

| **X** | **Y** | **Cin** | **S** | **V** | **C4** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 2 | 0 | 0 |
| 4 | 3 | 0 | 7 | 0 | 0 |
| 4 | 4 | 0 | 8 | 1 | 0 |
| 9 | 6 | 0 | F | 1 | 0 |
| 9 | 7 | 0 | 0 | 0 | 1 |
| C | A | 0 | 6 | 1 | 1 |
| E | A | 0 | 8 | 0 | 1 |
| E | A | 1 | 9 | 0 | 1 |

This is the Table of Values generated from the bug our TA added to our code

| X | Y | Cin | S | V | C4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 4 | 0 | 0 |
| 4 | 3 | 0 | 9 | 1 | 0 |
| 4 | 4 | 0 | 8 | 1 | 0 |
| 9 | 6 | 0 | 1 | 0 | 1 |
| 9 | 7 | 0 | E | 0 | 1 |
| C | A | 0 | 8 | 0 | 1 |
| E | A | 0 | 6 | 1 | 1 |
| E | A | 1 | 7 | 1 | 1 |

**Procedure for debugging the code:** We immediately noticed that both X and Y's 2nd switch (2's place) wasn't working. When one 2's place switch was flipped up, it showed a sum of 4. We thought that a wire was probably misplaced. However, if there is one wire misplaced, there has to be another wire misplaced in order to replace the original misplaced wire. So, we thought that one of the values in the Verilog code was switched. This way, it is possible for one value to be wrong by itself. However, we couldn't decide specifically what value was switched because there are too many possibilities. It is most likely in the line

myfulladd(X[1], Y[1], C1, S[1], C1);

Which controls the sum for the 2's place. It turns out the C1 and the Sout[1] were switched. We were correct when we said something was changed in the Verilog code and that it was in the line above.

**Questions:**

***Find one pair of X and Y inputs (with C0=0) that would result in the following:***

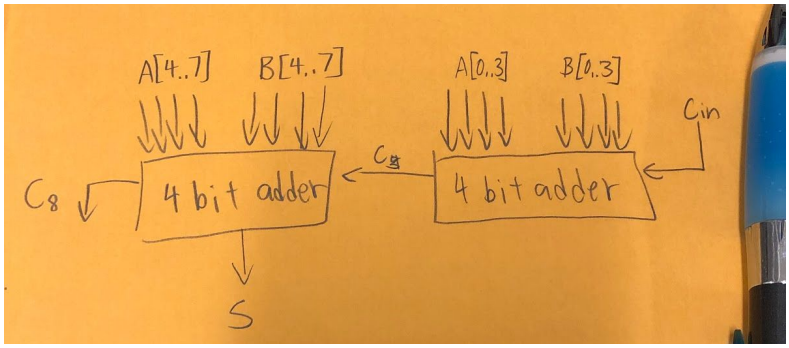
- C4 = 0 and V = 0
    - X = 0, Y = 0
- C4 = 0 and V = 1
    - X = 9, Y = 7
- C4= 1 and V = 0

    - X= 4, Y =3
- C4 = 1 and V = 1
    - X = C, Y = A

***If you had to make an 8-bit adder, show how would you do it using only instances of the 4-bit module you have built in this lab? Specifically show how you would create the C8 output and the V output.***

If we were to use an 8 bit adder using only instances of 8 bit adders, would just increase the number of inputs two times. This creates two 8 bit inputs. As a result, we would need two 4 bit adders. One four bit adder would deal with the first four bits of each input. The second four bit adder would deal with the second four bits of each input. This way, two segments of each input will be processed and added up at the end.

Actual representation:




***Discussion of testing procedures and results:***

Initial implementation: Did the initial testing of your circuit (i.e., before Section 5) go smoothly or did you encounter incorrect results? If the latter, describe how you determined what was wrong.

- One of our designs was not syncing onto the board correctly. Even if the pin assignments were correct, none of the 7 segment lights or RLEDs would light up. To fix this, we set the circuit with the 7 segment blocks to a top level entity. After this, everything worked perfectly. Other than small syntax/compile errors, the lab went smoothly

Testing Challenge: In Section 5 you tested a circuit to detect an error. Summarize what the steps you followed to identify the problem. Discuss whether you could have followed a shorter set of tests to identify the problem.

- Details on debugging code above in “Procedure for debugging the code”. We felt that looking at possible inputs and deriving hints from them was the fastest possible method. Without access to code and only access to inputs, we didn’t really have any other choices. We could have done a k-map and created the logic equation and identify the area with the bug, but that would have taken a lot more time and effort.

***If the circuit were completely correct except that X[1] and Y[1] were interchanged in a full adder input, would you be able to detect this based on observing outputs during testing? Why or why not?***
No, even if those values are changed, they are both involved in the same operations in the same 4 bit adder. Even if they were switched, we would still be adding them together and still producing the same sum of X[1] and Y[1]. Similar to the commutative property of addition, the order in which we add these numbers do not actually matter. It only matters that they are added together in the right place and operation.

**Conclusion:** The majority of the challenges we faced when we were trying to simulate the verilog code and schematic onto the FPGA were related to the pin assignments and the choice of our top level entity. What we did to solve our problems was directly edit our Lab 5 qsf file to include all of our HEX pin assignments. We also changed our top level entity to the large schematic labelled "Lab5." After this, our switches, LED's, and our seven segment display started working and we were able to complete the table of values. What we struggled with the most was trying to debug Ta's modification in our ripple carry adder verilog code. Essentially, we realized that the bug was in this particular line "myfulladd FA1(A[1], B[1], c0, Sout[1], c1);" with the Sout[1] and c1 switched. This caused the majority of our sums to be off by 2. Going forward, we believe that our pin assignments and verilog codes are very essential to every project and require careful and close attention.