Lab #2 – Aaron Gonzalez & Alma Alvarado

CECS 341 – Computer Architecture and Organization

Aaron Gonzalez, Student ID# 017718387

Alma Alvarado, Student ID# 015087681

Professor: Jose Aceves

California State University, Long Beach

College of Engineering

1250 Bellflower Blvd, Long Beach, CA 90840

September 15, 2020

Goal/Objective:

Part one of the lab requires us to develop two circuits using behavioral Verilog. In part two we need to design and develop a simple 4-bit adder/subtractor.

Technical Description/Steps:

For both part one/two the software used to design our program were Vivado and Verilog.

Part 1:

Circuit 1 - The steps we took to design circuit 1 included designing the design source and then the test bench.

For our design source we had 3 inputs: A, B, and C. Our output was F1. We then used an always block that stated that under the conditions that F1 is equal (A and B) or (not A and C) or (A and not B and not C) then run through this block.

The test bench required us to state the inputs of the unit under test(UUT) which again were A, B , and C. The output of the UUT is wire F1. The test bench requires a lot of instantiation so we also had to instantiate the unit under test and the test bench itself. To instantiate the unit under test we just had to write .A(A), .B(B), .C(C), and .F1(F1). We needed a for loop so we also had to initialize an integer i to use in our loop. The for loop stated that for integer i = 0 and if i is less than 8 then {A, B, C} equals i then display A, B, C, and F1. Finally increment i + 1 until i = 8.

Circuit 2 - The steps we took to design circuit 2 were very similar to circuit one with the exception of the always block. This design source had 3 inputs: A, B, and C. The output for this circuit was F2. We then used an always block that stated that under the conditions that F2 is equal to not A or B and A or not B then run through this block.

The test bench required us to state the inputs of the unit under test(UUT) which again were A, B , and C. The output of the UUT is wire F2. To instantiate the unit under test we just had to write .A(A), .B(B), .C(C), and .F2(F2). We needed a for loop so we also had to initialize an integer i to use in our loop. The for loop stated that for integer i = 0 and if i is less than 4 then {A, B} equals i then display A, B, and F2. Finally increment i + 1 until i = 4.

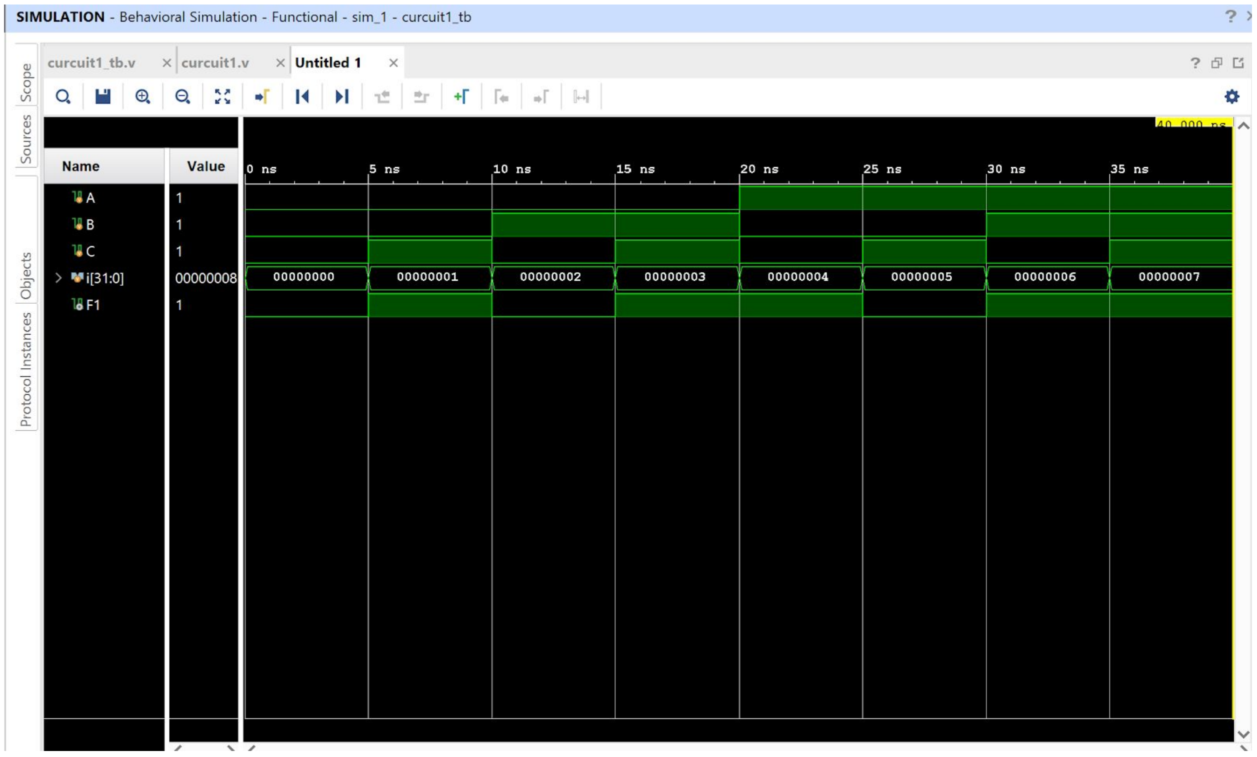
Part 2: The second part of the project required us to design and develop a simple 4-bit adder/subtractor. The code was based on a schematic that was given to us and we needed to model it. The first thing we did was add the Full Adder given to us from the first project and then create four instances of that full adder. We could have used either an xor gate or a mux to follow the schematic either one would work. We decided to go ahead and used an xor gate instead of the mux. We then proceeded to make out inputs [3:0] A, [3:0] B, which are both 4 bits and K. Out outputs included overflow, and [3:0] sum which again is also 4 bits. A localparam ZERO was also instantiated along with all the wires from the schematics. Our wires included C0, C1, C2, S0, S1, S2, S3, cout, Y0, Y1, Y2, and Y3. We then assigned all the Y wires to our B input. For example Y0 = B[0]^K, Y1 = B[1]^K, Y2 = B[2]^K. and Y3= B[3]^K. We then assigned overflow equal to cout^C2 while also assigning sum = {S3, S2, S1, S0}. Finally the last part of the design source was to create the four instances of the full adder with all our wires and inputs.

For our test bench we did several different test cases to make sure that we had the correct output and that our schematic correctly modeled the schematic that was given to us in the assignment. Our test cases had to be in the range between 7 to -8.

Results:

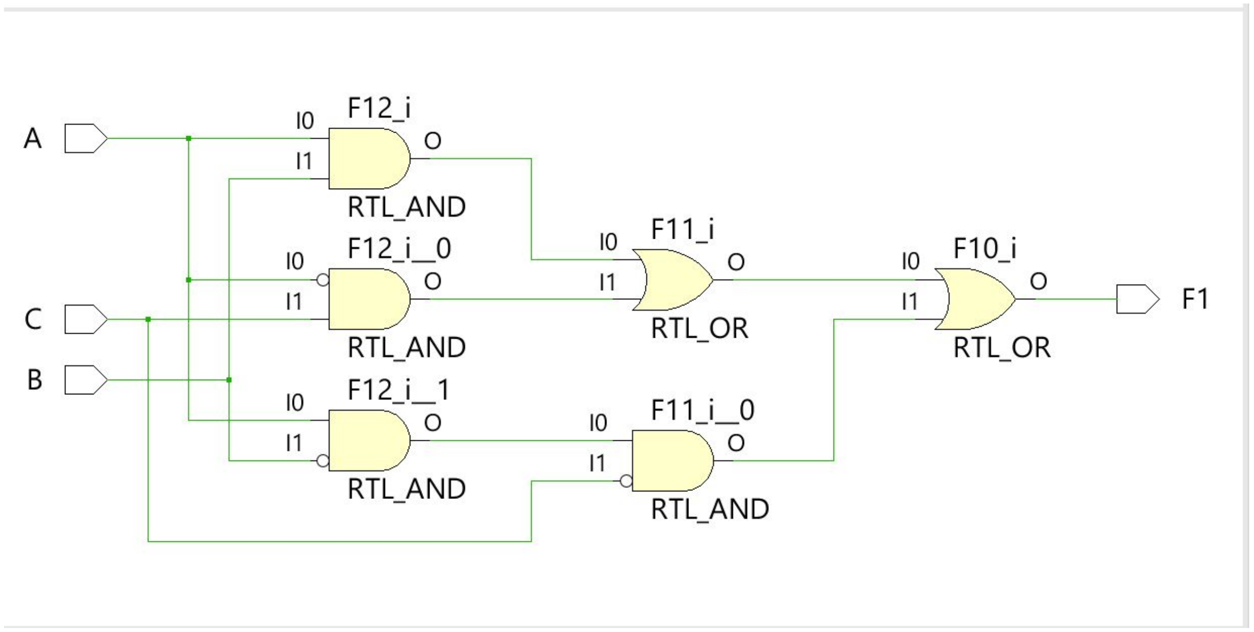
Our results for circuit one came out as they should following the truth table that was presented to us. We can see that the increment follows how we want it to. The truth table statement that we were asked to follow was: F1(A, B, C) = A.B + A’.C + A.B’,C’. When you scroll down to the second image you can see that our console output is the correct output. When A = 0 , B = 0, C= 0 then F1 = 0. This is correct for all the following lines. We can also see that our waveform is following the correct increments.

Circuit 1: WaveForm



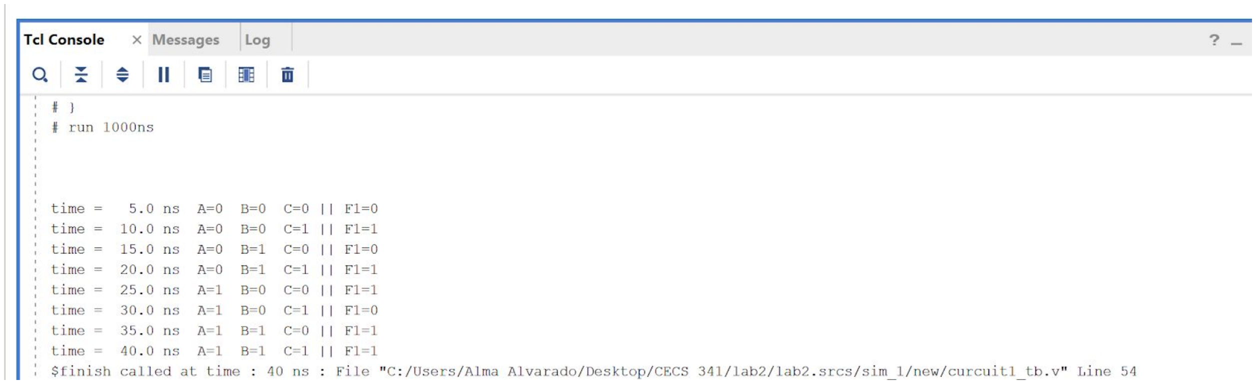
Circuit 1: Schematic

Our schematic is showing the correct inputs that were declared: A, B, and C. We can see that they are correctly going into the & gate and if we follow we can see that the correct inputs are outputting the correct truth value. They then follow to go into the or gates giving us the F1 outputs.



Circuit 1: Console Output:

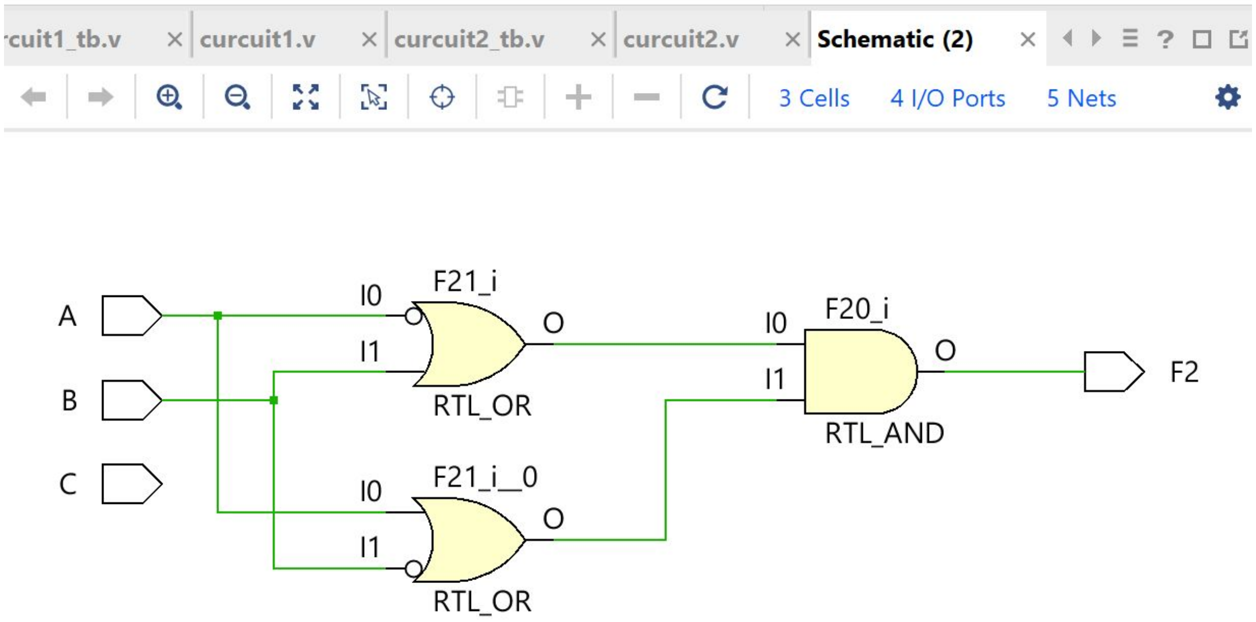
The console output correctly follows the schematic listed above and the truth table. Therefore our truth table is also correct.

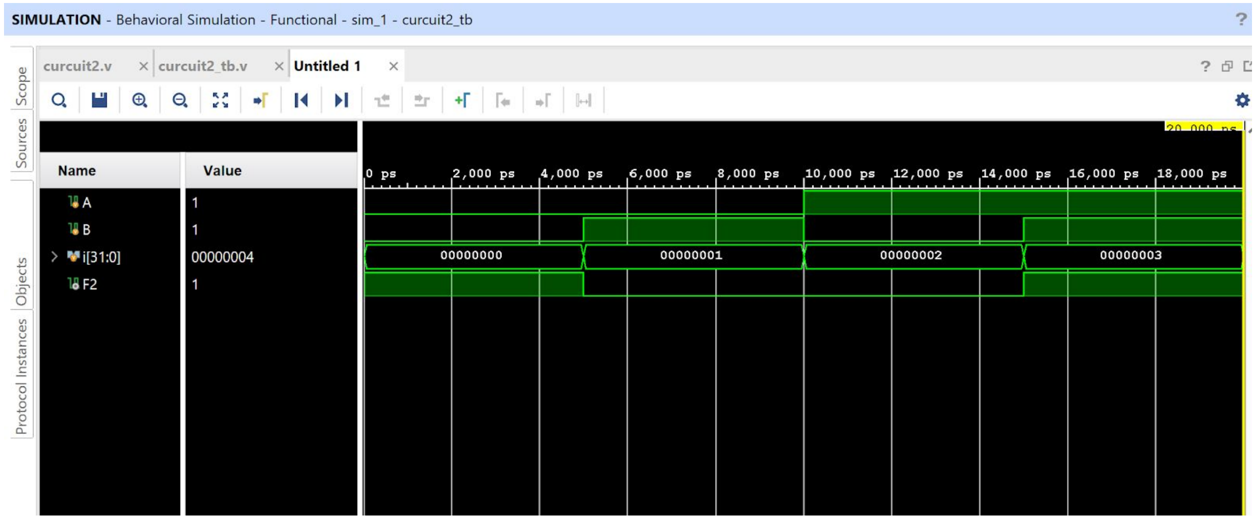


```
# }
# run 1000ns



time =   5.0 ns  A=0  B=0  C=0 || F1=0
time =  10.0 ns  A=0  B=0  C=1 || F1=1
time =  15.0 ns  A=0  B=1  C=0 || F1=0
time =  20.0 ns  A=0  B=1  C=1 || F1=1
time =  25.0 ns  A=1  B=0  C=0 || F1=1
time =  30.0 ns  A=1  B=0  C=1 || F1=0
time =  35.0 ns  A=1  B=1  C=0 || F1=1
time =  40.0 ns  A=1  B=1  C=1 || F1=1
$finish called at time : 40 ns : File "C:/Users/Alma Alvarado/Desktop/CECS 341/lab2/lab2.srcs/sim_1/new/curcuit1_tb.v" Line 54
```

Circuit 2: Schematics & Waveform

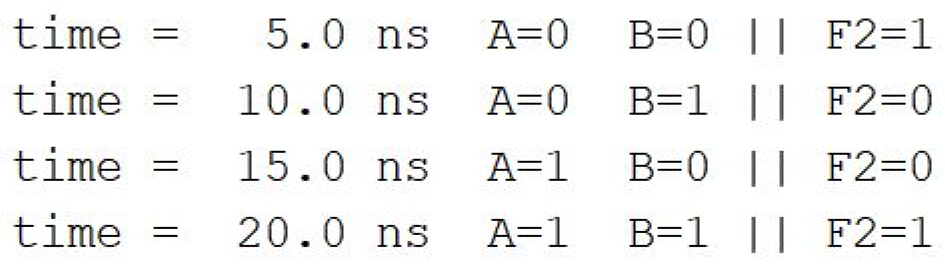
Our schematic is showing the correct inputs that were declared: A, B, and C. We can see that they are correctly going into the or gate and if we follow we can see that the correct inputs are outputting the correct truth value. They then follow to go into the and gates giving us the F2 outputs. We can also observe that this schematic is less complex than the first circuit we were modeling.





Circuit 2: Output

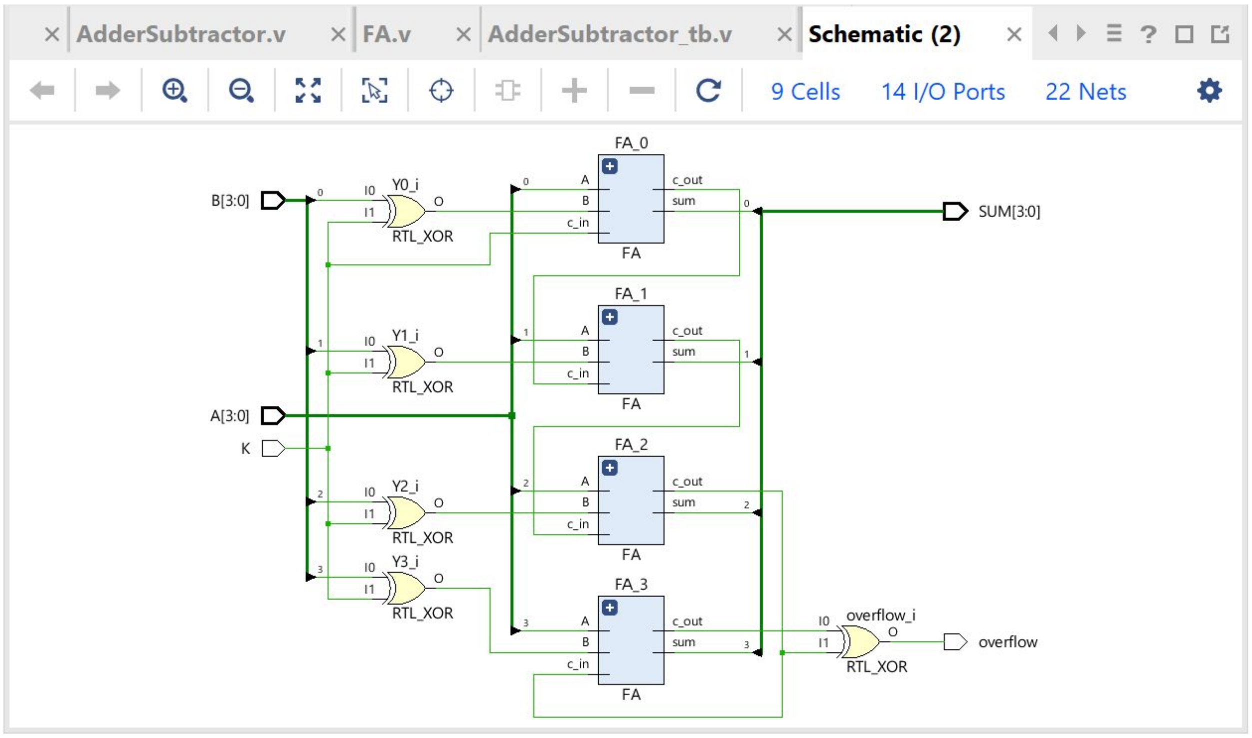
Again if we observe the schematic above this image we can see that the truth table is following the schematic of circuit 2 and our truth table shows the correct output if we follow the design above.



```
time =   5.0 ns  A=0  B=0 || F2=1
time =  10.0 ns  A=0  B=1 || F2=0
time =  15.0 ns  A=1  B=0 || F2=0
time =  20.0 ns  A=1  B=1 || F2=1
```

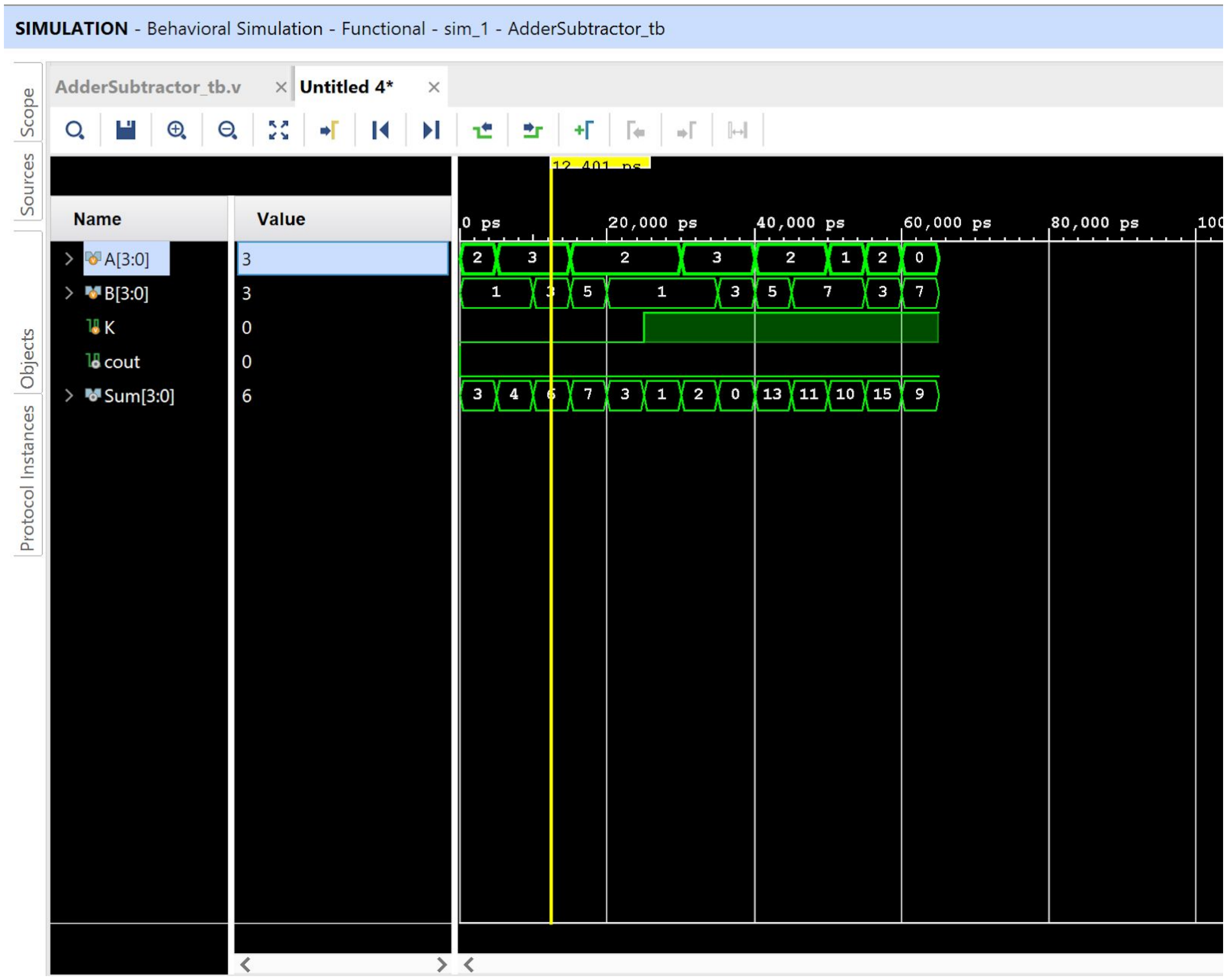
Part 2: Schematic

The schematic for the AdderSubtractor design is much more complex than the previous two circuits. We can see that for this schematic we have four instances of the Full Adder and we also have inputs that are 4 bits now. We also have xor gates and two different outputs like sum and overflow. Our results successfully model what a four bit adder subtractor should look like. We can also see that for every full adder we have the inputs A, B, c_in and the outputs are c_out and sum which is exactly what a full adder should look like. Although the full adder was given to us we used it to correctly model the rest. We also had an overflow output just in case any of our test cases needed it but after we got more information we realized we only needed a range of numbers from 7 to -8.


AdderSubtractor.v
FA.v
AdderSubtractor_tb.v
Schematic (2)
9 Cells
14 I/O Ports
22 Nets
B[3:0]
A[3:0]
K
Y0_i
Y1_i
Y2_i
Y3_i
RTL_XOR
I0
I1
O
FA_0
FA_1
FA_2
FA_3
A
B
c_in
c_out
sum
FA
SUM[3:0]
overflow_i
overflow

Part 2: Unsigned Radix Waveform

The unsigned radix waveform was used for adding. Our numbers show a correct output and it also follows the range correctly.

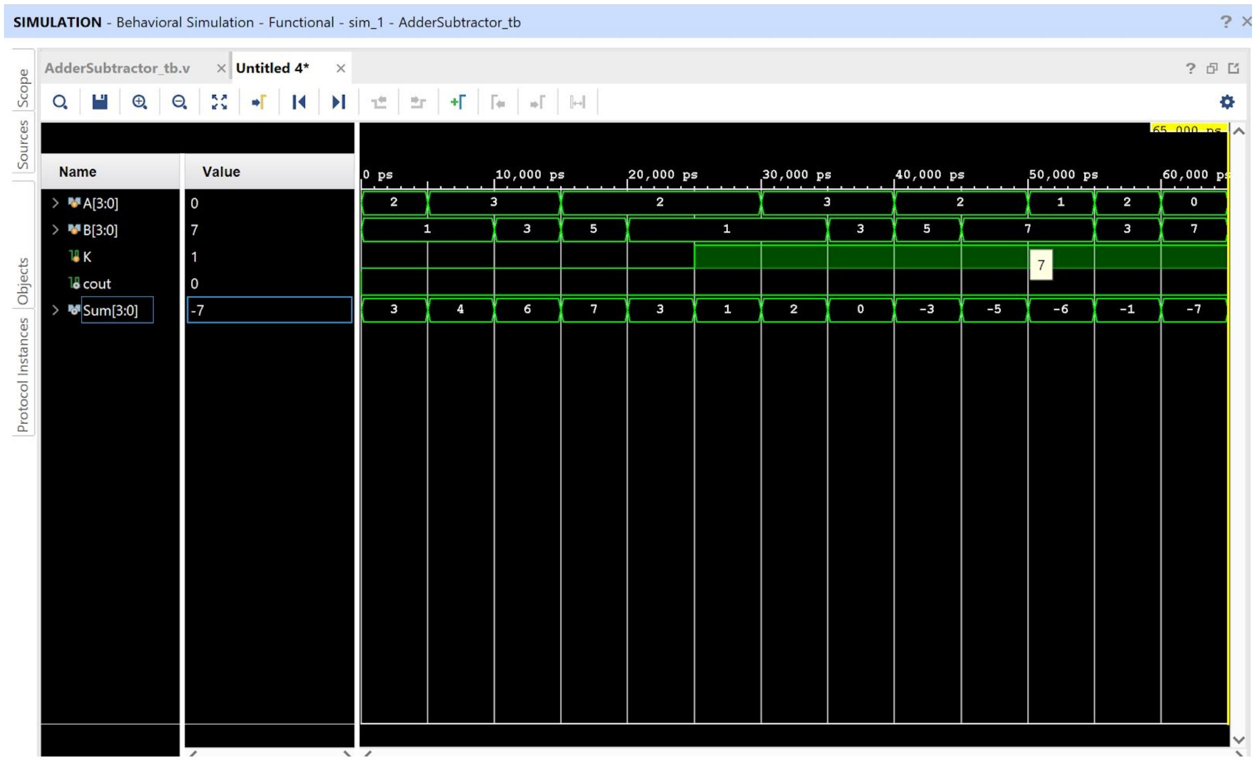


Name	Value
A[3:0]	3
B[3:0]	3
K	0
cout	0
Sum[3:0]	6

```
A[3:0]    2 | 3 | 2 | 3 | 2 | 1 | 2 | 0
B[3:0]    1 | 3 | 5 | 1 | 3 | 5 | 7 | 3 | 7
Sum[3:0]  3 | 4 | 6 | 7 | 3 | 1 | 2 | 0 | 13 | 11 | 10 | 15 | 9
```

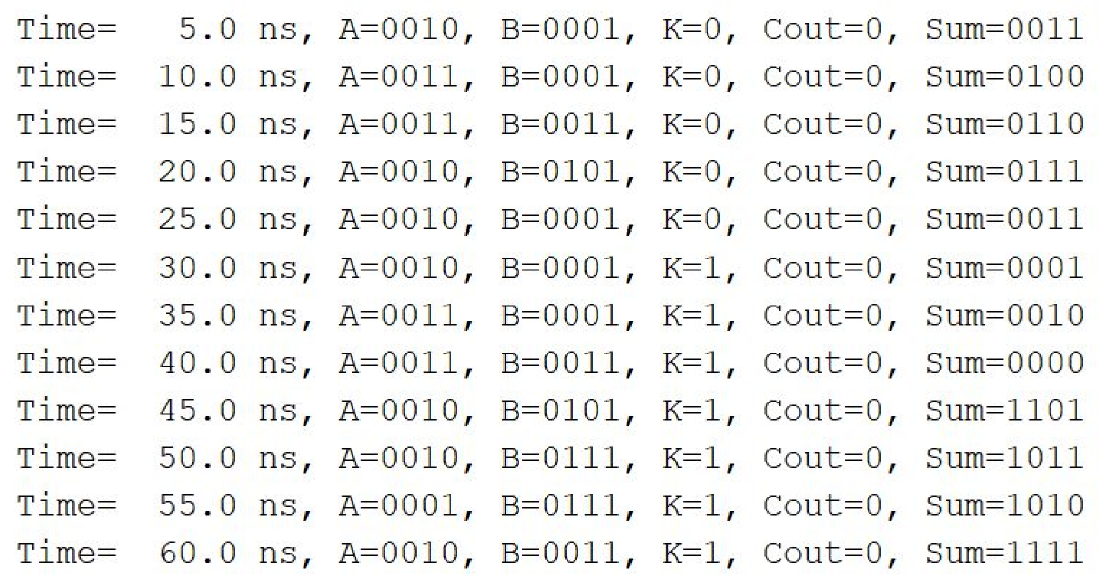
Part 2: Signed Radix Waveform

The signed radix waveform was used for subtracting. Our numbers show a correct output and we can also see that we get the negative numbers we should. The negative numbers also follow the range correctly which at first we were sure it was but again there was a range to follow.



Part 2: Console Output

This console output shows that we were dealing with 4 bits instead of two and our sum follows the correct binary output that follows a 4 bit binary adder/subtractor. We developed the required 1-bit logical components to make the necessary instantiations for our full design. For this specific testbench we did not use a loop because we wanted to choose different numbers, and it would have been difficult to acquire unique outputs with a sequential while loop.



```
Time=   5.0 ns, A=0010, B=0001, K=0, Cout=0, Sum=0011
Time=  10.0 ns, A=0011, B=0001, K=0, Cout=0, Sum=0100
Time=  15.0 ns, A=0011, B=0011, K=0, Cout=0, Sum=0110
Time=  20.0 ns, A=0010, B=0101, K=0, Cout=0, Sum=0111
Time=  25.0 ns, A=0010, B=0001, K=0, Cout=0, Sum=0011
Time=  30.0 ns, A=0010, B=0001, K=1, Cout=0, Sum=0001
Time=  35.0 ns, A=0011, B=0001, K=1, Cout=0, Sum=0010
Time=  40.0 ns, A=0011, B=0011, K=1, Cout=0, Sum=0000
Time=  45.0 ns, A=0010, B=0101, K=1, Cout=0, Sum=1101
Time=  50.0 ns, A=0010, B=0111, K=1, Cout=0, Sum=1011
Time=  55.0 ns, A=0001, B=0111, K=1, Cout=0, Sum=1010
Time=  60.0 ns, A=0010, B=0011, K=1, Cout=0, Sum=1111
```

Conclusion:

For this lab we learned how to utilize a full adder to design an adder and subtractor while focusing on 2-bits. The XOR gate would be the key to implement our subtracting because we would need to implement the two’s complements then add the bits together. Depending how many full adders we use, we can concatenate the S[x] outputs to get our final answer, but we must also keep in mind the sign bit. For a four bit, we must keep in mind that we would only have to use 3-bits because our fourth bit would be our sign bit. After we got clarification about this, we were able to do our test cases while not worrying about numbers that would not be in the range between 7 and -8. This lab would also keep us mindful of making sure our inputs and outputs would be precise, because if we made a slight mistake on passing on the wrong carry to the next full adder then our final output would be incorrect. This lab was able to show us how to properly use prior schematics and expand on that to make something greater and efficient.