ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	3
EXPERIMENT DATE	:	31.03.2021
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G1

GROUP MEMBERS:

150170908	:	MUHAMMAD RIZA FAIRUZZABADI
070170364	:	AİŞE HÜMEYRA BOZ

SPRING 2021

Contents

1 INTRODUCTION

In this project, we were required to complete two distinct tasks, preliminary and experiment. First in preliminary part, we recalled some Digital Circuits topics of signed and unsigned addition and substraction, alongside the concepts of carry, borrow, and overflow. In the experiment part, we implemented half adder and full adder modules alongside the logic gate modules. Then we tested them with the given values.

2 MATERIALS AND METHODS

2.1 MATERIALS

Tools Used

- Vivado Design Suite - Xilinx
- Latex (overleaf.com)
- Logisim

Firstly we recalled the aforementioned topics and noted the important characteristics we have to keep in mind in the implementation of the modules. We then implement and program the logic circuits alongside their NOT,AND,OR gates modules in Vivado Design Suite, lastly we used overleaf.com to prepare the report document in LaTex.

2.2 PRELIMINARY

In the preliminary part, we recalled and revised what we learnt in Digital Circuits course. There are some features we have to keep in mind in the implementation of the adder and substractor circuits:

- Substraction is enabled by 2's complement method for both signed and unsigned operations.
- Addition is conducted the same way for both of them, but the interpretation differs.
- For unsigned addition operations, (n+1)th bit is called carry and it can take 0 or 1.
- For signed addition operations, (n+1)th bit is always ignored.
- Overflow occurs when the result cannot be represented in total bit it is being calculated, resulting in false positive/negative conditions. It occurs in 4 cases: $pos + pos = neg\ neg + neg = pos|\ pos - neg = neg\ neg - pos = pos$

- Borrow occurs when there is no carry with unsigned numbers in substraction operations.
- For both borrow and overflow situations, the result cannot be represented.

2.3 PART 1

In the first part, we were asked to design AND, OR, NOT, XOR modules. Since we already designed the first three ones, we only designed the XOR modules for this experiment. XOR gate's output is only 1, if and only if either one of the inputs is 1.We will use these modules in the following part of the experiment.

2.4 PART 2

In the second part of the experiment, we were tasked to design and implement a half adder which adds two 1-bit numbers without a carry input. The result and carry is represented in two different outputs. In design part, we instantiated XOR module and AND module because for result, addition operation can be made only with that gate and for carry, if both inputs are 1 then the carry becomes 1 only in that condition. Then we chose A and B as inputs, S and Cout as outputs. The name of the module HalfAdder. The elaborated design schematic is as shown in Figure 1, the result of the simulation is also shown in Figure 2. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation.

A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

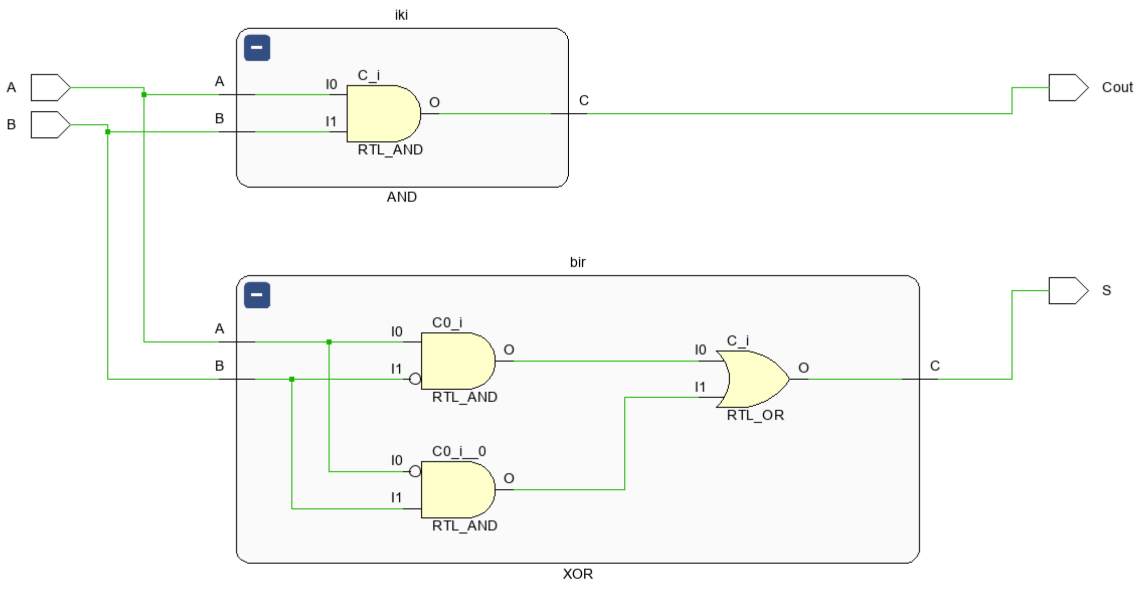


Figure 1: Elaborated Design Schematic of Half Adder

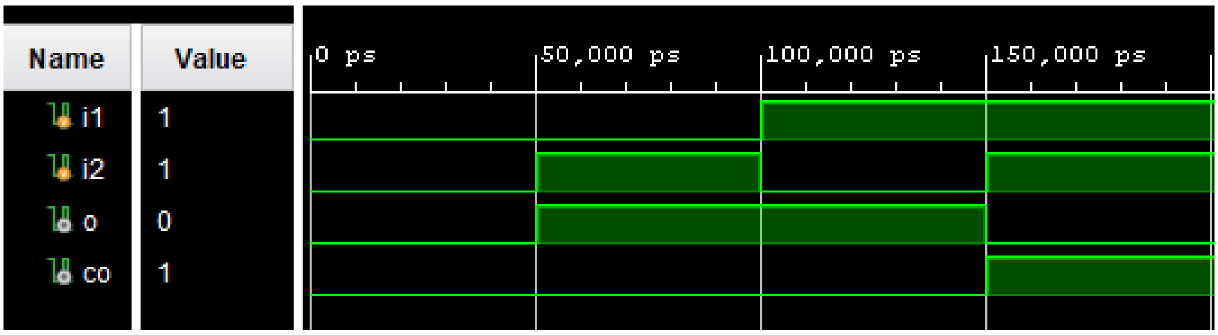


Figure 2: Simulation of Half Adder

2.5 PART 3

In the third part of the experiment, we were tasked to implement 1-Bit Full Adder by using OR module half-adder we implemented before. The main difference between half adder and full adder is that in Full Adder, there exists Carry-In input to take into consideration. For that reason, instead of one half adder, we had to use one extra half adder to add the Carry In input value with the first half adder's output. Then lastly, we took both outputs of the half adders as inputs to an OR, to check if there is any carry in any one of them. In its implementation in Verilog, we set A, B and Cin as input wires, and S and Cout as output wires. We then instantiated Half Adder twice, and one OR to implement the calculation we explained above. In the simulation.v part, we created one simulation module Sim1 where we tested/instantiated the module with every possible value 0/1 for A and B in 50 ns delay. The name of the module is FullAdder.

The elaborated design schematic of the module is as shown in Figure 3, the result of the simulation is also shown in Figure 4. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation.

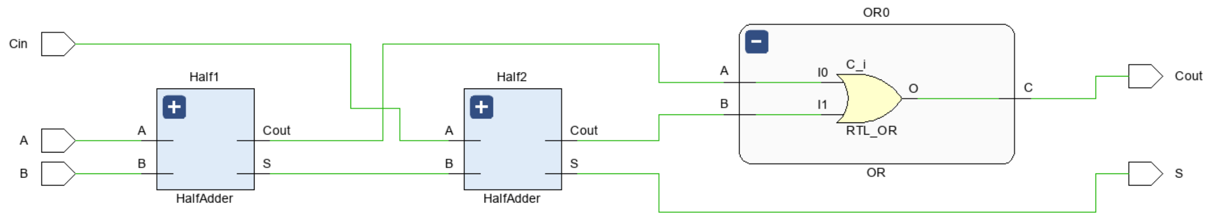


Figure 3: Elaborated Design Schematic of 1-Bit Full Adder

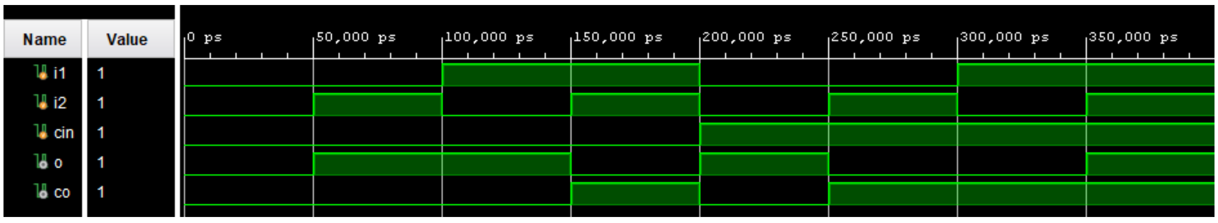


Figure 4: Simulation of 1-Bit Full Adder

A	B	Cin	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

2.6 PART 4

In the fourth part of the experiment, we designed a 4-Bit full adder using four FullAdder modules. Both inputs and outputs are now 4-Bits and there is an input named Cin and an output named Cout. Since there are wires between four full adders, we named them separately. In four FourBitFA module we designed, full adders are placed one after another that enables one's carry output to become another's carry input so that addition can be applied. In the first instantiating of FullAdder module, we took only 0th index between brackets of inputs and formed 0th index of the output. Cin is simulated as 0 in this part, because the operation is always addition. Then we practiced the same process over the other three full adders and reached to final output and Cout output. For simulation, we set three inputs namely i1,i2,cin and two outputs o, co. We instantiated FourBitFA module and set our variables inside. The elaborated design schematic of the module is as shown in Figure 5, the result of the simulation is also shown in Figure 6. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation. It is worth mentioning that S here is the output, not Signed/Unsigned Flag.

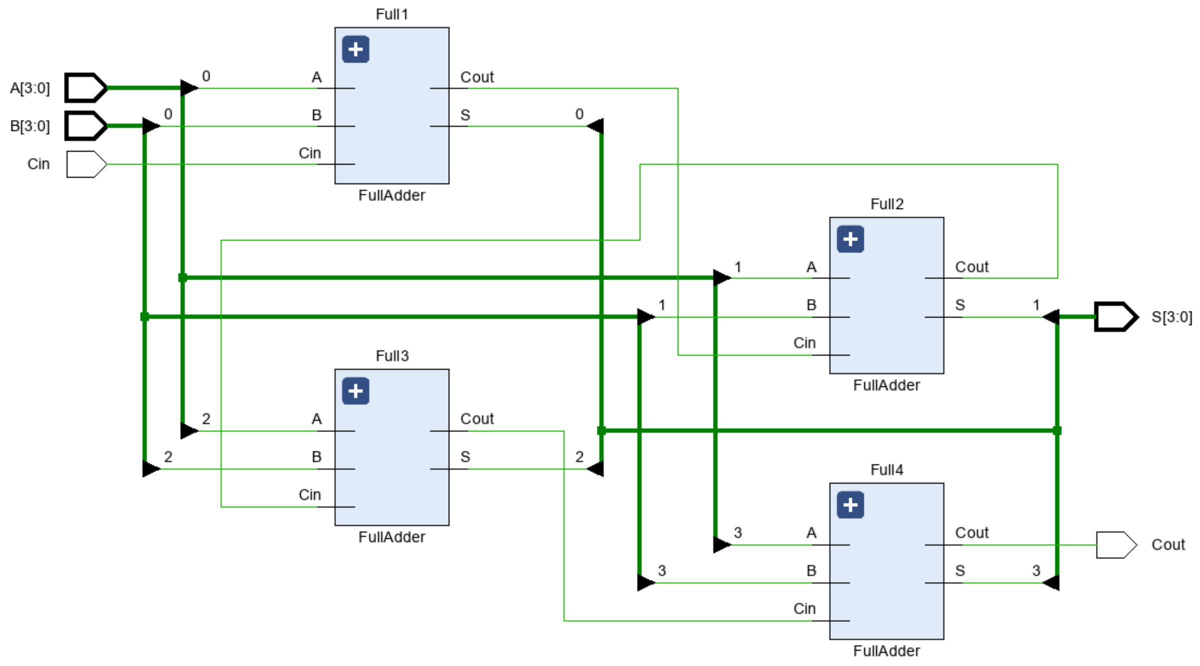


Figure 5: Elaborated Design Schematic of 4-Bit Full Adder

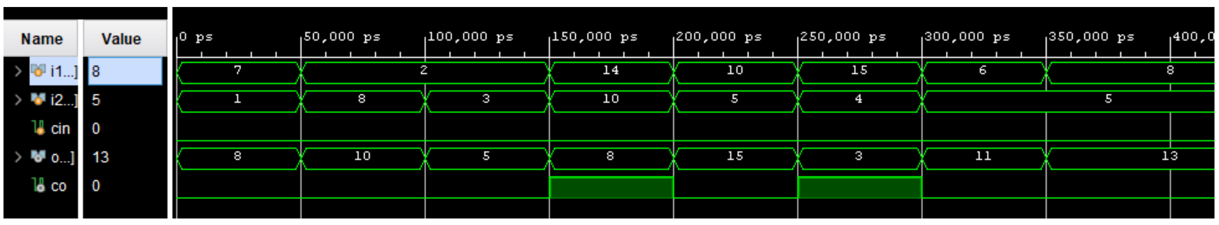


Figure 6: Simulation of 4-Bit Full Adder

A	B	Cin	S	Cout
7	1	0	8	0
2	8	0	10	0
2	3	0	5	0
14	10	0	24	1
10	5	0	15	0
15	4	0	19	1
6	5	0	11	0
8	5	0	13	0

2.7 PART 5

In the fifth part of the experiment, we were tasked to implement a 16-Bit Full Adder by using 4-Bit Full Adders modules we designed previously. Systematically we did the exact same steps as when we implemented the 4-Bit Full Adder. The main difference is, we instantiated FourBitFA instead of the previous FullAdder, and instead of taking and storing one bit from A/B to S, we took 4 bits at a time inside the square brackets in the format of [ending index:starting index]. ([3:0], [7:4], [11:8], [15:12], respectively.). To test the module, we created a new simulation module of Sim4, where we set the module's parameters as reg and wire as usual. Inside the initial block, we put the specific values we were tasked to simulate. The elaborated design schematic of the module is as shown in Figure 7, the result of the simulation is also shown in Figure 8. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation.

A	B	Cin	S	Cout
29	3	0	32	0
21	83	0	104	0
16800	16900	0	33700	0
65534	65100	0	130634	1
202	97	0	299	0
44	19	0	63	0
644	255	0	899	0
86	572	0	658	0

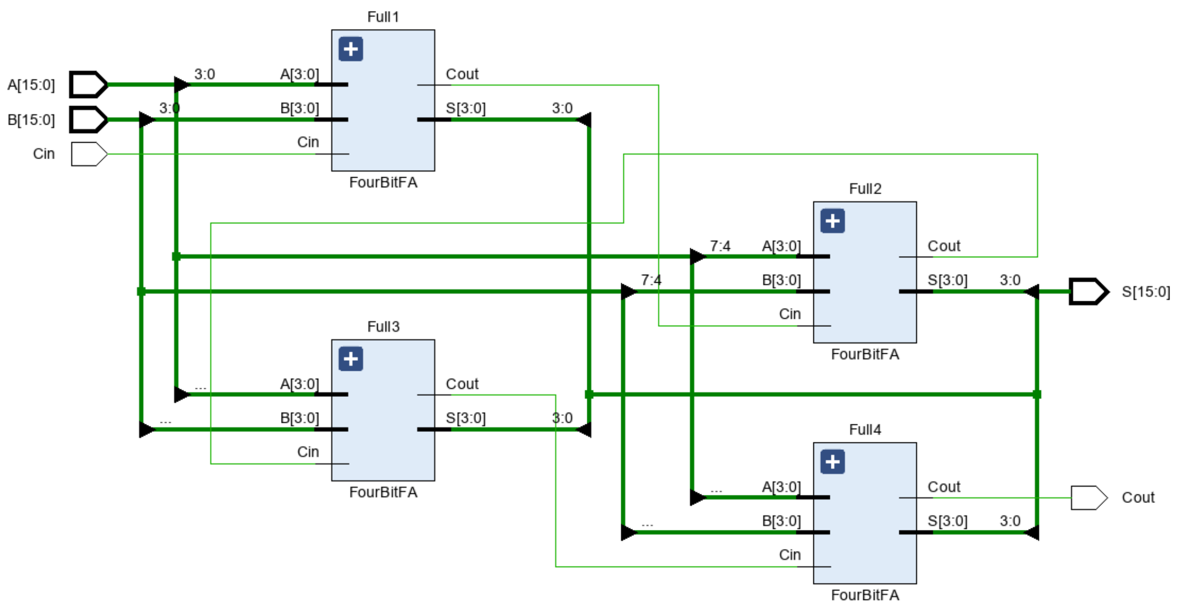
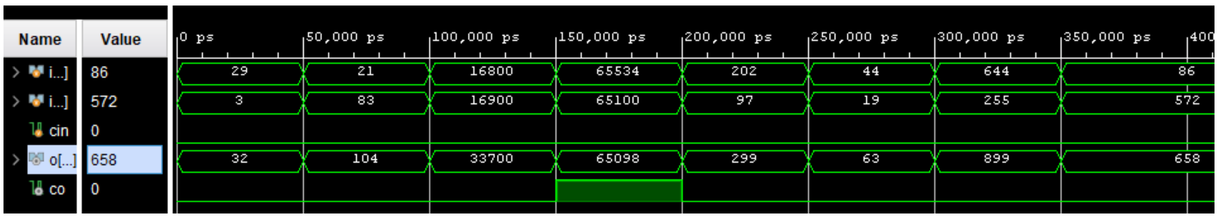


Figure 7: Elaborated Design Schematic of 16-Bit Full Adder



Name	Value	0 ps	50,000 ps	100,000 ps	150,000 ps	200,000 ps	250,000 ps	300,000 ps	350,000 ps
i...]	86	29	21	16800	65534	202	44	644	86
i...]	572	3	83	16900	65100	97	19	255	572
cin	0								
o[...]	658	32	104	33700	65098	299	63	899	658
co	0								

Figure 8: Simulation of 16-Bit Full Adder

2.8 PART 6

In the 6th part of this experiment, we were asked to implement a 16-Bit Full Adder-Subtractor by using 16-Bit Full Adder, NOT, XOR, AND and OR modules we designed previously. There are some factors we had to put into consideration in this part, firstly, since we already have 16-Bit Adder, we just have to instantiate the adder module once, and we didn't have to divide the bits in the calculation. Before going to that however, the values of B, have to first go through XOR gates since they have to change condition according to the calculation we do, which is either adding or substracting. In case of adding, the values would go through XOR to input the same, but when it comes to subtracting, the Carry In value 1 that determines the subtracting condition go through that XOR gates and flip the values, resulting in 2'Complement of B. To accomplish this, instantiate XOR gate 16 times using for loop, which takes B's value as input and stores the result of the calculation 16-Bit O1 variable bit by bit, using the square bracketes technique like we implemented previously. After then we instantiated the SixteenBitFA module normally, with A, and the O1 (the XOR gates input) as inputs, and Cin, O and Cout as outputs. Those are the implementations of Adder/Substracter calculation. Then we also had to check overflow and borrow conditions to check the validity of the calculation. To implement that we added overflag, borrowflag and validity output wires. As we have recalled in the preliminary part, overflow depends on the validity of four cases. The main characteristic of those cases is that all occurs when carry exists when the output value is positive (MSB being 0), or when carry does not exists when the output value is negative (MSB being 1). So it's 'either 1/0 or 0/1' scenario between Carry (Cout) and MSB. For that reason to check the overflow we take these two (Cout and O[15]) to a XOR gate. Then, since overflow only occurs in Signed operations, we then take the output of that XOR gate to an AND gate, alongside the S input wire, so overflow only occurs when S = 1 (Signed). Secondly, to check for borrow case the factors we have to put into consideration is that it depends on the existence of Carry (Cout) (Borrow exists when Carry doesn't, and vice versa), that it only occurs in unsigned calculation, and subtraction. To implement these factors into code we could use one AND, however since we only have two input ANDs, we first take Cout and S values (negated) to one AND, and then we take the output of that to one AND alongside Cin. Lastly, we check if whether one or more of these two's (overflag and borrowflag) value is 1 with an OR module to check the validity of the Adder/Substractor's output O. To test this AdderSubstractor we first set the module's parameters as reg or wire data types, as usual, instantiated the module and inside the initial block we put values to A, B, Cin, and S according to the expressions we were given in the Experiment's PDF. The elaborated design schematic of the module is as shown in Figure 9, the result of the simulation is also shown in Figure 10 and 11. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation. Since simulation in Vivado only shows on S condition at a time (Signed or Unsigned), we put two distinct simulation results as figure.

A	B	Cin	S	O	Cout	Overflag	Borrowflag	Validity
29	3	1	0	26	1	0	0	1
21	83	0	0	104	0	0	0	1
16800	16900	0	1	-31836	0	1	0	0
103	145	1	0	65494	0	0	1	0
202	97	0	0	299	0	0	0	1
32400	32200	0	1	-936	0	1	0	0
6478	2585	1	0	3893	1	0	0	1
8	52	1	0	65492	0	0	1	0

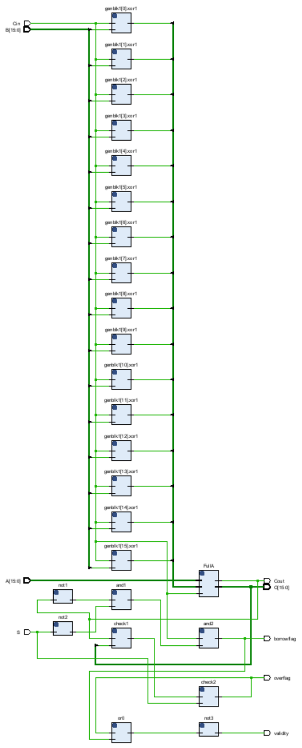


Figure 9: Elaborated Design Schematic of 16-Bit Adder/Subtractor

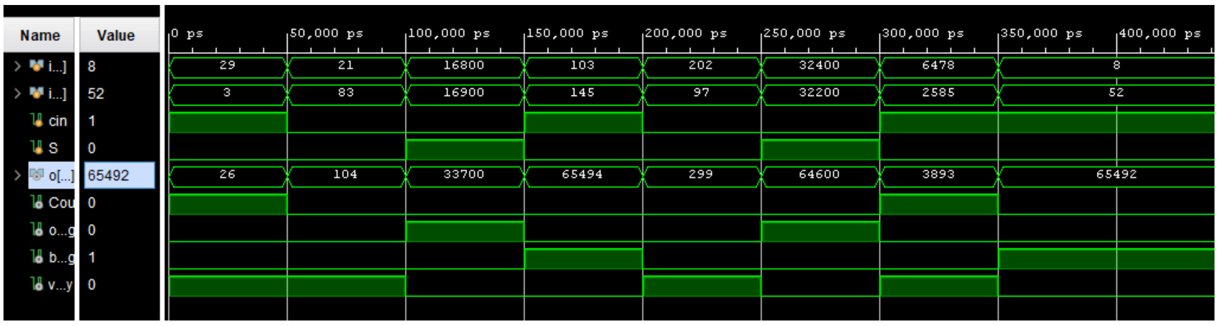


Figure 10: Simulation of 16-Bit Adder/Subtractor - Unsigned

2.9 PART 7

In the last part of the experiment, we created 3A - 2B operation using 16-Bit Full Adders and AdderSubstractor modules. In Part7 module, we took A,B inputs and O output as 16 bits. We also set input S, it works as same as in part 6 and also Cout, overflag, borrowflag, validity outputs. We first add A and A by utilizing SixteenBitFA then, with the output of this operation we again used SixteenBitFA module and reached to 3A. Likewise, we added two B's in the same way. Finally, we instantiated AdderSubstractor module and placed 3A and 2B as inputs 1 as for Cin because this is a substraction and give the rest their names in order to simulate them later. In simulation results, numbers varied when they are signed or unsigned. Overflow, borrow and validity situations are observed

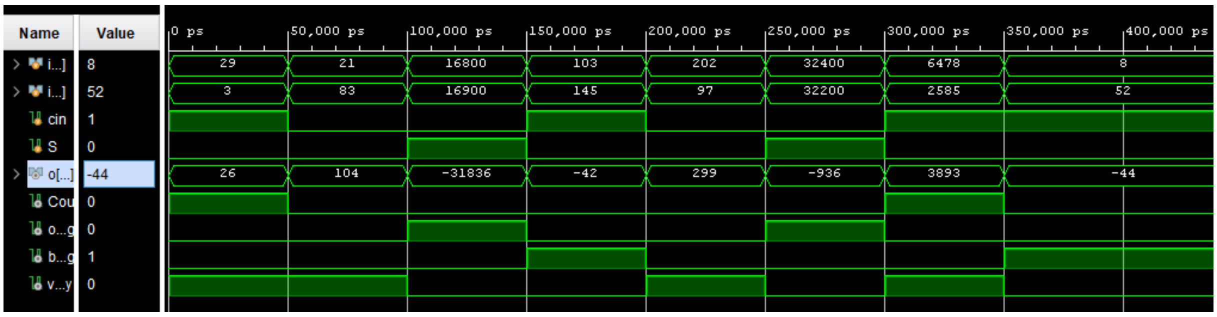


Figure 11: Simulation of 16-Bit Adder/Subtractor - Signed

in detail. We simulated the results for the given inputs for A and B. The elaborated design schematic of the module is as shown in Figure 12, the result of the simulation is also shown in Figure 1314. The truth table of the implementation can also be seen below the figures. Comparing the values in Truth Table with the ones in the simulation result, we can see the correctness of our implementation. Since simulation in Vivado only shows on S condition at a time (Signed or Unsigned), we put two distinct simulation results as figure

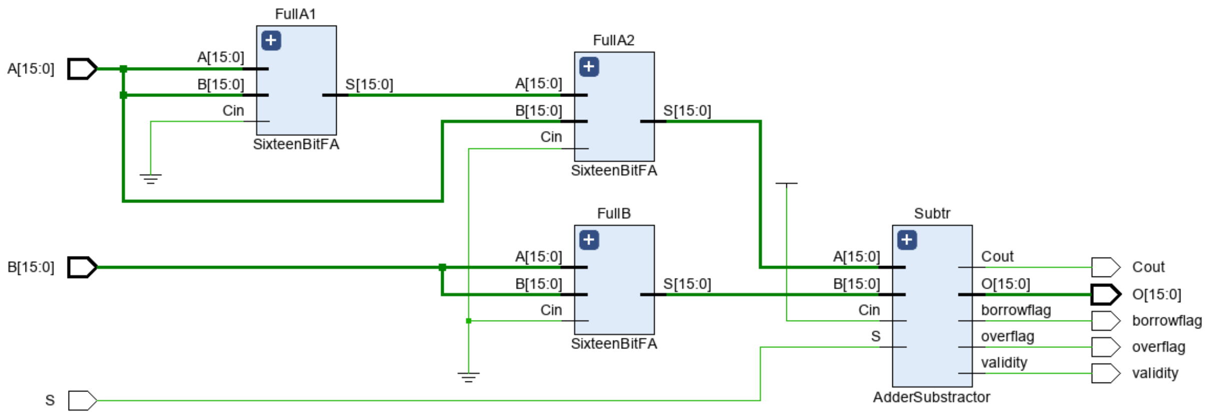


Figure 12: Elaborated Design Schematic of Part7

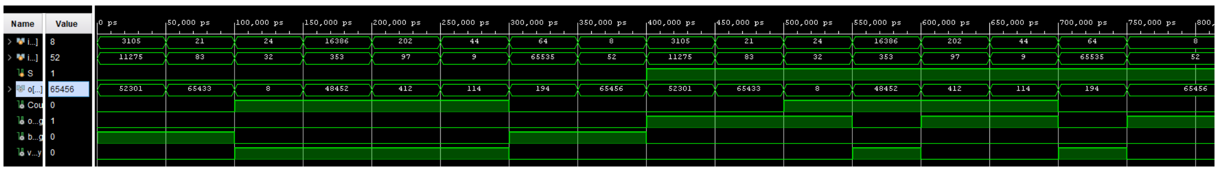


Figure 13: Simulation of 16-Bit Part 7 - Unsigned

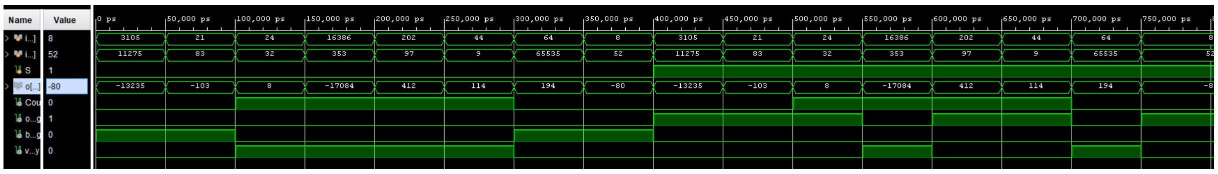


Figure 14: Simulation of 16-Bit Part 7 - Signed

A	B	S	O	Cout	Overflag	Borrowflag	Validity
3105	11275	0	52301	0	0	1	0
21	83	0	65433	0	0	1	0
24	32	0	8	1	0	0	1
16386	353	0	48452	1	0	0	1
202	97	0	412	1	0	0	1
44	9	0	114	1	0	0	1
64	65535	0	194	0	0	1	0
8	52	0	65454	0	0	1	0
3105	11275	1	-13235	0	1	0	0
21	83	1	-103	0	1	0	0
24	32	1	8	1	1	0	0
16386	353	1	-17084	1	0	0	1
202	97	1	412	1	1	0	0
44	9	1	114	1	1	0	0
64	65535	1	194	0	0	0	1
8	52	1	-8	0	1	0	0

3 RESULTS

We completed every task we were asked in the experiment. At first we drew the circuits in Logism, then we got to see their results clearly through simulations. Since we draw and design the expressions in Logisim, it was not hard to validate their correctness through simulations. We added required tables, images, circuits to the specific places materials methods part of the experiment. Lastly, we supplemented our images namely elaborated design, simulations and truth tables to the report for each part.

4 DISCUSSION

In preliminary part, we recalled addition and substraction operations with signed and unsigned numbers. We designed half adders using XOR and AND gates. We implemented full adder with half adder modules and OR gates. Then, we created 4-bit full adder module using four full adders. Likewise, we implemented 16-bit full adder with four 4-bit adders. Also, we designed adder-substractor module which enabled both adittion and substraction operations. The most challenging part was this since there were flags which indicates overflow, borrow and validity conditions. In for loop, we managed to set 1-bit and 16-bit inputs to be in the same gate for i times. Lastly, using the previous module we formed 3A - 2B operation, with same pattern.It was inconvenient to set as top in each time we need our modules. So, we uncommented the part we used during the experiment.

5 CONCLUSION

We successfully implemented the modules and simulated them, but apart from last weeks we learnt how to implement adders which are more than one bits using only allowed operators. We learnt how to implement more than one bit adders specifically 4-bits and 16-bits. In 16-bit adder, we learnt how to call bits of a number. Also in order to implement adder-substractor in part 6, we learnt for loop's detail. Lastly, it was a great chance for us to implement flags as overflow, borrow and validity signs.

REFERENCES

[1] LogicLab. Logic lab. *An example journal*, 22(4):10–16, February 2020.

[2] Overleaf documentation https://tr.overleaf.com/learn.