ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY
HOMEWORK 1

HOMEWORK NO	:	1
HOMEWORK DATE	:	20.03.2023
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G8

GROUP MEMBERS:

150200919 : Abdullah Jafar Mansour Shamout

150220762 : Muhammed Yusuf Mermer

SPRING 2023

Contents

1 INTRODUCTION

In this experiment we implemented combinational logic circuits and Full adder using the Verilog. In it we learned how to design and use a module. We found the minimal cost expression using prime implicants method and prime implicants chart then using our basic logic gates that we designed, we implemented the modules. We also recalled the signed and unsigned addition and subtraction using 2's complement, this helped us in implementing adders and subtractors using combinational logic circuits modules. Through out our designs and implementations, we made sure to test all of our modules with various inputs through simulations with different combinations to make sure they work properly.

2 PRELIMINARY

2.1 Question 1.a)

To find with K-Map, at first we find prime implicant for implicants.
Prime implicant is biggest implicants for that group of 1's.

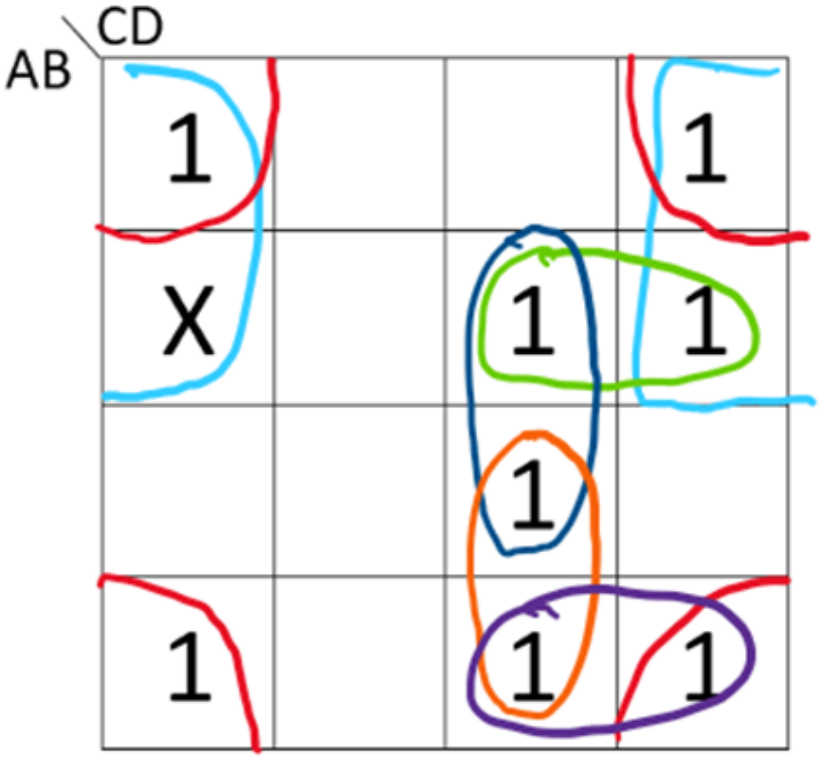

Figure 1: K-Map With All Prime Implicants

Then, to find essential prime implicants, we need to take prime implicants who are just covered by one and only one prime implicants.

After that, make other prime implicants with remaing 1's. Try to prevent collisions and make prime implicant big as possible.

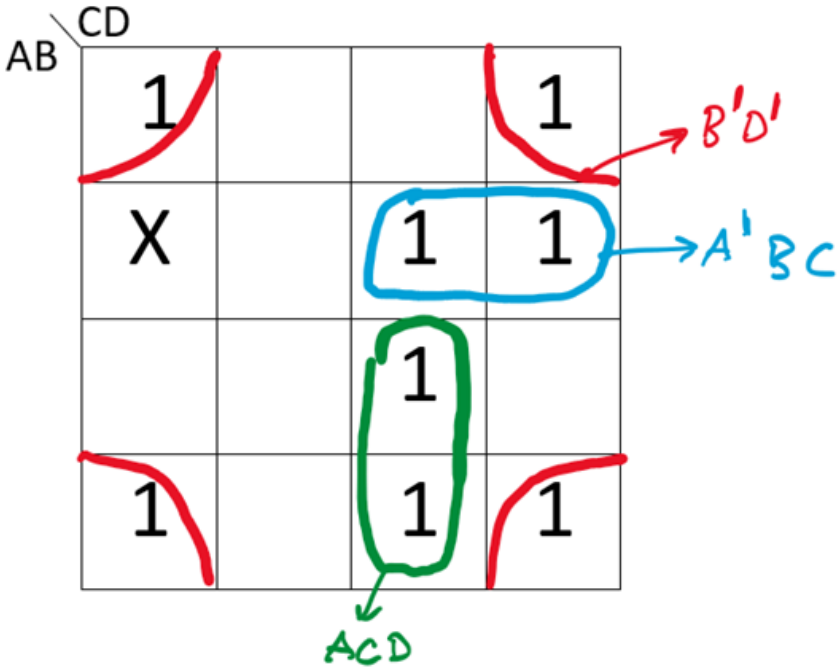

Figure 2: at end of the K-Map

$acd + a'bc + b'd'$ **will be the final result.**

2.2 Question 1.b)

To find with Quine Mccluskey method, at first we need to write all implicants(including don't care condition) to table (like binary digits) as in the figure:

Numbers	a	b	c	d	checked
0	0	0	0	0	✓
2	0	0	1	0	✓
4	0	1	0	0	✓
8	1	0	0	0	✓
6	0	1	1	0	✓
10	1	0	1	0	✓
7	0	1	1	1	✓
11	1	0	1	1	✓
15	1	1	1	1	✓

Figure 3: input combinations

After this operation, we made pairs from different groups which have only one digit difference. And we checked previous rows if they are used in new table:

Group	Numbers	a	b	c	d	checked
0	0,2	0	0	-	0	✓
0	0,4	0	-	0	0	✓
0	0,8	-	0	0	0	✓
1	2,6	0	-	1	0	✓
1	4,6	0	1	-	0	✓
1	2,10	-	0	1	0	✓
1	8,10	1	0	-	0	✓
2	6,7	0	1	1	-	X
2	10,11	1	0	1	-	X
3	7,15	-	1	1	1	X
3	11,15	1	-	1	1	X

Figure 4: Group with 2 points

Do same things for group of 4's:

0	0,2,4,6	0	-	-	0	X
0	0,2,8,10	-	0	-	0	X

Figure 5: Group with 4 points

2.3 Question 1.c)

To find cost, we used unchecked rows. We didn't include the don't care conditions but we used them to calculate costs. As it mentioned in the question, 2 cost for variables and 1 more cost for their inverts used to calculate over all cost.

	0	2	8	6	10	7	11	15	Cost
A				X		X			7
B					X		X		7
C						X		X	6
D							X	X	6
E	X	X		X					6
F	X	X	X		X				6

Figure 6: True Points of The Function

For the simplification, we need to find distinguished points. Distinguished points' column doesn't have any more X's.

	0	2	8	6	10	7	11	15	Cost
A				X		X			7
B					X		X		7
C						X		X	6
D							X	X	6
E	X	X		X					6
F	X	X	X		X				6

Figure 7: Distinguished Points

Then we deleted the row F but we will use it for the overall cost calculation. Not only row F, but we also deleted columns which are covered by the F.

	0	2	8	6	10	7	11	15	Cost
A				X		X			7
B					X		X		7
C						X		X	6
D							X	X	6
E	X	X		X					6
F	X	X	X		X				6

Figure 8: Simplification of Chart

In here, it is better to delete row B as **D < B + one more variable**. Do the same thing for E. row.

	6	7	11	15	Cost
A	X	X			7
B			X		7
C		X		X	6
D			X	X	6
E	X				6

Figure 9: Last Simplification

At the end, we will look distinguished points again. This time both A and D are distinguished points.

	6	7	11	15	Cost
A	X	X			7
C		X		X	6
D			X	X	6

Figure 10: Selecting Last Distinguished Points

Result will be $F + A + D = 6 + 7 + 6 = 19$

2.4 Question 1.d and 1.e)

After designing circuit with AND, OR, NOT gates, to convert equation to NAND gate, we take the NOT of whole equation and also to maintain accuracy, we distributed inverse to inside of the equation. Therefore, OR gate turned into AND gate.

$$((acd)'\&(a'bc)'\&(b'd')')'$$

Prime implicants already in AND form, so without distributing even more we can take their inverses to obtain NAND gates from them.

Circuit designs implemented in Figure 29 and 31.

2.5 Question 1.f)

Multiplexer has 3 selector. We can give a b c variables to this selectors. Then as a inputs of MUX, we can give d, d', 0 or 1 according to a, b, c values in the truth table.

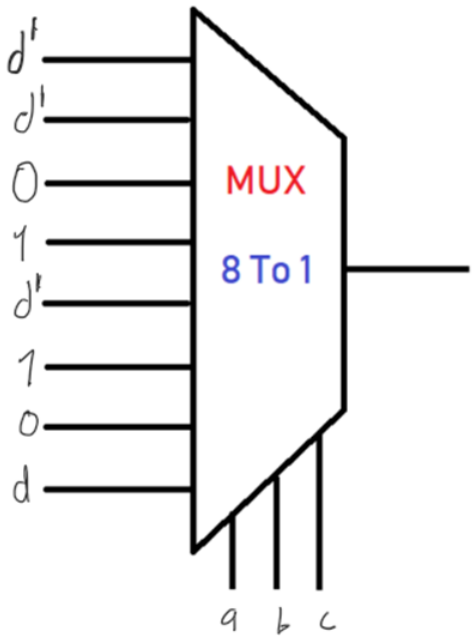

Figure 11: MUX 8x1

The circuit design implemented in Figure 33.

2.6 Question 2)

Using a decoder, we were able to create all possible minterms for (a,b,c), after that by using OR gates we where able to join them together to create the requested expressions, for F3 we also used our knowledge of boolean algebra to get rid of the (c) variable in the minterm.

2.7 Question 3)

2.7.1 Signed and Unsigned Addition For Binary Numbers

For adding two binary numbers we start form LSB (Least significant bit). Using logic, 0 and 0 results in a 0, 1 and 0 results in 1, 1 and 1 results in a carry to the bit more significant that it by 1. After we continue to the next left bit, after we make the addition we add the carry value to the result. For two n-bit unsigned integers added there can be a resultant carry needing a (n+1)th bit. This results in no problems if we use n+1 bits to represent the number, or else we cant represent it. For n-bit signed integers, its the same as unsigned integers, however, their MSB represents the sign, and bits greater than the nth bit (n+1) are ignored. If a carry occurs there could be an overflow.it happens when we add two positive numbers and get a negative number, or when we add two negative

numbers and get a positive one.

2.7.2 Signed and Unsigned Subtraction For Binary Numbers

In binary we do not use subtraction directly, we change the second operand to its negative value and add it to the first one, which is the same as subtraction. For that we use 2's complement: Flip all 1's and 0's to their counterpart, then add 1 to that value, this way we achieved that number's 2's complement. Then we add both numbers. For unsigned integer subtraction we can get either a carry or a borrow, if we get a carry then we just ignore it and the result is valid, if we get a borrow "no carry" then the result is invalid and the number can not be represented. For signed integer subtraction however,if there is a carry we just ignore it, but we need to check for the sign bits as an overflow can also occur here. if we have a negative - positive and get positive or if we have positive - negative and get negative then these are cases of overflow and we cannot represent them.

3 EXPERIMENT

3.1 Part 1

- To develop AND we used the bitwise operator & we created a normal 2-input AND module, a 3-input AND module for the 3 to 8 decoder, a 4-input AND module for the 8:1 multiplexer.
- To develop OR we used the bitwise operator |we created a normal 2-input OR module, an 8-input OR module for the 8:1 multiplexer.
- To develop NOT we used the bitwise operator ~ we developed a normal 1-input NOT module.
- To develop a XOR module we could not use its bitwise operator according to the question, so we had to develop it from its expression A'B + AB', thus we used two NOT gates to inverse A and B we also used two AND gates to join A' with B and A with B' the used an OR module to join the two terms to create an XOR.
- To develop a NAND module we used a normal 2-input AND gate and complemented its output to get a NAND.
- To develop a 8:1 MUX module we need three selector inputs s1-s3, then to create combinations from them we took the complement of each of these selectors. Then we utilized our 4 input AND modules each would take a unique combination of the 3 selectors and a unique input from 1 to 8. Then all the outputs of the AND modules

were (OR)ed using our 8 input or module, since with the selectors only 1 and gate could give an output so to take that output we OR all of them to get our value from the MUX.

- To develop a 3:8 decoder, we need 3 inputs, we also invert those inputs to be able to enumerate 8 values of output. Then we use three input AND modules to AND unique combinations of the 3 inputs to get a unique output from each AND module, thus creating 8 outputs.

3.2 Part 2

We used 4 wires for inputs. 8 wires used as intermediate section. Then, with the help of NOT modules, we inverted input a, b and d, and stored them in 3 intermediate wires. First $a'\&b$ calculated with AND gate. Again with AND gate result multiplied with c to get $a'bc$. For the second prime implicant, we multiplied a and c with AND module. Their result multiplied by d via AND gate to find acd. We store information in each step in different intermediate wires. At the end, b' and d' multiplied by AND gate. $a'bc$, acd, $b'd'$ prime implicants summed using 3 input OR gates. To verify our results we also ran simulations with different inputs.

3.3 Part 3

With the same 4 inputs, but with 6 intermediate wires, we implemeted the same equations with only using NAND gates. Some of the used NAND gates are 2 inputs some them are 3 inputs. Only difference between them is before the NOT operation, in 3 input NAND gate one more AND gate used.

At first, to gain a' we give two a inputs to the NAND gate, as $(a\&a)' = a'$. Did the same things to b and d inputs. Then a', b and c inputs given to 3-inputs NAND gate to obtain $(a'bc)'$. a, c and d inputs given to 3-inputs NAND gate to obtain $(acd)'$. At the end, b' and d' given to 2- inputs NAND gate to obtain $(b'd')'$. Then $(a'bc)'$, $(acd)'$, $(b'd')'$ all given to the 3-input NAND gate to calculate $((acd)'\&(a'bc)'\&(b'd')')'$ which is the equation we want to acquire. You can see the test results in the "RESULTS" section.

3.4 Part 4

We already defined 1 to 8 multiplexer. So what we do is to give correct wires to corect parts of the multiplexer. To do this, we again used 4 input wires and 3 more intermediate wires. In mux inputs, we need d, d', 0 and 1. d is already an input wire. To obtain d' we use NOT gate with d. To obtain 0 we know that $d\&d' = 0$ always. So, we used

AND gate with d and d' inputs. To obtain 1 we know that $d \| d' = 1$ always. So, we used OR gate with d and d' inputs. Then for mux module's inputs, we send a, b, c as selectors, $d', d', 0, 1, d', 1, 0, d$ as inputs respectively. This will get to our equation. Results and simulation can be observed in next section.

3.5 Part 5

For the implementation of function F2 we used a single 3:8 decoder and one OR gate. By utilizing the decoder we where able to attain all minterms for the inputs a,b,c. since we started with o1 as our first output, by going with the binary values required for F2, we picked o4 and o6 to be (OR)ed together to give us the required expression of a'bc + ab'c. For F3 we did the same thing but to attain the ab term, we (OR)ed abc and abc' to get ab then (OR)ed it with abc' again to get the desired expression. To verify our results we also ran simulations with different inputs.

3.6 Part 6

By writing out the truth table for the logic of the half adder we realised that the sum has the behavior of an XOR gate and the carry only exits when both inputs are on which results in an AND gate. So by adding those two modules we created a half adder, which we simulated for various inputs to check its correctness.

3.7 Part 7

To create a 1-bit full adder, we need to register a carry from a bit less significant that it, that is why we used a normal half adder with a and b as inputs, then for its outputs we took its sum with the carry from another bit (Cin) and put them in another half adder. to give us the total final sum. As for the carry from the first half adder and the second one, we used an OR gate for them, because if any of them result in a 1 then there is a carry.

3.8 Part 8

To implement a 4-bit full adder, we need two inputs with 4 bits and a Cin1. we partitioned the addition process for each bit on 4 separate 1-bit full adders. Thus our first 1-bit full adder takes the first bits from the inputs a and b and the Cin1 from the previous bit, the resulting carry would be Cin2 for the second adder that is taking a[1] and b[1]. So on the process goes till the 4th adder. after that all the sums for each adder

are concatenated together to give an output sum, and the resulting carry from the 4th adder would be the output carry for the 4-bit full adder.

3.9 Part 9

For creating an 8-bit full adder, just like what we did with the 4-bit full adder, we connect 8 1-bit full adders and partition the input bits to the adders accordingly and connect their output carrys as Cin for the following adders. Then the sum is concatenated with the correct order and the carry (out) from the last 1-bit adder would be the final carry from the 8-bit full adder.

3.10 Part 10

To make 16 bit adder - subtractor, we used 2 8-bit full adder. But the main difference from them is that we also used sub as a input wire. When sub=11111111, it will make substraction operation, but if sub=00000000, then it will make summation. And there is also minor differences on the bit sizes. In inputs we took 16-bits two number, and return value sum will also be 16-bits.

Two make substraction operation, we need 2's complement for a number. 2's complement for binary numbers can be applied by converting all the 0's to 1's and 1's to 0's and adding 1 to the obtained value.

Therefore, XOR gate can be used in here. If one of the inputs is 0, then other input's information will be transferred to output. However, if the one of the inputs is 1, then it will covert other one's value. We need also use XOR in carry section to acquire 2's complement rather then 1's complement.

In our implementation, we send 8-bits for sub because for the calculation of 8-bit inversion (which is required for 8-bit full adder) xor gates requires 8 bit for both inputs. You can see the simulation results from below section.

3.11 Part 11

For this question, all the input wires are same as previous example. For the subtraction with 16-bit adder, we can assign input wire sub=11111111. But to multiply input A by 2, we need another 16-bit adder or for the sake of this question, we used full adders to calculate sum for 16-bit input wire A with itself. Then result of this intermediate wire transferred to 16-bit adder. At the end, the module will do the same process as explained in earlier. All schematics and simulations of this example are under this section.

4 RESULTS

4.1 Schematics and Simulations

4.1.1 PART 1

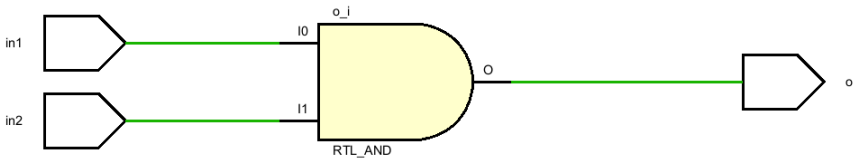

Figure 12: AND gate schematic

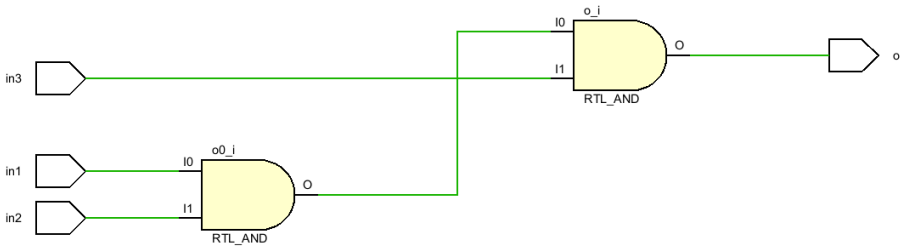

Figure 13: 3-input AND gate schematic

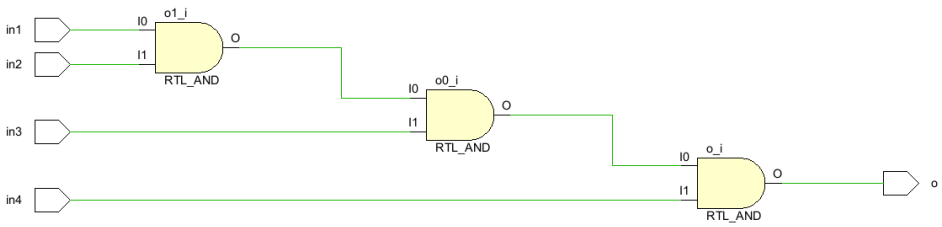

Figure 14: 4-input AND gate schematic

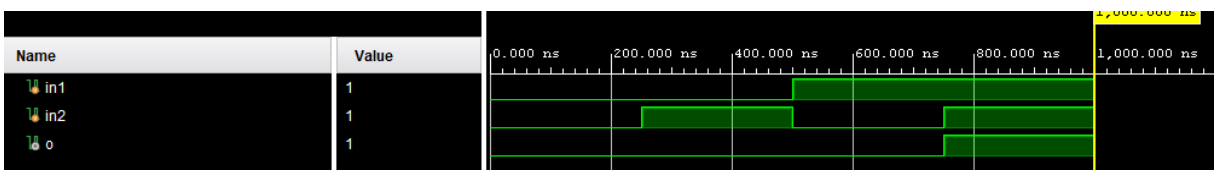

Figure 15: AND gate simulation

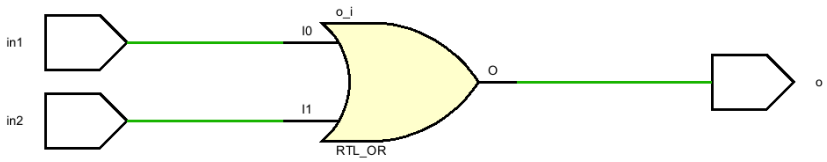

Figure 16: OR gate schematic

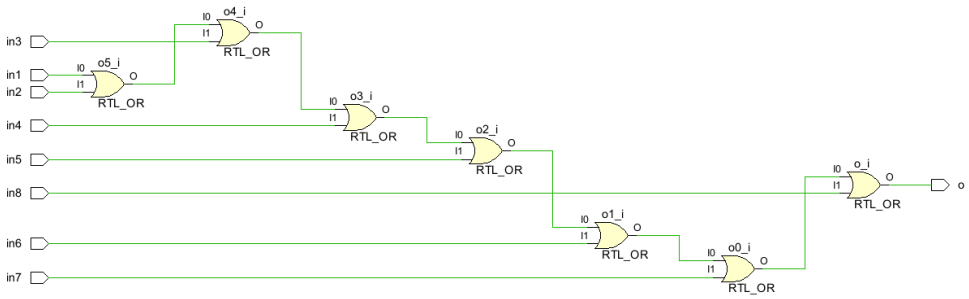

Figure 17: 8-input OR gate schematic

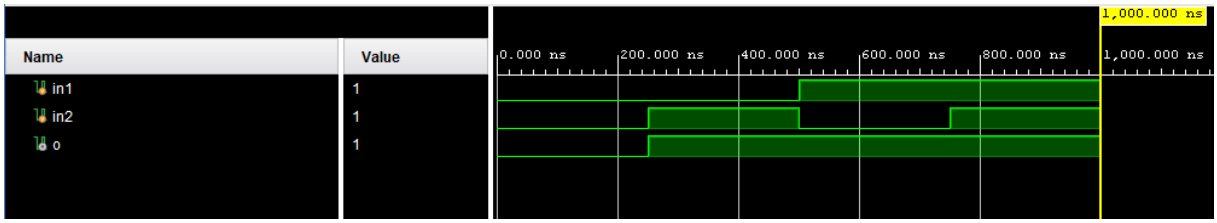

Figure 18: OR gate simulation

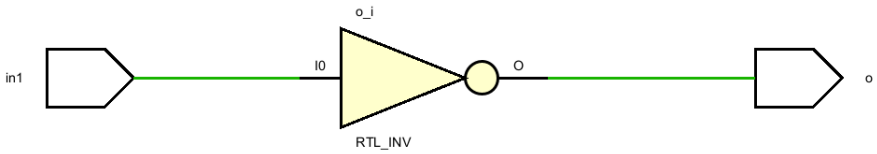

Figure 19: NOT gate schematic

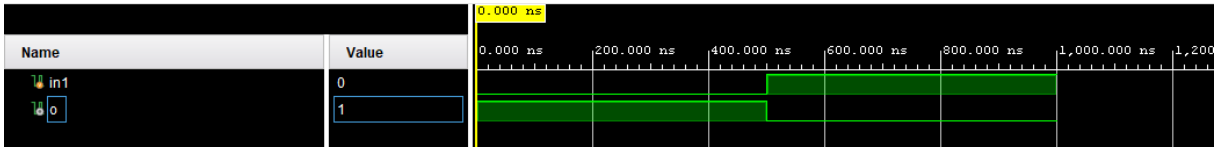

Figure 20: NOT gate simulation

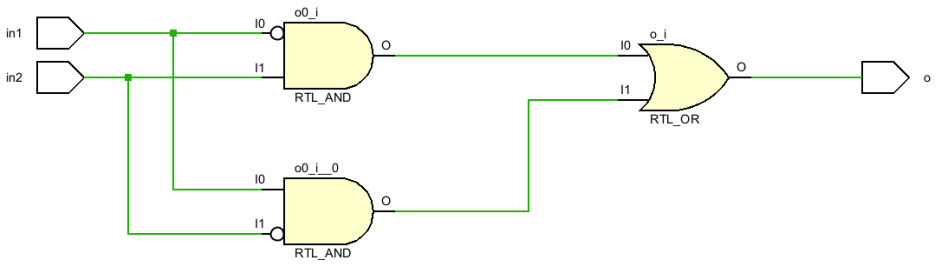

Figure 21: XOR gate schematic

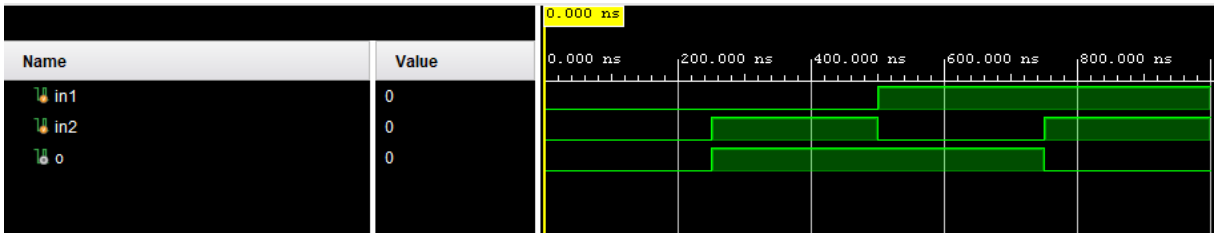

Figure 22: XOR gate simulation

Figure 23: NAND gate schematic

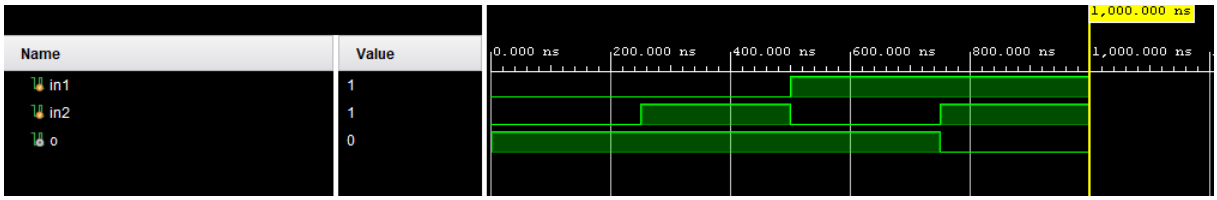

Figure 24: NAND gate simulation

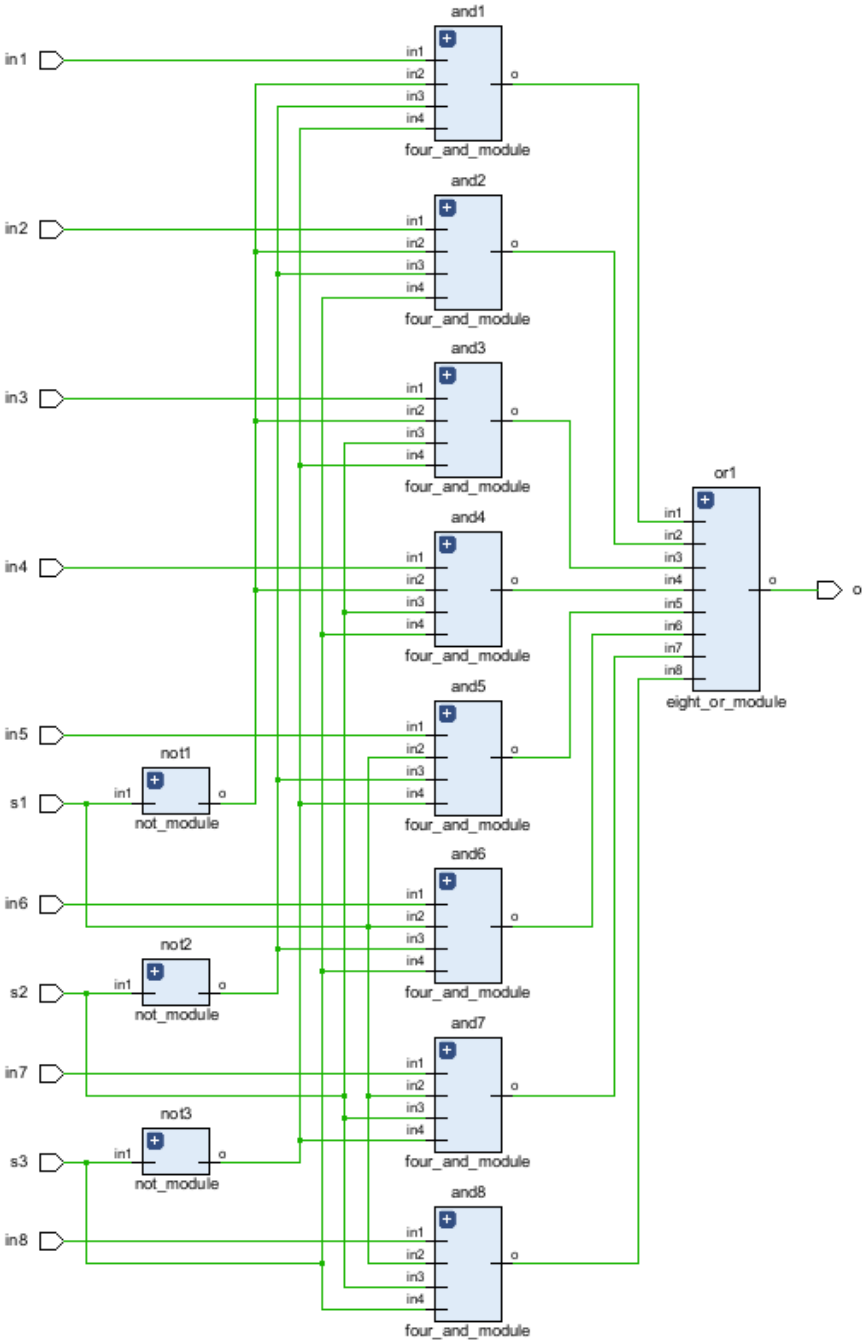

Figure 25: MUX schematic

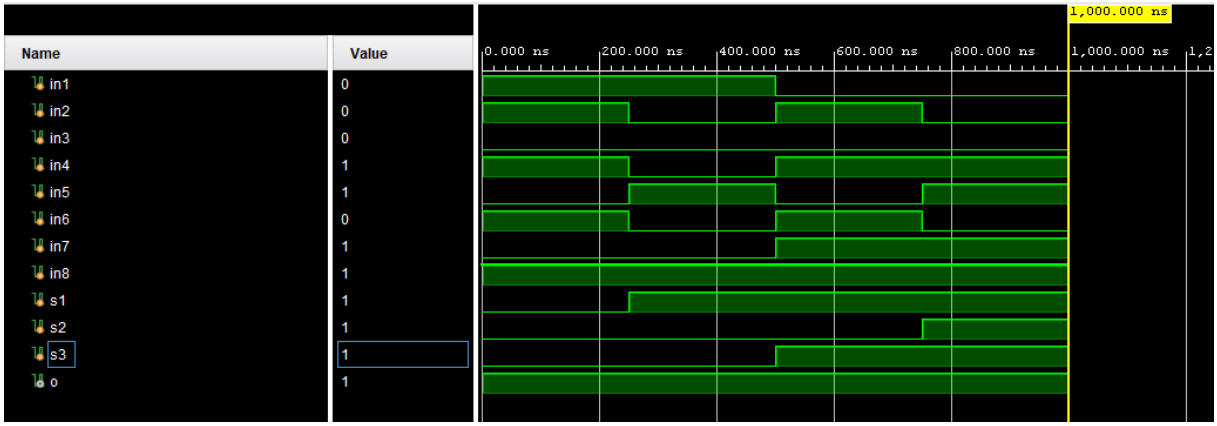

Figure 26: MUX simulation

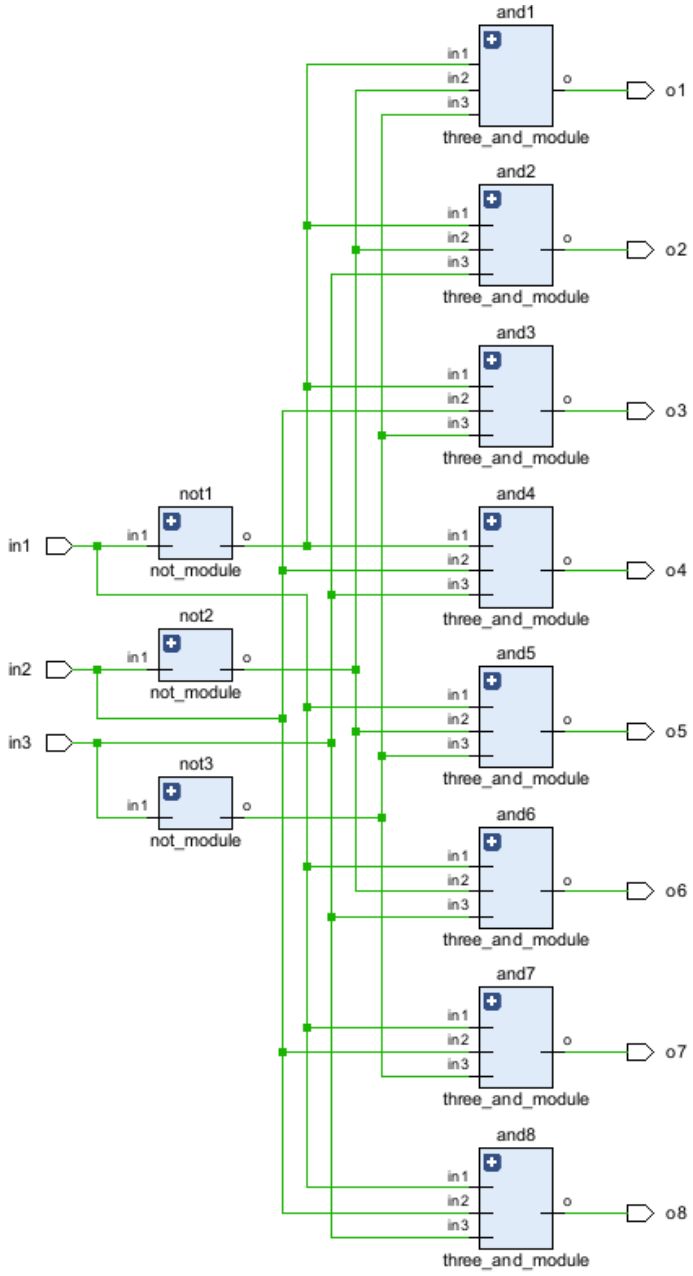

Figure 27: decoder schematic

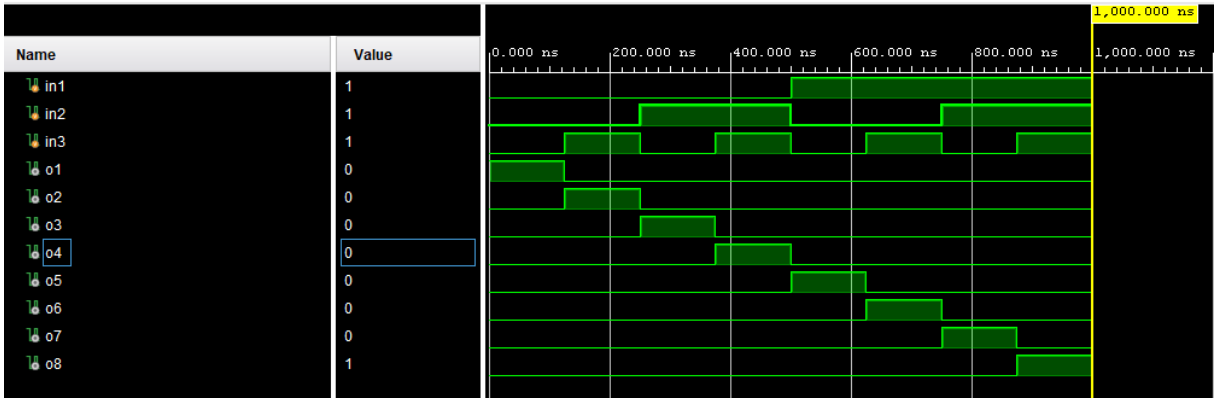

Figure 28: decoder simulation

4.1.2 PART 2

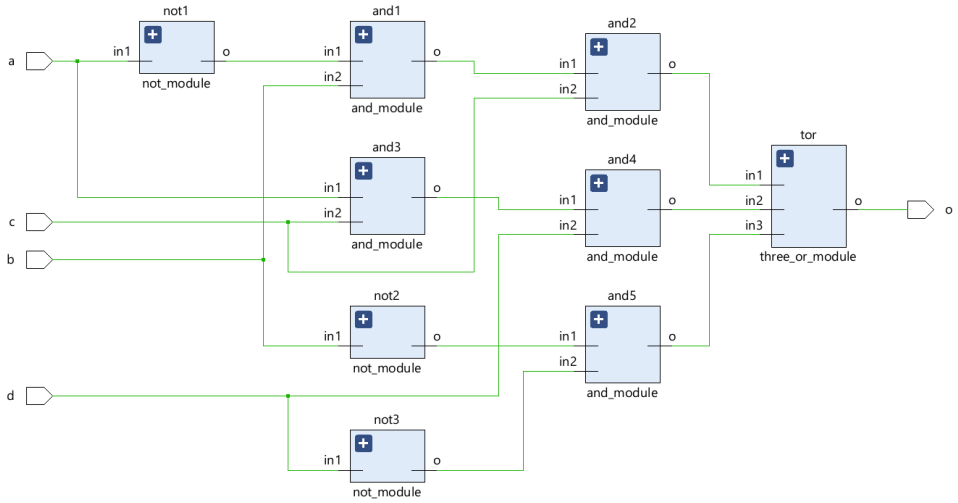

Figure 29: preliminary 1 using AND OR NOT gates

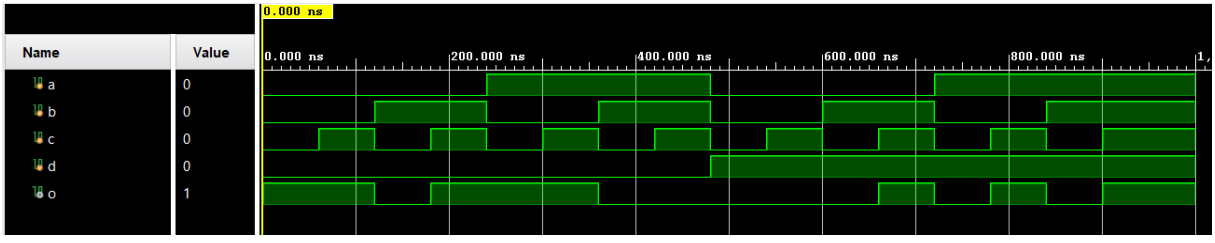

Figure 30: results for preliminary 1 using AND OR NOT gates

4.1.3 PART 3

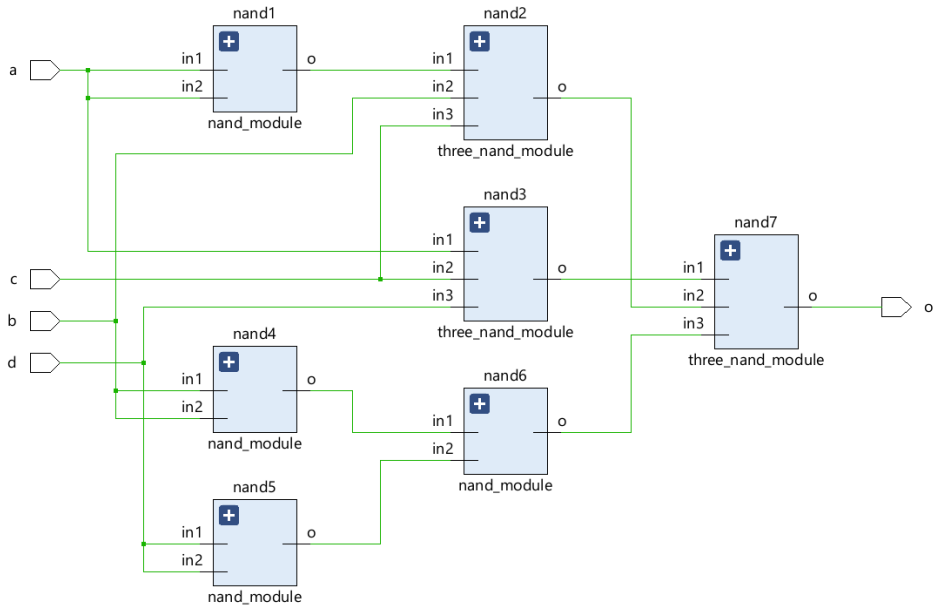

Figure 31: preliminary 1 using only NAND gates

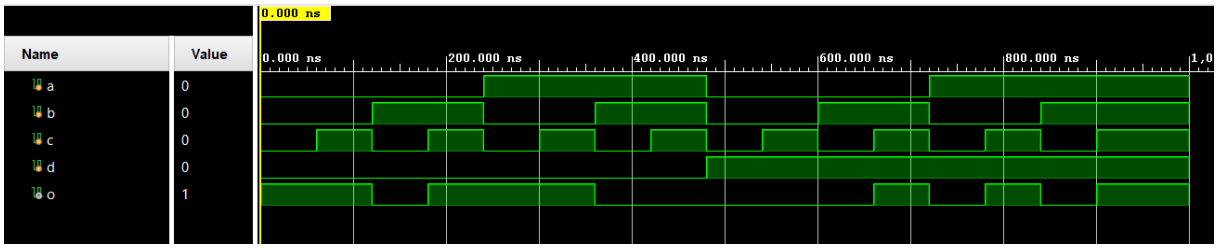

Figure 32: results for preliminary 1 using only NAND gates

4.1.4 PART 4

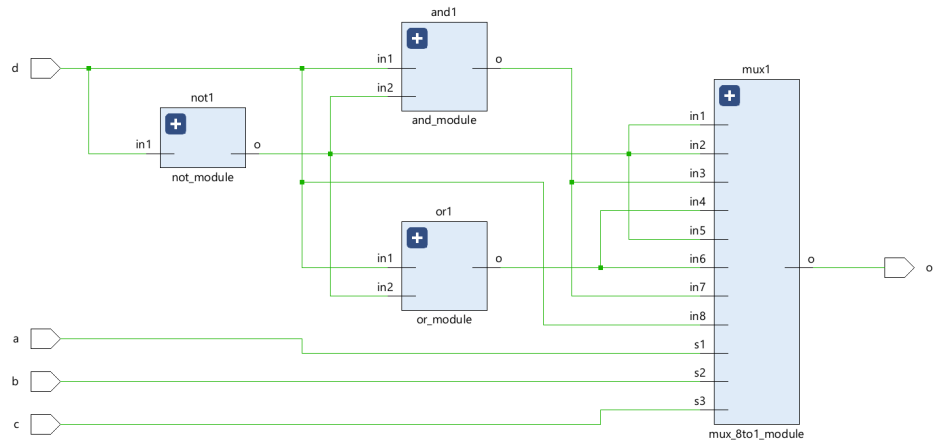

Figure 33: preliminary 1 using 8x1 MUX

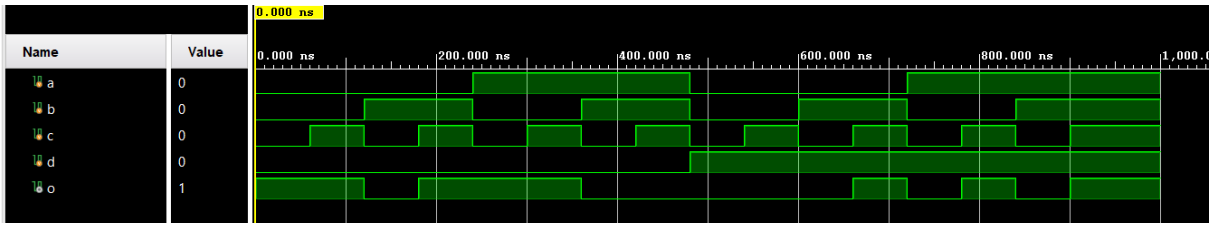

Figure 34: results for preliminary 1 using 8x1 MUX

4.1.5 PART 5

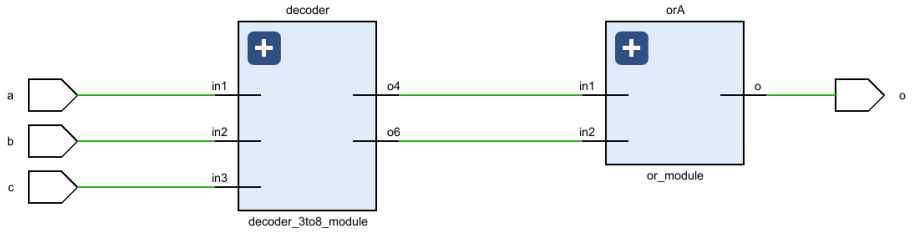

Figure 35: F2 schematic

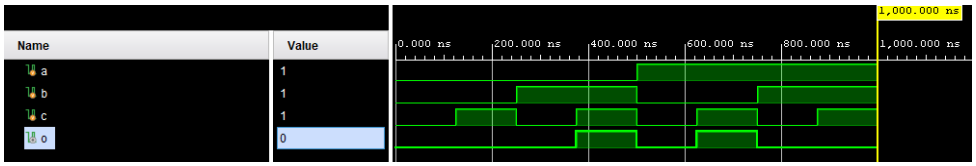

Figure 36: F2 simulation

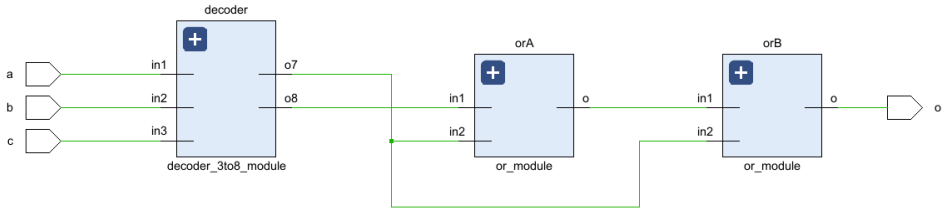

Figure 37: F3 schematic

Figure 38: F3 simulation

4.1.6 PART 6

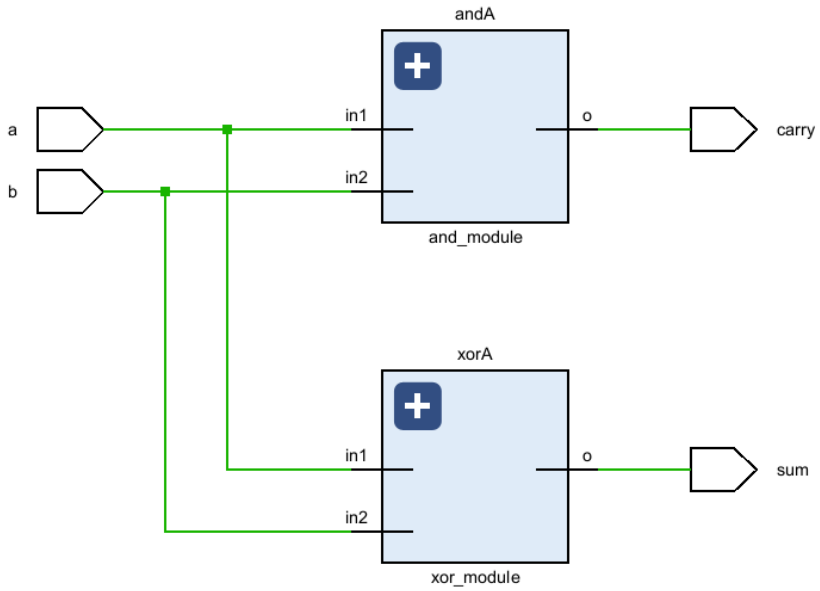

Figure 39: 1-Bit half adder schematic

Figure 40: 1-Bit half adder simulation

4.1.7 PART 7

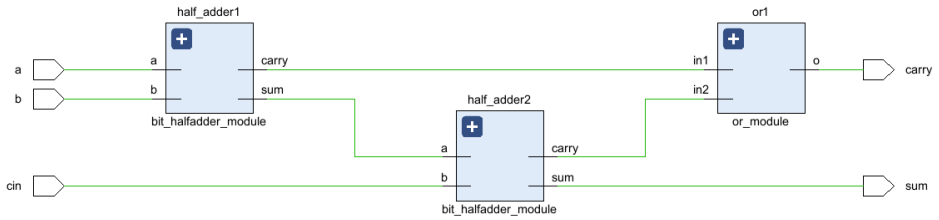

Figure 41: 1-bit full adder schematic

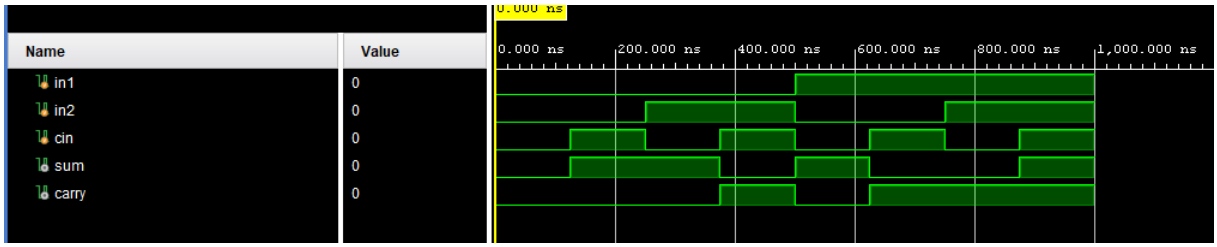

Figure 42: 1-bit full adder simulation

4.1.8 PART 8

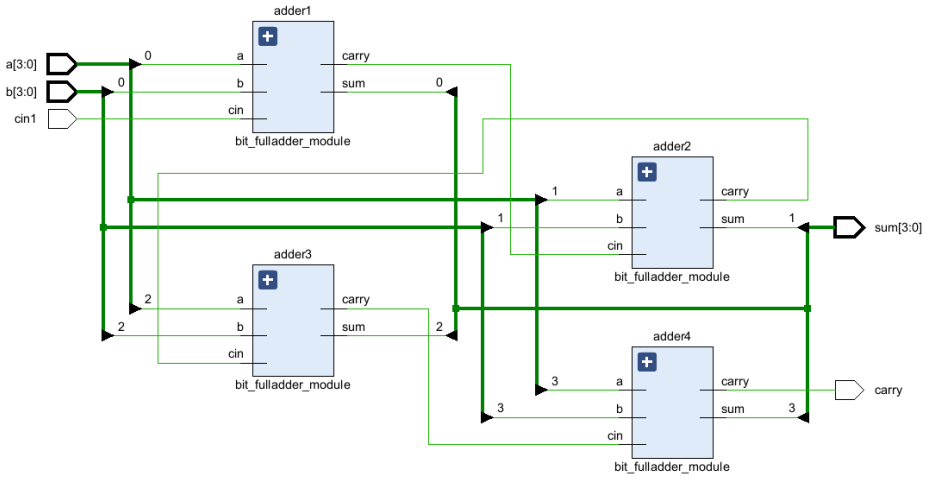

Figure 43: 4-bit full adder schematic

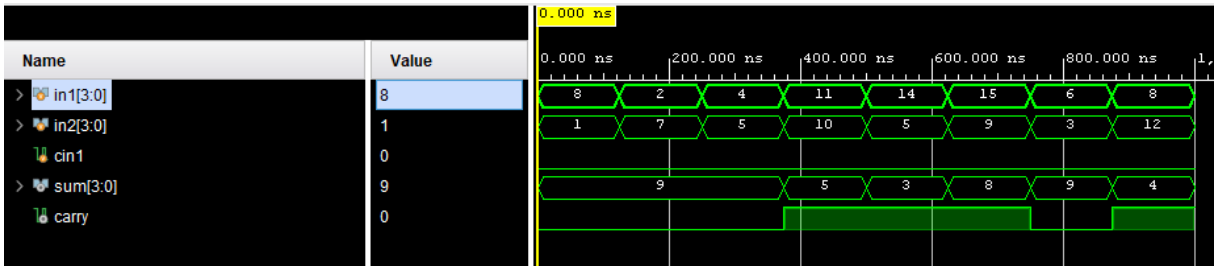

Figure 44: 4-bit full adder simulation

4.1.9 PART 9

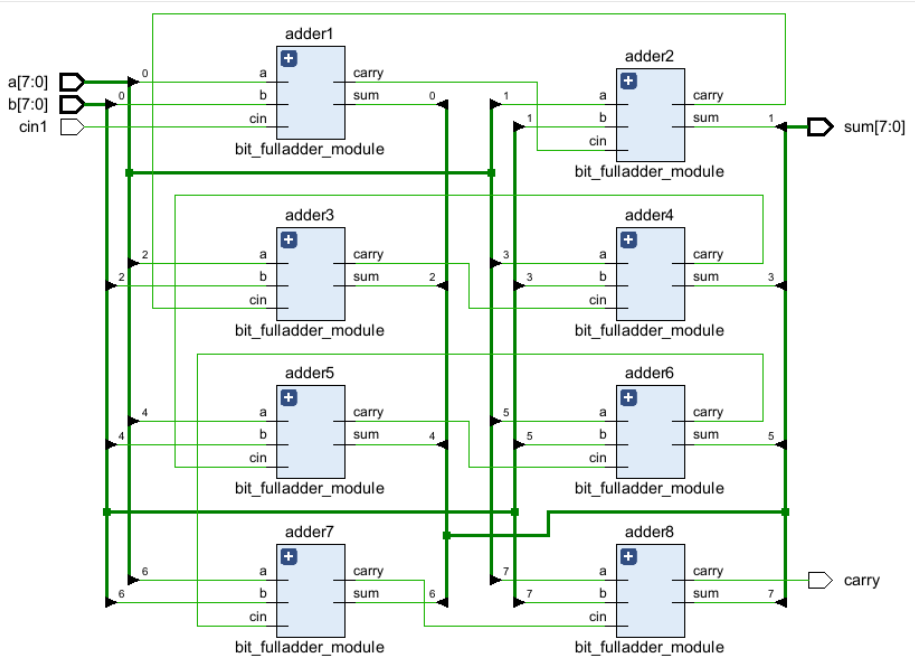

Figure 45: 8-bit full adder schematic

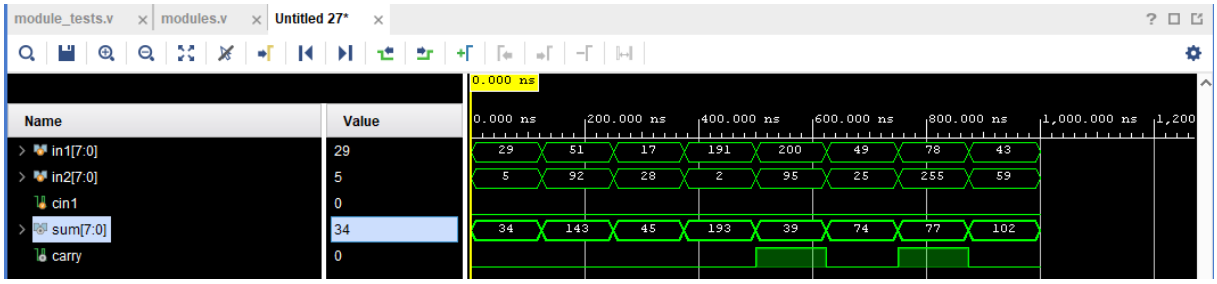

Figure 46: 8-bit full adder simulation

4.1.10 PART 10

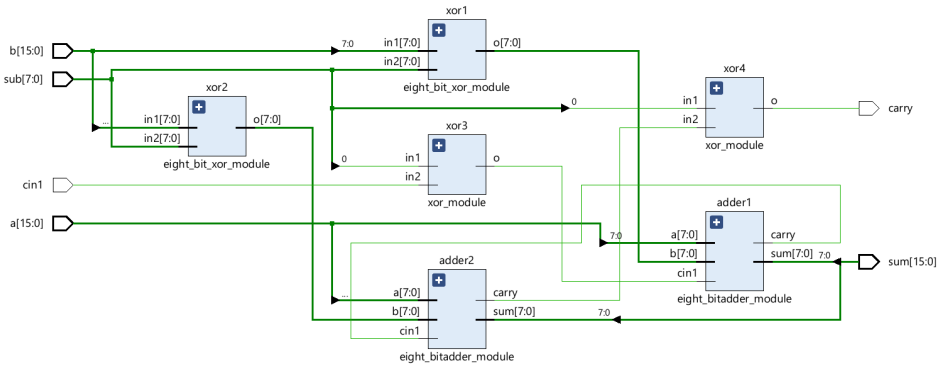

Figure 47: schematic of 16-bit-adder substractor

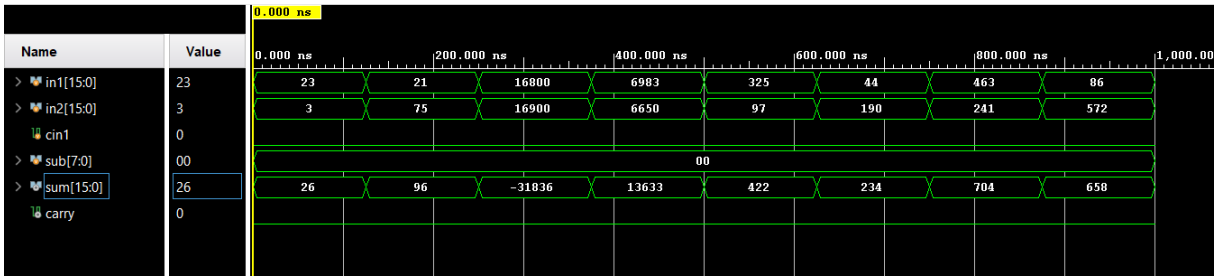

Figure 48: simulation of part 10

However, due to the overflow, result of 16800+16900 shows wrongly.

4.1.11 PART 11

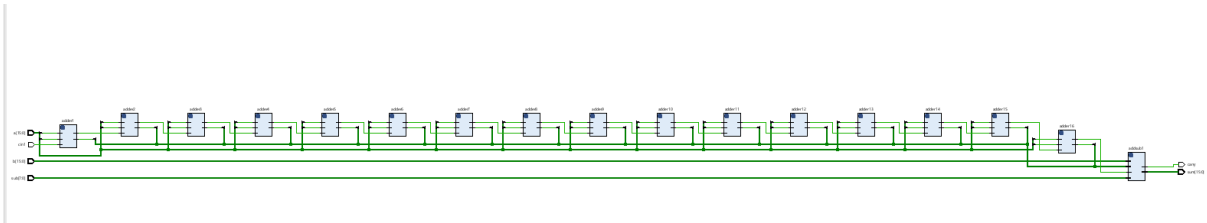

Figure 49: schematic of B-2A using 16-bit-adder-substractor(rightmost one) & full adders

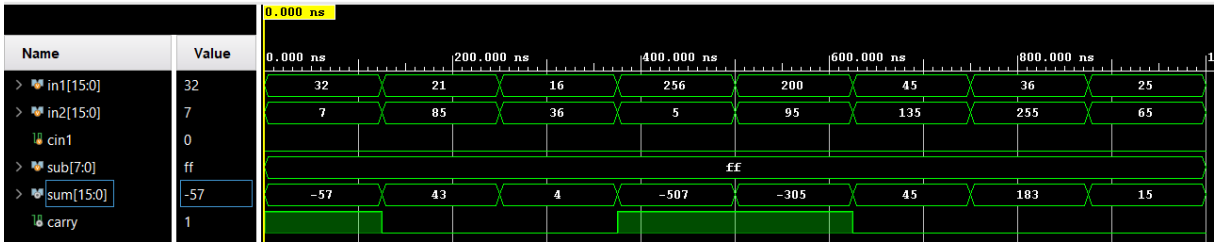

Figure 50: simulation of part 11

4.2 truth tables

a	b	(a ∧ b)
F	F	F
F	T	F
T	F	F
T	T	T

Figure 51: and gate truth table

a	b	(a v b)
F	F	F
F	T	T
T	F	T
T	T	T

Figure 52: or gate truth table

a	¬a
F	T
T	F

Figure 53: not gate truth table

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 54: xor gate truth table

A	B	A **NAND** B
0	0	1
0	1	1
1	0	1
1	1	0

Figure 55: nand gate truth table

INPUTS			Output
S_2	S_1	S_0	Y
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

Figure 56: 8:1 mux truth table

c	b	a	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Figure 57: 3:8 decoder truth table

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 58: 1-bit half adder truth table

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 59: 1-bit full adder truth table

5 CONCLUSION

With this homework, we learned how to use verilog in different applications. We write modules for NOT, AND, OR, XOR, NAND gates with different input size capabilities and designed 8x1 multiplexer and 3:8 decoder. For summation we designed half 1- bit adder, 1-bit full adder, 4-bit full adder, 8-bit full adder. Not only for summation but also for subtraction, we designed 16-bit adder substractor. We used our designed modules in various application. Such as with given expression we designed implementations with using AND, OR, NOT gates; using only NAND gates and using 8x1 MUX, NOT, AND, OR gates. We made functions with decoders and or gates. At the end, with using 16-bit adders and full-adders we designed implementation for the expression B-2A.