Q.no.1) Simplify the following Boolean logic functions to the format in sum of product first and then create a truth table in Excel for each as the verification reference of the circuit design, finally implement by online tools.

1a. f = $(A + C' + D')(B' + C' + D)(A + B' + C')$

= (A+ C' + D')(B' + C' + D)(A+ B' + C')
(first, outer, inner, last):

= (A+C" + D')(B' + C" + D)(A+ B' + C')
= A(B' + C" + D)(A + B' + C') + C'(B' +C" + D)(A + B' + C') + D'(B' + ' + D)(A + B' + C')

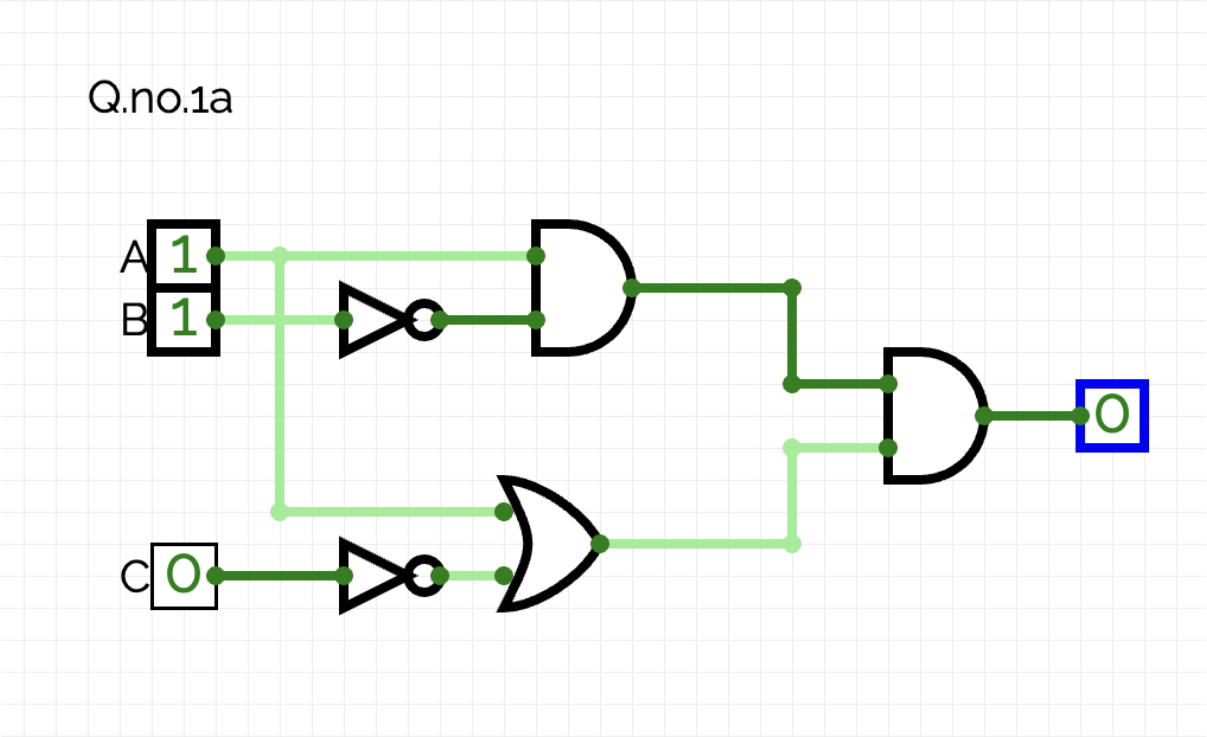
= AB'A+ AB'B' + AB'C' + AC'A + AC'B' + AC'C' + AD'A+ AD'B' + AD'C" †C' B'A + C'B'B' + C'B'C' + C'C'A+C'C' B' + C'C'C' + C'D'A+ C' D'B' + C' D'C' +D'B'A + D'B'B' + D'B'C' + D'C' A + D'C' B' + D'C'C" + D'D'A + D'D'B' + D'D'C'

= AB'A + AB'C' + AC'A + AC' B' + AD'A + AD'B' + AD'C' + C' B'A+ C'B'C" + C'C'A + C'C'B' + C'D'A + C'D'B' + C'D'C' + D'B'A + D'B'C' + D'C' A + D'C' B' + D'D'A + D'D' B'

[AB'A + AB'C' = A(B' + C')]
AC' A + AC' B'
= AC' (A + B')

A	B	C	B'	C'	A+B'	AC'	AC'(A+B')
0	0	0	1	1	1	0	0
0	0	1	1	0	1	0	0
0	1	0	0	1	0	0	0
0	1	1	0	0	0	0	0
1	0	0	1	1	1	1	1
1	0	1	1	0	1	0	0
1	1	0	0	1	1	1	1
1	1	1	0	0	1	0	0

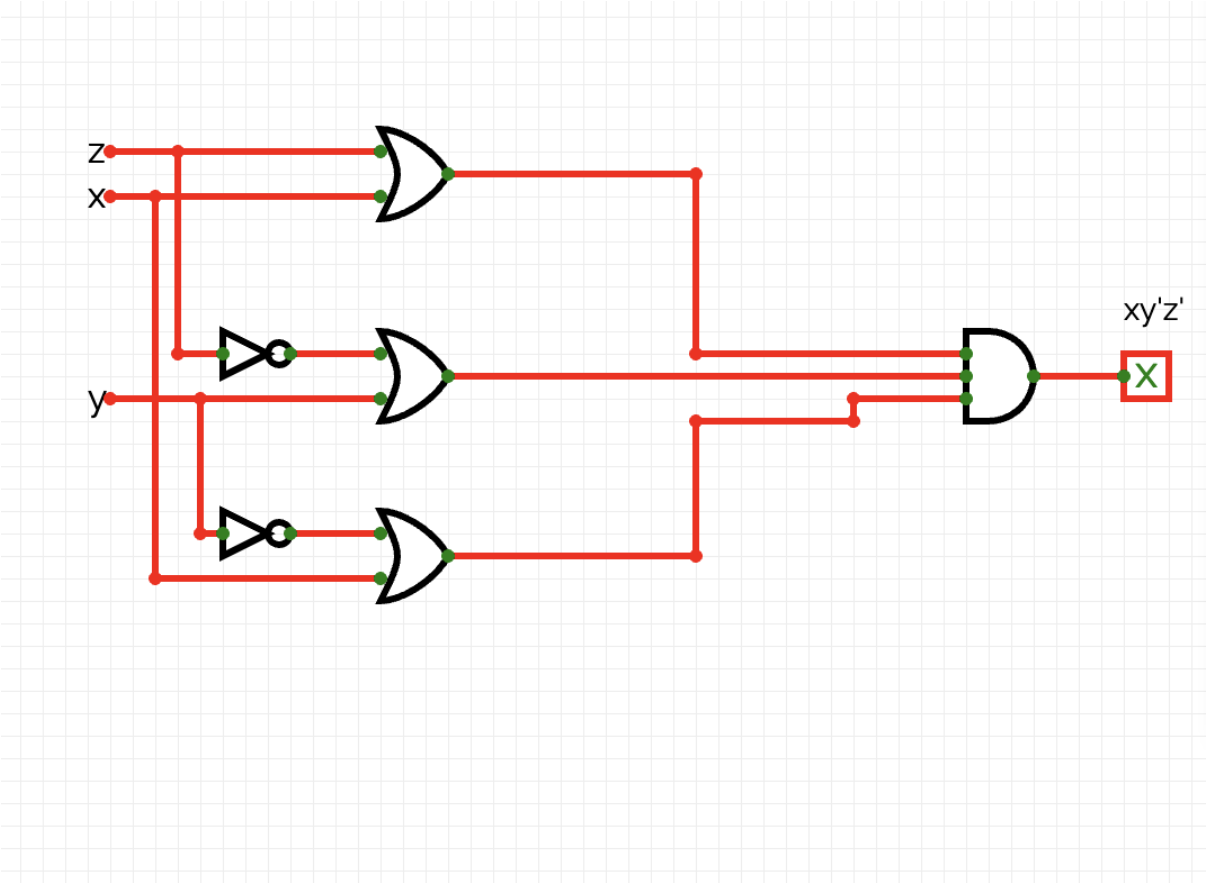
Test:



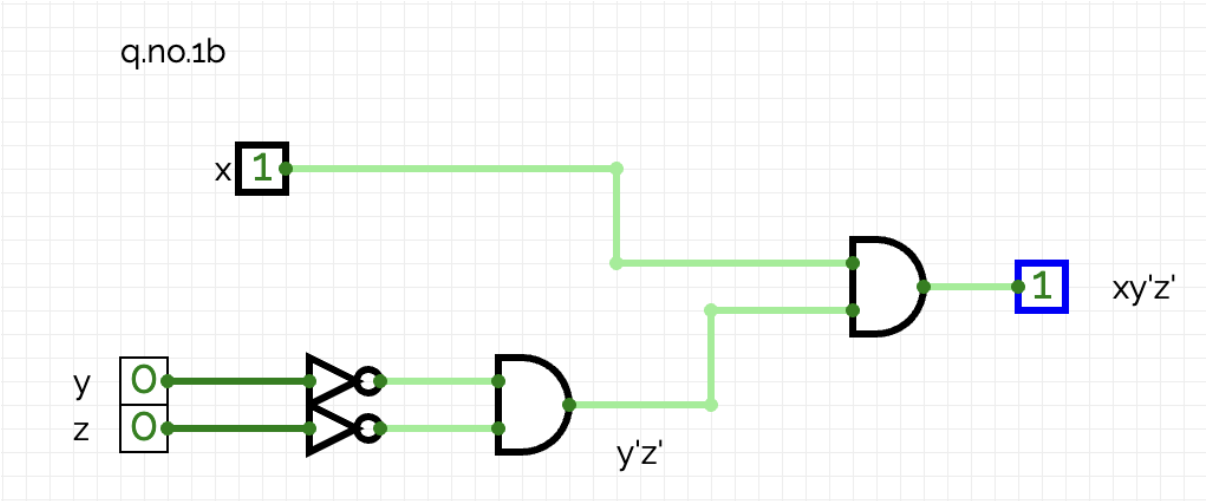
1b. f= (z+x)(z'+y')(y'+x)

= {(z + x)(x+y')}(y'z') [Using Demorgan's law: (x'+y' = x'y')]
= (x + y'z)(y'z') [By using Distributive law: (p+q)(p+r)=p+qr]
= xy'z' + y'y'zz'
= xy'z' + y'.0 [y'.y' = y' , z.z'=0]
= xy'z' + 0
= xy'z'

x	y	z	y'	z'	xy'z'
0	0	0	1	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	0	1	0
1	1	1	0	0	0



Test:

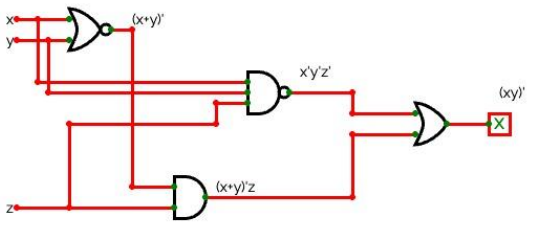


1c. f = (x + y)'z + x'y'z'

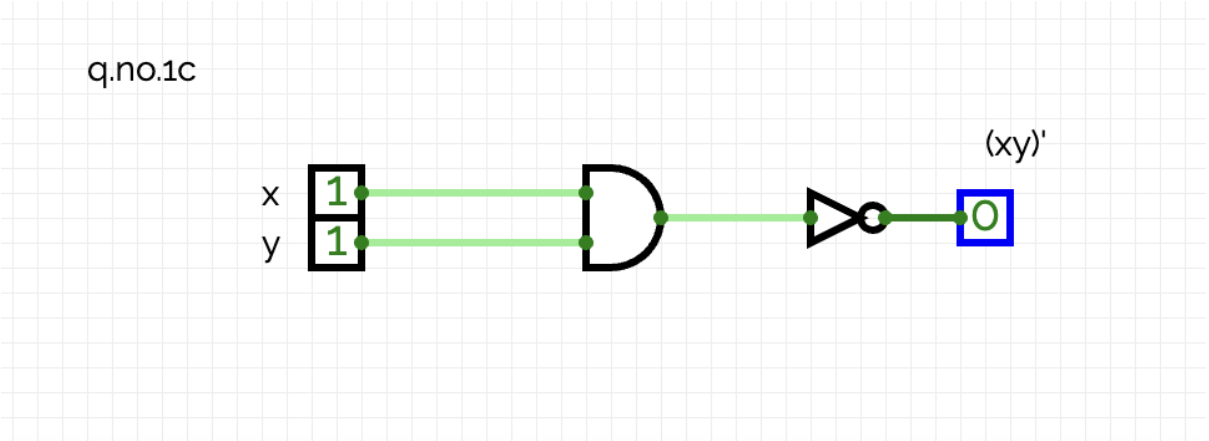
=(x'y')z + x'y'z' [By using Demorgan's law: (x+y)' = x'y']
= x'y'(z + z')
= x'y'.1 [Identity law : x+x' =1]
= x'y'

= (xy)’

x	y	z	xy	(xy)'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0



Test:



Q.no.2) Design signal bit full adder based on the truth table and circuit on chapter 3 lecture handouts first and then create 4-bit "Adder-Subtractor circuit" as follows to implement arithmetic addition and subtraction operations.
If S = 0 in the circuit, then $X3$ = $B3$, $X2$ = $B2$, $X1$ = $B1$ and $X0$ = $B0$, so the adder circuit simply adds A and B when $C0$ = S = 0 (carry in = 0). - If S = 1, then $X3$ = $B3$, $X2$ = $B2$, $X2$ = $B2$ and $X0$ = $B0$. Since $C0$ = S = 1, the circuit is equivalent to adding the 2's complement of B to A, that is, implementing subtraction operation "A-B"

Verify your design with the following testcases by the conversion from decimal to binary as inputs A and B (a) A + B = 7 + (-3) = 4 (b) A - B = -6 - (-1) = -5

Solution:

Full Adder Design: The diagram already provides a design of a single-bit full adder with three inputs: A, B, and a carry-in (C_in), and two outputs: the sum (S) and a carry-out (C_out). The logic operations being used here are AND, OR, and XOR gates.

4-bit Adder-Subtractor Design: For a 4-bit adder-subtractor circuit, you would essentially need four such full adders cascaded. The adder-subtractor works by taking advantage of the fact that the subtraction of a number is the same as the addition of its 2's complement.
If S = 0 (signifying addition):
• X3 = B3, X2 = B2, X1 = B1, and Xo = Bo
• The carry in Co is set to O.
• The operation performed is A + B. 2.

If S = 1 (signifying subtraction):
• X3 = B3, X2 = B2, X1 = B1, and Xo = Bo.
• The carry in Co is set to 1.
• The operation performed is A + (2's complement of B) which is equivalent to A - B.

Now, let's convert the test cases from decimal to binary to verify the design:

(a) A + B = 7+ (-3)

• A = 7 in binary is 0111.
• B = -3 in 2's complement representation is 1101.
(3 in binary is 0011. Inverting gives 1100. Adding 1 gives 1101).

So, when adding A and B, the circuit should output 4 which is 0100 in binary.

(b) A - B = -6 - (-1)

• A = -6 in 2's complement representation is 1010.
(6 in binary is 0110. Inverting gives 1010.Adding 1 gives 1010).

• B = -1 in 2's complement representation is 1111.
(-1 is represented as all ones in 2's complement).

So, when subtracting B from A (A - B), the circuit should output -5 which is 1011 in binary.

The provided circuit diagrams are for the full adder. For a 4-bit "Adder-Subtractor circuit", we'll chain 4 of these full adders together, ensuring the carry output from one feeds into the carry input of the & Regenerate next.

When you want to perform an addition (S=0): The bits of the B number are passed directly to the adder. The carry-in of the least significant bit (C_0) will be 0. The adder then simply adds A and B.

When you want to perform a subtraction (S=1): The B input will be inverted (to get 1's complement) and a carry-in of 1 is introduced into the least significant bit adder (to make it 2's complement). The adder then adds A to the 2's complement of B, which is equivalent to subtracting B from A.

Test the Design:
For the given test cases:
(a) A + B = 7 + (-3) = 4
A = 7 = 0111 in binary
B = -3, which is the 2's complement of 3
3 = 0011 in binary 2's complement of 3 = invert all bits and add 1 = 1101

When S=0 (Addition), use:
A = 0111
B = 1101
Expected result = 1000, which is 8 in binary (Note: This isn't 4 due to overflow, since we are effectively adding 7 and -3 in a 2's complement system). The sign bit indicates that the result is negative, so the result is -8. To find the absolute value, take the 2's complement of 1000 to get 1000, which is 8 in decimal.

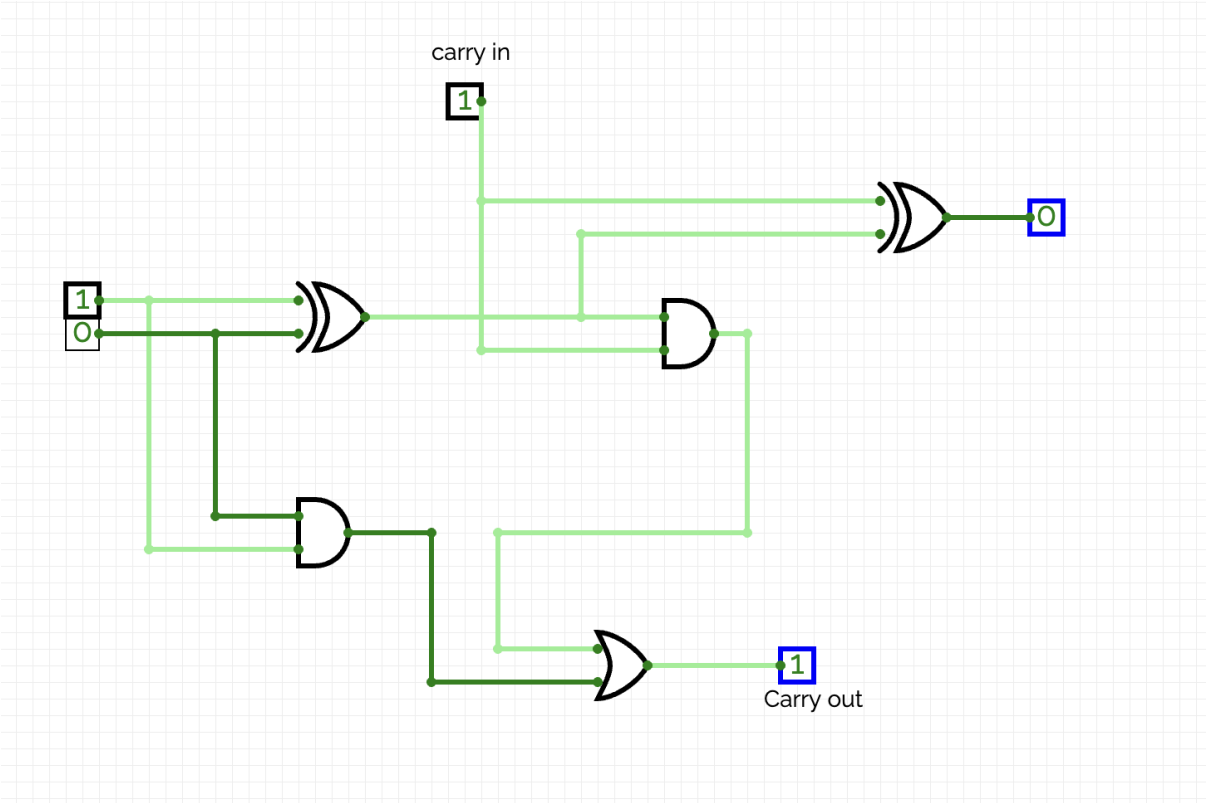
(b) A - B = -6 - (-1) = -5
A = -6 = 2's complement of 6 = 1010 in binary

B = -1 = 2's complement of 1 = 1111 in binary

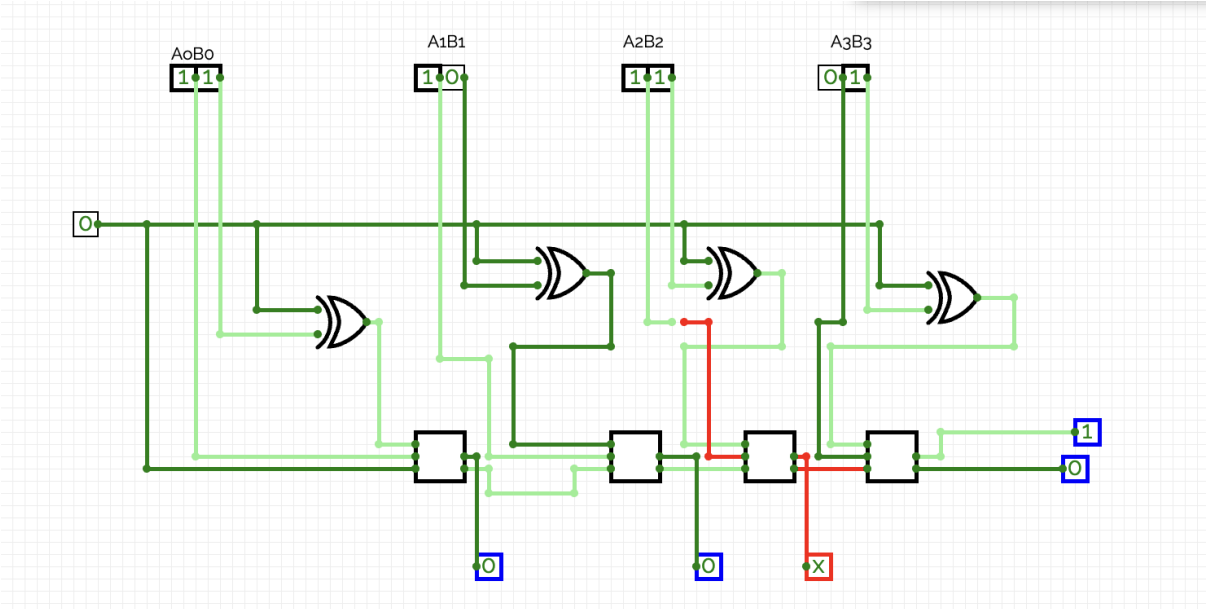
When S=1 (Subtraction), use:

A = 1010
B = 1111
Expected result = 1001, which is -7 in decimal.
However, it's important to note that when working with 2's complement arithmetic, we should interpret the results based on the sign bit (the leftmost bit). If the sign bit is 1, the result is negative and you need to take the 2's complement to get its absolute value.

Full adder:



Output:



Q.no.3)

The instruction for "Load X" is 0001, X represents the address to load from.
The instruction for "Add X" is 0011, X represents the address to add from.
The instruction for "Store X" is 0010, X represents the address to store to.
Load 104

- Binary: 0001 0000 1000 // "0001" is opcode for "Load", "0000 1000" is binary for 104
- Hex: 1 08 // "1" represents "Load", "08" is hex for 104

Add 105

- Binary: 0011 0000 1001 // "0011" is opcode for "Add", "0000 1001" is binary for 105
- Hex: 3 09 // "3" represents "Add", "09" is hex for 105

Store 106

- Binary: 0010 0000 1010 // "0010" is opcode for "Store", "0000 1010" is binary for 106
- Hex: 2 0A // "2" represents "Store", "0A" is hex for 106

So, the machine code for the given assembly instructions are: 1 08 3 09 2 0A

The binary representation of the instruction code determines which operation is to be executed. Here, we will represent these instructions in a 4-bit format (based on the provided table) and use the remaining 12 bits for the address.

Instruction	4-bit code	Address	Machine Code
Load	0001	0000 1000	0001 0000 1000
Add	0011	0000 1001	0011 0000 1001

Share	0010	0000 1010	0010 0000 1010

Now, let's design a simple circuit for the Load instruction as an example. For a "Load" operation, the output will be active (1) when the 4-bit code is 0001. Here's a simple representation: Inputs: A, B, C, D (representing the 4-bit code in reverse, i.e., D is the most significant bit) Output: Y (active when the input is 0001).

D	C	B	A	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0

Creating similar circuits for the "Add" and "Store" operations based on their specific 4-bit codes..