Combination Logic

Made by Md. Mehedi Hasan Rafy
with Assistance of Nur Alam Shikder

1. What are the advantages of encoding a decimal number in BCD as compared to straight binary? What is its disadvantage?

Encoding decimal numbers in Binary-Coded Decimal (BCD) has both advantages and disadvantages compared to straight binary representation.

Advantages of BCD

- ***Human Readability:*** BCD is more human-readable than straight binary because each decimal digit is represented by a 4-bit binary code. This makes it easier for humans to interpret the numbers directly. In applications where human interaction is common, such as displays and data entry, BCD is preferred.
- ***Simplified Arithmetic:*** BCD representation simplifies decimal arithmetic operations, like addition and subtraction. This is especially useful in applications that require precise decimal calculations, such as financial systems, calculators, and real-time clocks.
- ***Error Detection:*** BCD encoding can help in error detection. For example, it's easier to identify invalid BCD codes as they fall outside the range of valid BCD values (0000 to 1001 for each digit).

Disadvantages of BCD

- ***Wasteful Storage:*** BCD numbers encode less information than binary numbers with the same number of bits, as 6 of 16 possible states are not used for each 4-bit BCD number. Mathematically, each BCD “bit” only encodes 10^(1/4) = 1.778 states, while each binary bit encodes 2 states. So, the larger the BCD number, the more bits BCD coding wastes.
- ***Slower Computations:*** BCD arithmetic requires more complex circuitry and is generally slower than binary arithmetic. This can be a disadvantage in applications where speed is a critical factor, like high-performance computing.
- ***Limited Range:*** BCD can only represent the digits 0 to 9, while binary can represent a wider range of values with the same number of bits. This limitation can be a disadvantage in applications that need to handle a broader set of data types.
- ***Representation of states:*** BCD codes represents less number of states compare to binary numbers. Binary numbers can represent 2 states per bit, as every bit can be a 1 or a 0. A 4-bit binary number can represent 2^4 = 16 states.
  BCD numbers use 4 bits to represent one decimal digit. That means that a 4-bit BCD number can represent ten states (0-9).

2. What range of decimal values can be represented by a four-digit octal number?

A four-digit octal number represents values in base-8, which means each digit can have 8 different possible values (0 to 7). To find the range of decimal values that can be represented by a four-digit octal number, you need to consider the minimum and maximum values for each digit.

The minimum value for each digit in octal is 0, and the maximum value is 7.

Therefore, for a four-digit octal number:

- The minimum value is 0000 in octal.
- The maximum value is 7777.

Now, $(0000)_8 = 0\times8^3+0\times8^2+0\times8^1+0\times8^0=(0)_{10}$

and $(7777)_8 = 7\times8^3+7\times8^2+7\times8^1+7\times8^0=7+56+448+3584=(4095)_{10}$

So, a four-digit octal number can represent decimal values in the range of 0 to 4095.

3. A typical PC as a 20-bit address code for its memory locations-

- How many Hex digits are needed to represent a memory address?
  Each hexadecimal (hex) digit represents 4 bits. To represent a 20-bit memory address, you would need $\frac{20}{4}=5$ hexadecimal digits. Therefore, 5 hex digits are needed to represent a memory address.

- What is the range of addresses?
  With 20 bits, you can represent 2^20 unique addresses. The range of addresses starts from 0 and goes up to the maximum address, which is 2^20 - 1 in decimal. In hexadecimal, this range is from 0x00000 to 0xFFFFF.

- What is the total number of memory locations?
  The total number of memory locations is equal to the number of unique addresses that can be represented with a 20-bit address code. As mentioned above, this is 2^20, which is 1,048,576 memory locations. In hexadecimal, this is 0x100000.

4. Perform the subtractions $(01001)_2-(11010)_2$ and $(10010)_2-(10011)_2$ using 2's complement.

I. Suppose,
A=01001 [which is equivalent of decimal 9] and
B=11010 [which is equivalent of decimal 26].
using 1’s compliment -B=00101
using 2’s compliment -B=00101+1=00110
Now, suppose result, C=A-B=A+(-B)=01001+00110=01111 [which is equivalent of decimal -17]
To confirm this we can again perform,
1’s compliment on C=10000
2’s compliment on C=10000+1=10001[which is equivalent of +17].
So, the subtraction of $(01001)_2-(11010)_2=(01111)_2$

II. Suppose,
A=10010 [which is equivalent of decimal 18] and
B=10011 [which is equivalent of decimal 19].
using 1’s compliment -B=01100
using 2’s compliment -B=01100+1=01101
Now, suppose result, C=A-B=A+(-B)=10010+01101=11111 [which is equivalent of decimal -1]
To confirm this we can again perform,
1’s compliment on C=00000
2’s compliment on C=00000+1=00001[which is equivalent of +1].

So, the subtraction of $(01001)_2-(11010)_2=(01111)_2$.

5. Draw the logic diagram and truth table of OR, NOR and XOR gate using NAND gate only.

Implementation of OR Gate from NAND Gate

The NAND gate is a universal gate, therefore, it can be used to realize the OR gate. The implementation of OR gate using the NAND gate is shown in Figure-3.

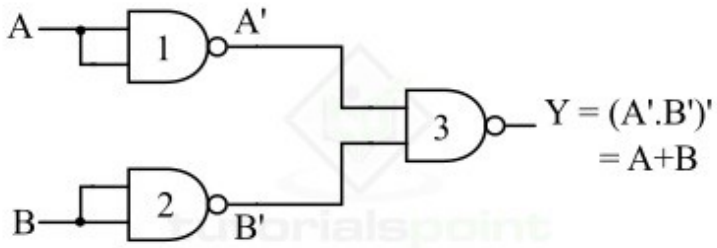


Figure 3 - OR Gate Using NAND Gate

To realize the OR gate using NAND gate, we first complement the inputs A and B. This is done by the NAND Gate 1 and 2 in the above Figure-3. Then, these complemented inputs, for example A' and B' are applied to a NAND Gate (NAND Gate 3). Thus, we get,

$Y=(A'+B')'$

Using De Morgen's Law, we have, $Y=(A')'+(B')'=A+B$

Truth table:

A	B	$Y_1=A'$	$Y_2=B'$	Y
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

This is the output equation of the OR gate. Therefore, the logic circuit of NAND gates in Figure-3 is equivalent to the OR Gate.

Implementation of NOR Gate from NAND Gate

The NAND gate is a universal logic gate, therefore, it can be used to realize the any other logic gate. The implementation of NOR gate using the NAND gate is shown in Figure-3.

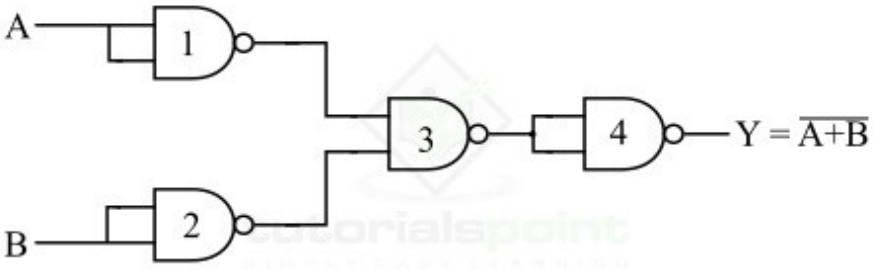


Figure 3 - NOR Gate using NAND Gate

From the logic circuit, it is clear that for the implementation of NOR gate using NAND gates only, we require 4 NAND gates.

The first two NAND gates perform the complement of input variables A and B.

The output of the first and second NAND gates is, $Y_1 = A'$ and $Y_2 = B'$

The third NAND gate produce the NAND output of the complemented inputs, in the example A' and B'.

The output of the third NAND gates is, $Y_3 = (A'.B')' = A + B$

Finally, the fourth NAND gate again operates as an inverter and produce the output Y. This output Y is the equivalent to the output of the NOR gate.

The output of the fourth NAND gate is, $Y = (A + B)'$

Truth table:

A	B	$Y_1 = A'$	$Y_2 = B'$	$Y_3 = (A'.B')' = A + B$	Y
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

Hence, this is the output of a NOR Gate. In this way, we can implement a NOR gate using NAND gates only.

<u>Implementation of XOR Gate from NAND Gate</u>

The NAND gate is a universal logic, hence, using which we may implement any other logic gate. Figure-3 shows how you can implement a XOR gate using only NAND gates.

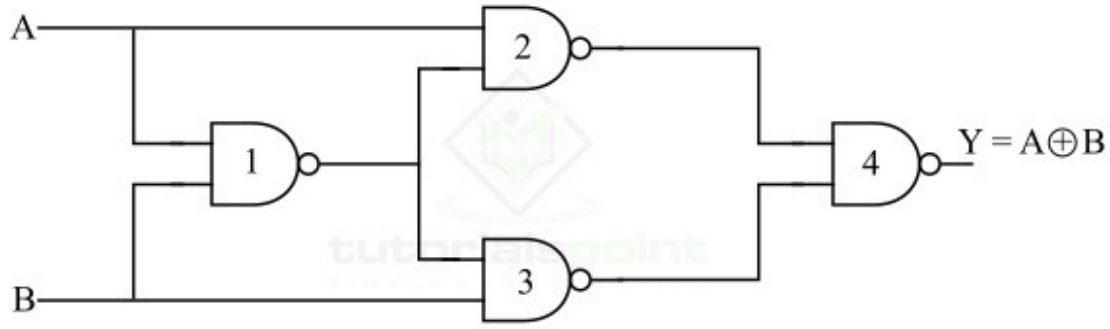


Figure 3 - XOR Gate using NAND Gates

From the logic circuit diagram of the XOR gate using NAND gates only, it is clear that we require 4 NAND gates.

Now, let us understand how this NAND logic circuit functions to produce an output equivalent to the XOR gate.

The output of the first NAND gate is, $Y_1=(AB)'$

The outputs of the secondary and third NAND gates are, $Y_2=(A.(AB)')'$ and $Y_3=(B.(AB)')'$

Finally, these two outputs (Y2 and Y3) are connected to the fourth NAND gate. This NAND gate will produce an output which is,

$Y=((A.(AB)')'.(B.(AB)')')'=A.(AB)'+B.(AB)'=A(A'+B')+B(A'+B')$

$\rightarrow Y=AA'+AB'+A'B+BB'=AB'+A'B$

$\therefore Y=A\oplus B$

Truth table:

A	B	$Y_1=(AB)'$	$Y_2=(A.(AB)')'$	$Y_3=(B.(AB)')'$	Y
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

This is the output of the XOR gate. Hence, in this way, we can implement the XOR gate from NAND gates only.

6. Simplify the following Boolean expression

- $X=ABC+A'BC+AC+AC'$
  $\rightarrow X=BC(A+A')+A(C+C')=BC+A$

- $X=A'B(D'+CD)+AB+A'BCD$
  $\rightarrow X=A'B((C+D')(D+D'))+AB+A'BCD=A'B(C+D')+AB+A'BCD$
  $\rightarrow X=A'BC+A'BD'+AB+A'BCD=A'BC(1+D)+AB+A'BD'$
  $\rightarrow X=A'BC+AB+A'BD'=B(A'C+A'D'+A)=B(A'C+(A+A')(A+D'))$
  $\rightarrow X=B(A'C+A+D')=B((A+A')(A+C)+D')=B(A+C+D')=AB+BC+BD'$

7. Which coding technique is good for error detection? Give example.

CRC is good for error detection rather than Parity code technique.
CRC (Cyclic Redundancy Check) is a widely used error detection technique used in digital communication and data storage systems which involves appending a specific checksum to a data message before transmission using a polynomial division.

Example:

Suppose we have a received message with an appended CRC code and we want to check if the message is error-free. We'll follow these steps:

1. **Received Message**: Let's say you received the following message along with the CRC code:

   Received Message: 10110101101 (message) 110 (CRC)

2. **Generator Polynomial**: You'll need to know the generator polynomial that was used to calculate the CRC. For this example, let's use the same CRC-3 polynomial, which is represented as " x^3+x+1 " and is equivalent to the binary "1011."

3. **Perform Polynomial Division**: Divide the received message (including the CRC code) by the generator polynomial using binary polynomial division. The goal is to check if the remainder is all zeros, which indicates an error-free message.

   10110101101110 (received message)
   1011 (generator polynomial)

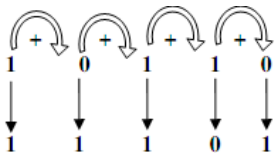
   000001001110 (remainder)

- **Error Detection**: If the remainder is all zeros (as it should be in an error-free message), then no error is detected. However, if the remainder is not all zeros, an error is present in the message, and it needs to be corrected.

In this example, the remainder is not all zeros (it contains some ones), which indicates that an error has been detected in the message.

8. Convert $(10110)_2=(?)_{gray}$ **

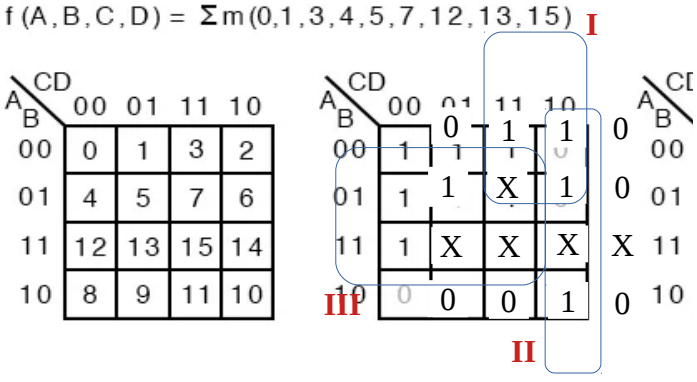
For changing binary number 10110 in its equivalent Gray code the rules are as, the left most bit that is MSB in Gray code is 1, similar as the left most bit in binary also add the MSB (1) to the adjacent bit (0) after that add the subsequent adjacent pair and discard the carry. Continue such process until completion.



Thus Gray equivalent is 11101 of Binary number 10110.

9. Minimize the following function: $f(a,b,c,d)=\sum_a(1,3,4,7,11)+\sum_d(5,12,13,14,15)$

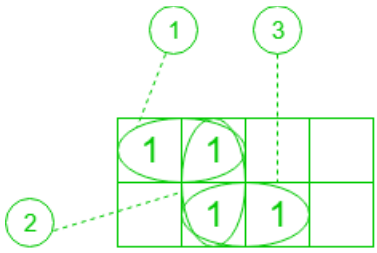
The given function $f(a,b,c,d)=\sum_a(1,3,4,7,11)+\sum_d(5,12,13,14,15)$ is being minimized using Karnaugh Map technique below:



So, $f(a,b,c,d)=I+II+III=cd+a'd+bc'$

10. Define prime implicant with example.

A prime implicant is a term or combination of literals in a Boolean expression that cannot be further reduced or combined with other implicants while covering the same set of minterms or don't-care conditions.



No. of Prime Implicants = 3

11. Define the following terms:

- Fan out

  Fan-out is a term used in digital electronics to describe the number of standard logic gates or other digital components that can be connected to the output of a particular gate or component without overloading it. It is a measure of the output's driving capability and is crucial for ensuring proper signal propagation and integrity in a digital circuit.

  Each digital gate or component has a specific fan-out rating, which indicates the maximum number of inputs it can drive while still maintaining the desired voltage levels. Exceeding the fan-out limit can lead to signal degradation, slower response times, or even incorrect logic levels at the output.

- Noise margin

  Noise margin refers to the difference between the minimum acceptable voltage level for a logical "0" and the maximum acceptable voltage level for a logical "1" in a digital circuit. It provides a safety margin against noise, electrical interference, and other factors that may cause signal distortion. Noise margin is essential to ensure that a digital circuit functions correctly, even in the presence of electrical noise.

  There are two types of noise margins:

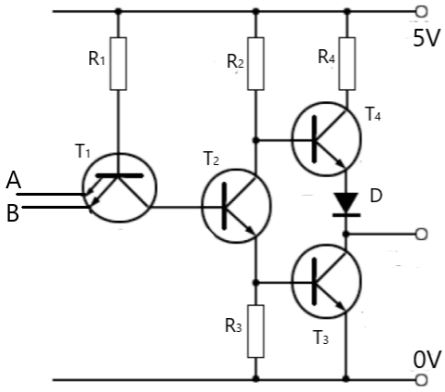
  - ***Low-Level Noise Margin (*** NM_L ***):*** This represents the difference between the minimum voltage level for a logical "0" (V_L) and the voltage level at which a signal is still recognized as a logical "0" (V_{IL}). Mathematically, $NM_L = V_{IL} - V_L$.
  - ***High-Level Noise Margin (*** NM_H ***):*** This represents the difference between the voltage level at which a signal is still recognized as a logical "1" (V_{IH}) and the maximum voltage level for a logical "1" (V_H). Mathematically, $NM_H = V_H - V_{IH}$.

    A larger noise margin indicates greater immunity to noise and interference, making the circuit more reliable. Designers strive to maximize both low-level and high-level noise margins to ensure robust digital circuit operation.

12. Draw and explain the circuit operation of 2 digit TTL NAND gate.

From the diagram, we shall explain the working. Now, as seen, the transistor T_1 has two emitters to allow two inputs into the transistor. Now, as connected the base voltage will be at 5V. If both inputs are logic 1 (usually means about 5V too), the potential difference across base and emitter would be zero or nearly. Hence, no current will flow and the transistor is turned off. So, the collector voltage would also be equal to about 5V. Hence, this potential can drive current through

the emitter of the transistor T_2 . This then will allow the collector voltage of the transistor T_2 to fall.
Now due to the current flowing through the emitter, there would be a voltage drop across the resistor R_3 . The desired voltage drop would be about 0.7V. As seen, this is the input of the transistor T_3 . Hence, the transistor is turned on. Due to saturation, the collector voltage will fall to about 0.2V which is a logic 0.



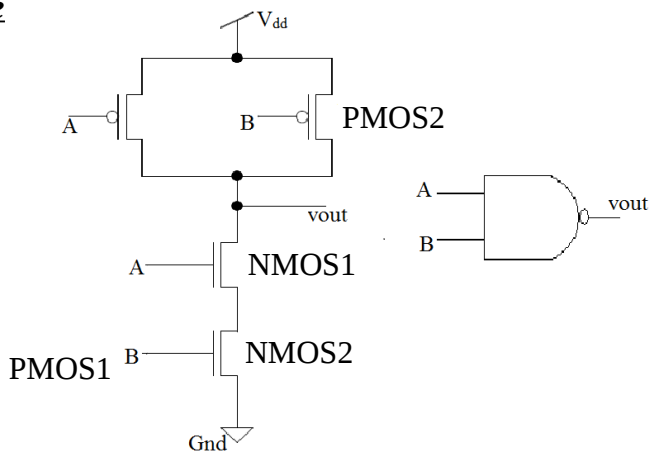
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

For the transistor T_4 , we can observe that the emitter voltage is made up of the entire voltage of the transistor T_3 plus the voltage drop across the diode D about 0.7V. Hence the emitter potential would be $0.7V+0.2V=0.9V$. Now the base voltage of the transistor T_4 , would be the voltage across the base-emitter of T_3 and the voltage of the entire transistor (i.e.) voltage across emitter-collector. This would also be equal to about 0.9V. Hence the emitter voltage and the collector voltage are equal. So the transistor T_4 will be turned off too. So the output is zero when both inputs are 1.

13. Design a two input CMOS NAND gate with necessary diagram and truth table.

Circuit and Truth table of 2 Input NAND Gate

V_A	V_B	V_{out}
Low	Low	High
Low	High	High
High	Low	High
High	High	Low



The above drawn circuit is a 2-input CMOS NAND gate. Now let's understand how this circuit will behave like a NAND gate. The circuit output should follow the same pattern as in the truth table for different input combinations.

Case-1: V_A – Low & V_B – Low As V_A and V_B both are low, both the pMOS will be ON and both the nMOS will be OFF. So the output V_{out} will get two paths through two ON pMOS to get connected with V_{dd}. The output will be charged to the V_{dd} level. The output line will not get any path to the GND as both the nMOS are off. So, there is no path through which the output line can discharge. The output line will maintain the voltage level at V_{dd}; so, High.

Case-2: V_A – Low & V_B – High

V_A – Low: pMOS1 – ON; nMOS1 – OFF

V_B – High: pMOS2 – OFF; nMOS2 – ON

pMOS1 and pMOS2 are in parallel. Though pMOS2 is OFF, still the output line will get a path through pMOS1 to get connected with V_{dd}. nMOS1 and nMOS2 are in series. As nMOS1 is OFF, so V_{out} will not be able to find a path to GND to get discharged. This in turn results the V_{out} to be maintained at the level of V_{dd}; so, High.

Case-3 : V_A – High & V_B – Low

V_A – High: pMOS1 – OFF; nMOS1 – ON

V_B – Low: pMOS2 – ON; nMOS2 – OFF

The explanation is similar as case-2. V_{out} level will be High.

Case-4 : V_A – High & V_B – High

V_A – High: pMOS1 – OFF; nMOS1 – ON

V_B – High: pMOS2 – OFF; nMOS2 – ON

In this case, both the pMOS are OFF. So, V_{out} will not find any path to get connected with V_{dd}. As both the nMOS are ON, the series connected nMOS will create a path from V_{out} to GND. Since, the path to ground is established, V_{out} will be discharged; so, Low.

In all the 4 cases we have observed that V_{out} is following the exact pattern as in the truth table for the corresponding input combination.

14. Perform subtraction using 2's complement method $85_{10} - 47_{10}$

Binary of 85_{10} ***and*** 47_{10} ***:***

$(85)_{10}=(\_\_\_\_\_)_2$

2	85	
2	42	1 ↑
2	21	0 ↑
2	10	1 ↑
2	5	0 ↑
2	2	1 ↑
2	1	0 ↑
	0	1 ↑

$\therefore (85)_{10}=(1010101)_2$

$(47)_{10} = (\_\_\_?\_\_\_)_2$

Solution:

$(47)_{10}=(\_\_\_\_\_)_2$

2	47	
2	23	1 ↑
2	11	1 ↑
2	5	1 ↑
2	2	1 ↑
2	1	0 ↑
	0	1 ↑

$\therefore (47)_{10}=(101111)_2$

+ 2's complement subtraction steps :

Here A = 1010101, B = 0101111.
Find A - B = ? using 2's complement
First find 2's complement of B = 0101111

Note : 2's complement of a number is 1 added to it's 1's complement number.

Step-1: 1's complement of 0101111 is obtained by subtracting each digit from 1

```
    1 1 1 1 1 1 1
-   0 1 0 1 1 1 1
-------------------
    1 0 1 0 0 0 0
```

Step-2: Now add 1 to the 1's complement to obtain the 2's complement :
1010000 + 1 = 1010001

Step-3: Now Add this 2's complement of B to A

```
1     1           1
    1 0 1 0 1 0 1
+   1 0 1 0 0 0 1
-----------------
1 0 1 0 0 1 1 0
```

$(0100110)_2$=(_____)$_{10}$

0100110

$= 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 0 \times 64 + 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$

$= 0 + 32 + 0 + 0 + 4 + 2 + 0$

$= 38$

∴ $(0100110)_2=(\underline{38})_{10}$

15. Convert $(541.203)_6$ to base 2, base 5, base 8, base 10 and base 16.*

First convert base-6 to decimal
$(541.203)_6$=(_____)$_{10}$

541.203

$$= 5 \times 6^2 + 4 \times 6^1 + 1 \times 6^0 + 2 \times \frac{1}{6^1} + 0 \times \frac{1}{6^2} + 3 \times \frac{1}{6^3}$$

$$= 5 \times 36 + 4 \times 6 + 1 \times 1 + 2 \times 0.166667 + 0 \times 0.027778 + 3 \times 0.00463$$

$$= 180 + 24 + 1 + 0.333333 + 0 + 0.013889$$

$$= 205.347222$$

∴ $(541.203)_6=(\underline{205.347222})_{10}$

Now convert decimal to base-5
$(205.347222)_{10}$=(_____)$_5$

5	205	
5	41	0 ↑
5	8	1 ↑
5	1	3 ↑
	0	1 ↑

0.347222 × 5 = 1.736110 ⇒ 1 ↓
0.736110 × 5 = 3.680550 ⇒ 3 ↓
0.680550 × 5 = 3.402750 ⇒ 3 ↓
0.402750 × 5 = 2.13750 ⇒ 2 ↓
0.13750 × 5 = 0.68750 ⇒ 0 ↓
0.68750 × 5 = 0.343750 ⇒ 0 ↓

∴ $(205.347222)_{10}=(\underline{1310.133200})_5$

∴ $(541.203)_6=(\underline{1310.133200})_5$

Now convert decimal to octal
$(205.347222)_{10}$=(_____)$_8$

8	205	
8	25	5 ↑
8	3	1 ↑
	0	3 ↑

0.347222 × 8 = 2.777776 ⇒ 2 ↓
0.777776 × 8 = 6.222208 ⇒ 6 ↓
0.222208 × 8 = 1.777664 ⇒ 1 ↓
0.777664 × 8 = 6.221312 ⇒ 6 ↓
0.221312 × 8 = 1.770496 ⇒ 1 ↓
0.770496 × 8 = 6.163968 ⇒ 6 ↓

∴ $(205.347222)_{10}=(\underline{315.261616})_8$

∴ $(541.203)_6=(\underline{315.261616})_8$

Now convert decimal to binary
$(205.347222)_{10}$=(_____)$_2$

2	205	
2	102	1 ↑
2	51	0 ↑
2	25	1 ↑
2	12	1 ↑
2	6	0 ↑
2	3	0 ↑
2	1	1 ↑
	0	1 ↑

0.347222 × 2 = 0.694444 ⇒ 0 ↓
0.694444 × 2 = 1.388888 ⇒ 1 ↓
0.388888 × 2 = 0.777776 ⇒ 0 ↓
0.777776 × 2 = 1.555552 ⇒ 1 ↓
0.555552 × 2 = 1.111104 ⇒ 1 ↓
0.111104 × 2 = 0.222208 ⇒ 0 ↓

∴ $(205.347222)_{10}=(\underline{11001101.010110})_2$

∴ $(541.203)_6=(\underline{11001101.010110})_2$

Now convert decimal to hexadecimal
$(205.347222)_{10}$=(_____)$_{16}$

16	205	
16	12	D ↑
	0	C ↑

0.347222 × 16 = 5.555552 ⇒ 5 ↓
0.555552 × 16 = 8.888832 ⇒ 8 ↓
0.888832 × 16 = 14.221312 ⇒ E ↓
0.221312 × 16 = 3.540992 ⇒ 3 ↓
0.540992 × 16 = 8.655872 ⇒ 8 ↓
0.655872 × 16 = 10.493952 ⇒ A ↓

∴ $(205.347222)_{10}=(\underline{CD.58E38A})_{16}$

∴ $(541.203)_6=(\underline{CD.58E38A})_{16}$

16. Represent the decimal number 28 to excess-3 and BCD code.

Decimal 28 to Excess-3:

Solution:

$(28)_{10}=(\_\_\_\_\_)_{XS3}$

2 8
+3 +3
=5 =11
0101 1011

$\therefore (28)_{10}=(01011011)_{XS3}$

Decimal 28 to BCD:

Solution:

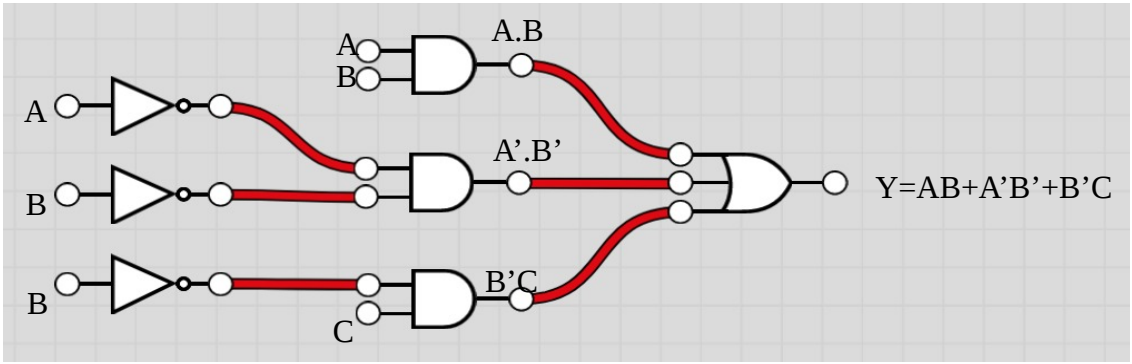
$(28)_{10}=(\_\_\_\_\_)_{BCD}$

2 8
0010 1000

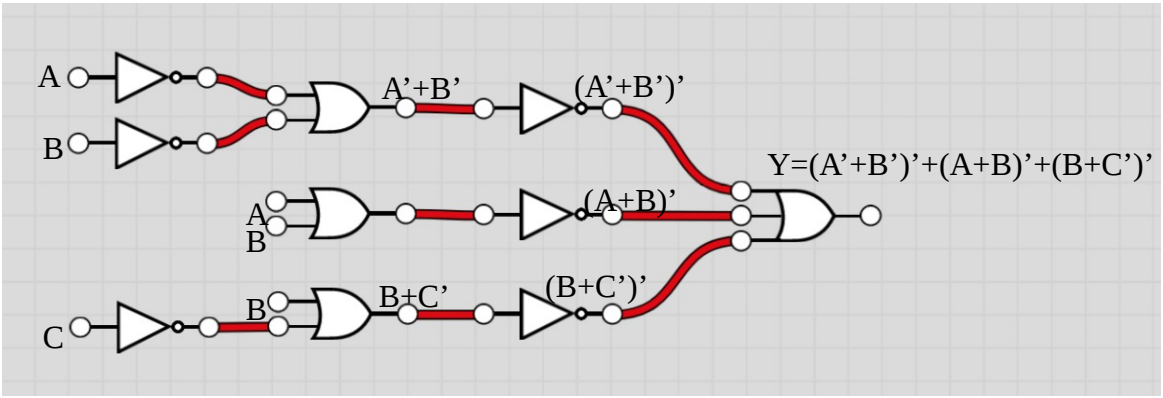
$\therefore (28)_{10}=(00101000)_{BCD}$

17. Given the boolean function $K=AB+A'B'+B'C$ Then-

- Implement it with AND, OR and NOT gates.



- Implement it with only OR and NOT gates.



18. Show that the dual of the exclusive-OR is equal to its complement.

A XOR B = A'B + AB'

(A XOR B)' = AB + A'B' .....(1)

In dual we replace AND with OR and OR with AND.

Dual of (A XOR B) = (A'+B) (A+B') = A'A +A'B' +AB+BB' = AB + A'B'..... (2)

from (1) and (2)

Dual of XOR = Complement of XOR

19. Simplify the following expression $z = A'BC' + ABC' + BC'D$

$Z = A'BC' + ABC' + BC'D = BC'(A + A' + D) = BC'(1 + D) = BC'.1 = BC'$

20. What are don't care terms? Why they're used? Explain with example.

Don't Care terms in a Karnaugh Map are input combinations for which the output value is not specified and can be either 0 or 1, as these combinations do not affect the desired behavior of the logic circuit.

Significance of "Don't Care" Conditions:

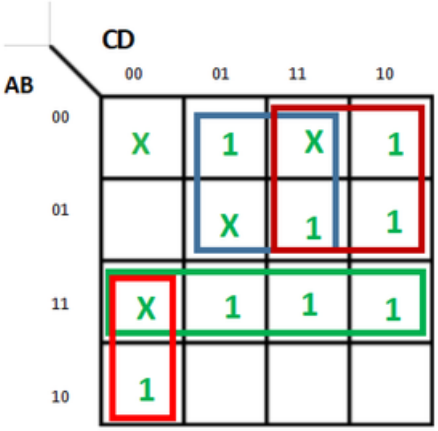
1. ***Simplification of the output:*** These conditions denotes inputs that are invalid for a given digital circuit. Thus, they can used to further simplify the boolean output expression of a digital circuit.
2. ***Reduction in number of gates required:*** Simplification of the expression reduces the number of gates to be used for implementing the given expression. Therefore, don't cares make the digital circuit design more economical.
3. ***Reduced Power Consumption:*** While grouping the terms along with don't cares reduces switching of the states. This decreases the memory space that is required to represent a given digital circuit which in turn results in less power consumption.
4. ***Represent Invalid States in Code Converters:*** These are used in code converters. For example- In design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don't cares.
5. **Prevention of Hazards in Digital Circuits:** Don't cares also prevents hazards in digital systems.

For example, minimise the following function in SOP minimal form using K-Maps:

$F(A,B,C,D) = m(1,2,6,7,8,13,14,15) + d(0,3,5,12)$

Explanation:

The SOP K-map for the given expression is:



Therefore, $f = AC'D' + A'D + A'C + AB$

21. What is parity bit? Design 3-bit odd parity generator and checker.

A parity bit is an additional bit used in digital communications to detect errors in transmitted data. It is a simple form of error-checking that helps ensure data integrity.

There are two types of parity: even parity and odd parity. In odd parity, the number of 1s in a data word, including the parity bit, is always maintained as an odd number. To design a 3-bit odd parity generator and checker, we'll start by creating the generator, which adds a

parity bit to a 3-bit data word to ensure it has odd parity. Then, we'll design the checker, which verifies if the received data has odd parity.

3-Bit Odd Parity Generator

For a 3-bit odd parity generator, you can use the following truth table:

D_2	D_1	D_0	$P(Parity\ Bit)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

The parity bit 'P' is calculated as the XOR of the three data bits (D2, D1, D0).

3-Bit Odd Parity Checker

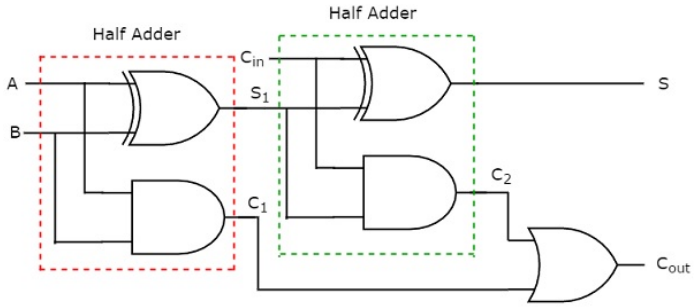
The checker will verify whether the received data has odd parity. To do this, calculate the parity of the received 4-bit word (3 data bits and 1 parity bit). If the parity is odd for example there is an odd number of 1s, then the data is considered valid. If the parity is even for example there is an even number of 1s, it indicates a transmission error.

Here's a truth table for the checker:

D_2	D_1	D_0	$P(Received\ Parity\ Bit)$	*Valid Data*
0	0	0	1	Yes
0	0	1	0	No
0	1	0	0	No
0	1	1	1	Yes
1	0	0	0	No
1	0	1	1	Yes
1	1	0	1	Yes
1	1	1	0	No

To check for odd parity, simply count the number of 1s in the received 4-bit word and determine if it's odd or even. If it's odd, the data is valid; if it's even, there's an error.

22. Implement a full adder using two half adder.

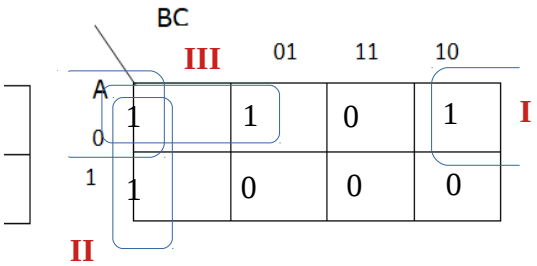


23. Design a logic circuit whose output is HIGH only when a majority of inputs A, B and C are LOW.

Truth table according to the condition:

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

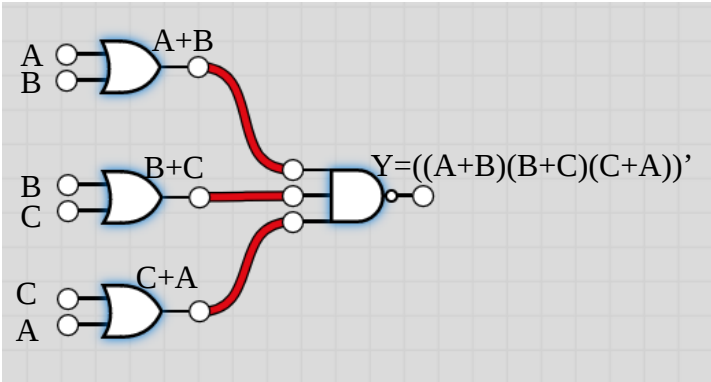
Implementing the truth table in a K-Map for minimization:



$F = I + II + III = A'B' + B'C' + C'A'$

$\rightarrow F = ((A'B' + B'C' + C'A')')' = ((A'B')'.(B'C')'.(C'A')')' = ((A+B)(B+C)(C+A))'$

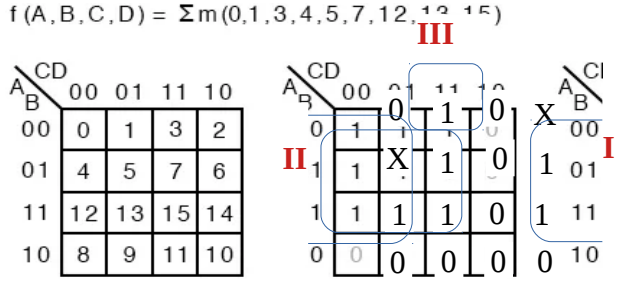
Logic Gate Design:



24. Simplify the following Boolean function using K-map and realize with basic gates:

$F(A,B,C,D)=\sum_m(1,5,6,12,13,14)+\sum_d(2,4)$

The given function $F(A,B,C,D)=\sum_m(1,5,6,12,13,14)+\sum_d(2,4)$ is being minimized using Karnaugh Map technique below:



So, $F(A,B,C,D)=I+II+III=BD'+BC'+A'B'C'D$

25. Find $(45)_{10}-(83)_{12}$ using two's complement format with 8-bit numbers. Then convert your result back to decimal.

$(45)_{10}$=(_____)$_2$

2	45	
2	22	1 ↑
2	11	0 ↑
2	5	1 ↑
2	2	1 ↑
2	1	0 ↑
	0	1 ↑

∴ $(45)_{10}=(\underline{101101})_2$

$(83)_{12}$=(_____)$_2$

First convert base-12 to decimal

$(83)_{12}$=(_____)$_{10}$

83

$= 8 \times 12^1 + 3 \times 12^0$

$= 8 \times 12 + 3 \times 1$

$= 96 + 3$

$= 99$

∴ $(83)_{12}=(\underline{99})_{10}$

Now convert decimal to binary

$(99)_{10}$=(_____)$_2$

∴ $(83)_{12}=(\underline{99})_{10}$

Now convert decimal to binary

$(99)_{10}$=(_____)$_2$

2	99	
2	49	1 ↑
2	24	1 ↑
2	12	0 ↑
2	6	0 ↑
2	3	0 ↑
2	1	1 ↑
	0	1 ↑

∴ $(99)_{10}=(\underline{1100011})_2$

∴ $(83)_{12}=(\underline{1100011})_2$

+ 2's complement subtraction steps :

Here A = 0101101, B = 1100011.
Find A - B = ? using 2's complement
First find 2's complement of B = 1100011

Note : 2's complement of a number is 1 added to it's 1's complement number.

Step-1: 1's complement of 1100011 is obtained by subtracting each digit from 1

```
  1 1 1 1 1 1 1
- 1 1 0 0 0 1 1
---------------
  0 0 1 1 1 0 0
```

Step-2: Now add 1 to the 1's complement to obtain the 2's complement :
0011100 + 1 = 0011101

Step-3: Now Add this 2's complement of B to A

```
  1 1 1 1   1
  0 1 0 1 1 0 1
+ 0 0 1 1 1 0 1
---------------
  1 0 0 1 0 1 0
```

+ Hints : (Move mouse over the steps for detail calculation highlight)

```
---------------
  1 0 0 1 0 1 0
```

+ Hints : (Move mouse over the steps for detail calculation highlight)

Here there is no carry, answer is - (2's complement of the sum obtained 1001010)

Note : 2's complement of a number is 1 added to it's 1's complement number.

Step-1: 1's complement of 1001010 is obtained by subtracting each digit from 1

```
  1 1 1 1 1 1 1
- 1 0 0 1 0 1 0
---------------
  0 1 1 0 1 0 1
```

Step-2: Now add 1 to the 1's complement to obtain the 2's complement :
0110101 + 1 = 0110110
So answer is -0110110

To confirm the result is accurate we can again perform 2's complement on the result.

$(0110110)_2=(\_\_\_\_\_)_{10}$

0110110

$= 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 0 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$

$= 0 + 32 + 16 + 0 + 4 + 2 + 0$

$= 54$

$\therefore (0110110)_2=(\underline{54})_{10}$

So, the result is -54 in decimal.

26. Add $(65)_{10}+(72)_{10}$ using 8-bit sign-magnitude format for the numbers. Convert your result to decimal. Is your answer correct? Why or why not?

65_10 =(01000001)_2
72_10 =(01001000)_2

_____
-9_10=(10001001)_2

The result we should have gotten is 137_10. So, the result isn't correct because of carry flag of a carry.
In an 8-bit sign-magnitude format, the largest positive number we can represent is 01111111 (127 in decimal), and the smallest negative number is 10000000 (-127 in decimal).
In our case, the result, 10001001, represents a negative number, and when we consider the magnitude part (the last 7 bits), which is 0001001, it converts to 9 in decimal.The addition of 65 and 72 in 8-bit sign-magnitude format resulted in a carry flag due to a carry occurring during addition, but there is no overflow because the result is well within the representable range for the given format.

27. Convert $(1000\,0110)_{BCD}$ to decimal, binary and octal.

Solution:

$(10000110)_{BCD}=(\_\_\_\_\_)_2$

1. Convert BCD to decimal
$(10000110)_{BCD}=(\_\_\_\_\_)_{10}$

1000 0110
8 6

$\therefore (10000110)_{BCD}=(\underline{86})_{10}$

2. Convert decimal to binary
$(86)_{10}=(\_\_\_\_\_)_2$

2	86	
2	43	0 ↑
2	21	1 ↑
2	10	1 ↑
2	5	0 ↑
2	2	1 ↑
2	1	0 ↑
	0	1 ↑

$\therefore (86)_{10}=(\underline{1010110})_2$

$\therefore (10000110)_{BCD}=(\underline{1010110})_2$

2. Convert decimal to octal
$(86)_{10}=(\_\_\_\_\_)_8$

8	86	
8	10	6 ↑
8	1	2 ↑
	0	1 ↑

$\therefore (86)_{10}=(\underline{126})_8$

$\therefore (10000110)_{BCD}=(\underline{126})_8$

28. Simplify $Z = A'C(A'BD)' + A'BC'D' + AB'$ using Boolean algebra.

$Z = A'C(A'BD)' + A'BC'D' + AB' = A'C((A')' + B' + D') + A'BC'D' + AB'$

$\rightarrow z = A'B'C + A'CD' + A'BC'D' + AB' = B'(A'C + A) + A'D'(C + BC')$

$\rightarrow z = B'((A + A')(A + C)) + A'D'((B + C)(C + C')) = B'(A + C) + A'D'(B + C)$

$therefore, z = AB' + B'C + A'BD' + A'CD'$

29. State and prove De Morgan's theorems with the help of truth tables.**

De Morgan's first Theorem: It states that the complement of a logical OR (NOR) operation is equivalent to the logical AND of the complements of the individual variables. Mathematically, it can be expressed as: $\overline{A + B} = \overline{A}\,\overline{B}$

De Morgan's second Theorem: It states that the complement of a logical AND (NAND) operation is equivalent to the logical OR of the complements of the individual variables. Mathematically, it can be expressed as: $\overline{A\,B} = \overline{A} + \overline{B}$

First De Morgan's Law Truth Table

The truth table for first De Morgan's Law is given as follows:

A	B	A + B	(A + B)'	A'	B'	A'. B'
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

From the table we can clearly see that, $\overline{A + B} = \overline{A}\,\overline{B}$

Second De Morgan's Law Truth Table

The truth table for the second De Morgan's Law is given as follows:

A	B	A . B	(A. B)'	A'	B'	A' + B'
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

From the table we can clearly see that, $\overline{A\,B} = \overline{A} + \overline{B}$

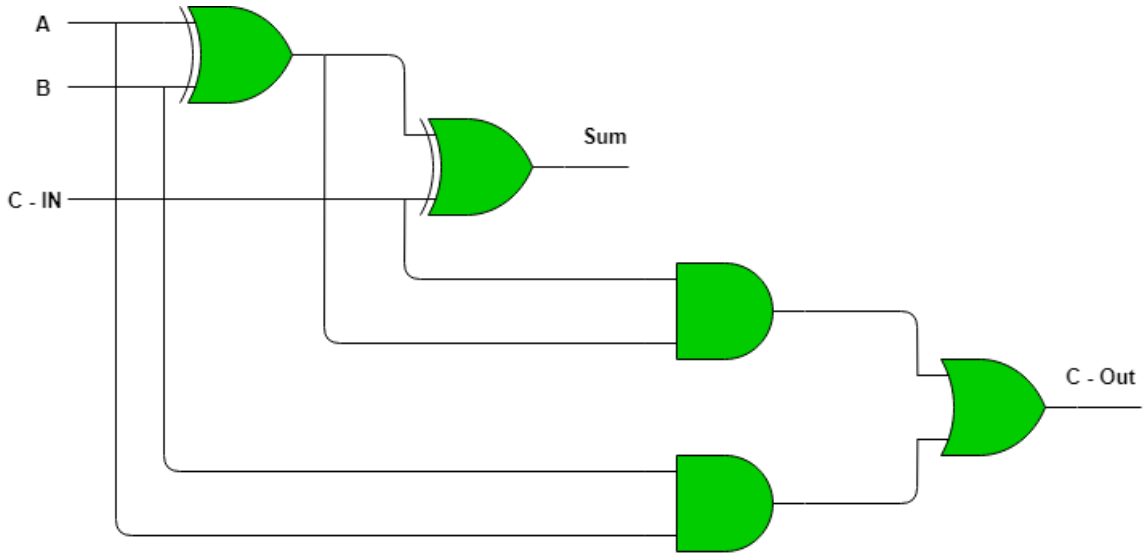
30. Design and explain a full adder in detail with circuit diagram and truth table.

Full Adder is the adder that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-in. The output carry is designated as C-out and the normal output is designated as S which is sum. The C-out is also known as the majority 1's detector, whose output goes high when more than one input is high.

Full adder truth table:

Inputs			Outputs	
A	B	C – IN	Sum	C - Out
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

Full adder logic circuit:



31. Design a combinational logic circuit to compare two 2-bit binary numbers A and B and to generate the outputs A< B, A=B and A>B. Is there a way to derive the third output from the first two outputs?

2-Bit Magnitude Comparator

A comparator used to compare two binary numbers each of two bits is called a 2-bit Magnitude comparator. It consists of four inputs and three outputs to generate less than, equal to, and greater than between two binary numbers.

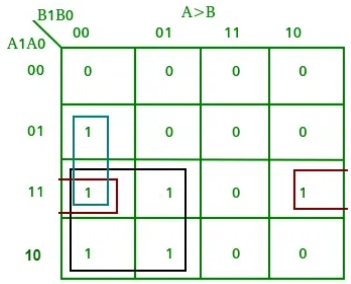
The truth table for a 2-bit comparator is given below.

	INPUT			OUTPUT		
A1	**A0**	**B1**	**B0**	**A<B**	**A=B**	**A>B**
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1

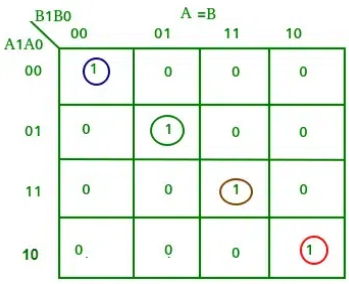
	INPUT				OUTPUT	
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

From the above truth table, K-map for each output can be drawn as follows.

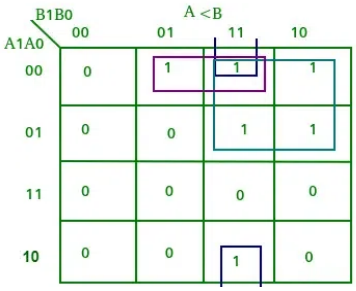
Truth Table of Output A>B



Truth Table of Output A=B



Truth Table of Output A<B



From the above K-maps logical expressions for each output can be expressed as follows:

A>B : A1B1’ + A0B1’B0’ + A1A0B0’

A=B : A1’A0’B1’B0’ + A1’A0B1’B0 + A1A0B1B0 + A1A0’B1B0’

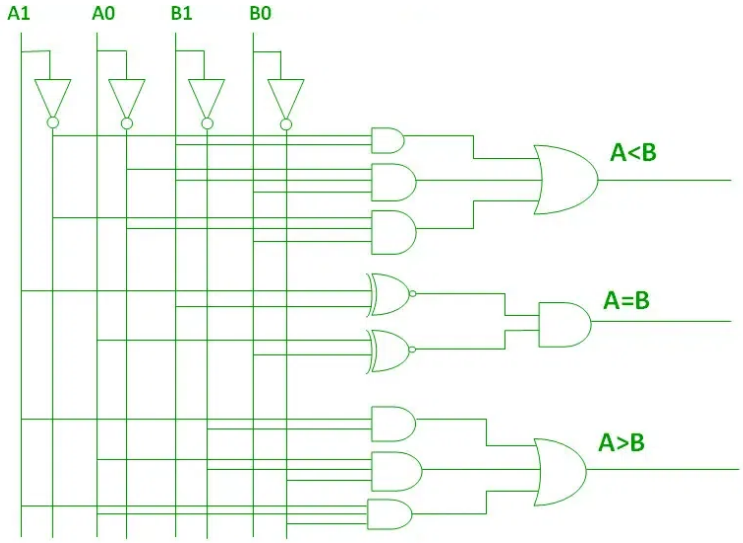
: A1’B1’ (A0’B0’ + A0B0) + A1B1 (A0B0 + A0’B0’)

: (A0B0 + A0’B0’) (A1B1 + A1’B1’)

: (A0 Ex-Nor B0) (A1 Ex-Nor B1)

A<B :A1’B1 + A0’B1B0 + A1’A0’B0

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below.



We can derive the third output from the first two outputs. We can put first two output as input in a NOR gate and the resultant output is A>B because if both A = B and A < B are true, it implies that A is not greater than B, but you can conclude A > B is false. Again if A=B and A>B is false then, A>B is true.
We can see it from the truth table:

Input		**Output**
A<B	**A=B**	**A>B**
0	0	1
0	1	0
1	0	0
1	1	0

32. Simplify the following Boolean expression using Quine-McCluskey technique
$f(A,B,C,D)=\Sigma_m(0,1,3,7,8,9,11,15)$ ***

Representing the minterms in terms of binary:

	Binary			
Minterms	A	B	C	D
0	0	0	0	0
1	0	0	0	1
3	0	0	1	1
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
11	1	0	1	1
15	1	1	1	1

Step-01: Grouping the minterms in terms of the number of 1's they have:

		Binary			
Groups	Minterms	A	B	C	D
0	0	0	0	0	0
1	1	0	0	0	1
	8	1	0	0	0
2	3	0	0	1	1
	9	1	0	0	1
3	7	0	1	1	1
	11	1	0	1	1
4	15	1	1	1	1

Step-02: Comparing adjacent group and find if it differs by only one bit and put '_' where the bit differs.

		Differ Bits			
Groups	Minterms	A	B	C	D
0 and 1	0,1	0	0	0	_
	0,8	_	0	0	0
1 and 2	1,3	0	0	_	1
	1,9	_	0	0	1
	8,9	1	0	0	_
2 and 3	3,7	0	_	1	1
	3,11	_	0	1	1
	9,11	1	0	_	1
3 and 4	7,15	_	1	1	1
	11,15	1	_	1	1

Prime Implicants

Prime Implicants

Prime Implicants

Prime Implicants

Step-03: Repeating the same process in step-02.

		Differ Bits			
Groups	Minterms	A	B	C	D
(0,1) and (1,2)	0,1,8,9	_	0	0	_
(1,2) and (2,3)	1,3,9,11	_	0	_	1
(2,3) and (3,4)	3,7,11,15	_	_	1	1

Prime Implicants

Step-04: List all the prime implicants.

Minterms	Prime Implicants
0,8	B'C'D'
1,9	B'C'D
3,11	B'CD
11,15	ACD
0,1,8,9	B'C'
1,3,9,11	B'D
3,7,11,15	CD

Step-05: Further minimize the prime implicants using PI Chart.

Minterms	PI	0	1	3	7	8	9	11	15
0,8	B'C'D'	×				×			
1,9	B'C'D		×				×		
3,11	B'CD			×				×	
11,15	ACD							×	×
0,1,8,9	B'C'	×	×			×	×		
1,3,9,11	B'D		×	×			×	×	
3,7,11,15	CD			×	×			×	×

Here, (3,7,11,15) is essential prime implicant. So it must be considered and this implicant covers 3,7,11,15. Now we have to choose prime implicants such that they covers the remaining minterms.
As we can see from the chart (0,1,8,9) prime imolicant covers the remaining minterms.
So, F= CD+B'C' [ans]

33. List out the advantages and disadvantages of Quine-McCluskey Method.

Advantages

- **Guaranteed Minimal Expression:** The Quine-McCluskey method guarantees that it will find the minimal sum-of-products (SOP) or product-of-sums (POS) expression, which means the simplest form of the Boolean function.
- **Systematic Approach:** It follows a systematic and algorithmic approach to simplify Boolean expressions. This makes it suitable for automation and computer-based simplification.
- **Useful for Larger and Complex Expressions:** It is especially useful when dealing with complex Boolean expressions with a large number of variables and min-terms.

Disadvantages

- **Complexity:** The method can become increasingly complex and time-consuming as the number of variables and min-terms in the expression increases.
- **Manual Grouping:** While the method can be automated, manual grouping of min-terms is often required, especially for expressions with a large number of variables.
- **Limited to Canonical Forms:** The Quine-McCluskey method is primarily used to find the canonical (standard) form of a Boolean expression, which may not be the most intuitive representation.
- **Resource-Intensive:** For very large problems, the method can be resource-intensive in terms of computational power and memory, potentially leading to inefficiencies.

34. Do the following conversion:

- $(378)_{10}$ to 16 bit binary number.

Solution:

$(378)_{10}$=(_____)$_2$

2	378	
2	189	0 ↑
2	94	1 ↑
2	47	0 ↑
2	23	1 ↑
2	11	1 ↑
2	5	1 ↑
2	2	1 ↑
2	1	0 ↑
	0	1 ↑

∴ $(378)_{10}=(101111010)_2$

- $(1010110.1100)_2=(?)_{10}$

Solution:

$(1010110.1100)_2$=(_____)$_{10}$

1010110.1100

$$= 1\times 2^6 + 0\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0 + 1\times \frac{1}{2^1} + 1\times \frac{1}{2^2} + 0\times \frac{1}{2^3} + 0\times \frac{1}{2^4}$$

$$= 1\times 64 + 0\times 32 + 1\times 16 + 0\times 8 + 1\times 4 + 1\times 2 + 0\times 1 + 1\times 0.5 + 1\times 0.25 + 0\times 0.125 + 0\times 0.0625$$

$$= 64 + 0 + 16 + 0 + 4 + 2 + 0 + 0.5 + 0.25 + 0 + 0$$

$$= 86.75$$

∴ $(1010110.1100)_2=(86.75)_{10}$

- $(10101100)_2=(?)_8$

Solution:

$(10101100)_2$=(_____)$_8$

010	101	100
2	5	4

∴ $(10101100)_2=(254)_8$

- $(743)_{16}=(?)_2$

Solution:

$(743)_{16}$=(_____)$_2$

7	4	3
0111	0100	0011

∴ $(743)_{16}=(11101000011)_2$

35. Compare between BCD and Binary Code.

Here's a comparison between BCD and binary code:

Components	Binary	BCD
1. Number Representation	• Represents numbers in base-2 (binary) form. • Each digit is a power of 2, with only two possible values, 0 and 1. • Suited for general-purpose digital computation and arithmetic.	• Represents numbers in decimal form, where each digit is in the range 0-9. • Each decimal digit is typically represented using 4 binary bits, known as a nibble. • Suited for applications where decimal representation is important, such as in financial and calculator systems.
2. Density	• Binary representation is more space-efficient for representing numbers, as each bit has only two possible states. • More compact for storage and arithmetic operations.	• BCD representation is less space-efficient because it uses 4 bits to represent each decimal digit (0-9). • Requires more storage space than a pure binary representation.
3. Arithmetic Operations	• Well-suited for binary arithmetic operations, such as addition, subtraction, multiplication, and division. • Commonly used in computers for all mathematical calculations.	• Suited for decimal arithmetic operations but can be less efficient than binary arithmetic. • Often used in applications where exact decimal representation is required, such as financial calculations.
4. Range	• Represents numbers in a wider range, using only 0 and 1. • Suitable for a broader range of applications.	• Represents numbers in the limited range of 0-9 for each digit. • Primarily used for representing decimal digits.
5. Error Detection	• May not inherently detect decimal-related errors, as it doesn't differentiate between valid and invalid decimal representations.	• Provides some inherent error detection, as it ensures that each digit is within the valid decimal range (0-9).
6. Human Readability	• Not human-friendly for direct interpretation, as it requires conversion to decimal for understanding.	• More human-readable because it directly represents decimal digits.

36. What is Gray code? Explain with examples, how do you convert binary to Gray and Gray to binary?**

Gray code

Gray code, also known as reflected binary code or unit distance code, is a binary numeral system where two successive values differ in only one bit position. Gray code is often used in various applications, including rotary encoders, error detection, and analog-to-digital conversion.

Converting Binary to Gray Code

Converting a binary number to Gray code involves a process called "reflecting."
Here's a step-by-step example: Convert the binary number 1101 to Gray code.

Step-01: Start with the most significant bit (leftmost bit), which remains the same in Gray code.
Binary: **1**101
Gray: **1**???

Step-02: For the remaining bits, apply the XOR operation to each bit with the bit to its left.
Binary: 1**1**01
Gray: 1**0**??

Step-03: Continue this process for all bits.
Binary: 11**0**1
Gray: 10**0**0

So, the Gray code representation of the binary number 1101 is 1000.

Converting Gray Code to Binary

Converting Gray code to binary is done by reversing the process. Given a Gray code, you can obtain the corresponding binary representation as follows:
Example: Convert the Gray code 1000 to binary.

Step-01: Start with the most significant bit (leftmost bit), which remains the same in binary.
Gray: **1**???
Binary: **1**???

Step-02: For the remaining bits, apply the XOR operation to each bit with the previously determined bit.
Gray: 1**0**??
Binary: 10**1**?

Step-03: Continue this process for all bits.
Gray: 100**0**
Binary: 101**1**

So, the binary representation of the Gray code 1000 is 1011.

37. Subtract $(100111)_2$ from $(001100)_2$ using 2's complement method. Why do you need 2's complement method?

Solution:

+ 2's complement subtraction steps :

Here A = 100111, B = 001100.
Find A - B = ? using 2's complement
First find 2's complement of B = 001100

Note : 2's complement of a number is 1 added to it's 1's complement number.

Step-1: 1's complement of 001100 is obtained by subtracting each digit from 1

```
    1  1  1  1  1  1
-   0  0  1  1  0  0
-------------------------
    1  1  0  0  1  1
```

Step-2: Now add 1 to the 1's complement to obtain the 2's complement :
110011 + 1 = 110100

Step-3: Now Add this 2's complement of B to A

```
1           1
    1  0  0  1  1  1
+   1  1  0  1  0  0
-------------------------
1   0  1  1  0  1  1
```

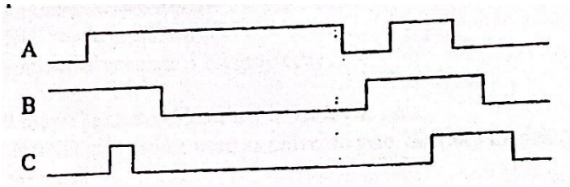
+ Hints : (Move mouse over the steps for detail calculation highlight)

The left most bit of the result is 1, called carry and it is ignored.
So answer is 011011

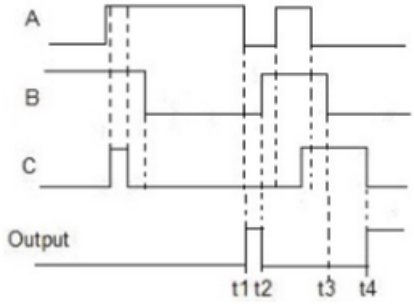
Here are several reasons why the 2's complement method is important and necessary:

- **Representation of Negative Numbers:** The 2's complement method is a widely used technique for representing negative numbers in binary form. It allows both positive and negative numbers to be represented.
- **Addition and Subtraction:** 2's complement simplifies the addition and subtraction of signed binary numbers. In 2's complement representation, subtraction is performed by adding the additive inverse, making it consistent with addition.
- **Efficient Arithmetic Circuits:** Digital arithmetic circuits, such as adders and subtractors, can be designed more efficiently using 2's complement representation. It reduces the need for complex logic to handle sign and magnitude, leading to faster and more compact circuitry.
- **Overflow Handling:** The 2's complement method allows for easy detection and handling of overflow in arithmetic operations. Overflow occurs when the result of an operation is too large to be represented using the available number of bits. 2's complement simplifies the detection of overflow, making it an important aspect of error handling.

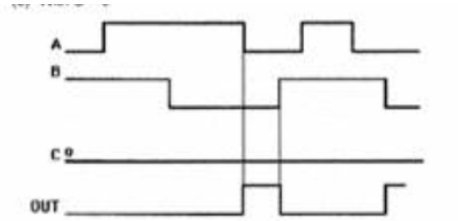
38. Apply the input waveforms of fig-1 to a NOR gate, and draw the output waveform. Then repeat the output waveform with C hold permanently LOW.



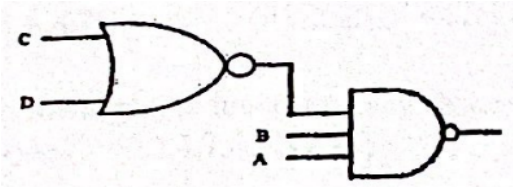
Applying the waveform of fig-1 to a NOR gate we get the output waveform:



Holding C permanently low and applying the waveform to a NOR gate, we get the output waveform:



39. Determine the truth table for the circuit of fig-2.



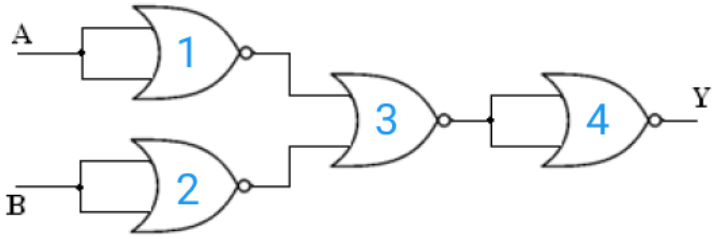
At first we should minimize the logic gates to determine a truth table. As we can see from the circuit,
Output X = ((C+D)'AB)'
Applying De Morgan's second law, we get,
X=((C+D)')'+A'+B'=A'+B'+C+D.
So, the truth table for the Output is:

Input						Output
A	A'	B	B'	C	D	X=A'+B'+C+D
0	1	0	1	0	0	1
0	1	0	1	0	1	1
0	1	0	1	1	0	1
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	0	1	1

0	1	1	0	1	0	1
0	1	1	0	1	1	1
1	0	0	1	0	0	1
1	0	0	1	0	1	1
1	0	0	1	1	0	1
1	0	0	1	1	1	1
1	0	1	0	0	0	0
1	0	1	0	0	1	1
1	0	1	0	1	0	1
1	0	1	0	1	1	1

40. Show that a two-input NAND gate can be constructed from two-input NOR gate. Simplify the following $(A+B)'(A'+B')'$

NOR gate is known as Universal Gate meaning every gate like NAND, Ex-OR etc even basic gates like AND, OR and NOT gate can also be constructed from NOR gate.
Here's logic circuit to construct NAND gate from NOR gate:



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Simplifying the logic gate: $(A+B)'(A'+B')'=(A'.B')((A')'.(B')')=(A'.B')(A.B)=0.$

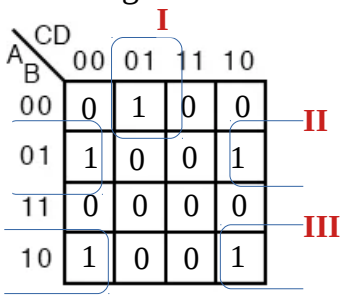
41. Simplify the expression $x=A'B'C'+A'BC+ABC+AB'C'+AB'C$ using Boolean algebra.

$X=A'B'C'+A'BC+ABC+AB'C'+AB'C=B'C'(A'+A)+BC(A+A')+AB'C$

$\rightarrow X=B'C'+BC+AB'C=BC+B'(AC+C')=BC+B'(A+C')=AB'+BC+B'C'$

42. Minimize the Boolean function $f(a,b,c,d)=\sum_M(1,4,6,8,10)$

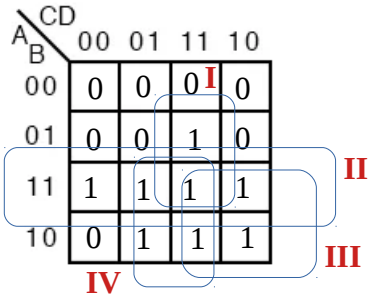
Minimizing the boolean function using K-Map:



$F=I+II+III=A'B'C'D+A'BD'+AB'D'$

43. Simplify the following expression using K-map

$f(A,B,C,D)=\sum_M(7,9,10,11,12,13,14,15)$



$F=I+II+III+IV=BCD+AB+AC+AD$

44. Design the circuit corresponding to the truth table shown in Table-1.

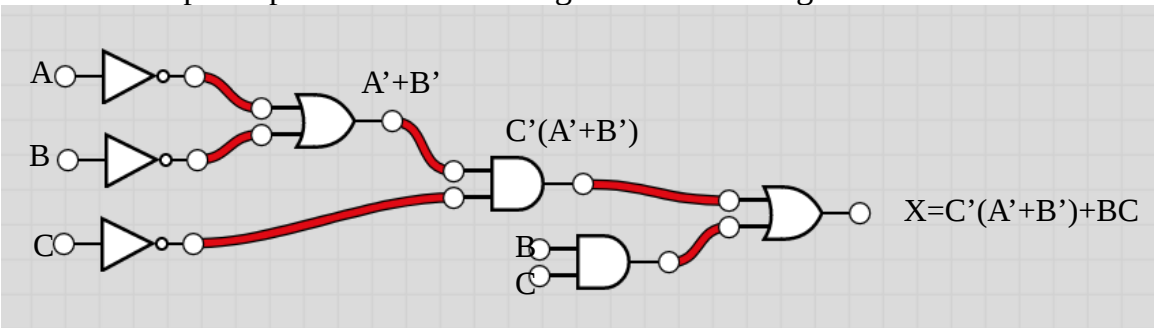
A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

From the truth table we can write,

$X=A'B'C'+A'BC'+A'BC+AB'C'+ABC=A'C'(B'+B)+BC(A'+A)+AB'C'$

$\rightarrow X=A'C'+BC+AB'C'=C'(A'+AB')+BC=C'(A'+B')+BC=A'C'+BC+B'C'$

From the output expression we can design this circuit diagram:



45. Do the following conversions:

- $(10101011.1101)_2=(?)_{10}$

10101011.1101

$= 1\times2^7+0\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0+1\times\frac{1}{2^1}+1\times\frac{1}{2^2}+0\times\frac{1}{2^3}+1\times\frac{1}{2^4}$

$= 1\times128+0\times64+1\times32+0\times16+1\times8+0\times4+1\times2+1\times1+1\times0.5+1\times0.25+0\times0.125+1\times0.0625$

$= 128+0+32+0+8+0+2+1+0.5+0.25+0+0.0625$

$= 171.8125$

$\therefore (10101011.1101)_2=(171.8125)_{10}$

- $(3AE8F.2D)_{16}=(?)_8$

First convert hexadecimal to binary
$(3AE8F.2D)_{16}=(\_\_\_\_\_)_2$

3	A	E	8	F	.	2	D
0011	1010	1110	1000	1111	.	0010	1101

$\therefore (3AE8F.2D)_{16}=(111010111010001111.00101101)_2$

Now convert binary to octal
$(111010111010001111.00101101)_2=(\_\_\_\_\_)_8$

111	010	111	010	001	111	.	001	011	010
7	2	7	2	1	7	.	1	3	2

$\therefore (111010111010001111.00101101)_2=(727217.132)_8$

$\therefore (3AE8F.2D)_{16}=(727217.132)_8$

- $(80914.25)_{10}=(?)_8$

8	80914	
8	10114	2 ↑
8	1264	2 ↑
8	158	0 ↑
8	19	6 ↑
8	2	3 ↑
	0	2 ↑

$0.25 \times 8 = 2.0 \Rightarrow 2 \downarrow$

$\therefore (80914.25)_{10}=(236022.2)_8$

- $(20345.125)_{10}=(?)_2$

2	20345	
2	10172	1 ↑
2	5086	0 ↑
2	2543	0 ↑
2	1271	1 ↑
2	635	1 ↑
2	317	1 ↑
2	158	1 ↑
2	79	0 ↑
2	39	1 ↑
2	19	1 ↑
2	9	1 ↑
2	4	1 ↑
2	2	0 ↑
2	1	0 ↑
	0	1 ↑

$0.125 \times 2 = 0.250 \Rightarrow 0 \downarrow$
$0.250 \times 2 = 0.500 \Rightarrow 0 \downarrow$
$0.500 \times 2 = 1.0 \Rightarrow 1 \downarrow$

$\therefore (20345.125)_{10}=(100111101111001.001)_2$

- $(1011001110)_2=(?)_4$

First convert binary to decimal
$(1011001110)_2=(\_\_\_\_\_)_{10}$

1011001110

$= 1\times2^9+0\times2^8+1\times2^7+1\times2^6+0\times2^5+0\times2^4+1\times2^3+1\times2^2+1\times2^1+0\times2^0$

$= 1\times512+0\times256+1\times128+1\times64+0\times32+0\times16+1\times8+1\times4+1\times2+0\times1$

$= 512+0+128+64+0+0+8+4+2+0$

$= 718$

$\therefore (1011001110)_2=(\underline{718})_{10}$

Now convert decimal to base-4
$(718)_{10}=(\_\_\_\_\_)_4$

4	718	
4	179	2 ↑
4	44	3 ↑
4	11	0 ↑
4	2	3 ↑
	0	2 ↑

$\therefore (718)_{10}=(\underline{23032})_4$

$\therefore (1011001110)_2=(\underline{23032})_4$

46. Add $(110111)_2$ with $(100111)_2$ Subtract $(100110)_2$ from $(110011)_2$ using 2's complement method.

Binary value:

110111 + 100111
= **01011110**

+ 2's complement subtraction steps :

Here A = 110011, B = 100110.
Find A - B = ? using 2's complement
First find 2's complement of B = 100110

Note : 2's complement of a number is 1 added to it's 1's complement number.

Step-1: 1's complement of 100110 is obtained by subtracting each digit from 1

```
   1  1  1  1  1  1
 - 1  0  0  1  1  0
-------------------
   0  1  1  0  0  1
```

Step-2: Now add 1 to the 1's complement to obtain the 2's complement :
011001 + 1 = 011010

Step-3: Now Add this 2's complement of B to A

```
 1  1        1
    1  1  0  0  1  1
+   0  1  1  0  1  0
-------------------
 1  0  0  1  1  0  1
```

+ Hints : (Move mouse over the steps for detail calculation highlight)

The left most bit of the result is 1, called carry and it is ignored.
So answer is 001101

47. Represent $(-17)_{10}$ in sign magnitude, 1's complement and 2's complement representation.

Since the number of bits used in the representation is 8, the binary equivalent of 17 is represented as:

$17_{10} = (00010001)_2$

Taking the 1's complement of the above, we get 11101110

Adding 1 to the 1's complement, we get the 2's complement representation of the number, i.e.

11101110 + 1 = 1110 1111

48. With the help of example explain excess-3 code.

Excess-3 code, also known as XS-3 or XS-3 binary-coded decimal (BCD), is a binary-coded decimal system where each decimal digit is represented by a 4-bit binary code. The term "excess-3" indicates that the binary representation of each decimal digit is 3 more than the decimal digit itself.

In Excess-3 code, the decimal digits 0 through 9 are represented as follows:

Decimal Digit	BCD Code	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Example: Convert the decimal number 648 to Excess-3 code.

1. Break down the decimal number into its individual digits:
    - Hundreds place: 6
    - Tens place: 4
    - Ones place: 8
2. Convert each decimal digit to its Excess-3 code:
    - 6 in decimal is represented as 1001 in Excess-3.
    - 4 in decimal is represented as 0111 in Excess-3.
    - 8 in decimal is represented as 1011 in Excess-3.
3. Combine the Excess-3 codes for each digit to represent the entire number:
    - 648 in decimal is represented as 1001 0111 1011 in Excess-3.

So, the Excess-3 code for the decimal number 648 is 1001 0111 1011.

49. Write the BCD code for $(9248)_{10}$

9 2 4 8
1001 0010 0100 1000

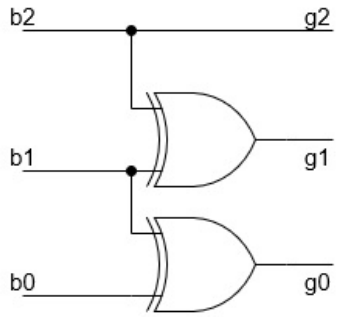
$\therefore (9248)_{10}=(1001001001001000)_{BCD}$

50. How can you easily generate 3 bit gray code.

We can easily generate 3 bit gray code from binary code using X-OR gate.
Truth table for 3 bit binary to 3 bit gray code is:

Decimal Equivalent	Binary			Gray Code		
	B2	B1	B0	G2	G1	G0
0	0	0	0	**0**	**0**	**0**
1	0	0	1	**0**	**0**	**1**
2	0	1	0	**0**	**1**	**1**
3	0	1	1	**0**	**1**	**0**
4	1	0	0	**1**	**1**	**0**
5	1	0	1	**1**	**1**	**1**
6	1	1	0	**1**	**0**	**1**
7	1	1	1	**1**	**0**	**0**

Now, we can construct a logic circuit using the truth table:



51. Define and draw the truth table of a 3-input X-OR gate.

A 3 input X-OR gate is basically the output of a 2 input X-OR gate is again used as an input with another input gate in a X-OR gate. More Formally,

$X = A \oplus B \oplus C$

We know that, if two input is same then, the output of X-OR gate is 0 otherwise 1. Now, using this principle of X-OR gate we can construct the truth table for a 3-input X-OR gate.

Inputs			outputs
W	X	Y	Q = A⊕B⊕C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

52. Show that NAND gate can be used as universal gate. Simplify the following using De-Morgan's theorem $(A+B')'(A'+B)'$

NAND gate can be used as universal gate basically it means Basic logic gates such as AND, OR and NOT gate can be constructed from NAND gate. Even other conjugate gates or universal gate such as Ex-OR gate, NOR can also be constructed from NAND gate.
Here's the construction of basic gates from NAND gate to proof its universality.

AND gate from NAND gate

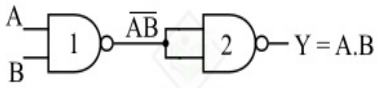


Figure 3 - AND Gate Using NAND Gate

OR gate from NAND gate

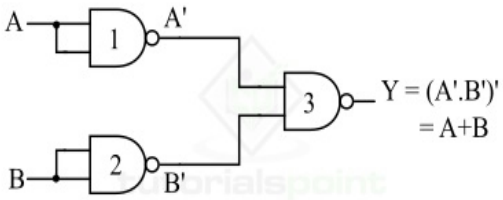


Figure 3 - OR Gate Using NAND Gate

NOT gate from NAND gate

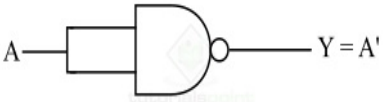


Figure 3 - NOT gate using NAND Gate

Now,
Simplification of the boolean expression using De Morgan's Theorem:
Simplifying the logic gate: $(A+B)'(A'+B')'=(A'.B')((A')'.(B')')=(A'.B')(A.B)=0.$

53. Simplify the following Boolean expressions to a minimum number of literals:

- $ABC+A'BC+A'B+AC$

  Suppose,
  $X=ABC+A'BC+A'B+AC=BC(A+A')+A'B+AC=BC+A'B+AC=AC+A'B$

- $A'B(D'+CD)+B(A+A'CD)$

  Suppose, $X=A'B(D'+CD)+B(A+A'CD)=A'B(C+D')+AB+A'BCD$
  $\rightarrow X=A'BC+A'BD'+AB+A'BCD=A'BC+A'BD'+AB=B(A+A'C)+A'BD$
  $\rightarrow X=B(A+C)+A'BD=AB+BC+A'BD=B(A+A'D)+BC=B(A+D)+BC$
  $\rightarrow X=AB+BC+BD$

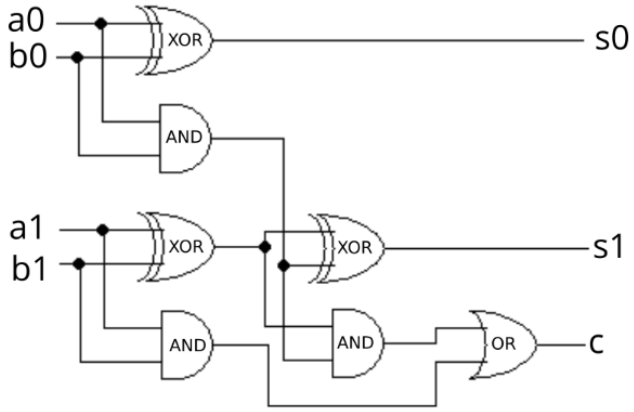
54. A combinational circuit produces the binary sum of two 2-bit numbers, a_0a_1 and b_0b_1. The outputs are C, S_1, and S_0. Provide a truth table of the combinational circuit, deduce the logic function and implement with logic gates. Also, design a circuit for the given problem using two full-adders block.

Here the question points to the 2 bit binary adder. A two-bit adder is a circuit the adds together two, 2-bit numbers. The first number, A, can be
representing using bits a1 and a0. The second number, B, is similarly represented.
The output consists of the sum of A and B, represented as two bits (s1 and s0) and one

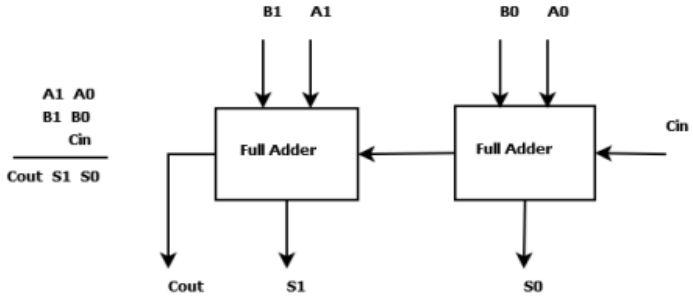
carry bit (c). A truth table for a 2-bit adder is as follows:

Inputs				Outputs		
a_1	a_0	b_1	b_0	c	s_1	s_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Now, we can deduce the logic expression using K-Map for the 3 outputs c, s0 and s1.



Block diagram using 2 bit Full Adder:



55. Construct a BCD-to-excess-3-code converter with a 4-bit adder. Remember that the excess-3 code digit is obtained by adding three to the corresponding BCD digit. What must be done to change the circuit to an excess-3-to-BCD-code converter?

1. Truth table
2. Find K-map for each output
3. Establish expression from each K-map
4. Draw circuit from those expressions

56. Design a combinational circuit with three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is one less than the input.*

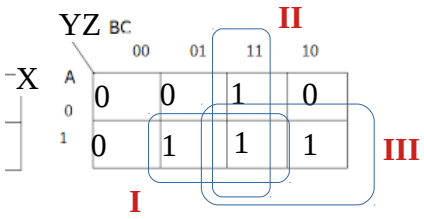
Truth table that defines the required relationship between inputs and outputs.

Input			Output		
X	Y	Z	A	B	C
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

Input 0,1,2,3 < output 1,2,3,4

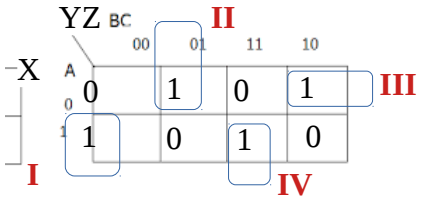
Input 4,5,6,7>output 3,4,5,6

K-Map for A:



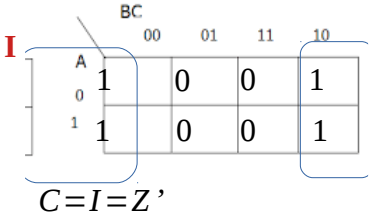
$A = I + II + III = XY + YZ + ZX$

K-Map for B:



$B = I + II + III + IV = XY'Z' + X'Y'Z + XYZ + X'YZ'$

K-Map for C:



$C = I = Z'$

Draw circuit from those expressions

57. Design a code converter that converts a decimal digit from "8 4-2-1" code to BCD ("8 4 2 1") code.

Truth table that defines the required relationship between inputs and outputs.

Input					Output			
Decimal	8	4	-2	-1	8	4	2	1
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	1
2	0	1	1	0	0	0	1	0
3	0	1	0	1	0	0	1	1
4	0	1	0	0	0	1	0	0
5	1	0	1	1	0	1	0	1
6	1	0	1	0	0	1	1	0
7	1	0	0	1	0	1	1	1
8	1	0	0	0	1	0	0	0
9	1	1	1	1	1	0	0	1

1. Find K-map for each output
2. Establish expression from each K-map
3. Draw circuit from those expressions

58. Design a combinational circuit with three inputs, x, y, and z, and four outputs, A, B, C and D. When the binary input is 0, 1, 3, or 5, 7 the binary output is double of the input. When the binary input is 2, 4, or 6, the binary output is half of the input.

Same as 56

59. Simplify the following Boolean function using K-map method

$F(A,B,C.D)=\sum_M(0,1,31,8\,14,14)+\sum_D(2,5,9,11)$. Implement the minimized function.

Same as 24

60. Represent $(-469)_{10}$ to sign-magnitude, 1's complement and 2's complement form.

Assuming -469_{10} for a 16 bit binary number:

$469_{10}=(0000\,0001\,1101\,0001)_2$

Sign magnitude : **1**000 0001 1101 0001 (here 1 is the sign bit for negative number)

1's complement : 1111 1110 0010 1110

2'a complement : 1111 1110 0010 1110 + 1 = 1111 1110 0010 1111

61. Convert the following expressions to sum-of-product (SOP) forms:

- $BC(C'D'+CE)$

  $X=BC(C'D'+CE)=BCE$
- $B+C[BD+(C+D')E]$

  $X=B+C(BD+CE+D'E)=B+BCD+CE+CD'E=B(1+CD)+CE(1+D')=B+CE$

62. Use a Karnaugh map to reduce each expression to a minimum SOP form:

- $A'B'C'D'+A'B'C'D+ABCD+ABCD'$
- $A'B(C'D'+C'D)+AB(C'D'+C'D)+AB'C'D$

Same as 24

63. Use The Quine-McCluskey Method to reduce each expression to a minimum SOP form:

- $X=ABC+A'B'C+ABC'+AB'C+A'BC$
- $X=A'B'C'D'+A'B'C'D+A'BC'D+ABC'D'+AB'CD'+A'BCD'+AB'C'D$

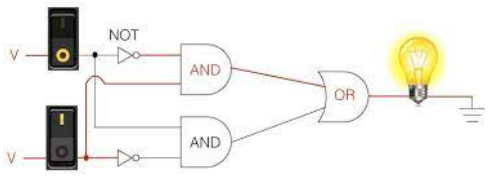
Same as 32

64. Develop the logic and implement the circuit necessary to meet the following requirements:

A battery-powered lamp in a room is to be operated from two switches, one at the back door and one at the front door. The lamp is to be on if the front switch is on and the back switch is off, or if the front switch is off and the back switch is on. The lamp is to be off if both switches are off or if both switches are on. Let a HIGH output represent the on condition and a LOW output represent the off condition.

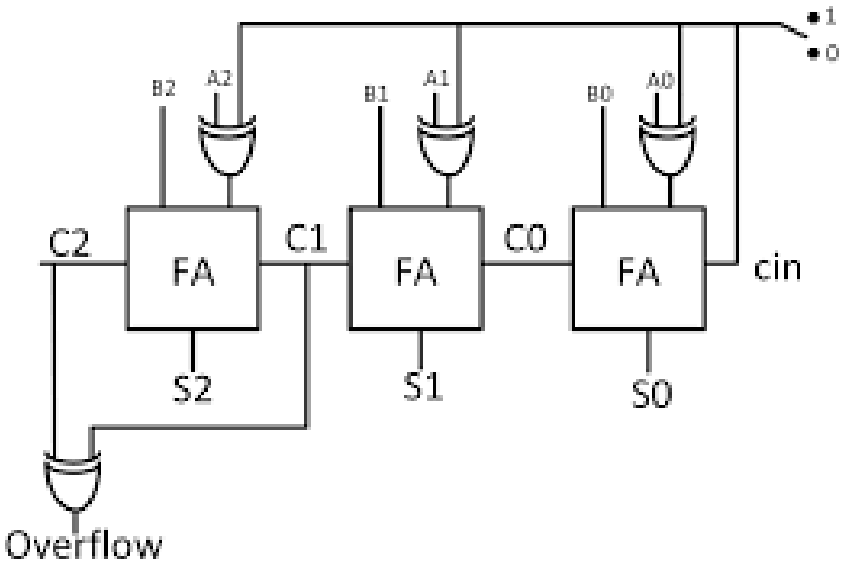
Its an X-OR gate because the output is high when only one of the input is 1 or HIGH otherwise output is 0 or LOW.

$Y=\bar{A}B+A\bar{B}=A\oplus B$



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

65. Design and discuss a 3 bit adder/subtractor circuit.



66. Design a combinational circuit with inputs A, B, C, D and output w, x, y, z. Assume that the inputs A, B, C, D represent a 4-bit signed number. The output is also a signed number, which is the 2's complement of the input.

2's complement input and output:

Input				Output			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

1. Find K-map for each output
2. Establish expression from each K-map
3. Draw circuit from those expressions