Digital Systems
18B11EC213

Binary Codes and Boolean Algebra

Contents

- Weighted and Non-weighted codes
- BCD 8421 Code
- XS-3 Code
- Gray Code
- Boolean Algebra Postulates
- Basic Theorems of Boolean Algebra
- Universal Gates: NAND and NOR

Weighted and Non-weighted Codes

- Numeric codes used to represent decimal digits are called Binary coded decimal (BCD) codes.
- BCD codes simplify the communication process between man and machine.
- A BCD code is one in which each digit of a decimal number is encoded into a group of four binary digits.
- The BCD codes may be weighted or non-weighted.
- The weighted codes are those which obey the position-weighting principle (means each digit has a positional weight). Examples : 8421, 2421
- The non-weighted codes do not obey the position-weighting principle. Examples : Excess-3 code, Gray code

Some commonly used Codes

1. 8421 BCD Codes
2. Excess-3 Codes
3. Gray – Codes

Straight Binary Codes

- One Binary Digit (one bit) can take on values 0, 1.
- This is used to represent numbers using natural (Straight) Binary form as discussed earlier
- $(65)_{10}$ in straight binary represented by 1000001

Natural BCD Code (8421 Code)

- In this Codes, decimal digit 0 through 9 are represented by their natural binary equivalents using four bits and each decimal digit of a decimal number is represented by this four bits code individually.
- It is also known as 8,4,2,1 code.
- 8,4,2,1 are the weights of the four bit of the decimal digit similar to straight binary number system.
- In BCD, there are six illegal combinations 1010, 1011, 1100, 1101, 1110, 1111.

Examples of 8421 BCD

- $(23)_{10}$ is represented by 0010 0011
- $(08)_{10}$ is represented by 0000 1000
- $(921)_{10}$ is represented by 1001 0010 0001

Excess-3 Codes (XS-3 code)

- XS-3 code is a non-weighted BCD Code, in which each decimal digit is coded into a 4-bit binary code
- The code for each decimal digit is obtained by adding decimal 3 to the BCD code of the digit.
- Decimal 2 is coded as 0010 + 0011= 0101
- It is a self complementing code, means 1’s complement of the coded number yields 9’s complement of the number it self .
- XS-3 has six invalid states 0000, 0001, 0010, 1101, 1110, 1111

Examples of XS-3

- $(8)_{10}$ is represented by 1011
- $(15)_{10}$ is represented by 0100 1000
- $(920)_{10}$ is represented by 1100 0101 0011

Gray Code

- It is a very useful code in which a decimal number is represented in binary form in such a way so that each Gray code number differs from the preceding and the succeeding numbers in a single bit.
- It is a non-weighted code.
- It is a reflected code.

Construction of Gray Code

- A 1-bit gray code has two code words 0 and 1 representing decimal numbers 0 and 1.
- An n-bit (n >=2) Gray Code will have first 2^{n-1} Gray Codes with n-1 bits (LSB) written in order with a leading 0 appended.
- The last 2^{n-1} Gray Codes with n-1 bits (LSB) written in reverse order(Mirror image) with a leading 1 appended.

Examples:-

- 1 Bit Gray Code

Decimal Number	Gray Code
0	0
1	1

- 2 Bit Gray Code

0	0	0
1	0	1
2	1	1
3	1	0

More Examples:-

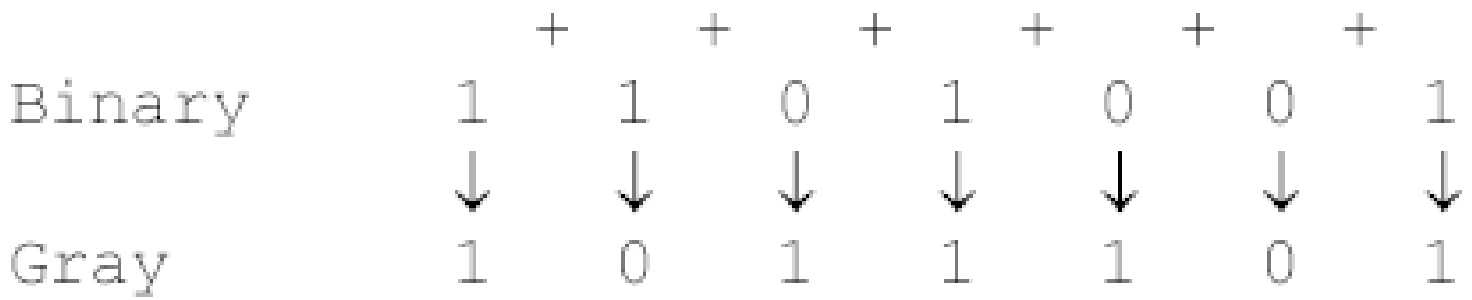
- 3- bit Gray Code

Decimal Number	Gary Code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Conversion From Binary to Gray

- Start with the MSB of the binary number.
- Copy this bit as the MSB of the gray code number.
- Xor the MSB of the binary to the next bit of the binary number.
- The sum (ignoring carry) is the next bit of the gray code number.
- Continue adding each bit of the binary to the next bit to its right to get the gray code for that position as shown in the next slide

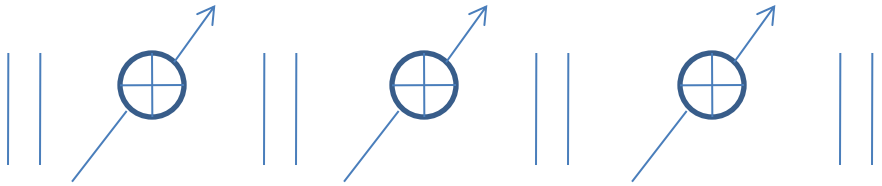
Conversion From Binary to Gray



** Start conversion from MSB. The MSB of Binary is same as MSB of Gray.*

Conversion From Gray to Binary

Gray	1	1	0	1
Binary	1	0	0	1



$(1101)_{\text{Gray}} = (1001)_{\text{Binary}}$

** Start conversion from MSB. The MSB of Binary is same as MSB of Gray.*

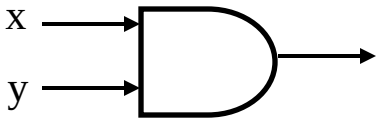
Two-valued Boolean Algebra

- Set of Elements: {0,1}
- Set of Operations: { ., + }

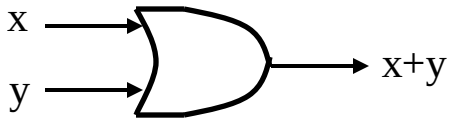
x	y	x . y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

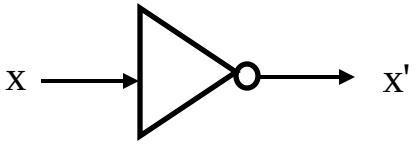
x	¬x
0	1
1	0



AND operation



OR operation



NOT operation

Signals: High = 5V = 1; Low = 0V = 0

Boolean Algebra Postulates

- A **Boolean algebra** consists of a set of elements *B*, with two binary operations {+} and {.} and a uniary operation {'}, such that the following axioms hold:
- The set *B* contains at least two distinct elements x and y.
- **Closure**: For every x, y in *B*,
  - x + y is in *B*
  - x . y is in *B*
- **Commutative laws**: For every x, y in *B*,
  - x + y = y + x
  - x . y = y . x

Boolean Algebra Postulates

- **Associative laws**: For every x, y, z in *B*,
  - (x + y) + z = x + (y + z) = x + y + z
  - (x . y) . z = x .(y . z) = x . y . z
- **Identities** (0 and 1):
  - 0 + x = x + 0 = x for every x in *B*
  - 1 . x = x . 1 = x for every x in *B*
- **Distributive laws**: For every x, y, z in *B*,
  - x . (y + z) = (x . y) + (x . z)
  - x + (y . z) = (x + y) . (x + z)

Boolean Algebra Postulates

- **Complement**: For every x in *B*, there exists an element x' in *B* such that
  - x + x' = 1
  - x . x' = 0

The set *B* = {0, 1} and the logical operations OR, AND and NOT satisfy all the axioms of a Boolean algebra.

A **Boolean function** maps some inputs over {0,1} into {0,1}

A **Boolean expression** is an algebraic statement containing Boolean variables and operators.

Duality

- **Duality Principle** – every valid Boolean expression (equality) remains valid if the operators and identity elements are interchanged, as follows:

  $+ \leftrightarrow .$

  $1 \leftrightarrow 0$

- Example: Given the expression

  $a + (b.c) = (a+b).(a+c)$

  then its dual expression is

  $a . (b+c) = (a.b) + (a.c)$

- If $(x+y+z)' = x'.y.'z'$ is valid, then its dual is also valid:

  $(x.y.z)' = x'+y'+z'$

Basic Theorems of Boolean Algebra

- Apart from the postulates, there are other useful theorems.

1. **Idempotency**.

   (a) $x + x = x$ (b) $x \cdot x = x$

2. **Null elements** for + and . operators.

   (a) $x + 1 = 1$ (b) $x \cdot 0 = 0$

3. **Involution**. $(x')' = x$

4. **Absorption**.

   (a) $x + x.y = x$ (b) $x.(x + y) = x$

5. **Absorption** (variant).

   (a) $x + x'.y = x+y$ (b) $x.(x' + y) = x.y$

Basic Theorems of Boolean Algebra

6. **DeMorgan**.

   (a) $(x + y + z)' = x'.y'.\ z'$

   (b) $(x.y.z)' = x' + y' + z'$

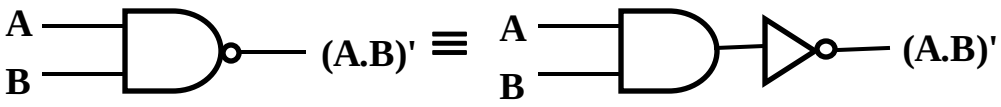
7. **Consensus**.

   (a) $x.y + x'.z + y.z = x.y + x'.z$

   (b) $(x+y).(x'+z).(y+z) = (x+y).(x'+z)$

Logic Gates: The NAND Gate

- The **NAND** Gate



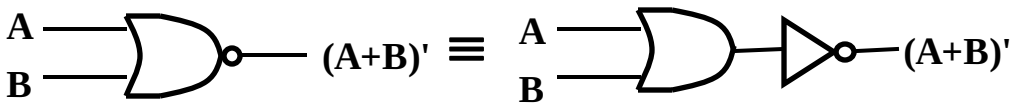
A	B	(A.B)'
0	0	1
0	1	1
1	0	1
1	1	0

NAND

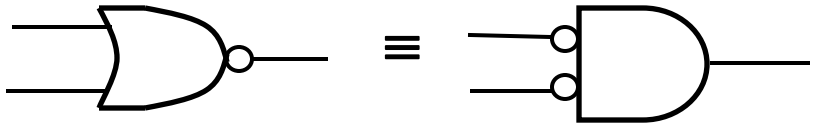
Negative-OR

Logic Gates: The NOR Gate

- The **NOR** Gate



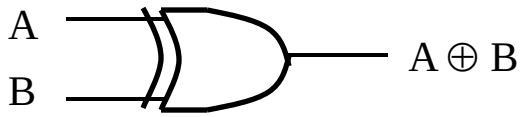
A	B	(A+B)'
0	0	1
0	1	0
1	0	0
1	1	0



NOR ≡ Negative-AND

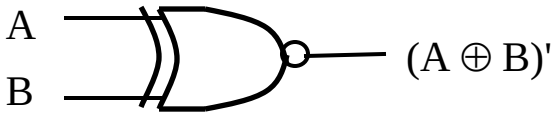
Logic Gates: The XOR & XNOR Gate

- The **XOR** Gate



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

- The **XNOR** Gate



A	B	(A ⊕ B) '
0	0	1
0	1	0
1	0	0
1	1	1

Universal Gates: NAND and NOR

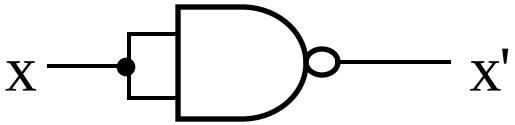
- AND/OR/NOT gates are sufficient for building any Boolean functions.
- We call the set {AND, OR, NOT} a complete set of logic.
- However, other gates are also used because:
  - (i) usefulness
  - (ii) economical on transistors
  - (iii) self-sufficient

  NAND/NOR: economical, self-sufficient

  XOR: useful (e.g. parity bit generation)

NAND Gate

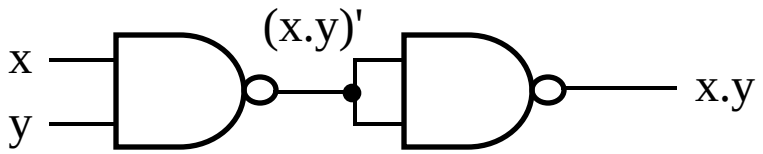
- NAND gate is self-sufficient (can build any logic circuit with it).
- Therefore, {NAND} is also a complete set of logic.
- Can be used to implement AND/OR/NOT.
- Implementing an inverter using NAND gate:



(x.x)' = x' (T1: idempotency)

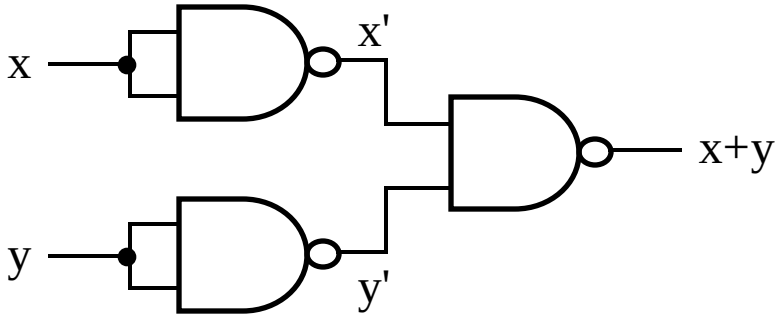
NAND Gate

- Implementing AND using NAND gates:



((xy)'(xy)')' = ((xy)')' idempotency
= (xy) involution

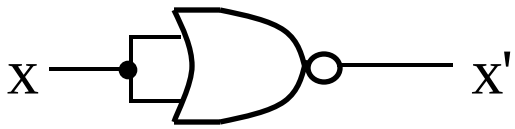
- Implementing OR using NAND gates:



((xx)'(yy)')' = (x'y')' idempotency
= x"+y" DeMorgan
= x+y involution

NOR Gate

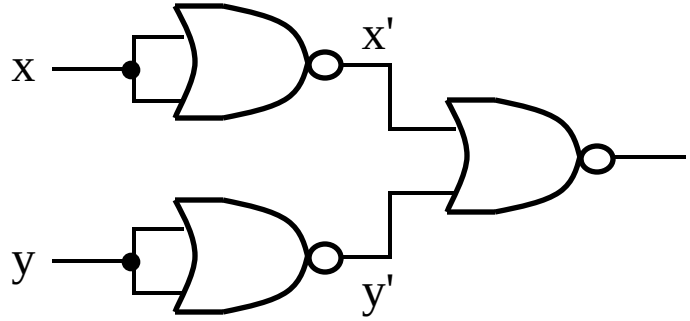
- NOR gate is also self-sufficient.
- Therefore, {NOR} is also a complete set of logic
- Can be used to implement AND/OR/NOT.
- Implementing an inverter using NOR gate:



(x+x)' = x' (T1: idempotency)

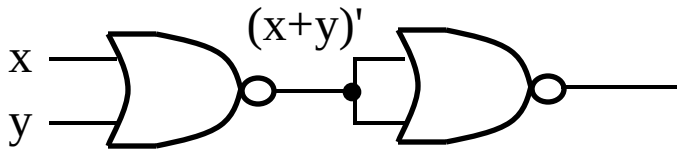
NOR Gate

- Implementing AND using NOR gates:



((x+x)'+(y+y)')'=(x'+y')' idempotency
= x".y" DeMorgan
= x.y involution

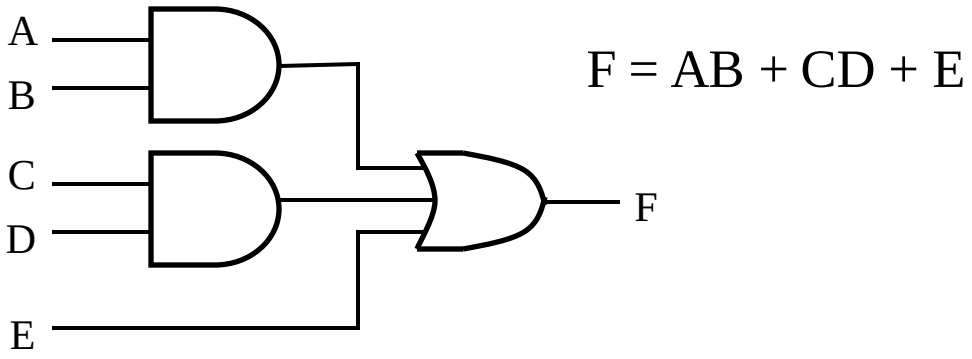
- Implementing OR using NOR gates:



((x+y)'+(x+y)')' = ((x+y)')' idempotency
= (x+y) involution

Implementation of SOP Expressions

- Sum-of-Products expressions can be implemented using:
  - 2-level AND-OR logic circuits
  - 2-level NAND logic circuits
- AND-OR logic circuit

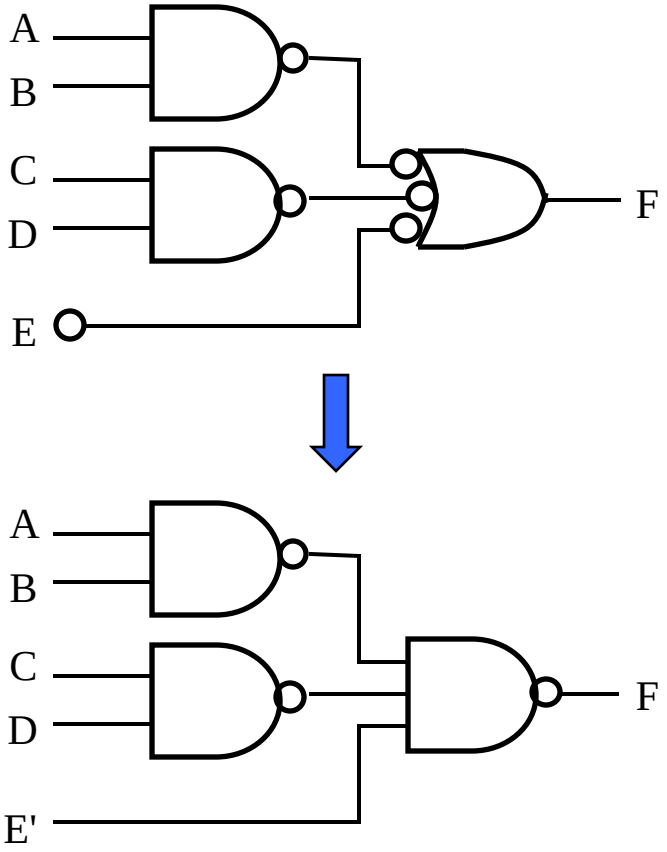


Implementation of SOP Expressions

- NAND-NAND circuit (by circuit transformation)

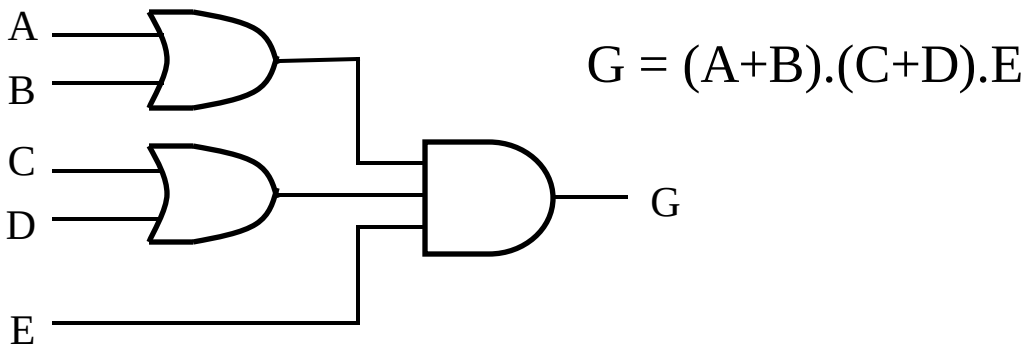
  a) add double bubbles

  b) change OR-with-inverted-inputs to NAND & bubbles at inputs to their complements



Implementation of POS Expressions

- Product-of-Sums expressions can be implemented using:
  - 2-level OR-AND logic circuits
  - 2-level NOR logic circuits
- OR-AND logic circuit

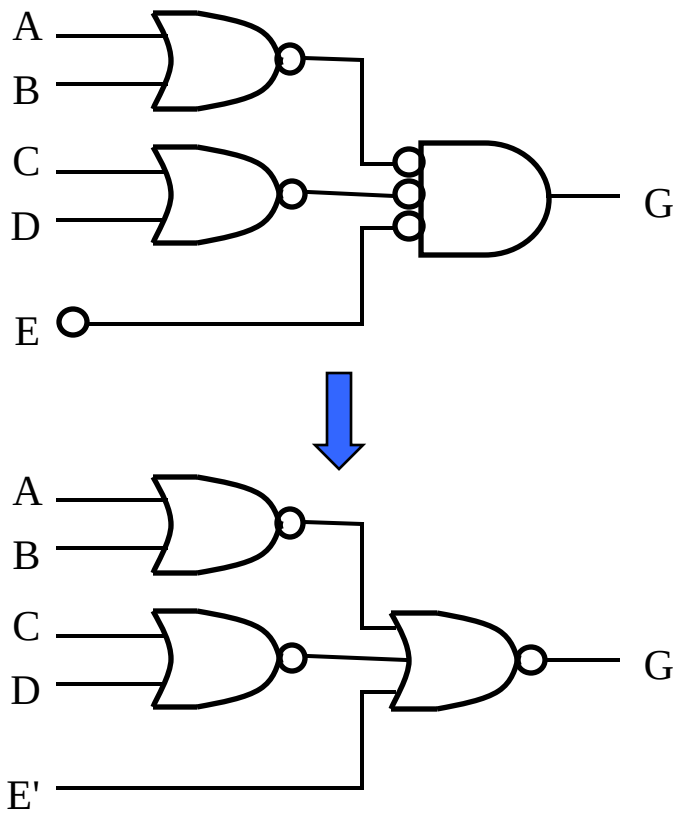


Implementation of POS Expressions

- NOR-NOR circuit (by circuit transformation):

  a) add double bubbles

  b) changed AND-with-inverted-inputs to NOR & bubbles at inputs to their complements



References

1. A. Anand Kumar, “Fundamentals of Digital Circuits”, PHI, Fourth Edition.

2. S. Salivahanan and S. Arivazhagan, “Digital circuits and design”, Vikas Publishing House PVT Limited, Fifth edition.