NUMBER SYSTEM AND BOOLEAN LOGIC

2.1 NUMBER SYSTEM AND CONVERSION

2.1.1 Decimal, Binary, Octal, Hexadecimal Number System & Conversion

Definition: A number system is a way of representing numbers using a fixed set of symbols (digits) and a base (radix), which determines how many unique digits are available and what each digit's position is worth.

Number System	Base (Radix)	Digits Used	Example
Decimal	10	0–9	245
Binary	2	0, 1	1101
Octal	8	0–7	372
Hexadecimal	16	0–9, A–F	1F3

In hexadecimal, the letters A to F represent the decimal values 10 to 15, since hexadecimal needs 16 unique symbols but only has 10 digits (0–9) available naturally.

Decimal to Binary Conversion

Repeatedly divide the decimal number by 2, noting the remainder each time, until the quotient becomes 0. Reading the remainders from bottom to top gives the binary equivalent.

Convert 25 to binary:

25 ÷ 2 = 12 remainder 1

12 ÷ 2 = 6 remainder 0

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1

Reading remainders bottom to top: 11001 So, $(25)_{10} = (11001)_2$

Binary to Decimal Conversion

Multiply each binary digit by 2 raised to the power of its position (counting from 0 on the right), then add all the results together.

Convert 1101 to decimal: $1101 = (1\times2^3) + (1\times2^2) + (0\times2^1) + (1\times2^0) = (1\times8) + (1\times4) + (0\times2) + (1\times1) = 8 + 4 + 0 + 1 = 13$ So, $(1101)_2 = (13)_{10}$

Decimal to Octal Conversion

Repeatedly divide the decimal number by 8, noting the remainder each time, then read the remainders from bottom to top.

Convert 100 to octal:

100 ÷ 8 = 12 remainder 4

12 ÷ 8 = 1 remainder 4

1 ÷ 8 = 0 remainder 1

Reading remainders bottom to top: 144 So, $(100)_{10} = (144)_8$

Decimal to Hexadecimal Conversion

Repeatedly divide the decimal number by 16, noting the remainder each time (converting remainders 10–15 into A–F), then read from bottom to top.

Convert 254 to hexadecimal:

254 ÷ 16 = 15 remainder 14 (E)

15 ÷ 16 = 0 remainder 15 (F)

Reading remainders bottom to top: FE So, $(254)_{10} = (FE)_{16}$

Binary to Octal and Hexadecimal (Shortcut Method)

Since $8 = 2^3$ and $16 = 2^4$, binary digits can be grouped directly without converting through decimal:

- **Binary to Octal**: group binary digits into sets of 3, starting from the right, then convert each group to its octal digit.
- **Binary to Hexadecimal**: group binary digits into sets of 4, starting from the right, then convert each group to its hexadecimal digit.

Convert 101110 to octal (group into 3s): 101 110 5 6 So, $(101110)_2 = (56)_8$ Convert 101110 to hexadecimal (group into 4s): 0010 1110 (pad with leading zeros to complete the group) 2 E So, $(101110)_2 = (2E)_{16}$

2.1.2 Calculation in Binary Addition, Subtraction

Binary Addition Rules

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Just like decimal addition, when a column adds up to 2 or more in binary, a carry is passed to the next column to the left.

Add 1011 and 1101: 1 1 1 (carry) 1 0 1 1 + 1 1 0 1 ——— 1 1 0 0 0 So, 1011 + 1101 = 11000

Binary Subtraction Rules

A	B	Difference	Borrow
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

Subtract 0101 from 1110: 1 1 1 0 – 0 1 0 1 ———— 1 0 0 1 So, 1110 – 0101 = 1001

2.1.3 One's and Two's Complement Methods of Binary Subtraction

Computers often perform subtraction using complements instead of direct borrowing, since this allows the same circuitry used for addition to also handle subtraction.

One's Complement

The one's complement of a binary number is found by simply flipping every bit. Every 0 becomes 1, and every 1 becomes 0.

Find the one's complement of 1010: Original: 1010 One's Complement: 0101

Subtraction Using One's Complement

Subtract 0101 from 1110 using one's complement:

Step 1: Find one's complement of 0101 → 1010

Step 2: Add it to 1110 1 1 1 0 + 1 0 1 0 ———— 1 0 1 0 0

Step 3: Since there is a carry out (leftmost 1), add it back to the result: 1 0 0 + 1 ——— 1 0 1 So, 1110 – 0101 = 1001 (using one's complement, with end-around carry)

Two's Complement

The two's complement of a binary number is found by taking the one's complement first, then adding 1 to the result. Two's complement is more commonly used in computers because it avoids some of the extra steps (like end-around carry) required by one's complement.

Find the two's complement of 1010: Original: 1010 One's Complement: 0101 Add 1: + 1 —— Two's Complement: 0110

Subtraction Using Two's Complement

Subtract 0101 from 1110 using two's complement:

Step 1: Find two's complement of 0101 One's complement: 1010 Add 1: 1011 (this is the two's complement)

Step 2: Add it to 1110 1 1 1 0 + 1 0 1 1 ———— 1 1 0 0 1

Step 3: Discard the final carry (leftmost 1) if present:

Result: 1001 So, 1110 – 0101 = 1001 (using two's complement)

Notice that with two's complement, there's no need to add back an end-around carry. Any carry generated beyond the number's bit length is simply discarded, which makes two's complement simpler to implement in hardware.

2.2 LOGIC FUNCTION AND BOOLEAN ALGEBRA

2.2.1 Introduction to Boolean Algebra

Definition: Boolean algebra is a branch of algebra in which variables can only hold one of two values, true (1) or false (0), and is used to analyze and simplify logical statements and digital circuits.

Boolean algebra was developed by mathematician George Boole, and it forms the mathematical foundation behind how digital computers make decisions and process logic, since every digital circuit is ultimately built from combinations of true/false (1/0) signals.

2.2.2 Introduction to Boolean Values, Truth Table, Boolean Expression and Boolean Function

Boolean Values

A Boolean value can only be one of two states: **True (1)** or **False (0)**. These represent concepts like "on/off," "yes/no," or "high/low" in a digital circuit.

Truth Table

A truth table is a table that lists every possible combination of input values for a Boolean expression, along with the resulting output for each combination. Truth tables are used to fully describe how a logic gate or Boolean expression behaves.

Boolean Expression

A Boolean expression is a combination of Boolean variables and operators (AND, OR, NOT) that evaluates to either true or false, similar to how an arithmetic expression evaluates to a number.

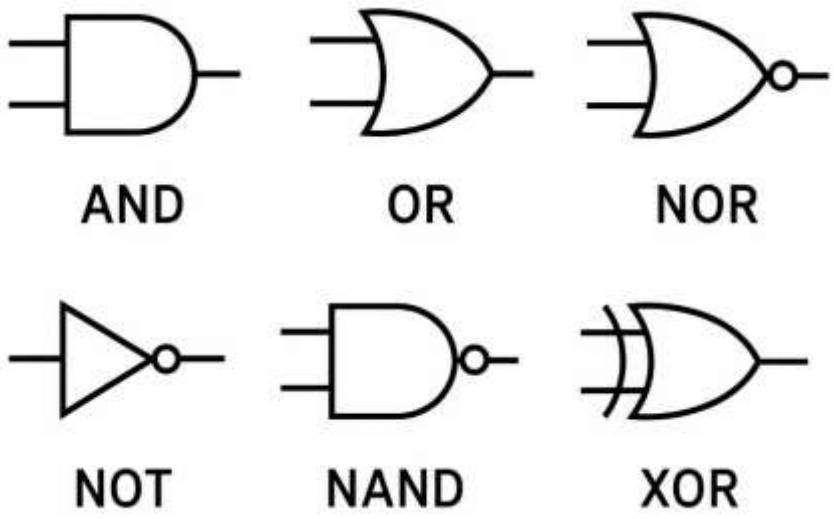
Example Boolean expression: Y = A.B + C (This reads as: Y is true if both A AND B are true, OR if C is true)

Boolean Function

A Boolean function is a function that takes one or more Boolean inputs and produces a single Boolean output, typically expressed using a Boolean expression, and is often represented visually as a truth table or a logic circuit diagram.

2.2.3 Logic Gates: AND, OR, NOT, NAND, NOR, XOR, XNOR

A logic gate is a basic building block of digital circuits that takes one or more binary inputs and produces a single binary output, based on a specific logical rule.



AND Gate

Definition: The output is true (1) only when all inputs are true (1).

Logic Function: Y = A . B

A	**B**	**Y = A.B**
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

Definition: The output is true (1) if at least one input is true (1).

Logic Function: Y = A + B

A	**B**	**Y = A+B**
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

Definition: A single-input gate that simply reverses (inverts) the input value. It is also called an inverter.

Logic Function: Y = A' (or $\bar{A}$)

A	Y = A'
0	1
1	0

NAND Gate (NOT + AND)

Definition: Produces the opposite output of an AND gate. The output is false (0) only when all inputs are true (1).

Logic Function: Y = (A.B)'

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

NOR Gate (NOT + OR)

Definition: Produces the opposite output of an OR gate. The output is true (1) only when all inputs are false (0).

Logic Function: Y = (A+B)'

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

XOR Gate (Exclusive OR)

Definition: Output is true (1) only when the inputs are different from each other.

Logic Function: Y = A ⊕ B

A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate (Exclusive NOR)

Definition: Produces the opposite output of an XOR gate. The output is true (1) only when both inputs are the same (both 0 or both 1).

Logic Function: Y = (A ⊕ B)'

A	B	Y = (A⊕B)'
0	0	1
0	1	0
1	0	0
1	1	1

2.2.4 Laws of Boolean Algebra

Just like ordinary algebra has rules (like the commutative or distributive property), Boolean algebra follows its own set of laws, which are used to simplify complex Boolean expressions and digital circuits.

Complement Laws

A + A' = 1 A . A' = 0

A variable combined with its own complement using OR always gives 1 (true), and combined using AND always gives 0 (false).

Identity Laws

A + 0 = A A . 1 = A A + 1 = 1 A . 0 = 0

Commutative Laws

A + B = B + A A . B = B . A

The order of variables does not affect the result of an AND or OR operation.

Associative Laws

(A + B) + C = A + (B + C) (A . B) . C = A . (B . C)

When combining three or more variables with the same operation, the way they are grouped does not affect the final result.

Distributive Laws

A . (B + C) = (A.B) + (A.C) A + (B.C) = (A+B) . (A+C)

Similar to how multiplication distributes over addition in ordinary algebra, AND distributes over OR in Boolean algebra (and, less intuitively, OR also distributes over AND).

2.2.5 Statement and Verification of Laws of Boolean Algebra Using Truth Table

Boolean laws can be verified by constructing a truth table for both sides of the equation and checking that the output columns match exactly for every possible input combination.

Example: Verifying the Commutative Law (A + B = B + A)

A	B	A + B	B + A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Since the "A + B" column and the "B + A" column produce identical results for every combination of A and B, the commutative law is verified.

Example: Verifying the Distributive Law (A.(B+C) = (A.B)+(A.C))

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

The “A.(B+C)” column and the “(A.B)+(A.C)” column match exactly for every combination of A, B, and C, so the distributive law is verified.

De Morgan’s First Theorem: (A + B)’ = A’ . B’

This theorem states that the complement of a sum (OR) equals the product (AND) of the individual complements.

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

The “(A+B)’” column and the “A’.B’” column produce identical results for every combination of A and B, so De Morgan’s first theorem is proved.

De Morgan’s Second Theorem: (A . B)’ = A’ + B’

This theorem states that the complement of a product (AND) equals the sum (OR) of the individual complements.

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

The “(A.B)’” column and the “A’+B’” column produce identical results for every combination of A and B, so De Morgan’s second theorem is proved.

Both theorems are widely used to simplify Boolean expressions and digital circuits, since they let you convert an OR-based expression into an AND-based one (and the other way round) without changing what the circuit actually does. NAND and NOR gates are often described as “universal gates” precisely because De Morgan’s theorems show they can be combined to build any of the other basic gates.