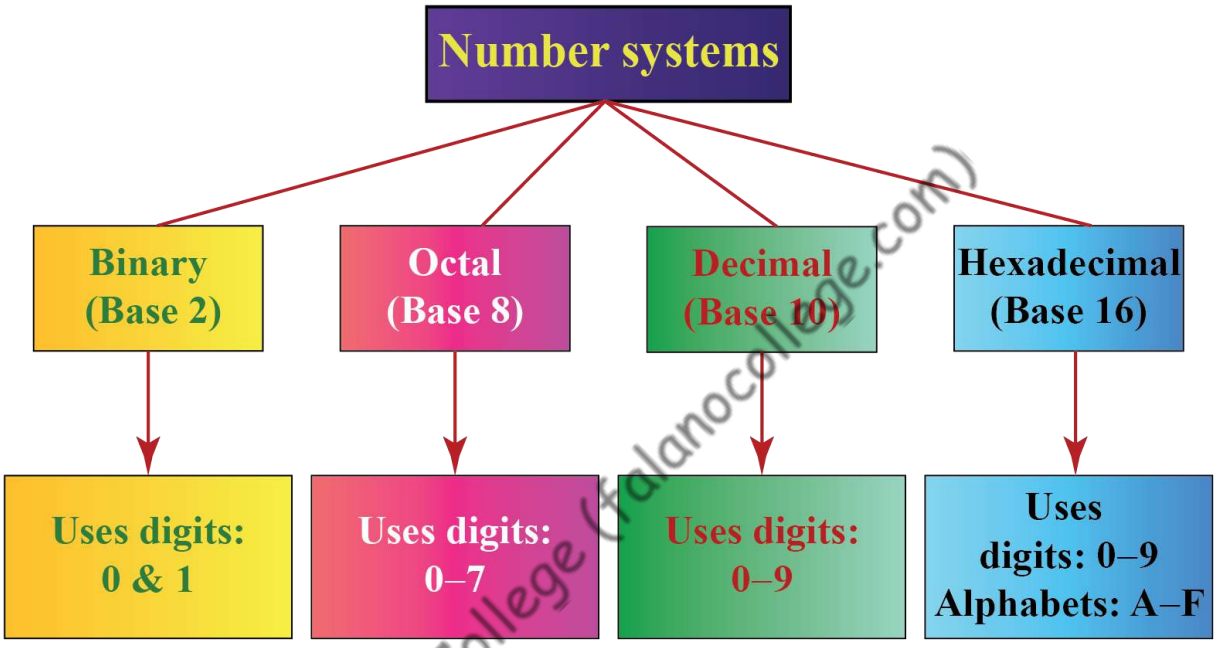
Number System



Source: https://d138zd1ktt9iqe.cloudfront.net/media/seo_landing_files/usha-number-system-06-1-1594730376.png

Introduction

A number system is a way of representing numbers using a set of symbols or digits. There are several types of number systems used in computing, including:

1. **Binary Number System:** The binary number system uses only two digits, 0 and 1, to represent numbers. It is the basis of all digital computing systems and is used to represent data and instructions in computer hardware. The binary number system is a base-2 number system that uses only two digits, 0 and 1, to represent numbers. It is the foundation of digital electronics and computer systems, where all data is represented using binary digits.

   In the binary number system, each digit represents a power of 2, starting from the rightmost digit which represents 2^0, then moving left to the next digit which represents 2^1, then 2^2, and so on. For example, the binary number 1011 can be interpreted as:

   1 x 2^3 (8) + 0 x 2^2 (0) + 1 x 2^1 (2) + 1 x 2^0 (1) = 11

   Binary numbers can be used to represent any decimal number. Conversely, decimal numbers can be converted to binary by repeatedly dividing by 2 and keeping track of the remainders, starting from the rightmost digit. For example, to convert the decimal number 13 to binary:

   Divide 13 by 2, with a remainder of 1. Write down the remainder, which is the rightmost digit of the binary number.

   Divide the quotient (6) by 2, with a remainder of 0. Write down the remainder, which is the next digit of the binary number.

   Divide the quotient (3) by 2, with a remainder of 1. Write down the remainder.

   Divide the quotient (1) by 2, with a remainder of 1. Write down the remainder.

   The binary representation of 13 is 1101.

   Binary numbers can also be used to represent characters and other data in computing systems. For example, the ASCII encoding system represents each character using a unique 8-bit binary code.

2. **Decimal Number System:** The decimal number system uses ten digits, 0 through 9, to represent numbers. It is the most commonly used number system in everyday life and is used to represent quantities such as money, time, and distance. The decimal number system is the most common base-10 number system used in everyday life. It uses ten digits, 0 through 9, to represent numbers. Each digit in a decimal number represents a power of 10, starting from the rightmost digit which represents 10^0, then moving left to the next digit which represents 10^1, then 10^2, and so on. For example, the decimal number 123 can be interpreted as:

   1 x 10^2 (100) + 2 x 10^1 (20) + 3 x 10^0 (3) = 123

   Decimal numbers can be used to represent any quantity, from the smallest to the largest. They are widely used in everyday life, from measuring quantities such as weight, length, and time, to representing values such as money and percentages.

   In computing systems, decimal numbers are typically represented using binary-coded decimal (BCD) or floating-point notation. BCD is a method of encoding decimal numbers using binary digits, where each decimal digit is represented by a four-bit binary code. Floating-point notation is a method of representing numbers with a fractional component, using a combination of a sign, exponent, and mantissa.

3. **Octal Number System:** The octal number system uses eight digits, 0 through 7, to represent numbers. It is used in some computer applications, particularly in the UNIX operating system. The octal number system is a base-8 number system that uses eight digits, 0 through 7, to represent numbers. It is often used in computing systems, particularly in older systems such as mainframe computers, where it is still used in some contexts.

   In the octal number system, each digit represents a power of 8, starting from the rightmost digit which represents 8^0, then moving left to the next digit which represents 8^1, then 8^2, and so on. For example, the octal number 245 can be interpreted as:

   2 x 8^2 (128) + 4 x 8^1 (32) + 5 x 8^0 (5) = 165

   Octal numbers can be used to represent any decimal number. Conversely, decimal numbers can be converted to octal by repeatedly dividing by 8 and

keeping track of the remainders, starting from the rightmost digit. For example, to convert the decimal number 57 to octal:

Divide 57 by 8, with a remainder of 1. Write down the remainder, which is the rightmost digit of the octal number.

Divide the quotient (7) by 8, with a remainder of 7. Write down the remainder, which is the next digit of the octal number.

The octal representation of 57 is 71.

Octal numbers are less commonly used than decimal or binary numbers in modern computing systems, but they can still be found in some contexts, such as in file permissions on Unix and Linux systems.

4. **Hexadecimal Number System:** The hexadecimal number system uses sixteen digits, 0 through 9 and A through F, to represent numbers. It is used in some computer applications, particularly in representing memory addresses and color codes. The hexadecimal number system is a base-16 number system that uses 16 digits, 0 through 9 and A through F, to represent numbers. It is widely used in computing systems as a convenient way to represent binary data in a more compact and human-readable form.

   In the hexadecimal number system, each digit represents a power of 16, starting from the rightmost digit which represents 16^0, then moving left to the next digit which represents 16^1, then 16^2, and so on. The letters A through F represent the decimal values 10 through 15, respectively. For example, the hexadecimal number 1A4 can be interpreted as:

   1 x 16^2 (256) + 10 x 16^1 (160) + 4 x 16^0 (4) = 420

   Hexadecimal numbers can be used to represent any binary data, such as memory addresses, register values, and ASCII characters. Each hexadecimal digit represents four binary digits (bits), so it is easy to convert between hexadecimal and binary. For example, the binary number 11011010 can be represented as the hexadecimal number DA.

   Hexadecimal numbers are also used in color representations, where each color component (red, green, and blue) is represented by a two-digit

hexadecimal number from 00 to FF. This allows for a total of 16.8 million possible colors to be represented.

In computing, binary is the most important number system because it is used to represent data and instructions in computer hardware. Decimal is the most commonly used number system in everyday life, and is used in applications such as finance, science, and engineering. Octal and hexadecimal are used in some computer applications for their ease of conversion between binary and decimal.

Conversion

Binary to Decimal

To convert a binary number to a decimal number, you need to multiply each binary digit by the corresponding power of 2 and add up the results. The rightmost digit of the binary number represents 2^0, the next digit to the left represents 2^1, and so on, with each subsequent digit representing the next higher power of 2.

For example, to convert the binary number 101010 to decimal:

1. Write down the binary number: 101010
2. Starting from the rightmost digit, multiply each digit by the corresponding power of 2: 0 x 2^0 = 0 1 x 2^1 = 2 0 x 2^2 = 0 1 x 2^3 = 8 0 x 2^4 = 0 1 x 2^5 = 32
3. Add up the results: 0 + 2 + 0 + 8 + 0 + 32 = 42

Therefore, the binary number 101010 is equivalent to the decimal number 42.

Decimal to Binary

To convert a decimal number to binary, you need to repeatedly divide the decimal number by 2 and write down the remainders in reverse order. The result is a binary number equivalent to the decimal number.

1. Here are the steps to convert a decimal number to binary:
2. Write down the decimal number that you want to convert to binary.
3. Divide the decimal number by 2.
4. Write down the remainder (either 0 or 1).
5. Divide the quotient from step 2 by 2.

6. Write down the remainder.
7. Continue this process of dividing by 2 and writing down the remainders until the quotient becomes 0.
8. Write down the remainders in reverse order to get the binary equivalent of the decimal number.

For example, let's convert the decimal number 27 to binary:

1. Start with the decimal number 27.
2. Divide 27 by 2 to get a quotient of 13 with a remainder of 1.
3. Write down the remainder of 1.
4. Divide 13 by 2 to get a quotient of 6 with a remainder of 1.
5. Write down the remainder of 1.
6. Divide 6 by 2 to get a quotient of 3 with a remainder of 0.
7. Write down the remainder of 0.
8. Divide 3 by 2 to get a quotient of 1 with a remainder of 1.
9. Write down the remainder of 1.
10. Divide 1 by 2 to get a quotient of 0 with a remainder of 1.
11. Write down the remainder of 1.
12. Write down the remainders in reverse order: 11011.

Therefore, the decimal number 27 is equivalent to the binary number 11011.

Binary to Hexadecimal

To convert a binary number to a hexadecimal number, you need to group the binary digits into groups of four, starting from the rightmost digit. If the number of digits in the binary number is not a multiple of 4, add 0s to the left of the number to make it a multiple of 4. Then, convert each group of 4 binary digits to its corresponding hexadecimal digit.

Here are the steps to convert a binary number to hexadecimal:

1. Write down the binary number that you want to convert to hexadecimal.
2. If the number of digits in the binary number is not a multiple of 4, add 0s to the left of the number to make it a multiple of 4.
3. Group the binary digits into groups of 4, starting from the rightmost digit.
4. Convert each group of 4 binary digits to its corresponding hexadecimal digit, using the following table:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Binary	Hexadecimal
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

5. Write down the hexadecimal digits in order from left to right.

For example, let's convert the binary number 11010110 to hexadecimal:

1. Start with the binary number 11010110.
2. Since the number of digits is not a multiple of 4, add a 0 to the left of the number to make it a multiple of 4: 11010110 becomes 1101 0110.
3. Group the binary digits into g roups of 4: 1101 0110.
4. Convert each group of 4 binary digits to its corresponding hexadecimal digit: D6.
5. Write down the hexadecimal digits in order from left to right: D6.

Therefore, the binary number 11010110 is equivalent to the hexadecimal number D6.

Binary Addition and Subtraction

Binary addition and subtraction are the fundamental operations used in binary arithmetic. Here are the step-by-step instructions for adding and subtracting binary numbers, along with examples:

Addition

To add two binary numbers, follow these steps:

1. Write the two binary numbers you want to add below each other, aligning the digits.
2. Start adding the digits from the right-hand side, just like in decimal addition. Add the two right-most digits, and write down the sum below them. If the sum is 0 or 1, write down that digit. If the sum is 2, write down 0 and carry over 1 to the next column. If the sum is 3, write down 1 and carry over 1 to the next column.
3. Move to the next column to the left, and repeat Step 2. Continue until you have added all the digits.
4. If there is a carry digit at the end of the addition, write it down as well.

Here's an example of binary addition:

```
 101101
+111010
--------
 1010111
```


Subtraction

To subtract two binary numbers, follow these steps:

1. Write the minuend (the number being subtracted from) and the subtrahend (the number being subtracted) below each other, aligning the digits.
2. Start subtracting the digits from the right-hand side, just like in decimal subtraction. If the digit in the subtrahend is greater than the corresponding digit in the minuend, you need to borrow 1 from the next column to the left in the minuend. Subtract the borrowed 1 from the next column, and add 2 to the

digit in the minuend column you are subtracting from. Then subtract the subtrahend digit from the minuend digit.

3. Write down the result below the digits you subtracted. If you had to borrow, cross out the borrowed 1 and write the new digit on top of the column you borrowed from.
4. Move to the next column to the left, and repeat Step 2. Continue until you have subtracted all the digits.
5. If there is a borrow digit at the end of the subtraction, write it down as well.

Here's an example of binary subtraction:

```
  10101010
– 11001100
----------
  -0100010
```


1’s and 2’s Complement method

One's complement and two's complement are two methods used to represent negative binary numbers and perform subtraction operations in binary arithmetic. Here's a brief explanation of each method:

1’s Complement:

In one's complement, the negative of a binary number is obtained by flipping all the bits in the number. For example, the one's complement of 10101010 is 01010101.

To perform subtraction using one's complement, you can use the following steps:

1. Find the one's complement of the subtrahend.
2. Add the minuend and the one's complement of the subtrahend.
3. If there is a carry out of the most significant bit, add it to the result.

Here's an example of binary subtraction using one's complement:

Minuend: 110101

- Subtrahend: 101011

Step 1: One's complement of subtrahend = 010100

Step 2: Minuend + one's complement = 1000101

Step 3: There is no carry out of the most significant bit, so the final result is 1000101 (in binary) or 37 (in decimal).

2's Complement

In two's complement, the negative of a binary number is obtained by inverting all the bits in the number and then adding 1. For example, the two's complement of 10101010 is 01010110 (invert all bits, then add 1).

To perform subtraction using two's complement, you can use the following steps:

1. Find the two's complement of the subtrahend.
2. Add the minuend and the two's complement of the subtrahend.
3. If there is a carry out of the most significant bit, discard it (it represents overflow).

Here's an example of binary subtraction using two's complement:

Minuend: 110101

- Subtrahend: 101011

Step 1: Two's complement of subtrahend = 010101

Step 2: Minuend + two's complement = 1000100

Step 3: There is no carry out of the most significant bit, so the final result is 1000100 (in binary) or 36 (in decimal).

Note that in both methods, the subtrahend is negated to perform subtraction. The one's complement method negates the subtrahend by flipping all its bits, while the two's complement method negates the subtrahend by inverting all its bits and adding 1.

Logic Function and Boolean Algebra

Logic function and Boolean algebra are two concepts that are used in digital electronics to analyze and design digital circuits.

Logic Function

A logic function is a mathematical function that takes binary inputs and produces binary outputs based on a set of logic rules. Logic functions are used to describe the behavior of digital circuits and can be expressed using truth tables or Boolean expressions.

Examples of logic functions include:

- AND: The output is 1 only if all the inputs are 1.
- OR: The output is 1 if at least one input is 1.
- NOT: The output is the complement of the input (i.e., 1 if the input is 0, and 0 if the input is 1).
- NAND: The output is the complement of the AND function.
- NOR: The output is the complement of the OR function.
- XOR: The output is 1 if the number of 1's in the inputs is odd, and 0 if the number of 1's is even.

Boolean Algebra

Boolean algebra is a mathematical system that deals with binary variables and logical operations. It is used to simplify logic functions and express them using Boolean expressions. In Boolean algebra, the two binary values are represented by 0 and 1, and the logical operations are represented by symbols such as + (OR), · (AND), and ' (NOT).

Boolean algebra has several laws and theorems that are used to simplify and manipulate Boolean expressions. Some of the laws include:

- Commutative law: $A + B = B + A$, and $A \cdot B = B \cdot A$.
- Associative law: $(A + B) + C = A + (B + C)$, and $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.
- Distributive law: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, and $A + (B \cdot C) = (A + B) \cdot (A + C)$.
- De Morgan's law: $(A + B)' = A' \cdot B'$, and $(A \cdot B)' = A' + B'$.

Using these laws, it is possible to simplify complex logic functions into simpler expressions, which can be easily implemented in digital circuits. Boolean algebra is an essential tool for digital circuit design, as it allows designers to optimize their circuits for performance, cost, and power consumption.

AND Gate

An AND gate is a basic digital logic gate that has two or more inputs and one output. The output of an AND gate is 1 (or high) only if all the inputs are 1 (or high). Otherwise, the output is 0 (or low).

The symbol for an AND gate is a triangle with a curved edge, and the inputs are placed on the left side of the triangle, while the output is on the right side.

Truth Table:

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

Here, A and B are two inputs and C is the output.

C = A AND B

In this truth table, A and B are the inputs to the AND gate, and C is the output. As you can see, the output C is 1 only when both inputs A and B are 1. Otherwise, the output is 0.

AND gates are used in many digital circuits, such as combinational logic circuits, arithmetic circuits, and memory circuits. They are also used in computer processors and microcontrollers for performing logical operations and calculations.

OR Gate

An OR gate is another basic digital logic gate that has two or more inputs and one output. The output of an OR gate is 1 (or high) if any of the inputs are 1 (or high). Otherwise, the output is 0 (or low).

The symbol for an OR gate is a triangle with a flat edge, and the inputs are placed on the left side of the triangle, while the output is on the right side.

Truth Table:

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

Here, A and B are two inputs and C is the

C = A OR B

In this truth table, A and B are the inputs to the OR gate, and C is the output. As you can see, the output C is 1 if either input A or input B is 1. If both inputs are 0, then the output is also 0.

OR gates are used in many digital circuits, such as combinational logic circuits, arithmetic circuits, and memory circuits. They are also used in computer processors and microcontrollers for performing logical operations and calculations.

NOT Gate

A NOT gate, also known as an inverter, is a basic digital logic gate that has one input and one output. The output of a NOT gate is the opposite (or complement) of its input. If the input is 1 (or high), then the output is 0 (or low), and vice versa.

The symbol for a NOT gate is a triangle with a small circle at the output, and the input is placed on the left side of the triangle.

Truth Table:

A	C	
0	1	Here, A is the input and C is the output.
1	0	C = A’

In this truth table, A is the input to the NOT gate, and C is the output. As you can see, the output C is the complement of the input A.

NOT gates are used in many digital circuits, such as combinational logic circuits, arithmetic circuits, and memory circuits. They are also used in computer processors and microcontrollers for performing logical operations and calculations. For example, a NOT gate can be used to invert the output of a sensor or to convert a positive logic signal to a negative logic signal.

NAND Gate

A NAND gate is a combination of an AND gate followed by a NOT gate (inverter). The output of a NAND gate is 1 (or high) unless all of the inputs are 1 (or high). In other words, the output of a NAND gate is the complement of an AND gate. The symbol for a NAND gate is a triangle with a curved edge and a small circle at the output, similar to an AND gate with an inverted output.

Truth Table:

A	B	C	
0	0	1	
0	1	1	Here, A and B are two inputs and C is the output.
1	0	1	C = A NAND B
1	1	0	

NOR Gate

A NOR gate is a combination of an OR gate followed by a NOT gate (inverter). The output of a NOR gate is 1 (or high) only if none of the inputs are 1 (or high). In other words, the output of a NOR gate is the complement of an OR gate. The symbol for a NOR gate is a triangle with a flat edge and a small circle at the output, similar to an OR gate with an inverted output.

Truth Table:

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

Here, A and B are two inputs and C is the output

C = A NOR B

NAND and NOR gates are useful for simplifying logic expressions and reducing the number of gates in a circuit. They are also used in memory circuits, arithmetic circuits, and other digital circuits.

X-OR Gate

An XOR gate, also known as an exclusive OR gate, has two inputs and one output. The output of an XOR gate is 1 (or high) if the inputs are different, and 0 (or low) if the inputs are the same. In other words, the output of an XOR gate is 1 if and only if one of the inputs is 1. The symbol for an XOR gate is a circle with a plus sign inside.

Truth Table:

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

Here, A and B are two inputs and C is the output.

C = A XOR B

X-NOR Gate

An XNOR gate, also known as an exclusive NOR gate, is the complement of an XOR gate. The output of an XNOR gate is 1 (or high) if the inputs are the same, and 0 (or low) if the inputs are different. In other words, the output of an XNOR gate is 1 if and only if both inputs are the same. The symbol for an XNOR gate is a circle with a plus sign and a small circle at the output.

Truth Table:

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

Here, A and B are two inputs and C is the output.

C = A XNOR B

Laws of Boolean Algebra

Boolean algebra is a mathematical system that deals with binary variables and logic operations. The laws of Boolean algebra are a set of rules that govern the manipulation of these binary variables and logic operations. The laws of Boolean algebra are based on the principles of identity, commutativity, associativity, distributivity, and complementarity.

Here are some of the most commonly used laws of Boolean algebra:

Identity laws:

$A + 0 = A$

$A \cdot 1 = A$

These laws state that the sum of any variable and zero is equal to the variable itself, and the product of any variable and one is equal to the variable itself.

Commutative laws:

$A + B = B + A$

$A \cdot B = B \cdot A$

A These laws state that the order of variables does not matter when performing addition or multiplication.

Associative laws:

$(A + B) + C = A + (B + C)$

$(A \cdot B) \cdot C = A \cdot (B \cdot C)$

These laws state that the grouping of variables does not matter when performing addition or multiplication.

Distributive laws:

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

$A + (B \cdot C) = (A + B) \cdot (A + C)$

These laws state that multiplication distributes over addition, and addition distributes over multiplication.

Complementarity laws:

$A + A' = 1$

$A \cdot A' = 0$

These laws state that the sum of any variable and its complement is equal to one, and the product of any variable and its complement is equal to zero.

These laws can be used to simplify and manipulate logic expressions, reduce the number of gates in a circuit, and check the correctness of a logic circuit design. They are also used in digital circuit design, computer architecture, and other areas of computer science and engineering.

De Morgan’s Theorem

De Morgan's laws are a set of two fundamental rules of Boolean algebra that relate the negation of logical expressions to their conjunction and disjunction. The laws are named after the British mathematician and logician Augustus De Morgan.

First Law: (A AND B)’ = A’ OR B’

The first law of De Morgan states that the negation of a conjunction (AND) is equivalent to the disjunction (OR) of the negations of the individual terms. In other words, the negation of "A AND B" is the same as "not A OR not B."

Second Law: (A OR B)’ = A’ AND B’

The second law of De Morgan states that the negation of a disjunction (OR) is equivalent to the conjunction (AND) of the negations of the individual terms. In other words, the negation of "A OR B" is the same as "not A AND not B."

For example, let's consider the expression "not (A and B)." Applying the first law of De Morgan, we can rewrite this expression as "not A or not B." Similarly, the expression "not (A or B)" can be rewritten as "not A and not B" using the second law of De Morgan.

De Morgan's laws are important in digital logic and computer science, as they can be used to simplify logic expressions and reduce the number of gates in a circuit. They also have applications in other areas of mathematics and logic, such as set theory and predicate logic.