SITE 1101: Principles of Information Systems

Week 03

Binary Representation, Arithmetic & Logic Operations

Authors: Rahida Asadli, Nilufar Ismayilova, Rahman Karimov, Ismayil Shahaliyev

Created / Updated: Oct 23 2025 / Dec 19 2025

Digital vs Analog

Discrete means values change in separate, well-defined steps with no possible values in between. A standard light switch is a simple example: it is either on or off – there's nothing between those two states. Likewise, a digital clock that shows 14:35 jumps straight to 14:36 with no "in-between" time displayed.

Continuous means values can vary smoothly and without breaks over a range. Between any two values, there are infinitely many possible intermediate values. A classic example is a mercury thermometer: if the temperature is 21 °C and then rises to 22 °C, it passes through 21.1 °C, 21.11 °C, 21.111 °C, and so on – endlessly many points. The temperature does not "jump" from one reading to another; it changes continuously.

This distinction is why *digital* systems are far more reliable than *analog* ones. In a continuous analog system, even a tiny fluctuation – a degree in temperature or an electrical noise – can change the value. In a discrete digital system, as long as the signal is close enough to one of the two states (say, above 3 V is "1" and below 1 V is "0"), the computer will interpret it correctly. That tolerance to noise is why modern computers abandoned continuous analog signals and rely almost entirely on discrete binary states.

Example. In an analog system, numbers 3 and 5 would be represented by physical quantities – for example, voltage levels of 3 V and 5 V. To add them, the machine might combine the voltages to produce 8 V. But if noise (e.g. temperature) affects hardware and changes voltage levels by just 0.2 V, the output might be 7.8 V or 8.2 V – which no longer corresponds exactly to 8. Repeated operations would accumulate these small errors, making the result unreliable.

The solution was to abandon continuous analog representation in favor of discrete digital states. Digital circuits only need to distinguish between a small set of clearly separated values, making them far more reliable, scalable, and resistant to noise. Once that decision was made, the simplest and most robust choice was to use two discrete states. This directly matched what electronic components could reliably detect: current flowing or not, voltage high or low, transistor open or closed. Those two states were naturally represented by 0 and 1, forming the binary number system.

Bit, Byte, Data Units

All digital information – including instructions themselves – is expressed as *bits* (binary digits, formally introduced by Claude Shannon in the paper A Mathematical Theory of Communication). A *bit* is the smallest unit of data and can hold one of two values: 0 or 1. Every number, letter, image, or instruction is ultimately **encoded** as sequences of bits.

With two bits, each bit can be either 0 or 1, so there are 2×2=4 possible combinations: 00, 01, 10, and 11. With three bits, there are 2×2×2=8 combinations: 000, 001, 010, 011, 100, 101, 110, and 111. In general, if you have **N** bits, each bit doubles the number of possible combinations because it can take two independent values. Therefore, the total number of combinations is 2^N. This simple rule explains why digital systems scale so effectively: every additional bit doubles the range of values that can be represented.

Eight bits grouped together form a *byte*, the standard unit for representing a single character or small piece of data. For example, the binary sequence 01000001 represents the letter "A" in the ASCII encoding system. Larger quantities of data are measured in multiples of bytes: kilobytes (KB), megabytes (MB), gigabytes (GB), and beyond – but all are built on the same binary foundation.

Data Unit	Size
Bit (b)	1
Byte (B)	8 bits
Kilobyte (KB)	$2^{10} = 1,024 \approx 10^3$ bytes
Megabyte (MB)	10^6 bytes
Gigabyte (GB)	10^9 bytes
Terabyte (TB)	10^{12} bytes
Petabyte (PB)	10^{15} bytes
Exabyte (EB)	10^{18} bytes
Zettabyte (ZB)	10^{21} bytes
Yottabyte (YB)	10^{24} bytes

Exercise: How many different colors can be represented in an RGB image if each of the three color channels (Red, Green, Blue) is stored using 8 bits?

Number Systems

Every piece of data inside a computer – numbers, letters, images, even videos – is represented using *number systems*. A number system defines how we represent and interpret numerical values using a specific set of symbols (digits) and a *base* that indicates how many symbols are available.

System	Base	Digits Used	Example	Usage
Decimal (Base-10)	10	0–9	245_{10}	Used by humans for everyday counting and arithmetic.

Binary (Base-2)	2	0, 1	101101_2	Used internally by all digital computers.
Hexadecimal (Base-16)	16	0–9, A–F	$2AF_{16}$	Used in memory addressing, machine instructions, representing colors, etc. Requires less symbols for describing large numbers.

Decimal System (Base-10)
The decimal system uses ten digits (0–9). Each position represents a power of 10.

Example: $245_{10} = (2 \times 10^2) + (4 \times 10^1) + (5 \times 10^0) = 200 + 40 + 5 = 245_{10}$

Binary System (Base-2):
The binary system uses only two digits: 0 and 1. Each position represents a power of 2.

Example: $1011_2 = (1\times2^3) + (0\times2^2) + (1\times2^1) + (1\times2^0) = 8 + 0 + 2 + 1 = 11_{10}$

Hexadecimal System (Base-16):
The hexadecimal system uses digits 0–9 and letters A–F, representing values 10–15.
Each hex digit equals 4 binary bits.

Example: $2AF_{16} = (2\times16^2) + (A\times16^1) + (F\times16^0) = (2\times256) + (10\times16) + (15\times1) = 687_{10}$

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Examples.

Decimal → Binary: To convert $\mathbf{13}_{10}$ to binary, write it in the powers of two (...16,8,4,2,1): 8+4+1=13. Hence, there is one 8, one 4, zero 2, and one 1: $\mathbf{1101}_2$

Binary → Decimal: 1011_2 means that there is one 1, one 2, zero 4, and one 8. 1+0+2+8 = 11_{10}

Binary ↔ Hexadecimal: Group binary digits into 4-bit sets and then convert. 10101110_2 = 1010 $1110_2 = AE_{16}$ ($1010_2 = 10_{10} = A_{16}$ and $1110_2 = 14_{10} = E_{16}$)

Exercise. Write down different numbers and convert back and forth in different number systems.

Binary Arithmetic and Two's Complement

Just like in the decimal system, numbers in the binary system can be added together - but since binary uses only 0 and 1, the addition rules are simpler. Binary addition is one of the most fundamental operations performed by the Arithmetic Logic Unit (ALU) in a computer's CPU.

Addition	Result	Carry
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

In digital systems, subtraction is often performed using addition. To make this possible, computers represent negative numbers in *two's complement* form, allowing a single adder circuit to handle both addition and subtraction. A - B → A + (-B)

With 3 bits, we can represent 8 different patterns (2^3). How we interpret those patterns depends on whether we want only positive numbers or both positive and negative. If we choose **unsigned representation**, all 3 bits are used for positive numbers. The patterns go from 000 to 111, which correspond to the numbers 0 through 7.

If we decide to include negative numbers, we can, for example, use one bit as a sign. The first bit indicates whether the number is positive (0) or negative (1), and the remaining two bits show the magnitude: 000 represents +0, 001 is +1, 010 is +2, 011 is +3, and the negative side goes from 100 (–0) to 111 (–3). This wastes one code for "–0" and makes arithmetic incorrect (try subtracting 2 from 1 in with sign bit representation).

Two's complement avoids two zeros (-0, and +0) and arithmetic issues. With three bits, the positive numbers still go from 000 (0) to 011 (3), but the negatives start from 100 (–4) to 111 (–1). There is only one zero, and addition or subtraction works naturally without special rules. *To find the negative of any number, we will invert the bits (0 becomes 1, and 1 becomes 0) and add 1 to the result. If there is overflow, we discard that bit.* That is two's complement.

Example 1: Positive Result. A=7, B=5. Compute A-B using 4-bit binary.[1]

Step	Operation	Result
1	Write in binary	($A = 0111_2$), ($B = 0101_2$)
2	Invert B → Add 1	$0101 = 1010 \rightarrow 1010 + 0001 \rightarrow \mathbf{1011}_2$
3	Add A + (Two's complement of B)	$0111 + 1011 = \mathbf{10010}_2$

[1] Instead of memorizing the provided actions, see the shared slides on the course page and the study materials for clear understanding.

4	Discard carry (leftmost bit)	Result = $\mathbf{0010}_2$
5	Convert to decimal	$\mathbf{2}_{10}$

Example 2: Negative Result. A=7, B=5. Compute B-A using 4-bit binary.

Step	Operation	Result
1	Write in binary	($A = 0101_2$), ($B = 0111_2$)
2	Two's complement of B	$0111 = 1000 \rightarrow 1000 + 0001 \rightarrow \mathbf{1001}_2$
3	Add A + (Two's complement of B)	$0101 + 1001 = \mathbf{1110}_2$
4	No carry → negative result. Find two's complement to revert it.	$1110_2 \rightarrow 0001 + 0011 = \mathbf{0010}_2$
5	Convert to decimal and attach negative sign.	$\mathbf{-2}_{10}$

Exercise. Subtract any two numbers in binary.

Transistors and Moore's Law

Transistor is a tiny electronic switch that controls the flow of electricity. Because computers use binary, transistors are the physical parts that store and process bits. Transistors can turn current on or off, just like a light switch, but they do this automatically using voltage. When a transistor is "on," electricity passes (1), when it's "off," electricity is blocked (0). An *NPN* transistor has layers forming three parts called the **collector (C)**, **base (B)**, and **emitter (E)**. Giving small charge to base opens up the "gate" for electrons to pass.

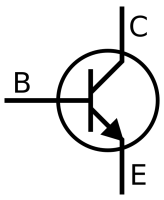

Figure 2: Symbol for an NPN transistor by Omegatron. This symbol was created with Inkscape by user Omegatron., CC BY-SA 3.0. Wikimedia Commons.

Modern computer chips contain billions of transistors packed into an area smaller than a fingernail. In 1965, Intel co-founder Gordon Moore observed that the number of transistors on a chip was roughly doubling every two years, which became known as Moore's Law (which is not a law). It meant that computers kept getting faster, smaller, and cheaper at a steady pace for decades. Although transistor growth has slowed recently due to physical limits (*what kind of limits?*).

Boolean Logic

All digital logic is based on a simple idea called Boolean algebra, named after George Boole. In this system, everything can be only **true** or **false**, or in computer terms, **1** or **0**.

There are three main logical operations. **NOT** simply flips the value: 1 becomes 0, and 0 becomes 1. **AND** gives 1 only if both inputs are 1. **OR** gives 1 if at least one input is 1. For example, if one switch is on (1) and another is off (0), AND gives 0 because both are not on.

These simple operations are what every computer uses. Inside a processor, they are built as small electronic parts called **logic gates**, and every complex calculation you see – from

adding numbers to running software – comes from millions of these tiny logical steps happening very fast.

NOT

A	Q
0	1
1	0

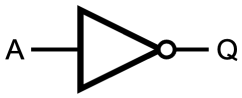

Figure 3: NOT logic gate symbol. Source: Inductiveload, "NOT symbol," Wikimedia Commons (Public Domain).

AND

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

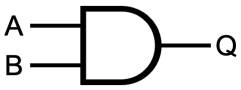

Figure 4: AND logic gate symbol. Source: Inductiveload, "AND symbol," Wikimedia Commons (Public Domain).

OR

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

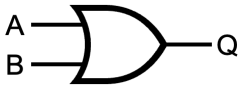

Figure 5: OR logic gate symbol. Source: Inductiveload, "OR symbol," Wikimedia Commons (Public Domain).

XOR

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

Figure 6: XOR logic gate symbol. Source: Inductiveload, "XOR symbol," Wikimedia Commons (Public Domain) .

Exercise. How would you implement XOR by using the three main gates? You can use online digital logic simulator.

Exercise. Use logic gates to add any two single-bit binary numbers. Which gates would you use? *Hint:* See addition table above.

Additional Material

- Early Computing: Crash Course Computer Science #1
- Electronic Computing: Crash Course Computer Science #2
- Boolean Logic & Logic Gates: Crash Course Computer Science #3
- Binary: Plusses & Minuses (Why We Use Two's Complement) - Computerphile
- Why It Was Almost Impossible to Make the Blue LED