

CS 110A: Introduction to Computer Systems

Lecture 02: Everything is a Number



ShanghaiTech University

School of Information Science and Technology (SIST)



September 16, 2025

Undergraduate Core Course · Fall 2026

"In computer architecture, every piece of data, including text, images, or code, ultimately boils down to binary numbers. This lecture uncovers the numerical foundation that underpins all computing operations."

Lecture Outline

01

Administrative Matters— Review of upcoming homework deadlines, lab session logistics, and discussion group assignments for the week.

02

The Binary System— Core fundamentals of base-2 numbering, techniques for decimal conversion, and binary arithmetic operations.

03

The Core Idea: Everything is a Number— Exploring the fundamental concept that all digital data (text, images, code) is ultimately encoded as numerical values.

04

Signed and Unsigned Integers— A detailed look at integer encoding schemes: Sign-Magnitude, One's Complement, and the ubiquitous Two's Complement.

05

Floating-Point Representation (IEEE 754)— Moving beyond integers: understanding the IEEE 754 standard for representing real numbers in modern computing.

An example

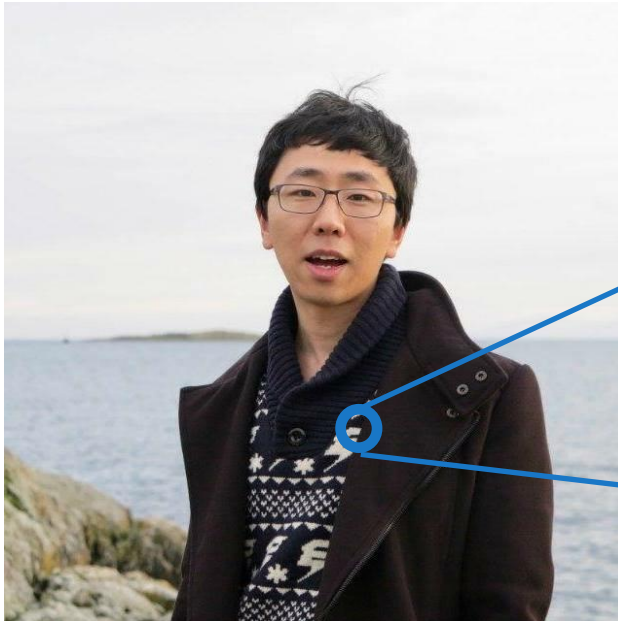
The Fundamental Principle

Inside a computer, **all information** is stored as binary numbers (0s and 1s). This universal encoding is the foundation of digital computing.

Number usually in a fixed size

8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, ...

A 640*640 pixel color image



Around 60 KB on disk

A 20*20 slice of the color image

HEX: #c6c3ba
RGB(198, 195, 186)



HEX: #292023
RGB(41, 32, 35)

HEX: #d5d2c9
RGB(213, 210, 201)

2. The Binary System: Fundamentals



What is a Bit?

The smallest unit of digital information.

- Represents a state of **0 (Off)** or **1 (On)**.
- Derived from the **bistable** nature of transistors, which physically switch between two voltage levels.
- All digital data, such as text, images, sound, and code, is ultimately encoded as sequences of bits.

Why Binary? Core Advantages



01. Robustness (Noise Immunity)

Two distinct states are far less susceptible to corruption from electrical noise or signal degradation compared to analog or multi-state systems.



02. Simplicity (Hardware Ease)

Directly maps to basic electronic components (transistors as switches) that naturally toggle between 'conducting' (1) and 'non-conducting' (0) states.



03. Boolean Logic (Mathematical Basis)

Perfectly aligns with Boolean Algebra, providing a rigorous mathematical framework to design and analyze complex computational logic circuits.

2. The Binary System: Representation & Conversion

01 / Positional Notation

Binary numbers follow the same positional rules as the decimal system, but with a base of 2. Each digit's value is determined by its position (starting from 0 on the right) multiplied by a power of 2.

Example: Convert 1011_2 to Decimal

$$\begin{aligned} 1011_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 8 + 0 + 2 + 1 = 11_{10} \end{aligned}$$

02 / Extended Representations

Long binary strings are cumbersome to read and write. We use more compact bases that align perfectly with binary bit groupings:

Octal (Base 8)

Each octal digit maps to exactly 3 binary bits (000 to 111). Historically used in early computing systems for simplifying bitwise operations.

Hexadecimal (Base 16)

Each hex digit represents 4 binary bits (0000 to 1111), using 0-9 and A-F. The de facto standard in modern programming for memory addresses and color codes.

3. The Core Idea: Everything is a Number

The Fundamental Principle

Inside a computer, **all information** is stored as binary numbers (0s and 1s). This universal encoding is the foundation of digital computing.



User Data: Text documents, images, audio, and video files



Program Logic: CPU instructions that execute software operations



System State: Hardware configurations, memory values, and flags

Fixed Bit Width: The "Words" of Computing

8 Bits

1 Byte

Basic unit for ASCII characters and small numerical values.

16 Bits

1 Halfword

Used for short integers, legacy systems, and some audio data.

32 Bits

1 Word

Standard for 32-bit operating systems and general-purpose calculations.

64 Bits

1 Doubleword

For large memory addressing, high-precision math, and modern OS.



Critical Takeaway: A sequence of bits has no inherent meaning. Its interpretation—whether as a number, letter, or instruction—is determined by the context and the system's design.

3. Everything is a Number: Examples



01 / Digital Audio

Sampling Continuous Waves

Sound is a continuous pressure wave. To digitize it, we take **samples** (measurements) at regular intervals (e.g., 44,100 times per second).

Numbers as Sound

Each sample is a numerical value representing the wave's amplitude. A music file is simply a long list of these numbers.



02 / Digital Images

Pixels as Numerical Grids

An image is a 2D matrix of **pixels**. Each pixel's color is defined by three integers (0-255) for its Red, Green, and Blue (RGB) components.

Color = Numbers

For example, pure red is (255, 0, 0) in decimal, or #FF0000 in hex. The entire image is a database of these color values.

CORE INSIGHT: All digital content—whether it's music, photos, or text—is fundamentally stored and processed as binary numbers (0s and 1s) by computers.

3. Everything is a Number: Beyond Data

01. Information is also Numbers



Identity & Credentials

Student IDs, bank accounts, and encrypted passwords are all stored as numerical data at their core.



Digital Assets

In-game currencies, NFTs, and virtual real estate exist purely as numerical entries in databases.



Personal Profiles

User preferences, social media content, and biometrics are encoded into binary numerical streams.

02. Program Instructions are Numbers



Stored Program Concept

A foundational principle: both program code and data reside in the same memory, as numbers.



ISA & Machine Code

Assemblers translate human-readable code into numerical machine instructions defined by the hardware's ISA.



CPU Execution

The CPU fetches these numerical instructions and executes operations based on predefined logic.



Key Insight: Meaning is Contextual

Bits are just electrical patterns with no inherent meaning. Their value—whether as data, code, or an instruction—derives entirely from how the system **interprets** the pattern, guided by software conventions and hardware design.

Everything is a Number

Instructions: e.g., move directions: forward, backward, left, right;
use $(00)_2$, $(01)_2$, $(10)_2$ and $(11)_2$

Assembly Language
Program (e.g., RISC-V)

Assembler

Machine Language
Program (RISC-V)

Stored program

```
lw    t0, 0(s2)
lw    t1, 4(s2)
sw    t1, 0(s2)
sw    t0, 4(s2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

- It is how you interpret the numbers decides the meaning

Anything can be represented as a *number*, i.e., data or instructions
ISA, defined by human, decides the meaning

3. Everything is a Number: Limitations

01. Overflow: The Boundary of Size

Caused by fixed bit-width storage, overflow happens when calculations exceed the maximum or minimum representable value. For integers, this leads to unexpected wrap-around (e.g., large positives become negatives). For floating-point numbers, results default to $\pm\text{Infinity}$, corrupting subsequent computations.

02. Underflow: The Vanishing Point

Occurs when values shrink below the smallest representable magnitude for the data type. The result is typically rounded down to zero, erasing critical precision in iterative calculations (e.g., repeated division or decay models). This "zeroing out" distorts proportional relationships in numerical models.



Mitigation: Scaling and Robust Data Types

Use larger bit-width types (e.g., 64-bit integers/floats) to expand the representable range and reduce risks. While this minimizes occurrences, it does not eliminate them entirely—rigorous input validation, range checks, and saturation arithmetic (capping values at limits) remain essential for building robust numerical systems.

4. Signed and Unsigned Integers

01 Unsigned Integers

Represents only non-negative values (zero and positive numbers). All bits are dedicated to encoding the magnitude, with no sign bit allocated.

32-bit Range: 0 to 4,294,967,295 ($2^{32} - 1$)

Covers a single range from zero to the maximum positive value.

Key Uses: Memory addressing, array indexing, counting iterations, file sizes, and network protocol fields where negative values are illogical.

02 Signed Integers

Designed to represent both positive and negative numbers. The sign is encoded using a specific bit pattern convention.

1. Sign-Magnitude: 1 bit for sign (+/-), rest for value. Flaw: Two representations of zero (0 and -0).

2. One's Complement: Invert all bits for negatives. Still has ± 0 issue; mostly of historical interest.

3. Two's Complement (Standard): Invert bits + 1. No ± 0 , and subtraction is implemented as addition, making it the modern standard.

Unsigned Integers

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

0000 0000 0000 0000 0000 0000 0000 0010_{two} = 2_{ten}

...

...

0111 1111 1111 1111 1111 1111 1111 1101_{two} = 2,147,483,645_{ten}

0111 1111 1111 1111 1111 1111 1111 1110_{two} = 2,147,483,646_{ten}

0111 1111 1111 1111 1111 1111 1111 1111_{two} = 2,147,483,647_{ten}

1000 0000 0000 0000 0000 0000 0000 0000_{two} = 2,147,483,648_{ten}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = 2,147,483,649_{ten}

1000 0000 0000 0000 0000 0000 0000 0010_{two} = 2,147,483,650_{ten}

...

...

1111 1111 1111 1111 1111 1111 1111 1101_{two} = 4,294,967,293_{ten}

1111 1111 1111 1111 1111 1111 1111 1110_{two} = 4,294,967,294_{ten}

1111 1111 1111 1111 1111 1111 1111 1111_{two} = 4,294,967,295_{ten}

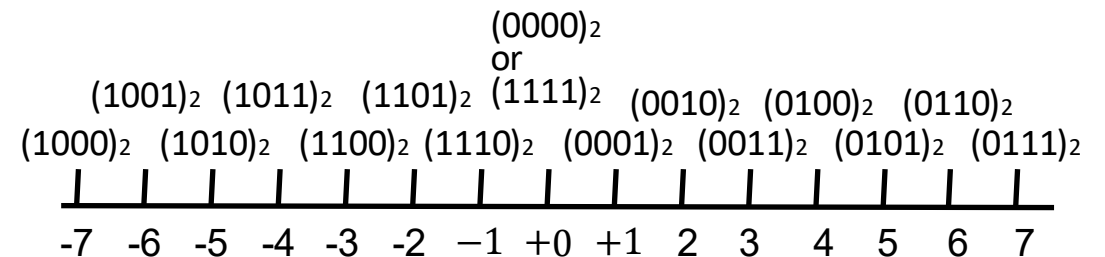
$$(a_n a_{n-1} \dots a_1 a_0)_2 = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Signed Integers

- A straight-forward method: add a sign bit (sign-magnitude)
- Most-significant bit (MSB, leftmost) is the sign bit, 0 means positive, 1 means negative; the other bits remain unchanged

0000 0000 0000 0000 0000 0000 0000 0011_{two} = 3_{ten}
 1111 1111 1111 1111 1111 1111 1111 1100_{two} = -3_{ten}
 Sign bit

- Range:
 - Positive: $0 \sim 2^{(n-1)}-1$
 - Negative: $-0 \sim -(2^{(n-1)}-1)$
 - Arithmetically unfriendly



Signed Integers

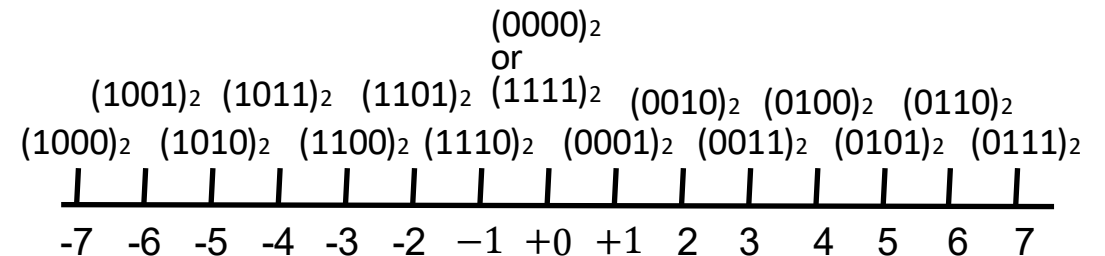
One's- & Two's-Complement Representation

- One's-complement representation
- Positive numbers, stay unchanged; Negative numbers, toggle all bits

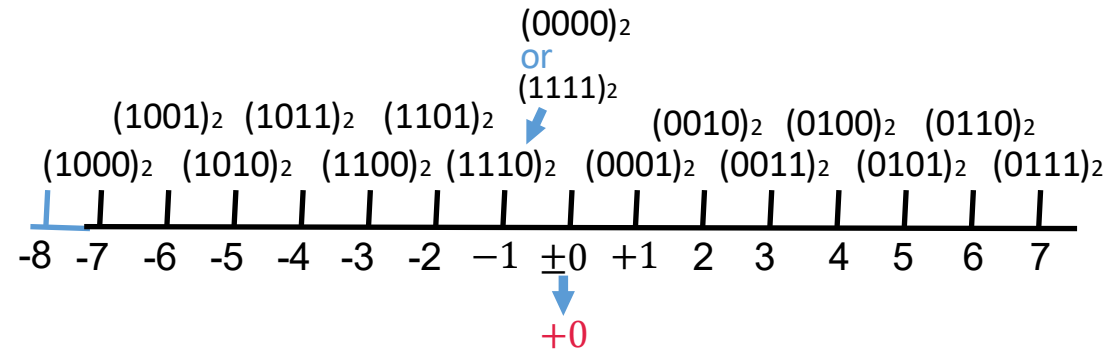
 0000 0000 0000 0000 0000 0000 0000 0011_{two} = 3_{ten}
1111 1111 1111 1111 1111 1111 1111 1100_{two} = -3_{ten}
Sign bit

- Range:

- Positive: $0 \sim 2^{(n-1)}-1$
- Negative: $-0 \sim -(2^{(n-1)}-1)$
- Arithmetically unfriendly



Two's-Complement Representation (Signed Integer)

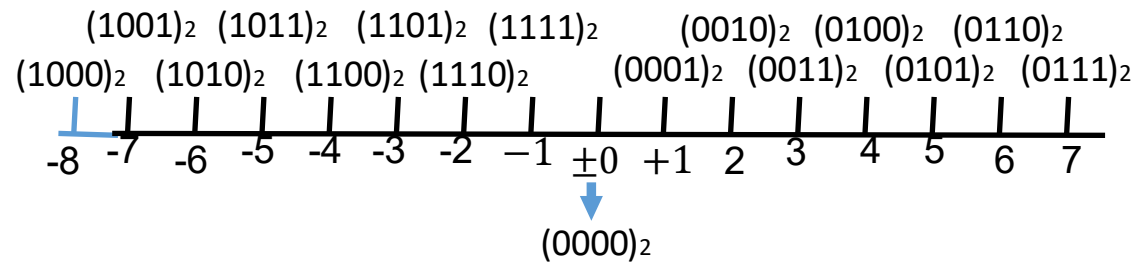


- Two's-complement representation:
 - Positive numbers, stay unchanged; Negative numbers, apply two's complement (for an n-bit number A, complement to 2^n is $2^n - A$, or toggling all bits and adding 1)
- Easy for arithmetic

0000 0000 0000 0000 0000 0000 0000 0011_{two} = 3_{ten}
 1111 1111 1111 1111 1111 1111 1111 1101_{two} = -3_{ten}

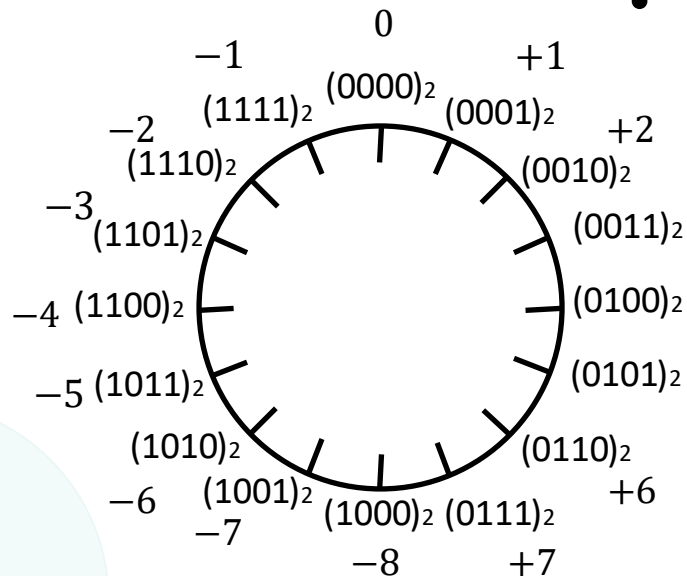
Sign
bit

Two's-Complement Representation (Signed Integer)



- 2's complement number $(a_n a_{n-1} \dots a_1 a_0)_2$ represents

$$(a_n a_{n-1} \dots a_1 a_0)_2 = -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$



- Sign extension

Two's-Complement Representation (Signed Integer)

- Two's complement treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} ($-2,147,483,648$) to $2^{31}-1$ ($2,147,483,647$)
 - Note: one negative number with no positive version
 - Every computer uses two's complement today
- Most-significant bit (MSB) (leftmost) is the sign bit, since 0 means positive (including 0), 1 means negative
 - Bit 31 is most significant (MSB), bit 0 is least significant (LSB)

4. Signed Integers: Sign-Magnitude & One's Complement

01 / Sign-Magnitude

The most intuitive method: The leftmost bit (MSB) acts as the sign flag (0 = positive, 1 = negative), while the remaining bits directly represent the absolute value (magnitude).

Critical Limitations

Dual Zero Representation: Wastes a bit pattern with two zeros (+0 = 0000, -0 = 1000), creating logical inconsistencies in calculations.

Complex Arithmetic: Requires separate hardware logic to check signs before addition/subtraction, increasing circuit complexity and cost.

02 / One's Complement

An incremental improvement: Positive numbers are identical to sign-magnitude, but negatives are formed by inverting (flipping) every bit of their positive counterpart.

Remaining Flaws

Zero Ambiguity Persists: Still suffers from +0 and -0 (all 0s vs. all 1s), a fundamental inefficiency in bit usage.

End-Around Carry: Addition requires an extra step to carry over the final overflow bit back to the sum, slowing down arithmetic operations.

4. Signed Integers: Two's Complement (The Winner)

How it Works



Positive Values: No Tricks

Stored exactly as standard binary numbers, identical to sign-magnitude for non-negative integers. The leading bit is 0.



Negative Values: The Two-Step Method

1. Compute the **one's complement** (flip all bits: $0 \leftrightarrow 1$).
2. Add 1 to the result.

Example: +5 (0101) → Flip (1010) → +1 → -5 (1011)

Why It's the Dominant Standard



1. Single, Unambiguous Zero

Eliminates the confusing duplicate representations of +0 and -0 found in other systems, ensuring clean numerical consistency.



2. Unified Arithmetic Logic

Addition and subtraction use the **same hardware**.

Subtraction is just adding the two's complement, drastically simplifying CPU design.



3. Seamless Sign Extension

Easily convert numbers to wider bit widths (e.g., 8-bit to 16-bit) by copying the sign bit, preserving the original value and sign perfectly.

4. Two's Complement: Range and Overflow

01. 32-bit Signed Integer Range

-2^{31} to $2^{31} - 1$

Decimal: $-2,147,483,648 \sim 2,147,483,647$

- **Asymmetry:** One more negative number exists because zero occupies a positive-side encoding, leaving -2^{31} without a positive counterpart.
- **Total Values:** Exactly 2^{32} (4,294,967,296) distinct integers can be represented.

02. Overflow Detection Logic

Carry In $\neq$ Carry Out (Sign Bit)

This mismatch indicates an overflow condition in addition.

- **Hardware Flag:** The CPU sets the Overflow Flag (OF) to signal out-of-bounds results.
- **Impact:** Unchecked overflow leads to silent data corruption, critical in low-level systems and cryptography.

Key Takeaway: The asymmetry of the two's complement range is the root cause of overflow edge cases. Unlike unsigned integers, signed integer overflow is undefined behavior in many programming languages (e.g., C/C++), making explicit range checks essential for robust code.

5. Floating-Point Representation

01 Fixed-Point Numbers

A simple method to represent fractions by placing a "binary point" at a fixed position in the bitstring, treating it like a decimal point.

Example: $0010.1010_2 = 2.625_{10}$

Breakdown: $(1 \times 2^1) + (1 \times 2^{-1}) + (1 \times 2^{-3}) = 2 + 0.5 + 0.125$

Key Limitations:

- Restricted range (can't handle extremes like 10^{30} or 10^{-30})
- Prone to overflow/underflow during multiplication/division

02 The Need for Floating-Point

Inspired by scientific notation, it dynamically adjusts the "binary point" to represent a vastly wider range of values efficiently.

Binary Scientific Notation:

Number = Mantissa $\times 2^{\text{Exponent}}$

Balances precision (mantissa) and scale (exponent) perfectly.

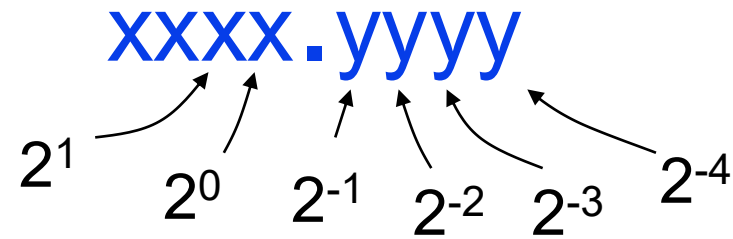
Transformative Advantages:

- Covers values from $\sim 10^{-38}$ to $\sim 10^{38}$ (single-precision)
- Enables scientific computing, AI, graphics, and financial modeling

 **Key Insight:** Floating-point is the foundation of modern computational science, solving the fixed-point problem by trading a small amount of precision for an enormous range.

Fractional

- “Binary Point” like decimal point signifies boundary between integer and fractional parts:



$$0010.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}}$$

- Fixed-point: 1234.5678; 1234.0

Fixed-Point Numbers

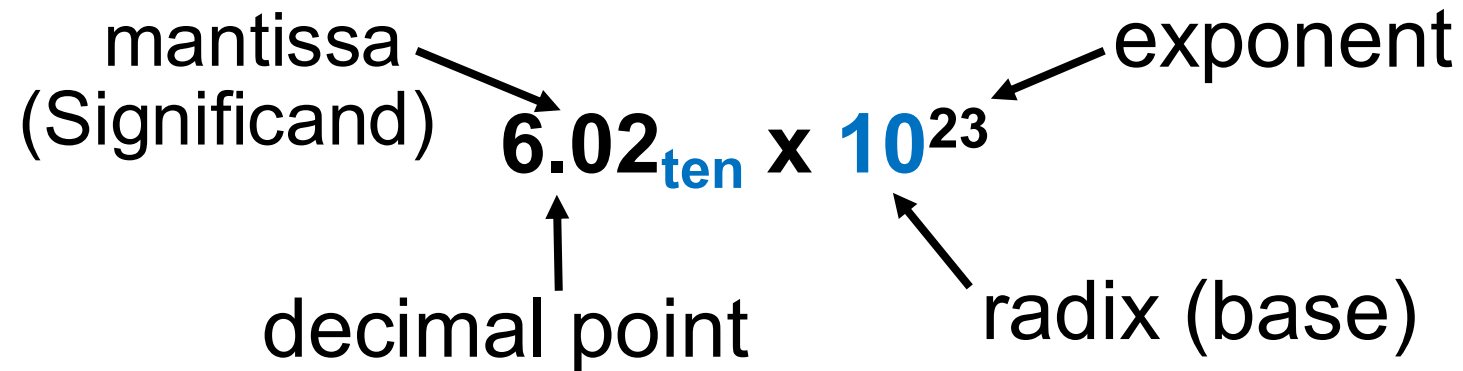
Addition is
straightforward

$$\begin{array}{r} 01.100 \ 1.5_{\text{ten}} \\ + 00.100 \ 0.5_{\text{ten}} \\ \hline 10.000 \ 2.0_{\text{ten}} \end{array}$$

$$\begin{array}{r} 01.100 \ 1.5_{\text{ten}} \\ 00.100 \ 0.5_{\text{ten}} \\ \hline 00 \ 000 \\ 000 \ 00 \\ 0110 \ 0 \\ 00000 \\ 00000 \\ \hline 0000110000 \end{array}$$

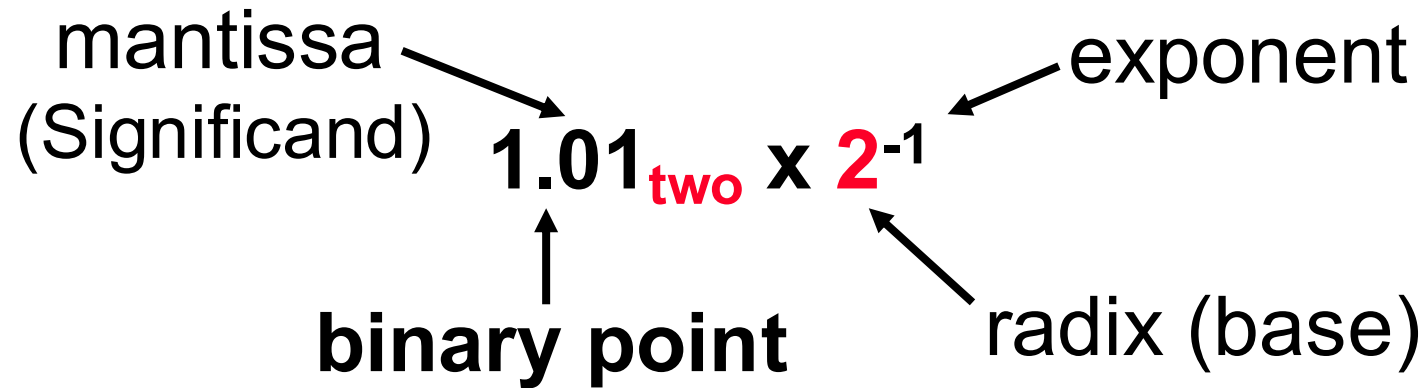
Multiplication a bit more complex
(Need to remember where point is)

Scientific Notation (in Decimal)



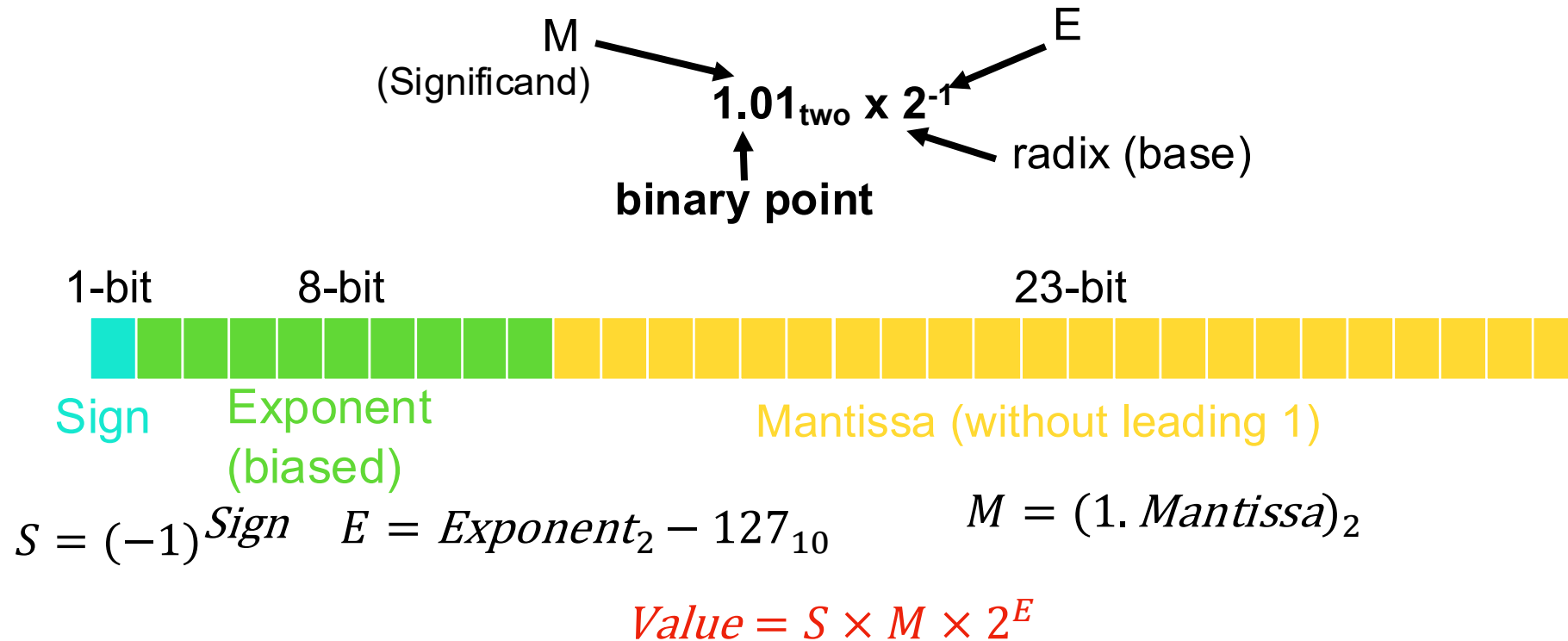
- Normalized form: no leading 0s (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
 - Normalized: 1.0×10^{-9}
 - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)



- Computer arithmetic that supports it called **floating point**, because it represents numbers where the binary point is not fixed
 - Declare such variables in C as float (32b); double for double precision (64b).
 - How to represent in computer? Everything is a number.

Single-Precision 32-bit floating point (IEEE 754)



<https://ieeexplore.ieee.org/document/8766229>

- Biased exponent: It can represent numbers in $[-127, 128]$, and allows comparing two floating point number easier (bit by bit) than the other representations.

How do we read a FP32 number?

Example

0 01111011 110000000000000000000000

- Step 1: determine the sign
 - $S = (-1)^{\text{Sign}} = (-1)^0 = 1$ (positive)
- Step 2: determine the unbiased exponent
 - $E = \text{Exponent}_2 - 127_{10} = 01111011_2 - 127_{10} = 123 - 127 = -4$
- Step 3: determine the Mantissa
 - $M = 1.\text{Mantissa}_2 = 1.11_2 = 1.75_{10}$
- Step 4: determine the converted decimal by using S , E and M
 - $\text{decimal} = S \times M \times 2^E = 1 \times 1.75 \times 2^{-4} = 0.109375$

Conversion—Store a FP32 number

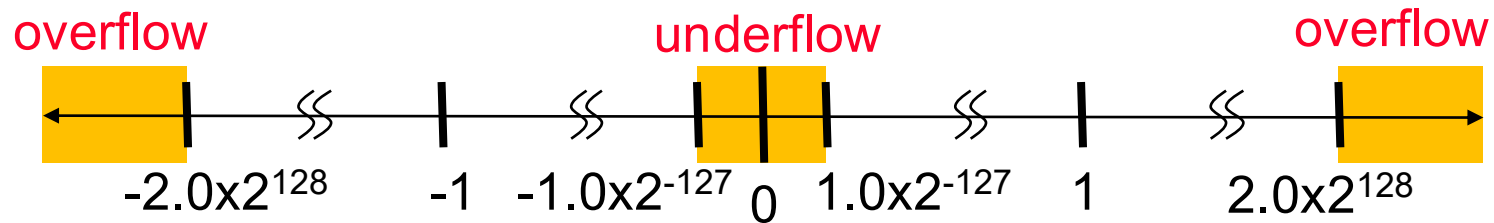
Example: 0.09375 into single precision floating point

- Step 1: determine the sign
 - Positive => Sign bit = 0
- Step 2: Convert the magnitude 0.09375 to binary
 - $0.09375_{10} = 0.00011_2$
- Step 3: Convert to scientific/normalized notation to obtain mantissa and unbiased exponent
 - $0.00011_2 = 1.1_2 \times 2^{-4}$
- Step 4: determine the biased exponent and remove the leading 1 from 1.1
 - Exponent = $-4_{10} + 127_{10} = 123_{10} = 01111011_2$, Mantissa = 1
- Step 5: padding 0s to the end of mantissa to make up to 23 bits/truncate if more than 23 bits

0 01111011 100000000000000000000000

Overflow vs. Underflow

- What if number too large/small? ($> 2.0 \times 2^{128}$, $< -2.0 \times 2^{128}$)
 - Overflow! $\Rightarrow$ Exponent larger than represented in 8-bit Exponent field
- What would help reduce chances of overflow?
- Double-precision (FP64): 1 (sign)- 11(exponent, bias 1023)- 52(mantissa) in IEEE 754 standard

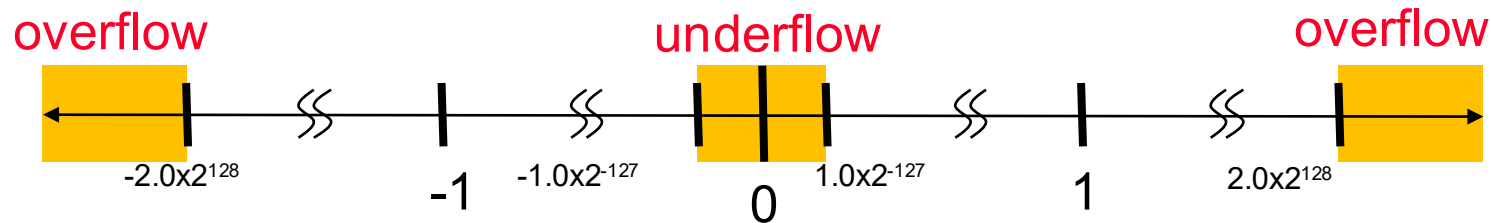


$$\pm\infty$$

- All exponent = 1s & All Mantissa = 0s
- Sign defined by the sign bit
- Valid Arithmetic
 - (+ - /) with finite numbers
 - Multiplication with finite/infinite non-zero numbers
 - Square root of $+\infty$
 - Conversion (e.g. fp32 ∞ to fp64 ∞)
 - remainder(x, ∞) for finite normal x
- Can be produced by
 - division (x/0, x $\neq$ 0), log(0), etc. along with divideByZero exception
 - Overflow with overflow exception

Overflow vs. Underflow

- What if result too small? (>0 & $< 1.0 \times 2^{-127}$, <0 & $> -1.0 \times 2^{-127}$)
 - Underflow! $\Rightarrow$ Negative exponent larger than represented in 8-bit Exponent field (have method solving it partially later)
- What would help reduce chances of underflow?
- Double-precision (FP64): 1 (sign)- 11(exponent, bias 1023)- 52(mantissa) in IEEE 754 standard

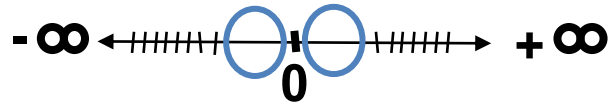


Denorms.

- Denormalized numbers
- Gap between smallest positive/largest negative FP numbers and 0

Normalization and hidden 1 is to blame!

Gaps!



- Use exponent all 0s, mantissa $\neq 0$
- No implicit leading 1, implicit exponent = -126
- Extend smallest pos./largest neg. single-precision FP to $\pm 2^{-149}$ (non-zero)
- Followed by $\pm 2^{-148}$, $\pm 1.5 \times 2^{-148}$, $\pm 2^{-147}$, $\pm 2^{-147}$, $\pm 1.25 \times 2^{-147}$, ...
- Underflow still exists



NaN (Not-a-Number)

- Resulting from invalid operations (neither overflow nor underflow)
 - e.g. operations with NaN (quiet invalid operations, generally)
 - $0 \times \infty$, $\sqrt{n}(n < 0)$, $0/0$, ∞/∞ , magnitude subtraction of infinities (signaling invalid operation exception)
- Exponent all 1s, mantissa non-zero, sign don't-care
- Why NaN?
 - Represent missing values
 - Find sources of NaNs using signaling NaNs

Representations for Special Cases

Exponent	Mantissa	Represented value
All ones	All zeros	$\pm \text{Inf}$
All ones	Not all zeros	Not a number (NaN)
All zeros	All zeros	$\pm \text{Zero}$
All zeros	Not all zeros	Sub/denormal

- <https://ieeexplore.ieee.org/document/8766229> See sections 3.4, 6&7 for more details.

Normal numbers: Exponent 1-254, -126-127 after biasing

$$\text{Value} = S \times M \times 2^E$$

5. IEEE 754: Special Cases and Edge Conditions

01. Zero (± 0)

Bits: Exponent = All 0s, Mantissa = All 0s
Represents signed absolute zero, distinguishing between positive and negative zero for directional calculations.

02. Denormalized Numbers

Bits: Exponent = All 0s, Mantissa $\neq 0$
Fills the gap between zero and the smallest normalized number, enabling representation of very small values.

03. Normalized Numbers

Bits: Exponent = 1–254, Any Mantissa
The standard range of representable floating-point values with implicit leading 1 for precision.

04. Infinity ($\pm\infty$)

Bits: Exponent = All 1s, Mantissa = All 0s
Result of arithmetic overflow (e.g., $1e308 * 10$) or division of a non-zero number by zero. Supports signed infinities for directionality.

05. NaN (Not a Number)

Bits: Exponent = All 1s, Mantissa $\neq 0$
Signals undefined or invalid operations (e.g., $0/0$, $\sqrt{-1}$). NaN is contagious: any operation involving NaN returns NaN, flagging errors.

Why Denormals Matter: Gradual Underflow

Without denormals, numbers below the smallest normalized value would abruptly underflow to zero, causing large errors. Denormals allow precision to degrade **gradually** as values approach zero, preserving numerical stability.

NaN: Robustness in Numerical Systems

NaN prevents crashes from invalid math by propagating through calculations. This allows software to detect and handle errors gracefully (e.g., in physics simulations or financial modeling) instead of aborting execution.

FP Arithmetic (normal numbers)

- Floating-point addition
 - 1.Alignment shift/preshift
 - 2.Add/subtract the significands
 - 3.Normalization shift/postshift
 - 4.Round
 - 5.Repeat 3,4 if not normalized
- (Detect inf and generate exception if necessary)

Rounding Modes

- Default to “round-to-nearest-ties-to-even” (avoid systematic biases)
- Other modes:
 - Round-to- $+\infty$ (up)
 - Round-to- $-\infty$ (down)
 - Round-towards-0
 - Round-to-nearest-ties-to-max-magnitude (roundTiesToAway, similar to SiSheWuRu)
- Used for FP arithmetics and FP-integer conversions

FP Arithmetic (normal numbers)

- Floating-point multiplication
 1. Calculate the actual exponents (if considering the bias)
 2. Add the actual exponents and add the bias (-127)
 3. Multiply the significands
 4. Normalize if necessary
 5. Round
 6. Repeat 4,5 if not normalized
 7. Calculate the sign bit
- (Detect inf and generate exception if necessary)

Summary



01. Everything is a Number

All digital data—text, images, or code—is stored as binary numbers. This universality unifies all forms of computer information at the lowest level.



02. Binary System & Logic

Built on bistable physical states (e.g., transistor on/off) and Boolean algebra. It's the mathematical and physical foundation for all computational operations.



03. Integer Representation

Unsigned integers for non-negative values; Two's Complement for signed values, simplifying subtraction and negation into efficient addition operations.



04. IEEE 754 Floating-Point

A standard for real numbers using binary scientific notation, supporting a vast dynamic range. Includes special values (± 0 , $\pm \infty$, NaN) and handles precision trade-offs.

Key Takeaway: Bits are meaningless in isolation—their value comes from **interpretation**. A sequence of bits could represent an integer, a floating-point number, a CPU instruction, or a pixel color, depending on the context defined by hardware and software.