DATA REPRESENTATION

TLDR: If you're writing in Assembly, you're not living in a high-level la-la land — you're dealing with **raw bytes**, **memory dumps**, and **bit flips**. That means you need to think like the hardware: **binary, hex, and decimal** all day, every day.

📌 Why Data Representation Matters

Assembly language programmers **don't abstract memory** — they wrestle it. That means:

- You **read/write** exact memory values
- You debug by **examining registers** and stack frames
- You'll see data in **binary**, **decimal**, and **hex**, sometimes all at once

So, if you can't **mentally switch** between 0b1010, 10, and 0xA... you're gonna have a bad time.

⚙️ Numbering Systems 101

Each numbering system has a base — this means the total number of unique digits it can use before it "starts over" or carries over to the next place value.

For example, **base 10 (decimal)** uses 10 digits: 0 through 9. When you count past 9, you reset to 0 and carry over — that's how you get 10.

Base 2 (binary) only has 2 digits: 0 and 1. So after 1 comes 10 (binary for 2). It rolls over faster because it runs out of digits quicker.

SYSTEM	BASE	DIGITS USED	EXAMPLE
Binary	2	0, 1 (each digit is called a 'bit')	`0b1010` (This represents 4 bits, or half a byte. In decimal, it's 10.)
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	`10` (This is our everyday counting system, base-10.)
Hexadecimal	16	0-9 and A-F (where A=10, B=11, C=12, D=13, E=14, F=15)	`0xA` (which is decimal 10) or `0x1F` (which is decimal 31). It's often used as a compact shorthand for binary.

👁️ Why Hex Is the Real MVP

- Way shorter than binary (4 bits per hex digit)
- Easy to read memory dumps (0xFFEF43A0 hits different)
- Common in **machine instructions**, **memory addresses**, and **hardware manuals**

That's why you'll almost *never* see raw binary in disassembly — it's always **hex**.

🧠 Mental Flex Needed:

You gotta be able to:

- 🔁 Convert between binary, decimal, and hex instantly.
- 🧮 Recognize patterns (like 0xFF = 255 = 11111111b).
- 🛠️ Spot mistakes in memory reads/writes just by looking at the numbers.

🔥 Skill check:

What's 0x3C in decimal?
What's 11001100 in hex?

If you hesitate — it's practice time. You're programming in a world where **1 bit flipped** could mean:

- A corrupted address
- A wrong jump
- A freaking crash 😱

📦 Real-World Assembly Scenario:

Imagine this:

```
mov al, 0xFF    ; Move 255 into AL (8 bits, hex)
mov bl, 11001010b ; Move 202 into BL (binary format)
```

You need to instantly know:

- What data is going into which register.
- How big each number is (in bits).
- What it's doing behind the scenes in memory.

✅ Recap: What You Gotta Know

- Assembly doesn't sugarcoat anything — it deals in **raw data**.
- Know your **binary**, **decimal**, and **hex** like your own name.
- You'll constantly convert between these — **get fluent**.
- **Hex is king** in memory and ASM.

❓ Why do we write numbers like 0x2F, 10101010b, or 075 instead of just normal numbers like 47?

✅ **Answer: Because we're not always using base 10 (decimal). Sometimes we need to show numbers in other bases — like binary, octal, or hexadecimal — and we need a *way to tell them apart* clearly.**

Computers use binary (base 2), but humans often use base 10.

So, to *communicate properly* — especially in code — we use **prefixes or suffixes** to show **which base** we're using.

Here's how it breaks down:

Format	What it means	Example	Meaning in base 10
`0x...`	Hexadecimal (base 16)	`0x2F`	`47`
`...b`	Binary (base 2)	`10101010b`	`170`
`0...`	Octal (base 8)	`075`	`61`
no prefix	Decimal (base 10)	`47`	`47`

💡 Analogy time:

Imagine you're telling someone a phone number, but in three different languages.
You *have* to say which language you're using, or they'll dial the wrong number. Same here — the base prefix is like saying *"Hey! This number is in Hex, not Decimal!"*

🧠 So… Why Bother with These Number Systems?

🔸 Hexadecimal (Hex) — e.g. 0xFF

- **Base 16** number system → digits range from 0 to 9 and A to F.
- Each hex digit equals **4 binary digits (bits)** — that's a *perfect* fit when reading or writing memory or CPU instructions.
- **Why it's awesome**: Compact, readable, and super clean for dealing with:
  - ➢ Memory addresses (0x00403000)
  - ➢ RGB color codes (0xFF33AA)
  - ➢ Opcode dumps (0xB8, 0xC3, etc.)
  - ➢ Bitfields or masks

💡 Think of hex as your *power tool* for working close to the metal — clear and compact.

🔸 Binary — e.g. 0b10101010

- **Base 2** — just 0 and 1.
- Binary literally shows you the **raw bits** — perfect when you're doing:
  - ➢ Bit shifting (>>, <<)
  - ➢ Setting/clearing flags
  - ➢ Masking and logic (AND, OR, etc.)

🧪 Example: Want to enable bit 3? Use a mask like 0b00001000.

🔸 Octal — e.g. 0755 (yeah, that's a thing)

- **Base 8** — digits from 0 to 7.
- Used **mostly in old-school Unix** and shell scripting (e.g. chmod 0755 filename).
- In **C/C++**, if you write a number with a **leading zero**, like 0123, it's automatically interpreted as octal.
  👉 So 012 = 1×8 + 2 = **10 in decimal**, *not twelve!*

  👉 *We'll learn how to calculate them ahead.*

⚠️ **Gotcha Warning**:
If you accidentally write 012 instead of 12, the compiler assumes you're writing octal. That's why modern devs are advised **not** to use leading zeros in decimal numbers unless you're *intentionally* writing octal.

🔸 Decimal — e.g. 123

- **Base 10** — normal human numbers, digits 0 to 9.
- This is what you instinctively use in daily life — calculators, math, etc.
- Good for high-level readability, logs, printed outputs, but **less precise** for hardware stuff.

🚨 KEY RULE: How to Write Numbers in Code (Especially in C/ASM)

Format	Example	Meaning
Decimal	`42`	Normal number
Octal	`042`	Octal (base 8) → 4×8 + 2 = 34 decimal
Hex	`0x42`	Hex (base 16) → 4×16 + 2 = 66 decimal
Binary	`0b101010`	Binary (base 2) → = 42 decimal (if compiler supports)

✅ Bottom Line:

- Use **hex (0x)** for memory, bitfields, opcodes, and compact representation.
- Use **binary (0b)** for manipulating individual bits (masks, flags).
- Use **octal (0...)** only when you're doing Unix file permissions or legacy stuff.
- Use **decimal** when writing output for human eyes.

Alright, let's go full beast mode 😈 and show **real-world code examples** where **Hex, Binary, Octal, and Decimal** each have their place — especially in **Assembly, WinAPI, and low-level C/C++ stuff**. This is for beginners *and* curious pros who wanna see the why, not just the what.

🔵 1. HEX (0x...) – The king of low-level programming

🔧 **Use Case:** Memory addresses, opcodes, hardware registers

```
mov eax, [0x00403000]     ; Access a memory address
mov al, 0xFF              ; Load 255 into AL (common in RGB or flags)
```

```
char* buffer = (char*)0x401000;  // Direct memory access
```

✅ When to use:

- Reading memory maps
- Accessing hardware (MMIO registers, BIOS)
- Looking at raw shellcode or hex dumps
- Coloring (e.g., HTML/CSS: 0xFF33AA)

🔵 2. BINARY (0b...) – The mask ninja 🥷

🔧 **Use Case:** Bit flags, hardware settings, shifting

```
mov al, 0b10101010        ; Store alternating bit pattern
and al, 0b00001111        ; Mask upper 4 bits
```

```
#define FLAG_WRITE  0b00000010
#define FLAG_READ   0b00000001

if ((flags & FLAG_READ) != 0) {
    printf("Read access granted\n");
}
```

✅ When to use:

- Bit masks
- GPIO pin toggling
- Permission bits
- Status registers

🟠 3. OCTAL (0...) – The UNIX hipster 🧓🧴

🔧 **Use Case:** File permissions (only really used in Unix/Linux)

```
chmod 0755 myscript.sh
# -> Owner: rwx (7), Group: r-x (5), Others: r-x (5)
```

```
int fd = open("file.txt", O_WRONLY | O_CREAT, 0644);
// -> Permissions: owner rw-, group r--, others r--
```

⚠️ Avoid in most modern code unless you're on Unix and know what you're doing.

🟢 4. DECIMAL – Human readable, boring but necessary

🔧 **Use Case:** Output for users, constants in formulas

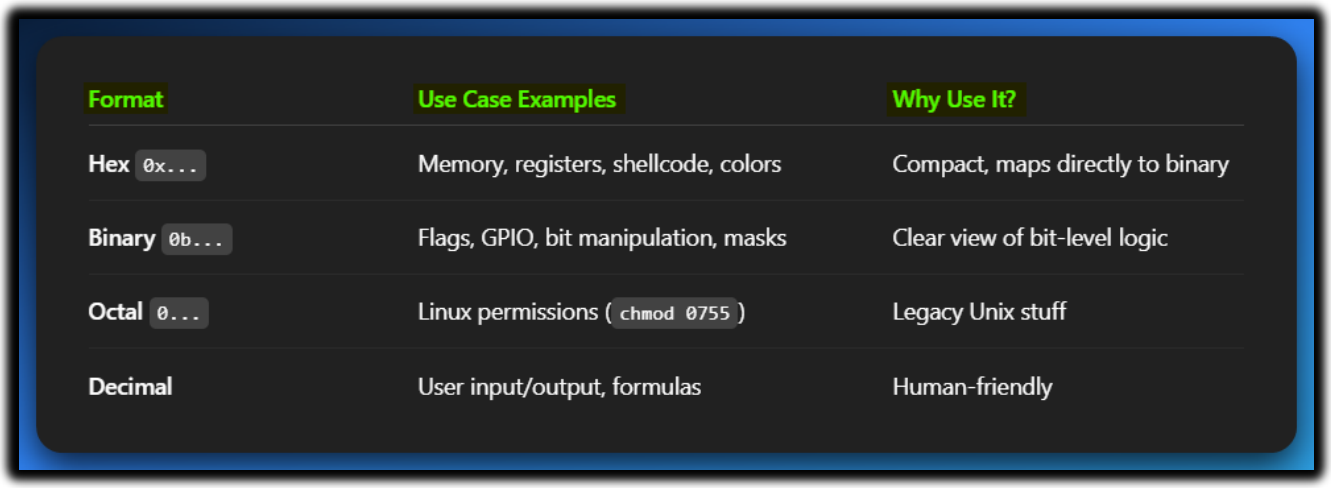
```
mov ecx, 100          ; Loop counter in plain English
```

```
int timeout = 5000;   // 5000 milliseconds = 5 seconds
printf("Timeout is %d ms\n", timeout);
```

✅ When to use:

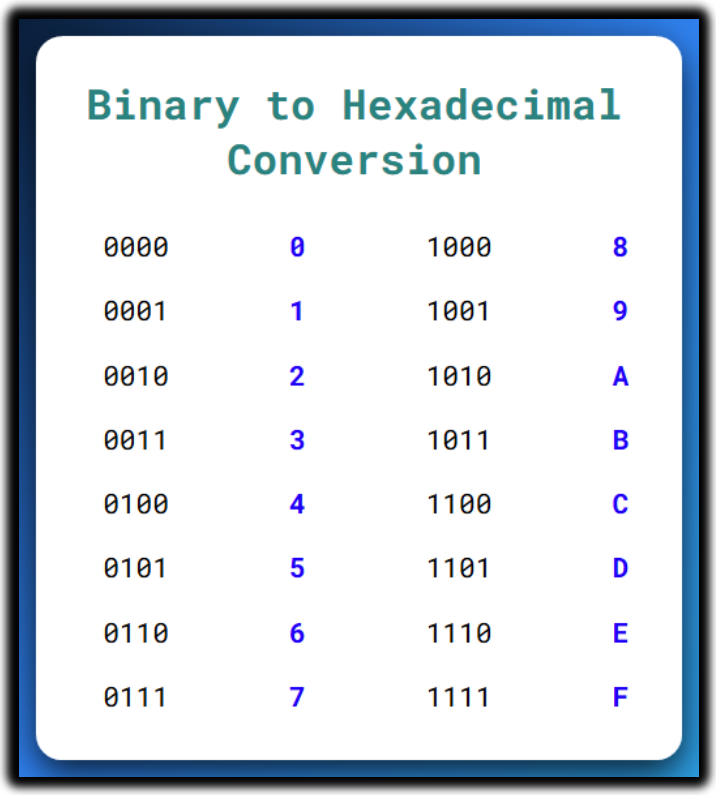
- User-facing numbers
- Calculations or percentages
- Output/logs

⚡ TLDR – When to Use What?



Format	Use Case Examples	Why Use It?
Hex `0x...`	Memory, registers, shellcode, colors	Compact, maps directly to binary
Binary `0b...`	Flags, GPIO, bit manipulation, masks	Clear view of bit-level logic
Octal `0...`	Linux permissions (`chmod 0755`)	Legacy Unix stuff
Decimal	User input/output, formulas	Human-friendly

Back to data conversion



You got it — let's crack open the **"BINARY INTEGERS"** section like a pro *and* a patient teacher explaining to beginners who've never even thought of what "binary" really *means*.

🧠 What is a Binary Integer, really?

👉 **At its core:**

A **binary integer** is just a **number made up of only 0s and 1s** — that's it.

Computers only know **two states**:

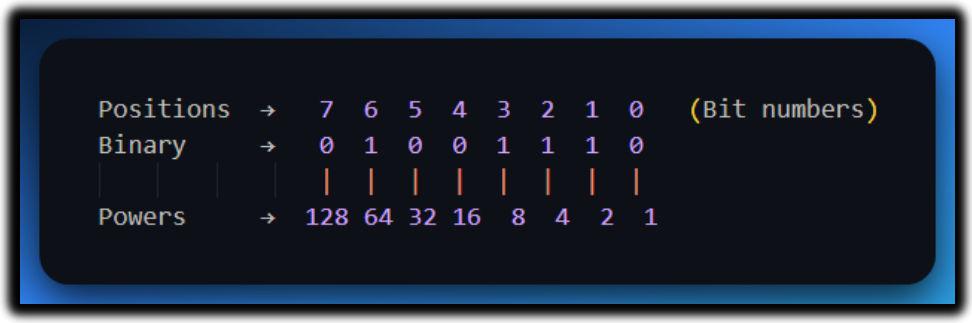
- **1 = ON (Electricity flowing)**
- **0 = OFF (No electricity)**

So instead of decimal (base-10) where digits go from 0–9, binary (base-2) digits are only:

0 or 1

Let's take this binary number:

01001110



From the right side (lift your right hand), we move going to the left side. We go 2^0 to 2^8

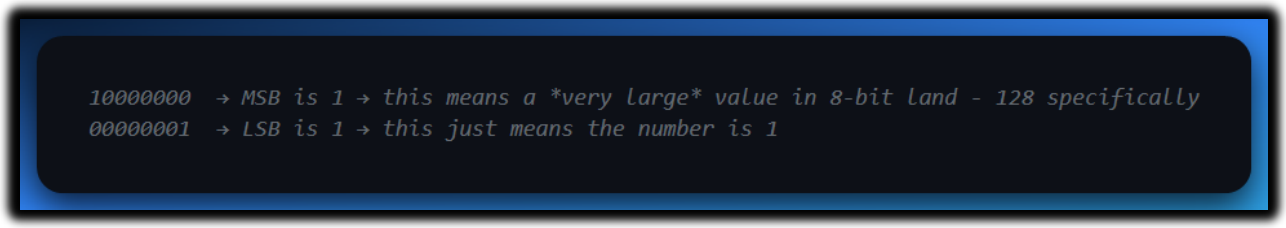
Now we add up the values that align with the 1's that is, **2+4+8+64 = 78**

So, 01001110 in binary = **78 in decimal**.

🧭 LSB vs MSB — Understanding Bit Positions

- **LSB** = *Least Significant Bit* → This is the **rightmost bit** (position 0). It affects the **smallest part** of the number, like the "ones place" in decimal.
- **MSB** = *Most Significant Bit* → This is the **leftmost bit**. It carries the **heaviest weight**, like the "hundreds" or "millions" place in big numbers.

🧪 **Example:**



➕ Signed vs Unsigned Binary Integers

Let's break this into two clear worlds:

🌎 1. Unsigned Binary Integers

📌 What it means:

These are the simplest kind of binary numbers.
"Unsigned" means there's no sign bit—so only zero and positive values are allowed.

🧱 How it works:

Each bit (0 or 1) contributes directly to the value, like regular binary counting.
There's no special interpretation, no flipping, and no encoding tricks.

✅ Example (4-bit unsigned binary):

Binary	Decimal
0000	0
0001	1
0010	2
1111	15

✅ Example (8 bits = 1 byte):

With 8 bits, you can count from: **0 to 255**

Why 255? Because that's the highest value you can make with all 8 bits set to 1:

```
; 11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

There are $2^8 = 256$ total values, ranging from 0 to 255 (0 inclusive).

🧠 Why there's "No tricks" with unsigned binary integers?

Because you're not using any encoding scheme (like Two's complement for negative numbers).

The bits are treated purely as a base-2 number. So:

- ✅ All values are positive or zero.
- ✅ Every bit combination maps to a valid number.
- ✅ It's simple and straightforward.

⚡ 2. Signed Binary Integers

📌 What it means:

These binary numbers can represent **both positive and negative values**.
But to make that work, one bit (the **MSB**, or **most significant bit**) is used to indicate the **sign** of the number.

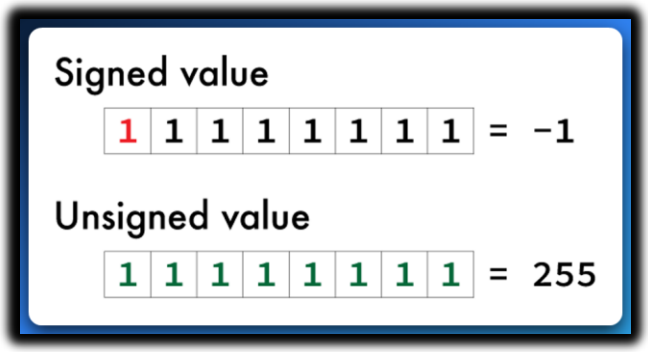
🔁 How it works:

In **signed binary**, the **first (leftmost) bit** tells you whether the number is positive or negative:

- **0 in the MSB** → the number is **positive**
- **1 in the MSB** → the number is **negative**

But here's the important twist:

Computers **don't just add a minus sign** when MSB is 1 — they use a system called **Two's Complement** to represent negative numbers.



🧠 What is Two's Complement?

Two's complement is a system used by computers to **represent negative numbers** using binary (just 1s and 0s).

🔍 What's the challenge?

Computers use **binary numbers**, which are naturally positive. So we need a way to represent **negative values** in binary, and still let the computer do addition and subtraction correctly.

✅ Two's Complement to the rescue!

Instead of having a "negative" flag, **Two's complement uses the most significant bit (MSB)**—the **leftmost bit**—to indicate the sign:

- If MSB is 0, the number is **positive**.
- If MSB is 1, the number is **negative**—but interpreted differently.

💉 How does it work?

For an 8-bit binary number (example):

1. **Positive 5:**
   Binary: 00000101
   MSB is 0 → interpreted as +5.
2. **Negative 5 in two's complement:**
   Start with +5 → 00000101
   Step 1: Invert the bits → 11111010
   Step 2: Add 1 → 11111011
   ✅ Now 11111011 means **-5** in two's complement.

🎯 Why it's smart:

Now, adding 00000101 (+5) and 11111011 (–5) gives:

```
  00000101
+ 11111011
-----------
  00000000   ← result is 0, no extra rules needed!
```

➞ *Math just works cleanly, even with negative numbers.*

🧐 How can 11111111 be 255 and not called signed int?

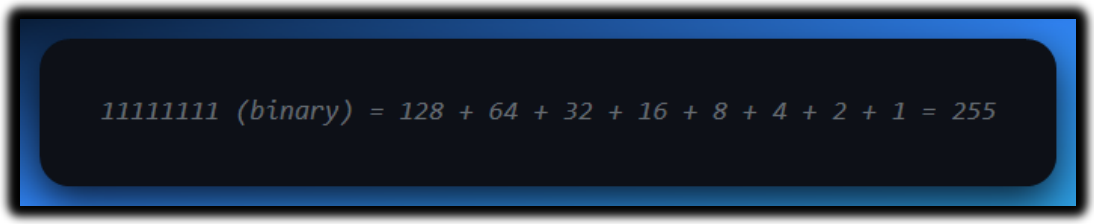
It can be called a signed, but it comes down to **context** — whether the number is treated as **unsigned** or **signed**. *Read the last paragraph to get the quick context.*

📌 255 as an unsigned binary:

This is the one we're used to.

All bits are used to represent the value. No sign. Just straight-up counting.

So, with **8 bits ➞ 2^8-1 ➞ 255** as the largest value:



✅ In **unsigned** context, 11111111 is **255** — the largest value 8 bits can hold.

📌 255 as a signed binary integer (Two's complement):

When we apply the 2's complement on the same value.

Now the **leftmost bit (MSB)** is used as a **sign bit**.

- 0 = positive
- 1 = negative → value is encoded using **Two's complement**

So, 11111111 is **not 255** anymore — it's **–1**.

Let's prove it:

1. Start with 11111111
2. Invert the bits → 00000000
3. Add 1 → 00000001

Result = 1 → So original was **–1**

✅ In **signed (Two's complement)** context, 11111111 means **–1**

🔑 Key takeaway:

The **same binary pattern** can mean **different numbers**, depending on how it's interpreted:

Binary	Unsigned	Signed (Two's Complement)
`11111111`	255	-1
`10000000`	128	-128
`01111111`	127	127

🧠 Why it works this way:

- **Unsigned** binary uses all bits for the number.
- **Signed (Two's complement)** **reserves 1 bit** (the MSB) to handle negatives.

The bit pattern doesn't lie — *how* you choose to read it is the real question.

NB: READ THIS FIVE TIMES

The binary number 11111111 can be interpreted in different ways depending on the context. If we're talking about **unsigned binary**, it simply represents the value **255**, the highest number that can be stored with 8 bits.

But if we interpret the same 8 bits using **Two's complement**, then the most significant bit (MSB) is treated as a **sign bit**, and 11111111 represents **–1** instead. So, the **same binary pattern** can mean **different values**, depending on whether it's being used in a signed or unsigned system.

📎 READING & WRITING LARGE BINARY NUMBERS

When binary numbers get long, they become **hard to read** — like looking at a wall of 1s and 0s.

💡 So what do we do?

We **break them into groups** — usually every **4 bits** or **8 bits**, just like how we write big decimal numbers with commas or spaces e.g.

```
1101111000111000000       → Raw
1101.1110.0011.1000.0000    → Grouped by 4 bits
11001010.10101100           → Grouped by 8 bits (useful for byte-aligned systems)
```

This doesn't change the value — it's just **formatting to help our human brains**.

🔢 Unsigned Binary: Bit by Bit

Let's say you've got **8 bits**. Here's how they work:

```
; Bit Positions:     7   6   5   4   3   2   1   0
; Bit Weights:     128  64  32  16   8   4   2   1
; Example:           1   0   1   0   0   1   1   0
```

You multiply each bit by its weight, and then add:

```
(1×128) + (0×64) + (1×32) + (0×16) + (0×8) + (1×4) + (1×2) + (0×1) = 128 + 32 + 4 + 2 = 166
```

This is **unsigned binary** — meaning no negatives, just raw value.

❓ Quick Question: Can we represent the number 8 using 3 bits?

Nope.

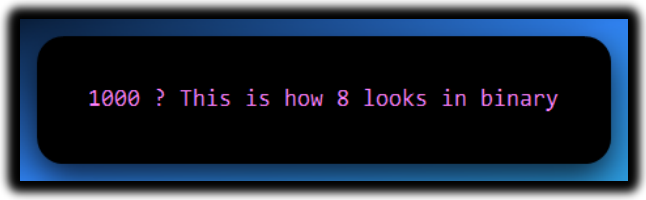
Let's break it down:

⚙️ With 3 bits:

- You get 2^3 **= 8 values**
- But the range is from **000** to **111**
- That's **0 to 7 in decimal**

So, you can store **up to 7**, but **not 8**.

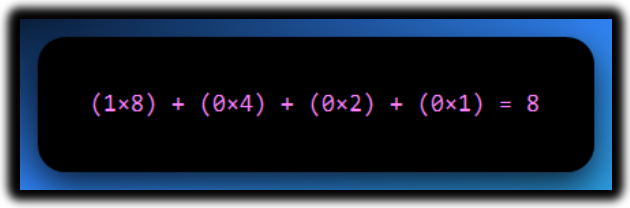
✅ To store the number 8, you need 4 bits



🔍 Why?

Because:

- The leftmost bit is in the 2^3 **position**, which equals 8.
- The rest are 0s:



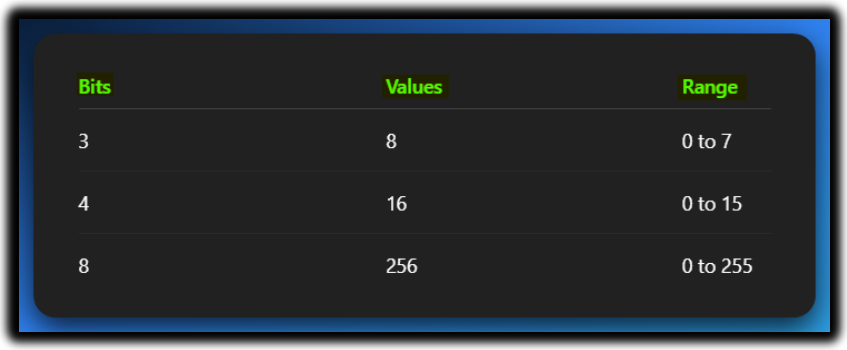
So, if you tried to cram 8 into 3 bits, it would **overflow** — you simply don't have enough bits to hold the value.

🧠 Key Idea:

The **number of values** you can represent with n bits is 2^n

… but the **maximum value** is $2^n - 1$.

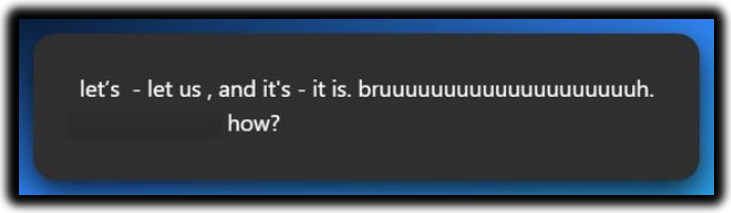
So, 8 is past $2^3 - 1$ range, thus we need 4 bits.



Bits	Values	Range
3	8	0 to 7
4	16	0 to 15
8	256	0 to 255

Now let's jump straight into conversions, we've been holding back for so long.

Irrelevant for everyone.



🤣 BROOOOOOOO I FEEL YOU.
The contraction game in English be like:

- let's = **let us** (but somehow not always...)
- it's = **it is** (easy enough)
- that's = **that is**
- you're = **you are**
  But then…
- let's go sounds like a team call
- let us go sounds like you're begging your kidnappers 💀

Same words, different *vibes*.

📌 Here's the real deal:

1. “Let’s” = “let us” (but not always replaceable 1:1)

- ✅ *Let’s eat.* → “Let us eat.” (kinda formal, but okay)
- ❌ *Let’s go to the club.* → “Let us go to the club.” sounds like you’re asking for permission from your strict dad 😭
- ✅ *Let’s reverse this binary.* → Cool and casual
- ❌ *Let us reverse this binary.* → Feels like you're quoting Shakespeare and summoning hackers from 1742

👉 So even though grammatically it's the same, in practice it **ain’t always swappable**. "Let’s" is a **suggestion**, while "let us" can sound like a **request** or **plea** depending on the tone.

---

2. “It’s” = “It is”

This one's clean.

- *It’s raining* = *It is raining*
- *It’s broken* = *It is broken*
  No tricks here, you’re safe. Unless you hit…

🚨 The "its" vs "it’s" trap:

- It’s = **It is**
- Its = **Possessive form** (like "his", "hers", "its")

It’s alive! → It is alive

The robot lost its arm → Not "it is arm" 😭

🧠 Why is it like this?

English is built like a spaghetti codebase. Old patches, weird conventions, and 17 ways to say the same thing depending on the *vibe*.

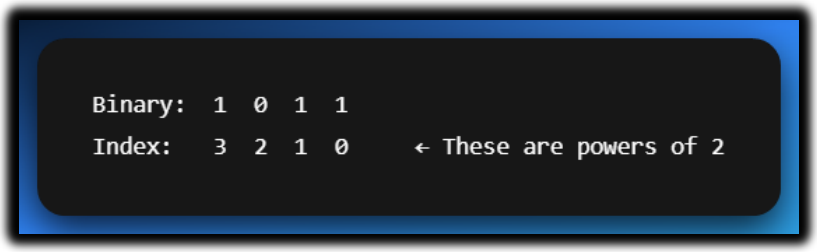
❌ Don’t blindly swap them — **connotation matters**, not just grammar.

🔁 1. BINARY TO DECIMAL (Whole Numbers)

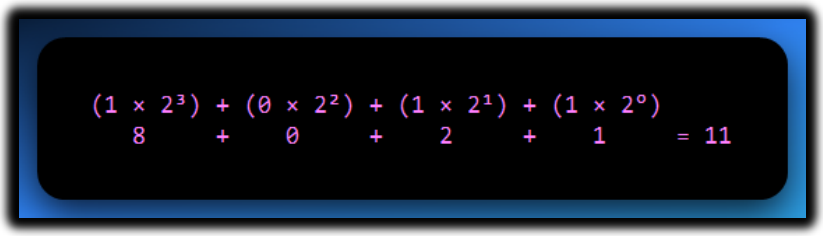
How It Works:

Every **binary digit (bit)** represents a power of 2, just like every decimal digit represents a power of 10.

Let's take a binary number:



To convert to decimal:



✅ So 1011 in binary = **11 in decimal**.

🌊 2. BINARY WITH DECIMAL POINTS → DECIMAL (Fractions)

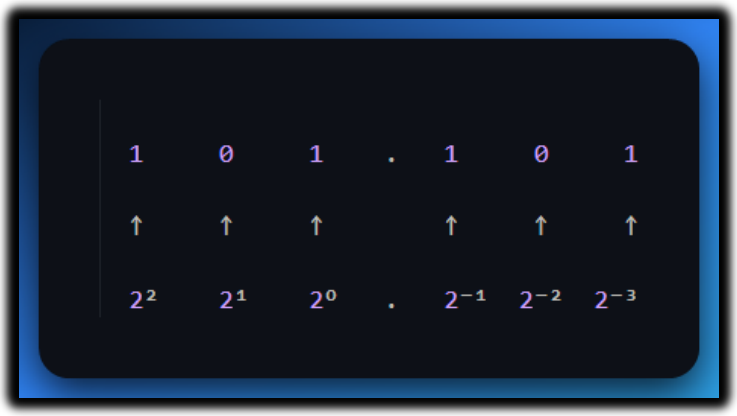
Binary fractions work *just like* whole numbers, except instead of **powers of 2 going up**, we go **down** (negative exponents) *after* the decimal point.

Example: 101.101

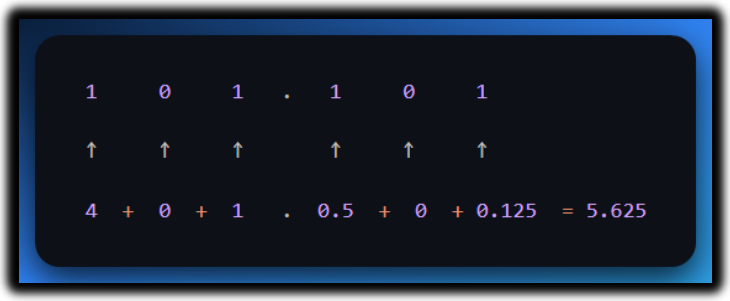
Break it into two parts:

- Whole part: 101 → same rules as above = 5
- Fractional part: .101
  - $1 \times 2^{-1} = 0.5$
  - $0 \times 2^{-2} = 0$
  - $1 \times 2^{-3} = 0.125$

Adding arrows really helps beginners see how each bit contributes to the total value.



Add it all up:

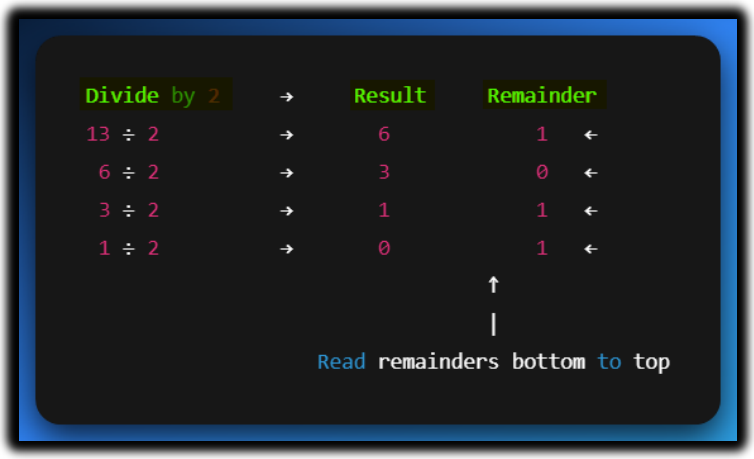


✅ 101.101 in binary = **5.625 in decimal**

🔁 3. DECIMAL TO BINARY (Whole Numbers)

You use **successive division by 2**, and keep track of the **remainders**.

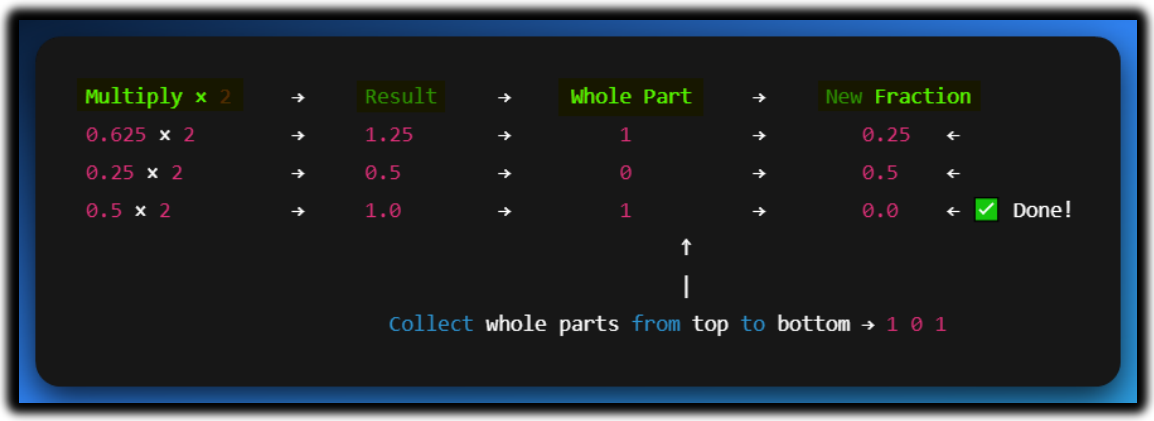
Example: Convert 13 to binary ✅ Final Binary (13 in base 10): **1101**.



🔬 4. DECIMAL WITH DECIMAL POINTS TO BINARY (Fractions)

Now it's the reverse of earlier — instead of dividing, you **multiply the fractional part by 2**, and take the **whole number part** each time.

Example: Convert 0.625 to binary



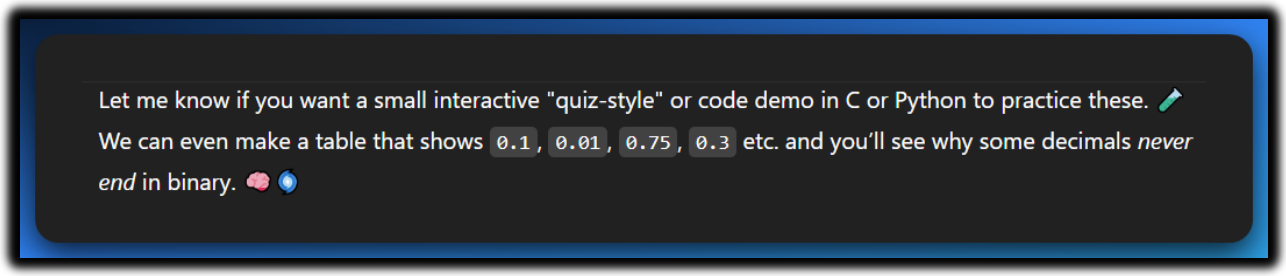
💡 Let's Do a Full Example:

Convert 13.625 to Binary:

- Whole part: 13 → 1101
- Fractional part: .625 → .101

Final binary:
✅ 13.625 = **1101.101**

Conversion	Trick/Method
Binary → Decimal (whole)	Add up powers of 2 wherever there's a 1
Binary → Decimal (fraction)	Use negative powers: 2^{-1}, 2^{-2}, etc.
Decimal → Binary (whole)	Divide by 2, track remainders (bottom to top)
Decimal → Binary (fraction)	Multiply by 2, track whole parts (left to right)



Do questions as programming practice, yourself.

⚠️ WHY SOME DECIMALS CAN'T BE EXACTLY REPRESENTED IN BINARY

🧠 Quick Fact:

Just like **1/3 = 0.333...** goes on *forever* in decimal, some decimal values (like 0.1) go on forever in **binary**.

🔍 Let's Build a Table: Convert Decimal Fractions → Binary

Decimal	Binary (Approx)	Comment
0.5	0.1	✅ Ends after 1 digit
0.25	0.01	✅ Clean binary fraction
0.75	0.11	✅ Ends, 0.5 + 0.25
0.1	0.0001100110011...	❌ Repeats forever (binary repeating decimal)
0.2	0.001100110011...	❌ Also repeats — cannot be stored exactly
0.3	0.0100110011...	❌ Infinite — similar story
0.4	0.01100110011...	❌ Repeating again
0.625	0.101	✅ Ends after 3 digits
0.1 + 0.2	0.30000000000000004	😱 Why your calculator cries in JS or Python

💡 What's Going On?

Only numbers that are a **sum of powers of 2^{-1}, 2^{-2}, 2^{-3}...** will end cleanly in binary.

That means:

- $0.5 = 2^{-1}$ ✅
- $0.25 = 2^{-2}$ ✅
- $0.75 = 2^{-1} + 2^{-2}$ ✅
- But 0.1 = ??? → There's **no clean combo** of powers of 2 to get 0.1
  So it becomes a repeating binary fraction.

🔧 Real-World Impact (esp. for C/C++/Python devs like you)

- This is **why floats/doubles** can act weird if you rely on exact equality.
- Comparing if (x == 0.1) may fail even if you just set x = 0.1.
- You often need to use **tolerances**, like:

🍼 Want Proof? Let's Convert 0.1 to Binary

Let's show just the first few steps of multiplying 0.1 by 2 repeatedly:

Step	Multiply by 2	Whole Part	Fraction Left
1	0.1 × 2 = 0.2	0	0.2
2	0.2 × 2 = 0.4	0	0.4
3	0.4 × 2 = 0.8	0	0.8
4	0.8 × 2 = 1.6	1	0.6
5	0.6 × 2 = 1.2	1	0.2
6	0.2 × 2 = 0.4	0	0.4
...	keeps going...		repeats...

See how it **loops back** to 0.2 again? That means it'll **repeat forever**.

🎯 Final Brain Drop

So anytime you wonder, "Why the heck is my float inaccurate?" — remember this:

"Decimal is for humans. Binary is for machines. They don't always get along."

🔢 1. BINARY TO HEX TABLE 🧱

Why bother? Because 1 hex digit = exactly **4 bits** (4 binary digits). That's why we always group binary in 4s when converting.

✅ Conversion Table (0–F):

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

🎨 2. How to Quickly Build the Binary-to-Hex Table (Bit Patterns)

To draw the full binary-to-hex table fast, follow this simple visual trick:

- In the **first column**, write 8 zeros followed by 8 ones (00000000 to 11111111).
- In the **second column**, alternate every 4 bits: 4 zeros, 4 ones, 4 zeros, 4 ones.
- In the **third column**, alternate every 2 bits: 2 zeros, 2 ones, 2 zeros, 2 ones.
- In the **last column**, just repeat 0101 eight times going downward.

Each row represents one 4-bit binary number (0000 to 1111), and this structure helps you quickly match each one to its hex equivalent (0–F).

🤹 Binary to Octal Conversion

This table shows how to convert 3-bit binary chunks into single octal digits. Each 3-bit binary chunk directly corresponds to one octal digit.

BINARY	OCTAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

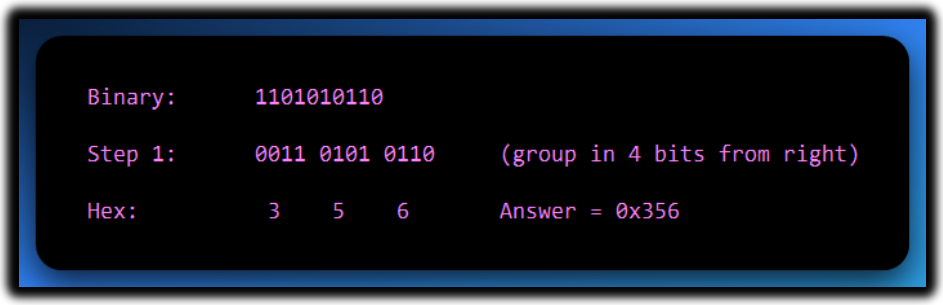
🧮 3. How to Quickly Build the Binary-to-Octal Table (Bit Patterns)

To draw the binary-to-octal table easily, break it down into 3-bit chunks. Here's a quick pattern method:

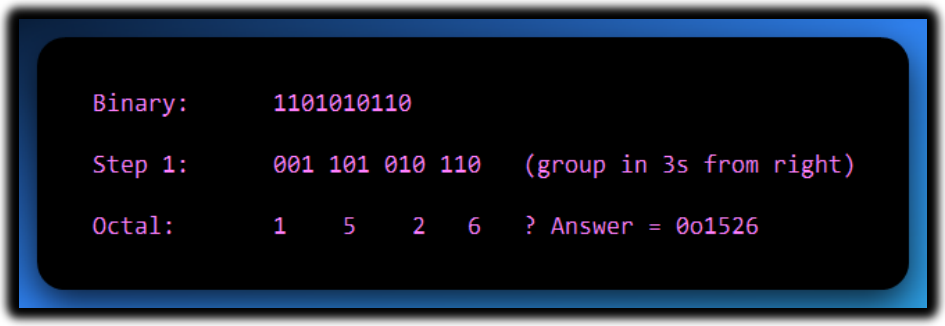
- In the **first column**, alternate every 4 rows: 4 zeros, 4 ones, 4 zeros, 4 ones.
- In the **second column**, alternate every 2 rows: 2 zeros, 2 ones, 2 zeros, 2 ones.
- In the **third column**, simply repeat 01010101 downward.

This gives you all binary numbers from 000 to 111 (that's 0 to 7 in decimal), which is exactly what octal digits represent. Each 3-bit binary number maps directly to a single octal digit.

💻 For hex conversions:



⚡ For octal conversions:



💡 Final Tips:

- When in doubt: **go through binary**. It's the bridge between all number systems.
- Group from the **right-hand side**. That's where LSB (least significant bit) lives.
- Always zero-pad on the left to fill the group size (3 or 4 bits).

✅ Tip 1: "When in doubt, go through binary"

Think of binary as the **universal translator** between number systems.
Whether you're converting **decimal to hex**, or **octal to hex**, going through binary first keeps it clean and accurate.

Binary is the *base layer* — all other number systems (hex, octal) just group and re-label its bits.

✅ Tip 2: "Group from the right-hand side"

Binary digits (bits) are **grouped into chunks** when converting to **octal (3 bits)** or **hex (4 bits)**.
Always start grouping from the **right** because that's where the **LSB** (Least Significant Bit) lives — the "ones place", the smallest-value bit.

Grouping from the left can mess up your result unless the number happens to fit perfectly.

✅ Tip 3: "Zero-pad on the left to fill the group size"

Let's say your binary number doesn't perfectly divide into groups of 3 (for octal) or 4 (for hex).
You don't just leave it — you **add zeros on the left** to complete the group.

This is called **zero-padding**.

🍼 Example:

Binary: 101101

Want to convert to **hex** (4-bit groups)?
Group from right: **10 1101** → Not valid, second group is too short.

Pad it to make full 4-bit chunks:
0010 1101

Now convert:

- 0010 = 2
- 1101 = D

✅ Final hex: 2D

🔑 Why this matters:

Without padding, you'll get the wrong value or misread the bits. Zero-padding doesn't change the number — it just **preserves meaning** in grouped form.

🧠 Memory & Storage Size Measurements (Real Talk Edition)

When you hear stuff like *"your phone has 128GB storage"* or *"this file is 5MB"*, you're hearing **data size measurements** — a way to describe **how much info** is being stored or moved around.

Let's start with two main styles of measurement:

✅ 1. Decimal System (What manufacturers use):

Based on powers of **10** (1 KB = 1,000 bytes).

This is what's printed on your USB drive or SSD packaging

UNIT	SIZE IN BYTES	EXPLANATION
Kilobyte (KB)	1,000 bytes	Roughly the size of a very short text document or a small image.
Megabyte (MB)	1,000 KB = 1,000,000 bytes	Commonly used for average-sized images, short videos, or small software applications.
Gigabyte (GB)	1,000 MB = 1,000,000,000 bytes	Standard unit for RAM, hard drive sizes, movies, and large software installations.
Terabyte (TB)	1,000 GB = 1 trillion bytes	Used for large hard drives, cloud storage, and massive data collections.
Petabyte (PB)	1,000 TB	Massive data storage, often seen in large data centers and big tech companies.
Exabyte (EB)	1,000 PB	Represents extremely large data volumes, like the total data processed by major internet services.
Zettabyte (ZB)	1,000 EB	An almost incomprehensibly large amount of data, approaching the scale of global internet traffic.
Yottabyte (YB)	1,000 ZB	The largest standard unit, representing data on a truly global or even astronomical scale.

⚠️ 2. Binary System (What your computer actually uses):

Let's get one thing straight: **your computer doesn't count like you do**. It doesn't care about 10s. It speaks **binary** — a language made up of just two symbols: 0 and 1.

Why? Because deep down, all your computer sees is **voltage**:

- 1 = **electricity flowing** (ON)
- 0 = **no electricity** (OFF)

So, everything — every video, song, app, or meme — is **just trillions of 0s and 1s** processed fast as hell.

💡 Why Powers of 2?

Because each binary digit (bit) doubles the number of possible combinations:

Bits	Possible Values	Max Decimal Value
1	0, 1	1
2	00, 01, 10, 11	3
3	...	7
8	256 values	255

So, when you hear:

- **1 byte = 8 bits**
- **1 KiB (kibibyte) = 1024 bytes**
- **1 KB (kilobyte) = 1000 bytes (SI definition) or 1024 bytes (binary) .**
- **1 kbit (kilobit) = 1000 bits**

Let's address this madness:

⚠️ 2. Binary System (What Your Computer Actually Uses)

💭 "So wait… 1KB is 1000 bytes? Or 1024? Or 8192 in Mars?"
Yep. Welcome to the madness of digital units.

Let's break this thing down like you're hearing it for the first time — because most people only pretend they understand this.

📌 Binary: The Language of Computers

At the hardware level, everything is just **ON (1)** or **OFF (0)** — two voltage states. That's why computers use **binary (base-2)** instead of human-friendly **decimal (base-10)**.

Binary example:

```
Decimal  :  0 1 2 3 4 5 6 7 8 9 10
Binary   :  0 1 10 11 100 101 110 111 1000 1001 1010
```

See that? It gets long fast. But it's perfect for computers because flipping switches (on/off) is fast, cheap, and reliable.

📦 Units in Binary World (the OG Nerd Units)

Now, let's talk *storage units* — this is where the confusion starts:

Name	Meaning	Value in Bytes	Based On
Bit (b)	Smallest unit (0 or 1)	N/A	Binary (0 or 1)
Byte (B)	8 bits	8	Used to store a character
Kilobyte (KB)	1000 bytes	1,000	**SI Decimal** (used by hard drive companies)
Kibibyte (KiB)	1024 bytes	1,024	**Binary** (used by RAM, OS)
Kilobit (kb)	1000 bits	1000 bits / 125 bytes	Used in internet speeds

💡 Important:

- **1 KB** = 1 Kilobyte ✅ = 1000 bytes
- **1 KiB** = 1 Kibibyte ✅ = 1024 bytes
- **1 kbit** = 1 Kilobit ✅ = 1000 bits
- **1 byte** = 8 bits ✅
- **1 kilobyte (KB)** = 1000 bytes (Decimal, SI standard) ✅
- **1 kibibyte (KiB)** = 1024 bytes (Binary, OS standard) ✅
- **1 Kilobit (kbit)** = 1000 bits (used in networking) ✅

Internet speeds are shown in **kilobits per second (kbps)**, not kilobytes — that's why 10 Mbps WiFi doesn't *feel* that fast.

🧠 Why Binary Sizes Even Exist (And Who Uses What)

📓 Binary Sizes (KiB, MiB, GiB, etc.)

These units are used in computing (especially in operating systems and hardware) and are based on powers of 2:

- **1 KiB (Kibibyte) = 1024 bytes**
  It's 2^{10} (which equals 1024).
- **1 MiB (Mebibyte) = 1024 KiB (Kibibytes) = 1,048,576 bytes**
  This is equal to 2^{20} bytes.
- **1 GiB = 1024 MiB = 1,073,741,824 bytes**
- **1 TiB = 1024 GiB = 1,099,511,627,776 bytes**

Each time, you multiply by 1024, which follows the binary system (base 2).

"Megabyte" and "kilobyte" are *officially* based on the SI standard.

But... when you're working in operating systems, file systems, and even low-level stuff like assembly, those units often use powers of 2.

This is where mebibyte (MiB), kibibyte (KiB), etc., come in.

So, yeah, everyone says "megabyte" (MB), "kilobyte" (KB), and so on, even though technically they often mean **MiB** or **KiB** in many contexts.

🏁 Used in:

- RAM/Memory.
- Operating Systems (Windows File Explorer, Linux ls, etc.)
- CPU-level code.
- Embedded systems, firmware.

When you see:

- 4 GiB RAM — that's **$4 \times 2^{30} = 4{,}294{,}967{,}296$ bytes**.

📄 Decimal Sizes (KB, MB, GB)

Based on **powers of 10**:

1 KB = 10^3 = 1000
1 MB = 10^6 = 1,000,000

Used in:

- Hard drive & SSD marketing (they'll say "500GB" but that's 500 × 10^9 bytes, not GiB)
- Internet Service Providers
- USB packaging
- SD card labels

📉 So your 1TB hard drive is *not* 1TB in Windows. It shows around **931GiB**. Why? Because **marketing uses decimal**, but **Windows shows binary**.

🎯 Real World Example: Where You'll Meet These

Situation	Unit You'll See	What It *Actually* Means
RAM (8GB)	GiB (binary)	8×2^{30} = 8,589,934,592 bytes
File sizes in Explorer	KiB, MiB (but written as KB, MB)	1 MB = 1,048,576 bytes
Internet speed (e.g., 20Mbps)	**Megabits per second** (Mb)	20 × 1,000,000 bits per sec
SSD packaging (e.g., 512GB)	**GB (decimal)**	512 × 1,000,000,000 bytes
Download size in browsers	Usually MB (binary-ish)	Might be 1,048,576 bytes per MB

🧪 Quick Quiz to Test You (Mentally)

Q1: If a file is 10 MiB, how many bytes is that?

10×2^{20} = 10,485,760 bytes

Q2: Your internet is 100 Mbps. How many megabytes per second can you download?

100 / 8 = 12.5 MBps
(Because 8 bits = 1 byte)

Term	Means	Where You'll See It
Bit (b)	0 or 1	Networking, logic
Byte (B)	8 bits	Files, RAM, OS
KB	1000 bytes	Internet, SSDs, USBs
KiB	1024 bytes	RAM, OS internals
kbit	1000 bits	Internet speeds

🧠 INTEGER STORAGE: HOW THE CPU SEES NUMBERS

The most **fundamental storage unit** in any modern computer (including x86 architecture) is:

⚙️ 1 byte = 8 bits

But that's just the start. Larger integer sizes are built by combining more bytes:

NAME	SIZE (BITS)	SIZE (BYTES)	COMMON ASSEMBLY KEYWORD
Byte	8	1	BYTE
Word	16	2	WORD
Doubleword (Dword)	32	4	DWORD
Quadword (Qword)	64	8	QWORD

📊 UNSIGNED INTEGER STORAGE TABLE (RAW BINARY)

When we talk **unsigned integers**, we're only representing **positive values**, including zero. This means the minimum is always 0, and the maximum depends entirely on how many bits are used.

DATA TYPE	BITS	BYTES	MIN VALUE	MAX VALUE (DECIMAL)	MAX VALUE (HEX)
Byte	8	1	0	255	0xFF
Word	16	2	0	65,535	0xFFFF
Doubleword (Dword)	32	4	0	4,294,967,295	0xFFFFFFFF
Quadword (Qword)	64	8	0	18,446,744,073,709,551,615	0xFFFFFFFFFFFFFFFF

These are the **raw** binary interpretations, not tied to a programming language (yet). Just what the CPU or memory sees.

⚠️ SIGNED vs. UNSIGNED: THE TWIST

When you introduce *signed* integers (which include negative numbers), the bit layout changes — usually the **most significant bit (MSB)** is used to indicate sign:

TYPE	FORMULA FOR MAX VALUE	EXAMPLE (32-BIT)
Unsigned	$2^n - 1$	$2^{32} - 1 = 4,294,967,295$
Signed	-2^{n-1} to $2^{n-1} - 1$	$-2,147,483,648$ to $2,147,483,647$

So, for signed 32-bit (int):

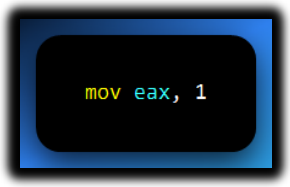
- MSB = 1 → negative number
- MSB = 0 → positive number

In **two's complement** format (what modern CPUs use), this makes arithmetic way easier for the hardware.

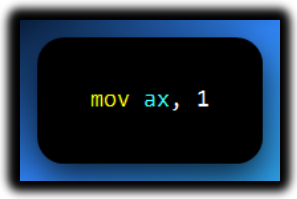
🧱 WHAT THIS MEANS IN PRACTICE (x86 & C/ASM)

✅ Assembly / WinAPI / x86 Systems

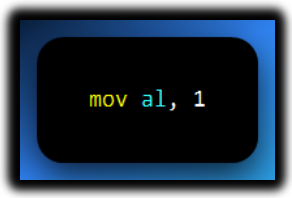
mov eax, 1 → eax is a **32-bit register** (DWORD)



mov ax, 1 → ax is the **16-bit word** part of eax. You're only manipulating the **lower half** of it.



mov al, 1 → al is the **8-bit low byte** of ax.



C TYPE	SIGNED?	BITS	BYTES	COMMON ON X86	NOTES
`unsigned char`	No	8	1	Yes	Maps directly to a single byte in memory. Used for small non-negative numbers or raw byte data.
`unsigned short`	No	16	2	Yes	Maps directly to a word in memory. Used for larger non-negative numbers than a byte.
`unsigned int`	No	32	4	Yes	Maps directly to a doubleword (dword) in memory. This is a very common size for general-purpose integers.
`unsigned long long`	No	64	8	Yes (x86_64)	Maps directly to a quadword (qword) in memory. Essential for 64-bit programming and handling very large non-negative numbers.
`int`	Yes	32	4	Yes	Signed by default. Maps to a doubleword (dword). This is the most common integer type for general-purpose signed numbers.

Where you'll mostly see these being used:

USE CASE	DATA TYPE COMMONLY USED	EXAMPLE
File size, disk size	`unsigned DWORD` (for 32-bit systems) or `QWORD` (for 64-bit systems)	Windows API function `GetFileSizeEx()` uses a `LARGE_INTEGER` structure, which is essentially a 64-bit value (like a `QWORD`) to accurately represent very large file sizes.
Pointers on 64-bit systems	`QWORD` (8 bytes)	A memory address like `0x7FFFAABBCCDD1122` is a 64-bit value, fitting perfectly into a `QWORD`. This allows 64-bit systems to access vast amounts of memory.
Buffers, loops for reading arrays	`DWORD` or `INT`	When you're iterating through an array or managing the size of a data buffer, 32-bit integers (`DWORD` or `int` in C) are commonly used for indexing and counting elements.
WinAPI handles	`DWORD`, `WORD`, or specific handle types	Windows API (WinAPI) functions often return or expect handles (unique identifiers for resources). Examples include file handles, process IDs, thread IDs, or window handles, which are typically 32-bit values (like a `DWORD`) or sometimes 16-bit (`WORD`) in older contexts.
System call arguments	Often `DWORD` or `QWORD`	When your program asks the operating system to do something (a "system call"), the arguments passed to the OS function are typically placed in specific registers (like `EAX/RAX`, `ECX/RCX`, etc.) and are sized according to the system's architecture (32-bit `DWORDs` or 64-bit `QWORDs`).

🧠 TLDR - INTEGER STORAGE DECODED

- 🧱 **Byte** = 8 bits = max 255 (unsigned)
- 🔧 **Word** = 16 bits = max 65,535
- 🧠 **Dword** = 32 bits = max ~4.2 billion
- 🧠 **Qword** = 64 bits = max ~18 quintillion
- ✅ **Unsigned** = only positive
- ⚠️ **Signed** = supports negatives, via two's complement
- 🔍 In **WinAPI** and **x86**, these terms show up *everywhere*