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

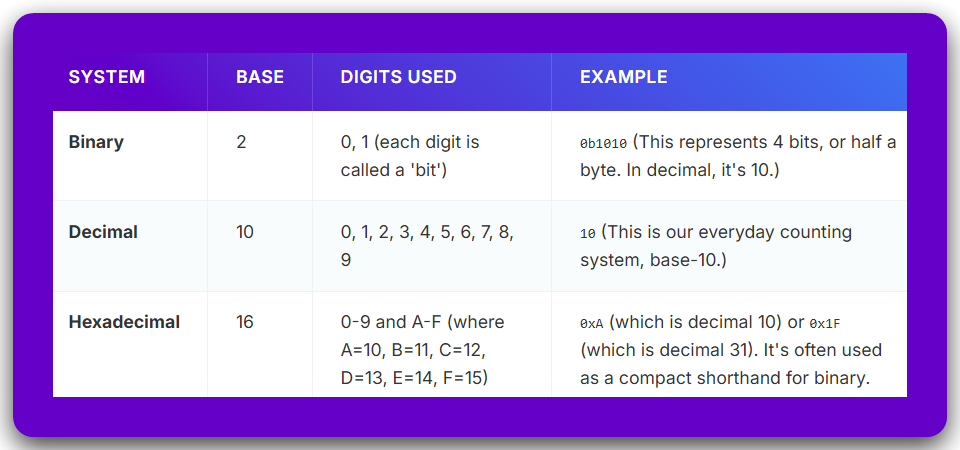
So, if you can’t **mentally switch** between 0b1010, 10, and 0xA… you’re gonna have a bad time.

**⚙️** 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.



**👁️** 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.

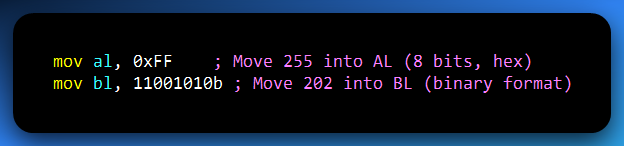
🔥 **Skill check:**What’s 0x3C in decimal?  
What’s 11001100 in hex?

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:



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:



**💡** 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!*

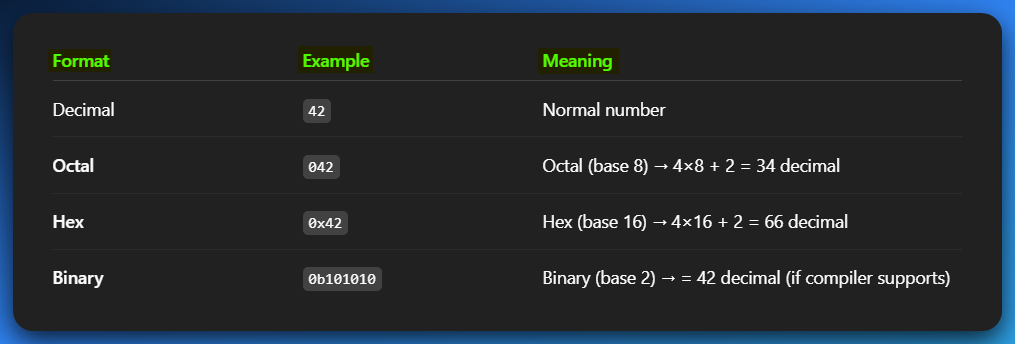
👉 *We’ll learn how to calculate them ahead.*

⚠️ **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)



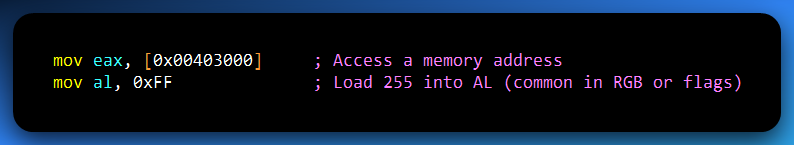
**✅** Bottom Line:

* 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



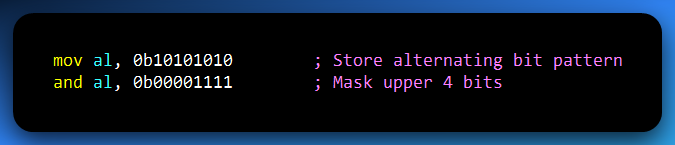


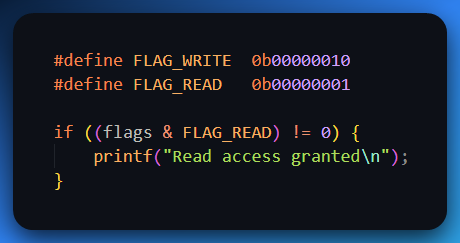
**✅ When to use:**

* 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



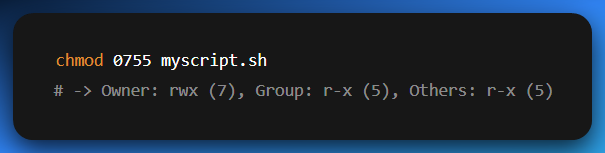


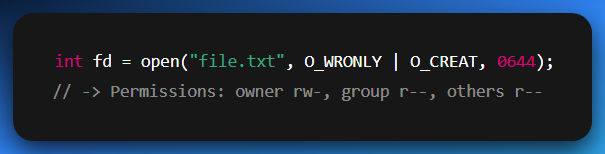
**✅ When to use:**

* 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)

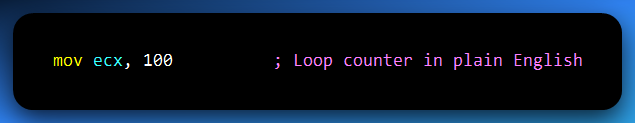


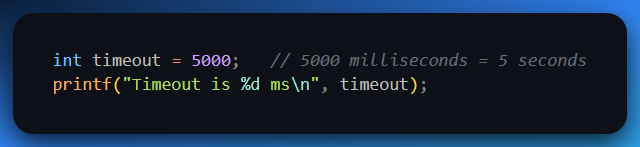


**⚠️** Avoid in most modern code unless you’re on Unix and know what you're doing.

**🟢** 4. DECIMAL – Human readable, boring but necessary

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

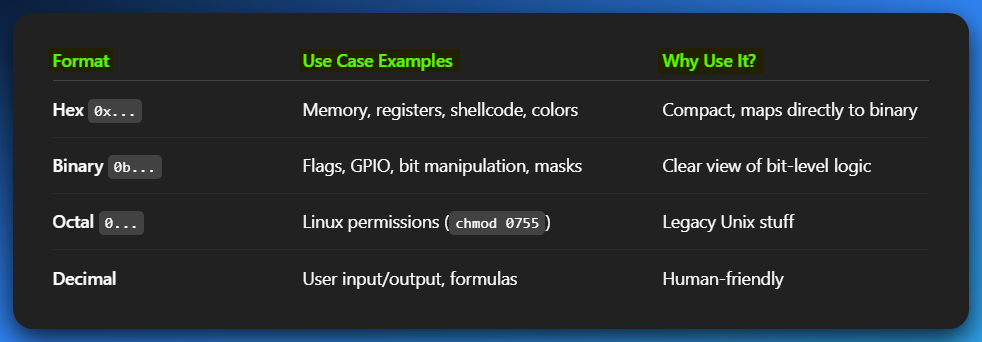




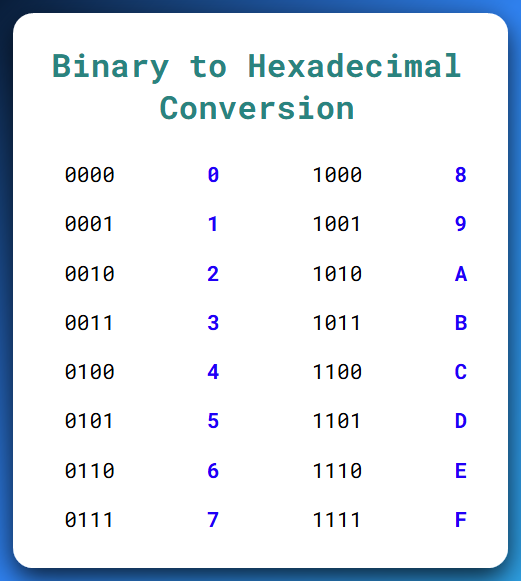
**✅ When to use:**

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

**⚡** TLDR – When to Use What?



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**:

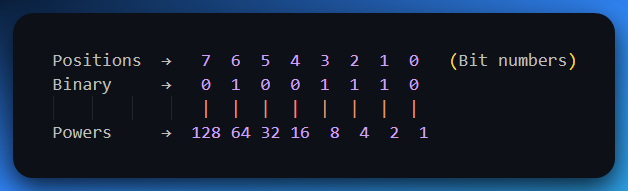
* **1 = ON (Electricity flowing)**
* **0 = OFF (No electricity)**

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 **20** to **28**

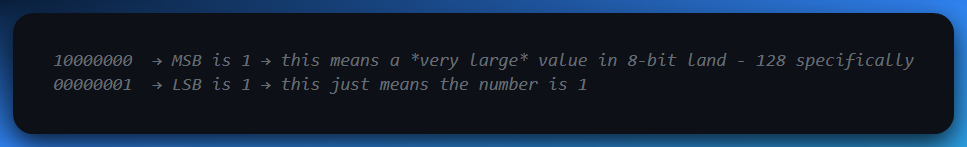
Now we add up the values that align with the 1’s that is, **2+4+8+64 = 78**

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

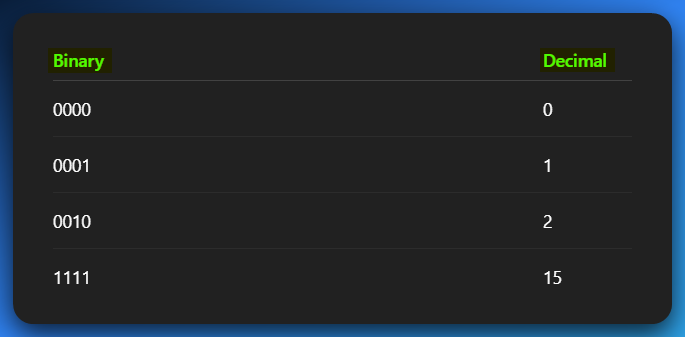
**📌 What it means:**

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.

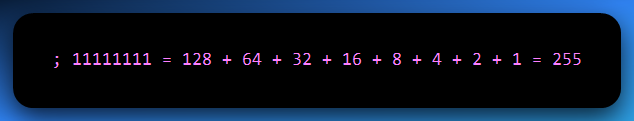
**✅ Example (4-bit unsigned binary):**



**✅ 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:



There are **2⁸ = 256** total values, ranging from 0 to 255 (0 inclusive).

**🧠 Why there’s “No tricks” with unsigned binary integers?**

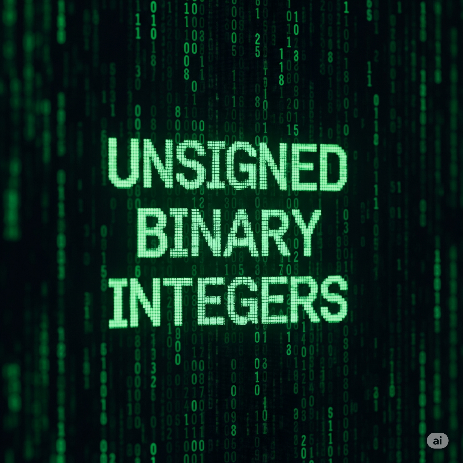
Because you’re not using any encoding scheme (like Two’s complement for negative numbers).

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.

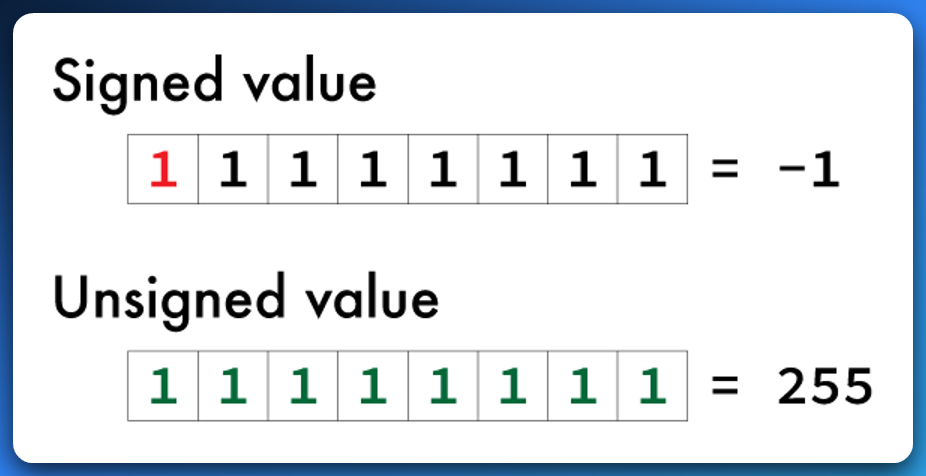
**🔄 How it works:**

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.

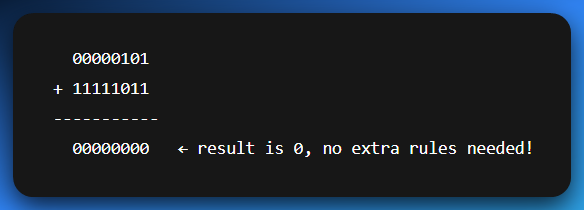
🧪 How does it work?

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:



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

**🤔** How can 11111111 be 255 and not called signed int?

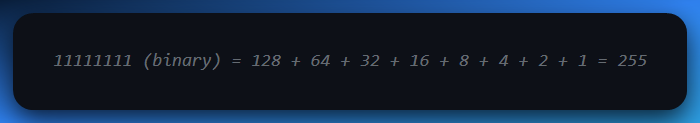
It can be called a signed, but it comes down to **context** — whether the number is treated as **unsigned** or **signed**. *Read the last paragraph to get the quick context.*

📌 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 ➡️ 28 -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:

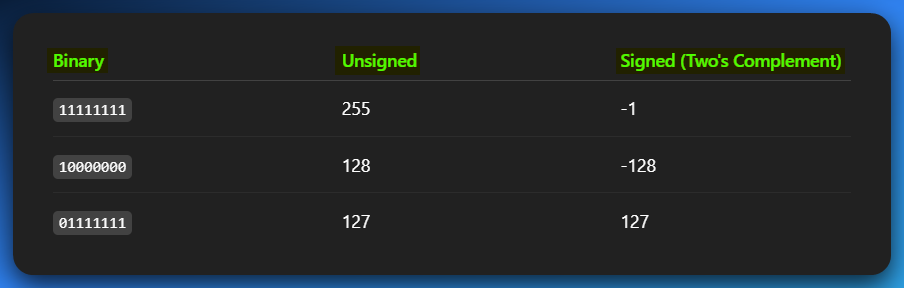
1. Start with 11111111
2. Invert the bits → 00000000
3. Add 1 → 00000001

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:



🧠 **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.

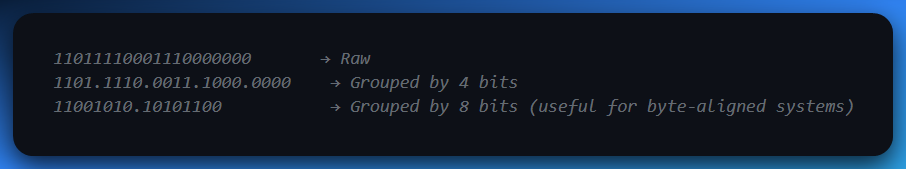
But if we interpret the same 8 bits using **Two’s complement**, then the most significant bit (MSB) is treated as a **sign bit**, and 11111111 represents **–1** instead. So, the **same binary pattern** can mean **different values**, depending on whether it’s being used in a signed or unsigned system.

**📎** 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?

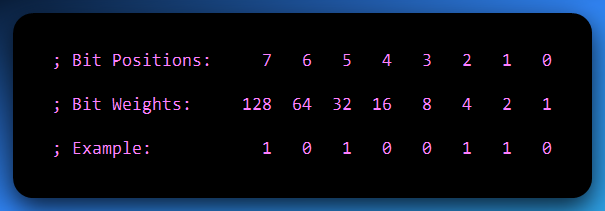
We **break them into groups** — usually every **4 bits** or **8 bits**, just like how we write big decimal numbers with commas or spaces e.g.



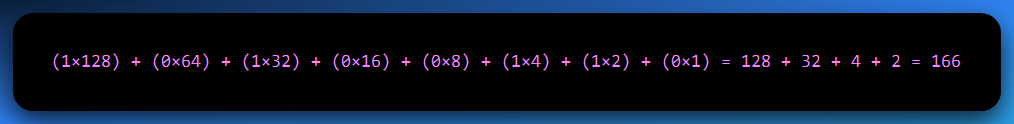
This doesn’t change the value — it’s just **formatting to help our human brains**.

**🔢** Unsigned Binary: Bit by Bit

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



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



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

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

**Nope.**

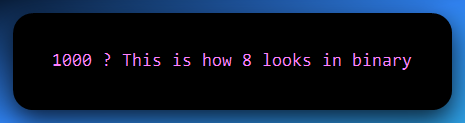
Let’s break it down:

⚙️ With 3 bits:

* You get **2³ = 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**.

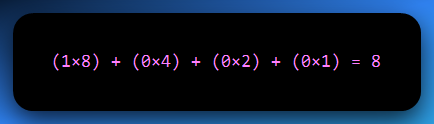
✅ To store the number 8, you need 4 bits



**🔍 Why?**

Because:

* The leftmost bit is in the **2³ position**, which equals 8.
* The rest are 0s:



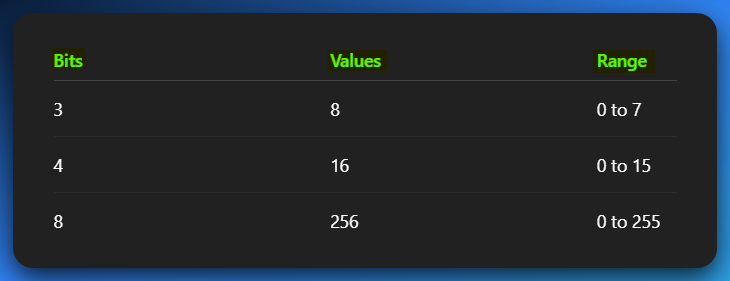
So, if you tried to cram 8 into 3 bits, it would **overflow** — you simply don’t have enough bits to hold the value.

🧠 Key Idea:

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

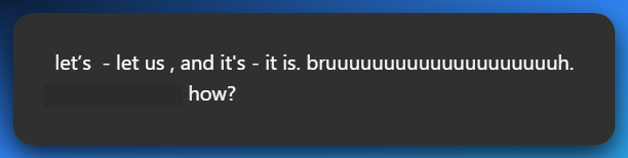
… but the **maximum value** is **2ⁿ - 1.**

So, 8 is past 23 – 1 range, thus we need 4 bits.



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*.

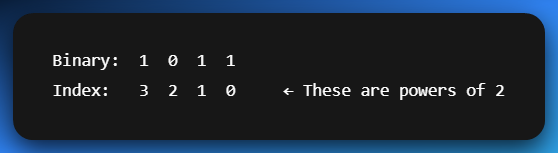
❌ Don’t blindly swap them — **connotation matters**, not just grammar.

🔁 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)

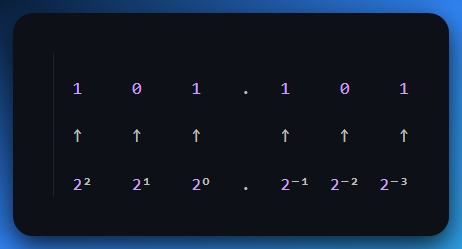
Binary fractions work *just like* whole numbers, except instead of **powers of 2 going up**, we go **down** (negative exponents) *after* the decimal point.

**Example: 101.101**

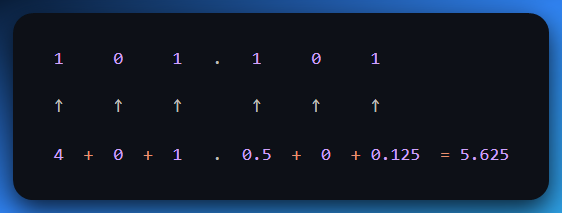
Break it into two parts:

* Whole part: 101 → same rules as above = 5
* Fractional part: .101
  + 1 × 2⁻¹ = 0.5
  + 0 × 2⁻² = 0
  + 1 × 2⁻³ = 0.125

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



Add it all up:

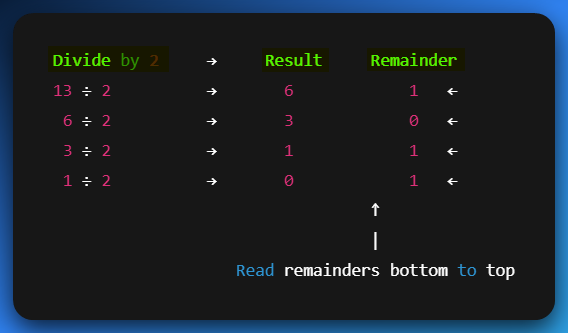


✅ 101.101 in binary = **5.625 in decimal**

**🔄** 3. DECIMAL TO BINARY (Whole Numbers)

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

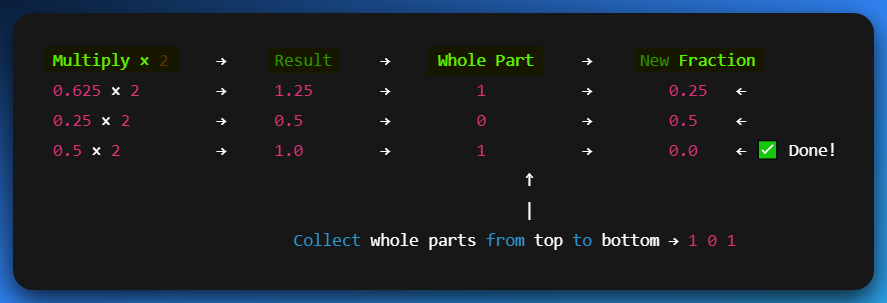
**Example:** Convert 13 to binary ✅ Final Binary (13 in base 10): **1101**.



**🔬** 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

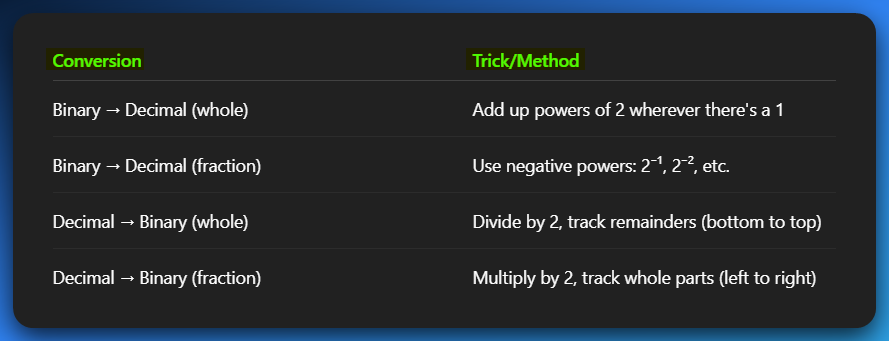


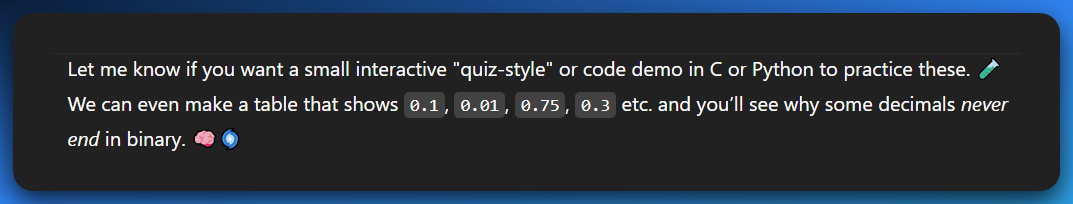
💡 Let's Do a Full Example:

**Convert 13.625 to Binary:**

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

Final binary:  
✅ 13.625 = **1101.101**





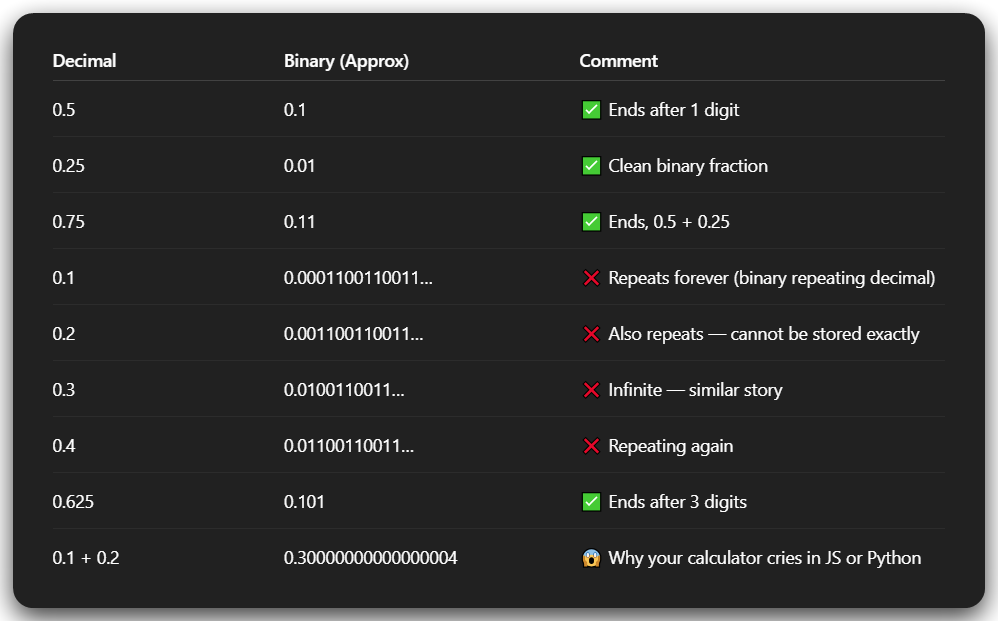
Do questions as programming practice, yourself.

**⚠️** 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



💡 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:



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

🎯 Final Brain Drop

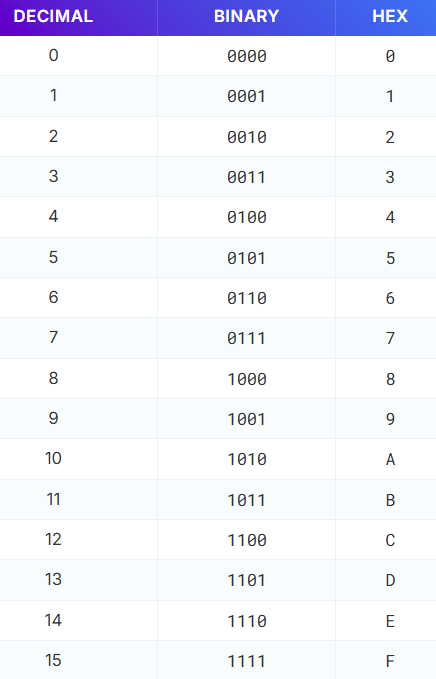
So anytime you wonder, “Why the heck is my float inaccurate?” — remember this:

*"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):



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

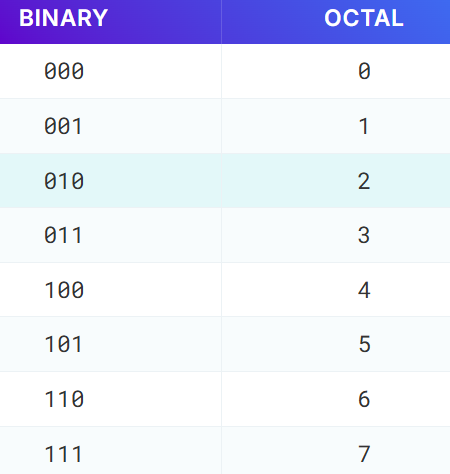
To draw the full binary-to-hex table fast, follow this simple visual trick:

* 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.



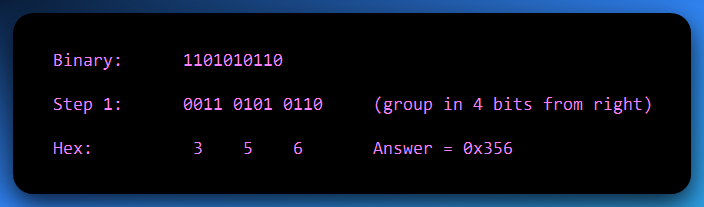
🧮 3. How to Quickly Build the Binary-to-Octal Table (Bit Patterns)

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

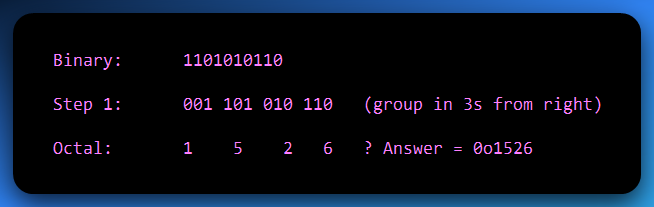
* In the **first column**, alternate every 4 rows: 4 zeros, 4 ones, 4 zeros, 4 ones.
* In the **second column**, alternate every 2 rows: 2 zeros, 2 ones, 2 zeros, 2 ones.
* In the **third column**, simply repeat 01010101 downward.

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.