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

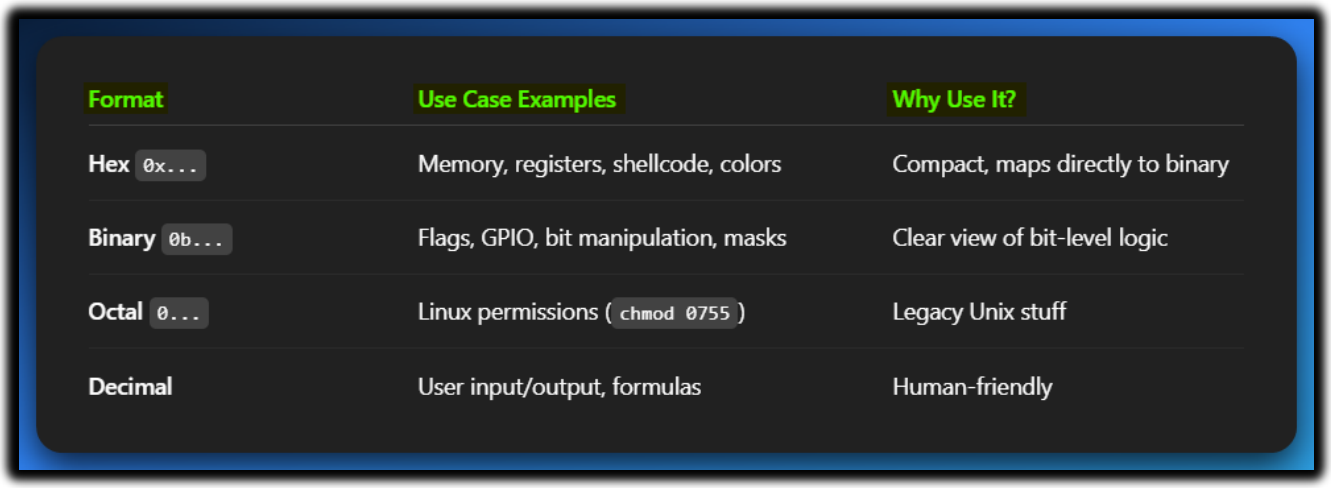
```
mov ecx, 100          ; Loop counter in plain English
```

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

✅ When to use:

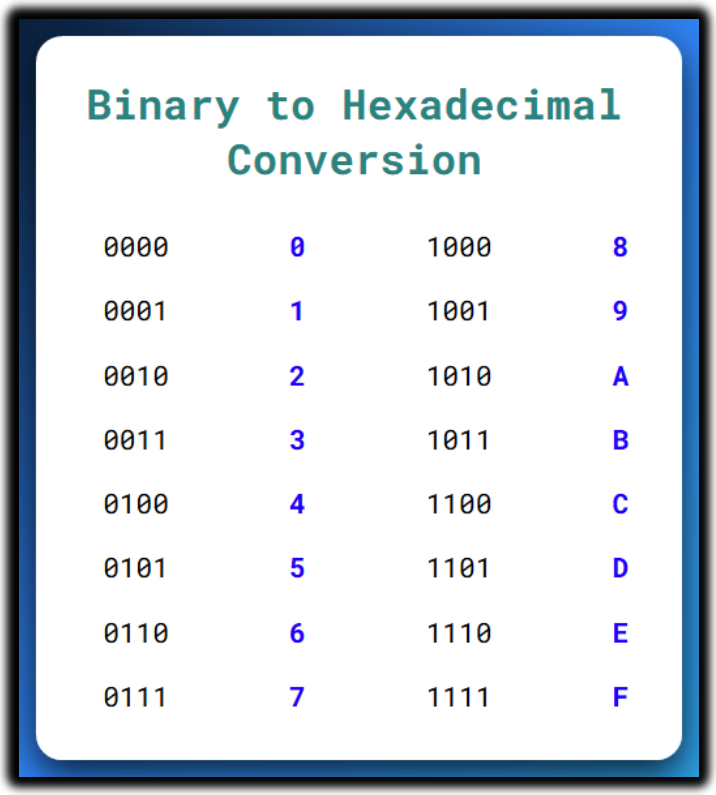
- User-facing numbers
- Calculations or percentages
- Output/logs

⚡ 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**:

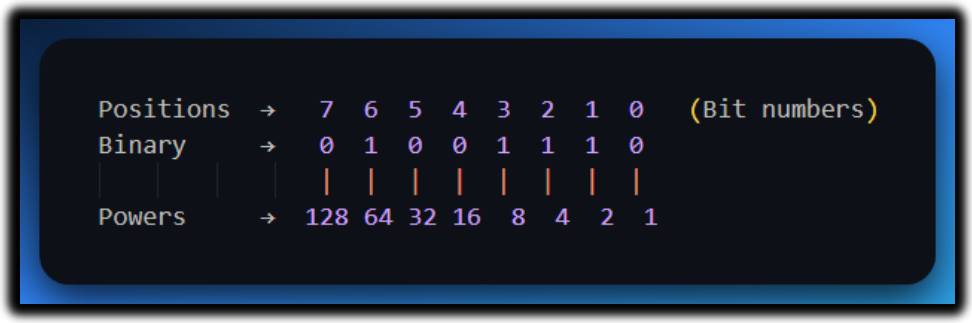
- **1 = ON (Electricity flowing)**
- **0 = OFF (No electricity)**

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

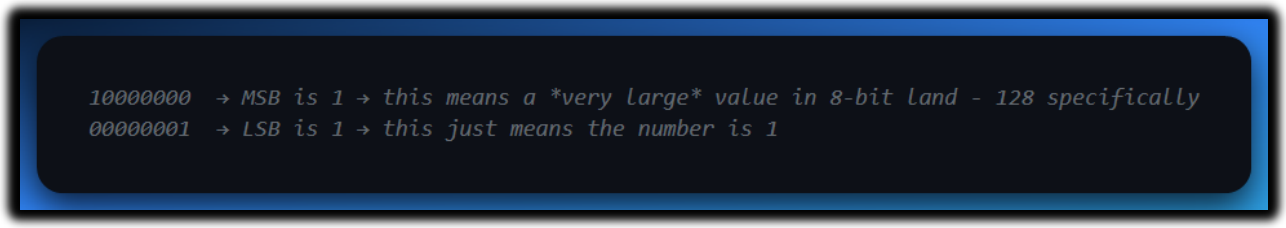
Now we add up the values that align with the 1's that is, **2+4+8+64 = 78**

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.

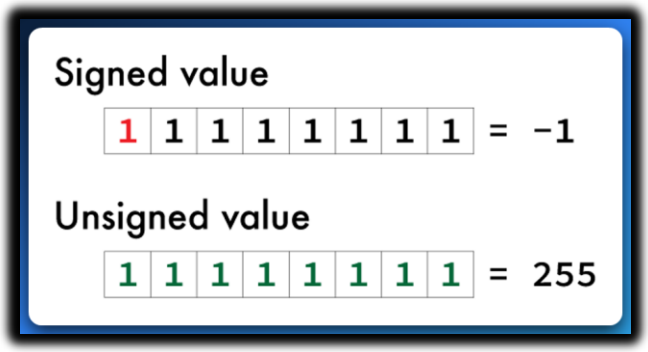
🔁 How it works:

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?

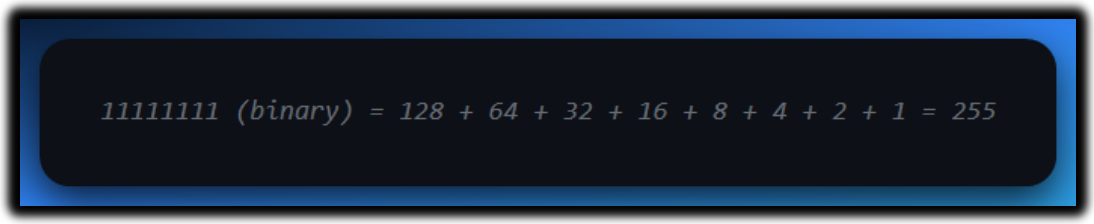
It can be called a signed, but it comes down to **context** — whether the number is treated as **unsigned** or **signed**. *Read the last paragraph to get the quick context.*

📌 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.

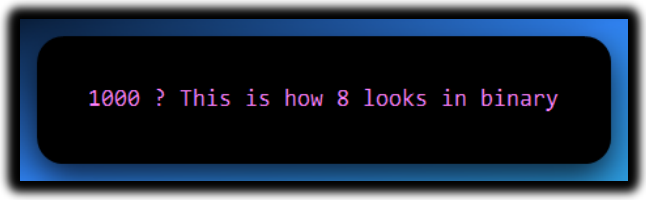
Let's break it down:

⚙️ 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**.

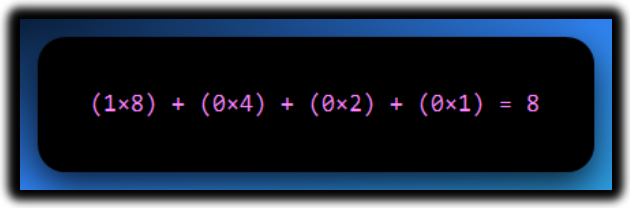
✅ To store the number 8, you need 4 bits



🔍 Why?

Because:

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



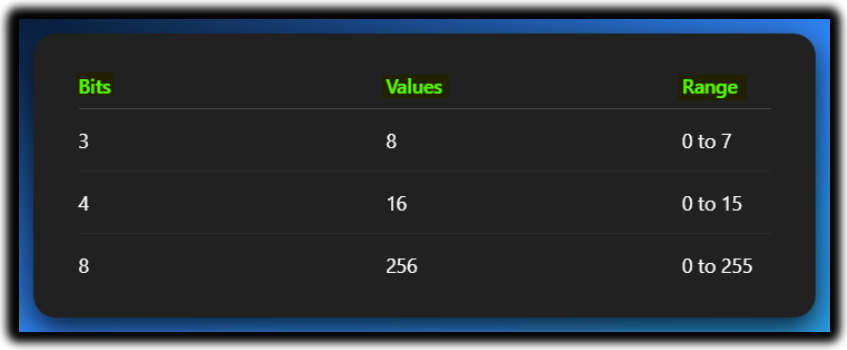
So, if you tried to cram 8 into 3 bits, it would **overflow** — you simply don't have enough bits to hold the value.

🧠 Key Idea:

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

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

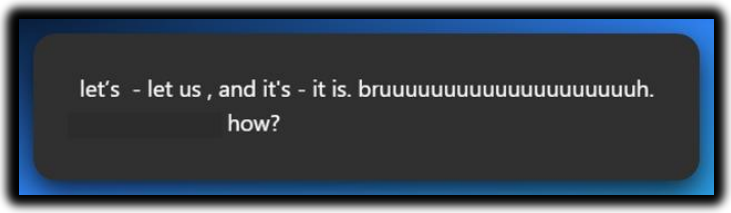
So, 8 is past $2^3 - 1$ range, thus we need 4 bits.



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*.

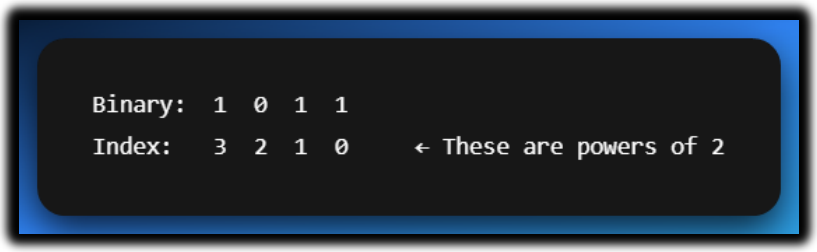
❌ Don’t blindly swap them — **connotation matters**, not just grammar.

🔁 1. BINARY TO DECIMAL (Whole Numbers)

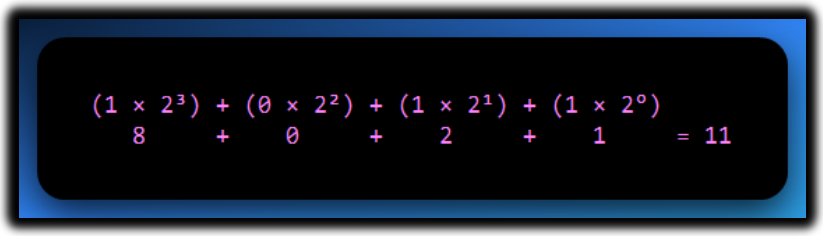
How It Works:

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)

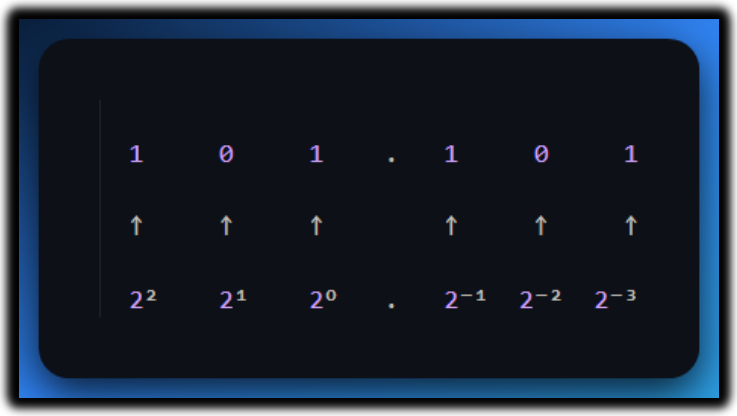
Binary fractions work *just like* whole numbers, except instead of **powers of 2 going up**, we go **down** (negative exponents) *after* the decimal point.

Example: 101.101

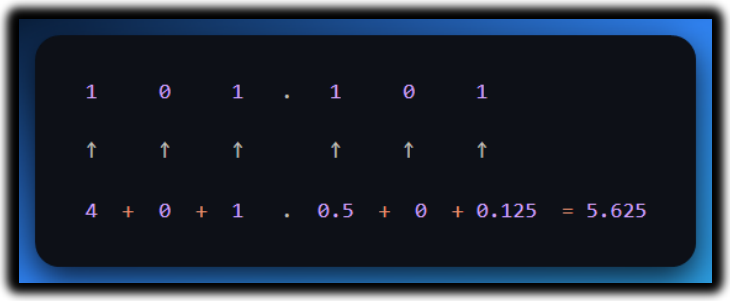
Break it into two parts:

- 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:

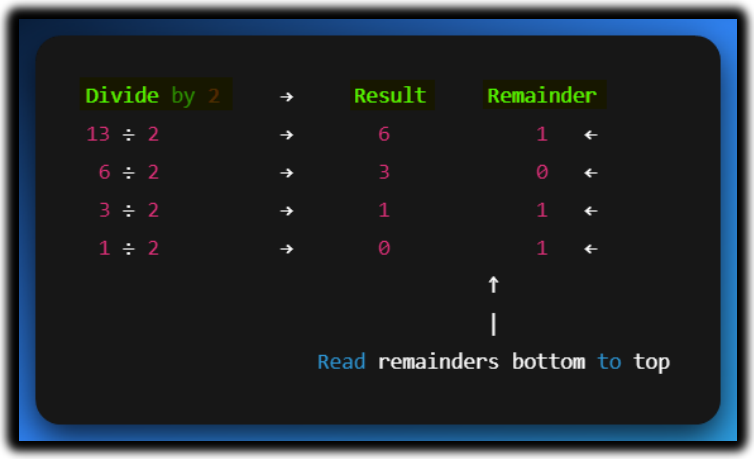


✅ 101.101 in binary = **5.625 in decimal**

🔁 3. DECIMAL TO BINARY (Whole Numbers)

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

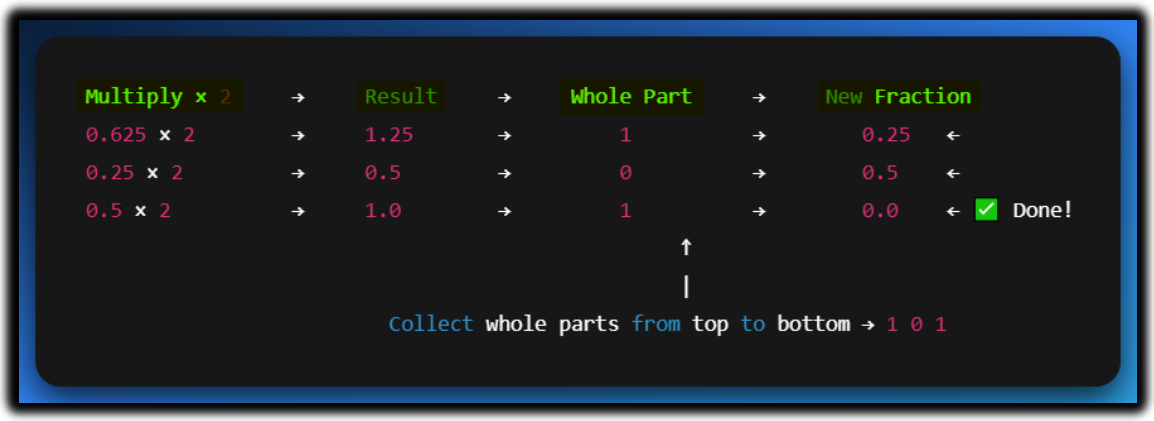
Example: Convert 13 to binary ✅ Final Binary (13 in base 10): **1101**.



🔬 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



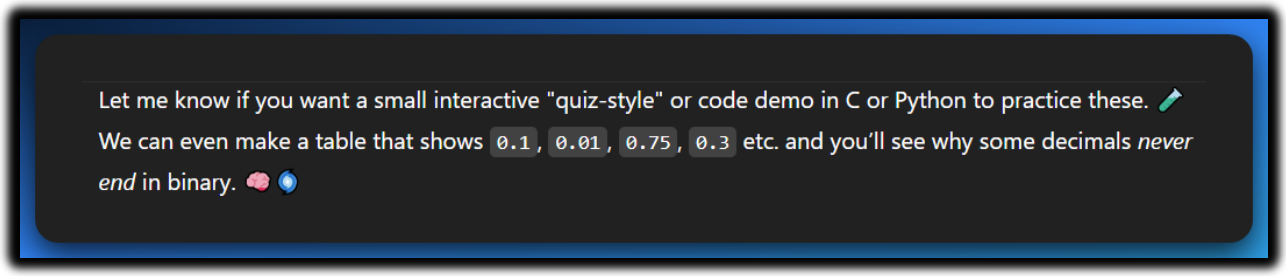
💡 Let's Do a Full Example:

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

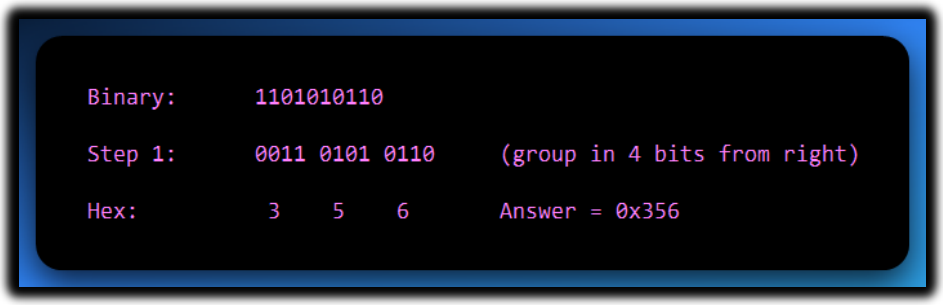
🧮 3. How to Quickly Build the Binary-to-Octal Table (Bit Patterns)

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

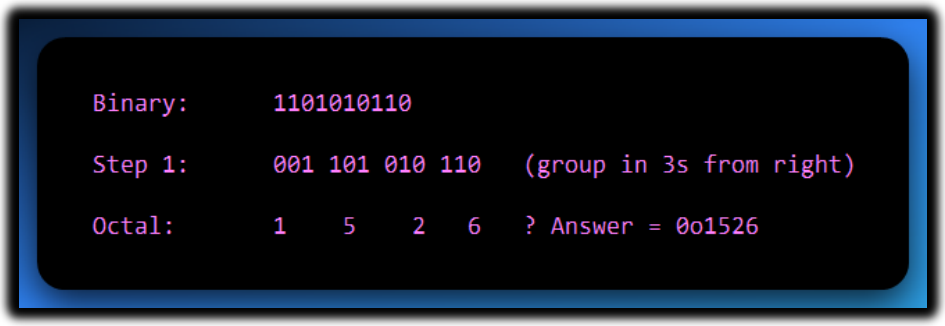
- In the **first column**, alternate every 4 rows: 4 zeros, 4 ones, 4 zeros, 4 ones.
- In the **second column**, alternate every 2 rows: 2 zeros, 2 ones, 2 zeros, 2 ones.
- In the **third column**, simply repeat 01010101 downward.

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):

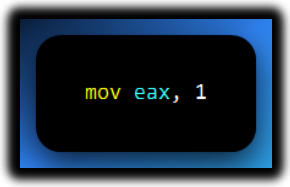
- MSB = 1 → negative number
- MSB = 0 → positive number

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

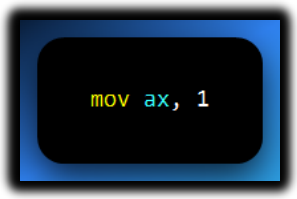
🧱 WHAT THIS MEANS IN PRACTICE (x86 & C/ASM)

✅ Assembly / WinAPI / x86 Systems

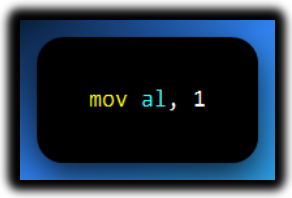
mov eax, 1 → eax is a **32-bit register** (DWORD)



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 **(Zoom PDF for clarity):**

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*

NB from the last topic:

✅ **KB, MB, GB, TB** = marketing numbers (1KB = 1,000 bytes)

💻 **KiB, MiB, GiB, TiB** = actual memory sizes in your system (1KiB = 1,024 bytes)

🧠 Be aware of both — especially if you're doing systems programming, OS work, or comparing file sizes accurately.

🧠 WHAT'S THE DEAL WITH "SIGN EXTENSION" AND "ZERO EXTENSION"?

Imagine you're taking a number written in a smaller box (like 8 bits) and putting it into a **bigger box** (like 16 or 32 bits). You can't just copy-paste the bits — the computer needs to **preserve the meaning** (especially the sign if it's signed).

🧿 Zero Extension (Used for Unsigned numbers)

What it does:
Just adds 0s to the **left** (the higher bits).

Why?
Unsigned numbers don't care about sign. So, filling with zeroes is always safe.

Example (8-bit to 16-bit):

```
mov al, 10001100b     ; al = 0x8C = 140 unsigned
movzx ax, al          ; Zero-extend to 16-bit : ax = 0x008C
```

```
; Results

; Original (8-bit):       10001100
; Zero-extended (16-bit): 00000000 10001100
```

See that? We kept the value **exactly the same** but just padded it with zeroes on the left.

🧭 Sign Extension (Used for Signed numbers — 2's complement)

What it does:

Copies the **sign bit** (leftmost bit) to the new bits on the left.

Why?

To **preserve the sign** and value of the number in 2's complement.

Example (8-bit to 16-bit):

```
mov al, 11111111b    ; al = -1 in 2's complement
movsx ax, al         ; Sign-extend to 16-bit : ax = 0xFFFF
```

```
; Original (8-bit):        11111111 (-1)
; Sign-extended (16-bit):  11111111 11111111
```

See how it **copied the sign bit (1)**? That's how −1 stays −1 even after "growing".

📍 Key Differences: Zero Extension vs. Sign Extension

FEATURE	ZERO EXTENSION	SIGN EXTENSION
Used for	Unsigned values (numbers that are always positive or zero).	Signed values (numbers that can be positive, negative, or zero), specifically using Two's Complement representation.
What's added	When a smaller value is moved to a larger register, the new, higher-order bits are filled with **zeros (0s)**. This preserves the original non-negative value.	When a smaller signed value is moved to a larger register, the new, higher-order bits are filled with **copies of the MSB (Most Significant Bit)** of the original value. This preserves the original signed value (positive or negative).
x86 Instruction	`MOVZX` (Move with Zero-Extend)	`MOVSX` (Move with Sign-Extend)

💡 Real-Life Analogy

Imagine someone is writing a note on a small piece of paper (8-bit value), and now they need to stick it on a large poster (16-bit register).

- If it's a **positive message (unsigned),** they just tape it on and fill the background with blank space (zero extension).
- If it's a **negative message (signed),** they fill the background with angry red ink (1s) to **keep the same emotional vibe** across the bigger paper (sign extension).

💉 Common Use Cases in Assembly

Reading bytes from memory and processing them as 32-bit values:

```
movsx eax, byte ptr [value]   ; read signed byte → sign-extend to 32-bit
movzx eax, byte ptr [value]   ; read unsigned byte → zero-extend
```

Before arithmetic: Some arithmetic instructions (like IMUL, IDIV) require 16 or 32-bit operands, so you must **extend** your smaller value first — **correctly**, based on its sign.

⚠️ Important Reminder

On x86:

- If you use MOV directly, no extension happens. Just copy. If you move al into ax, it won't zero-extend!
- Use MOVZX (zero extension) or MOVSX (sign extension) **explicitly**.

✅ TLDR

- 🔹 **Use MOVZX for unsigned values** — it adds zeroes.
- 🔹 **Use MOVSX for signed values** — it replicates the sign bit.
- 💥 Crucial when extending from 8-bit to 16-bit or 16-bit to 32-bit.
- 💀 If you mess this up? Your arithmetic gets WILDLY wrong, especially with negatives.

🧮 TWO'S COMPLEMENT REPRESENTATION pt 2

How your CPU thinks about **negative numbers** — and still sleeps well at night.

🧠 Why Do We Need It?

Computers store everything as **0s and 1s** — and binary is naturally positive.
So… how do you **store negative integers**? 🤔

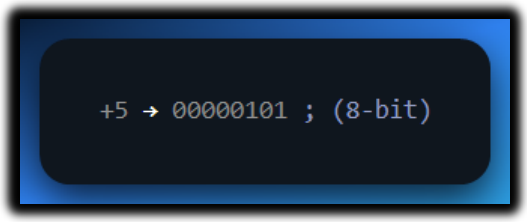
- Option 1: use a "sign bit" — not efficient.
- Option 2: use **two's complement**, the industry standard.

Two's complement is the system used in almost all CPUs today (x86, ARM, RISC-V, etc.) to store **signed integers** — i.e., both positive and negative values — **using only binary math**.

✅ THE RULES (in human words)

📈 For positive numbers:

Just write the binary as usual.
Example:



📉 For negative numbers:

Use **two's complement** steps:

⚙️ Step-by-step for -5:

1. Start with the binary of +5:
   00000101
2. Invert the bits:
   11111010
3. Add 1:
   11111011

✅ So, the two's complement of -5 is: **11111011**

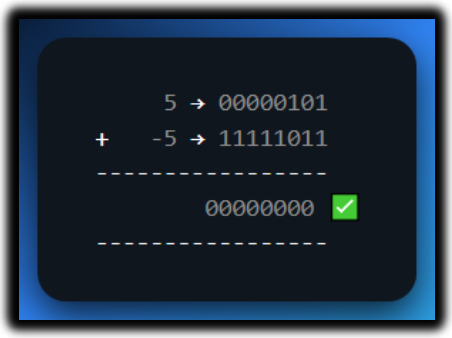
🧠 Bonus: the **leading 1** shows it's negative. Always.

💥 WHY TWO'S COMPLEMENT IS AWESOME

You can add and subtract negative numbers **just like positives**.

The CPU **doesn't need separate logic** for subtraction.

It simplifies circuitry and is way faster. Example:



We'd done this as the former example.

🔢 VALUE RANGE PER SIZE (Signed Integers)

BITS	RANGE (SIGNED, TWO'S COMPLEMENT)
8	-128 to +127
16	-32,768 to +32,767
32	-2,147,483,648 to +2,147,483,647
64	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Why is the **minimum value bigger** than the maximum?
Because zero uses up a positive slot (e.g., 00000000 = 0).

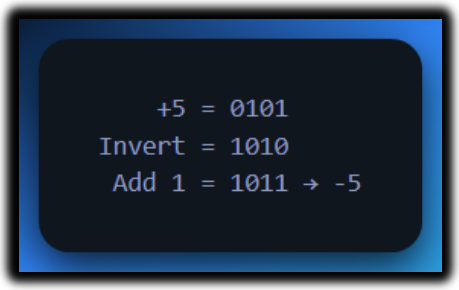
✅ 1. Bit-width Importance (Overflow Awareness)

We briefly mentioned **"make sure both have the same number of bits",** but we need to **hammer that home**.

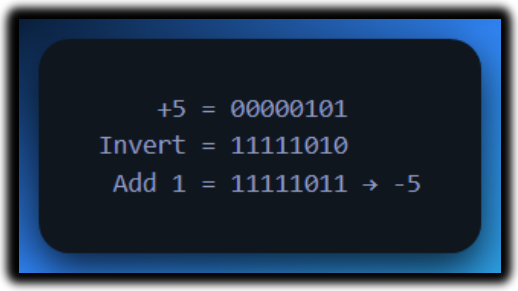
✍️ If you're working with 8-bit, 16-bit, 32-bit systems, the two's complement **depends** on that width.

Example: Represent -5 in 4 bits vs 8 bits

4-bit:



8-bit:



⚠️ If you try to add or subtract outside the allowed bit-width, you'll get **overflow**.

✅ 2. Two's Complement Range and Why MSB = Negative

This is **often misunderstood** by beginners:

For n bits:

- Range of **signed integers (two's complement)** = -2^{n-1} to $(2^{n-1}) - 1$ as we've shown in the table above.

Examples:

- 4-bit: -8 to +7
- 8-bit: -128 to +127

Why MSB = 1 Means Negative:

Here's where it gets interesting: the **MSB** (most significant bit) is often called the **sign bit** in two's complement. It's **the leftmost bit in a binary number**, and it **indicates the sign** of the number.

- **If the MSB is 0**, the number is **positive** or zero.
- **If the MSB is 1**, the number is **negative**.

The **reason the MSB represents a negative number** is because it is weighted as a **negative power of 2**. This is what makes two's complement so powerful for representing negative numbers.

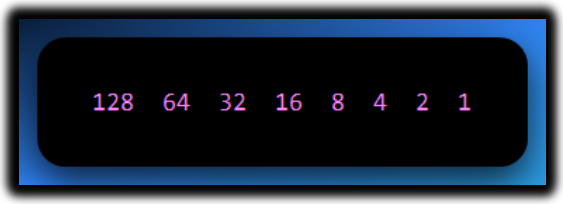
How the MSB Works:

Let's look at an **8-bit** example: 10000000.

In decimal, we calculate the value of each bit in the 8-bit binary number:

Starting from the rightmost bit, we assign powers of 2 to each position:
$2^0, 2^1, 2^2$ ...

So, for an **8-bit** value, the positions would look like this:



For the number 10000000, the leftmost bit (MSB) is **1**, and it represents the **128 place**:

- The first (leftmost) bit is worth -2^7 = **-128** (this is the negative part of the number).
- All other bits are 0, so the value is just **-128**.

Thus, 10000000 in **8-bit two's complement** represents **-128**.

To Summarize:

The **MSB** in two's complement is treated as a **negative** value because it corresponds to the most significant bit's weight, which is a **negative power of 2** (specifically, -2^{n-1}). This is why the **MSB = 1** signals a negative number in two's complement representation.

✍️ WORKED EXAMPLE: Subtracting 39 - 25 using 2's Complement

Let's manually walk through it so beginners *see the real magic*.

➖ Step 1: Convert numbers to 8-bit binary

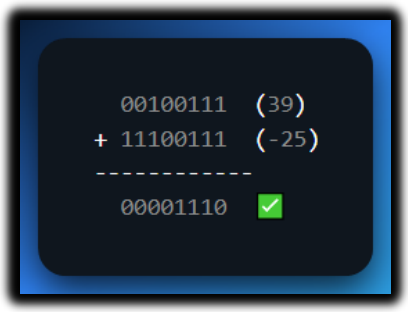
- 39 = 00100111
- 25 = 00011001

➖ Step 2: Take the two's complement of 25 (to make -25)

1. Invert: 11100110
2. Add 1: 11100111

So: **-25 = 11100111**

➖ Step 3: Add the two values



➖ Step 4: Convert 00001110 to decimal

✅ **Result: 39 + (-25) = 14**

✅ *It just works*. No separate subtraction logic needed.

Another example:

Example: -5 + (-3) in 8-bit two's complement.

- -5 = 11111011
- -3 = 11111101

🧠 Even with overflow, we discard extra carry in fixed-width arithmetic. This confuses many new devs.

✅ Why Two's Complement is Better than 1's Complement

FEATURE	1'S COMPLEMENT	2'S COMPLEMENT
Extra Zero?	Yes: You have both a positive zero (e.g., `0000`) and a negative zero (e.g., `1111` for 4 bits) 😬. This wastes a bit pattern.	No: Only one representation for zero (all zeros) 😎. This is more efficient and simplifies logic.
Arithmetic Simpler?	No: Addition and subtraction require an "end-around carry" adjustment if there's a carry out of the most significant bit. This makes hardware design more complex.	Yes: Addition and subtraction work cleanly and directly, just like with unsigned numbers, without any special adjustments for carries. This simplifies CPU arithmetic logic significantly.
Used in real world?	Rarely (mostly for historical context or very niche applications).	Everywhere 🌍: It's the standard method used by virtually all modern computers to represent signed integers because of its efficiency and simplicity in arithmetic operations.

So yeah, **no one really uses 1's complement anymore** — just good to mention it for historical flavor.

One's Complement is where you flip all the bits (0s become 1s, 1s become 0s) to get the negative representation of a number.

Two's Complement is where you flip all the bits (like 1's complement), and then **add 1** to the result to get the negative representation of a number.

✅ 6. Two's Complement of Hex Values — The Shortcut, Demystified

⚙️ PART 1: What's Actually Happening in Two's Complement?

Two's complement is a **way to store negative numbers** using binary.

It's smart because it allows **the same addition circuitry** to handle both positive and negative values. No separate subtractor needed.

But you already knew that.

What's less obvious is how that idea translates into **hex math** — where people often say:

"Just subtract from 2^n." Wait... *Why? What is 2^n? Why are we subtracting?*

🧠 PART 2: What "Subtract from 2^n" Really Means

Let's say you're working in **8 bits**, which is very common.

In 8 bits, the **total number of possible values** is: **$2^8 = 256$**

So, you can represent:

00000000 → 0

11111111 → 255

Now imagine this like a **circular number line** that wraps around from 255 back to 0.

💡 THE CORE IDEA:

When you want to get the **negative version of a number**,
you want to find **what value would "wrap it around" to 0**
in this circular world of 0 to 255.

This "wrapping" is **modulo math** — specifically **mod 256** (because $2^8 = 256$).

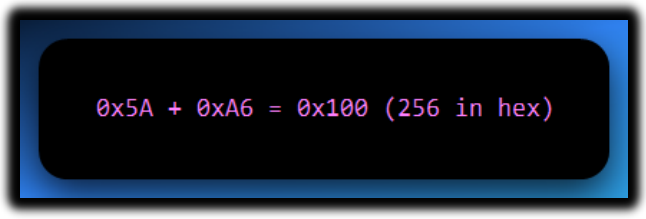
So, you're asking:

What number can I add to 90 to get 256?

Answer:

166 (because 90 + 166 = 256)

In hex:



So clearly, 0xA6 is the **two's complement of 0x5A**.

You flipped it around the 256 mark. That's it.

🎯 SO THE TRICK WORKS BECAUSE...

In two's complement:

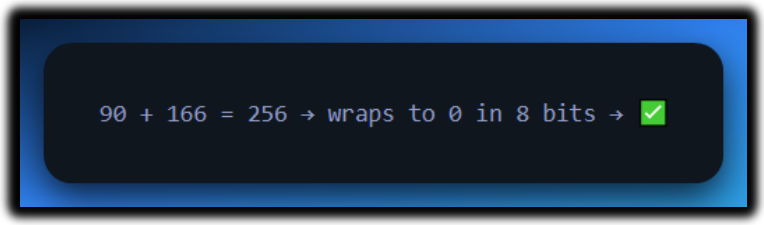
Negative numbers are stored such that:

X + (-X) = 0 (mod 2^n)

So, if:

- X = positive number (e.g. 90)
- -X = two's complement of that number (e.g. 166)

Then:



🧮 NOW DO IT STEP BY STEP WITH EXPLANATIONS

Let's say we want to find -0x5A in 8-bit

Step 1: Understand what 0x5A means

- 0x5A is hex
- 5A in hex = 90 in decimal

Step 2: What's the total possible value in 8 bits?

- $2^8 = 256$
  (because 8 bits can store values from 0 to 255)

Step 3: Subtract 90 from 256

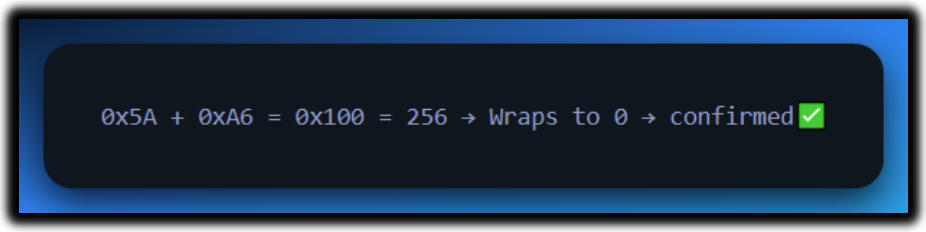
- 256 - 90 = 166

This means that:

- -0x5A = 166 in decimal
- 166 in hex = 0xA6

✅ **Final Answer:**

- Two's complement of 0x5A = 0xA6



Another example: What is -0x2F in 8-bit?

Step 1: Convert hex to decimal

0x2F = 47

Step 2: Use the full range of 8 bits

$2^8 = 256$

Step 3: Subtract 47 from 256

256 - 47 = 209

Step 4: Convert that back to hex

209 = 0xD1

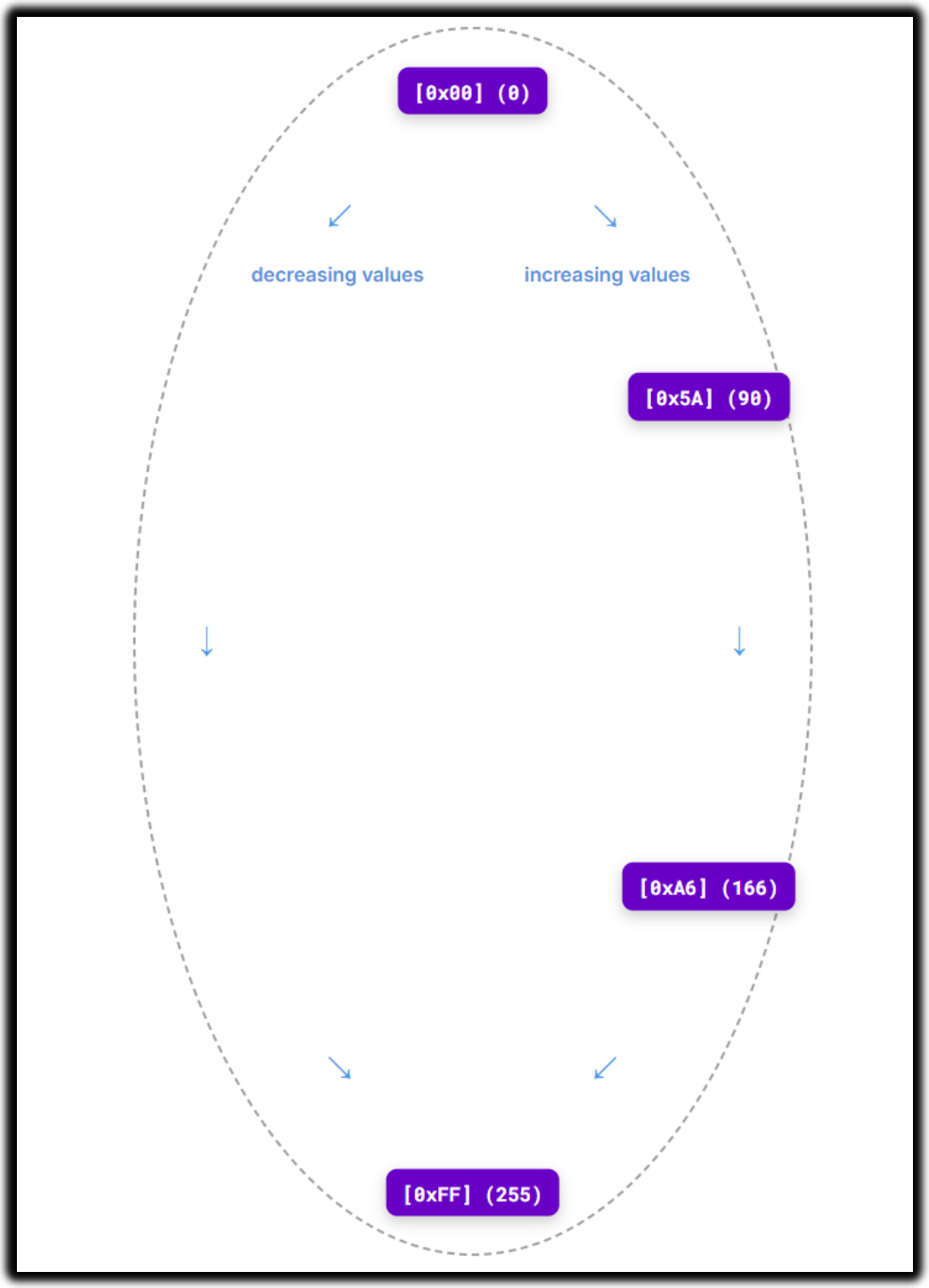
✅ **Final Answer:**

-0x2F = 0xD1 in 8-bit two's complement

✅ **Because:**

0x2F + 0xD1 = 0x100 = 256 → wraps to 0 → valid

🤯 Let's Try to Visualize It with a Circle (Mod 256)



This is why subtraction works:

You're finding the number that would complete the circle to 256.

⚠️ Two's Complement Shortcut: Important Conditions

The "subtract from 2^n" shortcut for Two's Complement is incredibly powerful, but like any powerful tool, it comes with specific conditions for when it works correctly. Make sure you keep these in mind to avoid unexpected results!

CONDITION	REASON
You know how many bits you're working with (n)	The value of 2^n (the "wrap-around" point) depends entirely on the bit width. For example, 2^8 is 256, but 2^{16} is 65,536. Using the wrong 'n' will give you an incorrect result.
The original value fits within that bit width	If the number you're trying to find the two's complement of is too large for the specified 'n' bits, it will overflow, and the shortcut won't yield the correct signed representation. Always ensure your number is within the unsigned range of 0 to 2^n-1.
You subtract from 2^n (in decimal or hex), not from 0xFF, 0xFFFF, etc.	This is a common pitfall! 0xFF (for 8 bits) is 255, which is 2^8 - 1. You need to subtract from the total number of states (2^n), not the maximum value. In hex, for 8 bits, this means subtracting from `0x100`, not `0xFF`. For 16 bits, it's `0x10000`, not `0xFFFF`.

🤖 USING THIS IN REAL LIFE

In C or Assembly:

You're storing something like:

```
char x = -90;  // gets stored as 0xA6
```

In Malware:

You encode a positive value like 0x2F, but store it as 0xD1 (which looks like junk) to hide meaning.

In File Formats or Debuggers:

You read a hex dump and see 0xA6. What does that mean?

- If unsigned: 166
- If signed (two's complement): -90

Now you know how to **decode and encode both ways**.

📌 TLDR REWRITE — MAKE THIS UNDERSTANDABLE

What is Two's Complement in Hex?

Two's complement is how computers represent **negative numbers** using only binary.

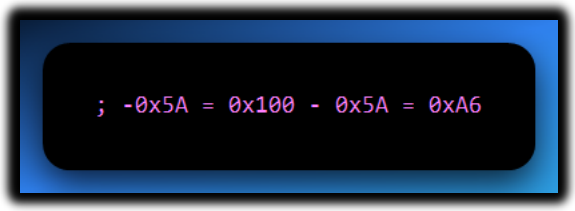
Hex is just a cleaner way to write binary. So to represent -X in hex, we can use math.

The Shortcut

To find the two's complement of a hex value,
subtract it from 2^n, where n is the number of bits.

In 8 bits, that's $2^8 = 256 = 0x100$

So:



That means:

- 0x5A = +90
- 0xA6 = -90

Because they add up to 0x100, which wraps to 0.

When Should You Use This?

Situation	Use the shortcut?
You're comfortable with hex math	✅ Yes
You're reading binary files	✅ Yes
You're teaching someone new	❌ Explain bit inversion first
You're working with 32-bit or 64-bit	✅ Just use 2^{32}, 2^{64}, etc.

Final Examples

Example 1: **-0x2F**

```
0x2F = 47

256 - 47 = 209 = 0xD1

→ -0x2F = 0xD1
```

Example 2: **-0x7A**

```
0x7A = 122

256 - 122 = 134 = 0x86

→ -0x7A = 0x86
```

✅ 1. Signed Binary to Decimal — Two's Complement Style

💡 Problem: How do you convert binary like 10101011 into a negative decimal?

When you're dealing with signed binary (two's complement), the highest bit (leftmost one) tells you **if it's negative**:

- 0 = positive
- 1 = negative (so we need to decode it differently)

🎯 EXAMPLE: Convert 10101011 to decimal

Let's say it's an 8-bit signed number.

Step 1: Check the leftmost bit (MSB)

It's 1 → this number is **negative**

Step 2: Take two's complement (to find the positive version):

Invert the bits:
10101011 → 01010100

Add 1:
01010100 + 1 = 01010101
= **binary 01010101 = 85**

So, the original number was:

🔁 **-85**

✅ Final Answer: 10101011 = **-85 (in decimal)**

🍼 Another One: Convert 11010101 (8-bit) to decimal

1. Leftmost bit is 1 → negative
2. Invert: 11010101 → 00101010
3. Add 1: 00101010 + 1 = 00101011
4. 00101011 = 43

✅ So 11010101 = **-43**

🧠 Max and Min Values (8-bit Two's Complement)

TYPE	BINARY RANGE	DECIMAL RANGE
Unsigned	00000000 – 11111111	0 – 255
Signed	10000000 – 01111111	-128 – +127

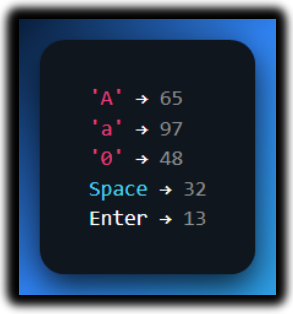
01111111 = 127

10000000 = -128 (the lowest value possible in 8-bit Two's Complement)

✅ 2. Character Sets: ASCII, Extended ASCII and Unicode

🧠 ASCII — American Standard Code for Information Interchange

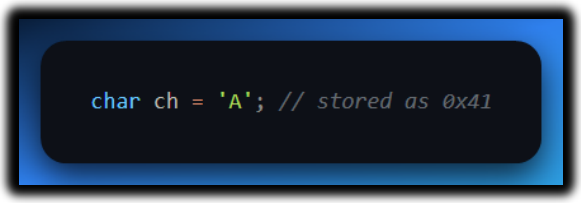
A **cheat sheet** that assigns specific numbers to each keyboard key and character, making it easier for computers to process and understand text.



ASCII uses a **7-bit system**, which allows it to represent **128 unique characters** in total.

These characters include English letters (both uppercase and lowercase), digits, punctuation marks, and some control characters (like newline, carriage return).

In memory, each ASCII character is **stored as a byte** (8 bits).

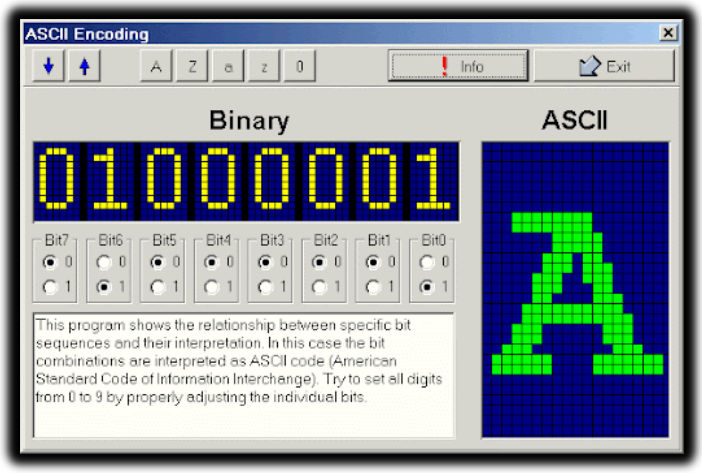


Although ASCII itself only uses **7 bits for each character**, it is typically stored in an 8-bit byte, with the **extra bit** usually set to 0.

Example:

The ASCII value for the letter **"A"** is **65**. In binary, this is represented as **01000001** (7 bits, with an extra 0 bit for storage).

In memory, this would be stored as the **byte 01000001**, which is equivalent to **0x41** in hexadecimal.



🧠 Extended ASCII

So, the **Extended ASCII** set uses 8 bits (1 byte), allowing for a total of **256 characters** (0-255). It has the ordinary 0 to 127… but also…

The characters in the range 128-255 are typically used for non-English characters and special symbols (See the images and html files) e.g.

- **0x41 (65 in decimal)** = "A" (Standard ASCII)
- **0xE9 (233 in decimal)** = "é" (Extended ASCII)

This extension was useful in environments that needed to support multiple languages or special symbols, but it has limitations, which is why it was eventually superseded by the **more powerful Unicode** standard.

The range 128 to 255 expands the original set to use the 8th bit, allowing for additional characters. This part includes:

- **Special symbols**, like **currency symbols** (€, £, ¥)
- **Characters from other alphabets**, such as **é**, **ñ**, and other accented characters.
- **Graphics**, like box-drawing characters, and some early **emojis** (though emojis today are much more advanced in Unicode).

🌎 Unicode — Go Global or Go Home

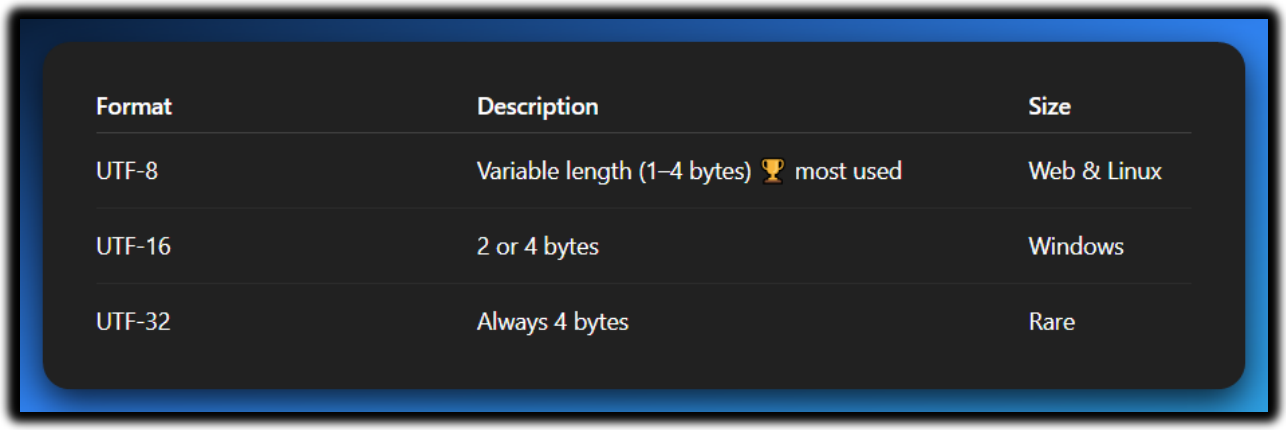
ASCII wasn't enough when humans needed:

- Arabic.
- Chinese.
- Emoji.
- Mathematical symbols.

So, we got **Unicode**, which maps **over 100,000+ characters**.

🔁 Unicode Transformation Formats (See the html and images for Unicode)

Read the Unicode.html and the image for more information.

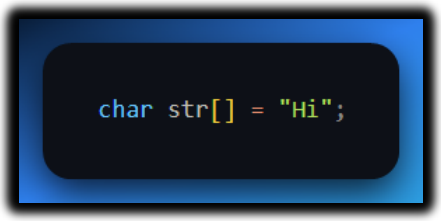


Format	Description	Size
UTF-8	Variable length (1–4 bytes) 🏆 most used	Web & Linux
UTF-16	2 or 4 bytes	Windows
UTF-32	Always 4 bytes	Rare

📄 *Let's first divert from Unicode for a minute...*

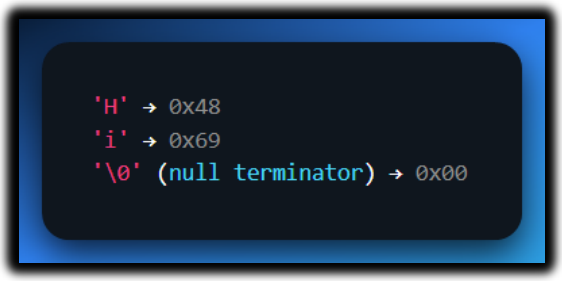
💡 How Are Strings Stored?

Characters are stored as bytes representing their character code. Example:



When you declare a string like above in C, the characters **'H'** and **'i'** are converted into their corresponding numerical ASCII (or other encoding) values.

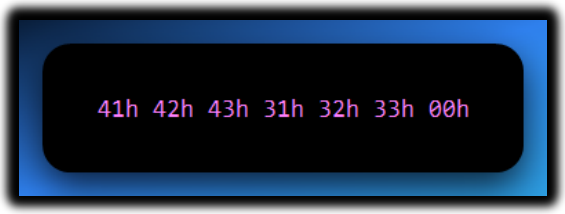
Memory view (ASCII):

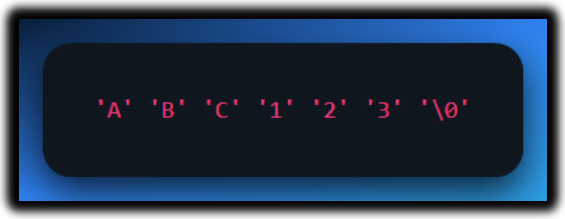


- **'H'** is stored as 0x48 (hexadecimal).
- **'i'** is stored as 0x69 (hexadecimal).

Null Terminator: A crucial part of C-style strings is the **null terminator**, represented as '\0' or 0x00. This byte marks the end of the string in memory.

The string "ABC123" is stored in memory like this:



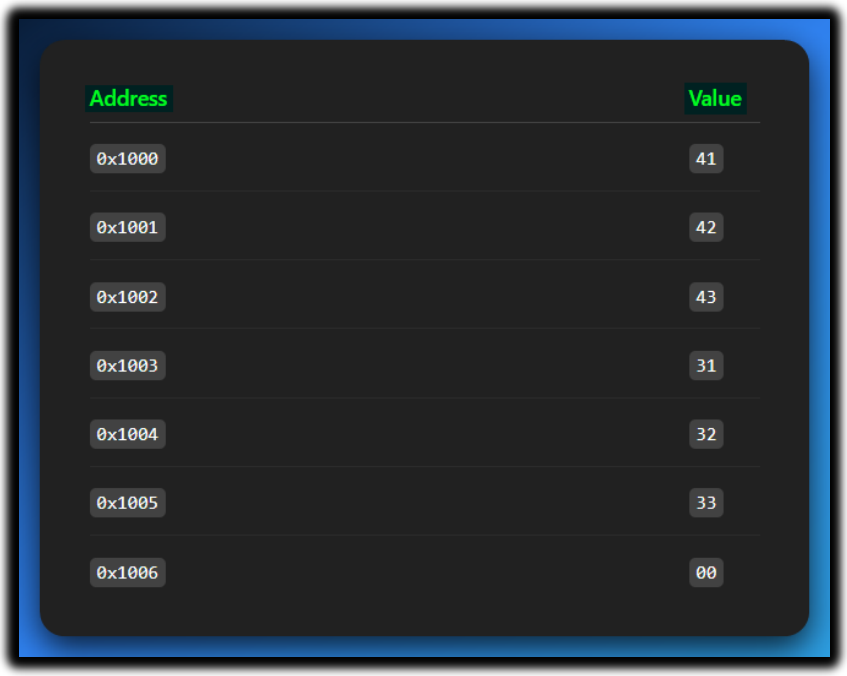


Every character = 1 byte. Last byte = 00h (null terminator)

🧠 So wait... Does 41h mean memory location or character?

- 🔍 **Answer:** 41h is **not** a memory address.
- 🔍 It's the **numeric value of the ASCII character 'A'**.

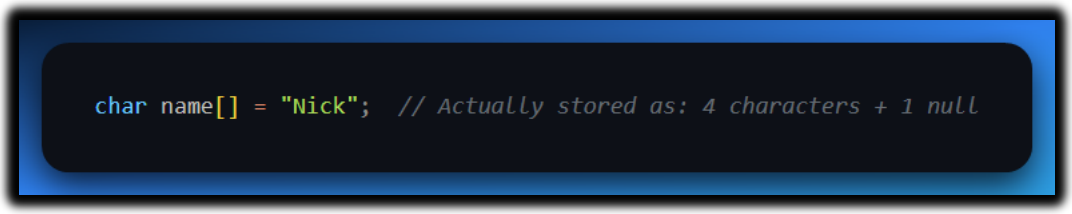
Memory might look like:



Address	Value
0x1000	41
0x1001	42
0x1002	43
0x1003	31
0x1004	32
0x1005	33
0x1006	00

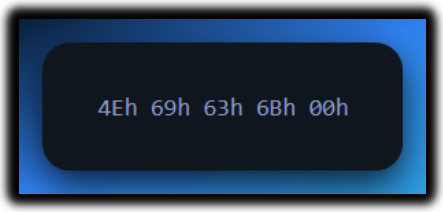
⚠️ Why the Null Terminator?

Because in **C, C++, WinAPI, Assembly** — a string doesn't store its length.
So the only way to know where a string ends is to look for the **00h null byte**.



```
char name[] = "Nick";  // Actually stored as: 4 characters + 1 null
```

In memory:



```
4Eh 69h 63h 6Bh 00h
```

🔧 Control Characters (ASCII 0–31) – As seen in the ASCII.table.html

ASCII 0–31 = **non-printable control characters**, like:

- 0x0A (10d) = \n (newline)
- 0x08 (8d) = \b (backspace)
- 0x00 (0d) = \0 (null terminator)

In C/ASM, you'll see them written using **escape sequences**:

```
printf("Hello\nWorld");   // Moves to new line
```

CONCEPT	KEY POINT
Signed Binary to Decimal	If MSB is 1, invert bits and add 1, then make it negative.
Max/Min for Signed (8-bit)	-128 to +127
ASCII	7-bit, basic characters (0-127).
Extended ASCII (ANSI)	8-bit, includes symbols and accented letters (128-255).
Unicode	Supports every language on Earth (and emoji 😎).
UTF-8	Most common encoding — variable byte length (1-4 bytes).
Null-Terminated String	Ends with a 0x00 byte (common in C/C++).

🔢 Why is $2^7 - 1 = 127$ the max for signed 8-bit integers?

🧠 The Core Rule:

If you're using **n bits** to store a **signed integer** in **two's complement**, then:

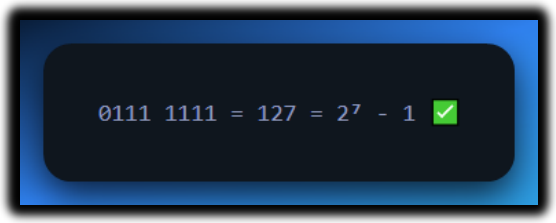
- You get **1 bit for the sign** (positive or negative)
- That leaves **(n - 1)** bits for the value

So, for **8 bits** total:

Part	Bits
Sign bit	1
Value bits	7

🎯 Max positive value?

You use all 7 value bits as 1s:



That's why $2^7 - 1 = 127$ is the max positive value in **signed 8-bit**

🎯 Total number of values?

For n bits, total combinations = 2^n

So:

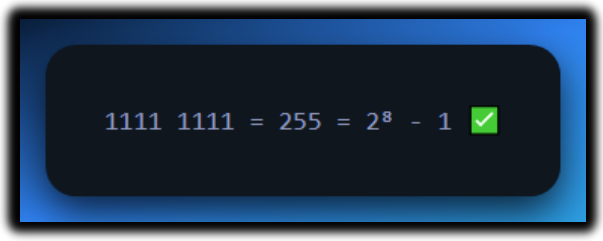
- 8-bit → 256 total values
- That's the range of **signed** 8-bit values: **-128 to +127**

Because:

- Min: 1000 0000 = -128
- Max: 0111 1111 = 127

🧠 Why $2^8 - 1 = 255$ for unsigned?

If you're not using a sign bit (unsigned), **all 8 bits** are for value:



🔁 Summary Table

Type	Bits Used	Formula	Range
Unsigned 8-bit	8	`0 to 2⁸ - 1`	`0 to 255`
Signed 8-bit	7 (value)	`-2⁷ to 2⁷ - 1`	`-128 to 127`

I was **mixing unsigned vs signed logic** in my head.

📜 *Back to Unicode...*

📦 Unicode Encoding Formats — UTF-8, UTF-16, UTF-32

✨ What is Encoding? (all these are mentioned in the htmls and images)

Encoding = turning characters into binary for storage or transmission.
Decoding = converting the binary back to characters.

Unicode is the global standard that assigns **each character** (letters, numbers, emoji, symbols, etc.) a **code point** — a unique number.

But we still need to *encode* those code points into bytes so computers can store them in memory or send them across the web.

That's where **UTF-8**, **UTF-16**, and **UTF-32** come in.

🔢 UTF-8 (Most Popular)

- 🔧 **Variable-length encoding**: uses **1 to 4 bytes per character**
- ✅ First **128 characters** (ASCII) = 1 byte (so it's backwards compatible with ASCII)
- 📦 Supports all Unicode characters (up to U+10FFFF)
- ⚡ **Efficient for English and common languages** — because most characters use only 1 byte
- 🌐 **Most widely used on the web, Linux, APIs, and emails**

✅ Example

CHARACTER	CODE POINT	UTF-8 ENCODING
A	U+0041	0x41
€	U+20AC	0xE2 0x82 0xAC
𐍈	U+10348	0xF0 0x90 0x8D 0x88
👋	U+1F44B	0xF0 0x9F 0x91 0x8B
你好	U+4F60 U+597D	0xE4 0xBD 0xA0 0xE5 0xA5 0xBD

🧪 UTF-16

- 🔧 **Variable-length encoding**: uses **2 or 4 bytes per character**
- 🧠 First **65,536 characters** (Basic Multilingual Plane) use **2 bytes**
- 👀 Rare characters like emojis and historical scripts use **surrogate pairs** (4 bytes)
- 📌 Used in **Windows**, **Java**, and .NET

✅ Example

CHARACTER	CODE POINT	UTF-16 ENCODING
A	U+0041	0x0041
€	U+20AC	0x20AC
𐍈	U+10348	0xD800 0xDF48 (surrogate pair)
👋	U+1F44B	0xD83D 0xDC4B (surrogate pair)
你好	U+4F60 U+597D	0x4F60 0x597D

🚧 UTF-32

- 🧱 **Fixed-length encoding**: **4 bytes per character**
- 🔍 Every character — no matter how simple or complex — uses **exactly 4 bytes**
- ⚖️ Simple and predictable (no need to deal with variable lengths or surrogate pairs)
- 😬 Very **inefficient** in terms of memory/storage
- 📌 Used mainly in **low-level systems**, like C/C++ internal handling

✅ Example:

CHARACTER	CODE POINT	UTF-32 ENCODING
A	U+0041	0x00000041
€	U+20AC	0x000020AC
𐍈	U+10348	0x00010348
👋	U+1F44B	0x0001F44B
你好	U+4F60 U+597D	0x00004F60 0x0000597D

Pick your Unicode Transformation Format

FEATURE	UTF-8	UTF-16	UTF-32
🧠 Efficiency	✅ Best for most text (especially English-centric)	Good for Asian scripts (fixed-size for common characters)	❌ Worst (wastes space for common characters)
📦 Size	1–4 bytes per character (variable)	2–4 bytes per character (variable)	4 bytes always per character (fixed)
🌍 Popularity	⭐⭐⭐⭐ Most popular, internet standard	Windows, Java, JavaScript internal representation	Rare (used in some low-level systems for consistency)
🔙 ASCII Support	✅ Perfect (ASCII characters are 1 byte UTF-8)	✅ Yes (first 128 characters are 2 bytes, leading zeros)	✅ Yes (first 128 characters are 4 bytes, leading zeros)
💪 Use Case	Web pages, APIs, email, files, Linux/Unix systems	Internal string representation in Windows, Java, JavaScript; some text editors	Niche applications requiring fixed-width character access, some internal string processing

🧠 Pro Tip:

UTF-8 is the default for files, web, APIs, and databases.

UTF-16 is mostly for Windows apps or Java-based systems.

UTF-32 is best used when you need fixed-width encoding and don't care about memory (e.g., internal char buffers).

📚 TERMINOLOGY IN DATA REPRESENTATION – FOR BEGINNERS

In assembly language and low-level systems work, you've got to be **ultra-precise** with how you describe numbers, characters, and what's actually in memory.

Why? Because the **same number** (like 65) can have different meanings depending on:

- ⚙️ where it is (memory vs screen).
- 📺 how it's interpreted (ASCII vs integer).
- 💾 how it's formatted (binary vs hex vs string).

👇 Example: The Number 65

Representation Type	Value	Meaning
Binary integer	`01000001`	Just 65 in binary, ready for math
Hexadecimal	`41h`	Hex version of 65
Decimal digit string	`"65"`	Two ASCII characters: `'6'`, `'5'`
ASCII character	`'A'`	Because `65` = ASCII code for `'A'`

🧠 Memory Contexts

Let's break it down in a real-world scenario:

Action	What happens
You store `65` as a byte in RAM	It's stored as `01000001` in binary
You view it in a debugger	It shows up as `41h` (hex)
You print it on screen as a char	You see `'A'`
You treat it as a string	It's not `'A'`, it's the characters `'6'` and `'5'` (ASCII `36h` and `35h`)

🔠 Number 65: Character vs String in ASCII

When you store the value **65**, how it's interpreted depends on the **data type**:

- If you treat it as a **character** (like chr(65) in Python), it becomes 'A', because 65 is the ASCII code for 'A'.
- If you treat it as a **string** (like str(65)), it becomes '65', which is literally the two characters '6' and '5', not 'A'.

🧠 TLDR:

65 → 'A' if treated as a **character**
65 → '6' and '5' if treated as a **string of digits**

They look the same on the outside, but they're **totally different under the hood**. One is a single byte with value 65 ('A'), the other is two bytes: **0x36** ('6') and **0x35** ('5').

📌 Key Terminology

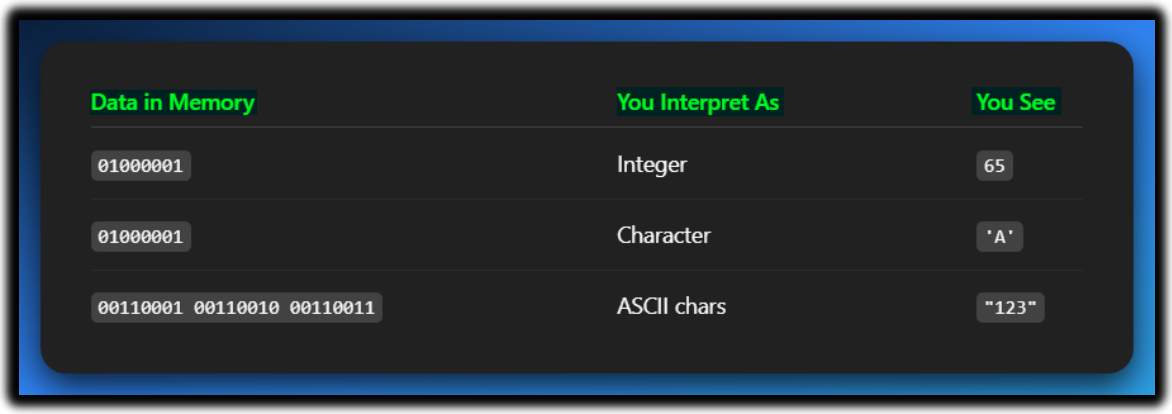
Let's define the exact terms you'll see in docs and assembly manuals:

Term	Meaning
Binary Integer	A number stored in raw binary, e.g. `65` as `01000001`. Used in math.
Digit String	A sequence of ASCII characters that *represent* digits. `"123"` is 3 bytes: `31h 32h 33h`.
ASCII Character	A symbol like `'A'` or `'0'`, stored as its ASCII code in memory.
Hexadecimal	A human-readable base-16 format. `41h` = 65 decimal = `'A'` in ASCII.
Octal	Base-8 version of a number. `101o` = 65 decimal. Still useful in Unix permissions.

📦 Data Storage: You Decide the Meaning

What's wild is — the **computer doesn't care** what 01000001 *means*.

It's just 8 bits. You — the programmer — **decide how to interpret it.**



Data in Memory	You Interpret As	You See
`01000001`	Integer	`65`
`01000001`	Character	`'A'`
`00110001 00110010 00110011`	ASCII chars	`"123"`

🧠 TLDR

Same binary data can be an **integer**, a **character**, or a **string — context defines the meaning**.

A **binary integer** is raw math data. A **digit string** is made of ASCII bytes that look like numbers.

Hex (41h) is just another way to write the same data — it's easier to read for humans.

ASCII **'A'** = **65** = **0x41** = **01000001**.

🔥 Not important but good to know notes: Hex and ASCII Fast-Lookup Tip

If you're working in low-level code or reading memory:

- **'A'** → 0x41 → 01000001
- **'0'** → 0x30 → 00110000
- **'9'** → 0x39 → 00111001

Once you memorize these ranges (0x30–0x39 for digits, 0x41–0x5A for uppercase letters), you'll **read hex dumps like the Matrix**.

Let's go **deep** — not just on how to read ASCII and hex dumps, but also why it's *super important* when you're inside tools like **x64dbg**, **Ghidra**, or **IDA**.

🔥 WHY YOU NEED TO RECOGNIZE HEX ⟶ ASCII RANGES

When you reverse engineer software, you're constantly reading **memory dumps**, **disassembly**, or **binary blobs**.

Inside those dumps are:

- Strings (like usernames, passwords, commands)
- Constants (e.g., 'A', '0', '9', etc.)
- File formats (PDF, PE headers, image metadata)
- Protocols (HTTP, binary protocols, etc.)

But guess what? **Check the html's and images for more context to be honest.**

All of those are just **bytes** — and those bytes often encode **ASCII characters**. If you can spot those by eye, you start **seeing** stuff that's hidden to normal devs.

💡 ASCII CHEAT TIP

Here's the gold — memorize these three **ASCII hex ranges**:

Character Type	Decimal Range	Hex Range	Notes
Digits '0'–'9'	48–57	`0x30–0x39`	`'0' = 0x30`, `'9' = 0x39`
Uppercase 'A'–'Z'	65–90	`0x41–0x5A`	`'A' = 0x41`, `'Z' = 0x5A`
Lowercase 'a'–'z'	97–122	`0x61–0x7A`	`'a' = 0x61`, `'z' = 0x7A`

Let's continue with shifts and rotates from the next docx, cool?