

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 (0xFFE43A0 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
<code>0x...</code>	Hexadecimal (base 16)	<code>0x2F</code>	47
<code>...b</code>	Binary (base 2)	<code>10101010b</code>	170
<code>0...</code>	Octal (base 8)	<code>075</code>	61
<i>no prefix</i>	Decimal (base 10)	<code>47</code>	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 $012 = 1 \times 8 + 2 = 10$ 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) $\rightarrow 4 \times 8 + 2 = 34$ decimal
Hex	0x42	Hex (base 16) $\rightarrow 4 \times 16 + 2 = 66$ decimal
Binary	0b101010	Binary (base 2) $\rightarrow = 42$ 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 <code>0x...</code>	Memory, registers, shellcode, colors	Compact, maps directly to binary
Binary <code>0b...</code>	Flags, GPIO, bit manipulation, masks	Clear view of bit-level logic
Octal <code>0...</code>	Linux permissions (<code>chmod 0755</code>)	Legacy Unix stuff
Decimal	User input/output, formulas	Human-friendly

Back to data conversion

Binary to Hexadecimal Conversion

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

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

Positions	→	7	6	5	4	3	2	1	0	(Bit numbers)
Binary	→	0	1	0	0	1	1	1	0	
Powers	→	128	64	32	16	8	4	2	1	

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:

10000000 → MSB is 1 → this means a *very Large* value in 8-bit Land - 128 specifically
00000001 → LSB is 1 → this just means the number is 1

⊕ 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.

Signed value

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = -1

Unsigned value

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = 255

🧠 What is Two's Complement?

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



Two's
Complement

🌐 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

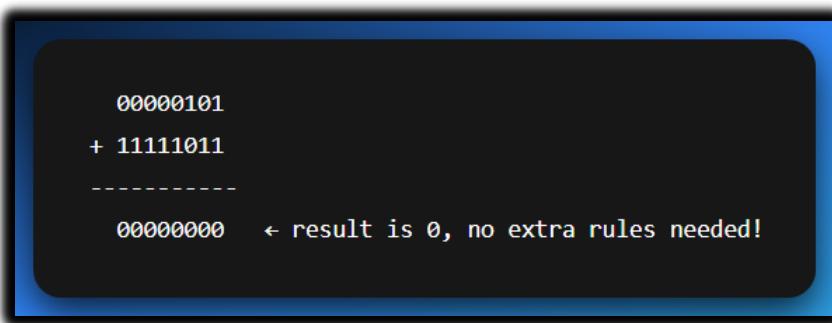
Step 1: Invert the bits → 11111010

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:


$$\begin{array}{r} 00000101 \\ + 11111011 \\ \hline 00000000 \end{array}$$

← 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 → $2^8 - 1 \rightarrow 255$** as the largest value:

$$11111111 \text{ (binary)} = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

✓ 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.

```
11011110001110000000          → 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**.

12 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:

$$(1 \times 128) + (0 \times 64) + (1 \times 32) + (0 \times 16) + (0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 128 + 32 + 4 + 2 = 166$$

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

1000 ? This is how 8 looks in binary

⌚ Why?

Because:

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

$$(1 \times 8) + (0 \times 4) + (0 \times 2) + (0 \times 1) = 8$$

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.

let's - let us , and it's - it is. bruuuuuuuuuuuuuuuuuuuh.
how?

 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:

Binary: 1 0 1 1
Index: 3 2 1 0 ← These are powers of 2

To convert to decimal:

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 11$$
$$8 + 0 + 2 + 1 = 11$$