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

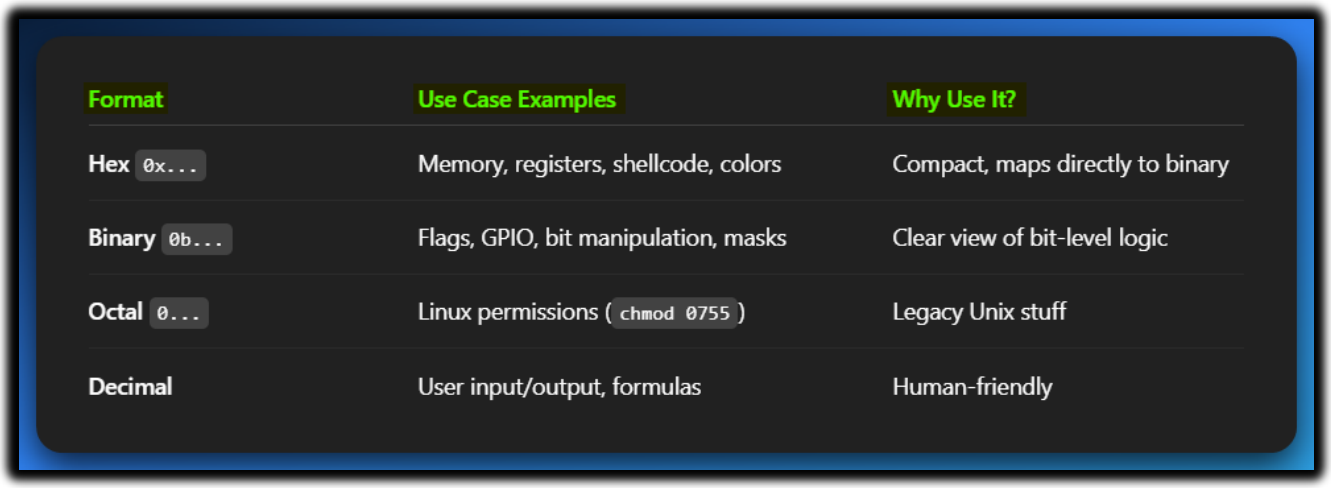
```
mov ecx, 100          ; Loop counter in plain English
```

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

✅ When to use:

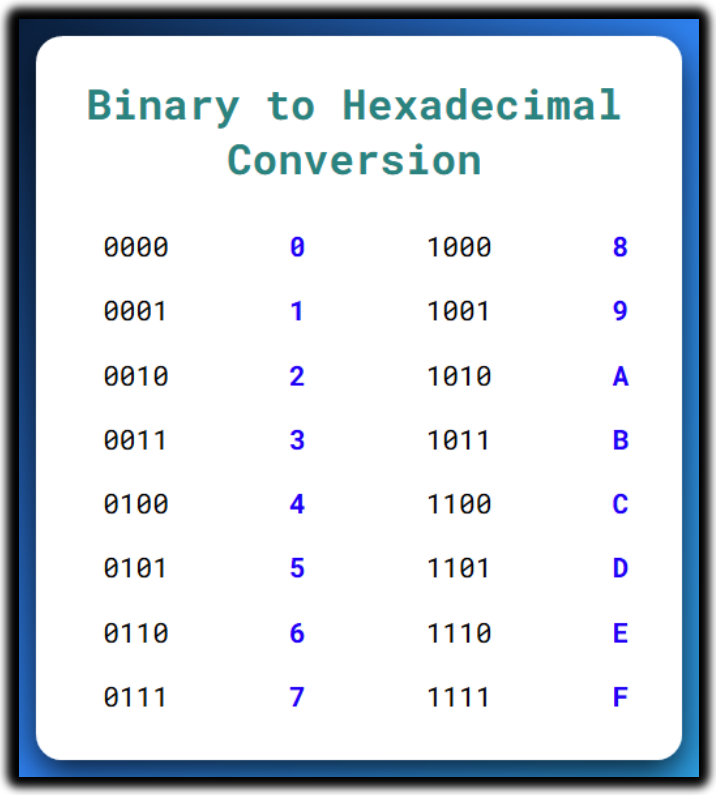
- User-facing numbers
- Calculations or percentages
- Output/logs

⚡ 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**:

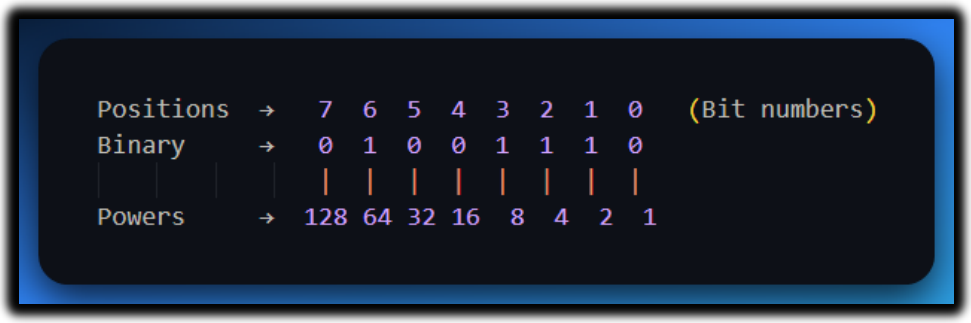
- **1 = ON (Electricity flowing)**
- **0 = OFF (No electricity)**

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