💡 FIRST PRINCIPLES: WHAT IS ADDRESSING, REALLY?

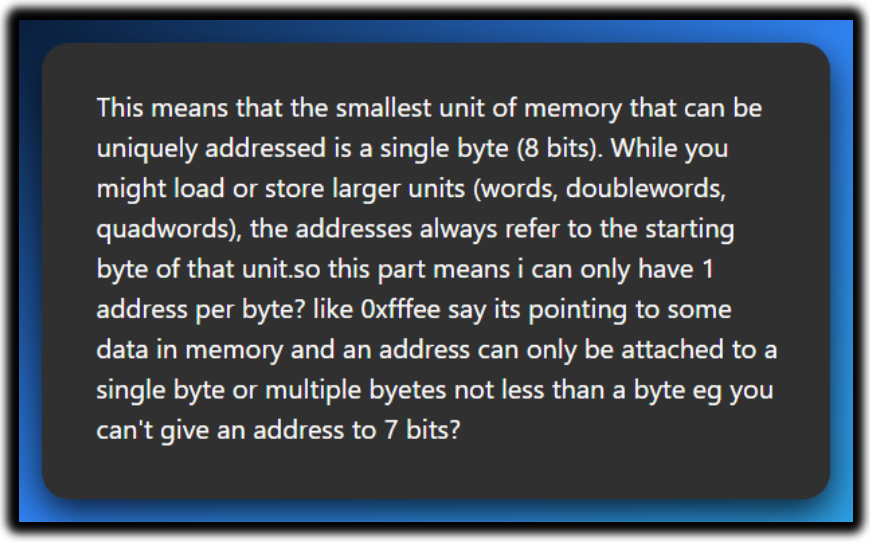
Think of the CPU like a blind beast that only knows how to poke memory by **number** — it doesn't "see" variable names, doesn't care about objects or data types — it just sees **addresses**.

So:

📍 **Addressing** = Telling the CPU *where* to look (or write) in its giant memory book.
🧠 **Address Space** = The full size of that book — how many unique "pages" (addresses) it can have. The range of memory addresses available to a process or system.

🔎 Byte-Addressability — Why It Matters

x86 is byte-addressable.



👉 **Each unique memory address points to exactly one byte.**

👉 **Not a bit. Not 5, 6, 7 or 9 bits. Just 1 full byte = 8 bits.**

You *"can only have 1 address per byte? like 0xfffee pointing to some data..."*

Yes! 0xfffee is the address of **a single byte** in memory.

This means that the smallest unit of memory that can be uniquely addressed is a single byte (8 bits).

While you might load or store larger units (words, doublewords, quadwords), the addresses always refer to the starting byte of that unit.

Even if that byte is *part* of a bigger thing (like a 4-byte integer), the CPU still treats addresses as pointing to **bytes only** — and you just use consecutive addresses to grab the rest:

- **32-bit value?** → uses 4 bytes → starts at 0xfffee, continues to 0xfffef, 0xffff0, 0xffff1
- **64-bit value?** → uses 8 bytes → starts at one byte address, spans 8 bytes. But each of those 8 bytes still has **its own unique address** like the 32-bit version shown above.

You can't address individual bits (like "give me bit #3 in byte at address 0xfffee").

If you need to access a bit, you do:

See? Still fetched the whole byte — then *you* manually isolate the bit.

🍼 Analogy Time (Gen Z style):

🧊 Imagine memory is like a line of lockers.

- Every locker = 1 byte
- Locker number = memory address
- You can open locker #101 (0x65) and get what's inside — one byte
- Want a 64-bit doubleword? You just open 8 lockers in a row starting from some number.

BUT…

❌ You **can't** label **"the left sock inside locker #101"** as its own address
✅ You just open locker #101 and **dig into its contents** if you need something smaller.

⚠️ Fun fact:

Some really old or embedded systems are **bit-addressable**, but **x86, ARM, RISC-V, etc — all byte-addressable.**

So, you're absolutely right to say:

An address can only refer to a byte — nothing smaller.

No "bit-addressing" unless you're working with custom hardware or ancient microcontrollers.

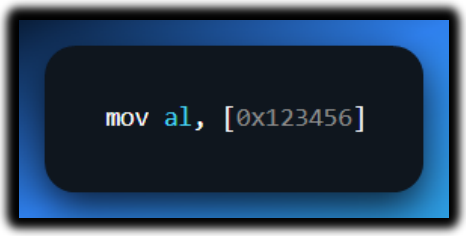
Byte-addressable sounds like a boring fact until you realize…

- You can modify **one single character in a password string**.
- You can NOP out a **single instruction** in shellcode.
- You can corrupt just the **last byte** of a return address, and change program flow.

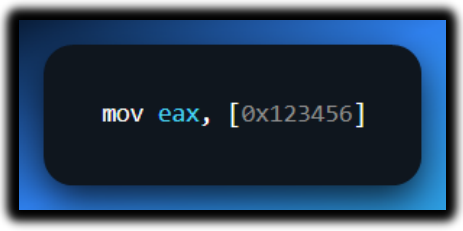
Byte-level control = surgical power.

In older architectures (word-addressable), you couldn't just poke one byte. But x86? You've got scalpel-level access. So:

Grab 1 byte from memory.



Grab 4 bytes starting from that address.



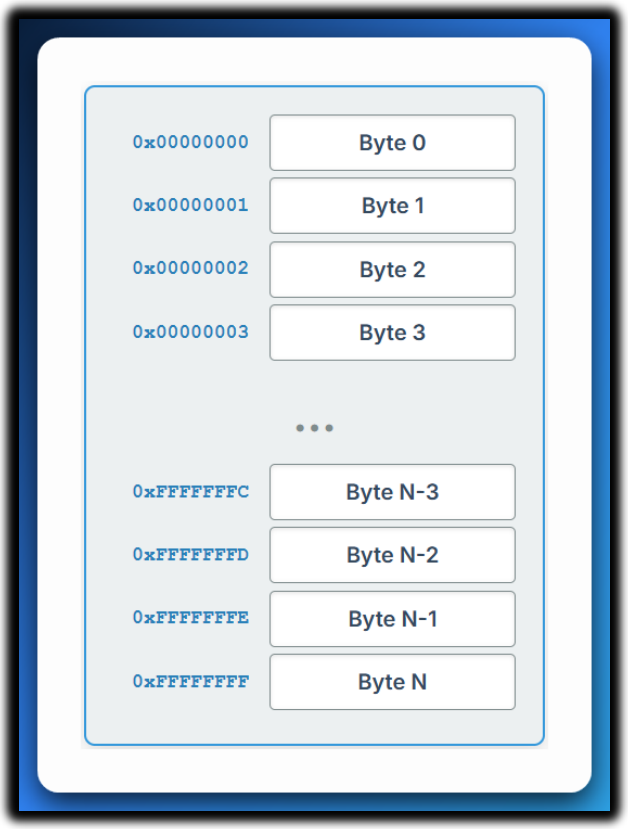
But the address always refers to the first byte.

That's like saying:

"Go to house 123456. If I say byte, steal one envelope. If I say word, take four in a row, starting from that house."

📏 Linear Memory Model: The Illusion of Simplicity

This is the *illusion* every beginner sees:



See the Addressing.html

Looks clean, right? Like RAM is just a big array of bytes you can index into like RAM[0x123456].

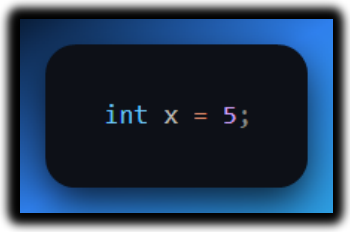
But reality? 👇

🚨 The Plot Twist: Linear View Is a Lie (Sort of)

Underneath this smooth address space is a **warzone** of mappings, protections, virtual memory tricks, segmentation (in real/protected mode), and paging tables.

- In **real mode**, segment:offset math gives fake "linear" addresses.
- In **protected mode**, the OS maps your 0x00400000 to some other **physical** location via page tables.
- In **long mode (x86-64)**, even *more* indirection.

So, when you think:



Your code might think x lives at 0x00401234, but under the hood, that:

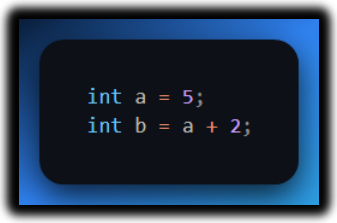
- Is virtual.
- Mapped to a page.
- Backed by physical RAM, or not.
- Maybe even swapped to disk.
- All depending on your mode and OS.

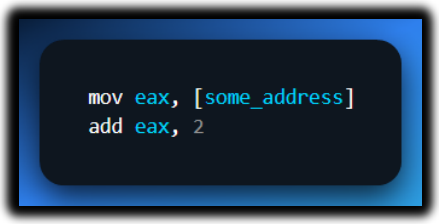
So yeah — it **starts** as a linear view. But when reversing or writing exploits, you'll often have to:

- Find the **real** physical or kernel address.
- Traverse **page tables.**
- Bypass **address randomization (ASLR).**
- Reconstruct **segment mappings** in 16-bit code.

🔥 Assembly vs High-Level: Why You NEED to Know This

In C:





You have to **know** where that address points.

- Did the OS give you that address?
- Is it in the data section?
- Heap? Stack?
- Is that memory even mapped and readable?

One wrong assumption = segmentation fault, or worse, silently reading junk data.

🎯 Final Thought (For This Chunk)

Addressing isn't just "how memory works."

It's **how everything works** once you're outside the compiler's sandbox.

- Want to write a shellcode? You'll hand-write addresses.
- Want to trace a malware unpacker? You'll follow memory derefs.
- Want to write a bootloader? You'll directly manage segment:offset pairs.
- Want to debug a crash? You'll match an instruction with a corrupt address.

So, here's a rule to tattoo mentally:

❗ **If you don't control the address, you don't control the code.**
❗ **If you don't understand the address, you're blind.**

🔁 Endianness — The CPU's Accent When Speaking Memory

Endianness is like the **dialect** your CPU uses to write down numbers in memory.

You and your friend both know what the number "0x12345678" means — but if one writes it **backwards**, and the other reads it **forwards**, y'all are gonna get very confused real quick.

Let's break it down to **core concepts**, **visuals**, **real-world cases**, and why it matters in RE/malware/file parsing.

📦 What is Endianness?

When a value spans **multiple bytes** (e.g., a 32-bit int = 4 bytes), the CPU has to decide:

"Which byte goes to the *lowest* memory address? The most significant one (biggest part) or the least significant one (smallest part)?"

That decision = **endianness**.

✊ Little-Endian (Used by Intel x86/x64, AMD64)

"Put the Least Significant Byte (LSB) at the Lowest Address."

It's **backwards** to how humans normally write numbers — but not to the CPU.

Example:

Let's store 0x12345678 into memory.

Byte	Value
+0	`0x78` ← LSB (least significant byte)
+1	`0x56`
+2	`0x34`
+3	`0x12` ← MSB (most significant byte)

So, in **hex dumps or memory editors**, you'll see: **78 56 34 12**

And you gotta mentally flip it to read the real number. This **byte reversal** is why malware analysts constantly “swap bytes” in IDA or Ghidra.

🧠 *Mnemonic:*

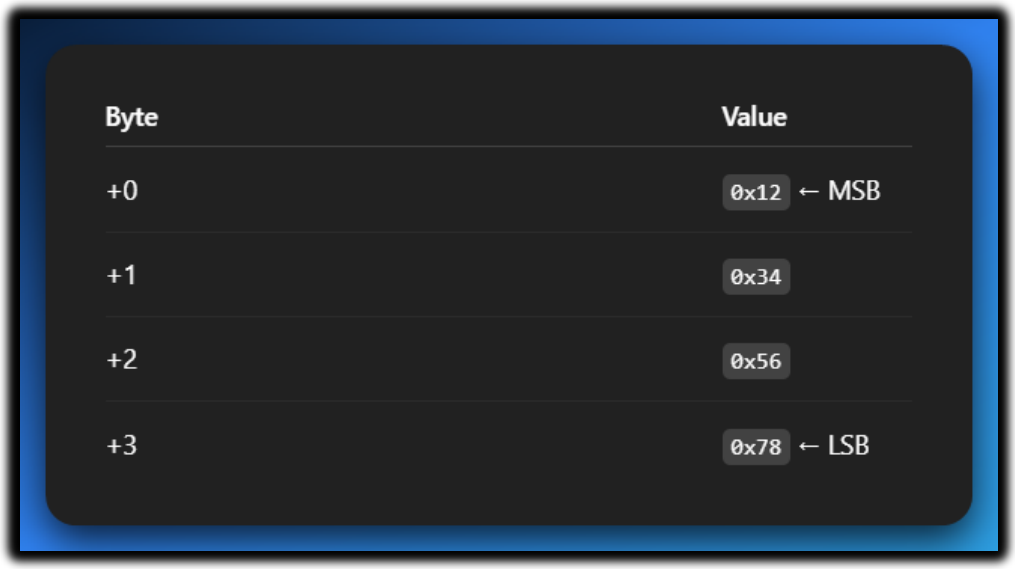
*"Little-endian puts the **little stuff first**." (just like how babies say "me first" 😆)*

🤯 Big-Endian (Used in old-school CPUs, network protocols)

"Put the **Most Significant Byte (MSB)** at the Lowest Address."

It feels more “natural” to humans. Example:

Same number: **0x12345678**



Byte	Value
+0	`0x12` ← MSB
+1	`0x34`
+2	`0x56`
+3	`0x78` ← LSB

So, in memory dump: **12 34 56 78**

Looks cleaner, right? But almost *none* of your PCs today use it natively — unless you're reading:

- Network packets (TCP/IP headers use big-endian)
- Older PowerPC binaries
- Foreign firmware formats (e.g., routers, smart devices)

🧠 *Mnemonic:*

"Big-endian puts the big stuff first."

📚 Real-World Analogy

Imagine you have a box labeled “4-digit PIN: 1234.”

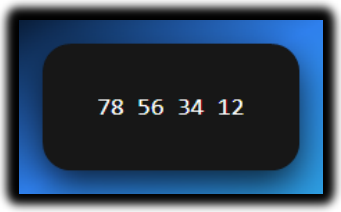
- Big-endian guy: opens the box and reads left to right: 1, 2, 3, 4 ✅
- Little-endian guy: opens the box and reads **right to left**: 4, 3, 2, 1 ❌ (unless he knows to flip it)

So, if someone drops a little-endian number on a big-endian parser = chaos. 😬

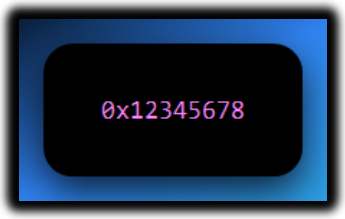
🕵️ Why This Matters in Reversing / Malware Analysis

1. Hex Dumps Look “Reversed”

When analyzing a binary or memory:



You must **reverse** it to understand what it *really* means:



If you forget? You’ll misinterpret pointers, constants, even jump addresses.

2. Parsing File Formats

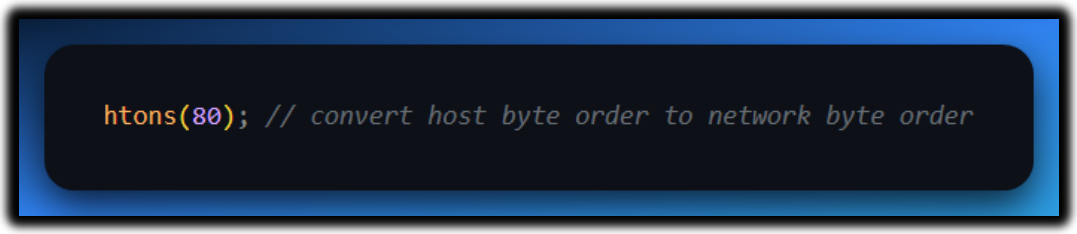
Think about parsing a BMP, PE header, ELF, ZIP, or MP3:

- The structure says: “next 4 bytes = offset”
- You read 40 00 00 00
- That’s 0x40 in little-endian… not 0x40000000!

3. Network Forensics

Network protocols use **big-endian** ("network byte order").
So if malware sends a packet like:

That might mean **port 80** (HTTP), not 0x5000. In C:



💡 **htonl, ntohl, htons, ntohs** = functions to flip bytes correctly.

4. Cross-Platform Hell

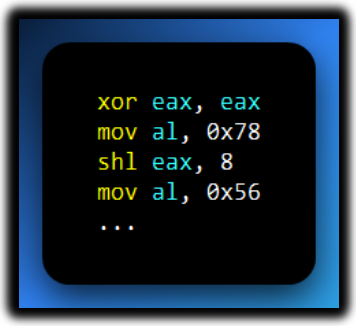
You're dealing with:

- An iOS app (uses ARM).
- A router firmware (MIPS).
- A PC binary (x86).

All of these may interpret the **same bytes** differently unless you respect their endianness. Binary parsing tools (like Binwalk) usually try to auto-detect, but you must know what's up.

⚔️ Bonus Fun: Obfuscation via Endianness

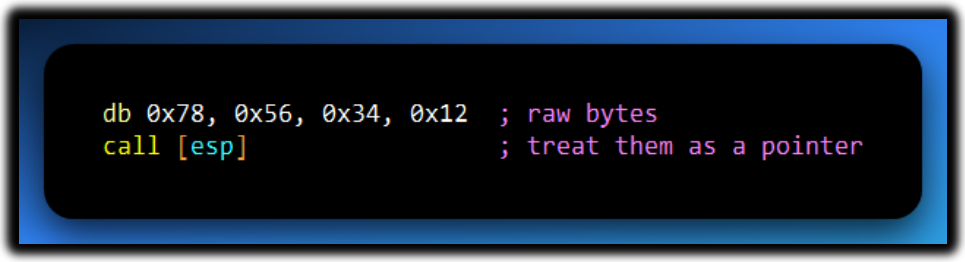
Some malware writers or packers intentionally **flip bytes** or **misalign fields** to throw off analysts e.g. a pointer to code might be stored as:



```
xor eax, eax
mov al, 0x78
shl eax, 8
mov al, 0x56
...
```

So, you get: 0x12345678 → but not in one go.

Or they do something like:

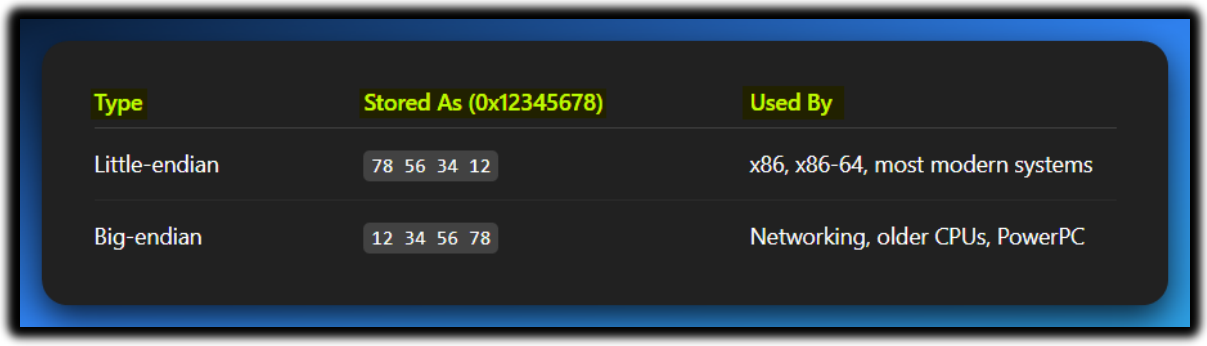


```
db 0x78, 0x56, 0x34, 0x12  ; raw bytes
call [esp]                 ; treat them as a pointer
```

Knowing endianness lets you spot this and go:

"Aha! This junk is a valid pointer in disguise."

🧠 TLDR Summary



Type	Stored As (0x12345678)	Used By
Little-endian	`78 56 34 12`	x86, x86-64, most modern systems
Big-endian	`12 34 56 78`	Networking, older CPUs, PowerPC

- **Always know** the target architecture.
- **Always flip** bytes if the dump looks weird.
- **Never assume** raw bytes = the final number.

This knowledge stacks FAST. You're not just learning addressing — you're building the instincts of a *binary surgeon*.

🧠🪄 THE HOLY TRINITY OF ADDRESSING: LOGICAL VS LINEAR VS PHYSICAL

⚙️ From Code to Execution:

🔧 Compilation Time (Before Running):

- You write: int x = 5;
- Compiler translates it into machine code.
- That machine code becomes part of an .exe or binary.
- The binary stores instructions + metadata like "I need space for variable x".

🚀 Runtime (When You Run the Program):

1. **OS Loader loads your program** into RAM (but not just anywhere).
2. The OS gives your program its own **virtual address space** (like a private sandbox).
3. The **CPU starts executing** the instructions using **logical (virtual) addresses** only.
4. The **MMU (Memory Management Unit)** kicks in and translates **logical → physical** addresses.

🔄 Behind the Scenes – Address Translation:

5. The program says: "I need memory for x."
6. The CPU generates a **logical address** (e.g. 0x7FF51234ABCD) for it.
7. The **MMU + Page Tables** look up where that maps to in **physical RAM** (e.g. 0x0012F000).
8. This is done through **paging** (modern systems), and maybe **segmentation** (rare in 64-bit mode).
9. You never see the physical address — only the **logical one** in your code/debugger.

🎮 While Running:

10. Your program reads/writes to x using the logical address.
11. The CPU + MMU **magically redirect** those reads/writes to the real RAM.
12. The OS ensures **memory protection**, so other apps can't touch your stuff.

🧼 When You Exit:

13. The OS unmaps your virtual address space.
14. All physical memory you used is **freed up** for other programs.

✅ Summary:

Write code → compile → load into RAM → virtual memory created → logical addresses used → paging maps them to physical RAM → program runs like it owns the place 🧠💥

Welcome to the **Matrix of Memory**. What you see as [my_variable] in assembly?

That ain't real. It's a *lie the OS and CPU tell you* to keep your code sane.

Here's the three layers of that lie:

1. Logical Address (aka Virtual Address): Your Personal Map

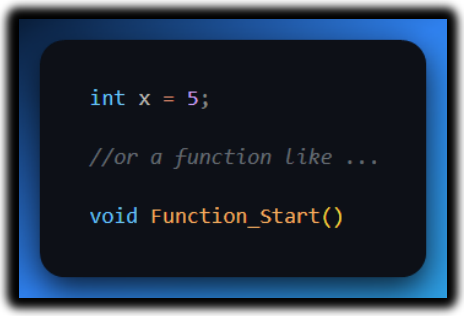
A **logical address** is indeed the address generated by the CPU when your program is running.

When you write something like **int x = 5;** in your C code, you're telling the compiler, "Hey, I need a spot in memory to store this integer x."

Think of the **logical address** as your **program's personal map or a wish list for memory**. 🗺️

This address is **virtual** because it's not the actual physical spot in RAM. It's more like a promise or a placeholder.

When you're writing code in C, Assembly, or Python, and you declare a variable like…

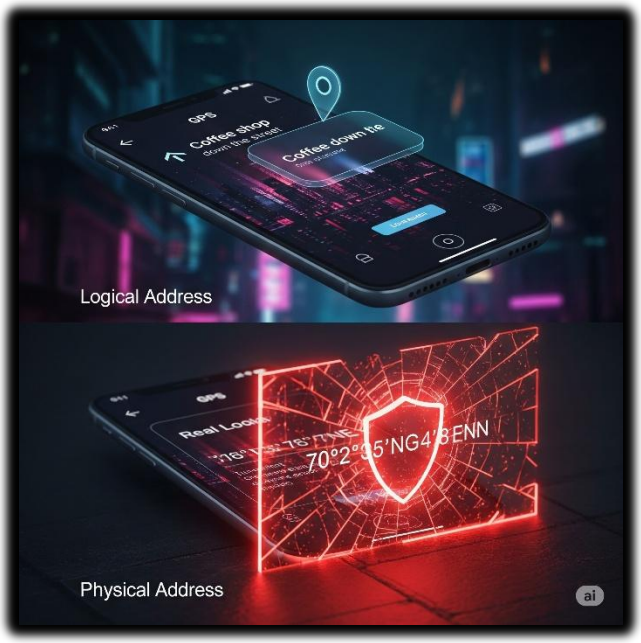


… you're essentially giving these pieces of your program **logical names** and assuming they'll reside at a specific, predictable "address" within your program's memory space.

Real-world example:

When you tell your friend, "Meet me at 'the coffee shop down the street'," you're giving them a **logical address**.

You don't know the exact street number or building ID yet, but you know the *concept* of where you want to go.



Your program thinks similarly: **"I need a variable here,"** or **"My function starts there."**

It's an address *relative to your program's perspective*.

ASM Code you write:

```
mov eax, [0x00401000] ; some variable or code
```

That 0x00401000? That's a **logical address**.

But like a street address on a letter, you don't actually deliver it.

You hand it to the system, and say **"make sure this reaches the correct house."**

👨‍💻 Used in:

- Assembly code.
- Decompiled functions in Ghidra.
- High-level program instructions.
- ptrace, read()/write() syscalls.
- Malware writing to its own heap/stack.

🧠 Pro Tip:

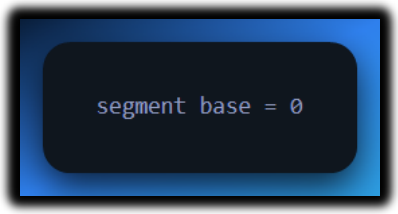
Every process sees a **clean, fake world** of memory starting at 0. That's the illusion virtual memory provides.

🧮 2. Linear Address (aka Post-Segmentation Address)

If you're in Protected Mode (which all modern OSes are), then the CPU first checks segmentation (CS, DS, SS, etc.), applies any segment base, and gives you a **linear address.**

✅ Most of the time:

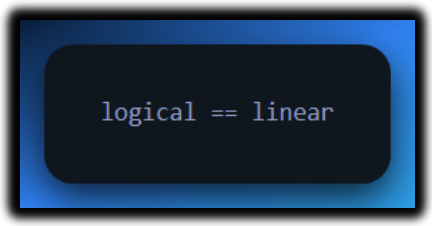
For modern 32-bit or 64-bit systems running in Protected Mode (especially with flat memory models like Linux/Windows user programs:



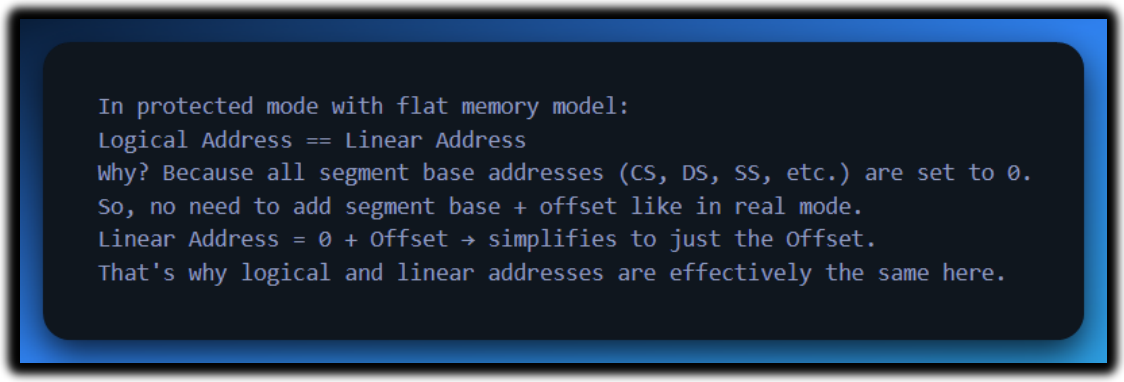
Segments can have custom base addresses in protected mode…BUT, in modern systems? We just **set all segment bases to 0** to simplify life.

The MMU doesn't go straight from logical to physical — instead, it first converts the **logical address to a linear address** using the segment's base, then translates the **linear address to a physical address** using **paging** (if enabled).

Again, we already know that…



… in protected mode because:



That's the modern twist:

99.9% of OSes configure segments with base = 0, limit = 4GB+, and basically ignore segmentation.

So, in most systems: **Logical Address ≈ Linear Address**

But if you're in:

- **Real mode** (bootloader/BIOS)
- **VM or sandbox**
- **Old malware or self-modifying code**
- **Or messing with LDT/GDT**

...then this difference **matters**. Like, *"you will misread a pointer and crash"* kind of matters.

📦 Analogy:

This is the **fully qualified address** the post office uses internally — like *"Banana Street, Zone 17, Nairobi, Kenya, 00505."*

👮 CPU uses this as an **intermediate step**:

- If paging is OFF → this = physical
- If paging is ON → this goes to MMU → becomes physical

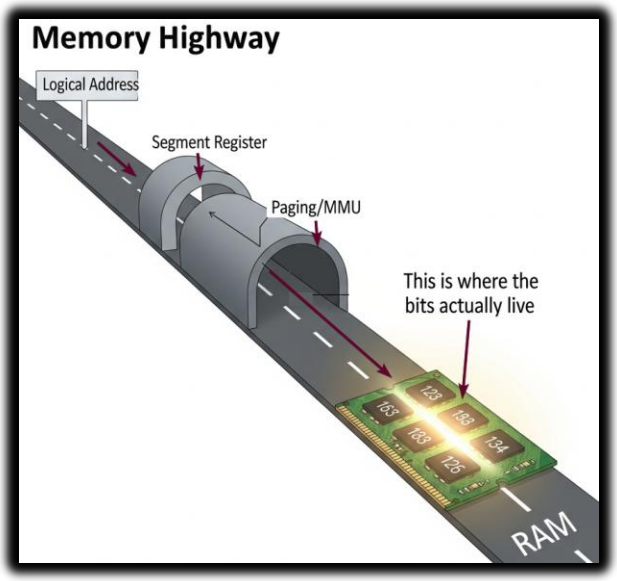
🧠 You only really *see* linear addresses when you're working in:

- Kernel exploit dev.
- BIOS/bootloaders.
- IDT/GDT/LDT manipulation.

🧱 3. Physical Address

This is where the bits **actually** sit on your RAM sticks. The real, raw deal.

It's what the **Memory Controller** sees after the MMU's done playing translator.



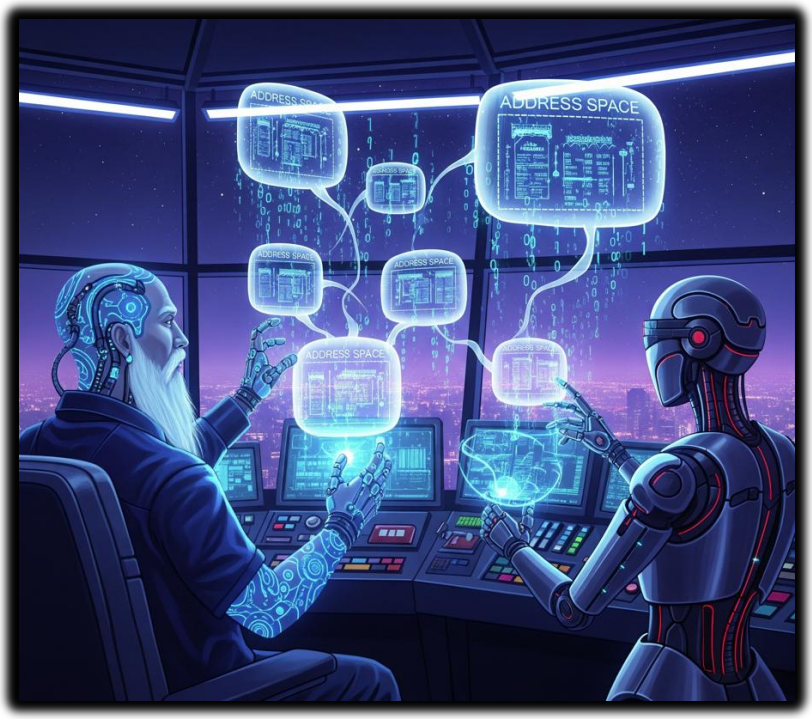
It's the **GPS coordinates** of the exact house. No ambiguity. If you gave this to a drone, it'd drop a missile precisely.

👾 Malware sometimes **fakes page table entries** to point logical addresses to custom physical regions (e.g. rootkits hiding in unused RAM).

💾 Address Space: The Digital Sandbox

Alright, picture this: your computer's memory isn't just one big, open field where all your programs run wild.

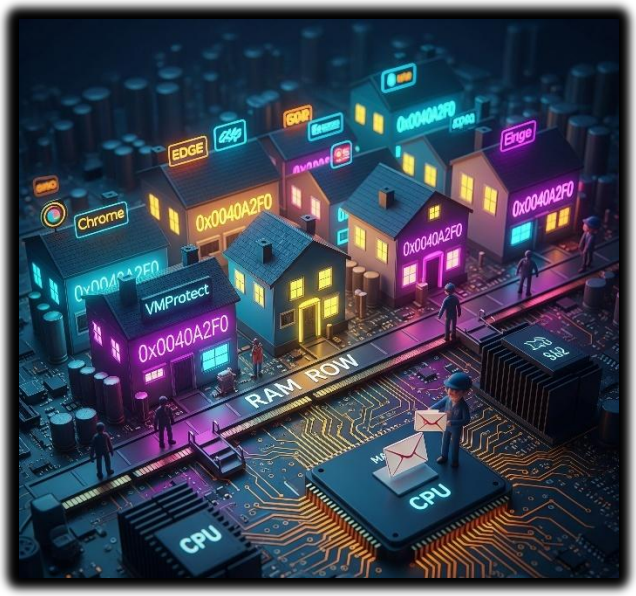
Nah, that'd be chaos! Instead, your **CPU** (the brain of your computer) and the **OS** (the air traffic controller) team up to create an incredible **illusion** for every program running. This illusion is called **Address Space**.



We can see that in this program the computer is giving each program an illusion, but each has its own space of operation.

Memory Addresses:

A unique identifier that specifies the location of a specific piece of data within a computer's memory, like a street address for a house.



Address Space as a Range:

An address space defines the set of all possible addresses that can be used within a system or a specific program to store and retrieve data.

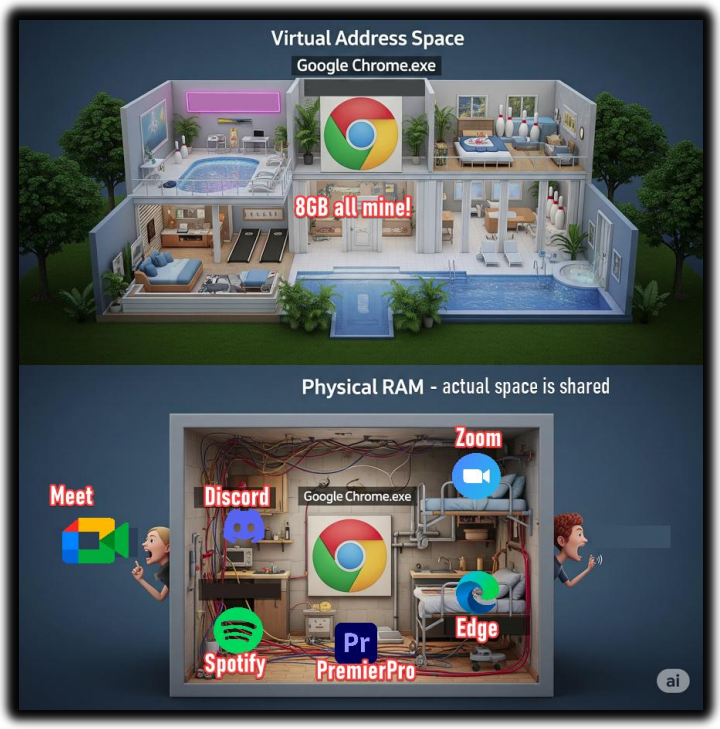
It's essentially the "map" that the system or program uses to navigate memory, allowing it to access and manage different locations for storing information.

It's crucial to get this straight: *Address Space is NOT your physical RAM.*

🎭 The Illusion of Address Space

Each program thinks it owns a massive private estate—like a Sims mansion with a pool, home theater, bowling alley, and a data jacuzzi. But in reality?

You're in a cozy 8GB RAM apartment with shared plumbing and maybe a noisy neighbor (aka another program).



🗺️ Address Space = Memory Map (But Not Physical RAM)

Think of address space as the **map** your program uses to navigate memory. But it's just the map, not the territory.

- You're seeing **logical/virtual addresses**.
- The actual RAM lives elsewhere, behind the curtain, like Oz.

This map isn't always the same across systems:

After this section, if you don't know OS at all, you might suffer. 😣 You can skip for now.

📚 Memory Mapping Styles Through the Ages:

- **Flat Addressing**: A simple street where every house is numbered sequentially.
- **Segmented Addressing**: Sections like neighborhoods; offsets are used instead of global coordinates.

CHAPTER 1: THE ILLUSION OF MEMORY - UNDERSTANDING ADDRESS SPACES

Before we plunge deeper into the intricate dance of memory management, let's establish a crystal-clear understanding of the fundamental concepts that govern how your programs interact with the system's memory.

This is the bedrock upon which all advanced memory techniques, exploits, and security measures are built.

This part is technical, no vibes, just raw notes. 😣 Sometimes, we need raw notes…

1.1 The CPU's Perspective: The Logical Address

🧠 Logical Address

A logical address is the address that the Central Processing Unit (CPU) generates during the execution of a program. Think of it as the address your code *thinks* it's talking to when you write instructions like **MOV EAX, [0x12345678]** in assembly, or when you allocate memory with **ptr = malloc (100)** in C.

It's entirely independent of the actual physical layout of the RAM chips on your motherboard. This is a critical distinction: your program operates in its own imagined world of memory, unconcerned with the underlying hardware specifics.

✅ **It is what your program inherently sees and requests.**

1.2 The OS's Masterpiece: Virtual Memory

🌌 Virtual Memory

Virtual memory is a sophisticated memory management technique, a collaborative effort between the Operating System (OS) and a dedicated hardware component called the **Memory Management Unit (MMU)**.

Its primary role is to create a seamless, *virtual address space* for each running process.

This is achieved by meticulously mapping the logical addresses generated by the CPU to the actual physical addresses in RAM (or even on disk) using complex data structures known as **page tables**.

This ingenious system allows for several powerful capabilities:

- **Address Space Isolation:** Each program lives in its own sandboxed memory environment, preventing one misbehaving application from corrupting or accessing the memory of another. This is the first line of defense against many common bugs and a cornerstone of system stability.
- **Overcommitting Memory:** The OS can promise more memory to running programs than is physically available. This works because most programs don't use all their allocated memory simultaneously. It's like a restaurant taking more reservations than it has tables, banking on the fact that not everyone shows up exactly on time.
- **Swapping to Disk (Paging):** When physical RAM becomes scarce, the OS can temporarily move less frequently used pages of memory from RAM to a designated area on the hard drive (known as the **swap file** or **pagefile**). When those pages are needed again, they are "paged back in" to RAM. This allows you to run applications that collectively require more memory than your physical RAM can hold.
- **Lazy Memory Allocation:** Memory isn't physically allocated until it's actually accessed. When a program mallocs a large chunk of memory, the OS might just update its internal tables, providing the *illusion* of allocated memory, but not actually reserving physical RAM until a byte within that allocated region is written to or read.
- **Memory-Mapped Files:** This powerful feature allows a file on disk to be treated as if it were part of a process's virtual address space. The OS transparently loads parts of the file into memory as needed, making file I/O operations feel like direct memory access.

✅ **It is how the OS, with the MMU's help, transforms the program's perceived logical address space into a functional reality.**

1.3 The Scope of Perception: Virtual/Logical Address Space

📏 Virtual Address Space / Logical Address Space

These terms refer to the total range of memory addresses that a process is theoretically allowed to use within a virtual memory system.

It's the entire "mansion layout" that your program believes it has at its disposal, irrespective of whether every single room (page) in that mansion is currently backed by real physical space.

For example:

- **On a 32-bit OS:** The address range spans from 0x00000000 to 0xFFFFFFFF, providing a massive 4 Gigabyte (GB) range.
- **On a 64-bit OS:** This expands dramatically, allowing for addresses up to 0xFFFFFFFFFFFFFFFF (16 Exabytes!). However, most modern 64-bit operating systems and CPUs typically only implement and use a smaller, albeit still enormous, 48-bit range (256 Terabytes), often due to practical limitations and current hardware capabilities.

✅ **This is the complete set of logical addresses your program can touch, operating under the grand illusion of having its very own, private memory realm.**

1.4 The Ground Truth: Physical Address

🧱 Physical Address

The physical address is the **true, concrete location** in your system's Random Access Memory (RAM) chips.

When data is swapped out, it could also be the **location on your hard drive** (within the pagefile).

This is what your hardware, specifically the memory controller and the CPU, actually deals with at the lowest, most fundamental level.

Programs never see or interact with physical addresses directly. This *vital translation* is exclusively handled by the MMU, acting as the system's tireless, vigilant gatekeeper between the logical world of software and the physical reality of hardware.

CHAPTER 2: THE GRAND DUALITY - LOGICAL VS. PHYSICAL

Alright, it's time to peel back the layers and truly understand the ingenious duality that powers every modern computing system. This is where the magic of virtual memory comes into its own, presenting an abstract, simplified world to your programs while efficiently managing the complex reality of physical hardware.

2.1 The Program's Dream: Logical Address Space

🎭 Logical Address Space = "What Your Program Thinks It Sees"

Imagine you're playing a massive open-world video game. Your character might wander into a grand, seemingly endless city, complete with towering skyscrapers and sprawling parks.

Your character (the program) genuinely believes this entire 10,000 square-foot metropolis exists for its sole enjoyment.

But behind the scenes, the game engine (CPU generating logical addresses) is a master of illusion.

It's cleverly reusing assets, dynamically loading textures only when you approach a specific building, and giving every single player the vivid illusion of having their own personal, *impossibly vast mansion.*

That seamless, *boundless illusion of memory*, the one your program perceives as its exclusive playground? That's your **logical address space**. It's the blueprint, the mental map that the program follows, oblivious to the deeper mechanics.

2.2 The Hardware's Reality: Physical Address Space

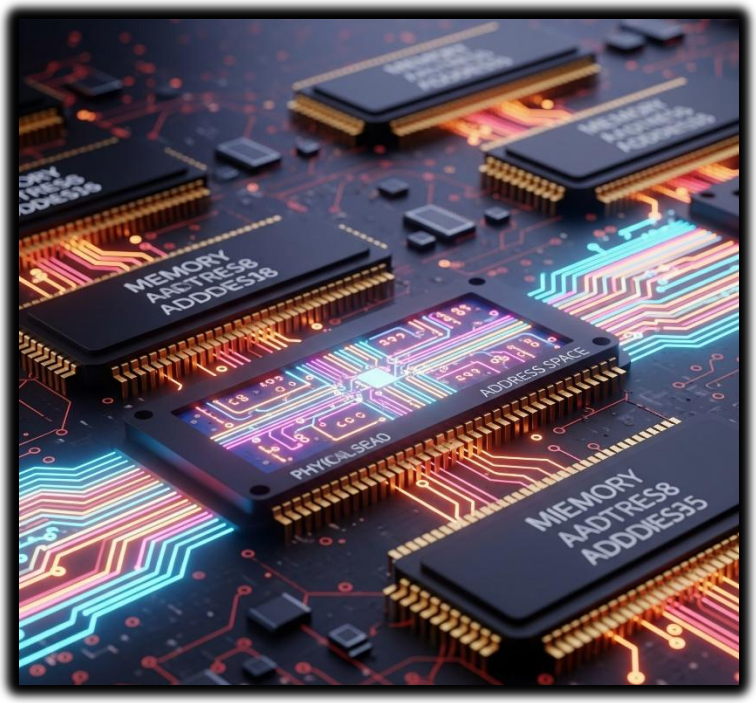
🦾 Physical Address Space

This is the tangible reality: the raw RAM chips soldered onto your motherboard, the solid-state drives, or the spinning platters of your hard drive. This is the actual, finite storage.

- **32-bit CPUs:** Historically, these could directly address a maximum of ~4GB of physical RAM. To access more, clever "hacks" like **Physical Address Extension (PAE)** were introduced, allowing a 32-bit CPU to *manage* more than 4GB by manipulating the page tables more extensively, but still accessing it in 4GB chunks.
- **64-bit CPUs:** These are built to handle truly colossal amounts of memory, capable of addressing terabytes or even petabytes of RAM, depending on the specific hardware implementation and the operating system's configuration.

But here's the kicker, and it's a crucial point:

🧠 **Your program *never* directly talks to physical addresses.** It sends out logical addresses, and then a series of translations occur before the data is actually fetched from or written to physical RAM.



2.3 The Master of Illusion: Virtual Memory

🌌 Virtual Memory = The Trick That Makes That Mansion Work

Virtual memory is the sophisticated system that brings this grand illusion to life. It's the benevolent, yet strict, landlord of your system, orchestrating the memory landscape for every tenant (process).

It says: *"Yo, Chrome, yeah, you can have 4GB. And Discord? Sure, you too. Visual Studio Code? Absolutely, here's your own 4GB. Just don't actually try to use all of it at once, and we're good."*

This "landlord" (the OS) keeps a colossal ledger, meticulously maintained in **page tables**, which contains the real mappings:

LOGICAL ADDRESS (PROGRAM'S VIEW)	PHYSICAL ADDRESS (HARDWARE'S REALITY)
0x401000 (Chrome's perceived address)	0x0001ABCD (Actual RAM location)
0x401000 (Discord's perceived address)	0x0002EFGH (Another actual RAM location)

This table allows multiple processes to *believe* they have the same logical address (e.g., 0x401000), but the MMU ensures they point to entirely different, isolated physical locations.

If Chrome, through a bug or malicious intent, somehow tries to "touch" memory that belongs to Discord's space (e.g. tries to access 0x401000 when the page table for Chrome doesn't map that logical address to an allowed physical frame), what happens?

💥 **Boom. Segmentation fault (Segfault).** The MMU instantly detects this unauthorized access and raises an exception. The OS, like a bouncer at an exclusive club, steps in and terminates the offending process. "Nah, bruh, that's not your table."

2.4 The Private Blueprint: Virtual Address Space

🚪 Virtual Address Space = The Private Mansion Layout

Every process gets its own pristine, idealized apartment layout within the mansion. This layout is standard and consistent:

- **Stack** on top (for local variables, function calls)
- **Heap** in the middle (for dynamically allocated memory like malloc or new)
- **Code (.text section)** at the bottom (for the executable instructions)
- And various **other sections** like .data, .bss, shared libraries, etc.

But really, underneath this beautiful illusion, they all reside in the same physical apartment complex.

The OS, **through the MMU and page tables**, masterfully remaps everything so that each process's "apartment" appears clean, contiguous, and isolated, even though the physical pages might be scattered across RAM, or even temporarily evicted to disk.

It's like residents in an apartment building **sharing common resources (RAM)** but having their own unique keys and spaces.

2.5 Unraveling the Nomenclature: Where Confusion Happens

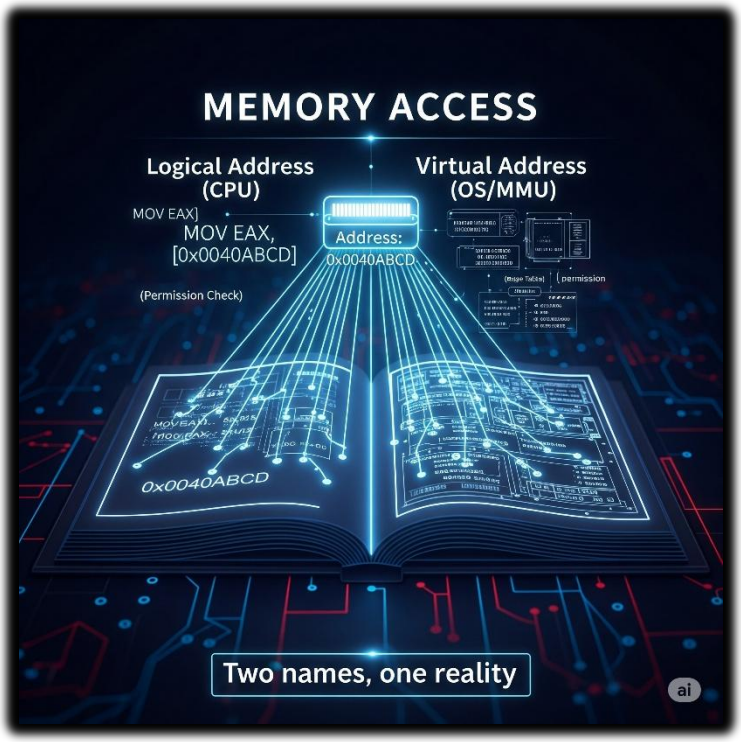
You might have previously thought that "logical address" and "virtual memory" were entirely distinct concepts. This is a common point of initial confusion!

But actually?

The logical address space is the virtual address space. They are two terms describing the *same concept* from slightly different perspectives, highlighting different aspects of the same underlying mechanism:

- **"Logical address"** emphasizes *what the CPU emits* when it needs to access memory. It's the address generated by the program's instructions.
- **"Virtual address"** emphasizes *what the OS and MMU interpret and manage* to perform the translation to physical memory. It's the abstract address that is subject to the virtual memory system's rules and mappings.

💡 And the virtual memory system is the *entire, comprehensive mechanism* that connects this logical/virtual address space to the physical address space.



It's the *complete infrastructure*, including the page tables, MMU, and OS algorithms, that makes the illusion work.

CHAPTER 3: THE MAGIC BEHIND THE CURTAIN: MMU AND EFFICIENCY

Let's quickly reinforce this concept with a relatable scenario, and then pull back the final curtain to see how the MMU truly operates and why this entire system is indispensable for modern computing, especially from a security and performance standpoint.

3.1 Quick Real-World Example: Lazy Allocation

📜 Quick Real-World Example:

Imagine you walk into a store and buy a gift card for 100MB of storage.

🧠 **You just got 100MB of logical address space.** The store (OS) has assigned you a section on its ledger (page table) indicating you "own" this space. You can refer to addresses within this 100MB range.

📦 **But physically? The actual USB stick (RAM) isn't fully used until you start putting files (data) onto it.** The store doesn't physically hand you a 100MB blank USB stick when you buy the gift card; it just notes your ownership.

💾 **If you only put 10MB of photos on that gift card, the remaining 90MB might never even get physically transferred to a real USB stick.** It's only when you actually *access* (write to or read from) a specific portion of that 100MB logical space that the system dedicates a physical block for it.

🧙‍♂️ **That's the true magic of virtual memory:** it's **lazy**, **overcommitted**, and beautifully **efficient**. It gives programs the freedom to ask for large chunks of memory without the immediate burden of physical allocation, conserving precious RAM.

3.2 The Master Magician: How the Trick Works

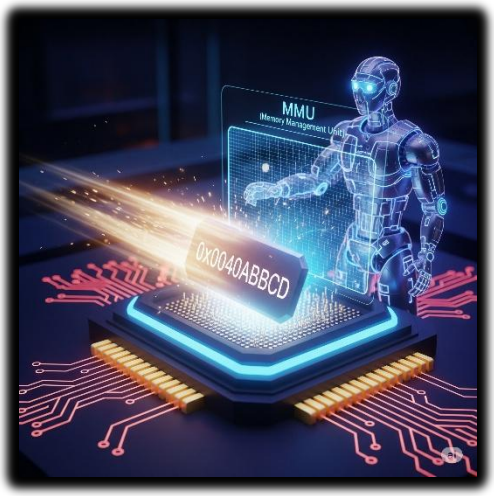
⚙️ How the Trick Works: The MMU in Action

The **Memory Management Unit (MMU)** and the Operating System are the true magicians behind the curtain, constantly performing address translation on every memory access.

When your program says: "Yo, give me the byte at 0x0040ABCD," it's not directly pointing to that spot in physical RAM. Instead, a precise sequence of events unfolds:

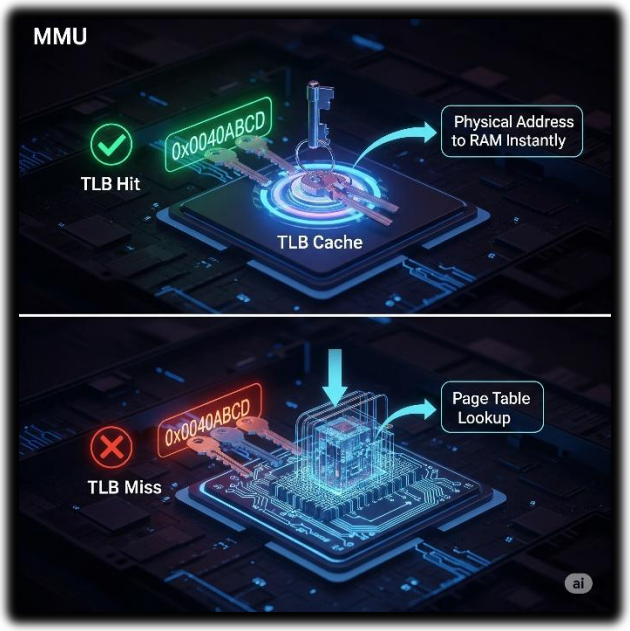
CPU Generates Logical Address: Your program executes an instruction (e.g., MOV EAX, [0x0040ABCD]). The CPU generates 0x0040ABCD as a *logical address*.

MMU Intercepts: This logical address is immediately intercepted by the MMU, a hardware component integrated into the CPU or its memory controller.



TLB Lookup (First Check): The MMU first checks its **Translation Lookaside Buffer (TLB)**. The TLB is a small, ultra-fast cache that stores recently used virtual-to-physical address translations.

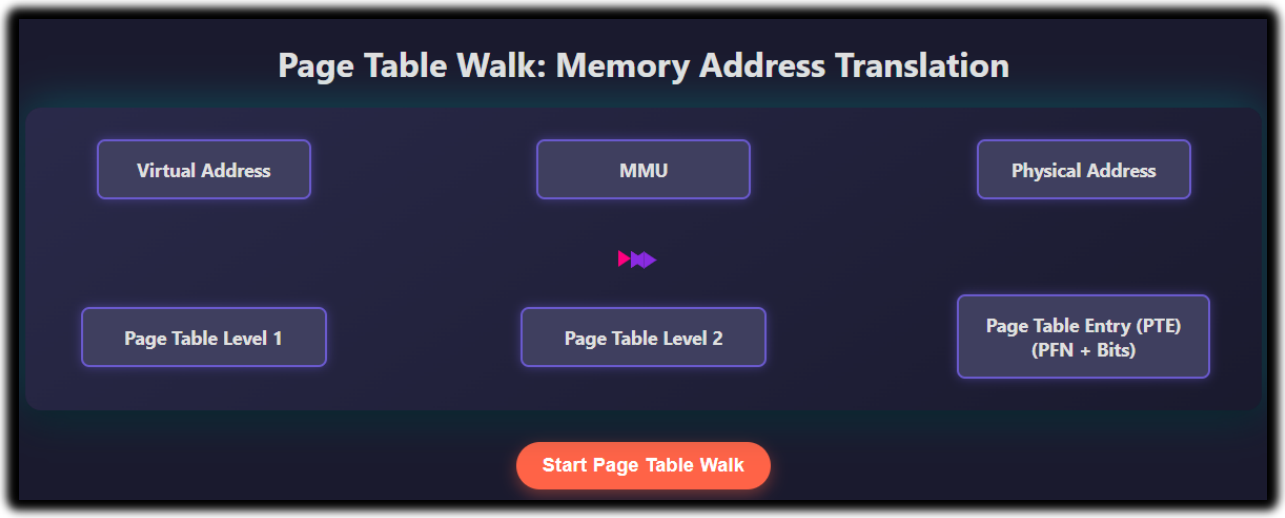
- **TLB Hit:** If the translation for 0x0040ABCD is found in the TLB (a "TLB hit"), the MMU quickly retrieves the corresponding physical address and its permissions. This is incredibly fast, akin to finding a frequently used key on a small keychain.
- **TLB Miss:** If the translation is *not* in the TLB (a "TLB miss"), the MMU proceeds to the next step.



Page Table Walk (The Ledger Check): For a TLB miss, the MMU initiates a "page table walk." This involves traversing a hierarchical data structure called the **page table**, which is stored in physical memory. The OS configures a special CPU register (e.g., CR3 on x86, or similar registers on ARM) to point to the base of the current process's page table.

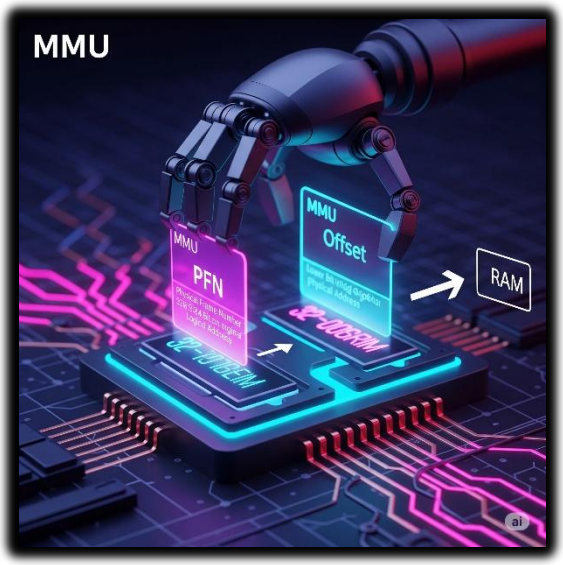
- The MMU breaks down the logical address into a **virtual page number (VPN)** and an **offset within the page**.
- It uses the VPN to index into the page table (or multiple levels of page tables) to find the corresponding **page table entry (PTE)**.
- Each PTE contains crucial information:
  - **The physical frame number (PFN)** where the virtual page resides in physical RAM.
  - **Permission bits** (Read, Write, Execute, User/Supervisor mode access).
  - **Present bit:** Indicates whether the page is currently in physical memory or has been swapped out to disk.
  - **Dirty bit:** Indicates if the page has been modified since it was loaded into RAM.
  - **Accessed bit:** Indicates if the page has been recently accessed.

Checkout the html for this part it has some JS:



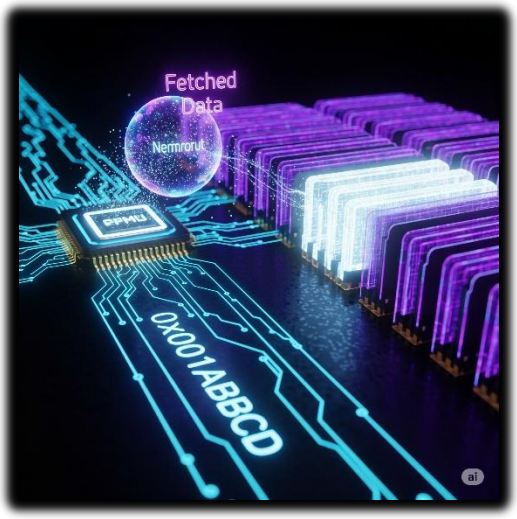
Permission Check: Once the MMU retrieves the PTE, it checks if the requested memory access (read, write, execute) is allowed for that page based on the permission bits. RWX lights all symbolize the process of page-level protection. MMU (Memory Management Unit) gatekeeping access to memory is crucial, see this image now…

Physical Address Construction: If the page is present in RAM and permissions are granted, the MMU combines the physical frame number (from the PTE) with the original offset (from the logical address) to construct the complete **physical address**.



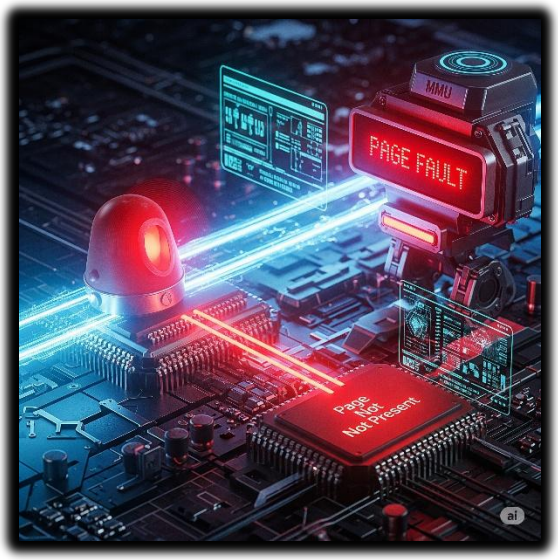
Malware often tries to manipulate page tables or exploit vulnerabilities in the MMU's translation process to gain unauthorized access to critical system memory or to hide its presence.

Data Fetch/Write: The CPU then uses this physical address to access the actual RAM location, fetching or writing the data.



Page Fault Handling (If Not Present): If the "present bit" in the PTE is clear (meaning the page is not in physical RAM but on disk), the MMU triggers a **page fault interrupt**. The OS's page fault handler then steps in. It will:

- Identify the missing page.
- Find an available physical frame (potentially evicting another less-used page to the swap file if RAM is full).
- Load the required page from disk into that physical frame.
- Update the page table entry to reflect the new physical location and set the "present bit."
- Resume the interrupted instruction, allowing the original memory access to complete successfully.



This entire process, involving TLB lookups and potential page table walks, happens with incredible speed, often **within a few clock cycles**, making the illusion completely transparent to the running program.

It's just like a magician (MMU) pulling a glowing memory page from a hat labeled 'Virtual Address 0x0040ABCD', with a hidden backstage showing a physical RAM chip and a meticulously updated page table ledger being used behind the scenes to make the magic happen.

CHAPTER 4: THE STRATEGIC IMPORTANCE - WHY THIS ILLUSION MATTERS

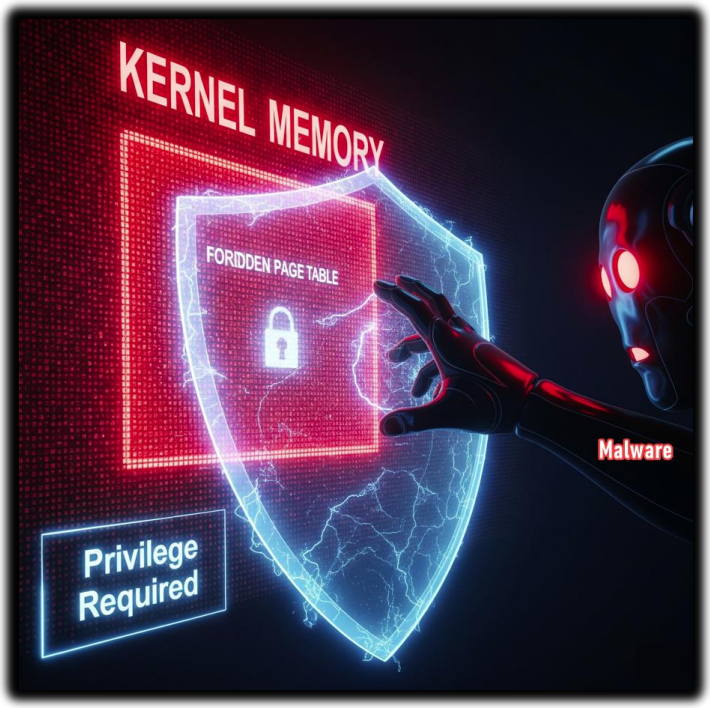
For anyone diving into reverse engineering, malware analysis, or operating system internals, understanding this memory illusion is not just academic; it's absolutely critical. It's the foundation of modern system security, stability, and efficiency.

🔐 4.1 Security: Fort Knox for Your Programs

Imagine each program running in its own secure, private apartment within a larger building. This is **address-space isolation.** Every program gets its own dedicated chunk of memory, separate from all others. Peeking isn't allowed. Accidental bugs can't escalate systemwide. Crashes and malware can't affect the others. In reversing, your work is finding these loopholes that cause crashes and violations.

📦 4.2 Multitasking: A Symphony of Processes

Modern OSes use virtual memory to let many programs run at once without stepping on each other. Even if Chrome and Photoshop both load code at the same logical address (like 0x400000), their page tables map that address to different physical RAM locations.

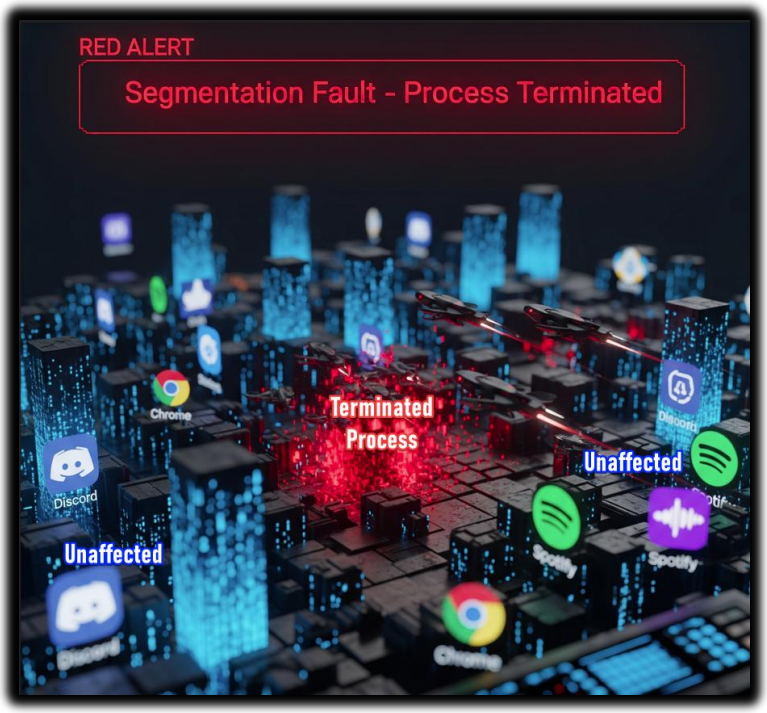


This keeps their memory totally separate. Malware can't poke into other processes or the kernel unless it exploits the OS to hijack those mappings — which is exactly what rootkits and kernel exploits are designed to do. 😈

🛡️ 4.3 Crash Containment: Fireproof Walls

If one program experiences a fatal error, such as trying to access an invalid memory address or dividing by zero, the OS can simply terminate that single, offending process.

The *virtual memory system*, by **isolating its memory space**, prevents the crash from cascading and bringing down the entire system.



It's like having **fireproof walls** between apartments. A fire (crash) in one unit doesn't necessarily spread and burn down the entire building.

The OS can contain the damage and extinguish the fire, allowing other processes to continue running undisturbed.

📦 4.4 Portability: Code That Works Anywhere

You don't need to know where your code actually lands in RAM — that's the OS + MMU's problem.

As a dev (or malware author 😏), you just write your code assuming a clean layout, like **"put my code at 0x00400000,"** and everything works. Behind the scenes, the system remaps it to wherever there's space in physical memory.

This image visually represents how a malware like **payload.bin** can effectively execute across different computer systems, regardless of their varying RAM configurations, due to the consistent logical mapping of virtual addresses.

It's like **unpacking your stuff in a new apartment** — same furniture, same layout, new building. You don't care how the plumbing works or where the pipes go — the building crew (OS + MMU) makes sure everything functions like you never left.

This **abstraction** makes your code portable across machines, even if their physical RAM looks totally different under the hood.

TLDR: What's What?

The essay is finally over, if you made it here, you're my legend.

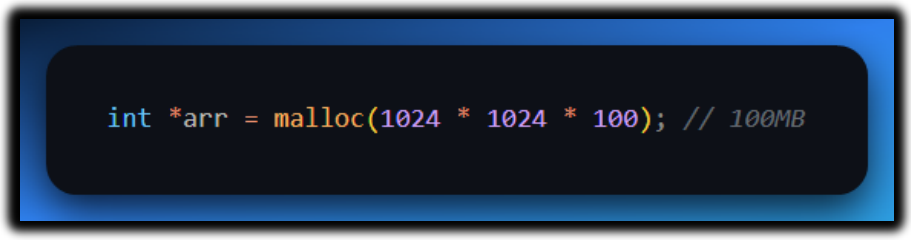
Your one step closer to being an assembly god. Keep going my guy/girl! 🥰

Term	Description
Logical Address	The address generated by your program (what your code uses)
Virtual Memory	The OS/MMU trick that maps logical → physical; enables isolation, paging, etc.
Virtual Address Space	The range of addresses available to your program, made possible by virtual memory
Physical Address	The actual RAM location holding the data

💡 And the **virtual memory system** is what connects logical ↔ physical.

Layer	What's Happening
You (Program)	"I need address 0x404000 for my array."
CPU	"Gotcha — that's a logical address."
MMU + OS	"Let's check the page tables... ah, that maps to physical RAM at 0x00F0A000."
RAM	"Here's your data."

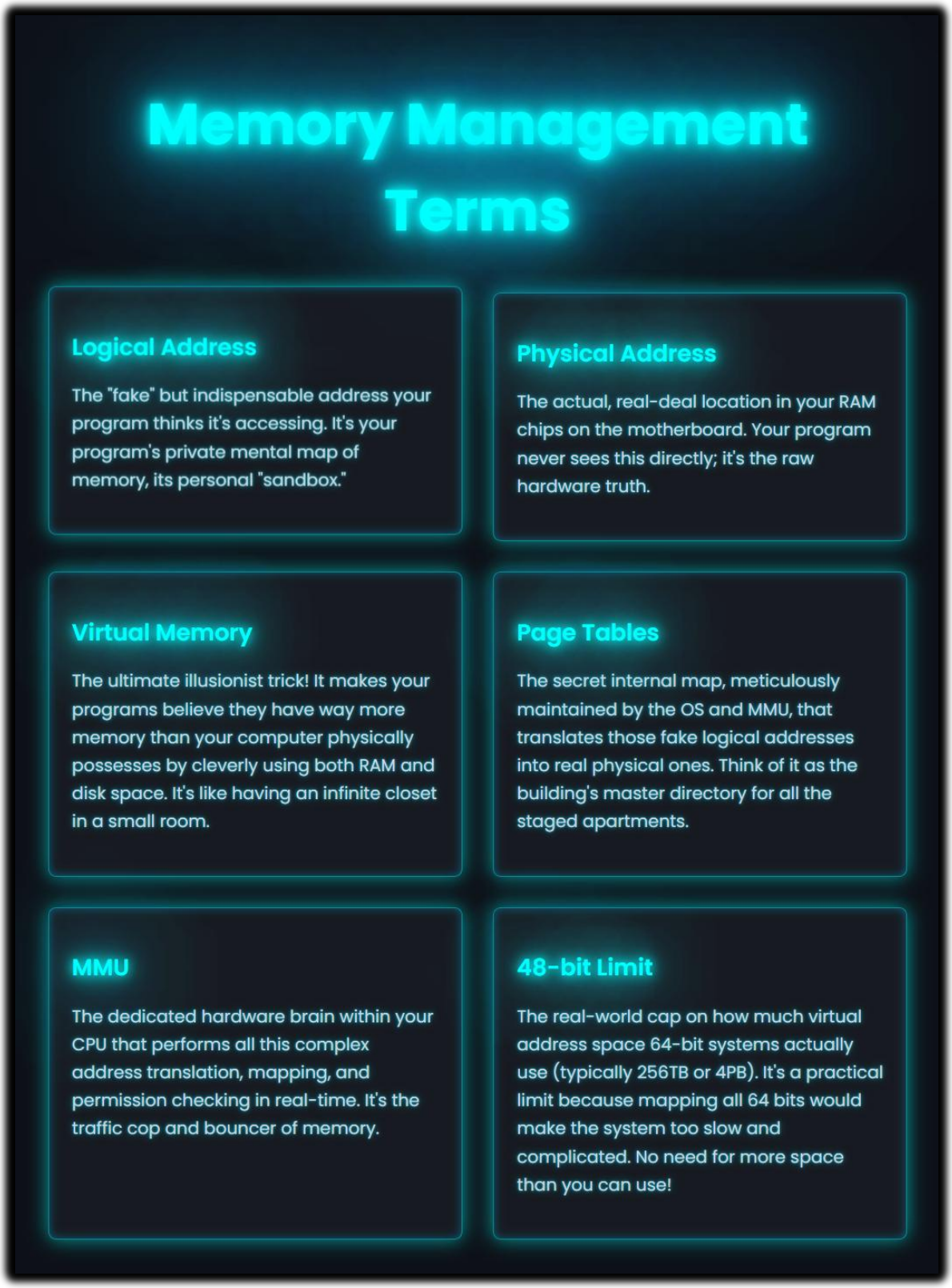
📄 Quick Real-World Example:



🧠 You just got 100MB of logical address space.
📦 But physically? The RAM isn't used until you *touch* the memory.
💾 If you don't access all of it, it might never even get loaded into RAM.
🧙‍♂️ That's the magic of virtual memory: lazy, overcommitted, and beautifully efficient.

We have an image in the next page... the book's not over yet.

TLDR: The Hacker's Quick Refresher



💣 Attack Surface for Malware & Reverse Engineers

🔐 **Manipulating Page Tables:** Rootkits can hide themselves by mapping their memory as "unmapped" to tools like ps, top, or memdump.

🛑 **Anti-Debugging via Memory Layout:** Malware may check if certain addresses are mapped to detect debuggers or sandboxes.

💉 **Android Reverse Engineering:**

- SELinux policies can stop access to kernel pages.
- mmap() tricks can obfuscate payload loading.
- Native libraries (via NDK) use virtual memory like crazy.

🧠 **You (the analyst) must:**

- Know which layer you're looking at.
- Translate logical ↔ physical ↔ file offset when needed.
- Understand how malware might *trick* tools like IDA by manipulating memory maps.

🔧 OPCODE vs OPERAND – The CPU's To-Do List 📃

Let's keep it simple to start:

⚡ Opcode = What to do

Think of it like the *verb* in a sentence. It's the **operation code** – a small binary value that tells the CPU *what instruction to execute*. Examples:

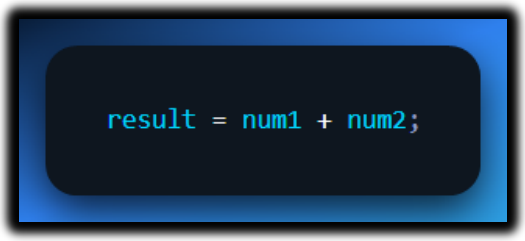
- ADD – Add values
- MOV – Move data
- JMP – Jump to a new address
- INT – Interrupt

💻 Operand = What to do it on

This is the **data, register, or memory location** the opcode will work on. It can be:

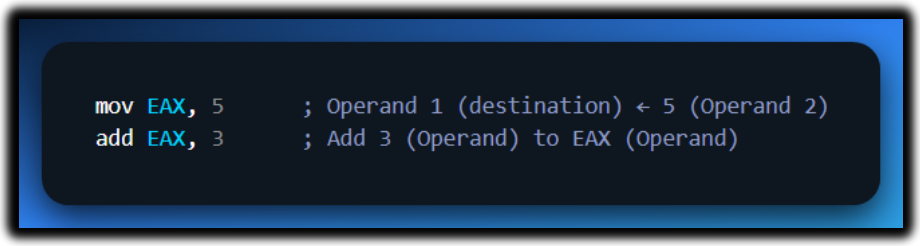
- A literal number (e.g. 5)
- A register (e.g. EAX)
- A memory address (e.g. [0x123456])

🔬 A Real-Life Analogy (Gen Z Edition):



- + is the **opcode** → "Add stuff"
- num1, num2 are the **operands** → "Stuff to add"

Now in **Assembly**:



Here's what's happening:

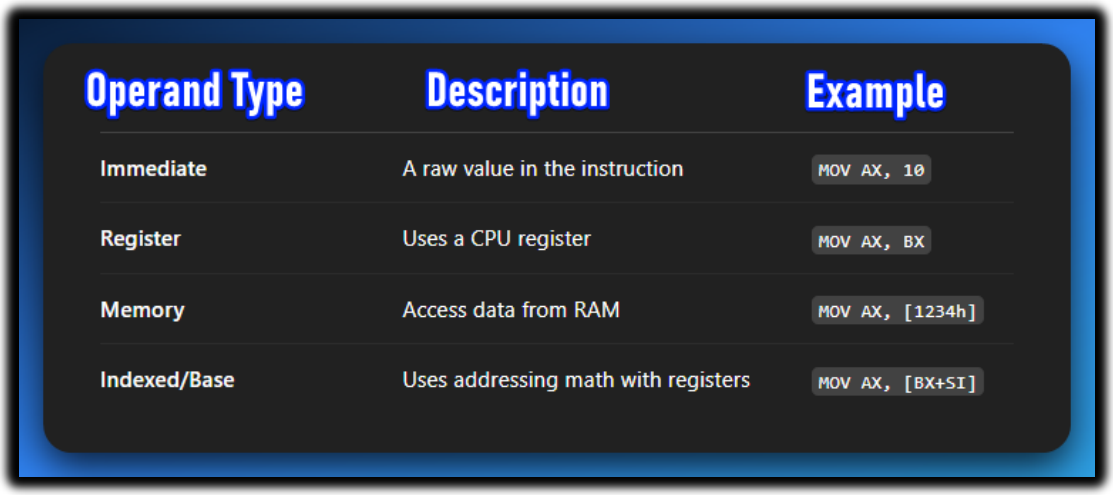
- MOV is the **opcode** (it says: "move data")
- EAX and 5 are **operands.**
- ADD is another opcode (says: "do math")

So, the instruction **ADD EAX, 3** is:

"Hey CPU, take the value in EAX and add 3 to it."

🧠 Types of Operands

Let’s go deeper — operands come in flavors. Here’s how the CPU sees 'em:



Operand Type	Description	Example
Immediate	A raw value in the instruction	`MOV AX, 10`
Register	Uses a CPU register	`MOV AX, BX`
Memory	Access data from RAM	`MOV AX, [1234h]`
Indexed/Base	Uses addressing math with registers	`MOV AX, [BX+SI]`

Modern CPUs use **effective address calculation** to resolve complex operands. Think of it like pointers on steroids.

🤯 The CPU's Inner Language: Opcodes and Operands Demystified

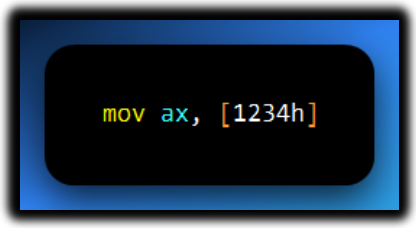
Think of it like this: your CPU is a super-fast chef 👨‍🍳.

- ### Opcode (Operation Code) = The Recipe Step 📜
  - This is *what* the chef needs to do. "Chop," "Mix," "Bake." Each opcode is a unique instruction the CPU understands.
  - **Example:** ADD (add), MOV (move), JMP (jump). These are **verbs** for the CPU.
- ### Operand = The Ingredients / Tools 🥕🔪
  - This is *what* the chef performs the action *on*, or *with*. "Chop carrots," "Mix batter in bowl," "Bake cake at 350 degrees."
  - **Example:** In MOV EAX, 5, EAX is a **register** (a small, fast storage location inside the CPU), and 5 is a **literal number**. EAX is where we put the 5.
  - **Analogy:** If MOV is "Put," then EAX is "into this box" and 5 is "this number."

Key Takeaway: An **instruction** is the complete sentence: **Opcode + Operand(s)**. It tells the CPU exactly what to do and where to do it.

🛠️ Anatomy of an Instruction: Breaking It Down

Let's use an example and visualize it. Consider the instruction:



This instruction tells the CPU:

"Move the 16-bit value located at memory address 1234h into the AX register."

Here's how the CPU actually sees it in its raw machine code (binary):

Opcode (A1): 🎨

- This A1 is the specific binary code (in hexadecimal) that represents the MOV AX, moffs16 instruction. moffs16 means "memory offset 16-bit."
- Each unique operation the CPU can perform has its own specific **opcode byte(s)**. These are hardcoded into the CPU's design.

Operands (34 12): 🔢

- The 34 12 represents the memory address 1234h. Why 34 12 instead of 12 34? Because of **Little Endian** byte order, which is common in x86 CPUs. The **least significant byte** comes first. So 0x1234 is stored as 0x34 then 0x12.
- These bytes are the data or address that the A1 (MOV) opcode will operate on.

The CPU's Process:

When the CPU fetches A1 34 12:

1. It reads A1 and recognizes it as the "Move to AX from memory" instruction.
2. It then knows that the *next two bytes* (34 12) represent the memory address it needs to read from.
3. It fetches the data from 1234h in memory and places it into the AX register. Boom!

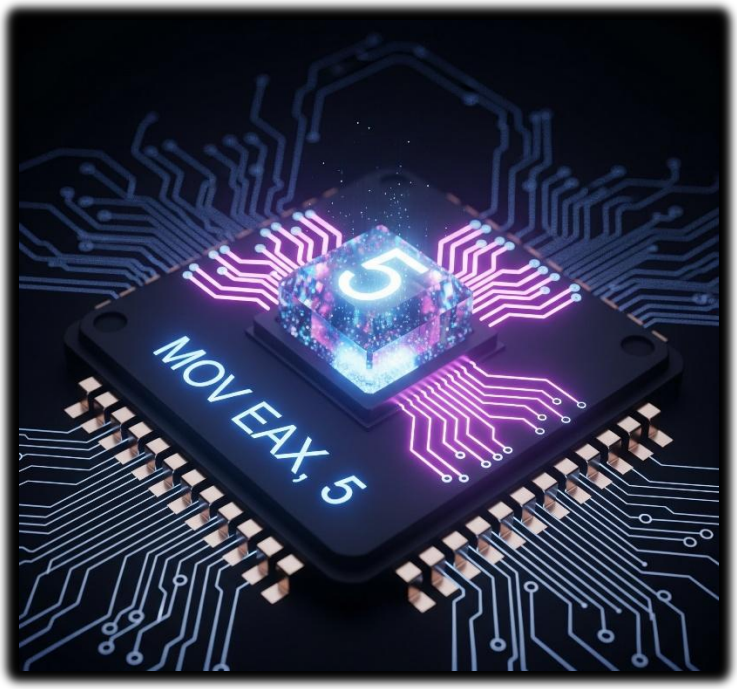
🧩 Types of Operands: How the CPU Gets Its "Stuff"

Operands tell the CPU *where* to find the data. Let's make this super clear:

Immediate Addressing/Immediate Operand (The "Raw Value") 📦

What it is: The actual data value is embedded directly *within the instruction itself*.

Example: MOV EAX, 5



Explanation: The number 5 is literally part of the machine code instruction. The CPU doesn't need to look anywhere else for 5.

Analogy: The recipe says "add 2 cups of sugar." The "2 cups of sugar" is right there in the instruction.

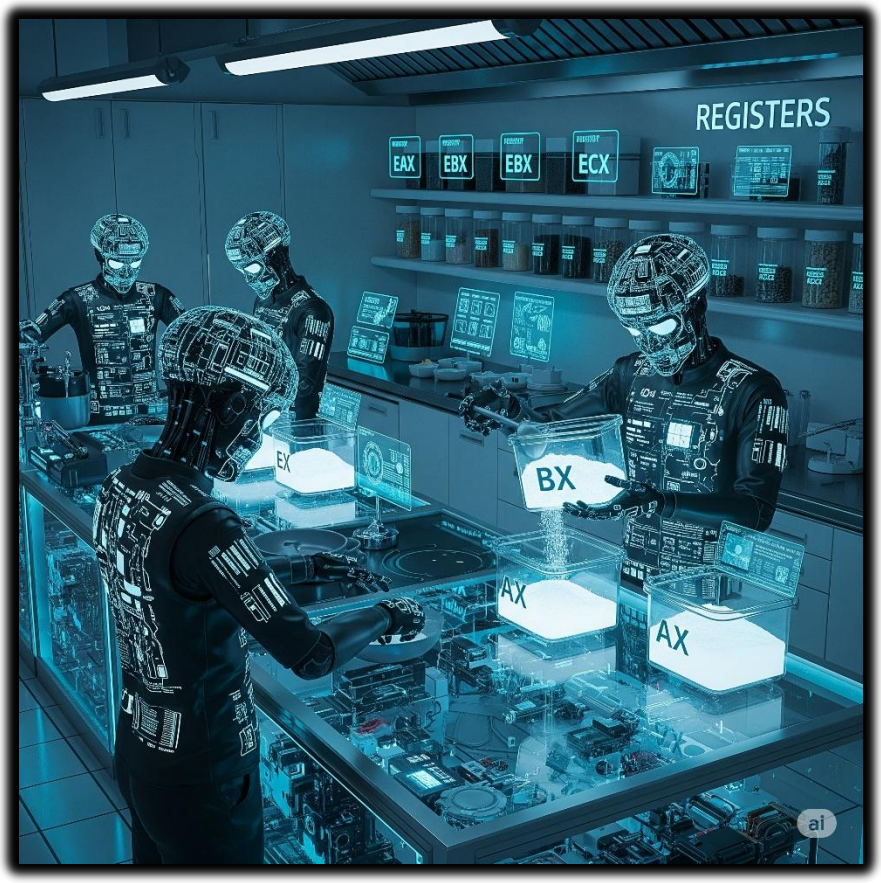
Register Addressing/Register Operand (The "Named Box") 📦

What it is: The data is stored in one of the CPU's internal registers (like EAX, EBX, ECX, EDX).

Example: MOV AX, BX

Explanation: The CPU is told to take the value *from* the BX register and put it *into* the AX register. Registers are super-fast, because they're right on the CPU chip.

Analogy: The recipe says "take the flour from the large container." The "large container" (register) is a known, quick place to get the ingredient.



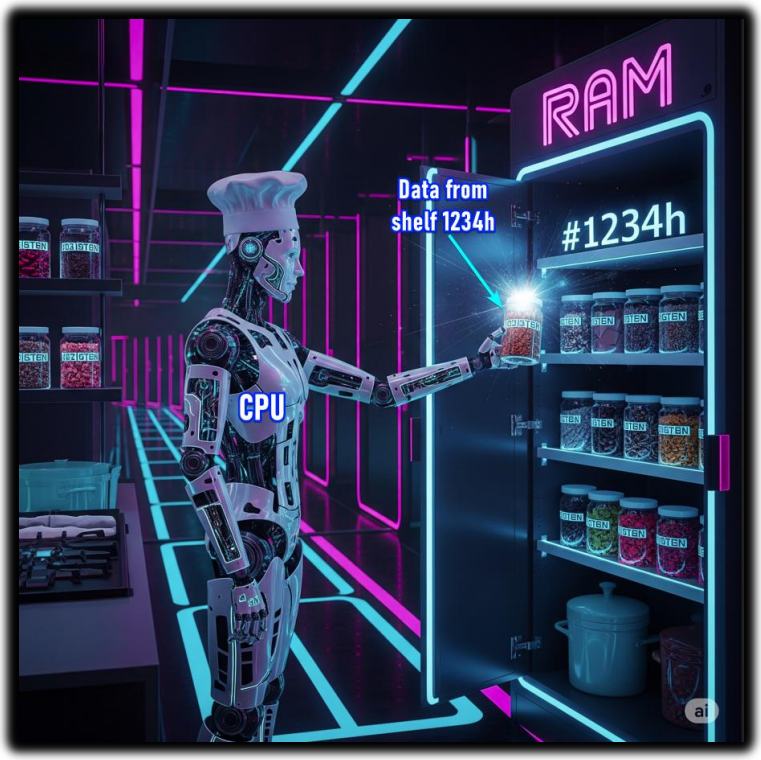
Direct (or Absolute) Memory Addressing/Memory Operand/Storage Unit

What it is: The data is located in your computer's main memory (RAM).

Example: MOV AX, [1234h]

Explanation: The [1234h] tells the CPU to go to memory address 1234h and fetch the data stored there. Memory access is slower than register access.

Analogy: The recipe says "get the special spice from the pantry shelf #1234h." The CPU has to go out to the "pantry" (RAM) to find the "spice."

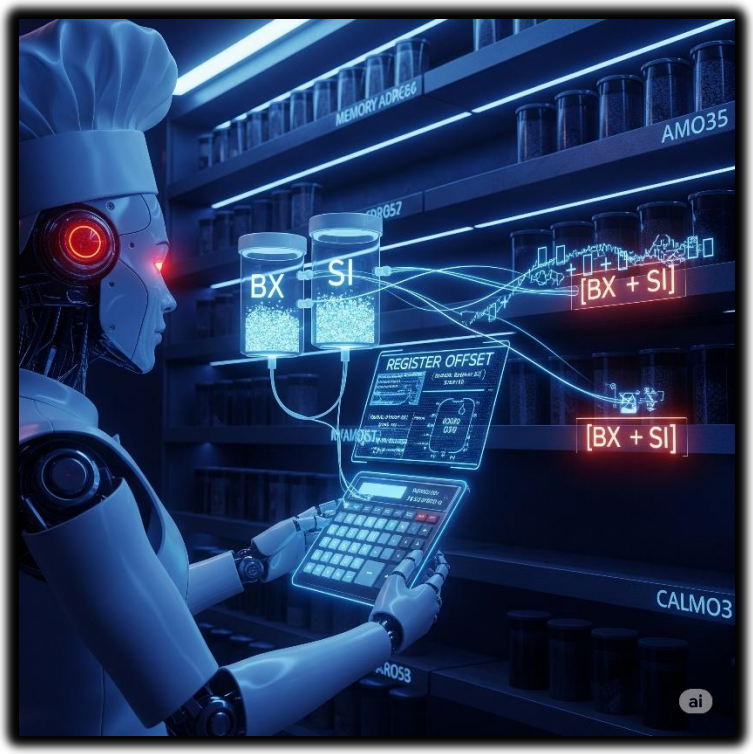


Register Indirect Addressing/ Based-Indexed/ Effective Addressing/ Indexed/ Base/ Scaled Operands/ Smart Address Calculation 🧠

What it is: These are more complex memory addresses. The CPU calculates the *final* memory address by combining values from multiple registers and/or a displacement. This is powerful for arrays, structures, and dynamic memory access.

Example: MOV AX, [BX+SI]

Explanation: The CPU adds the current value in register BX to the value in register SI to get the *actual* memory address to read from. [BX+SI] is the "effective address."



Analogy: *"Get the ingredient from the shelf that's X steps from the main cabinet, and then Y steps to the right."* The CPU calculates X+Y to find the exact spot.

🕵️ Mini Disassembler Challenge!

Let's test your understanding. Based on what we've discussed:

Given the following raw hexadecimal bytes, what do you think they represent in x86 assembly?

1. B8 05 00
2. 8B C3
3. 89 C8
4. CD 21

(Don't worry about being perfect, just try to recognize the patterns we've discussed.)

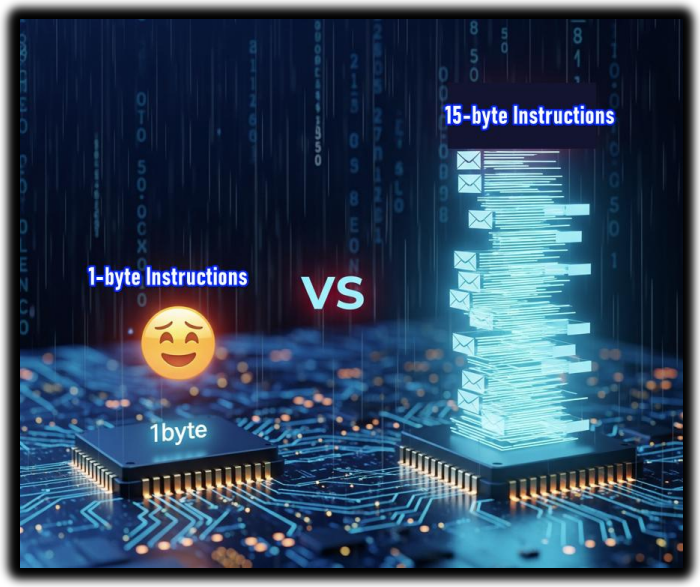
🚀 Deeper Dive: Instruction Encoding & Opcode Size

🤖 WTF Is This About?

This is about how the CPU **reads instructions** — and how each one is made up of a bunch of **bytes** behind the scenes.

🧠 KEY IDEA:

In x86, **instructions don't all take the same number of bytes**. Some are short like a text emoji (1 byte), others are long like an email rant (up to 15 bytes). That's what **"variable-length"** means.



🚀 Now Let's Break It Down Clean:

1. Variable-Length Instructions

In x86:

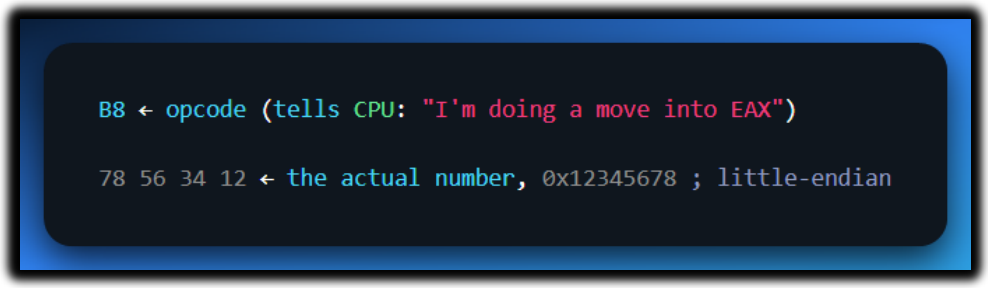
Every instruction is made of bytes.

Some are tiny.

👉 **NOP (do nothing) = just 1 byte: 0x90**

Others are chunky.

👉 **MOV EAX, 0x12345678 = 5 bytes:**



So even though you write 1 line in assembly… it could take 1 byte or 15 bytes in machine code.

2. Prefixes: The Hidden Tweaks

Sometimes the instruction starts with a special **prefix byte** — like an accessory that tells the CPU,
"Yo, treat this like a 16-bit move, not 32-bit."

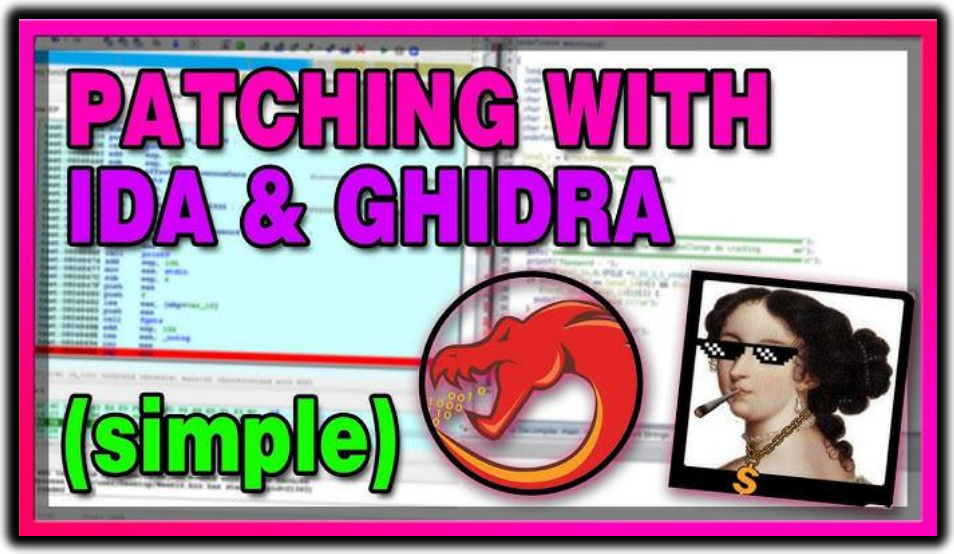
- 🔧 Example: 0x66 is the **"operand-size override"** prefix.
  → It's like saying: *"I know we're in 32-bit mode, but let's act 16-bit just this once."*

These prefixes come **before** the main instruction byte(s) and modify how it's interpreted.

🤯 Why This Matters?

Because:

- You can't just say “1 line of ASM = 1 byte.” That’s not true.
- When reverse engineering, you need to decode the bytes correctly.
- Tools like disassemblers (IDA, Ghidra) need to read all those prefixes, opcodes, operands, etc., or you’ll misinterpret instructions.



Let’s do **registers** in the next document…

By orders of the peaky blinders! 😤

Specifically, IDA SHELBY…