Let's separate the CPU operation modes to this document.

🧠 CPU MODES EXPLAINED:

PART 1 – REAL MODE (THE WILD WEST OF COMPUTING)

⚙️ What Even Is a CPU Mode?

Before we get deep, here's the vibe:

A CPU "mode" is like the **operating style** of the processor.
It defines what the CPU is allowed to do:

- *How much memory it can touch.*
- *Whether it has access control/security.*
- *What kind of instructions and registers it can use.*

It's like your CPU switching between "beginner," "intermediate," and "pro" modes depending on what it's trying to run.

🧱 Real Mode: The 16-Bit Legacy Arena

🕰️ This is the **original** mode of the x86 CPU family. Born with the 8086 processor (1978), and every modern x86 CPU **still starts** in Real Mode when powered on — even your Core i9 or Ryzen 9.

🔍 Key Traits of Real Mode:

💾 16-Bit Real-Address Mode: Key Features

FEATURE	EXPLANATION
16-bit registers	You mostly use registers like AX, BX, CX, DX, SI, DI, SP, BP. These registers can hold 16 bits (2 bytes) of data.
1MB memory limit	The CPU can only directly address up to 1 megabyte (MB) of memory, from address 00000h to FFFFFh. This is also referred to as a 20-bit address space ($2^{20} = 1MB$).
No memory protection	Any program running in this mode can write to any part of memory, including memory used by other programs or the operating system itself. There is no safety mechanism to prevent programs from interfering with each other.
No multitasking	You typically run one program at a time. That program has complete control over the CPU and all system resources until it finishes or explicitly yields control.
Direct hardware access	Programs can read from and write to hardware devices like the keyboard, screen, or disk controller directly, without needing the operating system's permission or mediation.

💉 Addressing Style

Real Mode uses **Segment:Offset** addressing. It breaks up memory access like this:

```
mov ax, 0x1234   ; Segment
mov bx, 0x5678   ; Offset
; Physical address = (Segment × 16) + Offset = 0x12340 + 0x5678
```

🧠 Basically: **Segment × 16 (or left-shift 4 bits)** + **Offset.** That's how Real Mode squeezes 20-bit memory access out of 16-bit registers.

I know you didn't get anything, lets revisit this madness about 16-bit real mode. Okay, I already made the image and html for you to go read, this is too hard to just write it out here.

You'll not meet this stuff a lot, this is just for the old systems, for understanding.

🎮 Where You'll Still See Real Mode in Action:

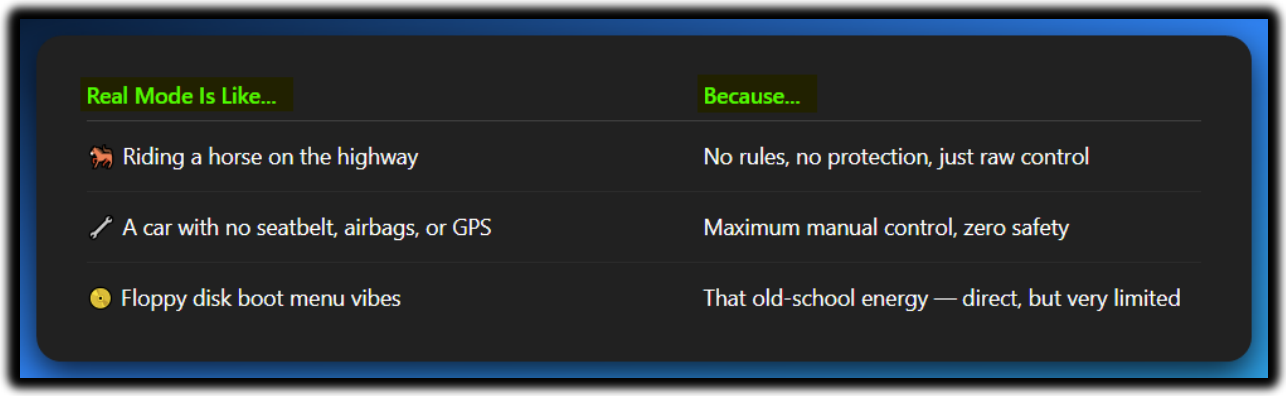
Use Case	Why It Uses Real Mode
BIOS	It's the first thing your CPU runs. BIOS firmware is old-school and still coded for Real Mode to stay compatible with all systems.
Bootloaders	The first stage of your OS loader (like GRUB or BOOTMGR) starts in 16-bit real mode because the CPU boots there by default.
MS-DOS / DOSBox	Games from the 80s–90s (like Prince of Persia, Doom 1, etc.) were all written for Real Mode. DOSBox emulates this environment.
Embedded Systems	Tiny microcontrollers (washing machines, basic control systems) don't need advanced modes — just raw direct control.

🚫 Why Modern OSes Abandoned Real Mode:

Reason	Why It's a Problem
❌ **No memory protection**	Any program can accidentally (or maliciously) overwrite system memory. Major security risk.
🧠 **Can't multitask**	No scheduling or isolation — one program at a time.
🧱 **Too little memory**	1MB just isn't enough for modern applications or even drivers.
😵 **Too manual**	You have to manage everything: memory, stack, device IO, etc. No OS to help.

⚠️ That's why modern OSes like Windows 10/11 or modern Linux **don't allow 16-bit Real Mode programs** to run natively anymore. You need emulators or virtual machines.

🧠 Analogy Time:



Real Mode Is Like...	Because...
🐎 Riding a horse on the highway	No rules, no protection, just raw control
🔧 A car with no seatbelt, airbags, or GPS	Maximum manual control, zero safety
💿 Floppy disk boot menu vibes	That old-school energy — direct, but very limited

📜 Summary: Real Mode

- ✅ 16-bit legacy mode — max 1MB memory
- ✅ No protection, no multitasking
- ✅ Still used in BIOS, bootloaders, and tiny embedded systems
- ❌ Not suitable for modern multitasking OSes
- ❌ Needs emulation on modern 64-bit systems

⚡⚡ *Let's go to 32-bit. Remember, we're using the 007 html file, just expanding that one for maximum impact and understanding.*

🛡️ 32-Bit Protected Mode – The Secure Apartment Building of Computing

🧠 What is Protected Mode?

Protected Mode was a **game-changer** when it dropped with the Intel **80386** processor. This mode introduced *true multitasking, memory protection, and virtual memory* — which are **core features** of every modern OS.

Imagine going from a **wild jungle (Real Mode)** to a **secure, gated apartment complex** where every resident (program) has their own key, walls, and alarm system.

🔐 Key Features of Protected Mode:

🧱 **Memory Protection**: Each program runs in its own isolated memory space, so if it tries to access memory it doesn't own, it crashes without affecting other programs or the operating system.

🌐 **Virtual Memory**: Every application is given the illusion of having access to a full 4GB (or more) of memory, even if the physical RAM is smaller. The operating system makes this possible by using disk space as overflow, through a technique called paging.

🔄 **Multitasking**: The CPU can rapidly switch between multiple programs or tasks, allowing you to run things like Chrome, Spotify, and Visual Studio simultaneously without conflict.

🧩 **Privilege Levels (Rings)**: The CPU enforces a hardware-based separation between user-mode (applications) and kernel-mode (the OS). This ensures that applications cannot directly interfere with or compromise the operating system.

📦 **Flat Memory Model Support**: Although segmentation still technically exists, modern systems often use a flat memory model where memory is accessed linearly, byte by byte, making addressing simpler and more intuitive.

REAL MODE (THE WILD WEST HOUSE)	PROTECTED MODE (THE SECURE APARTMENT BUILDING)
Everyone lives in the same big house. All programs share the same memory space.	Everyone has a private, locked apartment. Each program gets its own isolated memory space.
Anyone can open any door and walk into any room. Programs can access any part of memory directly.	You need permission (handled by the Operating System) to access memory. You can't just walk into another program's space.
If someone breaks something (e.g., spills juice on the carpet), it's chaos for everyone. If one program crashes, it can take down the whole system.	If one tenant (program) crashes, others are safe. The problem is contained within their own apartment, and the rest of the system keeps running.

🗂️ Where Protected Mode Is Used Today (And why):

🪟 **Windows 32-bit Operating Systems** like Windows XP, Vista, 7, 8, and 10 (32-bit editions) rely entirely on Protected Mode to function.

🎮 **Older games and applications from the 2000s** were mostly compiled as 32-bit programs, which means they still run perfectly well in Protected Mode environments.

💡 **WoW64 (Windows-on-Windows 64-bit)** allows modern 64-bit versions of Windows to run older 32-bit applications by emulating a Protected Mode environment for compatibility.

🐧 **32-bit Linux distributions**, such as Ubuntu x86, older versions of Raspberry Pi OS, and many embedded Linux systems, still use Protected Mode under the hood.

🔧 **MASM and NASM tutorials** often teach Protected Mode (32-bit assembly) first because it's cleaner, simpler, and requires less setup than diving straight into 64-bit assembly.

🖱️ **Legacy drivers and low-level tools** are still sometimes compiled in 32-bit mode, even on modern systems, to ensure compatibility with older hardware or software layers.

💡 Why 32-bit Protected Mode Was Such a Leap:

Before Protected Mode, you had:

- No app isolation
- No memory management
- No multitasking
- No security

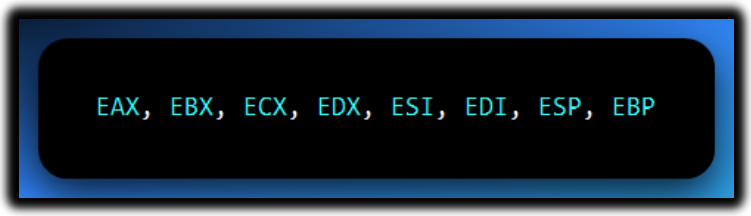
After Protected Mode:

- You could have a full OS with **apps crashing independently**, **virtual RAM**, **security per program**, and **multitasking**.

That's why OSes like **Windows NT**, **Windows 95**, and modern Linux were only possible with this mode.

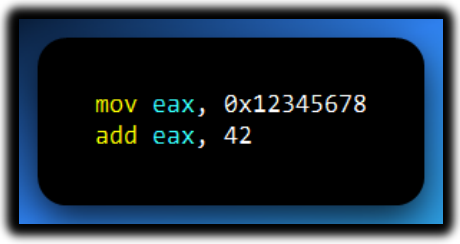
💾 Registers in Protected Mode

You gain access to **extended 32-bit registers**:



Also, segmentation is still there (DS, CS, ES, etc.), but most tutorials flatten it for simplicity.

Example:



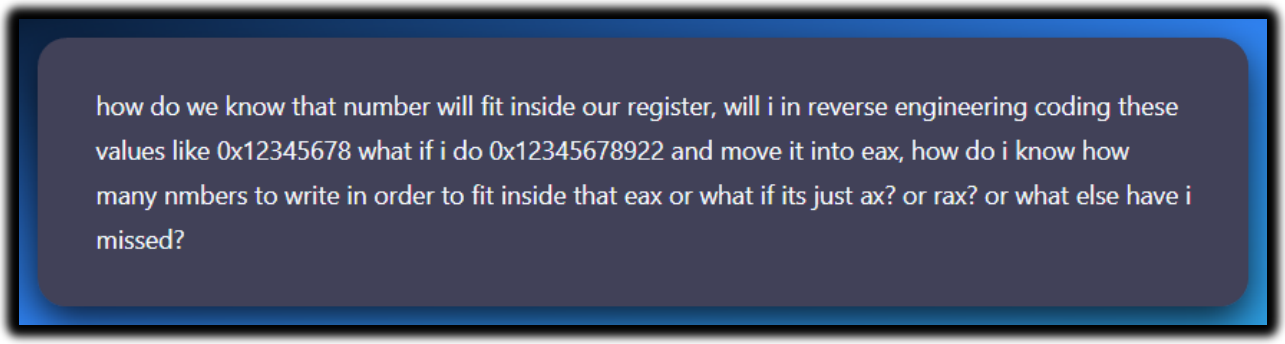
This assembly snippet **moves** the hexadecimal value 0x12345678 into the 32-bit EAX register, then **adds** 42 to it.

Both instructions **operate directly on 32-bit data**, which is standard in **protected mode** environments.

In 32-bit protected mode, registers like EAX, EBX, and ECX are designed to handle 32-bit values, and memory addressing is structured around these 32-bit operations — making this kind of code the norm for systems like 32-bit Windows and Linux.

Register Sizes (x86/x86-64 Architecture):

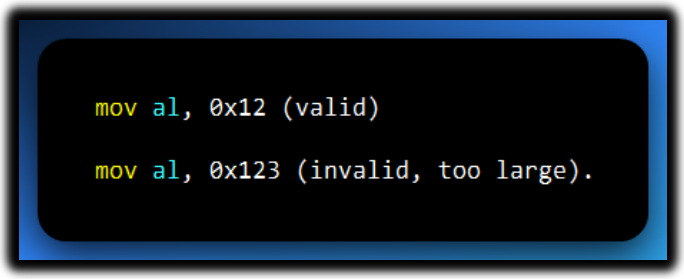
Before we continue, let's address a small issue here:



8-bit registers: AL, AH, BL, BH, CL, CH, DL, DH

Can hold values from 0x00 to 0xFF (0 to 255 unsigned, or -128 to 127 signed).

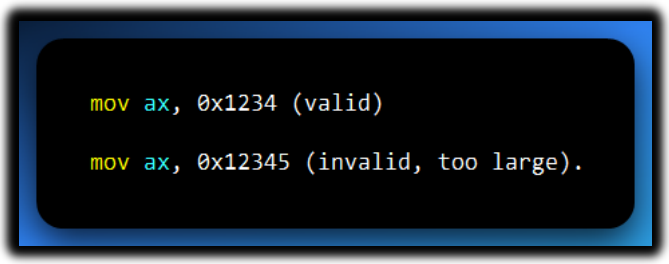
Example:



16-bit registers: AX, BX, CX, DX, SI, DI, SP, BP

Can hold values from 0x0000 to 0xFFFF (0 to 65,535 unsigned, or -32,768 to 32,767 signed).

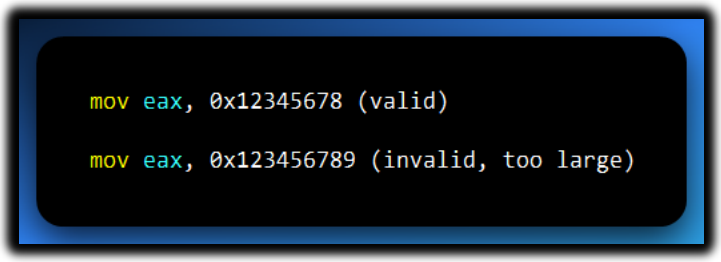
Example:



32-bit registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

Can hold values from 0x00000000 to 0xFFFFFFFF (0 to 4,294,967,295 unsigned, or -2,147,483,648 to 2,147,483,647 signed).

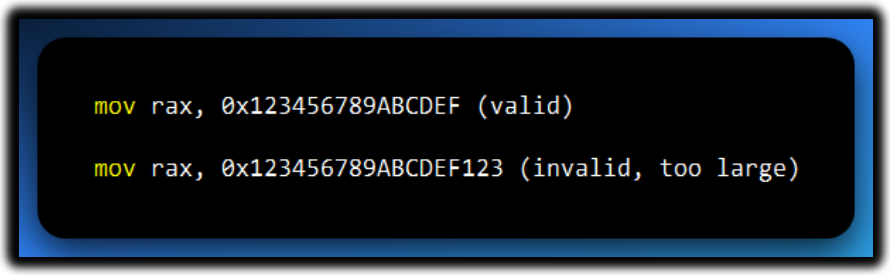
Example:



64-bit registers: RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP

Can hold values from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF (0 to 18,446,744,073,709,551,615 unsigned, or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 signed).

Example:



Checking if a Value Fits

Hexadecimal digits per register size:

- 8-bit: **2 hex digits** (e.g., 0x12).
- 16-bit: **4 hex digits** (e.g., 0x1234).
- 32-bit: **8 hex digits** (e.g., 0x12345678).
- 64-bit: **16 hex digits** (e.g., 0x123456789ABCDEF).

Example:

- mov eax, 0x12345678 → Valid (8 hex digits, fits 32-bit).
- mov eax, 0x123456789 → **Invalid** (9 hex digits, exceeds 32-bit). It **won't fit** — the CPU will take only the **lower 4 bytes – TRUNCATION.**
- mov rax, 0x123456789 → Valid (fits in 64-bit).

What Happens if You Exceed the Limit?

Most assemblers (like NASM, MASM, FASM, GAS) will **throw an error** if you try to move a too-large value into a register.

Example (NASM error):

```
mov eax, 0x123456789  ; Error: value too large for register
```

Some assemblers might **truncate** the value (keep only the lowest bits), but this is **not reliable** and should be avoided.

Truncation

Say you do:

```
mov eax, 0x12345678922 ; 0x12345678922 is 9 hex digits (too big)
```

It **won't fit** — the CPU will take only the **lower 4 bytes** (last 8 hex digits):
0x12345678922 → gets **truncated** to 0x345678922 → then **only the last 8 digits** → 0x45678922 (lower 32 bits).

```
0x12345678922
          └─ keep last 8 hex digits =   0x45678922
```

👎 The extra 0x1 at the beginning gets **cut off silently**. It's like trying to pour 1.5 liters of soda into a 1-liter bottle. The rest just spills.

🦾 What Happens if You Use a Smaller Register?

Same logic, smaller limit:

```
mov ax, 0x1234       ; ✅ fits fine (16-bit)
mov ax, 0x12345678   ; ❌ too big, gets chopped to 0x5678
```

AX is only 16 bits → only takes last 4 hex digits.

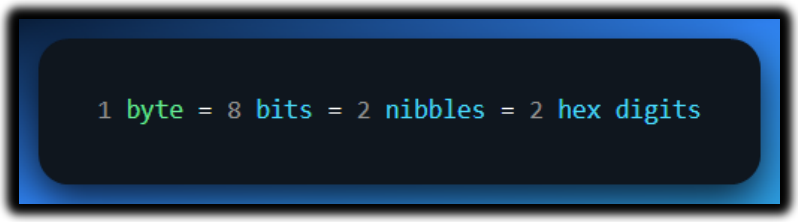
🔥 Cheat Sheet: How Many Hex Digits per Register?

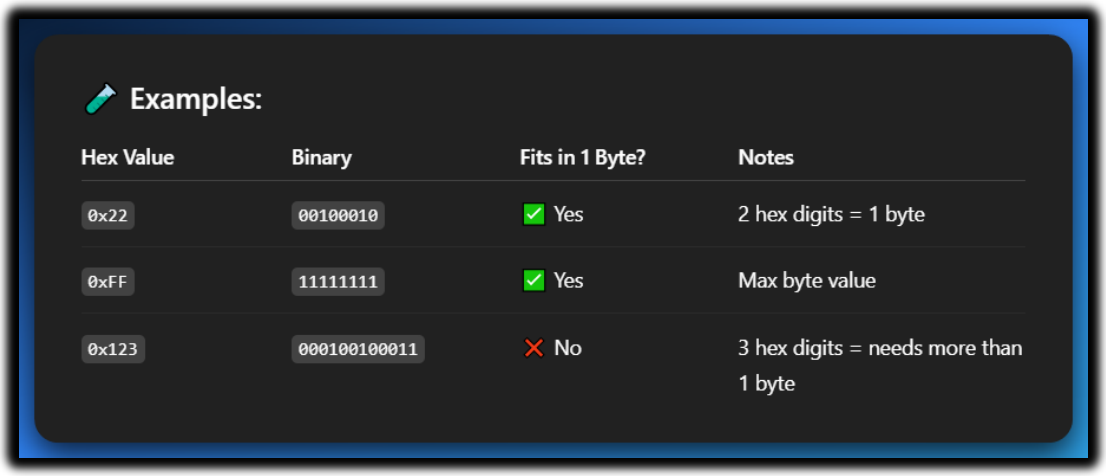
REGISTER	MAX HEX DIGITS	TIP	EXAMPLE
AL, AH, BL, BH, CL, CH, DL, DH	2	1 byte (8 bits) = 2 hex digits	`0x7F`
AX, BX, CX, DX, SI, DI, SP, BP	4	2 bytes (16 bits) = 4 hex digits	`0xBEEF`
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP	8	4 bytes (32 bits) = 8 hex digits	`0xDEADBEEF`
RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15	16	8 bytes (64 bits) = 16 hex digits	`0xCAFEBABEDEADBEEF`

For the confused backbenchers, lets fix you:

✅ One Byte = 8 Bits = 2 Hex Digits

A byte holds 8 bits. Each hex digit represents 4 bits (aka a nibble).





"If my value is 2 hex digits (like 0x7F, 0x22, 0xB4), it fits in a byte."

⚠️ Don't Confuse with Decimal

Some decimal numbers look small but still take more than 1 byte:

- 200 (decimal) = **0xC8** → ✅ fits
- 300 (decimal) = **0x12C** → ❌ 3 hex digits → needs 2 bytes

Special Cases:

Sign Extension: If you move a smaller value (e.g., mov eax, -1) into a larger register (e.g., rax), the value is sign-extended.

Zero Extension: Moving unsigned values (e.g., movzx eax, al) fills upper bits with zeros.

We'll see these in future topics.

Key Takeaway:

- **Count the hex digits** to ensure the value fits the register.
- **Assemblers will warn you** if the value is too large.
- **Reverse Engineering Tip**: When analyzing code, check the register size to understand how much data is being manipulated.
- 1 byte = 2 hex digits (a nibble each).
- AL / AH can store anything up to 0xFF.
- When writing hex, count the digits to know how many bytes you're dealing with.
- Don’t confuse hex with regular base-10 numbers.