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:

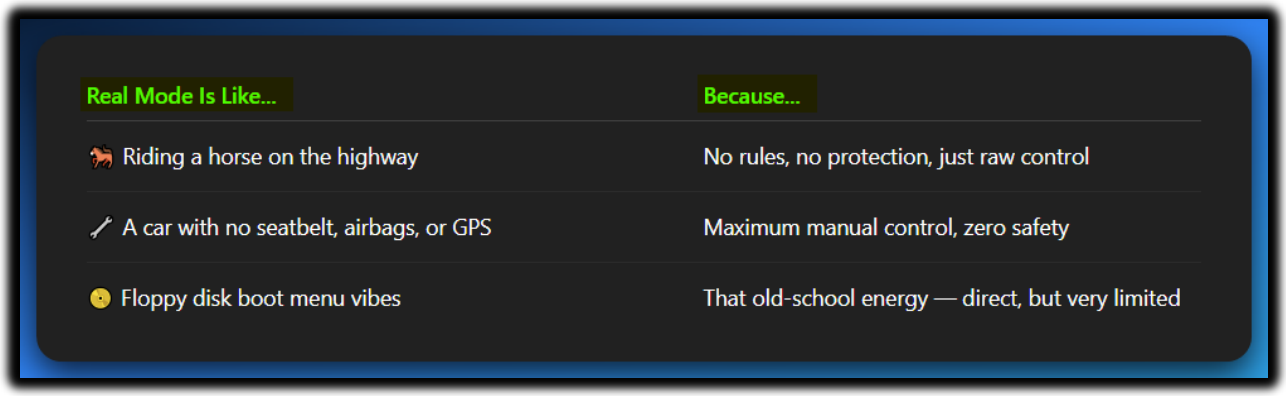
Use Case	Why It Uses Real Mode
BIOS	It's the first thing your CPU runs. BIOS firmware is old-school and still coded for Real Mode to stay compatible with all systems.
Bootloaders	The first stage of your OS loader (like GRUB or BOOTMGR) starts in 16-bit real mode because the CPU boots there by default.
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**:

```
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
```

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

Example:

```
mov eax, 0x12345678
add eax, 42
```

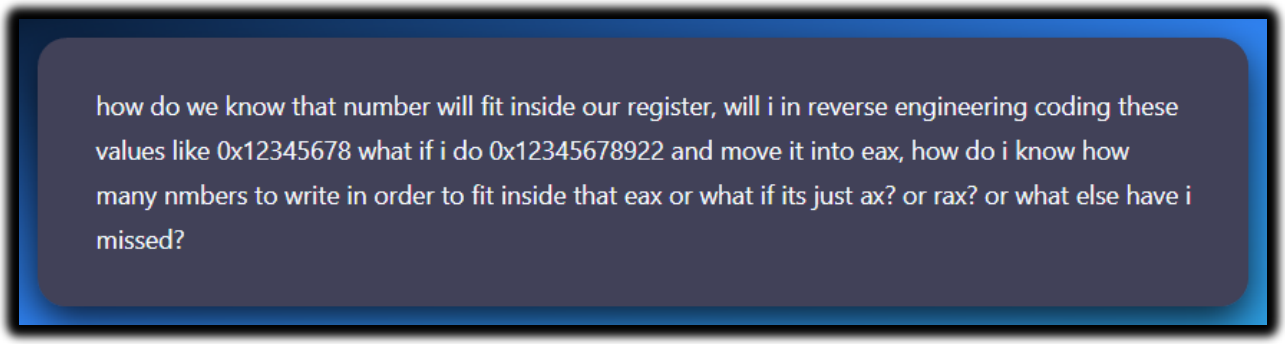
This assembly snippet **moves** the hexadecimal value 0x12345678 into the 32-bit EAX register, then **adds** 42 to it.

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):

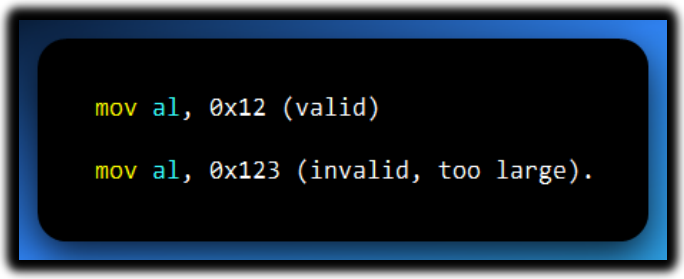
Before we continue, let's address a small issue here:



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).

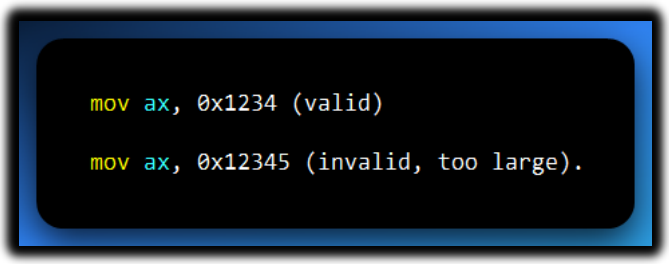
Example:



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).

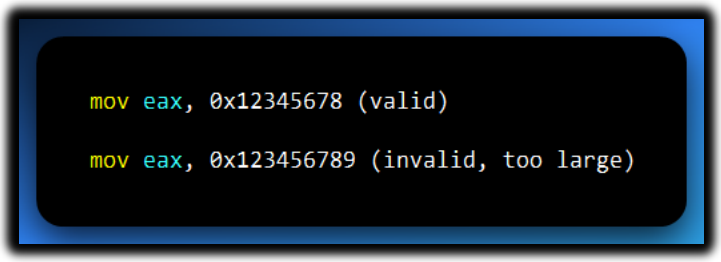
Example:



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).

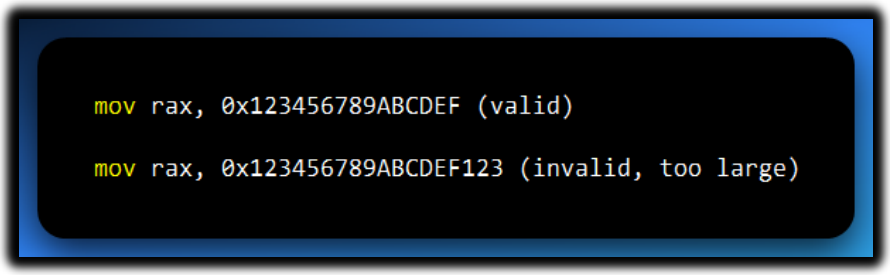
Example:



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.

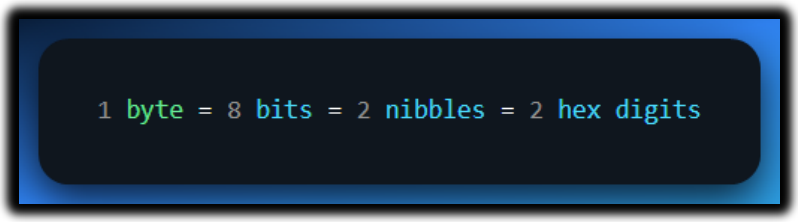
🔥 Cheat Sheet: How Many Hex Digits per Register?

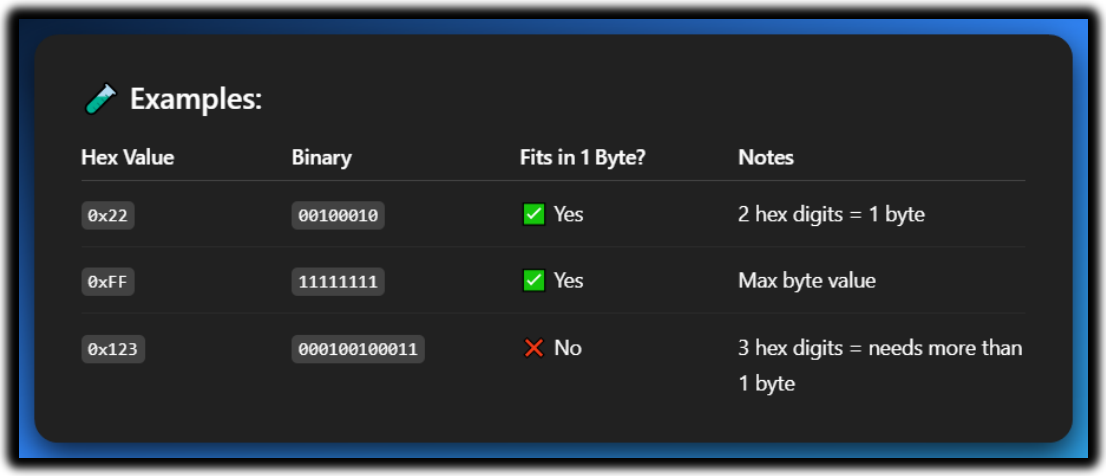
REGISTER	MAX HEX DIGITS	TIP	EXAMPLE
AL, AH, BL, BH, CL, CH, DL, DH	2	1 byte (8 bits) = 2 hex digits	`0x7F`
AX, BX, CX, DX, SI, DI, SP, BP	4	2 bytes (16 bits) = 4 hex digits	`0xBEEF`
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP	8	4 bytes (32 bits) = 8 hex digits	`0xDEADBEEF`
RAX, RBX, RCX, RDX, RSI, RDI, RSP,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

⚠️ 32-bit protected mode is Legacy today

But still essential for Reverse Engineering and Kernel work… etc

🌐 64-Bit Long Mode – The Skyscraper City of Modern CPUs

🚀 What is Long Mode?

Welcome to the **current era** of computing. Long Mode is how **modern 64-bit CPUs** run your operating systems and apps today.

Introduced with **AMD64** (yup, AMD beat Intel here), this mode **unlocked way more RAM**, **better performance**, and **modern security features** — without throwing away what made Protected Mode great.

Think of Long Mode like a **future-proof skyscraper city**:

Massive vertical space (more memory), more elevators (registers), and smarter infrastructure (paging, security).

📦 Long Mode = Protected Mode ++

Technically, Long Mode is a **supercharged version** of Protected Mode.
It still supports:

- Paging (virtual memory)
- User/kernel isolation
- Multitasking
  ...but adds **64-bit registers** and **64-bit address spaces**.

⚙️ Key Features of Long Mode:

- 🧠 **64-bit Registers**: In Long Mode, traditional 32-bit registers like EAX, EBX, and ECX are replaced with their 64-bit counterparts — RAX, RBX, RCX, etc. In addition, the architecture introduces eight brand-new general-purpose registers: R8 through R15, giving developers more flexibility and faster data handling.
- 💾 **Huge Address Space**: Long Mode unlocks a theoretical memory address space of up to **16 exabytes** (that's 18,446,744,073,709,551,616 bytes). In practice, most modern CPUs support up to **256 TB** of addressable space, which is still astronomically higher than 32-bit limits.
- 🚀 **Faster Performance**: With more registers and wider 64-bit data paths, CPUs in Long Mode can handle larger numbers and datasets more efficiently — which means faster calculations, better multitasking, and improved performance for heavy applications.
- 🧱 **RIP-Relative Addressing**: Long Mode introduces **RIP-relative addressing**, which allows code to access memory locations relative to the current instruction pointer (RIP). This makes **position-independent code (PIC)** easier to write and more secure — something modern operating systems rely on for features like shared libraries and code randomization.
- 🔐 **Stronger Isolation and Security**: Long Mode supports a hardened separation between **kernel** and **user space**, along with advanced security features like the **NX (No-eXecute) bit**, **ASLR (Address Space Layout Randomization)**, and **SMEP (Supervisor Mode Execution Prevention)**. These features work together to protect against modern memory-based attacks and vulnerabilities.

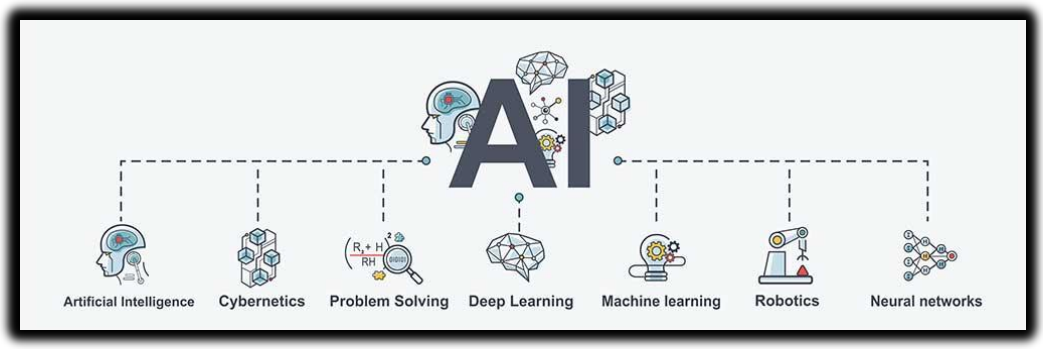
💻 Real-World Use Cases of 64-bit Long Mode (a.k.a. Where It's Actually Used)

🪟 **Modern Operating Systems**: Pretty much every current OS — Windows 10, Windows 11, macOS, and modern Linux distros — runs entirely in 64-bit Long Mode. If your computer is less than 15 years old, you're already living in it.

🎥 **Heavy-Hitter Apps (Video, Databases, etc.)**: Apps like video editors, big databases, and 3D rendering engines need access to more than 4GB of RAM — which 32-bit systems just can't handle. Long Mode makes that possible.

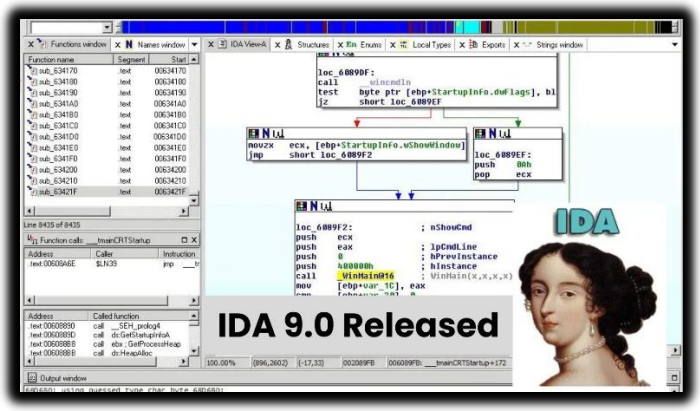
🎮 **Modern Games**: Games today eat RAM like snacks. 8GB+ is standard, 16GB+ is common — and that means they *have to* be 64-bit. Most AAA titles won't even launch in a 32-bit world.

🔬 **Scientific Computing & Machine Learning**: When you're working with huge arrays, neural networks, or massive datasets, 32-bit systems just tap out. Long Mode opens the door for processing at scale — think AI, simulations, bioinformatics, physics engines, all that jazz.



🛡️ **Malware (and Anti-Malware)**: Modern malware is built to target 64-bit OSes, and defenders (a.k.a. reverse engineers like you) need 64-bit tools to analyze and unpack them. Long Mode isn't just for legit programs — it's the battlefield for digital warfare.

🔧 **Reverse Engineering EXEs**: Most executables on a 64-bit Windows system use the PE64 format (Portable Executable, 64-bit). If you're cracking, tracing, or dissecting apps, you gotta know how 64-bit registers, memory layout, and instructions work — or you'll be totally lost.



📏 Register Breakdown in Long Mode

In Protected Mode (32-bit), you had:

```
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
```

Now in Long Mode (64-bit), you've got:

```
RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP   ; Base set, 64-bit
R8  to R15                               ; New registers!
```

Here are their full names:

🧱 RBX – The Extended Base Register

Used for holding base addresses in memory.
Think: a pointer to the start of your giant data structure — like the foundation of a skyscraper.

🔁 RCX – The Extended Count Register

Used in loops, counts, and string operations.
Think: a digital clicker counting how many reps your CPU has left to do.

📤 RDX – The Extended Data Register

Handles I/O and large-number math.
Think: your CPU's multipurpose toolbelt — for division, data transfer, etc.

📦 RSI – The Extended Source Index

Points to where data is *coming from* (like for string/memory ops).
Think: a chef's hand reaching into the pantry — grabbing the source.

📥 RDI – The Extended Destination Index

Points to where data is *going*.
Think: that same chef dumping the food into a bowl — the destination.

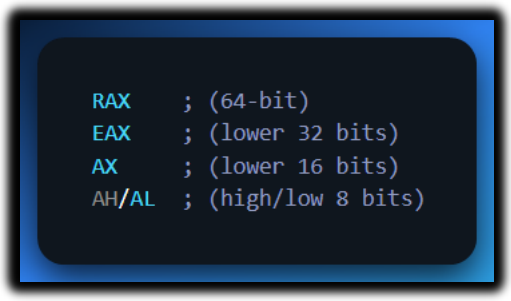
📚 RSP – The Extended Stack Pointer

Always points to the top of the stack.
Think: a stack of plates — this register tracks the one on top.

📌 RBP – The Extended Base Pointer

Used to anchor the current function's stack frame.
Think: a fixed bookmark inside your temporary memory, pointing to where local variables live.

And yes, each of these can be broken down further:



So, you still get backward compatibility with older 32-bit and 16-bit code, but now with way more horsepower.

🧠 R8 to R15 – The New Recruits (64-bit Only)

When CPUs evolved from 32-bit to 64-bit, they didn't just stretch existing registers (like EAX → RAX).

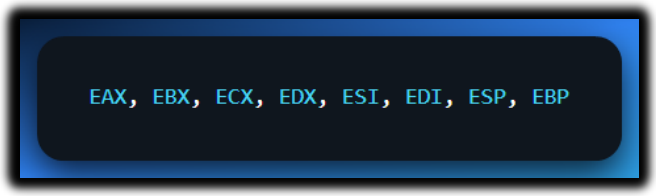
When we made the jump to 64-bit, Intel said: **"You know what? 8 general-purpose registers just ain't enough anymore."** So, they gave us **8 more**: R8 to R15.

These are full 64-bit general-purpose registers — just like RAX, RBX, etc. — but **exclusively available in 64-bit mode** (Long Mode). You won't see these in 32-bit assembly at all.

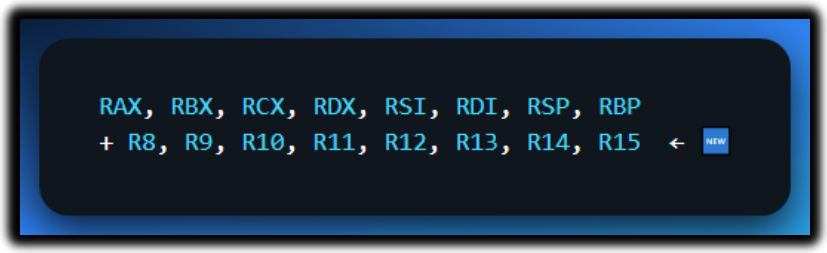
🔧 What They're Used For:

- Used heavily in **function parameter passing** (the Windows/Linux 64-bit calling conventions rely on them)
- Great for **extra temporary storage** when your code needs more than the classic 8 registers
- Super handy in **loop unrolling, SIMD routines**, or **low-level optimization**
- You'll see malware, obfuscators, and compilers use them for sneaky tricks or performance

So instead of:



We now get:



That's **16 total general-purpose registers** in 64-bit mode. Huge boost.

🎮 Why do we care about R8–R15?

1. Function Parameter Passing in 64-bit Linux (System V ABI)

When you call a function in 64-bit Linux (or compile with GCC, Clang, etc.), the **first six arguments** are passed using registers (not on the stack like in 32-bit).

The order is:

Argument	Register
Arg 1	RDI
Arg 2	RSI
Arg 3	RDX
Arg 4	RCX
Arg 5	R8
Arg 6	R9

This table shows a common calling convention, specifically for Linux (and other Unix-like systems) on x86-64 architecture, often referred to as the System V AMD64 ABI.

For the first six arguments, it prioritizes using **specific general-purpose registers**, including the "new" registers like **R8** and **R9**, to pass data directly to a function, which is **much faster** than pushing them onto the stack.

Example:

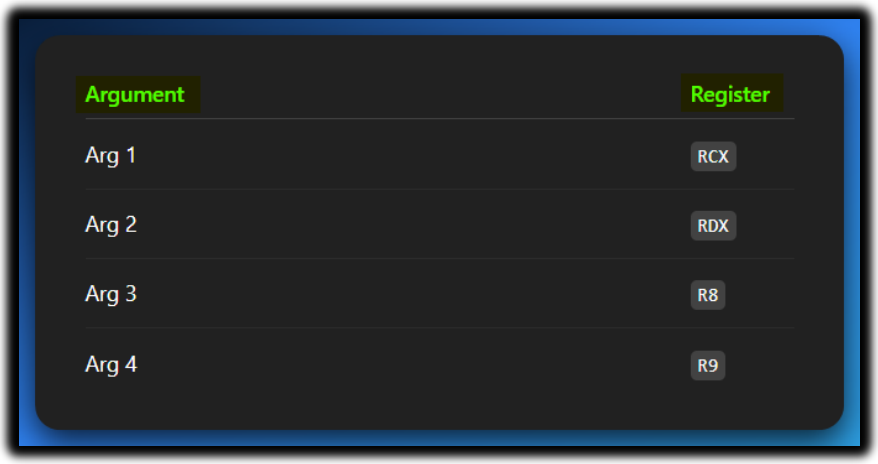
```
; Calling a function like this:
foo(1, 2, 3, 4, 5, 6)

; The compiler translates it so:
RDI = 1
RSI = 2
RDX = 3
RCX = 4
R8  = 5
R9  = 6
```

That's why R8 and R9 **aren't optional weird extras** — they are baked into how functions work in 64-bit!

📬 What about Windows?

In **Windows 64-bit (Microsoft x64 calling convention)**, it's a little different:



So, in both Linux and Windows, R8 and R9 are **used early** in parameter passing.

🧩 Sub-registers of R8–R15

Just like how RAX has smaller siblings:

- EAX (32-bit)
- AX (16-bit)
- AL (8-bit low)
- AH (8-bit high)

The new registers R8–R15 also have sub-registers:

In 64-bit systems, Intel introduced a set of new general-purpose registers: R8 through R15.

Just like the older AX, BX, CX, DX registers, these new 64-bit registers also have sub-registers that allow you to access smaller portions of their data (32-bit, 16-bit, and 8-bit parts).

REGISTER	64-BIT	32-BIT	16-BIT	8-BIT
R8	R8	R8D	R8W	R8B
R9	R9	R9D	R9W	R9B
R10	R10	R10D	R10W	R10B
R11	R11	R11D	R11W	R11B
R12	R12	R12D	R12W	R12B
R13	R13	R13D	R13W	R13B
R14	R14	R14D	R14W	R14B
R15	R15	R15D	R15W	R15B

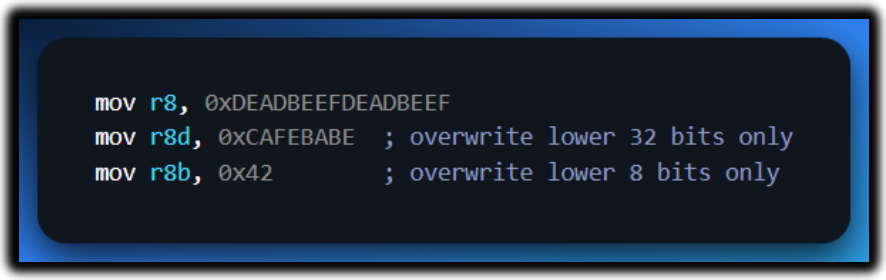
✅ So yes, you can move 8-bit values into **R11B**, 16-bit values into **R12W**, 32-bit values into **R9D**, and so on.

✅ This works just like how you'd use **AL** (8-bit), **AX** (16-bit), or **EAX** (32-bit) with the legacy **RAX** register.

✅ This flexibility allows for efficient manipulation of data of different sizes within the larger 64-bit registers.

🧪 Real Usage

Example 1 – Simple data move



```
mov r8, 0xDEADBEEFDEADBEEF
mov r8d, 0xCAFEBABE  ; overwrite lower 32 bits only
mov r8b, 0x42        ; overwrite lower 8 bits only
```

Example 2 – Passing function args in Linux

```
mov rdi, msg        ; arg1
mov rsi, 13         ; arg2
mov rdx, 1          ; arg3
mov rcx, 0          ; arg4
mov r8,  0          ; arg5
mov r9,  0          ; arg6
call syscall_func
```

🎯 Why This Matters for You:

- If you're writing **shellcode**, reversing malware, or working on **system-level C or C++**, you *must* understand how args are passed.
- If you're building a compiler, parser, or learning ABI design – this is ground truth.
- If you're debugging a crash and see R8 = 0x0BADF00D – you now know it might be **parameter 5**.

🧠 TLDR – R8 to R15 in 64-bit Assembly (Cleaned Up)

R8 to R15 are **extra general-purpose registers** introduced in **64-bit mode** — they **don't exist at all in 32-bit Protected Mode**. These registers give you more firepower for handling data, optimizing performance, and passing function arguments.

In the 64-bit calling convention (especially on Linux and Windows), they help carry function arguments:

- 📄 Example: R8 and R9 come in right after RCX, RDX, RSI, and RDI.
- 💻 So, if you're writing shellcode, reversing binaries, or tracing sys-calls, **you *need* to know their role**.

Just like RAX breaks into EAX → AX → AL, these registers have sub-registers too:

- **R8D–R15D → 32-bit**
- **R8W–R15W → 16-bit**
- **R8B–R15B → 8-bit**

🔎 Why it matters: Long Mode didn't just make registers bigger — it added **more**.
More registers = more freedom, more complexity, more control.
If you're in 64-bit land, these are *not optional knowledge*. Period.

🧠 Memory Access in Long Mode

You now have:

- **64-bit flat address space**
- **Paging with 4 levels (PML4)** to map virtual addresses
- Still **no segmentation** like in Real Mode — segmentation is mostly disabled (yay, simplicity!)

That's why most 64-bit assembly tutorials say: "Forget segments. Think in pages."

🚧 Why 64-bit Mode Isn't Always Taught First – Painful AF

- 🤯 **It's more complex under the hood**:
  System calls don't work the same — you can't just drop a casual int 0x80 anymore like it's 2003. Instead, 64-bit uses a totally different ABI (Application Binary Interface), and the registers behave differently. The rules changed, and you gotta learn the new playbook.

- 🧠 **Debugging is trickier**:
  You're now juggling wider 64-bit registers like RAX, dealing with **RIP-relative addressing** (yeah, your instructions reference memory based on the current address), and following new calling conventions. It's like graduating from checkers to 4D chess.

- 📉 **Less hand-holding for beginners**:
  Most tutorials out there still cling to 32-bit because it's simpler and easier to teach. That means you'll find fewer guides, fewer StackOverflow answers, and more "figure it out yourself" moments. But hey…