GENERAL PURPOSE REGISTERS (GPRs)

The CPU's Workbench

Think of the CPU as a worker. **Registers** are the worker's hands or small workbench.

They hold the tools and materials (data, addresses, pointers) the CPU needs right now.

I. Why they matter:

Registers give the CPU instant access to information.

Without them, the CPU would waste time going back and forth to RAM (the big library) or the Hard Drive (the warehouse).

II. How they work:

The CPU can't use RAM directly. It must first copy data into a register.

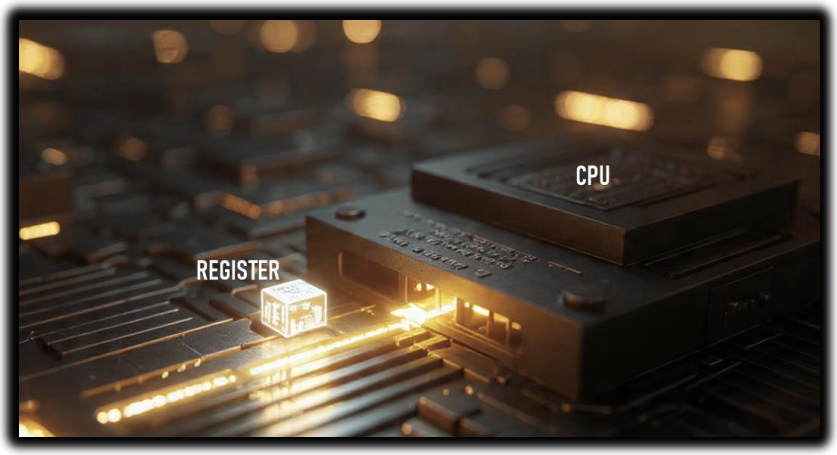
Once in a register, the CPU can add numbers, compare values, or move data quickly.

III. Key Points:

- **Location:** Inside the CPU chip.
- **Speed:** Extremely fast — one cycle vs. hundreds for RAM.
- **Volatility:**

The 32-bit Registers

In this modern x86 architecture, we have eight special types of registers called general-purpose registers (GPRs). Each of these GPRs is a single 32-bit block. This is different from the earlier 16-bit registers that were used in older CPU models.



I. Breaking Down Each Register

Let's explore each register and its typical uses:

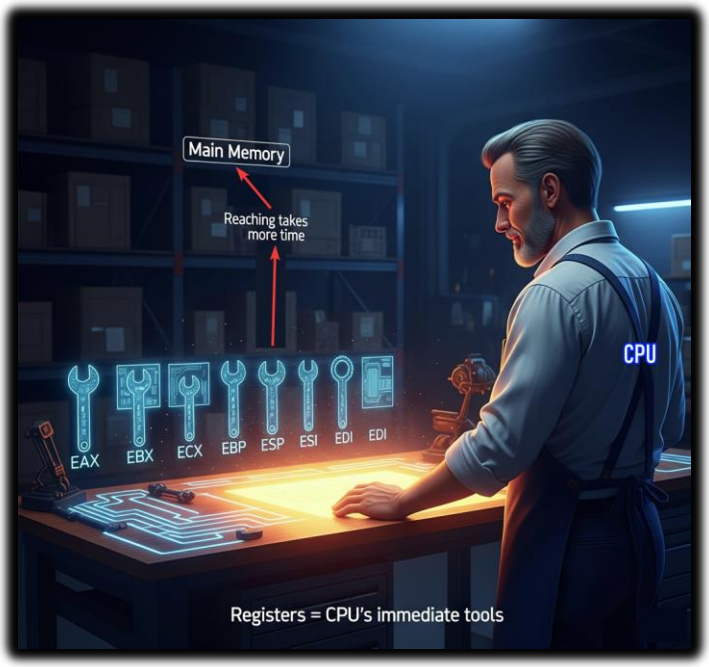
1. **AX**: Used for addressing data, pointing to specific locations.
2. **BX**: Similar to AX, used for addressing data, but often used for larger values (like 16-bit numbers).
3. **CX**: Often used for addressing data in addition to what's already stored in AX and BX.
4. **DX**: A bit different from the others, it's usually set by the program or hardware to point to a specific address.
5. **BP**: Used for storing addresses of previously loaded data (or code).
6. **SP**: This register holds the stack pointer, which is used to manage memory and return values.
7. **SI** (or **DI**): Usually linked to BP, SI stores the base address of a block of memory.
8. **EAX**, **EBX**, **ECX**, **EDX**: These registers are like AX, BX, CX, and DX, but with 16-bit widths instead of 32 bits.

So, in short, these eight registers provide direct access to your CPU's data storage locations, making them incredibly useful for efficient program execution.

In the x86 architecture, registers have grown over time. Intel maintained backward compatibility, meaning the names tell you the size.

- **8-bit (1970s):** AL, AH, BL, etc.
- **16-bit (1980s):** AX, BX, CX. (The "X" usually stands for pair or extended from 8-bit).
- **32-bit (1990s):** EAX, EBX, ECX. (The **"E"** stands for **Extended**).
- **64-bit (2000s):** RAX, RBX, RCX. (The **"R"** stands for **Register** or Re-extended).

Note: In this chapter, we focus on the 32-bit "E" registers, as they are the standard baseline for malware analysis and reverse engineering.



II. The Multi-Role Power Tools

The general-purpose registers in x86 are not just simple scratchpads. They're actually **multi-role power tools** that handle various tasks, such as data, pointers, counters, parameters, addresses, and even system calls.

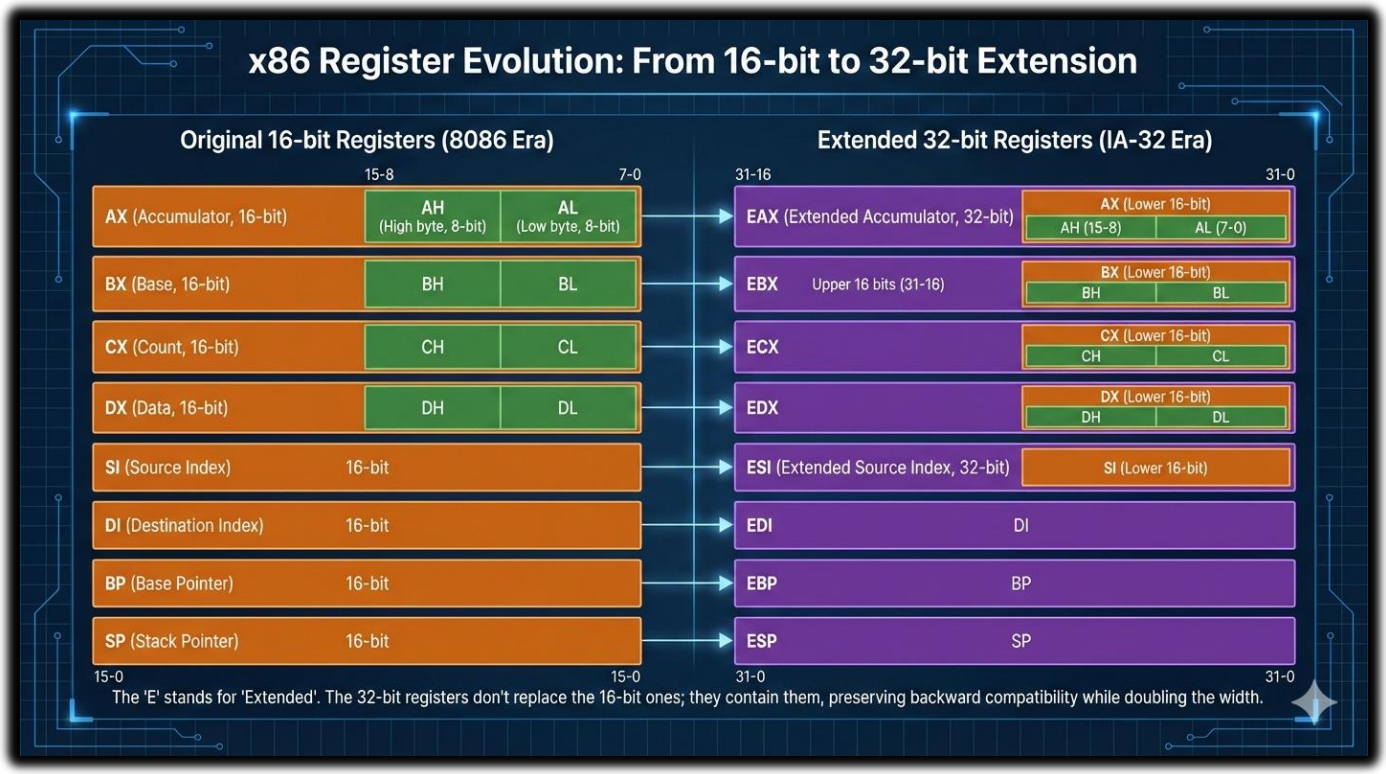
Think of them like a set of versatile hammers, each one capable of performing different functions. For example:

- **AX** is for moving data - it can be used to load, store, or manipulate data in various locations.
- **BX** is similar to AX, but often used for *larger values* or *addressing higher memory* locations.
- **CX** is another type of register that's often used in addition to AX and BX.

III. Gaining Muscle in Protected Mode

In **protected mode (32-bit),** the general-purpose registers grew stronger and more powerful.

The first eight registers **AX to DI** became **EAX to EDI**, which are like a new set of tools with 16-bit widths instead of the original 8-bit width.



Later, in long mode (64-bit), even more registers were added:

- RAX to RDI: This is another "set of 8" with increased functionality.
- R8 to R15: These additional registers bring the total count to 16, providing a wider range of operations and tools for your CPU.

It's worth noting that these multi-role power tools are not just used in x86 processors; they're also found in other architectures, such as ARM and PowerPC.

3.2 The Core Four: Arithmetic & Logic

These four registers—EAX, EBX, ECX, and EDX—are special because they can be split into smaller pieces.

This comes from very early x86 designs in the 1970s, but it's still useful today.

It lets the CPU work with single bytes, like characters in a string, instead of always handling full numbers.

I. EAX

EAX is mainly used for math and logic.

Most calculations end up here, and the CPU is optimized to use it fast.

When a function finishes running, its return value is placed in EAX.

Because of this, programs often check EAX right after a function call to see if something succeeded or failed.

For example, after a password check, the code might look at EAX to see if the result was 1 or 0.

EAX can be split into smaller parts: AX is the lower 16 bits, and that can be split again into AH (high 8 bits) and AL (low 8 bits).

II. EBX

EBX is commonly used to hold a base address in memory.

Think of it as pointing to where some data starts, like the beginning of an array or structure.

Older code used EBX heavily for memory access, but modern compilers don't give it a strict role anymore.

Because of that, it's often used as a general-purpose register to store values that need to stay around for a while. Like EAX, it can be split into BX, BH, and BL.

III. ECX

ECX is mostly used as a counter.

The CPU relies on it for loops, especially when repeating an instruction multiple times.

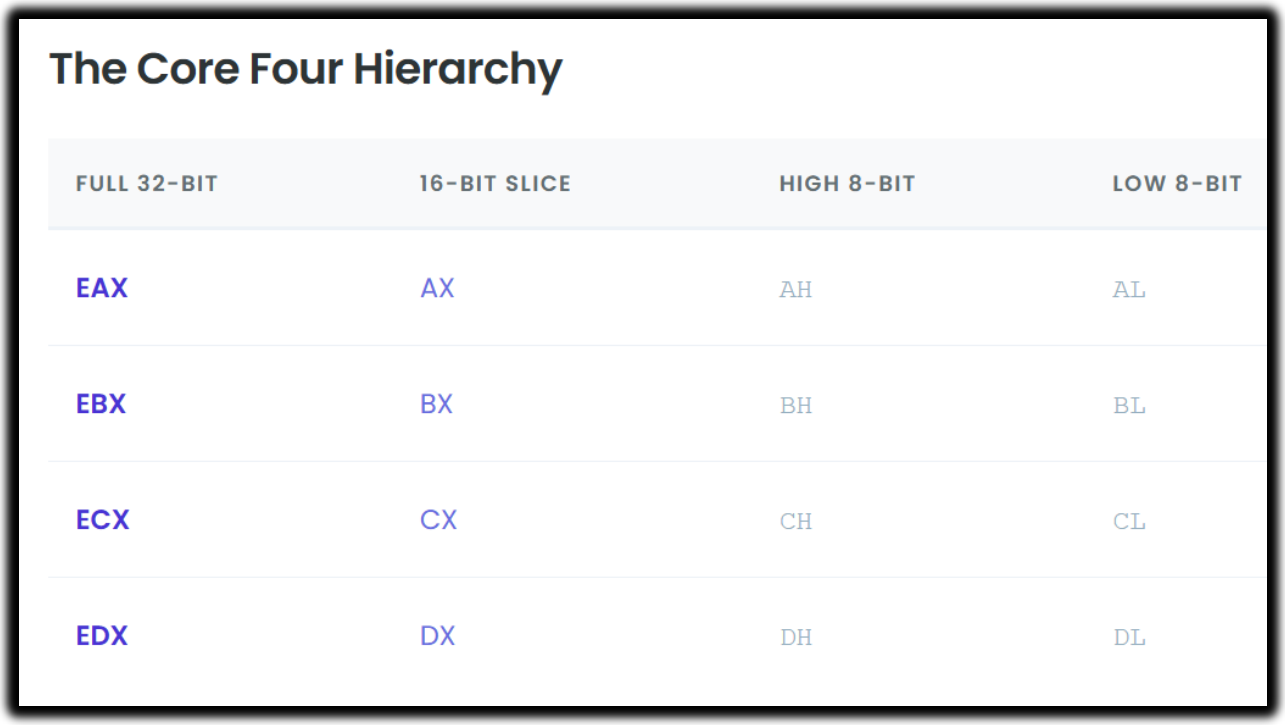
There's even a special LOOP instruction that automatically decreases ECX by one each time it runs.

When copying or moving data in chunks, ECX usually holds how many bytes or items need to be processed. ECX can also be split into CX, CH, and CL.

IV. EDX

EDX is often used as a helper register for EAX. When multiplying or dividing large numbers, the result may not fit in EAX alone, so the CPU uses EDX together with EAX to hold the full result.

EDX is also commonly used in input and output operations, such as storing port addresses. Like the others, it can be split into DX, DH, and DL.



The Core Four Hierarchy

FULL 32-BIT	16-BIT SLICE	HIGH 8-BIT	LOW 8-BIT
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

3.3 The Index Registers

ESI and EDI are used when the CPU is moving data in memory.

This usually means copying arrays, strings, or blocks of bytes from one place to another.

You'll see them a lot in code that handles buffers and string operations because they're built to do this kind of work fast.

I. ESI - Extended Source Index

Holds the address of the data being read.

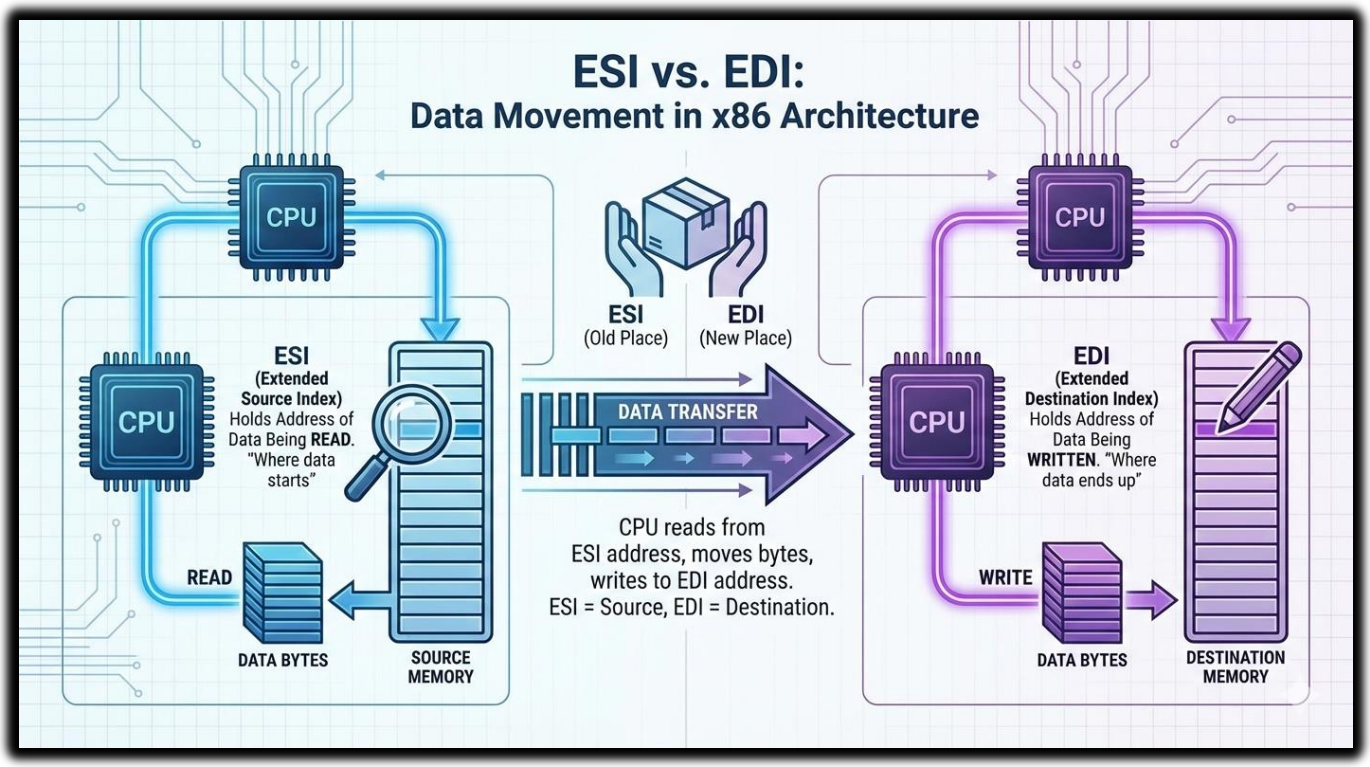
It points to where the data starts, like the source of a copy.

When the CPU is moving bytes, it reads them from the memory location stored in ESI.

II. EDI – Extended Destination Index

Holds the address of where the data is being written. It points to the destination, where the copied data will end up. As the CPU copies each byte, it writes it to the address in EDI.

You can think of it like moving stuff from one place to another. ESI is the old place where the boxes are, EDI is the new place, and the CPU walks back and forth moving things from ESI to EDI.



3.4 The Pointer Registers (The Stack)

ESP and EBP are used to manage the stack. These registers are extremely important, and you normally don't use them for math or random storage.

If they get messed up, the program will usually crash because the CPU no longer knows where the stack is.

I. ESP – Extended Stack Pointer

Always points to the top of the stack.

When a value is pushed onto the stack, ESP moves downward to make space.

When a value is popped off, ESP moves back up.

The CPU constantly updates ESP as the program runs, so it is always changing.

Because of this, you generally should not manually change ESP unless you are writing very low-level code like a compiler or an operating system.

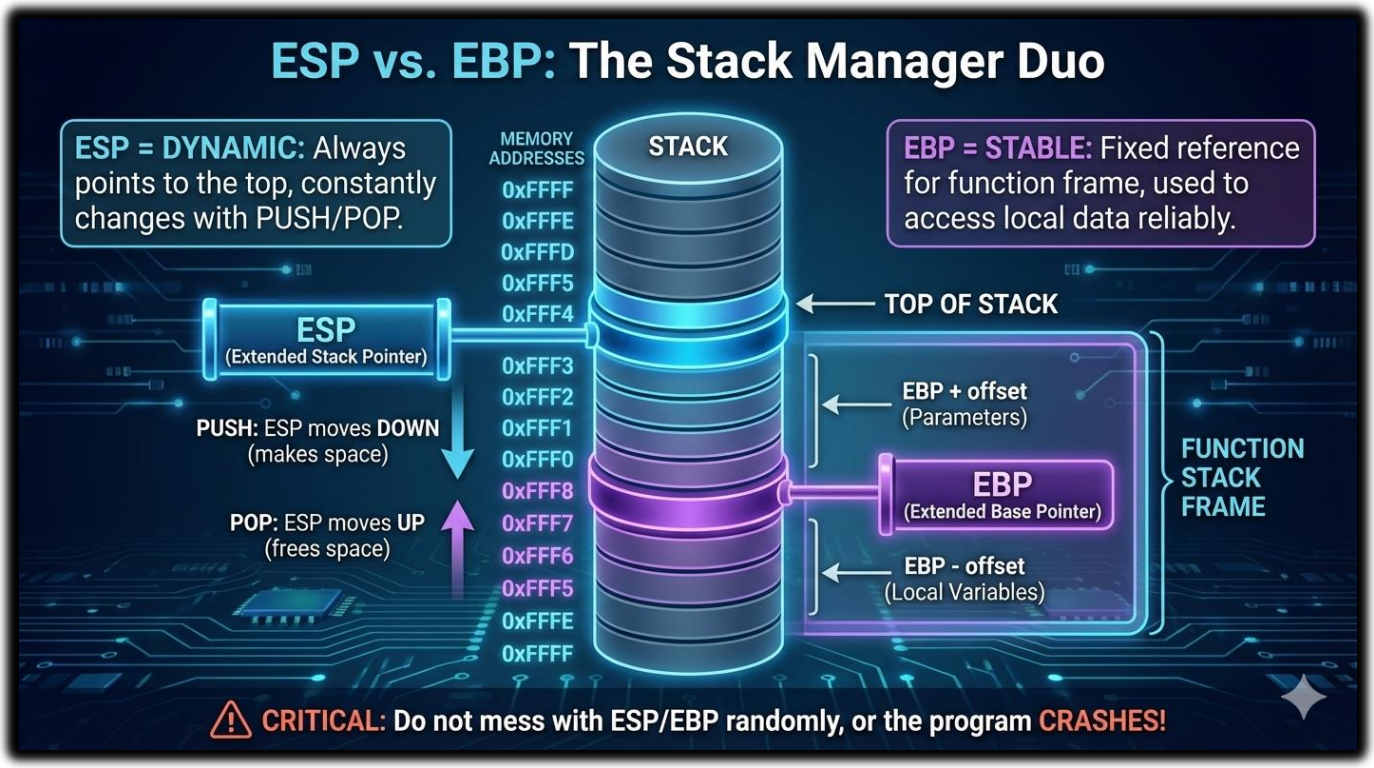
II. EBP – Extended Base Pointer

Used as a stable reference point for a function.

When a function starts, EBP is set to mark the base of that function's stack frame and then stays mostly the same while the function runs.

This is important because ESP keeps moving as values are pushed and popped.

By using EBP, the program can reliably access local variables and function parameters using fixed offsets, like **"EBP minus 4"** or **"EBP plus 8."**



3.5 Register Breakdown Table (32-bit Architecture)

Visualizing how the registers overlap is crucial. Remember: Changing AL **changes** EAX. They are the same physical wires.

32-BIT (E-REG)	16-BIT PART	8-BIT HIGH	8-BIT LOW	PRIMARY FUNCTION
EAX	AX	AH	AL	Accumulator: Arithmetic & Return Values
EBX	BX	BH	BL	Base: Memory Addressing & Storage
ECX	CX	CH	CL	Counter: Loops & Repeating Instructions
EDX	DX	DH	DL	Data: Large Math & I/O Operations
ESI	SI	*None*	*None*	Source Index: Reading Data Strings
EDI	DI	*None*	*None*	Destination Index: Writing Data Strings
EBP	BP	*None*	*None*	Base Pointer: Stable Stack Reference
ESP	SP	*None*	*None*	Stack Pointer: Top of Current Stack

3.6 Reverse Engineering Insight: Register Slicing

Even in a 64-bit world, the small pieces of registers like AL and AH still matter.

A lot of real code works on bytes, not full 32-bit or 64-bit values.

Characters in strings, for example, are only 8 bits, so when a program checks a password one character at a time, it compares the value in AL against a single byte like 0x41 for the letter 'A', not the entire EAX register.

This also shows up in malware and obfuscated code.

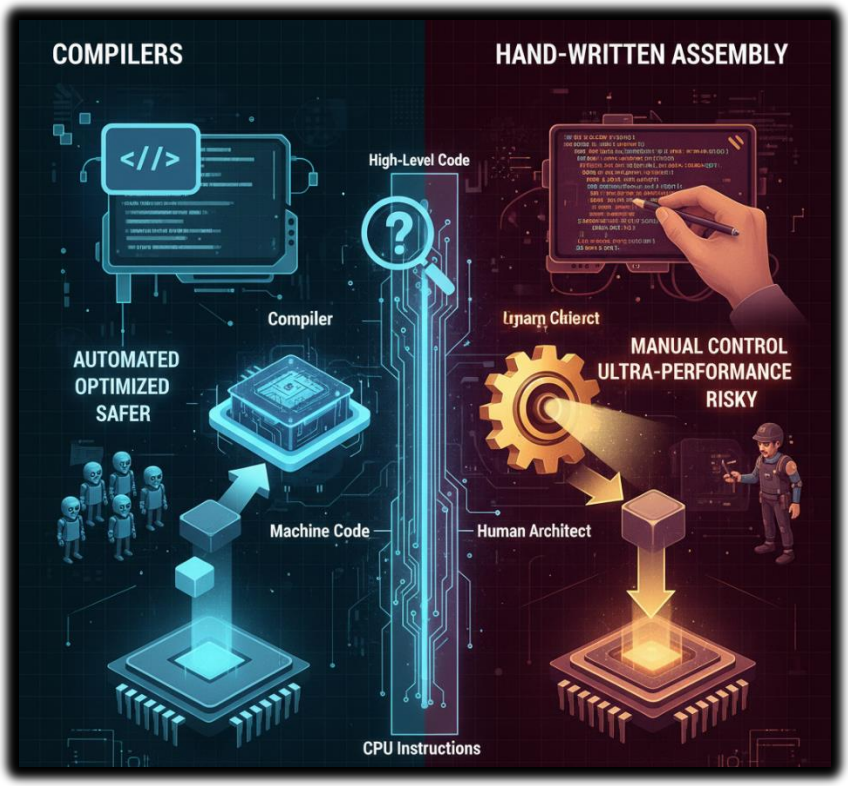
Some malware avoids detection by building values one byte at a time using 8-bit registers.

Since many antivirus signatures look for known 32-bit patterns, assembling instructions byte-by-byte can help malicious code slip past simple pattern matching.

There’s also a practical performance reason.

Using smaller registers can reduce instruction size and sometimes be slightly more efficient.

Compilers and hand-written assembly will still use 8-bit operations when they make sense.



3.7 Chapter Review

If EAX contains the value 0x12345678, the lower 16 bits are stored in AX, so AX would be 0x5678.

The lowest 8 bits are stored in AL, which would be 0x78.

You can't access a "high" or "low" 8-bit part of registers like ESI or EDI because only the core registers (A, B, C, and D) were designed with addressable 8-bit sub-registers.

Registers like ESI, EDI, EBP, and ESP must be accessed as 16-bit or 32-bit values.

When a function finishes running, its return value is placed in EAX.

If you want to know whether a function succeeded or failed, that's the register you check.

SPECIAL PURPOSE REGISTERS (x86)

1.1 What "Special Purpose" Really Means

Special-purpose registers are registers that exist to **control the CPU itself**, not just to hold random data. They:

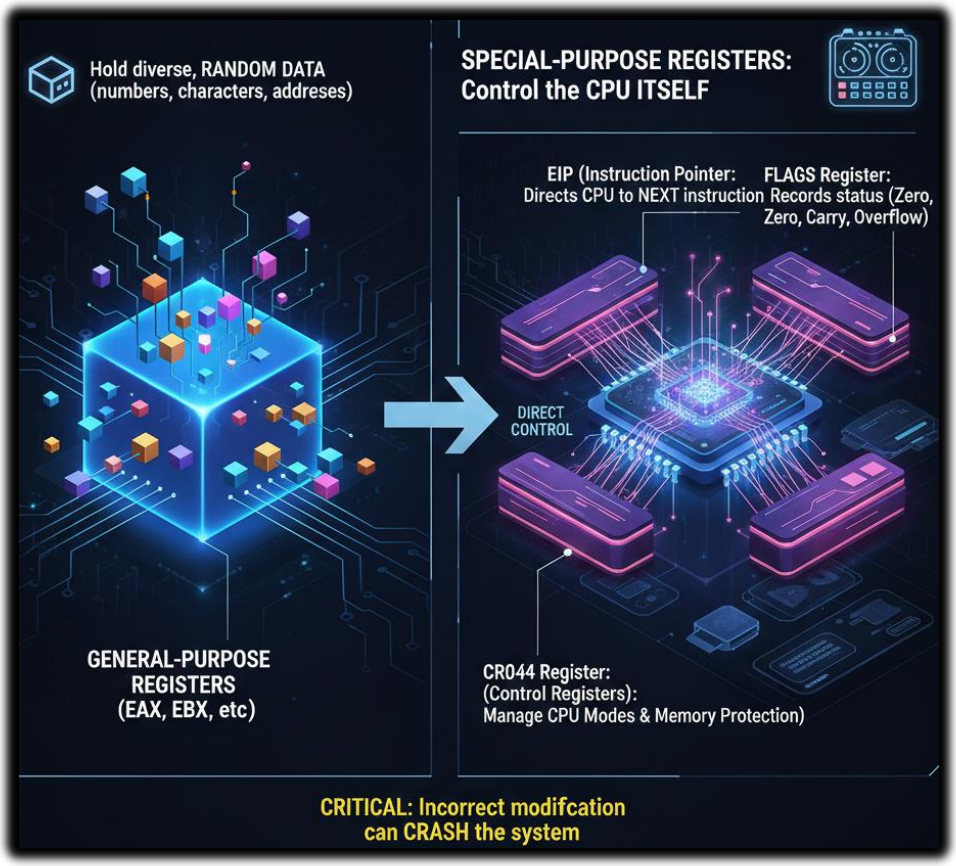
- Control **execution flow**
- Control **memory access**
- Track **CPU state**
- Enable **debugging, protection, and task switching**

Unlike general-purpose registers:

- Some of them **cannot be freely used**
- Some are **invisible to normal programs**
- Some can crash the system if misused

Think of them as: *"The steering wheel, brakes, and dashboard of the CPU"*

You don't casually play with them.



1.2 Instruction Pointer (IP / EIP) 🧭

What it is

The **Instruction Pointer** holds the memory address of the **next instruction** the CPU will execute.

- 16-bit mode → IP
- 32-bit mode → **EIP**
- 64-bit mode → RIP

You almost never write to EIP directly.

Instead, it changes through:

- JMP
- CALL
- RET
- Interrupts
- Exceptions

Example mental model:

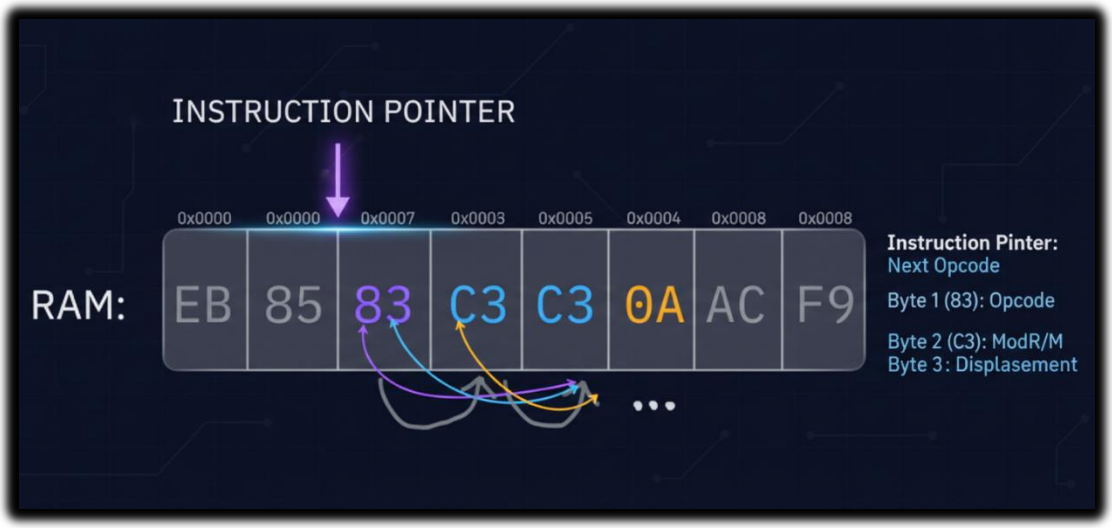
EIP is the CPU's "you are here" arrow.

If you control EIP:

- You control **execution**
- You control **program flow**
- You own the process

This is why exploits aim to:

Overwrite EIP



1.3 Flags Register (FLAGS / EFLAGS) 🚦

What it is

The Flags Register stores **status bits** describing what just happened in the CPU.

After arithmetic or logic:

- Was the result zero?
- Was there a carry?
- Was there an overflow?
- Was the result negative?

Common flags:

- ZF → Zero Flag
- CF → Carry Flag
- OF → Overflow Flag
- SF → Sign Flag

Example:

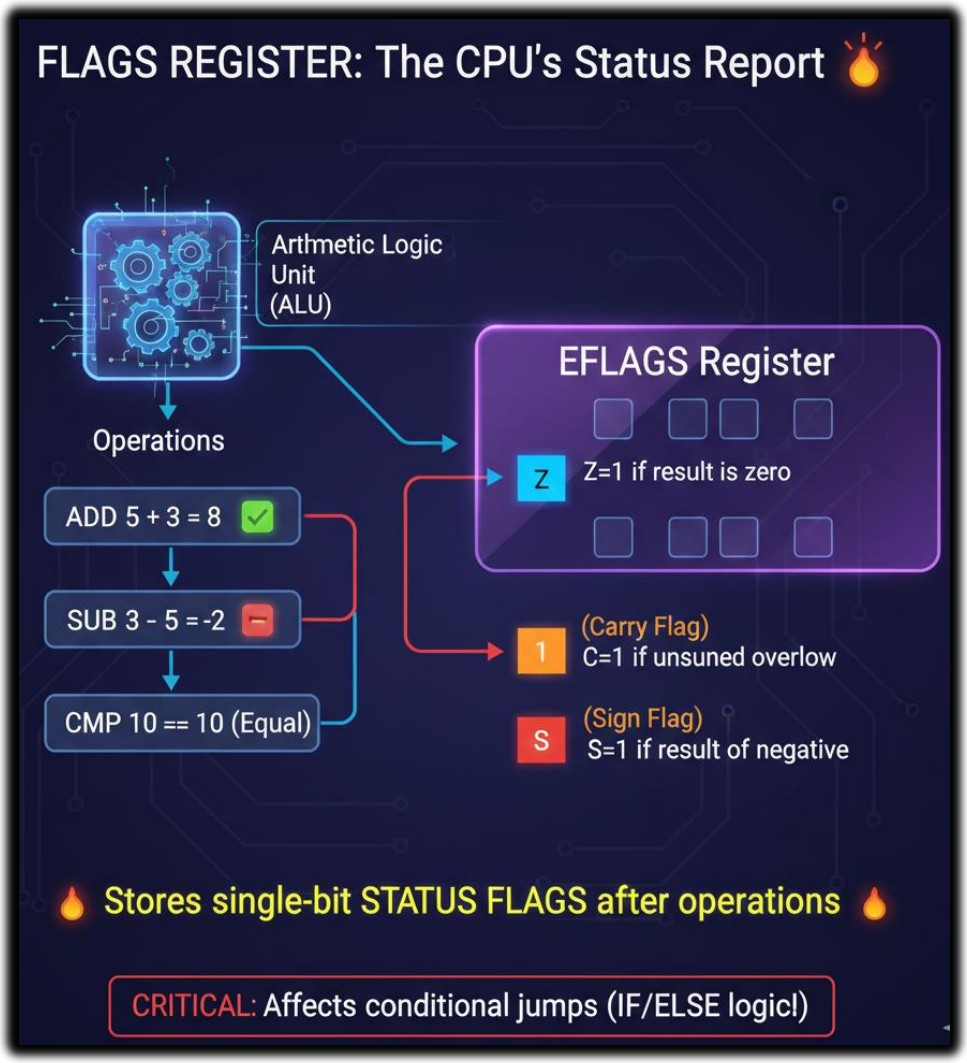
```
CMP EAX, EBX
JE equal
```

CMP does **not** jump.
It only sets flags.

JE checks:

"Is ZF = 1?"

Flags are the **decision-makers** behind conditional jumps.



1.4 Stack Pointer (SP / ESP)

What it is

ESP always points to the **top of the stack**.

The stack is used for:

- Function calls
- Return addresses
- Local variables
- Saved registers

Key rules:

- PUSH → ESP decreases
- POP → ESP increases

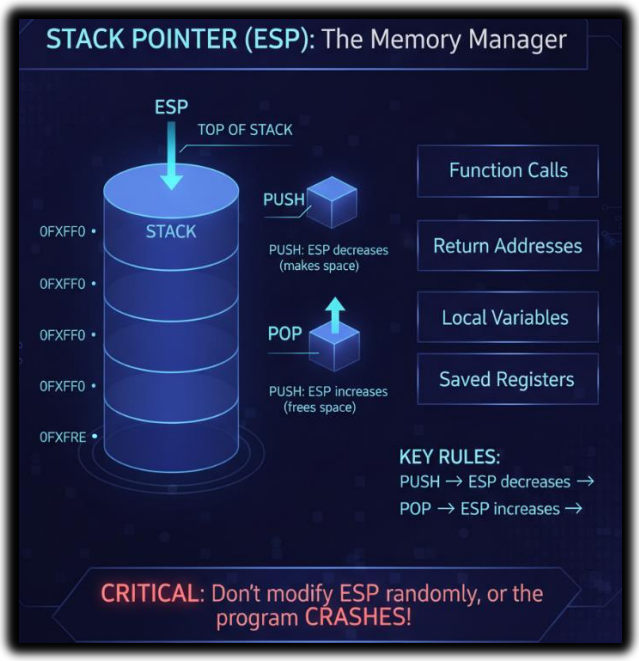
ESP is automatically modified by the CPU.

Important:

ESP is both **general-purpose in theory** and **special-purpose in reality**

You *can* use it as a normal register.
You *shouldn't* unless you're advanced.

Corrupt ESP → corrupted execution.



1.5 Base Pointer (BP / EBP)

What it is

EBP points to the **base of the current stack frame**.

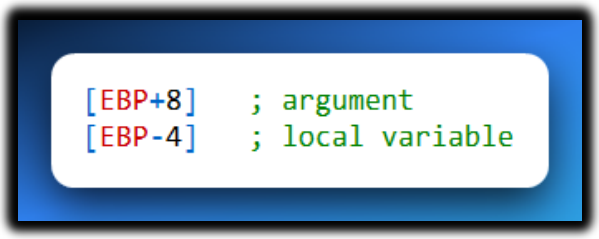
It provides:

- Stable access to function arguments
- Stable access to local variables

Unlike ESP:

- EBP usually stays fixed during a function

Example:



Why debuggers love EBP:

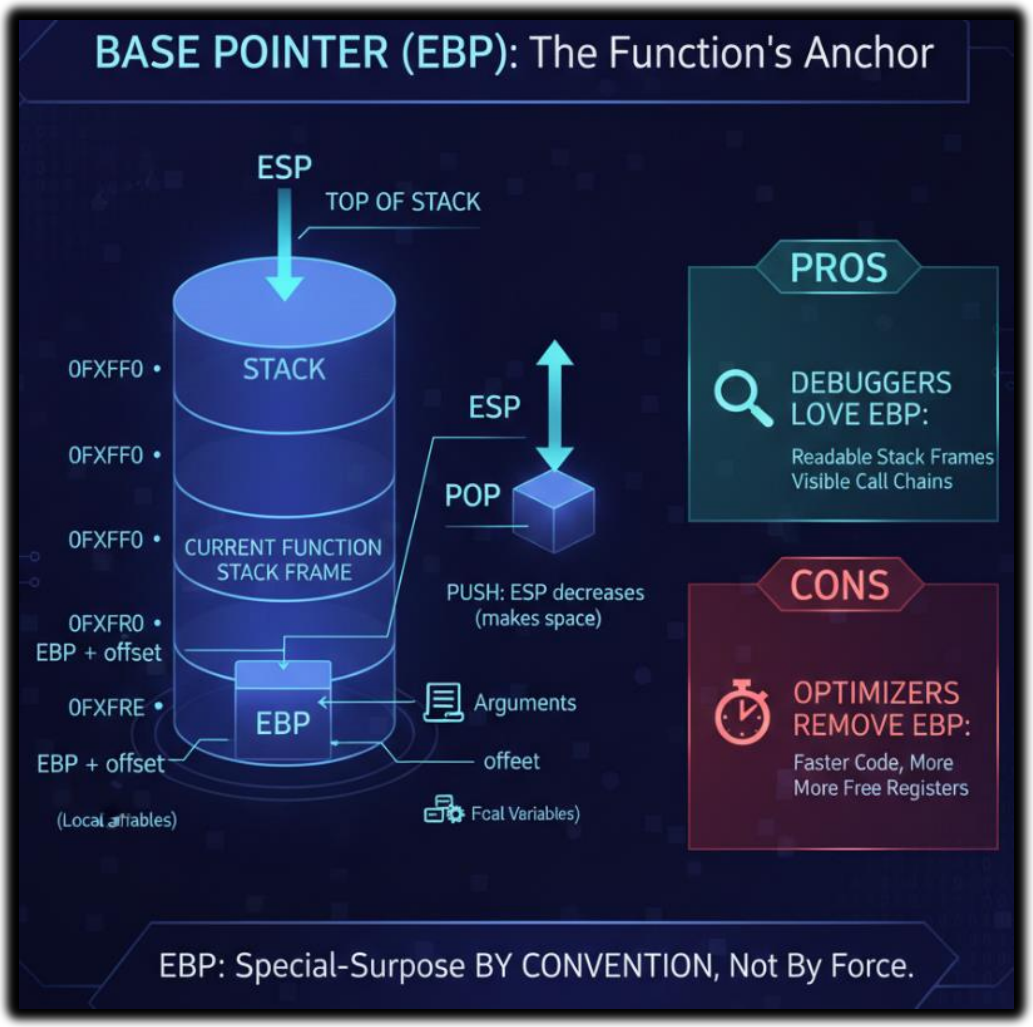
- Stack frames become readable
- Call chains become visible

Why optimizers remove it:

- One less register used
- Faster code

So yes:

EBP is special-purpose **by convention**, not by force.



1.6 Index Registers (SI, DI, ESI, EDI) 🔁

What they are: Index registers are optimized for **string and array operations**. They shine in: Memory copying, Memory comparison and Buffer processing.

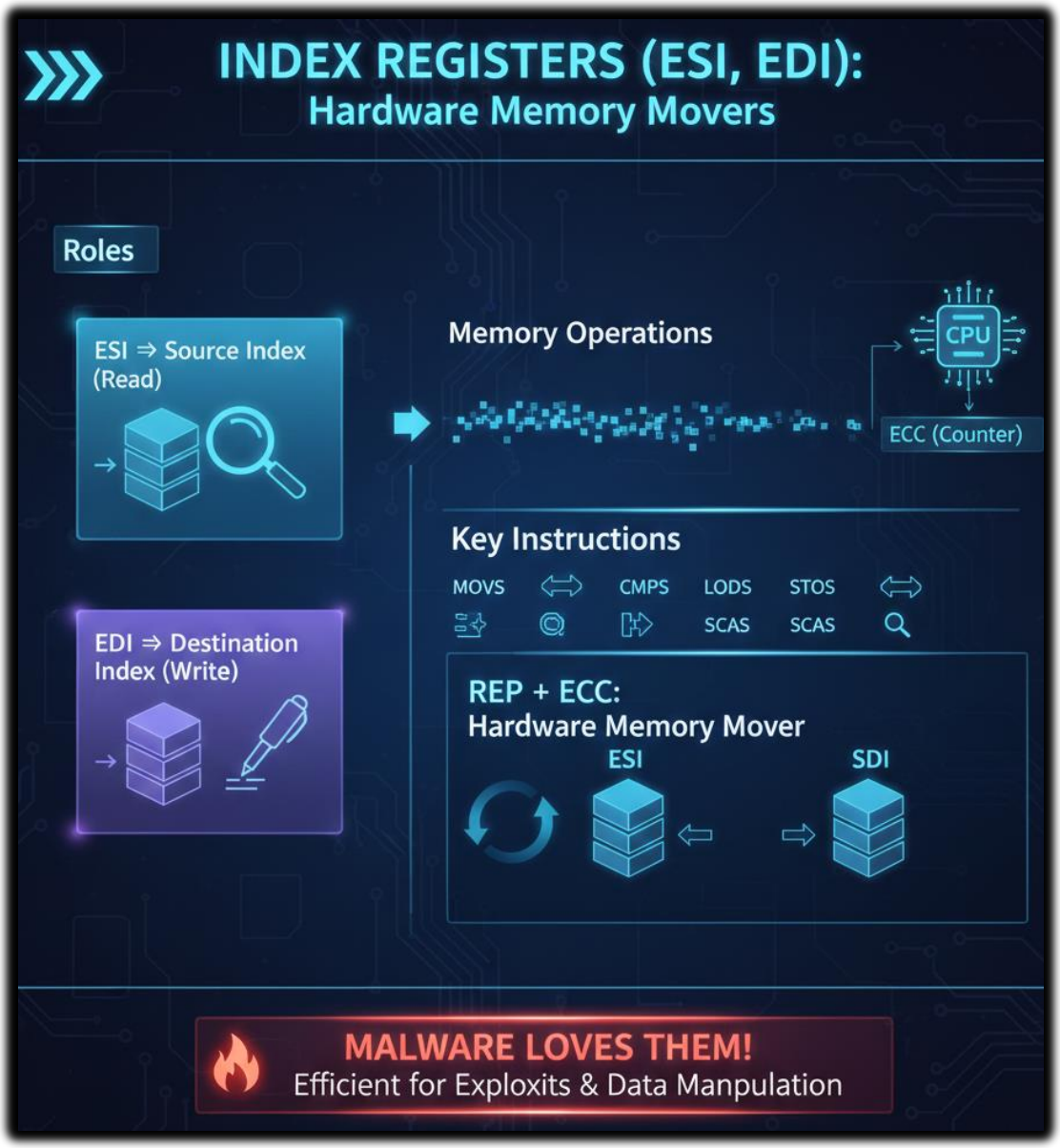
Roles:

- ESI → Source Index
- EDI → Destination Index

Used by instructions like:

- MOVS
- CMPS
- LODS
- STOS
- SCAS

With REP + ECX: The CPU becomes a **hardware memory mover**. This is why malware loves them.



1.7 Segment Registers (CS, DS, SS, ES, FS, GS) 🧠📦

What they are

Segment registers define **which segment of memory** you are accessing.

Historically:

- Memory was divided into segments
- Each segment had a base address

Key ones:

- CS → Code Segment
- DS → Data Segment
- SS → Stack Segment

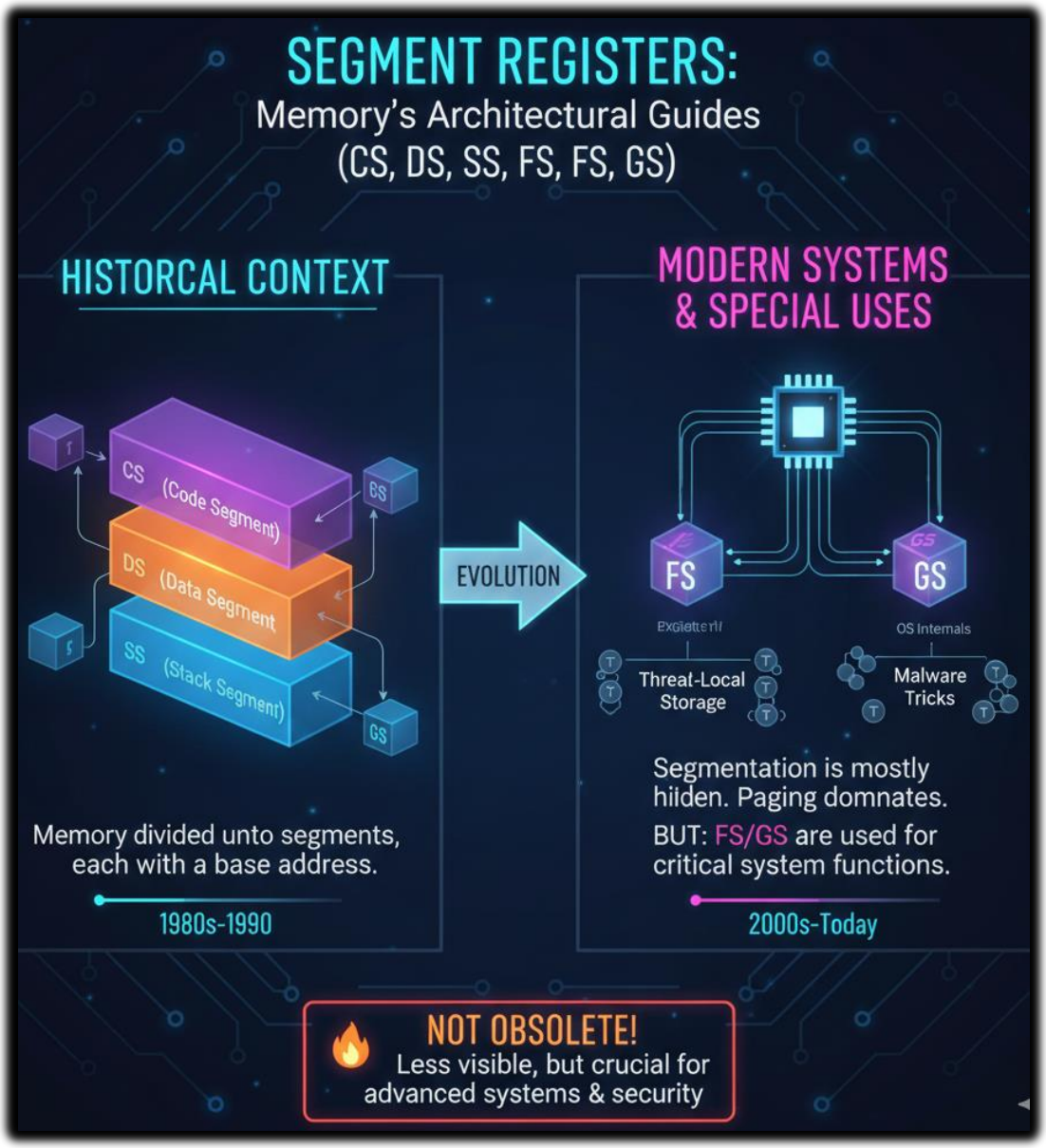
Modern systems:

- Segmentation is mostly hidden
- Paging dominates

But:

- FS / GS are still heavily used
- Thread-local storage
- OS internals
- Malware tricks

Segment registers are **not obsolete** — just less visible.



1.8 Control Registers (CR0, CR2, CR3, CR4, CR8) 💪

What they are

Control registers define **how the CPU operates**.

They control:

- Protected Mode
- Paging
- Virtual memory

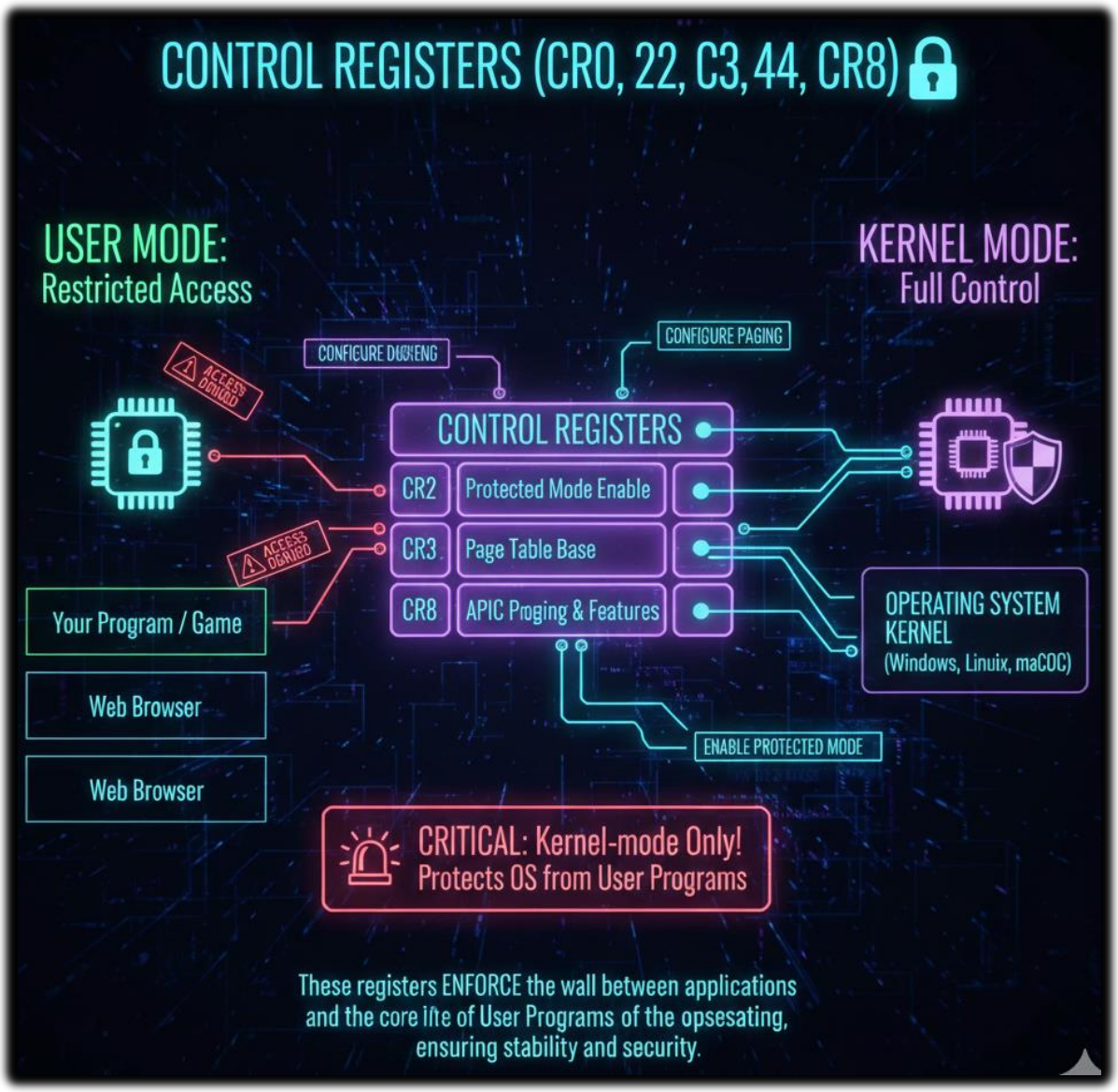
- Write protection
- Supervisor/User mode behavior

Examples:

- CR0 → enables protected mode
- CR3 → holds page directory base
- CR2 → stores faulting address (page fault)

Normal programs **cannot touch these**. Only kernel-mode code can.

These registers separate the user code from the operating system



1.9 Debug Registers (DR0–DR7) 🐞

What they are: Hardware debugging registers.

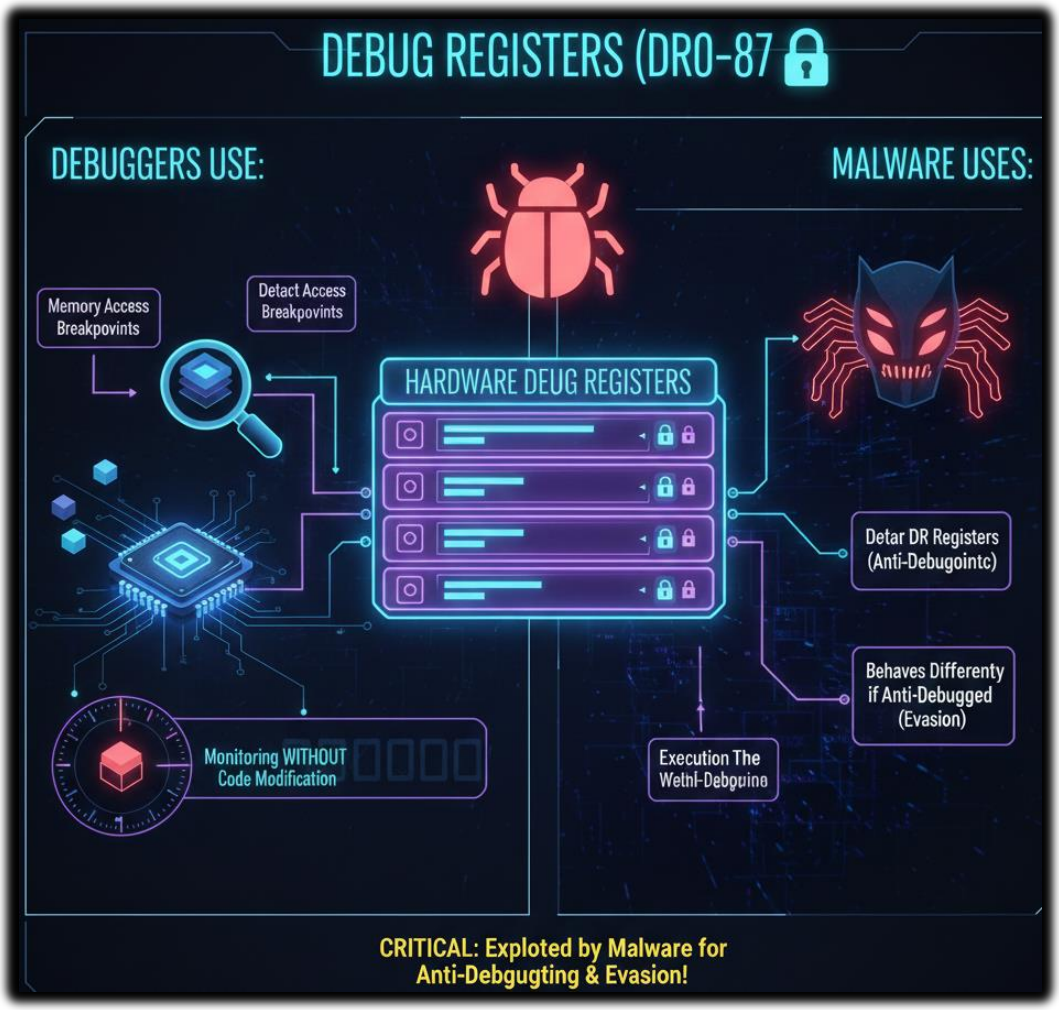
They allow:

- Breakpoints on memory access
- Breakpoints on execution
- Monitoring without code modification

Used by debuggers, anti-debugging checks and malware evasion.

Malware often:

- Detects DR registers
- Clears them
- Behaves differently if debugging is detected



1.10 Model-Specific Registers (MSRs) 🧪

What they are

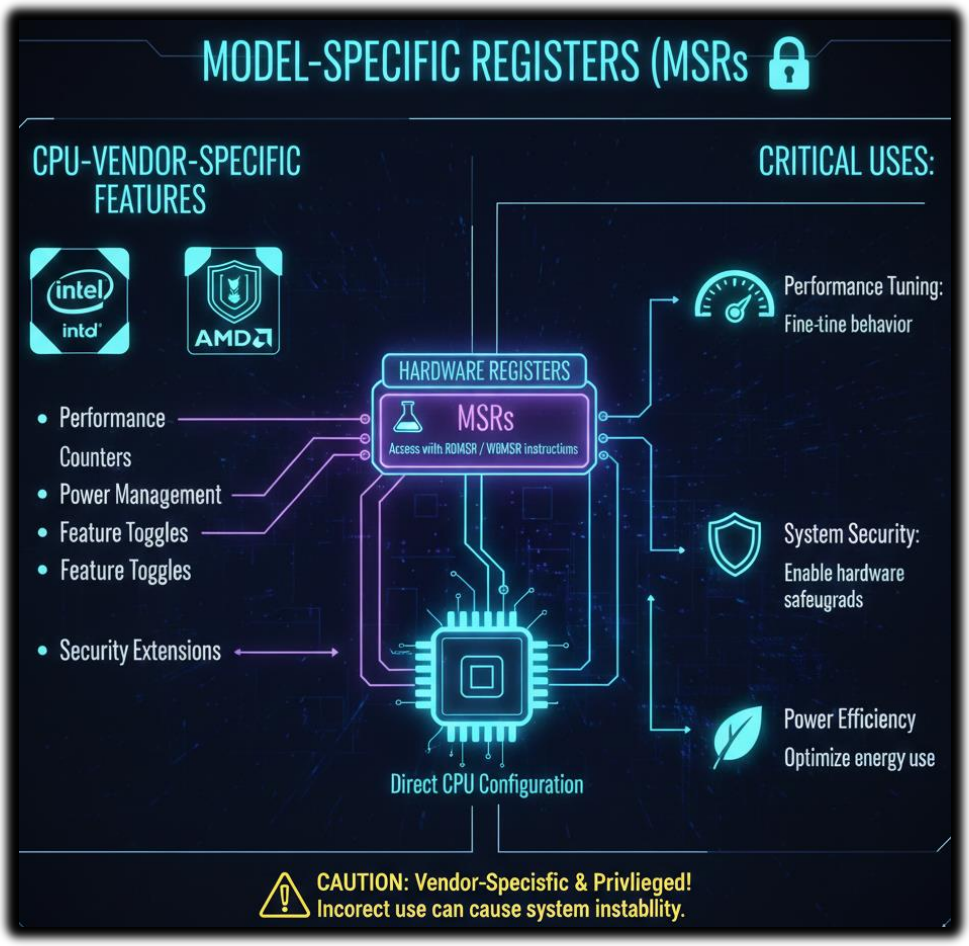
CPU-vendor-specific registers.

Used for:

- Performance counters
- Power management
- Feature toggles
- Security extensions

Accessed using: **RDMSR** and **WRMSR**

Highly privileged.
Highly dangerous.
Highly powerful.



1.11 Task Register (TR) 🔄

What it is

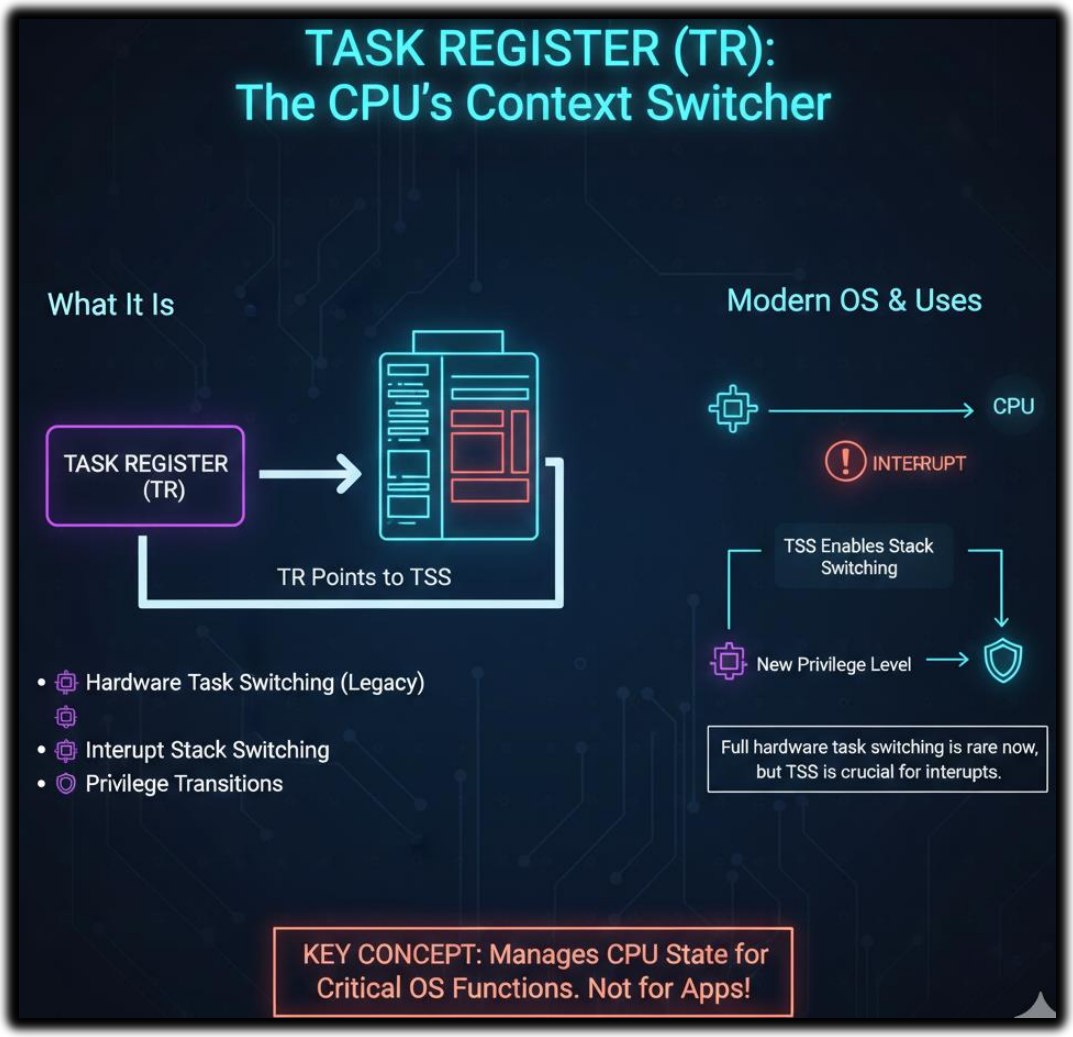
The Task Register points to the **Task State Segment (TSS)**.

Used in:

- Hardware task switching (mostly legacy)
- Interrupt stack switching
- Privilege transitions

Modern OSes:

- Rarely use full hardware task switching
- Still rely on TSS for interrupts



1.12 Local Descriptor Table Register (LDTR) 📋

What it is

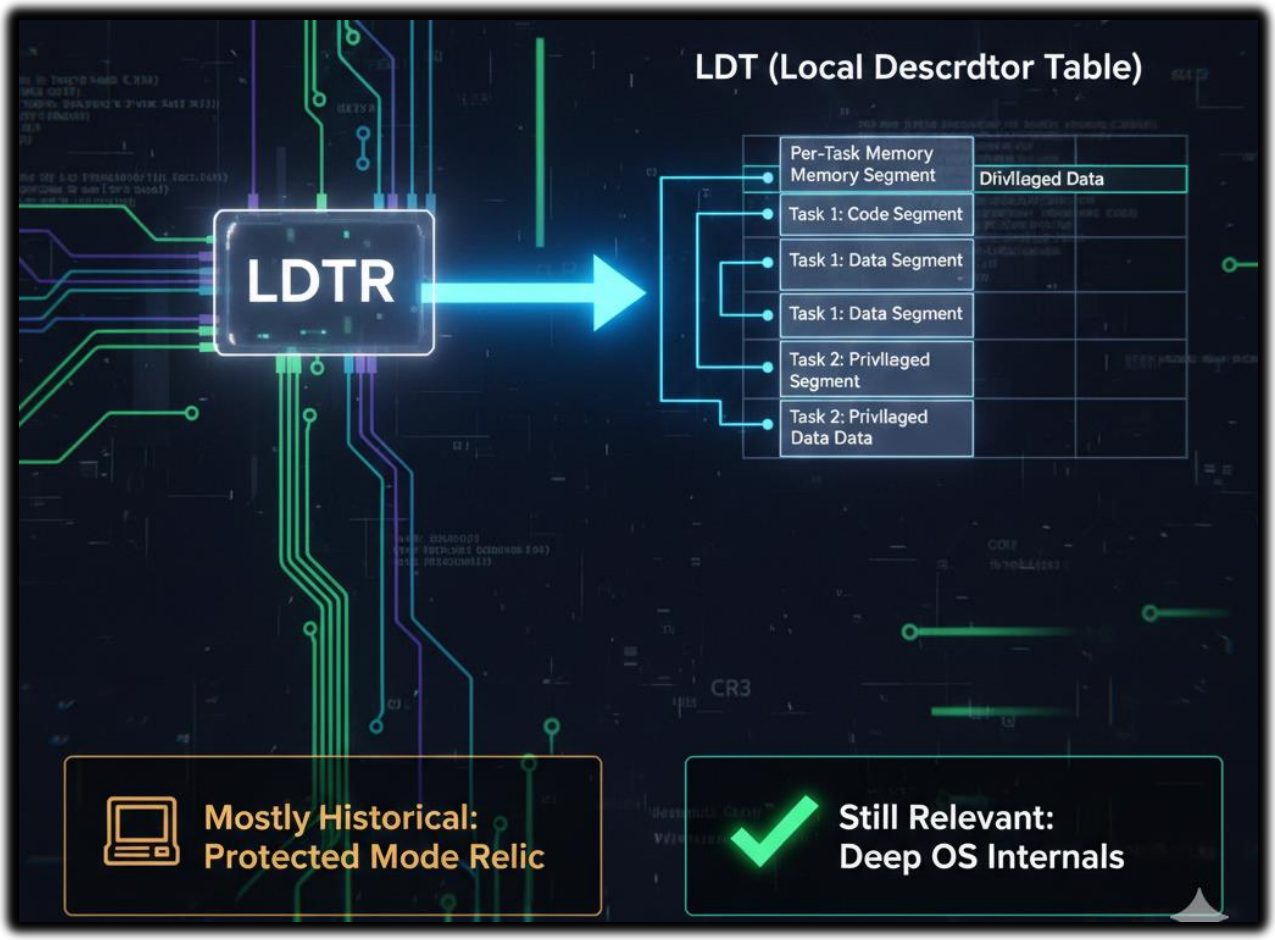
LDTR points to the **Local Descriptor Table**.

LDT:

- Defines per-task memory segments
- Used in protected mode

Mostly historical, but:

- Still exists
- Still relevant in deep OS internals



1.13 The ESP / EBP "Double Identity"

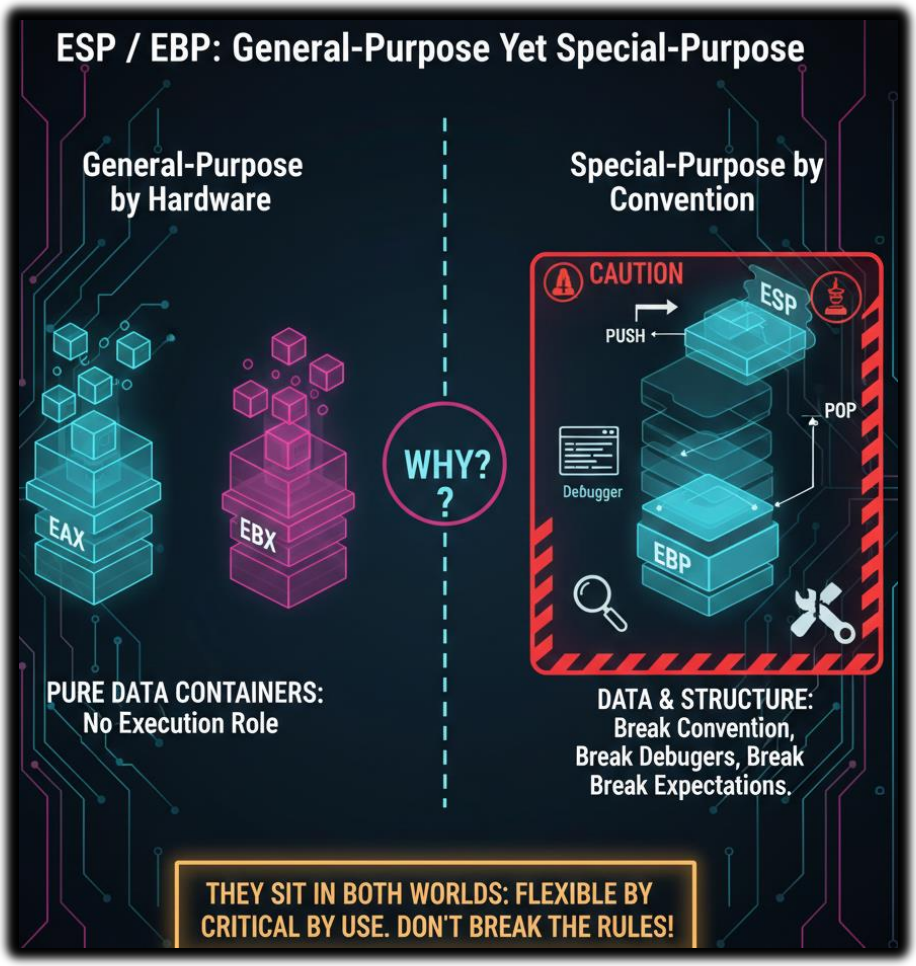
ESP and **EBP** live in two worlds at the same time. At the hardware level, they are just general-purpose registers.

The CPU doesn't magically protect them or force special rules on how they're used. In theory, you *could* use them like any other register.

In practice, they have special meaning because of convention. Compilers, debuggers, and operating systems all expect ESP to track the stack and EBP to describe the current stack frame. When code follows these conventions, tools work correctly and stack traces make sense.

When those conventions are broken, everything starts to fall apart. Debuggers lose track of variables, stack walking fails, and even simple analysis becomes confusing or impossible.

Registers like EAX or EBX don't have this problem. They are just containers for data and have no built-in structural role. ESP and EBP are different because they don't just hold values — they describe the shape of the program's execution. That's what gives them their "double identity."



1.14 “E”, “R”, and the Evolution of Register Names

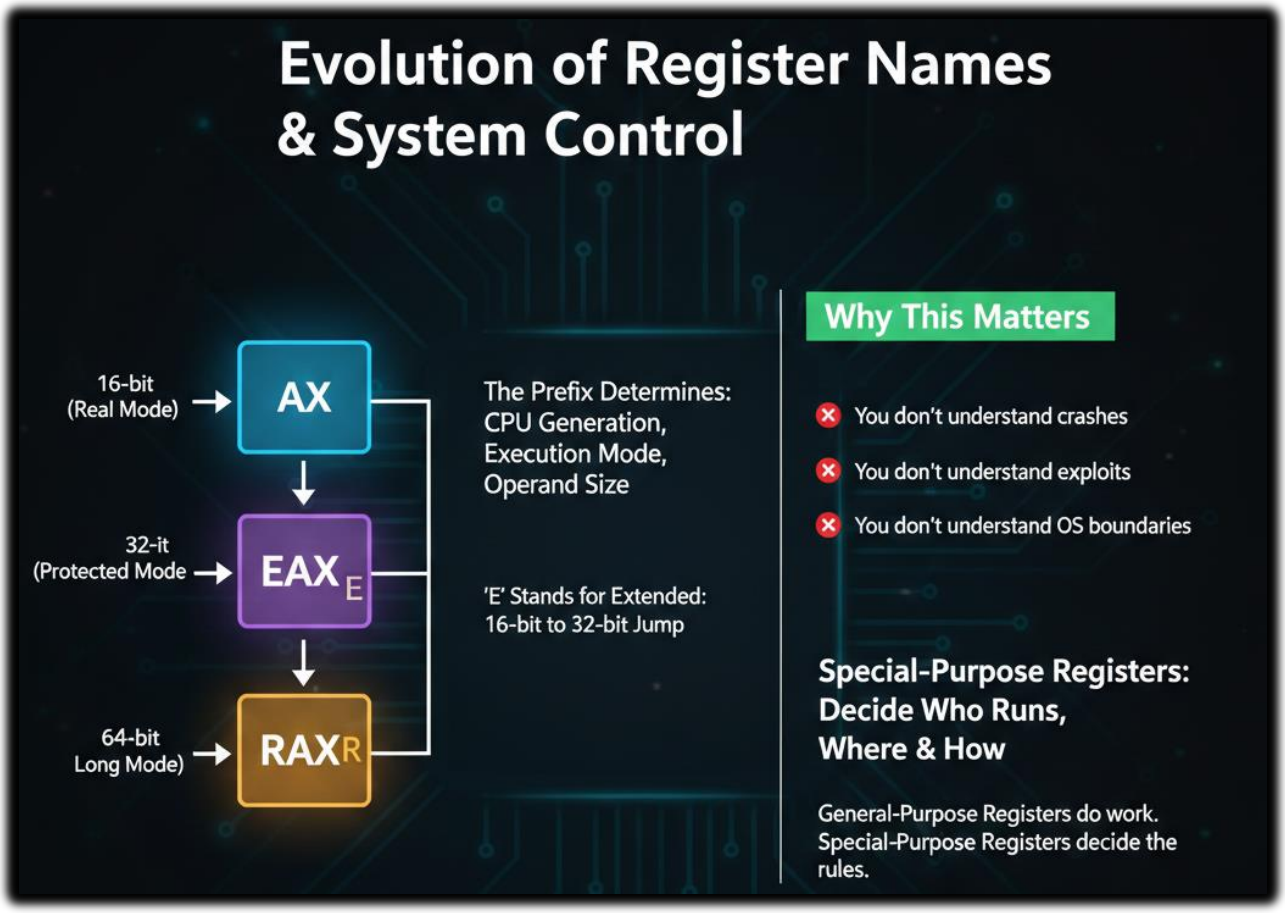
Register names changed as CPUs grew wider over time. Originally, registers like AX were 16-bit and used in early real-mode systems.

When x86 moved to 32-bit protected mode, those same registers were extended, and an **E** was added to the name. That’s where EAX comes from — the **E** simply means *extended*.

Later, when 64-bit CPUs were introduced, the registers were extended again and given an **R** prefix instead. AX became RAX, and the same pattern applies to the other core registers.

Because of this, the prefix on a register name tells you a lot at a glance. It hints at the CPU generation, the execution mode the code is running in, and the size of the values the register can hold.

Once you know that, the naming scheme stops feeling arbitrary and starts feeling very deliberate.



1.15 Why This Chapter Matters

If you don't understand special-purpose registers:

- You don't understand crashes
- You don't understand exploits
- You don't understand OS boundaries

These registers:

Decide *who* runs, *where*, and *how*

General-purpose registers do work.
Special-purpose registers **decide the rules**.

SEGMENT REGISTERS (x86) 

2.1 Why Segment Registers Exist (Historical Truth)

Segment registers exist because early x86 CPUs could **not directly address large memory**.

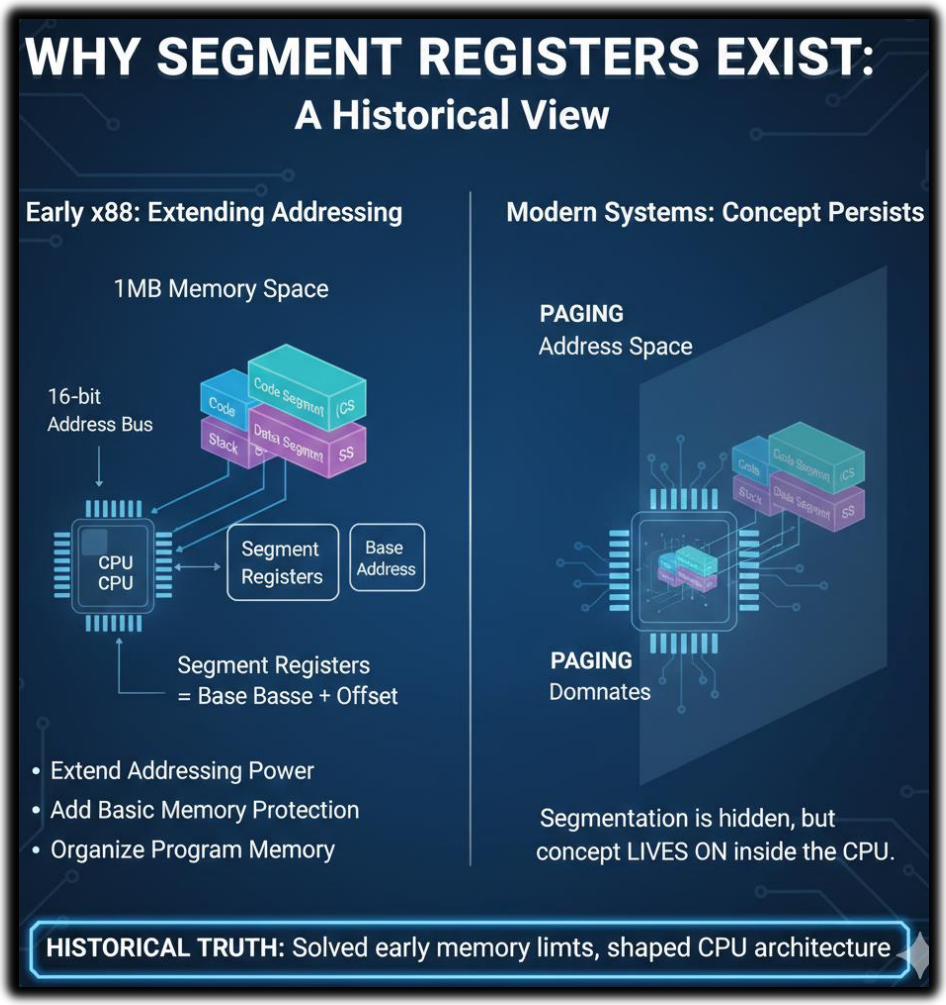
Instead of one flat address space, memory was divided into **segments**:

- Code lived in one place
- Data lived in another
- Stack lived somewhere else

Segment registers were created to:

Extend addressing power
Add basic memory protection
Organize program memory

Even though modern systems hide segmentation, the **concept still lives inside the CPU**.



2.2 What Segment Registers Actually Hold

Segment registers **do not hold raw memory addresses** (in modern modes).

They hold **segment selectors**.

A **segment selector** is a small value that:

- Points to an entry in a table
- That entry describes a memory segment

Think of it like this: Segment register → index → descriptor → real memory info

So:

- Segment register = “which segment?”
- Descriptor = “where is it and what are the rules?”

2.3 Segment Registers in Real Mode (16-bit World)

In **real-address mode**:

- Segment registers are **16 bits**
- They directly represent memory segments

Registers used:

- CS → Code Segment
- DS → Data Segment
- SS → Stack Segment
- ES → Extra Segment

Address calculation: Physical Address = Segment × 16 + Offset

Example: CS = 0x1234 and IP = 0x5678

Physical Address: 0x1234 × 16 + 0x5678

This is:

- Simple
- Fast
- Unsafe
- No real protection

This is why real mode is called **real-address mode**.

2.4 Protected Mode: Segments Become Descriptors 🔐

In **protected mode**, everything changes.

Segment registers:

- Still exist
- Still matter
- But no longer point directly to memory

Instead, they hold **selectors** that reference:

- Global Descriptor Table (GDT)
- Local Descriptor Table (LDT)

Each descriptor defines:

- Base address
- Limit (size)
- Access rights
- Privilege level

So, the CPU now says:

"Before I touch memory, check the rules."

This is where **protection** comes from.

2.5 The Six Segment Registers in Protected Mode

Protected mode uses **six segment registers**:

CS — Code Segment

- Points to executable code
- Controls instruction fetch
- Enforces execution permissions

Changing CS:

- Usually requires a far jump or call
- Changes execution context

DS — Data Segment

- Default segment for data access
- Used when reading or writing variables

Most memory instructions implicitly use DS unless overridden.

SS — Stack Segment

- Defines where the stack lives
- Used with ESP / EBP

SS violations:

- Cause immediate faults
- Are tightly protected

ES — Extra Segment

- Additional data segment
- Commonly used with string operations

Often paired with EDI.

FS and GS — Special Purpose Segments

FS and GS were added later.

File Segment or **Fiber/Thread Segment** (used for Thread Information Block/Thread Local Storage).

General Segment or **Graphics Segment** (used for CPU-specific data or OS-defined structures in 64-bit Windows/Linux).

They are commonly used for:

- Thread-Local Storage (TLS)
- Per-thread data
- OS internals

Modern examples:

- Windows → FS or GS for TEB
- Linux → GS for per-CPU data

These are **very important in reverse engineering**.

2.6 Base Address and Limit: Memory Boundaries

Each segment descriptor defines:

- **Base** → where the segment starts
- **Limit** → how big it is

Why this matters:

- Access beyond limit → fault
- Access with wrong privilege → fault

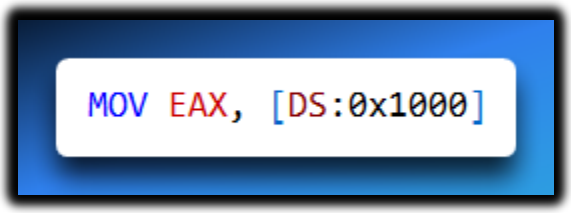
This is early memory safety.

Segments enforce:

"You may access THIS memory, and nothing else."

2.7 Logical Address → Physical Address (Step-by-Step)

When a program accesses memory:



The CPU:

1. Reads DS selector
2. Looks up descriptor in GDT/LDT
3. Gets base address
4. Adds offset (0x1000)
5. Checks limit and permissions
6. Produces linear address

Later, paging may translate that to physical memory

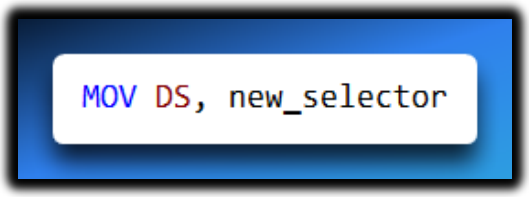
Segments happen **before paging**.

2.8 Modifying Segment Registers ⚠

Segment registers can be modified using:

- MOV
- PUSH
- POP

Example:



But important:

Loading a segment register does **not** directly change memory

It triggers:

- Descriptor lookup
- Validation
- Caching inside the CPU

If the selector is invalid:

- General Protection Fault occurs

2.9 Why Changing a Segment Register Feels "Delayed"

When you load a segment register:

- The CPU reads the descriptor
- Stores its info internally
- Uses that cached data for future access

So, memory behavior changes **after descriptor resolution**, not immediately.

This is why:

- Segment changes feel indirect
- Debugging segmentation is tricky

2.10 Segment Registers vs Paging (Modern Reality)

Modern systems:

- Use flat segments (base = 0)
- Rely on paging for protection

But:

- Segment registers still exist
- FS / GS are actively used
- CS still controls privilege

Segmentation is not dead.
It's just **quiet**.

2.11 Why This Chapter Matters (ASM + RE Perspective)

If you ignore segmentation:

- You misunderstand OS internals
- You miss TLS tricks
- You misread memory accesses

Segment registers explain:

- Why exploits jump through hoops
- Why kernel/user boundaries exist
- Why certain memory reads behave "magically"

Segments define:

Where you're allowed to be
What you're allowed to touch

Paging decides *where memory lives*.
Segmentation decides *who may touch it*.

2.12 Lock-In Questions

1. Why does protected mode use descriptors instead of raw addresses?
2. Why are FS and GS still relevant today?
3. Why does segmentation occur before paging?
4. Why does loading DS not immediately change memory behavior?

INSTRUCTION POINTER (IP / EIP) 🧭🔥

3.1 What the Instruction Pointer Really Is

The **Instruction Pointer** is the register that tells the CPU:

"This is the address of the next instruction to execute."

In 32-bit x86:

- The instruction pointer is called **EIP**
- It is **32 bits wide**

Naming across modes:

- Real mode → IP (16-bit)
- Protected mode → **EIP (32-bit)**
- Long mode → RIP (64-bit)

If you understand EIP, you understand **program flow**.

3.2 How the CPU Uses EIP (Fetch–Execute Reality)

Execution follows a simple loop:

1. CPU reads EIP
2. Fetches instruction from memory at EIP
3. Decodes the instruction
4. Executes it
5. Updates EIP to point to the next instruction

Important detail: EIP is **not incremented by a fixed value**

Instruction length varies:

- Some instructions are 1 byte
- Some are many bytes

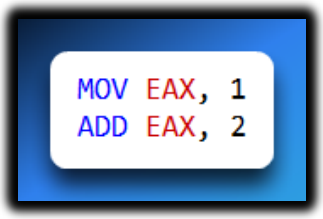
The CPU figures this out during decoding.

3.3 How EIP Changes Normally

Most of the time, EIP:

- Automatically moves forward
- You never touch it directly

Example:



EIP simply walks forward through memory.

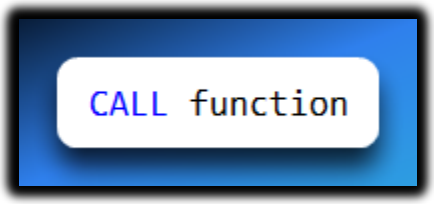
3.4 Instructions That Modify EIP Explicitly 🔀

Some instructions **change execution flow** by modifying EIP indirectly.

JMP is an unconditional jump. When the CPU executes it, it stops following the normal instruction order and loads the target address directly into **EIP**. Execution then continues from that new location, no questions asked.

CALL transfers execution to a function. Before the jump happens, the CPU pushes the address of the next instruction onto the stack.

Then it loads EIP with the function's address. This saved address is what allows the function to return to the correct place when it finishes.



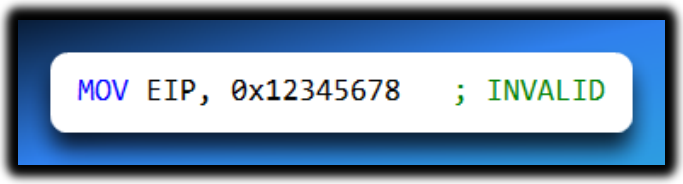
RET returns execution to the caller. The CPU pops an address off the stack and loads it into **EIP**. Execution resumes at that location.

If the stack has been corrupted, the value popped is wrong. That means **RET** jumps to a random address, and execution goes straight into chaos.

INT triggers a software interrupt, like INT 0x80 in Linux. When the CPU executes it, it automatically changes **EIP** to point to the interrupt handler. The CPU may also switch privilege levels, depending on the interrupt.

3.5 Can You Directly Modify EIP?

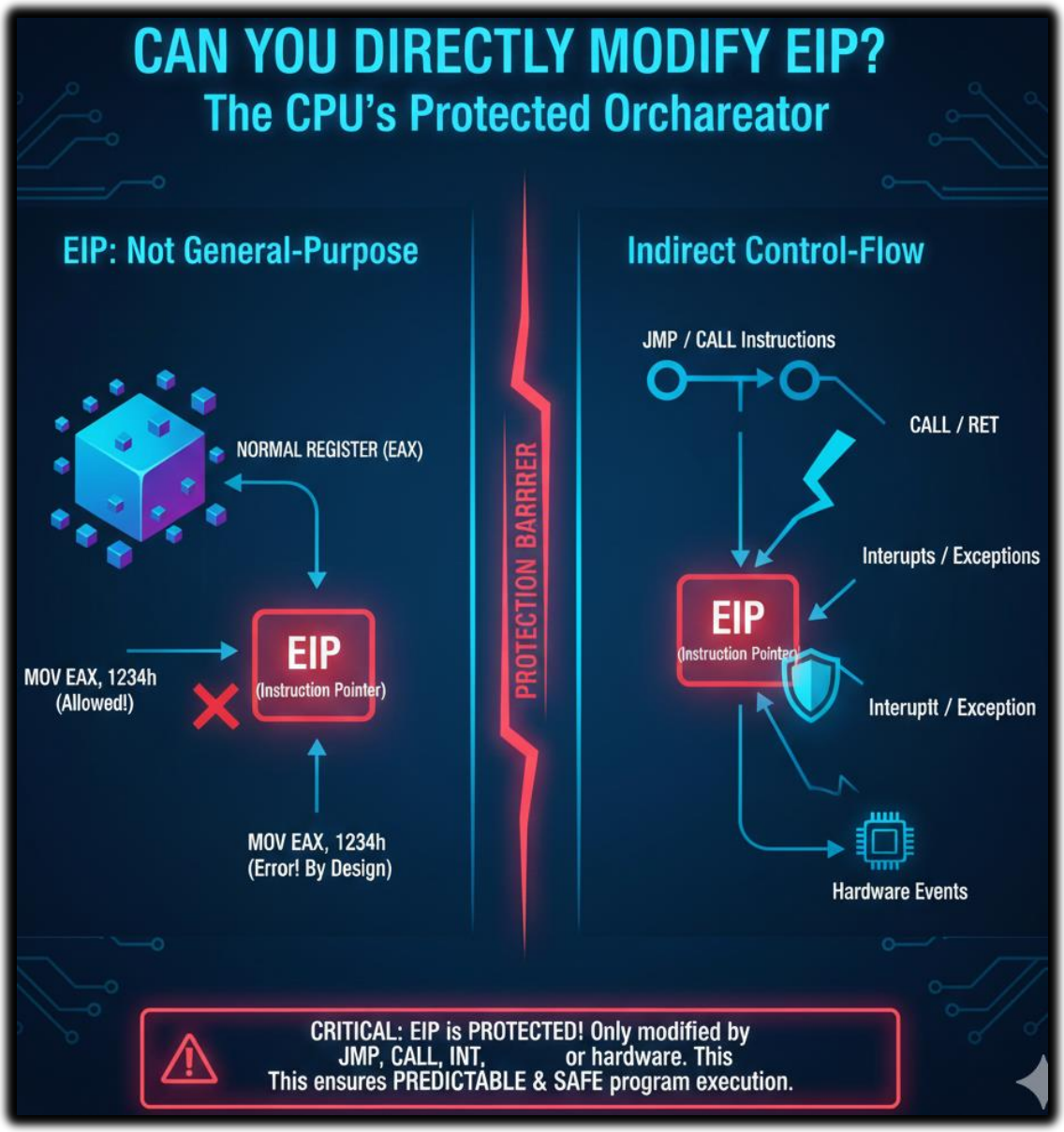
No, you can't just write a value into **EIP** like a normal register:



EIP isn't a general-purpose register.

You can only change it indirectly — through control-flow instructions like JMP or CALL, through interrupts or exceptions, or by certain hardware events.

This is by design. The CPU protects EIP to make sure program execution stays predictable and safe.



3.6 EIP and EFLAGS Are NOT the Same ❌

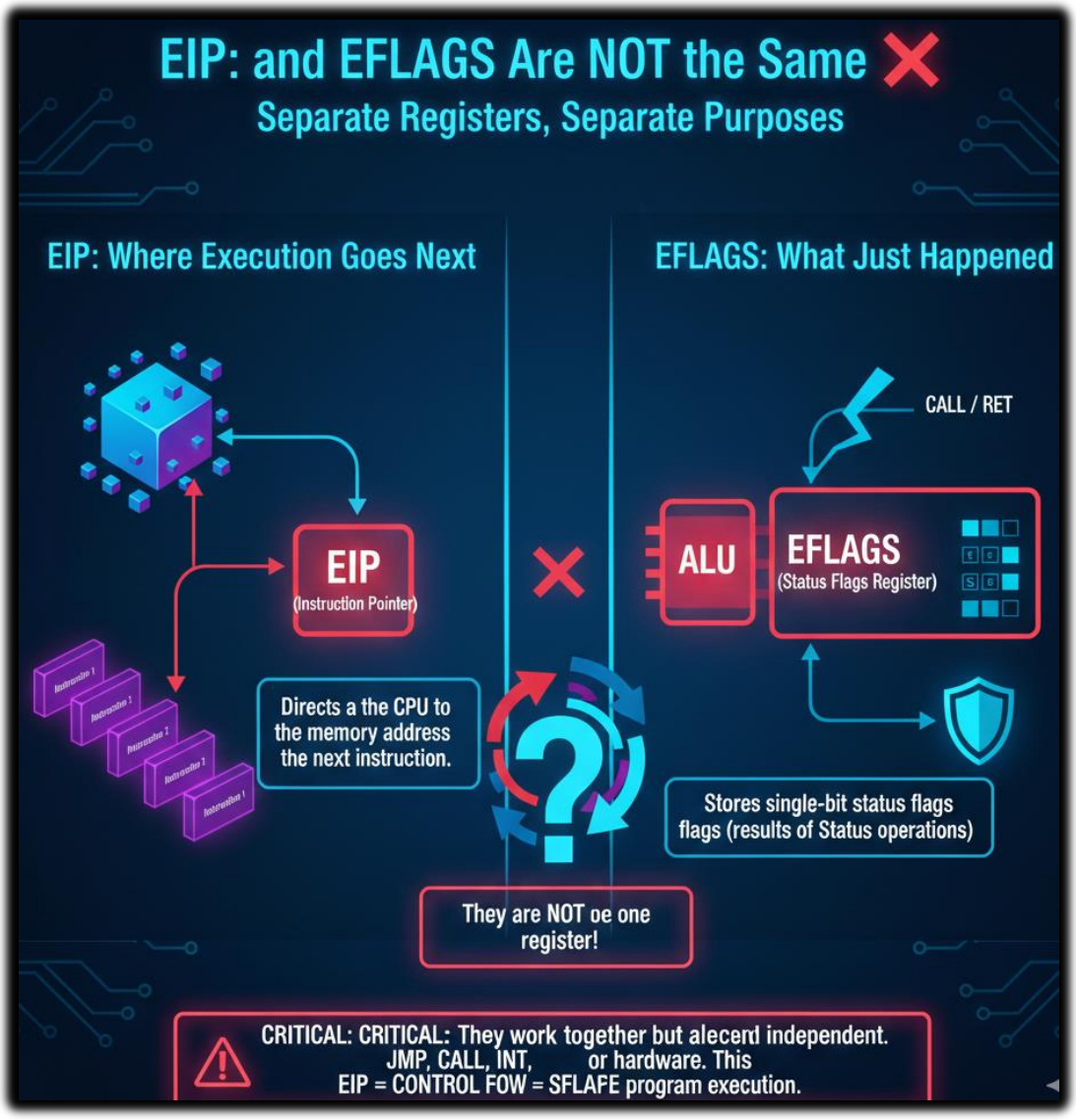
Let's fix the earlier mistake clearly.

- **EIP** → where execution goes next
- **EFLAGS** → what just happened

They are:

- Separate registers
- Separate purposes
- Updated independently

They work together, but they are not one register.



3.7 Why EIP Is the Crown Jewel 👑

If an attacker controls:

- EAX → they control data
- ESP → they control stack behavior
- **EIP → they control the program**

This is why:

- Buffer overflows target return addresses
- Exploits aim to overwrite EIP
- ROP chains exist

EIP control = **execution control**

3.8 EIP in Reverse Engineering 🔍

When reversing:

- Watch EIP to understand flow
- Breakpoints trigger on EIP values
- Call graphs are EIP graphs

Debuggers live and breathe EIP.

3.9 What Happens on Crashes

Common crash message: **"Access violation at EIP = 0x41414141"**

That tells you:

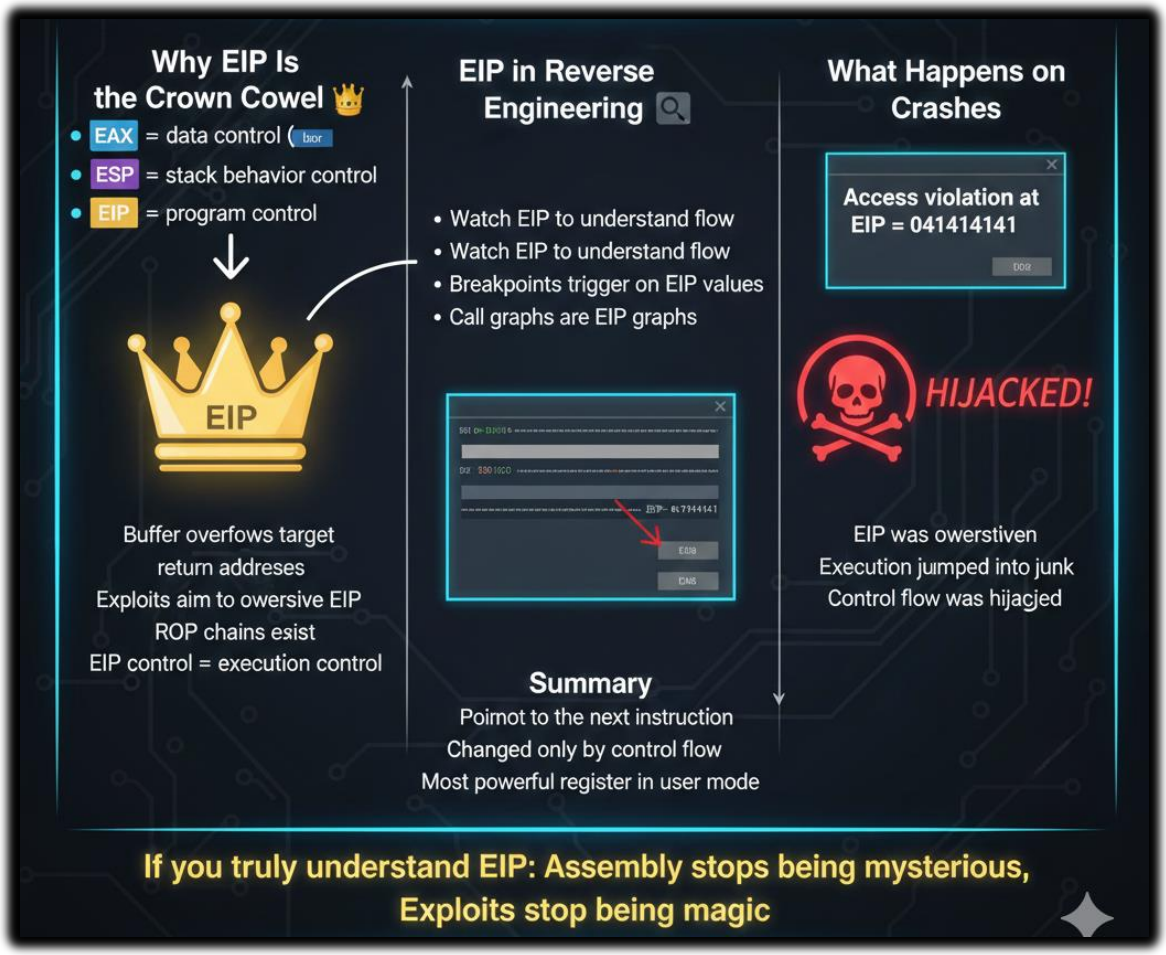
- EIP was overwritten
- Execution jumped into junk
- Control flow was hijacked

This is not random. It's mechanical.

3.10 Summary

EIP:

- Points to the next instruction
- Cannot be written directly
- Is changed only by control flow
- Is the most powerful register in user mode



3.11 Lock-In Questions

1. Why can't EIP be modified with MOV?
2. How does CALL know where to return?
3. Why is EIP control more powerful than EAX control?
4. Why is instruction length important for EIP updates?

FLAGS REGISTER (FLAGS / EFLAGS)

4.1 What the Flags Register Really Is

The **FLAGS / EFLAGS register** is a **32-bit register** that stores small 1-bit values called **flags**.

Each flag answers a question like:

- Did the result become zero?
- Was there a carry?
- Did an overflow happen?
- Should interrupts be handled?
- Should execution pause after each instruction?

Think of EFLAGS as:

The CPU's **status dashboard + control panel**

It both:

- Reports what just happened
- Controls how the CPU behaves next

4.2 Status Flags vs Control Flags (Clear Separation)

EFLAGS contains **two main kinds of flags**:

1. Status Flags (Results of operations)

These are set automatically by the CPU after math or logic instructions.

2. Control Flags (CPU behavior)

These affect **how the CPU runs**, not math results.

This distinction is important.

4.3 Core Status Flags (Arithmetic & Logic Results)

I. Carry Flag (CF)

Set when:

- An addition produces a carry
- A subtraction produces a borrow

Used for:

- Multi-precision arithmetic
- Unsigned math
- BCD operations

CF answers:

“Did the operation overflow unsigned range?”

II. Parity Flag (PF)

Set when:

- The **lowest byte** of the result has an even number of 1 bits

Used historically for:

- Error checking
- Hardware communication

Today:

- Rarely used directly
- Still updated by the CPU

III. Auxiliary Carry Flag (AF)

Set when:

- A carry occurs between bit 3 and bit 4

Used for:

- BCD arithmetic
- Legacy instructions

You'll mostly see AF in:

- Old code
- Emulators
- CPU internals

IV. Zero Flag (ZF)

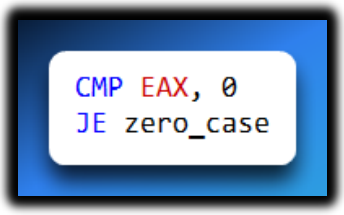
Set when:

- Result equals zero

Used constantly for:

- Equality checks
- Loop termination
- Null pointer tests

Example:



ZF is the backbone of conditional jumps.

V. Sign Flag (SF)

Reflects:

- The most significant bit of the result

If SF = 1:

- Result is negative (signed)

Used for:

- Signed comparisons
- Signed branching

VI. Overflow Flag (OF)

Set when:

- Signed arithmetic exceeds representable range

Example:

- 127 + 1 (8-bit signed) → overflow

OF answers:

"Did this operation overflow **signed** range?"

This is different from CF.

4.4 Control Flags (CPU Behavior Switches) 🔧

I. Direction Flag (DF)

Controls:

- Direction of string operations

DF = 0:

- Forward (low → high memory)

DF = 1:

- Backward (high → low memory)

Affects instructions like:

- MOVS
- STOS
- CMPS

This flag is often:

- Set intentionally
- Cleared immediately after use

Leaving DF set accidentally causes chaos.

II. Interrupt Flag (IF)

Controls:

- Whether the CPU responds to **external interrupts**

IF = 1:

- Interrupts enabled

IF = 0:

- Interrupts disabled

Used by:

- OS kernels
- Critical sections
- Interrupt handlers

Normal user programs:

Usually cannot freely change IF

III. Trap Flag (TF) 🐞

TF enables:

- **Single-step execution**

When TF = 1:

- CPU triggers an interrupt after **every instruction**

This is how:

- Debuggers step through code
- Instruction-by-instruction tracing works

Why it's powerful:

- Complete visibility
- Complete slowdown

Malware may:

- Detect TF
- Clear it
- Alter behavior

4.5 EFLAGS Is Not "One Control Flag" ❌

There is **no flag called "Control Flag"**.

Instead:

- EFLAGS contains **many independent flags**
- Some control behavior
- Some report status

Calling EFLAGS a *control register* is **conceptual**, not literal.

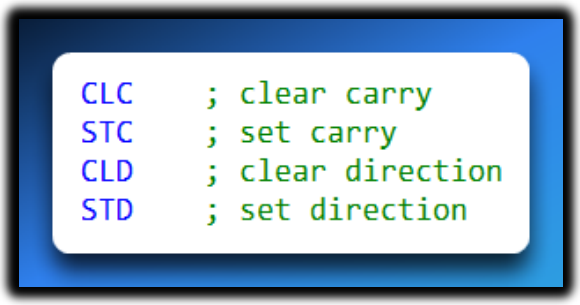
Precision matters here.

4.6 How Programs Read and Modify Flags

Flags are:

- Set automatically by the CPU
- Read by conditional jumps
- Modified by specific instructions

Examples:



You usually:

- Don't write flags directly
- Let instructions update them naturally

4.7 Flags and Conditional Branching 🔀

Most jumps depend on flags:

INSTRUCTION	CONDITION TO JUMP	MEANING
JE / JZ	ZF = 1	Jump if Equal / Zero
JNE / JNZ	ZF = 0	Jump if Not Equal / Not Zero
JC	CF = 1	Jump if Carry Flag is set
JO	OF = 1	Jump if Overflow Flag is set
JS	SF = 1	Jump if Sign Flag is set (Negative)

This means flags are the **decision engine** of the CPU

4.8 Flags in Reverse Engineering & Exploits 🔍

When reversing:

- Watch flags after CMP
- Understand which jump depends on which flag
- Spot fake comparisons

Exploits may:

- Manipulate flags
- Abuse signed vs unsigned checks
- Bypass logic using CF vs OF confusion

Flags lie only if you misread them.

4.9 Summary: EFLAGS

The **EFLAGS** register is 32 bits wide and holds many independent flags.

These flags **report the results of operations** — for example, whether the last calculation produced zero, a negative value, or caused a carry/overflow. They also **control execution**, influencing things like conditional jumps or interrupts.

In short: **registers do the work**, and **flags decide what happens next**.

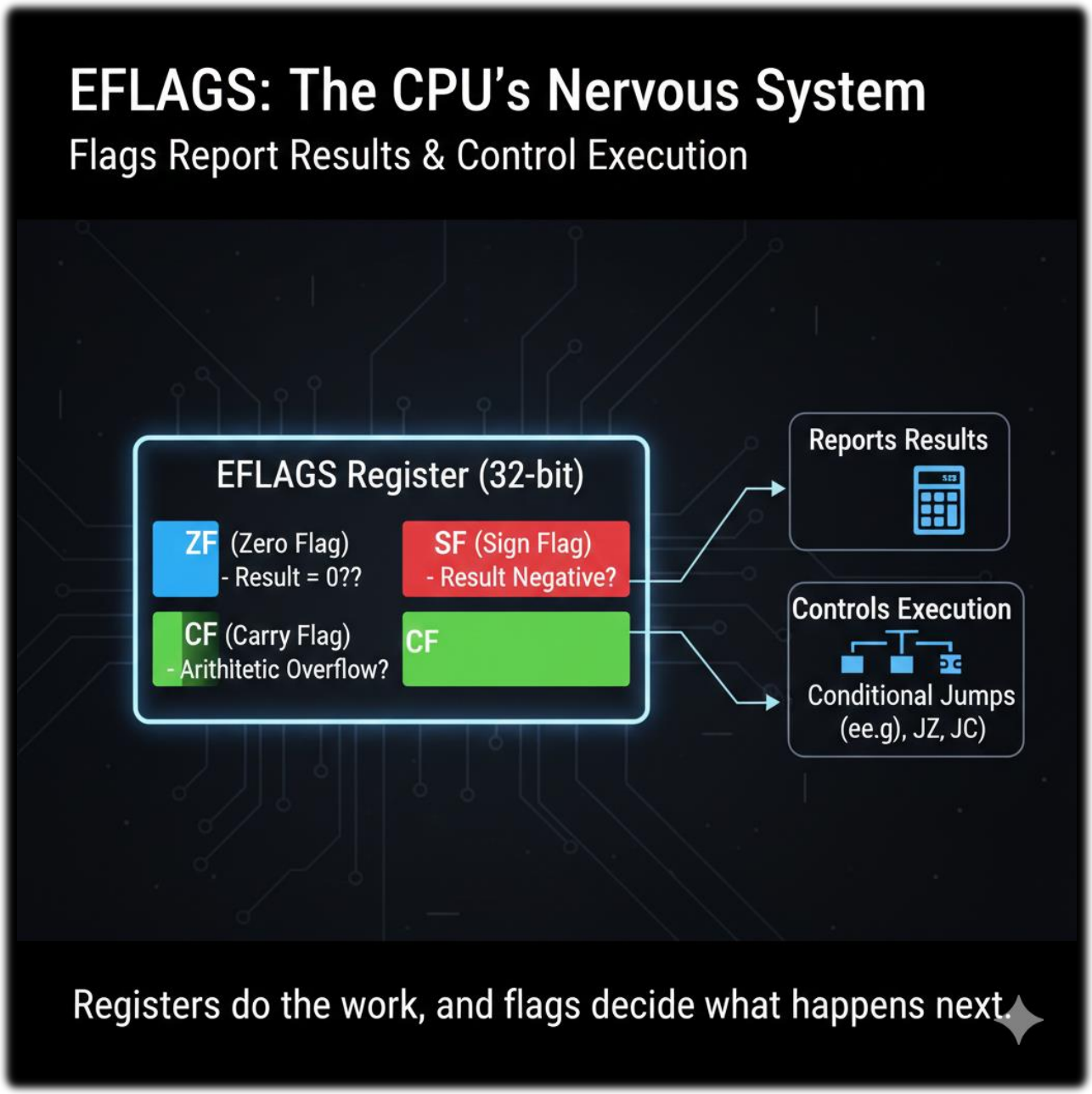
So, when it says "reports results," it means EFLAGS tells the CPU what happened during the last instruction. For example:

- Zero Flag (ZF) → Was the result zero?
- Sign Flag (SF) → Was the result negative?
- Carry Flag (CF) → Did an arithmetic operation overflow?

These flags are what let instructions like JZ (jump if zero) or JC (jump if carry) know whether to take a branch.

4.10 Lock-In Questions

1. Why are CF and OF different?
2. Why does CMP not store a result?
3. Why is DF dangerous if left set?
4. How do debuggers use TF?



MMX, XMM, AND FPU REGISTERS (x86 SIMD & FLOATING POINT)

5.1 First: Three Different Worlds (Do Not Mix Them)

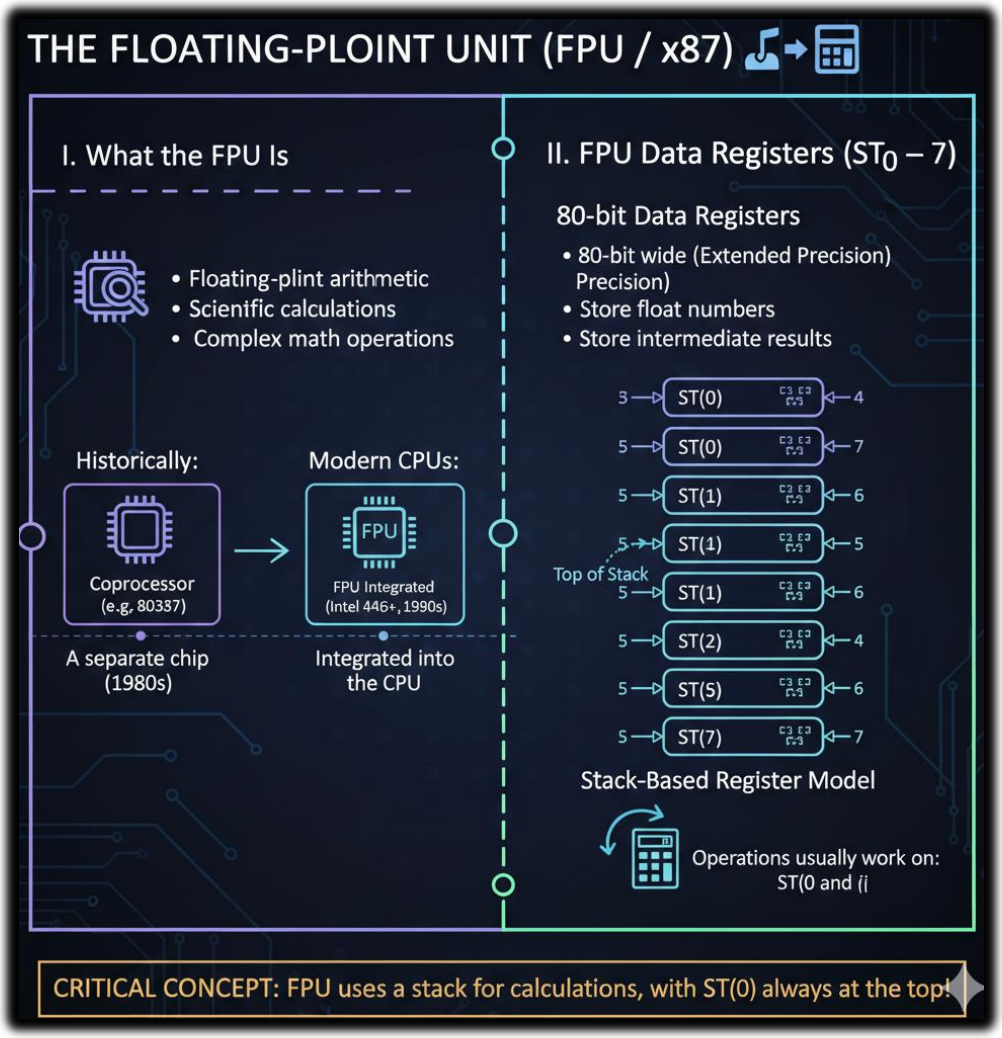
Before details, lock this in:

There are **three related but distinct systems**:

1. **FPU (x87 Floating Point Unit)**
2. **MMX (Multimedia Extensions)**
3. **XMM (SSE / Streaming SIMD Extensions)**

They exist for different eras and purposes.

If you mix them mentally, debugging becomes hell.



5.2 The Floating-Point Unit (FPU / x87)

I. What the FPU Is

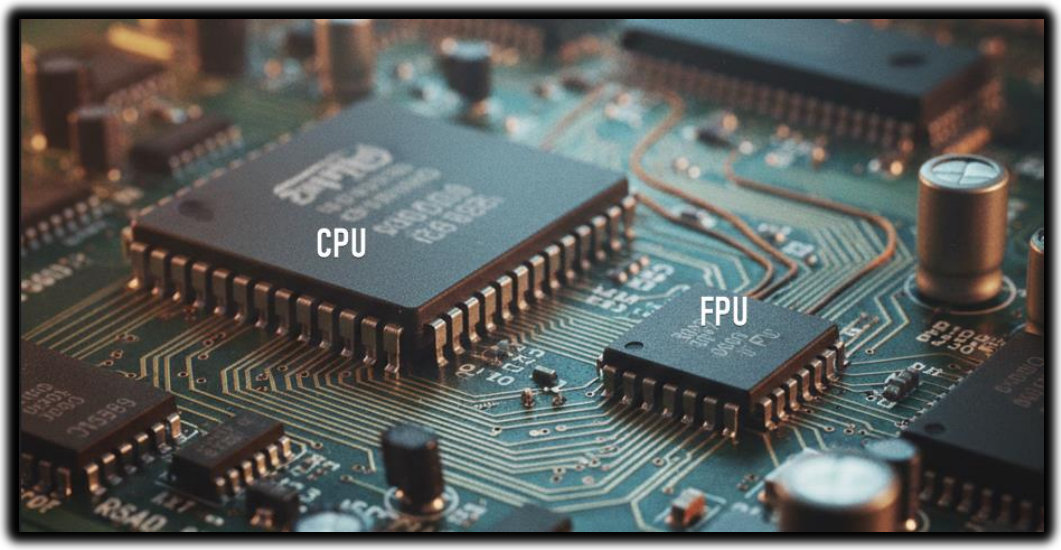
The **FPU** is the part of the CPU responsible for:

- Floating-point arithmetic
- Scientific calculations
- Complex math operations

Originally:

- It was a **separate chip** (coprocessor)
- Starting with **Intel 486**, it was integrated into the CPU

The 1980s, Intel 8086/8088 CPUs often used a separate Floating-Point Unit (FPU) chip, like the 8087. It was a real beast, handling all the floating-point calculations, freeing up the main CPU.



II. FPU Data Registers (ST0 – ST7)

The FPU has:

- **8 floating-point data registers**
- Named **ST(0) to ST(7)**
- Each register is **80 bits wide**

The FPU registers are where the CPU keeps floating-point numbers and intermediate results during calculations.

They're special because they support **extended precision**, giving more accuracy than standard 32-bit or 64-bit floats.

One important idea is that the FPU works like a **stack**.

The top of the stack is always **ST(0)**, and most operations involve **ST(0)** along with another register somewhere deeper in the stack, like **ST(i)**.

This stack-based model is different from the general-purpose registers you've seen before, but once you get used to it, it makes managing floating-point calculations very efficient.

REGISTER	PRECISION	ROLE DESCRIPTION
`ST0` TOP	80-bit	The current working register. All math happens here.
`ST1`	80-bit	The second value used in math (like the 'Y' in X + Y).
`ST2`	80-bit	Extra space for long calculations.
`ST3`	80-bit	Temporary storage for math bits.
`ST4`	80-bit	Temporary storage for math bits.
`ST5`	80-bit	Temporary storage for math bits.
`ST6`	80-bit	Temporary storage for math bits.
`ST7`	80-bit	The bottom of the stack.

5.3 FPU Control & Status Registers (Often Ignored, Very Important)

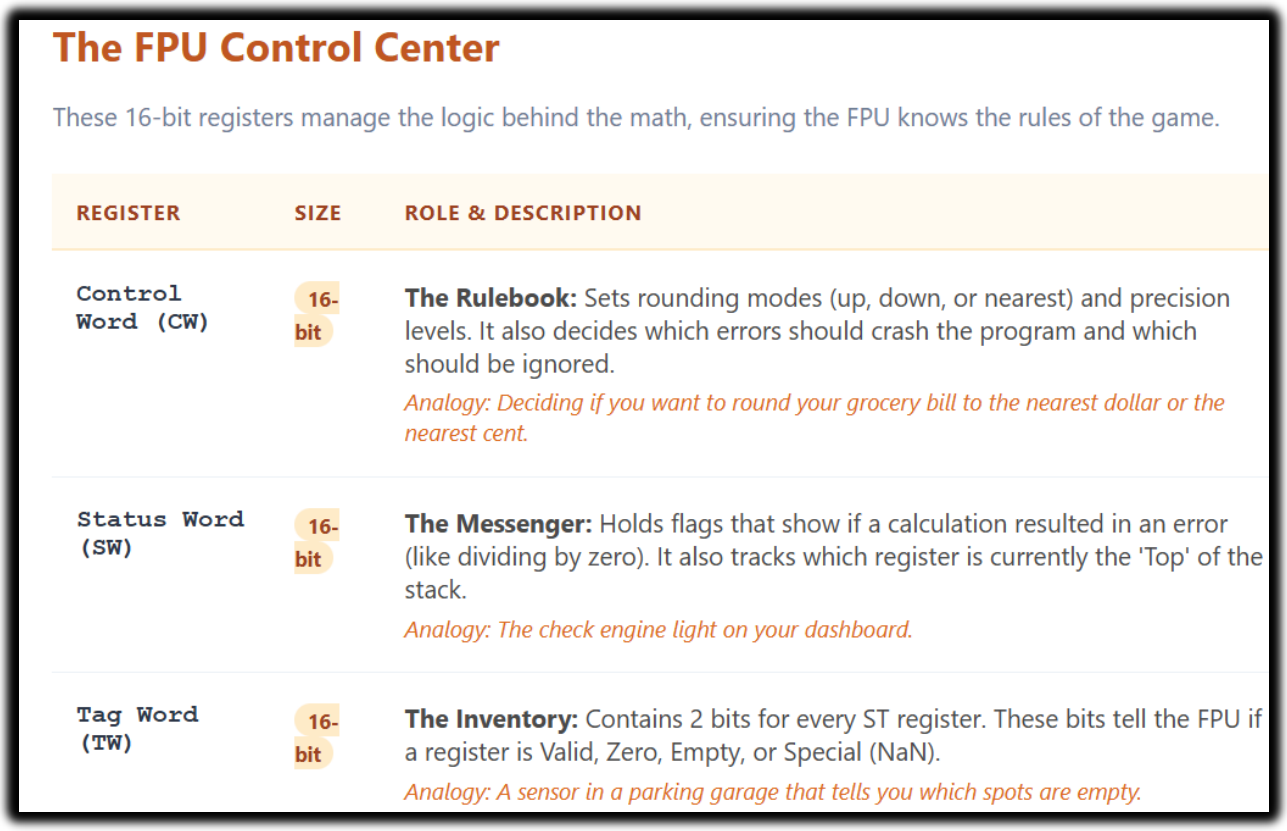
The FPU doesn't just have data registers; it also has several **control and status registers** that help manage how calculations are done and track the state of the FPU.

The **FPU Control Register** is 16 bits and lets the CPU control things like the precision of calculations, the rounding mode, and which exceptions are masked.

The **FPU Status Register**, also 16 bits, reports what's happening inside the FPU: it shows any exceptions, condition codes, and the current state of the stack.

Then there's the **FPU Tag Register**, which keeps track of which ST registers are empty and which contain valid data.

These control and state registers are especially important during **context switching**. When the operating system switches between programs, it must save and restore these registers to make sure each program continues calculations correctly and consistently.



The FPU Control Center

These 16-bit registers manage the logic behind the math, ensuring the FPU knows the rules of the game.

REGISTER	SIZE	ROLE & DESCRIPTION
Control Word (CW)	16-bit	**The Rulebook:** Sets rounding modes (up, down, or nearest) and precision levels. It also decides which errors should crash the program and which should be ignored. *Analogy: Deciding if you want to round your grocery bill to the nearest dollar or the nearest cent.*
Status Word (SW)	16-bit	**The Messenger:** Holds flags that show if a calculation resulted in an error (like dividing by zero). It also tracks which register is currently the 'Top' of the stack. *Analogy: The check engine light on your dashboard.*
Tag Word (TW)	16-bit	**The Inventory:** Contains 2 bits for every ST register. These bits tell the FPU if a register is Valid, Zero, Empty, or Special (NaN). *Analogy: A sensor in a parking garage that tells you which spots are empty.*

5.4 Opcode Register (FPU Opcode Register) ⚠

This is where the old notes get shaky — let's fix it.

What the Opcode Register Is

- **16-bit register**
- Stores the opcode of the **last executed FPU instruction**
- **Read-only**
- Updated automatically by the processor

It is **part of the FPU state**, not general CPU execution.

It exists mainly for:

- Debugging
- Exception handling
- Error reporting

📌 It does **not** control execution
📌 It does **not** replace EIP
📌 It belongs to the **FPU subsystem**

5.5 FPU Pointer Registers (48-bit) 📍

The FPU also has **pointer registers**, and understanding them is really important, even though they're often poorly explained.

In protected mode, the FPU keeps track of two things: the **instruction pointer** and the **data pointer**. Each of these pointers is 48 bits wide, made up of a 16-bit **segment selector** and a 32-bit **offset**.

These pointers tell the FPU exactly **which instruction caused an exception** and **which memory location was being used**.

This information is crucial for the operating system, allowing it to recover from FPU faults and help with debugging floating-point errors.

When we say "protected mode," we mean the CPU is using **segmentation and memory protection** — this is not real mode and not 64-bit long mode.

FPU TRACKING REGISTERS

Register	Standard Size	What it remembers
`FIP` (Instruction Pointer)	48-bit	**The Address of the Command:** This stores the exact location in memory of the last math instruction that was executed. > Analogy: The page number in a recipe book where the last instruction was found.
`FDP` (Data Pointer)	48-bit	**The Address of the Data:** This stores the location in memory of the number (the operand) used by that instruction. > Analogy: The address of the grocery store where you bought the ingredients.
`FOP` (Last Opcode)	11-bit	**The Action:** This stores a small code representing the actual math operation (like Add, Subtract, or Square Root). > Analogy: Remembering that the instruction was 'Bake' and not 'Boil'.

5.6 MMX Registers (Legacy SIMD) 📼

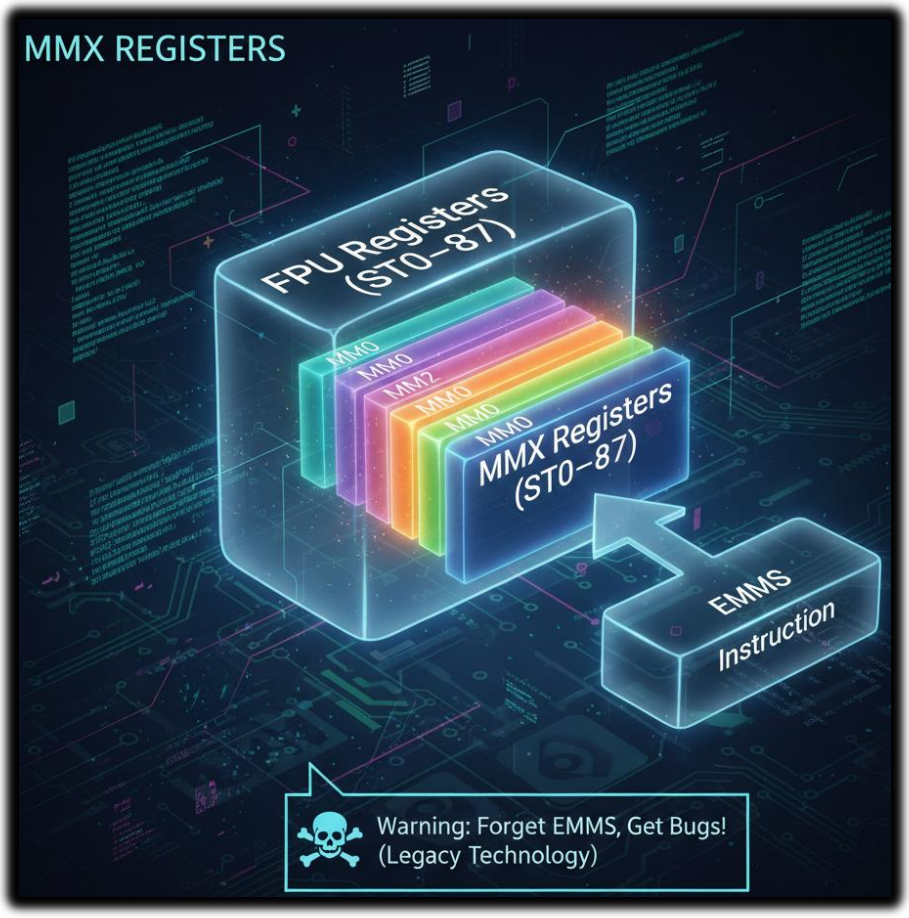
MMX was created to make multimedia, image processing, and audio/video tasks faster. It gives you **8 registers**, each **64 bits wide**, named **MM0–MM7**.

Here's the important part: **MMX registers aren't separate from the FPU**. They are actually **aliases of the FPU registers**. That means:

- MM0 uses the same physical storage as ST(0)
- MM1 uses ST(1)
- …and so on.

Because of this overlap, you have to **clean up the FPU state after using MMX**. The instruction for this is **EMMS**.

If you forget, floating-point calculations can break, sometimes causing subtle and confusing bugs. This overlap and cleanup requirement is why MMX is considered **legacy technology** today.



5.7 XMM Registers (SSE / Modern SIMD) ✨

XMM registers were created to fix the limitations of MMX. Unlike MMX, they are **completely separate from the FPU** and don't share storage with it.

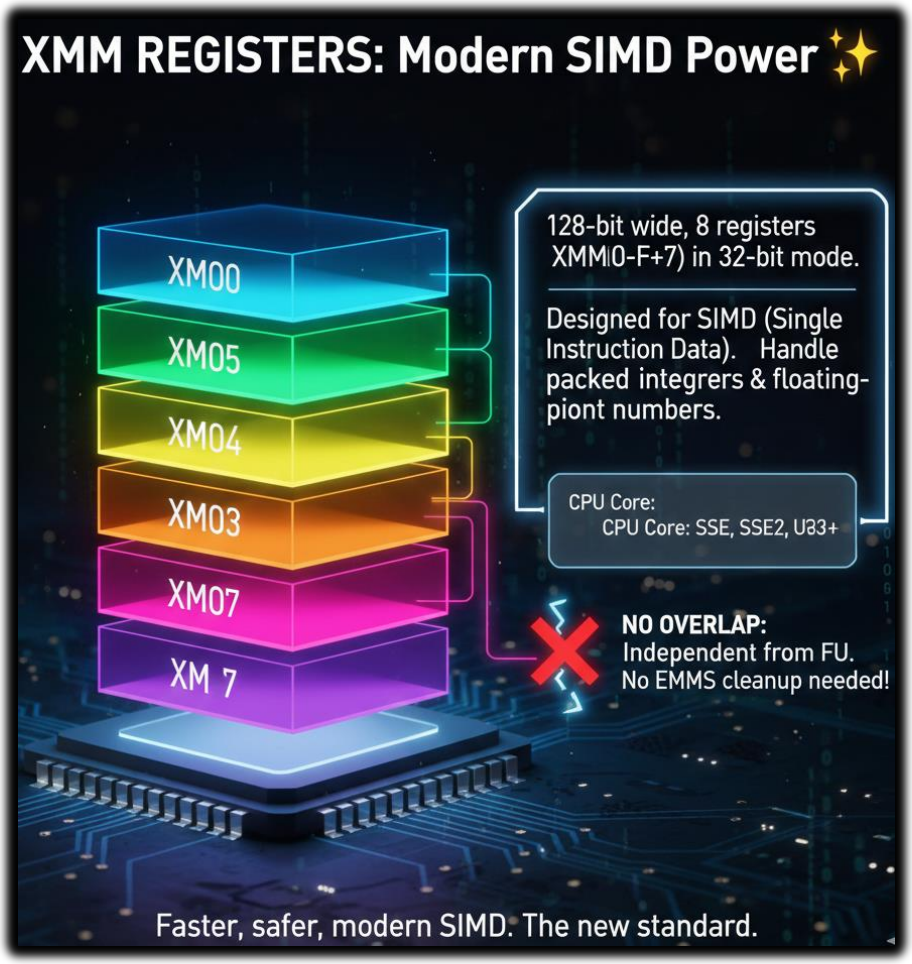
They were designed for **SIMD (Single Instruction, Multiple Data)** operations from the start.

In 32-bit mode, there are **8 XMM registers**, named **XMM0–XMM7**, and each one is **128 bits wide**.

These registers are used by SSE, SSE2, SSE3, and later SIMD extensions.

They can handle integers, floating-point numbers, and packed data, making them the **modern standard for SIMD calculations**.

Because they are independent from the FPU, there's no need to clean up the state like with MMX, which makes them safer and easier to use in modern code.



5.8 Why XMM Replaced MMX

I. MMX Problems

- **Aliases FPU registers** – MMX registers aren't separate from the FPU. MM0 uses ST(0), MM1 uses ST(1), and so on. This means using MMX can interfere with any floating-point calculations already in progress.
- **Breaks floating-point code** – Because MMX shares storage with the FPU, failing to clean up properly can cause strange bugs in floating-point operations. Programs that rely on FPU math can suddenly give wrong results.
- **Requires EMMS cleanup** – After using MMX instructions, you must run the EMMS instruction to reset the FPU state. Forgetting this step can corrupt calculations and cause subtle, hard-to-find errors.
- **Hard to context-switch safely** – Saving and restoring MMX/FPU state during task switches is tricky because the registers are shared. The OS has to be very careful, or the next program could inherit a broken FPU state.

II. XMM Advantages

- **Separate register file** – XMM registers are independent of the FPU, so using them doesn't interfere with floating-point math.
- **Cleaner design** – There's no need for a cleanup instruction like EMMS. This reduces the chance of bugs and makes code more predictable.
- **Better OS support** – Because XMM registers are distinct, context switches are simpler and safer. The OS can save and restore them without worrying about corrupting the FPU.
- **Scales to AVX/YMM/ZMM later** – XMM was designed with future SIMD extensions in mind. It naturally extends to 256-bit YMM registers (AVX) and 512-bit ZMM registers (AVX-512).

5.9 Control Registers (CR0–CR3) – Fixing Old Mistakes

The old notes say: **"control registers are 16-bit"** ❌ Incorrect.

Correct Facts: Control registers are special **32-bit registers** (in 32-bit mode) used by the CPU to manage how the system runs. They are named **CR0, CR2, CR3, CR4**, and later **CR8** in 64-bit mode.

These registers control key CPU features, including:

- **Protected mode** – enabling features like memory protection and privilege levels
- **Paging** – mapping virtual memory to physical memory
- **Write protection** – preventing certain memory writes
- **Virtual memory management** – helping the OS manage memory efficiently

Access to control registers is restricted:

- **User mode** programs can only read them
- **Supervisor/kernel mode** can read and write them

One important control register is **CR3**, which holds the **page directory base**. This makes it central to the system's memory management, telling the CPU where to find the page tables for virtual memory.

It's worth noting that **control registers are not part of the FPU**. They serve a completely different purpose, managing the CPU and the system rather than floating-point calculations.

5.10 Putting Everything in the Right Buckets 🧠

FPU Subsystem

- ST0–ST7 (80-bit)
- Control register
- Status register
- Tag register
- Opcode register
- Instruction pointer (48-bit)
- Data pointer (48-bit)

MMX Subsystem

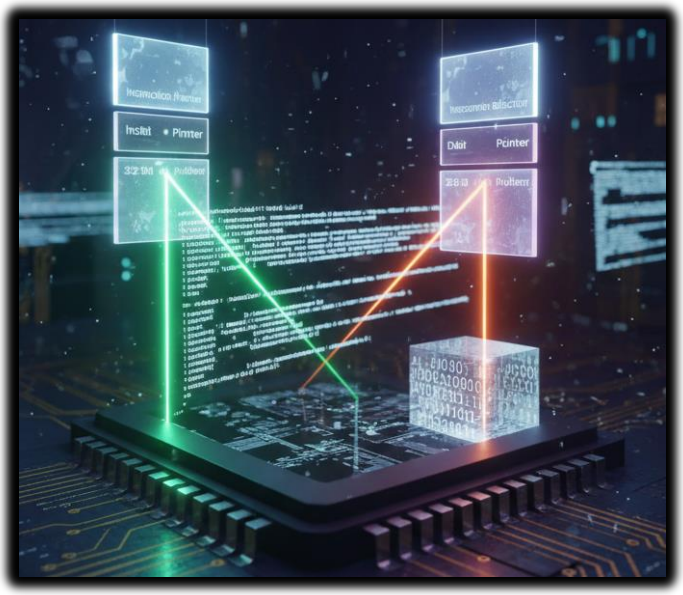
- MM0–MM7 (aliases of FPU)

SIMD Subsystem (SSE)

- XMM0–XMM7 (128-bit, separate)

System Control

- CR0–CR4 (privileged)



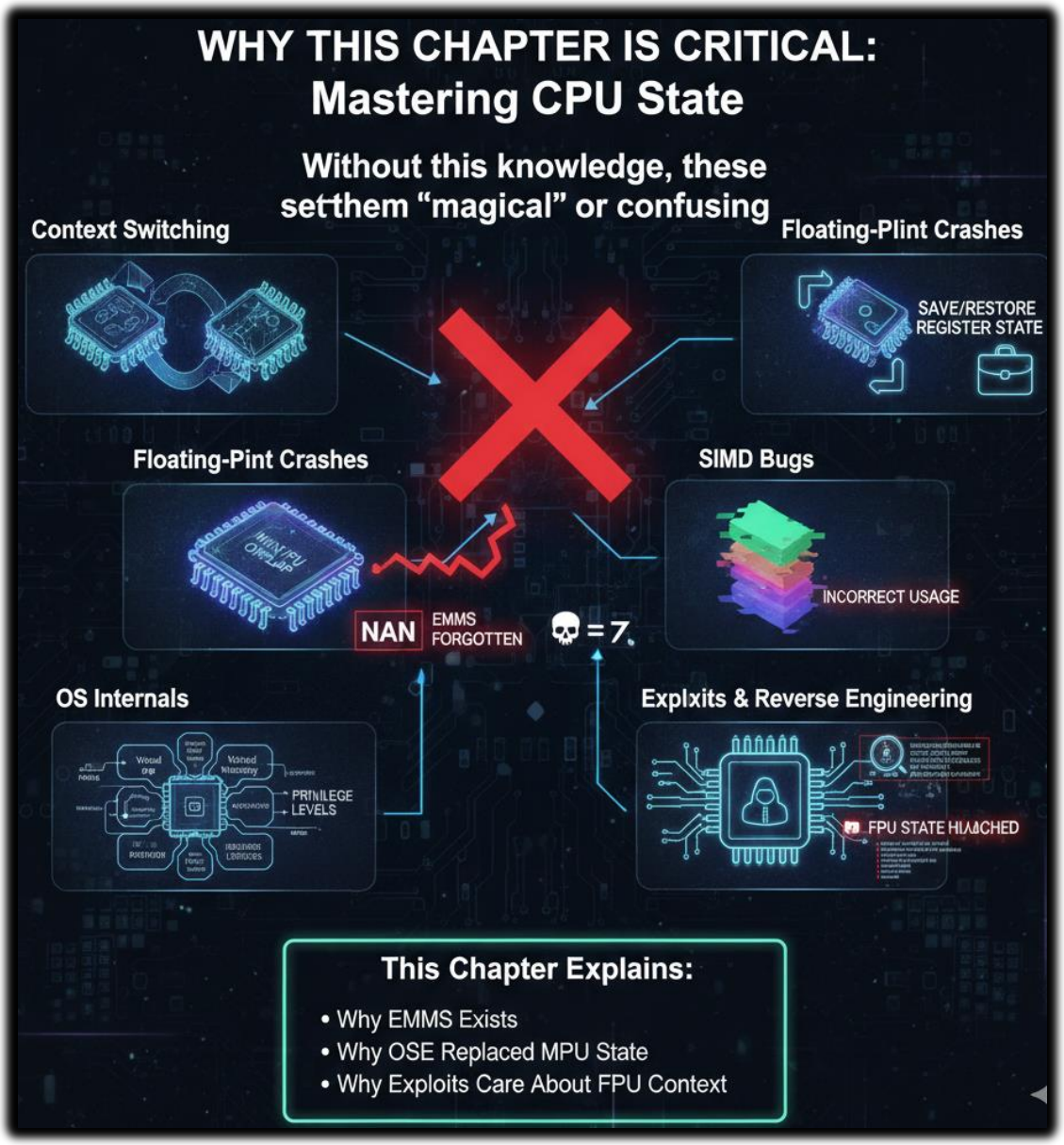
5.11 Why This Chapter Is Critical

If you don't understand this material, several things will seem confusing or "magical":

- **Context switching makes no sense** – without knowing how registers and FPU/MMX/XMM state are saved/restored, task switching seems mysterious.
- **Floating-point crashes look random** – forgetting EMMS or mixing MMX/FPU registers can cause subtle math errors that appear out of nowhere.
- **SIMD bugs feel supernatural** – using XMM/YMM/ZMM incorrectly can produce strange results in multimedia or scientific code.
- **OS internals feel impossible** – memory management, privilege levels, and exception handling are all intertwined with control and pointer registers.
- **Exploits and reverse engineering get confusing** – FPU and SIMD state can be targeted by malware or influence how code behaves.
- **High-performance programming becomes risky** – if you misuse SIMD instructions without understanding the underlying registers, calculations may silently break.
- **Debugging and optimization are harder** – knowing how registers interact helps you trace bugs and write faster code safely.
- **Historical context matters** – understanding MMX vs. XMM vs. AVX shows why older code behaves differently and why modern compilers avoid legacy instructions.

This chapter explains:

- **Why EMMS exists** – to reset the FPU after MMX usage
- **Why SSE replaced MMX** – cleaner, independent registers
- **Why OSes save FPU state** – to avoid crashes during context switches
- **Why exploits care about FPU context** – registers can leak info or be misused by attackers



5.12 Lock-In Questions

1. Why do MMX registers break floating-point code?
2. Why are FPU registers 80-bit instead of 64-bit?
3. Why are FPU pointers 48-bit in protected mode?
4. Why did SSE need separate registers?
5. What does EMMS do and why is it necessary after using MMX?
6. How does the FPU stack model work?

7. Why is ST(0) considered the top of the FPU stack?
8. How do FPU control and status registers affect calculations?
9. What does the FPU tag register track and why is it important?
10. How do context switches affect FPU and SIMD registers?
11. Why can't you directly write to EIP?
12. How do CALL and RET instructions manage the stack and EIP?
13. What happens if ESP or EBP is modified incorrectly?
14. How do interrupts (INT) change the flow of execution and EIP?
15. What is the difference between MMX and XMM in terms of register aliasing?
16. How does AVX extend XMM registers to YMM and ZMM?
17. Why must operating systems save and restore FPU/XMM state during a task switch?
18. Why is CR3 central to memory management in protected mode?
19. What kinds of operations are handled faster with SIMD registers than with the FPU?
20. How does the CPU know which ST registers are empty or contain valid data?
21. Why are control registers read-only in user mode but read/write in kernel mode?
22. How do segmentation and protected mode influence FPU pointer size?
23. Why is understanding register slicing (AL, AH, AX, EAX) still relevant today?
24. What could go wrong if MMX instructions are used without EMMS cleanup?
25. Why do modern compilers avoid MMX entirely?
26. How do FPU exceptions relate to instruction and data pointers?
27. Why is it critical to know which register holds the result of a function call?
28. How does REP MOVSB use ECX as a counter?
29. How do ESI and EDI registers simplify string/memory copying operations?
30. Why are ESP and EBP considered "double identity" registers?