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.

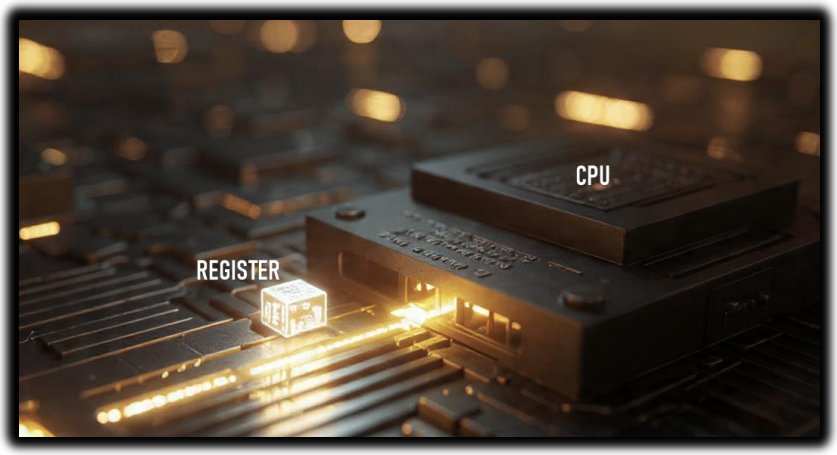
Once in a register, the CPU can add numbers, compare values, or move data quickly.

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:

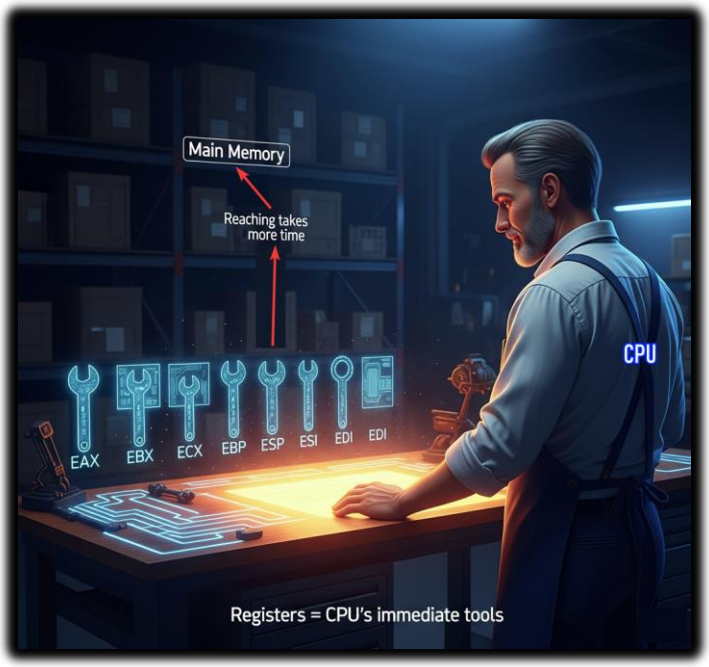
1. **AX**: Used for addressing data, pointing to specific locations.
2. **BX**: Similar to AX, used for addressing data, but often used for larger values (like 16-bit numbers).
3. **CX**: Often used for addressing data in addition to what's already stored in AX and BX.
4. **DX**: A bit different from 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.

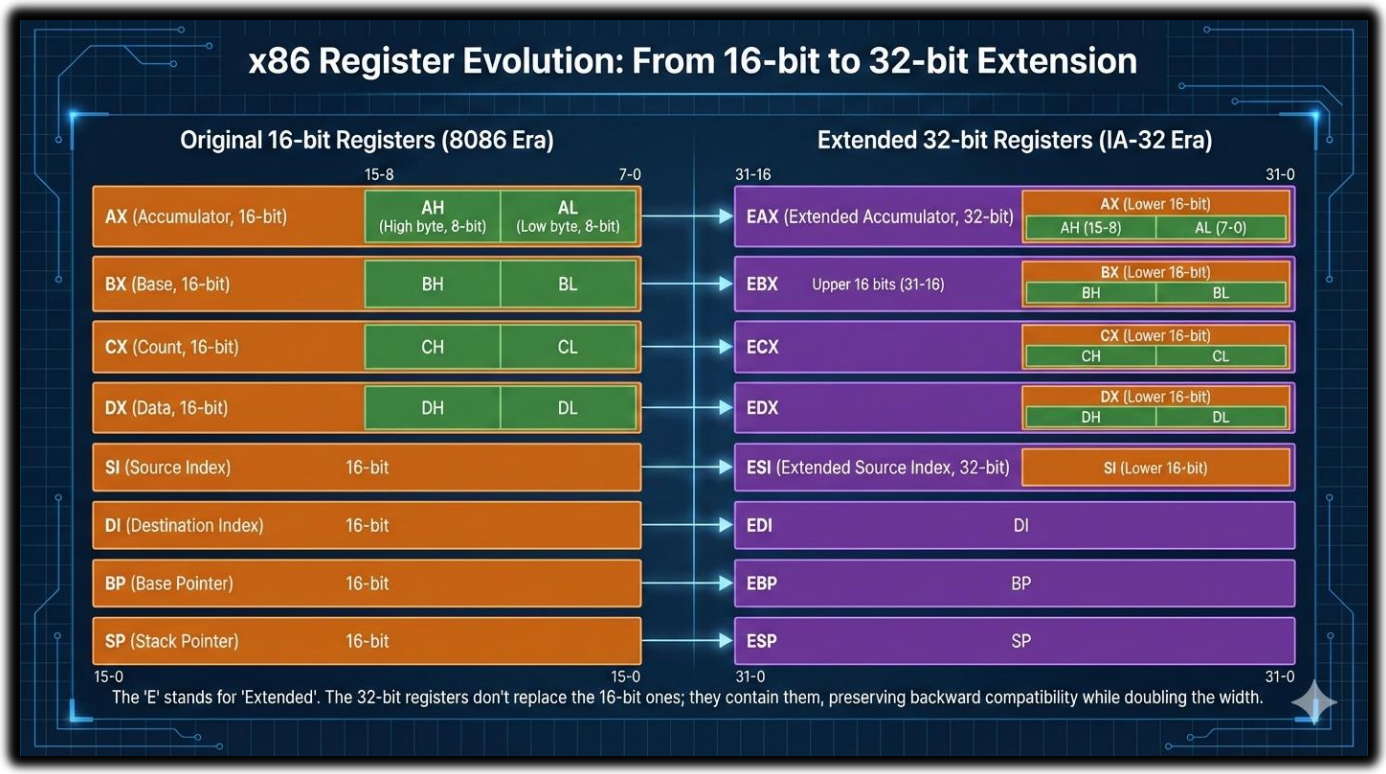
Think of them like a set of versatile hammers, each one capable of performing different functions. For example:

- **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.

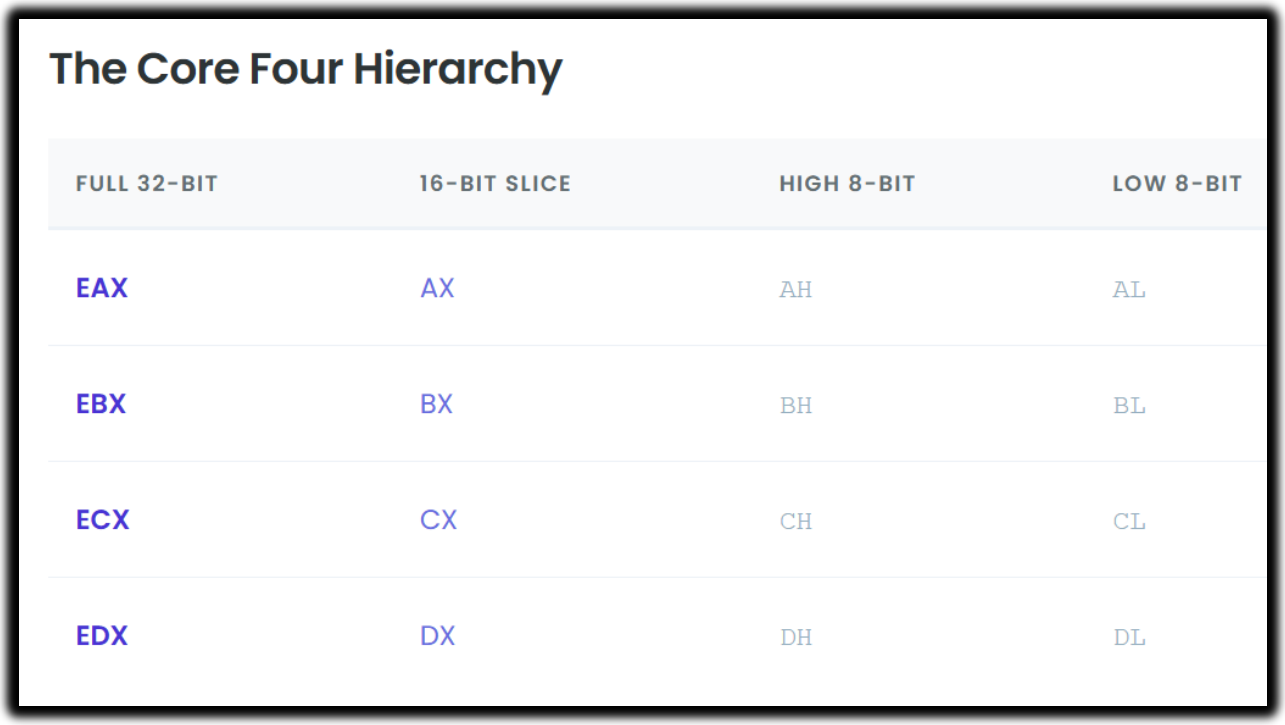
There's even a special LOOP instruction that automatically decreases ECX by one each time it runs.

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.

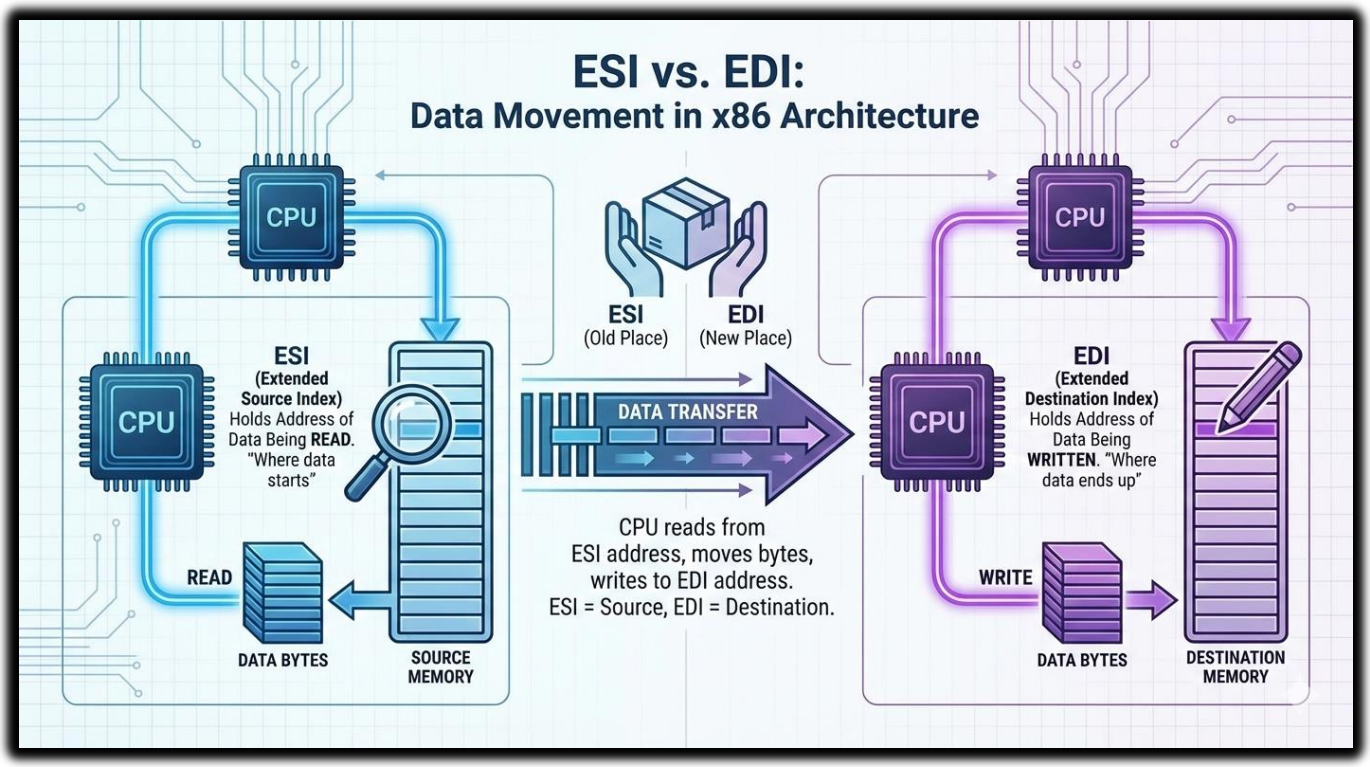
It points to where the data starts, like the source of a copy.

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.

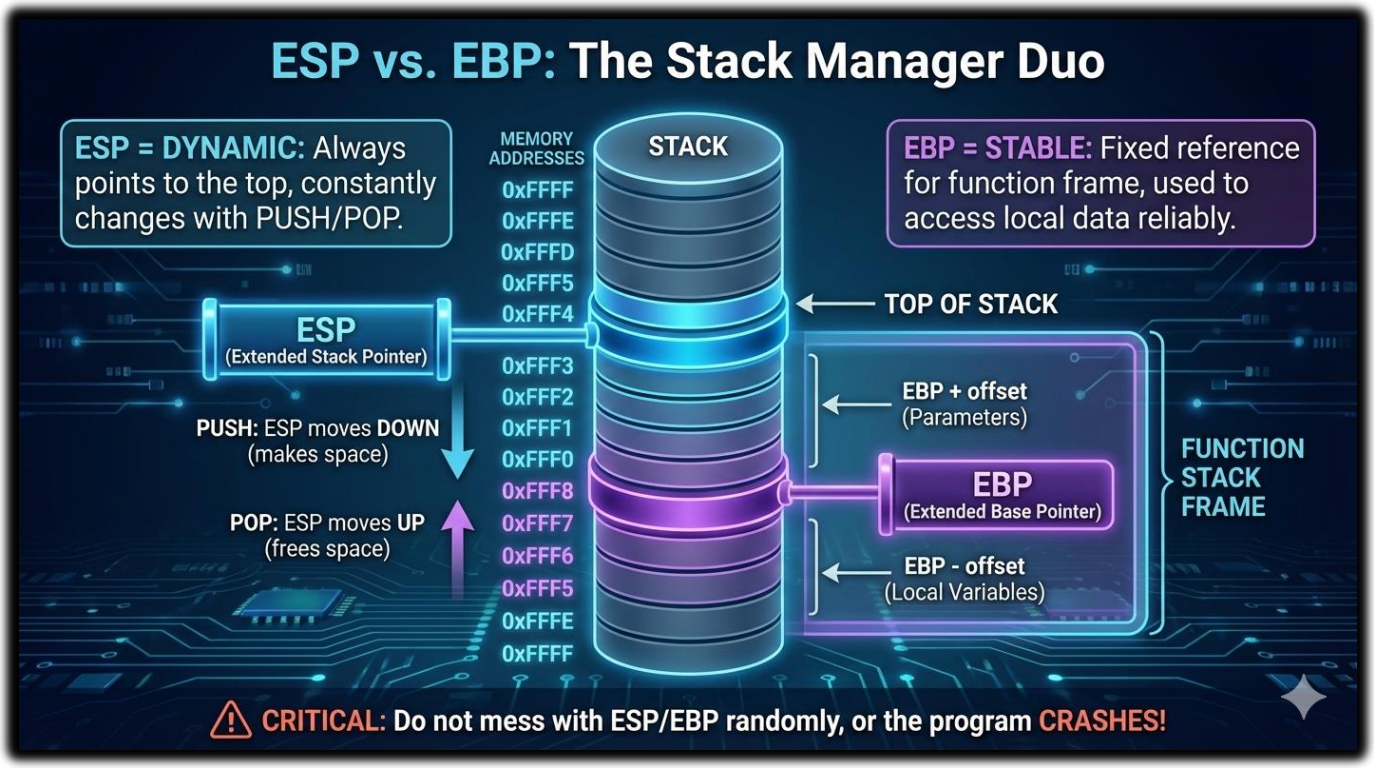
II. EBP – Extended Base Pointer

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.