

## ⌚ x86 PROCESSOR BASICS (HOW THE CPU ACTUALLY RUNS THE SHOW)

Imagine the CPU as the **brain** of your computer.

But not a chill brain — a **cracked-out microsecond freak** that runs everything on caffeine and electricity. Here's how it works:

### 🏛️ The CPU – Central Processing Unit

This is where all the **thinking, math, and decision-making** happens.

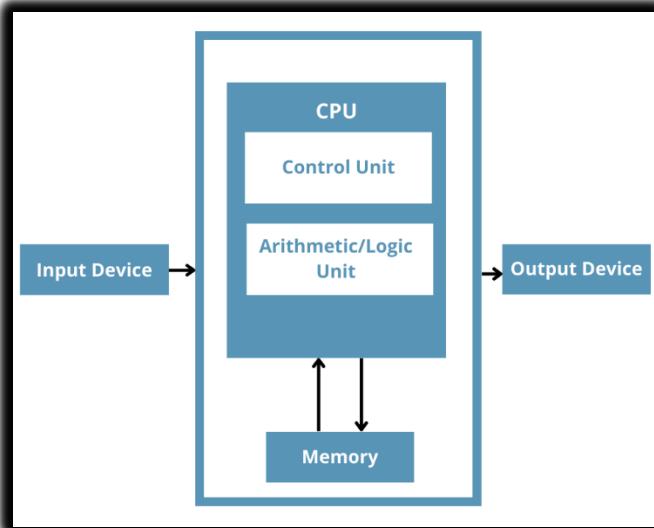
It has:

- **Registers** – tiny super-fast storage slots (think 32-bit pockets for numbers)
- **Clock** – keeps time like a heartbeat so stuff happens in sync
- **Control Unit (CU)** – the **boss** that decides what happens next
- **ALU (Arithmetic Logic Unit)** – the **muscle** that does all the math and logic ops (ADD, SUB, AND, OR, NOT, etc.)

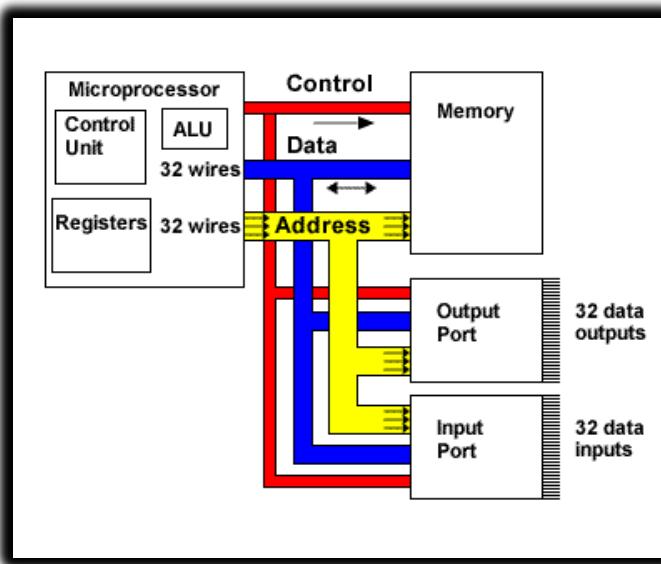


## 💡 How the CPU Connects to the World

The CPU talks to the rest of the PC through **pins** on its socket. These pins connect it to **buses** — long electric highways carrying signals.



## 💡 The 3 main buses:



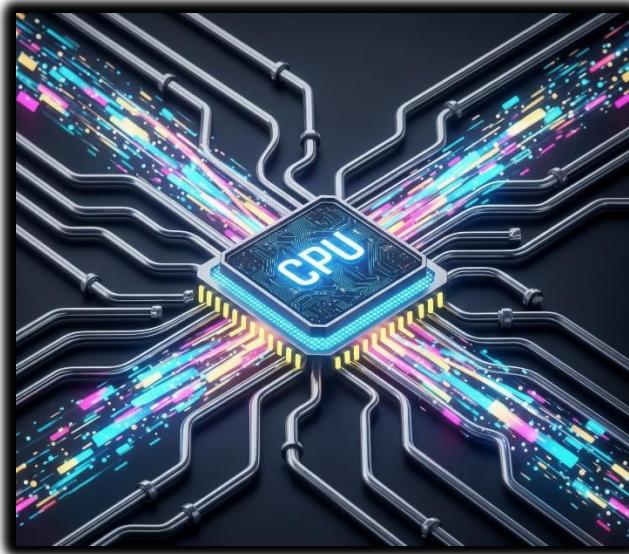
## ■ Data Bus

Moves the *actual data* and *instructions* between the CPU, memory and I/O devices.

The data bus is bidirectional, meaning information can flow in both directions.

The "*width*" of the data bus (how many parallel wires it has) determines how much data can be transferred at once.

A *64-bit data bus* can move 64 bits of data simultaneously.



**Analogy:** The data bus is like a fleet of delivery trucks that transport goods (data) and mail (instructions) between the city hall (CPU), the library (memory), and various businesses (I/O devices). These trucks can deliver or pick up cargo.

## ■ Address Bus

Says *where* in memory we're looking.

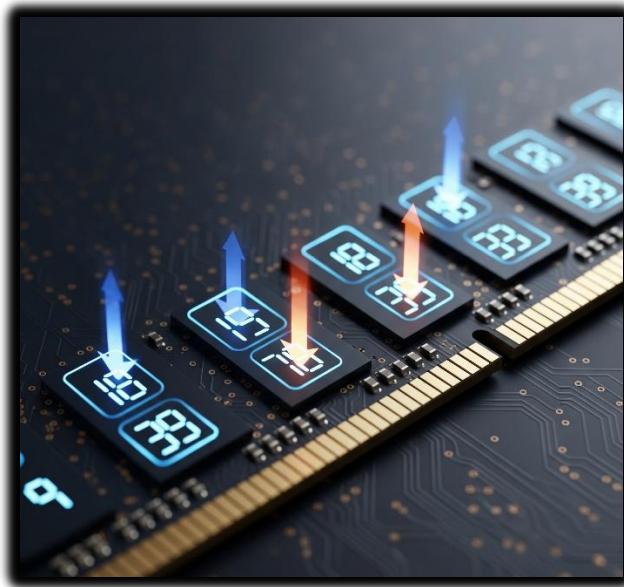
The address bus is *unidirectional*, meaning information flows only from the CPU to other components.

It carries the **memory addresses or I/O port addresses** where data is to be read from or written to.

When the CPU wants to access a specific piece of data or instruction, it places its memory address on the address bus, telling the memory unit *exactly where to find* or store that information.

The *width of the address bus* determines the maximum amount of memory the CPU can access.

A *32-bit address* bus can address  $2^{32}$  unique memory locations (4 Gigabytes).



Imagine your *computer's RAM as a massive library*, and each book in that library has a unique shelf and position. When the CPU wants to read a specific piece of information (a "book"), it doesn't just shout out the book's title.

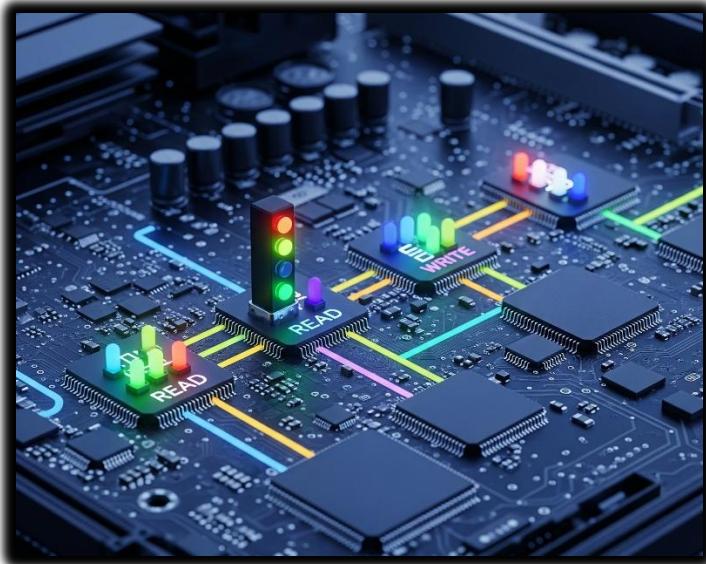
Instead, it sends out the exact "*shelf number*" and "*position*" through the address bus. This "shelf number and position" is what we call a **memory address**.

This *one-way communication* ensures that the CPU can accurately request data from, or send data to, a specific spot in memory.

## ■ Control Bus

Uses binary signals (on/off) to tell devices **when** to send or receive. It synchronizes the actions and manages the flow of information among all devices attached to the system bus.

**Think:** "Hey RAM — CPU wants to read now!"



It carries control signals that dictate operations like "memory read," "memory write," "I/O read," "I/O write," "interrupt request," and "bus grant."

These signals ensure that devices don't try to use the buses simultaneously or perform conflicting operations.

The control bus is like the city's traffic light system.

## Other Buses

### ● I/O Bus

Handles data moving between CPU and input/output devices (keyboard, mouse, etc.)

Also called the Peripheral bus, considered part of the system bus, but, yeah, it's a bit different coz its *dedicated* to transferring data between the CPU and the system I/O devices.

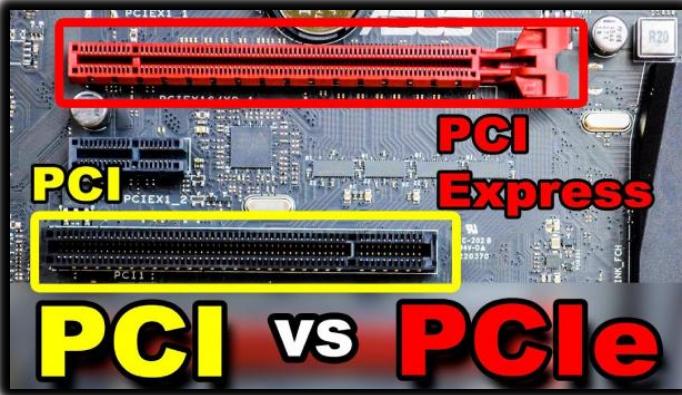
Modern systems often use high-speed serial buses like *PCI Express (PCIe)* for this purpose.



This bus is all about getting data to and from your **input/output devices**. Imagine:

- **Keyboard Input:** When you type "hello," that information needs to travel from your keyboard into the computer. The I/O bus is the route that data takes, like supplies being delivered to a restaurant. 
- **Printer Output:** When you hit "print," the document data needs to go from your computer out to the printer. The I/O bus handles this, much like official documents being sent out to residents. 

The I/O bus is considered part of the overall system bus, but it's "a bit different" because it's dedicated to these specific external communications. Modern systems use advanced, high-speed **I/O buses like PCI Express (PCIe)**.



## 🧠 Memory – Where Programs & Data Live

All your running programs and variables are stored in **RAM**. But here's the kicker: The CPU can't run them straight from **RAM** coz it is a temporary storage locker for the CPU.



It always does this:

1. Grabs the instruction from memory.
2. Brings it into the CPU which has small temporary storage locations, registers, the ones we covered in the previous chapter.
3. Executes it.
4. Maybe sends a result back to memory.

So, your code doesn't *run in RAM*, it runs **inside the CPU** — one piece at a time, or in chunks.

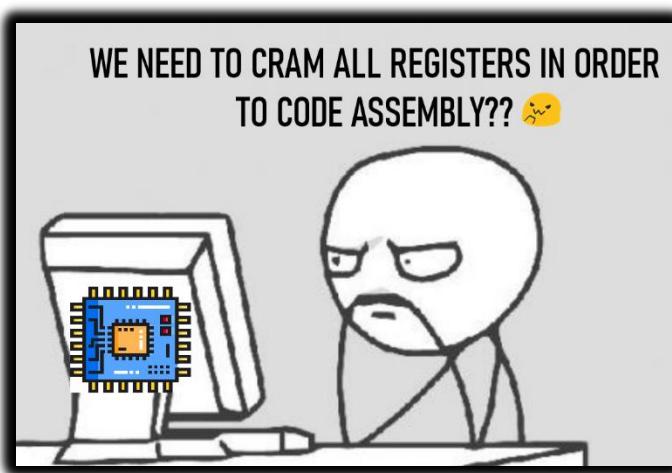
*We're going to see more about this Fetch, Decode, Execute cycle ahead.*

## 📦 Buses Summary (Quick Table):

Bus Type	What it Moves	Between	🔗
Data Bus	Actual values, instructions	CPU ⇌ Memory	
Address Bus	Memory addresses	CPU → Memory (to say where to go)	
Control Bus	Control signals (like READ/WRITE)	CPU → All hardware	
I/O Bus	Device-level data	CPU ⇌ Keyboard, Mouse, etc.	

## TLDR – Reverse Engineering Focus:

- Know the **ALU** is where bitwise ops live (AND, OR, SHL, etc.)
- Know that **registers** are the CPU's playground — what you see in disasm (like eax, edx, rsi, etc.)
- Remember: instructions **run inside** the CPU, not memory. Memory just holds them until they're needed.
- Buses = wires that move the ops around. If you're watching malware move code into memory and jump to it — that's this system in action.



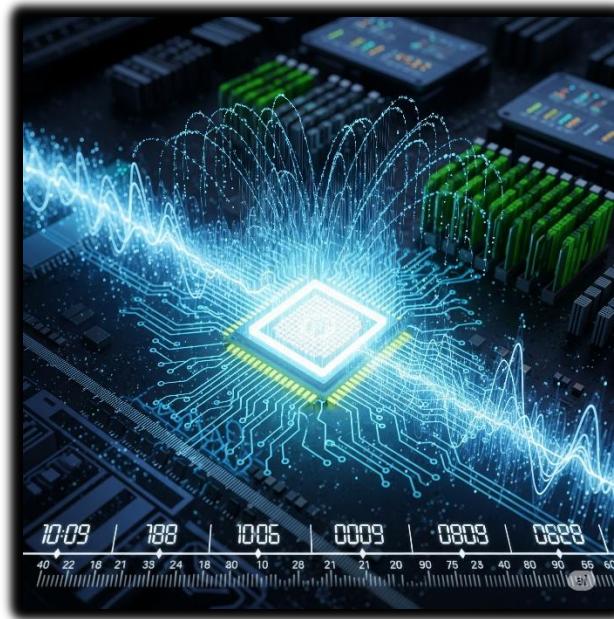
A *register's purpose* often becomes clear from the instructions around it. Is it being used as a counter in a loop? An argument for a function? The return value? The context will clue you in.

*Learn by Doing:* The more assembly code you read and write (even small snippets!), the more you'll see how registers are actually used in real programs. This hands-on experience beats rote memorization any day.

## ⌚ CLOCK & CLOCK CYCLE (X86 CPU TIMING EXPLAINED)

### ⌚ The Unseen Rhythm: What's the Clock?

The **CPU clock** is like the relentless, precisely timed heartbeat of your processor — ticking at a fixed speed (e.g., 1 GHz = 1 billion ticks per second).



It is an *internal electronic signal* that oscillates at an incredibly fixed and high frequency.

This isn't just a simple timer; it's the *master synchronizer* that orchestrates every single operation within the processor and its interactions with the rest of the computer system.

This clock ticks at a specific, fixed speed, often measured in Gigahertz (GHz). For example, a **3 GHz CPU** means the clock "ticks" 3 billion times every second. This incredible speed allows for billions of individual operations to occur in a mere blink of an eye.

The *clock keeps the CPU, RAM, and buses perfectly in sync*. It ensures data moves smoothly and at the right time — no timing chaos, no crashes. Without it, everything would fall apart.

## What's a Clock Cycle?

One clock cycle = **one complete tick** = the smallest unit of time the CPU understands.

*One clock cycle is equivalent to one complete tick. It represents the smallest indivisible unit of time the CPU understands and utilizes to perform any action. Nothing, absolutely nothing, can happen for a duration shorter than one clock cycle.*

The duration of a single clock cycle is simply the inverse of the clock speed.

For a CPU running at 1 GHz (1,000,000,000 cycles per second), one clock cycle lasts:

$$\frac{1 \text{ second}}{1,000,000,000 \text{ blinks}} = 0.000000001 \text{ seconds}$$

This is an incredibly tiny slice of time, emphasizing the sheer speed at which modern processors operate.

$$\frac{1 \text{ second}}{1,000,000,000 \text{ cycles}}$$

$$= 0.000000001 \text{ seconds}$$

(which is 1 nanosecond)

## ⌚ Clock Cycle in Action

- Every CPU instruction takes **at least 1 clock cycle** to run.
- Thanks to **pipelining**, modern CPUs can crunch simple operations super fast — even finishing one per cycle.
- But older CPUs? Different story.

On something like the **Intel 8088**, a single MUL instruction could eat up **tens or even hundreds** of cycles. 🎉

## 🧠 Meet the 8088 – The OG PC Chip

- Dropped in **1981**, the **Intel 8088** powered the first IBM PCs. That moment? Kicked off the whole *personal computer era*.
- It was a **cost-cut** version of the 8086 — same 16-bit CPU inside, but with an **8-bit external data bus** instead of 16.



**Why?** So, IBM could use cheaper 8-bit parts and simpler motherboard designs. 💰

**Downside?** To move 16-bit data, the 8088 had to do **two 8-bit transfers**. Slower memory and I/O — but it was worth it for the cost savings at the time.

## ❖ Segmented Memory (Remember this?)

- The 8088, like the 8086, used **segment:offset** addressing to get around the 64KB memory limit.
- It combined:
  - ✓ A **16-bit segment register** (points to a 64KB block)
  - ✓ A **16-bit offset**
- Together = a **20-bit address** → Boom, access to **1MB of RAM**.

( $2^{16}$  segment shifted left by 4 bits + offset = 20-bit address) – *we discussed this before.*

## ✓ Compatibility Bonus

- The 8088 ran the *same instructions* as the 8086 — full instruction set compatibility.
- So devs didn't have to rewrite anything. If it ran on 8086, it ran on 8088.

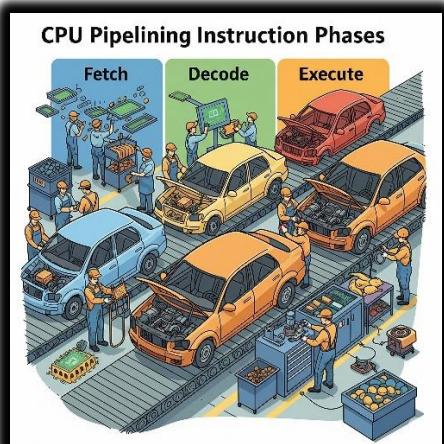
That made adoption easy and fast — crucial for software devs.

Modern x86 CPUs are incredibly sophisticated. They employ techniques like *pipelining* and *out-of-order execution*.

### Pipelining:

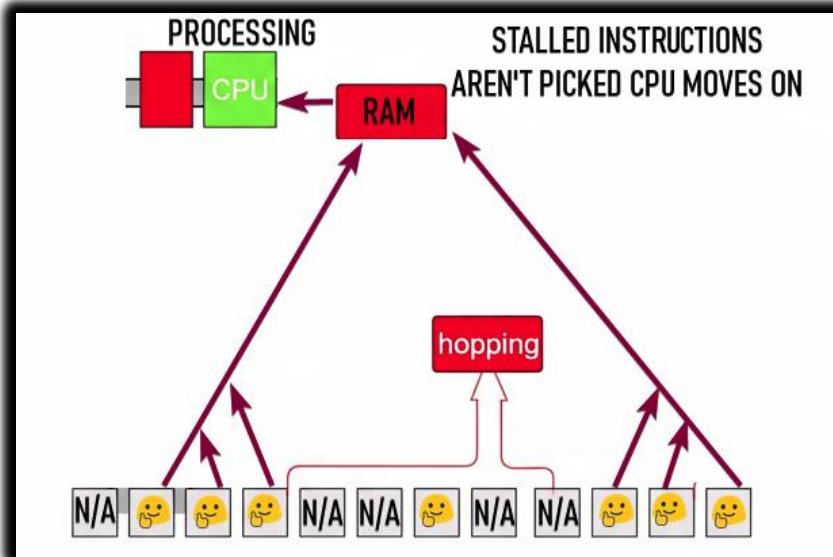
Imagine an assembly line. Instead of one worker building an entire car from start to finish, different workers perform different stages simultaneously on different cars.

In a CPU, this means that while one instruction is in its "**execute**" phase, another might be in "**decode**", and a third in "**fetch**."



## Out-of-Order Execution:

The CPU skips stalled instructions and runs independent ones first, then reorders the results. It's like working on what's ready instead of waiting — keeps the clock cycles busy.



If the CPU has to wait for slow memory? That gap = **wait states** (empty cycles where CPU chills while memory catches up).

**N/A** can represent wait states.

## ⌚ INSTRUCTION EXECUTION CYCLE

This is what the CPU **does for every instruction**, in order — over and over and over:

1. **Fetch**
  - Grab the instruction from memory (using the IP/EIP/RIP register).
  - It sends the memory address through the **address bus**, gets the data back via the **data bus**.
2. **Decode**
  - The CPU figures out:
    - What op it is (ADD, MOV, etc)
    - What operands are involved (registers, memory, etc)
3. **Execute**

- The instruction runs.

This is where the ALU steps in if there's math or logic.

#### 4. **Store**

- The result is written somewhere — back into a register or memory.

🧠 This 4-step cycle is repeated for *every single instruction* — from boot to shutdown.

---

#### 💡 **Real Talk:**

- If the clock is the **beat**, then the instruction cycle is the **dance** — the same steps every time, just with different moves.
- Your malware, game, compiler — all it's doing is **riding these cycles**.