PLC REPEATED NOTES FULL

📌 DEFINITION OF A PLC (Programmable Logic Controller)

A Programmable Logic Controller (PLC) is a solid-state, specialized industrial computer built for **harsh environments**, **storing** and **executing** control instructions to manage machines or processes in real-time with extreme reliability, essentially acting as the rugged "brain" of industrial automation engineered for 24/7 operation.

⚙️ TYPICAL INSTRUCTIONS STORED IN A PLC

When you write a PLC program, you're basically feeding it a bunch of instructions. Here are the classics:

🔄 Sequencing

Controlling the step-by-step progression of processes, much like following a TikTok recipe, is achieved through sequencing, often utilizing methods such as step coils, state machines, or sequential function charts.

🛺 Timing

Timers (like ON-delay, OFF-delay, and retentive timers) are crucial for precisely controlling time-based actions and transitions in processes, allowing you to implement delays or specific durations, such as waiting a set number of seconds before opening a valve.

🔢 Counting

Counters, including up-counters, down-counters, and combined counter-timer logic, precisely track events and control actions, such as counting products on a line and triggering an event when a specific number is reached, while also enabling batching and overflow management.

➕ Arithmetic

Supporting both integer and floating-point operations, arithmetic functions allow for basic mathematical calculations on machine data, such as adding, subtracting, multiplying, or dividing numbers, which is essential for scaling sensor data, calculating setpoints, or implementing custom algorithms.

📦 Data Manipulation

Bit-level and word-level operations, including masking, shifting, rotating, and BCD conversions, facilitate moving, comparing, or swapping numbers like playing cards, enabling complex signal conditioning or data packing and unpacking.

🌐 Communication

Industrial protocols like Modbus, Profibus, and Ethernet/IP enable PLCs to "gossip" with each other, HMIs, or SCADA systems, thus coordinating with external devices, peer PLCs, or supervisory systems.

Typical Instructions in a PLC program:

- 🔄 **Sequencing** – step-by-step process control.
- 🕑 **Timing** – timers for delays and durations.
- 🔢 **Counting** – counting events, products, or cycles.
- ➕ **Arithmetic** – math on inputs, outputs, or variables.
- 📦 **Data Manipulation** – move, compare, or edit data.
- 🌐 **Communication** – talk to other devices and systems.

📌 What Are PLCs Used For?

A PLC's whole reason for existing is **to control and automate stuff in the real world** — especially in places too rough or complex for a normal computer.
Here's where you'll see them flexing:

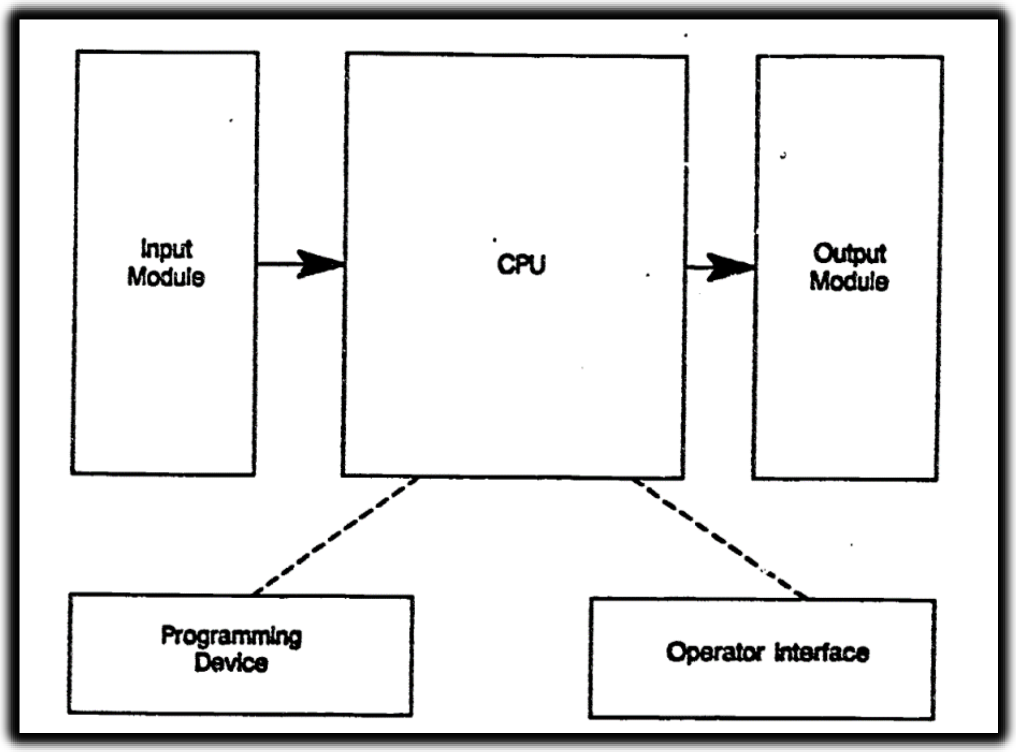
- ✅ **Machine Control:**
  They're the brain behind automated machines on factory floors — controlling motors, valves, and sensors with split-second precision.
- ✅ **Car Wash Systems:**
  Every step in an automatic car wash — soap, brushes, rinse, dryers — is sequenced by a PLC so your car gets clean without human intervention.
- ✅ **Bottling & Packaging Lines:**
  PLCs handle the timing, counting, and coordination needed to fill bottles, cap them, label them, and pack them at insane speeds.

- ✅ **Material Handling:**
  Conveyor belts, robotic arms, elevators, and palletizers all rely on PLCs to move products smoothly and safely through a plant.
- ✅ **Data Acquisition:**
  PLCs don't just control — they also **collect data** from sensors (temperatures, pressures, flow rates) and send it to HMIs, SCADA systems, or cloud dashboards.
- ✅ **Pipeline Monitoring:**
  Oil, gas, and water pipelines use PLCs to watch pressures, control valves, and trigger alarms or shutdowns if something goes wrong.
- ✅ **Hydroelectric Dams:**
  Gates, turbines, and safety interlocks are coordinated by PLCs to maintain power output and protect equipment.
- ✅ **Process Control:**
  Industries like pharmaceuticals or refineries use PLCs to handle multi-stage processes with loops, PID control, and precise logic.
- ✅ **Food Mixing or Cooking:**
  In large-scale food plants, PLCs run mixers, heaters, and coolers to hit exact recipes, temperatures, and timings.
- ✅ **Chemical Processing:**
  When handling chemicals, you need perfect timing, flow control, and safety shutdowns — PLCs are built for that level of reliability.

💡 Bottom line:

If it moves, mixes, counts, heats, cools, measures, or sequences — a PLC can control it. They're everywhere in modern automation, silently running the show behind the scenes.

Typical PLC System Components



A PLC isn't just a single magic box — it's a **team of modules** working together:

✅ Input Module

- Brings signals from the field (sensors, switches, limit detectors).
- Converts those real-world electrical signals into data the CPU can read.

✅ CPU (Central Processing Unit)

- The *brain* of the PLC.
- Reads input data, runs your program logic, makes decisions, and figures out what outputs should do.

✅ Output Module

- Takes decisions from the CPU and drives actuators (motors, solenoids, lights, valves) in the real world.
- Converts CPU signals into the correct power levels for field devices.

✅ Programming Device

- This is how *you* talk to the PLC.
- Used to create, download, and edit the logic (ladder diagrams, structured text, etc.).
- Think laptops, handheld programmers, or special config tools.

✅ Operator Interface (HMI – Human Machine Interface)

- The dashboard for operators.
- Shows process info (temperatures, states, alarms) and lets operators tweak parameters (like setting a new speed or temperature).

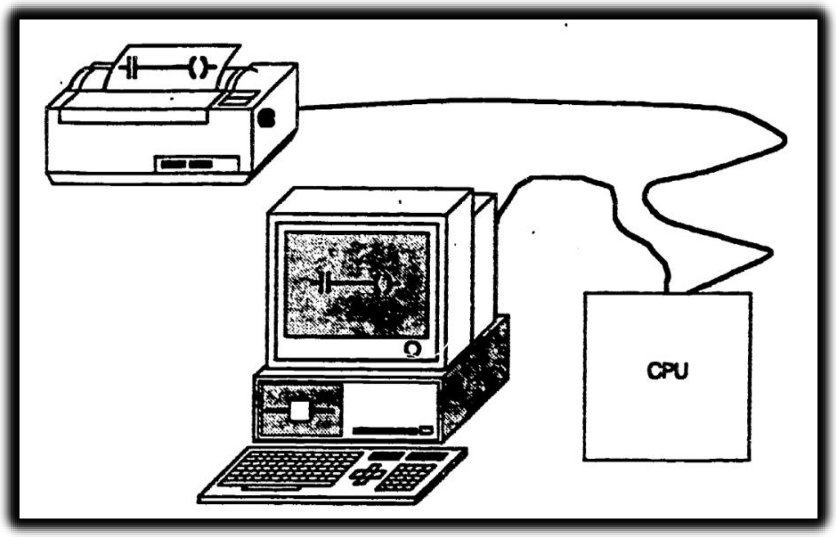
✨ How it all flows (based on the diagram):

1. **Input Module ➡ CPU:** Sensors send signals → CPU reads them.
2. **CPU does the logic:** Your program decides what should happen.
3. **CPU ➡ Output Module:** Sends commands to field devices.
4. **Programming Device:** Engineers use it to load or change the logic.
5. **Operator Interface:** On the plant floor, operators monitor and adjust the system in real time.

💡 Big picture:

A PLC is basically *eyes (inputs), a brain (CPU), hands (outputs)* — with engineers (programming device) teaching it, and operators (HMI) talking to it while it's running.

TERMINAL



📌 Terminals in PLC Systems

A **terminal** is a hardware device that acts as a **bridge between a human and a computer system** — basically, it's how people interact with machines before modern graphical interfaces became common.

🖥 Types of Terminals:

- **CRT (Cathode Ray Tube Terminal)**
  - ➢ Old-school monitor + keyboard setup.
  - ➢ Used for fast, real-time interaction with the system (entering commands, viewing status, editing programs).
  - ➢ Think of it as the OG "command-line interface" workstation in industrial settings.
- **PRINTER Terminal**
  - ➢ Not for controlling — but for **documenting**.
  - ➢ Used to get a printed version of your PLC program, error logs, process reports, or configuration files.

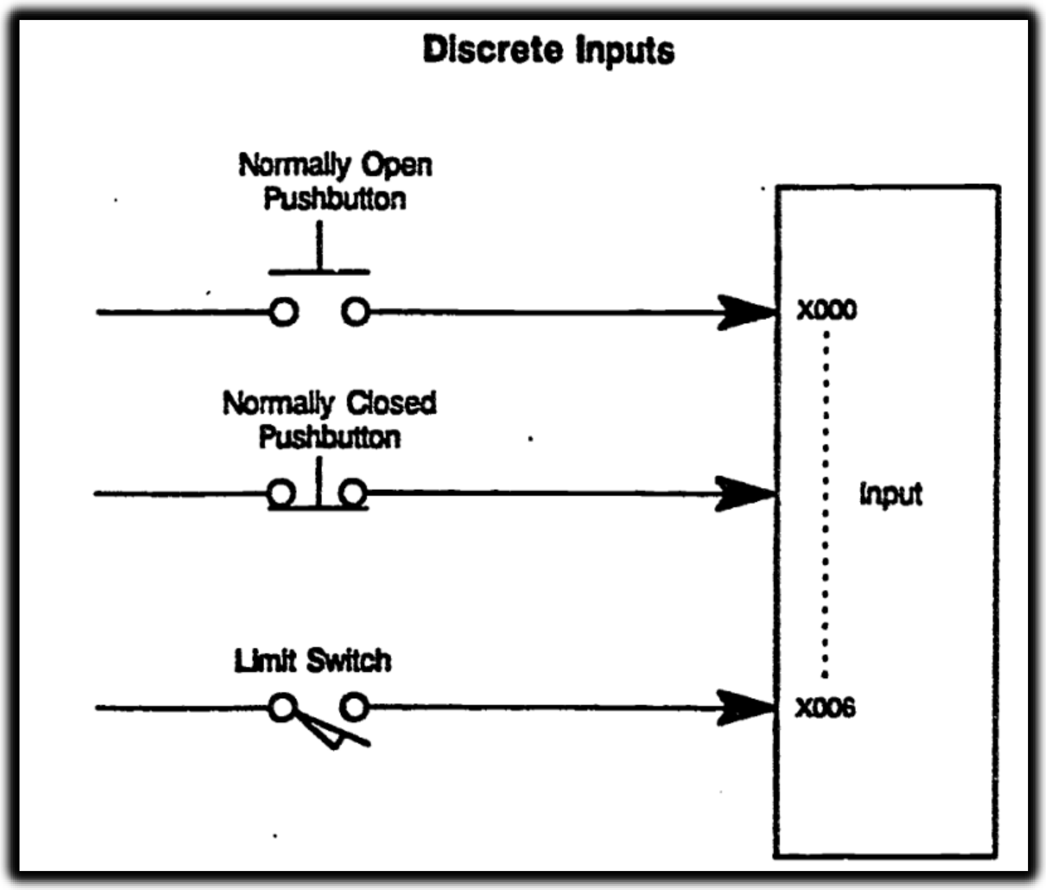
📄 Hardcopy

- Means a **physical printout**, usually on paper.
- For backups, records, or sharing with other engineers.

💡 Real Talk:

Before modern touchscreen HMIs and laptops, **terminals were the only way** to talk to your PLC. Today, they're mostly legacy — but understanding them helps when working on older systems or reading historical documentation.

📌 Discrete Inputs (a.k.a Digital Inputs)



💡 What is a Discrete Input?

A **discrete input** is a simple ON or OFF signal sent from a field device to the PLC.
It's **binary** — either HIGH (1) or LOW (0), no in-between.

🧠 Examples of Discrete Input Devices:

These devices act like *switches* — either giving power or not:

- 🟢 **Pushbuttons** (Normally Open / Normally Closed)
- ⚙️ **Limit switches** (detect positions of machine parts)
- 💧 **Float switches** (detect fluid levels)
- 🔄 **Toggle switches**
- 📈 **Flow switches / pressure switches**
- 🦶 **Foot pedals / safety interlocks**
- 🧲 **Proximity switches** (detect presence of objects without contact)

🔌 Wiring & Addressing: I/O Points

- **I/O points** are the terminals where field devices physically connect to the PLC input modules.
- Each input is mapped to a specific address in the PLC's memory.
  For example:

      X000 = 1st discrete input
      X006 = 7th discrete input

- This address can be auto-assigned or manually configured based on your PLC type and software.

✍️ Diagram Breakdown:

Let's decode the diagram you shared:

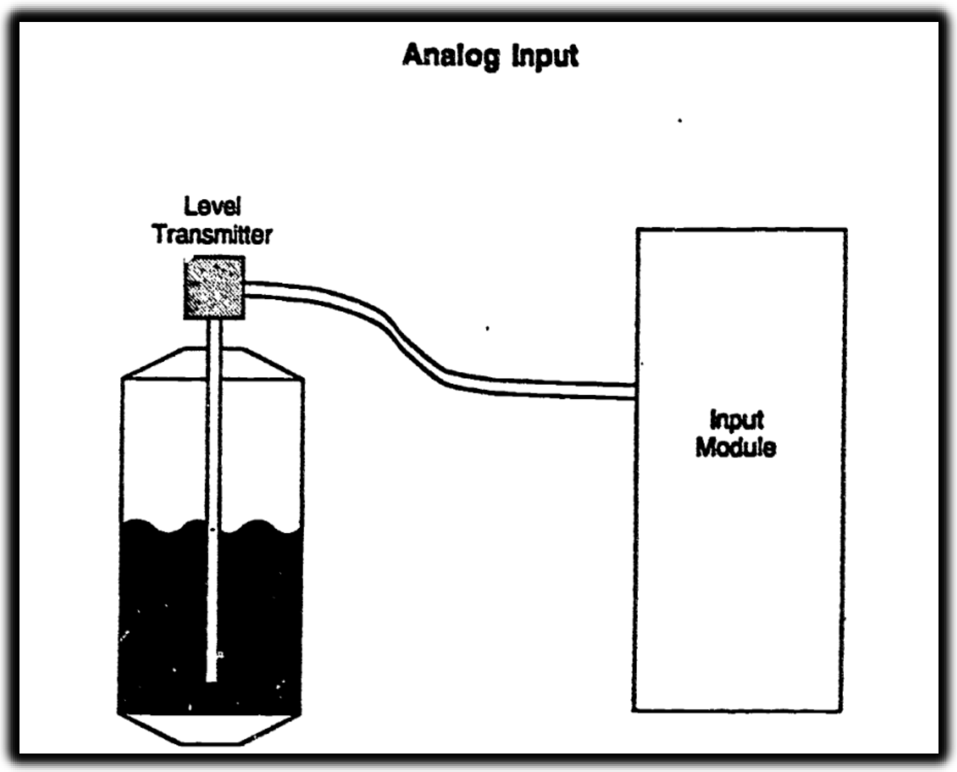
Device	Type	Connected to
Pushbutton	Normally Open	X000
Pushbutton	Normally Closed	X001 (assumed)
Limit Switch	Normally Open/Closed	X006

Each arrow represents a **signal wire** going into an **input point** on the PLC module. When the device is activated, the corresponding input bit (like X000) flips from **0 → 1** or **1 → 0** depending on the wiring logic.

🧠 Nick's Mental Model:

Think of discrete inputs as **the PLC's ears** — they tell it,
"Hey! This button was pressed."
"That valve reached its end."
"A box just passed the sensor."
And the CPU listens and acts based on your program logic.

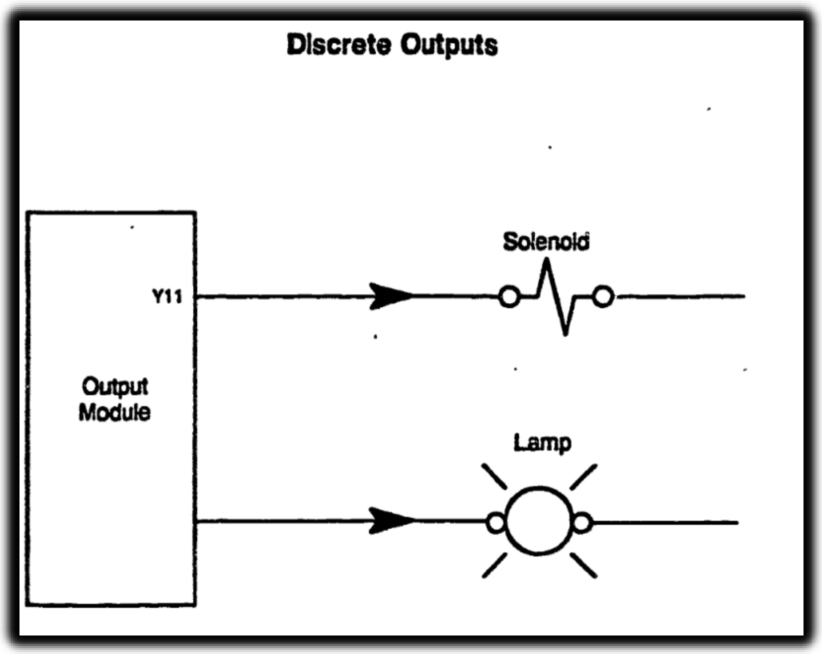
ANALOG INPUT



📌 Discrete Outputs (Digital Outputs)

Just like inputs tell the PLC what's happening outside, **discrete outputs** are how the PLC **tells machines what to do.**

A **discrete output** sends an ON or OFF signal to a field device — nothing fancy, just pure binary power delivery.



⚡ What Are Discrete Outputs Used For?

They control devices that either:

- turn ON/OFF
- open/close
- light up or go dark

🔧 Common Examples:

- **Solenoids** – to open/close fluid or air valves
- **Contactor Coils** – for switching high-power circuits
- **Indicator Lamps / Pilot Lights** – to show machine status
- **Buzzers / Alarms** – to signal errors or events
- **Small Motors or Relays** – in basic control setups

🧰 Wiring & Addressing:

Just like inputs, outputs are also connected to specific terminal points on the **output module**.

Each output gets a **unique address**, e.g.:

Y011 = 12th discrete output (starting from Y000)

These addresses can be set automatically or manually during setup/configuration.

🧠 Nick's Mental Model:

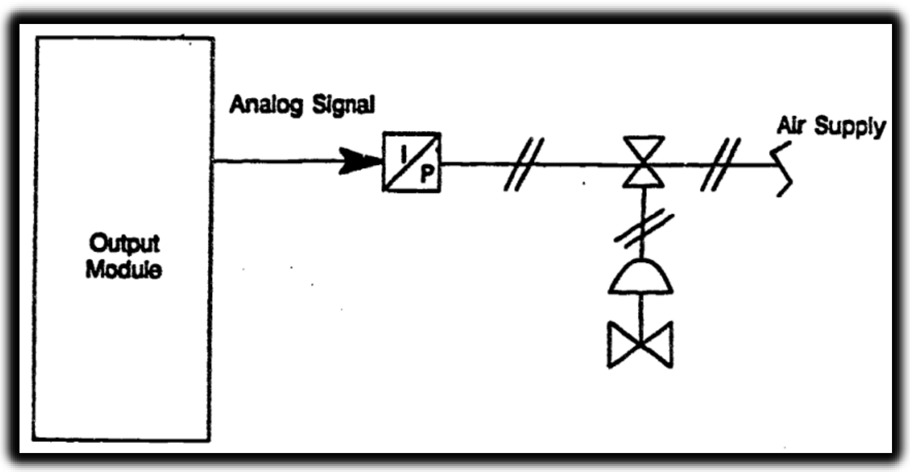
If **discrete inputs are the PLC's ears**,
then **discrete outputs are its hands.**
The CPU listens to what's happening → runs the logic → and then uses outputs to take action.

Example:

If X000 (button press) is ON, then Y011 (lamp) turns ON.

That's classic **input → logic → output** flow.

Analog Outputs



Analog Output refers to an electrical signal sent *from* the PLC *to* an external field device, where the value of the signal is continuously variable, not just ON or OFF. These outputs are typically used to control devices that need **precise, adjustable control**, rather than just binary states.

🌀 What Is an Analog Signal?

An **analog signal** is a continuously variable signal. It can represent a whole range of values — for example, anything from **0 to 10 volts**, or **4 to 20 milliamps** (mA). This is different from digital/discrete signals which are either **on (1)** or **off (0)**.

Think of it like a dimmer switch on a light — not just ON/OFF, but gradually increasing brightness. That's what analog signals do: provide a smooth range of control.

📤 How Analog Outputs Are Used

They're mainly used for **fine control** of physical devices that require variable levels of operation. Here are key examples:

- **Motor speed controllers** – to control the RPM of a motor smoothly
- **Flow control valves** – to let more or less fluid through
- **Temperature regulation systems** – for variable heating or cooling
- **Positioning systems** – like for a robotic arm that needs gradual, smooth movement

🛠️ Real-World Example:

Current-to-Pneumatic Transducer (I/P Transducer):

The PLC sends a **4–20 mA** current signal to the transducer. The transducer converts this electrical signal into a proportional **air pressure** that adjusts a **pneumatic valve** in a flow-control system.

- If the PLC outputs **4 mA**, the valve may open only slightly.
- If the PLC outputs **20 mA**, the valve may fully open.
- Anything in between gives **partial openings**, allowing **precise control of flow**.

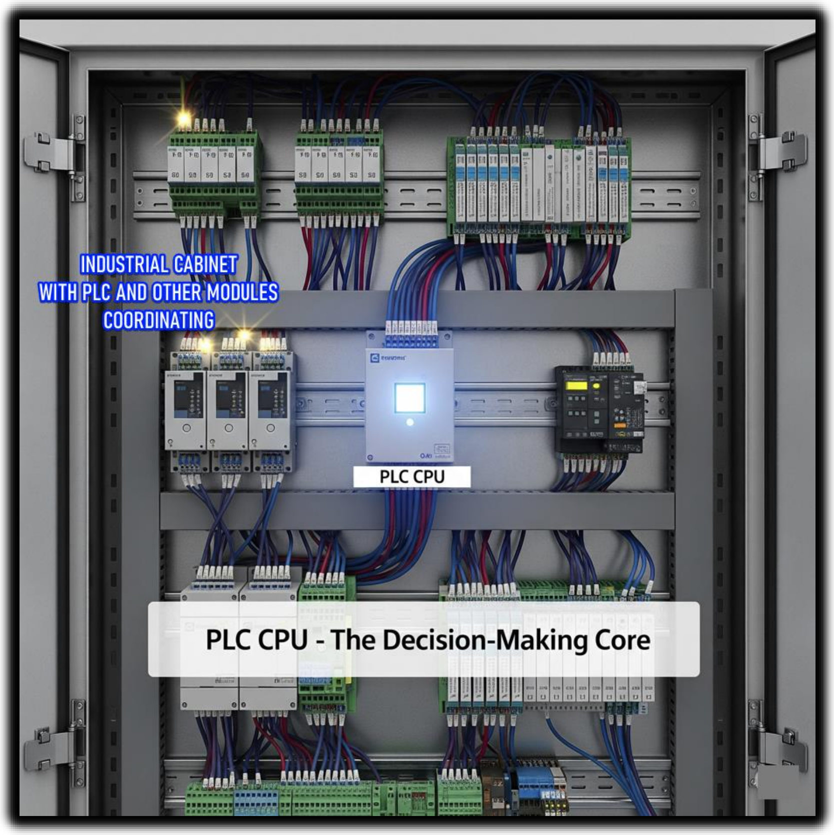
🧠 Extra Details to Keep in Mind:

- The **proportionality** of analog output depends on the whole control loop:
  - The **primary element** (sensor or input device)
  - The **transmitter** (converts sensor data to electrical signal)
  - The **controller (PLC)** and its output scaling
  - The **actuator** or device receiving the analog signal
- Sometimes, due to mechanical or electrical factors, the system might behave in a **non-linear (disproportionate)** way — meaning that doubling the signal doesn't necessarily double the effect.

📜 Summary (Quick Bites):

- **Analog output = variable signal from PLC (e.g., 4-20mA or 0-10V)**
- Used for **precise control** over devices like valves and motors
- Enables **gradual change**, not just ON/OFF
- Real-life use: controlling a valve in a water plant, adjusting motor speed in a conveyor system, etc.

🧠 Central Processing Unit (CPU) — The PLC's Brain



The **CPU** is the heart of the PLC. It's the boss that runs the show. It handles **decision-making** and manages **user memory**, both of which are **fully controlled by the programmer i.e. you 😈).**

The CPU has two main responsibilities:

🧩 1. Decision-Making Section

This is the logic engine. It's where all the **"if this, then that"** type of rules are executed.

✅ What does it do?

- It reads signals coming in from the field (sensors, switches, etc.)
- It **compares those inputs against your programmed instructions**
- Based on that, it decides what outputs to activate or deactivate

⚙️ Example:

"If the water tank is full, then turn off the inlet valve."

This logic is written in your program using conditions. The CPU constantly checks input values (like a float switch in the tank) and executes this rule if the condition is true.

So really, it's like a 24/7 vigilant robot, constantly scanning inputs, and flipping outputs based on your custom logic.

📦 2. User Memory Section

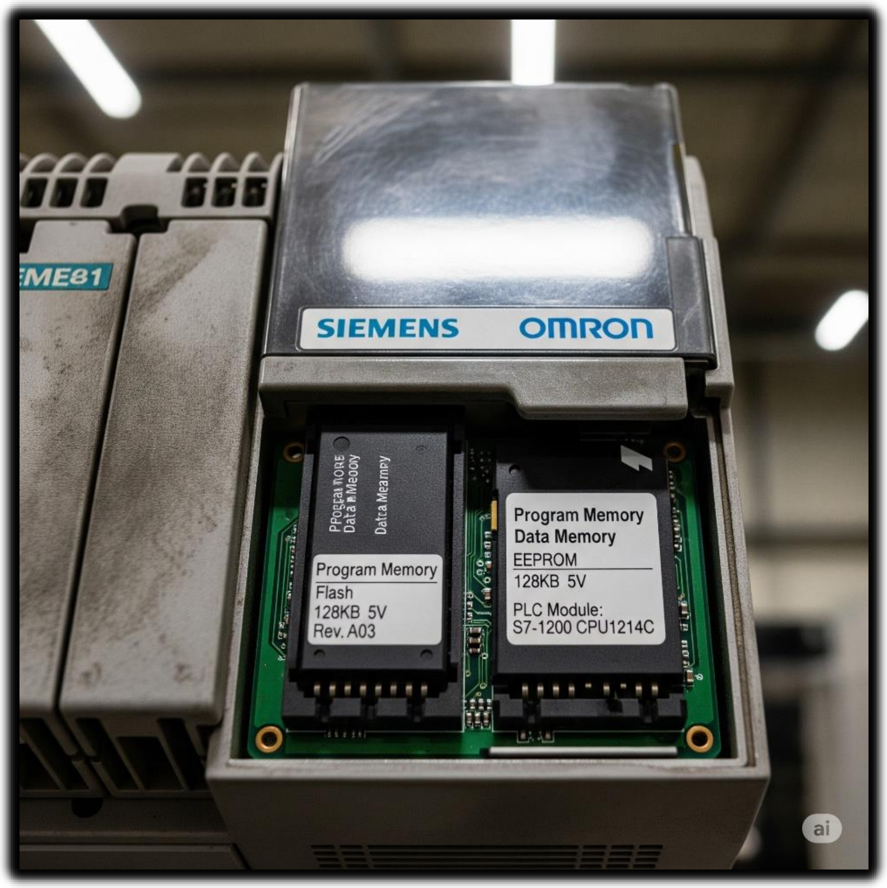
This is where your program and working data live.

🧠 What's stored here?

Instructions (Your actual program — logic, sequences, etc.)

Application-specific data, like:

- Current status of inputs and outputs
- Temporary storage for values (like timers, counters, internal flags)
- Recipes, setpoints, operator-entered data



📝 Real Talk:

If the CPU is the brain, then the **user memory** is like a combo of short-term memory (RAM) and long-term habits (program logic). Your logic tells it what to do, and the CPU keeps checking reality to follow those instructions.

🧪 Summary with Example Flow

1. Input says: “Tank is full” → sensed by a float sensor.
2. CPU reads this input.
3. It checks your program (user memory): IF tank is full THEN close valve.
4. Decision is made: Close the valve.
5. Output is activated to shut off the valve via a relay/solenoid.

That’s how **industrial automation magic** happens. Simple, logical, powerful.

🧠 Central Processing Unit (CPU) — The Brain of the PLC part 2

💭 Simple Explanation (Teen Mode)

The CPU in a PLC is like the brain of the system. It:

- **Thinks (Decision-Making):** Like deciding, “Hey, the tank is full — I better turn off the water valve.”
- **Remembers (User Memory):** It holds two things:
    1. ***The “how-to” list (instructions you give it).***
    2. ***The “what's happening” info (current readings or values from sensors).***

The CPU does *exactly* what it’s told. If the instructions are dumb, it’ll do dumb stuff — just *very* fast and *very* accurately. It doesn’t ask questions. It just obeys.

🛠 Software, Hardware, Firmware — Let's Demystify This

Term	Meaning
Program	A sequence of instructions written to perform a specific task. Organized Instructions work together to form a program.
Software	A collection of programs and related data that work together to control a system. Software enables computers and devices to carry out complex operations.
Hardware	The physical parts of a computer or control system — such as the CPU, memory, and input/output devices. These are the tangible components required for system operation and user interaction.
Firmware	Software that is permanently stored on hardware components, typically in read-only memory. It provides low-level control for the device's specific hardware — for example, the internal operating system of a PLC.

🔁 The CPU runs two "layers" of code:

- **Firmware**: Low-level code (like an OS) that makes the PLC even *function*. You can't edit this — it's stored inside the chip.
- **User Software**: Your custom ladder logic or structured text. This is the code that makes your specific plant or machine behave how you want it to.

Just like a PC:

- **Windows 10** = PLC firmware
- **Chrome** = Your PLC program

💡 Languages the PLC Understands

1. **Relay Ladder Logic (RLL):**
   - Graphical.
   - Inspired by old-school relay schematics.
   - Most beginner-friendly.
   - Looks like electrical diagrams = makes sense to electricians.
2. **Machine Stage Programming:**
   - Think: ladder logic + sequential flowchart.
   - Perfect for machines with stages (like washing machines, bottling lines, etc.).

🔥 Wrap-Up Quote:

The CPU is not *smart*. It's *fast*. It's not creative — it's obedient. Give it bad logic, and it'll mess things up with **machine-level precision**.

🧠 PROGRAMMING A PLC: THE BIG PICTURE

When you hear "PLC Programming," you're basically talking about telling a small industrial computer (the CPU of the PLC) **what to do and when to do it**. But that involves several components working together:

🔧 1. HARDWARE

This is the **physical stuff** — the actual PLC box, the input/output modules, the wiring, power supply, etc.

- **Examples**:
  - CPU (Central Processing Unit)
  - Input/Output modules (for sensors, actuators)
  - Power supply
  - Communication ports
  - Mounting base or chassis

📀 2. FIRMWARE

This is the **built-in software** inside the PLC hardware — like its operating system.
It controls basic operations (scan cycle, timing, comms, etc.) and lives inside a chip.

- **Think of it like this**:
  - **Firmware = Windows for your PLC**, but you can't uninstall or modify it.
  - It's burned into the memory of the PLC (ROM or flash).

📄 3. SOFTWARE

This is **your program** — the logic the PLC will execute based on the conditions it reads.
It's created by you (the programmer), uploaded into the PLC, and stored in its **user memory**.

- **User memory** stores:
  1. The actual instructions (the program).
  2. Process-dependent info (e.g., sensor values, counters, flags).

💻 4. PROGRAMMING TOOLS

You need a device to **write, upload, and monitor** your programs. These are usually:

- **PCs** (modern choice)
- **Handheld programmers** (old-school, niche but still in use)

⚙️ 5. PROGRAMMING MODES

There are two ways to interact with your PLC during programming:

Mode	What it Means
Offline/Programming Mode	You're writing/editing the program on your PC, but it's **not yet in the PLC**.
Online/Running Mode	You're connected to the PLC, and you can **monitor, modify, or debug** the program live.

Offline Mode **-** This mode is where you do all your initial development, major overhauls, or bug fixes without affecting the live process. It's like working in a sandbox. Changes only exist on your computer. The PLC itself is either running an old program, or if it's new, it has no program at all.

Online Mode **-** Where you can make small changes to the program without stopping the PLC. This is usually for minor tweaks, debugging or bug fixes. For bigger changes, you'd typically go offline. It's like adjusting your car's mirrors while driving – small, immediate changes.

💬 6. PROGRAMMING LANGUAGES

You can't just write English and expect the PLC to understand. It needs **languages it understands**:

a) Relay Ladder Logic (RLL)

- Most common.
- Looks like electrical relay schematics.
- Visual and great for electricians and techs.
- Uses contacts, coils, timers, counters, etc.

b) Boolean Logic

- The math behind all digital logic:
    - ➢ AND, OR, NOT, NAND, NOR
    - ➢ Often represented graphically in RLL

c) Computer-like Languages (Text-Based)

Some PLCs support structured text, instruction lists, or even Python-like languages. This depends on the brand (Siemens, Mitsubishi, Allen-Bradley, etc.).

🔥 Bonus Insight: Double Execution in the CPU?

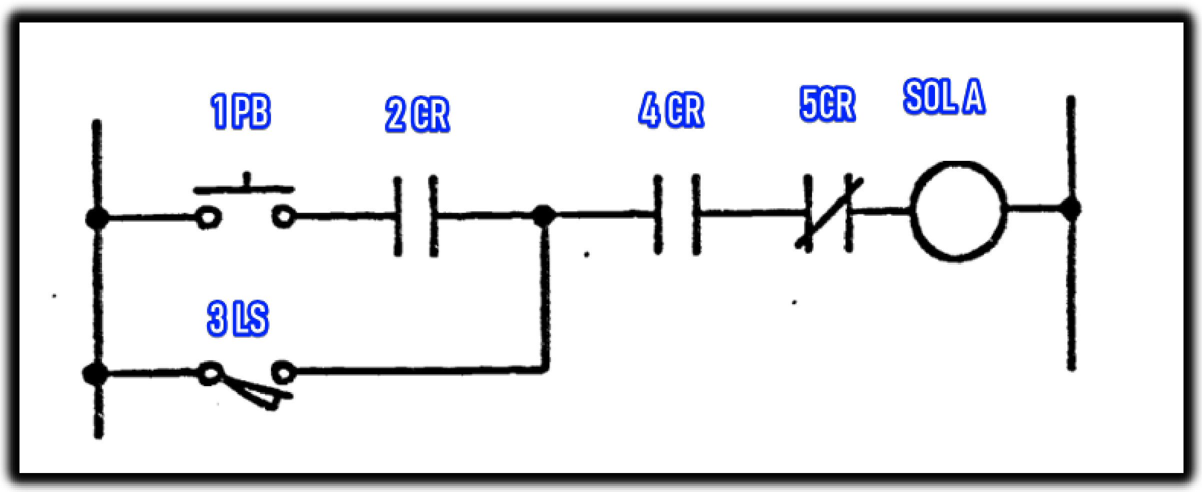
You're **spot on** with this analogy:

"The CPU handles double work — system stuff (like starting itself) and user programs (PLC code)."

Yes, the **firmware handles the PLC's own health and system tasks**, while the **user program runs your logic for controlling the machines**. They coexist but don't step on each other.

- **Hardware** = physical components
- **Firmware** = pre-installed OS of the PLC
- **Software** = the program you write
- **Programming devices** = PC or handheld (images in the other book)
- **Modes** = Off-line (write), On-line (monitor/test)
- **Languages** = Ladder Logic (visual), Boolean (logic base), or text-based (advanced)

RELAY LADDER DIAGRAM



Based on the relay ladder diagram, here's a short description of its components and operation:

This diagram represents a control circuit using rungs and rails, common in industrial automation. There are six distinct components involved in controlling a solenoid.

The six components are:

1. **1PB (Push Button):** This is a normally open (NO) push button. When pressed, it allows current to flow.
2. **2CR (Control Relay 2):** This is a normally open (NO) contact from control relay 2. It closes when control relay 2 is energized.
3. **3LS (Limit Switch):** This is a normally open (NO) limit switch. When actuated, it allows current to flow, providing an alternative path to the 1PB.
4. **4CR (Control Relay 4):** This is a normally open (NO) contact from control relay 4. It closes when control relay 4 is energized.
5. **5CR (Control Relay 5):** This is a normally closed (NC) contact from control relay 5. It opens when control relay 5 is energized, interrupting the circuit.
6. **SOL A (Solenoid A):** This is the output device, a solenoid, which is activated when current flows through it.

In operation, Solenoid A will energize if either the 1PB is pressed AND 2CR is closed, OR if the 3LS is actuated.

Additionally, for the solenoid to energize, 4CR must be closed AND 5CR must NOT be energized (meaning its normally closed contact remains closed).

Essentially, it's a logic circuit where multiple conditions must be met in sequence or parallel for the solenoid to activate.

CLARIFY FROM HERE:
This is my question: THERE'S SOEMTHING CONFUSING ABOUT SUCH LADDER DIAGRAMS COZ, HOW CAN YOU SAY 4 CR MUST BE CLOSED WHICH IS GOOD I UNDERSTAND, BUT 5CR MUST REMAIN AS IT IS, DOESN'T ELECTRICTY FLOW FROM POSITIVE RAIL TO NEGATIVE RAIL. AND MUST PASS THROUGH 5CR? OR THAT'S NOT HOW LADDER LOGIC WORKS?

🧠 First: YES — Electricity flows left to right in a ladder diagram.

- Just like a book, ladder diagrams are read from **top to bottom** and **left to right**.
- The **left vertical rail = L (Live)**
- The **right vertical rail = N (Neutral)**
- Current flows **from L to N** through the rungs — same way logic flows.

🧩 Now: Let's decode the mystery around 5CR

You're 100% right to question:

"Doesn't the current still have to go through 5CR? So how can you say '5CR must not be energized'?"

This is where **ladder logic's secret sauce** kicks in — the concept of **contacts being "controlled" by other relays or coils**.

🔧 THE CORE QUESTION:

> **If current flows through relay coil 4CR and it's wired in series with 5CR, doesn't that automatically energize 5CR and change its state too? Or is it just passing through a contact?**

Let's untangle that right now.

We gotta separate two things cleanly:

Relay Coil (e.g. 4CR, 5CR)	Relay Contact (e.g. 4CR NO, 5CR NC)
This is the **electromagnet** part that you energize with electricity.	These are the **switches** controlled by the relay coil.
You pass **current directly into the coil** to energize it.	They **change state** depending on whether the **associated coil** is energized or not.

Let's expand on that a little bit more...

CHARACTERISTIC	RELAY COIL	RELAY CONTACT
Function	Creates a magnetic field when energized.	Acts as a switch, opening or closing a circuit.
Nature	Electromagnet (input component).	Mechanical switch (output component).
Control Input	Requires direct electrical current to activate.	Controlled by the magnetic field of its associated coil.
Circuit Role	Part of the control circuit.	Part of the controlled (load) circuit.
State Change	Energized/De-energized.	Open/Closed (NO) or Closed/Open (NC).
Example Symbol	Circle with label (e.g., 4CR, 5CR).	Two lines (NO) or two lines with a diagonal slash (NC) with label (e.g., 4CR NO, 5CR NC).

👇 **Back to Your Circuit:**

Let's say you have:

Here’s what matters:

4CR and 5CR are not actual wires or power routes.

They're **contacts** controlled by relays **somewhere else**.

You’re not "passing electricity **through** a relay coil." You're passing electricity through **relay contacts**.

🔄 So when you close the 4CR relay, it energizes its **own coil**, and **its own contacts change** wherever they are.

✋ But if there's a **5CR NC contact** in that rung, whether it allows current or not **depends on the state of 5CR's coil**, **not** on the current coming from 4CR.

In short:

- **You’re not passing current into 5CR coil here.**
- You’re passing through the **contact** controlled by 5CR.
- So, if 5CR is energized **somewhere else**, its NC contact becomes **open**, breaking the circuit.

✅ The Key Takeaway:

Current doesn’t change a contact's state just by “passing through it.”

A contact changes state **only when its matching coil is energized**.

If you're going through a 5CR contact, its state depends only on whether **5CR’s coil** is energized.

🔄 So your thought:

“Since it must pass through 5CR, doesn’t it automatically open?”

❌ **Nope.** It opens only if the 5CR relay coil is energized. Otherwise, it stays in its default state (normally closed or normally open).

💡 Analogy (Gen Z mode):

Think of it like **smart home lights**.

- 4CR is a smart bulb. You control it with your phone (energize the coil), and it lights up (closes the contact).
- 5CR is another smart bulb, but its switch is somewhere else.
- Just because you're sending power to bulb A doesn't mean bulb B will react — unless **its own switch (coil)** is triggered.

👀 Bonus: Physical vs Ladder Logic

In real-world relays:

- **The contact is just a mechanical switch** inside a relay block. No power goes *through* a relay coil by just touching a contact.
- You energize a coil -> internal switch flips -> your circuit reacts.

✍️ Bottom Line:

🔌 Electricity **through a contact** ≠ electricity **through a relay coil**

🧠 Contacts change state only if their **relay coil is energized**.

So, your 4CR energizing doesn't affect 5CR unless there's a link somewhere in the logic to **energize 5CR's coil**.

💡 The BIG mistake students and teachers make:

🔌 There are two things going on with a relay:

1. **The coil** (e.g., 4CR or 5CR coil) – when energized, **it changes the state** of that relay's contacts.
2. **The contacts** (NO or NC) – are just **switches** that open or close **based on whether the coil is energized**.

Mistake? Thinking **current "passing through" a contact** changes its state.

FALSE. The contact's state **is not changed** by current *passing through* it — its state is only changed if the **coil** tied to it is energized.

🧠 Example Walkthrough:

Let's say we have:

- 4CR coil (controlled by some condition)
- 5CR contact (let's say it's a Normally Closed [NC] contact)

And 4CR's current is flowing **through** that 5CR contact. Here's what matters:

- If **5CR coil** is **not energized**, the contact **stays NC** → current flows normally.
- If **5CR coil is energized**, then and only then does its contact **open** → breaking the path.

➡ The relay **contact is just a puppet.**
➡ The relay **coil is the puppeteer.**

🔁 Don't Mix Up These Two:

What happens when you energize a COIL?	What happens when you energize a contact?
All contacts **linked to that relay** change state (NO ↔ NC).	Nothing. Contacts don't *do* anything themselves; they just pass or block current depending on their state.

🔧 Final Nail:

If current goes through a contact — **that does not change the contact.**
The only thing that **changes a contact's state** is **the COIL of that same relay being energized**.

💬 You Said:

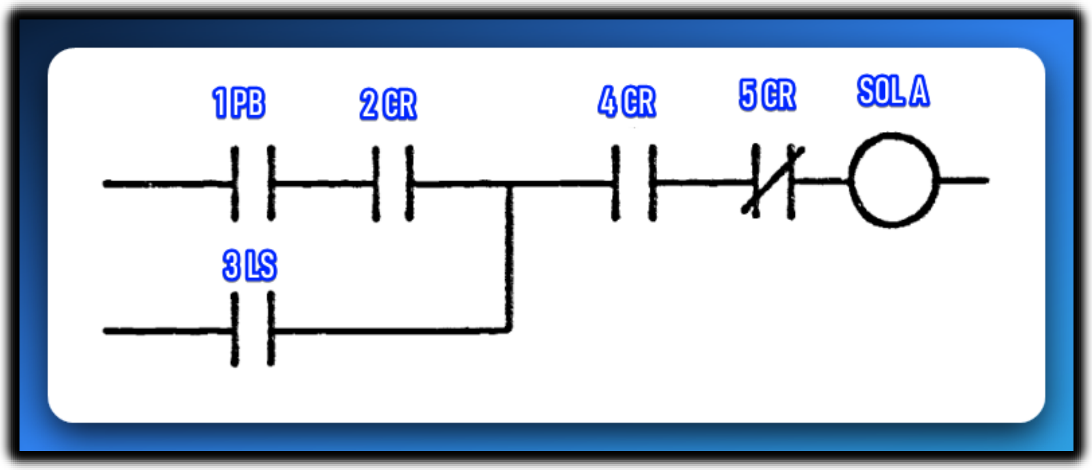
"Doesn't current already pass through 5CR so it becomes open contact?"

NOPE.
That's where it tripped you up. The 5CR contact will only become open if the **5CR coil** is energized — not because current passed through it.

🎯 Burn this in your memory:

“Coil controls the contact. Contact does not control the coil.”

FREE FORMAT EQUIVALENT PC DIAGRAM FOR THE SAME IMAGE DISCUSSED?



BOOLEAN STATEMENT?

$$((1PB \bullet 2CR) + 3LS) \bullet 4CR \bullet \overline{5CR} = SOLENOIDA$$

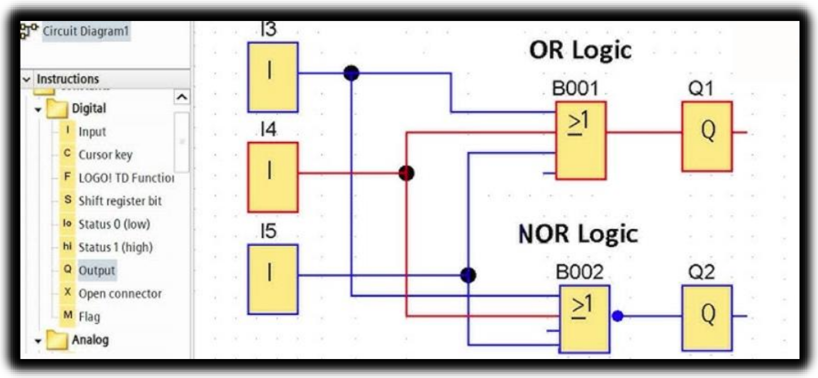
CODE OR MNEMONIC LANGUAGE?

```
LOAD   1PB    // Get 1PB's value.
AND    2CR    // Combine with 2CR.
OR     3LS    // Take previous result OR 3LS.
AND    4CR    // Combine with 4CR.
CAND   5CR    // AND with NOT 5CR.
STORE  SOLA   // Save final result to SOLA.
```

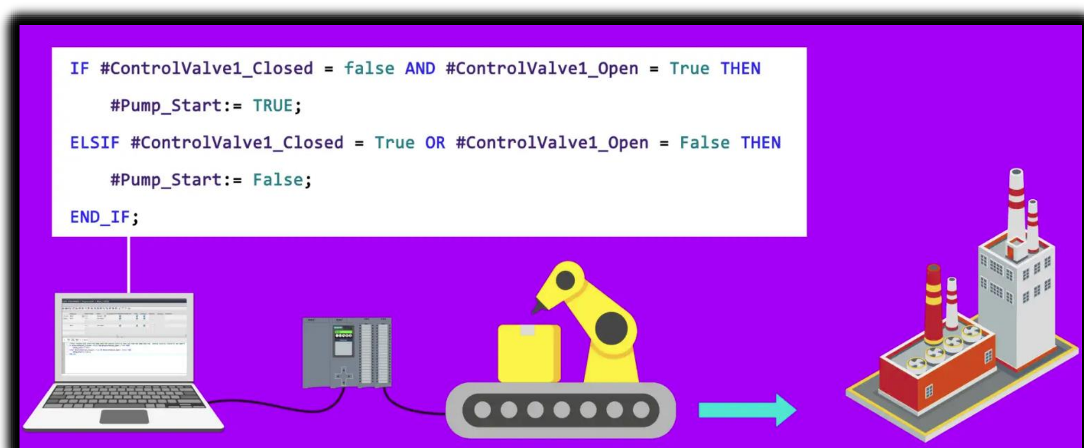
🔧 PLC Programming Languages — Comparison Table

Ladder Logic (LD) is the most commonly used language in PLCs, especially in standard industrial applications. It resembles electrical relay diagrams, which makes it intuitive for electricians to understand and troubleshoot.

Function Block Diagram (FBD) is often used in automation-heavy systems. It allows users to drag and drop logic blocks, making it especially appealing to visual thinkers who prefer graphical representations of control logic.



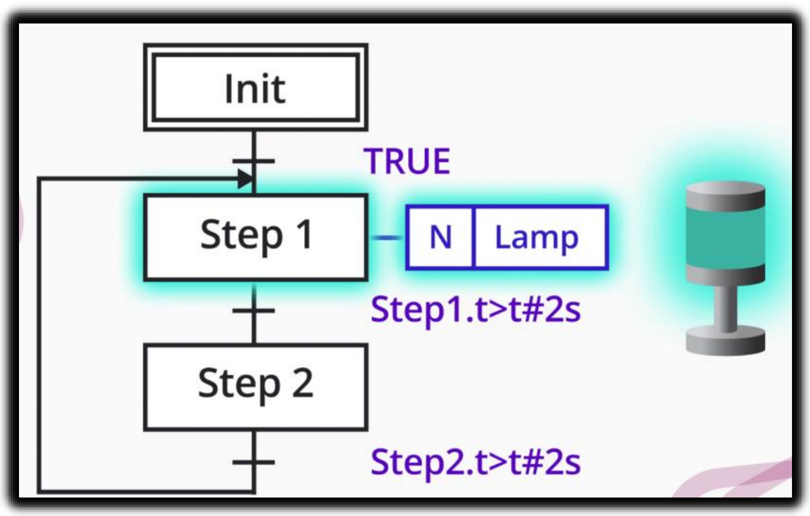
Structured Text (ST) is typically used in advanced control systems. It resembles the Pascal programming language and is best suited for applications involving complex logic or mathematical operations.



Instruction List (IL) is largely obsolete in modern PLCs. It is a low-level language that resembles assembly code. While deprecated, it may still be encountered in older systems.

```
# Function block call example
CAL INST_CMD_TMR(IN:=%IX5.0.0, PT:=T#300ms) # Call timer function
LD INST_CMD_TMR.Q # Load timer output
ST BOOL1 # Store to boolean variable
# Arithmetic operation example
LD 1.000e+3 # Load initial value
ST REAL1 # Store to REAL1
MUL REAL1 # Multiply REAL1 by itself
SUB (4 # Begin subtraction operation
    MUL( 1.0 # Multiply by 1.0
        MUL2_REAL((*IM1:=CR(REAL),*) IN2:=(2.0)) # Complex multiplication
    )
)
```

Sequential Function Chart (SFC) is ideal for batch processes. It is designed to represent step-by-step workflows using a visual state machine approach, making it excellent for modeling sequential operations.



🚨 **TLDR**: *Ladder Logic is king* because plant technicians already know relay circuits. Familiar = faster = fewer errors.

🤖 PLC vs. Traditional Control Systems

PLCs dramatically outperform traditional relay systems in every key area—including flexibility, reliability, data collection, expandability, repeatability, and space efficiency—making them the clear choice for modern industrial control.

COMPARISON POINT	PLCS 🔥	OLD-SCHOOL RELAYS 😵
Flexibility	Can reprogram in software. Update logic with a few clicks, like updating an app.	Need rewiring physically. Changing logic means literally moving wires around.
Reliability	Designed for harsh environments. Solid-state parts mean fewer things to break.	Prone to mechanical failure. Physical contacts wear out, leading to misfires.
Data Collection	Built-in sensors, counters, memory. They can log everything, like a machine's fitness tracker.	Basically none. They just switch things on/off; no data insights.
Expandability	Add new modules anytime. Just snap in extra I/O or comms cards as needs grow.	Not scalable easily. Adding functions often means building a whole new panel.
Repeatability	High. Logic doesn't drift. Once programmed, a PLC executes the same sequence perfectly every single time, like a robot's exact dance moves.	Can wear out, misfire. Mechanical parts degrade, causing inconsistent timing or faulty operations over time.
Space	Compact modules. A small PLC can control a huge system, saving precious cabinet space.	Requires large panels. Each relay is bulky, so complex systems need massive control boards.

💻 PLC vs. Personal Computers (PCs)

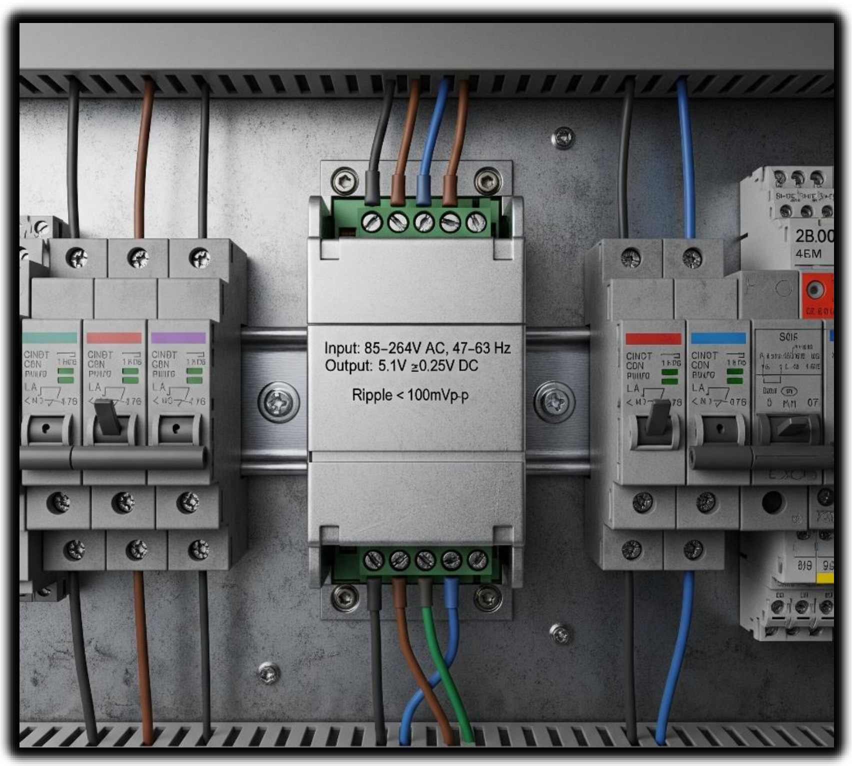
PLCs are purpose-built for the brutal realities of industrial environments. PCs, while powerful, just can't hack it when things get tough…💪

FEATURE	PLCS	PCS (GENERAL USE)
Operating Temp	0° to 60°C. Built to handle extreme heat or cold on the factory floor.	10° to 35°C. Designed for comfy office environments, not harsh conditions.
Storage Temp	-20° to 70°C. Can survive being stored in unheated warehouses or hot transport.	Varies, usually not ruggedized. Needs stable conditions; extreme temps can damage components.
Humidity	5–95%, non-condensing. Handles damp factory air without breaking a sweat.	Often lower tolerance. High humidity can cause short circuits or corrosion.
Air Quality Tolerance	No corrosive gas allowed. Built to resist industrial fumes and particles.	Needs clean, dust-free air. Dust and corrosive gases quickly degrade internal parts.
Vibration Resistance	MIL-STD 810C Level. Designed to withstand constant machine vibrations without issues.	Weak to medium. Continuous vibration can loosen connections or damage hard drives.
Shock Resistance	MIL-STD 810C. Can take a bump or drop; built for tough industrial handling.	Weak. A sudden jolt can easily damage sensitive internal components.
Voltage Withstand	1500VAC between key sections. Built to handle power spikes and electrical noise.	Not rated for that. Vulnerable to voltage fluctuations common in industrial settings.
Insulation Resistance	20MΩ @ 500VDC. High insulation prevents electrical leakage and ensures safety.	Standard at best. Less robust insulation means higher risk in high-voltage environments.
Noise Immunity	Meets industrial NEMA specs. Filters out electrical interference from other machinery.	No serious noise protection. Easily affected by electromagnetic interference, causing glitches.

🧠 Electrical Specs of a Sample PLC Power Supply:

- **Voltage Input Range**: 85–132V or 170–264V AC
- **Frequency Range**: 47–63 Hz
- **Input Current**: ~1.3A
- **Inrush Current**: Max 20A (short burst)
- **Input Power**: ~50W
- **Output Voltage**: 5.1V ± 0.25V (low ripple)
- **Ripple**: < 100mVp-p
- **Output Current**: Ranges 0.1A – 3.7A depending on voltage

📎 *Basically, PLCs are built to work in factories, next to loud motors, heat, vibration, electrical spikes... PCs would be crying in a corner.*



Summary Box:

🧠 **Why Pick PLCs?**
Because they're *optimized for the real world*. They're not fragile, fussy computers. They're reliable, rugged, and repeatable — and they speak the language electricians already know: ladder logic.

💻 PLCs vs. Computers & PCs (Part B: Software + Programming)

FEATURE	PLCS	COMPUTERS / PCS
Diagnostics	Built-in self-diagnosis on I/O modules. Can pinpoint faults fast, like a car with its own check engine light.	Usually needs external software/tools for diagnostics. You often need to download apps or run scans to find issues.
Programming Style	Uses ladder logic. Plant electricians love it because it's visual and similar to relay schematics they already know.	Can run any language (C, Python, Java). Super flexible, but needs a trained programmer who knows the code.
Program Flow	Executes one program from start to end, line-by-line (sequential execution). It's like following a recipe step-by-step.	Can run multiple programs in parallel using multi-threading or multitasking. Juggles many tasks at once, like a busy chef with multiple dishes cooking.
New PLC Features	Modern PLCs support: Subroutines (for modular code). Interrupt routines (event-driven logic). Jump instructions (skip parts of code when needed).	PCs already had this — but now PLCs are catching up and becoming more "intelligent" with these advanced features.

🛠 Part C: Maintenance Comparison – PLCs vs PCs

PLCs win big on maintainability and ease of integration for industrial settings. They're designed for quick fixes by on-site teams, minimizing downtime.

FEATURE	PLCS	COMPUTERS / PCS
Repair & Replacement	Uses modular hardware – just pull out and replace the faulty part. Like swapping a LEGO brick.	Repairs are trickier; often need a tech-savvy person or full replacement. More like fixing a complex puzzle.
Field Device Integration	I/O modules and interfaces built directly into the PLC — plug-and-play. Connects directly to sensors and actuators without hassle.	PCs need extra interface cards/adapters to connect to real-world devices. Requires specialized hardware add-ons and drivers.
Ease of Use	Designed for electricians & technicians – simple setup, clear documentation. Built for quick, practical use by on-site teams.	Requires a trained engineer, especially for low-level hardware interfacing. More complex setup and troubleshooting often needed.

🧠 Quick Summary: Why PLCs Win in the Factory

- They **diagnose themselves**.
- Use **relay-style logic** so any technician can understand.
- **Handle harsh environments** without crying.
- Can **run 24/7** with minimal maintenance.
- PCs may be smarter, but **PLCs are built tougher** for plant floors.

PLC Sizes and Their Applications (Simplified & Upgraded)

🔢 1. Understanding PLC Size Categories

PLCs come in different "sizes" — not physically, but in terms of memory, input/output capacity, and what kinds of tasks they can handle.

Think of it like phones: a basic **flip phone** vs. a **midrange Android** vs. a **flagship iPhone Pro Max.**

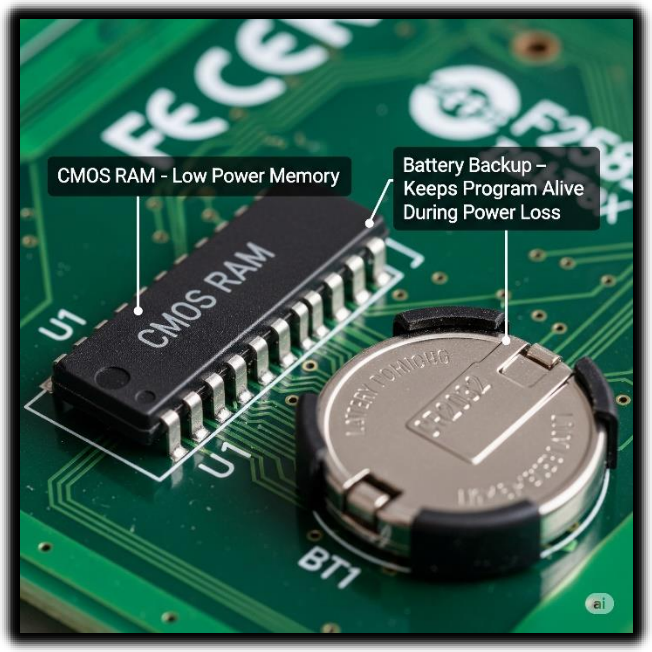
CATEGORY	I/O POINTS	MEMORY (RAM)	USE CASE	FEATURES
Small	Up to 128 I/O	~256 bytes to 2 KB	Basic control tasks. Ideal for simple, standalone machines or small processes, like a single conveyor or a packaging unit.	Only discrete (on/off) control. Think simple switches and lights.
Medium	128 – 2048 I/O	~2 KB to 32 KB	Moderately complex systems. Great for controlling multiple interconnected machines or a small production line.	Handles both discrete and some analog signals (like temperature or pressure readings).
Large	Up to 8192+ I/O	32 KB to 750 KB+	Large industrial plants. The powerhouse for managing entire factories, complex processes, or multiple production lines.	Advanced capabilities: extensive analog control, data logging, and robust communication options (like connecting to databases or other PLCs).

💾 2. Memory Tech Used in PLCs

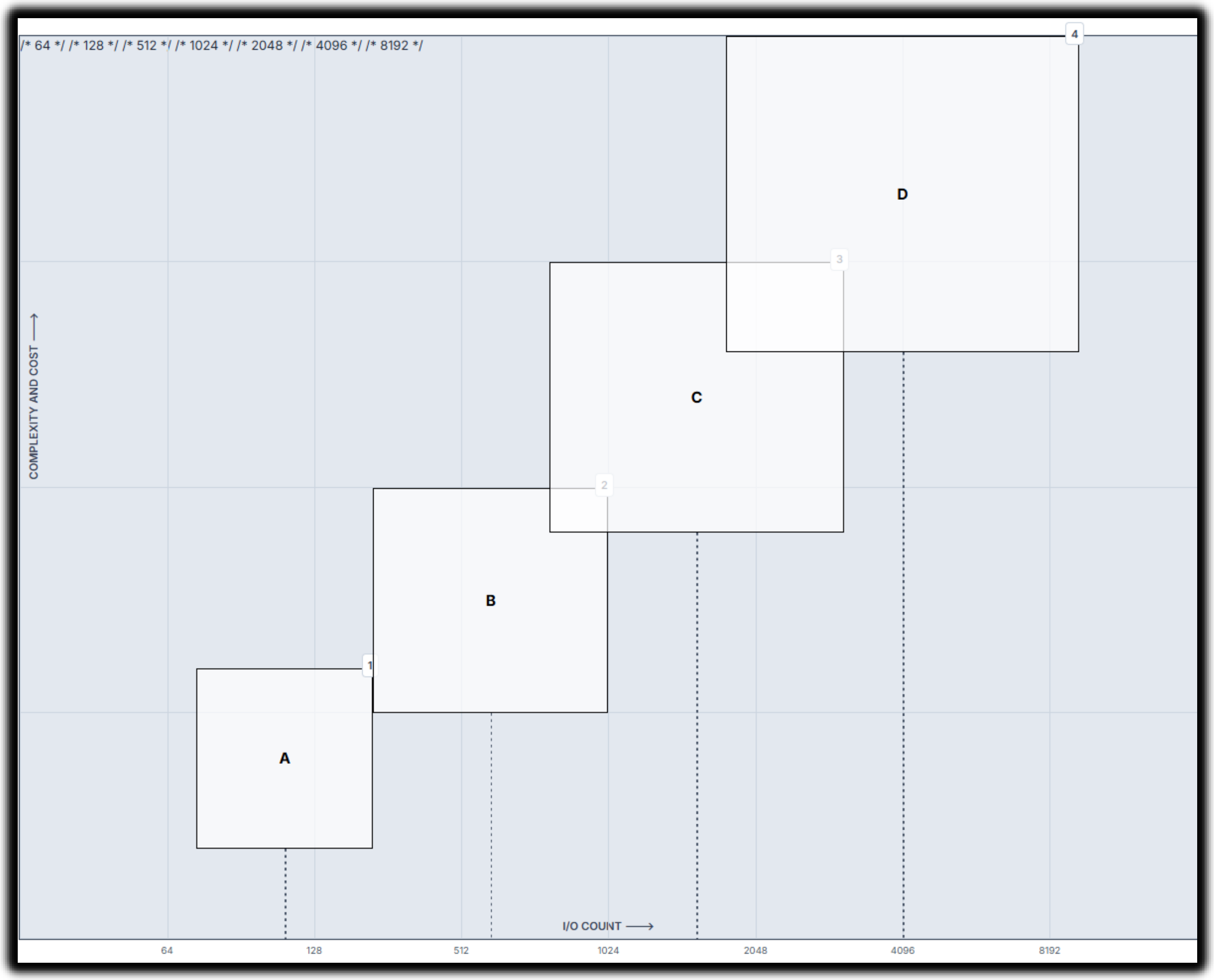
PLCs rely on specific memory types to keep their programs running, even when the power flickers. Here's a quick look at the key tech.

MEMORY TYPE	NOTES
CMOS RAM	Used in most PLCs. It's super low power, which is why it works great with a battery backup.
Battery Backup	Keeps your PLC program safe and sound even after power loss. Think of it as a tiny, dedicated power bank for your code.

Here’s a realistic sample…



3. PLC Size, Complexity, and Cost Relationship



Quick Notes on PLC Sizing:

- **I/O Count (X-axis):** This is like the number of "ports" or connections your PLC has for sensors and actuators. More I/O means it can control more stuff.
- **Complexity & Cost (Y-axis):** As you move up, the PLC can handle more intricate tasks and, naturally, costs more. Think of it as going from a basic smartphone to a high-end server.
- **Boxes (A, B, C, D):** These represent different PLC categories (Small, Medium, Large) and show that as I/O count increases, so does the complexity it can manage and its price.
- **The Trend:** The chart clearly shows an upward trend – bigger PLCs handle bigger jobs and come with a bigger price tag.

Just like upgrading your gaming rig, the more "muscle" (I/O count) a PLC has, the more complex tasks it can handle, and yep, the higher the price tag. This chart shows that sweet spot.

🔗 3. Scope of Applications

🚀 PLC Capabilities: What They Do	
Discrete Control	ON/OFF control (e.g., conveyor belts, push buttons). Simple binary decisions.
Analog Control	Varying input/output (e.g., temperature, pressure). Handling continuous signals for fine-tuning.
Data Communication	Talking to other systems (via RS-232, 20mA loop, etc.). Sharing info across the factory floor.
Master/Slave Networking	One PLC controls several others. Like a lead robot coordinating its team.
Servo Drives	Precision motor control. For tasks needing exact movement, like robotics or precise cutting.
Whole Plant Control	Big boy stuff – think beverage factories, water plants. Managing entire complex operations.

🤖 Bottom Line (Real Talk)

- **Small PLCs**: Like using a calculator — great for simple math, not TikTok edits.
- **Medium PLCs**: Your daily driver — handles most tasks without crying.
- **Large PLCs**: Supercharged server-tier boss mode — can run a whole factory and still chill.

💡 PLCs Usability Benefits + Practical Benefits Table

Let's go beyond marketing fluff and break it down: *Inherent Features of PLCs*

- **Solid-State Components (no moving parts = fewer failures)**
- **Programmable Memory (non-volatile, editable logic)**
- **Compact Size (small footprint)**
- **Microprocessor-Based Operation (real-time logic control)**
- **Built-in Timers/Counters**
- **Software-based relays (replace physical hardware)**
- **Modular Architecture (easily swappable modules)**
- **Multiple I/O Interfaces (digital, analog, remote)**
- **Remote I/O Stations (connected via coax/twisted pair)**
- **Real-Time Monitoring & Diagnostics**
- **User-editable memory and logic**

AREA	BENEFIT DESCRIPTION
💪 Reliability	Fewer mechanical parts = fewer breakdowns. Solid-state design means less wear and tear.
🔁 Flexibility	Easy to change logic – no need to rewire hardware. Just update the software code.
🧱 Space-Saving	Compact units reduce control panel size. Frees up valuable floor space in a factory.
🖊 Easy Maintenance	Diagnostic LEDs and modular parts = fast repairs. Quickly identify and swap out faulty components.
💬 Communication	RS-232, Ethernet, etc. – talk to SCADA, sensors, drives. Integrates seamlessly with other systems.
🛠 Simple Troubleshooting	View live logic with a laptop, no guessing. See exactly what's happening in real-time.
⚙ System Integration	Can control motors, valves, sensors – no need for extra boxes. A single PLC can manage diverse equipment.
📦 Expandability	Add I/O modules, memory, communication cards as needed. Scales up easily with growing demands.
📉 Cost Savings	Reduce wiring, panel size, downtime, and custom hardware. Saves money in installation and operation.
🧠 Smart Programming Tools	Simulate, test, and edit without touching physical wires. Develop and refine logic safely offline.
🔁 Change Management	Modify settings remotely or via software – no production halt. Make updates without stopping the line.

PLC MEMORY AND DATA REPRESENTATION

🧠 Understanding PLC Memory & Data Representation

When working with PLCs, it's crucial to know how they store and interpret data. At their core, PLCs **don't "see" numbers or words** the way humans do — they see **voltage levels**, **binary digits**, and **memory addresses**.

🧩 1. What Does a PLC Store?

PLCs handle two main categories of data:

Type	Description	Example
Instructions	Pre-programmed tasks the PLC must perform	e.g., `Turn ON motor`, `Check input X`
Process Data	Real-time info collected from sensors, timers, counters, etc.	e.g., `Temperature = 55°C`, `Items counted = 24`

🔍 Think of it like this: instructions = the brain's to-do list, process data = the brain's current awareness of the environment.

⚡ 2. How Is This Data Stored?

Unlike us, who use **decimal numbers (0–9)**, a PLC stores data using **binary (0 and 1)** — not just for fun, but because it physically uses **electrical signals** to represent these states.

Binary Digit	Voltage Level	Meaning
`0`	`0V`	Off / False / No
`1`	`5V` (or 24V etc.)	On / True / Yes

💡Inside the PLC's memory chips (RAM or ROM), data is stored as millions of tiny switches that are either ON (1) or OFF (0), represented by voltage levels.

🧠 3. Human vs. PLC Perspective

Let’s take a real-life example:
You want the PLC to keep track of how many **cans** were counted.

🧑 Human:

You think in decimals:
"24 Cans" → makes sense to you.

🤖 PLC:

It can’t store “24” as-is. It converts it:

- Decimal 24 → Binary 00011000
- Each binary bit gets mapped to voltage:
  0 → 0V, 1 → 5V

Binary Bits	Voltage Signals
`00011000`	`0V 0V 0V 5V 5V 0V 0V 0V`

📌 *This is how the PLC internally “thinks” about the number 24.*

🧰 4. Why This Matters in Practice

- You’ll often **see, debug, and manipulate** data in its **binary** or **hex** form inside the PLC. Understanding this voltage-level mapping helps when working with:
  - **Input/Output signals**
  - **Timers/Counters**
  - **Word/Bit-level instructions**
  - **Data type conversions (BCD <-> Binary, etc.)**

🔄 *Binary ↔ Decimal conversion is your daily bread in PLC land. Don't skip it.*

📷 Visual Breakdown (Explained)

Imagine this image setup:

- **Left side**: A person is thinking "24 cans"
- **Right side**: A PLC's thought bubble showing: **0V 0V 0V 5V 5V 0V 0V 0V**

This diagram is not just cute — it teaches you the mental translation process:

Human-friendly data → PLC-friendly voltage signals → Actual binary bits in memory

✅ Summary Cheat Code

Concept	Human Side	PLC Side
Number	24	`00011000`
System	Decimal	Binary
Storage format	None (abstract)	Electrical signal (0V/5V)
Underlying representation	Meaning	Voltage = Bit = Data

The last one is a bit tricky…

For us, numbers, words, and concepts carry inherent meaning. "24" immediately means a quantity, an amount, something we can understand and relate to. Our brains are wired for abstract thought and interpretation.

For a PLC, there's no "meaning" in the human sense. It's all about physical electrical states.

What Are Numbering Systems in PLCs?

Data in PLCs = Either Instructions or Process Info.
But here's the catch: *PLCs don't "think" in decimal.* Humans type numbers in base-10 (like 25 or 90) — but PLCs store them in **binary** using 1s and 0s.

So… **translation is needed** between human-readable numbers and machine-level binary.

Binary Number System (Base-2)

- **Only 2 digits:** 0 and 1.
- These are called **bits.**
- Groupings:
    - 1 bit = 1 binary digit (duh)
    - 4 bits = **Nibble**
    - 8 bits = **Byte**
    - 16 bits = **Word**
    - 32 bits = **Double Word**

Each bit in a binary number has a weight — just like place values in decimal — but in **powers of 2**.

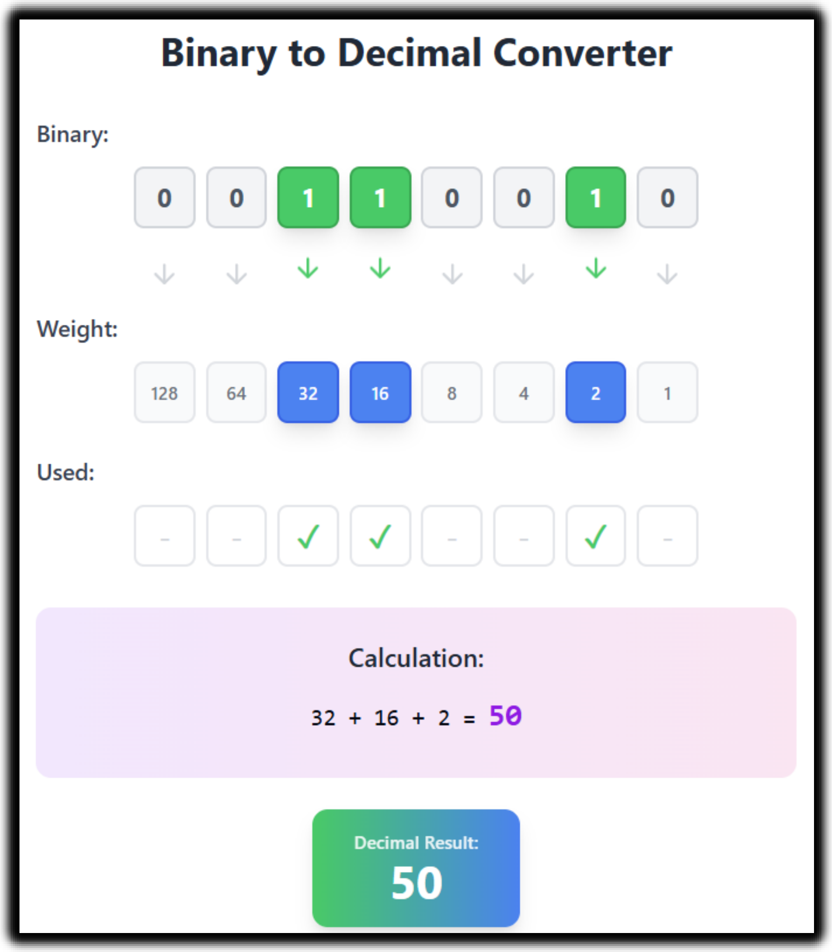
Binary Column	128	64	32	16	8	4	2
Example Bits	0	0	1	1	0	0	1

🔢 Convert Binary to Decimal

Steps:

1. Start from the right.
2. Only look at positions that have a 1.
3. Add their **weight** (the column value).

🧠 Example:

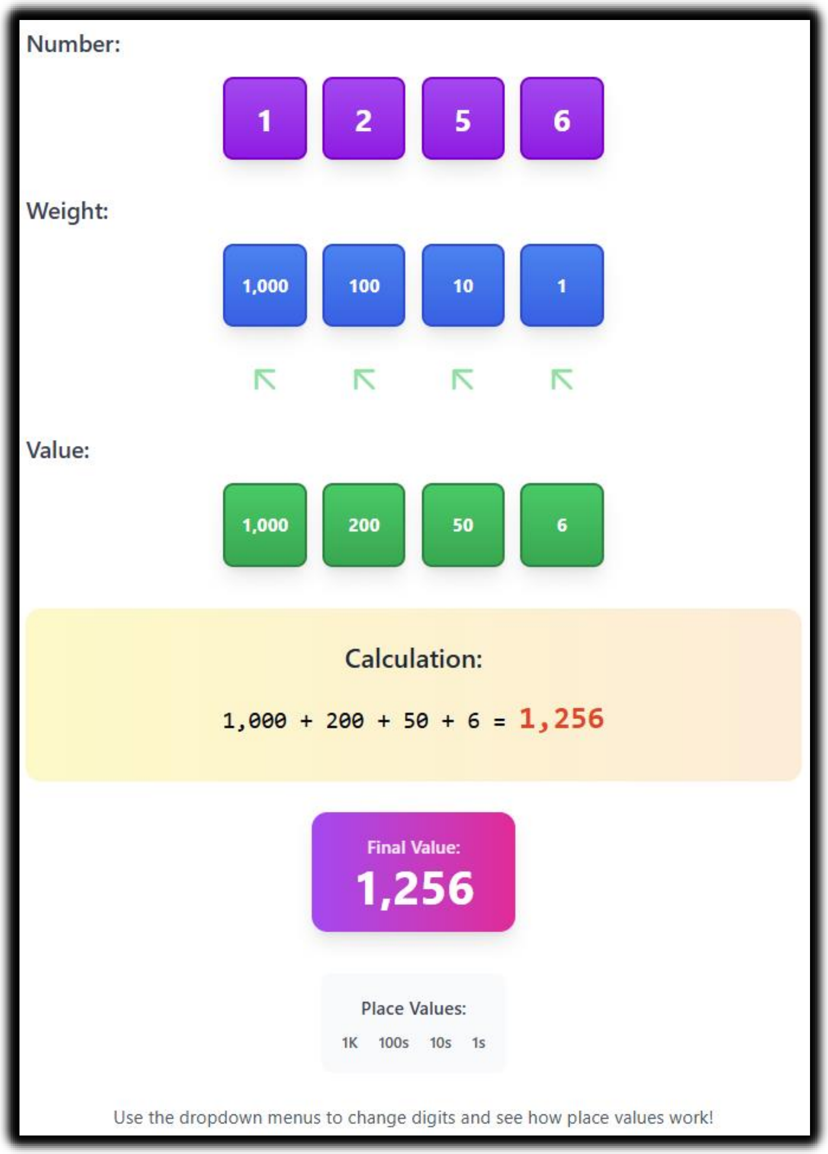


That's how the PLC remembers it — as 00110010. But when you check on your HMI or software, it'll show **50** in decimal.

🔟 Decimal Number System (Base-10)

- Humans use **10 digits**: 0 through 9
- Each digit has a place value weighted in powers of **10**

Example:



The concept of **weighted columns** exists in both decimal and binary.
The difference is that:

Binary → Powers of 2

Decimal → Powers of 10

🧠 Why Does This Matter in PLCs?

Because whenever we:

- Type a value into the PLC
- Store a sensor reading
- Set a timer

...The PLC stores it in binary, but you *see it* in decimal.

Understanding how those 1s and 0s are turned into human-friendly numbers helps you:

- Debug logic better 💻
- Spot errors quickly 🕵
- Know how memory is being used 🧠

PLC Numbering Systems + Data Types Simplified

🚀 BIT, BYTE, WORD — What's What?

Term	Size	Meaning
Bit	1 binary digit (0 or 1)	Smallest unit of data
Nibble	4 bits	Half a byte
Byte	8 bits	Common for ASCII chars & simple values
Word	Typically 16 or 32 bits	Depends on the PLC hardware

✅ In PLCs:

- 16-bit word = max value **65535** (if unsigned)
- 32-bit word = up to **~4.2 billion**

Word length determines how big a number your PLC can store or calculate.

🧮 Numbering Systems Recap

Let's blitz through these like a pro:

1. Binary (Base-2)

- Digits: 0 and 1
- Every bit = power of 2
- Used **internally** in PLC memory & logic

🧠 Example:

```
Binary:  00000000 00011000
Decimal:        24
```

2. Octal (Base-8)

- Digits: 0–7
- No 8 or 9 exists!
- Some old-school PLCs (like Allen-Bradley) used **octal addressing** for I/O: 001, 002, ..., 007, 010 (which is 8 in decimal).

🧠 Rare today but *still shows up* in legacy systems.

3. Decimal (Base-10)

- Digits: 0–9
- What humans use by default.
- PLC HMIs show **decimal**, but store as **binary** internally.

4. Hexadecimal (Base-16)

- Digits: 0–9 and A–F
  - A = 10
  - B = 11
  - ...
  - F = 15
- Shorter than binary but stores the same info
- Used heavily in:
  - **Timer values**
  - **Memory locations**
  - **Comms settings**
  - **Hex instructions**

🧠 **Example:**

```
Hex:  D3h
= D (13) × 16 + 3 = 208 + 3 = 211
```

5. BCD (Binary-Coded Decimal)

- Each decimal digit is stored separately in binary
- e.g., 45 = **0100 0101** (not 00101101)
- Used in:
  - Thumbwheel switches
  - Old school panels
  - PLCs that interface with operator input modules

BCD is weird but still lingers in industrial gear.

🔄 Conversions You'll Actually Use:

From → To	Example
Binary → Decimal	`00011000` = **24**
Decimal → Binary	`50` = `00110010`
Hex → Decimal	`2A` = 2×16 + 10 = **42**
Decimal → Hex	`173` = `AD`
Binary → Hex	`11010100` = `D4`

You rarely convert octal or BCD manually, but binary/hex/dec happens *daily* in ladder logic debugging and SCADA.

🚨 Max Values Recap

Type	Max Unsigned Decimal Value
8-bit Byte	255
16-bit Word	65,535
32-bit DWord	4,294,967,295
BCD (4 digits)	9999

🎯 What's New Here for PLC-Specific Context?

- **PLC I/O addressing might use Octal or Hex** (e.g., Address 020 in Octal is not 20 in decimal—it's 16!)
- **Timers/Registers** often appear in **Hex** in programming software
- **Memory** segments are word- or byte-addressed, so knowing limits (8-bit/16-bit) = less debugging rage 😤

🔥 HEX & DECIMAL CONVERSIONS

🧠 First, the Big Idea: Place Value Table

Just like decimal (base 10) uses powers of 10, hexadecimal (base 16) uses powers of 16.

Position	Decimal (10^n)	Hexadecimal (16^n)
4th from right	1000 (10^3)	4096 (16^3)
3rd from right	100 (10^2)	256 (16^2)
2nd from right	10 (10^1)	16 (16^1)
1st from right	1 (10^0)	1 (16^0)

Shows the comparison between decimal (10^n) and hexadecimal (16^n) place values.

When we say **"1st from the right"**, **"2nd from the right"** and so on, we are talking about the position of a digit in a number, starting with the rightmost digit as the first position.

Think of it like reading numbers. In the number 1234:

- 4 is the **1st from the right** (the ones place).
- 3 is the **2nd from the right** (the tens place).
- 2 is the **3rd from the right** (the hundreds place).
- 1 is the **4th from the right** (the thousands place).

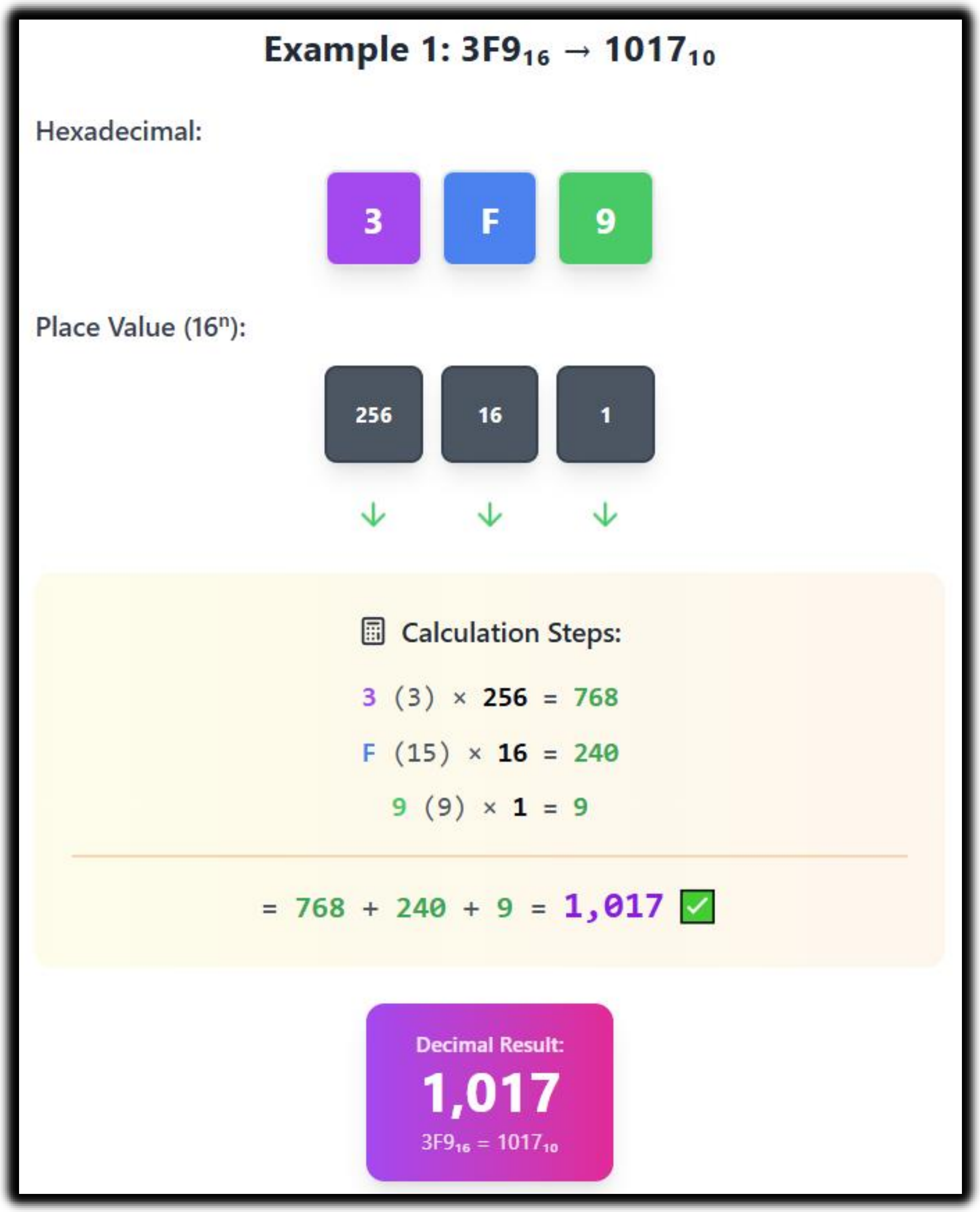
This applies to any number system, whether it's decimal, hexadecimal, or binary – you always count positions starting from the rightmost digit and moving left.

✨ Convert Hex to Decimal – Easy Mode:

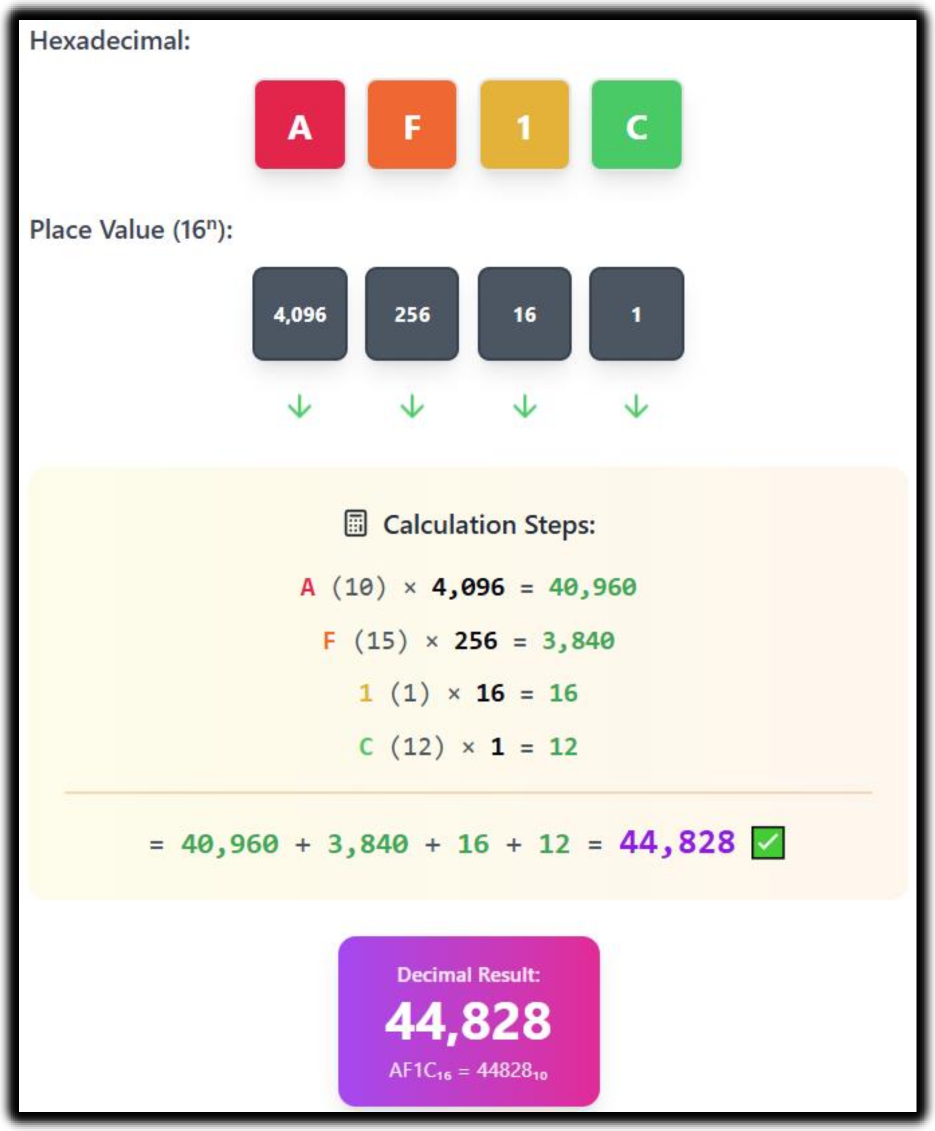
You multiply each digit (starting from the right) by powers of 16, then add them all up.

💡 **Pro tip:** A-F in hex = 10-15 in decimal - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

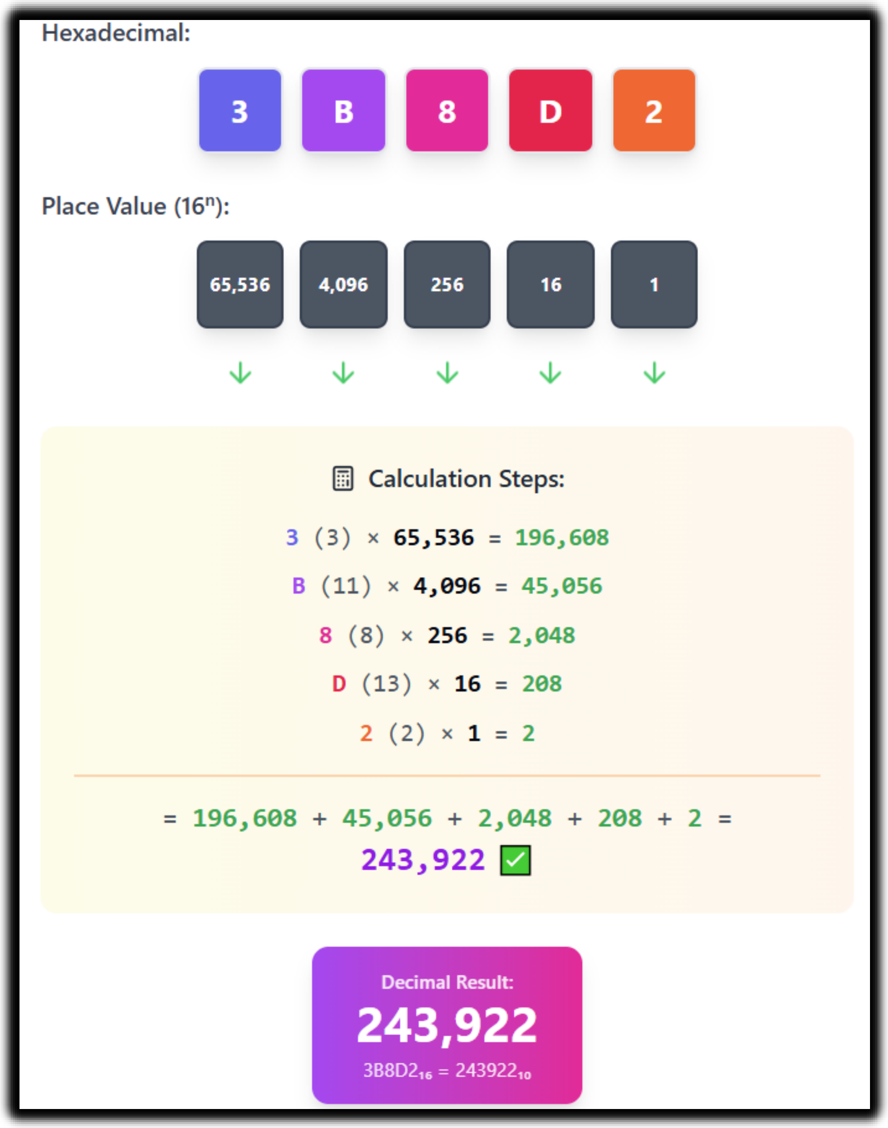
Section 2: Example 1 - $3F9_{16}$ to decimal



Example 2: $AF1C_{16} \rightarrow 44{,}828_{10}$



Example 3: $3B8D2_{16} \rightarrow 243{,}922_{10}$

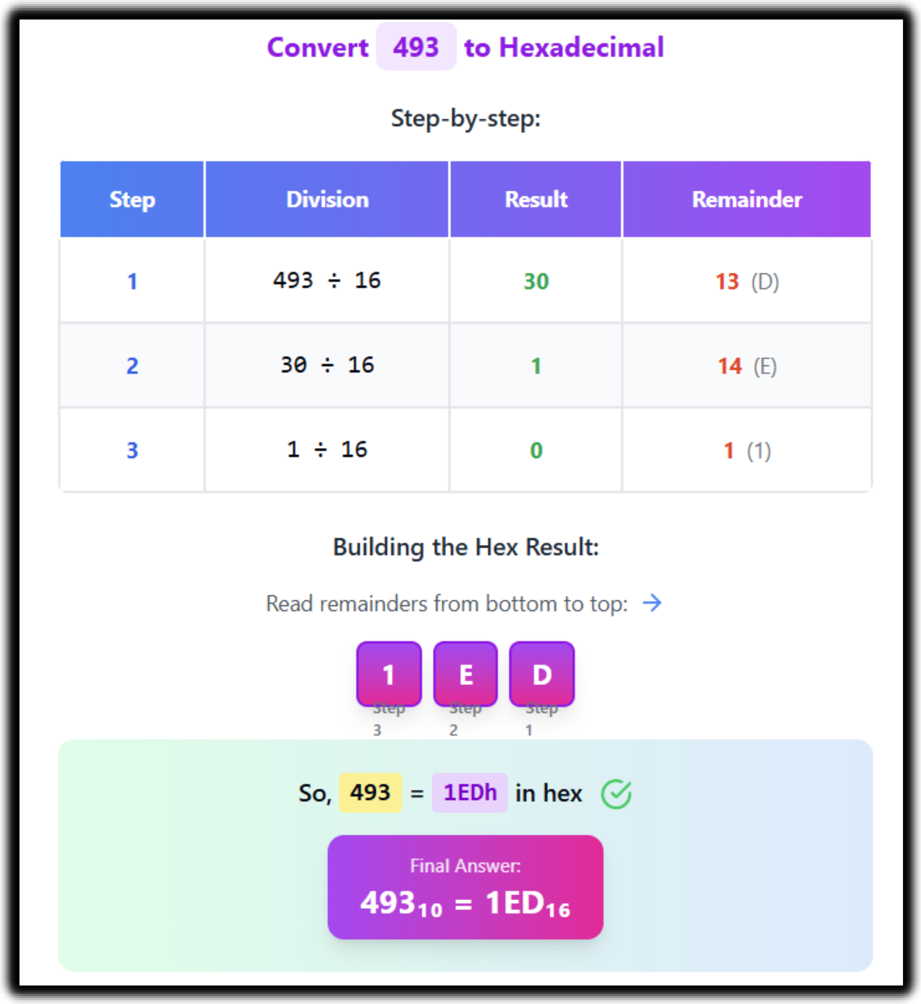


🔄 Decimal to Hex: Division and Remainders

This is like reverse engineering. You divide by 16 repeatedly, collecting **remainders**. These remainders (in reverse order) give you the hex value.

💻 Example: Convert 493 to Hex

Step-by-step:



Convert 493 to Hexadecimal

Step-by-step:

Step	Division	Result	Remainder
1	493 ÷ 16	30	13 (D)
2	30 ÷ 16	1	14 (E)
3	1 ÷ 16	0	1 (1)

Building the Hex Result:

Read remainders from bottom to top: →

1 E D

Step 3 Step 2 Step 1

So, 493 = 1EDh in hex

Final Answer:

$493_{10} = 1ED_{16}$

💻 Example: Convert 57392 to Hex

Convert 57392 to Hexadecimal

Step-by-step:

Step	Division	Result	Remainder
1	57392 ÷ 16	3587	0 (0)
2	3587 ÷ 16	224	3 (3)
3	224 ÷ 16	14	0 (0)
4	14 ÷ 16	0	14 (E)

Building the Hex Result:

Read remainders from bottom to top: →

E	0	3	0
Step 4	Step 3	Step 2	Step 1

So, 57392 = `E030h` in hex ✓

Final Answer:

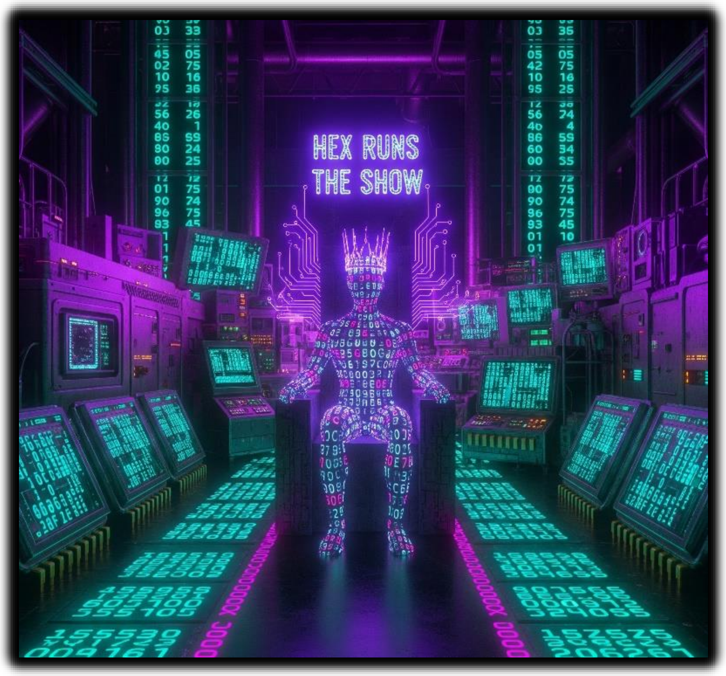
$57392_{10} = E030_{16}$

How the Division Method Works

1. **Divide** the decimal number by 16
2. **Record** the quotient and remainder
3. **Convert** remainder to hex digit (10=A, 11=B, 12=C, 13=D, 14=E, 15=F)
4. **Repeat** with the quotient until it becomes 0
5. **Read** the hex digits from bottom to top

🧠 Final Boss Trick: Why Hex is 🔥

- **Compact**: 1 hex digit = 4 binary bits. Clean.
- **Readable**: Easier to debug than long binary strings.
- **Popular**: Memory addresses, opcodes, machine-level data—hex runs the show.



🧠 LOGIC CONCEPTS - SUBTOPIC: BOOLEAN ALGEBRA

📜 Quick History Drop

- **Year:** 1849
- **Inventor:** George Boole (England)
- **Why it matters:**
  He created a mathematical way to describe logic—like "If A is true, then B must be false."
  Every digital circuit, from your calculator to a PLC to an Intel CPU, is built on his rules.
  This isn't just dusty philosophy—it *runs the world.*

🔌 The Core of Boolean Algebra

- It's a way to write **logic expressions** (decisions, conditions) that can only be **True or False**.
- In digital electronics:
  - ➢ **True = 1**
  - ➢ **False = 0**
- Called a **two-valued (binary) system.**

📐 Why Use Boolean Algebra in PLCs?

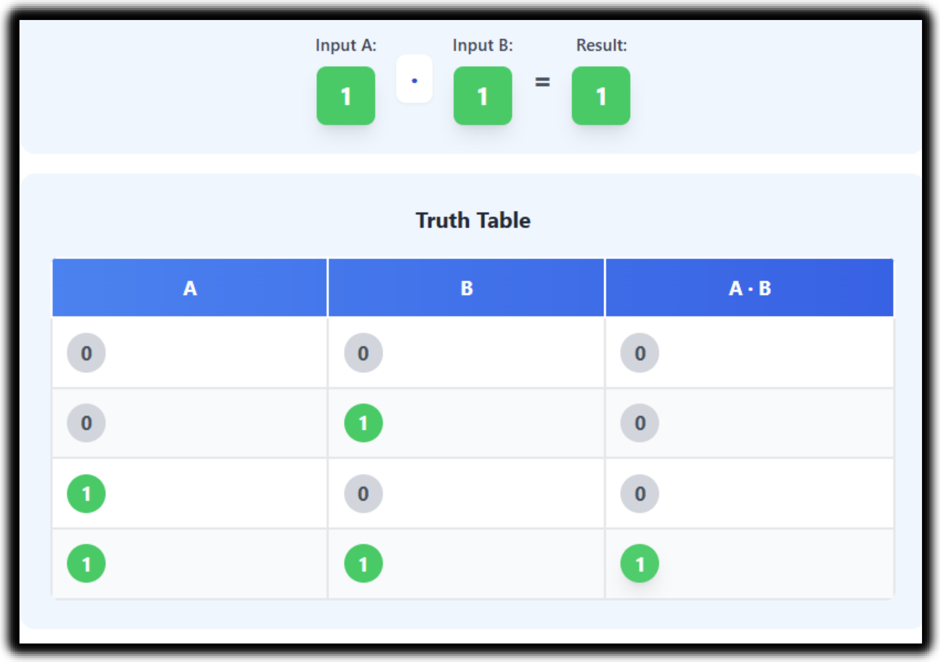
Because ladder logic *is literally* Boolean logic in disguise:

- A contact closed? = 1
- A coil energized? = 1
- Two conditions in series? = AND logic
- Two in parallel? = OR logic

✅ Boolean Operators (The Building Blocks)

Logic	Symbol	Meaning
AND	• or AB	Both A and B must be 1 *Returns 1 only when both inputs are 1*
OR	+ or A + B	Either A or B (or both) is 1 *Returns 1 when at least one input is 1*
NOT	‾ or !A	Opposite of the value (1 → 0, 0 → 1) *Inverts the input value*

AND Gate



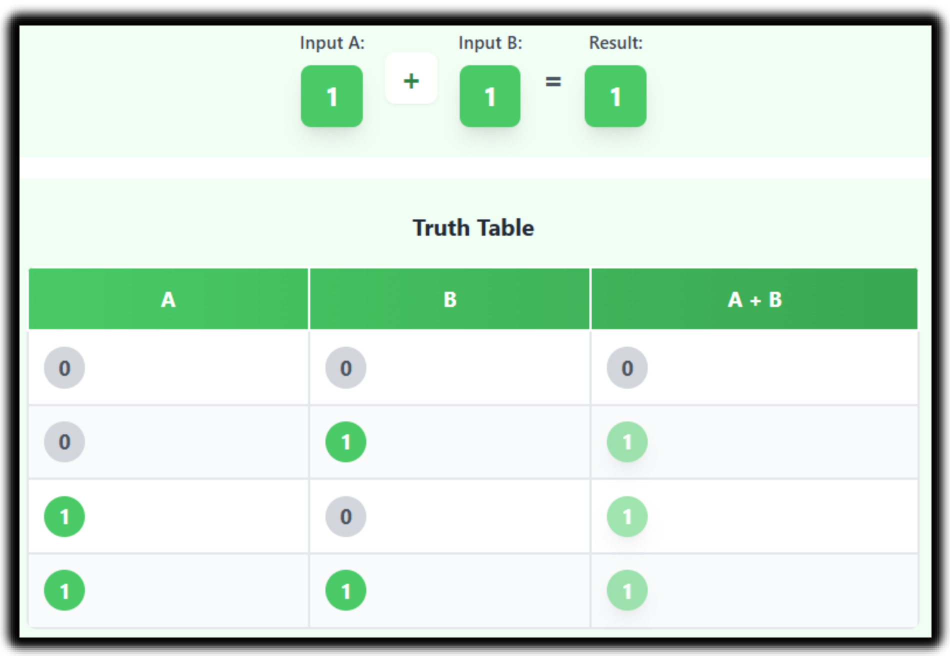
Input A: 1 · Input B: 1 = Result: 1

Truth Table

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

Both A and B must be 1.

OR Gate



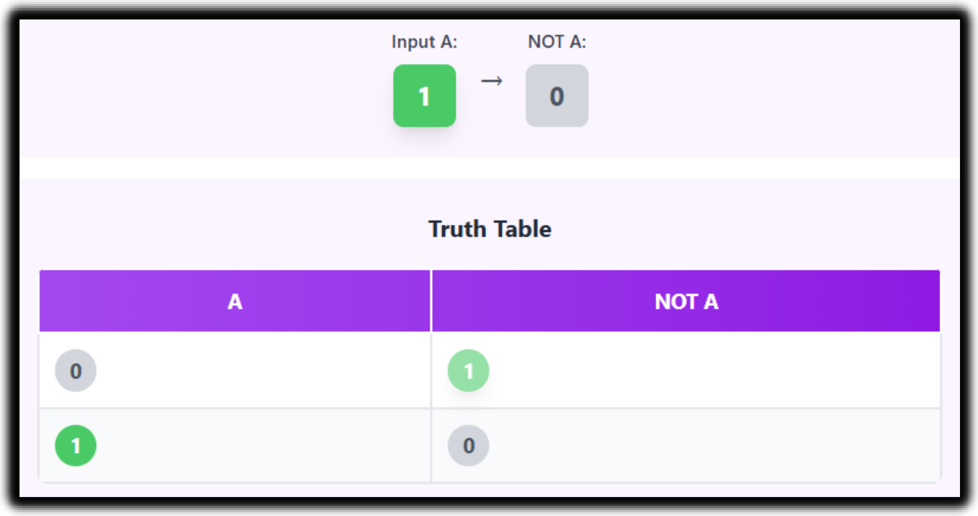
Input A: 1 + Input B: 1 = Result: 1

Truth Table

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Either A or B must be 1.

NOT Gate



Opposite of the value. Flips it.