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:

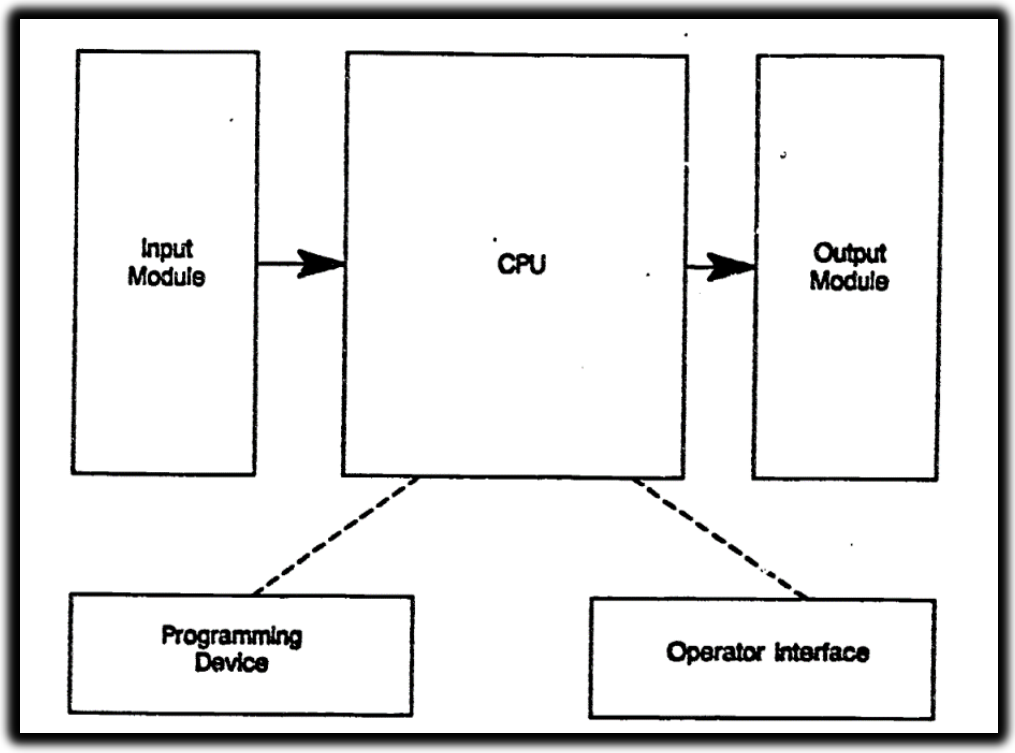
- ✅ **Machine Control:**
  They're the brain behind automated machines on factory floors — controlling motors, valves, and sensors with split-second precision.
- ✅ **Car Wash Systems:**
  Every step in an automatic car wash — soap, brushes, rinse, dryers — is sequenced by a PLC so your car gets clean 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).

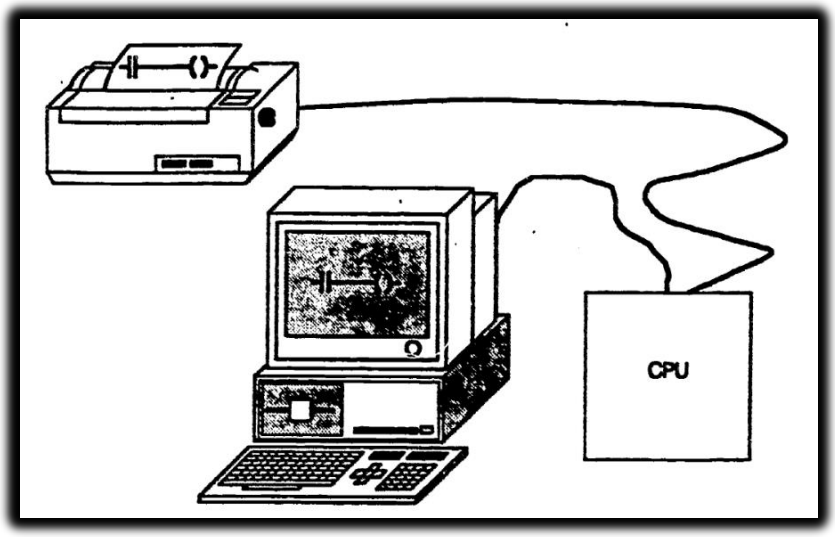
✨ How it all flows (based on the diagram):

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.

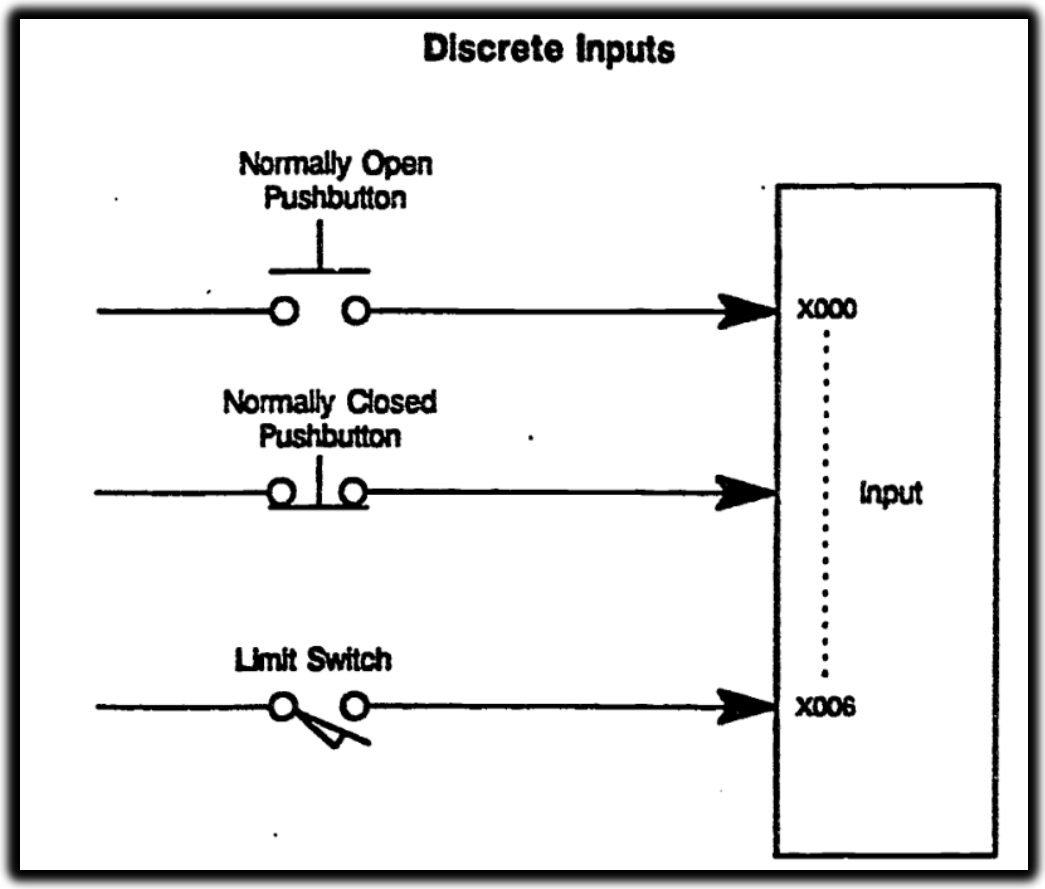
📜 Hardcopy

- 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:

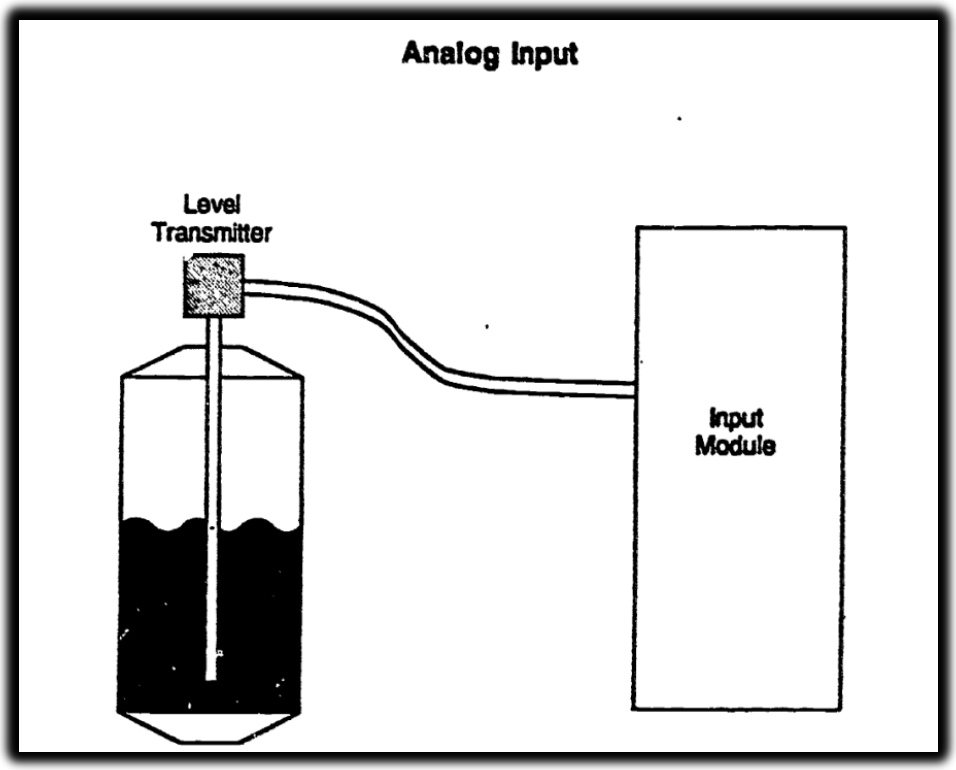
Device	Type	Connected to
Pushbutton	Normally Open	X000
Pushbutton	Normally Closed	X001 (assumed)
Limit Switch	Normally Open/Closed	X006

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.

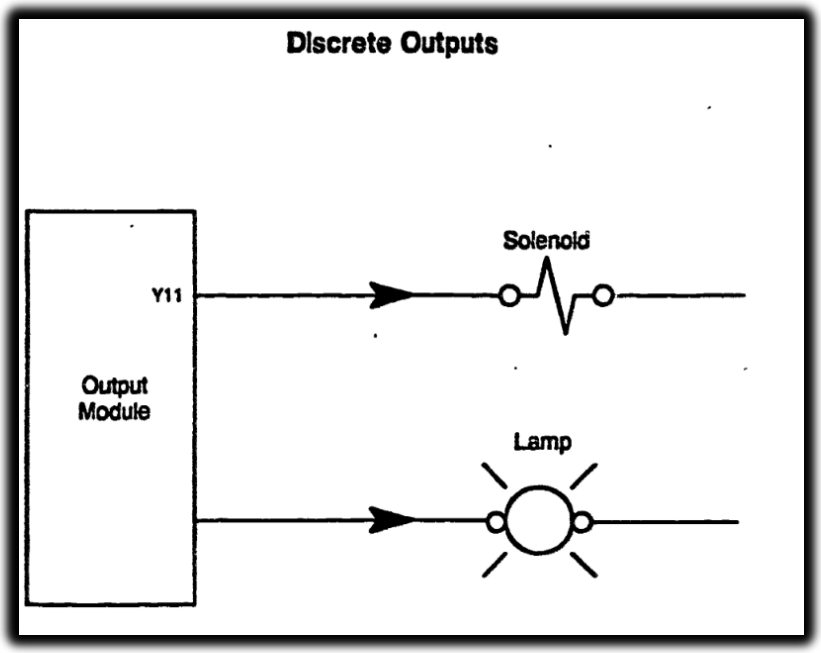
ANALOG INPUT



📌 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:

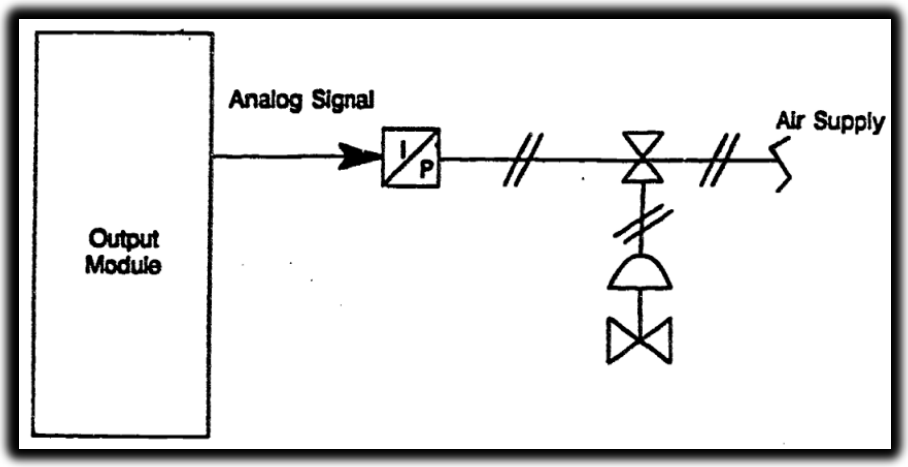
If **discrete inputs are the PLC's ears**,
then **discrete outputs are its hands.**
The CPU listens to what's happening → runs the logic → and then uses outputs to take action.

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**.

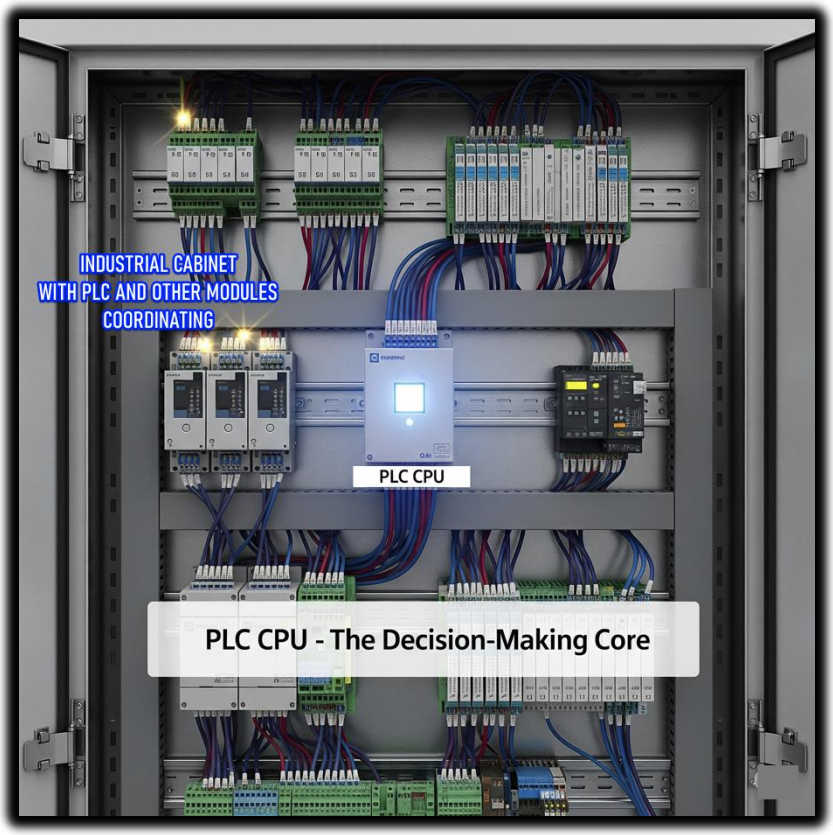
🧠 Extra Details to Keep in Mind:

- 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

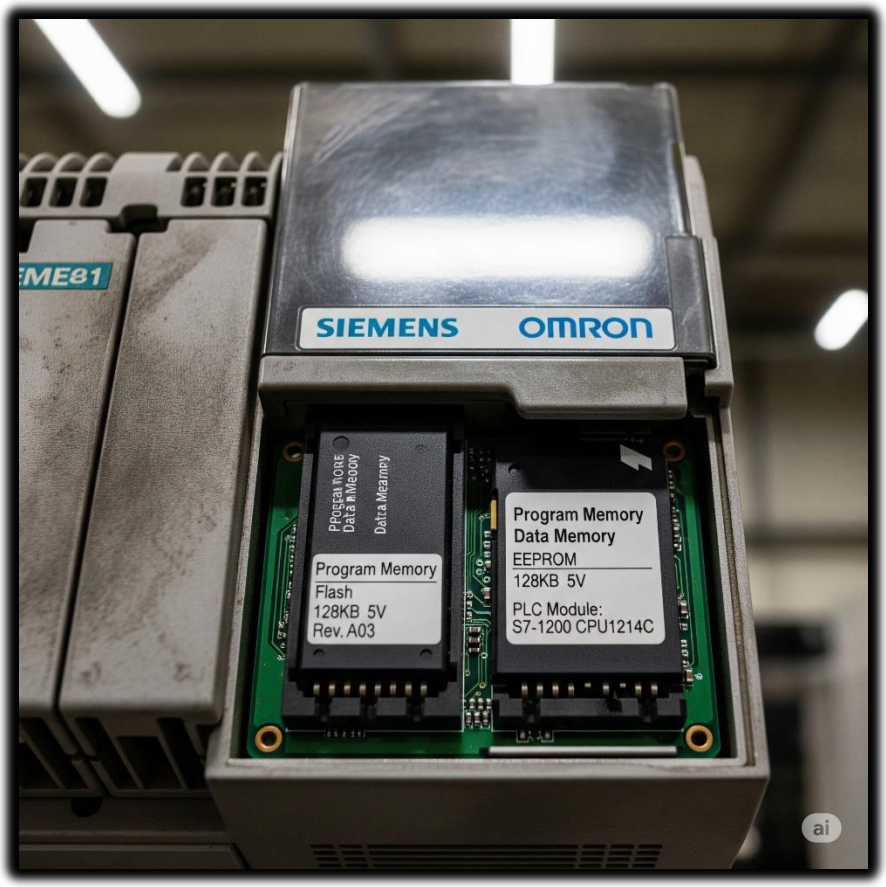
This is where your program and working data live.

🧠 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?

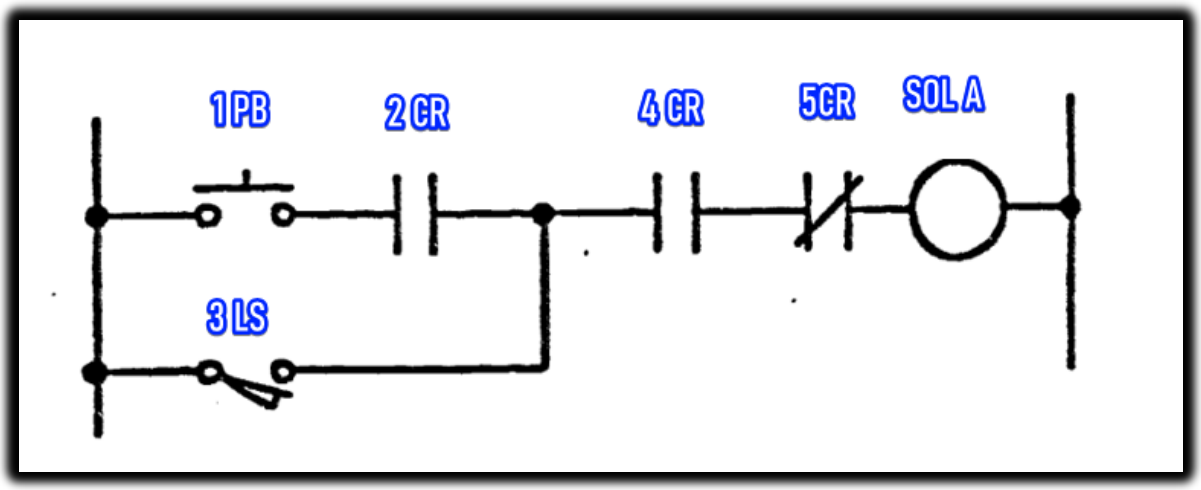
You're **spot on** with this analogy:

“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:

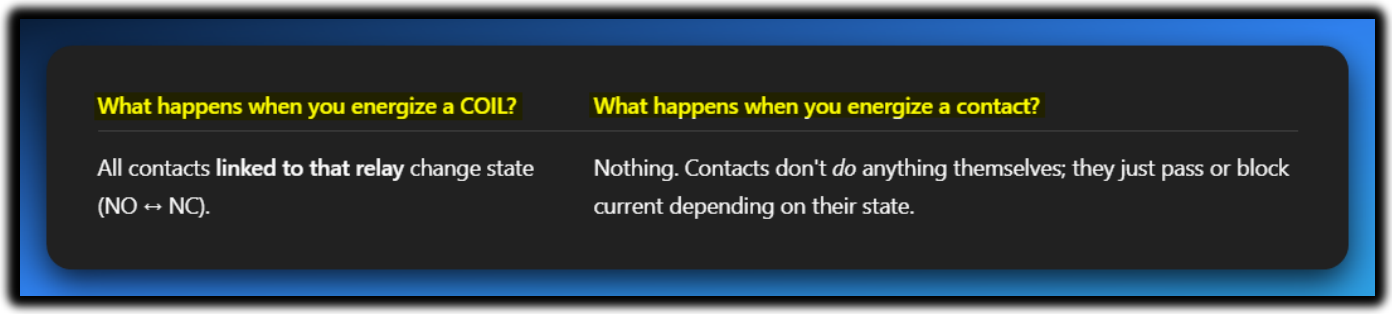
- 4CR coil (controlled by some condition)
- 5CR contact (let's say it's a Normally Closed [NC] contact)

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:

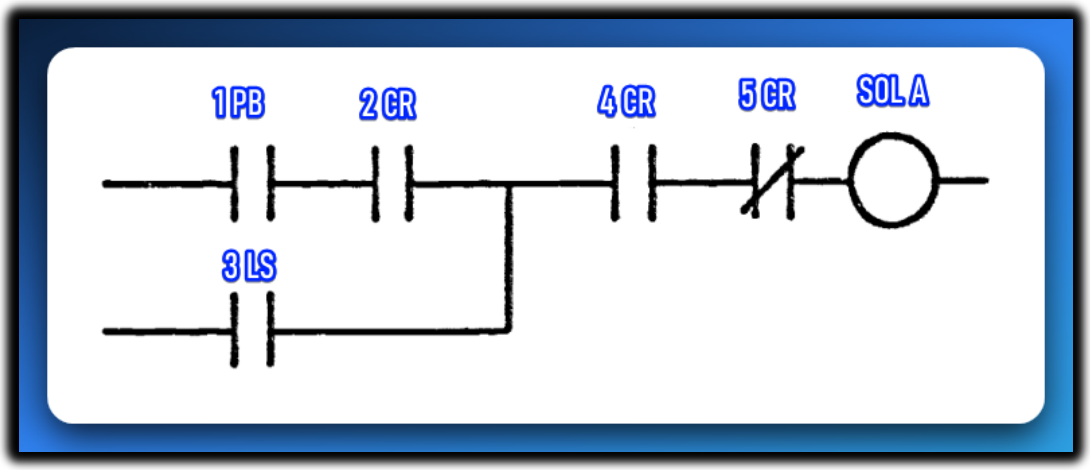
"Doesn't current already pass through 5CR so it becomes open contact?"

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?

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

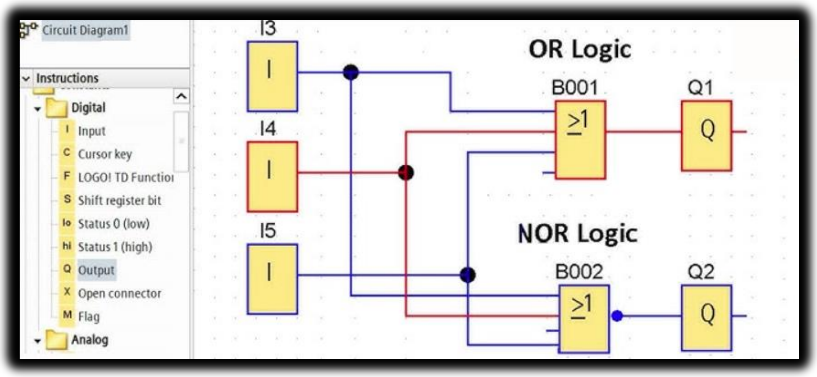
CODE OR MNEMONIC LANGUAGE?

```
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.
```

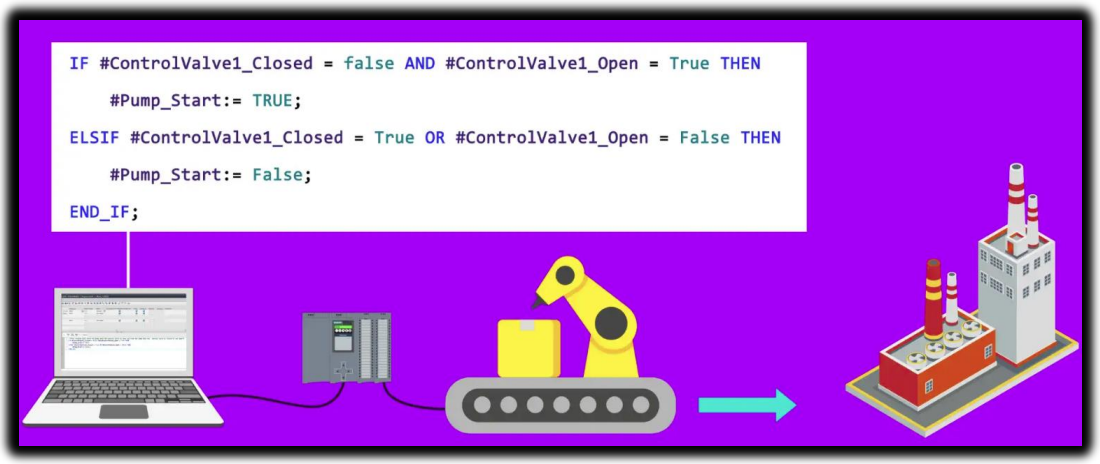
🔧 PLC Programming Languages — Comparison Table

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.



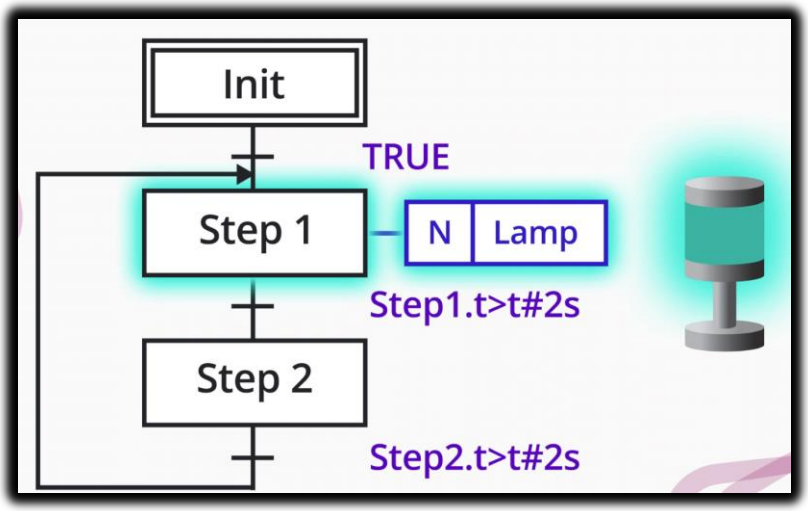
Structured Text (ST) is typically used in advanced control systems. It resembles the Pascal programming language and is best suited for applications involving complex logic or mathematical operations.



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)

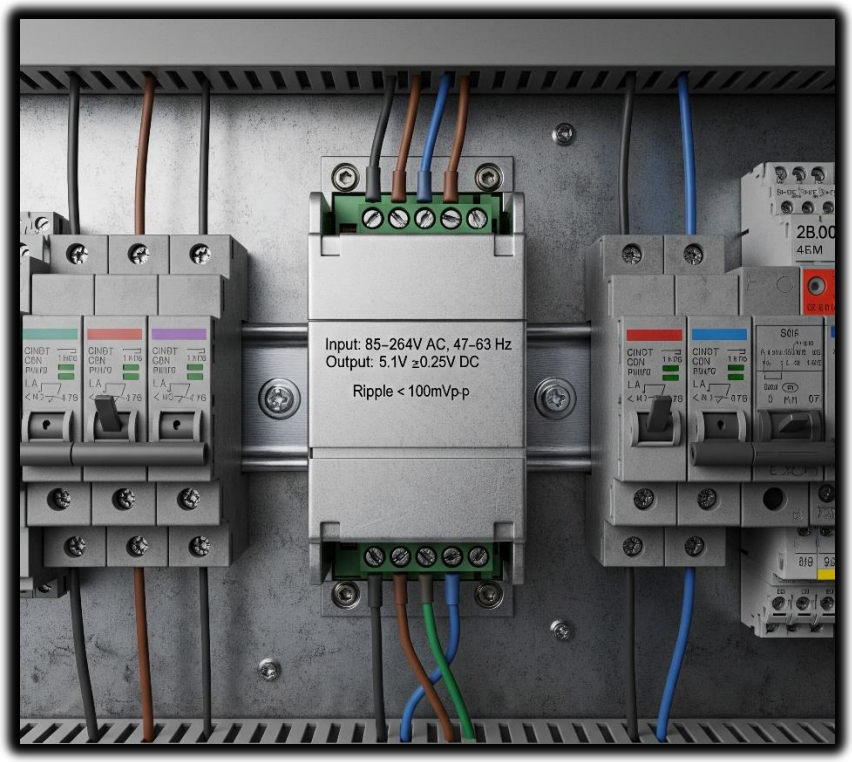
PLCs are purpose-built for the brutal realities of industrial environments. PCs, while powerful, just can't hack it when things get tough…💪

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.**

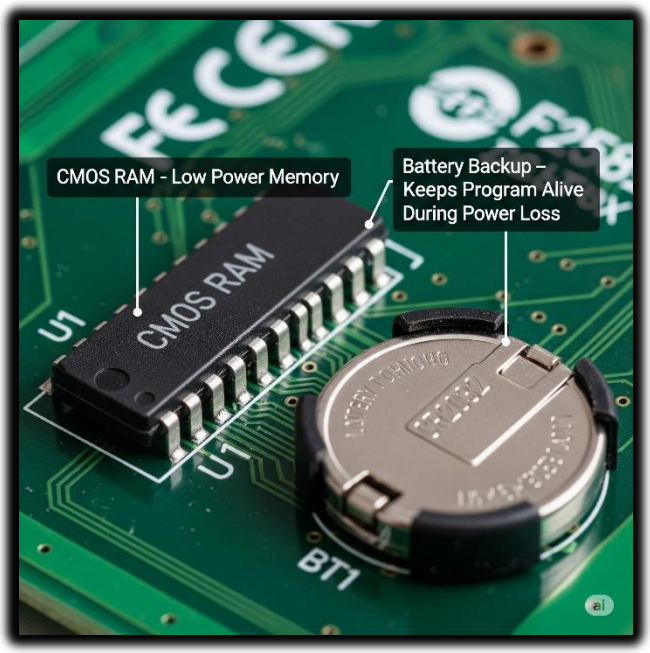
CATEGORY	I/O POINTS	MEMORY (RAM)	USE CASE	FEATURES
Small	Up to 128 I/O	~256 bytes to 2 KB	Basic control tasks. Ideal for simple, standalone machines or small processes, like a single conveyor or a packaging unit.	Only discrete (on/off) control. Think simple switches and lights.
Medium	128 – 2048 I/O	~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

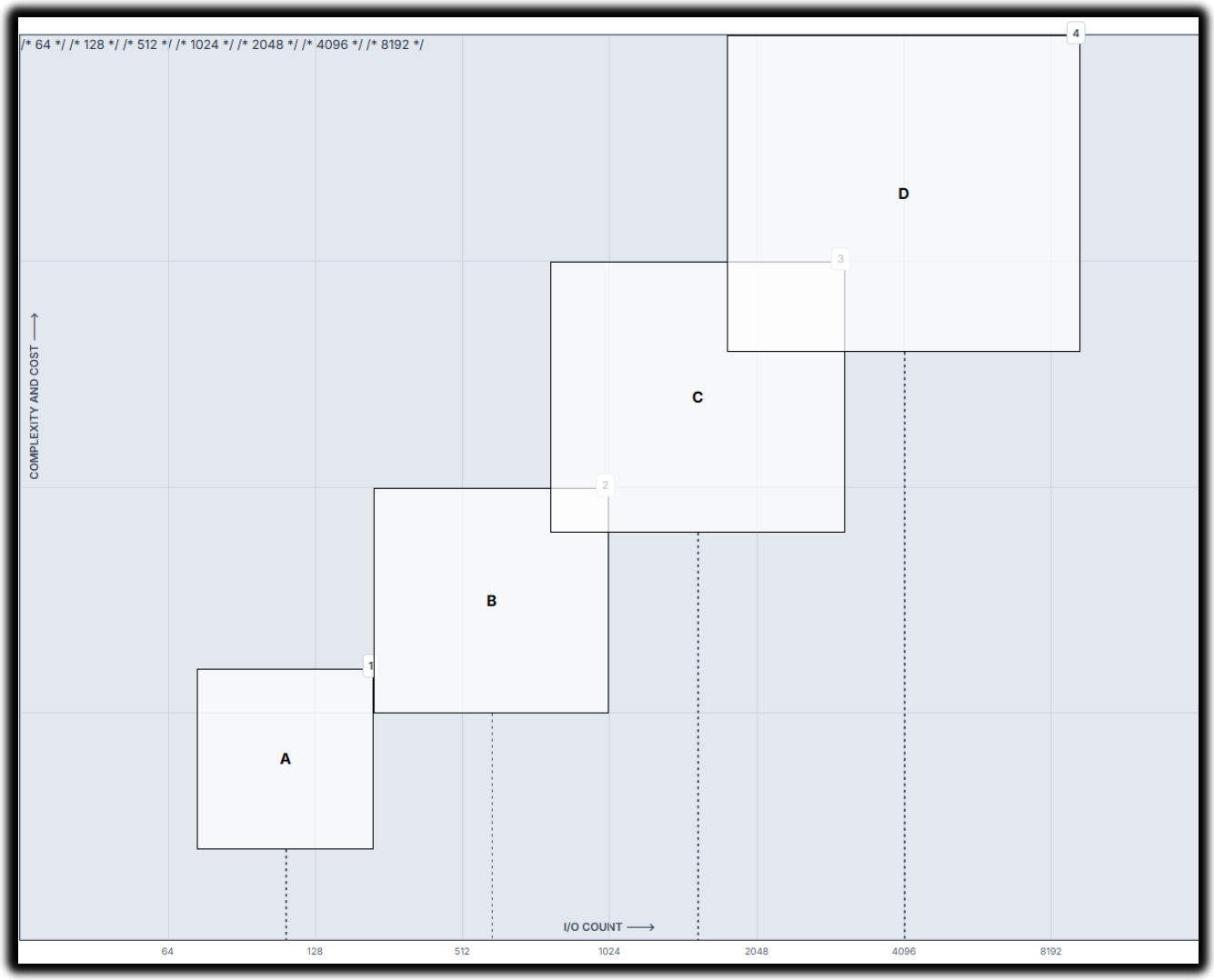
PLCs rely on specific memory types to keep their programs running, even when the power flickers. Here's a quick look at the key tech.

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

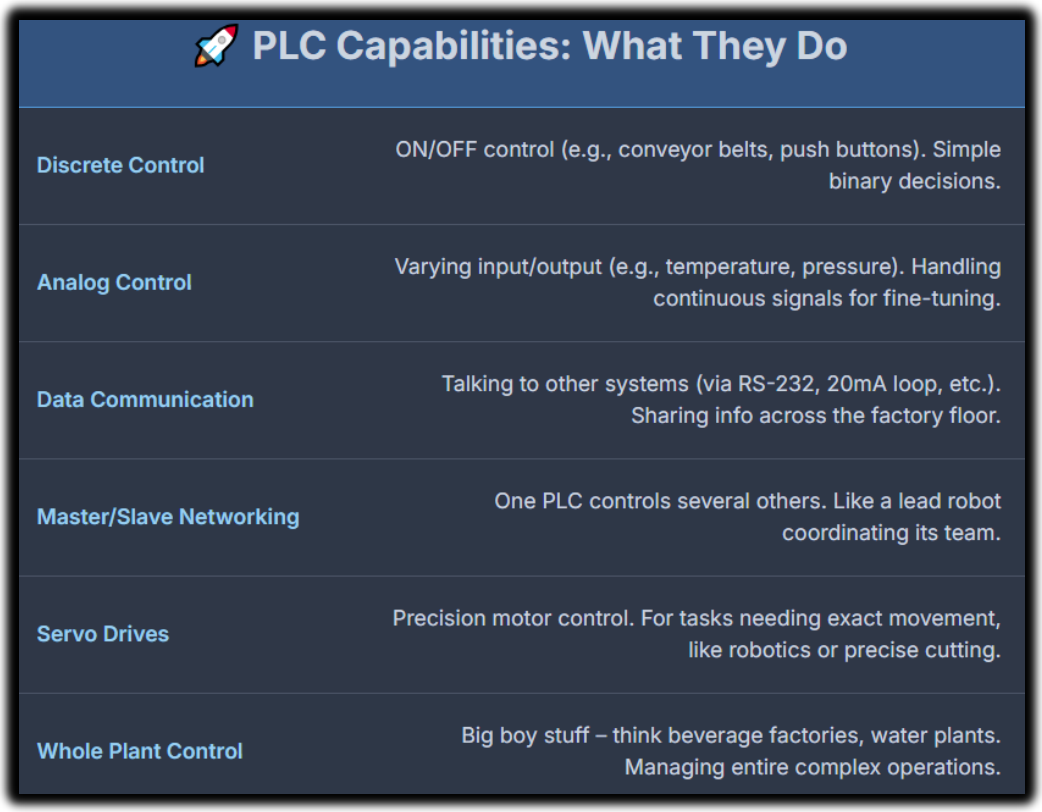


Quick Notes on PLC Sizing:

- **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



🤖 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.

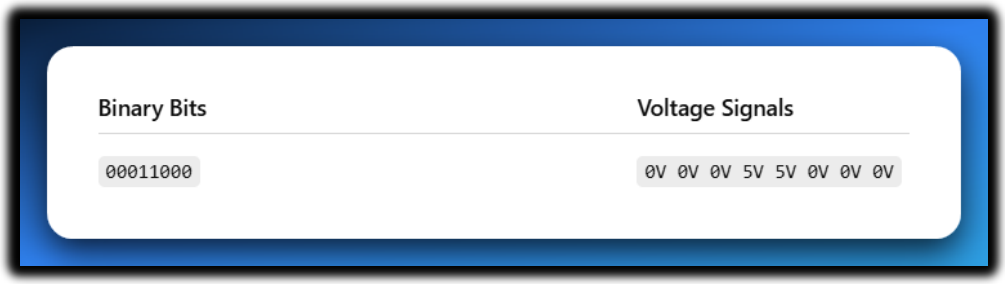
🧍 Human:

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



📌 *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**.

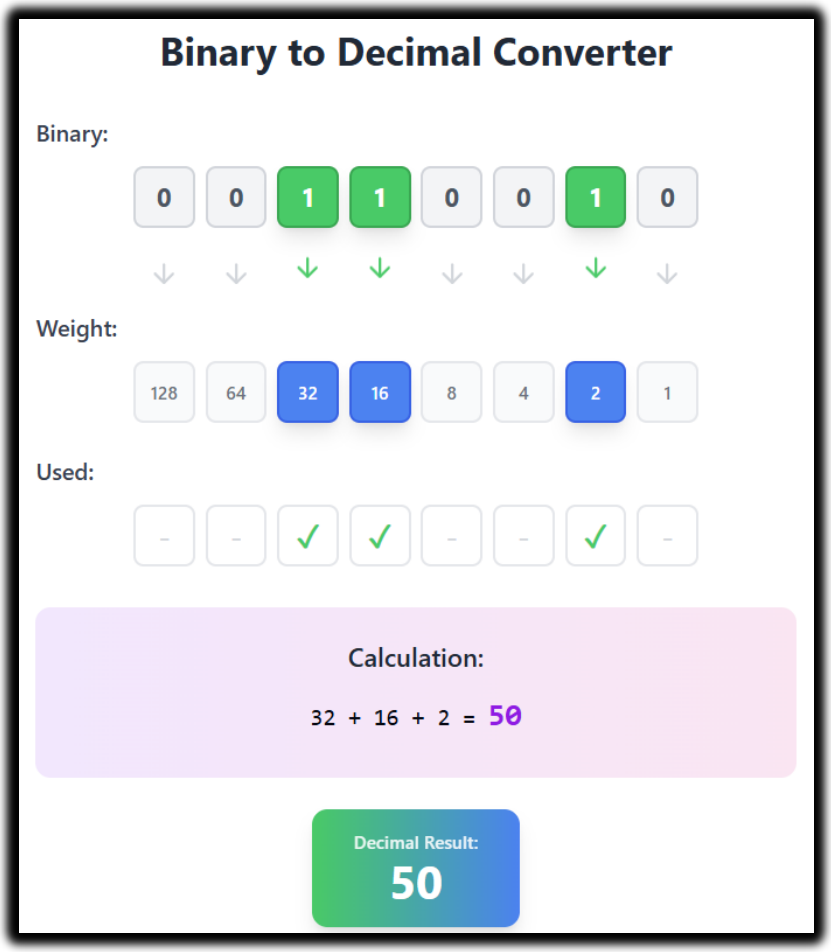
Binary Column	128	64	32	16	8	4	2
Example Bits	0	0	1	1	0	0	1

🔢 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:

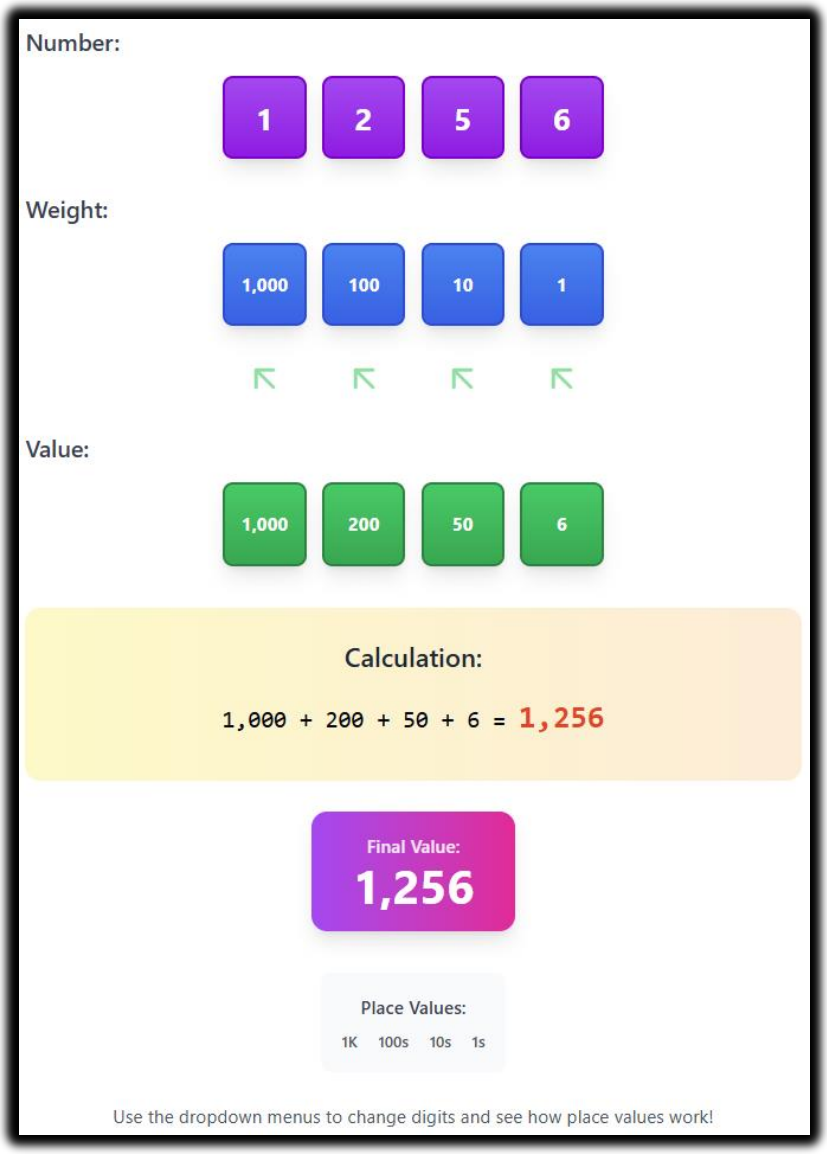


That's how the PLC remembers it — as 00110010. But when you check on your HMI or software, it'll show **50** in decimal.

10 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:**

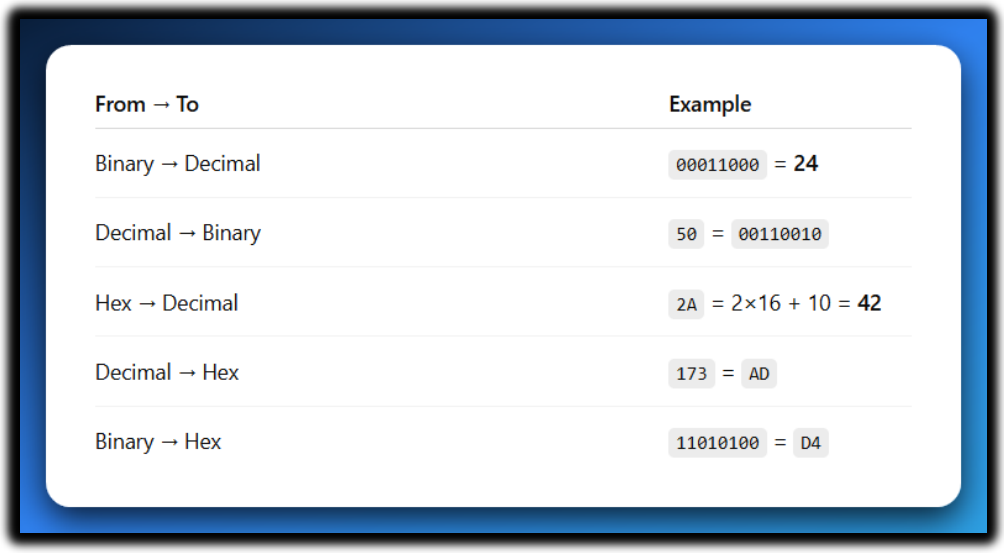
```
Hex:  D3h
= D (13) × 16 + 3 = 208 + 3 = 211
```

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.

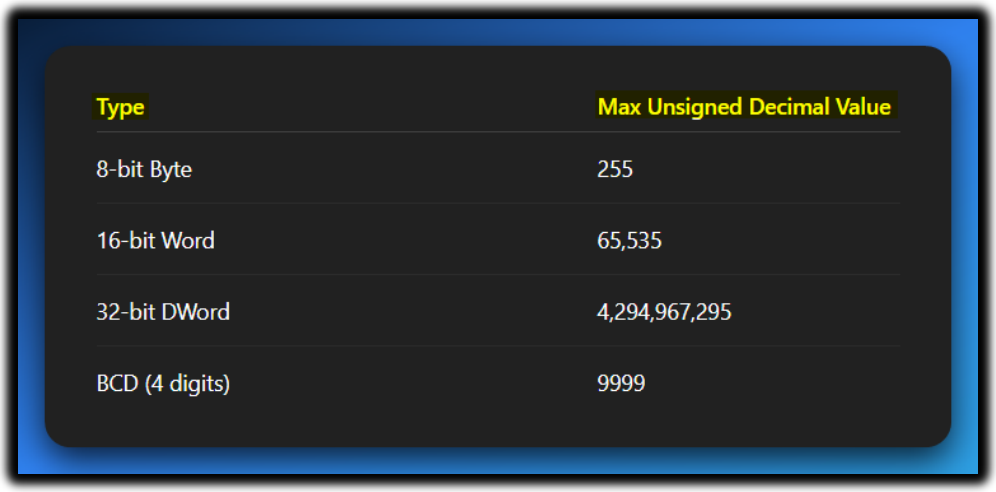
🔁 Conversions You'll Actually Use:



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).

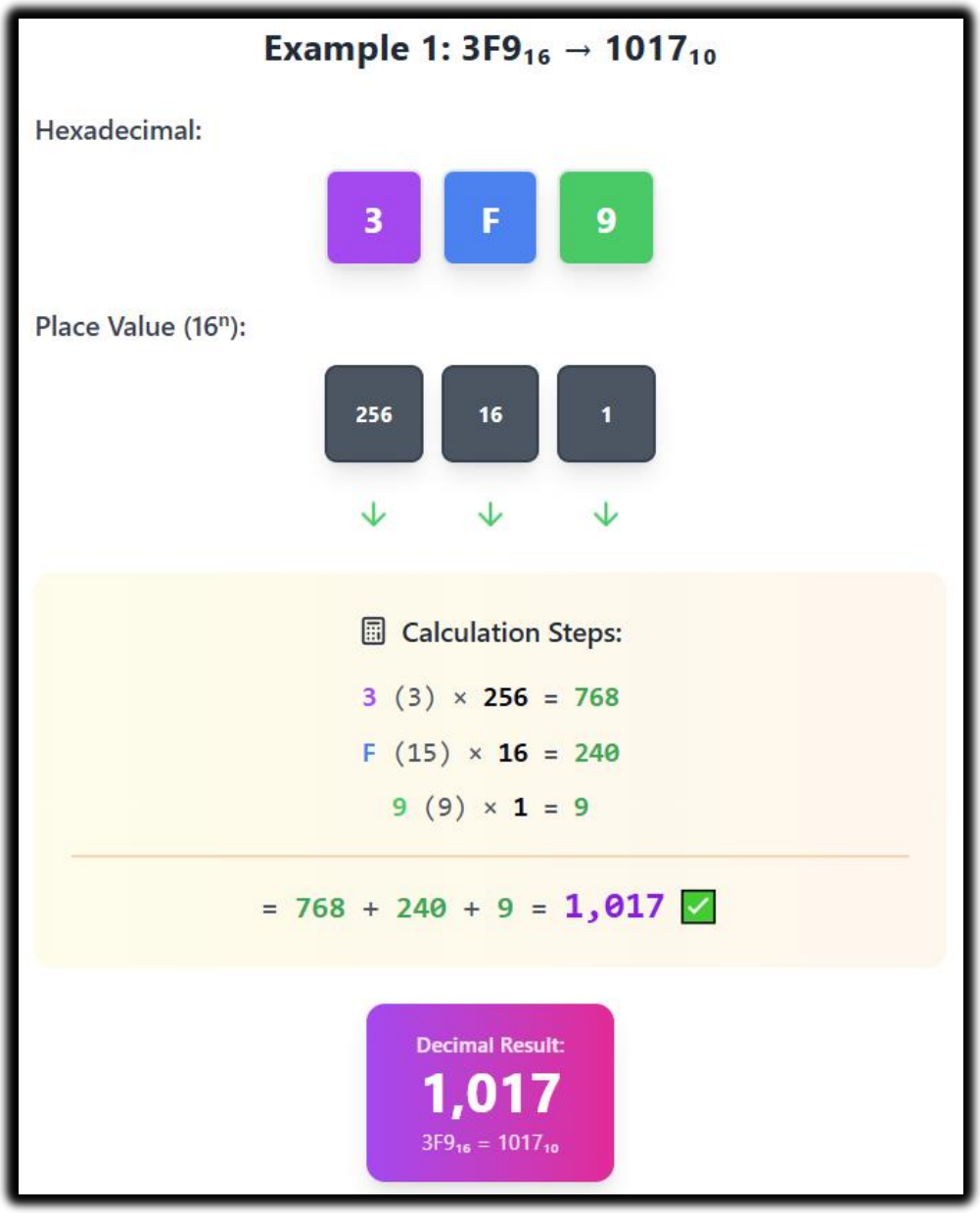
This applies to any number system, whether it's decimal, hexadecimal, or binary – you always count positions starting from the rightmost digit and moving left.

✨ Convert Hex to Decimal – Easy Mode:

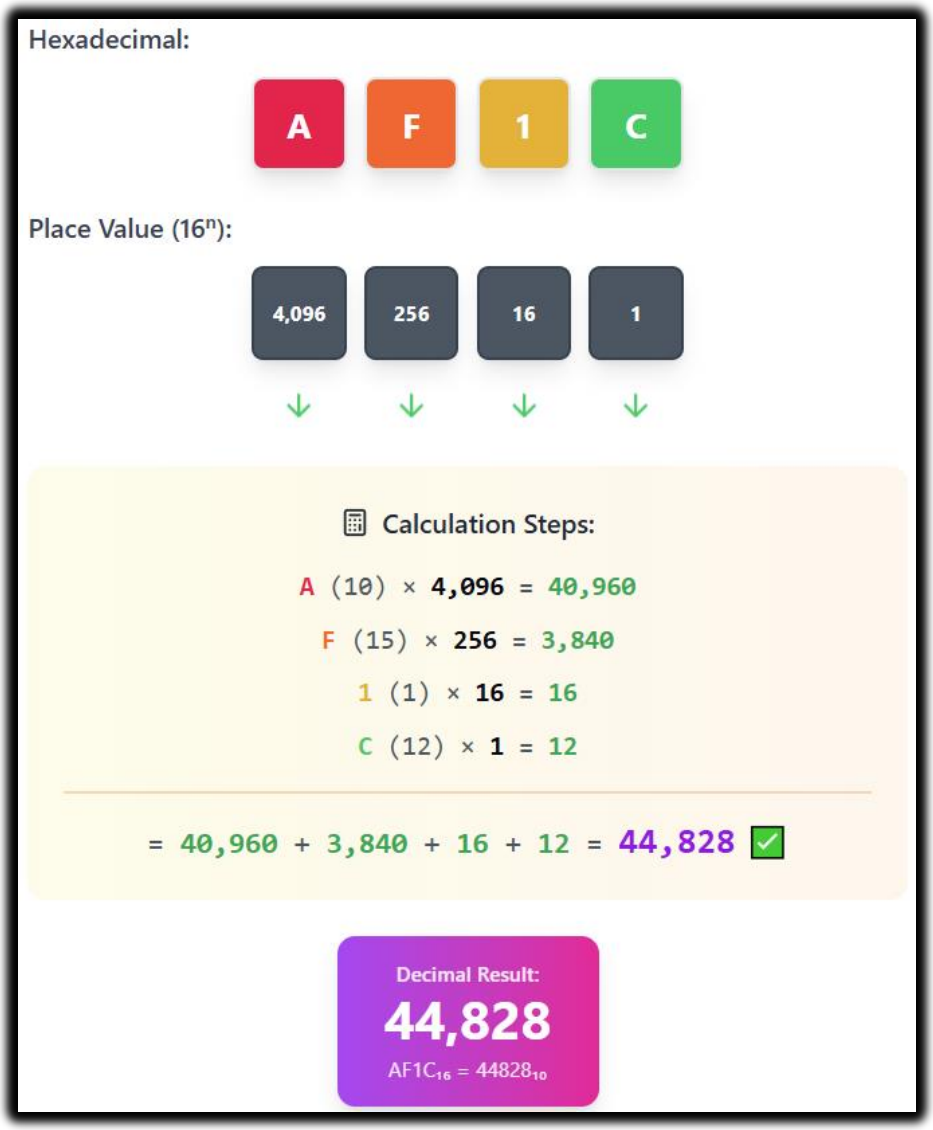
You multiply each digit (starting from the right) by powers of 16, then add them all up.

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

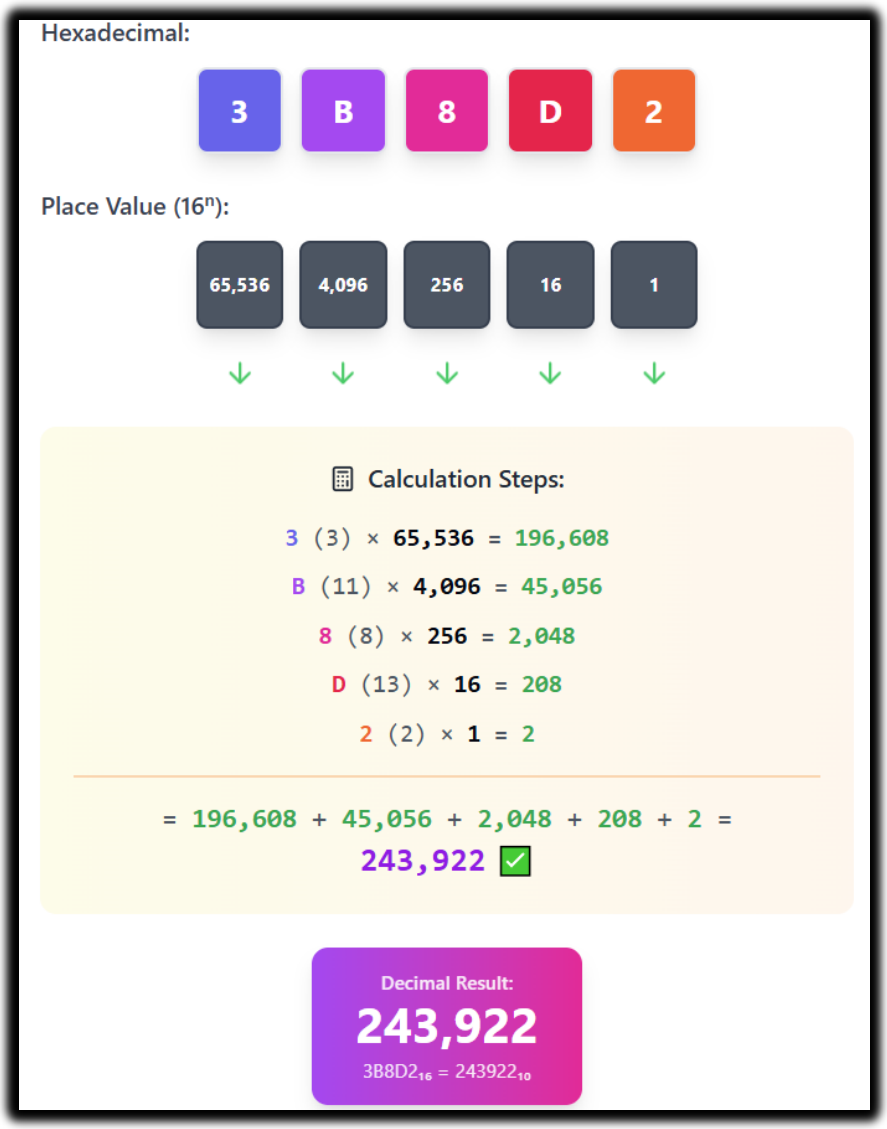
Section 2: Example 1 - $3F9_{16}$ to decimal



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



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

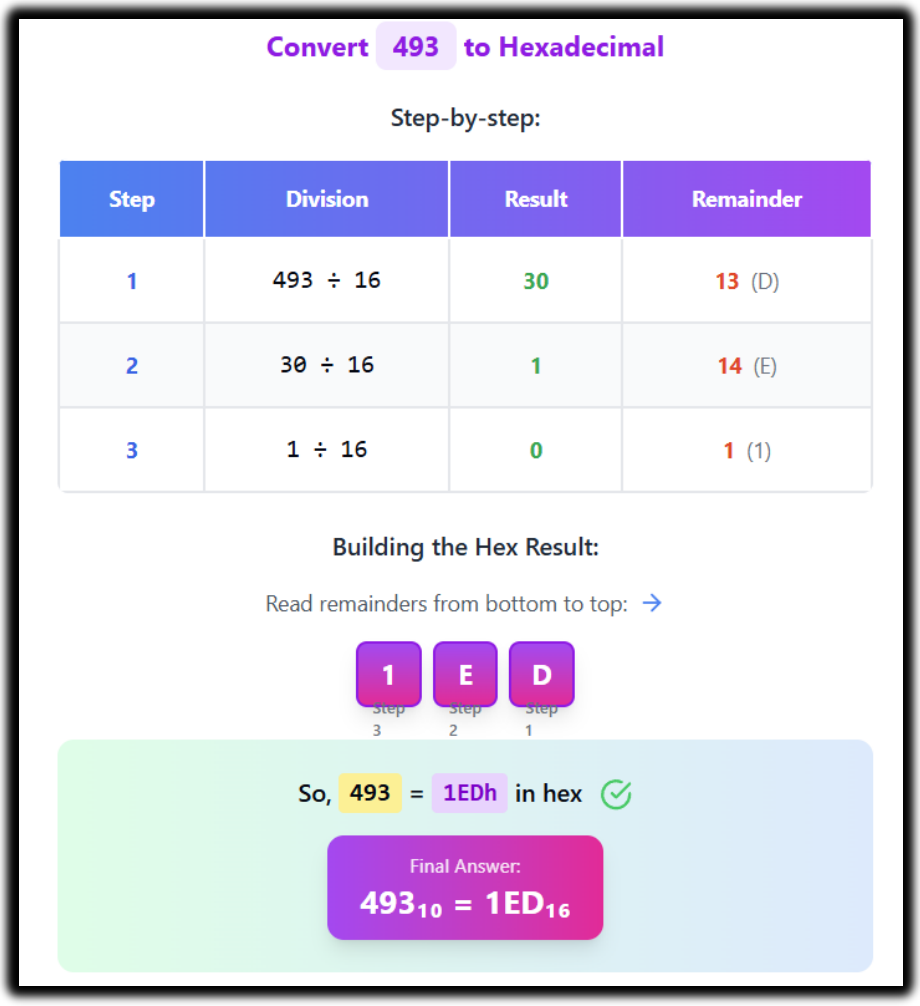


🔄 Decimal to Hex: Division and Remainders

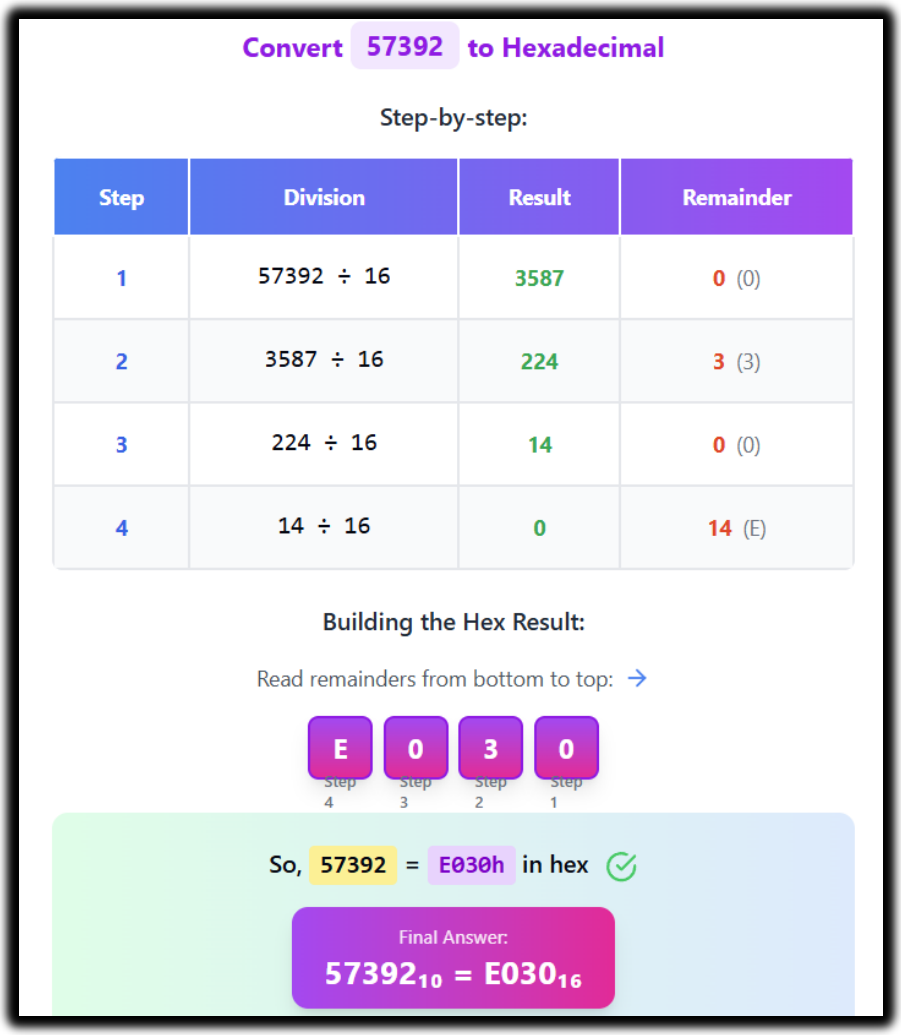
This is like reverse engineering. You divide by 16 repeatedly, collecting **remainders**. These remainders (in reverse order) give you the hex value.

💻 Example: Convert 493 to Hex

Step-by-step:



💻 Example: Convert 57392 to Hex

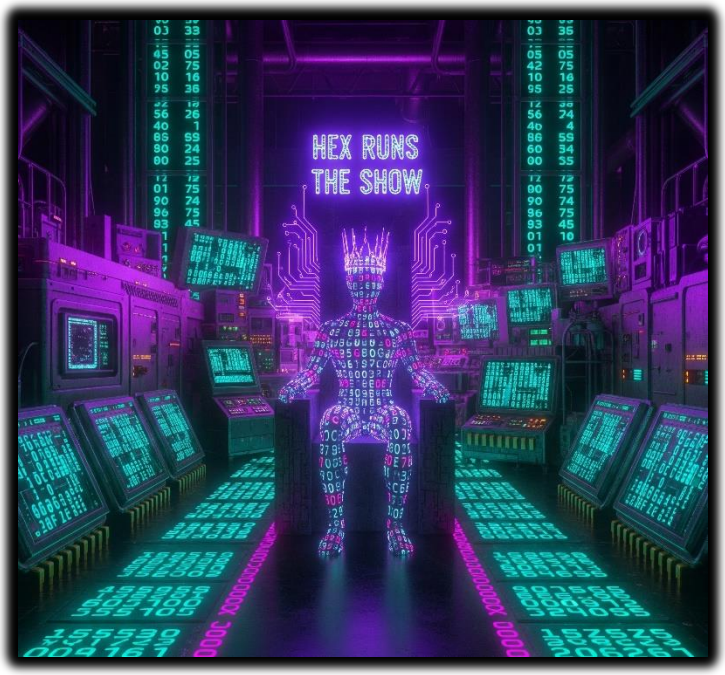


How the Division Method Works

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?

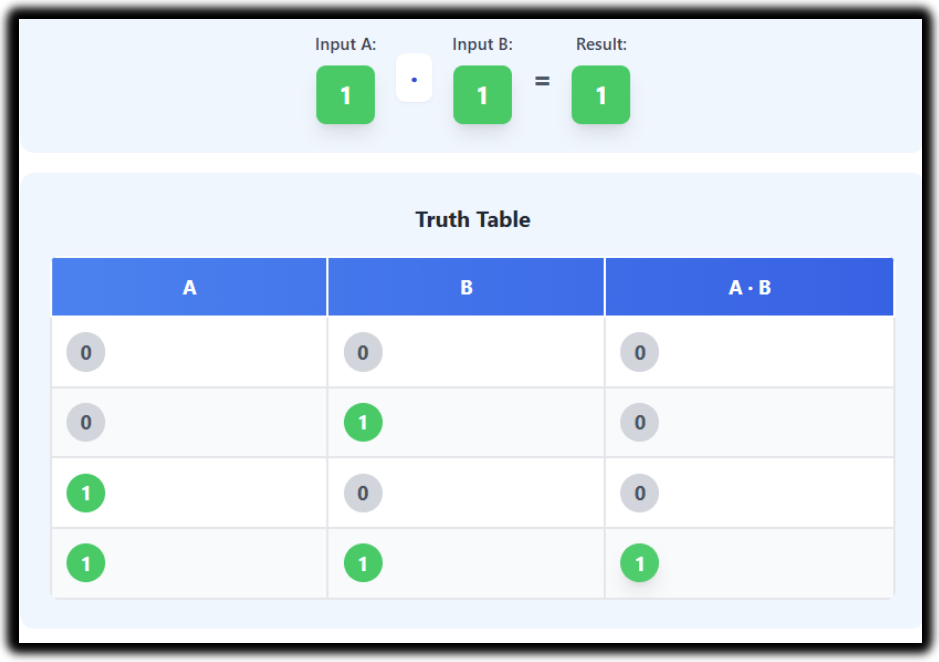
Because ladder logic *is literally* Boolean logic in disguise:

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

✅ Boolean Operators (The Building Blocks)

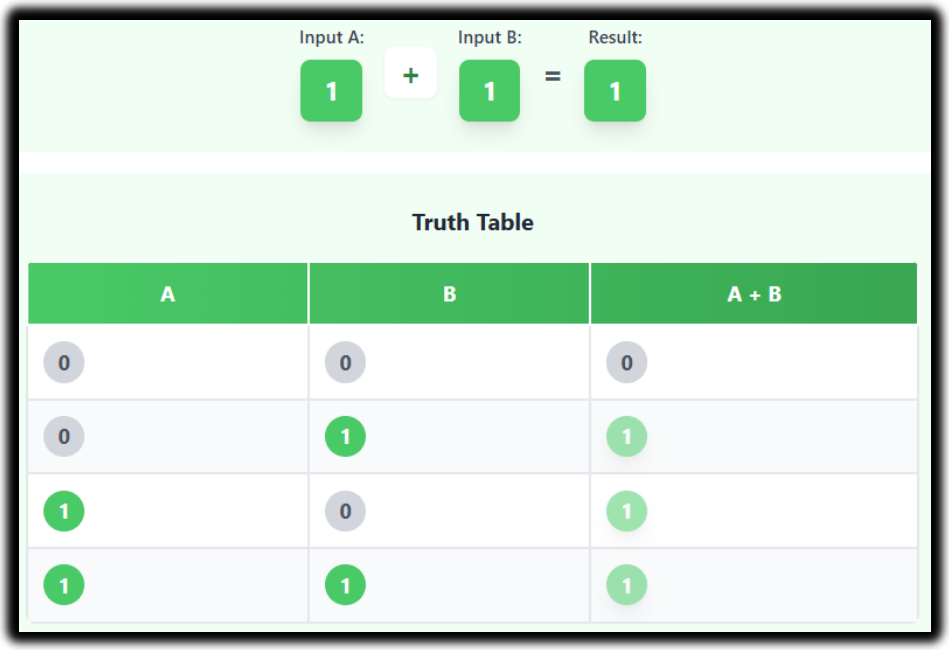
Logic	Symbol	Meaning
AND	· or AB	Both A and B must be 1 *Returns 1 only when both inputs are 1*
OR	+ or A + B	Either A or B (or both) is 1 *Returns 1 when at least one input is 1*
NOT	¯ or !A	Opposite of the value (1 → 0, 0 → 1) *Inverts the input value*

AND Gate



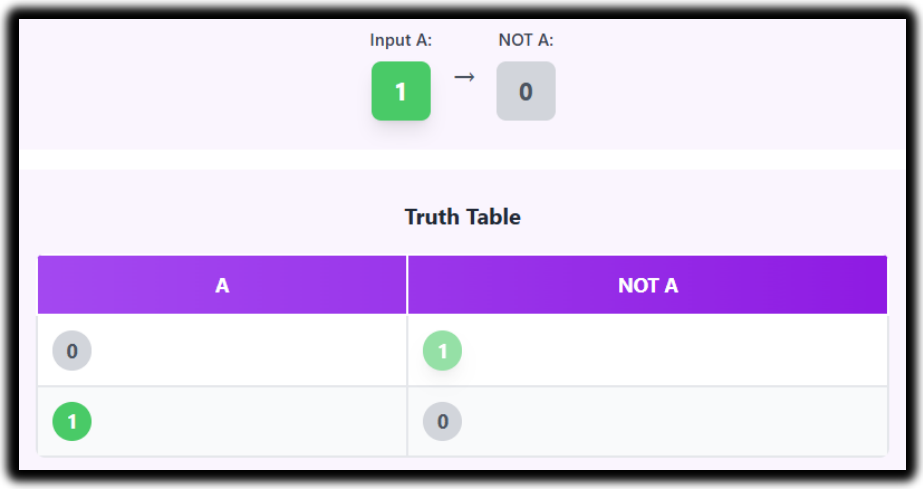
Both A and B must be 1.

OR Gate



Either A or B must be 1.

NOT Gate



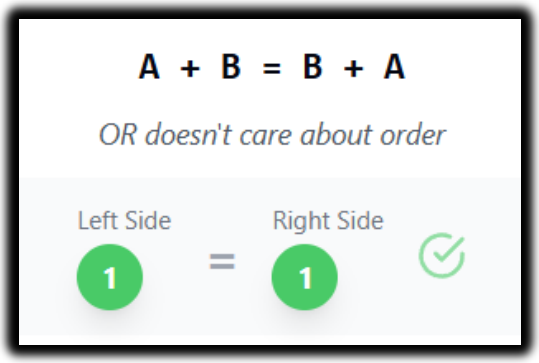
Opposite of the value. Flips it.

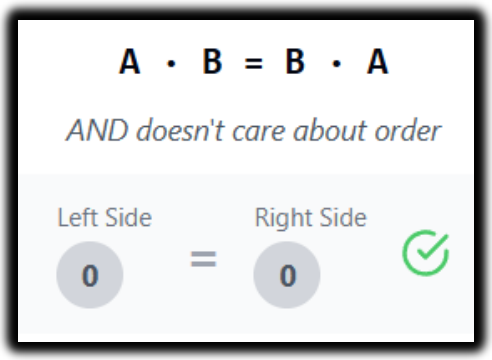
📜 7 Main Boolean Laws (Clean Version)

Master the fundamental laws that govern Boolean algebra - the building blocks of digital logic! 🚀

1. Commutative Laws

- **A + B = B + A** (OR doesn't care about order)
- **A · B = B · A** (AND doesn't care about order)



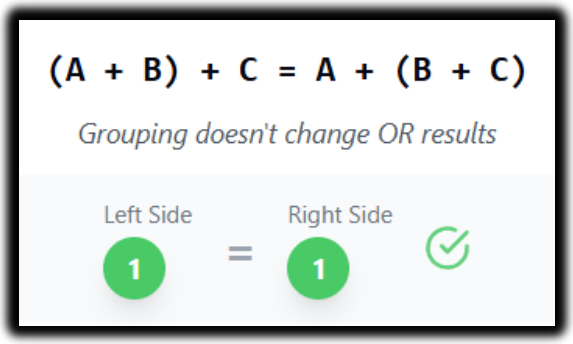


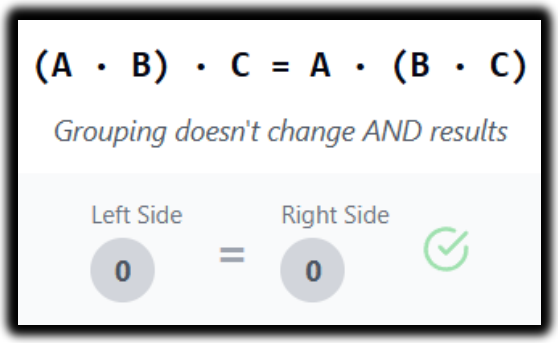
Just like 2 + 3 = 3 + 2 in regular math, Boolean operations don't care about the order of inputs!

2. Associative Laws

Grouping doesn't change the result.

You can group operations however you want - the parentheses can move around freely!





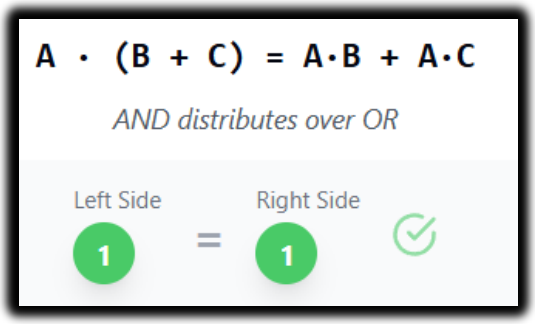
Parentheses can group operations differently without changing the result –

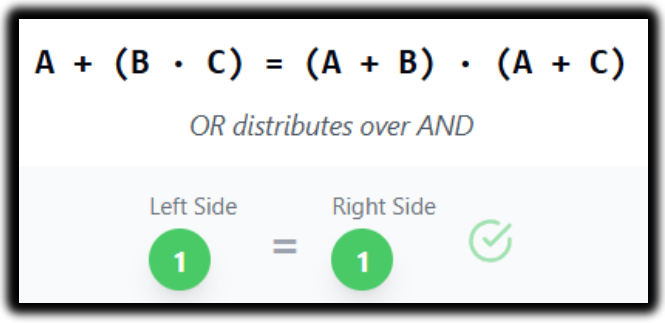
(2 + 3) + 4 = 2 + (3 + 4)!

3. Distributive Laws

Like algebra with a logic twist.

You can distribute operations across parentheses, just like in regular algebra!





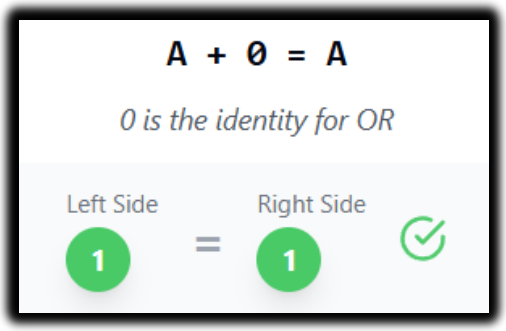
You can 'distribute' one operation across another, just like **a(b + c) = ab + ac** in algebra!

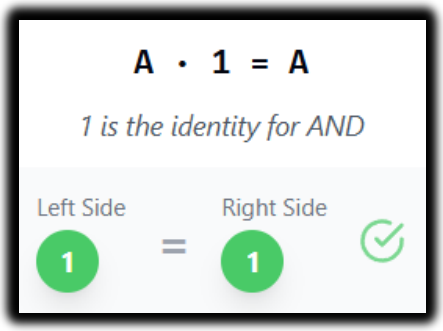
The dot doesn't mean anything, just multiply. That's why it could be written as above highlighted in yellow. **AND** means multiply. **OR** means addition. Have that in mind.

4. Identity Laws

0 is the identity for OR, 1 is the identity for AND.

These are the 'do nothing' operations - they leave values unchanged!





These values don't change the result - like multiplying by 1 or adding 0 in regular math!

In boolean algebra, **"Identity Laws"** describe **"do nothing"** operations.

For the **OR operation** (represented by +), adding 0 to any value A keeps A unchanged (e.g. A + 0 = A), just like adding zero in regular math.

For the **AND operation** (represented by • or multiplication), multiplying any value A by 1 keeps A unchanged (e.g., A • 1 = A), similar to multiplying by one in regular math.

These specific values, 0 for OR and 1 for AND, are called **"identities"** because they don't alter the original input.

5. Null Laws

1 in OR dominates, 0 in AND kills everything.

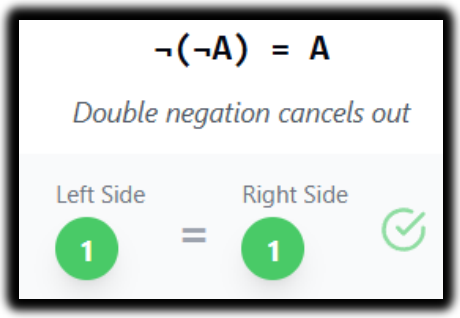
These are the **'dominating'** values that override everything else!

These values **completely dominate** the result, no matter what the other input is!

6. Involution Law

Double negation cancels out.

Two NOTs make a positive - just like in regular language!

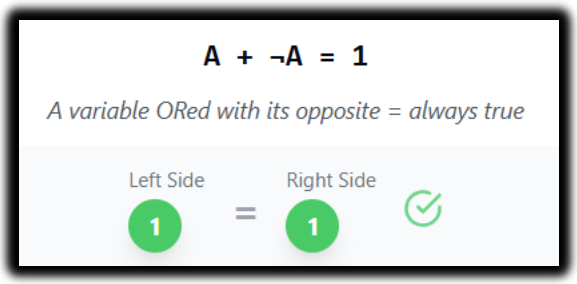


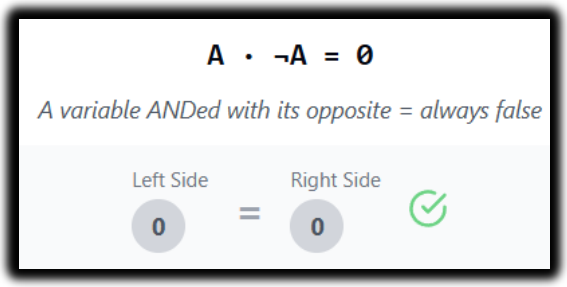
Two negations cancel each other out - 'It's not true that it's not raining' means 'It's raining'!

7. Complement Laws

A variable with its opposite.

When you combine a variable with its opposite, you get predictable results!





A variable combined with its opposite gives predictable results - something can't be both true AND false!

Simple, isn't it? These 7 laws are the foundation of all digital logic. You're now ready to tackle any Boolean algebra problem! 🎉

Let's quickly do the truth tables for these Logic Gates and then we shall jump straight into PLCs. All this was just an introduction, laying the groundwork.

LOGIC GATES

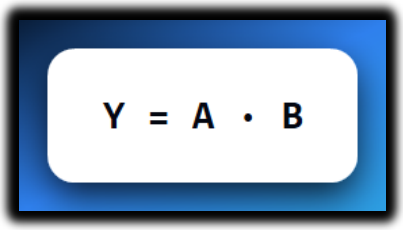
The Building Blocks of Digital Logic

Welcome to the most epic logic gates journey ever! We'll master the triple threats (AND, OR, NOT) and then dive deep into the advanced squad! 🚀

1. AND Gate

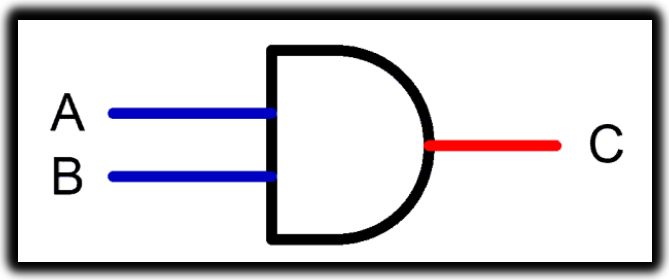
Output is 1 only when BOTH inputs are 1.

Boolean Equation:



PLC Application

Motor runs only when START button is pressed AND safety guard is closed.



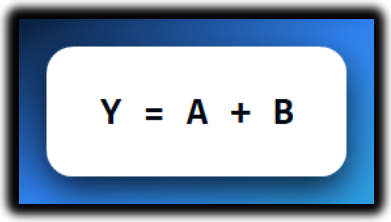
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

🌎 **Real World Example**

Car engine starts only when key is turned AND seatbelt is fastened.

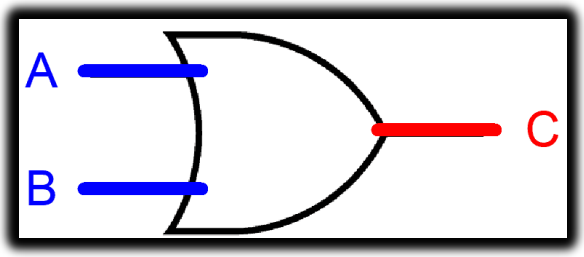
2. OR Gate

Output is 1 when AT LEAST ONE input is 1.



PLC Application

Alarm sounds when temperature is high OR pressure is high.



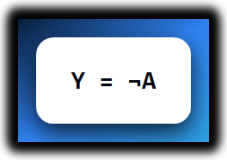
Input		Output
A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

🌎 **Real World Example**

Room light turns on from wall switch OR bedside switch.

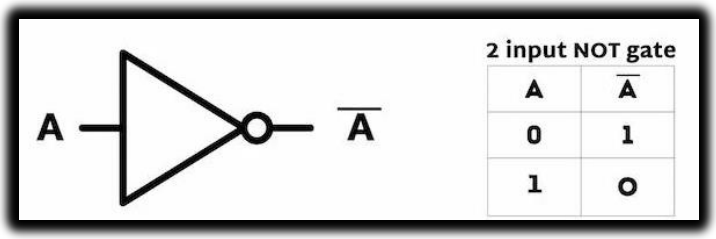
3. NOT Gate

Output is the OPPOSITE of the input.



PLC Application

Conveyor stops when emergency stop button is NOT pressed.

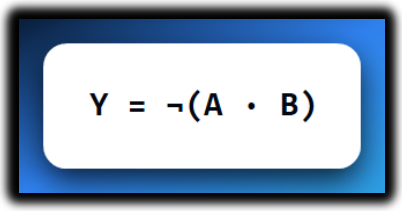


🌍 Real World Example

Security light turns ON when motion sensor is OFF (no motion).

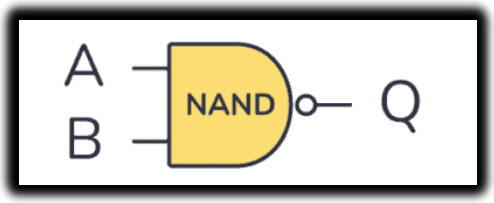
4. NAND Gate

NOT-AND: Output is 0 only when BOTH inputs are 1.



PLC Application

Safety system triggers unless BOTH sensors detect normal conditions.



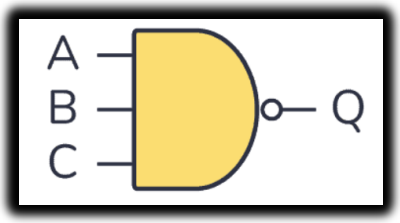
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

🌍 Real World Example

Car alarm is OFF only when key is in AND door is locked.

If A or B is false, then Q is true.

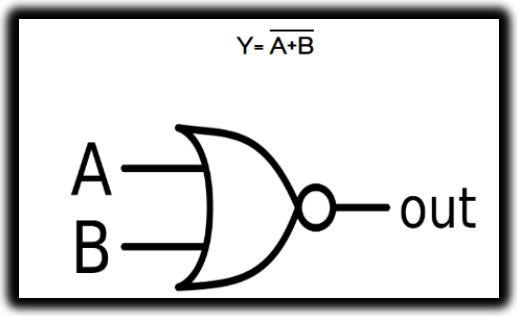
5. 3-input AND



Input A	Input B	Input C	Output Q
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

4. NOR Gate

NOT-OR: Output is 1 only when BOTH inputs are 0



A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

PLC Application:

System is safe only when NO faults are present.

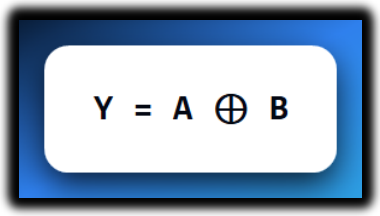
🌎 **Real World Example:**

Safe opens only when NEITHER alarm is active NOR motion is detected.

5. XOR Gate

Exclusive OR: Output is 1 when inputs are DIFFERENT.

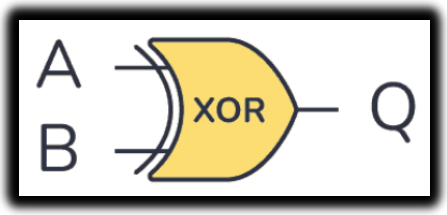
Boolean Equation



PLC Application

Parity checker: output is 1 when odd number of 1s.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



🌎 Real World Example

Stairway light: either upstairs OR downstairs switch (but not both).

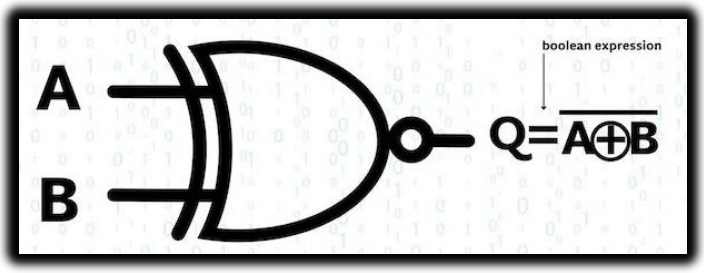
- **Binary Addition**: XOR gates are used in half adders and full adders to perform binary addition of two bits, producing the sum bit.
- **Parity Generation and Checking**: XOR is employed to generate and check parity bits, ensuring data integrity by verifying whether the number of 1's is odd or even.
- **Data Encryption and Decryption**: XOR is widely used in cryptographic algorithms, where it combines data during encryption and reverses the process during decryption.
- **Digital Signal Processing (DSP)**: XOR gates are used in signal processing applications, such as modulation, demodulation, and noise shaping in audio systems.
- **Generating Random Numbers:** XOR gates are often part of pseudorandom number generators in digital systems, contributing to the creation of random sequences.
- **Bitwise Comparison:** XOR is used for bitwise comparison, highlighting positions where two binary numbers differ.

- **Error Detection and Correction:** XOR plays a key role in error-detecting and error-correcting codes like Hamming code, allowing for identification and correction of single-bit errors.
- **Parity Bit in Memory Storage:** XOR gates are involved in computing and verifying parity bits in memory cells, ensuring data consistency and error correction.

6. XNOR Gate

Exclusive NOR: Output is 1 when inputs are the SAME.

Boolean Equation and Logic Gate



An XNOR gate produces a high output (1) only if its two inputs are equal. Commonly used in digital circuits to perform arithmetic and data processing operations.

0	0	1
0	1	0
1	0	0
1	1	1

If A and B are the same, then Q is true.

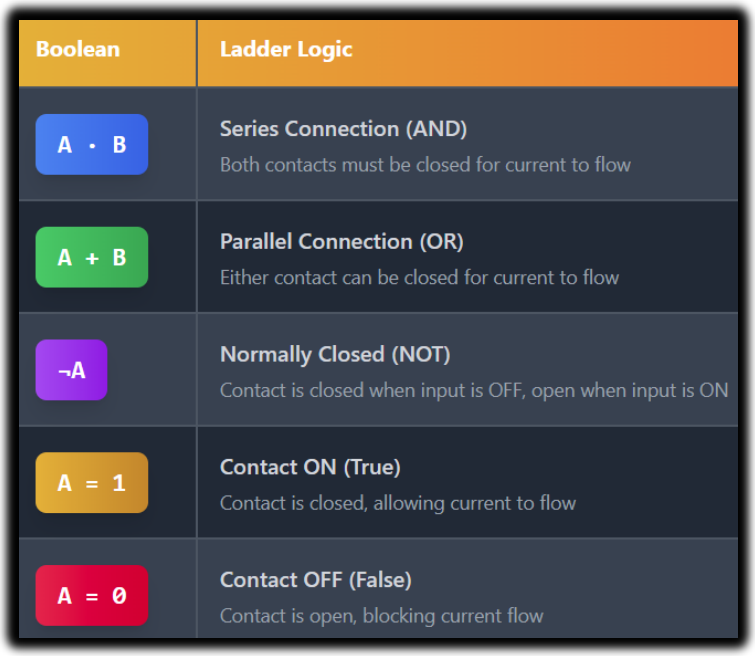
PLC Application

Quality control: pass when measurements are equal.

🌍 **Real World Example**

Garage door: opens when remote signal MATCHES security code.

BOOLEAN TO LADDER LOGIC CONVERSION



The Building Blocks of Logic

Think of these diagrams as a series of instructions that a machine follows, one line at a time. It's like a simplified programming language where everything is either "ON" or "OFF."

🧠 Think of Ladder Logic like Industrial Pseudocode

Ladder Logic is basically **visual code** where each rung is a logic statement:

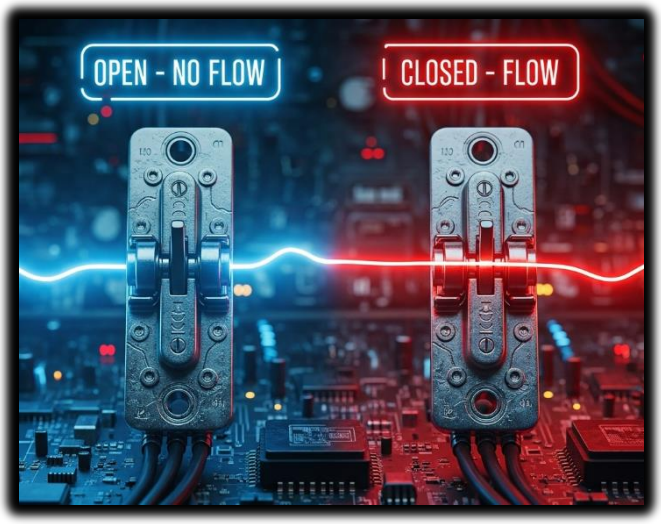
- It runs **top to bottom, left to right**—just like reading a book.
- Each rung is like:
  if (condition) → then (do something)
- And everything is **binary**—either ON (true, 1) or OFF (false, 0).
  Let's break down the pieces.

🔌 Contacts = The If-Checks

Contacts are your conditional checks. They read inputs from switches, sensors, or internal memory bits.

1. Normally Open (NO) Contact: | |

A Normally Open contact is like a gate that starts **closed** (blocking the path). For electricity to flow through it, something must happen first to **open the gate**.



How It Works

- **Normal state:** The contact is OPEN = No electricity flows through
- **When activated:** The contact CLOSES = Electricity can flow through
- **Think of it as:** A switch that must be turned ON for power to pass

Real World Example: Coffee Machine Start Button

Imagine the start button on your office coffee machine:

- **Normal state:** Button is not pressed → Contact is OPEN → Coffee machine won't start
- **When you press it:** Button gets pressed → Contact CLOSES → Power flows → Coffee machine starts brewing

The button "normally" sits there doing nothing (open), but when you press it, it closes the circuit and lets power through.

Other Common Examples

- **Car ignition button** - Normally off, press to start
- **Doorbell button** - Normally quiet, press to ring
- **Elevator call button** - Normally waiting, press to call elevator
- **Motion sensor** - Normally detects nothing, activates when it sees movement

Key Point

The name "Normally Open" means the contact is open (blocking power) when nothing is happening to it. You need to DO something (press, activate, trigger) to close it and let power flow.

2. Normally Closed (NC) Contact: |/|

A Normally Closed contact is like a gate that starts **open** (allowing the path). For electricity to STOP flowing through it, something must happen to **close the gate**.

How It Works

- **Normal state:** The contact is CLOSED = Electricity flows through freely
- **When activated:** The contact OPENS = Electricity STOPS flowing
- **Think of it as:** A switch that's normally ON, but turns OFF when triggered

Real World Example: Safety Door on a Microwave

Think about your microwave oven door:

- **Normal state:** Door is closed → Contact is CLOSED → Power flows → Microwave can run
- **When you open the door:** Door opens → Contact OPENS → Power stops → Microwave shuts off immediately

The door switch is "normally" letting power through (closed), but when you open the door, it cuts the power for safety.

Other Common Examples

- **Car seatbelt warning** - Normally quiet when buckled, beeps when unbuckled
- **Fire alarm pull station** - Normally allows system to work, triggers alarm when pulled
- **Emergency stop button** - Normally lets machine run, stops everything when pressed
- **Refrigerator door light** - Normally off when door closed, turns on when door opens

Why Use Normally Closed?

Safety First! NC contacts are perfect for safety systems because:

- If a wire breaks or sensor fails → The contact opens → Machine stops (safe)
- You want things to work normally, but stop immediately when something goes wrong

Key Point

The name "Normally Closed" means the contact allows power to flow when nothing is happening to it. When something activates it (danger, problem, or trigger), it opens and cuts the power - making it perfect for safety systems that need to "fail safe."

⚡ Coils = The Action (Then Do This)

Coils are outputs. They trigger actions: start motors, turn on lights, activate alarms, etc.

Coils are like the "do something" part of your PLC program.

When all the conditions on a line are met (all the switches, sensors, etc. are in the right position), the coil gets power and makes something happen in the real world.

1. Output Coil ( )

This is your basic **"turn something ON"** coil. When the logic is satisfied, it energizes and activates whatever it's connected to.

How It Works

- **Logic satisfied:** All contacts are in the right position → Coil gets power → Device turns ON
- **Logic not satisfied:** One or more contacts block the path → Coil has no power → Device stays OFF

Real World Example: Garage Door Opener

Think of your garage door opener system:

- **Conditions:** Remote button pressed AND safety sensors clear AND power is on
- **Action:** When ALL conditions are met → Output coil energizes → Garage door motor starts → Door opens

Other Examples:

- Turn on conveyor belt motor
- Activate warning light
- Start air conditioning
- Open valve for water flow

2. Negated Output Coil (/)

This is the "turn something OFF when conditions are met" coil. It works opposite to the regular coil.

How It Works

- **Logic satisfied:** Conditions are met → Coil energizes → Connected device turns OFF
- **Logic not satisfied:** Conditions not met → Coil not energized → Connected device stays ON

Real World Example: Smart Thermostat Fan

Think of a cooling fan controlled by temperature:

- **Normal state:** Fan runs to keep things cool
- **Condition:** Temperature drops below 20°C
- **Action:** When temperature IS low → Negated coil energizes → Fan turns OFF (no longer needed)

Other Examples:

- Turn off heater when room gets warm enough.
- Stop alarm when problem is fixed.
- Close valve when tank is full.

3. Latching Coil (L) and Unlatching Coil (U) - Symbols: (L) and (U)

These work like a **"sticky switch"** - once you turn something ON with the Latch coil, it STAYS on even if you let go. The only way to turn it OFF is with the Unlatch coil.

How They Work

- **Latch (L):** Quick pulse of power → Device turns ON and STAYS on forever
- **Unlatch (U):** Quick pulse of power → Device turns OFF and STAYS off forever

Real World Example: Hotel Room Lights

Think of those hotel room card key systems:

- **Insert card:** Latching coil (L) energizes → Room lights turn ON and STAY on
- **Remove card:** Nothing happens → Lights STAY on (that's the latch working)
- **Press main switch:** Unlatching coil (U) energizes → All lights turn OFF and STAY off

Even Better Example: Emergency Stop System

Emergency Stop Button Pressed:

- Latching coil (L) energizes → All machines STOP and STAY stopped
- Even if someone accidentally bumps the E-stop button again, machines stay OFF

Reset Button Pressed (by supervisor with key):

- Unlatching coil (U) energizes → System is ready to start again (but doesn't start automatically)

Why This Matters:

Safety! Once an emergency stop happens, you can't accidentally restart dangerous machinery. Someone has to deliberately reset the system with a key.

Other Examples:

- Fire alarm system (latch when smoke detected, unlatch when reset by fire department)
- Security gate (latch open with access card, unlatch closed with timer)
- Process start/stop (latch production line on, unlatch when shift ends)

Key Takeaway

- **Output Coil ( )**: Turn something ON when conditions are right
- **Negated Coil (/)**: Turn something OFF when conditions are right
- **Latch/Unlatch (L)/(U)**: Turn something ON/OFF and make it STAY that way until told otherwise

LOGIC CONCEPTS ILLUSTRATIONS

Boolean equation: AB = Y

Boolean equation: $A \cdot \bar{B} = Y$

See the symbol above B it's a NOT.

In a truth table, if B is true (1), then B' (or ¬B) is false (0), and vice versa.

Boolean equation: A + B = Y

That's an OR gate. If either switch is pressed, the Solenoid gets energized.

LS typically stands for **"Limit Switch"** a common sensor in industrial applications.

SOL stands for **"Solenoid"** which is a device that uses an electromagnet to create a linear or rotational motion. It's a common output device.

Because LS1 and LS2 are in parallel, the current can flow through either one to reach the SOL coil.

- **Scenario 1:** If you actuate LS1 (press the switch) but not LS2, the circuit is complete through LS1, and the SOL is energized.
- **Scenario 2:** If you actuate LS2 but not LS1, the circuit is complete through LS2, and the SOL is energized.
- **Scenario 3:** If you actuate both LS1 and LS2, the circuit is complete through both paths, and the SOL is energized.

The only way for the SOL to *not* be energized is if *neither* LS1 nor LS2 is actuated.

Boolean equation: **(A + B) C = Y**

Boolean Equation: **(A + B) (C + D) = Y**

We're done with note-taking and basic PLC theory. Now it's time to focus on understanding ladder logic through practice, discussions here, and using online PLC simulators for better clarity.

Let's first do Timers and Counters which are the final topics that we shall need for this ladder logic drawing… then we shall jump straight into questions beginner to advanced topics….

🦴 Mastering PLC Timers, Counters & Internal Memory

✨ For Future Automation Beasts

I. Intro: Welcome to the Machine

PLCs — they're not just little boxes with blinking lights. These bad boys are the **real MVPs** of industrial automation. Imagine a PC, but way tougher. We're talking full-time, no-sleep, heat-proof, noise-resistant, punch-a-wall-and-still-run kind of rugged.

To learn timers, I will take you back to the beginning and come building up.

What is a PLC?

It's a dedicated industrial computer built to **run machines**, handle **real-world inputs and outputs**, and do it all **nonstop, 24/7**, without throwing errors or crashing like your laptop mid-Zoom call.

Why not use a regular PC?

Because PCs can't handle the factory life — not the heat, not the voltage spikes, not the dust storms inside a bottling plant. PLCs thrive in the chaos. They were literally **engineered for the madness** of motors, valves, and sensor storms.

Where are they used?

Everywhere serious automation happens:

- Factory assembly lines 🏭
- Chemical plants 🧪
- Hydroelectric dams 💧⚡
- Food processing lines 🍟
- Basically, anywhere "don't mess this up" is part of the job description.

🚀 From Relays to Real-Time Logic

How PLCs Leveled Up

PLCs didn't start off as the digital ninjas they are today.
In the beginning? They were just **replacements for clunky relay panels** — good for basic ON/OFF control, a few timers, a few counters. Nothing fancy.

But as industry scaled up, the pressure grew. More machines. More speed. More precision.
So PLCs had to **evolve** — and they did. Fast.

Now?
Modern PLCs can:

- Run advanced math operations
- Handle complex logic structures
- Talk to other devices over Ethernet, Modbus, Profibus — you name it
- React to events, not just fixed scans
- Store and manipulate huge sets of internal data

Translation?

We're no longer just flipping coils and pushing bits.
We're now working with **dynamic, intelligent control systems** that expect you to know how to juggle memory, timing, and communication like a pro.

📘 Why This Guide Exists

Let's be honest: learning timers, counters, and memory bits online can be a **hot mess**.
Some tutorials assume you already know everything. Others teach like it's still 1995.
And nobody connects the dots.

So, this guide?
We're breaking it down *properly*.
You'll learn:

- What each timer and counter really does (not just what the manual says).
- How to actually wire these things in ladder logic.
- Where and **why** to use internal memory bits.
- Real-life examples from automation scenarios.
- Plus, solid resources so you're not just guessing with YouTube videos.

This is about **leveling up**.
Not just learning a new instruction — but understanding a whole new mindset:

🔄 One that's built around timing, memory, logic flow, and real-world constraints.

You ready? Good.
Let's build that PLC brain.

🔁 The Scan Cycle: The Pulse of the PLC

At the core of every PLC's brain is one relentless rhythm — the **scan cycle**.
It's a super-fast loop that never stops, running **thousands of times per second**:

1. 🧠 **Read Inputs** – What's the world saying right now?
2. 🔍 **Execute Logic** – Based on those inputs, what should we do?
3. ⚡ **Update Outputs** – Send the commands to motors, lights, valves, etc.

This cycle is *everything*.
It's how your PLC makes sense of the world — input, decide, act — over and over, like a machine heartbeat.

But here's the twist that trips up a lot of beginners:

Outputs don't change the moment an input changes.

Nope — they only update **after** the next full scan.
So if your program relies on microsecond timing or lightning-fast sensors, that tiny delay?
It can make or break your whole logic.

That's why **understanding the scan cycle isn't optional**.
It's essential for:

- Using timers and counters correctly
- Catching fast events (like a sensor that only blinks for a millisecond)
- Designing **reliable**, **predictable**, and **safe** control logic

If your control system is misfiring or feels out of sync?
🧠 First place to look: **your scan cycle timing.**

⚡ II. Ladder Logic: The OG Language of Control

Ladder Logic isn't just some random PLC programming language — it's a **deliberate throwback** to the old-school way factories used to automate stuff.

🧲 Flashback to the Relay Era:

Before 1968, industrial automation meant **relay cabinets** — gigantic walls stuffed with electromagnetic relays wired up in crazy patterns.
Each relay was basically an electrical switch, and wiring up a control system meant **drawing out a ladder**:

- Two vertical lines = power rails
- Horizontal lines = logic paths (aka "rungs") with contacts and coils

It worked, but man — it was a nightmare:

- 💸 Expensive
- 🧱 Took up entire rooms
- 🧩 Changing anything meant ripping wires and redoing the layout

💡 Enter the PLC Revolution

Then came Modicon. Their first PLCs — the **Modicon 084** and the game-changing **Modicon 184** — flipped the entire industry on its head.

Instead of using physical relays, they said:

"Let's simulate those relay diagrams in software."

And just like that, the first programmable controller was born. Same ladder style. Same logic flow. But now? It's all done in code.

This was *brilliant*, because:

- Electricians and technicians didn't need to learn a new language
- They could **read ladder logic just like a wiring diagram**
- No more rewiring — you just changed the program

That's why Ladder Logic is still the **king** in automation today. It's familiar. It's visual. And it gets the job done — especially in environments where clarity, speed, and reliability matter more than code elegance.

🧠 Relay Logic vs. Ladder Logic: Same Look, Different Game

Here's where a lot of people trip up — especially electricians making the jump from physical panels to PLCs.

At first glance, **Relay Logic** and **Ladder Logic** look almost identical.
Same rungs. Same coils. Same contacts. But **under the hood?** Totally different beasts.

⚙️ Relay Logic: Real Wires, Real Current

Old-school relay logic = **actual electrical current** flowing through **actual wires**, switching **real coils and contacts** in big, noisy panels.

- Contacts are real metal switches.
- Coils are electromagnetic relays.
- Power actually flows left to right on the ladder.

When you press a button, **electricity** physically moves across the rung and energizes something — a motor, a lamp, a siren.

💻 Ladder Logic (PLC): Virtual Wires, Logical Flow

Now flip to PLC ladder logic.
Same ladder? Yeah. But now:

- The "contacts" are just **memory bits** inside the PLC.
- The "coils" are software instructions.
- And there's **no real power** flowing — it's all just 1s and 0s.

When you press a button, the PLC **scans the input**, evaluates the logic, and if the conditions are right? It **sets a memory bit to TRUE**. That bit tells the **output module** to energize the actual hardware — the motor, the lamp, whatever.

So, in the PLC:

"Power flow" = **Logical flow**
If the rung is TRUE → Output turns ON
If the rung is FALSE → Output stays OFF

No sparks. Just **binary truth** inside the CPU.

🔍 What This Means for Troubleshooting

In relay logic, if something breaks, you grab a multimeter and follow the wire.

In PLC logic? You grab your **programming software** and trace the logic:

- Is the input bit changing?
- Is the rung evaluating to TRUE?
- Is the output bit firing?
- Is the output module receiving the command?

It's not about voltage anymore — it's about **logic tracing** and **bit behavior**.

This shift to virtual logic is why PLCs are so flexible — but also why new learners need to rewire their brains a bit.

You're not chasing electrons.
You're chasing logic. 🧠⚡

🧱 Essential Ladder Logic Building Blocks

💯 Know These, or Don't Even Start Programming

Before you start writing your first PLC program, you need to **master the basics** — the components that make up every single rung of ladder logic. If ladder logic is a language, these are its alphabet.

⚡ Rails & Rungs: The Skeleton of Every Ladder

- **Rails** are the **two vertical lines** running down the left and right edges of your ladder diagram.
  Think of them as the start and end of your logic — like a power source and return line.
- **Rungs** are the **horizontal lines** that stretch between the rails.
  Each rung = one line of logic. Just like a sentence in programming.

🔀 **Logic flows left to right** — not top to bottom. The PLC reads each rung from left to right, top to bottom, and decides whether it evaluates to TRUE or FALSE.

🔘 Contacts: Inputs That Control the Flow

Let's break down the two core types:

1. Normally Open (NO) Contact – --| |--

- Starts out **open** (no logical flow).
- Closes (allows logic to pass) **only when the input is ON** (e.g., a pushbutton is pressed).
- Think: Doorbell. Nothing happens until you press it. 🛎️

If the input bit is TRUE → the contact closes → logic flows through.

2. Normally Closed (NC) Contact – --|/|--

- Starts out **closed** (logic can flow).
- **Opens** (blocks logic) **when the input turns ON**.
- Used in safety systems where failure = shutdown.

🔒 **Example:**
An emergency stop (E-Stop) is usually wired as NC.

- If it's working and untouched → logic flows.
- If someone hits the button *or* the wire breaks → the circuit opens → the machine stops.
  That's called a **fail-safe design** — *if anything goes wrong, it shuts things down*.

🧠 Why This Matters

Every control routine starts with understanding:

- What your **inputs** are doing (contacts),
- How the **logic flows** across the rung (left to right),
- And how **your outputs** will respond (which we'll hit in the next part).

💡 Output Coils: Lighting Up the Real World

🔘 OTE — Output Energize Instruction --( )--

This is your **go-to output instruction** in ladder logic. Simple, reliable, and used in almost every basic control task.

🔌 How It Works

- The OTE coil sits at the **far right** of the rung.
- It waits for **logical power** (i.e., a TRUE condition) to reach it.
- If the rung's logic evaluates to TRUE → the OTE **energizes** → turns on a real-world output device.

💡 Think:

- Motor starts spinning
- Light turns on
- Solenoid activates
- Buzzer screams at 120dB

🧠 Important Behavior: No Memory

This part is key:
The OTE **doesn't remember anything**. It only stays ON as long as the rung conditions are TRUE.

- Rung TRUE = Output ON
- Rung FALSE = Output OFF
  No in-between. No delay. No memory.

It's just like a basic light switch:

Flip the switch ON = light is ON
Flip it OFF = light is OFF immediately

So, if the input conditions blink ON for just one scan and go back to FALSE, the output also turns OFF — **instantly**.

Use OTEs when you want **simple, responsive, no-state-needed control**.

🎯 One-Shot Contacts: Mastering Momentary Logic

🔫 Fire Once, Then Chill.

In ladder logic, sometimes you don't want an output to stay ON — you just want to **trigger it once** when something changes. That's what **one-shot** contacts are for. They're like digital snipers: **One pull. One shot. One scan cycle. Done.**

⚡ Positive Transition (Rising Edge) — P_TRIG / ONS / --[↑]--

This is your **"On-to-Off" gatekeeper**. It only activates when the input **changes from 0 to 1** — aka a **positive edge** or **rising signal**.

🧠 Example:
A part passes a photo-eye sensor. The sensor stays ON for a few milliseconds, maybe even several scan cycles.
You only want to **count the part once**, right?
If you use a regular contact — it'll count every scan while the sensor stays ON.
But with a **one-shot P_TRIG**? Boom — **only counts once**, right when the sensor flips ON.

📌 Real-world use cases:

- Incrementing counters
- Triggering alarms once
- Saving data on an input spike

🧯 Negative Transition (Falling Edge) — N_TRIG / --[↓]--

Now flip it.
N_TRIG activates **only when the input falls from 1 to 0** — aka **OFF trigger**.

🧠 Example:
You've got a conveyor. You want to **log when a box exits a station**, not when it enters.
The sensor is ON while the box is present. When the box leaves → sensor turns OFF → N_TRIG fires once. 📌 Perfect for:

- Cleanup logic
- End-of-task triggers
- Logging event completions
- Resetting temporary bits or conditions

⏱️ One-Shots & Scan Cycles: Timing Is Everything

Here's the **catch** that most rookies miss:

If the input transitions **faster than your PLC's scan cycle**, the one-shot might **completely miss it**.

Let's say your PLC scans every 5ms, but your sensor goes ON and OFF in 2ms.
Too fast. The PLC never sees the edge → your P_TRIG never fires → missed event. 😮

That's why:

- You need to know your PLC's scan time
- For **high-speed inputs**, consider using **dedicated high-speed input modules** or **interrupt routines**

🧠 Bottom Line:

- One-shots are **event detectors**, not state detectors.
- They let you act on **transitions**, not just steady signals.
- Use them when **"just once"** matters more than **"while true"**.

They're small, sharp, and surgical. Use with precision. 🧬

🤖 Boolean Logic in Ladder Diagrams

The Hidden Math Behind the Magic

Even though it looks like wiring, ladder logic is secretly just **Boolean algebra in disguise**.
Everything boils down to **TRUE (1)** or **FALSE (0)** — that's it.
You're not just placing contacts and coils…
You're **building logic gates** with layout alone.

Let's break down the **three core operations** hiding in your ladder rungs:

🔗 AND Logic — Contacts in Series

--| |--| |--( )

Place two or more contacts **in a row**, and you've got an **AND condition**.

- All contacts must be TRUE (ON)
- If *any one* is FALSE → output = FALSE

🧠 Example:
A secure lab door that needs:

- ✅ Keycard swipe
- ✅ Password entry
- ✅ Biometric scan

All must pass → THEN door opens.

🌿 OR Logic — Contacts in Parallel

We already have several images for this.

Put contacts **in parallel**, and boom — it's **OR logic**.

- Only *one* contact needs to be TRUE
- If *any* path is TRUE → output = TRUE

🧠 Example:
An alarm system that triggers if:

- 🔥 Fire detected
- 🛑 E-Stop pressed
- 🚪 Door forced open

One sensor goes off → system reacts.

🔁 NOT Logic — Normally Closed (NC) Contact

--|/|--( )

This is ladder logic's version of **NOT** — just a single **NC contact** flips the logic:

- If the input is ON → NC contact opens → Output = FALSE
- If the input is OFF → NC contact stays closed → Output = TRUE

🧠 Example:
A machine that should run **only when the safety gate is closed**.

- Gate open = sensor ON → NC breaks → machine won't start
- Gate closed = sensor OFF → NC allows logic through

It's **inverted behavior** — and super useful in safety and interlock circuits.

🛠️ It's All About Layout

Here's the catch:

In ladder logic, **you don't write AND/OR/NOT — you draw them.**
Mess up a contact's placement? You could flip the whole meaning of a rung.

That's why **meticulous design matters**.
This isn't like block-based programming (where you drop an AND gate from a menu).
Here, the logic is **implied** by how you arrange contacts — and that makes clarity and testing *crucial*.

You're not just placing instructions —
You're crafting **logic flows with spatial awareness**.

That's why ladder logic is powerful, but **you gotta think like both an electrician and a programmer**.

🔒 The Latching (Seal-In) Circuit

Holding Power Without Holding the Button

This is one of the **most common control patterns** you'll ever use in ladder logic — the **seal-in**, also known as a **latching circuit** or **holding contact**.

What does it do?
It **keeps an output ON** even after the button that started it has been released.

🧠 Why is this useful?

Imagine pressing a "Start" button and having to **hold it forever** just to keep a machine running.
Ridiculous, right? That's why we build a **logic memory loop** that says:

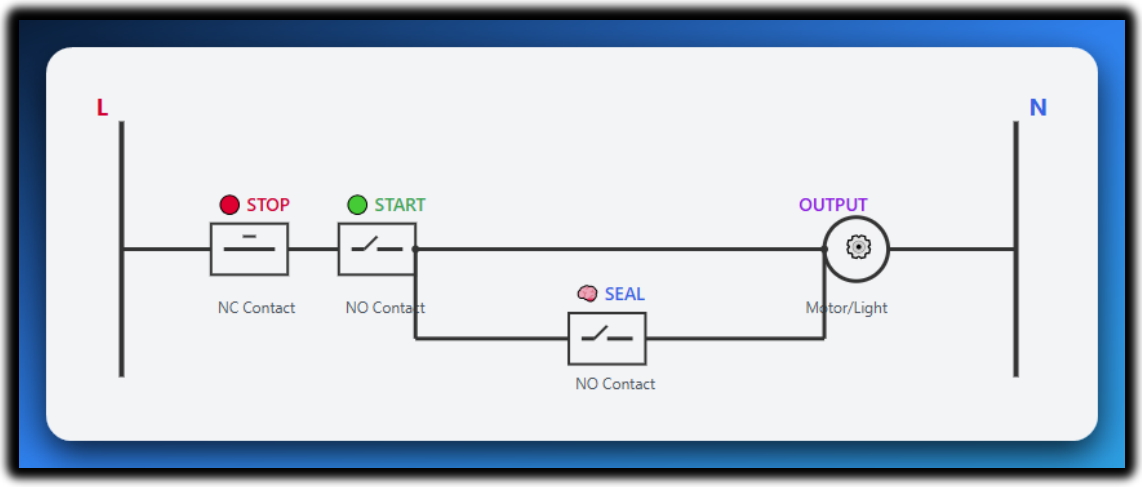
"If I started, I'll stay ON — until told to stop."

🔧 Classic Latching Circuit Breakdown

Components:

- 🟢 **Start button** → Normally Open (NO) contact
- 🔴 **Stop button** → Normally Closed (NC) contact
- ⚙️ **Output coil** (motor, light, etc.) → OTE
- 🧠 **Seal-in contact** → NO contact tied to the output

How It Works:



1. **Start pressed**:
    - Logic flows through **NC Stop** and **NO Start** → energizes output coil
2. **Output turns ON**:
    - Its associated contact (seal-in) **closes** — now there's a **new path** for logic
3. **Start released**:
    - Doesn't matter — seal-in is doing the job now. Output stays ON.
4. **Stop pressed**:
    - NC Stop opens → logic is cut off → output de-energizes → seal-in opens → resets the loop

🧰 Real-World Use Case: Motor Start/Stop

- Press "Start" → Motor turns ON
- Release "Start" → Motor keeps running
- Press "Stop" → Motor shuts off

This behavior is **crucial** in:

- Motor control
- Conveyor systems
- Pumps
- Anything where **continuous operation** is needed after a momentary trigger

🧠 Why It Matters

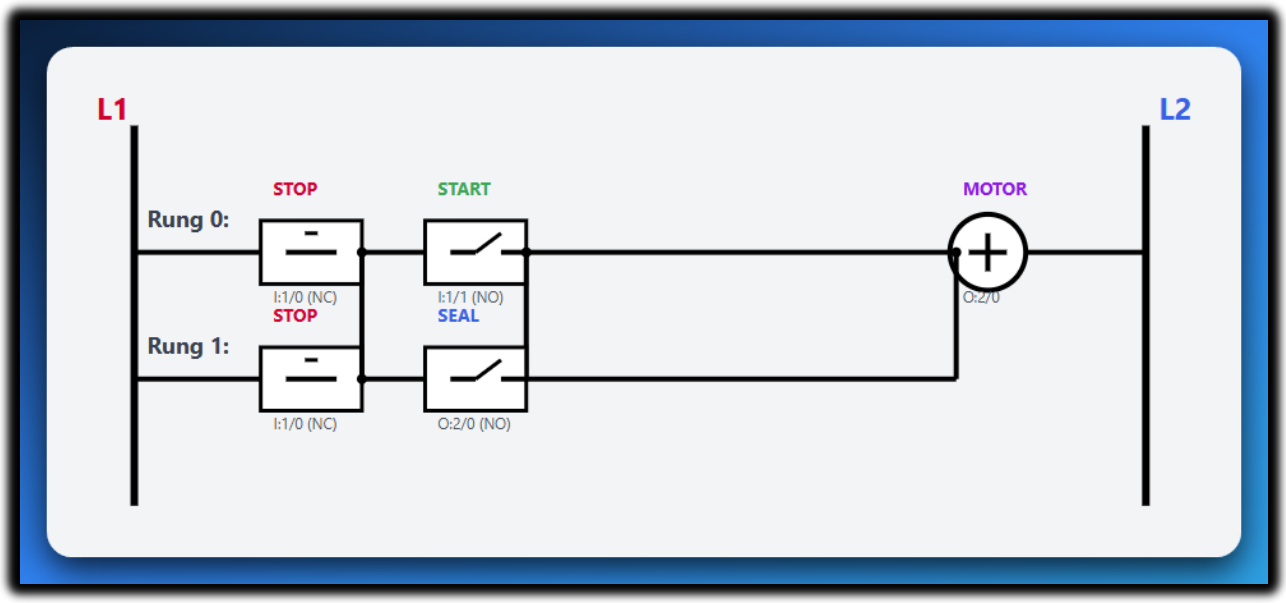
Seal-in logic introduces the idea of **memory** — holding a state across time.
You're not using a retentive bit yet, but it's the **first step** to understanding:

- Retentive timers.
- Latching bits.
- Preserving state through logic loops.

And it's also critical for **safety** — a machine shouldn't stop just because a button was accidentally let go. It should stop only when the operator **deliberately presses STOP**.

💡 Key Concept: Self-Holding

The seal-in contact creates a **parallel path** around the start button. Once energized, the output "holds itself ON" through this parallel path, even when the start button is released. Only the stop button can break the circuit and reset the system.



🔍 What You See Here:

- **Two Rails:** L1 (left) and L2 (right) - like + and - on a battery
- **Two Rungs:** Two parallel paths for electricity to flow
- **Rung 0:** STOP → START → MOTOR (main control)
- **Rung 1:** STOP → SEAL → MOTOR (holds motor ON)
- **Connection Dots:** Show where wires connect together

⚡ How It Works:

1. **Press START:** Electricity flows through STOP→START→MOTOR
2. **Motor Turns ON:** This closes the SEAL contact
3. **Release START:** Electricity now flows through STOP→SEAL→MOTOR
4. **Press STOP:** Both paths are broken, motor turns OFF

💡 The Magic of the Seal-In:

Look at the **SEAL contact (O:2/0)** - it's controlled by the same motor output! When the motor turns ON, it automatically closes this contact, creating a second path to keep itself running.

This is called **"self-holding"** or **"latching"** - the circuit holds itself ON!

🏷️ Address Labels Explained:

I:1/0 = Input 1, Bit 0 (Stop button)

I:1/1 = Input 1, Bit 1 (Start button)

O:2/0 = Output 2, Bit 0 (Motor)

🧠 OTL & OTU: Retentive Control Made Easy

🔒 When You Want Full Manual Control Over ON/OFF States

Sometimes the classic seal-in circuit isn't enough — you need **precise control** over what stays ON or OFF, even if the logic around it changes.

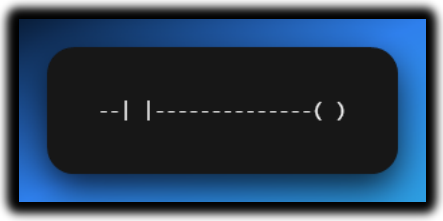
That's where the **OTL (Output Latch)** and **OTU (Output Unlatch)** instructions come in.

These are **retentive instructions** — meaning once they set or clear an output, it stays that way **until you explicitly change it**.

Let's break it down:

💥 OTE — Output Energize

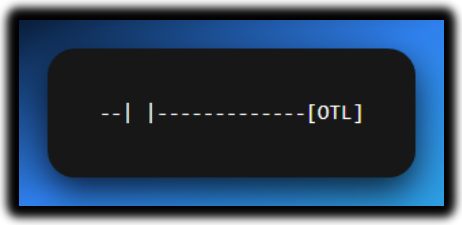
The default. The basic.



- 🧠 **Memory?** None. Volatile.
- ON if rung logic is TRUE.
- OFF the moment logic goes FALSE.
- Great for: Lights, fans, motors — anything that should turn off instantly if the input fails.

💥 OTL — Output Latch

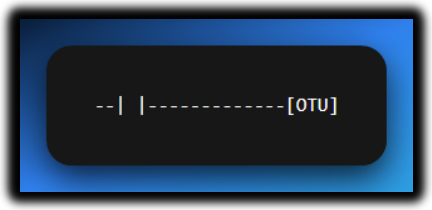
Sets the output **ON**, and keeps it ON.



- 🧠 **Memory?** YES. Retentive.
- ON when rung goes TRUE (even for just **one scan**)
- **Stays ON** until a matching OTU resets it
- Useful for:
  - 🔔 Emergency alarms (require manual reset)
  - 🚩 Status flags (“process complete”)
  - 🛑 Holding a machine in a STOPPED state after fault

💥 OTU — Output Unlatch

Turns the same output **OFF**, and keeps it OFF.



- 🧠 Also **retentive**
- OFF when this rung becomes TRUE (even once)
- **Stays OFF** until something else (like an OTL) turns it back on
- Used for:
  - 🔁 Resetting alarms
  - ✅ Clearing flags or error states
  - 🧼 Cleanup logic after a batch run

🧩 Example Use Case: Fault Alarm Logic

If fault → OTL → Alarm ON

If reset button → OTU → Alarm OFF

This lets you keep the alarm blaring *even if the fault clears*.
The operator must press RESET — ensuring they **saw and acknowledged** the error.

⚠️ Pro Tips:

- OTL/OTU pairs **must use the same address/tag**
- Don't mix OTL with OTE on the same output — it causes unpredictable behavior
- Retentive = **survives logic changes**, but *may not* survive power loss unless stored properly

Use OTL/OTU when you need **intentional, human-triggered state changes**, not just auto-on/off behavior. It's all about **control and safety**.

🧠 Memory vs. Moment: Why This Difference Matters

Here's the real takeaway from the table:

- **OTE = reactive**
  - ➢ It mirrors the rung logic — ON when true, OFF when false.
  - ➢ Like a light switch: flip it, and it instantly obeys.
- **OTL / OTU = retentive**
  - ➢ They don't care what the logic *currently* says.
  - ➢ They remember the last command — ON or OFF — and **hold it** until told otherwise.

🎯 Why That's Huge

This difference isn't just technical — it's **philosophical**.
You're deciding *how much control the PLC keeps vs. how much you, the programmer, take over*.

🔐 Use retentive logic (OTL/OTU) when:

- You want a machine to **remember a fault** until the operator resets it
- You're building a **multi-step sequence** (like filling → mixing → draining)
- You're creating **interlocks** for safety that must be manually cleared

⚡ Use volatile logic (OTE) when:

- You want instant response to input changes
- The output should reflect **real-time** status (like lights or fans)
- Memory would actually cause confusion or danger

📦 It's the Foundation of Sequential Control

Retentive logic is the backbone of:

- Step-by-step automation
- Process hold states
- Fault detection and reset
- Operator-controlled resets and acknowledgements

If you want your PLC to **act like a brain**, not just a switchboard —

You've got to master the art of memory-based output logic.

🧠 Internal Memory Bits — Your PLC's Hidden Superpower

📦 M-Bits / Internal Relays / Flags — Whatever You Call Them, They're Game-Changers

Okay, so you already know that X is for inputs and Y is for outputs in Mitsubishi PLCs.

But what if you want to store a result, remember a condition, or build logic that has **nothing to do with physical wiring**?

That's where **internal memory bits** come in — the M range.

⚙️ What Are M-Bits?

M-bits (M0, M1, M100...) are **virtual switches** inside the PLC's brain.

- They're **Boolean variables**: either ON (1) or OFF (0).
- You **don't wire them** to anything.
- They live entirely **in software.**

💡 You can think of them like this:

If physical inputs/outputs are the muscles, M-bits are the neurons — internal signals making decisions behind the scenes.

🧩 How Are They Used?

Let's say you have a huge condition like:

- Input A is ON.
- AND Input B is OFF.
- AND the motor isn't already running.
- AND a timer has expired.
- AND another condition from 5 rungs back is true.

😵 Trying to cram that into one rung? Good luck debugging that in an exam or real system.

Instead:

1. **Break it into smaller parts.**
2. Store partial results in M bits (M100, M101, etc.)
3. Use those M bits as contacts in other rungs.

Now your logic is **clean**, **modular**, and **easy to follow**.

🔁 Example in Mitsubishi:

```
--|X001|------|X002|------------(M100)  ; Store condition 1

--|M100|------|X003|------------(Y001)  ; Use that condition to drive an output
```

This lets you "build logic in layers" — just like you do with if statements or functions in Python or C.

🧠 Why Use M-Bits?

- ✅ To **store results** from previous logic.
- ✅ To **simplify complex rungs.**
- ✅ To **add memory** to your logic without using OTL/OTU.
- ✅ To create **modular, reusable chunks** of control.
- ✅ To make your program **easy to debug** and maintain.

It's the difference between spaghetti logic and **engineered control flow**.

🧠 Pro Tip for Mitsubishi Folks:

- Inputs → X (e.g., X001)
- Outputs → Y (e.g., Y001)
- Internal relays → M (e.g., M100)
- Timers/Counters → T, C
- Constants/Data → D, K, etc.

Think of M-bits as your **scratchpad memory** — they hold your thoughts while you solve the puzzle. Once you master them, your ladder diagrams will go from basic to pro-level real quick.

This next chunk right here is **the juice**.

You're learning how to **build real-world PLC behavior**, not just light bulbs and motors — but structured, modular, and *safe* systems.

M-bits don't just store logic; they **organize and control entire machine states**.

Let's refactor this into something 🔥sharp, real-world, and easy to visualize.

⚙️ Practical Power: How to Use M-Bits Like a Pro

Flags, Intermediate Logic, and Master Control — The Real Deal

Internal memory (M) bits are not just for breaking long rungs — they're your **logic control tools**, the way you make your PLC **think clearly** and **act smartly**.

Let's walk through **three powerful ways** to use them:

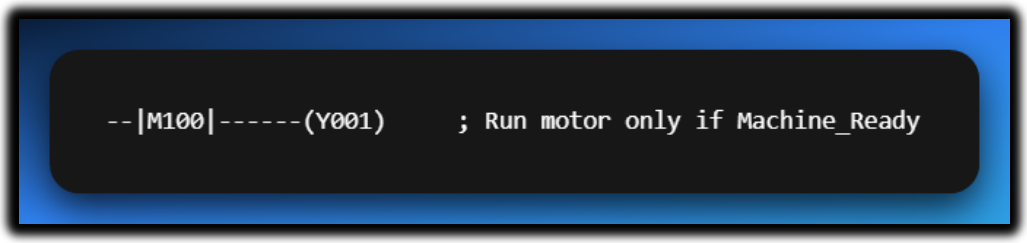
🏳️ 1. Flags — Status Indicators

Flags are M-bits that act like labels or checkpoints for your program.

📍 Examples:

- **M100 → Machine_Ready**
- **M101 → Fault_Active**
- **M102 → Sensor_Triggered**

Once a flag is ON, **anywhere else** in your program can check it like:

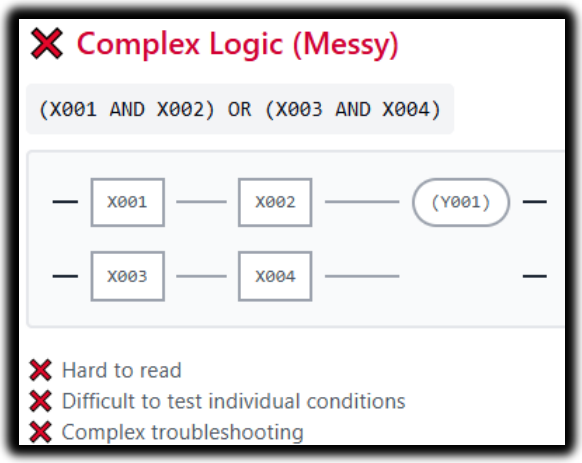


Think of it like this:

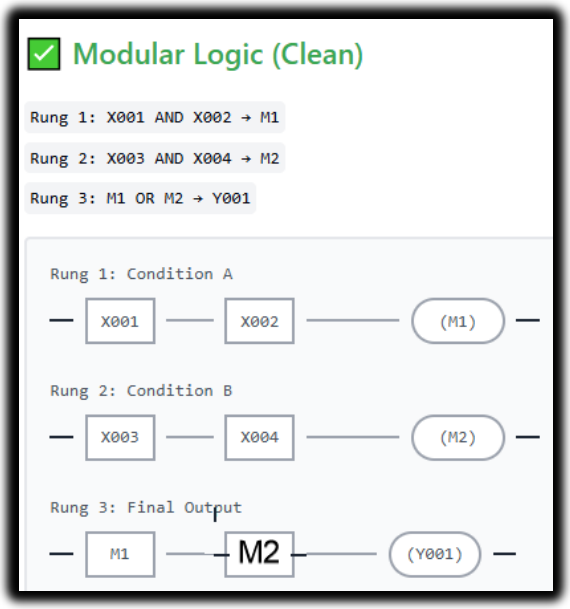
"Hey PLC, remember this condition happened? Cool — now let's use that info later."

🧱 2. Intermediate Logic — Break It Down to Build It Right

Complex logic can get ugly fast.
Instead of this mess:



Split it up like a smart coder:



✅ Easier to read
✅ Easier to test
✅ Easier to debug during an exam or live run

It's **ladder logic modularity** — just like clean code in Python or C.

🔧 3. Master Control Relay — The Big Switch

This is **crucial for safety and system structure**.

Let's say you've got a whole section of outputs — motors, valves, alarms — and you want to **disable them all instantly** when something goes wrong or the operator hits STOP.

🔧 Create a master relay like M200 (e.g., System_ON)

Then put that M200 contact in every rung:

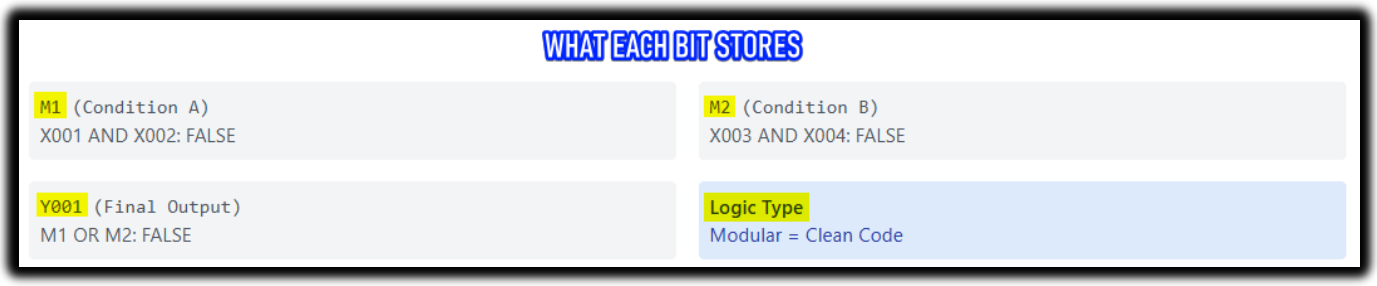
```
--|M200|--|X010|----------(Y001)   ; Output only if system is ON
--|M200|--|X011|----------(Y002)
```

Control M200 with your start/stop logic:

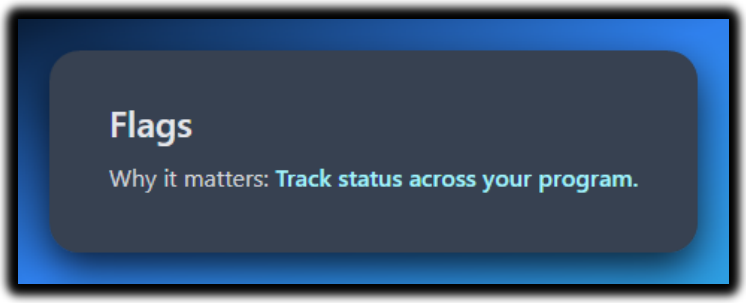
```
--|/|----| |----------------(M200)   ; NC Stop + NO Start = Enable System_ON
 Stop     Start
```

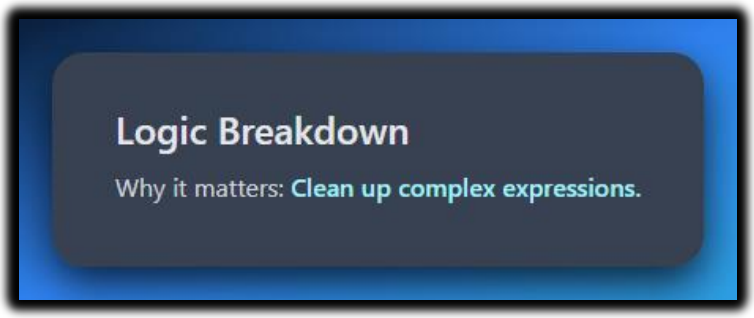
Now pressing STOP **cuts off everything**, like a master switch.

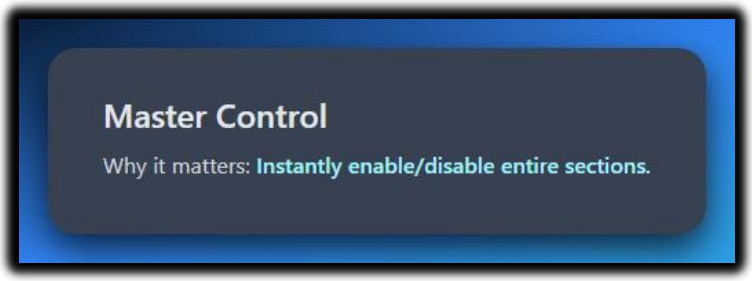
✅ Safe.
✅ Predictable.
✅ Easier to read
✅ Easier to test (M1, M2 individually).
✅ Easier to debug during exams/live runs.



Internal M-Bits: Structured Control







Internal memory is what takes your program from "works" to **works well** — clean, scalable, and *factory-ready*.

➞ *Does your PLC remember... or forget everything the moment power dies?*

Let's break this down so it **sticks like glue**, and you **never misuse a memory bit** again.

🧠 Retentive vs Non-Retentive M-Bits

What Happens When the Power Goes Out?

Not all M-bits are created equal. Some forget like goldfish 🐠... others remember like elephants 🐘.

And you, the programmer, decide which behavior you want.

🧊 1. Non-Retentive M-Bits — The Forgetful Kind

Most M-bits (by default) are **volatile** — meaning:

- If the PLC **loses power** or is switched to **PROGRAM mode**
- These bits **reset to OFF (0)**
- Everything they "remembered" is gone

✅ Good for:

- Temporary flags
- Intermediate logic
- States that should *always reset* on power-up

📍Example:

*M100: **"Start_Pressed"** → Only relevant during runtime.*

You *want* this to reset so no ghost input causes logic to fire when power comes back.

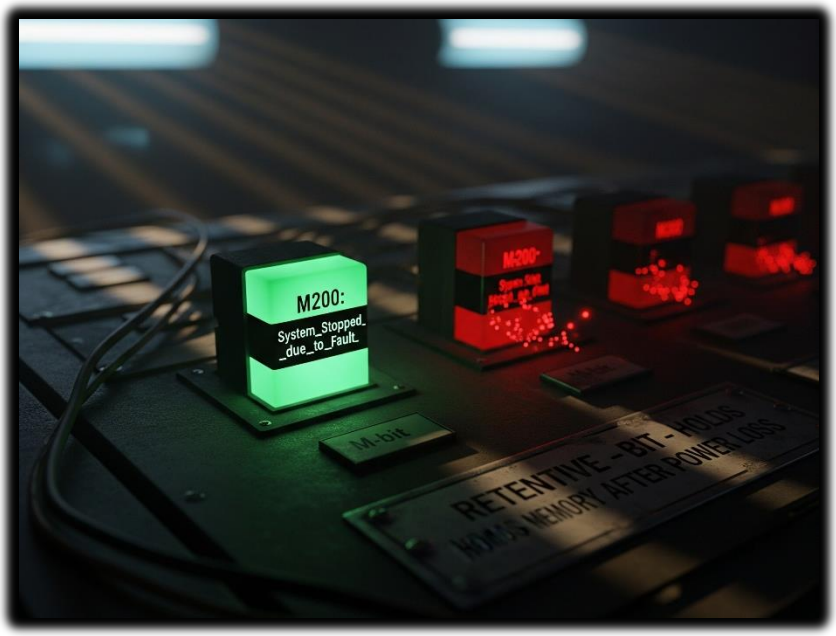
🧲 2. Retentive M-Bits — The Memory Keepers

Some M-bits are **retentive**. These are backed by internal battery power or stored in a persistent way inside the PLC.

- They **remember their state** even if:
  - ➢ Power goes out.
  - ➢ PLC is restarted.
  - ➢ Scan cycle is interrupted.

✅ Use them when you **must preserve state**. Example:

*M200: **"System_Stopped_Due_to_Fault"** → You don't want this cleared, until operator resets it.*



⚠️ Why It Matters

Scenario A:

A batch mixing system is halfway through step 3 when the power dies.

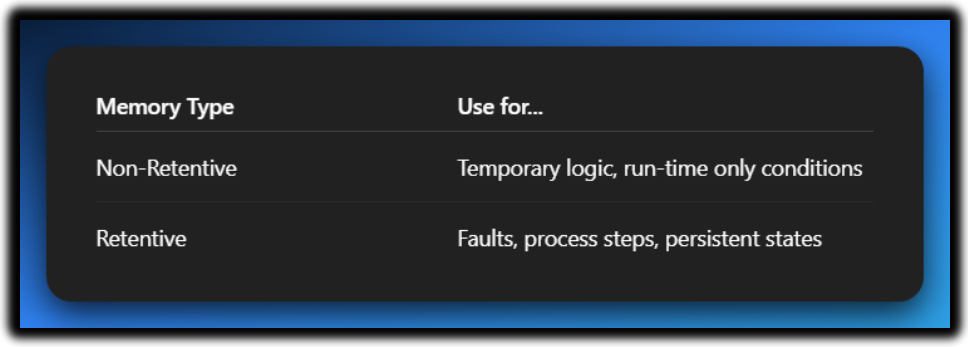
- ❌ If all your M-bits were non-retentive → the PLC boots up and has no clue where it left off
- ✅ If you saved step progress in retentive bits → the system resumes like nothing happened

Scenario B:

A fault occurred. You want the operator to see it after reboot.

- ❌ Non-retentive = fault flag disappears
- ✅ Retentive = fault flag still ON → Operator sees and clears it

🔐 Design Rule of Thumb



Memory Type	Use for...
Non-Retentive	Temporary logic, run-time only conditions
Retentive	Faults, process steps, persistent states

Mixing them up = recipe for **bad logic** or **unsafe resumes**.

📌 In Mitsubishi PLCs:

You'll often configure **ranges** of M-bits as retentive in the **PLC parameters**.

For example:

- **M0–M499 → Non-retentive.**
- **M500–M999 → Retentive.**

Check your software (like GX Works) to define or view this split.

🧠 Final Word:

When power comes back, your program should either:

- **Restart cleanly** like a fresh boot.
- Or **pick up where it left off**, like nothing happened.

Your **choice of retentive vs non-retentive bits** is what makes that happen.

🧩 Real-World Uses of M-Bits

Practical Logic with Flags, Intermediates, and Master Control

M-bits aren't just "extra space" — they're your secret tools for organizing, controlling, and simplifying your entire program.

Here's **how pros actually use them** in everyday logic:

Memory Type	Behavior on Power Loss	Best For
🧊 Non-Retentive (Volatile)	Resets to **OFF**	Temporary logic, one-shots, momentary triggers
🧠 Retentive (Persistent)	**Remembers** its state (ON/OFF)	Critical states, alarms, batch step memory

✅ 1. Flags / Status Indicators

Flags = M-bits that mark something as ON/OFF (TRUE/FALSE) in your program.

Think of them as **checkpoints**:

- **M100 → Machine_Ready**
- **M101 → Fault_Active**
- **M102 → Sensor_Clear**

Instead of checking multiple raw inputs again and again, just check the flag:

```
--|M100|-------------|Y001|     ; Start motor only when system is ready
```

💡 **Flags simplify conditions and make logic readable**.

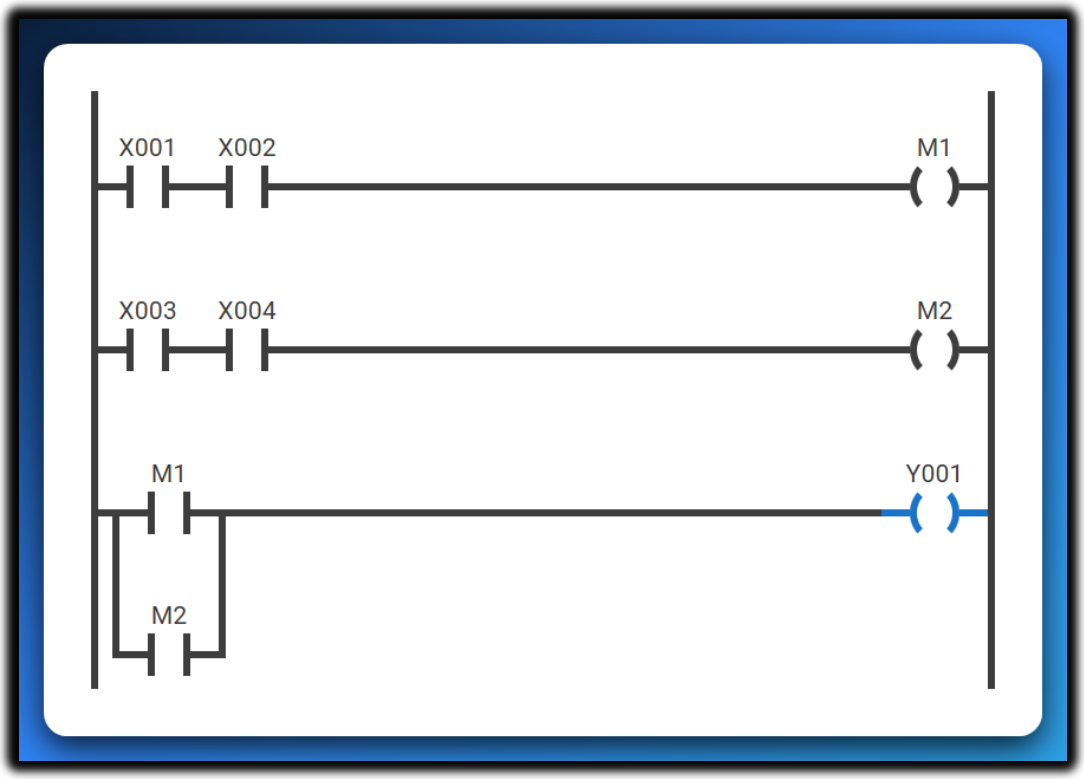
🧠 2. Intermediate Logic Storage

Ever seen an ugly rung like this?

```
(X001 AND X002) OR (X003 AND X004)
```

Messy and hard to debug, right?

💡 Solution: Break it up using M-bits:



✅ Cleaner
✅ Easier to troubleshoot
✅ You'll thank yourself during exams

🚨 3. Master Control Bit (MCB)

Use an M-bit like a **main switch** to enable or disable entire sections of the program.

Example: M200 → System_ON (controlled by start/stop buttons)

Then use it like this:

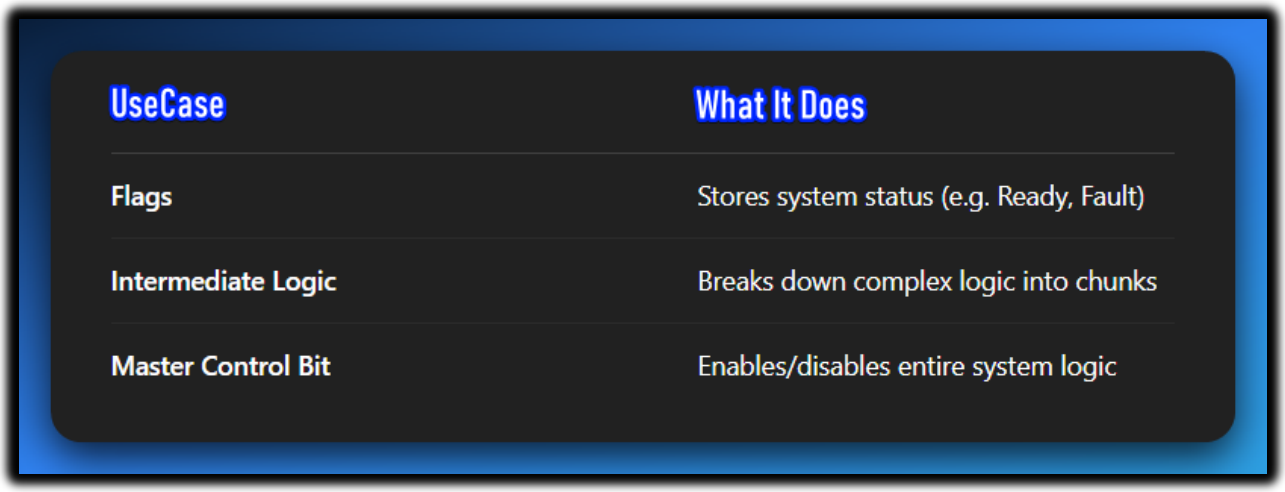
```
--|M200|--|X010|------------(Y001)     ; Motor only works if system is ON

--|M200|--|X011|------------(Y002)     ; Valve only opens if system is ON
```

If the STOP button breaks the latch and M200 turns OFF, **everything shuts down** — instantly.

✅ Safety-compliant.
✅ Easy to organize.
✅ Clean sequencing control.

🔑 Summary Table



UseCase	What It Does
Flags	Stores system status (e.g. Ready, Fault)
Intermediate Logic	Breaks down complex logic into chunks
Master Control Bit	Enables/disables entire system logic

M-bits = **modular design, safety control, and clean programming.**
Every decent PLC project uses these like glue holding everything together.

⏱ PLC TIMERS DEMYSTIFIED: TIME-BASED CONTROL MADE SIMPLE

Chapter IV: Mastering Timing Logic

We're now in the deep waters, let's go! 💥
In the real world, **not everything happens instantly**.
Need a delay before a conveyor starts?
Want a light to stay on for 10 seconds?
Or pause between steps in a process?

That's what **timers** are for.
They let you go **beyond simple ON/OFF** logic — adding time-based control to your ladder program.

🚀 What Are Timers in PLC?

Timers are **built-in instructions** that count time when conditions are met.
They trigger events **after** a specified time delay.

Examples:

- Keep a fan running for 5s after machine stops
- Flash a light every 0.5s
- Add a delay between solenoid activations

🧩 All PLC Timers Have These Core Parts

Let's break this down in plain English: PLC Timer parameters

Parameter	What It Does	Example
Timer Tag/Address	Unique name/ID for your timer (varies by brand)	*T0 (Mitsubishi), T4:0 (A-B)*
Time Base	The unit of time used by the timer	*1s, 0.1s, or 0.01s per tick*
Preset Value (PRE)	How long the timer should wait before finishing (in units of Time Base)	*If base = 1s and PRE = 10 → 10s*
Accumulated Value (ACC)	Current time that has passed (auto-increments while timing)	*If ACC = 3, timer's been ON for 3s*
Enable Bit (EN)	TRUE when the timer rung is ON (just means "timer is allowed to run")	*TRUE = it's powered*
Timer Timing Bit (TT)	TRUE while the timer is still counting (ACC < PRE)	*Think: "It's ticking..."*
Done Bit (DN)	Turns TRUE when ACC reaches PRE	*"We're done timing!"*

PLC Timer Parameters: Quick Notes

Timer Tag/Address

This is the unique name or ID that identifies a specific timer in your program.

Time Base

This defines the unit of time, like seconds or milliseconds, for each "tick" of the timer.

Preset Value (PRE)

This is the total amount of time, in Time Base units, that the timer must count up to.

Accumulated Value (ACC)

This tracks the current time that has passed since the timer started running.

Enable Bit (EN)

This bit is TRUE when the timer's circuit is powered, allowing it to start counting.

Timer Timing Bit (TT)

This bit is TRUE only while the timer is actively counting, before it reaches its Preset Value.

Done Bit (DN)

This bit turns TRUE as soon as the timer's Accumulated Value is equal to or greater than the Preset Value.

🔁 Real-Life Analogy

Let's say you're boiling an egg for 5 minutes:

- **EN** = you turned on the stove.
- **TT** = it's still boiling.
- **DN** = timer rings after 5 mins (egg done!).
- **ACC** = shows how long the egg has been boiling.
- **PRE** = your 5-minute goal.
- **Time Base** = counting in seconds or tenths.

⚙️ Use Case Examples

Light turns off 10s after button press

Use ON-delay timer with PRE = 10

The ON-delay timer starts counting when the button is pressed, and after 10 seconds, it activates the output to turn the light off.

Motor starts 3s after system power-up

Delay with PRE = 3 before output

A timer is initiated on system power-up and counts for 3 seconds before activating the output that starts the motor.

Flash alarm light every second

Toggle logic + timer reset loop

A timer with a Preset of 1 second is used in a loop that resets itself, and its timing bit is used to toggle the alarm light on and off.

Timers *control over when things happen, not just what happens. They're the heartbeat of delays, sequences, and safety timeouts.*

⏳ On-Delay Timer (TON): Wait Before You Act

Think: *"I'll wait for X seconds, then I'll turn ON"*

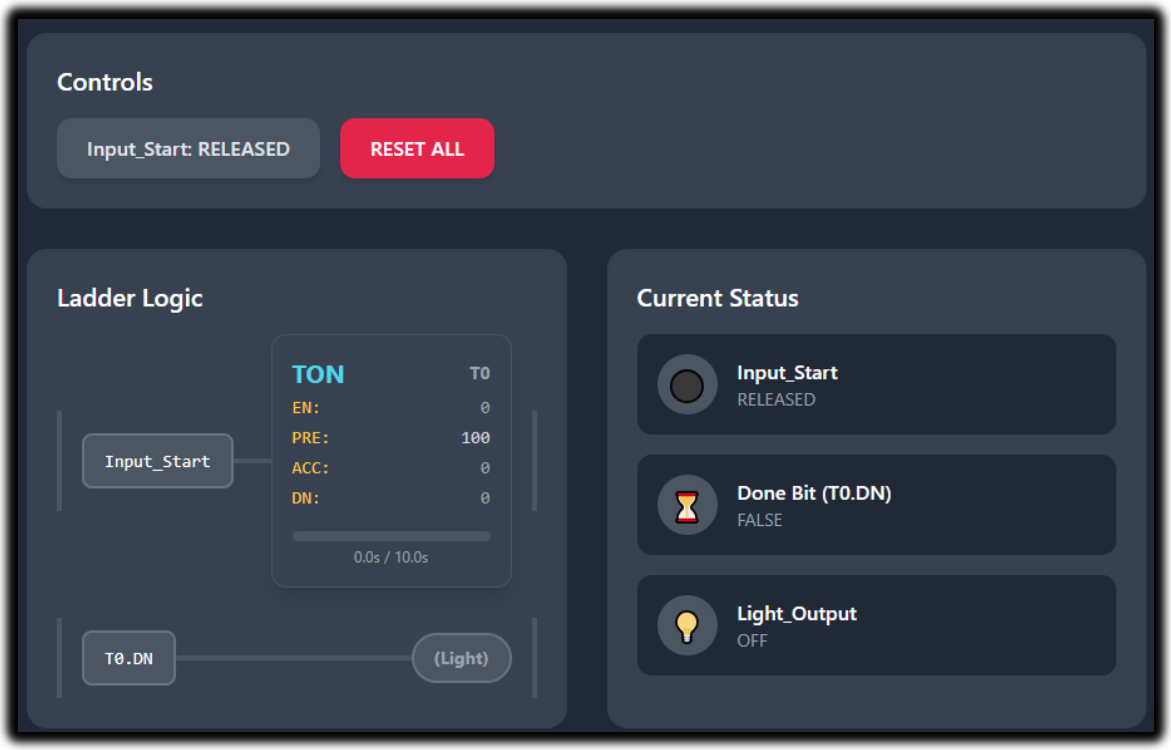
🔍 What Does TON Do?

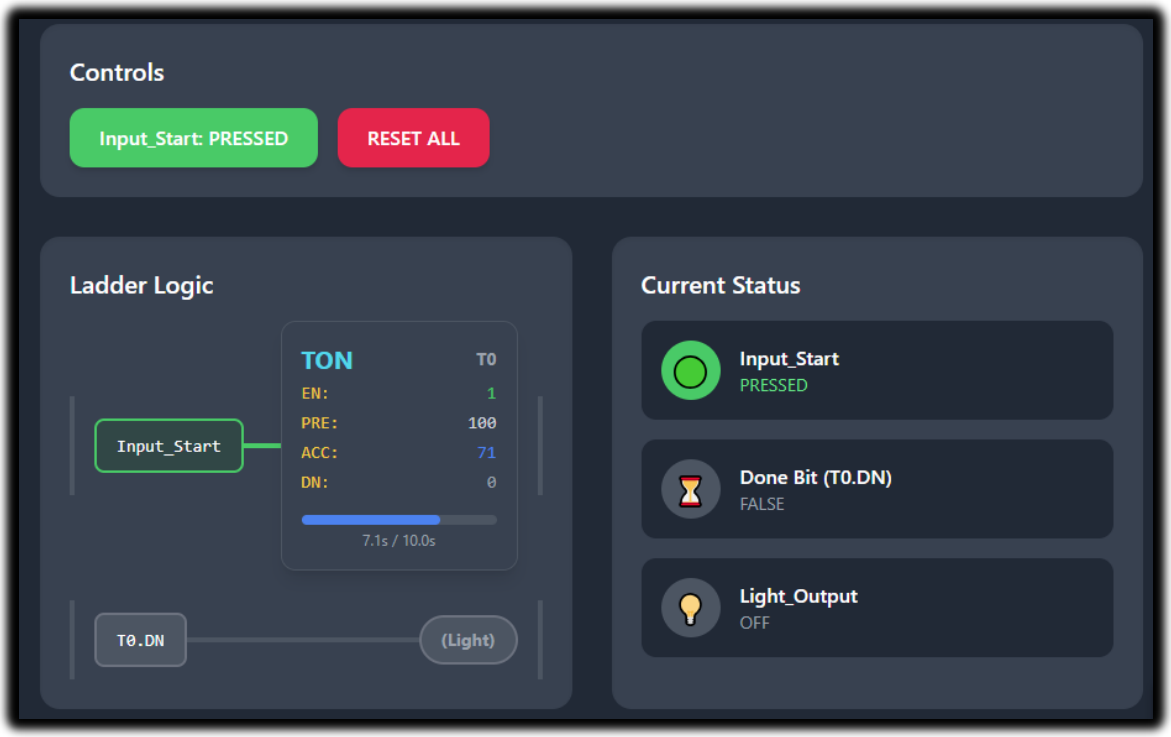
An **On-Delay Timer** delays its output from turning ON until a preset time passes — *but only if the input stays ON the whole time*.

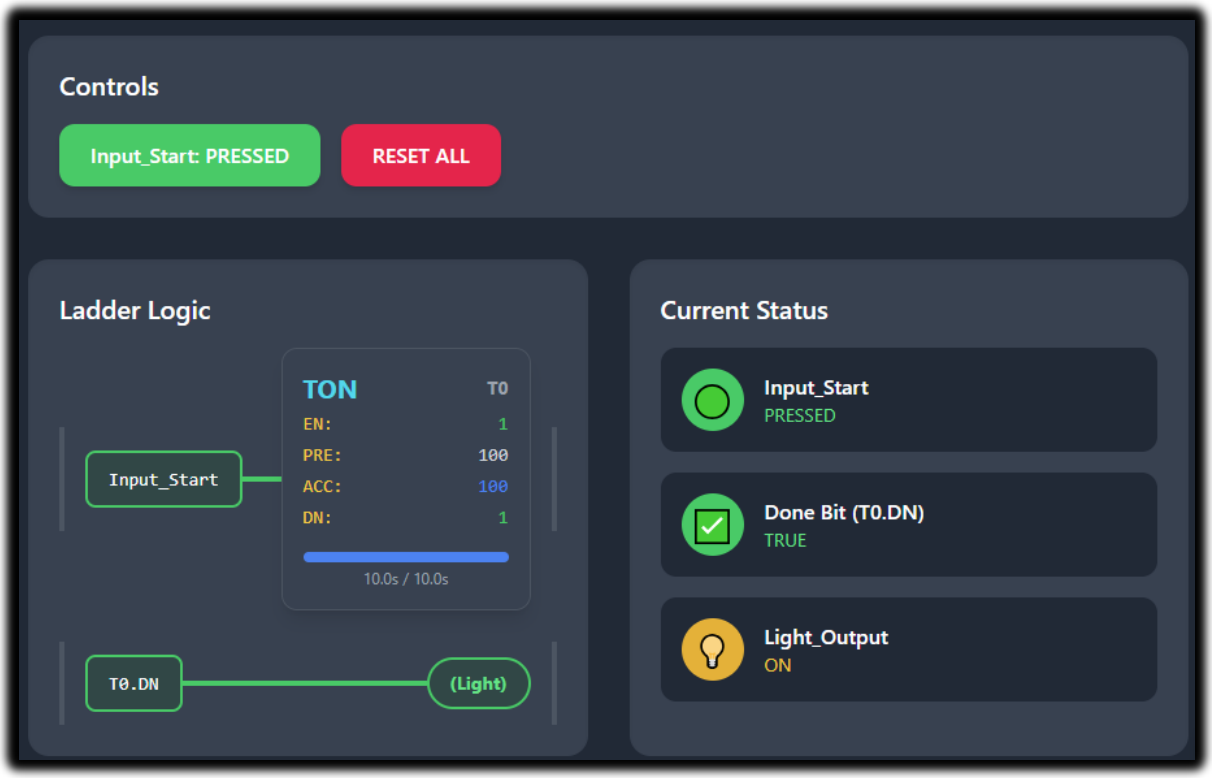
So, it's basically saying:

"If you're serious about this signal, hold it steady. No quick taps."

⚙ TON Timer Behavior Breakdown







When the input rung turns **TRUE**, the timer starts counting up, and the accumulated value (ACC) increases.

If the input remains **TRUE** until the accumulated value (ACC) equals the preset value (PRE), the **Done (DN)** bit turns **TRUE**, activating the output.

If the input goes **FALSE** before the accumulated value (ACC) reaches the preset (PRE), the ACC resets to **0** and the DN bit remains **FALSE**.

If the input goes **FALSE** after the DN bit is **TRUE**, the ACC resets and the DN bit turns **FALSE** again.

🧠 In short:

- **It waits** to turn ON.
- **It resets instantly** if input is lost.
- **No memory** — doesn't hold state once input dies.

🛠 Real-World Uses for TON:

- Safety delays (e.g., wait 5s before re-starting)
- Signal debouncing (ignore fast button taps)
- Sequential process delays (e.g., wait before opening next valve)

🔁 TON = Patience Timer

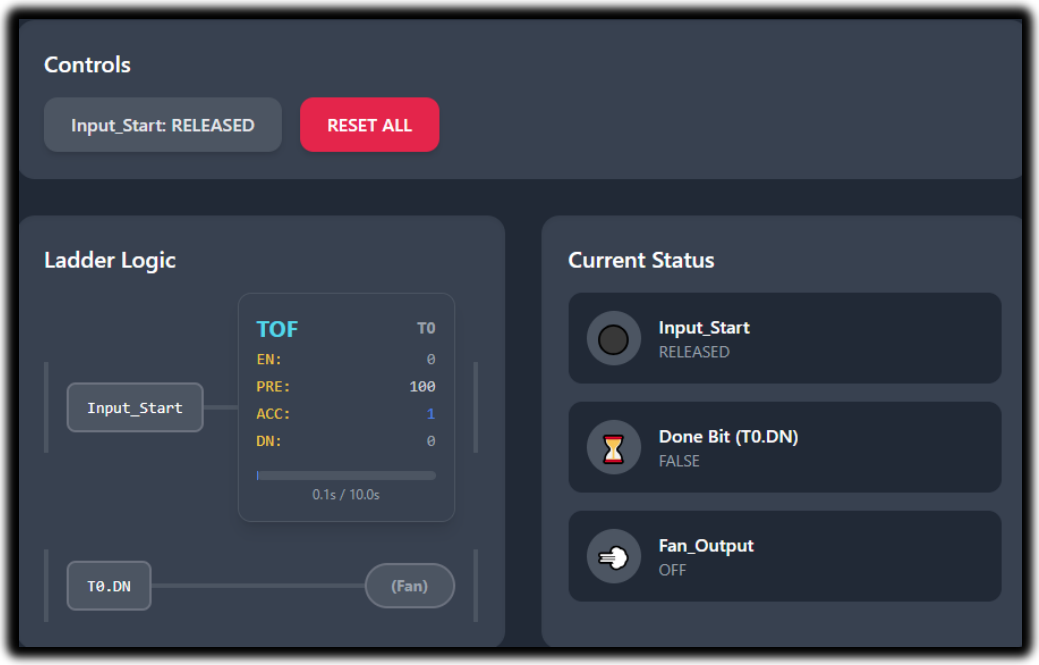
- It **doesn't rush** into action.
- It needs **consistent input.**
- It **resets instantly** if trust is broken.

🧯 Off-Delay Timer (TOF): Stay ON a Bit Longer

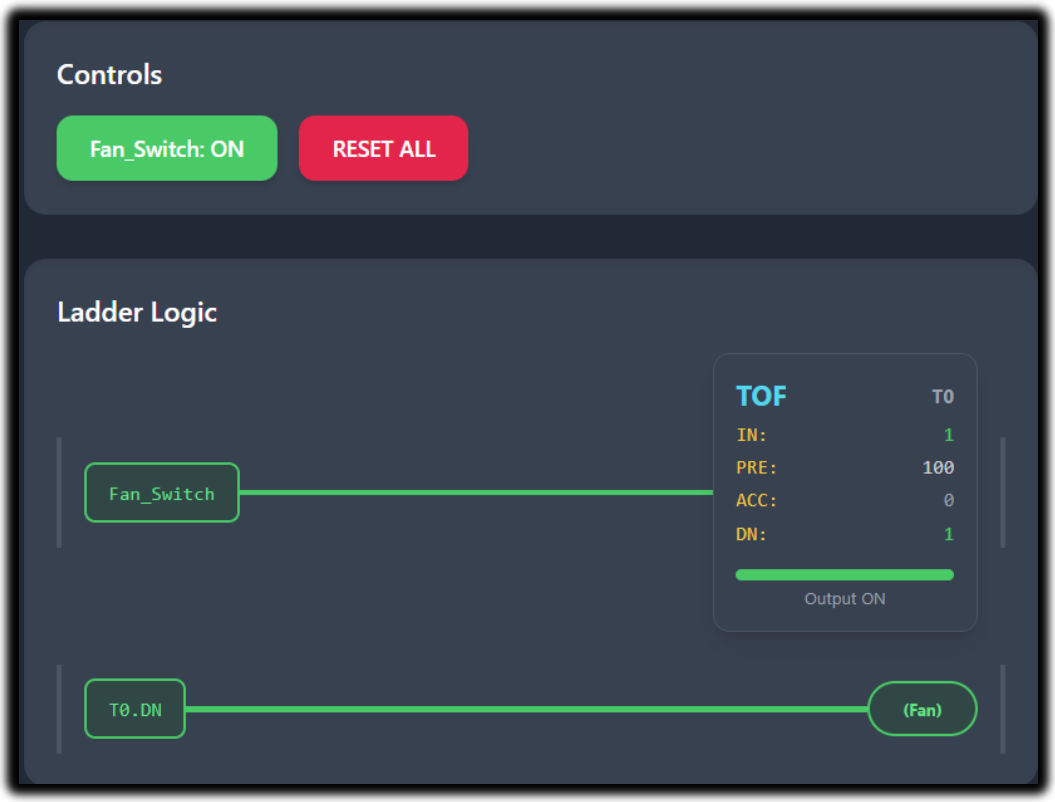
Think: *"I'll stay ON for X seconds after input goes OFF."*

⚙️ TOF Behavior Breakdown

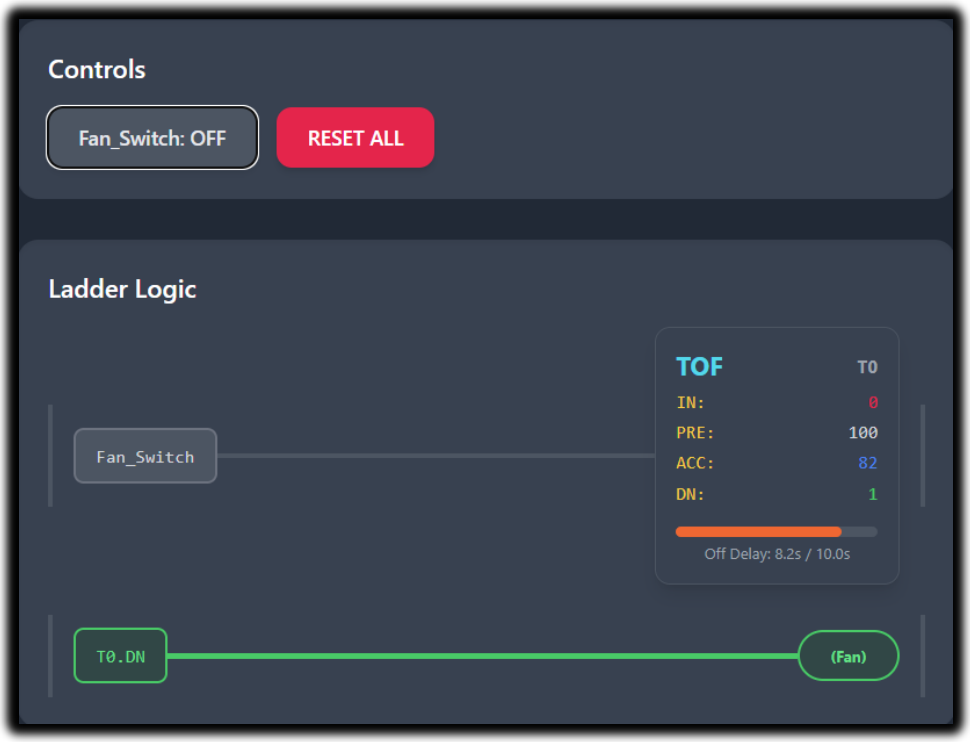
Normal state.



Input pressed:



Input released and becomes False counts, then goes back to state 1:



When the input rung turns **TRUE**, the **DN (Done)** bit turns **TRUE** immediately.

While the input remains **TRUE**, the timer stays idle and the accumulated value (**ACC**) remains at **0**.

When the input turns **FALSE**, the timer begins counting and the **ACC** value starts increasing.

When the **ACC** reaches the preset value (**PRE**), the **DN** bit turns **FALSE**, which causes the output to turn **OFF**.

If the input turns **TRUE** again during the countdown, the **ACC** resets to **0**, the **DN** bit stays **TRUE**, and the delay restarts.

🧠 In short:

- **Output turns ON immediately.**
- **Stays ON** even after input is OFF — but only for the preset time.
- **Cancels countdown** if input turns ON again.

🛠 Real-World Uses for TOF:

- Cooling fans (e.g., after welding, drying, or heating)
- Ventilation after motor stops
- Delay-off buzzers or alarms
- Lights that stay ON for a few seconds after leaving a room

💥 TOF = "Let me finish my job before I shut off"

- It's **immediate on**, **delayed off**
- Adds **grace period** before shutting down
- Helps prevent **premature shutoffs**

You've now mastered both ends of the timer spectrum:

- **TON** = waits to turn ON
- **TOF** = waits to turn OFF