

# PROGRAMMABLE LOGIC CONTROLLERS

## ✓ 1. PLC Fundamentals (Core Concepts)

- **What is a PLC?**
  - Definition & role in automation (why PLCs?) ✓
  - Advantages over hard-wired relay logic. ✓
- **PLC Architecture**
  - CPU (control + scan cycle) ✓
  - Memory (program memory, data table, I/O image table) ✓
  - Input modules (digital/analog) ✓
  - Output modules (digital/analog) ✓
  - Power supply ✓
- **PLC Scan Cycle**
  - Input scan → Program execution → Output update ✓
  - Relation to continuous loops in software ✓
- **PLC Programming Languages (overview)**
  - Ladder Diagram (main focus) ✓
  - Mention of others: FBD, ST, IL, SFC (just awareness) ✓

## ✓ 2. Ladder Logic Essentials (Main Exam Meat)

- **Contacts and Coils**

- Normally Open (or XIC) ✓
- Normally Closed (or XIO) ✓
- Positive Transition Sensing Contact ✓
- Negative Transition Sensing Contact ✓
- Output Coil / Output Energize (OTE) ✓
- Output Latch (OTL) and Output Unlatch (OTU) ✓



Tom Jenkins  
May 14, 2003

#5

Forget XIO and XIC. That is specifically and only Allen Bradley nomenclature. Any other ladder logic I can think of uses NO and NC.

XIO is A-B speak for NC and XIC is NO.

Source: [plctalk.net](http://plctalk.net)

- **Logical Operations**

- AND (series contacts) ✓
- OR (parallel contacts) ✓
- NOT (using NC contacts) ✓

- **Internal Memory Bits**

- How to use internal relays (B3, M bits) for intermediate logic ✗

- **Basic Programs**

- Start/Stop motor latch circuit ✗
- Simple interlock (two conditions must be true) ✗

## 3. Timers (Time-Based Control)

- On-Delay Timer (TON) ✗
- Off-Delay Timer (TOF) ✗
- Retentive Timer (RTO) ✗
- Timer Reset (RES) ✗  
*(Know their use-cases and how to wire them in ladder logic)*

## 4. Counters (Count-Based Control)

- Count Up (CTU) ✗
- Count Down (CTD) ✗
- Counter Reset (RES) ✗  
*(Be able to create simple counting circuits)*

## 5. Data Handling & Math

- Comparison Instructions
  - Greater Than (GRT) ✗
  - Less Than (LES) ✗
  - Equal (EQU) ✗
- Math Instructions
  - ADD, SUB, MUL, DIV ✗
  - Scaling analog inputs (basic understanding) ✗
- Move Instruction (MOV)
  - Copy data between memory locations ✗

## 6. Program Structuring Techniques

- Sequencing (step-by-step control, e.g. washing machine cycle) ✗
- Subroutines / JSR (jump to subroutine) ✗
- State machines (using memory bits to track steps) ✗

## 7. Troubleshooting & Simulation

- **Diagnostics**
  - Going online with PLC software 
  - Forcing I/O (know concept) 
  - Monitoring and interpreting real-time values 
- **Common Faults-**
  - Inputs not wired or misaddressed 
  - Timers/counters not resetting as expected 
- **Simulation Practice** (*you already reached this in class*)
  - Start/Stop motor latch 
  - Traffic light (timers) 
  - Bottle filling/counting (counters) 
  - Door interlock (AND/OR logic) 

## Your Study Priority

If time is tight, focus in this order:

**Ladder Logic Basics → Timers → Counters → Comparisons/Math → Structuring → Troubleshooting/Simulation.**

## Practical Tip

-  Spend most of your time in a simulator (OpenPLC or LogixPro).
-  Build small circuits for each topic.
-  Write tiny notes after each topic with a quick ladder snippet.

## WHAT IS A PLC?

A **PLC** is a ruggedized, solid-state industrial computer designed to run 24/7 in harsh environments. It **stores and executes control instructions** in real time to monitor inputs, make logic decisions, and control outputs. Think of it as the **brain of industrial automation**, replacing bulky relay systems with reliable digital control.

 It *takes in signals* from sensors and switches, processes them according to a stored program, and then sends commands to outputs like motors, valves, or lights.

## Typical Instructions in a PLC Program

When programming a PLC, you're basically giving it a toolbox of standard instructions. These fall into six core categories:

1.  **Sequencing** – Controls step-by-step process flow (e.g., a car wash sequence: rinse → soap → scrub → dry).
2.  **Timing** – Timers (ON-delay, OFF-delay, retentive) introduce precise delays or durations (e.g., keep a valve open for 5 seconds).
3.  **Counting** – Counters (up, down, combined with timers) track events, products, or cycles (e.g., count 100 bottles, then trigger packaging).
4.  **Arithmetic** – Math operations (add, subtract, multiply, divide, floating-point) for scaling sensor data, setpoints, or calculations.
5.  **Data Manipulation** – Bit/word operations (masking, shifting, rotating, BCD conversions) to move, compare, or reformat data.
6.  **Communication** – Lets PLCs exchange data with other PLCs, HMIs, or SCADA systems via protocols like Modbus, Profibus, or Ethernet/IP.

## 📌 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:**

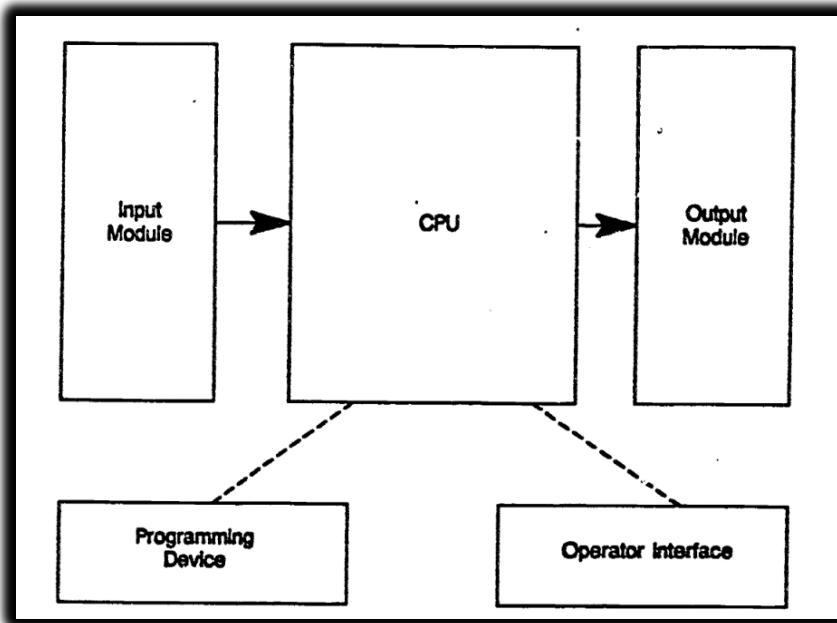
When handling chemicals, you need perfect timing, flow control, and safety shutdowns — PLCs are built for that level of reliability.

## **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.

## **Main Parts of a PLC**

PLC has two main parts: The **CPU** which has the memory and the **I/O subsystem**.



### 1) CPU (Central Processing Unit)

- **The brain of the PLC.**
- Contains the **processor + memory**.
- Reads inputs, executes your program logic, and decides what outputs should do.
- Memory stores the control program, I/O status, and working data e.g. flags, registers.

### 2) I/O Subsystem (Input & Output Modules)

- **Input Module** → Brings in signals from sensors, switches, etc. Converts them into digital data the CPU can understand.
- **Output Module** → Takes CPU decisions and drives actuators like motors, lights, or valves. Converts low-level CPU signals into usable power for devices.

## Supporting Components of a PLC System

Besides the core **CPU + I/O subsystem**, PLCs often use external devices that help engineers and operators interact with the system:

### 1) Programming Device

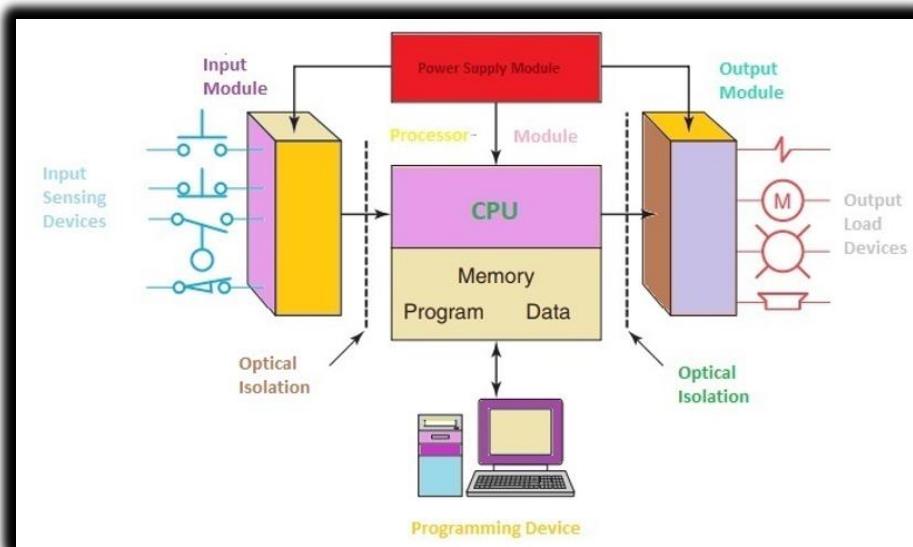
- Laptop, handheld programmer, or software tool.
- Used to **create, edit, and download logic** into the PLC.
- Without this, you can't even "teach" the PLC what to do.

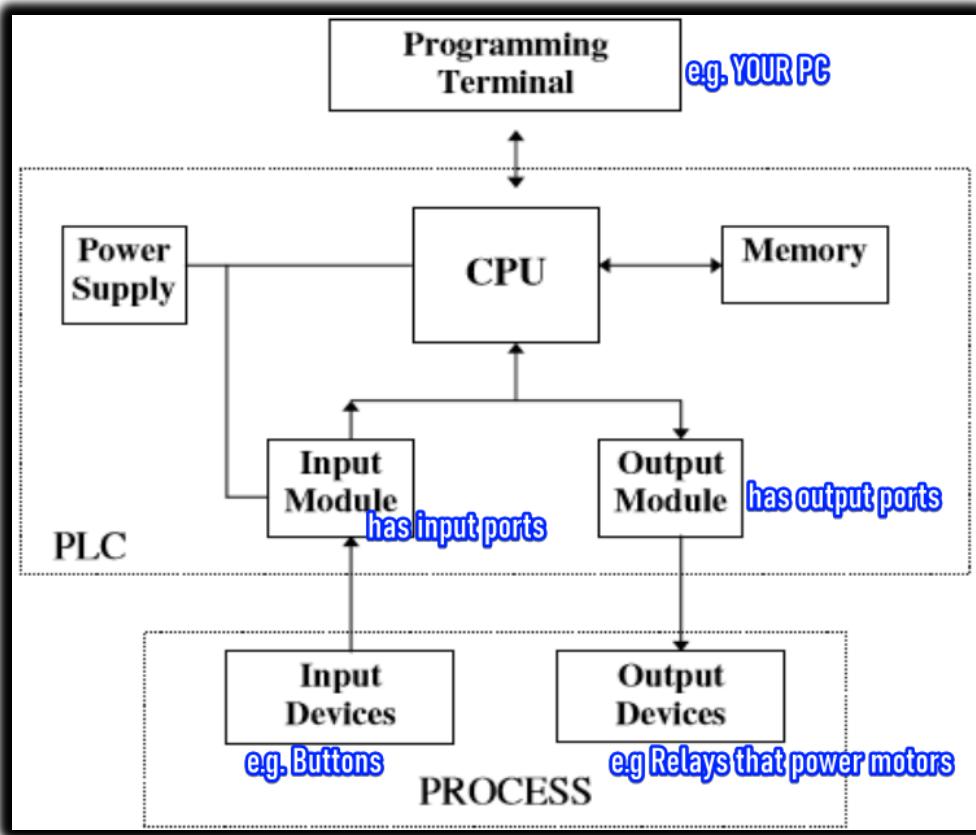
### 2) Operator Interface (HMI – Human Machine Interface)

- Acts as the **dashboard** for the operator.
- Displays process info (e.g., temperatures, alarms).
- Allows parameter changes (e.g., speed, setpoints).
- Modern HMIs are usually **touchscreens** with graphics.

### 3) Terminals (Legacy Interfaces)

- **CRT Terminal** → Old-school monitor + keyboard setup. Worked like a command-line interface for real-time PLC interaction.
- **Printer Terminal** → Produced hardcopy printouts of programs, error logs, or reports for documentation.
- **Hardcopy** → Physical records on paper, important for backups or compliance in older systems.





## Process 1: Initializing the PLC System

You walk in, flip the switch, and the PLC comes to life. The power supply juices up the CPU and I/O modules. It's on but in **Programming Mode**, ready for you to tell it what to do.



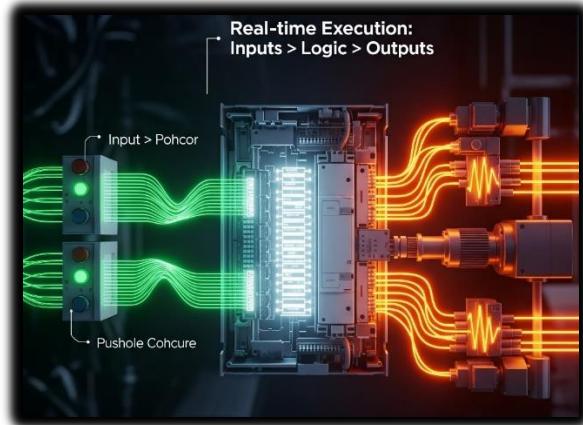
## Process: Programming and Downloading Logic

Using your computer, you write the program in a language like **Ladder Logic**. Once you're happy with it, you download the code to the PLC's memory. This gives the PLC its new instructions and capabilities. It's like teaching it a new trick. 🧠



## Process: Transition to Run Mode and Program Execution

When the PLC switches to **Run Mode**, it starts its endless scan cycle. It first **reads all the inputs**, then **runs the program's logic**, and finally, **updates all the outputs**. It's like a constant loop, making sure everything is working as it should. 🔍



## Process: Real-World Actuation via Output Chain

- The PLC CPU decides an action, like starting a motor.
- It sends a small, low-power signal to the Output Module.
- The Output Module converts this weak digital signal into a stronger electrical one.
- This stronger signal is sent to an external relay.
- The relay acts like a security guard for the motor.
- When energized, it closes its heavy-duty contacts.
- This allows high-power electricity to flow to the motor.
- The motor turns on and does its job.
- Small PLC command → big real-world action. 💥

### Key real-world use cases:

Controlling machines on a factory assembly line.



Managing amusement ride rollercoasters as they twist and turn.



Automating food-processing machinery that mix ingredients for your favorite snack.



## Why PLCs instead of normal PCs?

### Designed to Handle Digital & Analog I/O:

PLCs have built-in I/O modules to directly handle digital and analog signals from industrial devices, while regular PCs need extra hardware and software, making them slower and harder to use for real-time control.

### Survive Extreme Temperatures:

PLCs are rugged and built to survive extreme temperatures, while regular PCs can't handle the heat—or the cold—of harsh industrial environments.

### Immune to Electrical Noise:

PLCs are built to resist electrical noise from heavy machinery, while regular PCs can glitch or fail in such interference-heavy environments.

### Resist Vibration and Impact:

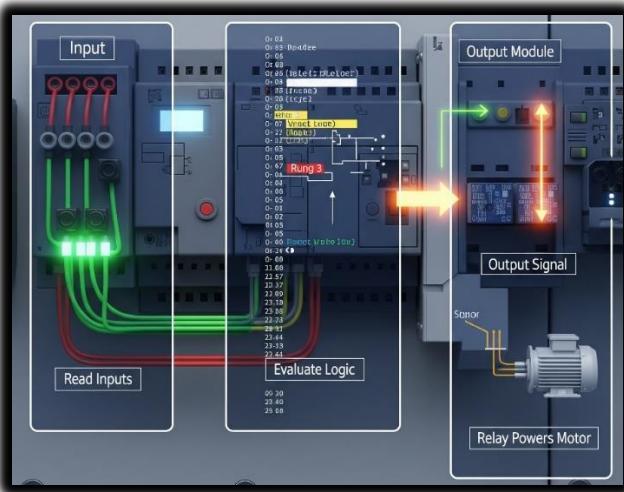
PLCs are designed with sturdy, shock-resistant casings and components, enabling them to withstand vibrations, impacts, and rough handling while regular PCs are too fragile and prone to hardware failures under physical stress.

## ⚡ CPU Operating Modes

We said that the **CPU** is the brain of the PLC. It has a **microprocessor** that handles all the calculations, **memory chips** for storing the program and data, and **control circuits** that let it talk to everything else, like the I/O modules and other devices. 🧠

The PLC's CPU has distinct operating modes:

- **Programming Mode/Offline Mode:** The PLC is essentially stopped. Its CPU is not executing any instructions that control machinery, so it's in a safe state ready to accept new instructions. Control logic can be uploaded without the output being affected coz the PLC is off.
- **Run Mode/Online Mode:** The PLC's CPU continuously executes the program stored in its memory. It performs a high-speed scan cycle, reading inputs, processing logic, and updating outputs to control the connected machinery in real-time.



The **CPU** is the heart of the PLC. It handles **decision-making** and manages **user memory**, both of which are **fully controlled by the programmer i.e. you 🦸‍♂️**.

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

## 1. Decision-Making Section

The CPU is the brain of the PLC. Constantly *reads signals* from inputs like sensors, *compares* them to your preprogrammed instructions, and then *decides which outputs* to turn on or off.

Always checking inputs and making things happen 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

## Inputs and Outputs (The Senses – How the PLC Gathers Info):

This part of the PLC is constantly **listening and watching** for signals from the outside world.

**Input Field devices** gather real-world info (like temperature, position, pressure)

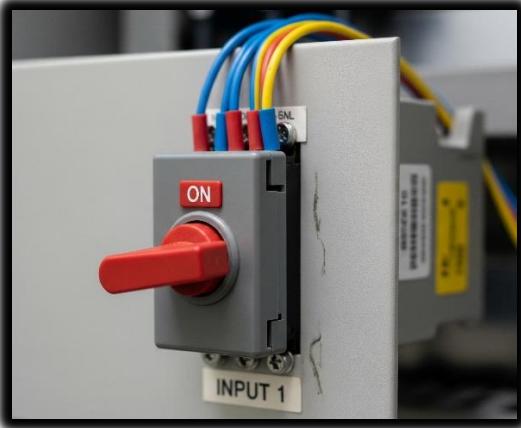
**Output Field devices** receive and act on commands to perform physical actions e.g. The PLC processes the signal from the proximity sensor and sends a command to a motor (an output device) to push the box off the conveyor.

**Sensors:** Sensors give the PLC *status updates*.

- A **proximity sensor** says, "Product is in place."
- A **temperature sensor** warns, "It's getting hot!"  
- A **magnetic proximity sensor** detects when a metallic object is nearby. 
- A **photoelectric sensor** sees if an object is present or absent by using a beam of light.
- A **limit switch** signals when a mechanical part has reached its full travel distance e.g. garage door hits a limit switch signal sent to stop motor meaning gate is fully open or fully closed. 

- A **pressure sensor** measures the force of a gas or liquid in a system e.g. hydraulic press. 
- A **flow sensor** checks if a liquid or gas is moving through a pipe. 
- A **level sensor** tells you how full a tank or container is. 
- An **encoder** tracks the rotation or position of a shaft. 
- A **vision sensor** uses a camera to inspect, identify, or locate an object. 
- An **ultrasonic sensor** detects objects using sound waves. 
- An **infrared sensor** senses heat or motion. 

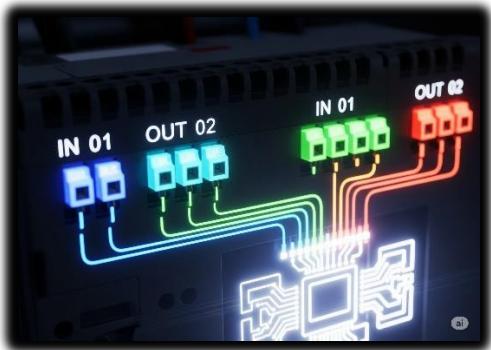
**Switches:** A switch is a device that completes or breaks an electrical circuit to send a signal to the **PLC**. It acts as a digital input, like a simple on/off button, telling the PLC to start or stop a process.



**I/O points** are the physical terminals on the PLC where you connect field devices like sensors and switches.

Each of these points is assigned a **unique memory address** (for example, X000), allowing the PLC's CPU to monitor its state.

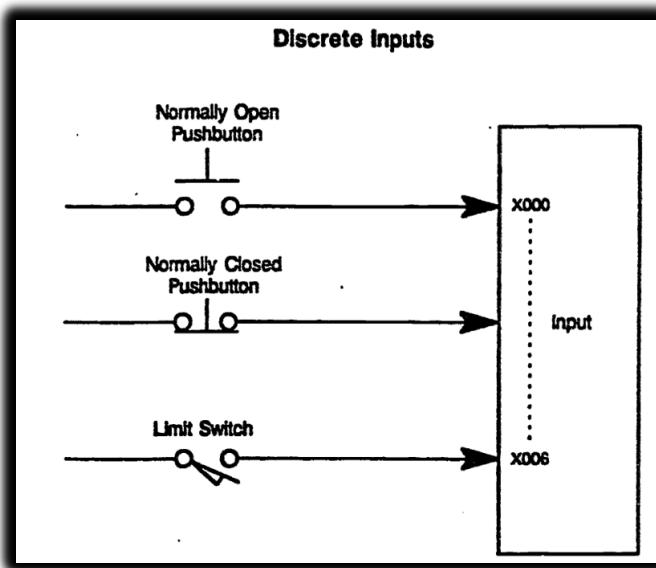
When a device is activated, the **corresponding bit in the PLC's memory flips**, changing its value from **0 to 1** to signal the change. 



## ❖ Discrete Inputs (a.k.a Digital Inputs)

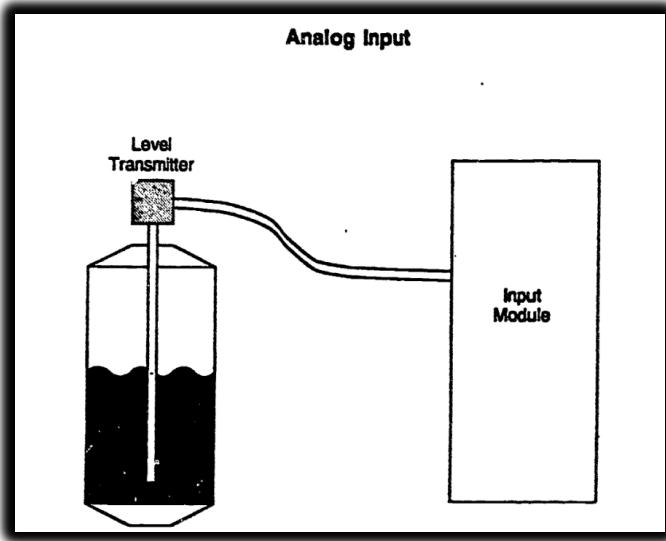
**Discrete inputs**, also known as digital inputs, are *signals sent from devices* that can only be in one of two states: **ON** or **OFF**, no in-between.

They are used by the PLC to know if a switch is pressed, a sensor detects an object, or a motor is running e.g.



- **Pushbuttons (Normally Open / Normally Closed):** Think of a "start" or "stop" button on a conveyor belt. You push it to either turn the machine on or off. 
- **Limit Switches:** These are like a tripwire for a machine's movement. For example, they tell a robotic arm that it has reached the end of its path. 
- **Float Switches:** Imagine a water tank. A float switch tells the PLC when the water level is too high or too low, so it knows when to turn a pump on or off. 
- **Toggle Switches:** These are just your basic light switches. You flip it on or off to send a constant signal to the PLC. 
- **Flow/Pressure Switches:** A flow switch might tell a PLC that water is flowing through a pipe, or a pressure switch could warn that the air pressure in a tank is too low. 
- **Foot Pedals / Safety Interlocks:** A foot pedal lets a worker control a machine hands-free, like a sewing machine. A safety interlock on a machine door prevents it from running if the door is open. 
- **Proximity Switches:** These are like magic eyes for machines. A proximity switch on an assembly line detects a metal soda can as it passes by, triggering the next step in the process. 

## **Analog Input**

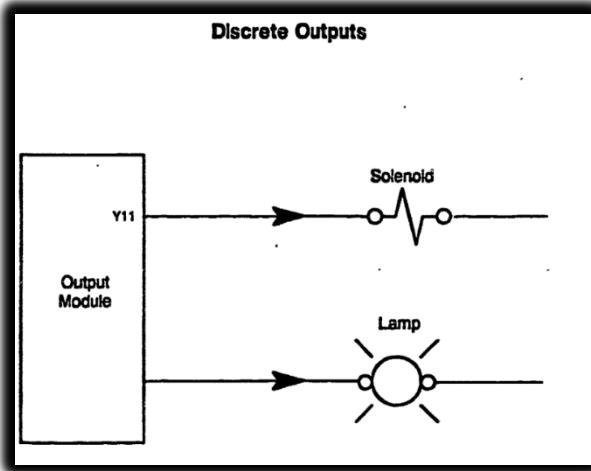


**Analog inputs** are signals that represent a range of continuous values, not just ON or OFF. They provide the PLC with a detailed measurement of something in the real world. Think of them as a dimmer switch rather than a light switch.

- **Temperature Sensor:** Measures a continuous range of temperatures, from freezing to boiling. 
- **Pressure Sensor:** Reports the exact pressure in a pipe, not just if it's high or low. 
- **Flow Meter:** Measures the specific rate at which a liquid is flowing, like a certain number of gallons per minute. 
- **Strain Gauge:** Detects and quantifies a tiny change in a material's shape, like how much a metal beam bends under a heavy load. 
- **Potentiometer:** A variable resistor that can be used to control things like the speed of a motor by providing a variable voltage signal. 

## ⚡ Discrete outputs

**Discrete outputs**, or **digital outputs**, are signals from the PLC that can only be **ON** or **OFF**. They are used to turn devices on or off, just like a light switch. The PLC sends a signal to a relay or a switch that then controls a device.



They control devices that either:

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

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.

- **Solenoids and Solenoid Valves** 💧

A solenoid works like an automatic gate lock — when the PLC sends an ON signal, it either opens a valve to let air or liquid pass, or closes it to stop the flow.

- **Motor Starters and Contactors** ⚙️

Since PLC outputs are low-power, they send signals to contactor coils, which then close heavy-duty switches to safely start large motors or pumps with high voltage.

- **Indicator Lamps / Pilot Lights** 🌈

These are the colored status lights controlled by the PLC that show machine states — for example, green for running and red for stopped.

- **Buzzers and Alarms** 🚨

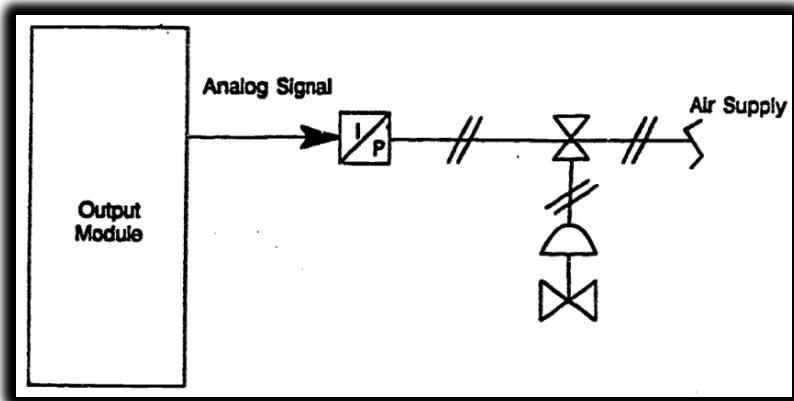
When the PLC detects a problem or a critical event, it turns on a buzzer or alarm to immediately alert human operators.

- **Relays** 🔗

A relay is an electrically operated switch; the PLC energizes it with a small signal, and in response it controls bigger devices or acts as part of more complex circuits.

## Analog Outputs

**Analog outputs** are signals that can have a continuous range of values, not just ON or OFF. They are used to control the speed, position, or intensity of a device. Think of them like the volume knob on a stereo, allowing for fine-tuned control.



- **Variable Frequency Drives (VFDs)** 

The PLC sends an analog signal to a VFD to smoothly adjust the speed of a motor, such as controlling the RPM of a fan or pump.

- **Proportional Valves** 

Using analog outputs, the PLC can regulate how much a valve opens, allowing precise control of fluid flow instead of just ON/OFF.

- **Heater Control** 

An analog output from the PLC can regulate temperature to a specific degree, making heating systems variable rather than simply full power or off.

- **Position Control** 

PLCs use analog signals to control positioning systems, like robotic arms, moving them smoothly and precisely to exact locations.



## Real World Example:

- An **I/P (Current-to-Pneumatic) Transducer** converts the PLC's analog electrical signal into a proportional air pressure.
- For example, the PLC sends a **4-20 mA current signal** to the transducer, which then adjusts a pneumatic valve.
- A low signal like **4 mA** opens the valve slightly, while a high signal like **20 mA** opens it fully, allowing for precise control of fluid or air flow.
- This entire process is part of a larger control loop, where the PLC's output is just one piece of the puzzle.

**Beware!** 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 for PLC Inputs/Outputs

The PLC's **output interface** is how it sends commands to devices in the real world.

**Actuators** are the "doers" that receive these commands and perform a physical action.

The PLC can control things like **motors** to start a machine or **lamps** to show a machine's status.

The PLC often uses a **relay** as a remote-controlled switch to safely control larger devices, as it cannot directly power them. 

**A PLC is a programmable controller composed of a CPU, power supply, and I/O modules that work together to replace complex, hard-wired relay logic with flexible software control.**

## 💡 PLC CPU Memory – What it stores and tracks

Stores the **program logic**, the actual instructions.

Tracks the current **status of inputs and outputs**.

Data Values – Stores values for:

- Timers ⏳ (e.g., wait 5 seconds before starting)
- Counters 📊 (e.g., count 10 items passing a sensor)
- Internal Bits ⚡ (virtual switches used inside the program)



## 📝 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. The memory holds the “how-to” list (instructions you give it) and the “what's happening” info (current readings or values from sensors).

## ✍️ 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.

## The Scan Time: Blazing Fast Automation! 🚀

A PLC is a dedicated industrial controller, built to run a single control program — and it does so continuously and extremely fast.

### 1. 🔎 Read Inputs

The PLC checks the status of **all input devices** (e.g., are switches pressed? Is the sensor triggered?).

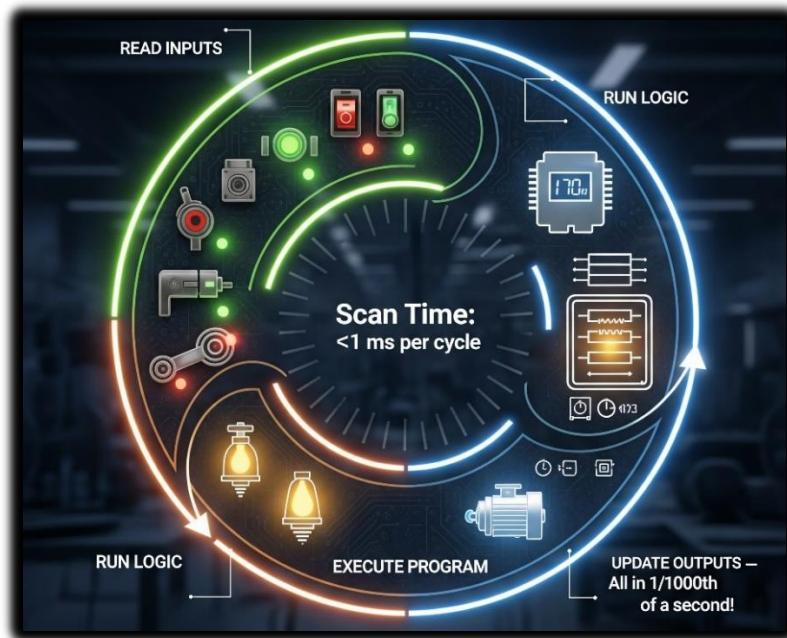
### 2. 🧠 Execute Program

Using the input data, the CPU **runs the logic** stored in its memory — this includes timers, counters, and all the if/then conditions in the control program.

### 3. ⚡ Update Outputs

Based on the results, the PLC **sends commands to output devices** (e.g., turn on a motor, light a lamp, or open a valve).

The time it takes for the CPU to complete **one full cycle** – reading inputs, executing the program, and updating outputs – is called the **scan time**.



This happens with mind-blowing speed, often in the range of **1/1000th of a second** (that's 1 millisecond!).

Imagine blinking, and the PLC has already completed several hundred scans!

This rapid cycling is crucial for ensuring that industrial processes respond instantly to changes in the real world.

## ⌚ PLC Scan Cycle Recap:

Every PLC constantly runs in a **loop** like this:

1. Input Scan → read sensor/input states
2. Logic Execution → run the ladder program using those inputs
3. Output Update → update real-world outputs based on logic

Except it's optimized, real-time, and done in **hardware + firmware**.

```
while (1) {  
    read_inputs();  
    run_logic();  
    update_outputs();  
}
```

## ⌚ Why the Scan Cycle Matters

- **Inputs don't update outputs instantly** → outputs only change on the *next scan*.
- **Timing issues** can happen if the program takes too long to execute (long scan time).
- **Instruction behavior**: some (like **SET/RESET**) persist across scans, others (like **OUT**) only work if the rung is true on *every* scan.
- A PLC is **scan-loop based** (not event-driven like normal PC code).

## 🔥 Exam Essentials

- Define what a PLC is and where it's used.
- Explain its main parts (CPU + I/O).
- Distinguish between **program mode** and **run mode**.
- Describe the **scan cycle** and what "scan time" means.
- Show why **memory** is crucial for holding programs, I/O states, and values.

## Main parts of a Programmable Logic Controller Summary

### 1. CPU (Central Processing Unit)

 *The brain.*

- Executes the control program (ladder logic, etc.)
- Does all the decision-making. Handles communication with other modules.
- Stores the logic in memory and updates outputs based on inputs.

### 2. Power Supply

 *The heart pumping electricity.*

- Feeds the CPU and I/O modules with stable DC power (often 24volt DC). Without this? Dead PLC. 

### 3. Input Module(s)

 *The senses (eyes, ears).*

- Reads signals from field devices (sensors, pushbuttons, limit switches). Converts them into logic levels the CPU understands.

### 4. Output Module(s)

 *The muscles (hands, feet).*

- Sends commands to actuators (motors, relays, lamps, solenoids). Converts CPU logic to real-world signals.

### 5. Programming Device (not always mounted, but essential)

 *The interface.*

- Laptop or handheld used to load/edit programs into the PLC. Without this, you can't tell the PLC what to do.

### 6. Communication Ports / Interfaces

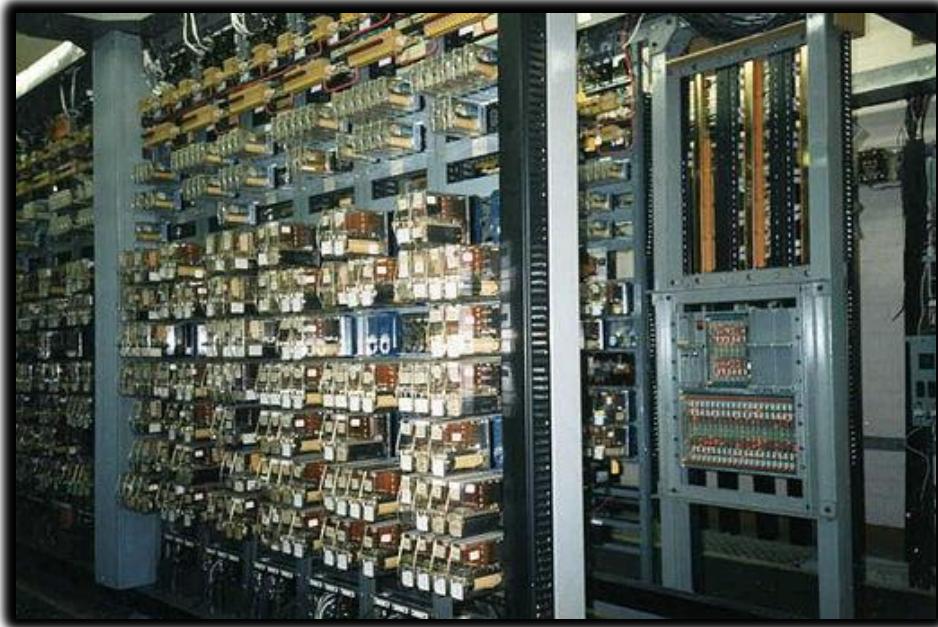
 *The mouth and ears for networking.*

- Ethernet, RS-232, Profibus, etc. for talking to HMIs, SCADA, or other PLCs.

## 💡 HISTORY OF THE PLCs

Before 1968, factories-controlled machines using **relay-based systems**:

- Relays = little electromagnetic switches turning things on/off.
  - To control one motor, you needed a power relay.
  - To control *that relay*, you needed control relays.
  - Add timers, counters... soon you had cabinets stuffed with hundreds of relays.
- 👉 Result: a hot mess. Hard to wire, hard to change, a pain to troubleshoot. 🤬



Enter 1968 (Hydra-Matic division of General Motors):

- Replace relay logic with a **solid-state controller**.
- Must support **ladder logic** programming (familiar to electricians).
- Must handle **harsh industrial environments** (dust, vibration, electrical noise).
- Must be **modular** (easy to expand and maintain).

## Dick Morley's team delivered:

Dick Morley's team first created the **Modicon 084**, the first commercial PLC, but it had limited success due to its speed and memory issues.



The subsequent **Modicon 184** was a more robust and user-friendly design that became an industry standard, replacing relay-based control systems.



## Key innovation:

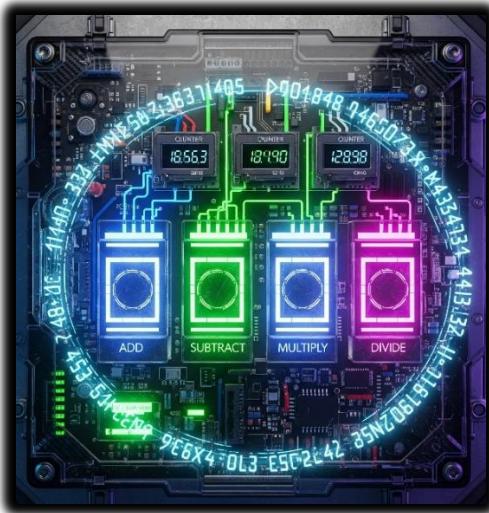
Instead of physically rewiring circuits, engineers now just modify software logic.

- Faster changeovers.
- Reduced downtime.
- Lower maintenance.
- Space-saving, scalable control.

## Leveling Up: New Powers for the PLC Brain 🧠🌟

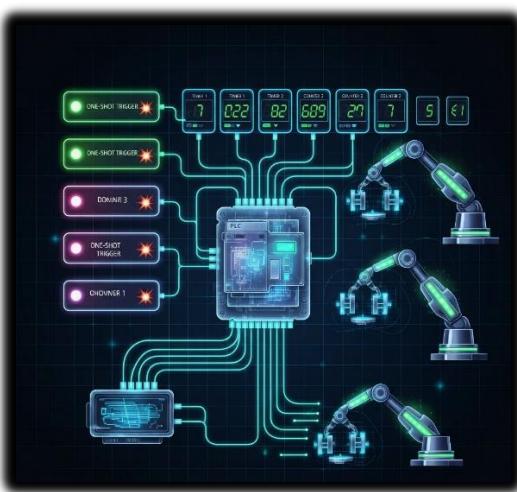
**Early PLCs** were limited, only handling basic ON/OFF inputs/outputs, simple logic, and basic timers and counters. This contrasts with **modern PLCs** which have much more advanced capabilities.

**1) Math Calculations:** Building on their ability to handle numbers for timers and counters, PLCs quickly learned to perform basic math and later mastered floating-point calculations for precise measurements and complex operations.



### 2) Enhanced Timing & Counting:

PLCs advanced from basic timers to include **one-shots** and more sophisticated timers and counters, allowing them to precisely control complex sequences and react to quick signal changes. 🧠



### 3) Process Control Superpowers (PID):

PLCs gained built-in **PID controllers (Proportional-Integral-Derivative)**, which act like a lightning-fast autopilot to maintain perfect stability in a system. They constantly fine-tune things like temperature or pressure to keep them exactly where they need to be without a human having to constantly make adjustments. 🌡️ A pharmaceutical company uses PID controllers to maintain precise pressure in tablet compression machines.



### 4) Drum Sequencers:

A **drum sequencer** is a type of PLC instruction that controls a machine through a fixed series of steps, one after the other. It's like a checklist for a robot, ensuring each task is done in the right order and completed before moving to the next. 🚗 Auto-Carwash: *pre-rinse → soap → scrub → rinse → dry → wax*. This ensures consistent results for any repetitive process.



## 5) Smarter Programming:

**Fill-in-the-blank programming** made it easier to set up common functions without writing a lot of code, similar to filling out a digital form.



**Meaningful Tag Names** replaced cryptic addresses (like I:0/0) with clear, descriptive names (like Start\_Button). This is like assigning names to your phone contacts instead of just using their numbers, making the program much easier to read and debug.



The ability to Import/Export Tags allowed these names to be easily shared between different devices, like a PLC and an HMI, saving time and reducing errors.

**Human-Machine Interfaces (HMIs)** replaced old-school, hardwired panels with digital, on-screen controls, leading to less physical clutter and better data visibility.



We moved from hardwired panels to smart HMIs. Less physical clutter, more data visibility, fewer mistakes.

## TALKING THE TALK: PROGRAMMING & COMMUNICATION EVOLUTION

As PLCs got smarter, so did the tools used to program them and the ways they communicated with the outside world.

### Programming Devices:

**The "Suitcase" Era:** Early programming devices were dedicated, clunky, and literally the size of suitcases! Not exactly portable.



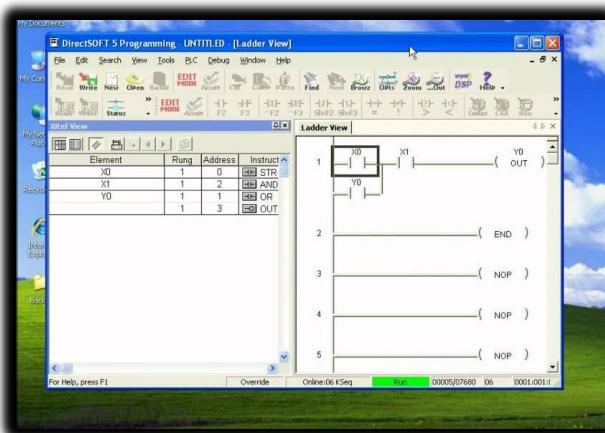
**Handhelds:** Then came smaller, handheld devices, offering a bit more freedom.



**PC Software Takes Over:** The real revolution happened when proprietary programming software moved to personal computers (PCs).



AutomationDirect's **DirectSOFT** was a pioneer as the first Windows-based PLC programming package. This was huge!



## Software, Hardware, Firmware

| 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.

## Communication Protocols:

PLCs needed to talk to other devices, and this area saw rapid growth.

### 1) The Early Days (MODBUS & RS-232):

Used basic serial protocols.

Direct cable link between two devices.

Slow, point-to-point communication.

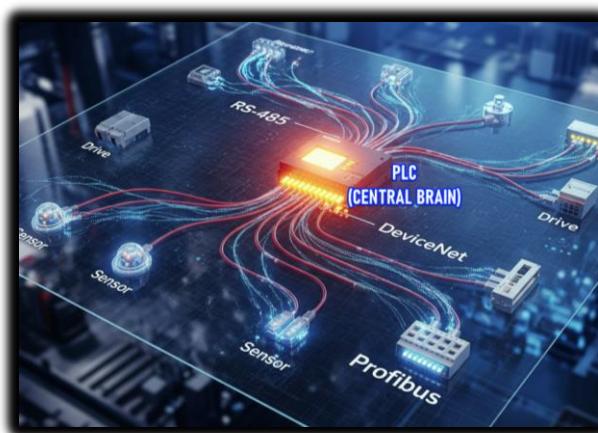


### 2. Expanding the Network (RS-485, DeviceNet, Profibus):

Allowed multiple devices over longer distances.

Think of it like a small office network for machines.

- **Sensors** = factory's eyes and ears (temperature, pressure).
- **Drives** = factory's muscles (motor speed, direction).
- **Remote I/O** = factory's nerve endings (extend reach across plant).



### **3. The Ethernet Era:**

Fast and flexible.

Standard for industrial communication.

Connects PLCs with drives, robots, HMIs, and SCADA.

Enables “smart factories” where everything communicates.



# HOW TO CHOOSE YOUR PLC

Choosing the right PLC for a project is like picking a perfect teammate; it requires careful consideration of key factors to ensure it's a good fit for the system's needs. 🤝

## Step 1: New System or Existing System? 🏠🔄

When choosing a PLC, you must first determine if you're building a **new system** or **upgrading an existing one**, as your PLC needs to be compatible with all existing equipment to avoid costly communication issues. 🏠



## Step 2: Environmental Check 🌡️

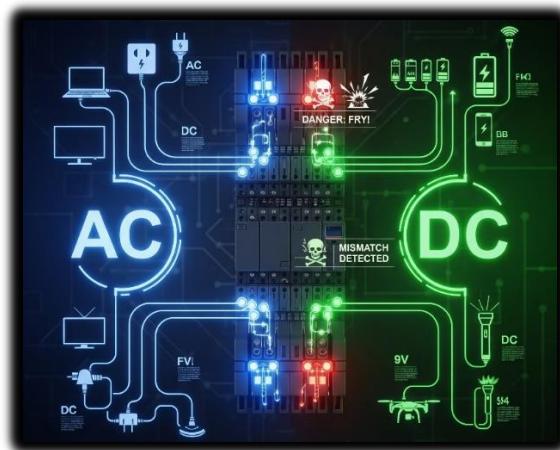
When choosing a PLC, you must consider the environment it will be in, ensuring it can handle factors like temperature, dust, vibration, and safety codes e.g. explosion-proof, waterproof, to prevent premature failure. 🌡️

Choose **ruggedized controllers** for harsh environments or **protect standard ones** with proper enclosures. Skip this step and your whole operation stops when the controller dies.



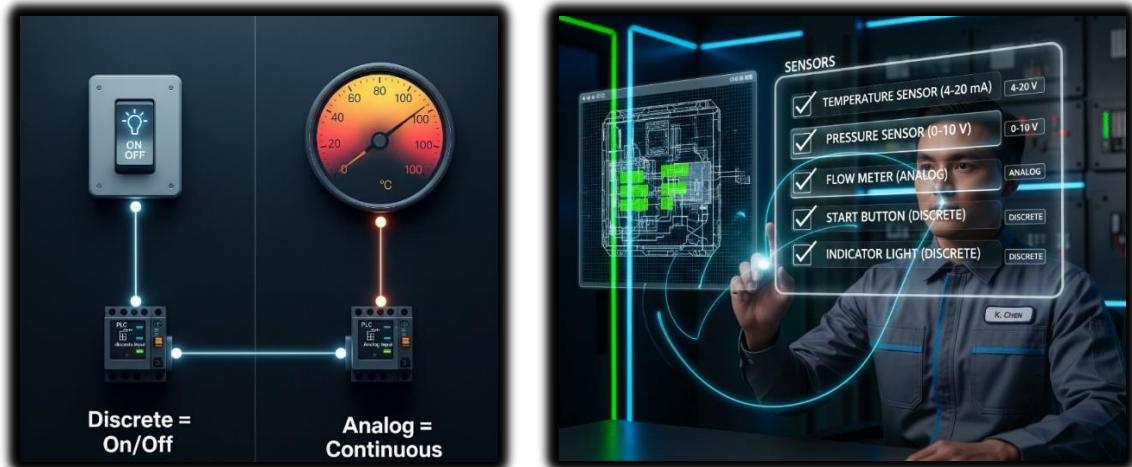
## Step 3: Counting the "On/Off" Crew (Discrete Devices) ⏪

Before selecting a PLC, you must count all the **discrete devices** (on/off components like buttons and lights) in your system and determine if they use AC or DC power to ensure your controller has enough compatible I/O points. ⏪



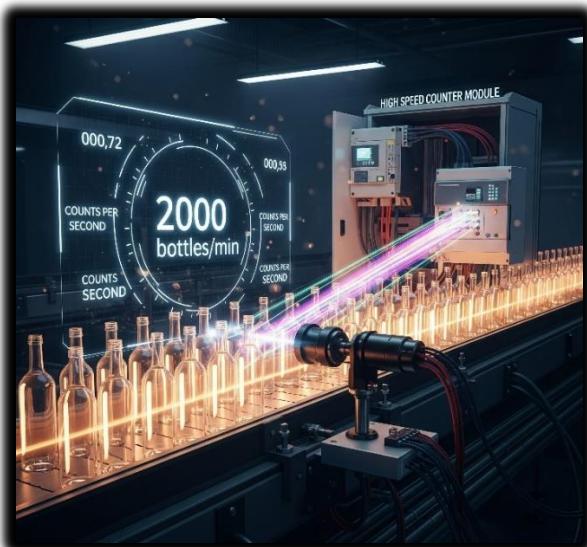
## Step 4: Count Your Analog Devices 📈

These are the ones that **measure a continuous range of values**, like a temperature sensor. You need to count how many of these devices you have and what type of signal they use (e.g., voltage or current) to ensure your PLC has the correct analog input modules. ✎



## Step 5: Special Features Check ✨🚀

Special features like **high-speed counting** or **precise positioning** are needed for systems that require more than basic on/off control. **Standard PLCs** often cannot handle these tasks because their scan cycle is too slow, so they need specific, and often expensive, **add-on modules** to work correctly. For example, a *high-speed counter module* is required to accurately count items on a fast-moving production line to prevent the PLC from missing events. ✨



## Step 6: CPU Power Check 🖥⚡

Choose your controller's CPU well, check the PLC's CPU power by looking at two types of memory.

- **Data memory** is for temporary storage of live values like sensor readings.
- **Program memory** is where your permanent code instructions are stored.

You need to ensure the PLC has enough of both to handle your system's data and code.

To estimate how much memory you need, a simple rule of thumb is to [allocate about 5 words per on/off device](#) and 25 words per measuring device. A "**word**" is a unit of memory, typically 16 bits, which acts like a small storage box for data. 🖥



**Real-world example:** A simple bottling line with 20 on/off devices (start/stop buttons, motors, lights) and 4 measuring devices (temperature sensors, flow meters) would need roughly:

- **20 devices × 5 words = 100 words**
- **4 devices × 25 words = 100 words**
- **Total: ~200 words of program memory**

So, you'd need a PLC with at least 200+ words of program memory capacity.



### Step7: How fast does it need to be? (Scan Time):

When selecting a PLC, you need to consider its **scan time**, which is how fast the CPU can read all inputs, execute the program, and update outputs.

A faster scan time is crucial for **high-speed operations**, such as fast packaging lines, to prevent the PLC from missing events or reacting too slowly.

For complex tasks like **PID control**, choose a CPU with a fast scan time for heavy calculations, not just simple on/off logic.  



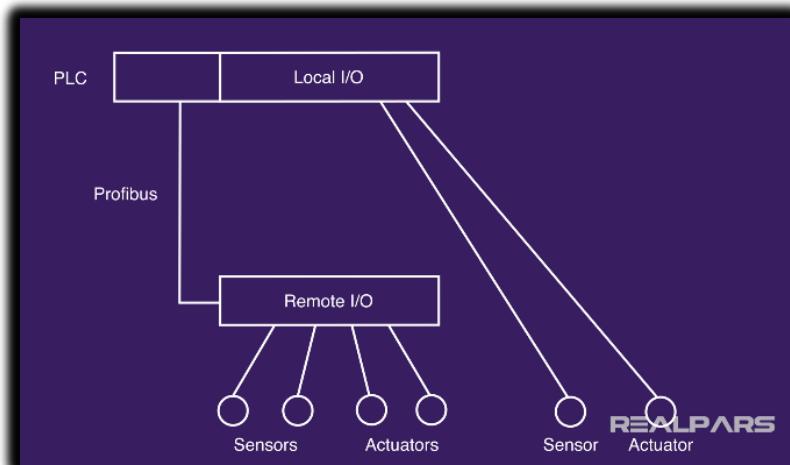
## Step 7: Where Are Your Devices Located? 🔑🌐

Are all your sensors, buttons, and motors close to the main PLC/controller, or spread across a big factory or different buildings?

**Remote I/O:** Devices are far away, so you use small connection boxes near the devices that talk back to the main PLC through one cable – similar to using Wi-Fi extenders across a big office.



Running many individual wires over long distances is expensive and complicated. **Remote I/O** solves this by using a *single communication cable* instead. Just like a Wi-Fi extender broadcasts your internet signal, remote I/O modules extend the PLC's reach to communicate with devices that are far from the main control cabinet. 🌐

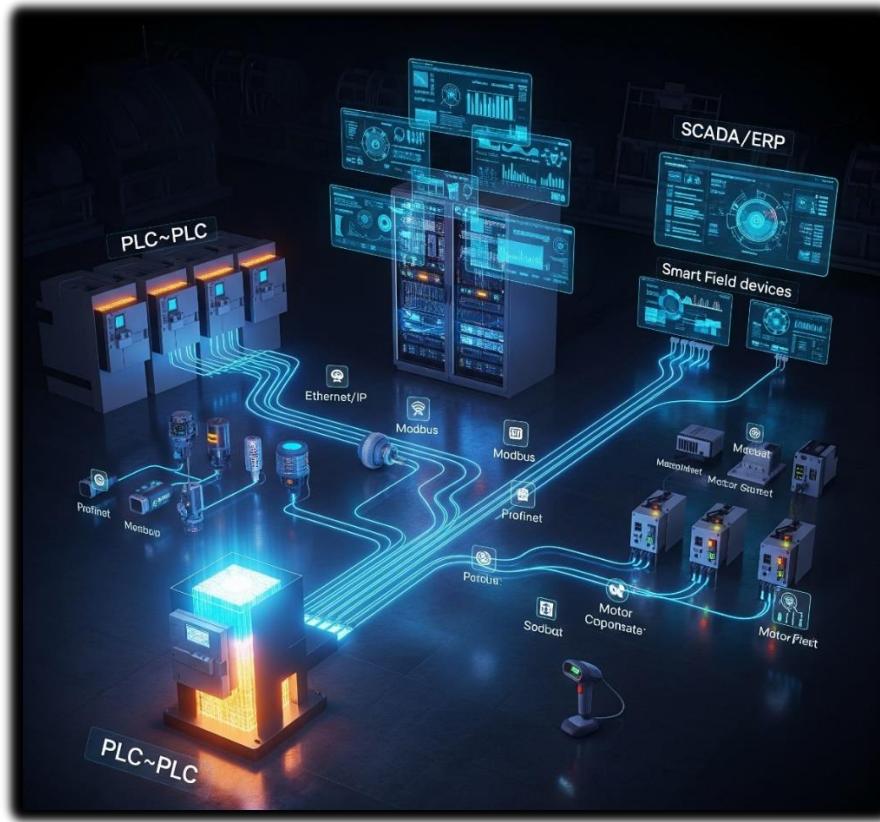


## Step 8: Who Needs to Talk to Whom? (Communication Requirements)

Before buying a PLC, you must identify all the other systems it needs to communicate with to ensure you choose a CPU with the right communication ports and protocols, or budget for any necessary add-on modules. 

Your PLC needs to communicate with other devices in the system, and you must know who it needs to talk to before you choose one. This communication can happen in a few ways:

- **PLC to PLC:** For two or more PLCs to share data and coordinate tasks.
- **PLC to Higher-Level Systems:** To send production data to systems like **SCADA** for monitoring or **ERP** for business management.
- **PLC to Smart Devices:** To communicate directly with things like **smart sensors** or **barcode scanners**. 



## Step 9: Programming Requirements

### Do You Need More Than the Basics?

Choosing a PLC requires you to first determine if your application only needs basic instructions like timers and counters, or if it also requires more specialized, built-in instructions. Not every controller supports all types of instructions, so you must select a model that can handle everything your project needs. 

**Standard instructions** cover the basics of PLC programming, including **timers** for counting time, **counters** for counting events, and **basic logic** for making simple decisions.



**Special instructions** are advanced, built-in functions that handle complex tasks like **PID control** for smooth process regulation or **motion control** for precise movements, saving you from writing complicated code.



# WHY WE USE PLC SOFTWARE??

## 🛠️ PLC Software: Your Digital Toolbox

We use PLC software because it is the essential digital toolbox for programming, debugging, and bringing the PLC to life.

It's **where you write the commands** that tell the hardware what to do. Choosing the right software is just as important as choosing the right PLC itself. 💻



## 🎮 1. Built-in Simulator: Your Virtual Playground

It gives you a safe virtual environment to test your ladder logic (or other code) *without touching the real machine*.

- You can **test your code**.
- Try out different scenarios to catch errors.
- You can mimic analog sensors and buttons, giving you a full-blown run.



## 🔥 2. Hot Swapping & Run-Time Transfers: No Downtime, Baby!

**Hot swapping** and **run-time transfers** are crucial features in modern PLC software that prevent system shutdowns.

Hot swapping allows you to replace hardware modules while the system is still running, similar to changing a part on a live machine.



**Run-time transfers** also known as online edits, let you update the PLC's program code without stopping the controller, saving significant time and money on maintenance. These features are essential for 24/7 production environments. 🚀

## ✨ 3. Auto Discovery: Plug-and-Play Magic

**Auto discovery** is a time-saving feature in PLC software that allows it to automatically detect and identify hardware modules as you connect them.

Instead of **manually configuring** each module's address, the software instantly recognizes what's been plugged in, making setup a seamless.



## 📊 4. Data View & Histograms: Seeing What's Up, Live!

PLC software includes powerful diagnostic tools that give you a live view into your system's operation.

**Data View Windows** let you watch and even adjust live values as the program runs, while **Graphical Trend Charts** and **Histograms** visualize how values change over time, making it much easier to debug and fine-tune your process. 🛠️📈



## 🔒 5. Security: Who's Got the Access? 🚫

PLC software includes **security features** that let you control access to your system. This is crucial because it allows you to set up user accounts with different permission levels, ensuring that only authorized individuals can make changes to the control logic. This protects your system from both accidental and intentional errors. 🔒



## 🔍 6. Search & Cross Reference: Your Code GPS

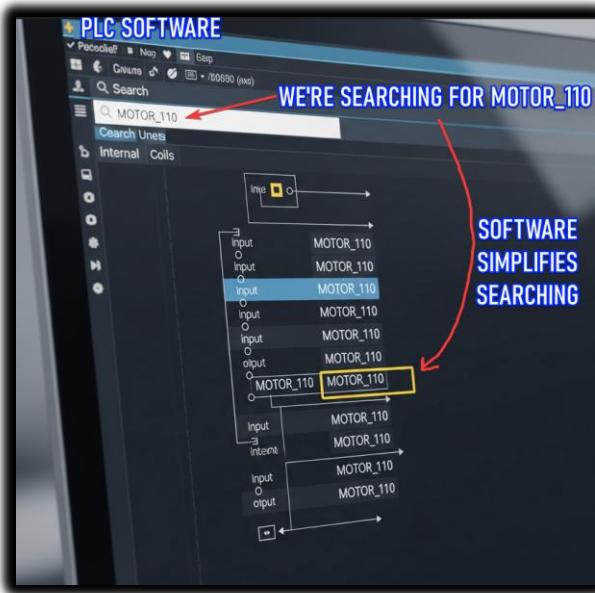
When your program gets big and messy, these tools save your brain (and your time). You'll use them all the time.

🔍 **Search:** Find exactly where an address, variable, comment, or instruction is used.

🔍 **Search & Replace:** Find something and update it everywhere it appears.

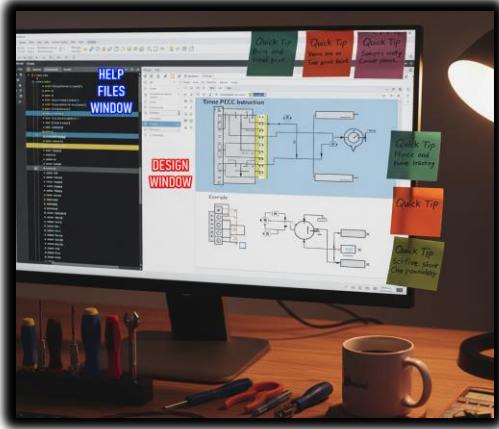
📍 **Cross Reference:** Shows you all instances of a particular tag or address.

🔴 **Why it's clutch:** It's like having GPS in a huge city of code, for precision.



## □ 7. Help Files: Your Built-In Guru

Great PLC software includes excellent help files that act as an experienced mentor. These files provide clear explanations, visuals, and examples to help you understand new instructions, troubleshoot code, and interpret error messages, so you're not stuck when you hit a wall. 



## 🌐 8. Connectivity: Plugging In & Getting Online

### 💻 USB Connections:

Connecting a PC to a PLC for programming or updates is usually done directly via **USB connections** using an industrial-grade cable. An engineer simply plugs in their laptop, downloads the new logic, and the PLC accepts the updates. This direct connection is fast and easy, but it requires the laptop to be physically close to the PLC. 



## **Ethernet:**

For networked PLC systems, **Ethernet** is the standard communication method. It allows you to connect to the PLC over long distances, similar to how you connect devices on a home internet network. However, for some setups, you might need to install extra drivers or software to ensure a proper connection. 



## **USB Project Transfers:**

Some PLCs let you load a project onto a **USB stick** and plug it straight into the PLC. Perfect for remote sites where lugging a laptop isn't practical.



## 🎨 9. Customizable Layouts: Make It Yours! ⚡

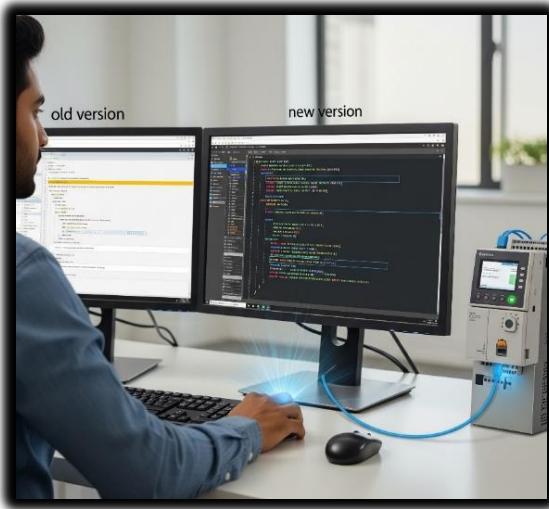
User-friendly PLC software is crucial because it allows you to **customize your workspace**, much like setting up a personalized coding environment. 💻



## 🤔 10. Project Compare: What Did I Change?! 📄

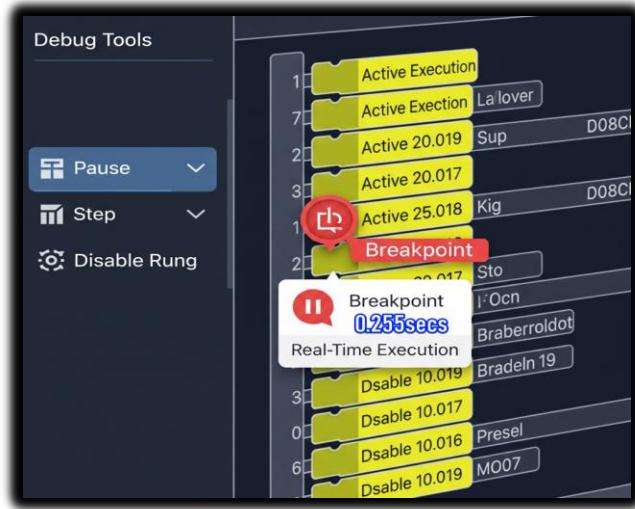
Ever download a project and later wonder, “*Wait... why's it acting weird? What did I even change?*”

**Project compare tools** are essential for this, as they show you the differences between your current project and an older version, or the one currently loaded in the PLC.



## 🐛 11. Debugging Tools: Squashing Bugs Like a Pro 💥

Debugging tools are essential because they allow you to step through your program's logic, pause execution, and simulate conditions in real time, helping you quickly pinpoint and fix exactly where a problem is occurring. 🐛🔧



## 🌐 12. Web Server & Mobile Apps: Control on the Go 🌎

Modern PLCs aren't chained to the factory floor anymore—now you can keep tabs from anywhere.

### ✓ Web Server Functionality:

Your PLC can host its own little web page. Just type in the PLC's IP address in a browser and boom—you can see live diagnostics, process values, and status updates from anywhere with internet access.

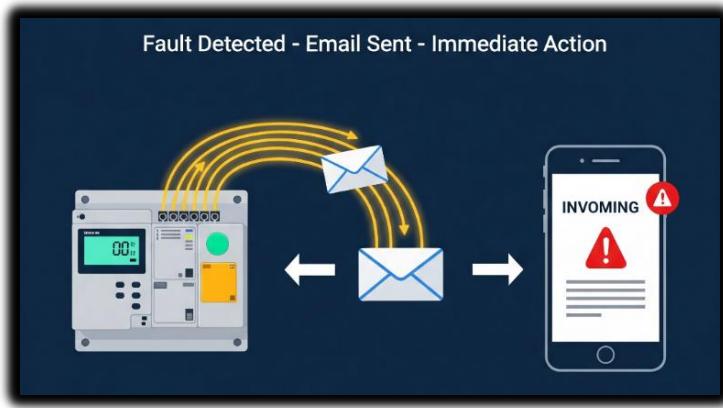
### ✓ Mobile Apps:

Many brands offer mobile apps that talk directly to your PLC.



## ✉ 13. Email Integration: Get Notified! ⚡

Some PLCs have the ability to **send email alerts** on their own for critical events, instantly notifying you and your team of issues like machine faults or low material warnings b4 issues escalate.



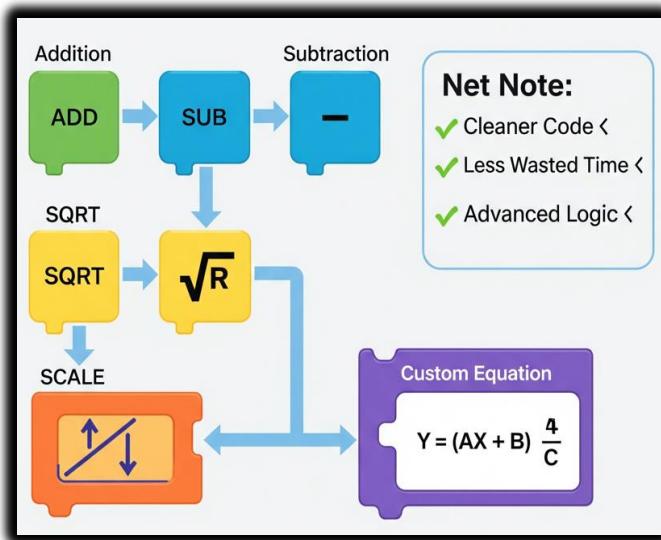
## ⚙ 14. PID Options: Smooth Operator Mode ⚙

A **PID (Proportional-Integral-Derivative)** controller is a control loop feedback mechanism used in PLC programming to precisely regulate variables like temperature or pressure. It works by continuously calculating the difference between a desired setpoint and the current value, then applying a corrective action based on three terms (**Proportional, Integral, and Derivative**) to maintain stability. ⚙



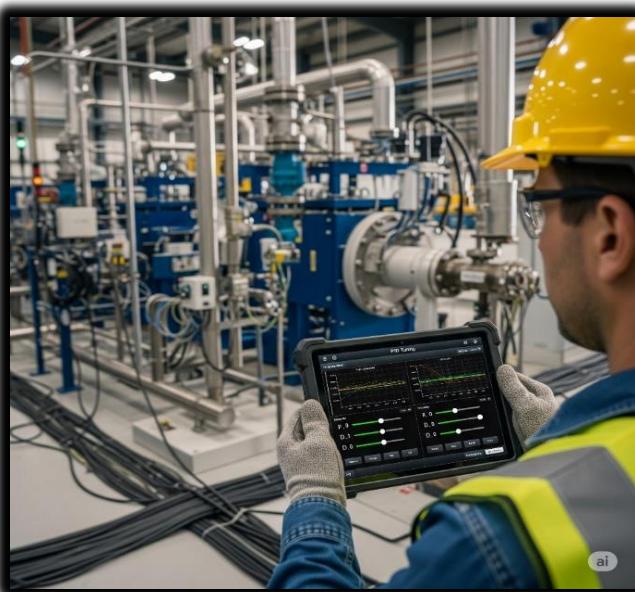
## 15. Powerful Math Functions: Crunch Those Numbers!

For modern processes that require complex calculations, great PLC software allows you to directly enter advanced mathematical equations, resulting in cleaner code and saving you from using messy workarounds. 



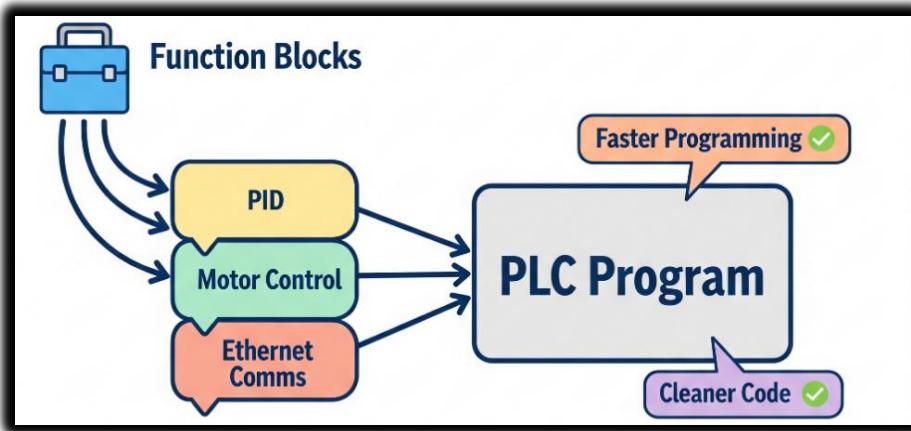
## 16. Task Manager: Organize Your Program's Flow

A **task manager** is a PLC software feature that allows you to divide a large program into smaller, more manageable pieces. This lets you run each task only when necessary, which optimizes the PLC's scan time and makes your logic easier to maintain. 



## 📦 17. Integrated Function Blocks: Programming Shortcuts 🚀

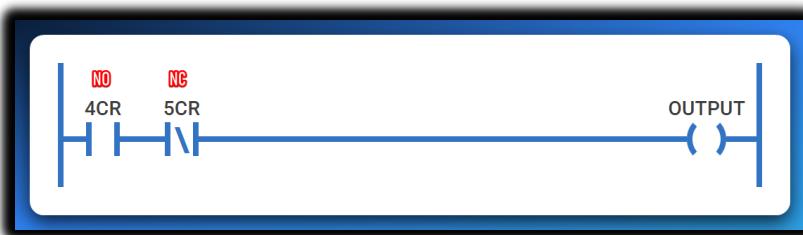
**Function blocks** are like ready-made mini-programs for common tasks. Instead of writing dozens of lines of code, you just insert a single block and configure it, which leads to faster programming, fewer errors, and cleaner, more modular code. 📦



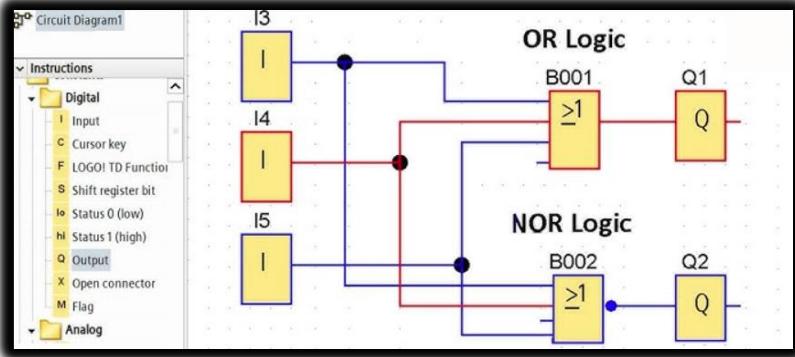
## 🗣 Programming Languages: Speaking Your Language

Ladder Logic still rules the industrial world, but PLCs speak more than one language — so pick what vibes with you:

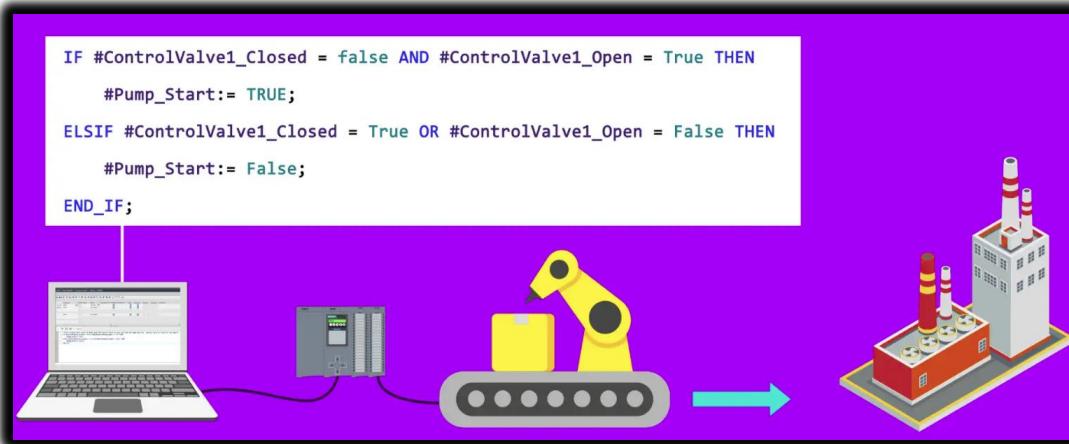
**Ladder Logic (LD)** is the most commonly used language in PLCs, especially in standard industrial applications. It resembles electrical relay diagrams, which makes it intuitive for electricians to understand and troubleshoot.



**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.



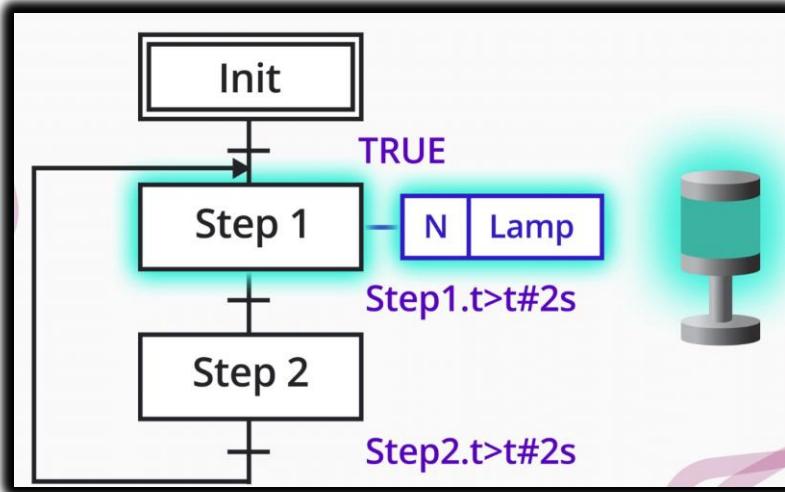
**Structured Text (ST)** is typically used in advanced control systems. It resembles the Pascal programming language and is best suited for applications involving complex logic or mathematical operations.



**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.

```
1 # Function block call example
2 CAL INST_CMD_TMR(IN:=%IX5.0..0, PT:=T#300ms) # Call timer function
3 LD INST_CMD_TMR.Q # Load timer output
4 ST BOOL1 # Store to boolean variable
5 # Arithmetic operation example
6 LD 1.000e+3 # Load initial value
7 ST REAL1 # Store to REAL1
8 MUL REAL1 # Multiply REAL1 by itself
9 SUB (4 # Begin subtraction operation
10      MUL( 1.0 # Multiply by 1.0
11      MUL2_REAL((*IM1:=CR(REAL),*) IN2:=(2.0)) # Complex multiplication
12    )
13 )
```

**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 are a superior choice for modern industrial control because they outperform traditional relay systems in:

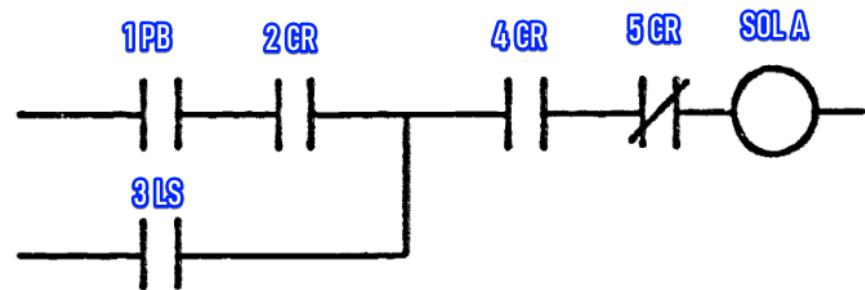
- **Flexibility:** PLCs can be reprogrammed easily with a few clicks, while relays require physical rewiring to change their logic.
- **Reliability:** PLCs are designed for harsh environments and are more reliable due to their solid-state components. Relays are prone to mechanical failure as their physical contacts wear out over time.
- **Data Collection:** PLCs can log data using built-in sensors, counters, and memory, providing valuable insights. Relays have almost no data collection capabilities.
- **Expandability:** You can easily expand a PLC's functionality by adding new modules, but expanding a relay system often means building a new control panel.
- **Repeatability:** PLCs execute the same sequence perfectly every time. Relays can wear out, causing inconsistent timing and faulty operations.
- **Space:** PLCs are compact modules that save cabinet space, whereas relays are bulky and require large control panels for complex systems.

## PLC vs. Personal Computers (PCs)

- **Operating & Storage Temperature:** PLCs are built to withstand extreme heat and cold on a factory floor, with a wider range for both operating and storage temperatures. PCs are designed for stable, comfortable office environments.
- **Humidity & Air Quality:** PLCs can handle damp, non-condensing air and corrosive gases, resisting industrial fumes and dust. PCs have a lower tolerance for humidity, and dust and corrosive gases can quickly damage their internal components.
- **Vibration & Shock Resistance:** PLCs are designed to military-grade standards (MIL-STD 810C) to withstand constant vibration and physical shocks. PCs have weak resistance, which can easily damage hard drives and internal components.
- **Voltage & Electrical Noise:** PLCs can handle power spikes and electrical noise, and they have high insulation resistance. PCs are not rated for such conditions and are easily affected by voltage fluctuations and electromagnetic interference.
- **Noise Immunity:** PLCs meet industrial NEMA specs, filtering out electrical interference from other machinery. PCs have no serious noise protection, making them vulnerable to glitches caused by electromagnetic interference.

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

## LADDER LOGIC



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

```
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.
```

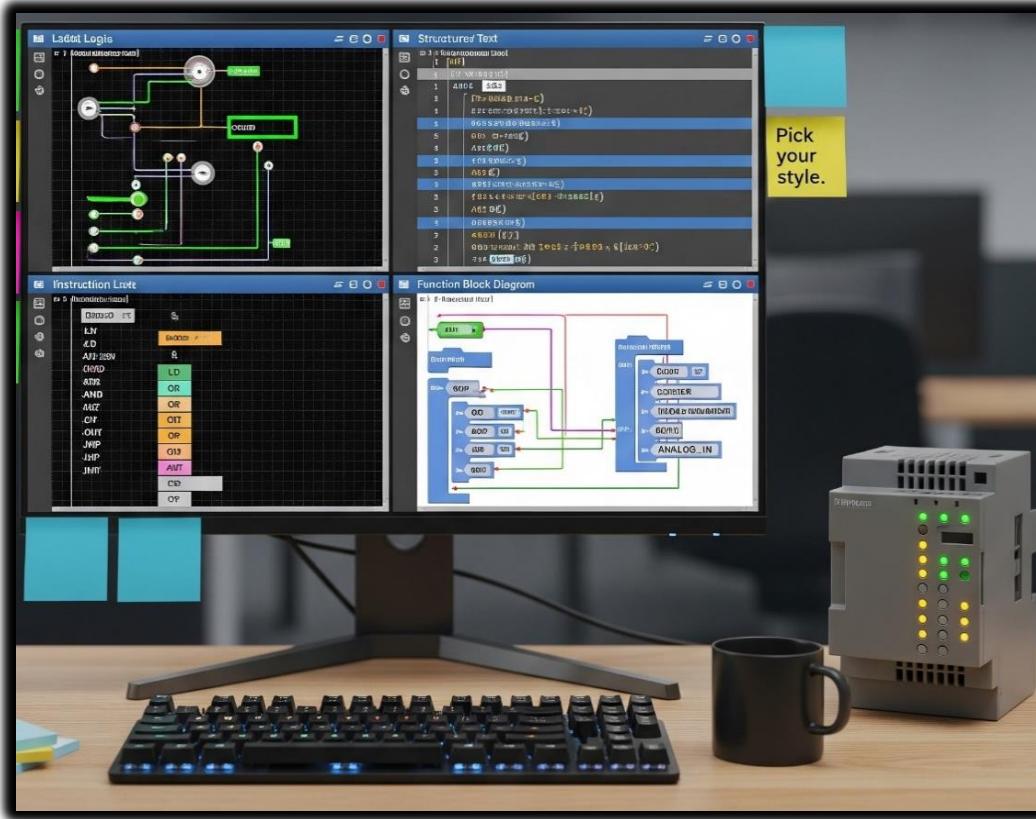
👉 **Ladder logic** is still the *main language everyone uses in factories*, but PLCs aren't stuck with only that.

You also have other language options:

- **Structured Text (ST)**: looks like regular programming (if...then, loops, math).
- **Instruction List (IL)**: looks like assembly language (low-level instructions).

- **Sequential Function Charts (SFC)**: flowcharts with steps and transitions.
- **Function Block Diagram (FBD)**: drag-and-drop blocks, very visual.

**The point:** Pick the one that feels easiest for *you* to program, debug, and maintain. Ladder logic is most common, but if your background is coding or assembly, you might feel more at home in ST or IL.



## 👉 Tag Names vs. Fixed Memory:

This is about how the PLC stores and labels data.

- **Tag Name Based:**

Instead of remembering cryptic addresses (I:0/0), you give your inputs/outputs friendly names like Start\_Button or Motor\_Speed.

Easier to integrate with HMIs, easier to read, way more beginner-friendly.

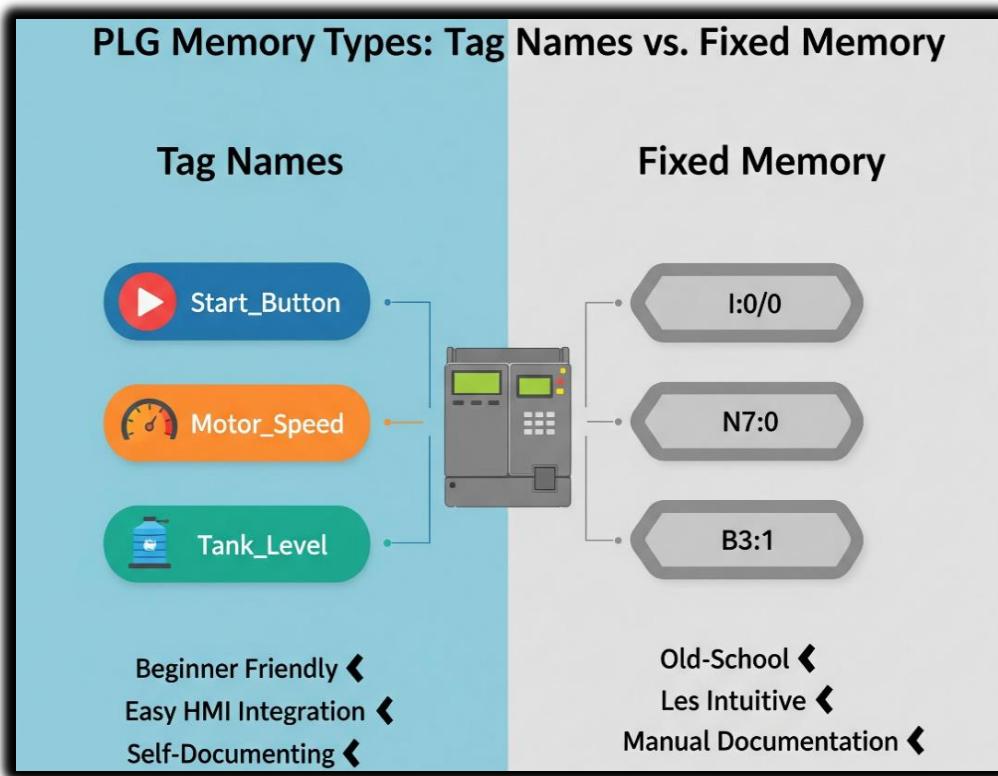
- **Fixed Memory:**

Uses hard-coded addresses like N7:0.

✓ Some old-school folks like it because they think it's easier to search or track in certain software.

✗ But it's less intuitive when your project grows big.

**The point:** Try both. They even suggest you download their free tools (Productivity Suite for tag names, Do-more Designer for fixed memory) and see which style clicks with you.



👉 **Final takeaway of that whole section:**

They're not giving you *everything* about PLC software, just highlighting a few extras that can really help your workflow. And if you're curious, they're inviting you to test out their free software and watch tutorials.

💡 **Your Move:**

Go with whatever keeps your code clean and makes sense for your project. If you're working with modern HMIs or big teams, tag names are a lifesaver.

### Wrap-up:

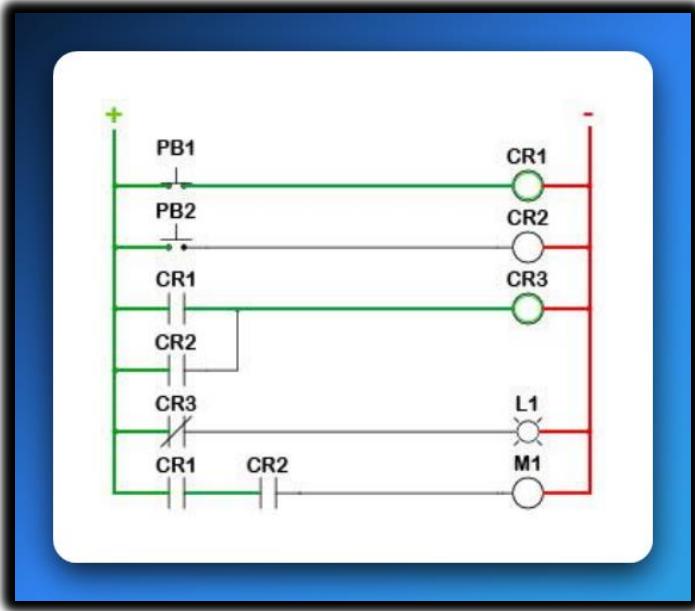
These extra considerations aren't the whole universe of PLC features, but they're the ones that can save your time, sanity, and project deadlines.

Pick tools and approaches that feel natural and fit your workflow — your future self (and your team) will thank you.

## The Foundational Structure: From Physical Relays to Digital Logic

At its heart, **Ladder Logic** is the main language used to program **PLCs**—those industrial computers that keep machines running like clockwork.

But to really understand it, you've gotta know where it came from: the old days of **relay logic** and **hard-wired control panels**.

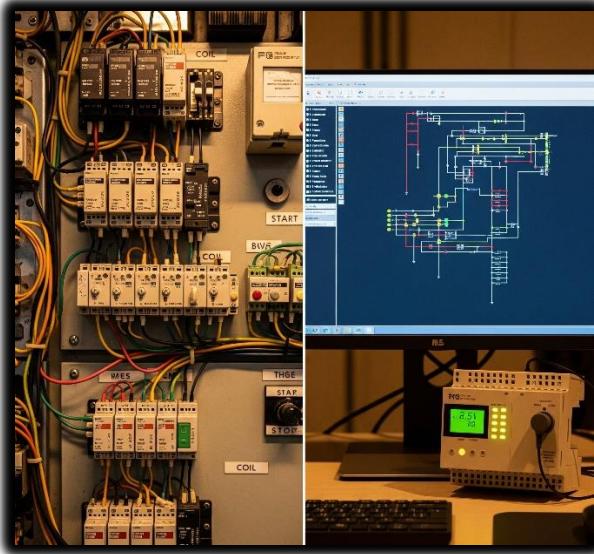


## ⚡ 1. The Roots: Relay Logic & Hardware Automation

Before microprocessors were everywhere, factories used **physical relays** to automate machines.

A relay is basically an electrically operated switch:

- Send a small current through its coil → it creates a magnetic field → that pulls or releases a metal arm → opening or closing contacts to let power through.



## 💡 Why It's Called a "Ladder"

It's not just a cute name. Ladder Logic diagrams **look exactly like old relay wiring diagrams**:

### Rails (Vertical Lines):

- The **left rail** is the power source (e.g., +24 V DC or 120 V AC).
- The **right rail** is the return path (common/ground). Power always flows left to right.

### Rungs (Horizontal Lines):

- Each rung is like one mini-circuit or one logic rule.
- On each rung, you place components—switches, contacts, coils—in **series** or **parallel**.
- If the path on that rung is complete, the output on the far right energizes.

## ⚙️ How It Worked (and Still Does in Principle)

In an old relay cabinet:

- Power starts at the **left rail**, passes through inputs like pushbuttons or sensors, then through relay contacts, and finally reaches an output device like a motor, light, or another relay coil.

- If every condition on that rung is true (closed contacts, active sensors) → the circuit completes → the output device gets power and turns on.

Modern Ladder Logic in a PLC mimics this exact same idea—but instead of physical wires and relays, it's all done in software.

## 2. The Evolution: From Tangled Wires to Clean Software Bits

The genius of **Ladder Logic** is how it took the messy world of physical relays and turned it into a neat software language.

Instead of grabbing a screwdriver and wiring real relays together, you “**draw**” your logic **on-screen** with symbols that look just like the old electrical components.

When the PLC runs your program, it's basically *pretending* electricity is flowing through those virtual circuits.

### Digital Translation – How It Maps Over

#### Power Flow:

In software, there's no real current. Instead, a logical **TRUE (1)** means “power is flowing” along that rung. If the path from the left rail to an output is logically true, that output gets activated.

#### Open vs. Closed Contacts:

Physical relay contacts become simple Boolean conditions in the PLC:

- **Normally Open (NO)** – shown as —| |—  
 True when the input is ON (button pressed, sensor triggered, relay energized).
- **Normally Closed (NC)** – shown as —|/|—  
 True when the input is OFF (button released, sensor not detecting, relay not energized).

#### Coils / Outputs:

Outputs are shown as —( )—.

 When the rung logic leading to that coil is TRUE, the PLC sets that output bit to **1**, which energizes the real-world device (motor spins, light turns on, valve opens) or even an internal memory relay.

## 👉 Big Picture:

Instead of digging through wires in a control cabinet, you're now dragging and dropping logic in software.

**Same principles, zero mess.** That's the magic of Ladder Logic.



## Key Ladder Logic Components & Their Analogies

Let's break down those classic symbols from your image and see how they vibe both in hardware *and* in software logic:

### ○ PB (Pushbutton) – Your Event Trigger

**PB1, PB2:** These are **inputs**.

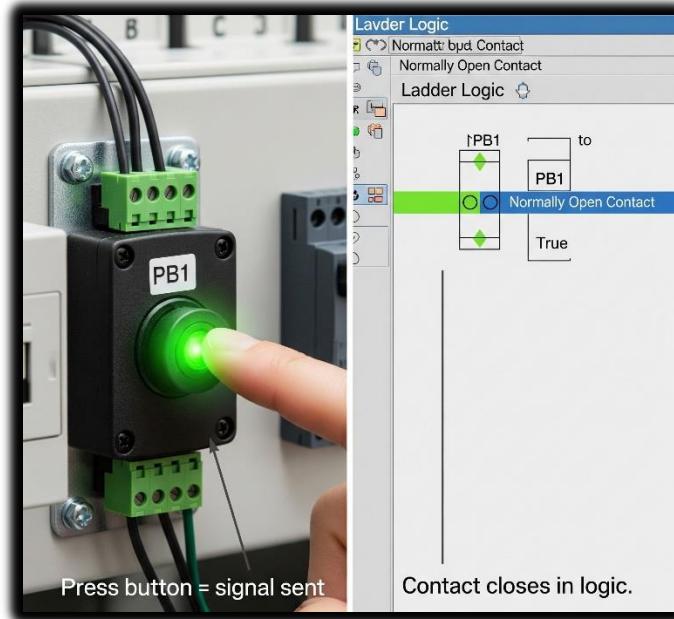
On a real panel, pressing a pushbutton closes a circuit and sends a signal into a PLC input terminal.

In Ladder Logic, it's represented by a contact symbol.

### 👉 **Analogy:**

A pushbutton is like calling a function or triggering an interrupt in code.

When you hit it, you're saying: "Yo, start that sequence!"



### ⚡ **CR (Control Relay) – Your Internal State & Your Electrical Middle-Man**

**CR1, CR2, CR3:**

Inside Ladder Logic, we treat these as **internal memory bits** — energize the coil (CR1) and every CR1 contact in your logic instantly follows that state.

**But in the real hardware world?**

A control relay also acts as an **electrically controlled switch**.

It lets the PLC — which only pushes tiny, low-power signals — safely control

**higher-power devices** like motors, solenoids, or large lamps.

It's the buffer between the fragile logic electronics and the beefy machinery, giving you:

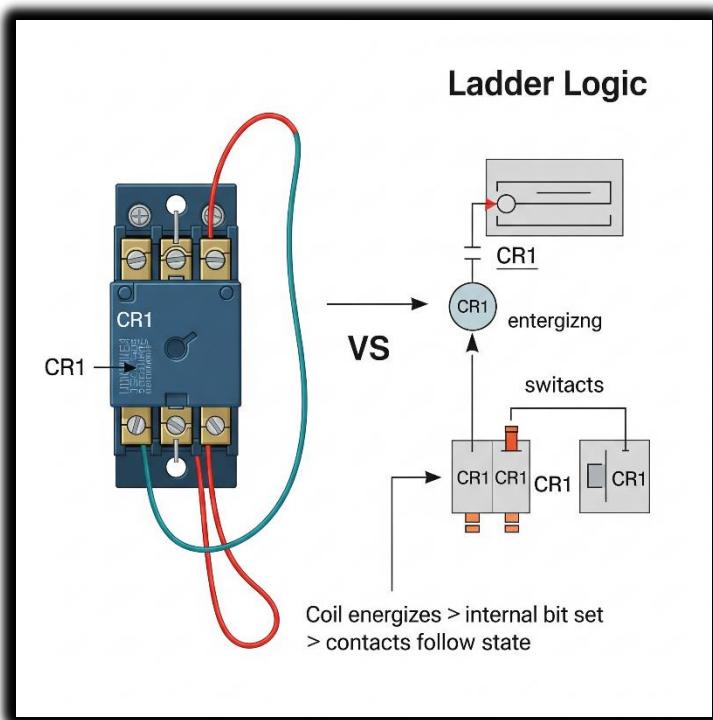
- Electrical isolation** (protects your PLC from power spikes),
- Safety** (no direct heavy current through your delicate PLC board),
- Stability** (reliable switching for big loads).

### 👉 **Analogy:**

Think of it as a bouncer at a club.

Your PLC quietly whispers, "let them in," and the control relay (the bouncer) swings open the big heavy door to the motor or solenoid.

The PLC never has to wrestle with high current directly — the relay does the heavy lifting.



Let's get a bit clear, we're talkign of ladder logic, but say, the plc takes your ladder logic and translates it to its own code for executing it right? so is that when it sees the code for control relays and says bet ! as we're executing, here's the parts where this bouncer steps in and does something for us eg? handling heavy power machines? like you mean a plc itself has some code in it that when it executes that code talks to the hardware telling it to do something special, is that what we mean?

Inside your ladder, when you drop in a coil (CR1) or an output (M1) you're basically saying to the PLC:

*"Hey PLC, if the logic on this rung is true, energize this output (or set this internal bit)."*

**What happens next depends on what type of coil it is:**

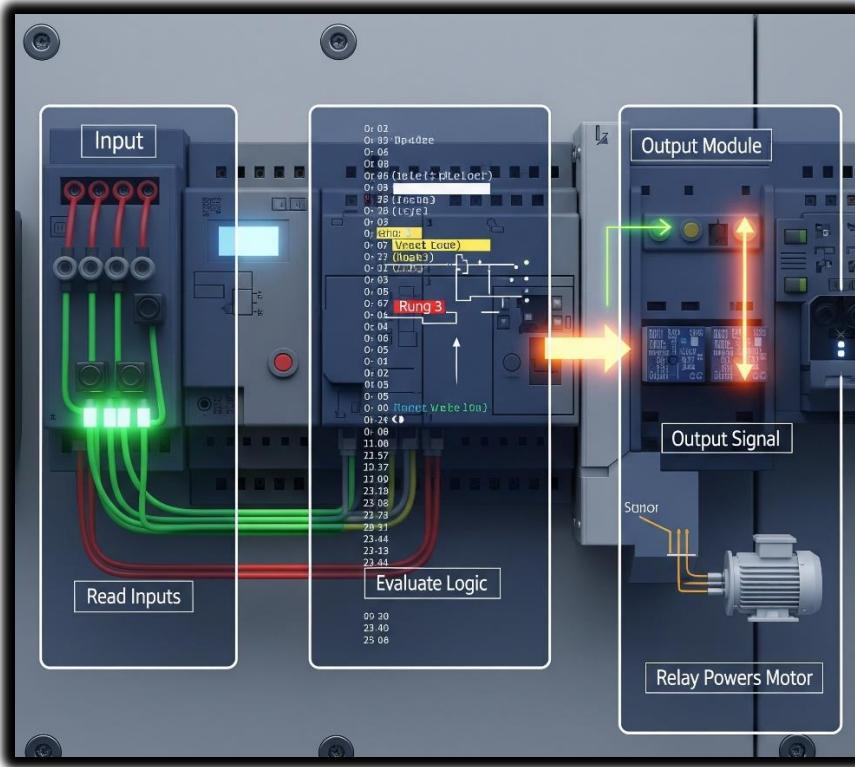
### 💥 Internal CR (Control Relay):

- This one lives purely in PLC memory.
- When the CPU executes that rung and it's true, it sets a *bit* in memory to 1.
- Any rung reading that contact (—| |— CR1) will now see it as "closed."  
👉 No actual electricity is switching heavy loads yet — it's just logic inside the PLC.

### 💥 Physical Output Coil (like M1 for a motor):

- When the CPU executes that rung and it's true, it flips a transistor or energizes a tiny driver circuit on the PLC's output card.
- That tiny driver is *not* strong enough to power a motor directly — and that's where real **control relays** or **contactors** out in the panel come in.
- The PLC output energizes the relay coil (low current).
- The relay's contacts then safely switch the **big current** going to your motor or solenoid.

The PLC scan loop runs through your compiled logic thousands of times a second. Each time it hits an output coil instruction, the CPU updates the hardware output register.



**That register is connected to output modules** — which *physically* drive relay coils, solid-state switches, or transistors on your control panel.

The **PLC isn't** literally muscling 10 amps into a motor — it's *flipping a small digital signal* that *commands* the relay/contactors to handle the big current.

I get it let me try explain, after compilation, you had your program like this rung had this control relays that handle the conveyor belt right? so when left vertical rail sends the input electricity, in the code its like plc speaking to itslef, we just received a power on singal now lets check what the code said, go to rung 3 execute it and it has this control relay that when it is in normally opened is it? so it executes that rung's code by telling it, send a singal i will hand you to this external module that is connected to me, so that it can start the motors of the conveyor belt right?

After your ladder logic is compiled, the PLC is constantly scanning. Imagine we're mid-scan:

#### **Input side:**

The PLC reads the status of all input terminals (pushbuttons, sensors, etc.).

 The left rail is like "power" coming in, but in PLC land it's really just reading TRUE/FALSE from each input.

#### **Logic side:**

Now the PLC runs through your ladder rungs, top to bottom: "*Ok, rung 3... oh, this rung has a coil that controls the conveyor motor. Let's check the contacts in this rung.*"

It evaluates your contacts (normally open/closed) based on the input states and internal bits. If the logic path is TRUE (like that normally open contact is now closed because the button was pressed): "*Alright, condition is true — energize that output coil!*"

#### **Output side:**

The PLC doesn't directly blast **3-phase power**(typically provides higher electrical power). Instead, it sends a low-power signal out of an output pin on the output module:

*"Hey output module, set your transistor/relay ON for Conveyor\_Motor."*

#### **External world:**

The output module energizes a control relay or contactor coil out in the panel.

**That relay/contact closes heavy-duty contacts that feed the actual conveyor motor's power circuit.**  And boom — your conveyor motor spins up. 

---

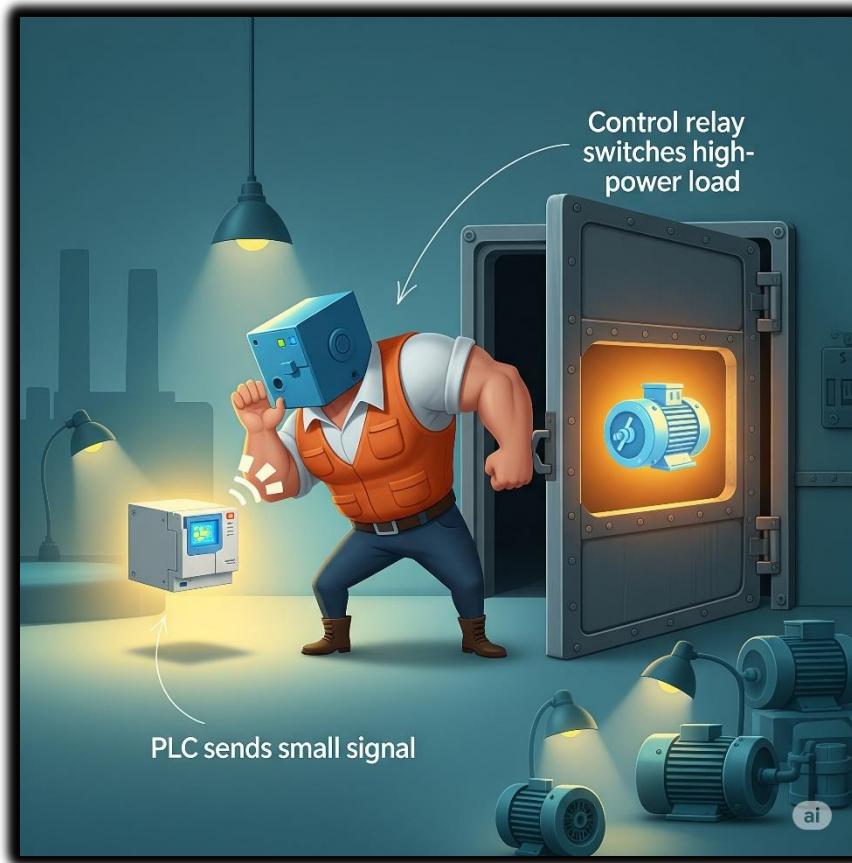
## So your sentence becomes:

*"When the PLC is scanning and gets to rung 3, it evaluates that rung's logic (including your control relay contact). If the logic is TRUE, the PLC sends a signal from its output module to energize an external relay or contactor, which then powers the conveyor motor."*

---

## Your idea is absolutely right.

- Left rail = inputs read.
- Code rung-by-rung.
- If logic says "go," PLC outputs a low-power signal.
- Output module reads the signal → control relay/contact closes → heavy machine (like the conveyor starts moving).



We talked about (CR1) as a "coil" and (L1) or (M1) as "output devices." These are essentially the "coils" that get energized.

When a rung's conditions are met, the "output coil" on that rung gets "energized." If it's an internal CR (Control Relay), it sets a memory bit.

If it's a physical output like L1 or M1, it triggers the corresponding output module on the PLC to provide power to the actual device.

## ⚡ The Evolution from Wires to Code – Ladder Logic's Secret Sauce

*Ladder Logic* didn't just give old relay panels a digital facelift — it *fundamentally changed* how we think about control systems.

Back in the day, every single relay, switch, and motor in a factory had to be hard-wired together.

If you wanted to change how it behaved, you grabbed a screwdriver and rewired the whole panel (and probably swore a lot).

Now? Those same circuits *live as memory bits* inside a PLC. Instead of cutting and crimping wires, you "draw" your circuit in software — and the PLC turns that into real-world action.

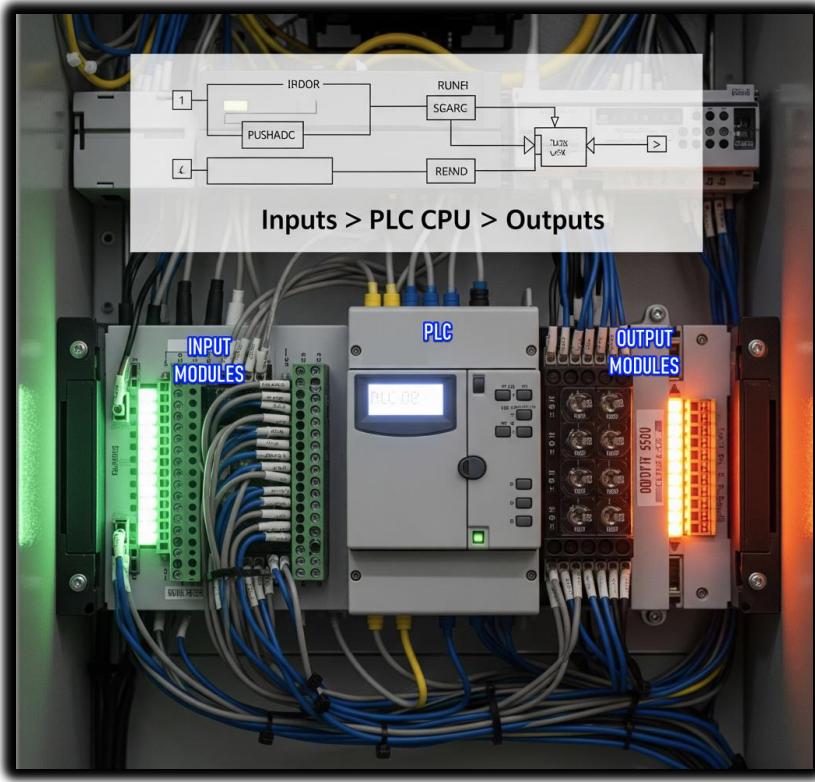


## 🔗 The Ins and Outs: Where Hardware Meets Software

A PLC lives in two worlds at once:

- **Physical world:** pushbuttons, sensors, motors, lights.
- **Digital world:** memory tables, logical rungs, and CPU scans.

The bridge between them? **Input/Output (I/O) modules.**



## 👁️ Inputs: The PLC's Eyes and Ears

Inputs are like sensors feeding data into your system.

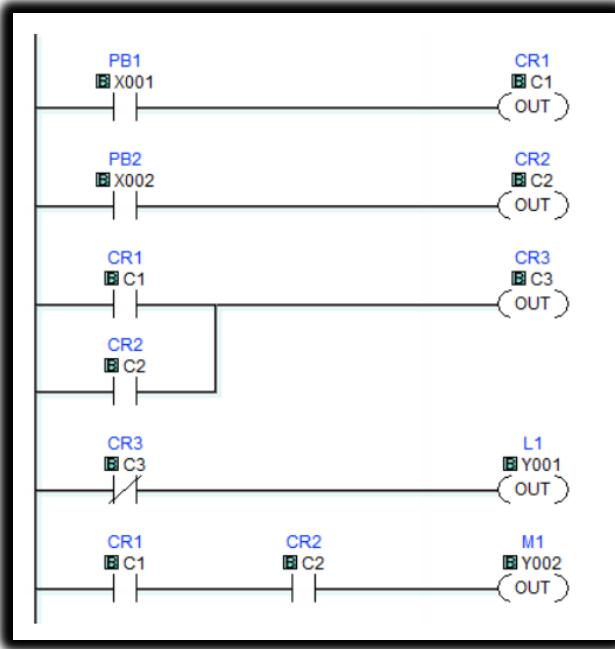
They are physical devices — like **pushbuttons (PB)**, **limit switches**, **sensors**, etc. These devices connect to the **PLC's input module**, which monitors whether there's voltage at each terminal.

When voltage is detected, the PLC sets an internal **memory bit** for that input to TRUE (or 1). That bit can then be referenced in your ladder program.

*Ladder logic was designed to have the same look and feel as ladder diagrams, but with ladder logic the physical contacts and coils are replaced with memory bits.*

## 💡 Example: PB1 and X001

In the diagram, you see:



- PB1 is a **physical pushbutton** connected to input terminal **X001**.
- When you press PB1, voltage is sent to that terminal → PLC sets **X001 = TRUE**.
- In the ladder diagram, the —| |— symbol means "check if this bit is TRUE".
- If X001 is 1, the virtual contact "**closes**" and logic can flow to the right.

💡 This is what we mean by "normally open":

The contact —| |— is **open** (no logic flow) unless the bit it watches becomes TRUE.

## ● Outputs: The PLC's Muscles

Outputs are how the PLC **acts on the world** — turning on lights, motors, alarms, etc.

When your ladder logic energizes an **output coil**, the PLC sets the corresponding **Y-bit** in memory. That tells the **output module** to physically energize the circuit.

### ⚡ Example: L1 and Y001

On this rung:



- CR3 (internal bit C3) must be TRUE.
- If so, the output coil Y001 is energized.
- That flips the bit in the output module → current flows → **L1 (a light) turns on**.

### 🧠 Analogy:

- Inputs = "Status updates" from the real world.
- Outputs = "Commands" sent out to the real world.
- The CPU = The brain that interprets inputs and triggers outputs via logic.

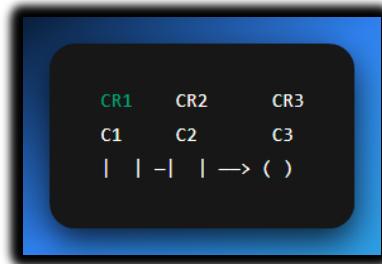
## ⌚ Internal Bits (Like CR1, CR2, CR3)

Not all contacts and coils represent **real-world hardware**.

CR1, CR2, CR3 are **internal control relays**:

- They're like virtual relays inside the CPU.
- They're used for logic control only — not connected to physical I/O.

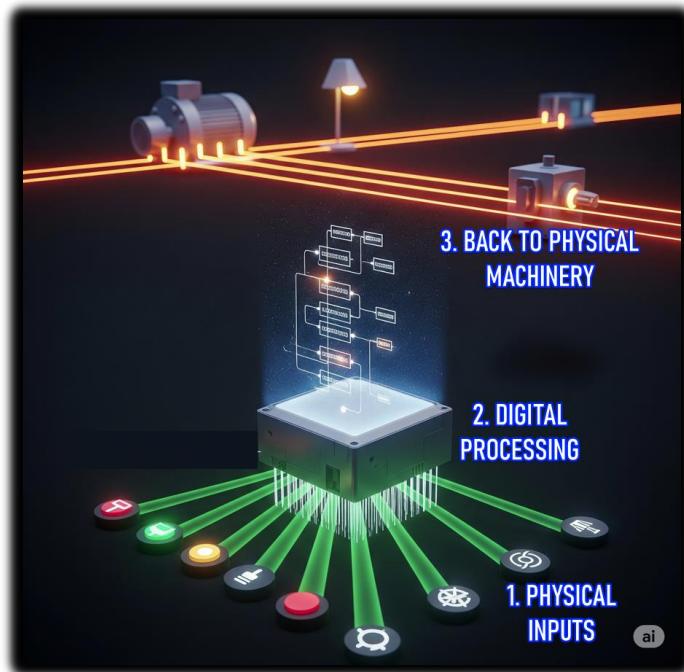
For example:



This rung says:

*"If internal relay C1 is TRUE and C2 is TRUE, then set C3 = TRUE."*

This kind of logic lets you build **intermediate steps**, sequences, and memory-like behavior.



## ⚡ TLDR: Why Ladder Logic Hits Different Now

Ladder Logic transformed factory control from **hardware headaches** to **software smoothness**.

Back then? Changing behavior meant rewiring physical panels — hours of work, high chance of mistakes.

Now? It's just: **edit → download → done**.

Behind the scenes, those chunky relays and switches are now just **memory bits** in the PLC's brain (X, Y, M, etc.).

## 🧐 So, what's really happening?

- 🧠 **Inputs** → Digital bits (from input modules)
- 💻 **Logic** → Runs through virtual rungs in PLC memory
- ⚙️ **Outputs** → Real-world actions (through output modules)

When your ladder rung hits true, it's like the PLC says:

*"Yo, flip this bit. Let's run that motor."*

And the output module?

*"Say less. Power's going live."*

## 🔗 Tying It All Together — One Rung at a Time

Let's walk through how **electricity "flows"** in this PLC program using your image:

1. **Rung 1:** PB1 (X001) must be pressed → sets CR1 (C1) = TRUE
2. **Rung 2:** PB2 (X002) must be pressed → sets CR2 (C2) = TRUE
3. **Rung 3:** CR1 and CR2 must both be TRUE → sets CR3 (C3) = TRUE
4. **Rung 4:** If CR3 is TRUE → turn on L1 (connected to Y001)
5. **Rung 5:** If both CR1 and CR2 are TRUE → turn on M1 (connected to Y002, possibly a motor)

## ⚠ Quick Note on Contacts

- | |— → **Normally Open Contact** (logic flows when bit is TRUE)
- | /|— → **Normally Closed Contact** (logic flows when bit is FALSE)

This diagram uses all **normally open** contacts, which means:

*"Let power through only when the bit is active."*

## 🧠 Memory View

| LABEL | MEMORY ADDRESS | TYPE          | FUNCTION   |
|-------|----------------|---------------|------------|
| PB1   | X001           | Input (Real)  | Pushbutton |
| PB2   | X002           | Input (Real)  | Pushbutton |
| CR1   | C1             | Internal Bit  | Logic flag |
| CR2   | C2             | Internal Bit  | Logic flag |
| CR3   | C3             | Internal Bit  | Logic flag |
| L1    | Y001           | Output (Real) | Light      |
| M1    | Y002           | Output (Real) | Motor      |

## ❖ Final Summary

*"Press PB1 and PB2 → set internal logic relays → light and motor activate. All through virtual contacts, no physical rewiring needed. Just bits flipping in the PLC's mind."*

Now that's how you do modern industrial control — with **digital muscle, logic memory, and clean ladder flow.**

## Internal Bits = Pure Logic Power

Not everything in your ladder program needs to control a real-world device. Some bits exist just to help you **structure logic cleanly and smartly**.

Control Relays like CR1, CR2, CR3 are **internal memory bits** — like **flags** or **global variables**.

- One rung might **set CR1 = TRUE** (e.g., after certain conditions are met).
- Other rungs can then **check CR1** as if it were a real input — no wire needed.

## Think of it like:

- CR1 is your custom signal — a pure memory trigger.
- It's like `bool CR1 = true;` in C.
- Or a malware flag: `isPayloadReady = true.`

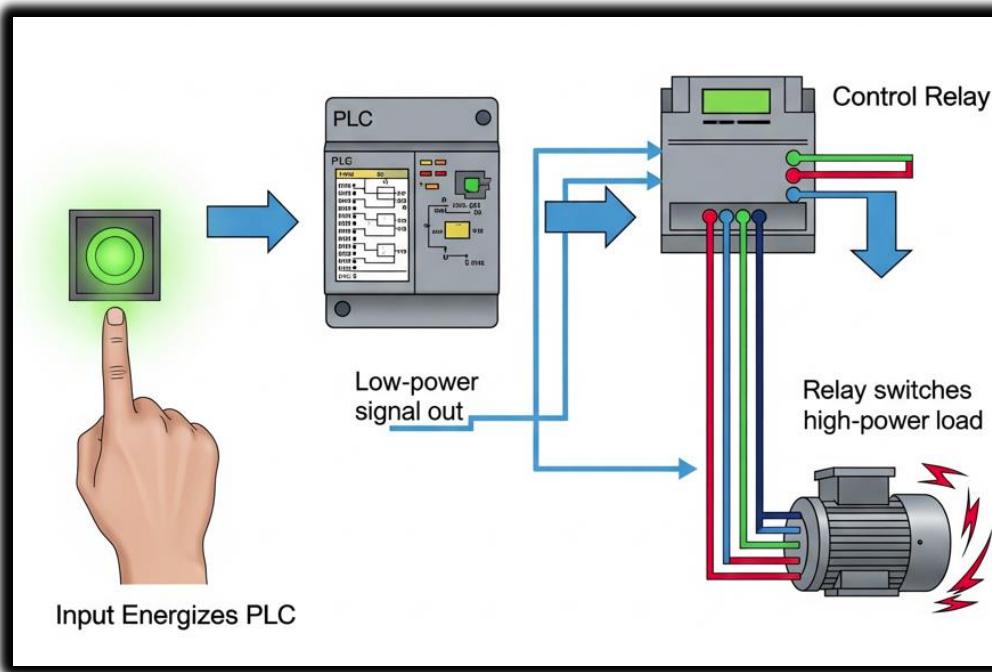
These bits **don't control power** directly — they **control logic**, and logic controls everything else.



*"The real power of PLCs isn't just physical I/O — it's how smartly you use the memory bits between them."*

 Yes, the sequence is:

1. An input energizes the PLC (button pressed / sensor triggered).
2. PLC logic decides → sends out a low-power signal from its output module.
3. That low-power signal drives a control relay coil (CR).
4. The relay's heavy contacts switch on and supply high power to a machine (motor, lamp, conveyor).



## 💡 Temporary Storage and Modular Thinking

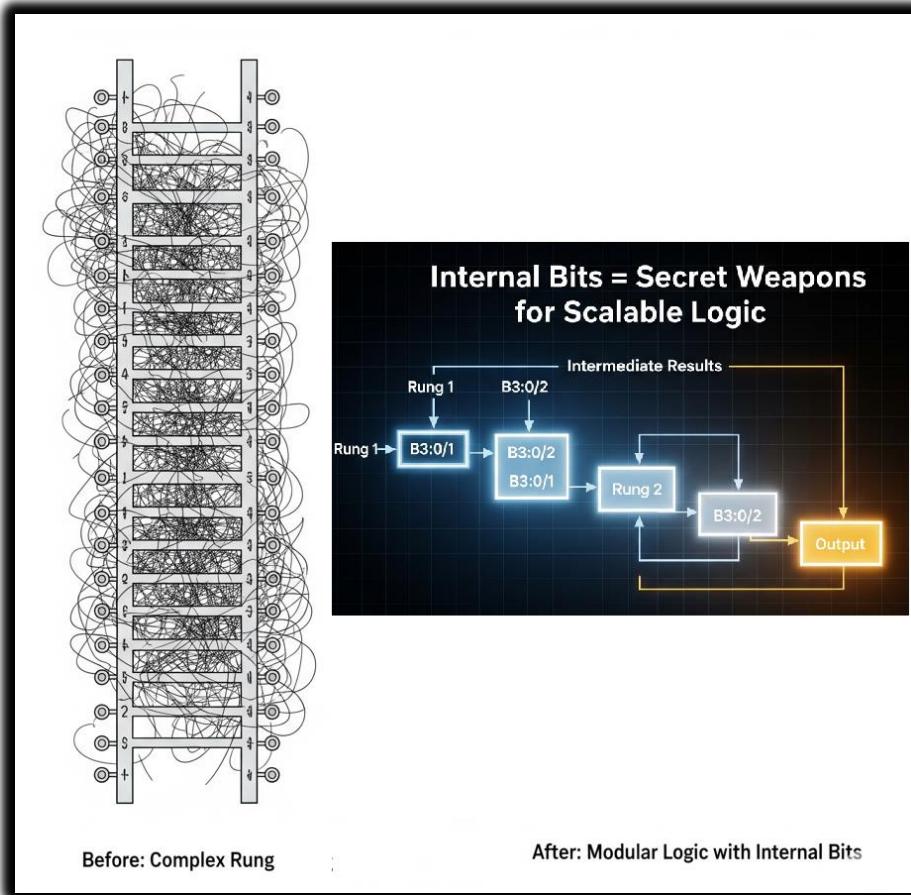
**Internal bits** aren't just for keeping states.

They're also great for **holding intermediate results**.

Instead of cramming everything into one messy rung, **split the logic into smaller rungs**.

**Store** each step's result in an internal bit.

It makes your program cleaner, easier to read, and way easier to debug later.



## 💡 Bottom line:

Internal bits are your secret weapons. They let you build smarter, layered logic without adding extra hardware. Combine them with your PLC's advanced features, and your ladder program becomes a powerful, professional-grade control system.

## Beyond Basic Contacts and Coils: The Modern PLC's Capabilities

While your provided text wisely advises keeping it simple for an introduction, it's vital to acknowledge that **today's PLC CPUs offer a vast array of sophisticated functions, far beyond mere contacts and coils.** This is where the PLC truly transcends its relay logic ancestors and becomes a powerful industrial computer:

- **Math Functions:** Addition, subtraction, multiplication, division, and more complex arithmetic operations. Essential for controlling variables like temperature, pressure, flow rates, and calculating production metrics.
- **Shift Registers:** Used for tracking items as they move along an assembly line, often used in conjunction with sensors to monitor position and sequence.
- **Drum Sequencers:** Imagine a virtual "drum" with multiple tracks, each controlling a specific output or action based on a sequence of steps. Ideal for automating repetitive, multi-step processes like batch mixing or complex assembly operations.
- **PID Control:** Proportional-Integral-Derivative controllers are advanced algorithms used for closed-loop feedback systems (e.g., maintaining a constant temperature in an oven by adjusting heater power based on temperature readings).
- **Data Handling:** Instructions for moving, comparing, and manipulating blocks of data.
- **Communication Protocols:** Built-in support for industrial networks (Ethernet/IP, Modbus TCP, Profinet, etc.) to communicate with other PLCs, HMIs (Human-Machine Interfaces), SCADA systems, and even enterprise databases.
- **Structured Text, Function Block Diagram, SFC:** Modern PLCs often support other IEC 61131-3 programming languages, allowing for more complex, high-level programming constructs similar to C or Pascal.

## ⌚ Chapter 5-2: Basic Instructions in Ladder Logic

### Boolean Logic and the DNA of Control

Now that you know what Ladder Logic looks like, it's time to understand **how it actually decides things**. And that takes us straight into **Boolean logic** — the bedrock of all decision-making in PLCs.

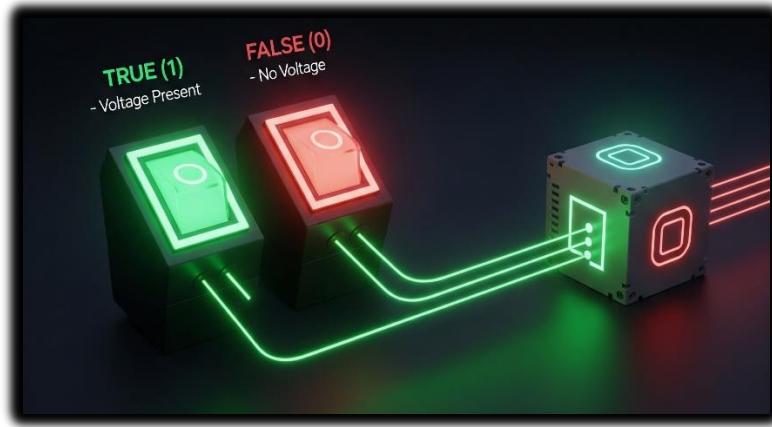
Don't worry, this isn't Digital Systems 101 — we're not breaking out Karnaugh maps or Boolean algebra proofs. We're just gonna look at the essentials: **AND** and **OR** logic.

### ▢ Boolean Logic in Plain Language

PLC logic is built on simple True/False decisions — like light switches:

- **TRUE (1)** → There's voltage or a condition is satisfied
- **FALSE (0)** → No voltage or the condition isn't satisfied

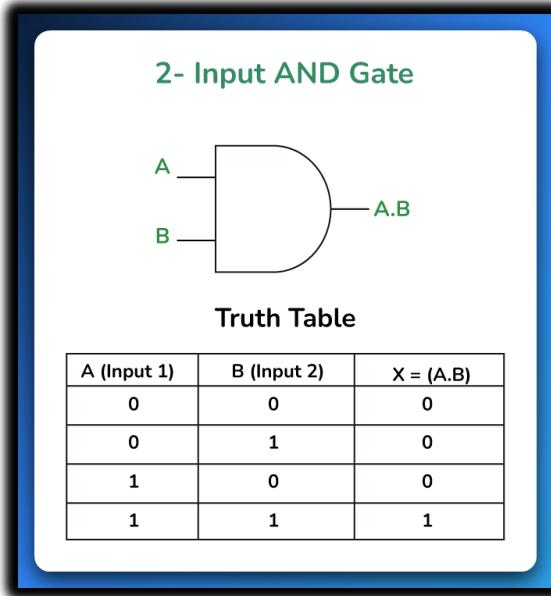
From this binary setup, we build logic gates. In Ladder Logic, these gates are **not separate components** — they're created through **how you arrange your rungs and contacts**.



## The Gates — Series Logic

### AND gate Ladder Implementation:

AND operations are analogous to **multiplication**.



Use **normally open contacts** in series.

This is saying:



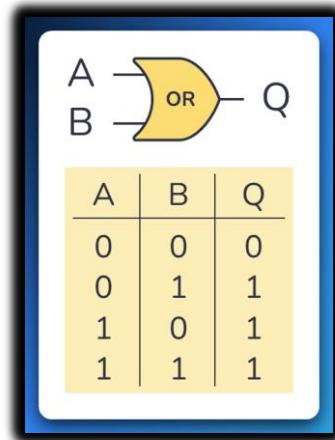
*"Only if A AND B are both TRUE, then turn ON the output."*

If either A or B is FALSE, current stops flowing — just like if one switch in a series circuit is off.

$$A \cdot B = X$$

## OR gate Ladder Implementation:

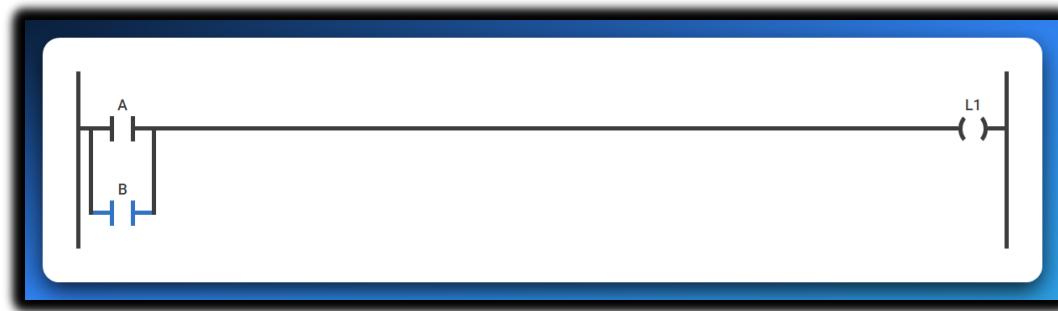
Use **normally open contacts** in parallel.  
OR operations are comparable to **addition**.



This is saying:

*"If A OR B is TRUE, then turn ON the output."*

As long as at least one of them is ON, the rung is complete and current flows to the output.

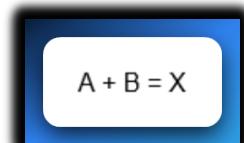


If **A** is **TRUE** (and B is FALSE), the path through A "closes," allowing logical power to flow to the output.

If **B** is **TRUE** (and A is FALSE), the path through B "closes," allowing logical power to flow to the output.

If **both A and B are TRUE**, both paths "close," and logical power still flows to the output.

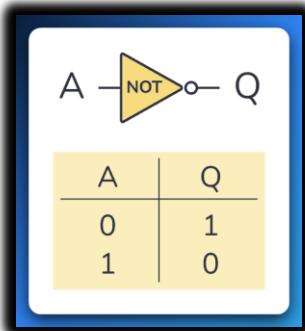
Only if **both A and B are FALSE** will both paths remain "open," stopping the logical power flow and keeping the output OFF.



## NOT Gate Ladder Implementation:

Use a **normally closed (NC)** contact in series.  
NOT operations are comparable to logical inversion.

**Truth Table:**



In short, **Q is the opposite of A**.

**A NOT gate can only have one input.** And its truth table is pretty simple since there are only two possible states; the input being HIGH (or "1") or the input being LOW (or "0").

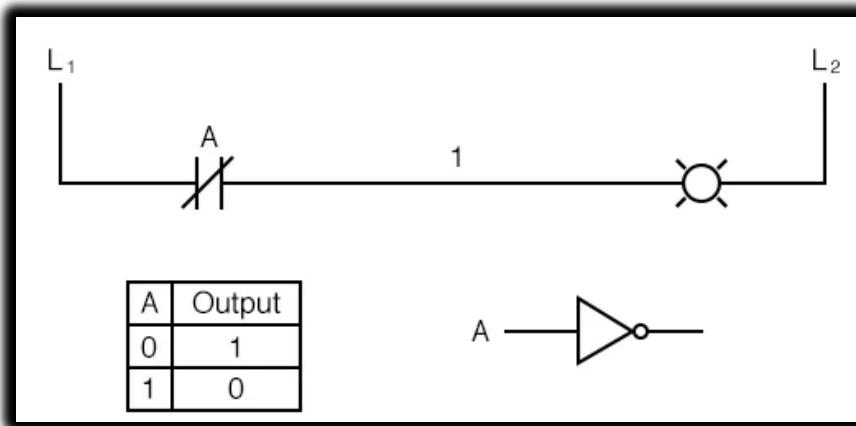
**This is saying:**

**"If A is NOT TRUE, then turn ON the output."**

As long as the input is **OFF (FALSE)**, the NC contact remains closed, allowing current (logical power) to flow to the output.

If the input becomes **ON (TRUE)**, the NC contact opens, breaking the circuit and stopping the output.

In ladder form, you simply place an NC contact in front of the coil:

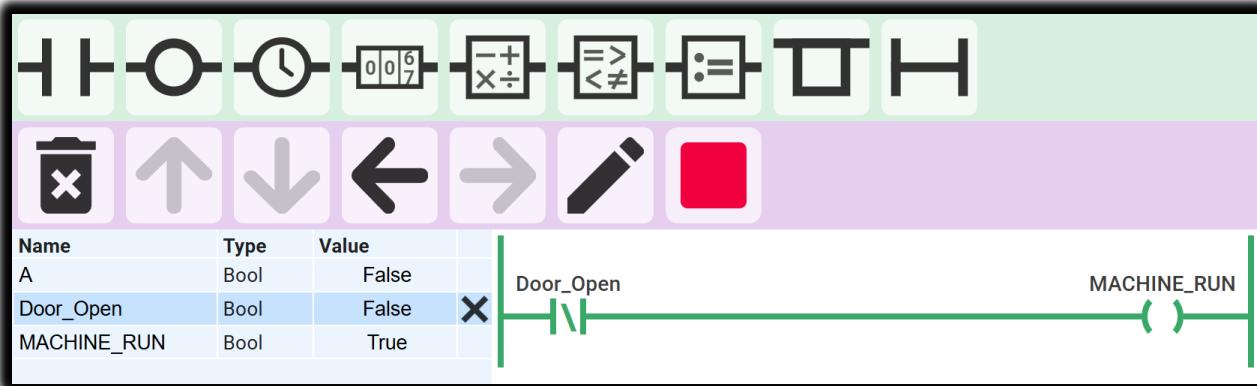


**What it does:** A NOT gate flips the input - if something is ON, it makes the output OFF, and vice versa.

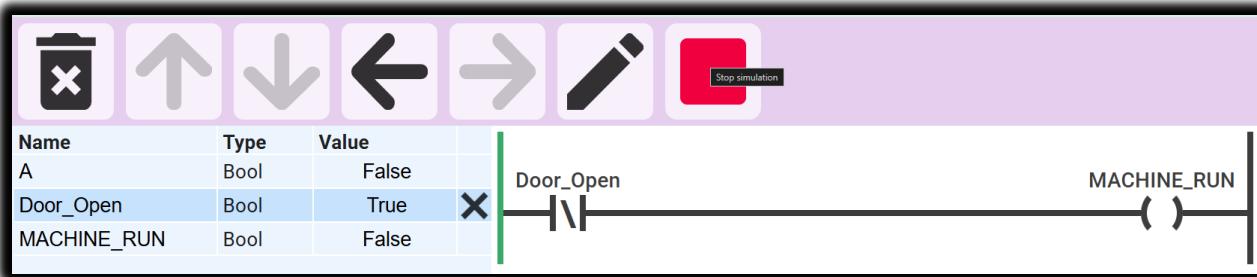
**How to build it in ladder logic:** Use a Normally Closed (NC) contact --|\|--

- When input A is OFF → NC contact is closed → power flows → output turns ON
- When input A is ON → NC contact opens → no power flows → output turns OFF

**Real-world example:** Safety interlock on a machine door:



Door is closed (DOOR\_OPEN = 0 / False / NOT gate active) → machine can run (output = 1)

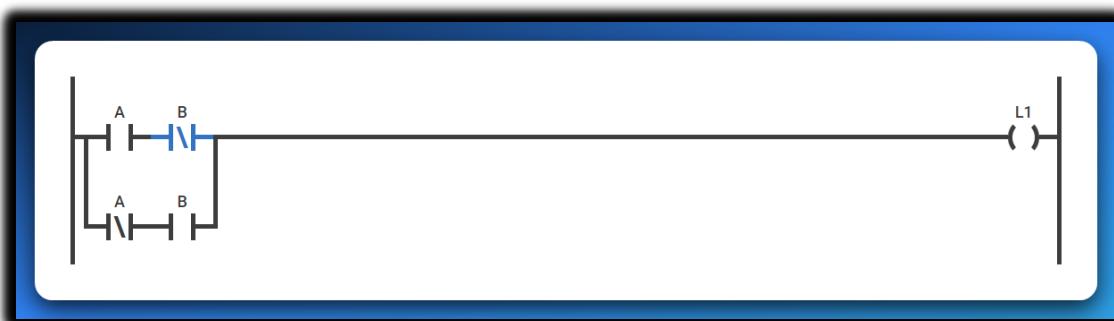
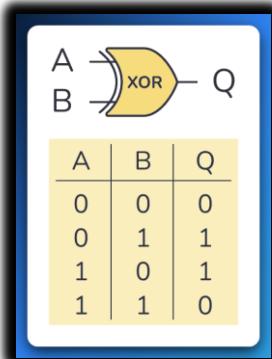


Door is open (DOOR\_OPEN = 1 / True) → machine stops (output = 0)

**In plain English:** "Run the machine only when the door is NOT open." The NC contact automatically inverts the door sensor signal - when the door opens, it immediately cuts power to the machine.

This is the most basic safety logic you'll see everywhere in industrial automation.

## XOR gate Ladder Implementation:



Use a combination of normally open (NO) and normally closed (NC) contacts.  
This represents the logic: **"If A OR B is TRUE, but not both, then turn ON the output."**

### How it works:

- **If A is TRUE and B is FALSE:**  
The path through A (NO) and B (NC) closes → Output turns ON.
- **If B is TRUE and A is FALSE:**  
The path through B (NO) and A (NC) closes → Output turns ON.
- **If both A and B are TRUE:**  
Both paths are blocked because NC contacts open → Output stays OFF.
- **If both A and B are FALSE:**  
Neither NO contact allows current → Output stays OFF.

 **Summary:** Output is ON **only when A or B is TRUE, but not both at the same time** — exactly what XOR logic means.

$$(A \cdot \bar{B}) + (\bar{A} \cdot B) = X$$

## Key Concept: Contacts Are Logic Conditions

In Ladder Logic, your contacts (—| |—) aren't checking voltage directly — they're checking **bit states** in memory.

- A contact in a rung is **like a mini IF-statement**.
- When you place them **in series**, it means "all must be true" (AND).
- When you place them **in parallel**, it means "any can be true" (OR).

And these logic structures determine whether **your output coils** (—( )—) get energized or not.

## Analogy Time:

- Think of series logic like **security clearance**:  
"You need both a keycard **and** a password to enter."
- Think of parallel logic like **alarm triggers**:  
*"If **any** of the sensors trip, sound the alarm."*

## Summary:

- **AND Logic = Series contacts** → All must be TRUE.
- **OR Logic = Parallel contacts** → At least one must be TRUE.
- No separate gate components — it's all about **how you arrange your contacts** in the rung.

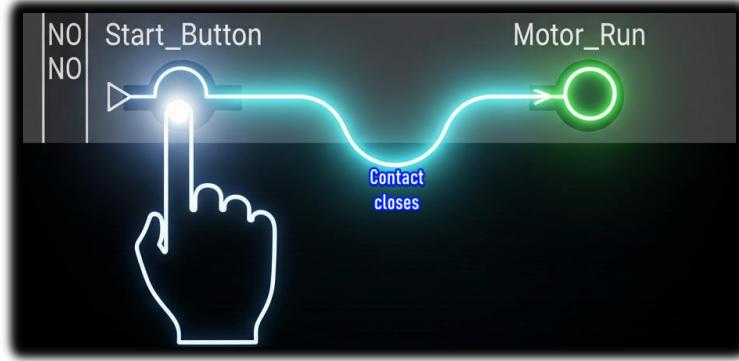
## The contacts

### ☛ Normally Open (NO) Contact – The “Push-to-Start”

Think of a normally open contact like a simple doorbell button.

- **Default (not pressed):** The internal path is open. No current flows. In PLC terms, the memory bit is FALSE (0).
- **When pressed:** The path closes. Electricity flows. In PLC terms, the bit flips to TRUE (1), and logical power can now travel through that rung.

✓ **Analogy:** Press button → bridge closes → current flows → light turns on.

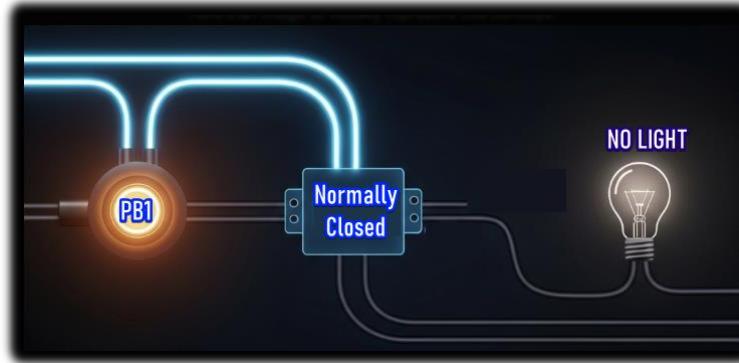


### ☛ Normally Closed (NC) Contact – The “Safety Gate”

Now flip the logic. A normally closed contact is like a safety gate that's already allowing current through until you interfere.

- **Default (not pressed):** The path is closed. Electricity flows. In PLC terms, the memory bit is TRUE (1).
- **When pressed:** The path opens. Current stops. In PLC terms, the bit reads FALSE (0), blocking logical power on that rung.

✓ **Analogy:** Press button → bridge opens → no current → light stays off.



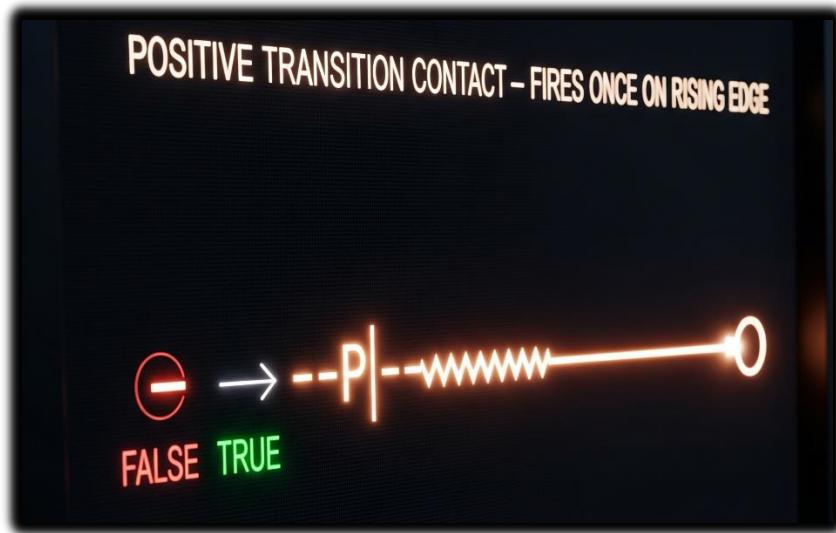
## 💡 Positive Transition-Sensing Contact ---|P|---

**What it does:**  Fires **only for one scan** when your variable flips from FALSE → TRUE  
 Think of it as a **one-shot rising edge detector**  It only passes power if the left input is already TRUE at that moment

### Real-world examples:

- **Start button:** Press to start a machine → you want it to start once, not keep trying to start every scan while held down.
- **Part counter:** When a sensor first detects a part → increment counter once, not continuously while the part passes by.
- **Door opening:** When a door switch goes from closed to open → log one "door opened" event, not hundreds while it stays open.
- **Emergency reset:** After fixing an alarm → acknowledge it once when the reset button is pressed, not repeatedly.

**Why it's essential:** Without this, holding a button would flood your system with repeated commands every scan cycle (potentially thousands of times per second), causing chaos in counters, alarms, and sequences.



## ⚠ Negative Transition-Sensing Contact ---|N|---

**What it does:**  Fires **only for one scan** when your variable flips from TRUE → FALSE  
 A **one-shot falling edge detector**  Left input must be TRUE at that moment too

**Real-world examples:**

- **Conveyor control:** When a box leaves the sensor → start the next conveyor section once, not continuously while the sensor is empty
- **Safety logging:** When an emergency stop is released → log one "system restored" event at the exact moment of release
- **Production tracking:** When a part exits the work station → trigger completion counter once as it leaves, not while the station stays empty
- **Door security:** When a door closes → send one "door secured" signal to the alarm system, not constant signals while it stays closed
- **Process completion:** When a tank empties (level sensor goes FALSE) → start the refill cycle once at that instant

**Why it's crucial:** This catches the exact moment something stops or leaves, preventing your system from missing the transition or triggering multiple times. Perfect for "cleanup" actions that should happen right when something ends.



## OTE (Output Energize) or Output Coil

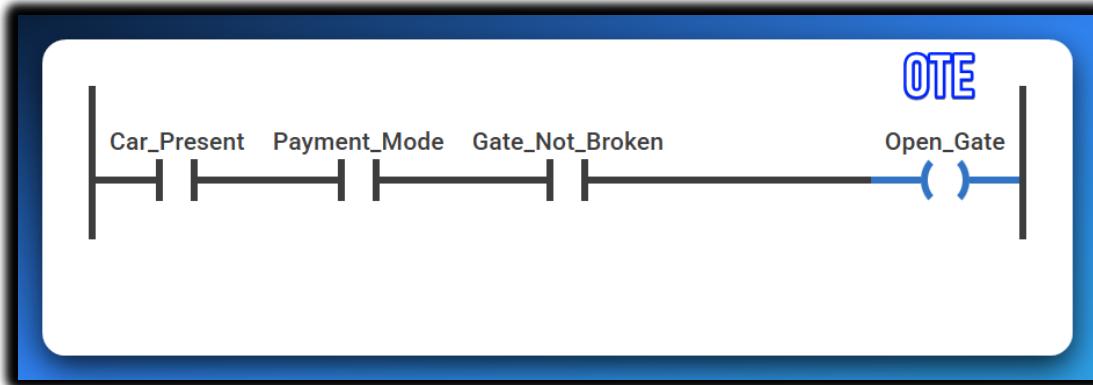
Think of OTE as the "**result**" of your ladder logic rung. It's what actually happens when all your conditions are met.

**Simple analogy:** You know how a regular light switch works? You flip it up → light turns on. You flip it down → light turns off. OTE is exactly like that light bulb, but instead of a physical switch, it's controlled by the logic you build.

**How it really works:** Every time the PLC scans your program (thousands of times per second), it checks: "Are all the conditions on this rung TRUE right now?"

- If YES → OTE energizes (turns ON)
- If NO → OTE de-energizes (turns OFF)

**Real-world example:** Imagine an automatic parking garage gate:



- Car sensor detects a vehicle AND payment was received AND gate isn't broken → gate opens
- Missing any one condition (no car, no payment, gate broken) → gate stays closed
- The gate follows these conditions in real-time - if payment expires while the car is there, gate closes immediately

**Key point:** OTE has zero memory. It's like a faithful dog - it does exactly what you tell it, when you tell it, every single scan. No thinking, no remembering yesterday, just pure obedience to your current logic.

**The OTE (Open\_Gate) is like the "DO IT!" command at the end of the line.** When everything to the left of Open\_Gate is TRUE (like, all the green lights are on and the vibes are right), then Open\_Gate gets powered up, and it stays on as long as those conditions are met.

## ▣ Latch (OTL) and Unlatch (OTU)

### OTL (Output Latch) - The Memory Keeper

**In simple terms:** OTL is like a **sticky switch** - once you turn it ON, it STAYS on until something specifically turns it OFF.

**How it's different from OTE:**

- **OTE:** Acts like a regular light switch - follows your logic constantly
- **OTL:** Acts like a **hotel room keycard** - once activated, it remembers and stays active

**How OTL works:** When the rung logic goes TRUE (even for just one scan), OTL sets the output to ON and then **remembers** that state. Even if the rung logic goes back to FALSE, the output STAYS ON.

**Real-world example:** Emergency alarm system(we'll unlatch it next):



- Smoke sensor triggers for 2 seconds → alarm turns ON
- Smoke clears (sensor goes FALSE) → alarm STAYS ON (latched)
- Alarm keeps blaring until someone manually resets it with an OTU instruction

**Why use latching?** Perfect for situations where you need something to **stay active** even after the trigger condition disappears:

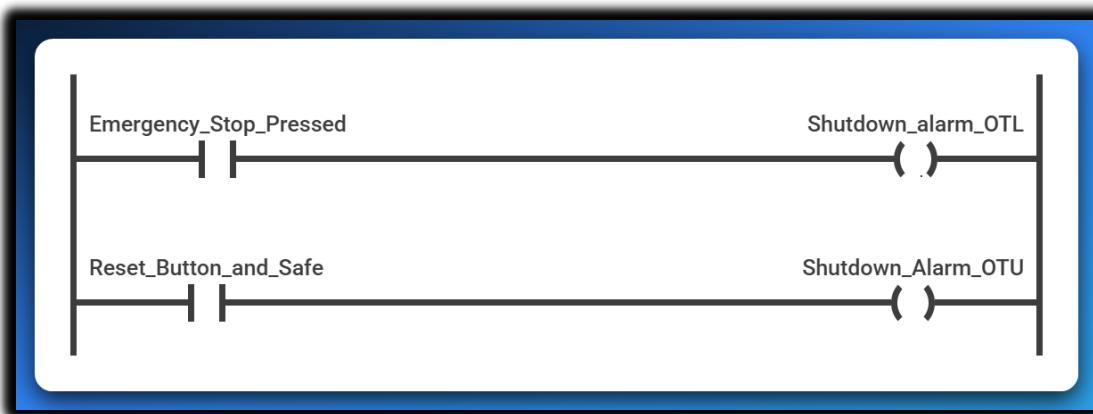
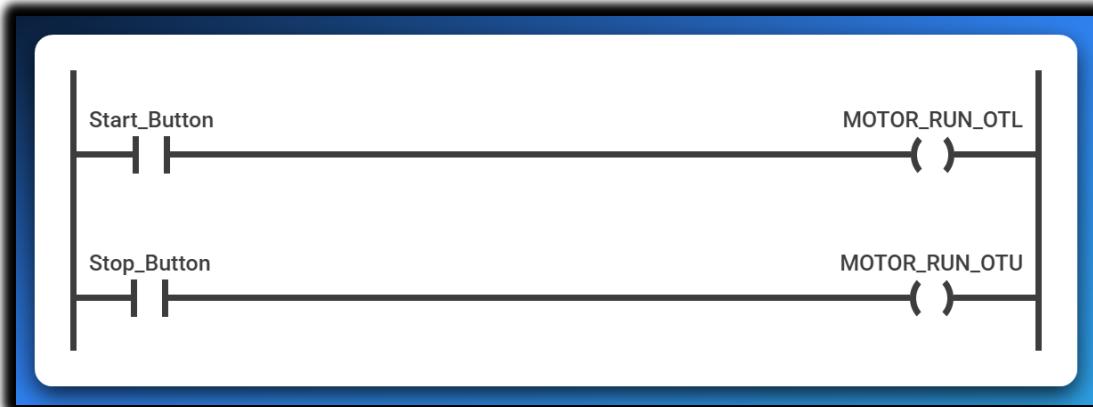
- Emergency alarms that need manual reset
- Equipment that should stay running once started
- Status flags that mark "something happened" until cleared

**Key point:** OTL has **permanent memory** - it's like writing with a pen instead of pencil. Once it's set, only an OTU (Output Unlatch) can erase it.

## OTU (Output Unlatch) - The Reset Button

**In simple terms:** OTU is the **eraser** for latched outputs. It specifically turns OFF outputs that were set by OTL instructions.

**How it works:** When the rung logic goes TRUE, OTU forces the specified output to turn OFF and clears its "memory." It's like hitting a reset button. The OTL/OTU partnership:



- Emergency button pressed → alarm latches ON and STAYS on
- Even if emergency button is released → alarm keeps sounding
- Only when technician presses reset AND system is safe → OTU turns alarm OFF

### Key differences:

- **OTL:** Sets something ON permanently (until reset)
- **OTU:** Sets something OFF permanently (until latched again)
- **OTE:** Just follows current logic (no memory)

**Important:** OTU only affects outputs that were previously latched with OTL. It's like having a specific key that only works on doors that were locked with a matching lock.

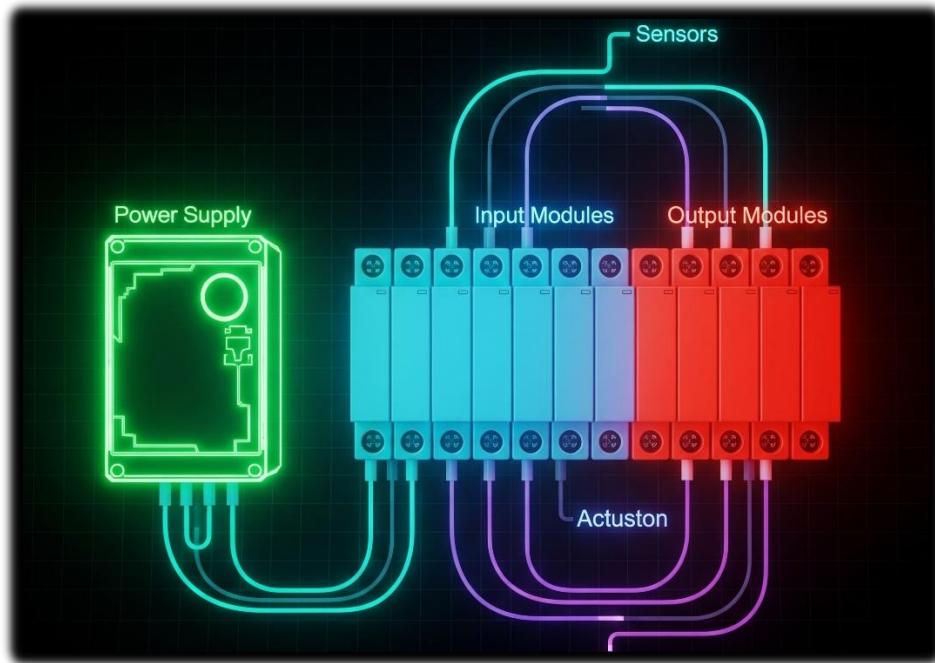
## Bottom line:

NO lets power through *when active*. NC stops power *when active*.

Together, they're the Lego blocks of ladder logic—defining exactly when your rungs conduct or block logical “power.”

Simple, powerful, and everywhere in your PLC world.

A PLC.



By now we're good to go forward with this topic.

## What are Internal Relays (B3, M bits)?

### Internal Relays (Also Called: Internal Coils / Memory Bits / Flags)

An **internal relay** is a **Boolean memory location** inside the PLC used **to store intermediate results or control logic flow**, without being physically tied to any I/O device.

It can be:

- **Turned ON or OFF** (just like a physical output coil)
- **Read using contacts** in other rungs (XIC = Normally Open, XIO = Normally Closed)
- Used across **multiple rungs** in your program to coordinate complex logic

### Key Properties

| FEATURE               | DESCRIPTION   |
|-----------------------|---|
| Type                  | Boolean (TRUE/FALSE)  |
| Physical hardware?    |  No – it's entirely inside the PLC's RAM |
| Can it be written to? |  Yes – via OTE instructions              |
| Can it be read from?  |  Yes – via XIC/XIO contacts              |
| Scope (Siemens vs AB) | M bits (Siemens), B3 file bits (Allen-Bradley)  |
| Typical Use Cases     | Logic steps, interlocks, sequences, memory flags, condition simplification  |

### Siemens:

- **M0.0** → Memory byte 0, bit 0
- Like accessing the first bit in a specific memory byte.

## Internal Relays vs. Physical Coils

| Coil Type                    | Controls Hardware?                      | Visible to Outside World?               | Purpose                                  |
|------------------------------|---|---|--|
| Physical Coil (e.g. O:0/1)   | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes | Turns on a motor, light, solenoid, etc.  |
| Internal Relay (e.g. B3:0/1) | <input checked="" type="checkbox"/> No  | <input checked="" type="checkbox"/> No  | Used to store and route logic internally |

### Real-World Scenario: Intermediate Logic Storage

Let's say your system needs to:

- Check if **Machine Ready**
- Check if **Safety Conditions Met**
- Check if **Operator Pressed Start**

Rather than one giant rung, we can break this down:

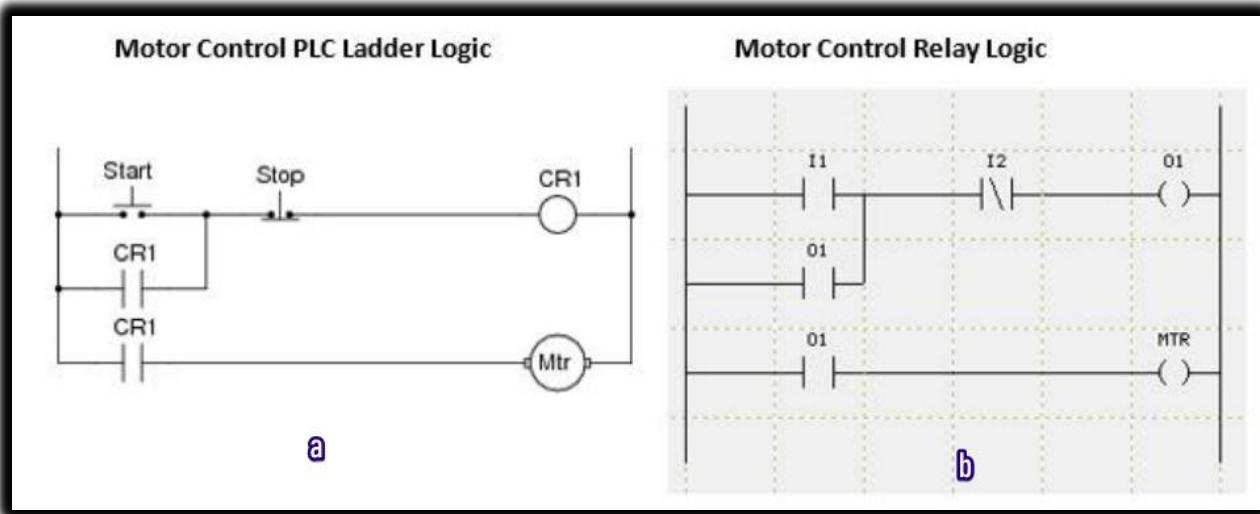
## LADDER LOGIC PART 3

### Elements of ladder logic

- **Rails** – Vertical lines that carry electrical power to the control circuit.
- **Rungs** – Horizontal lines where the logic is built; they contain inputs, outputs, and branches.
- **Branches** – Parallel paths on a rung that allow multiple input conditions.
- **Inputs** – Devices like switches or sensors that control the logic flow.
- **Outputs** – Devices like motors or lights activated by the logic.
- **Timer** – Delays actions for a set time (e.g., turn on a light after 5 seconds).
- **Counter** – Counts events or cycles and triggers actions after a set number.

I see a small naming confusion I have always had.

In my school, we used “Ladder Logic” to refer to the programs we were drawing.



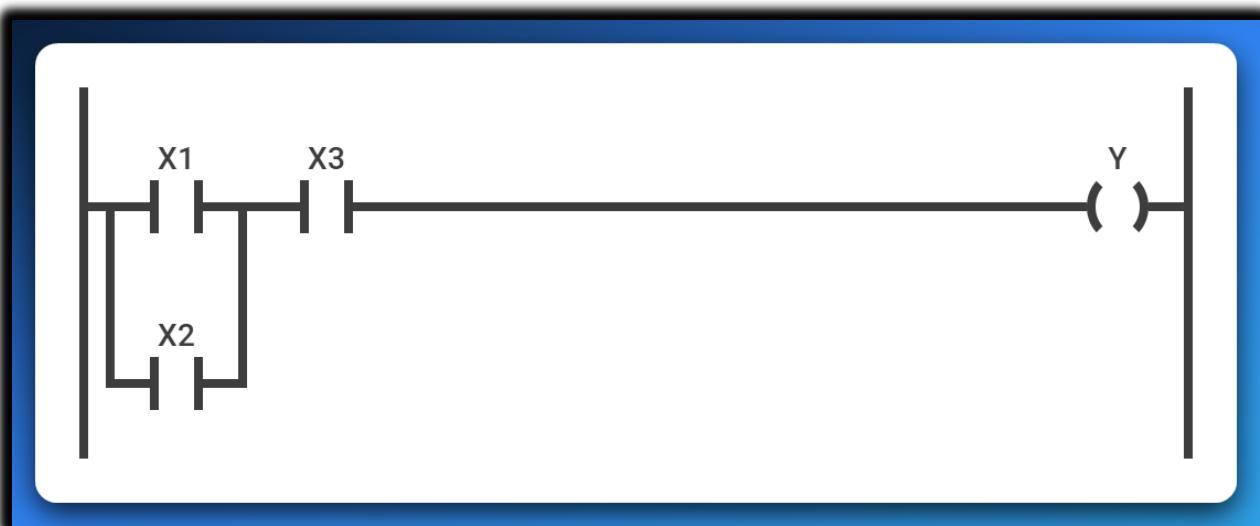
So, I had this conflict calling the **image b** ladder logic.

Now am seeing it being called **relay logic**.

- **Diagram 'a' (Motor Control PLC Ladder Logic):** This is a **program** that runs inside a PLC. It's software.
- **Diagram 'b' (Motor Control Relay Logic):** This is a **wiring diagram** for physical electrical components (relays, switches, motor). It's hardware.

## Some Practice

$$Y = (X_1 + X_2)X_3$$



Read it using the AND, OR gates.

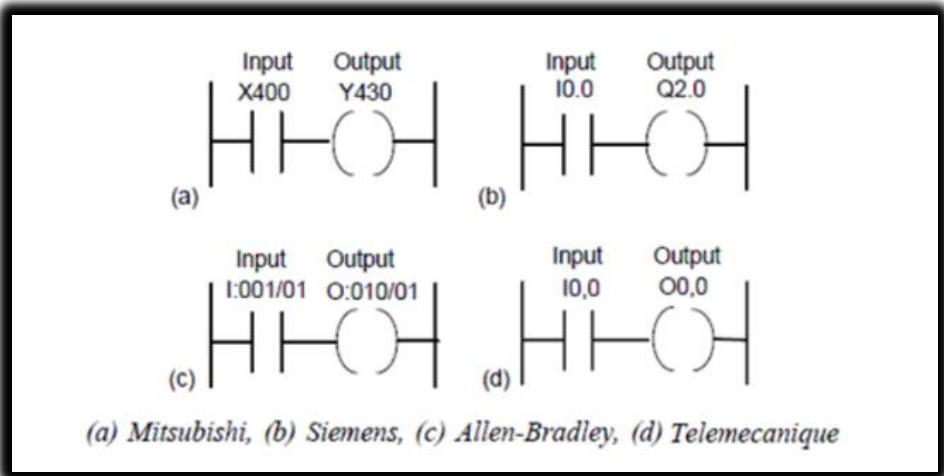
$$Y = (X_1 + X_2)(X_3 + X_4)$$



You could attach the midpoint lines as one, or separate them as above.

$$Y = (X_1 X_2) + X_3$$





## What are these diagrams called?

They are called **Ladder Diagrams** or **Ladder Logic**. They are a programming language used to program a **Programmable Logic Controller (PLC)**.

## What's it doing?

Each diagram represents a simple circuit. The vertical lines on the left and right are like power rails. The horizontal lines are called **rungs**. In a simple rung like these, a condition (the input) must be met for an action (the output) to occur.

## What are these rungs?

A "rung" is a single line of logic in a ladder diagram. Think of it like a rung on a ladder. Each rung represents a control circuit. The logic flows from left to right.

## What for?

Ladder logic is used to automate industrial processes. PLCs are essentially rugged computers that are used to control machinery on factory floors, such as conveyors, robotic arms, and assembly lines. The ladder logic program tells the PLC exactly how to control the physical inputs and outputs of the machine.

## A breakdown of the components:

### Input (X400, I0.0, etc.):

These are conditions that must be true for the circuit to be complete. They are typically contacts from a physical input device, like a push-button, a sensor, or a switch.

The symbol shown is an "examine if closed" or "normally open" contact. It's "closed" (and lets power through) when the input is activated.

### Output (Y430, Q2.0, etc.):

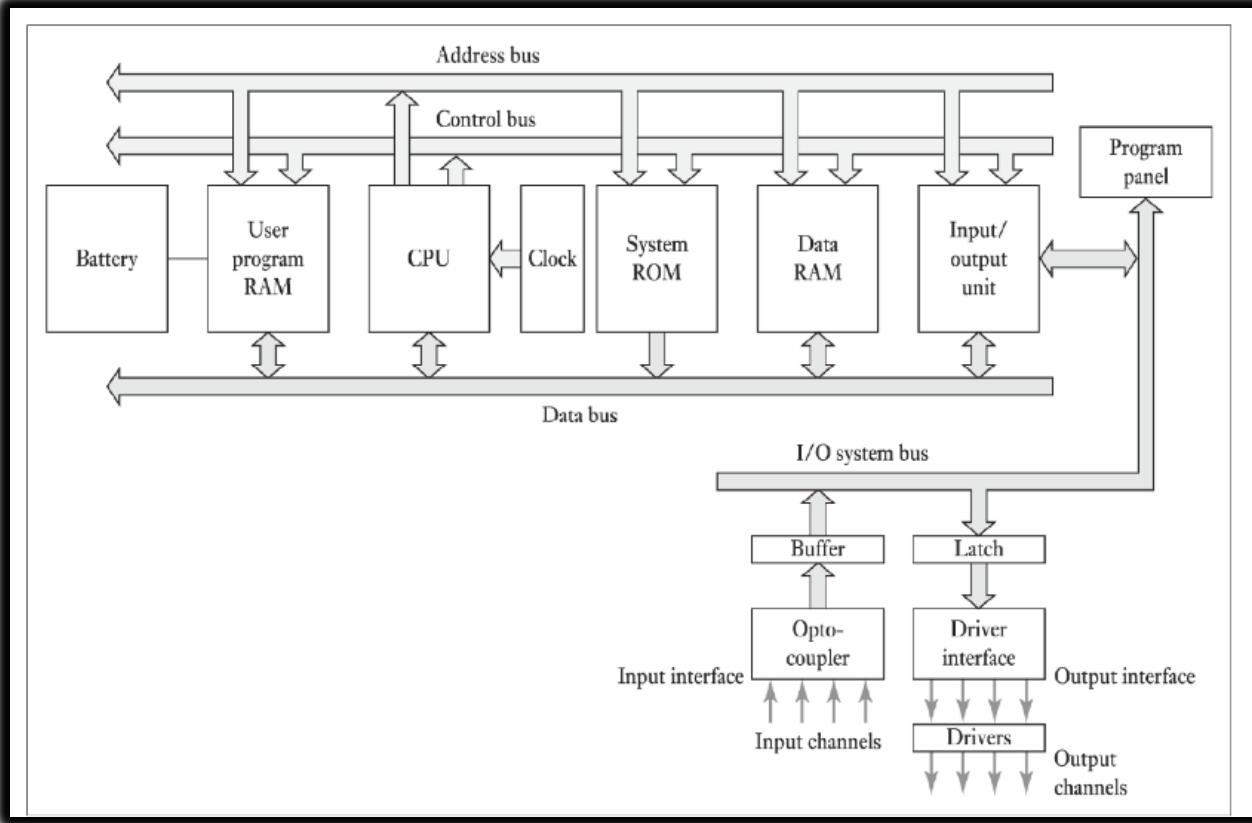
These are the actions that will happen when the input conditions on that rung are met. They represent a physical output device, like a motor, a light, or a solenoid.

The symbol shown is an "energize coil." When the rung is "true," the coil is energized, turning on the corresponding output device.

The different labels (a, b, c, d) show how different PLC manufacturers, like Mitsubishi, Siemens, Allen-Bradley, and Telemecanique, use slightly different naming conventions for their inputs and outputs, even though the fundamental logic is the same.

For example, Siemens uses 'I' for input and 'Q' for output, while Mitsubishi uses 'X' and 'Y'.

The logic itself, however, is universal.



This shows the internal architecture of a PLC, which is the foundation for how it runs the ladder logic programs you saw earlier.

Knowing these parts helps you understand why PLCs work the way they do and can be **very useful for troubleshooting**.

### CPU (Central Processing Unit):

The "brain" of the PLC. It's the microprocessor that executes the user program, processes input signals, and updates output signals.

## **Memory:**

This is where the PLC stores its program and data. There are a few different types shown:

- **User Program RAM:** This is where your ladder logic program is stored. The CPU reads instructions from here.
- **Data RAM:** This is where the PLC stores the current status of inputs, outputs, and any internal variables or data used by the program.
- **System ROM:** This holds the PLC's operating system, which is the permanent software that controls how the PLC runs.

## **Buses (Address, Control, Data):**

These are the internal communication highways of the PLC.

- **Data Bus:** Carries the actual data (instructions, values) between the CPU and the memory and I/O units.
- **Address Bus:** Specifies the location (address) in memory or the I/O unit that the CPU wants to read from or write to.
- **Control Bus:** Carries signals that control the timing and flow of data, like read/write commands.

## **Input/Output (I/O) Unit:**

This is how the PLC connects to the real world.

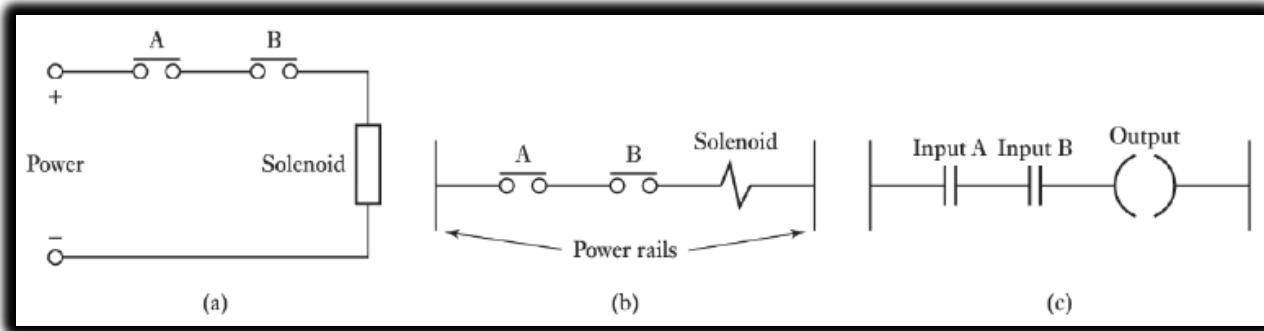
- **Input Unit:** Takes signals from external devices (sensors, switches) and converts them into a format the CPU can understand. It's shown with an **Opto-coupler**, which is an important component that uses light to electrically isolate the sensitive internal PLC circuitry from the external input signals, protecting it from electrical noise and voltage spikes.
- **Output Unit:** Takes signals from the CPU and converts them into a format that can control external devices (motors, lights). It often uses **Drivers** (like transistors or relays) to handle the higher voltage/current needed for these devices. The **Latch** shown helps hold the output state.

## Power Supply (Battery):

This provides power to the PLC's internal components. The battery is important because it often maintains the contents of the User Program RAM when the main power is off.

*In an exam, you might be asked to describe the function of these components or explain the flow of information during a typical PLC scan cycle.*

*The PLC scan cycle is a key concept that explains how the PLC reads inputs, executes the program, and updates outputs, and this diagram perfectly illustrates the hardware that makes that process possible.*



This image shows how a simple electrical circuit translates into ladder logic—a neat way to see how ladder diagrams were designed to feel familiar to anyone who understands traditional electrical relay schematics.

### Diagram (a): Standard Electrical Relay Circuit

This is your classic electrical schematic.

The solenoid (basically an electromagnet) only turns on when there's a complete path for current from the positive (+) to the negative (-) terminal.

The current path is controlled by switches A and B, which are wired in series.

In plain terms, *both* switches must be closed for the solenoid to activate. This setup is the physical equivalent of an **AND** logic gate.

## Diagram (b): Traditional Relay Logic Diagram

This is an old-school way of representing the same circuit, often used before PLCs came into the picture.

- The two vertical lines are the “power rails” (your positive and negative sources).
- The horizontal line between them is the “rung.”
- Switches A and B appear as “normally open” contacts, while the solenoid is shown as a coil.

This style is almost identical to what you’ll see in modern ladder logic, which is the next step.

## Diagram (c): PLC Ladder Logic Diagram

Here’s the PLC-friendly version. The logic hasn’t changed, just the symbols.

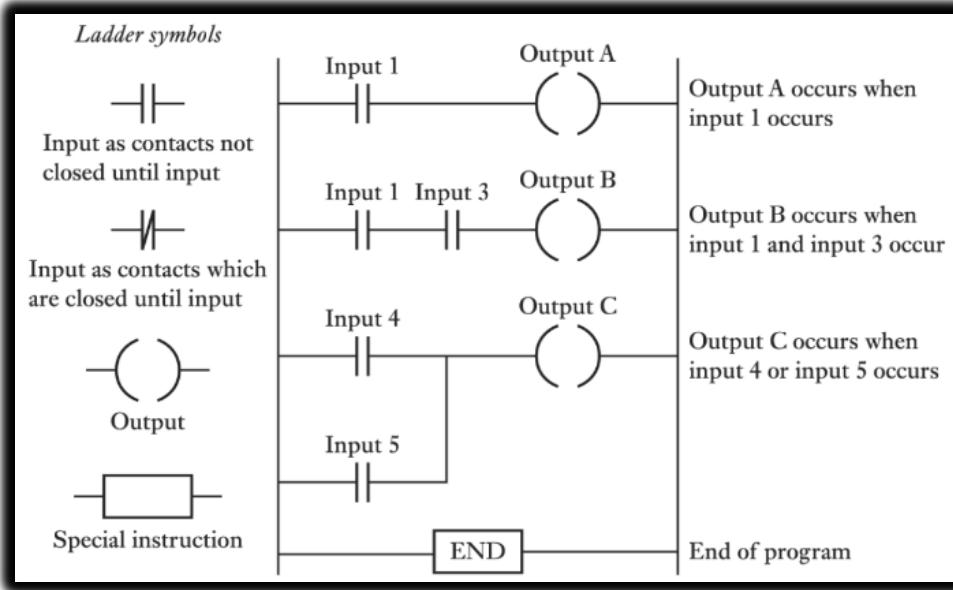
- **Inputs A and B** are represented as “normally open” contacts, wired in series. The PLC will only mark this rung as “true” if both inputs are active.
- **The output** is the coil symbol, which the PLC energizes (turns on) only when the rung condition is true.

## Big Picture

All three diagrams express the exact same rule:

**The output will activate only if Input A AND Input B are active.**

That’s the core principle of ladder logic—series connections always represent an **AND** condition.



Thanks, glad they're helpful! This image is a great summary of the basic symbols and logic gates used in PLC ladder programming.

## Basic Symbols (on the left)

### --| |-- (Normally Open Contact)

This is the **NO contact** symbol. By default, it's "open," meaning no current flows through it. It only "closes" (lets current pass) when the corresponding physical input—like a button or sensor—is activated. This is the most common type of input symbol you'll see.

### --| / |-- (Normally Closed Contact)

This is the **NC contact** symbol. By default, it's "closed," so current flows. It "opens" (stops the current) when the linked physical input is turned on. You'll often see this in safety circuits, like an emergency stop, where the system should stay active until the button is pressed.

### --( )-- (Output Coil)

This symbol represents an **output coil**. If the logic on the rung to its left is "true" (current can flow), the coil becomes energized, turning on the corresponding device—like a motor, light, or solenoid.

### --[ ]-- (Special Instruction)

This is a catch-all symbol for **special functions or instructions**. These are used for more advanced operations—things like timers, counters, math operations, or data handling inside the PLC program.

## Ladder Logic Rungs (on the right)

The right side shows three different rungs of logic, each with a corresponding explanation:

### Rung 1: Input 1 -> Output A

- This is the simplest form of logic.
- It uses a single **Normally Open** contact for Input 1.
- **Logic:** Output A will be energized **IF** Input 1 is turned on.
- This is a simple "direct control" or "Copy" logic.

### Rung 2: Input 1 -> Input 3 -> Output B

- This rung has two Normally Open contacts connected in series.
- Logic: Output B will be energized **IF** Input 1 is turned on **AND** Input 3 is turned on.
- This is the representation of an AND logic gate. Both conditions must be true for the output to activate.

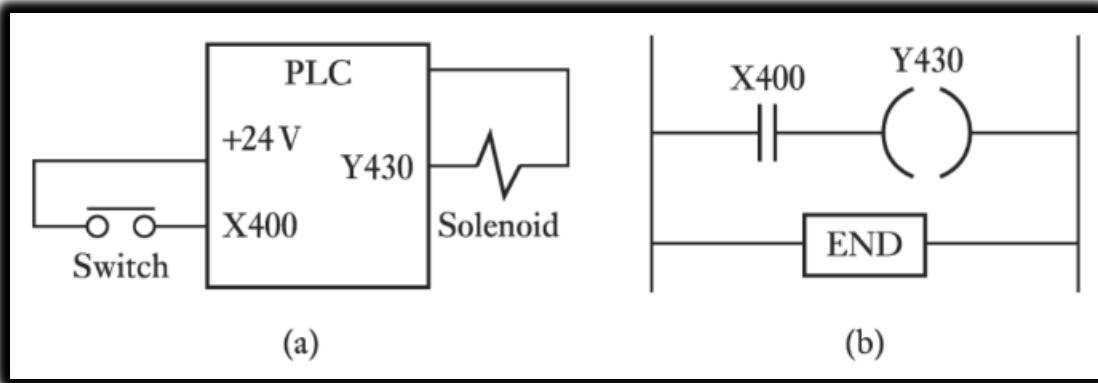
### Rung 3: Input 4 and Input 5 -> Output C

- This rung shows a different connection: two Normally Open contacts connected in parallel.
- Logic: Output C will be energized **IF** Input 4 is turned on **OR** Input 5 is turned on.
- This is the representation of an OR logic gate. The output will activate if at least one of the conditions is true.

## The END instruction

Signifies the end of the program. The PLC's scan cycle runs through all the rungs from top to bottom, then repeats.

This "END" tells the PLC that it has reached the end of the user-programmed logic.



### Diagram (a): The Physical Wiring

This diagram shows how the real-world components are hooked up to the PLC.

- **PLC:** The brain of the operation. It's a box with terminals for power, inputs, and outputs.
- **+24V:** The power source feeding both the PLC's internal logic and the connected devices.
- **X400 (Input Terminal):** A physical input point on the PLC. The switch is wired here along with the power source. When the switch is pressed, it sends a signal to X400. The PLC's hardware senses this change instantly.
- **Y430 (Output Terminal):** A physical output point on the PLC. The solenoid is wired here. When the PLC energizes Y430, power flows to the solenoid, causing it to activate.

### Diagram (b): The Ladder Logic Program

This is the “software version” of the hardware logic shown above.

- **--| |-- X400:** A Normally Open contact that mirrors the state of the physical switch on X400. When the switch is pressed, this contact “closes” in the program, allowing logic to flow through the rung.
- **--( )-- Y430:** The output coil symbol. If the rung's logic is true, this coil energizes the physical output Y430—powering the solenoid.
- **END:** Marks the end of the ladder logic program.

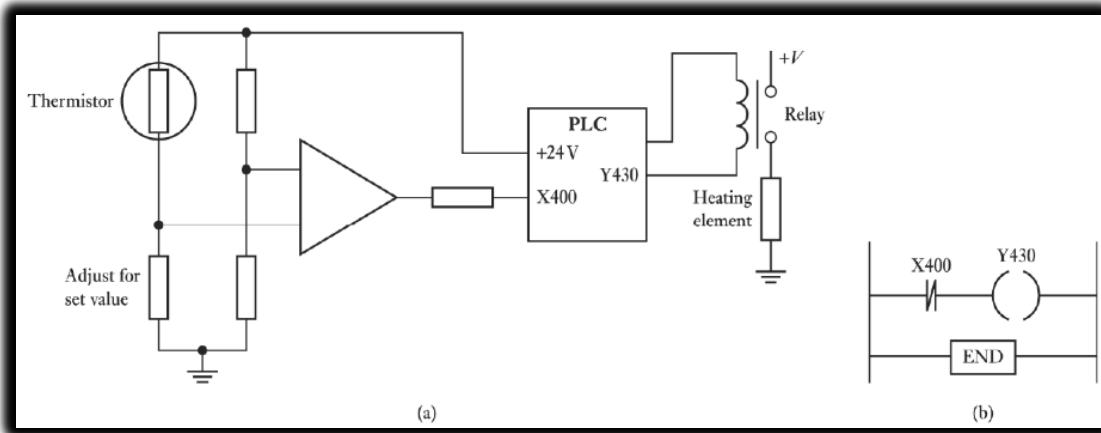
## How They Work Together

The hardware wiring provides the raw electrical connections, while the ladder logic decides what to do with those signals.

In this example, the rule is simple:

**IF the switch (X400) is ON, THEN energize the solenoid (Y430).**

The PLC constantly scans the input (X400), evaluates the logic, and updates the output (Y430) in real-time.



This image steps it up with a **real-world control loop**—combining an analog temperature sensor with a PLC's digital input to manage a heating element. This kind of ON/OFF control is everywhere in industrial systems, from ovens to HVAC units.

## Diagram (a): Physical Wiring and Analog Circuit

### Thermistor & Analog Comparator:

A **thermistor** is a temperature-sensitive resistor. Its resistance shifts with temperature changes.

It's wired into a **Wheatstone bridge** (those two parallel resistor dividers). On the other side of the bridge, a **variable resistor**—labeled “*Adjust for set value*”—lets you set the target temperature.

The **triangle symbol** is an op-amp (operational amplifier), acting as a **comparator**. It compares the voltage from the thermistor side to the voltage from the “set value” side.

When the thermistor detects the set temperature, the op-amp's output flips state (e.g., low → high), signaling that the threshold has been hit.

## PLC Integration:

- The **op-amp's output** connects to the PLC input **X400**. The PLC doesn't "know" the temperature; it just sees ON/OFF signals from the comparator.
- The PLC output **Y430** drives a **relay**, which works like an electrically controlled switch.
- When Y430 is energized, the relay closes its contacts, letting high-voltage power (+V) flow to the **heating element**. The relay keeps the heavy current away from the PLC while safely controlling big loads.

## Diagram (b): Ladder Logic Program

- **--| |-- X400:** Represents the ON/OFF signal from the op-amp comparator.
- **--( )-- Y430:** The output coil. When the rung is true, Y430 energizes the relay—and the heater kicks on.

The logic is identical to the earlier examples:

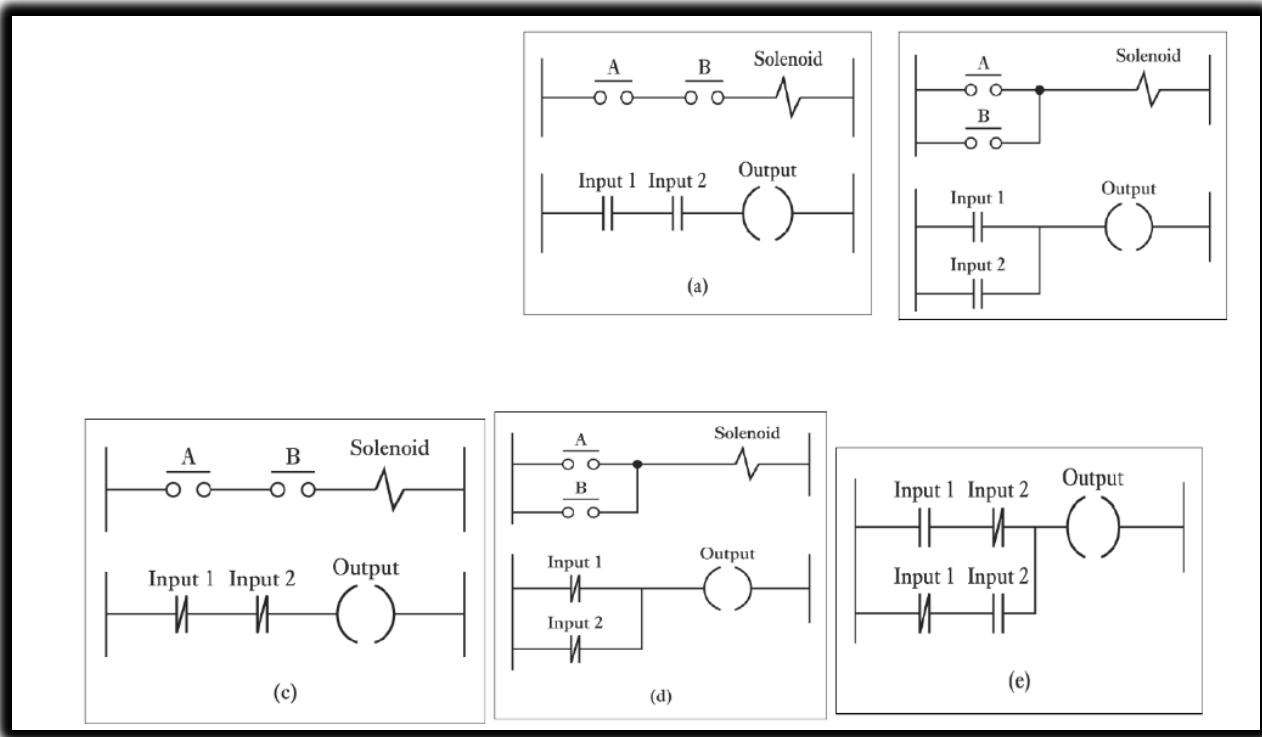
**IF X400 is ON, THEN energize Y430.**

## The Purpose of the Circuit

This forms a **basic temperature control system**:

- You set a target temperature with the "*Adjust for set value*" resistor.
- The thermistor constantly monitors the actual temperature.
- If the temperature drops below the setpoint, the op-amp output flips OFF. The PLC sees X400 as false, de-energizing Y430 and turning the heater off.
- If the temperature rises past the setpoint, the op-amp output turns ON. The PLC sees X400 as true, energizing Y430, which powers the heater.

This is called **bang-bang control** (or ON/OFF control): the heater is either fully ON or fully OFF—simple, reliable, and common in industrial automation.



This image is a fantastic summary that combines all the core concepts we've discussed so far. It shows pairs of traditional relay logic diagrams and their corresponding PLC ladder logic diagrams, illustrating how the same logic is represented in both formats.

Let's break down each pair:

### (a) Series Connection: AND Logic

- **Traditional:** The contacts A and B are connected one after the other in a **series**.
- **PLC:** Input 1 and Input 2 are represented by normally open contacts, also connected in **series**.
- **Logic:** The solenoid (or output) will energize **IF AND ONLY IF A AND B are both closed**. This is the **AND** logic gate.

### (b) Parallel Connection: OR Logic

- **Traditional:** The contacts A and B are connected in **parallel**, meaning there are two separate paths for the current to flow.
- **PLC:** Input 1 and Input 2 are represented by normally open contacts, connected in **parallel** on the rung.
- **Logic:** The solenoid (or output) will energize **IF A is closed OR B is closed (or both)**. This is the **OR** logic gate.

### (c) Series Connection with Normally Closed: NOT AND (NAND) Logic

- **Traditional:** The contacts A and B are normally closed (NC) and are in **series**.
- **PLC:** Input 1 and Input 2 are represented by **normally closed** contacts, also in **series**.
- **Logic:** The output will energize **IF** Input 1 is **NOT** activated **AND** Input 2 is **NOT** activated. This is a **NAND** gate in a non-standard form. In a more standard NAND gate, both inputs must be true for the output to be false. Here, both inputs must be false for the output to be true.

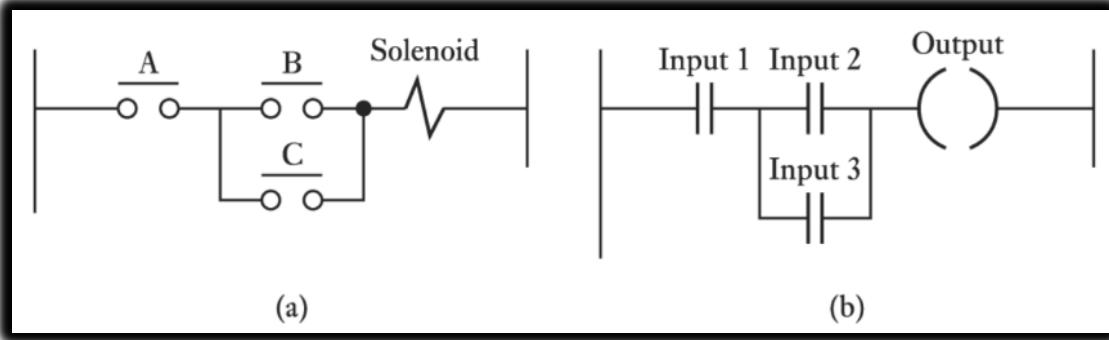
### (d) Parallel Connection with Normally Closed: NOT OR (NOR) Logic

- **Traditional:** The contacts A and B are normally closed (NC) and are connected in **parallel**.
- **PLC:** Input 1 and Input 2 are represented by **normally closed** contacts, connected in **parallel**.
- **Logic:** The output will energize **IF** Input 1 is **NOT** activated **AND** Input 2 is **NOT** activated. The current path is broken if **EITHER** Input 1 **OR** Input 2 is activated. This is a **NOR** gate.

### (e) Combined Logic (AND-OR)

- **PLC:** This shows a combined logic circuit with a parallel branch containing a series branch.
- **Logic:** This is a more complex rung. The output will be energized **IF** (Input 1 **AND** Input 2 are active) **OR** (Input 1 **AND** Input 2 are **NOT** active). This looks a bit like an XNOR gate, where the output is true if both inputs are the same.

The key takeaway from this image is that **series connections in ladder logic represent an AND condition**, while **parallel connections represent an OR condition**. The type of contact used (normally open or normally closed) determines whether you are looking for a condition to be TRUE or FALSE.



This diagram is another great example of combining the concepts of series and parallel logic to create a more complex control circuit. It shows a common arrangement for controlling a single output with multiple inputs.

Here's a breakdown of the two diagrams:

### Diagram (a): The Traditional Relay Logic Diagram

- **Series connection:** Contact A is in series with the rest of the circuit. This means contact A **must** be closed for power to reach the solenoid.
- **Parallel connection:** Contacts B and C are connected in parallel with each other. This means either contact B **OR** contact C can be closed to allow power to pass through this part of the circuit.
- **Logic:** The solenoid will be energized **IF AND ONLY IF** (A is closed) **AND** (B is closed **OR** C is closed).

### Diagram (b): The PLC Ladder Logic Diagram

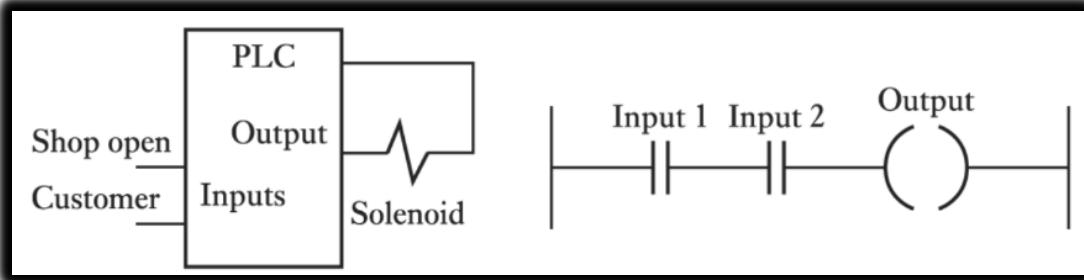
- **Series connection:** The Input 1 contact is in series with the parallel branch. This corresponds to the A contact from the relay logic.
- **Parallel connection:** The Input 2 and Input 3 contacts are connected in parallel. This corresponds to the B and C contacts.
- **Logic:** The Output coil will be energized **IF AND ONLY IF** (Input 1 is turned on) **AND** (Input 2 is turned on **OR** Input 3 is turned on).

## The Purpose of This Logic

This type of circuit is often used for a "master control" and "local control" scenario. For example:

- **A/Input 1** could be a master enable switch or a safety gate sensor. The entire system cannot run unless this master condition is met.
- **B/Input 2** could be a "manual start" button on a control panel.
- **C/Input 3** could be an automatic signal from another part of the system, like a sensor indicating a product is in position.

The logic ensures that the Output (e.g., a motor or a hydraulic cylinder) will only activate if the master switch is on **AND** either the manual button is pressed **OR** the automatic signal is received.



This final image is another excellent example of a real-world scenario, combining the physical wiring with the ladder logic program to control an output based on two simple conditions.

Here's a breakdown of the two parts:

### Diagram (a): The Physical Wiring

- **PLC:** This is the programmable logic controller.
- **Inputs:** The two physical inputs are labeled "Shop open" and "Customer."
  - The "**Shop open**" input is likely connected to a switch that's closed when the shop is open.
  - The "**Customer**" input is likely connected to a sensor (like a motion sensor or a pressure mat) that detects when a customer is present.
- **Output:** The physical output is connected to a **Solenoid**. A solenoid is often used to operate things like an automatic door lock or a buzzer. In this scenario, it's probably the latter—a buzzer or chime to alert a shopkeeper.



## Diagram (b): The Ladder Logic Program

- **--| |-- Input 1:** This is the normally open contact for the first physical input, "Shop open."
- **--| |-- Input 2:** This is the normally open contact for the second physical input, "Customer."
- **--( )-- Output:** This is the output coil that controls the solenoid.
- **Logic:** The two input contacts are in **series**. This means the logic is an **AND** condition. The rung will be "true" and the output will be energized **IF AND ONLY IF** Input 1 is on **AND** Input 2 is on.

## The Purpose of This Circuit

This setup creates a **simple automatic customer alert system** for a shopkeeper. The logic is:

**IF** the "Shop open" switch is closed **AND** a "Customer" sensor detects someone, **THEN** activate the "Output" (the solenoid, which could be a buzzer).



This prevents the buzzer from going off at night when the shop is closed. The "Shop open" switch acts as a master enable, and the "Customer" sensor is the trigger. The PLC's job is to continuously monitor these two inputs and, based on the AND logic programmed in the rung, activate the output at the right time.

| IEC 1131-3 | Mitsubishi | OMRON   | Siemens | Operation                                  | Ladder diagram                          |
|------------|------------|---------|---------|--|---|
| LD         | LD         | LD      | A       | Load operand into result register          | Start a rung with open contacts         |
| LDN        | LDI        | LD NOT  | AN      | Load negative operand into result register | Start a rung with closed contacts       |
| AND        | AND        | AND     | A       | Boolean AND                                | A series element with open contacts     |
| ANDN       | ANI        | AND NOT | AN      | Boolean AND with negative operand          | A series element with closed contacts   |
| OR         | OR         | OR      | O       | Boolean OR                                 | A parallel element with open contacts   |
| ORN        | ORI        | OR NOT  | ON      | Boolean OR with negative operand           | A parallel element with closed contacts |
| ST         | OUT        | OUT     | =       | Store result register into operand         | An output from a rung                   |

Here are some other **must-know concepts** that you'll run into as you dive deeper:

## 1. Memory Coils and Internal Bits

The table mostly focuses on physical inputs (like buttons) and outputs (like lights or motors), but PLCs also use **internal memory locations**—called **memory coils** or **internal bits** (labels like M, C, or T depending on the brand).

Think of these as **virtual outputs**: they don't control any physical device, but they store ON/OFF states inside the program. These are essential for building **latching circuits** or holding conditions active across multiple rungs.

## 2. Timers and Counters

These are the bread and butter of advanced control:

- **Timers:** Measure time. For example, you might trigger a light that stays on for 10 seconds after a button is pressed.
- **Counters:** Count events. A common example is counting how many items pass a sensor on a conveyor.

The table mentions “special instructions,” but timers and counters are the *go-to examples* of these.

### 3. Latching and Unlatching

This is a **core concept** you can't ignore.

A **latching circuit** (or *seal-in/holding circuit*) keeps an output **ON** even after the start button is released. It achieves this by "feeding" the output back into its own rung in parallel with the start signal.

The opposite is the **unlatch instruction**, which turns the output **OFF**. You'll see this pattern everywhere—especially for start/stop motor control.

### 4. Analog Inputs and Outputs

In a previous example, we saw an analog sensor feeding a digital input, but PLCs can handle **true analog signals**—values that aren't just ON/OFF but exist in ranges, like **0-10V** or **4-20mA**.

Working with analog signals often involves:

- **Reading the value** from an analog input module.
- **Scaling the value** (e.g., converting 0-10V to 0-100°C).
- **Using logic** to trigger actions based on thresholds or ranges.

### 5. Data Movement and Comparison

In many real-world systems, you'll need instructions to **move data** between memory locations or **compare values**.

For example:

- Move a number from one register to another (like copying a sensor reading).
- Compare if a counter has hit a certain target and trigger an alarm.

## 6. The PLC Scan Cycle

This is the **heartbeat of every PLC**.

The PLC continuously:

1. Reads all the input states.
2. Executes the logic program, rung by rung (top to bottom).
3. Updates all the outputs.

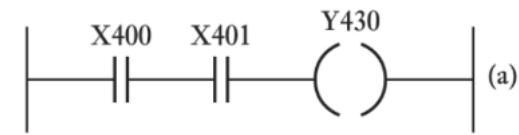
This loop—called the **scan cycle**—runs endlessly.

Understanding how it works is key to predicting your program's behavior, especially when working with fast-changing inputs.

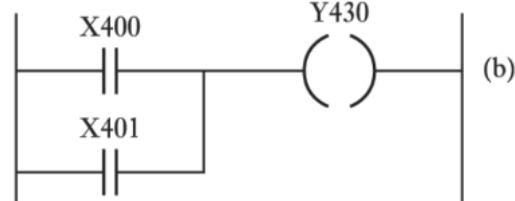
## The Bottom Line

The table is a **fantastic starting point**, but PLC programming is a huge iceberg. As you level up, you'll need to learn how to handle **advanced instructions**, **data manipulation**, and **program structure** to build reliable and flexible control logic.

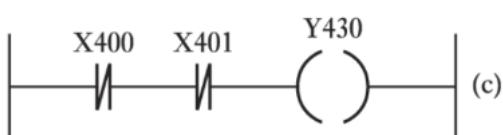
If you read the table above, you should be able to understand these.



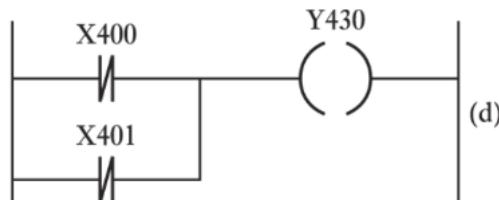
LD X400 (\*Input at address X400\*)  
AND X401 (\*AND input at address X401\*)  
OUT Y430 (\*Output to address Y430\*)



LD X400 (\*Input at address X400\*)  
OR X401 (\*OR input at address X401\*)  
OUT Y430 (\*Output to address Y430\*)



LDI X400 (\*NOT input at address X400\*)  
ANI X401 (\*AND NOT input at address X401\*)  
OUT Y430 (\*Output to address Y430\*)



LDI X400 (\*NOT input at address X400\*)  
ORI X401 (\*OR NOT input at address X401\*)  
OUT Y430 (\*Output to address Y430\*)

This image is an excellent summary that ties together the ladder logic diagrams with the corresponding **Instruction List (IL)** code.

The **Instruction List** is a low-level, text-based programming language for PLCs that directly corresponds to the graphical ladder logic. This image shows the fundamental logic gates in both formats.

Let's break down each example:

### (a) AND Logic

**Ladder Logic:** X400 and X401 are in **series** before the output Y430.

**Logic:** The output Y430 will be energized **IF** X400 is on **AND** X401 is on.

#### Instruction List (IL):

- **LD X400:** Load the state of input X400 into a temporary memory area called the result register.
- **AND X401:** Perform a boolean **AND** operation between the current value in the result register and the state of input X401.
- **OUT Y430:** Store the final result from the register to the output Y430.

**Summary:** LD starts the rung, AND adds a series element, and OUT controls the output.

### (b) OR Logic

**Ladder Logic:** X400 and X401 are in **parallel** before the output Y430.

**Logic:** The output Y430 will be energized **IF** X400 is on **OR** X401 is on.

#### Instruction List (IL):

- **LD X400:** Load the state of input X400.
- **OR X401:** Perform a boolean **OR** operation between the result register and X401.
- **OUT Y430:** Store the final result to the output Y430.

**Summary:** LD starts the rung, OR adds a parallel branch, and OUT controls the output.

### (c) NAND Logic (using NOT contacts)

**Ladder Logic:** X400 and X401 are **normally closed** contacts in **series** before the output Y430.

**Logic:** The output Y430 will be energized **IF** X400 is **NOT** on **AND** X401 is **NOT** on.

**Instruction List (IL):**

- **LDI X400:** Load the **inverse** (or **NOT**) of the state of input X400.
- **ANI X401:** Perform a boolean **AND** operation with the **inverse** of X401.
- **OUT Y430:** Store the result to output Y430.

**Summary:** LDI and ANI correspond to using normally closed contacts in the ladder diagram.

### (d) NOR Logic (using NOT contacts)

**Ladder Logic:** X400 and X401 are **normally closed** contacts in **parallel** before the output Y430.

**Logic:** The output Y430 will be energized **IF** X400 is **NOT** on **AND** X401 is **NOT** on. The circuit path is broken if **either** X400 **OR** X401 is on.

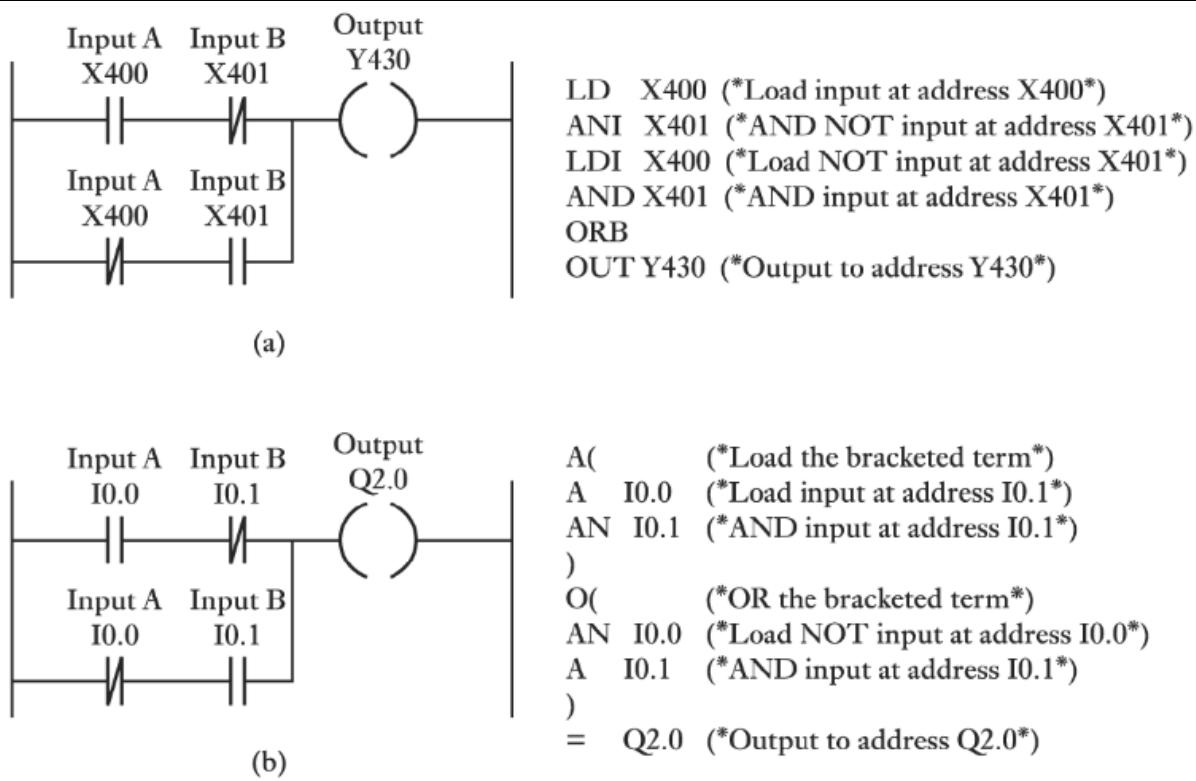
**Instruction List (IL):**

- **LDI X400:** Load the **inverse** of input X400.
- **ORI X401:** Perform a boolean **OR** operation with the **inverse** of X401.
- **OUT Y430:** Store the result to output Y430.

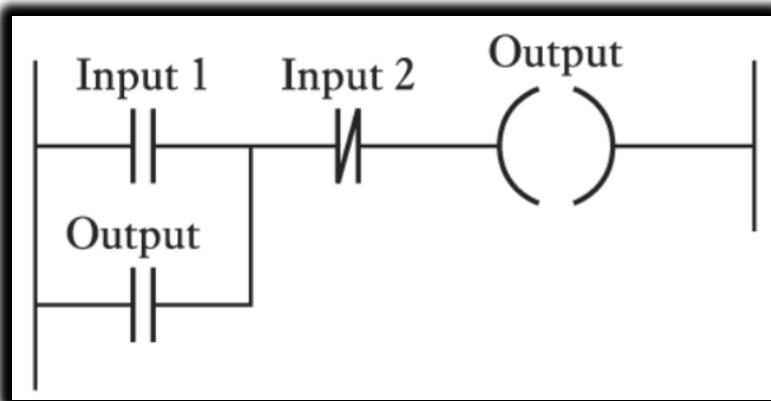
**Summary:** LDI and ORI correspond to using normally closed contacts in a parallel branch.

This image clearly shows that ladder logic is just a graphical way of representing the same instructions that are written in a text-based format like Instruction List. Both methods are used to program a PLC, and they are functionally equivalent.

Now do this one yourself, using the knowledge above, as practice.



Another one:



This ladder logic diagram shows a very common and important concept in PLC programming: a **latching** or **seal-in** circuit. It's used to make an output stay on even after the initial trigger input is removed.

Here's a breakdown of the logic:

## The Components

- **Input 1 (--| |--)**: This is a normally open (NO) contact. It's the "start" or "momentary on" button.
- **Input 2 (--|/|--)**: This is a normally closed (NC) contact. It's the "stop" or "reset" button.
- **Output (--( )--)**: This is the output coil that you want to control.
- **Output (--| |--)**: This is a second normally open contact, but it's controlled by the state of the Output coil itself. This is the "seal-in" or "latching" contact.

## The Logic (How it Works)

Let's trace the flow of logic step-by-step:

### 1. Initial State:

- Assume Input 1 and Input 2 are both off (as is normal for a start/stop button and an NC contact). The Output is off. The path through Input 1 is open, and the path through the parallel Output contact is also open. The path through Input 2 is closed.

### 2. Pressing the Start Button (Input 1):

- When you press Input 1, its contact closes.
- There is now a complete path for logic to flow through Input 1, through the closed Input 2 contact, to the Output coil.
- The Output coil is energized.

### 3. Latching Occurs:

- When the Output coil is energized, it "closes" its corresponding contact in the parallel branch.
- Now there are **two** paths for logic to flow to the Output coil: one through Input 1 and one through the Output's own contact.

### 4. Releasing the Start Button (Input 1):

- When you release Input 1, its contact opens again. The first path is now broken. **However**, the second path through the Output's own contact is still complete. The Output is on, so its contact stays closed, providing a path for the logic to keep flowing. The Output remains energized, even though the "start" button has been released. This is the "latching" action.

## 5. Pressing the Stop Button (Input 2):

- To turn off the Output, you must break the circuit.
- When you press the normally closed Input 2 contact, it opens.
- This breaks the single remaining path to the Output coil.
- The Output coil de-energizes.
- The Output's own contact then opens, breaking the latching path. The circuit is now in the initial state again, and the Output will remain off until Input 1 is pressed again.

This circuit is fundamental to PLC programming for things like motor controls, where you need to start a machine with a momentary button and have it stay on until a separate stop button is pressed.

## Let's re-explain to be honest:

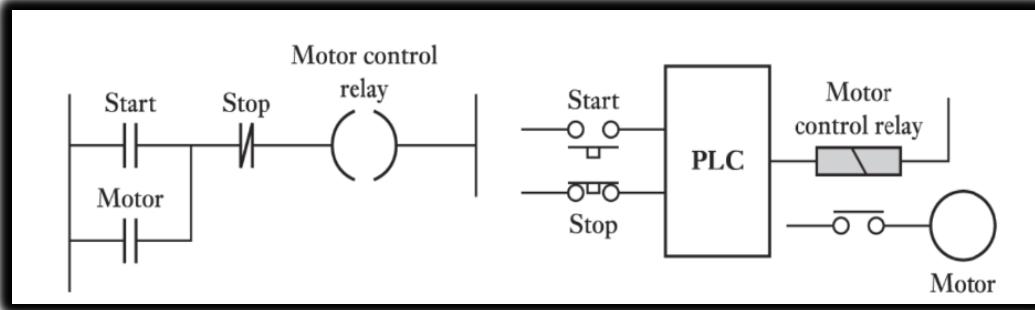
### The Components

- **Rung:** The entire horizontal line is a pipe.
- **Input 1 (--| |--):** This is a push-to-open valve (a momentary switch). You push it, the water flows, you let go, it closes.
- **Input 2 (--|/|--):** This is a push-to-close valve. It's normally open, so water is always flowing through it. You push it, it closes, and the water stops. (Wait, let's use a better analogy from the image).
  - Okay, let's reset the valve analogy based on the image's components.
  - **Input 1 (--| |--):** This is a "normally open" valve. Water won't flow until you push a button to open it.
  - **Input 2 (--|/|--):** This is a "normally closed" valve. Water is always flowing through it until you push a button to close it.
  - **Output (--)**: This is a solenoid valve. When water flows to it, it opens.
  - **Output (--| |--):** This is a "normally open" valve that is *mechanically linked* to the solenoid valve. If the solenoid opens, this valve also opens.

## The "Water Flow" Explanation

1. **STARTING POINT:** No water is flowing. Both Input 1 and the Output's own contact are closed valves. Input 2 is open, and water can flow through it.
2. **PUSHING THE START BUTTON (Input 1):**
  - You push Input 1's button. This opens its valve.
  - Water now has a path: it flows through the open Input 1 valve, through the open Input 2 valve, and reaches the Output solenoid valve.
  - The Output solenoid valve opens.
  - Because the Output solenoid valve opened, its linked contact valve also opens.
3. **RELEASING THE START BUTTON (Input 1):**
  - You let go of Input 1's button. The valve closes.
  - Water can no longer flow through the Input 1 path.
  - **But**, water can still flow through the other path! The Output's own linked contact valve is still open, so water flows through it, through the Input 2 valve, and back to the Output solenoid valve.
  - The Output solenoid valve stays open because it's getting a constant flow of water from its own linked contact valve.
  - **This is the latching!** The output "holds itself on" by providing its own path for the logic to continue flowing.
4. **PUSHING THE STOP BUTTON (Input 2):**
  - You push Input 2's button. This closes its valve.
  - The flow of water to the Output solenoid valve is now completely cut off. Both paths are blocked.
  - The Output solenoid valve closes.
  - Because the Output solenoid valve closed, its linked contact valve also closes.
  - You release the Input 2 button, and its valve opens again. But since the Output's linked valve is now closed, the circuit is back to the starting point, and no water is flowing.

This circuit is designed to work like the on/off switch on a machine. You push the "ON" button for a second, and the machine starts and stays on. You then push a different "OFF" button to stop it. The Output's own contact is the key to making this possible.



This image is a perfect example of how a **traditional motor control circuit**—built with relays—can be reimaged using a **PLC**. It showcases the core purpose of a PLC: replacing complex hard-wired logic with clean, flexible software-based logic.

### Left Diagram: Traditional Relay Logic (Latching Motor Circuit)

This is the old-school way of controlling a motor using physical relay components. It's essentially the **latching circuit** we discussed earlier, just applied to a motor.

- **Start (--| |--)** – A **Normally Open (NO)** push button. When pressed, it allows current to flow.
- **Stop (--|/|--)** – A **Normally Closed (NC)** push button. It allows current by default but opens (breaking the circuit) when pressed.
- **Motor Control Relay (--( )--)** – The coil of a relay. When energized, it switches its associated contacts (like the one labeled *Motor* below).
- **Motor Contact (--| |--)** – A NO contact tied to the Motor Control Relay. When the relay energizes, this contact closes, creating the “**seal-in**” or **latching path**.

### How It Works (Traditional):

1. **Initial State:** The Motor control relay is de-energized, so its Motor contact is open. The Stop button is closed, the Start button is open. The motor is off.
2. **Press Start:** Current flows through the closed Stop contact and the now-closed Start contact to energize the Motor control relay.
3. **Latching:** When the Motor control relay energizes, its associated Motor contact closes. This provides an alternative path for current, bypassing the Start button.
4. **Release Start:** The Start button opens, but current continues to flow through the Motor contact, keeping the Motor control relay energized. The motor continues to run.
5. **Press Stop:** The Stop button opens, breaking the entire circuit path. The Motor control relay de-energizes, its Motor contact opens, and the motor stops.

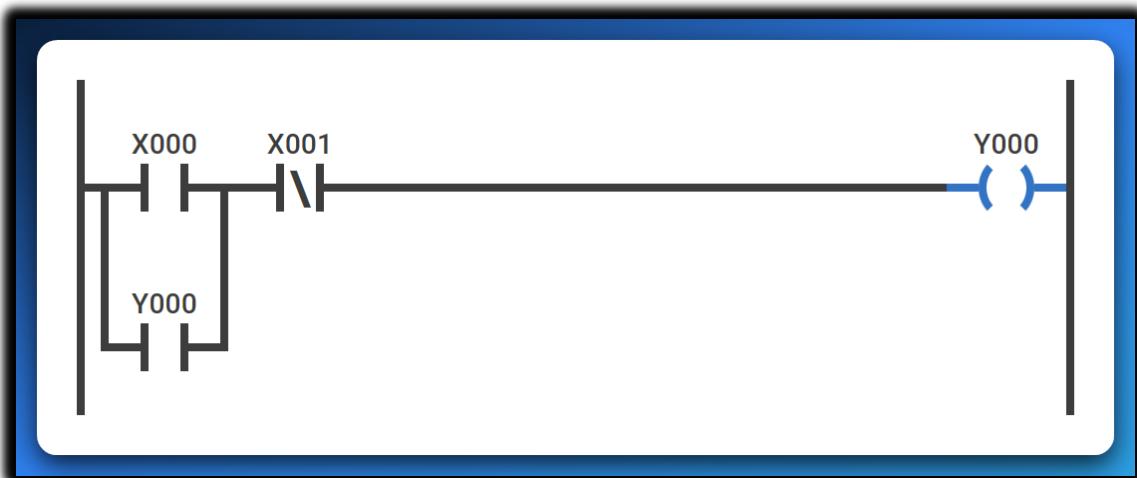
## Right Diagram: PLC-Based Motor Control

This diagram shows how the same motor control functionality is achieved using a PLC.

- **Start (Physical Button):** This is a physical Normally Open (NO) push-button wired to a digital input of the PLC.
- **Stop (Physical Button):** This is a physical Normally Closed (NC) push-button wired to another digital input of the PLC.
- **PLC:** The Programmable Logic Controller. It receives signals from the physical Start and Stop buttons.
- **Motor control relay (PLC Output):** This represents a physical relay whose coil is connected to a digital output of the PLC. The PLC energizes this output, which in turn energizes the physical Motor control relay.
- **Motor:** This represents the actual motor. The contacts of the Motor control relay (controlled by the PLC's output) switch the main power to the motor.

## How it works (PLC-based):

1. **Input Reading:** The PLC continuously monitors the state of its inputs connected to the Start and Stop buttons.
2. **Program Execution:** Inside the PLC, you would program a ladder logic rung *exactly* like the one shown on the left. Let's say Start is mapped to X000, Stop to X001, and the Motor control relay output is Y000.



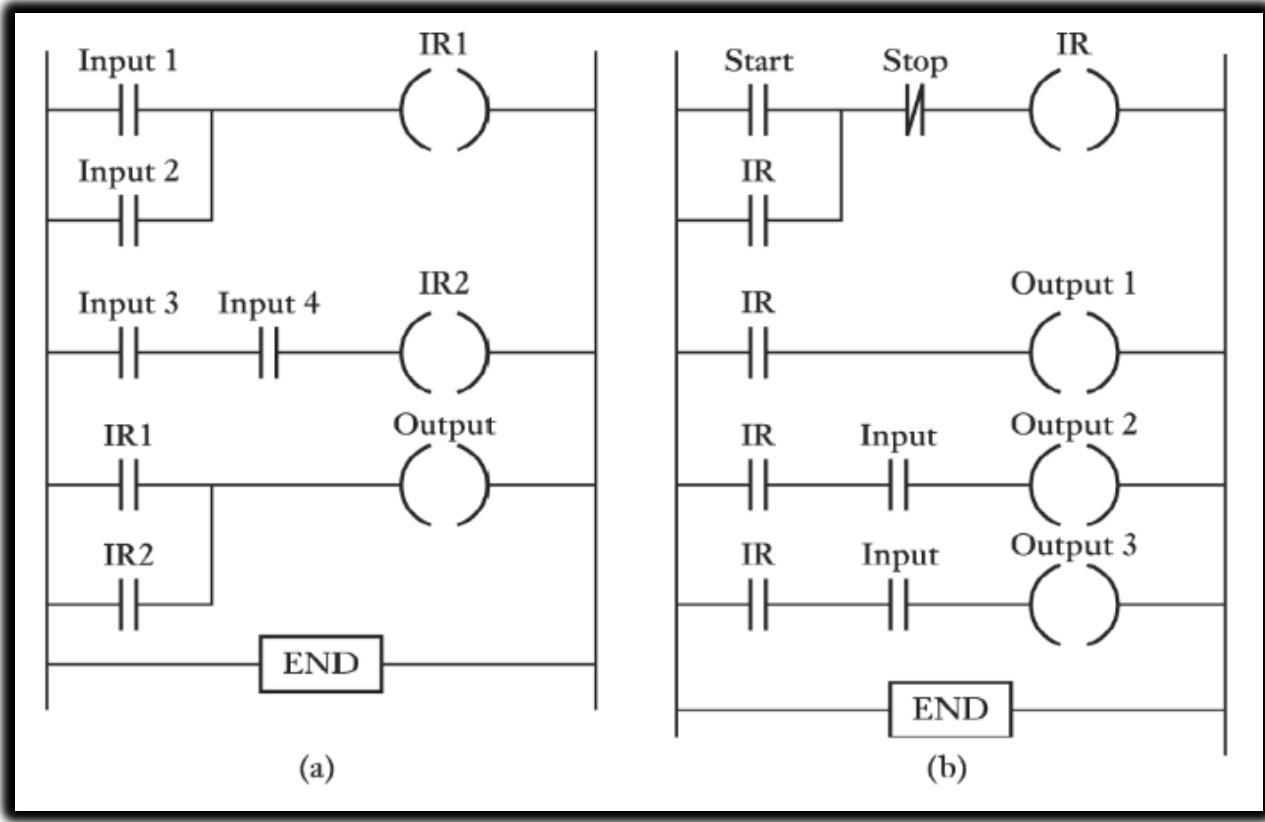
### 3. Output Control:

- When the physical Start button is pressed, the PLC detects X000 as ON. If X001 (from the Stop button) is also ON (meaning the button is not pressed), the PLC's internal logic energizes Y000.
- Y000 (the PLC's output) is then used in the latching contact within the PLC's program.
- The physical Motor control relay coil receives power from the PLC's Y000 output, activating the relay and thus starting the motor.
- When the physical Start button is released, the PLC detects X000 as OFF, but the Y000 internal contact within the PLC's program keeps the Y000 output energized (latching). The motor stays on.
- When the physical Stop button is pressed, the PLC detects X001 as OFF (because it's NC, pressing it makes the input false). This breaks the logic path in the PLC program, de-energizing Y000, and stopping the motor.

## The Key Difference (and Advantage) of a PLC

The logic itself—the latching principle—hasn't changed. What's different is **how it's implemented**. With a PLC:

- You no longer need to rewire physical relays to modify the control logic.
- Updates, debugging, or adding features become as simple as editing the program.
- Complex wiring diagrams shrink down to clear, readable ladder logic.



This image presents two multi-rung ladder logic programs, each demonstrating more complex control scenarios than single-rung examples.

The key concept introduced and heavily utilized here is the use of **Internal Relays (IR)**, also known as internal coils, memory bits, or flags.

These are **virtual outputs** within the PLC's memory that don't directly control a physical device but are used to store intermediate logic results, making complex programs more organized and efficient.

Let's explain each program:

## (a) Program (a) - Combining Logic with Internal Relays

This program shows how multiple conditions can be combined using internal relays to control a final output.

- **Rung 1: Input 1 OR Input 2 → IR1**

- **Logic:** IR1 (Internal Relay 1) will be energized **IF** Input 1 is ON **OR** Input 2 is ON.
- This is a simple OR gate. IR1 acts as an intermediate flag; it's ON if either of these conditions is met.

- **Rung 2: Input 3 AND Input 4 → IR2**

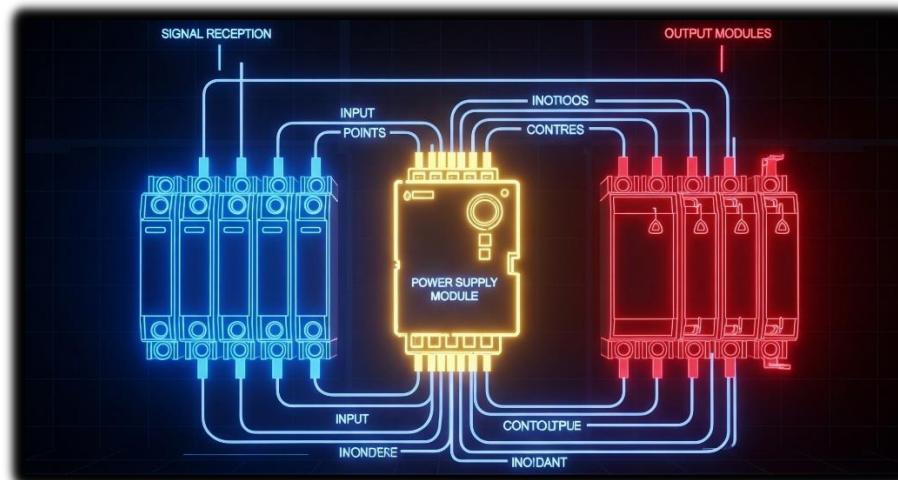
- **Logic:** IR2 (Internal Relay 2) will be energized **IF** Input 3 is ON **AND** Input 4 is ON.
- This is a simple AND gate. IR2 is another intermediate flag; it's ON if both these conditions are met.

- **Rung 3: IR1 OR IR2 → Output**

- **Logic:** The final Output will be energized **IF** IR1 is ON **OR** IR2 is ON.
- This rung takes the results from the previous two rungs. The Output will be ON if ( (Input 1 OR Input 2) OR (Input 3 AND Input 4) ) is true.

- **END:** Marks the end of the program.

**Purpose of IR1 and IR2:** They simplify the overall logic. Instead of creating one very long and potentially complex rung, the logic is broken down into smaller, manageable pieces. This makes the program easier to read, understand, debug, and modify. If Input 1 or Input 2 are also used elsewhere, IR1 can act as a single "summary" input.



## (b) Program (b) - Latching with Internal Relay and Dependent Outputs

This program demonstrates a common motor control type scenario where a main control (a latching circuit) enables or disables other outputs.

- **Rung 1: Start (NO) and Stop (NC) with IR (Latching) → IR**

- **Logic:** This is the standard latching (seal-in) circuit we discussed earlier.
- The internal relay IR (Internal Relay) will be energized when the Start button is momentarily pressed, and it will stay energized even after Start is released, as long as Stop is not pressed.
- IR essentially acts as a "System ON" or "Master Enable" flag.

- **Rung 2: IR → Output 1**

- **Logic:** Output 1 will be energized **IF** IR is ON. This means Output 1 will turn ON as soon as the main IR latching circuit is engaged, and it will stay ON as long as IR is ON.

- **Rung 3: IR AND Input → Output 2**

- **Logic:** Output 2 will be energized **IF** IR is ON **AND** Input is ON. This demonstrates that Output 2 can only be activated if the "System ON" (IR) is active **AND** a second condition (Input) is also met. If the IR is off, Output 2 can never turn on, regardless of Input.

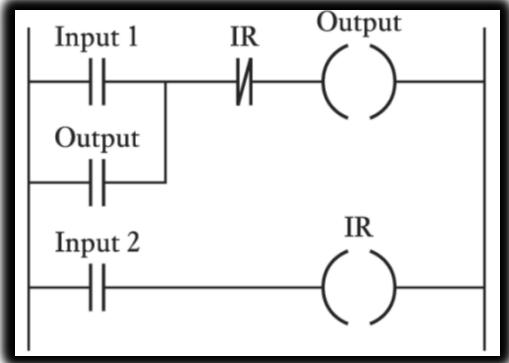
- **Rung 4: IR AND Input → Output 3**

- **Logic:** Output 3 will be energized **IF** IR is ON **AND** Input is ON.
- This is identical logic to Rung 3. It shows that multiple outputs can be controlled by the same combination of conditions. **END** does the obvious.

### Purpose of IR and Dependent Outputs:

The IR internal relay provides a master control. If IR is not latched ON, none of Output 1, Output 2, or Output 3 can be turned on. This is extremely useful for:

- **Safety:** A single stop button can de-energize a critical master relay, shutting down an entire process or machine.
- **Sequencing:** It ensures that certain operations (like Output 2 or Output 3) only occur when the system is in its enabled state.
- **Modularity:** You can easily add more outputs or complex logic that depends on the IR being active without affecting the main start/stop control.



This ladder logic diagram shows a **latching circuit** combined with an **internal relay (IR)** that acts as an override or condition for the main output. It's a common control pattern, especially when you need a start/stop behavior with a "master stop" or enable function.

### Rung 1: Latching Circuit with Input 1, Output, and IR

- **Input 1 (--) |--:** This is a Normally Open (NO) contact, likely representing a "Start" or "Enable" push-button.
- **Output (--) |-- in parallel with Input 1:** This is a Normally Open (NO) contact that is controlled by the Output coil itself. This forms the "seal-in" or "latching" part of the circuit.
- **IR (--|/|--):** This is a Normally Closed (NC) contact that is controlled by the IR coil in the second rung. This contact acts as a "stop" or "disable" for the first rung's logic.
- **Output (--) (--):** This is the output coil that this rung directly controls.

### Logic of Rung 1:

The Output coil will be energized **IF** (Input 1 is pressed **OR** Output is already latched ON) **AND** IR is OFF.

- **How to start Output:** You press Input 1. If IR is currently OFF (its NC contact is closed), then current flows to the Output coil, energizing it.
- **How Output latches:** Once Output is energized, its parallel contact closes, creating a self-holding path. You can release Input 1, and Output will remain ON, provided IR stays OFF.
- **How to stop Output:** If IR turns ON (from Rung 2, as we'll see next), its normally closed contact in Rung 1 will open, breaking the circuit and de-energizing Output. This acts as a "master stop" controlled by Input 2.

## Rung 2: Simple Control of IR

- **Input 2 (--)**: This is a Normally Open (NO) contact, likely a push-button or sensor.
- **IR (--)**: This is an **Internal Relay** (or memory bit) coil. It does not directly control a physical device but stores a logical state (ON or OFF) within the PLC's memory.

### Logic of Rung 2:

The internal relay IR will be energized **IF** Input 2 is ON.

- When Input 2 is ON, IR turns ON.
- When Input 2 turns OFF, IR turns OFF.

### How the Two Rungs Interact

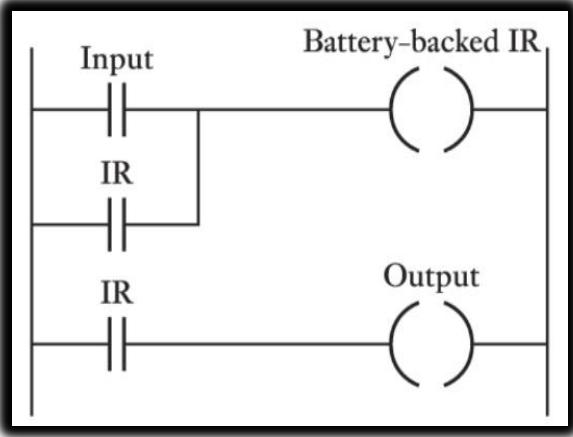
The key here is how IR in Rung 2 influences Rung 1.

- **Input 2 as a "Master Stop" for Output:**
  - When Input 2 is activated, IR turns ON (Rung 2).
  - When IR turns ON, its Normally Closed contact (--) in Rung 1 **opens**.
  - This opening breaks the power flow in Rung 1, forcing Output to turn OFF, regardless of Input 1 or Output's own latching contact.
  - So, pressing Input 2 effectively stops Output.

### Overall Function

This program creates a **robust start/stop setup**:

1. **Start**: Press Input 1 to energize Output.
2. **Latch**: Output stays ON even after Input 1 is released.
3. **Stop**: Press Input 2. This energizes IR, which overrides and turns Output OFF.
4. **Reset**: Releasing Input 2 allows Output to be restarted with Input 1.



This ladder logic diagram shows a specific type of internal relay: a **Battery-backed Internal Relay**. It also illustrates how such an internal relay can be used to control a physical output.

### Rung 1: Latching Circuit for "Battery-backed IR"

- **Input (--| |--):** This is a Normally Open (NO) contact, acting as a "start" or "enable" trigger.
- **IR (|--| -- in parallel with Input):** This is a Normally Open (NO) contact controlled by the Battery-backed IR coil itself. This creates the self-latching (seal-in) mechanism.
- **Battery-backed IR (-( )--):** This is an internal relay coil, similar to the IR we discussed before. The crucial difference is the "Battery-backed" designation. This means that if the PLC loses power, the state (ON or OFF) of this specific internal relay is *retained by a small battery inside the PLC*. When power is restored, the Battery-backed IR will return to its state before the power loss. This is also called a "retentive" memory bit.

### Logic of Rung 1:

The Battery-backed IR coil will be energized **IF** (Input is turned ON **OR** Battery-backed IR is already latched ON).

- **How to turn Battery-backed IR ON:** Press Input. The Battery-backed IR coil energizes.
- **How it latches:** Once energized, the Battery-backed IR's parallel contact closes, creating a holding path. You can release Input, and Battery-backed IR remains ON.
- **How to turn Battery-backed IR OFF:** There isn't an explicit "stop" button in this rung. To de-energize Battery-backed IR, you would typically need another rung with a normally closed stop contact in series, or a separate unlatch instruction for Battery-backed IR.

## Rung 2: Controlling a Physical Output with "Battery-backed IR"

- **IR (--| |--)**: This is a Normally Open (NO) contact controlled by the Battery-backed IR coil from Rung 1.
- **Output (--)**: This is a physical output coil, which would control an external device like a light, motor, or solenoid.

### Logic of Rung 2:

The Output coil will be energized **IF** IR (the Battery-backed IR) is ON.

- This means the Output will simply follow the state of the Battery-backed IR. If Battery-backed IR is ON, Output is ON. If Battery-backed IR is OFF, Output is OFF.

### Key Concept: Battery-backed / Retentive Memory

The standout feature of this diagram is the **Battery-backed Internal Relay (IR)**.

### Non-Retentive (Volatile) Internal Relays

Most standard internal relays—like the ones we saw earlier—are **non-retentive**.

- If the PLC loses power, their state (ON/OFF) is **reset to OFF** when power comes back.
- This is fine for simple systems but not ideal for processes that need to “remember” their state.

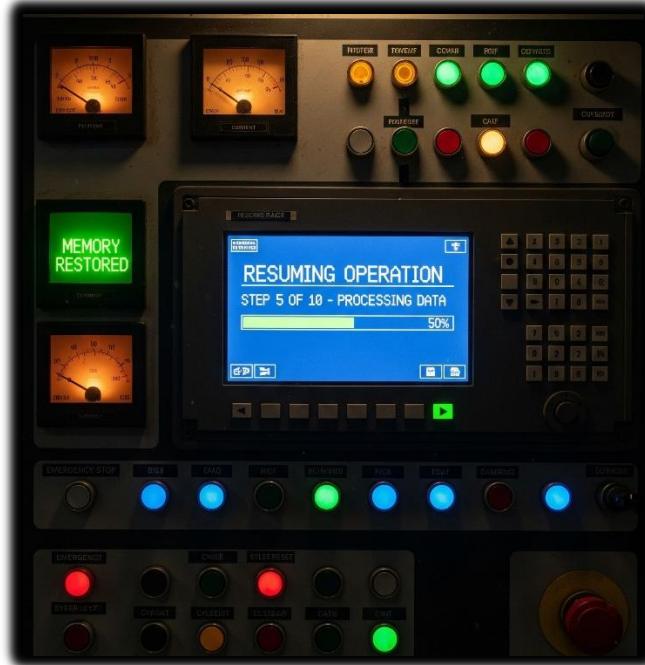
### Retentive (Battery-backed) Internal Relays

Battery-backed IRs are designed to **hold their state even through power cycles**.

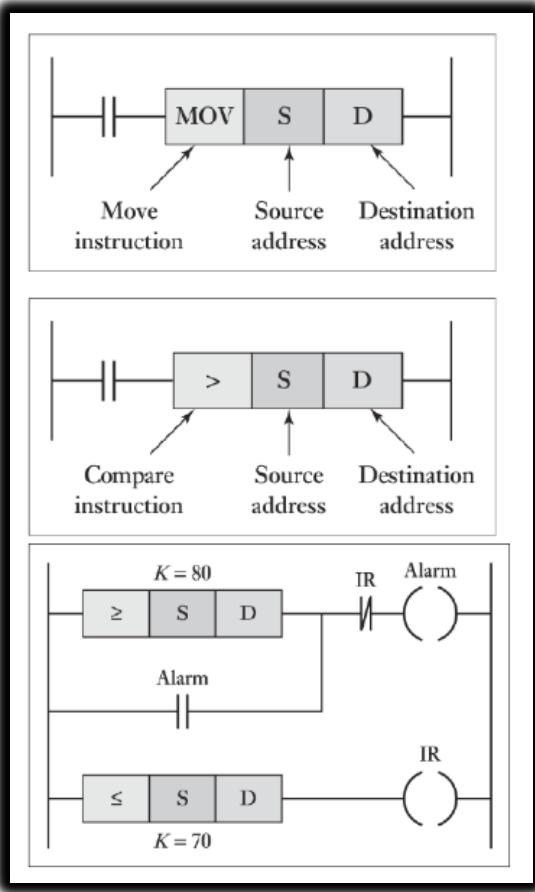
- A small internal battery inside the PLC provides the backup power needed to maintain this memory.
- This is crucial for machines that need to **resume where they left off after a power outage**—for example, remembering that a sequence was halfway done or that a certain mode was active.

## Overall Function of This Program

- Once the **Battery-backed IR** is latched ON (via an input signal), it **stays ON**, even if the PLC loses and regains power.
- The Output simply mirrors this persistent state—turning ON whenever the Battery-backed IR is ON.



This makes retentive memory **invaluable for industrial processes** where a power cut shouldn't mean starting over.



## Key Concept: Data Handling and Comparison in Ladder Logic

This image introduces **special PLC instructions**—specifically, **MOV (Move)** and **Compare instructions**—which are essential for going beyond simple ON/OFF control. These tools let you handle numeric data, set thresholds, and build smarter logic based on real-world values (like temperature or pressure).

### 1. MOV (Move Instruction)

- **Ladder Symbol:** A box labeled MOV with S (Source address) and D (Destination address).
- **Purpose:** Copies data from one memory location to another.
- **How It Works:**  
When the rung condition to the left of the MOV box is true, the PLC takes the current value at **S (Source)** and writes it into **D (Destination)**.
- **Example Use:** Move a setpoint value into a timer's preset register or copy a counter's current count into a display register.

## 2. > (Greater Than – Compare Instruction)

- **Ladder Symbol:** A box labeled  $>$  with S and D.
- **Purpose:** Checks if **S** is greater than **D**.
- **How It Works:**  
If **S > D**, the instruction evaluates as true and allows the logic to flow to the right. Otherwise, it evaluates as false, and the rung stops there.
- **Example Use:** Compare a temperature reading (S) against a high-limit setpoint (D) to trigger an alarm.

## 3. Practical Application: $\geq$ and $\leq$ for High/Low Limits

### Top Rung: High-Limit Alarm

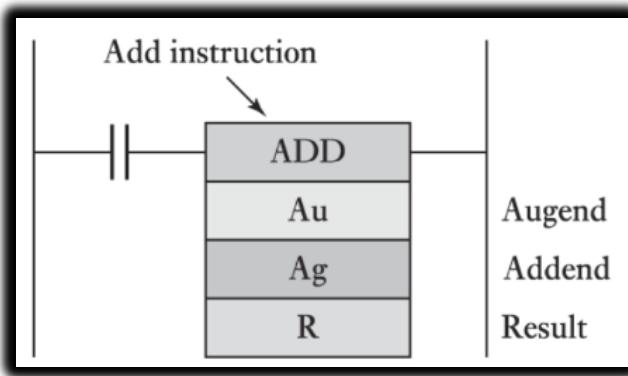
- **Instruction:**  $\geq K=80 S D$ 
  - **K=80** is a constant value (80). The PLC checks if **S  $\geq$  80**.
  - **S** could be a process variable like a temperature input.
- **Logic:**  
The Alarm coil turns ON if (**S  $\geq$  80**) AND (**IR is ON**).
- **Purpose:**  
This triggers an alarm when a high threshold is reached—e.g., if the temperature hits 80°C and the master alarm relay (IR) is active.

### Bottom Rung: Low-Limit Control (IR Relay)

- **Instruction:**  $\leq K=70 S D$ 
  - Compares **S  $\leq$  70**.
  - **IR (--)** is an internal relay coil that activates when this condition is true.
- **Purpose:**  
This IR can represent a **low condition trigger**, such as signaling that the temperature has dropped to 70°C or lower. IR can then be used to control heating or enable alarms in the rung above.

## Why This Matters

- **MOV and Compare instructions** give PLCs the ability to handle **numeric data** rather than just boolean states.
- They enable **threshold-based logic**, which is critical for analog sensors (temperature, level, flow, etc.).
- With these tools, you can implement smart control strategies, such as turning on a heater when the temperature is too low or sounding an alarm when it's too high.



This image introduces another type of special instruction in PLC ladder logic: an **Arithmetic Instruction**, specifically the **ADD instruction**. These are used for performing mathematical operations on data within the PLC.

## The ADD Instruction

- **Ladder Symbol:** A box labeled ADD with three fields below it: Au, Ag, and R.
- **Add instruction:** This indicates the type of mathematical operation to be performed. In this case, it's addition.
- **Au (Augend):** This is the memory address or value that represents the **first number** in the addition operation. Think of it as the "number to which something is added."
- **Ag (Addend):** This is the memory address or value that represents the **second number** in the addition operation. Think of it as the "number that is added."
- **R (Result):** This is the **destination memory address** where the sum of Au and Ag will be stored.

## How it Works

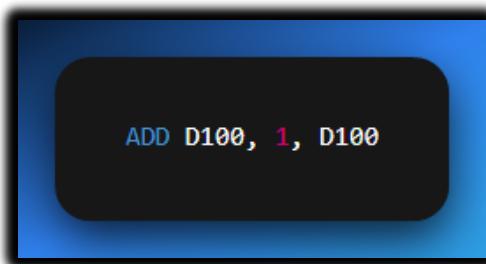
- When the contact(s) to the left of the ADD instruction are true (i.e., the rung condition is met), the PLC will execute this instruction.
- It will retrieve the value from the memory address specified by Au.
- It will retrieve the value from the memory address specified by Ag.
- It will add these two values together ( $Au + Ag$ ).
- The calculated sum will then be stored in the memory address specified by R.

## Example Use Case

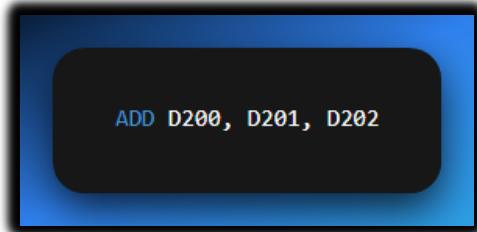
Imagine you have a process where you need to track a total count, or adjust a setpoint:

- **Counting:**

- Au could be a register holding a current total (e.g., D100).
- Ag could be a constant value 1 (if adding one each time) or another register (e.g., D101) that stores a batch quantity.
- R would be the same register as Au (D100) if you want to update the running total in the same location, or a new register (D102) if you want the sum in a separate spot.
- So, **ADD D100, 1, D100** would increment the value in D100 by 1 each time the rung is true.



- **Adjusting a Setpoint:**
  - Au could be a base temperature setpoint (e.g., D200).
  - Ag could be an offset value (e.g., D201) that changes based on production mode.
  - R would be the actual operating setpoint that a temperature controller uses (e.g., D202).
  - So, **ADD D200, D201, D202** calculates the final setpoint.



The ADD instruction—along with SUB (subtract), MUL (multiply), and DIV (divide)—is essential for handling numeric data from sensors and implementing calculations inside the PLC. These instructions are the backbone of more advanced control strategies.

**SUB, MUL, and DIV work the same way as ADD**—they just change the operation performed on the two values.

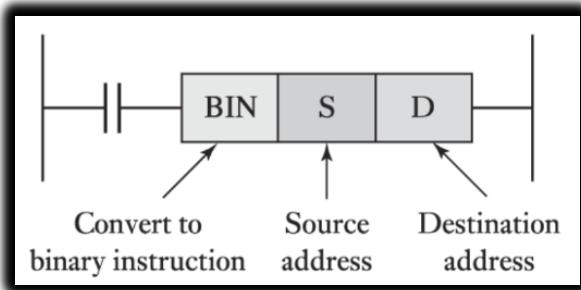
They all use:

- **Au (first value/memory)**
- **Ag (second value/memory)**
- **R (result destination)**

The only difference is:

- ADD → Adds **Au + Ag → R**
- SUB → Subtracts **Au - Ag → R**
- MUL → Multiplies **Au × Ag → R**
- DIV → Divides **Au ÷ Ag → R**

So yeah, we don't need to write separate full notes for each.



## Data Conversion Instruction: BIN

This image introduces another key PLC instruction—**BIN (Convert to Binary)**—which is used to **convert data from one numerical format (e.g., BCD) into pure binary** for internal processing.

### BIN Instruction Breakdown

- **Ladder Symbol:** A box labeled BIN with two fields: **S** (Source) and **D** (Destination).
- **S (Source):** Memory address holding the original data (e.g., from an input module).
- **D (Destination):** Memory address where the converted binary value will be stored.

### How It Works:

When the rung condition to the left is true, the PLC:

1. Reads the value from **S**.
2. Converts it into pure binary (if it's in another format, such as BCD).
3. Stores the converted value into **D**.

### Why Conversion Is Needed

- **BCD Inputs:** Devices like thumbwheel switches or certain displays often output **Binary Coded Decimal (BCD)**. In BCD, each decimal digit is stored as its 4-bit binary code (e.g., decimal 25 is 0010 0101 in BCD).
- **PLC Processing:** PLCs perform math and logic operations using **pure binary** (e.g., decimal 25 is 0001 1001 in binary).
- **Compatibility:** Data must be converted to binary before it can be used in instructions like ADD, SUB, or compare operations.

## Example Use Case

Suppose a thumbwheel switch outputs a BCD value (e.g., 25) into register **I\_Data\_Register**

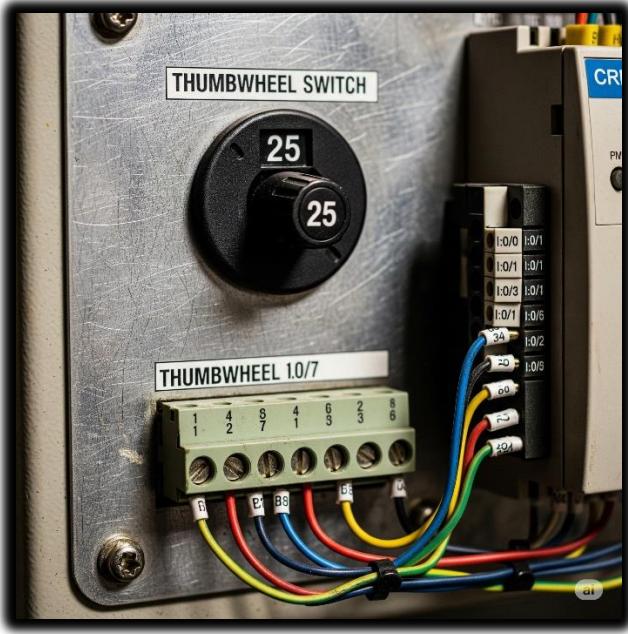
You want this number to be used as a **timer preset**. You'd use:



When the rung is true, the PLC converts 0010 0101 (BCD 25) into 0001 1001 (binary 25) and stores it in D\_Timer\_Preset.

## Why It's Important

The BIN instruction acts as a **bridge between external devices and internal PLC logic**, ensuring that data coming from different encoding schemes can be processed correctly.



These were my personal notes, they mostly have no order of topicS, I just threw in a bunch of pdfs together and made sense of them without caring about order, because I was just reading for basic understanding of PLCs and what they do. The next pdf is going to be hosting the ladder logic complex content, and then the questions on plcs will be in the final PDF.