Industrial automation and robotics

Logic control: the PLC

Prof. Paolo Rocco (paolo.rocco@polimi.it)

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria

Modulating and logic control

In process and factory automation we must control both continuous time physical quantities and discrete variables.

An automation system is in fact composed of many different controllers.

Part of them are digital control systems, i.e., PIDs, used to control continuous time physical quantities.

Another important part is made by devices used to control action sequencing, safety interlock logics, etc. These devices are called **Programmable Logic Controllers** (PLCs).

Sometimes the following terminology is used:

- **Modulating control** (control of continuous time physical quantities)
- **Logic control** (control of the sequence of actions)

Some of these slides are based on the lectures of Prof. Alberto Leva (Politecnico di Milano)

Production processes

Industrial processes can be classified by the type of operations and transformations they (mainly) carry out.

We then talk about:

- **continuous** processes
- **batch** processes
- **discrete or manufacturing** processes

Continuous processes

In **continuous processes** there are continuous transformations of mass and energy.

The goal of the process is to achieve a uniform product over time.

Examples are power plants, distribution plants, paper mills, ...

In such processes **modulating control prevails**.

The logic control deals essentially with ignition, shutdown, emergencies, supervision.

Batch processes

Products are processed in quantities or **batches** of scalable size.

The control of these plants is therefore based on the recipes of processing, i.e. the sequence of processes to be done to obtain the product.

Examples are chemical plants, food production, ...

In such processes **modulating control** operates essentially in the subsystems of production

The **logic control** implements the recipes and must ensure a correct use of plant resources.

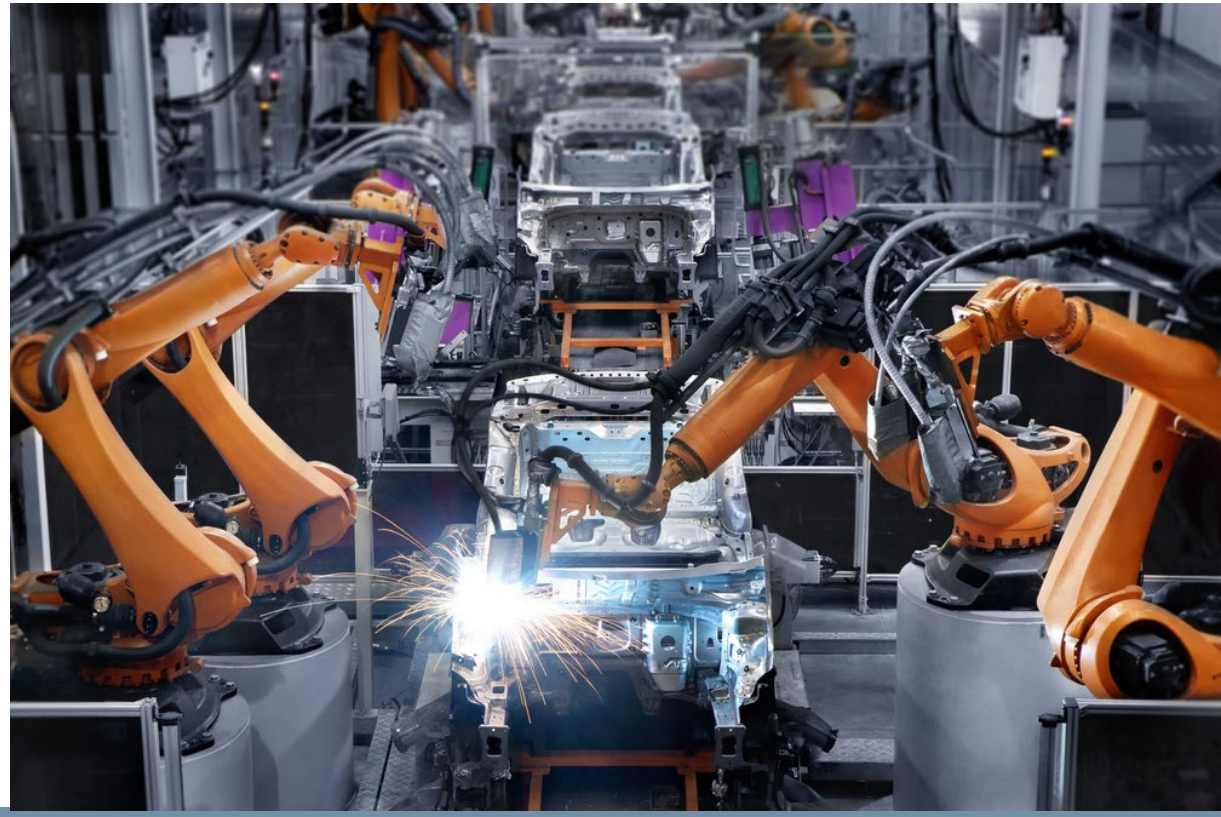
Discrete processes

Discrete processes are characterized by processing cycles involving individual parts or individual units of product (pieces). Processing, transport and storage operations are present.

Examples are machining centers, assembly, storage...

In such processes **modulating control** essentially deals with motion within individual machines (e.g. spindle speed).

The **logic control** coordinates all the machines, the lines of production, transport systems, and so on.



Action sequencing and logic control

Action sequencing is the control of a set of actions, triggered by events, that should be executed in the correct order, examples are:

- controlling an elevator
- controlling a vending machine
- controlling the lightning system
- controlling a production plant
- etc.

In general, the correct execution of an action and the correction action sequencing require different strategies for both the formalization of the problem and for the implementation of a proper **logic control**.

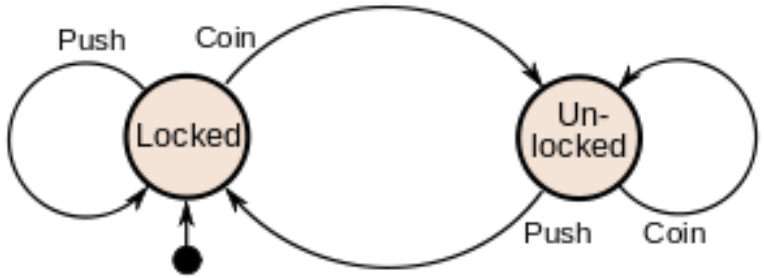
Discrete events systems

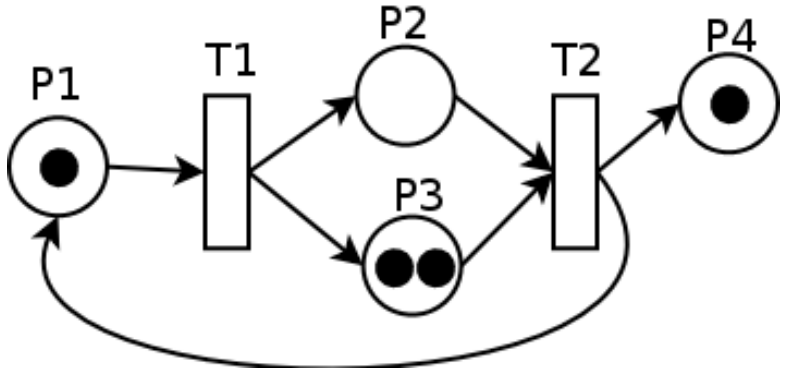
A **discrete events system** is a dynamic system (then endowed with a state) where:

- The state takes values only in a discrete set of values
- The evolution of the state is exclusively given by the occurrence of asynchronous events (i.e. whose timing does not obey any regularity property)

A classical example is the line of people at a counter, where the state is number of people queuing while the events are the arrival of new people or the exit of other people.

Finite state automata and **Petri nets** are among the most widely used mathematical tools for modeling discrete event systems (we will not cover them in this course).





Discrete events systems: an example

Consider an extremely simple example made by a **lamp and a button**.

When you press and release the button, if the lamp was off it turns on, and if it was on it turns off. Switching on and off takes place at the moment the button is released.

This is a **dynamic system** because just observing the input (the fact that the button is pressed or not), we cannot infer the state of the lamp (turned on or switched off): we also need an **initial state**.

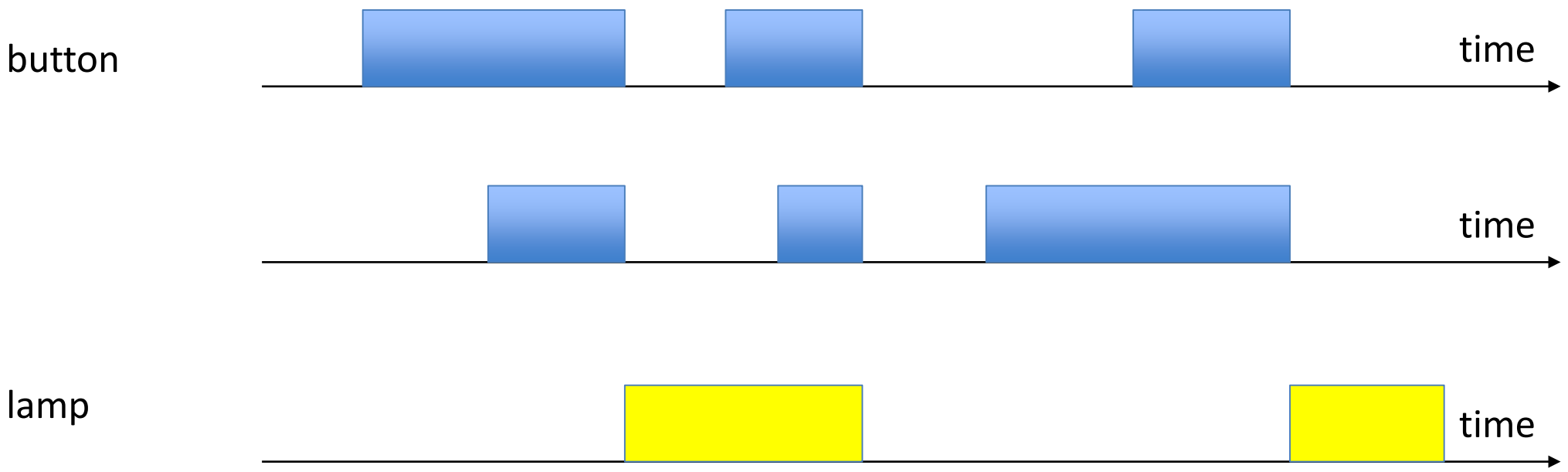
The **variables involved** are:

- The button {pressed, released} which serves as an input
- The lamp {on, off} which serves as state (and output)

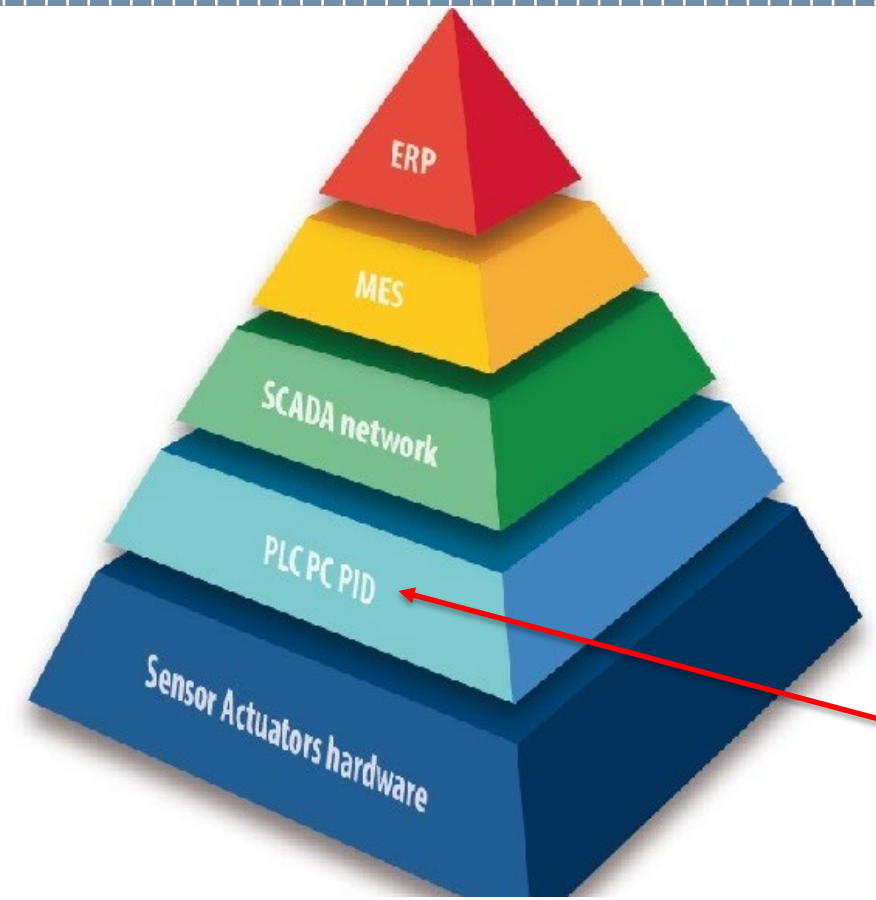
Discrete events systems: an example

The dynamic system does not evolve based on time but based on events.

The only significant event here is the release of the button, irrespectively of the time history of the button pressing/release:



Hierarchical automation systems



It is rather common that the automation system is organized in **hierarchical layers**.

At the highest level there is the ERP (Enterprise Resource Planning), a management systems that gathers all the functions at company level.

The lowest level corresponds to field devices like sensors and actuators.

We are now interested in **this** level

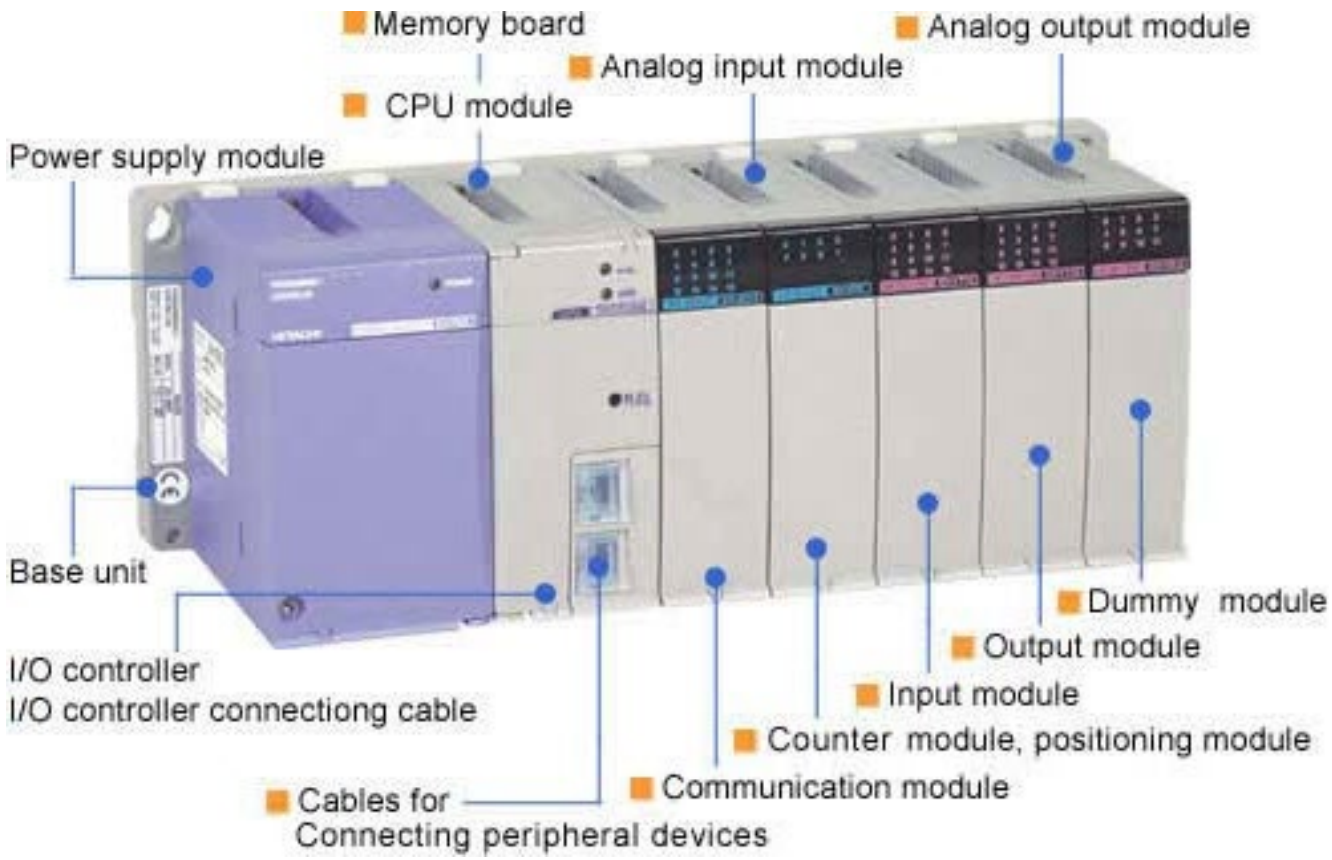
The programmable logic controller

The **PLC** (**Programmable Logic Controller**) was introduced in automation to replace sequential relay circuits, with a device capable of acquiring inputs and performing logic operations on them after programming.

The diffusion of PLCs in automation is enormous: in fact they realize **the logic control** that together with the modulating control performs low-level functions essential for the production process.

Typical PLCs

The programmable logic controller



The programmable logic controller

A PLC program is executed repeatedly, every T_c milliseconds, as long as the controlled system is running, and consists in the following steps:

- the status of physical inputs is copied to an area of memory accessible to the processor (I/O Image Table)
- the user program is run from its first instruction down to the last one, and the I/O image table is updated with the status of outputs
- operating system services are executed
- the status of outputs is copied from the I/O image table to physical outputs

T_c is called **cycle time** and is related to the specific application.

The previous steps are executed as a sequence of instructions, as a consequence the input signals can be acquired only at the beginning of the PLC cycle (no input variations during the cycle time can be detected).

Programming a PLC

Under the **IEC 61131-3** standard, PLCs can be programmed using standard-based programming languages.

IEC 61131-3 currently defines **five programming languages** for programmable control systems:

- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Sequential Function Chart (SFC)
- Structured Text (ST or STX), similar to Pascal
- Instruction List (IL), similar to machine assembly (now deprecated)

FBD, LD and SFC are graphical programming languages, while ST and IL are text programming languages.

In this course we will cover the Ladder Diagram and the Sequential Function Chart

Ladder diagram

The **Ladder Diagram** (LD) is derived from the drawings of control systems made with electromechanical relays.
It is based on the concepts of contact and coil and was initially designed for binary logic functions; then it was extended to cover more advanced functions.

It is a low-level and poorly structured language, not very suitable for complex systems. However, it is important because it was the first graphic language for PLCs, is present in all industrial PLCs and is a de facto standard of the American market.

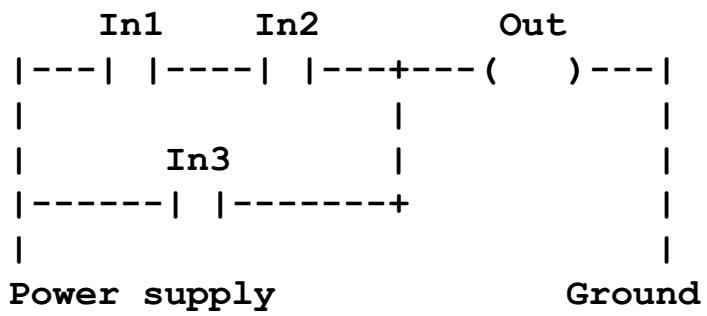
Ladder diagram

A ladder diagram consists of:

- two **vertical lines** called risers: the left riser is figuratively connected to the power supply, the right one to ground
- some horizontal connections between the vertical lines, called rungs that contain **contacts** on the left and **coils** on the right
- the “current” can only flow from left to right
- each variable and each coil are associated with logical variables
- the rungs are explored from the top to the bottom
- if there is electrical continuity between the left riser and the coil, the logical variable associated with the coil is 1 (true), otherwise it is 0 (false)

Ladder diagram: first example

Consider this very simple Ladder Diagram:



- When the variables In1 and In2 are “true” the, corresponding contacts are closed and the current flows from the power supply to the coil associated with the variable Out which is then true
- The same applies when variable In3 is true
- Then Out is true when either both In1 and In2 are true or when In3 is true
- The corresponding logic function is then:

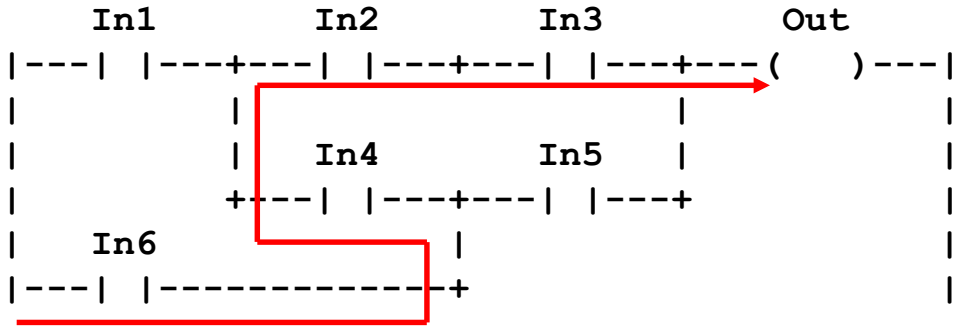
```
Out = (In1 AND In2) OR In3
```

Ladder diagram: basic rules

To avoid ambiguities, the definition of a Ladder Diagram obeys some rules:

Rule 1

The current can flow through contacts and coils only **from left to right**



In this diagram the current **cannot** flow like the red arrow.

The logic function represented by this block diagram is therefore:

```
Out = (In1 AND ((In2 AND In3) OR (In4 AND In5))) OR (In6 AND In5)
```

Ladder diagram: basic rules

Rule 2

The rungs are explored by the PLC from the first at the top to the last at the bottom, and at the last the PLC starts again from the first one.

As a result, the **order of the rungs is relevant** (as is that of the instructions in a program).

Rule 3

The synchronization of the variables of the program with inputs and outputs takes place according to the principle of **massive copying**:

- the inputs are read (and for the purposes of the program remain constant throughout the cycle)
- all the rungs are executed (unless there are jumps, see later) and all the coils are assigned a value
- each coil retains its value until it is rewritten in a subsequent cycle
- the outputs are updated
- the cycle is started all over again

The contacts

Normally open contact

Symbol --| |--

It is associated with a Boolean variable (a bit): if the bit associated with the contact is 1 (“true”) the contact is closed and there is logical (electrical) continuity, otherwise the contact is open and there is no continuity.

Normally closed contact

Symbol --|/|--

It is associated with a Boolean variable (a bit): if the bit associated with the contact is 0 (“false”) the contact is closed and there is logical (electrical) continuity, otherwise the contact is open and there is no continuity.

The coils

Normal coil

Symbol `--( )--`

It should always be inserted at the end of the rung and can be associated with an output bit or internal bit but not with an input (which would not make sense).

The coil is activated when current passes through it. The bit associated with it is 1 (“true”) if the logical conditions to its left are verified, otherwise it is 0 (“false”). The name of the bit is placed above the coil.

Latch coil

Symbol `--(L)--`

When it is activated, the associated bit becomes 1 and remains in this state until a coil associated with the same bit and of the unlatch type is activated

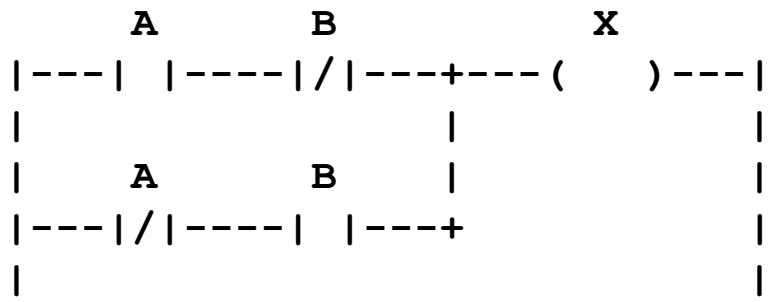
Unlatch coil

Symbol `--(U)--`

When it is activated, the associated bit becomes 0 and remains in this state until a coil associated with the same bit and of the latch type is activated

A second example

Consider the following ladder diagram:



The logic function is clearly:

X = (A AND NOT(B)) OR (NOT(A) AND B)

X is then true when one, and only one, within **A** and **B** is true. This logic function is also called **Exclusive OR**

X = A XOR B

Edge detection contacts

Other elements can be used in the ladder diagrams:

Positive edge detection contact

Symbol `--|P|--`

The positive edge detection contact closes **for a single cycle** when the bit associated with it changes from 0 to 1, and remains open in all other cases.

Negative edge detection contact

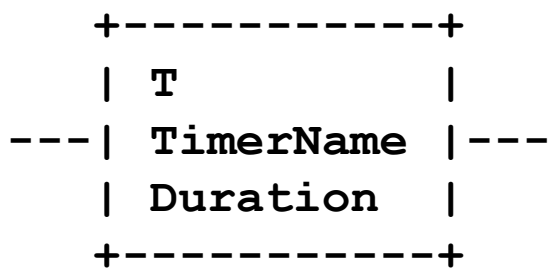
Symbol `--|N|--`

The negative edge detection contact closes **for a single cycle** when the bit associated with it changes from 1 to 0 and remains open in all other cases.

Timers

Normal timer

Symbol



- **Duration** is a parameter, usually expressed in ms, and obviously indicates how long the timer should count
- If the “current” reaches the timer (from the left) the timer counts the time until it reaches the duration
- At this point the variable **TimerName** is set to 1 and remains in this state until the reset of the timer, which occurs when the electrical continuity towards the timer stops
- In any other case, **TimerName** is equal to 0
- At all times, the time counted by the timer is accessible with a special symbolic name automatically defined by the system, usually **TimerName.acc** (accumulated time).

Timers

Latch timer

Symbol

```
    +-----------+
    | TL        |
---|  TimerName |---
    | Duration  |
    +-----------+
```

- It is similar to the normal timer
- When the electrical continuity towards the block is absent, the timer is not reset, instead it stops the time count at the value reached at that time.
- When continuity returns, the count then resumes from the previously reached value

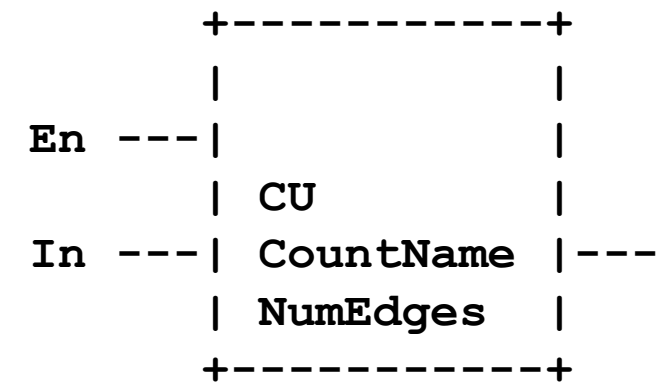
- To **reset the time count**, a special command is used

```
  TimerName
---(RES)---
```

Counter

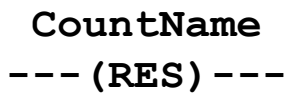
Counter

Symbol



- Similar to the latch timer
- The positive edges, from 0 to 1, of the input **In** are counted, up to the number **NumEdges**, if there is electrical continuity at the input **En** (enable)

- To **reset the counter**, a special command is used:



Control of the flow

Jump to a label

Symbol

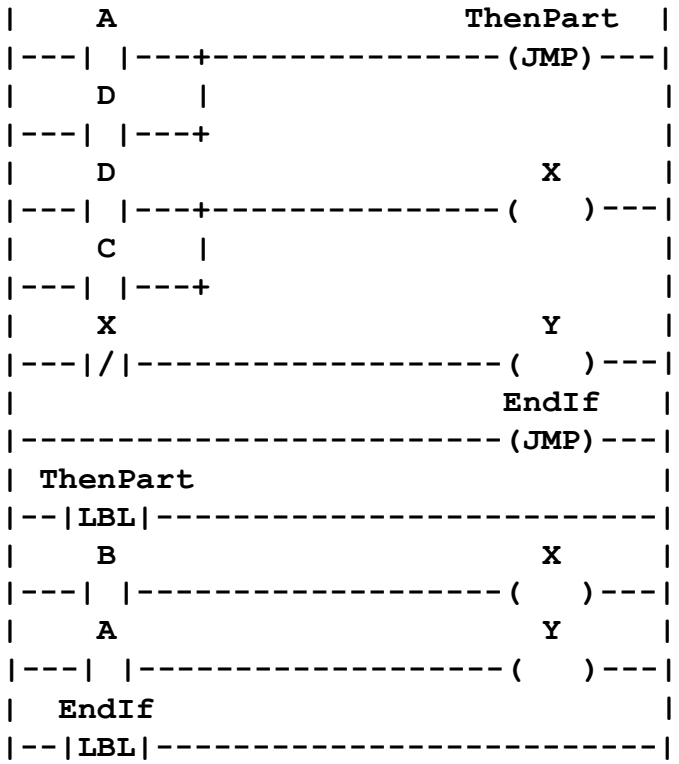
```
---(JMP)---

---|LBL|---
```

If the **JMP** coil is powered, the PLC jumps to the rung right after the one that contains only the **LBL** element

Sample of LD code:

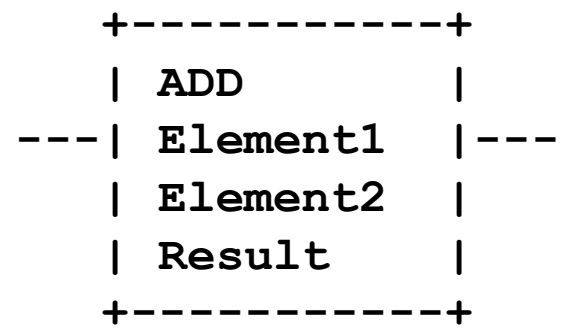
```
if (A or D) {
    X = B; Y = A;}
else {
    X = D or C; Y = not X;}
```



Other instructions

Arithmetic/logic instructions

Symbol (for sum)

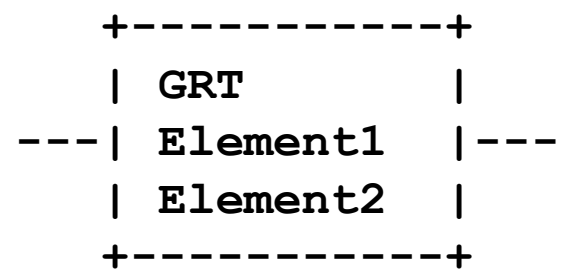


- If the current reaches the block, the variables **Element1** and **Element2** are added together and the result is placed in the variable **Result**.
- In addition to **ADD** there are **SUB** (subtraction), **MUL** (multiplication), **DIV** (division), **AND** (bit-to-bit binary multiplication), **OR** (bit-to-bit binary addition).

Other instructions

Comparisons

Symbol (for greater than)

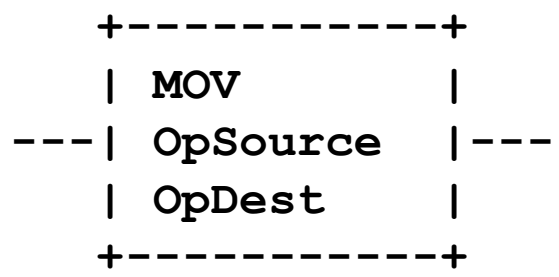


- The block behaves like a contact, which is closed if **Element1** is greater than **Element2** and opened otherwise.
- Other comparisons in addition to **GRT** area available, like **EQU** (equal to), **NEQ** (different from), **GEQ** (greater than or equal to), **LEQ** (less than or equal to), **LES** (less than).

Other instructions

Transfer

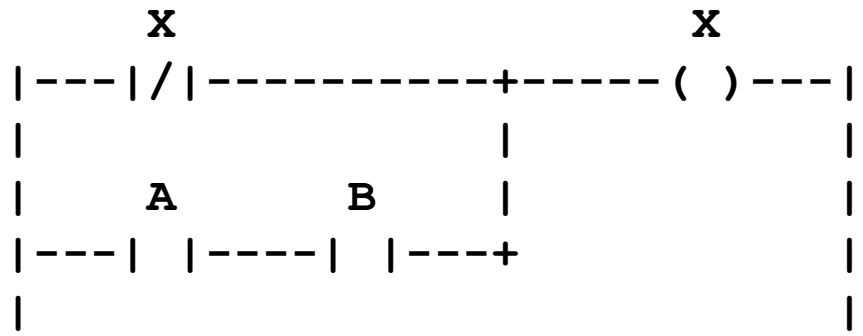
Symbol



- If the current reaches the block, the **OpSource** content is transferred (i.e. copied) to **OpDest**

Ladder diagrams: examples

Consider the following ladder diagram:



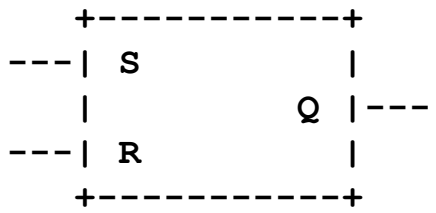
The logic function is:

X = NOT(X) OR (A AND B)

On the right side, **X** denotes the previous value of the variable, while on the left side the new value. This is reflected in the ladder diagram where the **X** on the contact denotes the original value of the variable, the **X** on the coil the new one.

Ladder diagram: examples

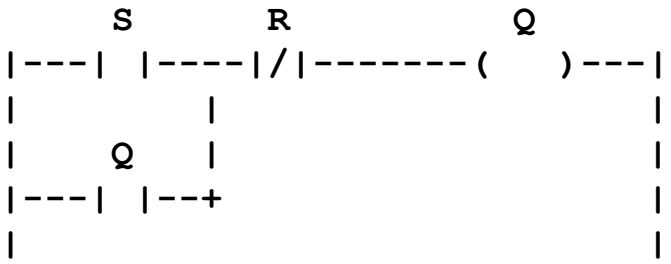
Flip-Flop SET RESET



The output Q is set to 1 if the input S (Set) is set to 1 and remains in such state until the input R (reset) is set to 1: this makes in any case the output switch to 0 (the Reset input is at higher priority than the Set input).

If neither the Set nor the Reset input are high (1), the output Q keeps the value of the previous cycle:

Q = (NOT R) AND (Q OR S)



This can be the logics to enable a motor:

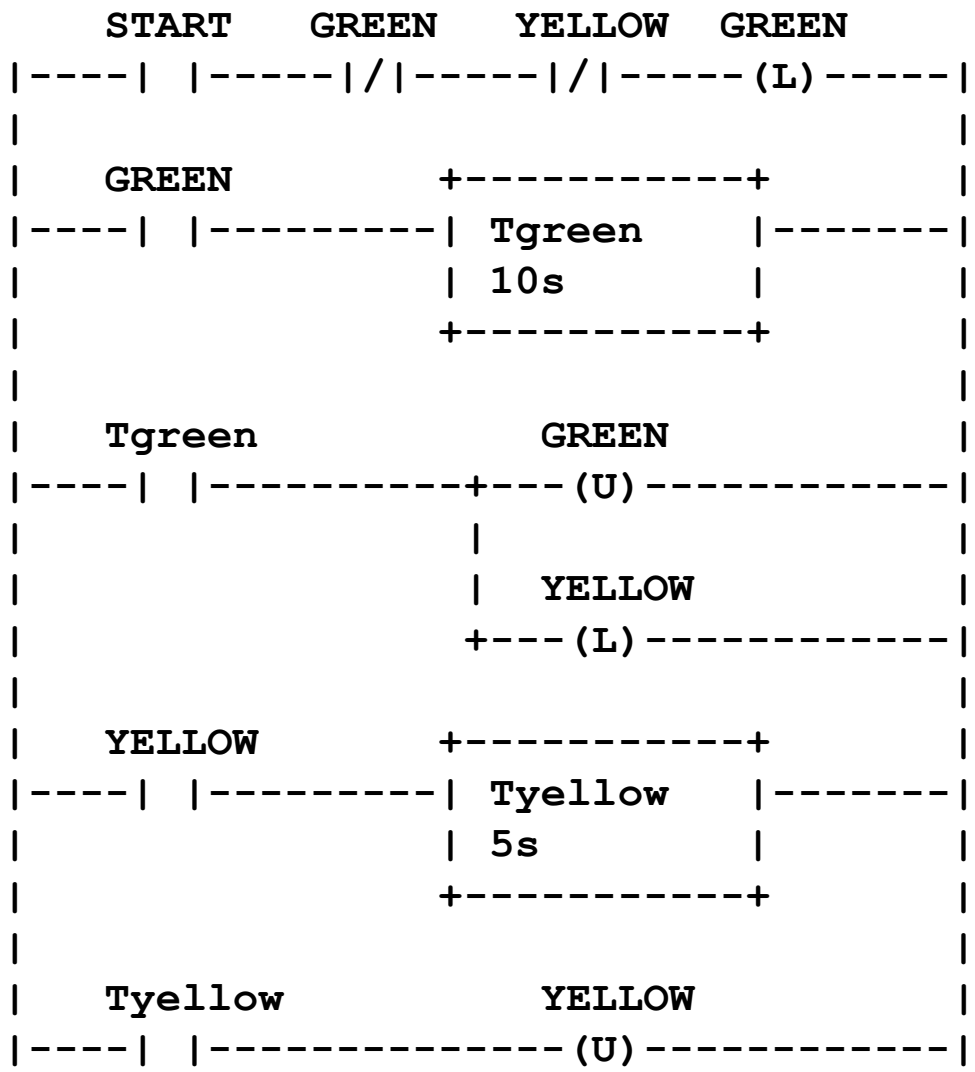
- S: enabling switch
- R: disabling switch
- Q: motor running

Ladder diagram: examples

Consider the following process:

Pressing a button (**START**), a green light (**GREEN**) turns on for 10s, then it turns off and a yellow light (**YELLOW**) turns on for 5s.

We want to write in LD a PLC program that implements these functions, using the **START** signal as input, and **GREEN** and **YELLOW** as outputs.



Ladder diagram: examples

Consider the following process:

Pressing a **START** button activates an irrigation system with two separate lines for a total time of 30 min. If the humidity sensor **UMID** is active, then **LINE1** comes into operation while **LINE2** remains off, otherwise **LINE2** comes into operation and **LINE1** remains off. This must happen for 30 min continuously. After 30 min everything must turn off, until the next **START**.