

PLC programming

Paolo Burgio

paolo.burgio@unimore.it



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

High Performance
Real Time **Lab**

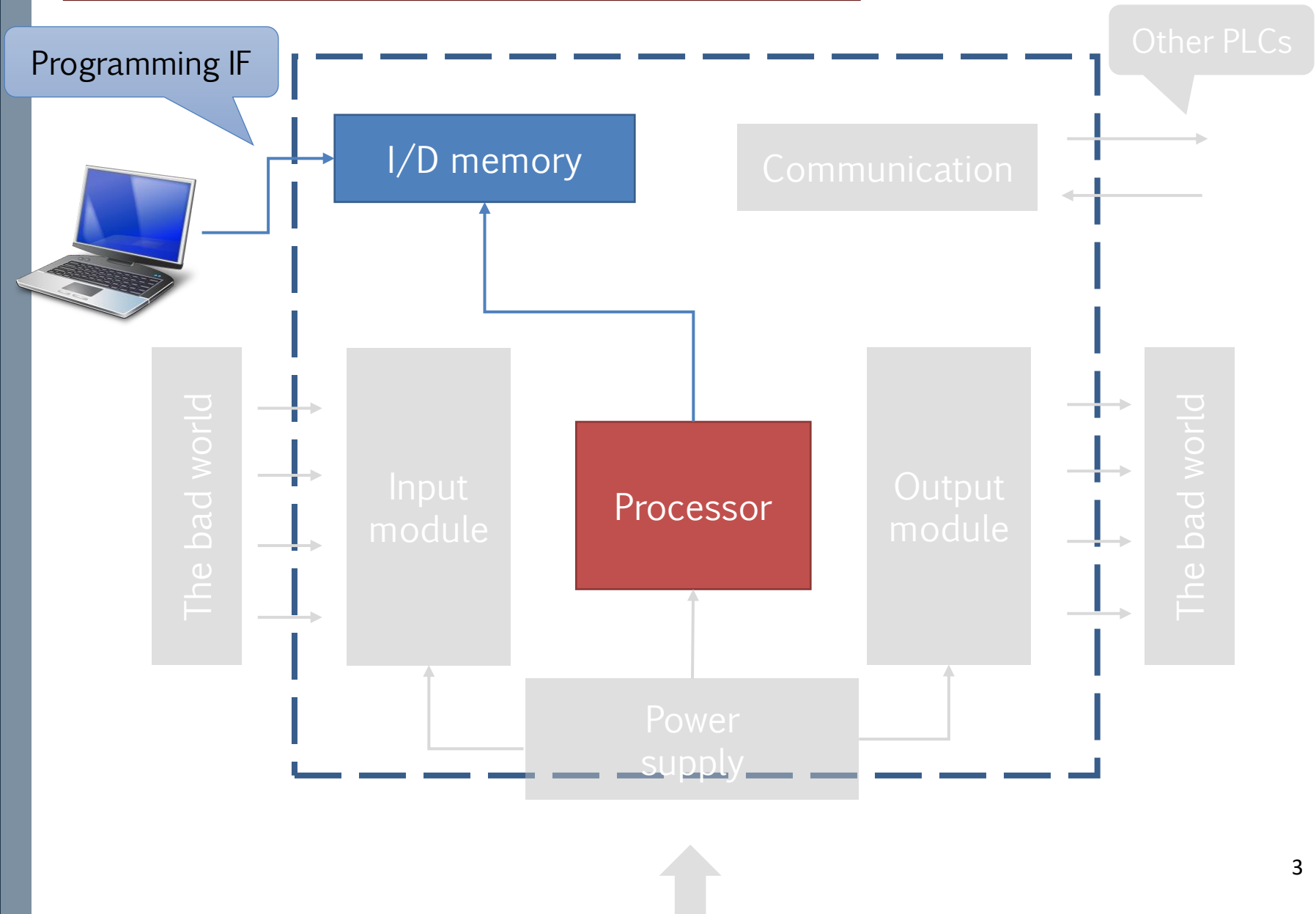
```
while (not edge) {  
    run();  
}
```

```
do {  
    run();  
} while (not edge);
```





Structure of a PLC





PLC programming with IEC 61131

First attempt of standardization => IEC 61131-3 standard

- › yr 1993, latest rev 2013
- › Before that, “the fish market” of languages
- › Still, Ladder was a prominent one...but everyone had its own variant!!

States that there are 5 “standard” ways of programming PLCs

- › **Ladder diagram**
 - Description of electrical wiring, designed for non-informatics
- › **Function Block Diagram – FBD**
 - From electronics
- › **Sequential Functional Chart – SFC**
 - Petri-net style
- › **Instruction List – IL**
 - ASM-like
- › **Structured Text – ST**
 - Similar to Pascal/VB



IEC 61131

Covers the complete lifecycle of PLC modules and ~~sw development for PLC~~ PLC programming

- › Part 1: definition of terminology and concepts
- › Part 2: electronic and mech equipment and verification/testing
- › Part 3: programming languages (5 types)
- › Part 4: how to choose, install and maintain
- › Part 5: how to communicate (MMS – Manufacturing Messaging Specification)
- › Part 6: communication via fieldbusses/other ind. standards
- › Part 7: fuzzy control (won't see this..if you don't want)
- › Part 8: sw dev guidelines



Ladder

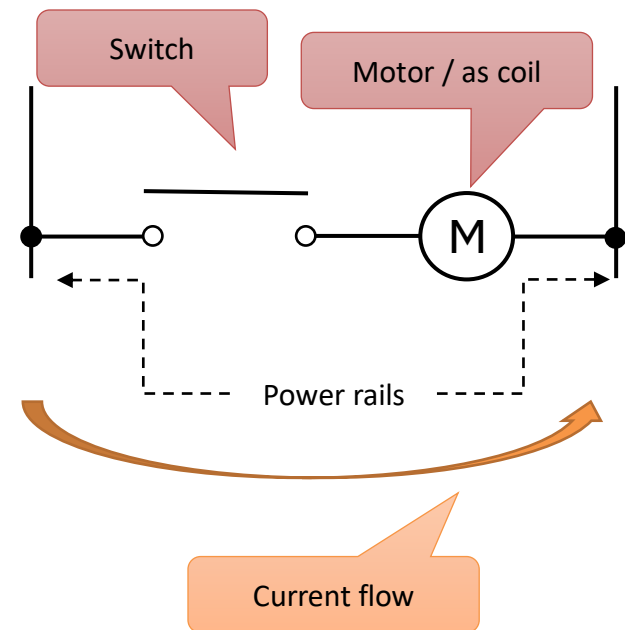
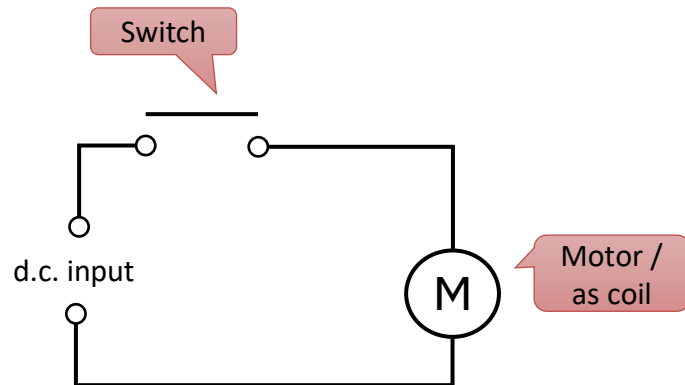




Ladder diagrams

Possible circuit to power on a motor

- › Left: electrical diagram; right: Ladder
- › Does it remind of something...?





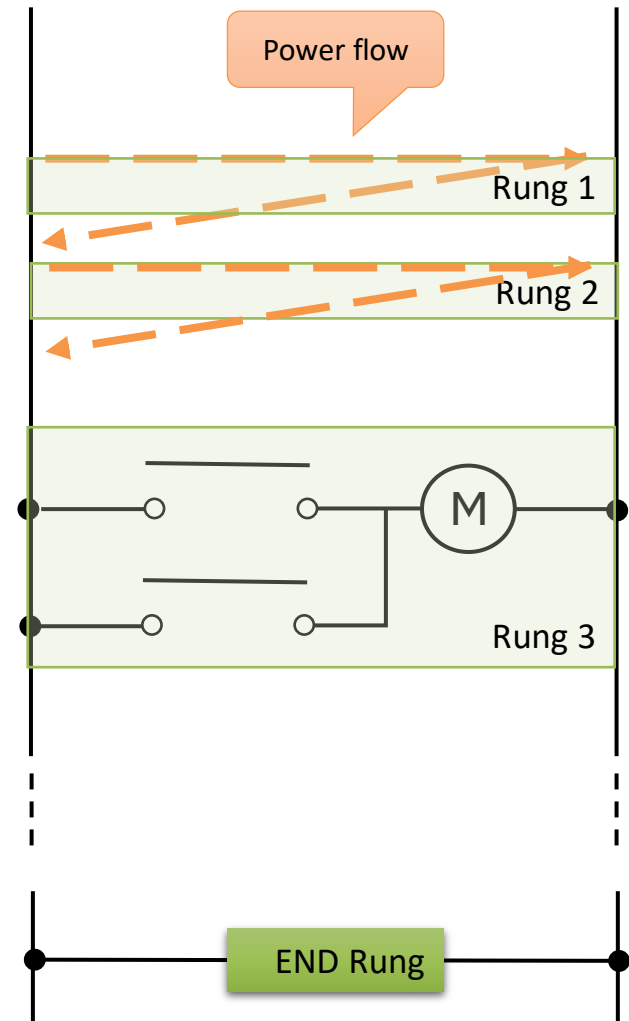
Ladder programs

Writing a program is as easy as writing a switching circuit

- › Vertical lines are power rails, and power flows from top-left to bottom-right
 - So does “program flow”
- › Horizontal lines (rungs) connect power rails
 - Every rung starts with one or more inputs, and ends with exactly one output
- › Typical exec time: 1ms for 1k bytes of program, so usually approx 10-50ms

Program flow

- › Store input status in mem
- › Read inputs from memory, run program, store out in mem
- › Update all outputs





Ladder symbols

› Power rails (Vert lines)

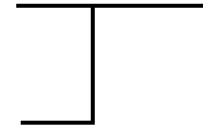


› Rungs (Horiz lines)



› Left/right connection between power rail and rungs

› Dual connection



› Normally open (NO) contact



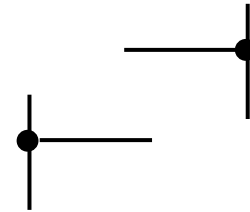
› Normally closed (NC) contact



› Output coil (from a lamp, a motor...)



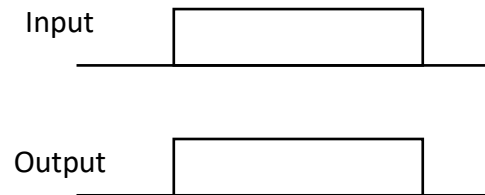
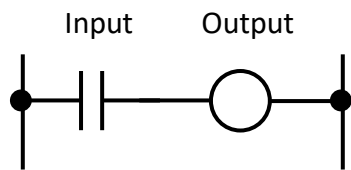
› A switch



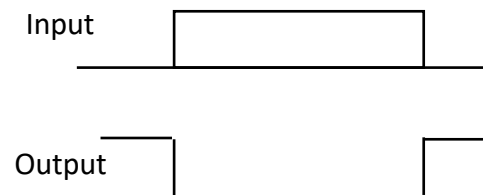
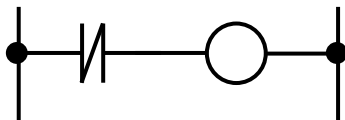



Example: a (lamp?) switch

- › When a NO contact is closed, the attached coil propagates the signal




- › When a NC contact is opened, the attached coil propagates the input signal





Structured Text





Structured text

- › Inspired by Pascal language, also Basic and VB programmer might find it familiar
- › Procedural language (can define Functions, or Function Blocks)
- › Or define a Program (and call it from within your Application MainTask)
 - Can have multiple programs...you must have one call for one program!

Can also comment code..

```
(* This is a comment! *)  
  
(* This is a  
multi-line  
comment! *)
```



Defining variables

Can also initialize them

› Don't forget semicolon ;

IEC 61131.3 defines several datatypes

› Few examples

```
Contact1: BOOL;
```

```
Contact2: BOOL := FALSE;
```

IEC Data Type	Format	Range
SINT	Short Integer	-128 ... 127
INT	Integer	-32768 ... 32767
DINT	Double Integer	$-2^{31} \dots 2^{31}-1$
UINT	Unsigned Integer	$0 \dots 2^{16}-1$
BOOL	Boolean	1 bit
BYTE	Byte	8 bits
WORD	Word	16 bits
STRING	Character String	'My string'
TIME	Duration of time after an event	T# 10d4h38m57s12ms TIME# 10d4h38m
DATE	Calendar date	D# 1989-05-22 DATE# 1989-05-22
REAL	Real Numbers	$\pm 10^{\pm 38}$



Variables: simple operations

- › Assigning immediate vals to variables

```
Contact1 := FALSE;  
Input1  := 11;  
Output1 := 5;
```

```
Contact1: BOOL;  
Input1: INT;  
Output1: BYTE;
```

- › Also, with operators in R-values

```
Output1 := Input1 - Output1 / ( Input6 + 3 );
```





Structured Text Operators (and their priority)

Operators	Description
(. . .)	Parenthesized (brackets) expression
Function (. . .)	List of parameters of a function
**	Power
-, NOT	Negation, Boolean NOT
*, / , MOD	Multiplication, division, modules operations
+, -	Add, subtract
< , > , <= , >=	Comparison
=, <>	Comparison
AND, OR	Boolean operator
XOR	Exclusive OR
OR	Boolean OR

Precedence



Define your own datatypes

```
(* Enum-like datatype *)  
TYPE Motor:( Stopped, Running );  
END_TYPE;
```

```
(* Analog value datatype *)  
TYPE Pressure: REAL;  
END_TYPE;
```




Variables: nomenclature

- › ST is not case sensitive; in case you might want to use capital letter for clarity
- › Use the AT keyword to fix the memory location of a variable

```
Contact1: BOOL AT %MX100; (* Internal memory Bit at address 100 *)  
Input1: INT AT %IW200; (* Input memory Word at address 200 *)  
Output1: BYTE AT %OB300; (* Output memory Byte at address 300 *)
```

% **M** **X** 100

Memory Address

Memory type

M - internal
I - input
O - output

Data type

X (bit) - 1 bit
B (byte) - 8 bit
W (word) - 16 bits
D (double word) - 32 bits
L (long word) - 64 bits



If-Then-Else

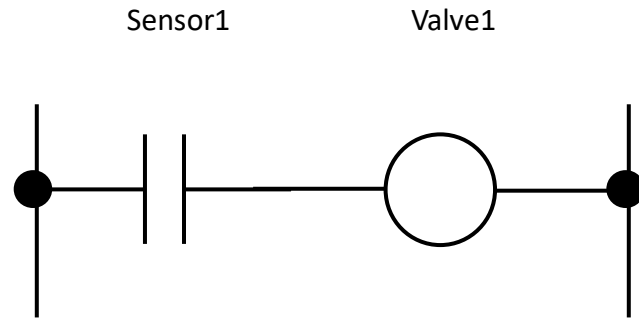
```
IF Contact1 = TRUE THEN;  
    Coi11 := TRUE;  
ELSE;  
    Coi11 := FALSE;  
END_IF;
```

```
IF NOT Contact1 = TRUE;  
    Coi11 := TRUE;  
END_IF;
```

```
IF Contact1 = TRUE OR Input1 = 11;  
    Coi11 := TRUE;  
END_IF;
```



Ladder vs. ST



```
Valve1 := Sensor1  
IF Sensor1 THEN;  
    Valve1 := 1;  
END_IF;
```





Switch-case

- › Multiple instructions on the same line
- › Default with ELSE keyword
- › Can also use ranges

```
CASE (State) OF
  1: nextState := 2; ERROR := FALSE;
  2: nextState := 3; ERROR := FALSE;
ELSE
  nextState := 4; ERROR := TRUE;
END_CASE;
```

```
CASE (Temperature) OF
  0...40: Furnace_switch := ON;
  40...100: Furnace_switch := OFF;
END_CASE ;
```



Loops

› For, While-Do, Repeat-Until

```
(* for (input1 = 10;  
      input1 >= 0;  
      input1-- *)  
FOR Input1 := 10 to 0 BY -1  
DO  
    Output1 := Input1;  
END_FOR;
```

```
(* do...  
  while (input1 < 3  
        && input2 == 5); *)  
Output1 := 0;  
REPEAT  
DO  
    Output1 := Output1 + 1;  
UNTIL Input1 < 3 AND Input2 = 5  
END_REPEAT;
```

```
(* while (input1 < 3  
          && input2 == 5) *)  
Output1 := 0;  
WHILE Input1 < 3 AND Input2 = 5  
DO  
    Output1 := Output1 + 1;  
END_WHILE;
```



Defining Programs

- › Read (input vars) – Exec – Write (output vars) paradigm

```
(* ... *)
```

```
VAR (* Input type, and datatype,  
    implicit by memory *)
```

```
    Temperature AT $IW100;
```

```
END_VAR;
```

```
(* ... *)
```

```
PROGRAM Example
```

```
VAR_IN (* Input *)
```

```
    Temperature: INT;
```

```
END_VAR;
```

```
VAR_IN (* Input *)
```

```
    Speed: INT = 50;
```

```
END_VARS;
```

```
VAR_OUT (* Outputs *)
```

```
    Motor_speed: REAL;
```

```
END_VAR;
```

```
(* Instructions here *)
```

```
END_PROGRAM;
```



Logic blocks

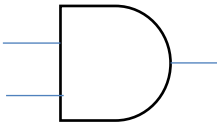


(Unnecessary) brief recap

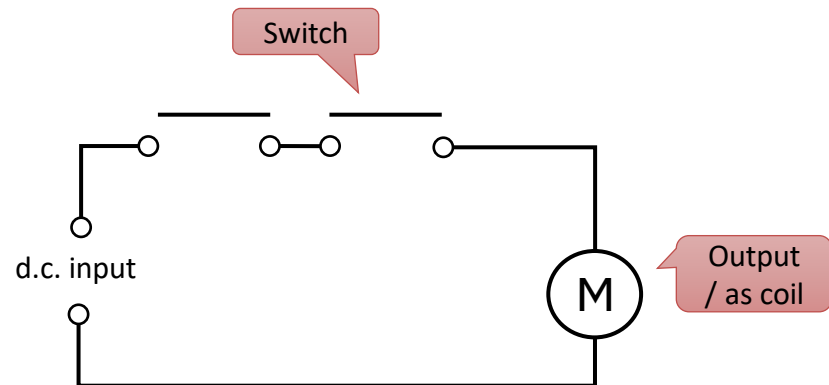
Logic functions

- › Already seen in previous courses
- › Express Boolean logics, adders, latches, ...

Example: Logical AND

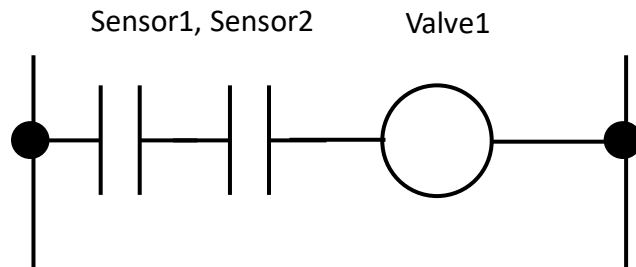


A	B	Output
0	0	0
0	1	0
1	1	1
1	0	0

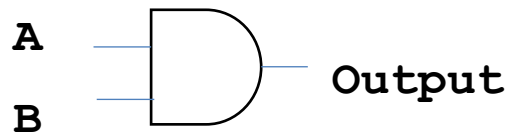




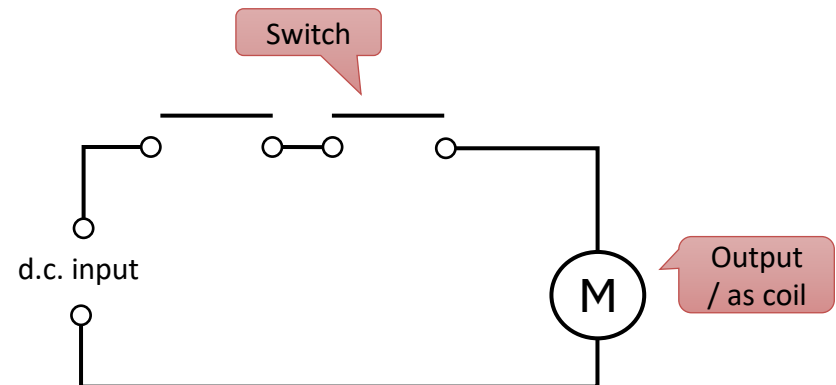
Ladder and ST AND



```
IF Sensor1 AND Sensor2 THEN;  
    Valve1 := TRUE;  
ELSE  
    Valve1 := FALSE;  
END_IF;
```

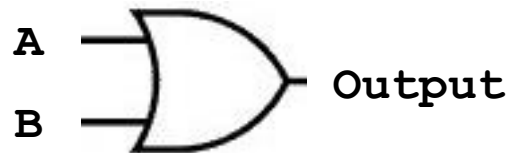


A	B	Output
0	0	0
0	1	0
1	1	1
1	0	0

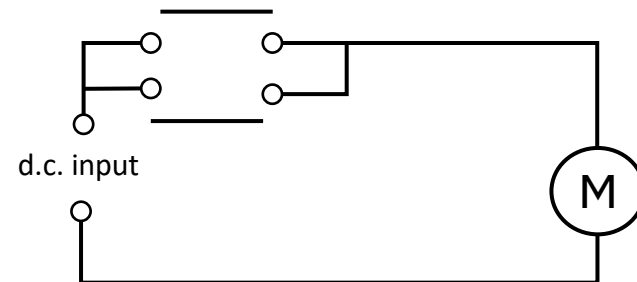




Ladder and ST OR

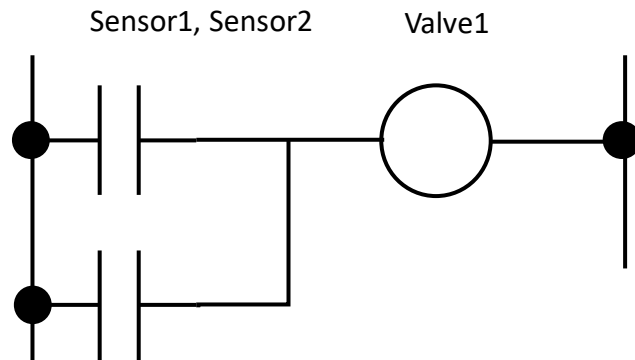


A	B	Output
0	0	0
0	1	1
1	1	1
1	0	1

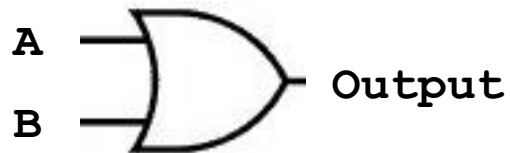




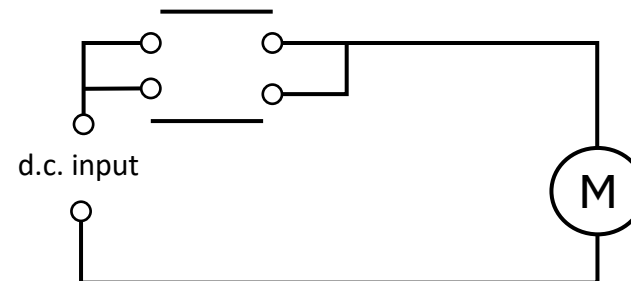
Ladder and ST OR



```
IF Sensor1 OR Sensor2 THEN;  
    Valve1 := TRUE;  
ELSE  
    Valve1 := FALSE;  
END_IF;
```

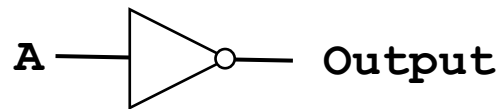


		Output
A	B	
0	0	0
0	1	1
1	1	1
1	0	1

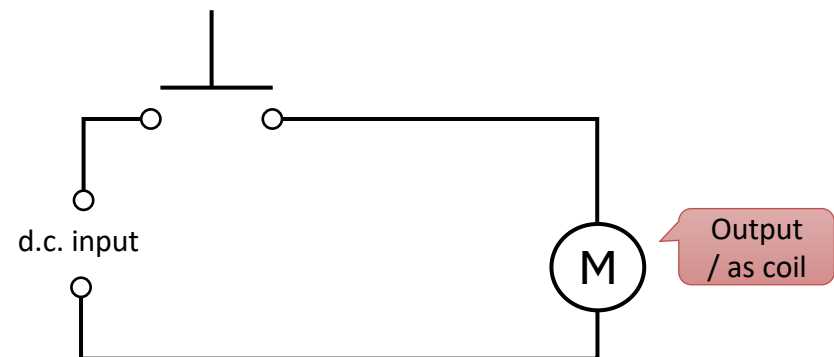




Ladder and ST NOT

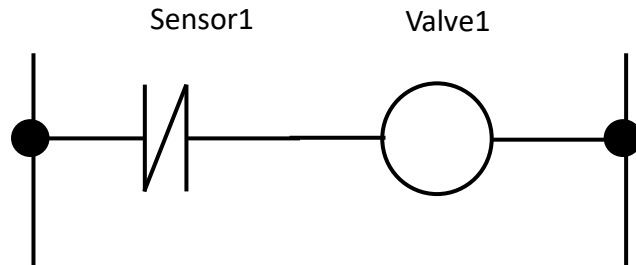


A	Output
0	1
1	0

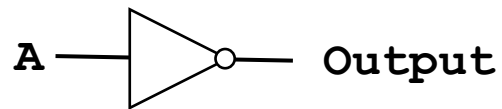




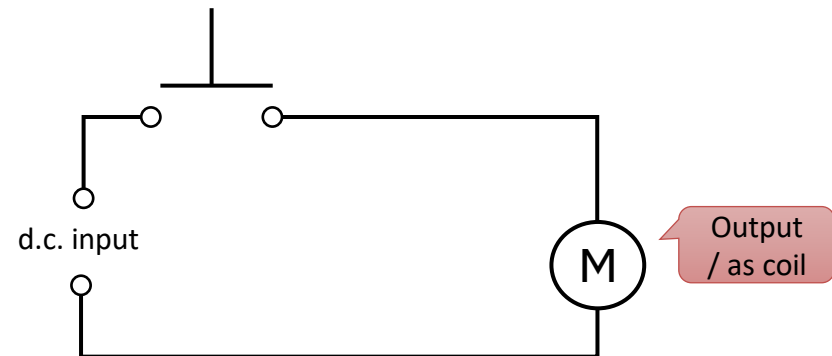
Ladder and ST NOT



```
IF NOT Sensor1 THEN;  
  Valve1 := TRUE;  
ELSE  
  Valve1 := FALSE;  
END_IF;
```



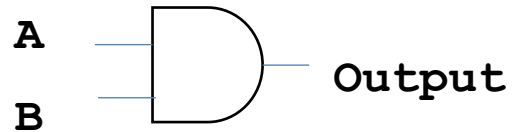
A	Output
0	1
1	0



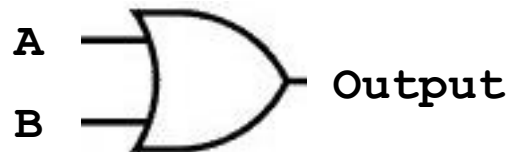


Boolean algebra

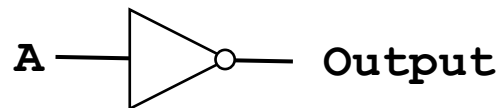
› $A * B = \text{Out}$



› $A + B = \text{Out}$

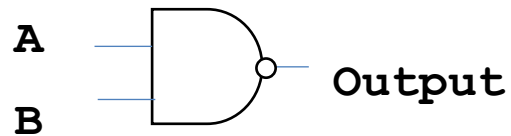


› $\overline{A} = \text{Out}$





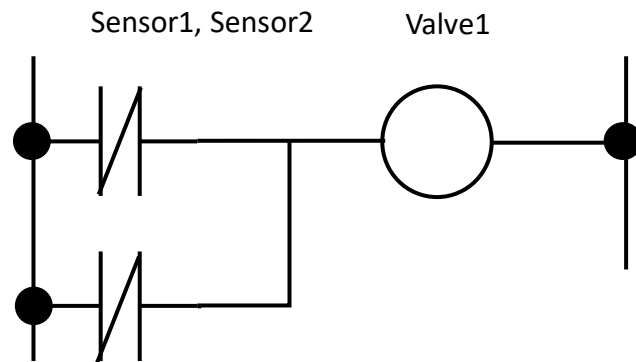
Ladder and ST NAND



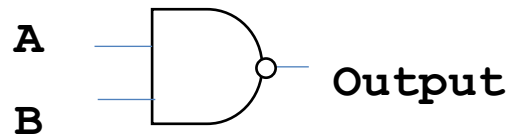
A	B	Output
0	0	1
0	1	1
1	1	0
1	0	1



Ladder and ST NAND



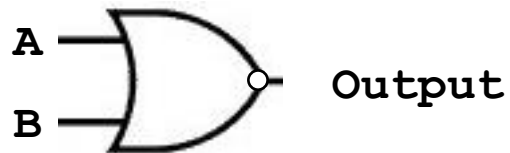
```
IF Sensor1 AND Sensor2 THEN;  
    Valve1 := FALSE;  
ELSE  
    Valve1 := TRUE;  
END_IF;
```



		Output
A	B	
0	0	1
0	1	1
1	1	0
1	0	1

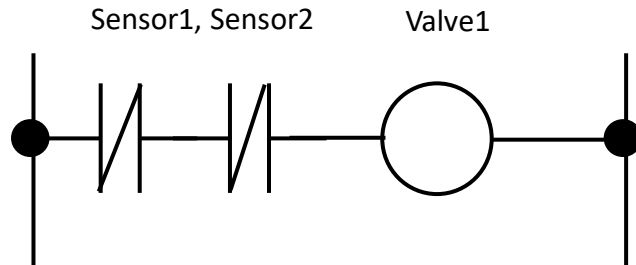


Ladder and ST NOR



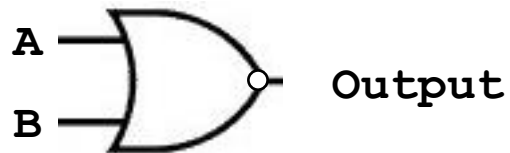
A	B	Output
0	0	1
0	1	0
1	1	0
1	0	0

Ladder and ST NOR



```

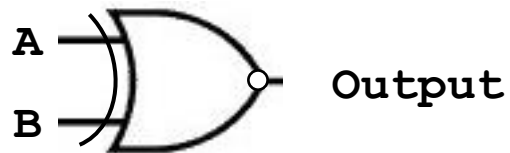
IF Sensor1 OR Sensor2 THEN;
    Valve1 := FALSE;
ELSE
    Valve1 := TRUE;
END_IF;
  
```



A	B	Output
0	0	1
0	1	0
1	1	0
1	0	0



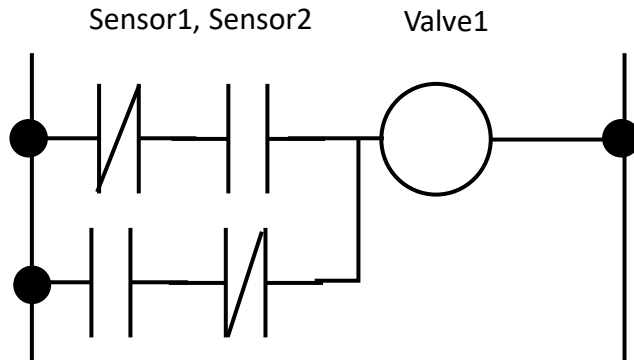
Ladder and ST XOR



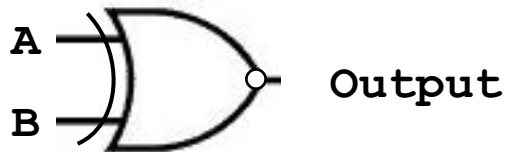
A	B	Output
0	0	0
0	1	1
1	1	0
1	0	1



Ladder and ST XOR



```
IF (Sensor1 OR Sensor2)
  AND NOT (Sensor 1 AND Sensor2)
THEN;
  Valve1 := FALSE;
ELSE
  Valve1 := TRUE;
END_IF;
```



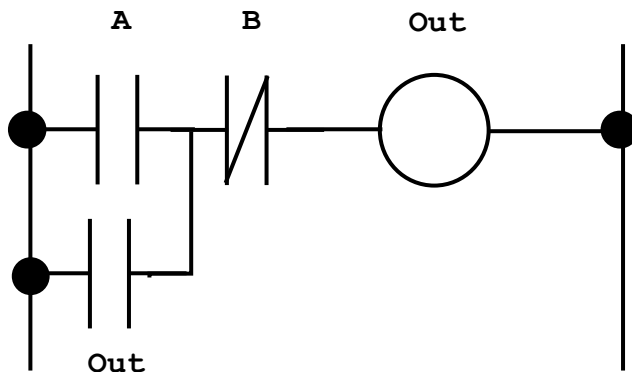
A	B	Output
0	0	0
0	1	1
1	1	0
1	0	1

Latching

- › We often need to store in memory a value (e.g. a bit)
 - I.e., to store a **state**
- › Can implement it with NANDs, or NORs

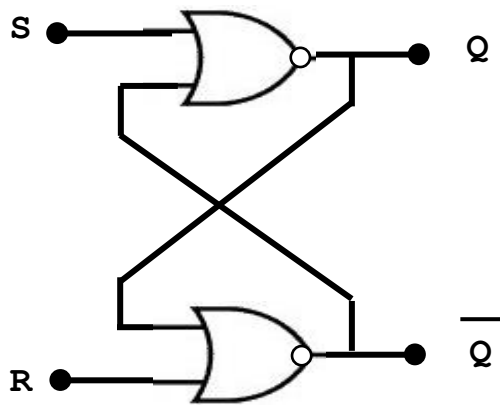
SR Latch circuit

- › When **A** closes ('1'), **Out** coil gets energy ('1')
- › Also, **Out** contact closes, so, even if **A** opens ('0'), the “OR” keeps the **Out** coil powered ('1')
- › Until the NC **B** contacts closes ('0'). Then the **Out** coil becomes '0'





Latch with NOR



We also have the negate available for free!

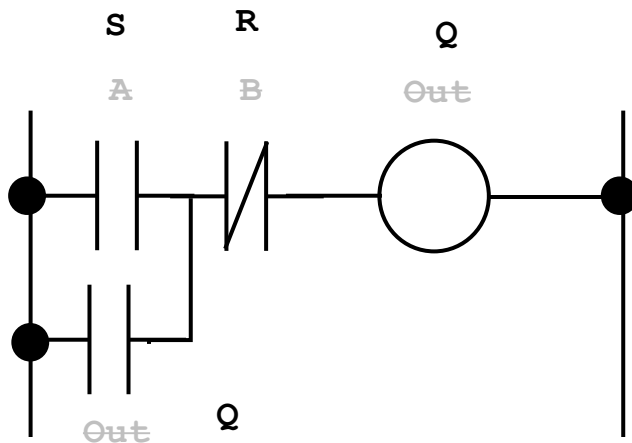
		Q
S	R	Q
0	0	LATCH
0	1	0
1	0	1
1	1	-

Memory

Reset

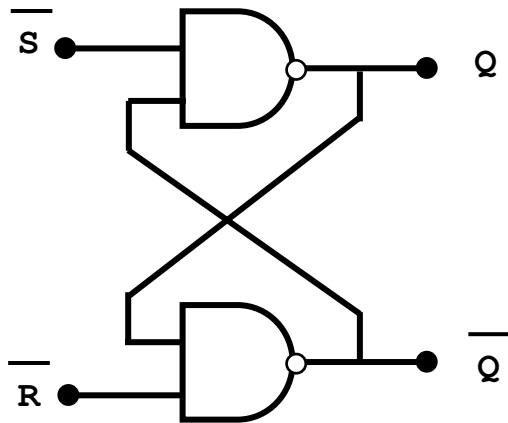
Set

Not legal

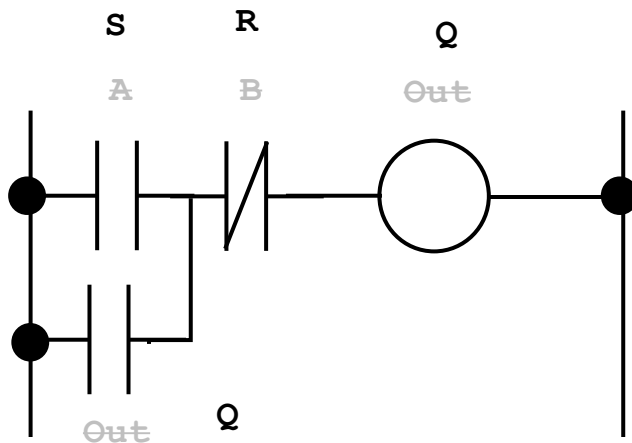





Latch with NAND




		Q
S	R	Q
0	0	-
0	1	0
1	0	1
1	1	LATCH





Functional Block Diagrams

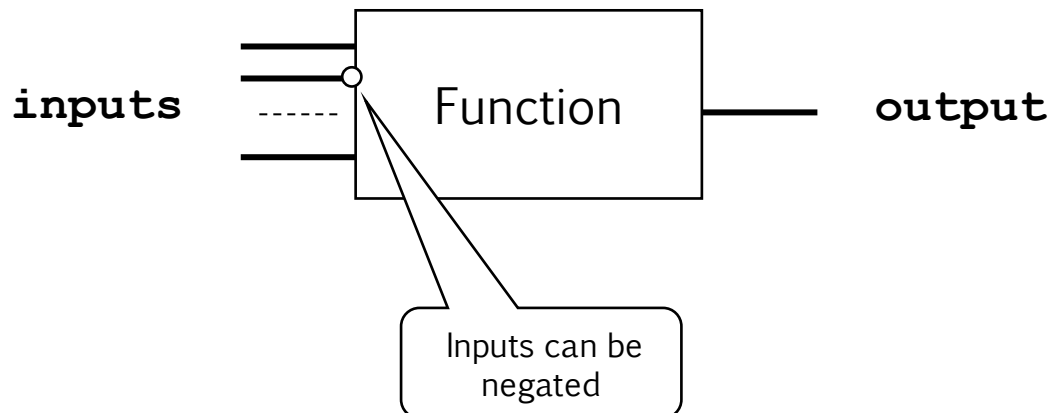


Function blocks

Enable us grouping a functionality in a reusable elements

- › “aka”: function 😊
- › Multiple inputs, only one output
- › Standard-defined functions vs. user-defined functions

IEC 61131-3 defines them graphically as

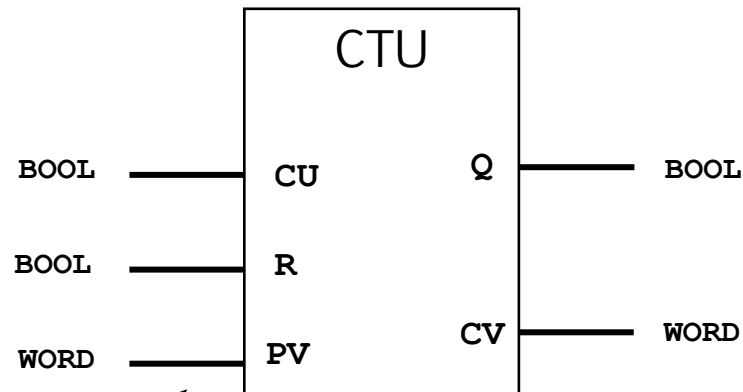




Standard blocks: CTU

Up counter **CTU**

- › Gives output **Q** when *input pulse* **CU** reaches the *set value* **PV**
- › Reset input **R**
- › Also, every time **Q** is set, **CV** is incremented



WORD is unsigned; **INT** is signed, which makes no sense here

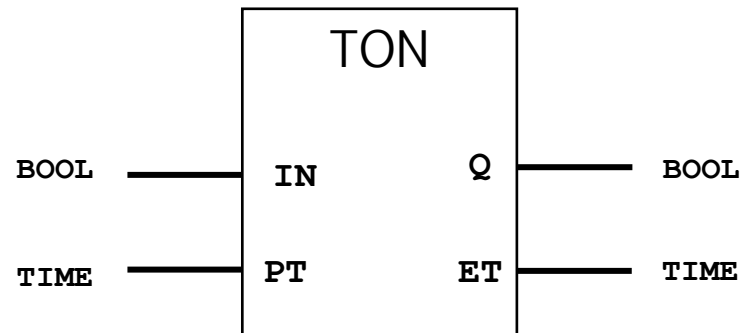




Standard blocks: TON

Timer ON **TON**

- › Gives output **Q** when timer gets to time to pass **PT** has a rising edge
- › Starts when **IN** has a rising edge, reset when **IN** has a falling edge
- › Also, *Elapsed Time* **ET** keeps trace of time





Defining our own Function Blocks

- › Here, written in ST
- › Define function to reuse code
- › Define in, out, internal vars

```
FUNCTION_BLOCK MyFb

VAR_INPUT (* Inputs *)
  (* ... *)
END_VAR;

VAR_OUTPUT (* Outputs *)
  (* ... *)
END_VAR;

VAR (* Internal *)
  (* ... *)
END;

(* Instructions here *)

END_FUNCTION_BLOCK;
```

References



Course website

- › http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

- › paolo.burgio@unimore.it
- › <http://hipert.mat.unimore.it/people/paolob/>

Resources

- › W. Bolton, "Programmable Logic Controllers", 6th edition, Newnes
- › "Industrial informatics" course by Proff. Vezzani and Pazzi @UNIMORE
- › A "small blog"
 - <http://www.google.com>