Lezione #8

Programmazione dei PLC

Sommario

- Variabili e tipi
- Unità di organizzazione della programmazione
- Linguaggi di programmazione per i PLC
  - Linguaggi testuali: Instruction List e Structured Text
  - Linguaggi grafici: Function Block Diagram, Ladder Diagram, Sequential Functional Chart

Introduzione

- La programmazione dei PLC è regolata dallo **standard CEI 61131, parte 3**
- Tale standard descrive in dettaglio anche aspetti non strettamente legati alla programmazione (e.g. il numero di bit da usare per una certo tipo)
- Lo standard è orientato alla progettazione di software **modulare**, **leggibile** e **ben strutturato**

Premessa

- In queste lezioni introdurremo lo standard “così com’è”, in particolare con riferimento al testo di Chiacchio-Basile [3]

- L’effettiva implementazione potrebbe essere leggermente diversa da un caso all’altro
(e.g. dichiarazione delle variabili in OpenPLC)

Costanti, variabili, tipi

- Costanti e variabili sono il mezzo tramite cui è possibile **rappresentare i dati** all'interno del dispositivo
- Lo standard prevede alcuni **tipi predefiniti**

- La definizione avviene in **forma testuale** (indipendentemente dal linguaggio usato)
- **Funzioni** e **blocchi funzionali** possono avere variabili di ingresso e uscita definite in forma grafica

Costanti, variabili, tipi

Tipi predefiniti

- **Numeri interi**
  - `INT` interi (16 bit, da -2^15 a 2^15 -1)
  - `UINT` interi (16 bit, da 0 a 2^16 -1)
- **Numeri reali**
  - `REAL`
  - `LREAL`
- **Variabili temporali**
  - `TIME` durata temporale (e.g. T#1d3h5m12s50ms)
  - `DATE, TIME_OF_DAY` rispettivamente data e ora del giorno
  - `DATE_AND_TIME` data e ora insieme

Costanti, variabili, tipi

Tipi predefiniti

- **Stringhe di caratteri**
  - `STRING`
- **Stringhe di bit**
  - `BOOL` singolo bit o variabile logica
  - `BYTE` 8 bit
  - `WORD` 16 bit
  - `DWORD` 32 bit
  - `LWORD` 64 bit

Costanti, variabili, tipi

Variabili generiche

- **Rappresentano qualunque tipo** (in un dato insieme)
  - `ANY` qualsiasi tipo
  - `ANY_NUM` qualsiasi tipo numerico
  - `ANY_INT` qualsiasi intero
  - `ANY_REAL` qualsiasi reale
  - `ANY_DATE` qualsiasi variabile temporale
  - `ANY_BIT` qualsiasi stringa di bit

Costanti, variabili, tipi

È inoltre possibile definire **tipi derivati**

- Per **equivalenza** a quelli definiti
- Per **enumerazione**
- Per **restrizione** di tipi già definiti
- Definendo insiemi ordinati di elementi dello stesso tipo (***array***)
- Definendo insiemi di elementi diversi (**strutture**)

Costanti, variabili, tipi

Esempio

```
TYPE
        (* tipo ottenuto come restrizione di UINT *)
        impulsi : UINT(0..1000);

        (* tipo ottenuto per enumerazione *)
        stato : (fermo, funzionante, guasto, attesa);

        (* tipo equivalente a REAL, ma *)
        (* con valore di default 20 e non 0 *)
        temperatura : REAL := 20.0;


        [...]
```

Costanti, variabili, tipi

Esempio (cont'd)

```
(* struttura *)
sensore_temperatura : STRUCT
      valore : temperatura;
      ultima_calibrazione : DATE;
      intervallo_calibrazione : TIME;
      valore_massimo : REAL := 100.0;
      diagnostica : BOOL;
END_STRUCT
(* array di strutture *)
dati_termici_forno : ARRAY[1..10] OF
            sensore_temperatura;
(* array di array *)
dati_termici_insieme_forni : ARRAY[1..4,1..4] OF
            dati_termici_forno;
END_TYPE
```

Costanti, variabili, tipi

L'accesso alle variabili con più elementi può avvenire

- Per **indice** nel caso di array
  - `forno_1[7]`
- Per **campo** nel caso di strutture
  - `sensore_1.valore`

Costanti, variabili, tipi

- **Dichiarazione delle variabili**
  - All'inizio di programmi, funzioni e blocchi funzionali
  - `VAR … END_VAR`

- **Inizializzazione**
  - L'inizializzazione può avvenire anche all'atto della dichiarazione
  - La **non ambiguità** è un requisito fondamentale: all'atto della dichiarazione di una variabile deve **sempre essere assegnato un valore,** eventualmente di default:
    - 0 per le variabili numeriche
    - Stringa nulla
    - 01/01/0001 per le variabili data

Costanti, variabili, tipi

Esempio inizializzazione

```
VAR
      A, B : REAL;
      abilitazione : BOOL;
      conteggio : impulsi;
      stato_tornio : stato;
      termometro_7 : temperatura := 0.0;
      termocoppia_1, termocoppia_2 :
            sensore_temperatura;
      forno_1 : dati_termici_forno;
END_VAR
```

Costanti, variabili, tipi

Un programma eseguito su un PLC tipicamente avrà variabili

- Di **ingresso**: in genere sono associate a un sensore (*read-only*) o agli ingressi di funzioni e blocchi funzionali
  - `VAR_INPUT … END_VAR`
- Di **uscita**: associate ad attuatori o a parametri di ritorno di un blocco funzionale
  - `VAR_OUTPUT … END_VAR`
- Di **ingresso/uscita**: fanno riferimento all'indirizzo di variabili esterne alla POU ma da essa modificabili
  - `VAR_IN_OUT…END_VAR`
- **Interne**: dati temporanei o di appoggio

Costanti, variabili, tipi

- La **visibilità** di una variabile è interna alla POU in cui è dichiarata; la variabile non è accessibile dall'esterno (ad eccezione di variabili di I/O)

- È possibile dichiarare **variabili globali**
  - `VAR_GLOBAL … END_VAR`

Costanti, variabili, tipi

- Le variabili globali sono visibili alle unità **interne a quella in cui sono dichiarate**
- È possibile dichiararle solo a livello di **programma, risorsa e configurazione** (vedi dopo)
- Per utilizzare una variabile globale esterna bisogna **dichiararla**
  - `VAR_EXTERN … END_VAR`

Costanti, variabili, tipi

- Le **variabili a rappresentazione diretta** consentono di riferirsi a specifiche locazioni di memoria

- Queste variabili hanno una notazione particolare

Costanti, variabili, tipi

- %ABxxx
  - A è il prefisso di **locazione**
    - I per locazioni riservate agli ingressi
    - Q per locazioni riservate alle uscite
    - M per locazioni di memoria generiche
  - B è il prefisso di **taglia**
    - X (o mancante) per un bit
    - B per un byte (8 bit)
    - W per una word (16 bit)
    - D per una word doppia (32 bit)
    - L per una word lunga (64 bit)
  - xxx è un codice che dipende dal dispositivo
    - In generale sono numeri interi, eventualmente separati da punti, che indicano la locazione precisa

Costanti, variabili, tipi

- Le **variabili accessibili** possono essere indirizzate da programmi remoti
  - `VAR_ACCESS … END_VAR`

- Astraggono le **funzionalità di comunicazione**
- Si possono riferire solo alle variabili di ingresso e di uscita di un programma, alle variabili globali e alle variabili a rappresentazione diretta
- Possono essere accessibili solo in lettura/scrittura

Costanti, variabili, tipi

- **Attributi delle variabili**
  - `RETAIN` variabile conservata in mancanza di alimentazione
  - `CONSTANT` variabile non modificabile dopo l'inizializzazione
  - `AT` specifica la locazione di memoria

- **Commenti:** `(* questo è un commento *)`

Unità di Organizzazione della Programmazione

- Lo standard prevede alcuni elementi comuni che prescindono dal linguaggio scelto

- In particolare, le **POU** **(Programmation Organization Units)** sono porzioni di codice riutilizzabili
  - Funzioni
  - Blocchi funzionali
  - Programmi

- Una POU è composta da
  - **Definizione** del tipo di POU e del nome
  - **Dichiarazione** di variabili e attributi
  - **Corpo** della POU (le istruzioni vere e proprie)

- **Funzioni**
  - Sono POU **riutilizzabili**
  - Calcolano **un solo dato in uscita**, rappresentato dal nome della funzione stessa (può essere usato direttamente nelle espressioni)
  - Il risultato **non deve dipendere da variabili interne** (stessi ingressi → stesse uscite)
  - Sono previste funzioni **predefinite** (`ADD, SUB, MAX, MIN, MUX, LIM, GT/LT/GE/LE/EQ/NE, SHL/SHR, AND/OR/NOT/XOR, LEN, INSERT…`)

- La **definizione delle funzioni** può avvenire in forma **grafica** o **testuale**

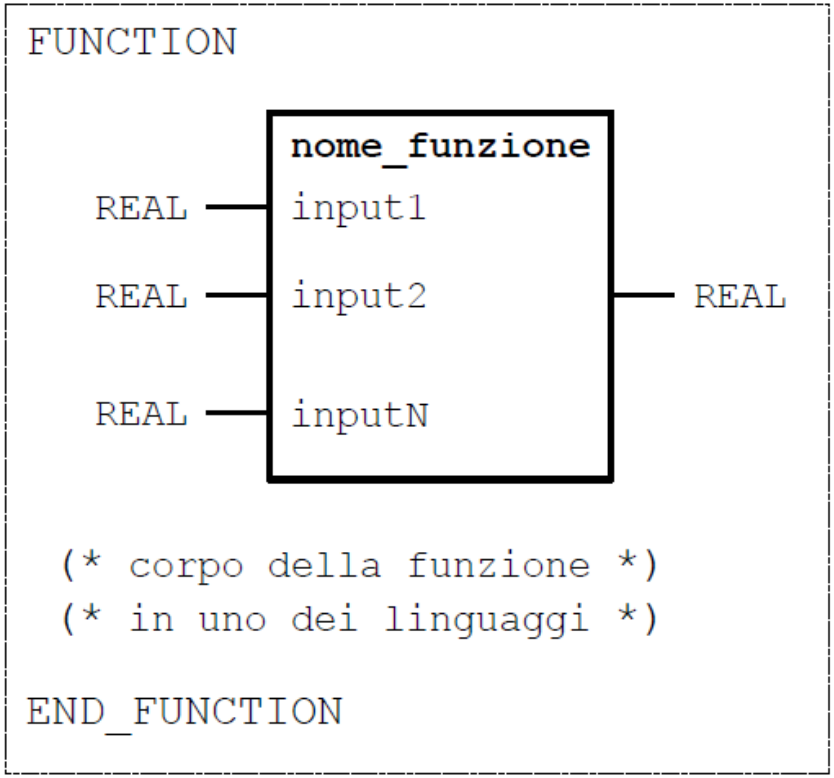
```
FUNCTION nome_funzione : tipo (* tipo della funzione *)
        VAR_INPUT
                (* definizione delle variabili di ingresso *)
                ....;
        END_VAR
                ...
                (* altre definizioni di variabili *)
                ...
                (* corpo della funzione in uno dei linguaggi *)
                (* deve prevedere l'assegnazione di un *)
                (* valore a nome_funzione*)
END_FUNCTION
```

- Nel caso di **definizione testuale**, il corpo della funzione può essere scritto in uno qualunque dei linguaggi dello standard (tranne SFC)

- **Non possono** essere definite variabili di uscita, di ingresso/uscita, a rappresentazione diretta, esterne, globali, accessibili, a ritenuta

- La **definizione delle funzioni** può avvenire in forma **grafica** o **testuale**

- Nel caso di **definizione grafica**, l'interfaccia della funzione è definita con un blocco rettangolare
- Vanno dichiarate le **variabili interne** e scritto il corpo in uno dei linguaggi disponibili



- L'uso di linguaggi grafici prevede
  - Un **ingresso implicito booleano** `EN`
    - Abilita l'esecuzione
  - Un'**uscita implicita booleana** `ENO`
    - Vera se la funzione viene eseguita senza errori

- Queste variabili sono usate nelle **catene di funzioni** per assicurarsi che un blocco abbia terminato l'esecuzione prima che cominci l'altro

- È possibile effettuare **overload** delle funzioni
  → usate con variabili di ingresso di tipo diverso (e.g. somma di reali o di interi)

- Si possono **estendere** alcune funzioni
  → possono avere un numero di ingressi non specificato (e.g. somma)

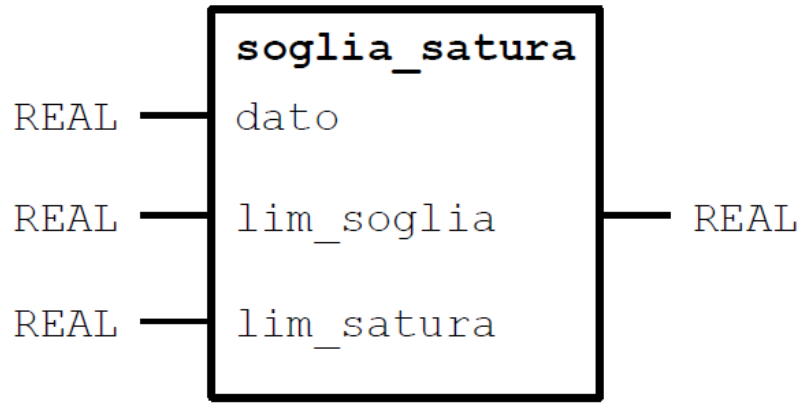
- Le funzioni possono richiamare programmi o blocchi funzionali, **ma non se stesse**
  (la *ricorsione* non è consentita)

- Esempio: definizione testuale

```
FUNCTION soglia_satura: REAL
      VAR_INPUT
            dato, lim_soglia, lim_sat : REAL;
      END_VAR
      IF ABS(dato) < lim_soglia THEN
            soglia_satura := 0.0;
      ELSE
            soglia_satura :=
                  MIN(MAX(dato, -lim_sat), lim_sat);
      END_IF
END_FUNCTION
```

- Esempio: definizione grafica

```
FUNCTION
```




```
  (* corpo della funzione *)
  (* in uno dei linguaggi *)

END_FUNCTION
```

- **Blocchi funzionali**
  - Le uscite possono **dipendere da uno stato interno, che viene conservato tra una chiamata e l'altra**
  - Un blocchi può chiamare altri blocchi o funzioni, ma **non è consentita la ricorsione**
  - Esistono blocchi **predefiniti** (ad esempio flip-flop SR, contatori…)

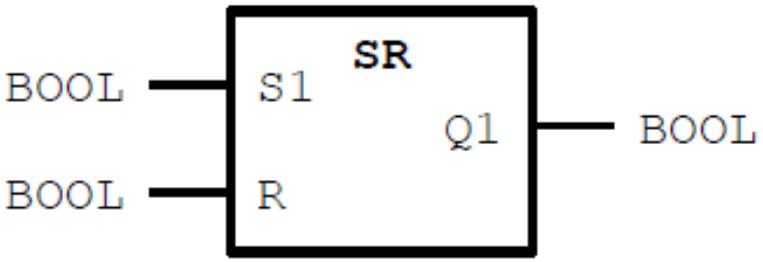
- Anche la **definizione dei blocchi funzionali** può avvenire in forma **grafica** o **testuale**

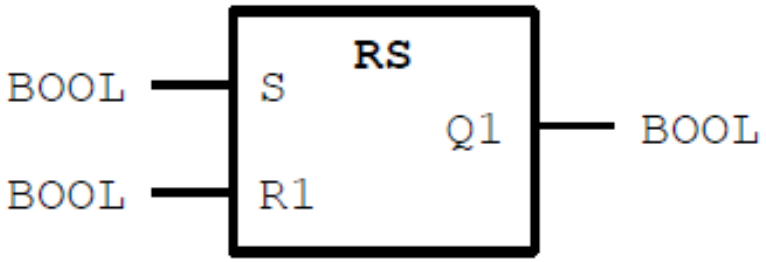
```
FUNCTION_BLOCK esempio
        VAR_INPUT
                (* definizione delle variabili di ingresso *)
                ....;
        END_VAR
        VAR_OUTPUT
                (* definizione delle variabili di uscita *)
                ....;
        END_VAR

                ...
                (* altre definizioni di variabili *)
                ...
                (* corpo del blocco funzionale *)
END_FUNCTION_BLOCK
```

- La **definizione grafica** si ottiene ancora con le parole chiave `FUNCTION_BLOCK … END_FUNCTION_BLOCK` ma ricorrendo a un blocco rettangolare

  – Esempio: flip-flop SR e RS





- Il corpo del blocco può essere scritto in uno qualunque dei linguaggi dello standard (incluso SFC)

POU
Blocchi funzionali

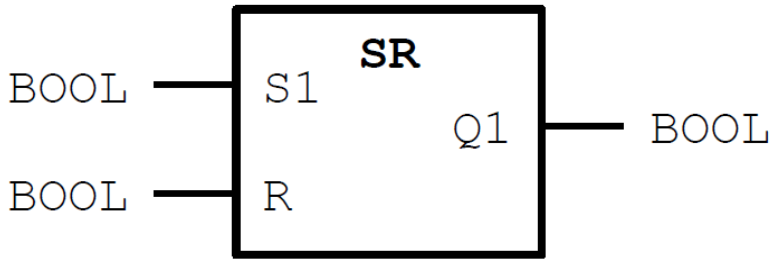
- All'atto della definizione di un blocco viene creata una sua **istanza**, in modo analogo a una variabile
  -
```
VAR
  istanza_1, istanza_1 : blocco_esempio
END_VAR
```
- L'utilizzatore può accedere solo alle variabili di **ingresso/uscita;** quelle interne non sono accessibili
- La definizione di un'istanza corrisponde a quella di una struttura dati
- Un'istanza può avere gli attributi `RETAIN` e `GLOBAL`
- Un'istanza può essere data in ingresso a un'altra POU come una qualunque variabile

POU
Blocchi funzionali

- Lo standard prevede un certo numero di **blocchi predefiniti:**
  - flip-flop
  - Contatori
  - Temporizzatori
  - Rilevatori di fronte
  - …
- Vediamo alcuni esempi

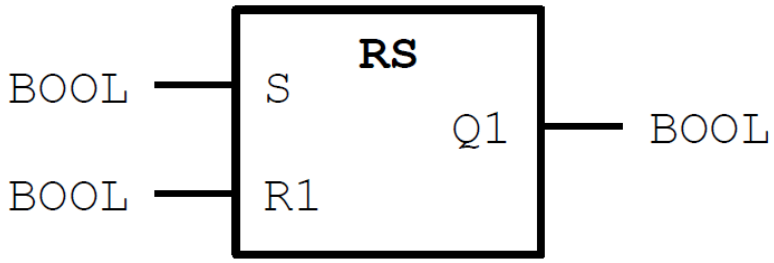
Esempio #1: flip-flop SR

```
FUNCTION_BLOCK SR (* priorità sul SET *)
    VAR_INPUT
        S1,R : BOOL;
    END_VAR
    VAR_OUTPUT
        Q1 : BOOL;
    END_VAR
    Q1 = S1 OR (Q1 AND NOT R)
END_FUNCTION_BLOCK
```



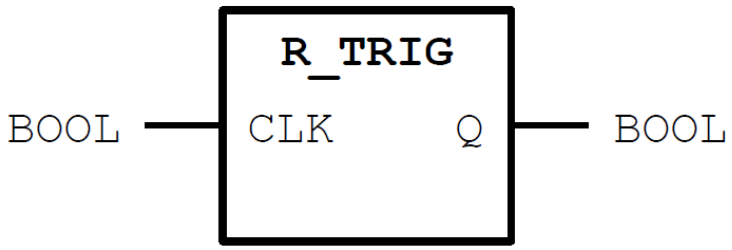
Esempio #2: flip-flop RS

```
FUNCTION_BLOCK RS (* priorità sul RESET *)
    VAR_INPUT
        S,R1 : BOOL;
    END_VAR
    VAR_OUTPUT
        Q1 : BOOL;
    END_VAR
    Q1 = NOT R1 AND (Q1 OR S)
END_FUNCTION_BLOCK
```



Esempio #3: rilevatore di fronte di salita

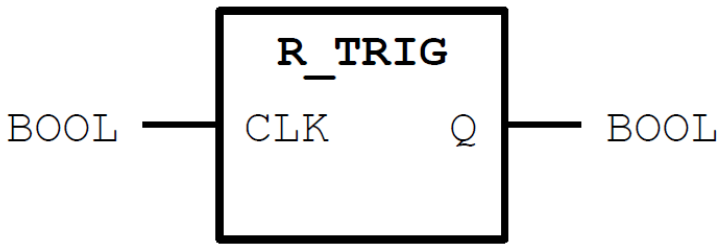
```
FUNCTION_BLOCK R_TRIG
        VAR_INPUT
                CLK : BOOL;
        END_VAR
        VAR_OUTPUT
                Q : BOOL;
        END_VAR
        VAR_RETAIN
                AUX : BOOL := 0;
        END_VAR
        LD CLK  (* corpo in IL *)
        ANDN AUX
        ST Q
        LD CLK
        ST AUX
END_FUNCTION_BLOCK
```

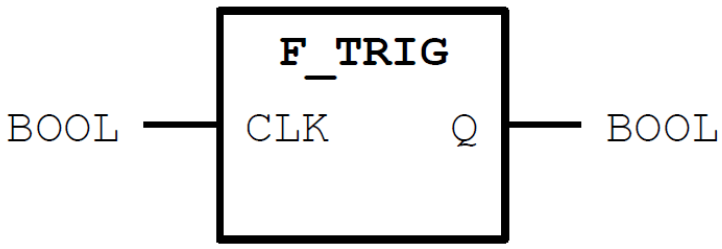


NOTE:

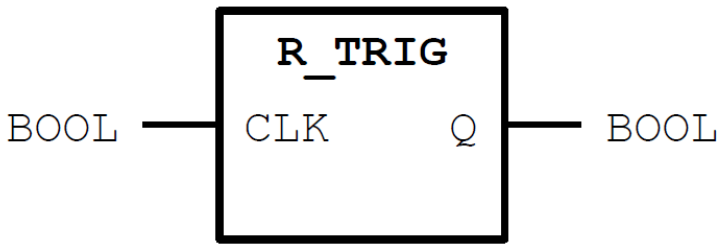
- alla prima chiamata del blocco si può avere il riconoscimento di un fronte se la variabile di ingresso è alta o bassa, rispettivamente
- il valore della variabile ausiliaria è mantenuto anche in assenza di alimentazione (`RETAIN`)

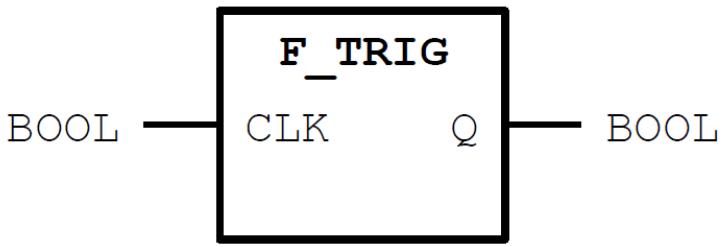
- Definizione grafica dei blocchi funzionali `R_TRIG` e `F_TRIG`





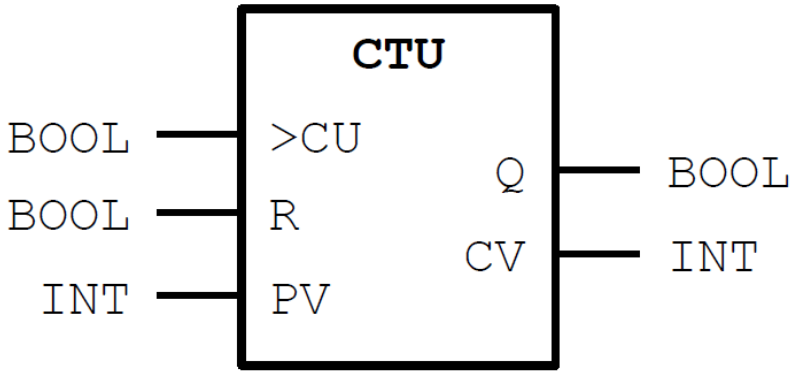
- I riconoscitori di fronte possono essere **utilizzati implicitamente** sulle variabili di ingresso di altri blocchi funzionali:
  - Le variabili vanno definite con il qualificatore `R_TRIG` e `F_TRIG`
  - Per la definizione grafica sulle linee di ingresso si possono usare i **simboli grafici** > e <



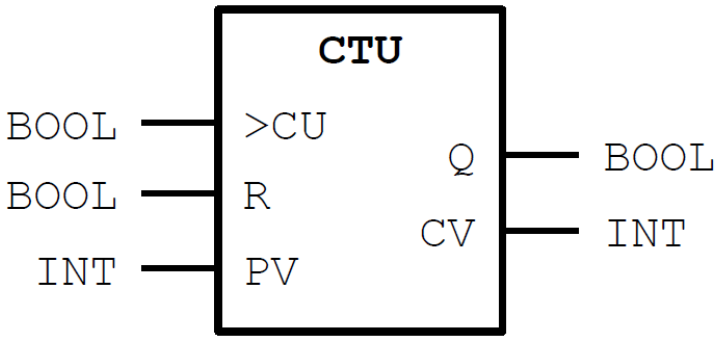


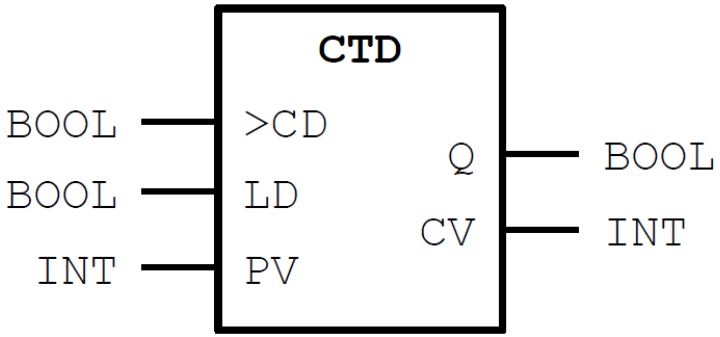
Esempio #4: contatore a incremento

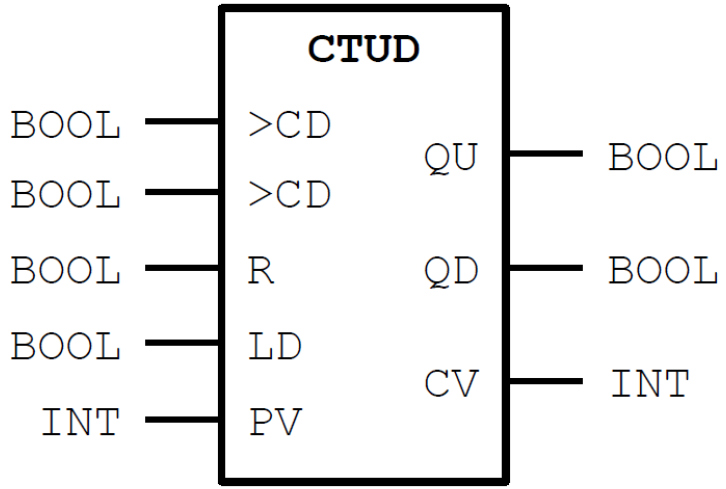
```
FUNCTION_BLOCK CTU
        VAR_INPUT
                CU : BOOL R_TRIG; (* def. implicita fronte *)
                R : BOOL; (* segnale di reset *)
                PV : INT; (* valore massimo di conteggio *)
        END_VAR
        VAR_OUTPUT
                Q : BOOL;
                CV : INT;
        END_VAR
        VAR_RETAIN
                AUX : BOOL := 0;
        END_VAR
        IF R THEN
                CV := 0;
        ELSEIF CU AND (CV<PV) THEN
                CV : = CV + 1;
        ENDIF
        Q = (CV = PV);
END_FUNCTION_BLOCK
```



Esistono contatori **a incremento** (CTU), **a decremento** (CTD) o **bidirezionali** (CTUD)







- `CU, CD` ingressi sui cui fronti di salita si incrementa/decrementa il contatore
- `R` è il reset (porta a zero il valore di conteggio)
- `PV` è il valore di conteggio di ingresso (*preset*)
- `LD` carica il contatore con il valore `PV`
- `Q` *end of count*
  - `CU = PV` (incremento)
  - `CD = 0` (decremento)
- `QD` segnala che il contatore bidirezionale ha raggiunto lo zero
- `QU` segnala che il contatore bidirezionale ha raggiunto il valore `PV`
- `CV` è il valore raggiunto dal contatore

- Per il contatore andrà previsto un valore di conteggio massimo `PV`
- Questo limite può essere aggirato usando più contatori in cascata

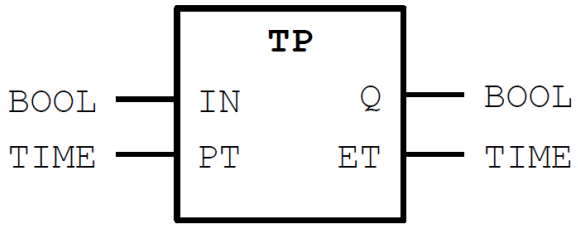
Esempio #5: timer

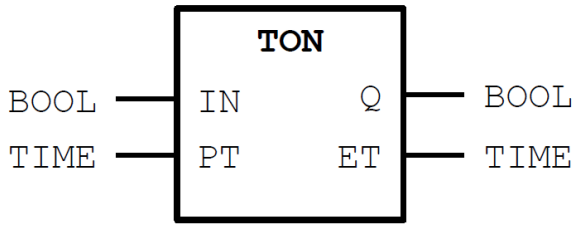
- `IN` segnale di ingresso
- `Q` uscita su cui agisce il timer
- `ET` tempo trascorso
- `PT` tempo massimo

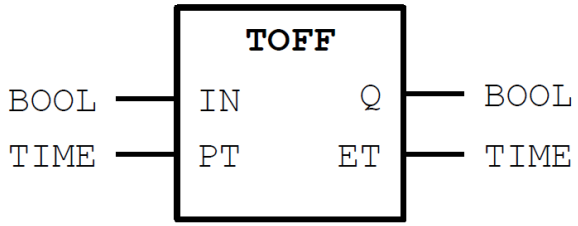
- Il timer genera una finestra rettangolare di durata prestabilita quando l'ingresso IN diventa vero
- *On delay*: l'uscita diventa vera dopo un certo tempo
- *Off delay*: l'uscita diventa falsa dopo un certo intervallo dallo spegnimento di IN

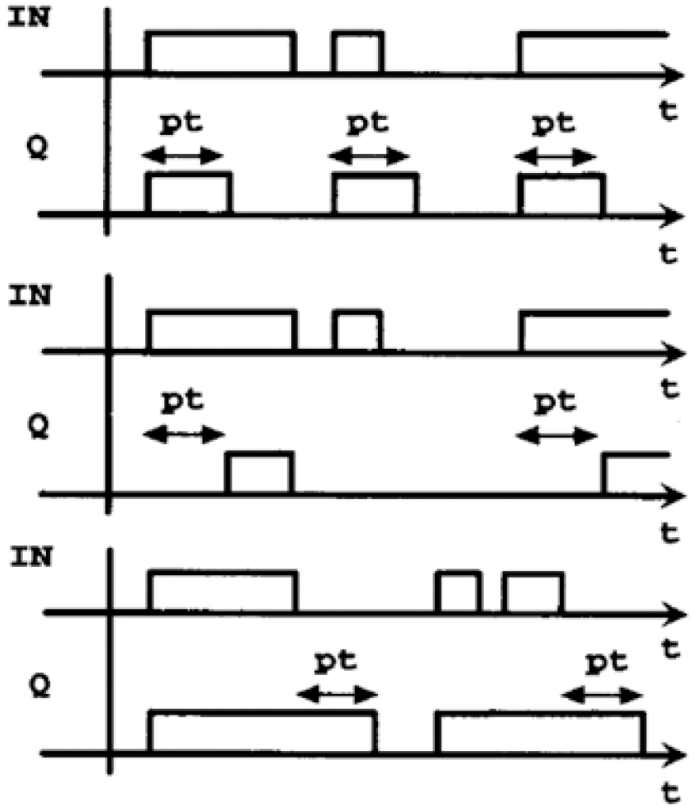
POU
Blocchi funzionali

- A impulsi
- *On delay*
- *Off delay*









- Anche in questo caso è possibile aggirare il limite massimo utilizzando
  - Più timer in cascata
  - Contatore in cascata al timer

Esempio #6: blocco funzionale definito dall'utente per la gestione di una risorsa comune (semaforo)

```
FUNCTION_BLOCK semaforo
     VAR_INPUT
          richiesta, rilascio : BOOL;
     END_VAR
     VAR_OUTPUT
          impegnata : BOOL;
     END_VAR
     VAR
          aux : BOOL := 0;
     END_VAR
```

Esempio #6: blocco funzionale definito dall'utente per la gestione di una risorsa comune (semaforo)

```
        impegnata := aux;
        IF richiesta THEN
             aux := 1;
        ELSEIF rilascio THEN
             impegnata := 0;
             aux := 0;
        END_IF
END_FUNCTION_BLOCK
```

Esempio #6: blocco funzionale definito dall'utente per la gestione di una risorsa comune (semaforo)

Definizione grafica →

```
FUNCTION_BLOCK
```


```
VAR
   aux : BOOL := 0;
END_VAR

 (* corpo in uno dei linguaggi *)

END_FUNCTION_BLOCK
```

Esempio #7: blocco funzionale definito dall'utente per la generazione di un'**onda quadra** su una variabile booleana quando un'altra variabile booleana è vera

```
FUNCTION_BLOCK ondaquadra
        VAR_INPUT
                abilita : BOOL;
                durata_ON, durata_OFF : TIME;
        END_VAR
        VAR_OUTPUT
                uscita : BOOL;
        END_VAR
        VAR                         (*sono usate due istanze di *)
                timer1, timer2 := TON;      (* timer on-delay *)
        END_VAR

        …
        (*corpo del blocco funzionale *)
        …
END_FUNCTION_BLOCK
```

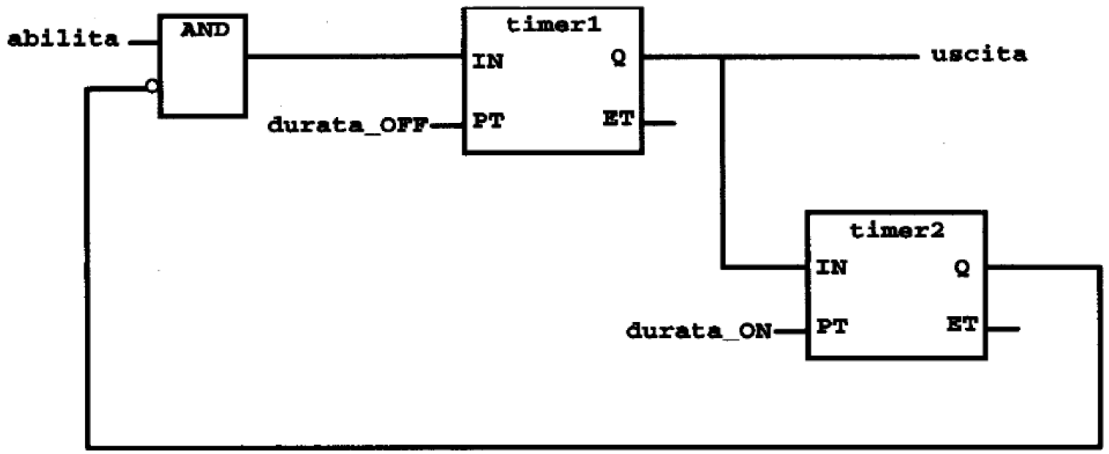
Esempio #7: blocco funzionale definito dall'utente per la generazione di un'**onda quadra** su una variabile booleana quando un'altra variabile booleana è vera

1. Corpo in ST

```
timer1(IN := NOT(timer2.Q) AND abilita, PT := durata_OFF);
timer2(IN := timer1.Q, PT := durata_ON);
uscita := timer1.Q;
```

NOTA: contiene chiamate a blocchi funzionali!

2. Corpo in FB

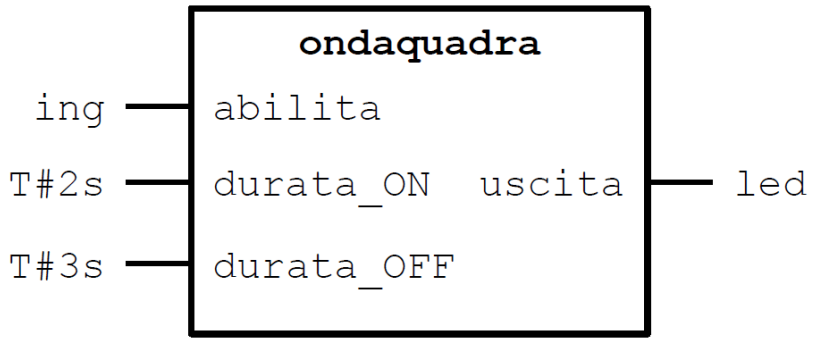


Esempio #7: blocco funzionale definito dall'utente per la generazione di un'**onda quadra** su una variabile booleana quando un'altra variabile booleana è vera

- Chiamata del blocco `ondaquadra` in testo strutturato:

```
ondaquadra(abilita := ing, durata_ON := T#2s,
           durata_OFF := T#3s, uscita := led);
```

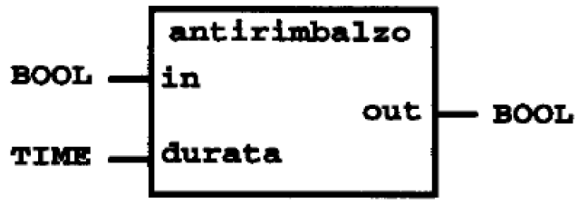
- Chiamata in forma grafica



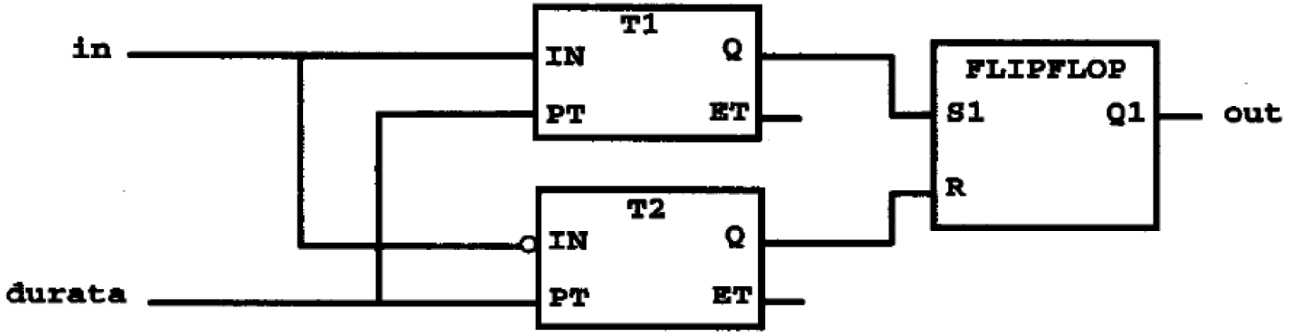
Esempio #8: antirimbalzo

`out` diventa vera solo se `in` resta vera per la durata specificata

```
FUNCTION_BLOCK
```




```
VAR
   T1,T2: TON; FLIPFLOP:SR;
END_VAR
```




```
END_FUNCTION_BLOCK
```

- **Programmi**
  - Rappresentano l'insieme logico di elementi e costrutti necessari per il controllo di una macchina o processo
  - Sono le uniche POU che possono accedere alle variabili rappresentative degli ingressi e delle uscite **fisiche** del dispositivo

POU
Programmi

- Un **programma** può essere visto come un grande blocco funzionale con alcune caratteristiche particolari:
  - Può avere definizioni di variabili direttamente rappresentate in memoria
  - Può avere definizioni di variabili globali
  - Può definire variabili indirizzabili da remoto
  - Non può contenere un'istanza di se stesso
  - Istanze di programmi possono essere dichiarate solo a livello risorsa

POU
Programmi

- Definizione

```
PROGRAM nome
    VAR_INPUT
          ...
    END_VAR
    VAR_OUTPUT
          ...
    END_VAR
...
(* altre variabili e corpo *)
END_PROGRAM
```

- Il corpo può essere scritto in un qualunque linguaggio previsto dalllo standard

POU
Programmi

- I programmi **sono le uniche POU che possono accedere alle variabili rappresentative degli ingressi e dalle uscite fisiche** del dispositivo!

- Potranno poi passare le variabili rappresentative ai blocchi può interni

Unità di Organizzazione della Programmazione

In aggiunta, si individuano

- **Compiti** o **task**: elementi che **controllano l'esecuzione di programmi e blocchi funzionali** su base periodica, ciclica o al verificarsi di eventi
- **Risorse**: corrispondono a entità in grado di **eseguire programmi**
- **Configurazione**: elemento del linguaggio che corrisponde a un dispositivo che **comprende una o più risorse**

Compiti o task

- Per definire un compito si possono utilizzare:
  - Il parametro `SINGLE` di tipo `BOOL`
    - Indica la variabile booleana il cui fronte di salita rappresenta il verificarsi dell'evento che causa un'**unica esecuzione** del programma o del blocco funzionale
  - Il parametro `INTERVAL` di tipo `TIME`
    - Indica la durata temporale del ciclo per un compito di tipo **periodico**
  - In caso di mancato uso di uno dei due parametri precedenti si definisce un compito **ciclico continuo**

Compiti o task

- Per definire un compito si possono utilizzare:
  - Il parametro `PRIORITY` di tipo `UINT`
    - Indica la priorità del compito utilizzata dallo schedulatore del dispositivo per gestire i vari compiti
    - **Il valore 0 indica la priorità massima**
    - Se non viene definito la priorità è quella minima
  - L'algoritmo di schedulazione dipende dal particolare dispositivo
    - Può essere *pre-emptive* (task a priorità minore possono essere interrotti da quelli a priorità maggiore) oppure *non pre-emptive*
    - Normalmente per i dispositivi di controllo è richiesta la disponibilità di uno schedulatore pre-emptive

- Esempi
  - `TASK ciclo_controllo(INTERVAL:=T#20ms, PRIORITY:=0);`
  - `TASK allarme(SINGLE:=flag_allarme, PRIORITY:=2)`
  - `TASK background`
  - `TASK ciclo_refresh(INTERVAL:=T#10s, PRIORITY:=5);`

- **Cosa fanno questi task?**

- Esempi
  - `ciclo_controllo` definisce un compito periodico da eseguire ogni 20 ms a priorità massima
  - `allarme` definisce un compito che va eseguito solo una volta quando la variabile booleana flag_allarme passa dal valore falso a quello vero
  - `background` definisce un compito ciclico alla minima priorità
  - `ciclo_refresh` definisce un compito periodico da eseguire ogni 10 ms

POU
Task

- Per assegnare un programma o un blocco funzionale a un particolare compito (task) si usa la parola chiave `WITH`
  - In maniera grafica, l'assegnazione al compito si ottiene scrivendo il nome del compito (task) all'interno del blocco
  - I programmi e i blocchi funzionali non esplicitamente associati ad un compito (task) vengono eseguiti in maniera ciclica continua con la minima priorità

Risorse

- **Definizione**: inizia con la parola chiave `RESOURCE`, seguita dal nome identificativo, dalla parola chiave `ON` e dal tipo di processore su cui dovrà essere caricata, e termina con la parola chiave `END_RESOURCE`
  - Vanno definite eventuali variabili globali, variabili ad accesso remoto, compiti e programmi che la compongono
  - Nel definire i programmi si assegnano anche le loro variabili di ingresso e di uscita, che vanno collegate agli indirizzi di memoria corrispondenti a I/O fisici

POU
Risorse

- Si noti che la definizione di una risorsa non prevede la scrittura di istruzioni, ma solo la definizione di **relazioni** tra elementi
- A questo livello viene creata un'**istanza** dei programmi, come per i blocchi funzionali
- L'istanza è collegata a un *task* e alle variabili su cui deve operare

Configurazione

- Corrisponde alla definizione di tutto il SW che deve essere caricato in un dispositivo di controllo
- Contiene **risorse, variabili globali e percorsi di accesso**

POU
Configurazione

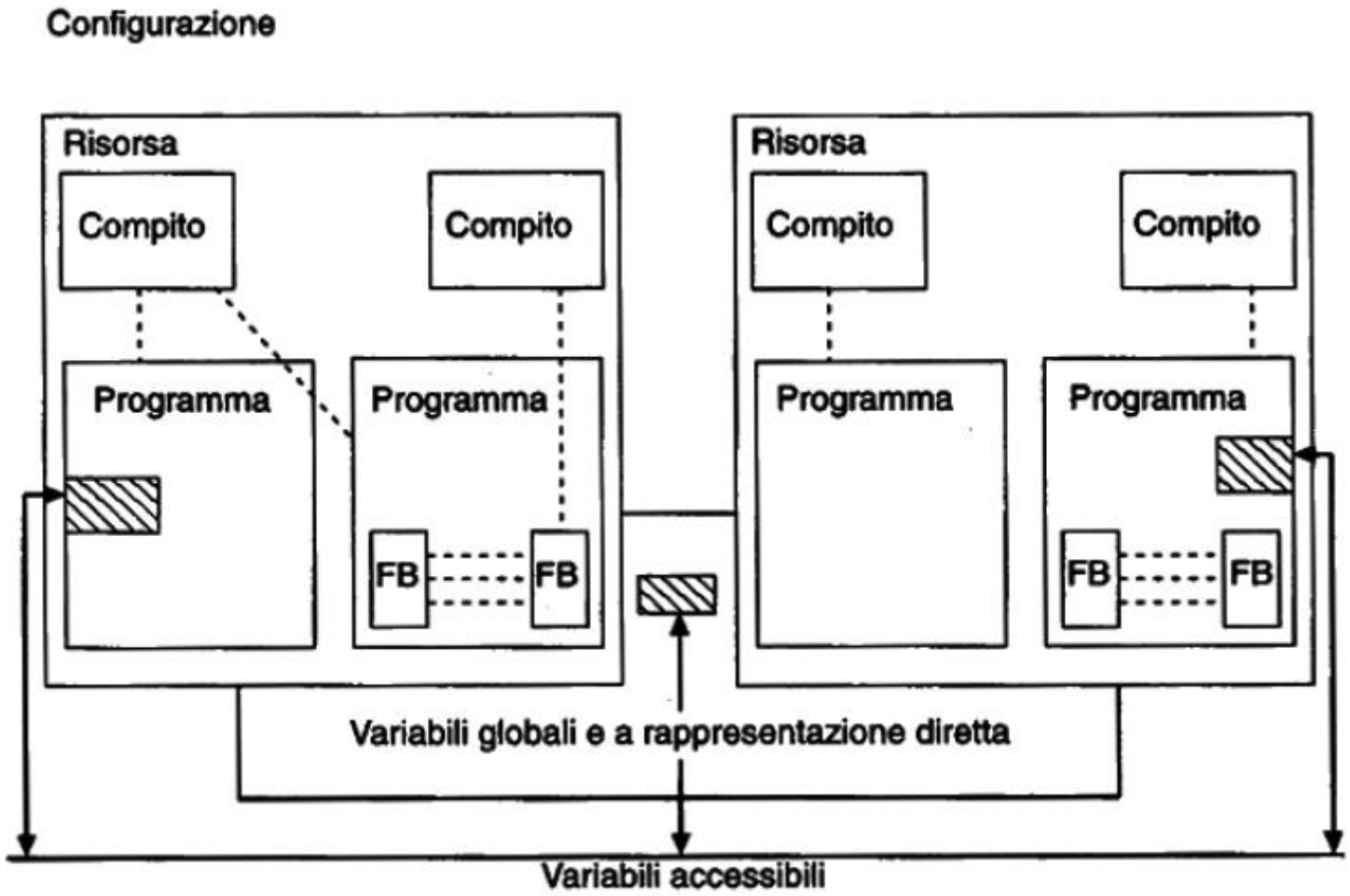
- Lo scopo della configurazione è
  - Definire il tipo di **risorse elaborative** a disposizione
  - **Allocare i programmi** sulle risorse
  - Indicare la **modalità** e la **priorità** di esecuzione dei programmi
  - Definire le **variabili globali** accessibili da tutti i programmi

POU
Configurazione

```
CONFIGURATION controllore_cella
        VAR_GLOBAL
                A : REAL;          (*queste variabili sono visibili in *)
                B : REAL;          (*tutta la configurazione           *)
        END_VAR
        RESOURCE processore_1 ON Pentium
                VAR_GLOBAL
                flag_allarme : BOOL;      (*visibile in tutta la *)
                                          (*risorsa              *)
                END_VAR
                TASK ciclo(INTERVAL:=T#20ms,PRIORITY:=0);
                TASK allarme(SINGLE:=flag_allarme,PRIORITY:=2);
                PROGRAM tornio : controllo_macchina(IN1:=…,IN2:=…,…,
                   OUT1:=…,OUT2:=…,…) WITH ciclo;
                PROGRAM all_tornio : gestione_allarmi(IN1:=…,IN2:=…,
                   …,OUT1:=…,OUT2:=…,…) WITH allarme;
        END_RESOURCE

        RESOURCE processore_2 ON AB486
                PROGRAM cella : coordinamento(IN1:=…,IN2:=…,…,
                   OUT1:=…,OUT2:=…,…);
        END_RESOURCE
END_CONFIGURATION
```

Unità di Organizzazione della Programmazione



Unità di Organizzazione della Programmazione

NOTE:

- È possibile eseguire un **blocco funzionale** con criteri specificati da un task diverso dal quello del programma che contiene il blocco
- È possibile definire variabili accessibili da parte di programmi remoti

Linguaggi di programmazione

- Un PLC può essere programmato utilizzando diversi linguaggi, tutti normati dallo standard **IEC 61131-3**

- La normativa comprende **3 linguaggi grafici** e **2 testuali**

- Lo standard è un **modello di riferimento non vincolante:** non tutti i PLC lo rispettano! (soprattutto PLC vecchi e/o precedenti allo standard)

Instruction List (IL)

- Linguaggio di basso livello simile all'**assembly**
- Sequenze di istruzioni, solitamente composte da un operatore che agisce su un unico operando; possono avere un **modificatore** e un'**etichetta**
- Gli operatori fanno riferimento a un **accumulatore** oltre all'operando indicato

Instruction List (IL)

- Operatori
  - `LD` load (operando -> accumulatore)
  - `ST` store (accumulatore -> operando)
  - `S` set (mette una variabile logica a 1)
  - `R` reset (mette una variabile logica a 0)
  - `AND/&, OR, XOR` (op. logiche)
  - `ADD,SUB,MUL,DIV` (op. aritmetiche)
  - `GT,GE,EQ,NE,LE,LT` (comparazione)
  - `JMP` (salta all'etichetta indicata dall'operando)
  - `CAL` (chiamata a blocco funzionale)
  - `RET` (ritorno da una funzione o un blocco funzionale)

Instruction List (IL)

- Modificatori
  - `N` negazione booleana dell'operando
  - `(` la valutazione dell'operatore va effettuata quando si trova la `)`
  - `C` l'esecuzione è condizionata al fatto che il valore dell'accumulatore sia 1 (o 0 se si usa anche `N`); utilizzabile con `JMP, CAL, RET`

Instruction List (IL)

- Esempio:
  flip-flop a commutazione su fronte di salita

```
        LD    i                 (* Queste istruzioni valutano *)
        ANDN  i_precedente      (*  il fronte di salita      *)
        JMPCN ava   (* Salta se non c'è il fronte di salita *)
        LD    u     (* Queste istruzioni assegnano alla  *)
        STN   u     (*  variabile u il suo valore negato *)
ava:    LD    i
        ST    i_precedente
```

Structured Text (ST)

- linguaggio di alto livello simile al **Pascal**

Structured Text (ST)

- Assegnazione `:=`
- Terminatore `;`
- Operatori aritmetici
  - `+`
  - `-`
  - `*`
  - `/`
  - `MOD` (resto di una divisione intera)
  - `**` (elevamento a potenza)

Structured Text (ST)

- Operatori logici
  - `AND` o `&`
  - `OR`
  - `XOR`
  - `NOT`
- Operatori relazionali
  - `<`
  - `>`
  - `<=`
  - `>=`
  - `<>` (diverso)

Structured Text (ST)

- Istruzioni condizionate
  - `IF … THEN …(ELSEIF … THEN) … (ELSE) … END_IF`
  - `CASE … OF … ELSE … END_CASE`
    (decisioni multiple che dipendono da una variabile intera)
- Cicli
  - `FOR … TO … BY … DO … … END_FOR`
  - `WHILE … DO … … END_WHILE`
  - `REPEAT … … UNTIL … END_REPEAT`
  - `EXIT` (uscita immediata da un ciclo)
- Gestione del flusso di istruzioni
  - Costrutti `EXIT, RETURN` (uscita da funzioni e blocchi funzionali)

Structured Text (ST)

- Esempi

```
IF abilitazione & (conteggio<100) THEN
  conteggio := 100;
  stato_tornio := fermo;
END_IF
```

```
IF abilitazione THEN
  termocoppia_1.valore := termocoppia_2.valore;
ELSE
  termocoppia_1.valore := 0.0;
END_IF
```

Structured Text (ST)

- Esempi

```
IF conteggio<100 THEN
  A := 1.0;
ELSEIF conteggio<500 THEN
  A := 2.0;
ELSEIF conteggio<800 THEN
  A := 3.0;
ELSE (* conteggio maggiore o uguale a 800 *)
  A := 4.0;
END_IF
```

Structured Text (ST)

- Esempi

```
CASE conteggio OF
  1 : stato_tornio := fermo;
  2, 3, 4, 5 : stato_tornio := funzionante;
  6..50 : stato_tornio := attesa;
ELSE
  stato_tornio := guasto;
END_CASE
```

Linguaggi di programmazione

Linguaggi testuali

- **NOTA:** i linguaggi testuali possono variare tra i diversi produttori di PLC (e a volte anche tra prodotti diversi dello stesso marchio)

- Ad esempio vedremo che la dichiarazione di variabili e POU è leggermente diversa in openPLC

- Nelle prossime lezioni ci concentreremo soprattutto sui linguaggi di programmazione **grafici** (in particolare Ladder e SFC)

Function Block Diagram (FBD)

- Prevede la costruzione di **reti di componenti**, in analogia a un circuito elettronico
- Ciascun blocco è una *black-box* che **processa gli ingressi e restituisce delle uscite**
- I singoli blocchi possono essere realizzati in **linguaggi differenti**

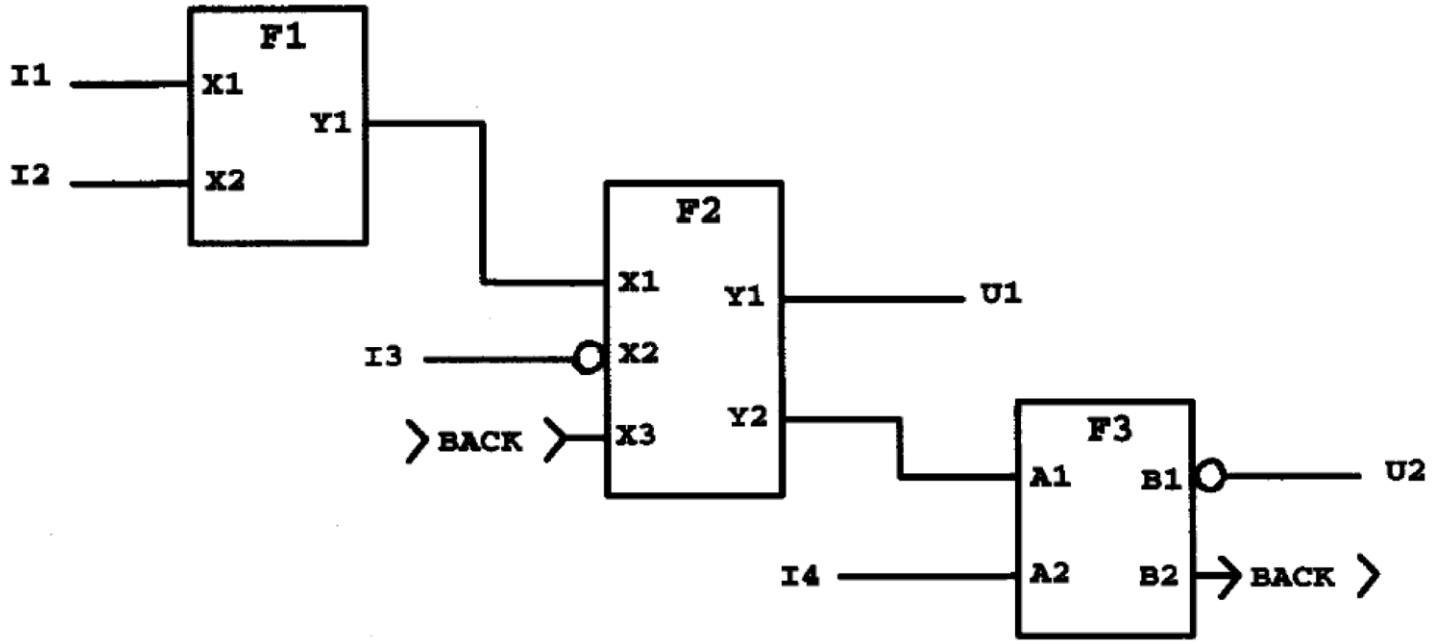
Function Block Diagram (FBD)

- I segnali viaggiano lungo le **connessioni** tra questi blocchi (da destra a sinistra)

- È possibile realizzare **anelli chiusi:** l'uscita viene conservata e usata come valore di ingresso alla successiva valutazione della rete

Function Block Diagram (FBD)

- Esempio

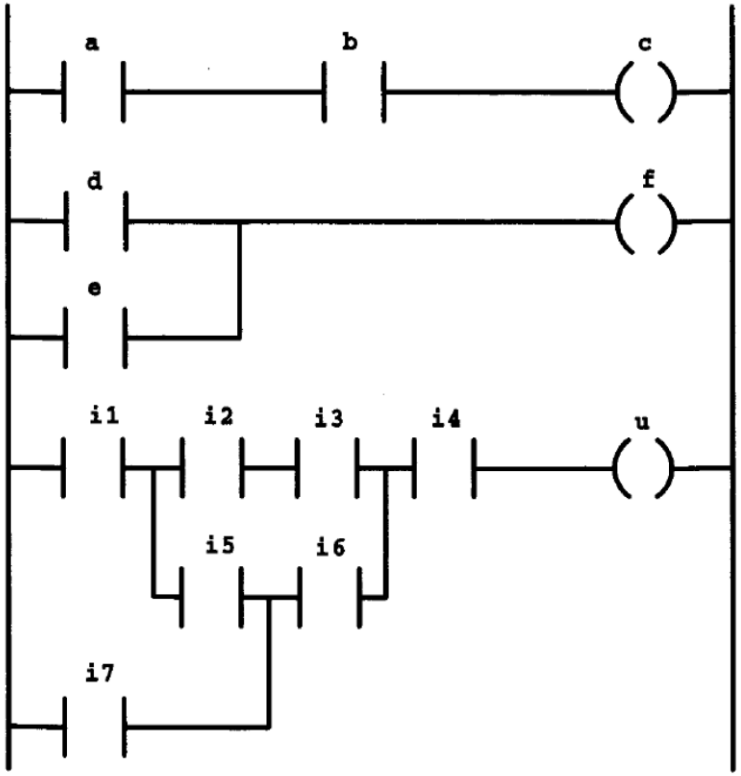


Ladder Diagram (LD)

- Nasce come “imitazione” dei **quadri a relé**
- L’idea era quella di rendere la **transizione** dai sistemi di controllo cablati a quelli programmabili il più **naturale** possibile
- Il nome *ladder* significa “**scala**” (con riferimento all’aspetto del listato; ogni linea è detta *rung*)

Ladder Diagram (LD)

- Esempio



Ladder Diagram (LD)

- I principali componenti sono
  - **Contatti:** ingressi esterni o condizioni interne del dispositivo
    (rappresentati da bit di memoria, ricorda la copia massiva…)
  - **Bobine:** uscite
    (bit di memoria che possono comandare uscite o variare condizioni interne)
- Il flusso di energia procede **da sinistra verso destra** e le istruzioni vengono eseguite **dall'alto verso il basso**

Sequential Function Chart (SFC)

- Linguaggio specifico per la scrittura di algoritmi per il controllo logico/sequenziale
- Nasce come risultato di un'apposita commissione costituita in Francia nel 1975
  - Scopo: descrivere sistemi complessi di automazione industriale **ad eventi discreti**
  - Proposta: **GRAFCET (GRAPHe de Coordination Etapes-Transitions,** grafo di coordinamento passi-transizioni**)**

Sequential Function Chart (SFC)

- Sono di fatto una versione semplificata delle **Reti di Petri**

→ La programmazione di un sistema di controllo per l'automazione industriale coincide con la descrizione del suo comportamento desiderato!

Sequential Function Chart (SFC)

- Il GRAFCET è stato incluso con il nome di **SFC** nello standard IEC 61131-3

- Elementi principali di un SFC:
  - Fasi
  - Transizioni
  - Archi orientati

Linguaggi di programmazione

Linguaggi grafici

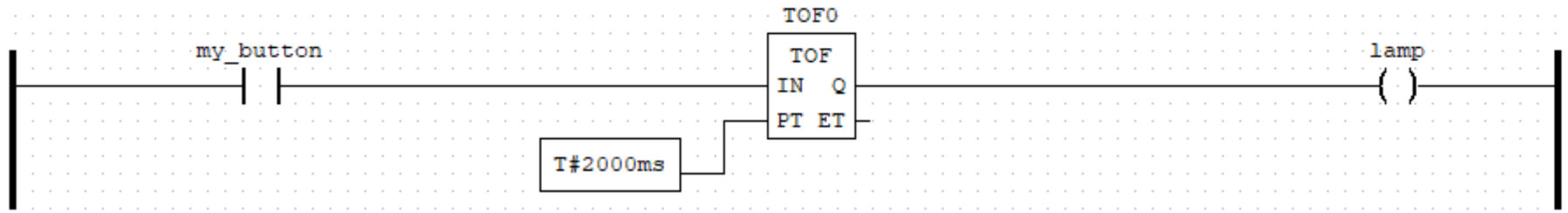
- Il **Ladder Diagram** e i **Sequential Functional Chart** saranno approfonditi nelle prossime lezioni

- **Le esercitazioni saranno realizzate prevalentemente in questi due linguaggi**

Esercizi

- Ricordate il progetto Hello World di OpenPLC? Ora dovreste essere in grado di capire cosa fa

- NB: TOF0 è un timer; i contatti sono **ingressi** e le bobine **uscite…**



Risorse e Riferimenti

- [1] Cap. 6
- [3] Cap. 2,3

- Programmazione PLC @RealPars

Fine Lezione #8

Programmazione dei PLC