



# Linguaggi di descrizione dell'hardware

**Richiami di VHDL**

# Preview

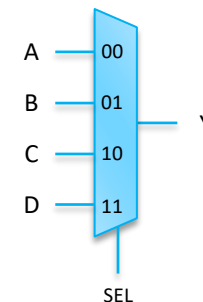
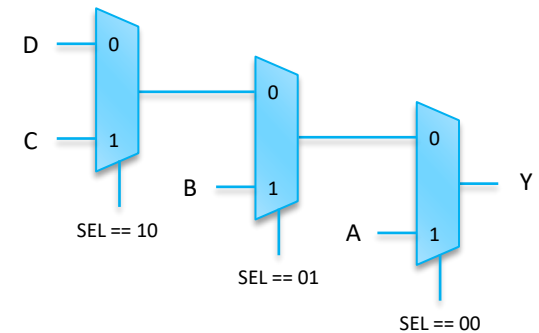
## ► Come funziona?

- Scriviamo del software che viene poi trasformato in un circuito
- Cerchiamo di usare concetti ad alto livello

## ► Esempi

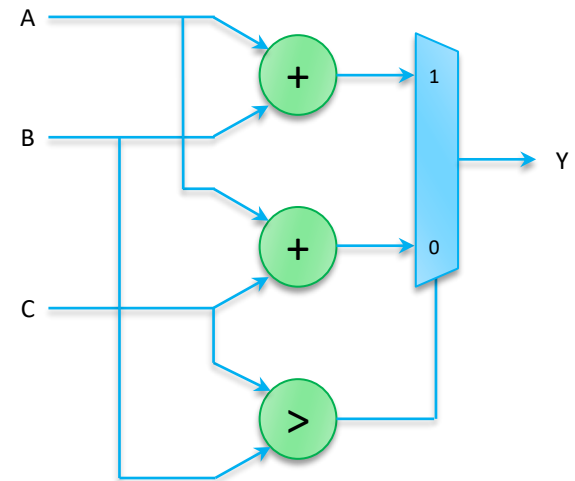
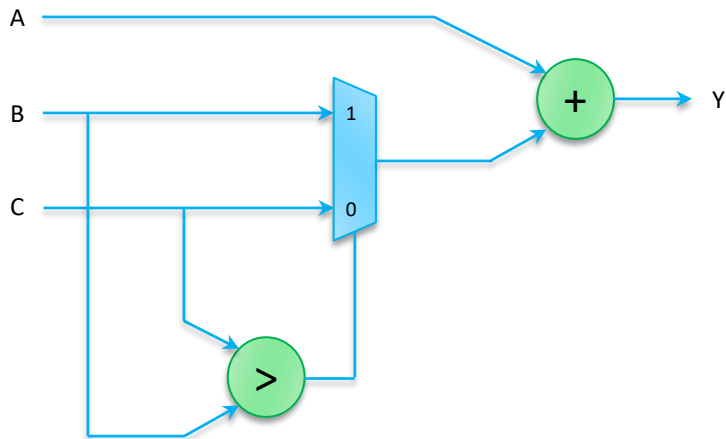
```
if      (sel == 00) then Y = A;  
elseif (sel == 01) then Y = B;  
elseif (sel == 10) then Y = C;  
else      Y = D;
```

```
case (sel) {  
  00: Y = A; break;  
  01: Y = B; break;  
  10: Y = C; break;  
  11: Y = D; break;  
}
```



# Preview, operazione complessa

```
if (B > C) then  
    Y = A + B;  
else  
    Y = A + C;
```



## II VHDL

**Lo prendiamo come esempio**

# Il linguaggio VHDL

---

- ▶ **Very High Speed Integrated Circuit Hardware Description Language**
  - ▶ VHSIC HDL = VHDL
- ▶ **Scopi**
  - ▶ Documentazione, simulazione, sintesi
- ▶ **Metodologie**
  - ▶ Top-down – procede da un livello alto ad uno basso scomponendo un sistema in sottosistemi
  - ▶ Bottom-up – ottiene sistemi complessi assemblando sistemi più semplici
  - ▶ Meet-in-the-middle – decompone un sistema in sottosistemi, fino ad arrivare ad elementi contenuti in una libreria
- ▶ **Viste**
  - ▶ Data flow
  - ▶ Strutturale
  - ▶ Comportamentale

# Concetti base: entità

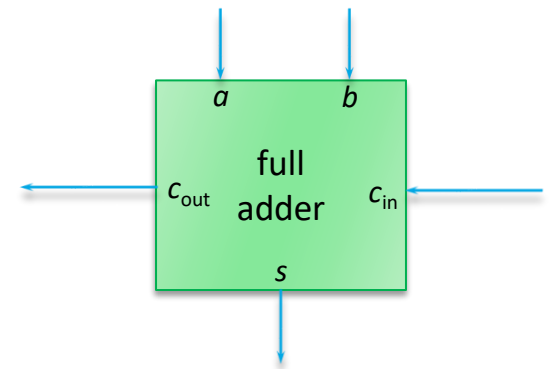
---

## ► L'entità e l'oggetto base del VHDL

- Corrisponde ad un blocco, un modulo, un elemento
- Simile ad una classe in C++, o una funzione in C
- E' identificata da un nome
- Dispone di un certo numero di ingressi ed uscite
- **Definisce l'interfaccia del blocco**

```
entity entity_name is  
  [port (interface-signal-declaration);]  
end [entity] [entity_name];
```

```
entity full_adder is  
  port (a, b, cin : in bit;  
        s, cout : out bit);  
end full_adder;
```

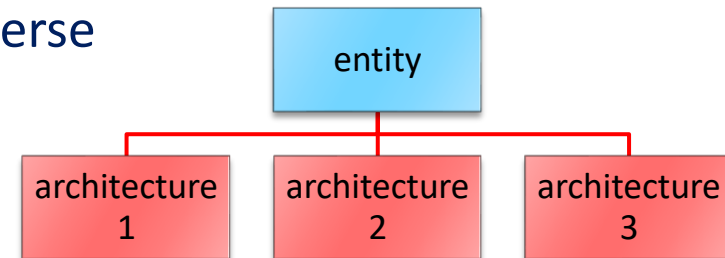


# Concetti base: architettura

---

## ► L'architettura definisce cosa fa un blocco

- Ogni architettura corrisponde ad una entità
- Ogni entità può avere più architetture diverse
- **Ne descrive il funzionamento** tramite equazioni, connessioni strutturali di altri blocchi (gerarchia), oppure tramite un algoritmo



```
architecture architecture_name of entity_name is  
    [declarations]  
begin  
    [architecture body]  
end [architecture] [architecture_name];
```

```
architecture componenti of full_adder is  
begin  
    blah blah blah  
end componenti;
```

# Stili di descrizione

---

## ▶ **Data flow o equazioni**

- ▶ Descrive il funzionamento tramite delle equazioni booleane
- ▶ Esprime la dipendenza delle uscite in funzione degli ingressi e di segnali interni
- ▶ Le varie equazioni sono messe a sistema

## ▶ **Strutturale**

- ▶ Descrive il funzionamento come collegamento di componenti in una gerarchia
- ▶ La nuova entità può essere usata a sua volta come componente in un'altra architettura

## ▶ **Behavioral o comportamentale**

- ▶ Descrive il funzionamento tramite un algoritmo
- ▶ Il codice descrive come derivare il valore dell'uscita in funzione di quello degli ingressi, dei segnali interni e di variabili



# Esempio di porta AND a 2 ingressi

## ► Interfaccia

- Ingressi *x* e *y*, uscita *z*

## ► Comportamento

- Usiamo l'operatore **and** già predefinito nel linguaggio
- L'operatore **<=** esegue l'assegnazione

Nomi dei segnali di interfaccia

**in** denota i segnali di ingresso

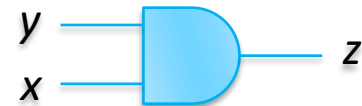
**out** denota i segnali di uscita

**bit** indica un tipo di segnali che può avere valori 0 e 1

Equazione che descrive il comportamento del componente

```
-- Porta AND a 2 ingressi
entity And2 is
  port (x, y : in bit; z : out bit);
end entity And2

architecture ex1 of And2 is
begin
  z <= x and y;
end architecture ex1;
```



# Testbench

- ▶ **Per poter eseguire una simulazione occorre fornire dei valori ai segnali di ingresso**
  - ▶ Il blocco che realizza questa funzione è solitamente chiamato testbench
  - ▶ E' possibile realizzarlo mischiando la rappresentazione data flow con quella strutturale
  - ▶ Il testbench chiude il sistema, quindi l'entità non ha ingressi e uscite

```
-- Porta AND a 2 ingressi
entity And2 is
  port (x, y : in bit; z : out bit);
end entity And2;
```

```
architecture ex1 of And2 is
begin
  z <= x and y;
end architecture ex1;
```

```
-- Testbench per porta And a 2 ingressi
```

```
entity TestAnd2 is
end entity TestAnd2;
```

```
architecture simple of TestAnd2 is
```

```
-- Segnali interni di interconnessione
```

```
signal a, b, c : bit;
```

```
begin
```

```
-- Istanza del modulo da testare
```

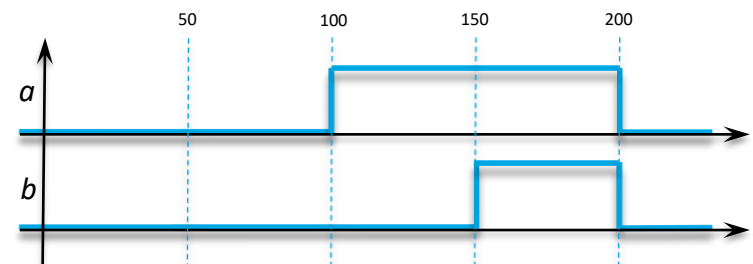
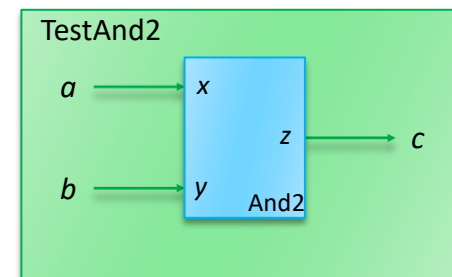
```
g1: And2 port map (x => a, y => b, z => c);
```

```
-- Definizione degli stimoli
```

```
a <= '0', '1' after 100 ns, '0' after 200 ns;
```

```
b <= '0', '1' after 150 ns, '0' after 200 ns;
```

```
end architecture simple;
```



# Operatori e identificatori

---

- ▶ Il VHDL dispone di tutti gli operatori dell'algebra booleana
  - ▶ not
  - ▶ and, or
  - ▶ nand, nor
  - ▶ xor, xnor
- ▶ **Precedenza degli operatori**
  - ▶ not ha la precedenza sugli altri
  - ▶ Gli altri hanno tutti la stessa precedenza, applicati da sinistra a destra
  - ▶ Quindi attenzione a mettere bene le parentesi!!
- ▶ **Identificatori**
  - ▶ Utilizzati per nomi di entità, architetture, segnali
  - ▶ Il VHDL è case insensitive, quindi x e X sono la stessa cosa!

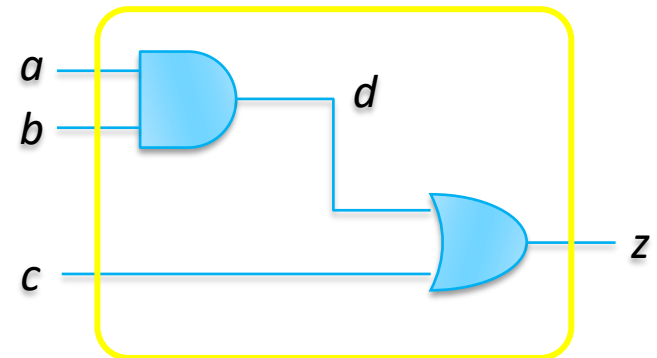
# Espressioni multiple

---

- ▶ Possiamo usare diverse espressioni legate da segnali interni
  - ▶ Dichiariamo il nome e il tipo dei segnali interni prima delle equazioni
  - ▶ L'ordine delle equazioni non ha importanza

```
-- Uso di espressioni multiple
entity comb_function is
  port (a, b, c : in bit; z : out bit);
end entity comb_function;

architecture expression of comb_function is
  signal d : bit;
begin
  d <= a and b;
  z <= d or c;
end architecture expression;
```

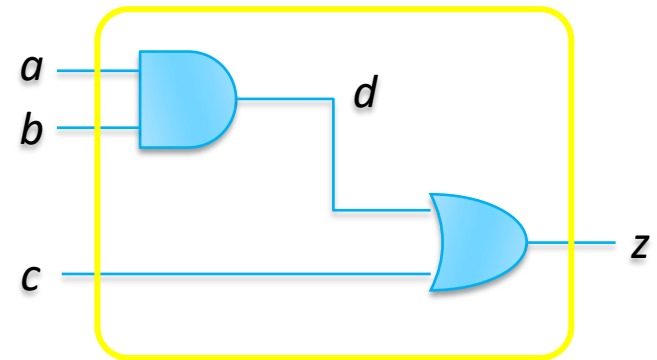


# Ritardi

- ▶ **Si può specificare il ritardo di una espressione**
  - ▶ Si usa l'operatore **after** seguito dall'indicazione del tempo
  - ▶ Il risultato non viene assegnato al segnale destinazione fin quando non è passato il tempo di simulazione
  - ▶ Se non viene specificato alcun ritardo si assume comunque un ritardo arbitrariamente piccolo indicato con  $\delta$

```
-- Uso di espressioni multiple
entity comb_function is
  port (a, b, c : in bit; z : out bit);
end entity comb_function;

architecture expression of comb_function is
  signal d : bit;
begin
  z <= d or c after 4 ns;
  d <= a and b after 5 ns;
end architecture expression;
```



# VHDL in pratica

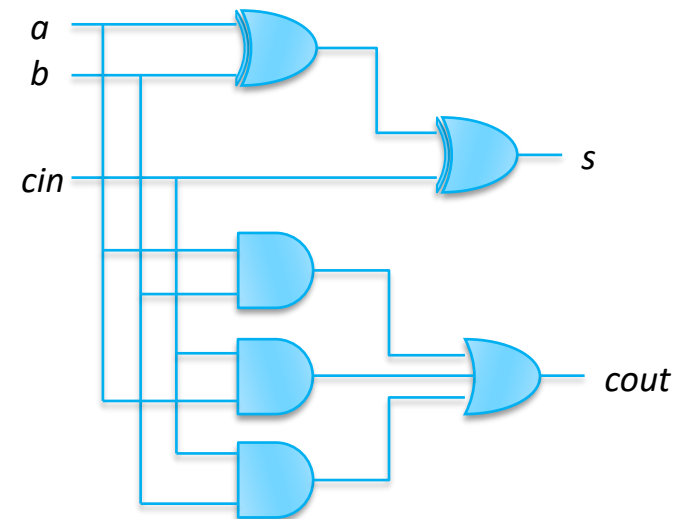
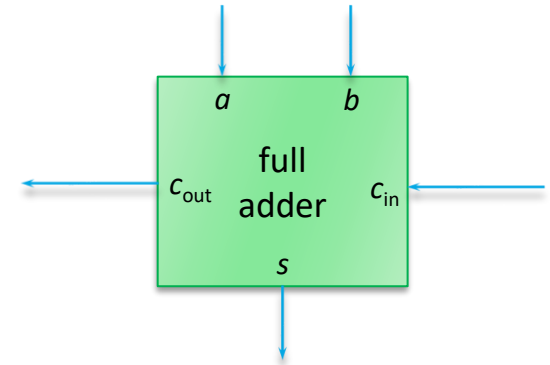
**Vista strutturale, sintassi, procedurale, etc.**

# II full adder

---

```
entity full_adder is  
  port (a, b, cin : in bit;  
         s, cout : out bit);  
end full_adder;
```

```
architecture equazioni of full_adder is  
begin  
  s <= a xor b xor cin;  
  cout <= (a and b) or (a and cin)  
         or (b and cin);  
end architecture equazioni;
```

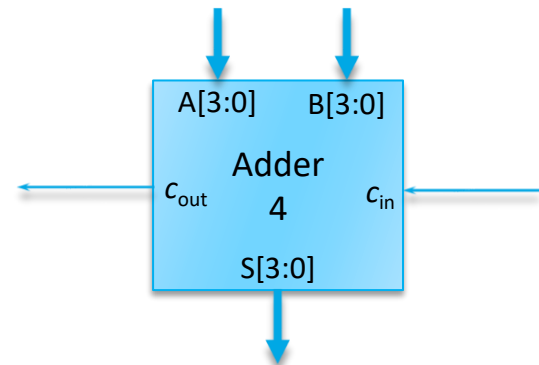


# Sommatore a 4 bit

---

- ▶ Dobbiamo poter definire un'interfaccia con segnali multibit
  - ▶ Il VHDL supporta la definizione di array di valori, usando il tipo `bit_vector`
  - ▶ Possiamo definire un range qualunque

```
entity Adder4 is
  port (A, B : in bit_vector(3 downto 0);
        cin : in bit;
        S : out bit_vector(3 downto 0);
        cout : out bit);
end Adder4;
```

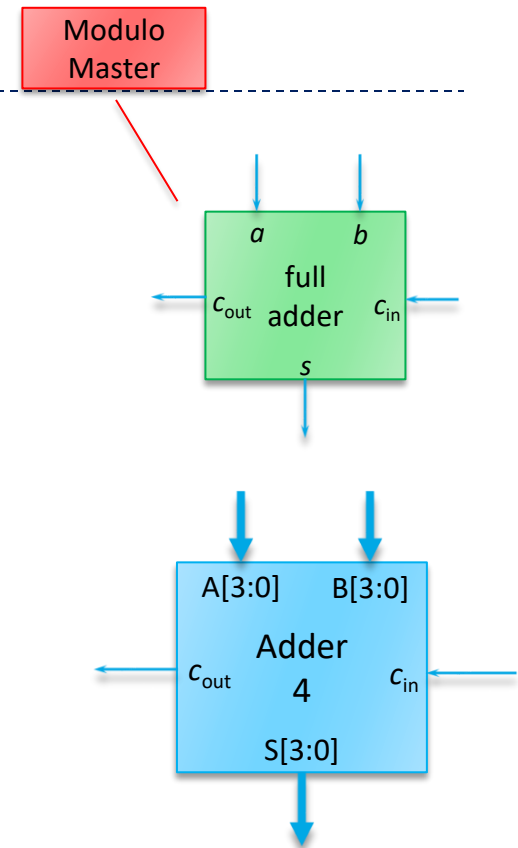
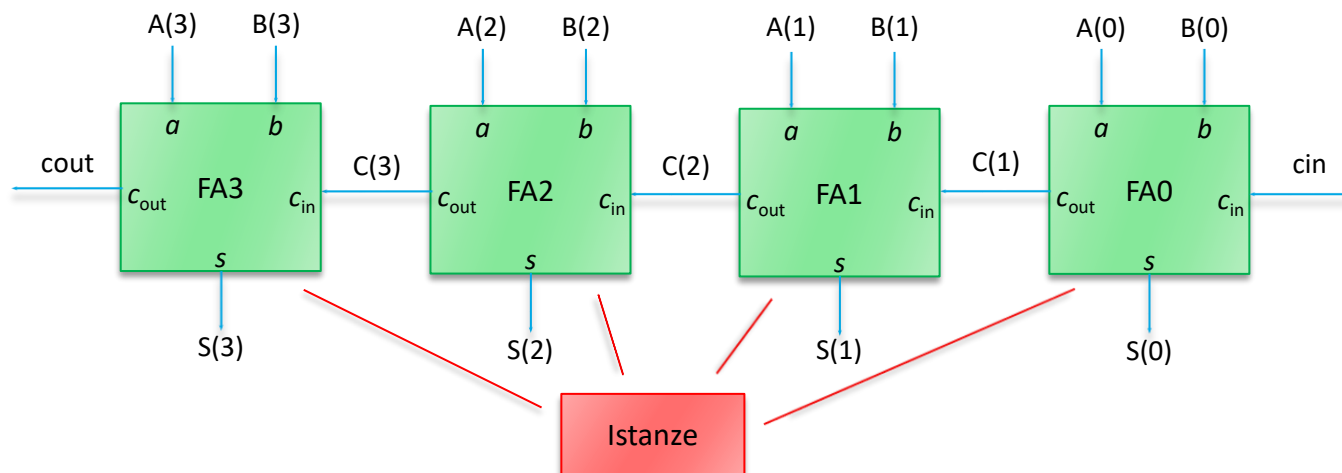




# Architettura strutturale

## ► Costruiamo l'addizionatore a 4 bit mettendo assieme 4 full adder

- Nell'architettura si specifica il modulo che si vuole istanziare e si dà un nome ad ogni istanza
  - Per esempio FA0, FA1, FA2 ed FA3
- Quindi si definisce una mappatura tra i segnali del modulo e quelli del composito
  - I segnali A(0) e B(0) andranno ad FA0
  - I segnali A(1) e B(1) andranno ad FA1, etc.



# Parametri formali ed effettivi

- ▶ **Ogni componente assegna un nome agli ingressi e uscite**

- ▶ In termini informatici, questo è chiamato *parametro formale*
- ▶ Tale nome ha significato all'interno del componente

```
entity full_adder is
  port (a, b, cin: in bit;
        cout, s: out bit);
end full_adder;
```

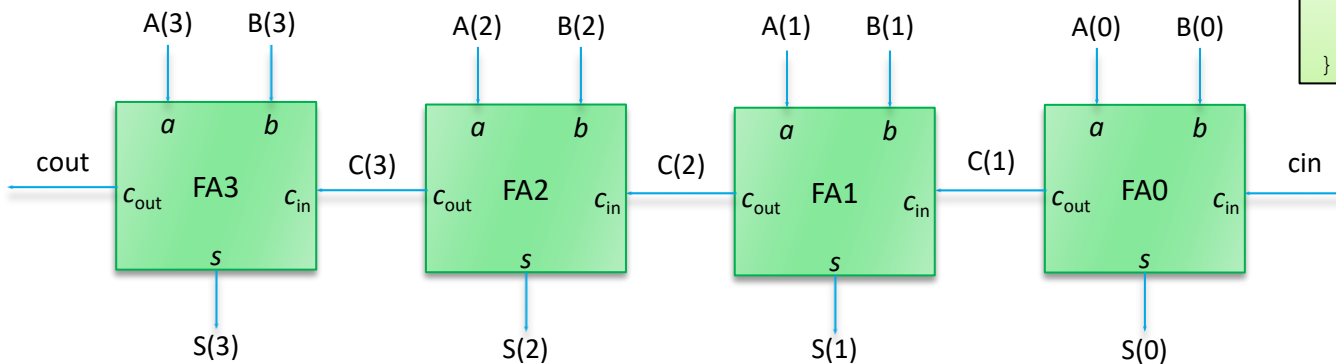
- ▶ **Quando si usa il componente, si assegna ad ogni parametro formale un valore effettivo**

- ▶ Si traduce la porta di ingresso/uscita nel contesto del livello di gerarchia superiore
- ▶ Per esempio, A(0) è collegato alla porta *a* di FA0
- ▶ Mentre A(1) è collegato alla porta *a* di FA1
- ▶ La porta *cout* di FA0 è collegata a C(1), al quale è collegata anche la porta *cin* di FA1

- ▶ **E' come quando si chiamano le funzioni in C**

- ▶ Si passano valori del contesto chiamante, che sono usati in quello chiamato

```
int f( int primo, int secondo ) {
  return primo + secondo;
}
int main( void ) {
  int a = 3;
  int b = 5;
  return f( a, b );
}
```



# Architettura strutturale in VHDL

## ► Segue la sintassi di chiamata a funzione

- Si indica nome dell'istanza e del modulo da istanziare
- Si indicano in ordine le porte che si vogliono collegare tramite un port map
- Notazione posizionale come in C

```
entity Adder4 is
  port (A, B : in bit_vector(3 downto 0);
        cin : in bit;
        S : out bit_vector(3 downto 0);
        cout : out bit);
end Adder4;
```

```
architecture strutturale of Adder4 is
  signal C : bit_vector(3 downto 1);
begin
  FA0: full_adder port map (A(0), B(0), cin, C(1), S(0));
  FA1: full_adder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: full_adder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: full_adder port map (A(3), B(3), C(3), cout, S(3));
end architecture strutturale;
```

```
entity full_adder is
  port (a, b, cin: in bit;
        cout, s: out bit);
end full_adder;
```

Nome dell'istanza

Nome del master

Corrispondenza porte

# Associazione esplicita

- ▶ Invece di usare una notazione posizionale è possibile indicare esplicitamente le connessioni
  - ▶ C'è più roba da scrivere
  - ▶ Non importa l'ordine
  - ▶ Più comodo quando il file VHDL viene generato da un tool

```
entity Adder4 is
  port (A, B : in bit_vector(3 downto 0);
        cin : in bit;
        S : out bit_vector(3 downto 0);
        cout : out bit);
end Adder4;
```

```
entity full_adder is
  port (a, b, cin: in bit;
        s, cout: out bit);
end full_adder;
```

Stesso nome ma oggetti diversi

```
architecture strutturale of Adder4 is
  signal C : bit_vector(3 downto 1);
begin
  FA0: full_adder port map (a => A(0), b => B(0), cin => cin, cout => C(1), s => S(0));
  FA1: full_adder port map (b => B(1), a => A(1), cout => C(2), cin => C(1), s => S(1));
  FA2: full_adder port map (cout => C(3), s => S(2), a => A(2), b => B(2), cin => C(2));
  FA3: full_adder port map (cin => C(3), a => A(3), b => B(3), cout => cout, s => S(3));
end architecture strutturale;
```

# Modi dei segnali

---

## ► Segnali di ingresso (in)

- Possono essere soltanto letti
- Nelle equazioni possono comparire solo a destra del segno <=

## ► Segnali di uscita (out)

- Possono essere soltanto scritti
- Nelle equazioni possono comparire solo a sinistra del segno <=
- Devono essere definiti da una sola espressione, altrimenti ci potrebbero essere dei conflitti

```
entity full_adder is
  port (a, b, cin : in bit; s, cout : out bit);
end full_adder;
architecture equazioni of full_adder is begin
  s <= a xor b xor cin;
  cout <= (a and b) or (a and cin) or (b and cin);
  a <= b xnor cin;
  s <= a and b;
  cout <= a or s;
end architecture equazioni;
```

Ingresso appare a sinistra

Equazioni di uscita in conflitto

Uscita appare a destra

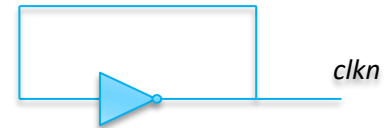
# Modi dei segnali

---

- ▶ **Segnali di uscita che possono essere anche letti (buffer)**
  - ▶ Devono essere definiti come **buffer** nella definizione di entità
  - ▶ Possono comparire da entrambi i lati del segno `<=`
  - ▶ Essendo uscite, devono comparire a sinistra di `<=` in una sola espressione

```
-- Generatore di clock a 100 MHz
entity clock_generator is
  port (clk_n : buffer bit);
end clock_generator;

architecture equazioni of clock_generator is
begin
  clk_n <= not clk_n after 5 ns;
end architecture equazioni;
```



# Modi dei segnali

---

## ► Segnali interni dell'architettura (signal)

- Possono essere sia letti che scritti
- Si comportano come i **buffer**
- Comunque devono essere scritti una volta sola
- In particolare, quando sono parte di istanziazioni, devono essere scritti una volta sola quando si espandono le istanziazioni con le equazioni
- Quindi attenzione a come si collegano i componenti (non collegare le uscite di due componenti assieme!)

```
architecture equazioni of Adder4 is
    signal C : bit_vector(3 downto 1);
begin
    S(0) <= A(0) xor B(0) xor cin;
    C(1) <= (A(0) and B(0)) or (A(0) and cin) or (B(0) and cin);
    S(1) <= A(1) xor B(1) xor C(1);
    C(2) <= (A(1) and B(1)) or (A(1) and C(1)) or (B(1) and C(1));
    S(2) <= A(2) xor B(2) xor C(2);
    C(3) <= (A(2) and B(2)) or (A(2) and C(2)) or (B(2) and C(2));
    S(3) <= A(3) xor B(3) xor C(3);
    cout <= (A(3) and B(3)) or (A(3) and C(3)) or (B(3) and C(3));
end architecture equazioni;
```

```

architecture equazioni of full_adder is
begin
    s <= a xor b xor cin;
    cout <= (a and b) or (a and cin)
           or (b and cin);
end architecture equazioni;

```

```

architecture strutturale of Adder4 is
    signal C : bit_vector(3 downto 1);
begin
    FA0: full_adder port map (A(0), B(0), cin, C(1), S(0));
    FA1: full_adder port map (A(1), B(1), C(1), C(2), S(0));
    FA2: full_adder port map (A(2), B(2), C(2), C(3), S(2));
    FA3: full_adder port map (A(3), B(3), C(3), cout, S(3));
end architecture strutturale;

```

```

architecture strutturale of Adder4 is
    signal C : bit_vector(3 downto 1);
begin
    S(0) <= A(0) xor B(0) xor cin;
    C(1) <= (A(0) and B(0)) or (A(0) and cin) or (B(0) and cin);
    S(0) <= A(1) xor B(1) xor C(1);
    C(2) <= (A(1) and B(1)) or (A(1) and C(1)) or (B(1) and C(1));
    S(2) <= A(2) xor B(2) xor C(2);
    C(3) <= (A(2) and B(2)) or (A(2) and C(2)) or (B(2) and C(2));
    S(3) <= A(3) xor B(3) xor C(3);
    cout <= (A(3) and B(3)) or (A(3) and C(3)) or (B(3) and C(3));
end architecture strutturale;

```



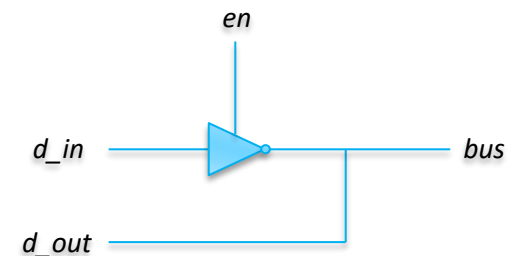
# Modi dei segnali

---

## ► Segnali sia di ingresso che di uscita (inout)

- Dal punto di vista del componente sono come i **buffer**
- Ma possono avere un driver anche dall'esterno, mentre i **buffer** no
- Possono apparire a sinistra di  $\leq$  in più espressioni
- Occorre però risolvere eventuali conflitti
- Usati per lo più per rappresentare buffer tri-state
- L'uso effettivo richiede un ulteriore tipo di dato (a breve!)

```
-- Interfaccia tri-state verso il bus
entity bus_interface is
  port (en, d_in : in bit;
        d_out : out bit
        bus : inout bit);
end entity bus_interface;
```



# Parametri (generic)

---

- ▶ **E' spesso utile poter parametrizzare un componente**
  - ▶ In questo modo ne scriviamo la descrizione una sola volta
  - ▶ Quando lo si istanzia si definisce il valore del parametro

```
-- Porta con ritardo parametrizzato
entity And2 is
  generic (delay : time);
  port (x, y : in bit; z : out bit);
end entity And2;
architecture ex2 of And2 is begin
  z <= x and y after delay;
end architecture ex2;

...
g1: And2 generic map (5 ns)
  port map (p, b, q);
g2: And2 generic map (delay => 3 ns)
  port map (z => r, y => p, x => s);
...
```

Definizione parametro

Uso del parametro

Istanziamento del  
parametro

# Parametri (generic)

---

## ► E' possibile fornire un valore di default

- Il valore di default viene usato quando non viene specificato nulla nell'istanza
- Si può comunque modificarne il valore come al solito

```
-- Porta con ritardo parametrizzato e default
entity And2 is
  generic (delay : time := 5 ns);
  port (x, y : in bit; z : out bit);
end entity And2;
architecture ex2 of And2 is begin
  z <= x and y after delay;
end architecture ex2;

...
g1: And2 port map (p, b, q);

g2: And2 generic map (delay => 3 ns)
  port map (z => r, y => p, x => s);
...
```

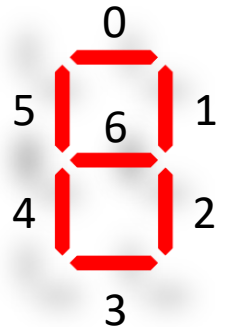
Definizione parametro  
con valore di default

Istanziamento con uso  
del valore di default

# Costanti

- ▶ **Assieme ai segnali è possibile definire delle costanti**
  - ▶ **constant** nome\_costante: tipo := valore
  - ▶ Utili per definire valori simbolici da poter usare nel resto del codice
  - ▶ Per esempio per definire le dimensioni degli array, o dare valori simbolici alla codifica degli stati
  - ▶ Se occorre modificarli, basta farlo in un punto solo del codice

```
-- Uso di costanti
architecture equazioni of display is
    constant segmenti : integer := 7;
    constant zero : bit_vector(segmenti-1 downto 0) := "0111111";
    constant uno : bit_vector(segmenti-1 downto 0) := "0000110";
    ...
    constant cinque : bit_vector(segmenti-1 downto 0) := "1101101";
    ...
begin
    uscita <= cinque when (ingresso = 5);
end architecture equazioni;
```



# Tipi

---

## ▶ Ogni segnale deve avere un tipo

- ▶ Ne caratterizza i possibili valori
- ▶ Abbiamo visto il tipo `bit` e `bit_vector`
  - ▶ I segnali di questo tipo possono assumere i valori '0' oppure '1'
- ▶ Con la keyword **downto** si definisce l'ordinamento dei bit
  - ▶ `b : in bit_vector (3 downto 0);`
  - ▶ `b <= "1100"` assegna a `b(3)` e `b(2)` il valore 1, e a `b(1)` e `b(0)` il valore 0
- ▶ La keyword **to** definisce l'ordinamento contrario
  - ▶ `b : in bit_vector (3 to 0);`
  - ▶ `b <= "1100"` assegna a `b(3)` e `b(2)` il valore 0, e a `b(1)` e `b(0)` il valore 1

## ▶ Altri tipi

- ▶ Abbiamo visto il tipo "integer"
- ▶ Abbiamo visto il tipo "time"

# Tipi definiti dall'utente

---

- ▶ **Possiamo definire i tipi che vogliamo**
  - ▶ Come il typedef in C
  - ▶ Aumenta la leggibilità del codice e permette di evitare errori
  - ▶ **type** word **is** bit\_vector (15 **downto** 0);

```
-- Uso di tipi definiti dall'utente
architecture equazioni of display is
    constant segmenti : integer := 7;
    type 7segments is bit_vector(segmenti-1 downto 0);
    constant zero : 7segments := "0111111";
    constant uno  : 7segments := "0000110";
    ...
begin
    uscita <= cinque and (ingresso = 5);
end architecture equazioni;
```

# Tipi compositi

---

- ▶ **Il VHDL offre i normali modi di definire tipi compositi**
  - ▶ Consentono di rendere il codice più simbolico e meno dipendente dalla specifica implementazione

```
-- Tipi enumerati
type op_select_type is (load, store, add, sub, div, mult, shiftl, shiftr);
type hex_digit is ('0', '1', '2', '3', '4', '5', '6', '7', '8',
                  '9', 'A', 'B', 'C', 'D', 'E', 'F');
type state_type is (S0, S1, S2, S3);

-- Tipi array
type my_word is array (15 downto 0) of hex_digit;

-- Tipi struttura
type packet is record
    RISE_TIME : time;
    FALL_TIME : time;
    SIZE : integer range 0 to 200;
    DATA : bit_vector (15 downto 0);
end record;
...
signal A, B : packet;
...
A.RISE_TIME <= 5ns;
A.SIZE <= 120;
B <= A;
```

# Il tipo std\_logic

---

- ▶ **Definito nella libreria standard 1164 della IEEE**
  - ▶ Tipologia non numerica che usa 9 valori

Valore	Significato
U	Non definito (a cui non è mai stato dato un valore)
X	Sconosciuto (il cui valore non è determinabile)
0	0 logico
1	1 logico
Z	Alta impedenza
W	Segnale debole, senza un valore determinabile
L	Segnale debole che probabilmente andrà a 0
H	Segnale debole che probabilmente andrà a 1
—	Indifferente (don't care)



# Uso del tipo std\_logic

## ► Sostituisce, estendendolo, il tipo bit

- Occorre però ridefinire tutti gli operatori (**and**, **or**, **xor**, etc.) per usufruire dei nuovi valori
- La ridefinizione si chiama *overloading* dell'operatore
- Per esempio l'operatore **and** è ri-definito come segue

	U	X	0	1	Z	W	L	H	–
U	U	U	0	U	U	U	0	U	U
X	U	X	0	X	X	X	0	X	X
0	0	0	0	0	0	0	0	0	0
1	U	X	0	1	X	X	0	1	X
Z	U	X	0	X	X	X	0	X	X
W	U	X	0	X	X	X	0	X	X
L	0	0	0	0	0	0	0	0	0
H	U	X	0	1	X	X	0	1	X
–	U	X	0	X	X	X	0	X	X



# Uso di espressioni condizionate

---

- ▶ **Come esprimere delle condizioni in una equazione?**
  - ▶ In una espressione non possiamo usare lo statement **if**, abbiamo invece solo operatori
  - ▶ C'è bisogno di un operatore condizionato
- ▶ **E' presente per esempio in C o in Java**
  - ▶ `int z = (b < 10) ? a : a * b;`
- ▶ **In VHDL ve ne sono di due tipi**
  - ▶ Assegnazione condizionata
  - ▶ Assegnazione selezionata

# Assegnazione condizionata

- ▶ Realizziamo, per esempio, un buffer tri-state
  - ▶ Se enable è attivo, l'uscita è uguale all'ingresso
  - ▶ Altrimenti è in alta impedenza 'Z'
- ▶ Direttiva **when ... else**
  - ▶ Funziona come l'if, ma in forma di equazione

```
-- Assegnazione condizionata, il buffer tri-state
library ieee;
use ieee.std_logic_1164.all;

entity three_state is
    port (a, enable : in std_logic;
          y : out std_logic);
end entity three_state;

architecture when_else of three_state is
begin
    y <= a when enable = '1' else 'Z';
end architecture when_else;
```

Inclusione della libreria standard 1164

if enable = 1 then y = a  
else y = 'Z';

L'operatore di uguaglianza  
è = invece di == come in  
altri linguaggi

# Scelte annidate

- ▶ La parte else può essere una qualunque espressione
  - ▶ In particolare può essere un altro **when ... else**
  - ▶ E' possibile quindi fare scelte multiple
- ▶ Realizziamo un decoder 2-4
  - ▶ Il simulatore analizza le varie condizioni nell'ordine in cui compaiono
  - ▶ La prima che risulta verificata viene presa, le rimanenti sono ignorate (quindi non devono essere necessariamente mutualmente esclusive)

```
-- Assegnazione condizionata, il decoder 2-4
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
    port (a : in std_logic_vector(1 downto 0);
          y : out std_logic_vector(3 downto 0));
end entity decoder;

architecture when_else of decoder is
begin
    y <= "0001" when a = "00" else
        "0010" when a = "01" else
        "0100" when a = "10" else
        "1000" when a = "11" else
        "XXXX";
end architecture when_else;
```

Array di ingressi e uscite

Doppi apici quando si  
definisce un valore multibit

Possibili alternative

Alternativa finale: il  
segnale *a* potrebbe avere X  
o altri valori non binari

# Assegnazione selezionata

- ▶ **E' come lo switch ... case di molti linguaggi di programmazione**
  - ▶ Si scelgono le alternative sulla base del valore di un segnale
  - ▶ Le alternative vengono considerate contemporaneamente, devono quindi essere mutualmente esclusive

```
-- Assegnazione selezionata, il decoder 2-4
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
  port (a : in std_logic_vector(1 downto 0);
        y : out std_logic_vector(3 downto 0));
end entity decoder;

architecture with_select of decoder is
begin
  with a select
    y <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when "11",
         "XXXX" when others;
end architecture with_select;
```

Segnale su cui eseguire la selezione

Valori dei vari casi.  
Attenzione: sono valori,  
non espressioni booleane!

Per tutti gli altri casi

# Alternative equivalenti

- ▶ Quando diverse alternative danno lo stesso risultato si possono raggruppare
  - ▶ Realizziamo un decoder a 7 segmenti decimale
  - ▶ Le configurazioni oltre il 9 mostrano la lettera E per segnalare errore

```
-- Decoder 7 segmenti
entity seven_seg is
  port (a : in std_logic_vector(3 downto 0);
        y : out std_logic_vector(6 downto 0));
end entity seven_seg;

architecture with_select of seven_seg is
begin
  with a select
    y <= "0111111" when "0000",
         "0000110" when "0001",
         "1011011" when "0010",
         "1001111" when "0011",
         "1100110" when "0100",
         "1101101" when "0101",
         "1111101" when "0110",
         "0000111" when "0111",
         "1111111" when "1000",
         "1101111" when "1001",
         "1111001" when "1010" | "1011" | "1100" |
                                "1101" | "1110" | "1111",
         "0000000" when others;
end architecture with_select;
```



Alternative equivalenti  
separate da |

# Differenza semantica

- ▶ Usare uno o l'altro metodo è spesso lo stesso
  - ▶ Ma vi sono delle sostanziali differenze semantiche
  - ▶ L'assegnazione condizionata può guardare per esempio condizioni su più segnali diversi

```
-- Multiplexer, assegnazione selezionata
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer is
    port (a, b, c : in std_logic;
          sa, sb, sc : in std_logic;
          y : out std_logic);
end entity multiplexer;

architecture with_select of multiplexer is
    signal selez : std_logic_vector(2 downto 0);
begin
    selez <= (sa, sb, sc);
    with selez select
        y <= a when "100",
            b when "010",
            c when "001",
            '0' when others;
end architecture with_select;
```

Concatenazione di  
segnali

```
-- Multiplexer, assegnazione condizionata
library ieee;
use ieee.std_logic_1164.all;

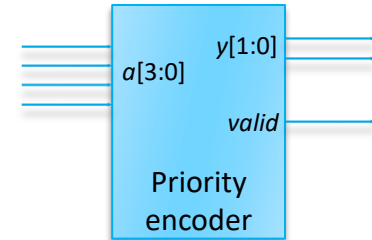
entity multiplexer is
    port (a, b, c : in std_logic;
          sa, sb, sc : in std_logic;
          y : out std_logic);
end entity multiplexer;

architecture when_else of multiplexer is
begin
    y <= a when sa = '1' else
        b when sb = '1' else
        c when sc = '1' else
        '0';
end architecture when_else;
```

Non mutualmente  
esclusivi

# Priority encoder e don't care

- ▶ I don't care in std\_logic sono valori come altri
  - ▶ Non vengono interpretati come ci aspettiamo
  - ▶ Meglio usare condizioni nidificate con **when ... else**



```
-- Priority encoder, don't care (non funziona)
library IEEE;
use IEEE.std_logic_1164.all;

entity priority is
    port (a: in std_logic_vector(3 downto 0);
          y: out std_logic_vector(1 downto 0);
          valid: out std_logic);
end entity priority;

architecture DontCare of priority is
begin
    with a select
        y <= "00" when "0001",
            "01" when "001-",
            "10" when "01--",
            "11" when "1---",
            "00" when others;
    valid <= '1' when a(0) = '1' or a(1) = '1' or
                  a(2) = '1' or a(3) = '1'
              else '0';
end architecture DontCare;
```

**Non funziona!**

```
-- Priority encoder, when ... else
library IEEE;
use IEEE.std_logic_1164.all;

entity priority is
    port (a: in std_logic_vector(3 downto 0);
          y: out std_logic_vector(1 downto 0);
          valid: out std_logic);
end entity priority;

architecture Ordered of priority is
begin
    y <= "11" when a(3) = '1' else
        "10" when a(2) = '1' else
        "01" when a(1) = '1' else
        "00" when a(0) = '1' else
        "00";
    valid <= '1' when a(0) = '1' or a(1) = '1' or
                  a(2) = '1' or a(3) = '1'
              else '0';
end architecture Ordered;
```



# Stile comportamentale

**I processi**

# Stile comportamentale

---

- ▶ **Prevedere molte alternative può diventare laborioso**
  - ▶ Ogni uscita deve avere una sola espressione
  - ▶ Diventa difficile riuscire a partizionare il calcolo
- ▶ **Per questo è possibile usare uno stile di specifica comportamentale**
  - ▶ Sono parti di codice eseguite in sequenza, invece che in concorrenza
  - ▶ Si realizzano all'interno di processi
  - ▶ Il codice è simile a quello che si scrive per i tradizionali linguaggi di programmazione
  - ▶ Utile sia per circuiti combinatori che per circuiti sequenziali
- ▶ **Fondamentale capire come e quando viene attivato il codice durante la simulazione**
  - ▶ Arricchisce la semantica vista fino ad ora

# Processo

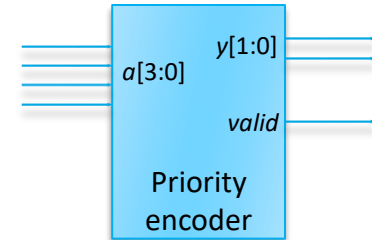
---

- ▶ **Si usa all'interno di una architettura**
  - ▶ Lo si usa al posto di una equazione
- ▶ **Sensitivity list**
  - ▶ Una lista di segnali che definiscono le condizioni di attivazione del processo
  - ▶ Il processo viene eseguito ogni volta che uno dei segnali nella sensitivity list cambia di valore

```
-- Sintassi di un processo
architecture comportamentale of mio_modulo is
begin
    label: process (sensitivity-list) is
        dichiarazioni di tipi, costanti, segnali, variabili;
    begin
        statement-sequenziale;
        ...
        statement-sequenziale;
    end process;
end architecture comportamentale;
```

# Rivediamo il priority encoder

- ▶ Le uscite vanno aggiornate ogni volta che cambia l'ingresso *a*
  - ▶ Definiamo allora un processo sensibile alle variazioni di *a*
- ▶ Una serie di if annidati controllano gli ingressi attivi
  - ▶ Cominciamo con quello a priorità più alta
  - ▶ Un ultimo caso gestisce tutte le altre alternative
- ▶ **Componente combinatorio**
  - ▶ Anche se abbiamo usato uno stile comportamentale, l'effetto totale è combinatorio
  - ▶ Dato *a* è sempre possibile determinare il valore di tutte le uscite
  - ▶ Il processo è come una espressione che pilota il valore di *y* e di *valid*



```
-- Priority encoder, comportamentale
architecture sequential of priority is
begin
  process (a) is
  begin
    if a(3) = '1' then
      y <= "11";
      valid <= '1';
    elsif a(2) = '1' then
      y <= "10";
      valid <= '1';
    elsif a(1) = '1' then
      y <= "01";
      valid <= '1';
    elsif a(0) = '1' then
      y <= "00";
      valid <= '1';
    else
      y <= "00";
      valid <= '0';
    end if;
  end process;
end architecture sequential;
```

# Semantica

---

## ▶ Attivazione

- ▶ Ogni volta che un segnale ha un evento il simulatore guarda quali processi sono sensibili verificando le sensitivity list
- ▶ Attiva quelli che hanno il segnale nella lista

## ▶ Esecuzione

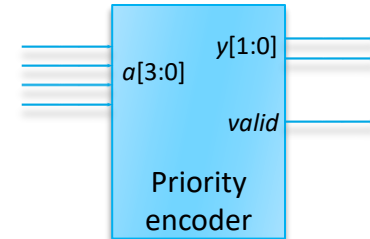
- ▶ Il simulatore esegue in sequenza gli statement contenuti nel processo
- ▶ Il tempo di simulazione NON avanza durante l'esecuzione
- ▶ I nuovi valori dei segnali NON vengono assegnati immediatamente
- ▶ L'assegnazione avviene invece solo dopo che il processo termina o viene sospeso (tipicamente quando si arriva al fondo del processo)

## ▶ Assegnazione dei segnali

- ▶ Uno stesso segnale (di uscita) può essere assegnato più volte in un processo, anche con valori diversi
- ▶ Tanto l'assegnazione non si fa subito, ma alla fine del processo
- ▶ Alla fine del processo si assegna effettivamente l'ultimo dei valori assegnati

# Valori di default per circuiti combinatori

- ▶ Utile assegnare un valore a tutte le uscite ad inizio processo
  - ▶ Nel processo è poi necessario solamente gestire i casi non già coperti dal default
  - ▶ Questo inoltre garantisce contro certi errori (inferenza di latch) che vedremo in seguito
  - ▶ Nel caso del priority encoder, di default assumiamo che l'uscita sia valida e poniamo `valid <= '1'`;
  - ▶ Se  $a = "0000"$  allora poniamo `valid <= '0'`
  - ▶ Notate che *valid* viene assegnato due volte durante l'esecuzione quando  $a = "0000"$



```
-- Priority encoder, comportamentale
architecture sequential of priority is
begin
  process (a) is
  begin
    valid <= '1';
    if a(3) = '1' then
      y <= "11";
    elsif a(2) = '1' then
      y <= "10";
    elsif a(1) = '1' then
      y <= "01";
    elsif a(0) = '1' then
      y <= "00";
    else
      y <= "00";
      valid <= '0';
    end if;
  end process;
end architecture sequential;
```

# Variabili

---

- ▶ Quando si scrivono algoritmi iterativi (loop), è utile aggiornare certe quantità in modo incrementale
  - ▶ Questo **non lo si può fare con i segnali**, perché vengono aggiornati solo a fine processo, e non cambierebbero durante tutta l'esecuzione del loop
  - ▶ Nei processi è allora possibile definire delle **variabili**
  - ▶ Le variabili prendono il nuovo valore immediatamente quando gli viene assegnato
  - ▶ Quindi se vengono assegnate più volte durante il processo, il loro valore cambia di volta in volta (mentre per i segnali cambierebbe una sola volta, alla fine)
  - ▶ Mantengono il valore tra una attivazione e la successiva
- ▶ **Assegnazione**
  - ▶ L'operatore di assegnazione per le variabili è `:=`
  - ▶ Per distinguerlo da quello dei segnali

# Controllo parità

Uscita a 1 se il numero di bit a 1 è dispari

## ► Realizziamo un controllo di parità in modo iterativo

- Assumiamo inizialmente che una parola sia pari
- Eseguiamo un ciclo **for** che analizza tutti i bit della parola di ingresso
- Ogni volta che si incontra un 1 aggiorniamo la parità
  - Se era pari diventa dispari, se era dispari diventa pari
- Quello che ci rimane alla fine è la parità effettiva

## ► Se la variabile fosse un segnale non funzionerebbe

- Il segnale non cambierebbe valore da una iterazione alla successiva

## ► Il modello è comunque combinatorio

- Anche se usa un ciclo, ricordate che il tempo non sta andando avanti
- Usiamo un segnale *y* per propagare il risultato all'esterno

```
-- Controllo di parità
library ieee;
use ieee.std_logic_1164.all;

entity parity is
  port (a : in std_logic_vector;
        y : out std_logic);
end entity parity;

architecture iterative of parity is
begin
  process (a) is
    variable even : std_logic;
  begin
    even := '0';
    for i in a'range loop
      if a(i) = '1' then
        even := not even;
      end if;
    end loop;
    y <= even;
  end process;
end architecture iterative;
```

No range

Definizione  
variabile

Range desunto  
dal contesto

Assegnamento  
variabile

Risultato finale



# Segnali e variabili

---

Segnali	Variabili
Usati per la comunicazione tra componenti e nello stile data flow	Usato solo per memorizzare dati
Può essere visto come un segnale reale di un circuito, essendo disponibile per essere condiviso tra più componenti	L'uso è locale al processo, non può essere condiviso tra componenti diversi
L'assegnazione avviene con un certo ritardo rispetto al tempo della sua esecuzione	L'assegnazione avviene immediatamente al tempo della sua esecuzione

# Confronto tra i due usi

---

- ▶ **I risultati usando variabili e segnali sono differenti**
  - ▶ Supponiamo che i segnali  $x$  e  $y$  valgano inizialmente 1
  - ▶ Se  $y$  va a 0, entrambi i processi vengono attivati
  - ▶ Nell'esempio di sinistra, l'assegnamento a  $x$  viene deferito fino alla fine del processo, quindi  $z$  a fine processo prende il valore 0, perché  $x$  valeva ancora 1 al momento della valutazione
  - ▶ Nell'esempio di destra,  $x$  è una variabile, quindi il suo valore viene aggiornato immediatamente a 0; alla valutazione dell'espressione di  $z$  si ottiene quindi 1

```
-- Algoritmo con segnali
architecture ... is
    signal x : std_logic;
begin
    process (y) is
    begin
        x <= y;
        z <= not x;
    end process;
end architecture;

x = 0; y = 0; z = 0
```

```
-- Algoritmo con variabili
architecture ... is
begin
    process (y) is
        variable x : std_logic;
    begin
        x := y;
        z <= not x;
    end process;
end architecture;

x = 0; y = 0; z = 1
```

# Equazioni e processi

---

	Equazioni	Processi
<b>Attivazione</b>	Ogni volta che un segnale a destra di $\leq$ cambia valore	Ogni volta che un segnale in sensitivity list cambia valore
<b>Pilota</b>	Determina il valore di un solo segnale	Può determinare il valore di più segnali contemporaneamente
<b>Assegnazione</b>	Evento inserito in coda dopo un delta cycle o il ritardo specificato	Evento/i inserito in coda a sospensione processo, dopo delta cycle o ritardo specificato
<b>Conflitti</b>	Altre equazioni o processi non possono pilotare lo stesso segnale	Altre equazioni o processi non possono pilotare gli stessi segnali
<b>Variabili</b>	No	Si

# Testbench con processi

---

## ▶ I processi semplificano la scrittura dei testbench

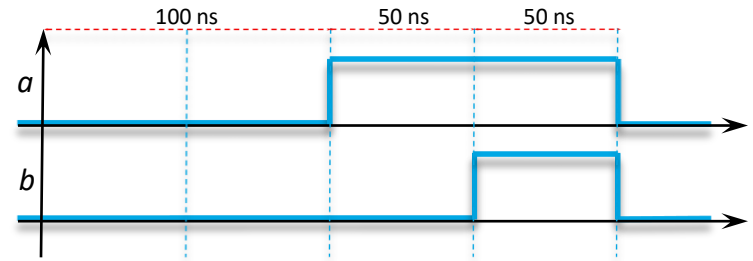
- ▶ Usiamo un processo senza sensitivity list per definire i segnali a priori
- ▶ Poiché non c'è sensitivity list, il processo viene attivato immediatamente all'inizio della simulazione
- ▶ Appena termina viene riattivato immediatamente
- ▶ Allora come lo fermo?

## ▶ Il comando wait

- ▶ Se incontriamo un comando **wait** all'interno di un processo l'esecuzione si interrompe per un tempo specificato dal comando
- ▶ Oppure si può interrompere il processo per sempre se non si specifica il tempo
- ▶ Quando si interrompe il processo, si eseguono gli assegnamenti ai segnali (come se terminasse)

# Uso del wait

- ▶ Si specifica un tempo di sospensione con **wait for**
  - ▶ Il processo verrà riattivato quando passa il tempo specificato dal punto in cui era stato sospeso
  - ▶ Questo ha il vantaggio di poter usare tempi relativi invece che assoluti come fatto precedentemente
- ▶ Sospensione completa
  - ▶ Si usa **wait** e basta per terminare il processo



```
-- Testbench per porta And a 2 ingressi
entity TestAnd2 is
end entity TestAnd2;

architecture simple of TestAnd2 is
    -- Segnali interni di interconnessione
    signal a, b, c : bit;
begin
    -- Istanza del modulo da testare
    g1: And2 port map (x => a, y => b, z => c);
    -- Definizione degli stimoli
    a <= '0', '1' after 100 ns, '0' after 200 ns;
    b <= '0', '1' after 150 ns, '0' after 200 ns;
end architecture simple;
```

```
architecture simple of TestAnd2 is
    -- Segnali interni di interconnessione
    signal a, b, c : bit;
begin
    -- Istanza del modulo da testare
    g1: And2 port map (x => a, y => b, z => c);
    -- Definizione degli stimoli
    process is
    begin
        a <= '0';
        b <= '0';
        wait for 100 ns;
        a <= '1';
        wait for 50 ns;
        b <= '1';
        wait for 50 ns;
        a <= '0';
        b <= '0';
        wait;
    end process;
end architecture simple;
```

# Libreria numeric\_std

---

- ▶ Introduce i tipi **signed** e **unsigned**
  - ▶ Sono dei vettori come gli std\_logic\_vector
  - ▶ La libreria definisce operatori aritmetici e di confronto (in modo differente per signed e unsigned)
  - ▶ Consente la trasformazione da e per std\_logic\_vector
- ▶ La libreria std\_logic\_arith è simile ma non è uno standard IEEE

```
-- Uso di operatori di confronto
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cp is
    port (s : in std_logic_vector (7 downto 0);
          s_big, s_neg : out std_logic);
end entity cp;

architecture beh of cp is
begin
    s_big <= '1' when ( unsigned(s) > 200 ) else '0';
    s_neg <= '1' when ( signed(s) < 0 ) else '0';
end architecture beh;
```

# Sommatore

- ▶ **Eseguiamo l'operazione come signed o unsigned**
  - ▶ Definiamo dei segnali temporanei di tipo signed o unsigned
  - ▶ Allineiamo a n+1 bit tutti i vettori
  - ▶ Per unsigned estendiamo con '0'
  - ▶ Per signed estendiamo con il valore del segno
  - ▶ Quindi si esegue la somma e si estrae il carry finale

```
-- Uso di operatori aritmetici
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

entity Adder is
    generic (n : natural := 4);
    port (a, b : in std_logic_vector (n-1 downto 0);
          cin : in std_logic;
          sum : out std_logic_vector (n-1 downto 0);
          cout : out std_logic);
end entity Adder;

architecture unsqnd of Adder is
    signal result, carry : unsigned(n downto 0);
    constant zeros : unsigned(n-1 downto 0) := (others => '0');
begin
    carry <= (zeros & cin);
    result <= ('0' & unsigned(a)) + ('0' & unsigned(b)) + carry;
    sum <= std_logic_vector( result(n-1 downto 0) );
    cout <= result(n);
end architecture unsqnd;

architecture sqnd of Adder is
    signal result, carry : signed(n downto 0);
    constant zeros : signed(n-1 downto 0) := (others => '0');
begin
    carry <= (zeros & cin);
    result <= (a(n-1) & signed(a)) + (b(n-1) & signed(b)) + carry;
    sum <= std_logic_vector( result(n-1 downto 0) );
    cout <= result(n);
end architecture sqnd;
```

# Libreria std\_logic\_signed/unsigned

---

- ▶ Permette di usare gli std\_logic direttamente come fossero dei numeri

```
-- Uso di operatori aritmetici
library ieee;
use ieee.std_logic_1164.all, ieee.std_logic_signed.all;

entity Adder is
  generic (n : natural := 4);
  port (a, b : in std_logic_vector(n-1 downto 0);
        sum : out std_logic_vector(n-1 downto 0);
        v : out std_logic);
end entity Adder;

architecture sgnd of Adder is begin
  process (a, b) is
    variable y : std_logic_vector(n-1 downto 0);
  begin
    y := a + b;
    if (a > 0 and b > 0 and y < 0) or
       (a < 0 and b < 0 and y > 0) then
      v <= '1';
    else
      v <= '0';
    end if;
    sum <= y;
  end process;
end architecture sgnd;
```



## **VHDL per circuiti sequenziali**

# Specificare memorie

---

- ▶ **Per ora abbiamo visto tutti oggetti combinatori**
  - ▶ Caratterizzati dalla mancanza di memoria
  - ▶ Quindi, per ogni valore degli ingressi, il codice deve specificare il valore delle uscite
- ▶ **I circuiti sequenziali devono essere in grado di memorizzare dei valori**
  - ▶ Le uscite devono poter dipendere dai loro valori precedenti
  - ▶ Basta usare un tipo **buffer**

```
-- Latch di tipo D
entity latch is
  port (d, c : in std_logic; q : buffer std_logic);
end entity latch;

architecture equazioni of latch is
begin
  q <= d when c = '1' else q;
end architecture equazioni;
```

Occorre ricordare il valore precedente di q, quindi serve una memoria

# Edge triggered

---

- ▶ **Come far aggiornare un segnale solo al fronte di un clock?**
  - ▶ Occorre verificare che ci sia stato effettivamente un evento sul clock
  - ▶ Lo si può fare guardando gli *attributi* di un segnale
    - ▶ 'event: il segnale ha un evento nell'istante di simulazione
    - ▶ 'stable: il segnale non ha un evento nell'istante di simulazione
- ▶ **Quindi basta aggiornare  $q$  solo quando  $c$  è ad 1 ed ha un evento, cioè siamo al fronte di salita**
  - ▶ Così  $d$  può cambiare mentre  $c = 1$ , ma il valore di  $q$  non viene aggiornato

```
-- Flip flop di tipo D edge triggered
entity flip_flop is
  port (d, c : in std_logic; q : buffer std_logic);
end entity flip_flop;

architecture equazioni of flip_flop is
begin
  q <= d when c = '1' and c'event else q;  -- oppure and not c'stable
end architecture equazioni;
```

# Funzioni pre-definite

---

- ▶ **L'edge triggered è così comune che si è pensato di definire una funzione apposta**
  - ▶ Si usa l'espressione `rising_edge( c )`
  - ▶ Ovviamente c'è anche `falling_edge( c )`
  - ▶ Nel `std_logic`, evitano di triggerare se c'è un evento da H a 1!
- ▶ **Facile aggiungere anche un reset**
  - ▶ Lo facciamo per esempio asincrono attivo basso
  - ▶ E se lo si volesse fare sincrono (che abbia effetto solo al fronte del clock)?

```
-- Flip flop di tipo D edge triggered
entity flip_flop is
  port (d, c, r : in std_logic; q : buffer std_logic);
end entity flip_flop;

architecture equazioni of flip_flop is
begin
  q <= '0' when r = '0' else
    d when rising_edge( c ) else
      q;
end architecture equazioni;
```

# Processi sequenziali

**Registri, contatori, etc.**

# Perché i modelli visti sono combinatori?

---

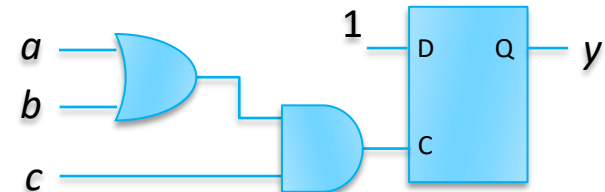
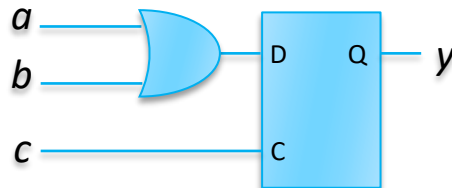
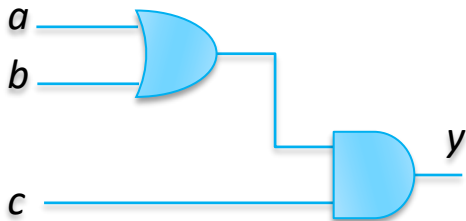
- ▶ Il circuito si comporta in maniera combinatoria se è sempre possibile assegnare un valore ai segnali senza dover ricordare il valore precedente
  - ▶ Quando si definisce un circuito **combinatorio**, occorre allora fare attenzione a fare **sempre un'assegnazione alle uscite per tutti i possibili valori degli ingressi**
  - ▶ **Se non lo si fa, gli strumenti di sintesi produrranno un latch o un flip flop per ricordare il valore precedente**, visto che uno nuovo non viene assegnato
  - ▶ Questo vale sia per lo stile data flow che per lo stile comportamentale
  - ▶ Le condizioni per le quali occorre ricordare vanno a formare il clock degli elementi sequenziali introdotti

# Inferenza di latch

```
process (a, b, c) is
begin
  if (c = '1') then
    if (a = '1') or (b = '1') then
      y <= '1';
    else
      y <= '0';
    end if;
  else
    y <= '0';
  end if;
end process;
```

```
process (a, b, c) is
begin
  if (c = '1') then
    if (a = '1') or (b = '1') then
      y <= '1';
    else
      y <= '0';
    end if;
  else
    y <= '0';
  end if;
end process;
```

```
process (a, b, c) is
begin
  if (c = '1') then
    if (a = '1') or (b = '1') then
      y <= '1';
    else
      y <= '0';
    end if;
  else
    y <= '0';
  end if;
end process;
```



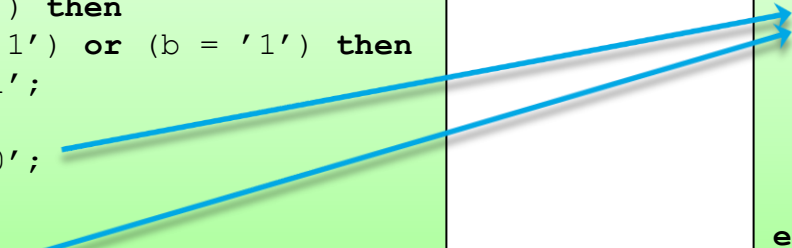
# Per evitare errori

---

- ▶ Se si vuole fare un circuito combinatorio è sufficiente dare alle uscite dei valori di default all'inizio del processo
  - ▶ In questo modo sono sicuramente sempre assegnate almeno una volta e non si inferisce un latch
  - ▶ Se il default non è corretto basta fare una seconda assegnazione (tanto vale l'ultima!)

```
process (a, b, c) is
begin
  if (c = '1') then
    if (a = '1') or (b = '1') then
      y <= '1';
    else
      y <= '0';
    end if;
  else
    y <= '0';
  end if;
end process;
```

```
process (a, b, c) is
begin
  y <= '0';
  if (c = '1') then
    if (a = '1') or (b = '1') then
      y <= '1';
    end if;
  end if;
end process;
```





# Registri

---

## ▶ Latch di tipo D

- ▶ Il latch è sensibile sia all'ingresso di dato  $d$ , sia al clock  $clk$
  - ▶ Il processo è sensibile sia alle variazioni di  $d$  sia alle variazioni di  $clk$
  - ▶ Si assegna a  $q$  il valore di  $d$  solo quando  $clk$  vale 1
  - ▶ Se  $d$  cambia e  $clk$  è 0, allora il circuito deve fornire il valore precedente di  $q$
  - ▶ Si crea quindi una memoria
- ▶ **Notare di nuovo la sensitivity list**
- ▶ Contiene sia  $d$  sia  $clk$

```
-- Latch di tipo D
library ieee;
use ieee.std_logic_1164.all;

entity D_latch is
    port (d, clk : in std_logic;
          q : out std_logic);
end entity D_latch;

architecture behavioral of D_latch is
begin
    process (d, clk) is
    begin
        if clk = '1' then
            q <= d;
        end if;
    end process;
end architecture behavioral;
```

# Registri edge triggered

---

## ► Flip flop di tipo D

- Il codice è simile a quello precedente
- Il processo però è sensibile solo al segnale *clk*, e non al segnale *d*
- Che non vuol dire che *q* non dipenda da *d*, ma solo che il valore di *q* cambia solo in corrispondenza delle variazioni di *clk*
- In questo modo il valore di *q* viene aggiornato solo quando *clk* esegue una transizione
- Poiché si verifica che *clk* sia 1 prima di fare l'assegnazione, questa avviene solo al fronte di salita

```
-- Flip flop di tipo D
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (d, clk : in std_logic;
          q : out std_logic);
end entity D_FF;

architecture behavioral of D_FF is
begin
    process (clk) is
    begin
        if clk = '1' then
            q <= d;
        end if;
    end process;
end architecture behavioral;
```

# Registri edge triggered

---

- ▶ Flip flop di tipo D con **reset asincrono**
  - ▶ Aggiungiamo un *reset* attivo basso alla interfaccia
  - ▶ Il processo è sensibile sia al clock che al reset
  - ▶ Per il segnale *clk* dobbiamo però ora controllare esplicitamente che ci sia un fronte
  - ▶ Se usassimo il codice di prima, si potrebbe assegnare *d* a *q* se *clk* è 1 e *res* va da 0 a 1

```
-- Flip flop di tipo D con reset async
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (d, clk, res : in std_logic;
          q : out std_logic);
end entity D_FF;

architecture behavioral of D_FF is
begin
    process (clk, res) is
    begin
        if res = '0' then
            q <= '0';
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end architecture behavioral;
```

# Registri edge triggered

---

- ▶ Flip flop di tipo D con **Set e Reset asincroni**
  - ▶ Come prima
  - ▶ A seconda dell'ordine in cui si verificano i valori di *set* e *res* si definisce la priorità tra i due

```
-- Flip flop di tipo D con set e
-- reset async
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (d, clk, set, res : in std_logic;
          q : out std_logic);
end entity D_FF;

architecture behavioral of D_FF is
begin
    process (clk, set, res) is
    begin
        if set = '0' then
            q <= '1';
        elsif res = '0' then
            q <= '0';
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end architecture behavioral;
```

# Registri edge triggered

---

- ▶ Flip flop di tipo D con **reset sincrono**
  - ▶ Questa volta il reset è operativo solo al fronte del clock
  - ▶ Ne controlliamo quindi il valore solo dopo aver verificato la presenza di un fronte
  - ▶ Notare che il processo non è sensibile al segnale *res*
    - ▶ Che succederebbe se lo mettessimo lo stesso nella sensitivity list?

```
-- Flip flop di tipo D con reset sync
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (d, clk, res : in std_logic;
          q : out std_logic);
end entity D_FF;

architecture behavioral of D_FF is
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            if res = '0' then
                q <= '0';
            else
                q <= d;
            end if;
        end if;
    end process;
end architecture behavioral;
```

# Registri edge triggered con load enable

---

## ► Flip flop di tipo D con **controllo di caricamento**

- L'uscita deve essere assegnata solo se il segnale *load* è attivo
- Il segnale *load* è sincrono, quindi non è necessario metterlo in sensitivity list
- Basta verificarne il valore prima di fare l'assegnazione
- Se *load* non è attivo non si fa nulla, quindi si mantiene il valore precedente

```
-- Flip flop di tipo D con load enable
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
    port (d, clk, load : in std_logic;
          q : out std_logic);
end entity D_FF;

architecture behavioral of D_FF is
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            if load = '1' then
                q <= d;
            end if;
        end if;
    end process;
end architecture behavioral;
```

# Registro a più bit

---

## ▶ Esattamente come un registro normale

- ▶ Solo che l'ingresso e l'uscita di dati sono dei vettori
- ▶ Possiamo parametrizzare la dimensione con un generic
- ▶ Notare come si assegna 0 a  $q$  senza saperne la dimensione
- ▶ Per il resto non cambia nulla
- ▶ Facile aggiungere anche il load enable, etc.

```
-- Registro multibit
library ieee;
use ieee.std_logic_1164.all;

entity reg is
    generic (n : natural := 4);
    port (d : in std_logic_vector(n-1 downto 0);
          clk, res : in std_logic;
          q : out std_logic_vector(n-1 downto 0));
end entity reg;

architecture behavioral of reg is
begin
    process (clk, res) is
    begin
        if res = '0' then
            q <= (others => '0');
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end architecture behavioral;
```

# Registro a più bit

---

## ► Registro con

- Reset asincrono attivo basso
- Init sincrono attivo alto
- Load enable attivo alto

```
-- Registro multibit
library ieee;
use ieee.std_logic_1164.all;

entity reg is
    generic (n : natural := 4);
    port (d : in std_logic_vector(n-1 downto 0);
          clk, res, init, load : in std_logic;
          q : out std_logic_vector(n-1 downto 0));
end entity reg;

architecture behavioral of reg is
begin
    process (clk, res) is begin
        if res = '0' then
            q <= (others => '0');
        elsif rising_edge(clk) then
            if init = '1' then
                q <= (others => '0');
            elsif load = '1' then
                q <= d;
            end if;
        end if;
    end process;
end architecture behavioral;
```



# Registro a scorrimento

---

## ► Ingresso seriale ed uscita seriale

- Viene anche chiamato Serial In Serial Out (SISO)
- Abbiamo bisogno di un segnale interno per memorizzare i dati
- Al fronte del clock si shiftano i bit
- Notare che l'ordine con cui si assegna non ha importanza
- Un'equazione al di fuori del processo assegna il valore di  $q$ 
  - Si poteva mettere dentro al processo?
  - Se si, come?

```
-- Registro a scorrimento
library ieee;
use ieee.std_logic_1164.all;

entity siso is
    port (d : in std_logic;
          clk : in std_logic;
          q : out std_logic);
end entity siso;

architecture behavioral of siso is
    signal reg : std_logic_vector(3 downto 0);
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            reg(3) <= reg(2);
            reg(2) <= reg(1);
            reg(1) <= reg(0);
            reg(0) <= d;
        end if;
    end process;

    q <= reg(3);

end architecture behavioral;
```

# Registro a scorrimento

---

```
-- Registro a scorrimento
library ieee;
use ieee.std_logic_1164.all;

entity siso is
  port (d : in std_logic;
        clk : in std_logic;
        q : out std_logic);
end entity siso;

architecture behavioral of siso is
  signal reg : std_logic_vector(3 downto 0);
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      reg(3) <= reg(2);
      reg(2) <= reg(1);
      reg(1) <= reg(0);
      reg(0) <= d;
      q <= reg(2);
    end if;
  end process;

end architecture behavioral;
```

```
-- Registro a scorrimento
library ieee;
use ieee.std_logic_1164.all;

entity siso is
  port (d : in std_logic;
        clk : in std_logic;
        q : out std_logic);
end entity siso;

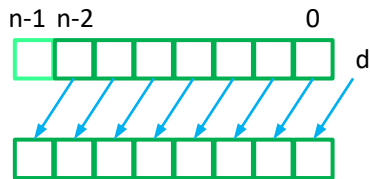
architecture behavioral of siso is
  signal reg : std_logic_vector(2 downto 0);
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      q <= reg(2);
      reg(2) <= reg(1);
      reg(1) <= reg(0);
      reg(0) <= d;
    end if;
  end process;

end architecture behavioral;
```

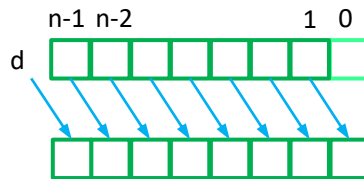
# Registro a scorrimento

## ► Ingresso seriale ed uscita parallela

- Viene anche chiamato Serial In Parallel Out (SIPO)
- Al fronte del clock si assegnano a  $q$  i bit meno significativi di  $q$ , tranne il più significativo, concatenati con  $d$
- L'operatore di concatenazione è &



shift a sinistra



shift a destra

```
-- Registro a scorrimento
library ieee;
use ieee.std_logic_1164.all;

entity sipo is
  generic (n : natural := 8);
  port (d : in std_logic;
        clk, res : in std_logic;
        q : buffer std_logic_vector(n-1 downto 0));
end entity sipo;

architecture behavioral of sipo is
begin
  process (clk, res) is
  begin
    if res = '0' then
      q <= (others => '0');
    elsif rising_edge(clk) then
      q <= q(n-2 downto 0) & d;
    end if;
  end process;
end architecture behavioral;
```

```
-- Per shiftare dall'altra parte
elsif rising_edge(clk) then
  q <= d & q(n-1 downto 1);
```

# Contatore binario

---

- ▶ Realizzato sommando o sottraendo il valore numerico 1
  - ▶ Reset attivo basso
  - ▶ Funzionamento sincrono
  - ▶ Comando che definisce la direzione di conteggio

```
-- Contatore binario
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity conta16 is
    port (res, clk, dir : in std_logic;
          y : buffer unsigned( 3_downto 0 ) );
end entity conta16;

architecture behavioral of conta16 is
begin
    process (res, clk) is
    begin
        if (res = '0') then
            y <= (others => '0');
        elsif rising_edge(clk) then
            if (dir = '1') then
                y <= y + 1;
            else
                y <= y - 1;
            end if;
        end if;
    end process;
end architecture behavioral;
```

# Contatore binario

## ► Contatore con

- Reset attivo basso
- Comando di count enable
- Uscita di terminal count

## ► Nota

- Il conteggio è solo un segnale interno, definito direttamente come tipo numerico
- Il terminal count è determinato in modo combinatorio fuori dal processo tramite una equazione

```
-- Contatore binario
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity conta256 is
    port (res, clk, count_en : in std_logic;
          tc : out std_logic);
end entity conta256;

architecture behavioral of conta256 is

    signal y : unsigned(7 downto 0);

begin

    process (res, clk) is
    begin
        if res = '0' then
            y <= (others => '0');
        elsif rising_edge(clk) then
            if count_en = '1' then
                y <= y + 1;
            end if;
        end if;
    end process;

    tc <= '1' when y = 255 else '0';

end architecture behavioral;
```

# Contatore binario

- ▶ **Terminal count nel processo**
  - ▶ Notare l'uso della variabile
  - ▶ Se y fosse un segnale non sarebbe aggiornato quando si fa il controllo

```
-- Contatore binario
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity conta256 is
    port (res, clk, count_en : in std_logic;
          tc : out std_logic);
end entity conta256;

architecture behavioral of conta256 is
begin

    process (res, clk) is
        variable y : unsigned(7 downto 0);
    begin
        if res = '0' then
            y := (others => '0'); tc <= '0';
        elsif rising_edge(clk) then
            if count_en = '1' then
                y := y + 1;
            end if;
            if y = 255 then tc <= '1';
            else tc <= '0';
            end if;
        end if;
    end process;

end architecture behavioral;
```

- ▶ **Il VHDL offre molte potenzialità**
  - ▶ Ne abbiamo viste per ora solo alcune
  - ▶ Ci forniscono comunque i blocchi base con cui sviluppare sistemi più complessi
- ▶ **Simulazione e sintesi**
  - ▶ Si scrive il modello del sistema in modo che possa essere sintetizzato, oltre ad essere simulato
  - ▶ Il modello del testbench e/o di oggetti già realizzati possono essere scritti solo per la simulazione
- ▶ **Data flow e comportamentale**
  - ▶ Due stili per rappresentare funzioni booleane combinatorie e sequenziali
  - ▶ Da scegliere a seconda della convenienza
  - ▶ Attenzione a garantire il comportamento combinatorio e sequenziale verificando gli assegnamenti lungo tutti i flussi di esecuzione

# Macchine a stati in VHDL

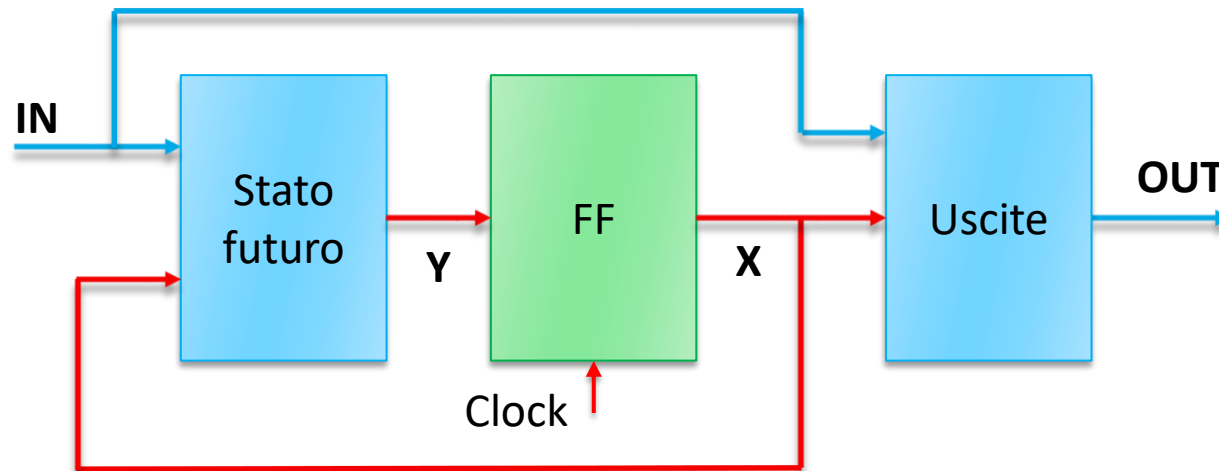
**Come rappresentare un digramma in testo**



# Macchine a stati

---

- ▶ **Le realizziamo seguendo la normale separazione tra parte sequenziale e combinatoria**
  - ▶ La parte sequenziale sarà sostanzialmente simile ad un registro a molti bit
  - ▶ La parte combinatoria può essere spezzata in due, per il calcolo dello stato futuro e delle uscite
  - ▶ Ogni blocco può essere rappresentato da un processo all'interno dell'architettura



# Esempio del derivatore

## ► Parte sequenziale

- Un segnale interno per lo stato presente ed uno per lo stato futuro
- Stato come tipo enumerato, per non decidere subito la codifica
- Al fronte del clock lo stato futuro diventa presente

```
-- Derivatore
library ieee; use ieee.std_logic_1164.all;

entity derivatore is
  port (s, clk, res : in std_logic;
        y : out std_logic);
end entity derivatore;

architecture behavioral of derivatore is
  type stato is (a, b, c, d);
  signal present_state, next_state : stato;
begin
  seq: process (res, clk) is
  begin
    if res = '0' then
      present_state <= a;
    elsif rising_edge(clk) then
      present_state <= next_state;
    end if;
  end process seq;
-- Continua alla slide successiva
```

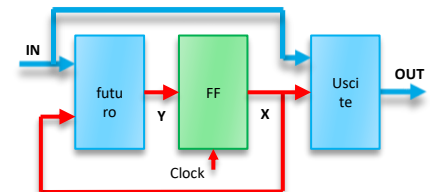
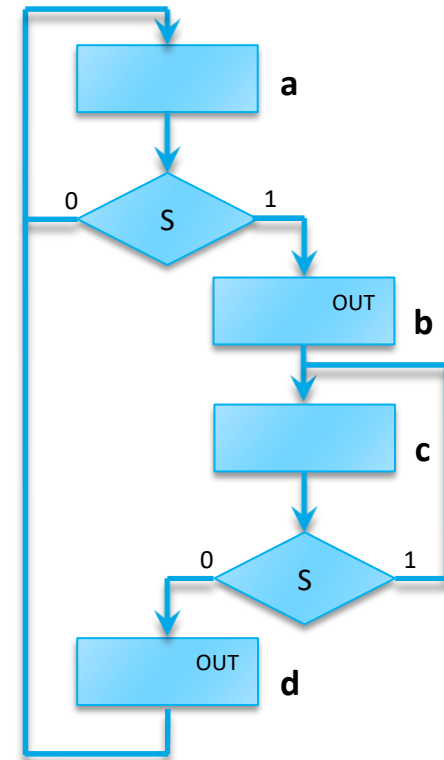
Definizione del tipo di stato

Segnali di stato

Processo per la parte sequenziale, sensibile a reset e clock

Stato iniziale

Aggiornamento stato



# Processi combinatori

-- Continua dalla slide precedente

```
futuro: process (present_state, s) is
begin
  case present_state is
    when a =>
      if s = '0' then
        next_state <= a;
      else
        next_state <= b;
      end if;
    when b =>
      next_state <= c;
    when c =>
      if s = '0' then
        next_state <= d;
      else
        next_state <= c;
      end if;
    when d =>
      next_state <= a;
    end case;
  end process futuro;

  uscite: process (present_state) is
  begin
    y <= '0';
    if present_state = b or present_state = d then
      y <= '1';
    end if;
  end process uscite;
end architecture behavioral;
```

Stato futuro dipende da quello presente e dall'ingresso

Costrutto **case** per i processi (non si può usare quello data flow)

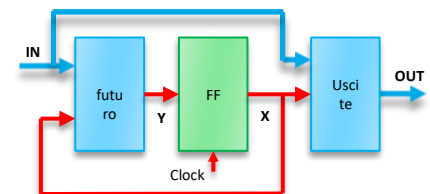
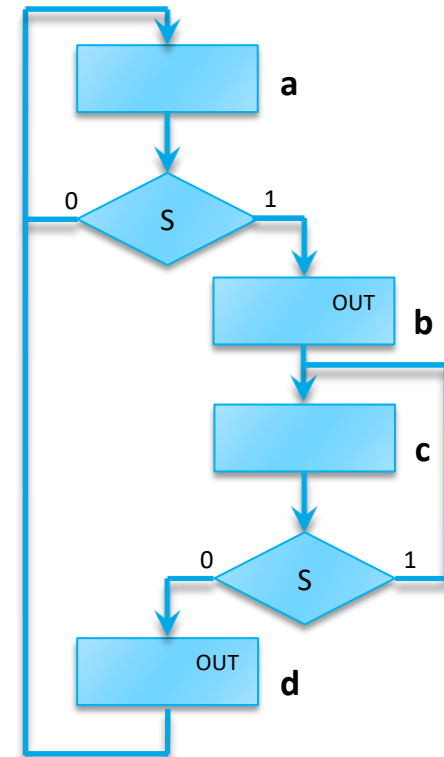
Un caso per ogni stato

Stato futuro dipende dall'ingresso

Si assegna qui, diventerà presente al fronte di clock

Macchina di Moore, uscite dipendono solo dallo stato

Uscita a 1 negli stati **b** e **d**



# Osservazioni

---

- ▶ **Solo il present\_state diventa un insieme di flip flop, mentre il next\_state no**
  - ▶ Il present\_state è pilotato da un processo sequenziale
  - ▶ Ci sono dei casi in cui viene attivato il processo e non gli viene assegnato alcun valore
  - ▶ Il next\_state è pilotato da un processo combinatorio
  - ▶ Per tutti i casi possibili si assegna un valore, quindi non genera un elemento di memoria
  - ▶ Le uscite sono anche pilotate da un processo combinatorio, quindi ovviamente non generano un elemento di memoria
- ▶ **Si possono mettere stato futuro e uscite insieme**
  - ▶ Si fa un solo processo che pilota entrambi
  - ▶ Riduce la quantità di codice da scrivere

# Testbench per circuiti sequenziali

---

- ▶ **Abbiamo già visto come creare testbench tramite processi**
  - ▶ Per i circuiti sequenziali occorre generare anche i segnali di clock e di reset
  - ▶ Il processo del clock non si ferma mai
  - ▶ Quello per il reset termina con l'ultimo **wait**
  - ▶ Nel resto del codice occorre quindi definire i restanti segnali di ingresso dell'oggetto da testare
  - ▶ Ed occorre istanziare l'oggetto da testare

```
-- Testbench sequenziale
library ieee; use ieee.std_logic_1164.all;

entity test_seq is
end entity test_seq;

architecture behavioral of test_seq is
    signal segnali-interni;
begin
    clk: process is
    begin
        clock <= '0';
        wait for 5 ns;
        clock <= '1';
        wait for 5 ns;
    end process clk;

    rst: process is
    begin
        reset <= '1';
        wait for 5 ns;
        reset <= '0';
        wait for 50 ns;
        reset <= '1';
        wait;
    end process rst;

-- Altri segnali

-- Istanziamento entità under test
```

# Derivatore come macchina di Mealy

## ► Gli stati ora sono solo due

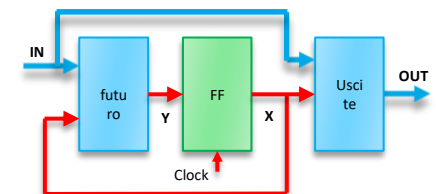
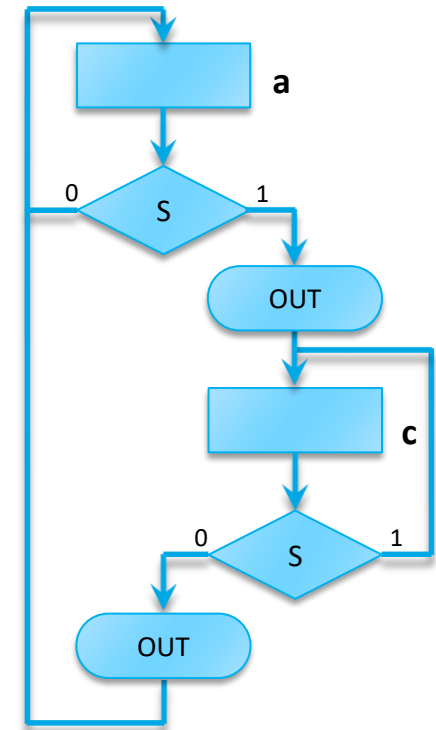
- Ma la parte sequenziale sostanzialmente non cambia
- Infatti è praticamente sempre uguale per tutte le macchine a stati!

```
-- Derivatore
library ieee; use ieee.std_logic_1164.all;

entity derivatore is
  port (s, clk, res : in std_logic;
        y : out std_logic);
end entity derivatore;

architecture behavioral of derivatore is
  type stato is (a, c);
  signal present_state, next_state : stato;
begin
  seq: process (res, clk) is
  begin
    if res = '0' then
      present_state <= a;
    elsif rising_edge(clk) then
      present_state <= next_state;
    end if;
  end process seq;
-- Continua alla slide successiva
```

Definizione del tipo di stato



# Processi combinatori

-- Continua dalla slide precedente

```

futuro: process (present_state, s) is
begin
  next_state <= present_state;
  case present_state is
    when a =>
      if s = '1' then
        next_state <= c;
      end if;
    when c =>
      if s = '0' then
        next_state <= a;
      end if;
  end case;
end process futuro;

uscite: process (present_state, s) is
begin
  y <= '0';
  if (present_state = a and s = '1') or
     (present_state = c and s = '0') then
    y <= '1';
  end if;
end process uscite;

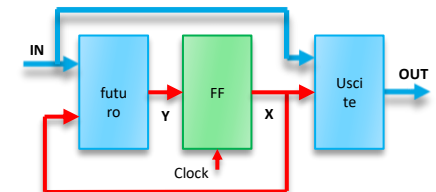
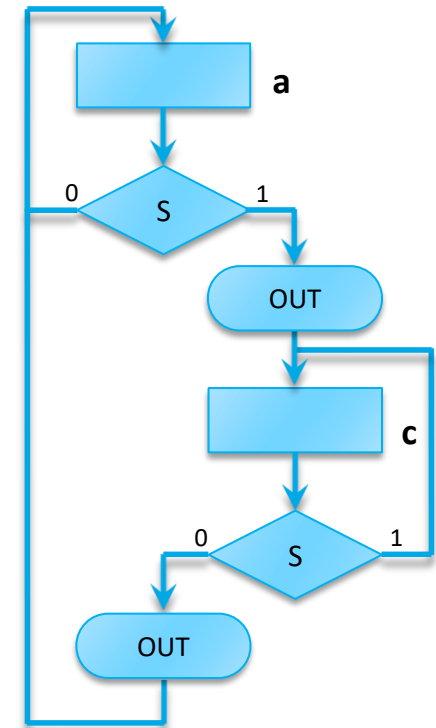
end architecture behavioral;
    
```

Stato futuro dipende da quello presente e dall'ingresso

Utile default: se non diciamo altro stiamo nello stesso stato

Macchina di Mealy, uscite dipendono dallo stato e dagli ingressi

Uscita a 1 nello stato a se s=1 e nello stato c se s=0



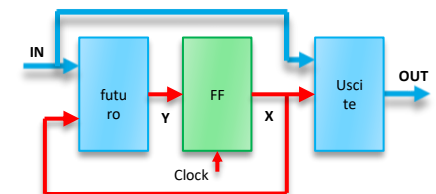
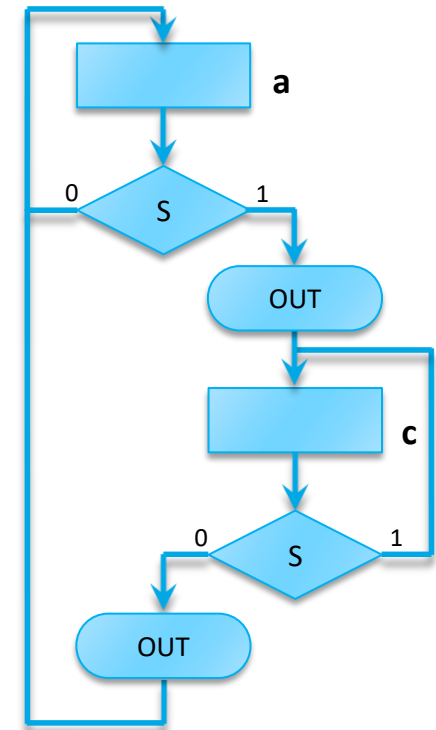
# Un solo processo combinatorio

-- Continua dalla slide precedente

```
comb: process (present_state, s) is
begin
  next_state <= present_state;
  y <= '0';
  if (present_state = a and s = '1') then
    next_state <= c;
    y <= '1';
  elsif (present_state = c and s = '0') then
    next_state <= a;
    y <= '1';
  end if;
end process comb;

end architecture behavioral;
```

Pilota sia lo stato futuro, sia le uscite





# Un solo processo sequenziale/combinatorio

```
-- Derivatore
library ieee; use ieee.std_logic_1164.all;

entity derivatore is
  port (s, clk, res : in std_logic;
        y : out std_logic);
end entity derivatore;

architecture behavioral of derivatore is
  type stato is (a, c);
  signal present_state : stato;
begin
  seq: process (res, clk) is
  begin
    if res = '0' then
      present_state <= a;
    elsif rising_edge(clk) then
      case present_state is
        when a => if s = '1' then
                    present_state <= c;
                  end if;
        when c => if s = '0' then
                    present_state <= a;
                  end if;
      end case;
    end process seq;

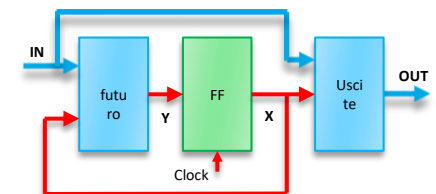
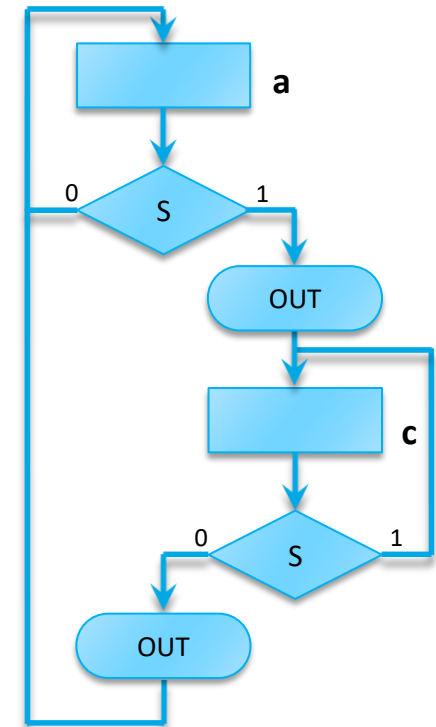
    y <= '1' when (present_state = a and s = '1') or
                  (present_state = c and s = '0')
              else '0';
  end architecture behavioral;
```

Stato futuro implicito, viene comunque sintetizzato

...ma assegnato solo al fronte del clock

present\_state calcolato in termini di se stesso...

uscite calcolate come equazione in questo caso



# Codifica dello stato

- ▶ **Abbiamo lasciato lo stato in forma simbolica**
  - ▶ Questo ci consente di verificare la funzionalità indipendentemente dalla codifica
  - ▶ Alcuni strumenti consentono di effettuare una codifica automatica
- ▶ **E' possibile specificare la codifica in forma esplicita**
  - ▶ Dipende dai tool usati
  - ▶ Si usano attributi

```
attribute enum_encoding : string;  
type stato is (a, b, c, d);  
attribute enum_encoding of stato : type is "00 01 11 10";
```

- ▶ **Oppure si usano costanti esplicite per dare valori agli elementi della enumerazione**
  - ▶ Indipendente dal tool

```
-- Derivatore  
architecture behavioral of derivatore is  
  type stato is std_logic_vector(1 downto 0);  
  signal present_state, next_state : stato;  
  constant a : stato := "00";  
  constant b : stato := "01";  
  constant c : stato := "11";  
  constant d : stato := "10";  
begin  
  seq: process (res, clk) is  
  begin  
    if res = '0' then  
      present_state <= a;  
    elseif rising_edge(clk) then  
      present_state <= next_state;  
    end if;  
  end process seq;  
  
  futuro: process (present_state, s) is  
  begin  
    case present_state is  
      when a => if s = '0' then  
        next_state <= a;  
      else  
        next_state <= b;  
      end if;  
      when b => next_state <= c;  
      when c => if s = '0' then  
        next_state <= d;  
      else  
        next_state <= c;  
    end case;  
  end process futuro;  
end architecture derivatore;
```

# Codifica one-hot

## ► Facile cambiare la codifica

- Basta modificare la definizione delle costanti e del tipo
- Occorre però fare attenzione alle possibili alternative per lo stato presente
- Per la simulazione non ci sono problemi
- Ma il sintetizzatore pensa che `present_state` possa essere per esempio 0110
- Non trovando un ramo di controllo per quel valore, aggiunge un latch spurio per ricordare il valore precedente
- Meglio allora mettere un caso di default al **case**

```
-- Derivatore
architecture behavioral of derivatore is
    type stato is std_logic_vector(3 downto 0);
    signal present_state, next_state : stato;
    constant a : stato := "0001";
    constant b : stato := "0010";
    constant c : stato := "0100";
    constant d : stato := "1000";
begin
    seq: process (res, clk) is
    begin
        ...
    futuro: process (present_state, s) is
    begin
        case present_state is
            when a =>
                if s = '0' then next_state <= a;
                else next_state <= b;
                end if;
            when b =>
                next_state <= c;
            when c =>
                if s = '0' then next_state <= d;
                else next_state <= c;
                end if;
            when d =>
                next_state <= a;
            when others =>
                next_state <= "XXXX";
            end case;
        end process futuro;
        ...
    end
```

# Memorie

---

- ▶ **Le memorie sono incluse in buona parte dei circuiti digitali**
  - ▶ ROM: Read Only Memory
  - ▶ RAM: Random Access Memory
- ▶ **E' utile avere modelli di memorie**
  - ▶ Soprattutto per la simulazione
  - ▶ Normalmente non vengono sintetizzate (verrebbero troppo grosse) ma realizzate a mano o tramite generatori automatici
- ▶ **Funzionamento**
  - ▶ Ad ogni indirizzo corrisponde un valore
  - ▶ In pratica si tratta di un array

# Modello per una ROM

---

- ▶ **Scriviamo i valori predefiniti in un array**
  - ▶ Per chiarezza ne definiamo il tipo
  - ▶ Usiamo un integer per indicizzare l'array (tanto non va sintetizzata)
  - ▶ Possibile specificare un tempo di ritardo
- ▶ **E' come fare una tabella della verità**
  - ▶ Possiamo usare una ROM per implementare una qualunque funzione combinatoria
  - ▶ Una EEPROM spesso usata per questo scopo

```
-- Modello di una ROM
library ieee;
use ieee.std_logic_1164.all;

entity ROM is
    port (address : in integer range (0 to 15);
          data : out std_logic_vector (7 downto 0));
end entity ROM;

architecture dataflow of ROM is
    type rom_array is array (0 to 15) of
        std_logic_vector(7 downto 0);
    constant rom : rom_array := ( "01010101",
                                    "10101010",
                                    "00000111",
                                    "01011101",
                                    "01110101",
                                    ...,
                                    "11111111",
                                    "00110111" );

    begin
        data <= rom(address) after 10 ns;
    end architecture dataflow;
```

# Modello di una RAM statica

- ▶ **I dati possono essere letti e scritti**
  - ▶ Simile alla ROM
  - ▶ Usiamo il bus dati per leggere e scrivere, quindi **inout**
  - ▶ Tre segnali di controllo **attivi bassi**
  - ▶ **cs**: chip select per selezionare la memoria
  - ▶ **oe**: output enable per leggere
  - ▶ **we**: write enable per scrivere
- ▶ **Usiamo un processo sequenziale**
  - ▶ Di default le uscite sono in alta impedenza
  - ▶ Vengono assegnate un valore solo se **cs** e **oe** sono entrambi attivi
  - ▶ Quando si scrive, il valore di **data** viene assegnato da qualche altro componente
  - ▶ La funzione resolve fa override dell'assegnazione a Z

```
-- Modello di una RAM statica
library ieee;
use ieee.std_logic_1164.all;

entity RAM16x8 is
    port (address : in integer range (0 to 15);
          data : inout std_logic_vector(7 downto 0)
          cs, oe, we : in std_logic);
end entity RAM16x8;

architecture behavioral of RAM16x8 is
begin
    process (address, cs, we, oe, data) is
        type ram_array is array (0 to 15) of
            std_logic_vector(7 downto 0);
        variable mem : ram_array;
    begin
        data <= (others => 'Z');
        if cs = '0' then
            if oe = '0' then
                data <= mem(address);
            elsif we = '0' then
                mem(address) := data;
            end if;
        end if;
    end process;
end architecture behavioral;
```

# Funzioni

- ▶ **E' possibile, come in altri linguaggi, fattorizzare il codice**
  - ▶ Una funzione prende dei parametri in ingresso e restituisce un valore in uscita
  - ▶ Utile per definire essenzialmente nuovi operatori
  - ▶ Fa uso al suo interno degli stessi statement sequenziali usati in un processo
  - ▶ A differenza di un processo viene attivata alla chiamata, invece che a causa del cambiamento di un segnale
  - ▶ I parametri formali sono sempre ingressi, e non sono modificabili
  - ▶ Una funzione può usare variabili, costanti, altre funzioni, etc.
  - ▶ Per ritornare un valore si usa **return** (valore);

```
-- Sintassi funzione
function nome-funzione (
  nome-parametro : in tipo-parametro;
  . . . . .
  nome-parametro : in tipo-parametro)
  return tipo-risultato is
  dichiarazioni-di-tipo
  dichiarazioni-di-costanti
  dichiarazioni-di-variabili
  dichiarazioni-di-funzioni
begin
  statement-sequenziale;
  . . . . .
  statement-sequenziale;
end nome-funzione;
```

Parametri formali, sempre **in**,  
non modificabili all'interno  
della funzione

Tipo del valore di ritorno

# Esempio di funzioni

## ► Definizione di una funzione add

```
-- Esempio funzione
library ieee; use ieee.std_logic_1164.all;

entity s is
  port (p1, q1 : in std_logic_vector (7 downto 0);
        p2, q2 : in std_logic_vector (3 downto 0);
        y1 : out std_logic_vector (7 downto 0);
        y2 : out std_logic_vector (3 downto 0));
end entity s;

architecture beh of s is
  function add (a, b : in std_logic_vector)
    return std_logic_vector is
    variable s : std_logic_vector(a'range);
    variable carry : std_logic;
  begin
    carry := '0';
    for i in a'low to a'high loop
      s(i) := a(i) xor b(i) xor carry;
      carry := (a(i) and b(i)) or
               (carry and a(i)) or
               (carry and b(i));
    end loop;
    return s;
  end function add;
begin
  y1 <= add(p1, q1);
  y2 <= add(p2, q2);
end architecture beh;
```

Usiamo variabili per fare una somma bit a bit iterativa

Instanziazione della funzione con parametri differenti



# Procedure

- ▶ **Simili alle funzioni, ma possono ritornare più di un valore**
  - ▶ I parametri formali possono essere sia di tipo **in** sia di tipo **out**
  - ▶ I parametri formali possono essere segnali o variabili (di default sono variabili)
  - ▶ La classe (variabile o segnale) dei parametri effettivi deve essere la stessa di quelli formali
  - ▶ Per usare segnali  
a, b : **in signal** std\_logic\_vector;

```
-- Esempio procedura
procedure somma (a, b : in std_logic_vector;
                 y : out std_logic_vector;
                 cout : out std_logic) is
    variable carry : std_logic;
begin
    carry := '0';
    for i in a'low to a'high loop
        y(i) := a(i) xor b(i) xor carry;
        carry := (a(i) and b(i)) or (carry and a(i))
                or (carry and b(i));
    end loop;
    cout := carry;
end procedure somma;
```

Ingressi

Valori di ritorno

# Configurazioni

---

- ▶ **Una specifica VHDL è costituita da dichiarazioni di entità e di architetture**
  - ▶ Ogni volta che viene utilizzata una entità si può scegliere una specifica architettura
  - ▶ Le configurazioni consentono di definire quali architetture utilizzare per il sistema
  - ▶ Facile cambiare architetture scambiando le configurazioni
- ▶ **Accesso a file**
  - ▶ Come nei normali linguaggi di programmazione è possibile scrivere e leggere i file
  - ▶ Utile solo per la simulazione
  - ▶ Si possono usare file per leggere valori per i segnali di ingresso o per scriverci valori di uscita

- ▶ **Abbiamo gli strumenti per realizzare sistemi complessi**
  - ▶ Il VHDL ci consente di esprimere il design in maniera semplice
  - ▶ Ci solleva da un gran numero di considerazioni implementative
- ▶ **Fondamentale la metodologia**
  - ▶ Separazione tra funzione e tempistiche tramite progetto sincrono
  - ▶ Separazione tra parte sequenziale e combinatoria
  - ▶ Separazione tra data path e unità di controllo
  - ▶ Suddivisione in componenti e macchine correlate
- ▶ **Il VHDL ci supporta con metodi di verifica e sintesi**
  - ▶ La verifica è necessaria per individuare errori presto durante il ciclo di progetto
  - ▶ E' una fase spesso più onerosa del progetto stesso
  - ▶ Utile verificare bene i componenti individualmente, e poi concentrarsi solo sulla loro interazione
  - ▶ La sintesi fornisce direttamente il circuito con diverse modalità realizzative

# Gestione delle uscite in VHDL

---

- ▶ **In VHDL, le uscite sono gestite da un apposito processo**
  - ▶ Abbiamo anche detto di non includerle nel processo sequenziale, altrimenti diventano dei registri
  - ▶ Ed entrano a far parte dello stato

```
-- Processo per il calcolo delle uscite

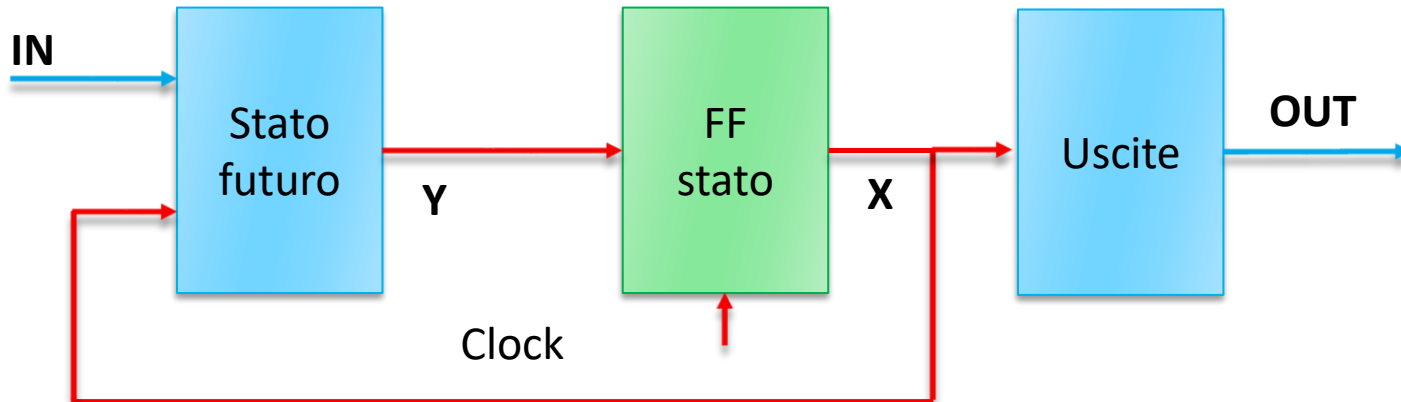
uscita: process (present_state) is
begin
  case present_state is
    when b =>
      y <= '1';
    when d =>
      y <= '1';
    when others =>
      y <= '0';
  end case;
end process uscita;

end architecture behavioral;
```

# Gestione delle uscite

---

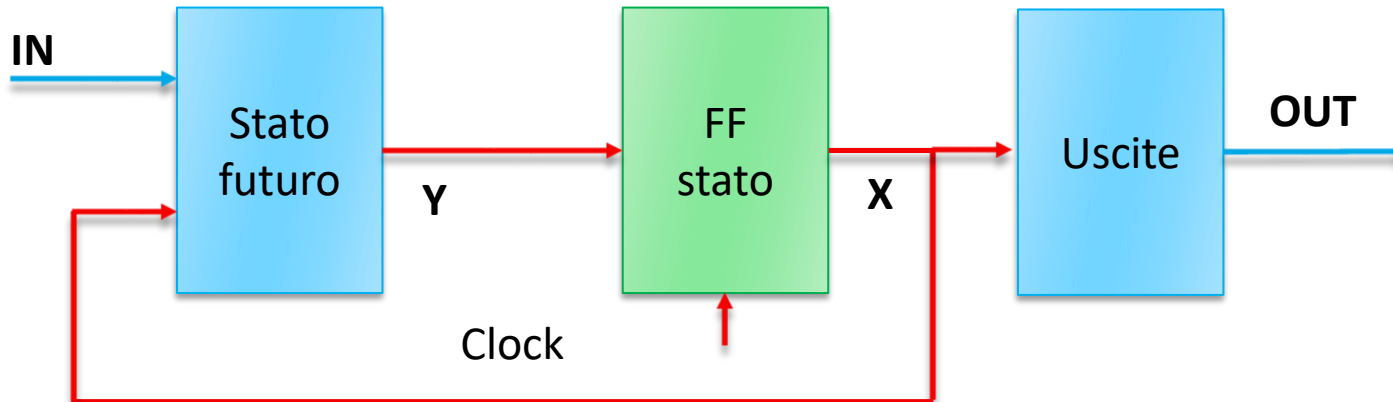
- ▶ In una macchina di Moore, le uscite dipendono solo dallo stato
  - ▶ Uscite calcolate con una rete combinatoria
  - ▶ La rete combinatoria potrebbe impiegare tempo a calcolare le uscite, che si stabilizzano durante il ciclo di clock
  - ▶ Le uscite potrebbero presentare dei glitch, magari indesiderati



# Gestione delle uscite

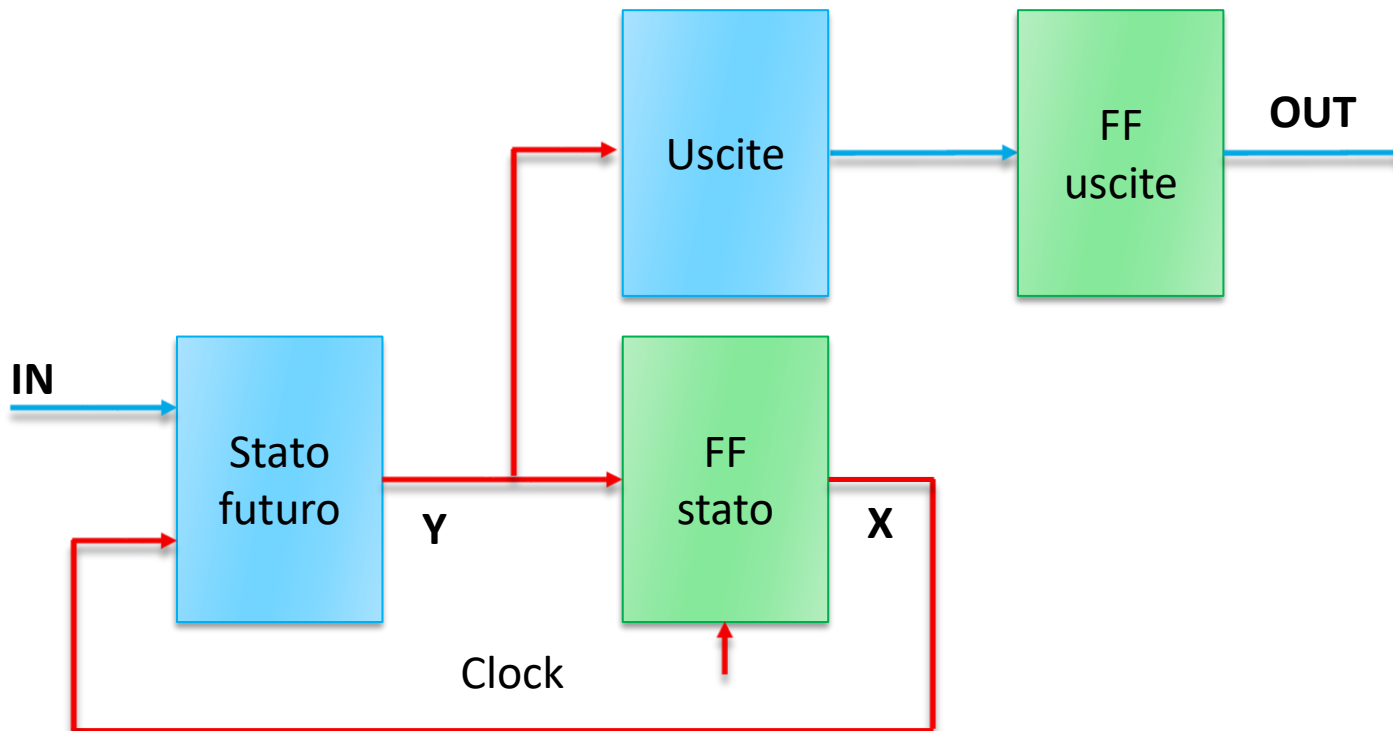
---

- ▶ Potrebbe essere utile fare in modo che le uscite corrispondano alle uscite di qualche flip flop
  - ▶ In questo modo si aggiornano subito dopo il fronte
  - ▶ E non hanno glitch
  - ▶ In certi casi occorre però aggiungere qualche variabile di stato
  - ▶ Occorre sapere il valore delle uscite *prima* dell'arrivo del fronte
  - ▶ Questo è semplice da fare nel caso delle macchine di Moore



# Uscite registrate

- ▶ **Anticipiamo il calcolo delle uscite**
  - ▶ Invece di usare lo stato presente, usiamo lo stato futuro, che conosciamo già il ciclo di clock prima
  - ▶ A questo punto basta registrare le uscite con dei flip flop



# Uscite registrate in VHDL

---

## ► Occorre gestirle come uno stato

- Ogni uscita avrà la versione presente (e.g., y) e la versione futura (e.g., next\_y)
- Il calcolo delle uscite si basa sul next\_state, e aggiorna next\_y
- Un processo sequenziale aggiorna le uscite effettive

```
-- Calcolo uscite

next_uscite: process (next_state) is
begin
  case next_state is
    when b =>
      next_y <= '1';
    when d =>
      next_y <= '1';
    when others =>
      next_y <= '0';
  end case;
end process next_uscite;
```

```
-- Registrazione uscite

uscite: process (res, clock) is
begin
  if res = '0' then
    y <= '0';
  elsif rising_edge( clock ) then
    y <= next_y;
  end if;
end process uscite;
```



# Uscite registrate in VHDL

---

- ▶ **Occorre gestirle come uno stato**
  - ▶ Volendo si può utilizzare un processo solo
  - ▶ Notate che al reset le uscite vanno messe al valore che assumono nello stato iniziale

```
-- Registrazione uscite

uscita: process (res, clock) is
begin
    if res = '0' then
        y <= '0';
    elsif rising_edge( clock ) then
        case next_state is
            when b =>
                y <= '1';
            when d =>
                y <= '1';
            when others =>
                y <= '0';
        end case;
    end if;
end process uscita;
```

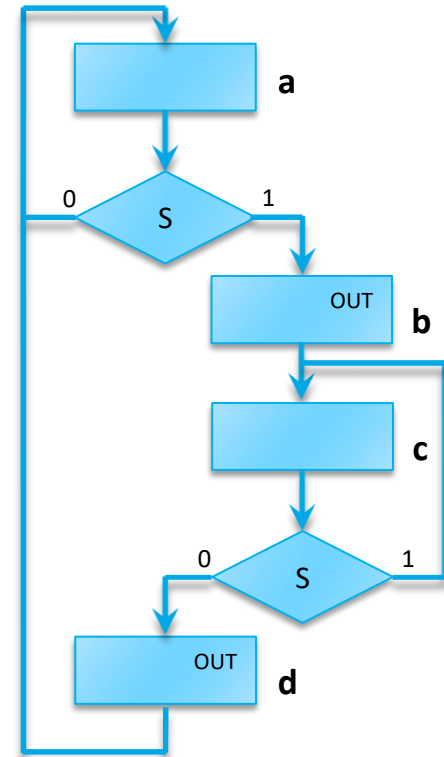
# Oppure si usa un singolo processo

```
-- Derivatore
library ieee; use ieee.std_logic_1164.all;

entity derivatore is
  port (s, clk, res : in std_logic;
        y : out std_logic);
end entity derivatore;

architecture behavioral of derivatore is
  type stato is (a, b, c, d);
  signal present_state : stato;
begin
  seq: process (res, clk) is
  begin
    if res = '0' then
      present_state <= a;
      y <= '0';
    elsif rising_edge(clk) then
      case present_state is
        when a => if s = '1' then
                     present_state <= b;
                     y <= '1';
                   end if;
        when b => present_state <= c; y <= '0';
        when c => if s = '0' then
                     present_state <= d;
                     y <= '1';
                   end if;
        when d => present_state <= a; y <= '0';
      end case;
    end if;
  end process seq;
```

Attenzione: le uscite si riferiscono allo stato di destinazione!

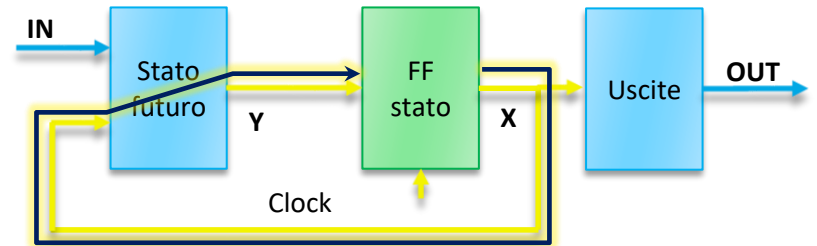


Soluzione molto utilizzata per le macchine di Moore

# Osservazioni

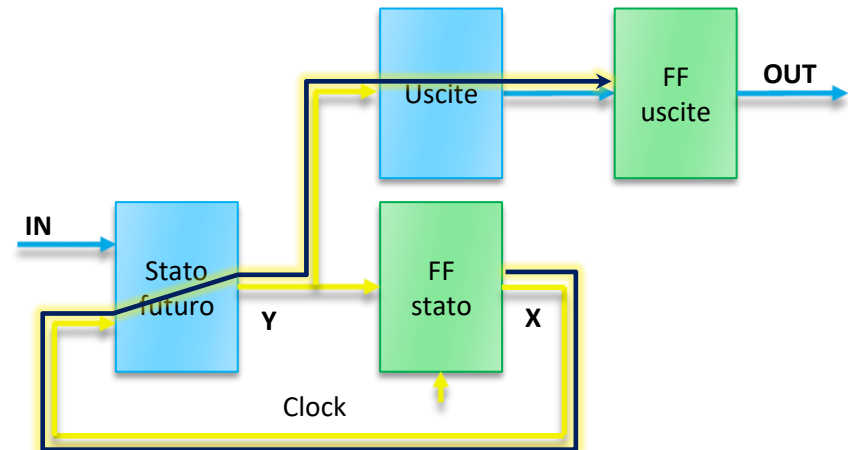
## ► Vantaggi

- Le uscite sono disponibili sin dall'inizio del ciclo di clock
- Le uscite rimangono stabili per tutto il ciclo di clock, e non hanno glitch



## ► Svantaggi

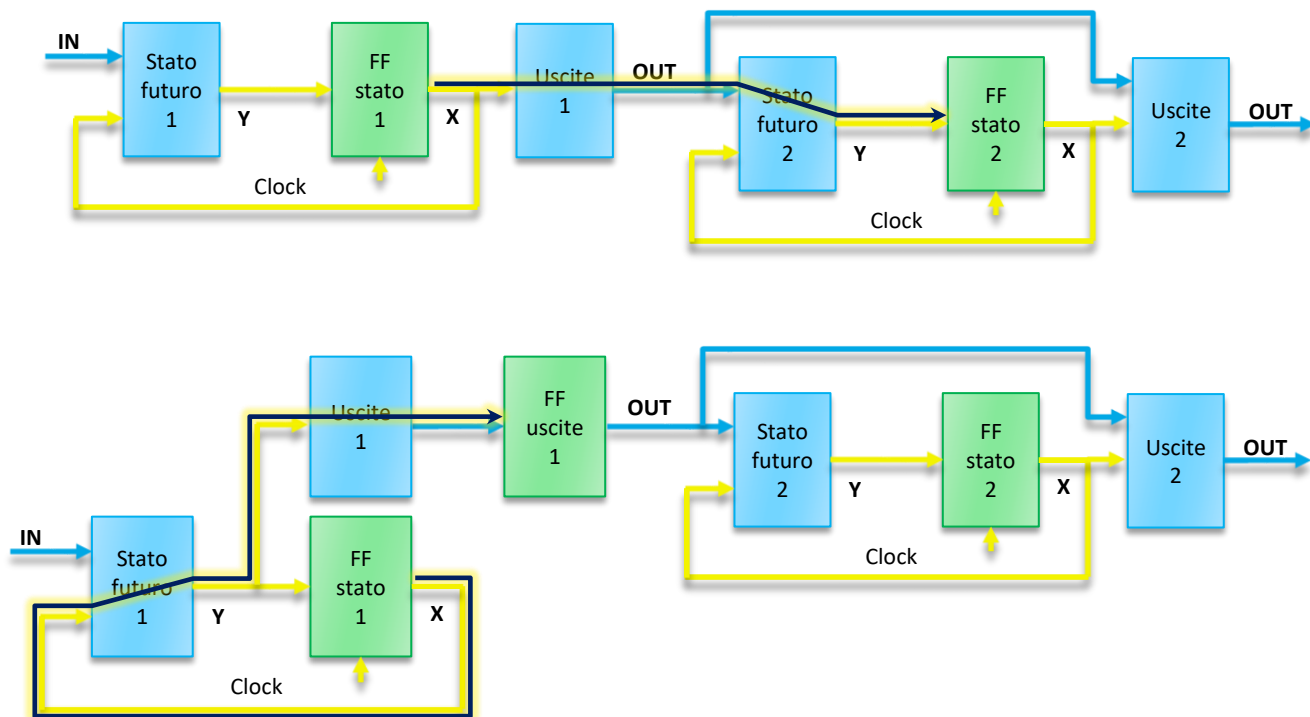
- Il circuito viene più grosso perché devo aggiungere dei FF
- Si rischia di dover allungare il ciclo di clock, perché attraverso due circuiti combinatori



# Ritardi tra macchine comunicanti

## ► Quale è meglio?

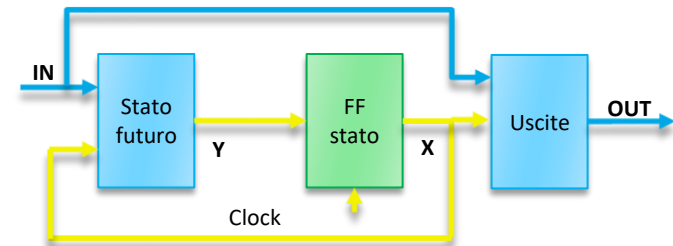
- Dipende dai valori dei ritardi dei vari blocchi combinatori



# Macchine di Mealy

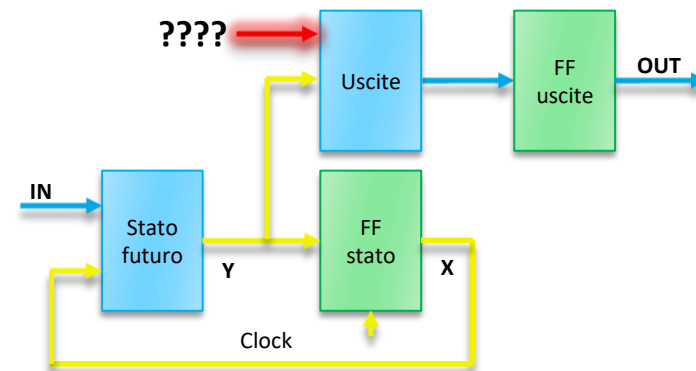
## ▶ La procedura funziona con una macchina di Moore

- ▶ Abbiamo spostato il blocco delle uscite da dopo i flip flop a prima dei flip flop
- ▶ Calcoliamo il valore delle uscite nel ciclo precedente usando lo stato futuro
- ▶ Poi lo ritardiamo con i FF per porlo in uscita nel ciclo giusto



## ▶ Non funziona per una di Mealy!!

- ▶ Per ogni ciclo, le uscite dipendono dal valore degli ingressi **nello stesso ciclo**
- ▶ Dovremmo sapere il valore degli ingressi con un ciclo di anticipo
- ▶ Ma questa informazione non è disponibile



# Macchine a stati correlate

---

- ▶ **Un sistema complesso non può essere progettato come una unica entità**
  - ▶ Lo abbiamo già visto più volte
  - ▶ Una sola macchina a stati avrebbe troppi stati per poter essere gestita in modo sensato
- ▶ **Normalmente diverse macchine a stati interagiscono tra loro per fornire la funzionalità richiesta**
  - ▶ Le singole macchine a stati possono avere uno spazio degli stati molto più ristretto
  - ▶ Supponiamo di avere due macchine a stati ognuna, con  $n$  stati
  - ▶ Ognuna può trovarsi in uno qualunque dei suoi stati (non è esattamente vero...)
  - ▶ Quindi, complessivamente, il sistema avrà  $n^2$  stati
  - ▶ Il partizionamento ci permette di trattare solo  $2n$  stati invece di  $n^2$
  - ▶ Occorre però tener conto dell'interazione

# Esempio

---

- ▶ **Supponiamo di voler progettare un sistema che sia in grado di ricevere ed eseguire vari comandi**
  - ▶ Un comando causa la trasmissione di un dato lungo una linea seriale
  - ▶ Un altro permette di calcolare una media mobile dei dati ricevuti
  - ▶ E così via
- ▶ **I comandi sono forniti su una linea seriale**
  - ▶ La linea è al valore 0 quando in idle, mentre un 1 introduce un comando (bit di start)
  - ▶ Per esempio la sequenza 1101 indica il comando per la linea seriale
  - ▶ La sequenza 1011 indica il comando per la media mobile

# Realizzazione

---

## ► Le specifiche sono un po' vaghe

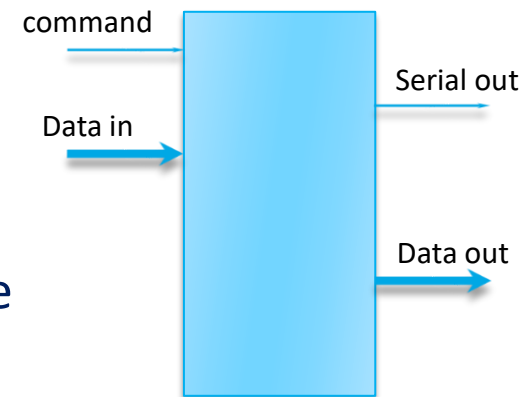
- Andiamo in modo incrementale per gradi
- L'oggetto avrà un ingresso seriale con cui riceve i comandi
- Avrà poi un ingresso parallelo con i dati
- Una uscita seriale in cui mandare il dato quando richiesto
- E magari una uscita parallela con la media mobile

## ► Occorre riconoscere i comandi

- Possiamo usare il nostro riconoscitore di sequenze
- Una volta riconosciuta una sequenza si esegue il comando

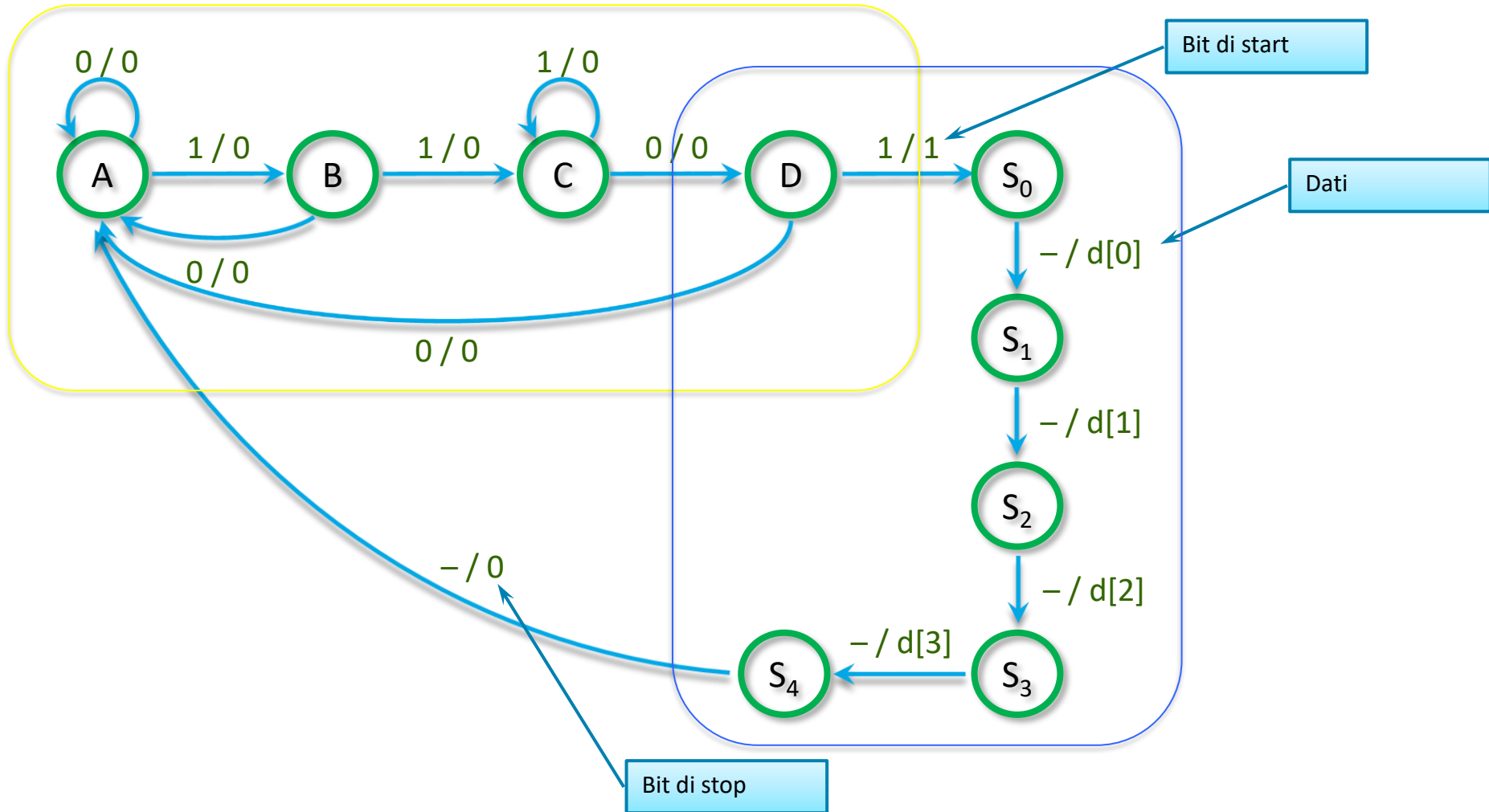
## ► Linea seriale

- Supponiamo di mandare i dati con un primo bit di start a 1, seguito dal resto dei bit, ed almeno un bit a 0 per lo stop





# Gestione della linea seriale



# Comandi ravvicinati

---

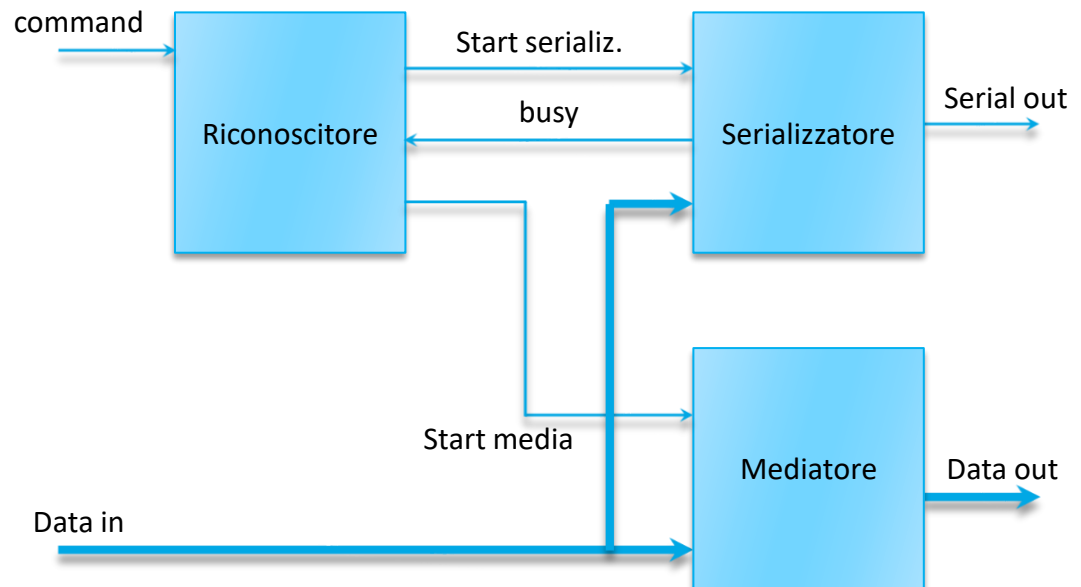
- ▶ **E se arriva un altro comando mentre ne eseguiamo uno?**
  - ▶ La nostra macchina a stati non guarda la linea comandi mentre esegue la serializzazione
  - ▶ Nulla vieta però di farlo: occorre riprodurre il riconoscitore mentre si esegue la serializzazione
  - ▶ Ma quanto si complica la gestione?
  - ▶ Dobbiamo tenere conto di quello che si vede sulla linea comandi, e allo stesso tempo contare i bit che serializziamo
- ▶ **Meglio separare le due funzioni**
  - ▶ Un blocco riconosce, l'altro esegue
  - ▶ Occorre prevedere una sincronizzazione

# Partizionamento

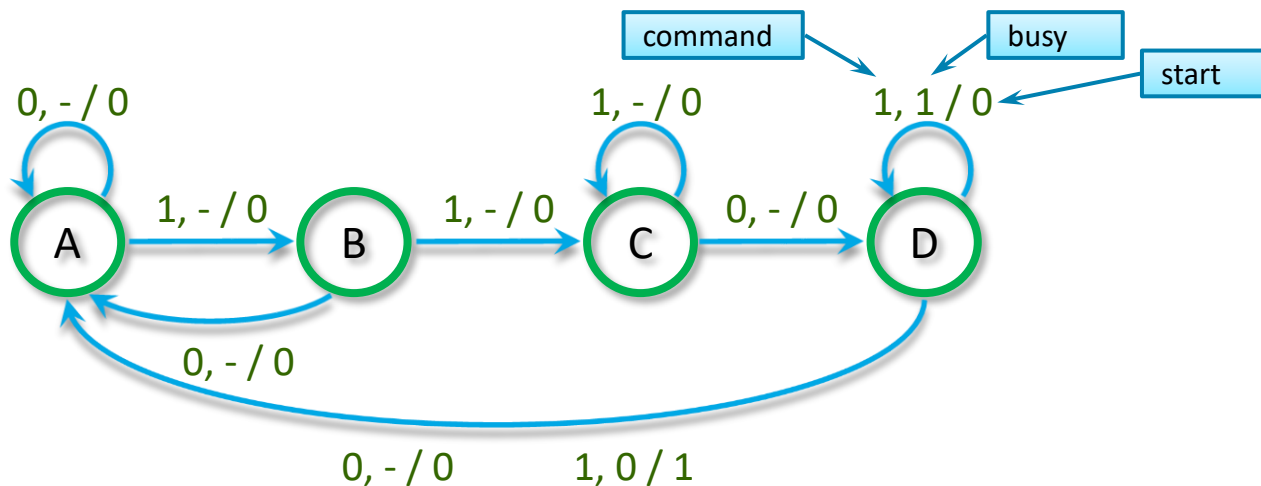
---

## ► Scomponiamo il sistema in diversi blocchi

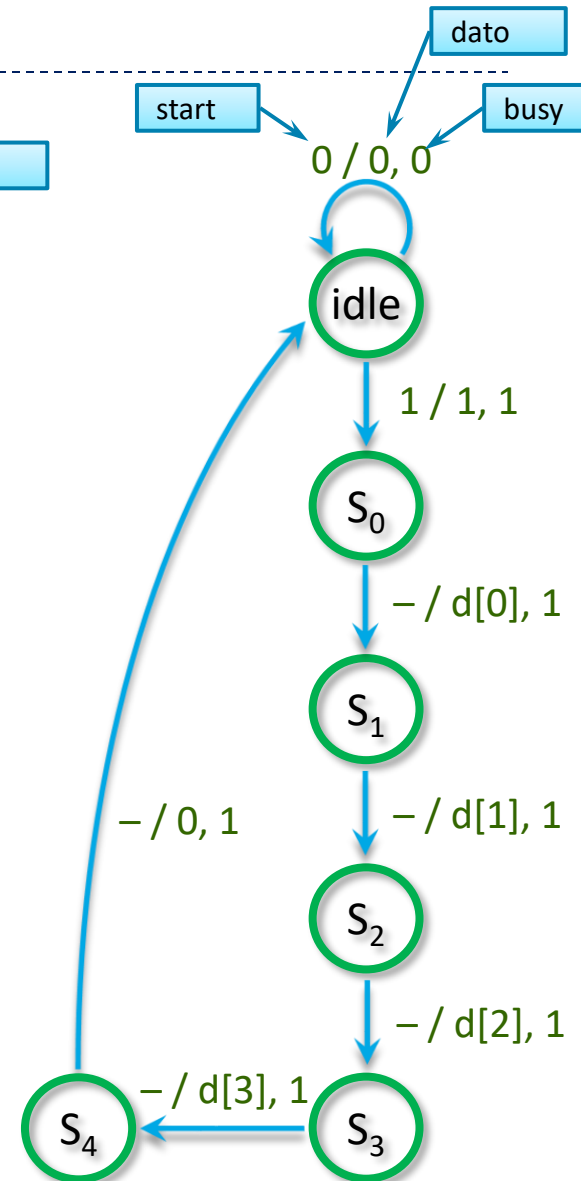
- Uno si occupa di riconoscere il comando
- Un altro di eseguire la serializzazione
- Un terzo di calcolare la media mobile
- Possono lavorare in parallelo



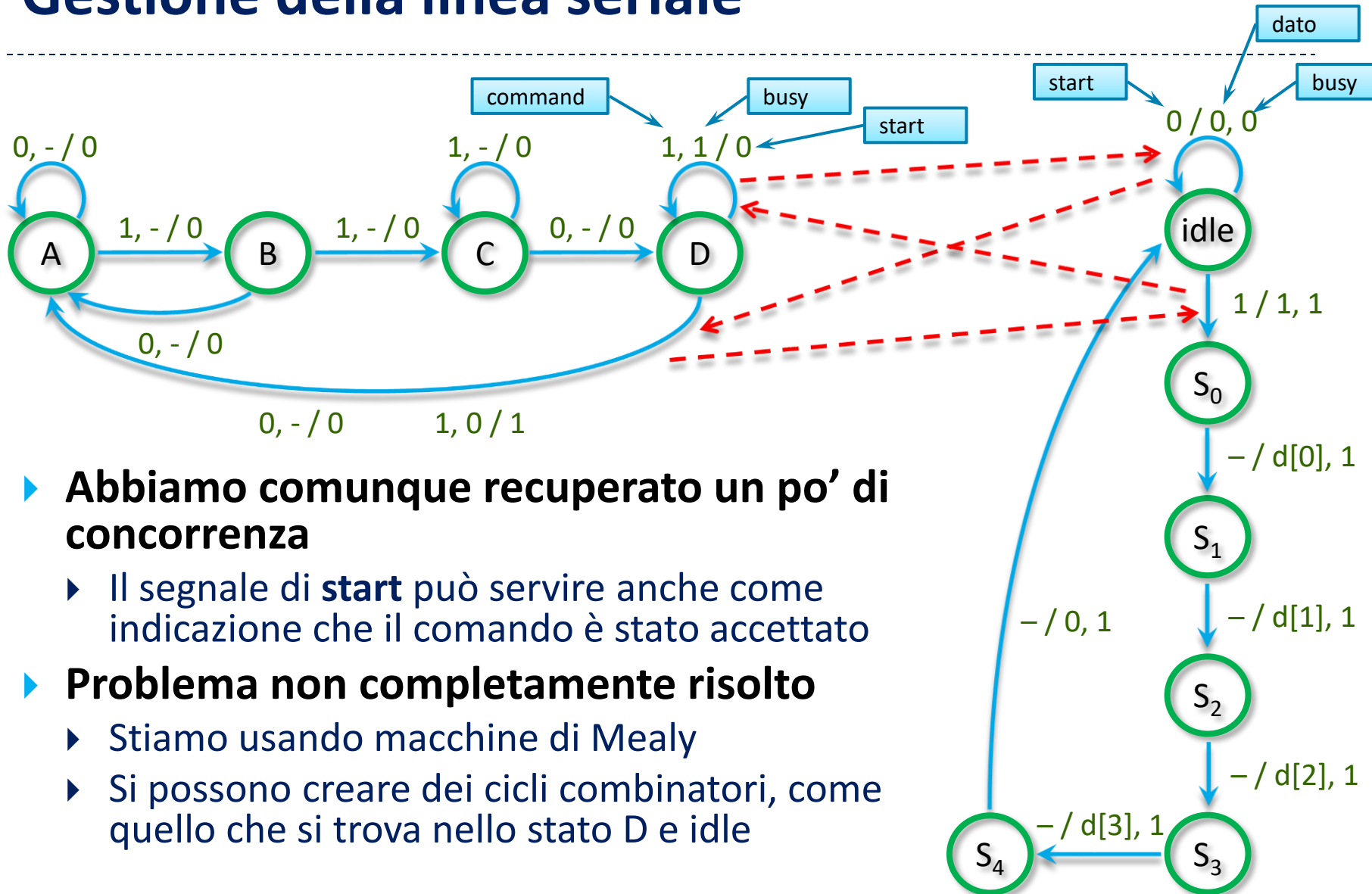
# Gestione della linea seriale



- ▶ **Abbiamo comunque recuperato un po' di concorrenza**
  - ▶ Il segnale di **start** può servire anche come indicazione che il comando è stato accettato
- ▶ **Problema non completamente risolto**
  - ▶ Stiamo usando macchine di Mealy
  - ▶ Si possono creare dei cicli combinatori, come quello che si trova nello stato D e idle

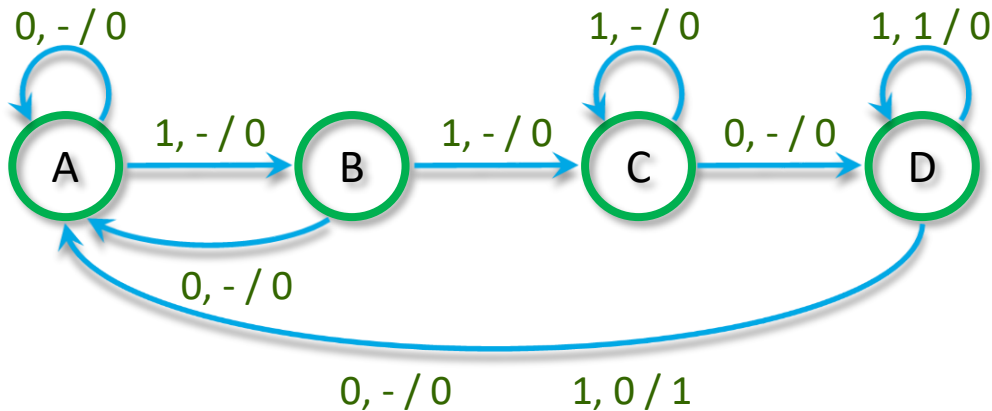


# Gestione della linea seriale

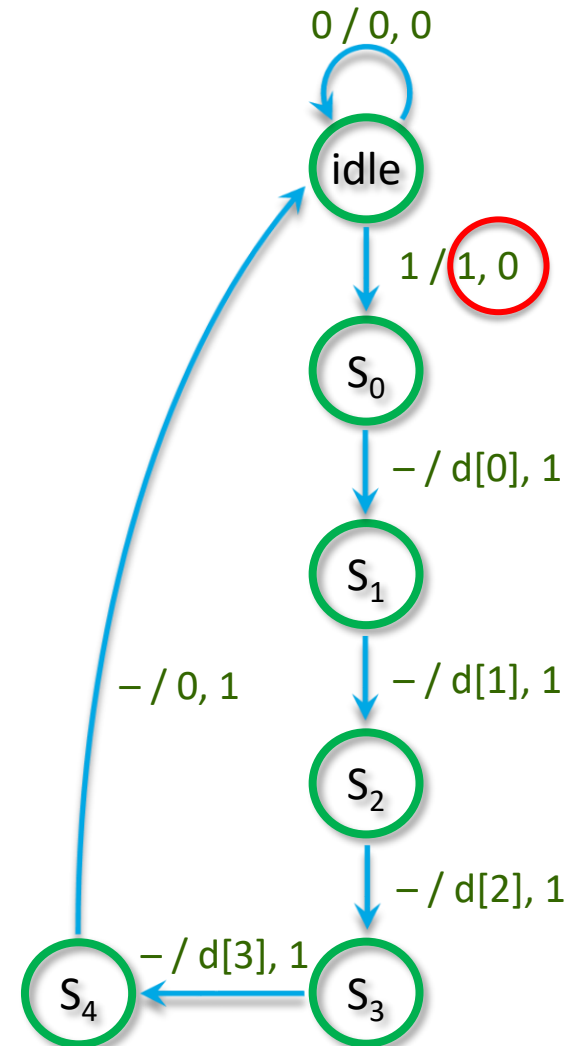


- ▶ **Abbiamo comunque recuperato un po' di concorrenza**
  - ▶ Il segnale di **start** può servire anche come indicazione che il comando è stato accettato
- ▶ **Problema non completamente risolto**
  - ▶ Stiamo usando macchine di Mealy
  - ▶ Si possono creare dei cicli combinatori, come quello che si trova nello stato D e idle

# Soluzione



- ▶ Il segnale *busy* diventa dipendente solo dallo stato
  - ▶ In questo modo si frappa almeno un registro e non c'è più il problema
- ▶ Si può ancora pensare di memorizzare il dato
  - ▶ In questo modo si svincola il resto del sistema che può fornire altri dati in concorrenza
  - ▶ Ma comunque deve aspettare se si sta serializzando quello di prima



# Realizzazione VHDL

---

- ▶ **Si fanno due macchine a stati separate**
  - ▶ Possono essere due entità distinte
    - ▶ Comunicano attraverso segnali di interfaccia
  - ▶ Oppure possono essere due processi distinti della medesima entità
    - ▶ Comunicano tramite segnali interni
    - ▶ Occorrono anche due segnali di stato separati
- ▶ **Stati totali**
  - ▶ Ci sono 4 e 6 stati nelle due macchine
  - ▶ Totale 24 stati nel sistema
- ▶ **Per casa, aggiungere il calcolo della media mobile**
  - ▶ Può essere utile suddividerlo in data path e controllo
  - ▶ Fossero comunque pure solo due stati, si ha in totale una cinquantina
  - ▶ Fondamentale in questo caso la divisione in componenti

- ▶ **Aumentiamo il livello di astrazione**
  - ▶ Passiamo ad una descrizione puramente funzionale del sistema per non limitare artificialmente lo spazio delle possibili implementazioni
  - ▶ Strumenti automatici consentono di raffinare il modello in modo efficiente
- ▶ **Linguaggi usati per descrivere i circuiti**
  - ▶ Possono esprimere facilmente il comportamento tramite equazioni booleane
  - ▶ Sono utili come standard per permettere a progettisti diversi di dialogare in maniera formale
- ▶ **Analisi basata su semantica di simulazione ad eventi discreti**
  - ▶ Il simulatore propaga gli eventi attraverso le equazioni
  - ▶ Mantiene una coda ordinata cronologicamente degli eventi da trattare
  - ▶ E' equivalente a costruire un diagramma temporale