

# Fondamenti di VHDL

# Sommario

---

- VHDL: premessa e introduzione
- Modellizzazione
- Sintassi
- Classi di Oggetti
- Tipi di Dati e Operatori
- Package e Librerie
- Processi
- Esempi di codice VHDL
- VHDL Testbenches

# Premessa

---

- ⊕ La tecnologia microelettronica si è evoluta molto rapidamente negli ultimi decenni
  - ⊕ La complessità dei circuiti è sempre maggiore
  - ⊕ Le prestazioni devono essere sempre migliori
  - ⊕ I costi devono essere continuamente ridotti
  - ⊕ L'affidabilità non deve essere penalizzata
- ⊕ La rapida evoluzione delle tecnologie favorisce l'obsolescenza dei circuiti
  - ⊕ Il time-to-market deve essere ridotto al minimo
  - ⊕ I tempi di sviluppo devono essere brevi
- ⊕ L'uso di strumenti CAD aiuta a soddisfare questi vincoli

# Perché c' è bisogno di un HDL?

---

- Perché c'è bisogno di un linguaggio di descrizione dell'hardware (HDL - Hardware Description Language)?
- I linguaggi di programmazione imperativi (es. C/C++) non supportano a pieno la specifica di diverse caratteristiche fondamentali dei componenti hardware:
  - ▶ Interfacce input/output/inout
  - ▶ Tipi di dati e specifica dell'ampiezza dei dati
  - ▶ Temporizzazione
  - ▶ Concorrenza
  - ▶ Sincronizzazione

# VHDL

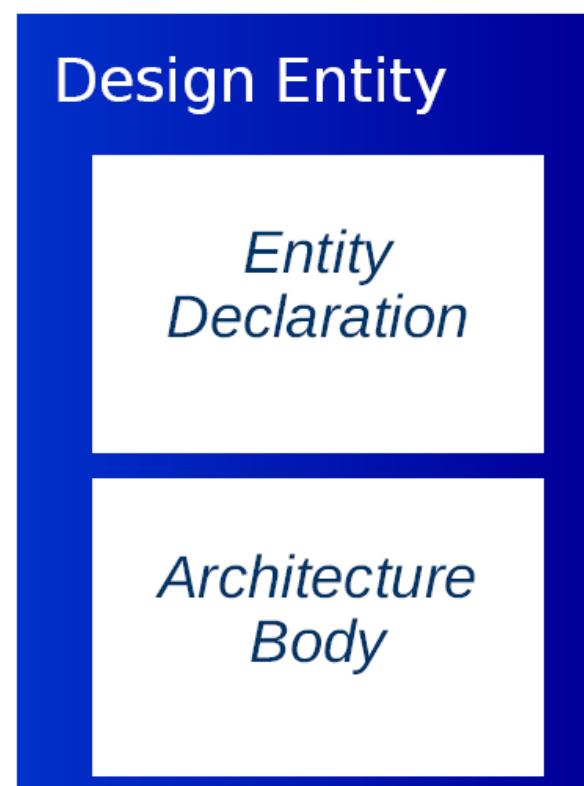
---

- Il **VHDL** è un linguaggio di descrizione dell'hardware
  - ▶ VHDL sta per VHSIC-HDL cioè Very High Speed Integrated Circuit – Hardware Description Language
  - ▶ Il VHDL è stato definito negli anni '80 dal dipartimento della difesa USA
  - ▶ L'ultima versione pubblica risale al 2008 (IEEE Std 1076-2008)
- Alternativa al linguaggio **Verilog**
- Viene utilizzato per: **documentare, descrivere, progettare, simulare e sintetizzare** circuiti e sistemi logici
  - ▶ Possibilità di descrivere il sistema a diversi livelli di astrazione
  - ▶ Progressivo raffinamento della specifica

# Il processo di modellizzazione

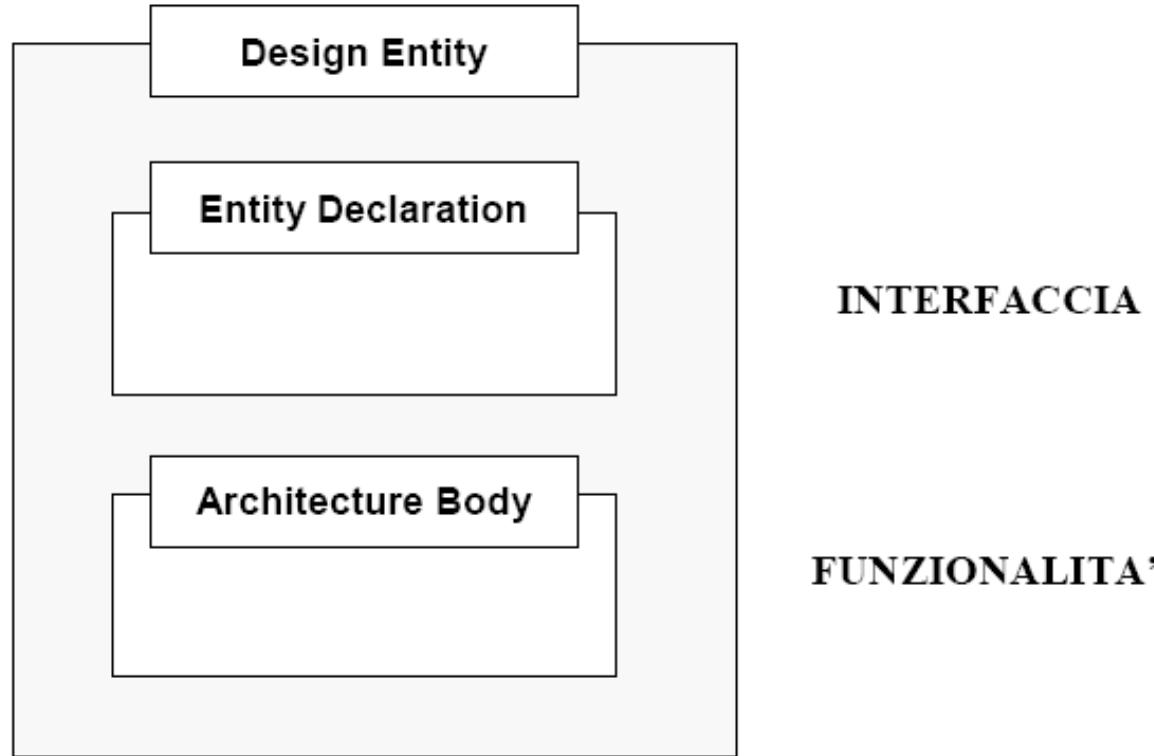
---

- ⊕ Lo sviluppo di un modello si distingue in varie fasi
  - ⊕ Analisi
  - ⊕ Progettazione
  - ⊕ Scrittura
  - ⊕ Compilazione
  - ⊕ Simulazione
  - ⊕ Validazione
- ⊕ La progettazione (o specifica concettuale) consiste nella descrizione di:
  - ⊕ Interfaccia
  - ⊕ Funzionalità



# Struttura di un modello

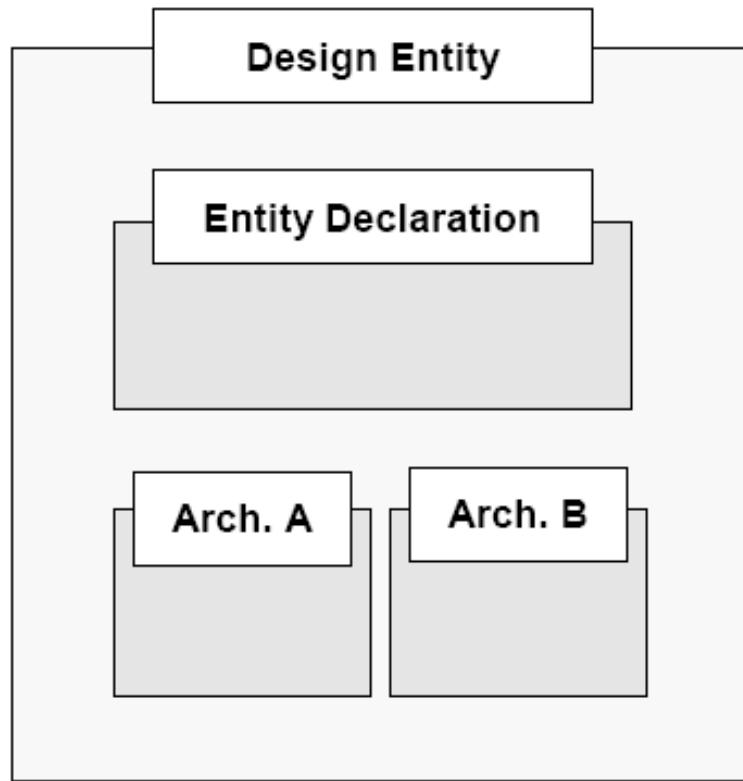
---



- ⊕ Diverse *Architecture Body* possono essere associate ad una stessa *Entity Declaration* all'interno della stessa entità

# Struttura di un modello

---



- ⊕ Ogni architettura rappresenta un diverso aspetto o modalità di realizzazione delle funzionalità

# Design Entity

---

- ❖ E' l'unità base della progettazione in VHDL

- ❑ Descrive un componente solo come interfaccia da e verso l'esterno
  - ❑ Fornisce una visione "ai morsetti"
  - ❑ Non fornisce alcun dettaglio sul funzionamento o sull'architettura



- ❖ Può rappresentare

- ❖ Una singola porta logica elementare
  - ❖ Un circuito integrato
  - ❖ Una scheda PCB (Printed Circuit Board)
  - ❖ Un intero sistema

# Entity Declaration

---

```
entity ENTITY_NAME is
    port (PORT_NAME : DIRECTION TYPE;
          ...
          PORT_NAME : DIRECTION TYPE );
end ENTITY_NAME;
```

- ⊕ La definizione dell'interfaccia inizia con la parola chiave **entity**
- ⊕ ENTITY\_NAME è il nome dell'interfaccia, da usare come riferimento
- ⊕ PORT\_NAME: Il nome dei segnali che permettono all'interfaccia di comunicare con il mondo esterno
- ⊕ DIRECTION: in/out/inout, indica la direzione del segnale rispetto all'entità stessa
- ⊕ TYPE: indica il tipo del segnale e con esso l'insieme dei valori ammissibili e le operazioni definite (es. interi, reali, valori logici, ...)
- ⊕ La definizione dell'interfaccia termina con la parola chiave **end** e il nome dell'interfaccia stessa

# Architecture Body (1/2)

---

- ⊕ Per ogni entità specifica
  - ⊕ Il comportamento
  - ⊕ Le interconnessioni
  - ⊕ Le relazioni tra ingressi e uscite
  - ⊕ I componenti in termini di entità già definite
- L'architettura può essere descritta tramite tre diversi approcci:
  - ▶ Descrizione **DATAFLOW**
    - Basata su formule logiche
  - ▶ Descrizione **STRUCTURAL**
    - Composta da componenti istanziati e interconnessi tra di loro
  - ▶ Descrizione **BEHAVIORAL** (comportamentale)
    - Basata su descrizioni algoritmiche
- La descrizione può essere anche di tipo **misto**

# Architecture Body (2/2)

---

**architecture** BODY\_NAME **of** ENTITY\_NAME **is**

*istruzioni dichiarative*

**begin**

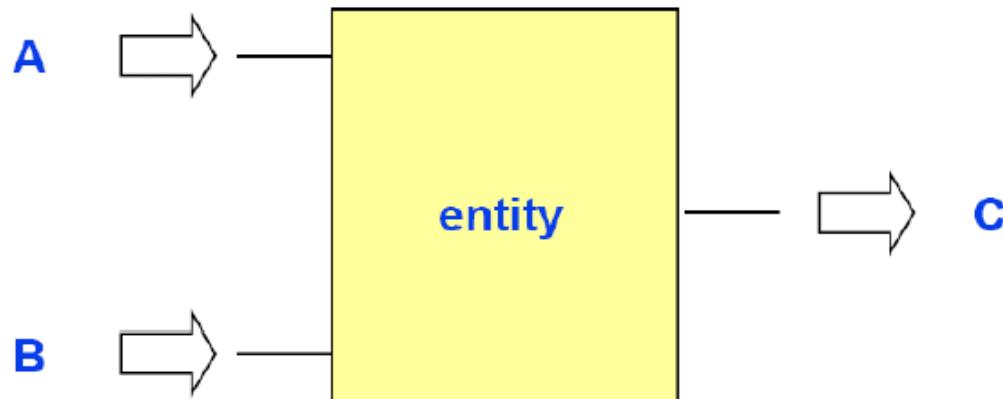
*istruzioni funzionali del modello*

**end** BODY\_NAME;

- ⊕ La definizione dell'interfaccia inizia con la parola chiave **architecture**
- ⊕ BODY\_NAME è il nome di riferimento dell'architettura
- ⊕ ENTITY\_NAME è il nome dell'entità di cui stiamo descrivendo le funzionalità
  - ⊕ Serve ad associare un'architettura all'entità corrispondente
- ⊕ Le *istruzioni dichiarative* permettono di dichiarare variabili, segnali o sottocomponenti che verranno usati nell'architettura
- ⊕ Le *istruzioni funzionali* permettono di definire il valore delle porte di uscita a diversi livelli di astrazione
  - ⊕ Queste istruzioni sono **CONCORRENTI**

# Esempio (1/4) - Specifica concettuale e analisi

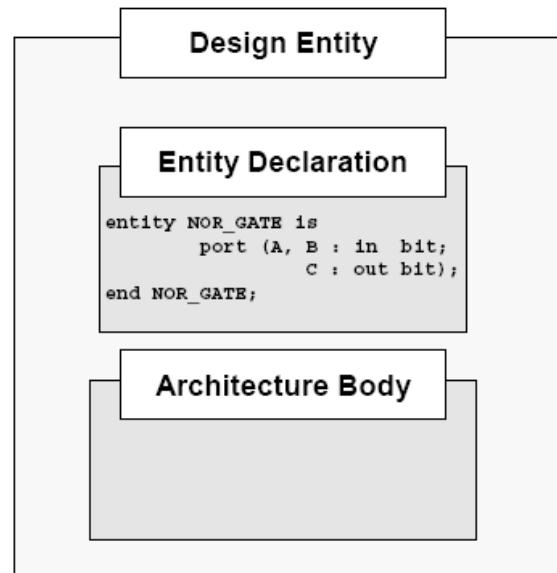
- ⊕ L'entità è dotata di due ingressi ed un'uscita
- ⊕ I segnali sono monodimensionali (1 bit...)
- ⊕ Se ambedue gli ingressi sono a livello logico basso (0), l'uscita è a livello logico alto (1)
- ⊕ In tutti gli altri casi, l'uscita è bassa (0)



Ingressi	Uscita	
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

# Esempio (2/4) – Entity Declaration

```
entity NOR_GATE is
  port ( A, B : in bit;
         C : out bit );
end NOR_GATE;
```



- ⊕ L'entità ha nome NOR\_GATE
- ⊕ A e B le porte di ingresso, ciascuna ampia 1 bit
- ⊕ C è la porta di uscita del risultato, anch'essa di 1 bit
- ⊕ Le porte dello stesso tipo e direzione possono essere elencate consecutivamente, separate da virgole (,)
- ⊕ Le porte di tipo o direzione differente vanno separate da ;

# Esempio (3/4) – Architecture Body

---

```
architecture DATA_FLOW of NOR_GATE is
```

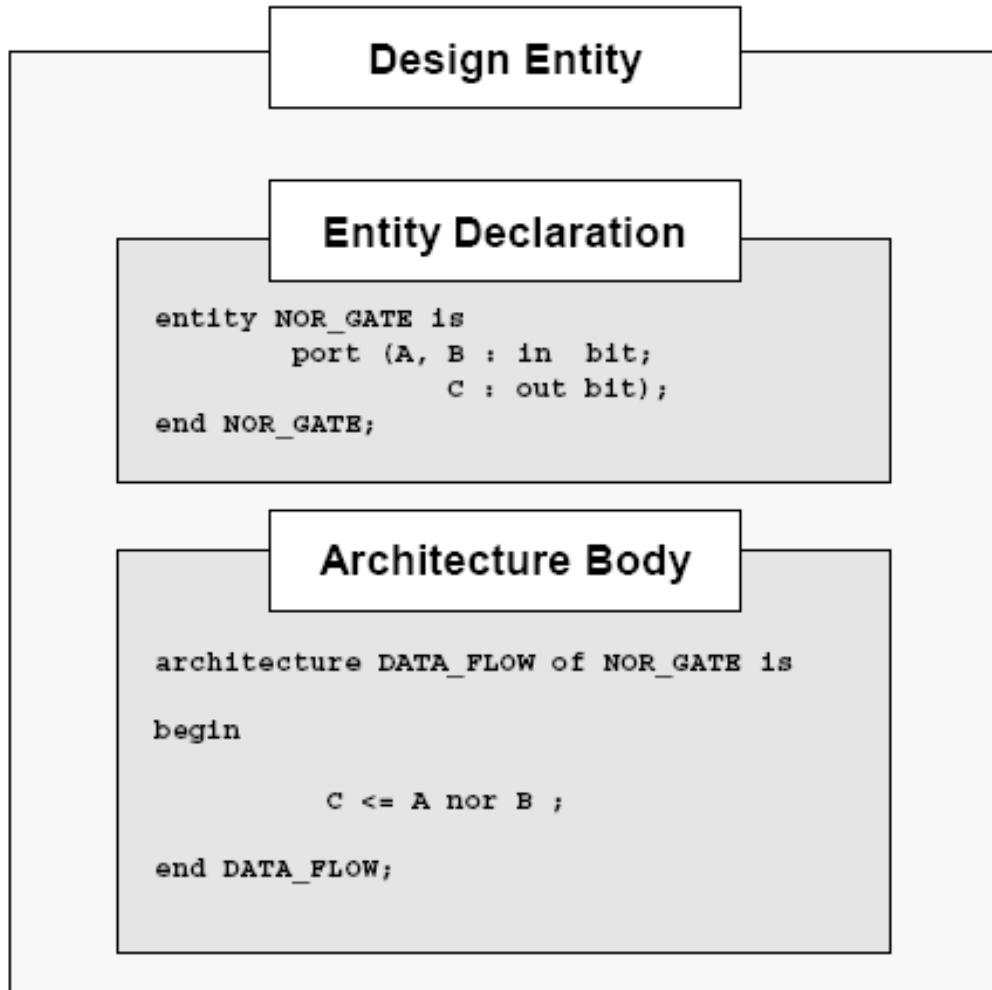
```
begin
```

```
    C <= A nor B;
```

```
end DATA_FLOW;
```

- ⊕ L'Architecture Body definito ha nome DATA\_FLOW ed è associato all'entità NOR\_GATE definita nelle slide precedenti
- ⊕ Non ci sono istruzioni dichiarative
- ⊕ Il corpo funzionale è formato da una sola istruzione NOR
- ⊕ L'operatore  $\leq$  è l'operatore di **assegnamento** tra segnali
- ⊕ Ogni istruzione è terminata da ;

# Esempio (4/4) – Modello VHDL NOR\_GATE



# Architettura Behavioral

---

- ❑ Descrizione di un'architettura con una sintassi algoritmica mediante processi
  - ▶ risulta molto simile ad un algoritmo espresso secondo i classici linguaggi sequenziali (C, Fortran, Pascal, ecc..)
- ❑ Le istruzioni sono eseguite sequenzialmente all'interno di un processo
- ❑ Ogni processo è visto dall'esterno come una sola istruzione concorrente
- ❑ Utile per simulare parti di circuito senza dover scendere troppo nel dettaglio del funzionamento
- ❑ Diversi processi comunicano tra loro mediante segnali ma al loro interno lavorano mediante variabili
- ❑ **ATTENZIONE:** non tutto ciò che viene descritto al livello comportamentale risulta sintetizzabile

# Architettura Dataflow (1/2)

---

- L'architettura è descritta in termini di una serie di espressioni
- Le espressioni sono eseguite in modo concorrente
  - ▶ La **posizione** relativa delle istruzioni che rappresentano le varie espressioni è **ININFLUENTE**
- Le espressioni possibili sono:
  - ❖ Assegnamento di segnali
  - ❖ Assegnamento condizionato di segnali
  - ❖ Assegnamento selettivo di segnali
  - ❖ Istanze di componenti
  - ❖ Dichiarazioni di blocchi
  - ❖ Procedure, asserzioni, processi

# Architettura Dataflow (2/2)

---

- Esempio:

```
architecture DATAFLOW of COMPARE is
begin
    c <= not (a xor b) after 1 ns;
end DATAFLOW
```

- Può contenere indicazioni timing da rispettare in fase di simulazione

# Architettura Strutturale (1/4)

---

- La descrizione strutturale è composta da istanze di componenti interconnessi tra loro
- La descrizione strutturale permette una specifica gerarchica del sistema
  - ▶ Supporta la progettazione incrementale sia top/down che bottom/up
- I vari componenti devono essere già presenti in una libreria di riferimento
- La dichiarazione dei componenti può essere raccolta in un “package”
- Se esistono più architetture per lo stesso componente si può usare una configurazione per specificare quale implementazione considerare

# Architettura Strutturale (2/4)

---

- Nella parte dichiarativa dell'architettura vanno specificati i componenti utilizzati
  - ▶ Si sostituisce la parola entity con la parola component

```
component COMPONENT_NAME is
    generic(
        GENERIC_NAME: TYPE := DEFAULT_VALUE
    );
    port( PORT_NAME: DIRECTION_TYPE;
        ...
        PORT_NAME: DIRECTION_TYPE;
    )
end component [ENTITY_NAME];
```

# Architettura Strutturale (3/4)

---

- I componenti vanno istanziati dopo l'istruzione **begin**

```
nome_istanza : nome_componente
[generic map (assegnamento costanti, . . .)]
port map (assegnamento segnali, . . .);
```

- La “**port map**” indica il collegamento fisico
  - ▶ Assegnamento posizionale
  - ▶ Assegnamento nominale
- Eventuali valori predefiniti (**generic**) possono essere sovrascritti

# Architettura Strutturale (4/4)

---

- Nel port map per ciascuna porta si può specificare
  - ▶ Un segnale
  - ▶ Una costante
  - ▶ **Open** (non connesso)
  - ▶ Una porta della entità
  
- Vengono solitamente impiegati segnali per connettere le varie istanze dei componenti

# Configurazioni

---

- - Le configurazioni permettono di configurare i componenti utilizzati da ciascuna unità
  - In particolare permettono di scegliere l'architettura da usare per una entità qualora ce ne siano diverse
    - ▶ Se non viene effettuata, viene usata l'architettura predefinita (in genere l'ultima descritta)
  - È possibile anche specificare un'unità di progetto chiamata configuration

```
nomi_oggetti : nome_locale_componente
use entity nome_entità(nome_architettura);
```

# Progettazione

---

- Gli strumenti di sintesi si appoggiano a librerie dove sono descritti i componenti che si possono utilizzare (forniti dal venditore)
- Sintesi logica: passaggio tra descrizione comportamentale a implementazione fisica
  - ▶ descrizione a porte logiche
  - ▶ processo delicato che deve essere opportunamente “guidato ed ottimizzato”
- Una delle funzioni del VHDL è quella di **descrivere/documentare** il funzionamento di un sistema in modo chiaro ed inequivocabile
- Solo un ristretto sottoinsieme del VHDL si presta ad essere sintetizzato automaticamente.
  - ▶ **Non tutto ciò che è scritto in VHDL è sintetizzabile**
  - ▶ Può essere comunque utile per la descrizione e per la simulazione del sistema

# Simulazione

---

- Un sistema descritto in VHDL viene solitamente simulato per analizzarne il comportamento (**simulazione comportamentale**)
  - ▶ Verifica della correttezza
  - ▶ Analisi dell'evoluzione temporale
- Bisogna:
  - ▶ fornire degli stimoli (INPUT)
  - ▶ avere un sistema capace di osservare l'evoluzione del modello durante la simulazione, registrarne le variazioni per un'eventuale ispezione di funzionamento
- NOTA BENE: Il simulatore deve aver la possibilità di rappresentare valori “*unknown*” o “*non-initialized*” o alta impedenza (valori elettrici tipici di sistemi hardware)
  - ▶ Logica multi-valore

# Sintesi vs. Simulazione

---

- All synthesizable designs can be simulated
- Not all simulation designs can be synthesized
- Consider the following VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY simple_buffer IS
    PORT ( din      : IN      std_logic;
           dout     : OUT     std_logic );
END simple_buffer;

ARCHITECTURE behaviourall OF simple_buffer IS
BEGIN
    dout <= din AFTER 10 ns;
END behaviourall;
```

- The input din is assigned to dout after 10 ns
  - Can this represent a real-world system? YES
  - Can this be implemented in a device? PERHAPS
  - Can this be implemented in all devices? NO
- This architecture can be simulated but not synthesized

---

**SINTASSI**

# Scrittura del codice sorgente (1/2)

---

- ⊕ Il codice sorgente di un modello VHDL è un file di semplice testo
- ⊕ In genere si usa un nome uguale al nome dell'entità; l'estensione **deve** essere \*.vhd
- ⊕ Una volta completata la stesura del codice, questo va compilato
  - ⊕ Correggete gli eventuali errori sintattici
  - ⊕ Includete le librerie:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

# Scrittura del codice sorgente (2/2)

---

- Il codice sorgente di un modello VHDL è un file di testo \*.vhd
  - ▶ “--” indica l'inizio di una riga di commento al codice
- Ogni elemento della specifica ha un nome simbolico
  - ▶ I nomi seguono le solite regole sintattiche dei linguaggi di programmazione
    - Il primo carattere deve essere una lettera
    - I seguenti possono essere alfanumerici
- Il VHDL e' un linguaggio “**case insensitive**”
  - ▶ (ossia abcd equivale a AbCd)
- Vi sono “nomi riservati” quali:  
`in, out, signal, port, library,  
map, entity, ...`

# Codice Sorgente nor\_gate.vhd

---

```
library IEEE;
use IEEE.std_logic_1164.ALL;

entity NOR_GATE is
    port ( A, B : in bit;
            C : out bit );
end NOR_GATE;

architecture DATA_FLOW of NOR_GATE is
begin
    C <= A nor B;
end DATA_FLOW;
```

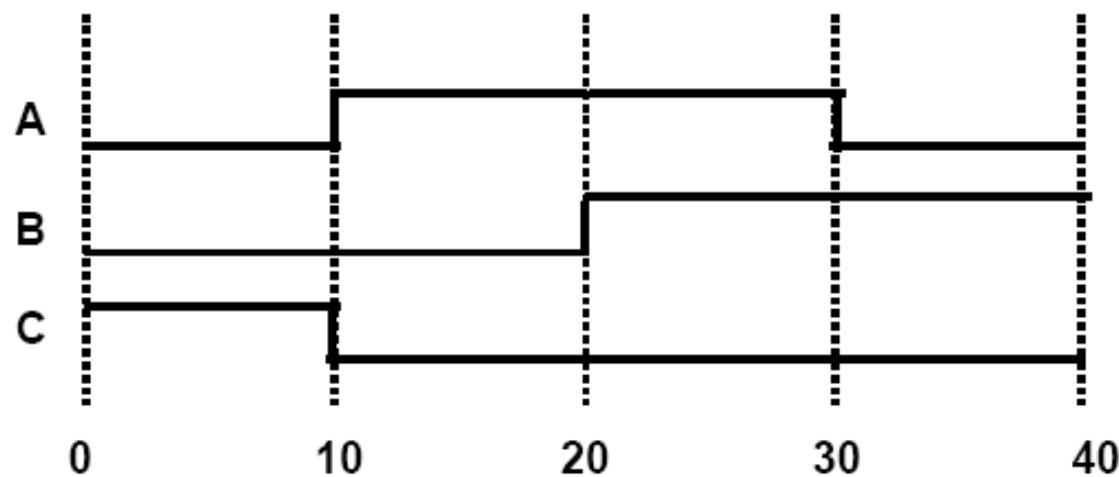
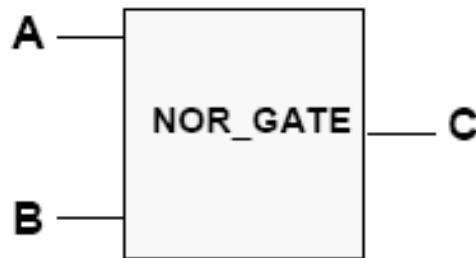
# Verifica

---

- ⊕ L'architettura va verificata con appositi strumenti
- ⊕ Tramite un editor di forme d'onde, vengono definiti gli ingressi di test
- ⊕ Con uno strumento di simulazione (ModelSim) possiamo analizzare l'andamento degli altri segnali (uscita, altri segnali ausiliari)

# Esempio di simulazione NOR\_GATE

---



# Principi di base del VHDL (1/2)

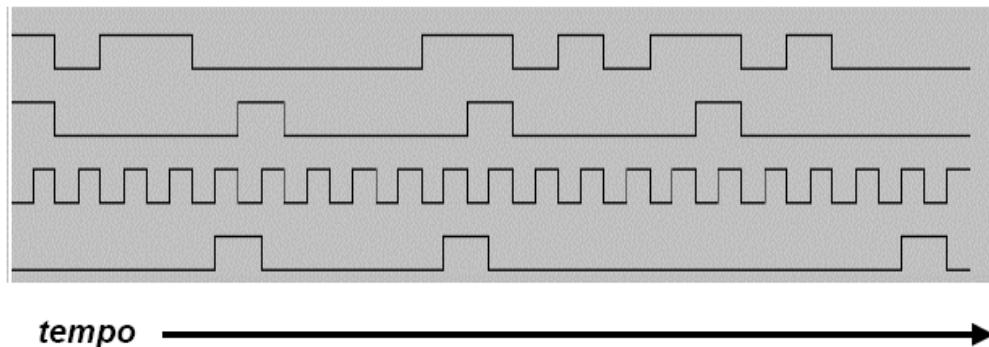
---

- Supporta la descrizione della funzionalità del modello a diversi livelli di astrazione.
  - ❖ Comportamentale
  - ❖ Flusso dati
  - ❖ Strutturale
  - ❖ Mista ...
- Concorrenza: le strutture hardware sono intrinsecamente concorrenti e composte dall'interconnessione di componenti elementari (le cui attività avvengono in parallelo).
  - ❖ A livello dei modelli strutturali
  - ❖ A livello dei processi ...
- Supporta istruzioni sequenziali all'interno di un processo.
  - ❖ E' possibile definire una sequenza di istruzioni esplicitamente sequenziali

# Principi di base del VHDL (2/2)

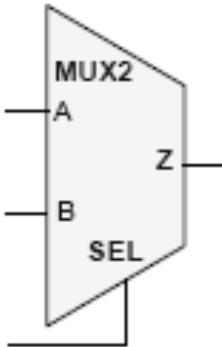
---

- **Gerarchia:** data la complessità progettuale occorre organizzare il progetto su diversi livelli gerarchici, che possono essere descritti a diversi livelli di astrazione.
  - ⊕ Supporta progettazione top/down o bottom/up
- **Temporizzazioni:** necessità di modellizzare l'andamento temporale dei segnali attraverso la descrizione di forme d'onda cioè di segnali il cui valore logico evolve nel tempo.



# Esempio – MUX2

- Esempio: Scrivere il modello VHDL *comportamentale e dataflow* del componente MUX2 descritto dalla seguente specifica:

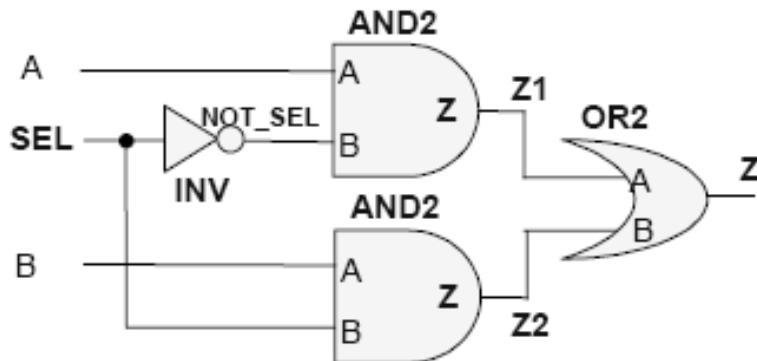
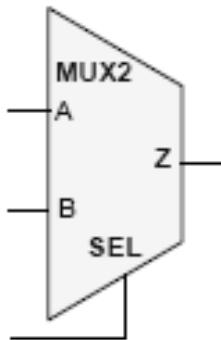


SEL	A	B	Z
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

```
entity MUX2 is
port (A, B, SEL: in bit;
      Z: out bit);
end MUX2;
-- modello comportamentale --
architecture ARC1 of MUX2 is
begin
    P1: process (A, B, SEL)
    begin
        if (SEL = '0') then
            Z <= A;
        else
            Z <= B;
        end if;
    end process P1;
end ARC1;
-- modello data-flow --
architecture DATA_FLOW1 of MUX2 is
begin
    Z <=     A when (SEL = '0') else
              B ;
end DATA_FLOW1;
-- modello data-flow --
architecture DATA_FLOW2 of MUX2 is
begin
    Z <=     (A and not SEL) or (B and SEL)
              B ;
end DATA_FLOW2;
```

# Esempio – MUX2

- Esempio: Scrivere il modello VHDL *strutturale* del componente MUX2 descritto dalla seguente specifica:



```
entity MUX2 is
port (A, B, SEL: in bit;
      Z: out bit);
end MUX2;
-- modello strutturale --
architecture STRUCT of MUX2 is
component inv is
    port (A: in bit;
          Z: out bit);
end component;
component and2 is
    port (A, B: in bit;
          Z: out bit);
end component;
component or2 is
    port (A, B: in bit;
          Z: out bit);
end component;
signal NOT_SEL, Z1, Z2: bit;
begin
    u1: inv port map (SEL, NOT_SEL);
    u2: and2 port map (A, NOT_SEL, Z1);
    u3: and2 port map (B, SEL, Z2);
    u4: or2 port map (Z1, Z2, Z);
end STRUCT;
```

---

# Classi di Oggetti

# Oggetti

---

- Possibili tre classi per gli oggetti:
  - Costanti
    - Oggetti assegnati ad un valore iniziale che non può essere cambiato.
  - Variabili
    - Oggetti che ricevono un valore che può essere cambiato attraverso un'istruzione di assegnamento tra variabili
  - Segnali
    - Oggetti che ricevono dei valori associati ad un fattore temporale e che possono essere cambiati attraverso un'istruzione di assegnamento tra segnali.

# Costanti

---

- Aiutano a migliorare la leggibilità del codice

- Sintassi:

```
constant nome: tipo := espressione;  
constant nome: tipo_array[(intervallo)] :=  
    espressione;
```

- Esempio:

```
constant GROUND: bit :=0 ;  
constant SYS_BUS : std_logic_vector (7 downto 0) :=  
    "00001111";  
constant SYS_ADD : std_logic_vector (0 to 7) :=  
    "00001111";
```

# Variabili

---

- Sintassi:

```
variable var_name : tipo [:= valore];
```

- La visibilità di una variabile è limitata al processo in cui è dichiarata
- L'assegnamento del valore ad una variabile avviene istantaneamente
- In un processo le variabili locali conservano in proprio valore nel tempo tra un'esecuzione e l'altra

```
variable INDEX : integer range 1 to 50;
variable CYCLE : time range 10 ns to 50 ns := 10ns;
variable MEMORY : bit_vector (0 to 7);
variable x,y,z : integer;
```

# Segnali

---

- Sono l'astrazione dei “collegamenti fisici”
- Connettono i vari elementi della specifica (l'entità, i componenti istanziati e i processi)
- Un segnale può essere inizializzato
  - ▶ **ATTENZIONE:** in fase di SINTESI l'inizializzazione potrebbe essere disattesa!

- Sintassi

```
signal nome: tipo <= espressione;  
signal nome: tipo_array[(intervallo)] <=  
espressione
```

- Esempio:

```
signal count: integer range 1 to 10;  
signal SYS_BUS : std_logic_vector (7 downto 0);
```

# Segnali e Variabili

	Segnali	Variabili
Dichiarazione	Parte dichiarativa di un'architettura	Parte dichiarativa di un processo
Valore predefinito		Valore minimo del dominio di appartenenza
Assegnamento	$<=$	$:=$
Inizializzazione		$:=$
Natura dell'assegnamento	Concorrente	Sequenziale
Utilizzo	In architetture e processi	Solo in processi
Effetto dell'assegnamento	Non immediato (in base ai "tempi" della simulazione)	Immediato

# Assegnamento di Segnali (1/2)

---

- Sintassi:

```
segnalet <= valore1 [after ritardo];
```

- Esempi

```
sum <= a xor b after 5ns;
```

```
carry <= a and b;
```

```
data_out(0) <= data_in(7);
```

```
data_out(6 downto 0) <= data_in(6 downto 0);
```

- Gli indici degli intervalli possono essere diversi!

# Assegnamento di Segnali (2/2)

---

- Assegnamento posizionale

```
vettore_di_bit <= (bit1, bit2, bit3, bit4);
```

- Assegnamento nominale

```
vettore_di_bit <= (2=>bit1, 1=>bit2, 0=>bit3,  
3=>bit4);
```

- Altri assegnamenti

```
vettore_di_bit <= (0 to 1 => '0', others =>'1');
```

# Concetto di ritardo (1/4)

---

- Il concetto di ritardo in un modello VHDL viene definito come il periodo di tempo che intercorre tra causa ed effetto.

- Modello senza ritardi:

```
C <= A and B;
```

- Modelli con ritardo inerziale utilizzano l'istruzione AFTER:

- Formato:

```
NEW_SIGNAL <= SIGNAL_EXPR after TIME_PERIOD;
```

- Esempi:

```
C <= A and B after 10 ns;  
  
D_OUT <= '1' after 10 ns,  
          '0' after 20 ns,  
          '1' after 30 ns,  
          '0' after 40 ns;  
  
Q <=      '1' after 10 ns when b = '1' else  
          '0' after 5 ns;
```

# Concetto di ritardo (2/4)

```
-- xor gate
-----
library IEEE;
use IEEE.Std_Logic_1164.all;
entity XOR_GATE is
    port (A, B : in std_ulogic;
          C : out std_ulogic);
end XOR_GATE;
architecture LOGIC of XOR_GATE is
begin
    C <= A xor B after 10 ns;
end LOGIC;
```

# Concetto di ritardo (3/4)

---

- I segnali la cui durata è inferiore del ritardo specificato nell'istruzione `after` sono ignorati
- Esempio:

```
A <= B after 50 ns;
```

- Stimoli:

```
force B 0 0  
force B 1 50  
force B 0 100  
force B 1 150  
force B 0 175  
run 250
```

- Risultato:
  - L'impulso di durata 25 ns imposto al segnale b a 150 ns non viene trasmesso al segnale a.

# Concetto di ritardo (4/4)

---

- L'opzione **transport** permette di trasmettere la forma d'onda presente su un segnale ad un altro segnale senza filtri.
- Esempio:

```
A <= transport B after 50 ns;
```

- Stimoli:

```
force B 0 0  
force B 1 50  
force B 0 100  
force B 1 150  
force B 0 175  
run 250
```

- Risultato:
  - L'impulso di durata 25 ns imposto al segnale **b** a 150 ns viene trasmesso al segnale **a**.

# Assegnamento condizionale

## ⊕ Sintassi

- ⊕ segnale <= valore1 **when** condizione1 **else**  
                  valore2 **when** condizione2 **else**  
                  ... ...  
                  Altro\_valore;

O

- ⊕ **if** condizione1 **then**  
                  segnale <= valore1  
**else if** condizione2 **then**  
                  segnale <= valore2  
                  ... ...  
**else**  
                  segnale <= Altro\_valore;

## □ Esempio:

```
architecture DATAFLOW of MUX is
begin
    out <= in1 when sel='1' else in2;
end DATAFLOW
```

*condizione* è **boolean**

**else** deve essere  
presente

La prima espressione  
vera viene assegnata

Possono essere usati i  
ritardi

# Assegnamento selettivo

---

- ❖ Sintassi

- ❖ **with** espressione **select**  
    segnalet <= valore1 **when** caso1,  
    segnalet <= valore2 **when** caso2,  
    ... ...  
    espressioneN **when** casoN  
    [,Altra\_espressione **when** **others**];

- ❖ **others** può essere esclusa se tutti i casi sono considerati

- ❖ Non si possono sovrapporre i casi

- Esempio:

```
with alu_func select
    res <= a+b when "000"|"001",
    res <= a-b when "010"|"011",
    res <= "0000" when others;
```

# Indirizzamento negli array

---

- Per riferirsi ad una posizione di un array si usano le parentesi tonde
  - ▶ Esempio: `my_array(2)`
- Per riferirsi ad un intervallo si usano le parole chiave to e downto
  - ▶ Esempio: `my_array(2 to 4)`, `my_array(5 downto 2)`
  - ▶ Diverso direzionamento delle porte
    - Utile per "incrociare" i segnali
  - ▶ `out1(2 to 4) <= in1(7 downto 5);`

# Attributi

---

- Esistono degli attributi predefiniti che sono associati ai segnali (e alle variabili)
- Si accede agli attributi tramite l'operatore ' :
  - ▶ Esempio: **sig\_array' lenght**
- Per i segnali è definito l'attributo **event**
  - ▶ assume valore “vero” se in un determinato istante si è verificato un evento sul segnale

---

# Tipi di Dati e Operatori

# Tipi di Dati

---

- Il VHDL e' costituito da vari tipi (**types**) per consentire simulazione e sintesi a vari livelli
  - **Ogni oggetto deve appartenere ad un tipo**
  - **Il tipo di dato identifica:**
    - un insieme di valori che un oggetto (costante, variabile o segnale) può assumere;
    - un insieme di operazioni che possono essere eseguite sull'oggetto stesso.
- Il VHDL e' un linguaggio fortemente tipizzato
  - ▶ Non è possibile fare il casting in modo implicito
  - ▶ Ci sono delle funzioni che permettono di "tradurre" da un tipo ad un altro

# Tipi di Dati

---

- Tipi di dati predefiniti - Esempi:  
`bit, bit_vector, boolean, integer, ....`
- Tipi di dati definiti dall'utente - Esempio:

```
type MY_STATE is (RESET, IDLE, READ, WRITE);
signal STATE: MY_STATE;
...
STATE <= RESET;

STATE <= READ;
```

- È possibile definire nuovi tipi e sottotipi

# Tipi di Dati

---

Scalari	Vettori
Character	String
Bit	Bit_vector
Std_logic	Std_logic_vector
Boolean	
Real	
Integer	
Time	
File	

# Character

---

- Il character assume i valori specificati nel set ISO 8859
- Il carattere va specificato tra apici
  - ▶ Es:      'a'      'b'
- Si può utilizzare la dichiarazione esplicita
  - ▶ Es:      **character**'('a')      **character**'('1')

# Bit

---

- Il bit assume solo valori '0' o '1' e va dichiarato tra virgolette singole
  - ▶ Es:      '0'      '1'
  
- Si può utilizzare la dichiarazione esplicita
  - ▶ Es:      **bit**'('0')      **bit**'('1')

# Integer

---

- ⊕ Integer: interi naturali positivi su 32 bit
  - Può essere utile per simulazioni ad alto livello
    - ▶ **NON SEMPRE VIENE SINTETIZZATO**
  - NON DEVE contenere il punto decimale ma può eventualmente contenere il segno
    - ▶ Es:      10      +223    - 456
  - Un intero può eventualmente essere espresso in un'altra base
    - ▶ Es:      16#00F0F#
  - Esistono due subset predefiniti degli integer: **positive** e **natural**

# Real

---

- ⊕ Real: reali da -1.0E-38 a +1.0E+38

- Può essere utile per simulazioni ad alto livello
  - ▶ Evita confusione con i tipi interi
- DEVE contenere il punto decimale ed eventualmente il segno
  - ▶ Es:        1 . 0        +2 . 23    -  4 . 56        -1 . 0E+38
- **ATTENZIONE: NON VIENE SINTETIZZATO!!!**
  - ▶ Deve essere descritta esplicitamente l'architettura dei componenti che eseguono le operazioni

# Time

---

- È la sola grandezza **fisica** predefinita in VHDL
- È importante separare il valore dall'unità di grandezza
  - ▶ Es: **10 ns**, **123 us**, **6.3 sec**
- Unità di grandezza consentite:  
**fs ps ns us ms sec min hr**

# Altri tipi scalari

---

- Boolean
  - ▶ Assume valore **true** o **false**
  - ▶ È il risultato di espressioni logiche e relazionali
  
- File
  - ▶ Vengono utilizzati nelle descrizioni di test (testbench) per caricare gli stimoli in ingresso al circuito e salvare gli output
  - ▶ (ovviamente) non possono essere sintetizzati

# **Std\_logic (1/2)**

---

- Per utilizzarlo va inclusa la libreria IEEE e specificato l'utilizzo del package std\_logic\_1164:

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;
```

- E' il tipo più usato per la sintesi logica
- Viene dichiarato racchiuso tra virgolette singole
  - ▶ Es:      'U'      'X'      '1'      '0'
- In caso di equivoco si usi la dichiarazione esplicita
  - ▶ Es:      **std\_logic'('1')**

# Std\_logic (2/2)

---

- Assume i valori:
  - ▶ U: uninitialized
  - ▶ X: unknown
  - ▶ 0
  - ▶ 1
  - ▶ Z: alta impedenza
  - ▶ W: weak unknown
  - ▶ L: weak 0
  - ▶ H: weak 1
  - ▶ - : don't care, indifferenza
- Esiste la versione **std\_ulogic** che non prevede la risoluzione dei segnali

# Vettori

---

- ⊕ I tipi possono essere aggregati in vettori (*array*) predefiniti
  - ⊕ String : array di caratteri
    - ⊕ “Tipo string”
  - ⊕ Bit\_vector
    - ⊕ “00011011”
  - ⊕ Std\_logic\_vector
    - ⊕ “11xxx00-”
- ⊕ Può essere necessario dover esplicitare il tipo
  - ⊕ String’(“1000”)
    - ⊕ Potrebbe essere interpretato altrimenti come bit\_vector o std\_logic\_vector

# string

---

- La string è un array di caratteri
  - ▶ Es:      **"ciao"**      **"345"**
  
- Si può utilizzare la dichiarazione esplicita
  - ▶ Es:      **string("011001010011")**

# bit\_vector (1/2)

---

- Il bit\_vector è un array di bit che assumono solo valori ‘0’ o ‘1’ e va dichiarato tra virgolette doppie
- È comunque consentito adottare una notazione ottale o esadecimale
- Il carattere ‘\_’ può essere adottato per comodità per dividere le cifre in gruppi (non viene interpretato)
  - ▶ Es: “0001\_1001” x”00FF”
- Si può utilizzare la dichiarazione esplicita
  - ▶ Es: **bit\_vector' (“0110\_0101\_0011”)**

Esempio:

```
signal Z_BUS: bit_vector(3 downto 0);
signal A_BUS: bit_vector(1 to 4);
...
Z_BUS <= A_BUS;
```

# bit\_vector (2/2)

---

Esempio:

```
signal Z_BUS: bit_vector(3 downto 0);
signal A_BUS: bit_vector(1 to 4);
...
Z_BUS <= A_BUS;
```

Equivale ad assegnamento bit per bit per posizione:

```
Z_BUS(3) <= A_BUS(1);
Z_BUS(2) <= A_BUS(2);
Z_BUS(1) <= A_BUS(3);
Z_BUS(0) <= A_BUS(4);
```

# **std\_logic\_vector**

- Viene dichiarato racchiuso tra virgolette doppie
    - ▶ Es:      "001XX"      "UUUU"
  - In caso si voglia esprimere un particolare valore espresso secondo una notazione di tipo “unsigned” o “signed” (complemento a 2) si deve impiegare il package STD\_LOGIC\_ARITH
    - ▶ Es:      signed    ("111001")      (ossia -7)  
              unsigned ("111001")      (ossia 57)
  - bisogna includere i package:

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all;
Use IEEE.STD_LOGIC_ARITH.all;
```

# Tipi e Sottotipi

---

- In VHDL si possono “inventare” nuovi tipi

```
TYPE mese IS (gennaio, febbraio, giugno);  
TYPE bit IS ('0', '1');  
TYPE mystring IS ARRAY (0 to 4) OF bit;  
TYPE norange IS ARRAY (natural range <>) OF bit;  
TYPE rec IS RECORD  
    campol: bit; ...  
END RECORD;
```

- E dei sottoinsiemi

```
SUBTYPE mesefreddo IS  
mese range gennaio to febbraio;
```

# Operatori

---

- Un tipo di oggetto identifica anche un insieme di operazioni che possono essere eseguite sull'oggetto.
- Operatori predefiniti:
  - Operatori aritmetici: +, -, \*, /, \*\*, abs, mod, rem
  - Segno: +, -
  - Operatori di scorrimento: rol, ror, sla, sll, sra, srl
  - Operatori misti: &
  - Operatori relazionali: =, /=, <, <=, >, >=
  - Operatori logici: and, or, nand, nor, not, xor, xnor

# Operatori aritmetici e di scorrimento

&	Concatenazione
+	Addizione
-	Sottrazione
+	Più unario
-	Opposto (unario)

*	Moltiplicazione
/	Divisione
<b>Mod</b>	Modulo
<b>Rem</b>	Resto (conserva il segno)
**	Esponenziazione
<b>abs</b>	Valore assoluto
<b>not</b>	Complemento

<b>sll, slr</b>	Scorrimento logico a sinistra/destra
<b>sra, sla</b>	Scorrimento aritmetico a sinistra/destra
<b>rol, ror</b>	Rotazione a sinistra/destra

# Operatori relazionali

---

- Operatori relazionali predefiniti per i tipi **bit** e **boolean**:

<	-- less than
<=	-- less than or equal to
>	-- greater than
>=	-- greater than or equal to
=	-- equal to
/=	-- not equal to

Esempio:

```
if (A = B) then  
    statement;
```

La condizione (A = B) ritorna un valore boolean

# Operatori logici

---

- Operatori logici predefiniti per i tipi `bit` e `boolean`:

`and`

`or`

`nand`

`nor`

`xor`

`xnor`

`not` -- Operatore a maggiore priorità

Esempio:

```
Z <= A and not (B or C)    -- necessaria la parentesi per  
                           -- alterare la precedenza degli  
                           -- operatori
```

---

# Package e Librerie

# Package

---

- La condivisione da parte di diverse entity di dichiarazioni di tipi e di operazioni tra tipi viene fatta attraverso dei costrutti noti come package.
  - Sono dei “contenitori” in cui vengono raccolte le definizioni di tipi, costanti, segnali, procedure, funzioni, configurazioni...
  - Nel package STANDARD si trovano descritti quegli oggetti destinati alla descrizione COMPORTAMENTALE (non sempre sintetizzabile)
- Tipi, procedure e funzioni comunemente usate sono poste in package standard come Std\_Logic\_1164 (IEEE standard).
  - Nel package IEEE1164 vi si trovano gli oggetti destinati alla sintesi ed alla simulazione logica

# Librerie

---

- La dichiarazione **library** rende visibile ad un modello VHDL una libreria selezionata che contiene i package desiderati.
- La dichiarazione **use** rende un package visibile ad un modello.
- Esempio:

```
library IEEE;  
use IEEE.Std_Logic_1164.all;
```

- In un dato istante c'è una sola **libreria di lavoro** e diverse librerie di risorse

# Libreria IEEE (1/2)

---

- La libreria standard IEEE è composta 6 package
  - ▶ std\_logic\_1164
  - ▶ std\_logic\_arith
  - ▶ std\_logic\_unsigned
  - ▶ std\_logic\_signed
  - ▶ std\_logic\_misc
  - ▶ std\_logic\_textio
- Solitamente si usano nel modo seguente

```
library IEEE  
use IEEE.<packagename>.all;
```

# Libreria IEEE (2/2)

---

- Definiscono
  - ▶ Tipi
    - std\_logic, std\_logic\_vector, std\_ulogic, ...
  - ▶ Operatori aritmetici
    - +, -, \*, shl, shr
  - ▶ Operatori relazionali
    - >, >=, <, <=, ...
  - ▶ Funzioni di conversione
    - conv\_integer, conv\_std\_logic\_vector, ...
    - to\_stdlogic, to\_stdlogicvector, ...

# Esempio di conversione

---

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity TB is
end TB;

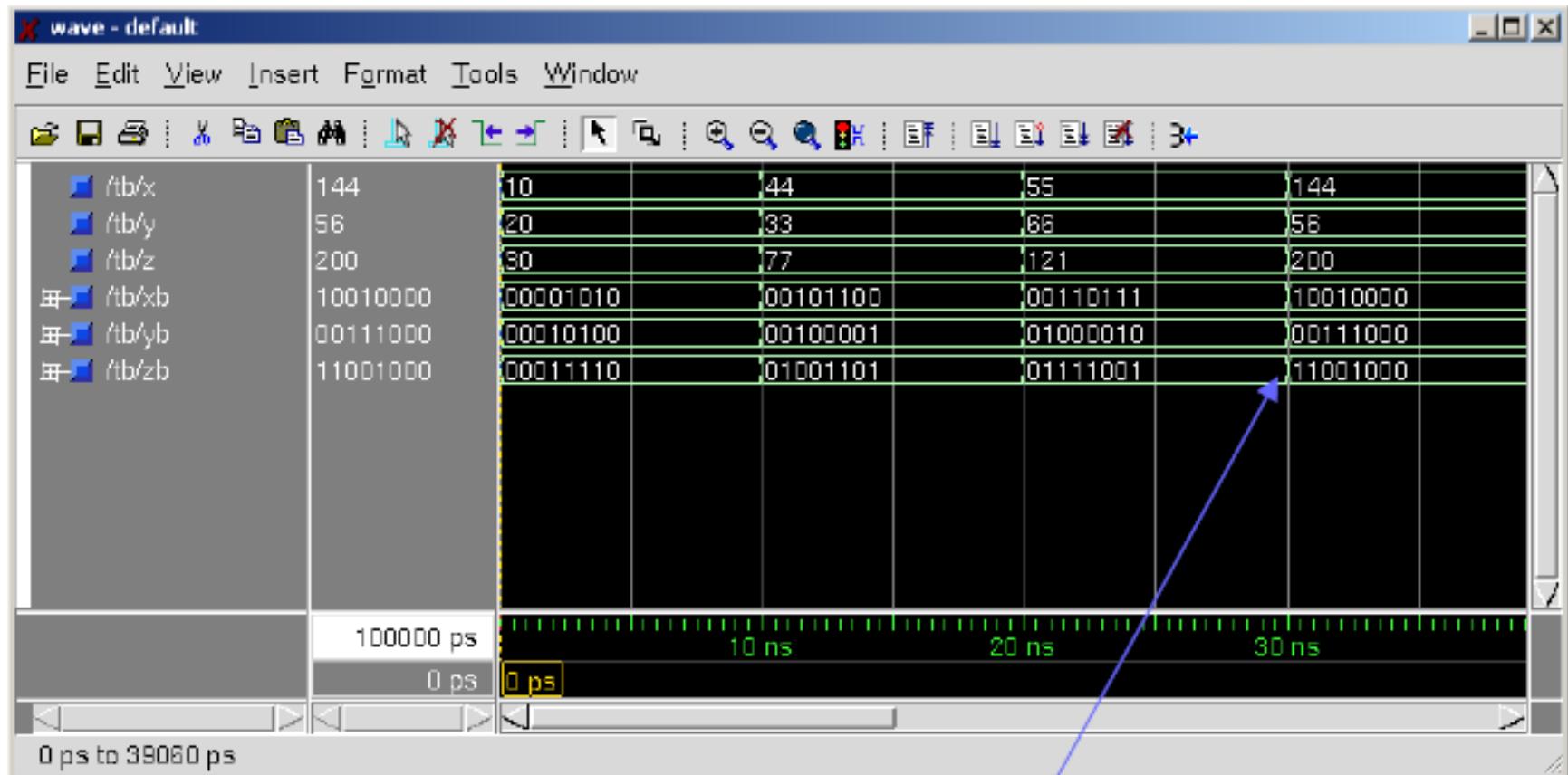
architecture BEH of TB is
    signal x, y, z: integer;
    signal xb, yb, zb: std_logic_vector(7 downto 0);
begin
begin
    gen: process
    begin
        x <= 10; y <= 20; wait for 10 ns;
        x <= 44; y <= 33; wait for 10 ns;
        x <= 55; y <= 66; wait for 10 ns;
        x <= 144; y <= 56; wait;
    end process;

    xb <= conv_std_logic_vector( x, 8 );
    yb <= conv_std_logic_vector( y, 8 );
    z <= conv_integer( zb );

    DUT: zb <= xb + yb;

end BEH;
```

# Esempio di conversione



Interpretato senza segno in modo corretto

# Libreria STD

---

- La libreria STD contiene due packages
  - ▶ **standard**
  - ▶ **textio**
- Il package **standard** definisce
  - ▶ Tipi
  - ▶ Unità di misura del tempo
  - ▶ Files
- Il package **textio** definisce
  - ▶ Il tipo necessario alla gestione dei file di testo
  - ▶ Funzioni per la lettura e scrittura dei file

# Apertura di un file

---

- L'apertura di un file avviene con la dichiarazione  
`file <handle>: text is in "<file>" ;`
- Il file è aperto in lettura/scrittura
  - ▶ Dipende solo dalle operazioni effettuate
- Da un file si leggono unicamente stringhe
- È necessario dichiarare almeno una variabile per la lettura di linee di testo  
`variable <var>: line;`
- Il testo letto non è interpretato

# Il package TEXTIO: Input functions

---

- **readline(F,L)**
  - ▶ Legge una linea dal file F
  - ▶ Lo memorizza nella variabile L
- **read(L,V,B)**
  - ▶ Estrae e converte un valore dalla linea L
  - ▶ Lo memorizza nella variabile V
  - ▶ Se la conversione ha successo B vale true
  - ▶ Il tipo della variabile V può essere
    - bit, bit\_vector
    - integer, real
    - character, string
    - time

# Il package TEXTIO: Output functions

---

- `writeline(F,L)`
  - ▶ Scrive sul file `F` la linea di testo `L`
- `write(L,V,J,W)`
  - ▶ Formatta il valore `V` e lo scrive nella variabile `L`
  - ▶ L'argomento `J` indica la giustificazione e può essere "`left`" o "`right`"
  - ▶ L'argomento `W` indica la dimensione del campo
  - ▶ Il tipo della variabile `V` può essere
    - `bit`, `bit_vector`
    - `integer`, `real`
    - `character`, `string`
    - `time`

# Esempio: Lettura da file

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
library STD;
use STD.textio.all;

entity TB is
end TB;

architecture BEH of TB is
    signal xb, yb, zb: std_logic_vector(7 downto 0);
begin
    gen: process
        file fp:      text is in "in.dat";
        variable ln:   line;
        variable x, y, z: integer;
    begin
        readline( fp, ln ); read( ln, x ); read( ln, y );
        xb <= conv_std_logic_vector( x, 8 );
        yb <= conv_std_logic_vector( y, 8 );
        if endfile( fp ) = true then
            wait;
        else
            wait for 10 ns;
        end if;
    end process;

    DUT: zb <= xb + yb;
end BEH;
```

in.dat

```
10 20
44 33
55 66
144 56
```

# Esempio: Lettura e scrittura da file

```
...
architecture BEH of TB is
    signal    xb, yb, zb: std_logic_vector(7 downto 0);
begin
    gen: process
        file      fpi: text open read_mode is "in.dat";
        variable lni: line;
        file      fpo: text open write_mode is "out.dat";
        variable lno: line;
        variable x, y, z: integer;
    begin
        readline( fpi, lni ); read( lni, x ); read( lni, y );
        xb <= conv_std_logic_vector( x, 8 );
        yb <= conv_std_logic_vector( y, 8 );
        wait for 9 ns;
        z := conv_integer( zb );
        write( lno, z );
        writeln( fpo, lno );
        if endfile( fpi ) = true then
            wait;
        else
            wait for 1 ns;
        end if;
    end process;

    ...
end BEH;
```

in.dat

```
10 20
44 33
55 66
144 56
```

out.dat

```
30
77
121
200
```

---

# Processi

# Processi (1/3)

---

- IL VHDL gestisce l' utilizzo di processi
- I processi inglobano parti di un progetto
- I processi hanno una lista di sensibilità che specifica i segnali che possono causare cambi negli outputs del processo stesso
  - La lista di sensibilità può essere usata per preservare lo stato di un sistema

Esempio, un flip-flop edge-triggered è sensibile solo ad un particolare fronte del clock: l' uscita cambia se e solo se è arrivato un particolare fronte del clock, altrimenti l' uscita rimane invariata

- I processi possono essere usati per implementare logica combinatoria, ma molto spesso inglobano logica sequenziale

# Processi (2/3)

---

- Sintassi:

```
[etichetta:] process [(lista di sensibilità)]
parte dichiarativa (variabili, ...)
begin
  istruzioni sequenziali
  ...
end process [etichetta];
```

- La lista di sensibilità indica i segnali le cui variazioni causano l'attivazione del processo
- Prima dell'istruzione begin vengono dichiarati le variabili usate nel processo
- Dopo l'istruzione begin viene specificato il funzionamento del processo

# Processi (3/3)

---

- Esempio:

```
architecture BEHAVIORAL of COMPARE is
begin
process (A,B)
begin
if (A = B) then
    C <= '1' after 1 ns;
else
    C <= '0' after 1 ns;
end if;
end process;
end BEHAVIORAL
```

# Processi – esecuzione

---

- ❑ Il processo è eseguito una prima volta durante l'inizializzazione
  - ⊕ Completamente se definito tramite lista di sensibilità
  - ⊕ Fino alla prima **wait** in caso contrario
- ❑ Il processo viene risvegliato alla variazione dei segnali della lista di sensibilità
  - ⊕ La lista può essere sostituita dall'istruzione `wait on lista_segnali;`
  - ⊕ La combinazione di diverse istruzioni **wait** permette di definire comportamenti complessi
- ❑ Il processo è eseguito interamente o interrotto nel caso si incontra un'istruzione wait
- ❑ I segnali sono aggiornati solo al termine dell'esecuzione **con l'ultimo valore assegnato**
- ❑ L'esecuzione al suo interno è strettamente sequenziale

# Wait

---

- Sospende il processo in attesa di un evento

- ⊕ **wait;**

- ⊕ Sospende il processo a tempo indefinito
    - ⊕ Utile in fase di test

- ⊕ **wait for time;**

- ⊕ Sospende il processo per il tempo specificato

- ⊕ **wait on lista segnali;**

- ⊕ Sospende il processo fino alla variazione di uno dei segnali elencati

- ⊕ **wait until condizione;**

- ⊕ Sospende il processo fino a quando una variazione dei segnali rende la condizione vera

# Wait

---

- ❑ Wait on `lista_segnali`; posto come ultima istruzione di un processo equivale alla specifica della lista di sensibilità
- ❑ Esempio:

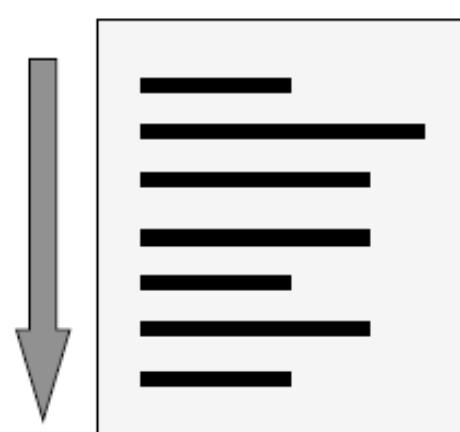
```
process
    begin
        if (A = B) then
            C <= '1' after 1 ns;
        else
            C <= '0' after 1 ns;
        end if;
        wait on a, b;
    end process;
```

# Istruzioni Sequenziali

- Eseguite in sequenza (una alla volta)
- Il loro comportamento dipende dall'ordine con il quale sono state scritte nel codice VHDL
- Esempio: un processo contiene istruzioni sequenziali (IF, CASE, ...):

```
MUX: process (A, B, SEL)
begin
    if (SEL = '1') then
        Z <= A;
    else
        Z <= B;
    end if;
end process MUX;
```

sensitivity list



# Costrutto condizionale if

---

- Consente il controllo condizionato di istruzioni sequenziali basato sulla valutazione di un valore booleano.

- Sintassi:

```
if condizione1 then
    istruzioni sequenziali;
{elsif condizione2 then
    altre istruzioni sequenziali;}
[else
    ultime istruzioni sequenziali;]
end if;
```

- Esempio:

```
if a > 0 then
    b <= '0';
else
    b <= '1';
end if;
```

- L'istruzione IF verifica una condizione ed esegue diverse istruzioni in base al risultato.

# elseif

---

- Consente il controllo condizionato di alternative multiple basato sulla valutazione di condizioni multiple.
  - Esempio:

```
if A = '1' then
    ONES_COUNT := ONES_COUNT + 1;
elsif A = '0' then
    ZEROS_COUNT := ZEROS_COUNT + 1;
else
    OTHERS_COUNT := OTHERS_COUNT + 1;
end if;
```

- In presenza di IF annidati viene eseguita la serie di istruzioni sequenziali corrispondenti alla prima condizione che risulti vera  
⇒ Importante l'ordine di scrittura delle condizioni ⇒ PRIORITÀ
- Più di una condizione può risultare vera ⇒ viene eseguita la prima condizione che risulta vera.

# Costrutto condizionale case

---

- Consente il controllo dell'esecuzione di istruzioni sequenziali basato sulla valutazione del valore assunto da un oggetto.

- Sintassi:

```
case espressione is
    when scelta1 => istruzioni sequenziali;
    when scelta2 => istruzioni sequenziali;
    [when others => ultime istruzioni
                           sequenziali;]
end case;
```

- Esempio:

```
case muxval is
    when 0 => q <= i0 after 10 ns;
    when 1 => q <= i1 after 10 ns;
    when scelta2 => istruzioni sequenziali;
    when others => null;
end case;
```

# Costrutto condizionale case

---

- ⊕ Equivale a **with ... select**
- ⊕ Tutte le scelte devono essere incluse
  - ⊕ E' possibile utilizzare **others**
- ⊕ Le scelte non possono sovrapporsi
- ⊕ E' possibile definire degli intervalli (1 to 4, ...)

Outside Processes	Inside Processes
WHEN..ELSE	IF..ELSIF..ELSE..END IF
WITH..SELECT..WHEN	CASE..WHEN..END CASE

# Costrutto condizionale case

- Esempio di processo contenente il costrutto CASE:

```
process (A, B, C, X)
begin
    case X is
        when 0 to 4 =>
            Z <= B;
        when 5 =>
            Z <= C;
        when 7 | 9 =>
            Z <= A;
        when others =>
            Z <= 0;
    end case;
end process;
```

The diagram shows a grey rectangular area containing the VHDL code. Two arrows originate from the right side of the slide and point to specific parts of the code. One arrow points to the 'when 0 to 4' clause, with the word 'range' written next to it. Another arrow points to the 'when 7 | 9' clause, with the word 'list' written next to it.

# Esempio di processo

---

```
mux: process (i0, i1, i2, i3, a, b)
variable muxval: integer range 0 to 4;
begin
    muxval := 0;
    if(a='1') then
        muxval := muxval +1;
    end if;
    if(b='1') then
        muxval := muxval +2;
    end if;
    case muxval is
        when 0 => q <=i0 after 10 ns;
        ...
    end case;
end process;
```

# Cicli for

- Consente l'esecuzione ripetuta di una sequenza di istruzioni.

```
[etichetta:] for indice in intervallo loop  
    istruzioni sequenziali;  
end loop [etichetta];
```

- indice è intero e non può essere modificato all'interno del ciclo
- intervallo è di tipo enumerativo

```
for I in 1 to 10 loop  
    I_SQUARED(I) := I * I;  
end loop;
```

```
exit [etichetta:] [when condizione];
```

- Termina l'esecuzione del ciclo specificato (anche se non il più interno)

```
next [etichetta:] [when condizione];
```

- Passa alla prossima iterazione

# Cicli while

---

```
[etichetta:] while condizione loop  
    istruzioni sequenziali;  
end loop [etichetta];
```

- La condizione è valutata prima di ogni iterazione

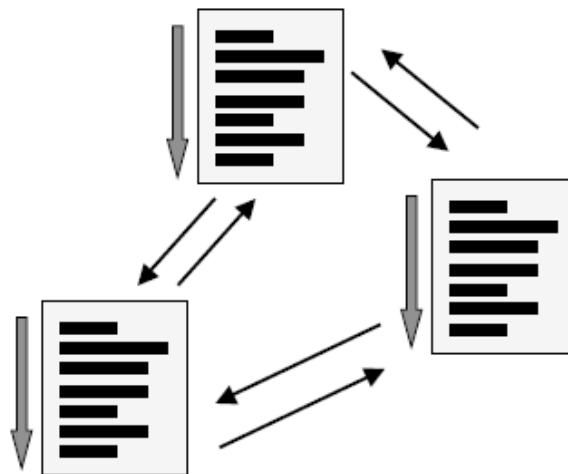
```
while (DAY = WEEK_DAY) loop  
    DAY := GET_NEXT_DAY(DAY);  
end loop;
```

- Si possono utilizzare le istruzioni **next** e **exit**

# Concorrenza tra Processi

Un modello VHDL è un insieme di PROCESSI che interagiscono tra loro in parallelo (CONCORRENZA TRA PROCESSI ) e che si scambiano informazioni tramite i SEGNALI.

- Le istruzioni presenti all'interno di un processo sono eseguite in sequenza, mentre processi multipli interagiscono in modalità concorrente.
- Un processo viene eseguito quando avviene un EVENTO nella sua SENSITIVITY LIST. L'esecuzione di un processo causa uno o più eventi alle sue uscite. Tali eventi possono attivare altri processi, ecc.



# Processi Multipli Concorrenti

---

- All'interno di un'architettura possono essere presenti processi multipli concorrenti e istruzioni concorrenti: ad esempio

```
architecture A of ENTITY is
begin
    -- concurrent statements
    P1: process (sensitivity list of signals)
        begin
            -- sequential statements;
        end process P1;
    -- concurrent statements
    P2: process (sensitivity list of signals)
        begin
            -- sequential statements
        end process P2;
    -- concurrent statements
end A;
```

# Aree concorrenti e sequenziali

```
architecture archi of circuito is
  -- istruzione concorrente 1
  -- istruzione concorrente 2
begin
  pa: process
  begin
    -- istruzione sequenziale pa_1
    -- istruzione sequenziale pa_2
    -- ...
  end;

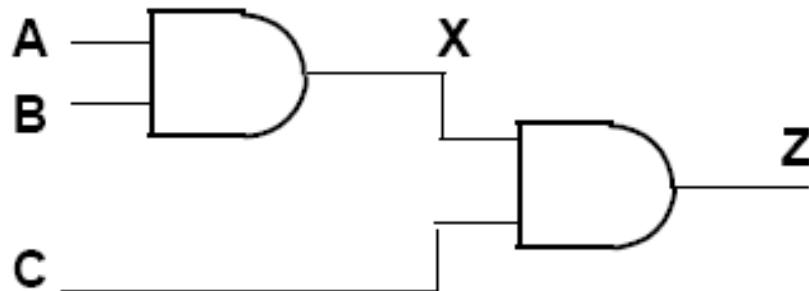
  pb: process
  begin
    -- istruzione sequenziale pa_2
    -- ...
  end;
end;
```

The diagram illustrates the structure of the VHDL code. A red vertical line on the left, labeled "Area concorrente", spans from the "begin" of process pa to the "end;" of the entire architecture. Inside process pa, two blue vertical lines, each labeled "Area sequenziale", span from the "begin" of pa to the "end;" of pa. Similarly, inside process pb, another blue vertical line, also labeled "Area sequenziale", spans from its "begin" to its "end;". This visual representation highlights the concurrent nature of processes pa and pb, and the sequential nature of the statements within each process.

# Istruzioni Concorrenti

- Eseguite in parallelo nello stesso istante di tempo.
- Il loro comportamento è indipendente dall'ordine con il quale sono scritte nel codice VHDL.
- Esempio: ISTRUZIONI DI ASSEGNAMENTO DEI SEGNALI:

```
X <= A and B;  
Z <= C and X;  
oppure  
Z <= C and X;  
X <= A and B;
```



- La struttura hardware corrispondente è intrinsecamente concorrente e composta dall'interconnessione di componenti elementari.
- Le attività sono svolte in parallelo dai diversi componenti.

# Sottoprogrammi (1/3)

---

- I sottoprogrammi sono utilizzati per descrivere calcoli, conversioni o altre funzionalità di comune utilizzo
- Sono descritti tramite un'insieme di **istruzioni sequenziali**
- Si possono definire **procedure** e **funzioni**
- Le procedure possono generare **effetti collaterali**
  - ▶ Modificano il valore dei parametri di uscita o i segnali visibili
- Le funzioni producono un solo risultato e non possono causare effetti collaterali

# Sottoprogrammi (2/3)

---

- Esempio di procedura

- ▶ Dichiarazione:

```
procedure min(a,b: in integer; c: out integer);
```

- ▶ Corpo

```
procedure min(a,b: in integer; c: out integer) is
Begin
```

```
    if a<b then
```

```
        c:=a;
```

```
    else
```

```
        c:=b;
```

```
    end if
```

```
End min;
```

# Sottoprogrammi (3/3)

---

## □ Esempio di funzione

► Dichiarazione:

```
Function min(a,b:integer) return integer;
```

► Corpo:

```
Function min(a,b:integer) return integer is
Begin
    if a<b then
        return a;
    else
        return b;
    end if
End min;
```

---

# Esempi di codice VHDL

# Processo combinatorio

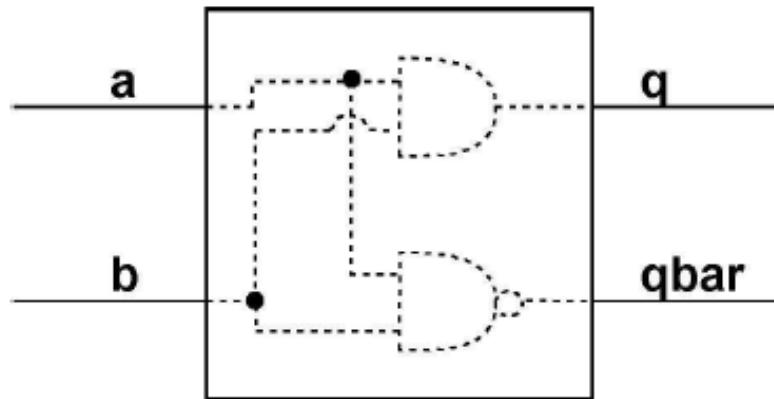


```
process (sensitivity_list_of_inputs)
begin
    -- default assignments;
    -- combinatorial logic assignments;
end process;
```



```
MUX: process (A, B, SEL)
begin
    if (SEL = '1') then
        Z <= A;
    else
        Z <= B;
    end if;
end process MUX;
```

# Processo combinatorio – Esempio 1



```
ENTITY andnand IS
  PORT (      a      : IN    std_logic;
              b      : IN    std_logic;
              q      : OUT   std_logic;
              qbar   : OUT   std_logic );
END andnand;
ARCHITECTURE synthesis1 OF andnand IS
BEGIN
  q    <= a AND b;
  qbar<= a NAND b;
END synthesis1;
```

# Processo combinatorio – Esempio 2

---

```
SIGNAL a, b, c, d          :std_logic;

PROCESS (a, b, d)
-- a, b, and d are in the sensitivity list to indicate that
-- the outputs of the process are sensitive to changes in them
BEGIN
    -- CASE keyword is only valid in a process
    CASE d IS
        WHEN '0' =>
            c <= a AND b;
        WHEN OTHERS =>
            c <= '1';
    END CASE;
END PROCESS;
```

**NOTE:**

This implements a combinational circuit.

# Processo combinatorio – Esempio 3

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

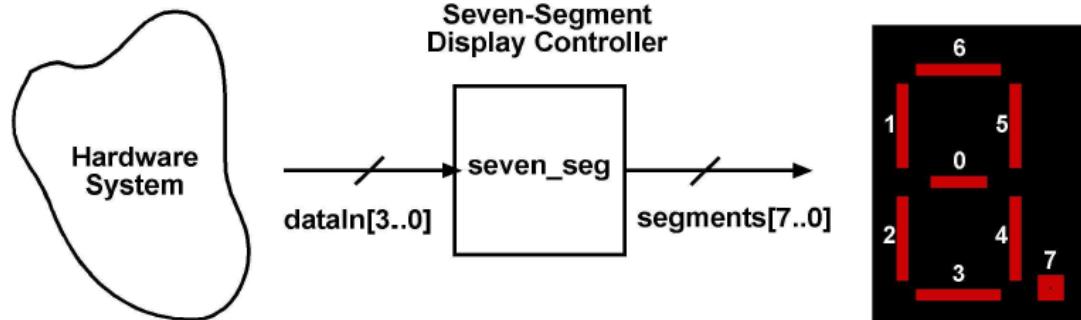
```
ENTITY seven_seg IS
    PORT( dataIn           : IN      std_logic_vector(3 DOWNTO 0);
          segments        : OUT     std_logic_vector(7 DOWNTO 0) );
END seven_seg;
```

```
ARCHITECTURE synthesis1 OF seven_seg IS
```

```
BEGIN
```

```
    WITH dataIn SELECT
        segments <=
            "10000001" WHEN "0000",    -- 0
            "11001111" WHEN "0001",    -- 1
            "10010010" WHEN "0010",    -- 2
            "10000110" WHEN "0011",    -- 3
            "11001100" WHEN "0100",    -- 4
            "10100100" WHEN "0101",    -- 5
            "10100000" WHEN "0110",    -- 6
            "10001111" WHEN "0111",    -- 7
            "10000000" WHEN "1000",    -- 8
            "10000100" WHEN "1001",    -- 9
            "11111111" WHEN OTHERS;
```

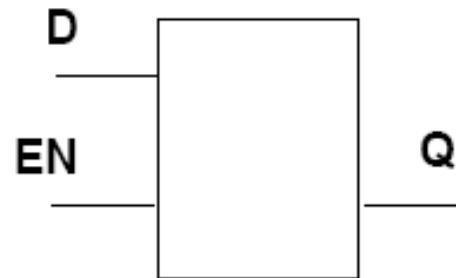
```
END synthesis1;
```



# D Latch

---

```
entity LATCH is
Port ( EN, D : in std_ulogic;
        Q: out std_ulogic);
end LATCH;
architecture A of LATCH is
begin
process (EN, D)
begin
    if (EN = '1') then
        Q <= D;
    end if;
end process;
end A;
```

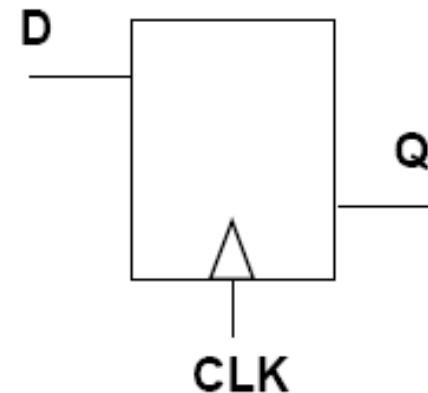


# D Flip-Flop

- The `EVENT` attribute can be used to check for the rising edge of a clock signal

```
entity FLOP is
port (D, CLK : in std_ulogic;
      Q      : out std_ulogic);
end FLOP;

architecture A of FLOP is
begin
  process
  begin
    wait until (CLK'event and CLK='1');
    Q <= D;
  end process;
end A;
```

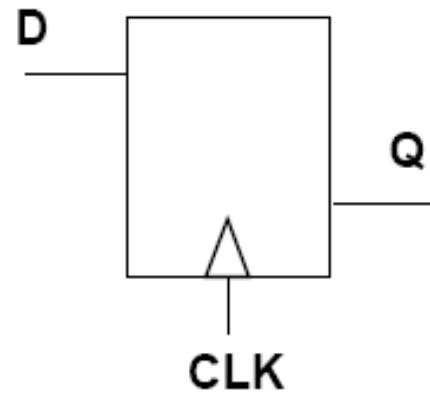


The `EVENT` attribute is true if an edge has been detected on the corresponding signal.

# D Flip-Flop

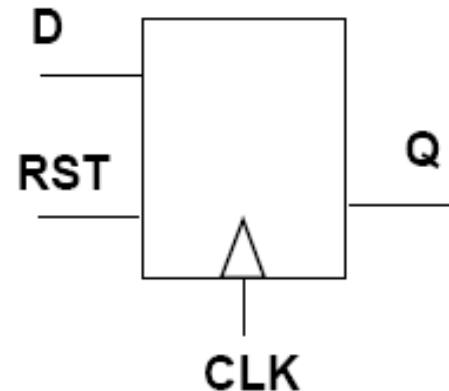
```
entity FLOP is
port (D, CLK : in std_ulogic;
      Q      : out std_ulogic);
end FLOP;

architecture B of FLOP is
begin
  process (CLK)
  begin
    if (CLK'event and CLK='1') then
      Q <= D;
    end if; -- no else clause
  end process;
end B;
```



# D Flip-Flop

```
entity FLOP is
port (D, CLK, RST : in std_ulogic;
      Q      : out std_ulogic);
end FLOP;
architecture A of FLOP is
begin
process (CLK, RST)
begin
  if (RST = '1') then
    Q <= 0;
  elsif (CLK'event and CLK='1') then
    Q <= D;
  end if; -- no else clause
end process;
end A;
```

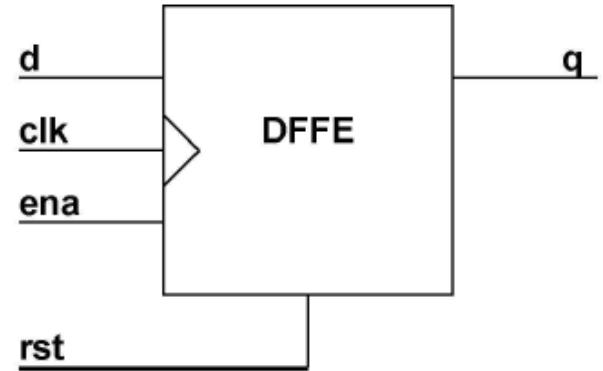


# D Flip-Flop

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dffe IS
    PORT(rst, clk, ena, d : IN      std_logic;
          q           : OUT      std_logic );
END dffe;

ARCHITECTURE synthesis1 OF dffe IS
BEGIN
    PROCESS (rst, clk)
    BEGIN
        IF (rst = '1') THEN
            q <= '0';
        ELSIF (clk'EVENT) AND (clk = '1') THEN
            IF (ena = '1') THEN
                q <= d;
            END IF;
        END IF;
    END PROCESS;
END synthesis1;
```

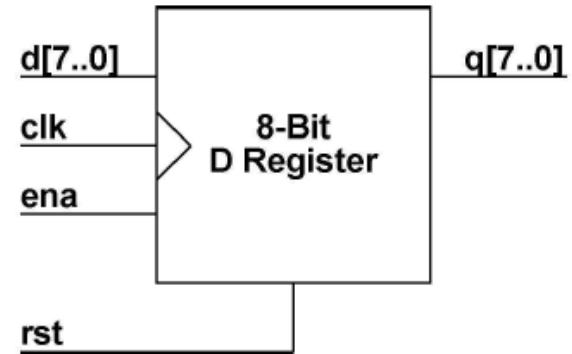


# 8-bit Register

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;
```

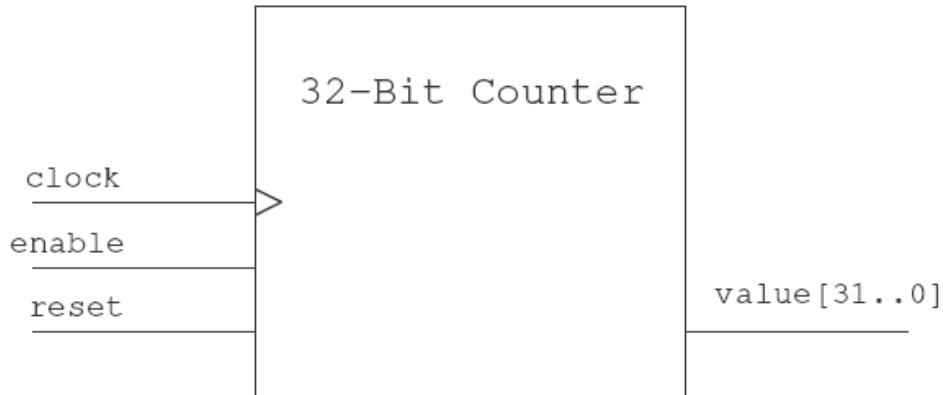
```
ENTITY dregister IS  
    PORT( rst, clk, ena : IN      std_logic;  
          d           : IN      std_logic_vector(7 DOWNTO 0);  
          q           : OUT     std_logic_vector(7 DOWNTO 0) );  
END dregister;
```

```
ARCHITECTURE synthesis1 OF dregister IS  
BEGIN  
    PROCESS (rst, clk)  
    BEGIN  
        IF (rst = '1') THEN  
            q <= X"00";  
        ELSIF (clk'EVENT) AND (clk = '1') THEN  
            IF (ena = '1') THEN  
                q <= d;  
            END IF;  
        END IF;  
    END PROCESS;  
END synthesis1;
```



# 32-bit Counter (1/2)

---



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY counter IS
  PORT (
    reset          : IN      std_logic;
    clock          : IN      std_logic;
    enable         : IN      std_logic;
    value          : OUT     std_logic_vector(31 DOWNTO 0)
  );
END counter;
```

# 32-bit Counter (2/2)

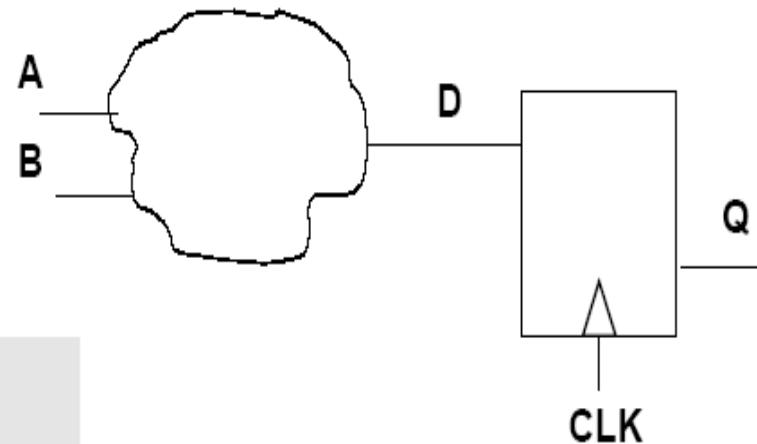
---

```
ARCHITECTURE synthesis1 OF counter IS
    SIGNAL count : unsigned(31 DOWNTO 0);          -- The unsigned type is used
                                                    -- so that unsigned arithmetic
                                                    -- will be synthesized

BEGIN
    PROCESS (reset, clock)
        BEGIN
            IF (reset = '1') THEN
                count <= X"00000000";
            ELSIF (clock'EVENT) AND (clock = '1') THEN
                IF (enable = '1') THEN
                    count <= count + 1;
                END IF;
            END IF;
        END PROCESS;                                -- Here, the count value is
                                                    -- converted to std_logic_vector
                                                    -- using a conversion function
        value <= std_logic_vector(count);
END synthesis1;
```

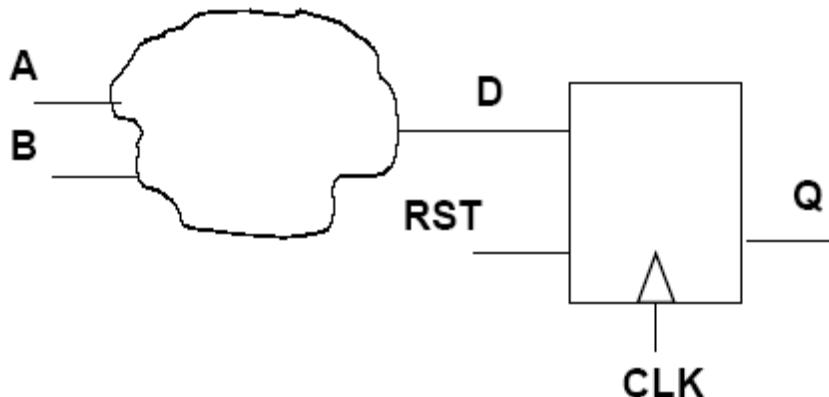
# Esempio di descrizione RTL (1/4)

```
process -- clocked process with no reset
begin
    wait until (CLK'event and CLK = '1');      ←
        Q <= A + B;
        -- all combinatorial logic assignments here;
end process;
```



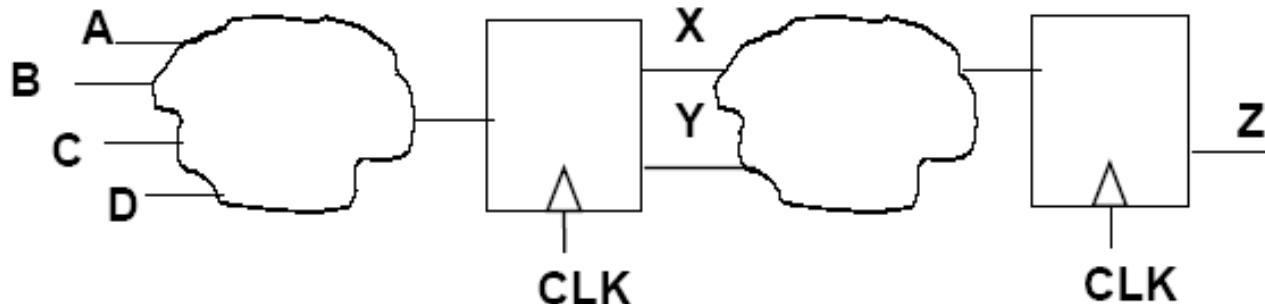
```
process (CLK) --clocked process with no reset
begin
    if (CLK'event and CLK = '1') then      ←
        Q <= A + B;
        -- all combinatorial logic assignments here;
    end if; -- no else clause
end process;
```

# Esempio di descrizione RTL (2/4)



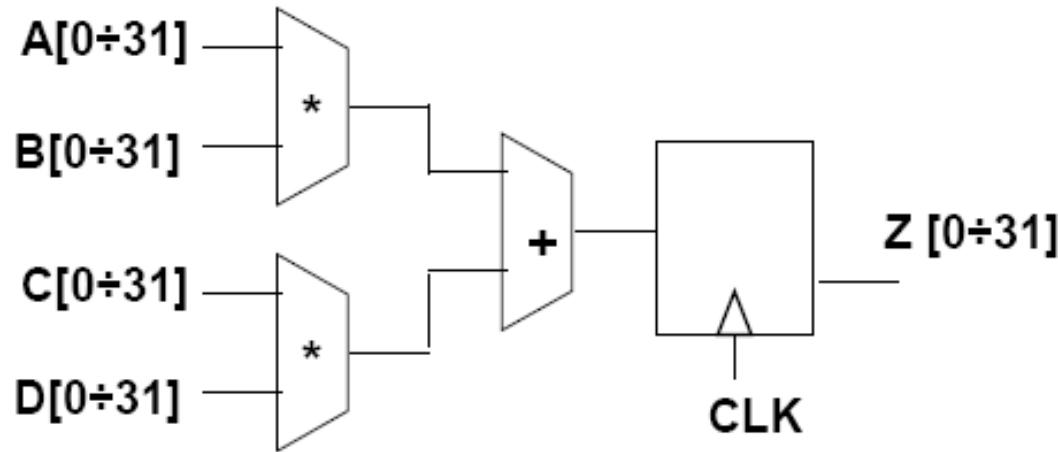
```
process (CLK, RST) - clocked process with asynch. reset
begin
    if (RST = '1') then
        Q <= 0;
    elsif (CLK'event and CLK = '1') then
        Q <= A + B;
        -- all combinatorial logic assignments here;
    end if; -- no else clause
end process;
```

# Esempio di descrizione RTL (3/4)



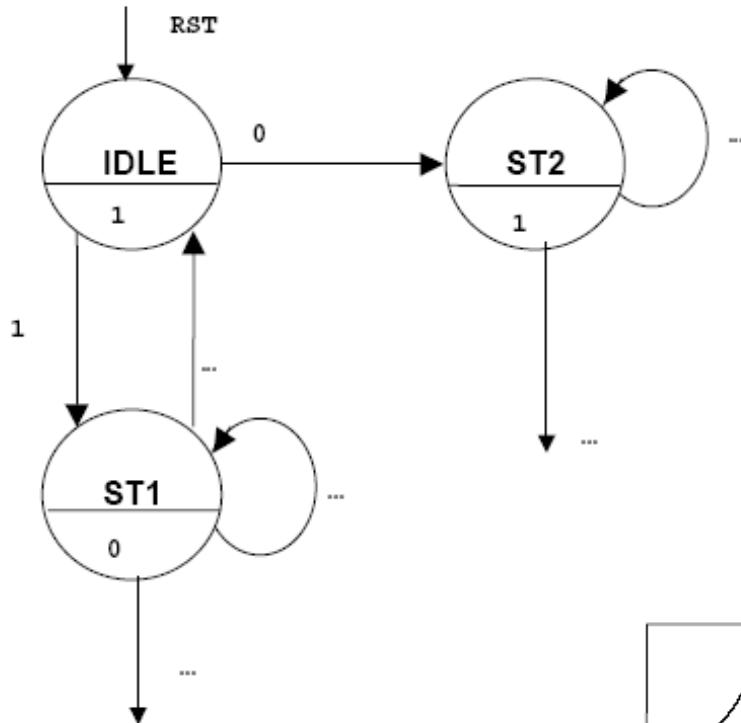
```
process
begin
    wait until (CLK'event and CLK = '1');
    X <= A + B;
    Y <= C + D;
    Z <= X + Y;
end process;
```

# Esempio di descrizione RTL (4/4)



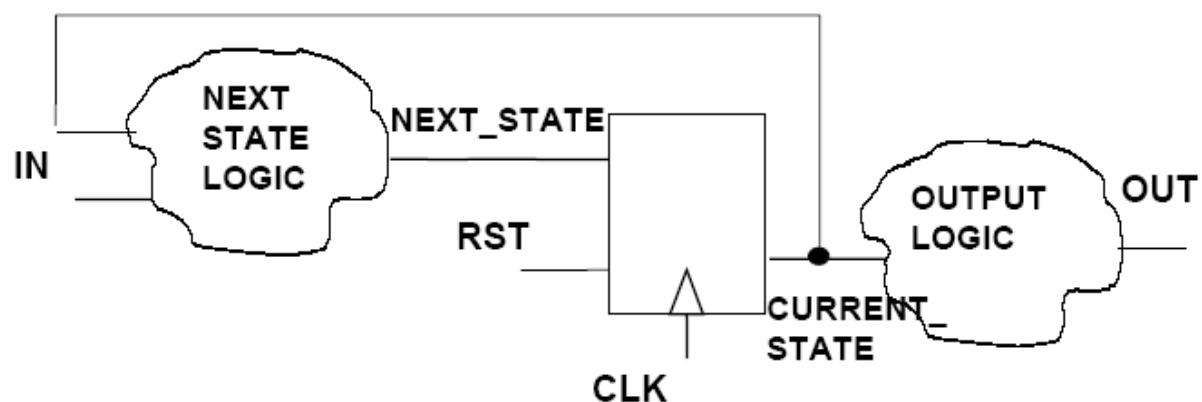
```
process
begin
    wait until (CLK'event and CLK = '1');
    Z <= (A * B) + (C * D)
end process;
```

# Macchine a stati finiti – Tipo Moore (1/5)



Si rappresenta lo stato presente e l' uscita corrispondente

Le frecce indicano l' evoluzione del circuito a seguito di un impulso di clock



# Macchine a stati finiti – Tipo Moore (2/5)

```
entity FSM is
    port ( RST, CLK, IN : in    bit;
           OUT : out   bit);

end FSM;
architecture MOORE of FSM is
    type STATE_TYPE is (IDLE, ST1, ST2, ...);
    signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
-----
-- Processo sequenziale
-----
    SEQ: process (CLK, RST)
begin
    if RST = '1' then
        CURRENT_STATE  <= IDLE;
    elsif (CLK'event and CLK = '1') then
        CURRENT_STATE <= NEXT_STATE;
    end if;
end process SEQ;
```

# Macchine a stati finiti – Tipo Moore (3/5)

```
-- Processo Combinatorio per Next State Logic
-----
COMB: process (CURRENT_STATE, IN)
begin
    case CURRENT_STATE is
        when IDLE =>
            if (IN='1') then
                NEXT_STATE <= ST1;
            else
                NEXT_STATE <= ST2;
            end if;
        when ST1 =>
            if (...) then
                NEXT_STATE <= ...;
            else
                NEXT_STATE <= ...;
            end if;
        when ... =>
            if (...) then
                NEXT_STATE <= ...;
            else
                NEXT_STATE <= ...;
            end if;
    end case;
end process COMB;
```

# Macchine a stati finiti – Tipo Moore (4/5)

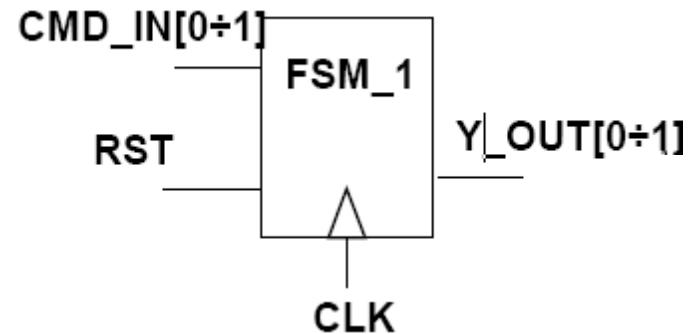
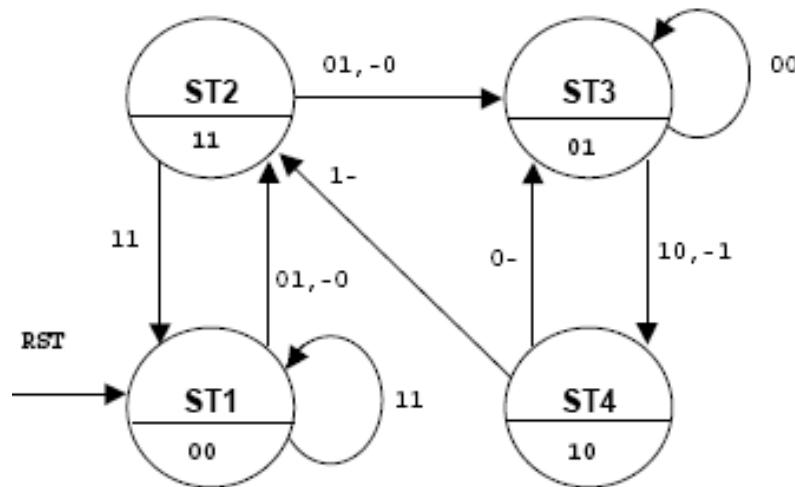
---

```
-----  
-- Processo Combinatorio per Output Logic  
-----  
      OUT_LOGIC: process (CURRENT_STATE)  
begin  
    case CURRENT_STATE is  
    when IDLE =>  
        OUT <= '1';  
    when ST1 =>  
        OUT <= '0';  
    when ... =>  
        OUT<= ...;  
    end case;  
end process OUT_LOGIC;  
end MOORE;
```

# Macchine a stati finiti – Tipo Moore (5/5)

```
-- Processo Combinatorio per Next State Logic e Output Logic
-----
COMB_OUT:process (CURRENT_STATE, IN)
begin
    case CURRENT_STATE is
        when IDLE =>
            OUT <= '1';
            if (IN='1') then
                NEXT_STATE <= ST1;
            else
                NEXT_STATE <= ST2;
            end if;
        when ST1 =>
            OUT <= '0';
            if (...) then
                NEXT_STATE <= ...;
            else
                NEXT_STATE <= ...;
            end if;
        when ... =>
            ...
    end case;
end process COMB_OUT;
end MOORE;
```

# Macchina di Moore – Esempio 1



	00	01	11	10	Y_OUT
ST1	ST2	ST2	ST1	ST2	00
ST2	ST3	ST3	ST1	ST3	11
ST3	ST3	ST4	ST4	ST4	01
ST4	ST3	ST3	ST2	ST2	10

# Macchina di Moore – Esempio 1

```
entity FSM_1 is
    port ( RST, CLK : in    bit;
           CMD_IN : in    bit_vector (0 to 1);
           Y_OUT : out   bit_vector (0 to 1));

end FSM2;
architecture MOORE of FSM_1 is
    type STATE_TYPE is ( ST1, ST2, ST3, ST4);
    signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
-----
-- Processo sequenziale
-----
    SEQ: process (CLK, RST)
    begin
        if RST = '1' then
            CURRENT_STATE <= ST1;
        elsif (CLK'event and CLK = '1') then
            CURRENT_STATE <= NEXT_STATE;
        end if;
    end process SEQ;
```

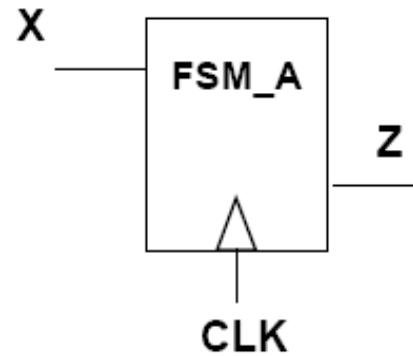
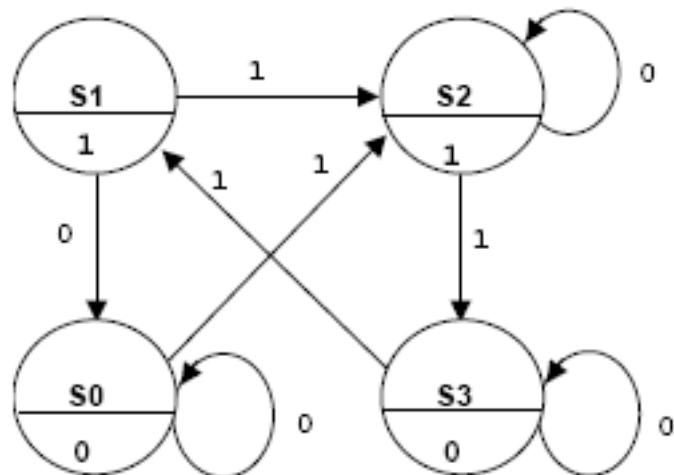
# Macchina di Moore – Esempio 1

```
-- Processo Combinatorio per Next State Logic
-----
COMB: process (current_state,CMD_IN)
begin
    case CURRENT_STATE is
        when ST1 =>
            if (CMD_IN="00" or CMD_IN="01" or CMD_IN="10") then
                NEXT_STATE <= ST2;
            else
                NEXT_STATE <= ST1;
            end if;
        when ST2 =>
            if (CMD_IN="00" or CMD_IN="01" or CMD_IN="10") then
                NEXT_STATE <= ST3;
            else
                NEXT_STATE <= ST1;
            end if;
        when ST3 =>
            if (CMD_IN="01" or CMD_IN="10" or CMD_IN="11") then
                NEXT_STATE <= ST4;
            else
                NEXT_STATE <= ST3;
            end if;
    end case;
end process COMB;
```

# Macchina di Moore – Esempio 1

```
when ST4 =>
    if ( CMD_IN="00" or CMD_IN="01") then
        NEXT_STATE <= ST3;
    else
        NEXT_STATE <= ST2;
    end if;
end case;
end process COMB;
-----
-- Processo Combinatorio per Output Logic
-----
OUT_LOGIC: process (CURRENT_STATE)
begin
    case CURRENT_STATE is
when ST1 =>
    Y_OUT <= "00";
when ST2 =>
    Y_OUT <= "11";
when ST3 =>
    Y_OUT <= "01";
when ST4 =>
    Y_OUT<= "10";
    end case;
end process OUT_LOGIC;
end MOORE;
```

# Macchina di Moore – Esempio 2



	X = 0	X = 1	Z
S0	S0	S2	0
S1	S0	S2	1
S2	S2	S3	1
S3	S3	S1	0

# Macchina di Moore – Esempio 2

```
entity FSM_A is
    port (  X, CLK : in  bit;
            Z: out   bit);

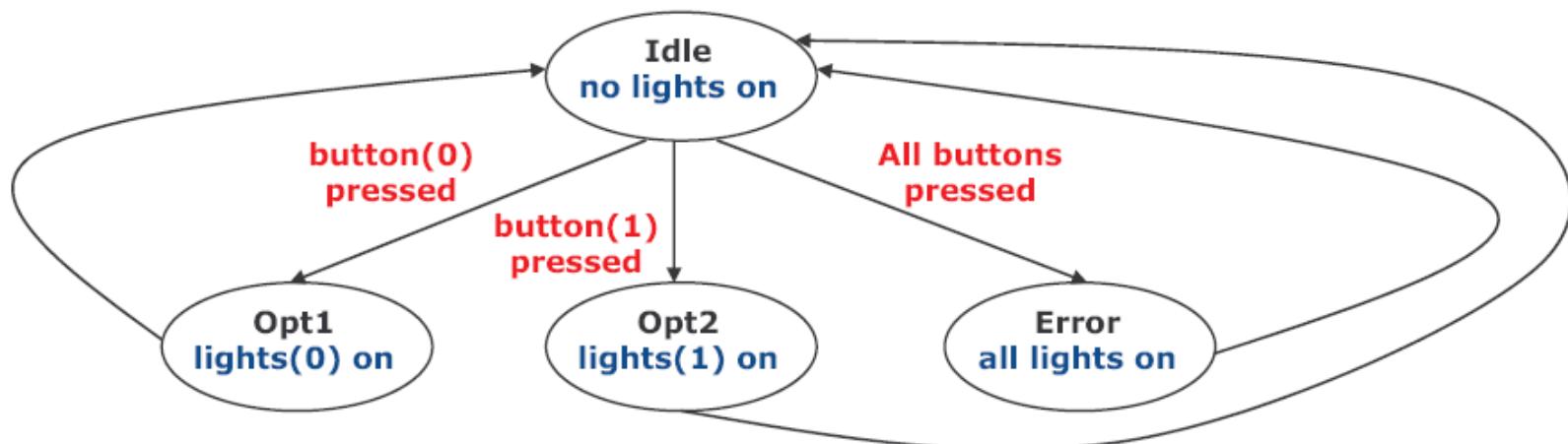
end FSM_A;
architecture MOORE of FSM_A is
    type STATE_TYPE is ( S0, S1, S2, S3);
    signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
    SEQ: process
    begin
        wait until(CLK'event and CLK = '1');
        CURRENT_STATE <= NEXT_STATE;
    end process SEQ;

    COMB_OUT:process (CURRENT_STATE, X)
    begin
        case CURRENT_STATE is
        when S0 =>
            Z <= '0';
            if  (X = '0') then
                NEXT_STATE <= S0;
            else
                NEXT_STATE <= S2;
            end if;
        end case;
    end process COMB_OUT;
end architecture;
```

# Macchina di Moore – Esempio 2

```
when S1 =>
    Z <= '1';
    if  (X = '0') then
        NEXT_STATE <= S0;
    else
        NEXT_STATE <= S2;
    end if;
when S2 =>
    Z <= '1';
    if  (X = '0') then
        NEXT_STATE <= S2;
    else
        NEXT_STATE <= S3;
    end if;
when S3 =>
    Z <= '0';
    if  (X = '0') then
        NEXT_STATE <= S3;
    else
        NEXT_STATE <= S1;
    end if;
end case;
end process COMB_OUT;
end MOORE;
```

# Macchina di Moore – Esempio 3



```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
  
ENTITY vending IS  
PORT(  
    reset : IN      std_logic;  
    clock : IN      std_logic;  
    buttons : IN    std_logic_vector(1 DOWNTO 0);  
    lights  : OUT    std_logic_vector(1 DOWNTO 0)  
);  
END vending;
```

# Macchina di Moore – Esempio 3

---

```
ARCHITECTURE synthesis1 OF vending IS
    TYPE statetype IS (Idle, Opt1, Opt2, Error);
    SIGNAL currentstate, nextstate : statetype;
BEGIN
    fsm1: PROCESS( buttons, currentstate )
    BEGIN
        CASE currentstate IS
            WHEN Idle =>
                lights <= "00";
                CASE buttons IS
                    WHEN "00" =>
                        nextstate <= Idle;
                    WHEN "01" =>
                        nextstate <= Opt1;
                    WHEN "10" =>
                        nextstate <= Opt2;
                    WHEN OTHERS =>
                        nextstate <= Error;
                END CASE;
            WHEN Opt1 =>
                lights <= "01";
                IF buttons /= "01" THEN
                    nextstate <= Idle;
                END IF;
```

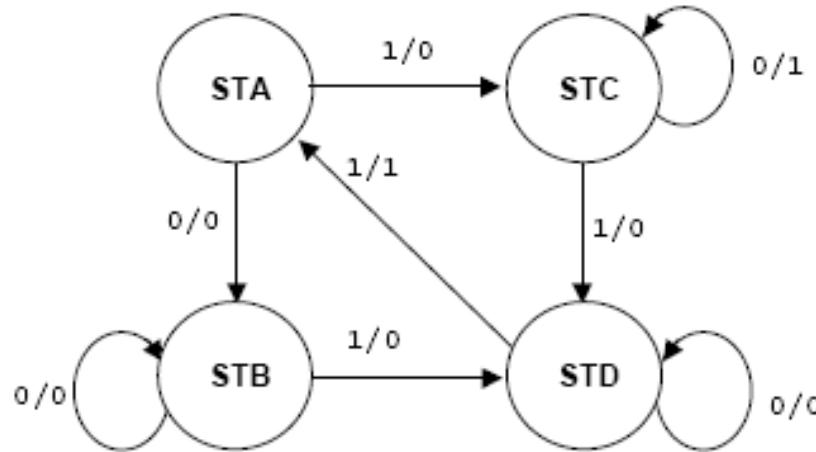
# Macchina di Moore – Esempio 3

---

```
        WHEN Opt2 =>
            lights <= "10";
            IF buttons /= "10" THEN
                nextstate <= Idle;
            END IF;
        WHEN Error =>
            lights <= "11";
            IF buttons = "00" THEN
                nextstate <= Idle;
            END IF;
    END CASE;
END PROCESS;

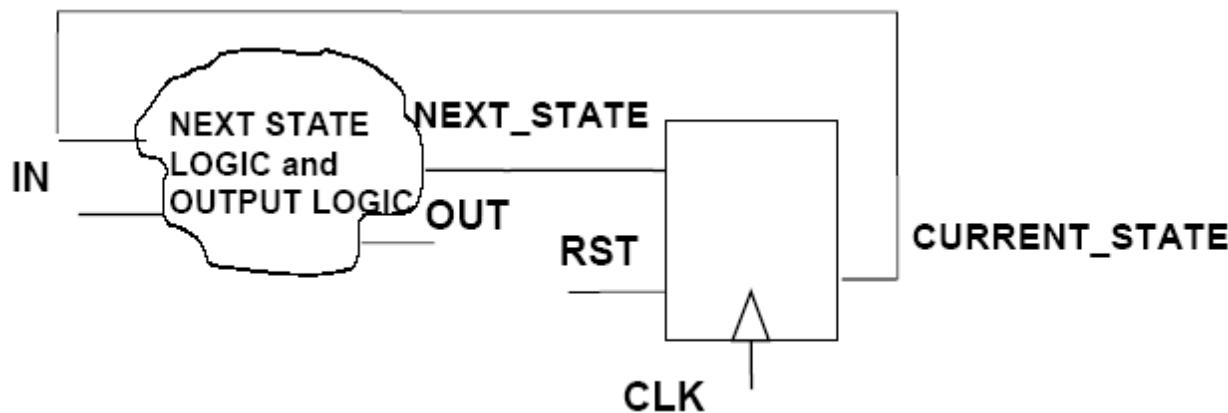
fsm2: PROCESS( reset, clock )
BEGIN
    IF (reset = '0') THEN
        currentstate <= Idle;
    ELSIF (clock'EVENT) AND (clock = '1') THEN
        currentstate <= nextstate;
    END IF;
END PROCESS;
END synthesis1;
```

# Macchine a stati finiti – Tipo Mealy

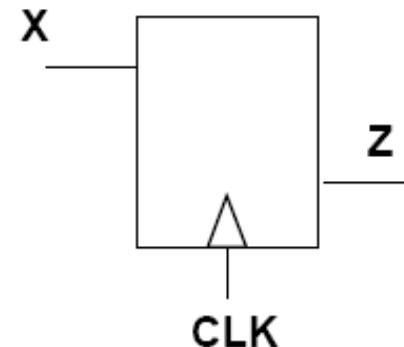
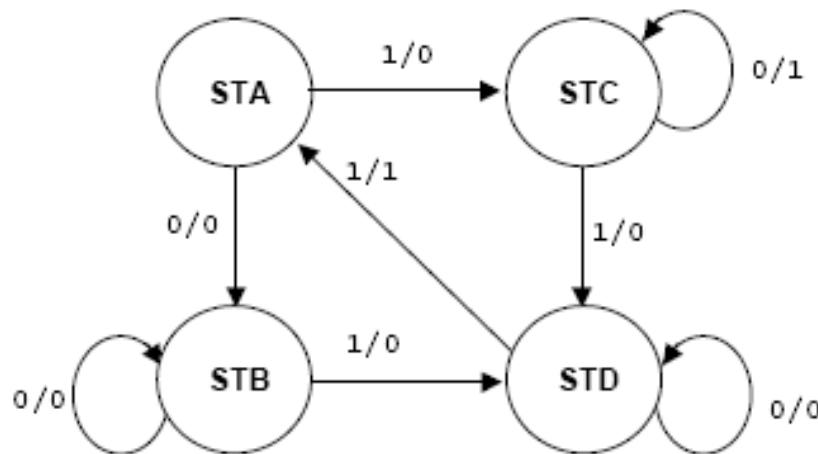


Si rappresenta solo lo stato presente

Le frecce indicano l'evoluzione del circuito e l'uscita corrispondente a seguito di un impulso di clock



# Macchina di Mealy – Esempio



	X = 0	X = 1
STA	STB / 0	STC / 0
STB	STB / 0	STD / 0
STC	STC / 1	STD / 0
STD	STD / 0	STA / 1

# Macchina di Mealy – Esempio

```
entity FSM_B is
    port ( X, CLK : in  bit;
           Z: out  bit);

end FSM_B;
architecture MEALY of FSM_B is
    type STATE_TYPE is ( STA, STB, STC, STD);
    signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
    SEQ: process
    begin
        wait until(CLK'event and CLK = '1');
        CURRENT_STATE <= NEXT_STATE;
    end process SEQ;

    COMB_OUT:process (CURRENT_STATE, X)
    begin
        case CURRENT_STATE is
        when STA =>
            if  (X = '0')  then
                Z <= '0';
                NEXT_STATE <= STB;
            else
                Z <= '0';
                NEXT_STATE <= STC;
            end if;
        end case;
    end process COMB_OUT;
end architecture;
```

# Macchina di Mealy – Esempio

```
when STB =>
    if  (X = '0') then
        Z <= '0';
        NEXT_STATE <= STB;
    else
        Z <= '0';
        NEXT_STATE <= STD;
    end if;
when STC =>
    if  (X = '0') then
        Z <= '1';
        NEXT_STATE <= STC;
    else
        Z <= '0';
        NEXT_STATE <= STD;
    end if;
when STD =>
    if  (X = '0') then
        Z <= '0';
        NEXT_STATE <= STD;
    else
        Z <= '1';
        NEXT_STATE <= STA;
    end if;
end case;
end process COMB_OUT;
end MEALY;
```

---

# VHDL TESTBENCHES

# Testbench (1/2)

---

- I moduli di test, chiamati testbench, sono specificati direttamente in VHDL
- Sono connessi al componente da testare
- Specificano gli stimoli in ingresso e collezionano i risultati
- Specificano gli stimoli in ingresso e collezionano i risultati
  - ▶ Queste due funzionalità possono essere eseguite direttamente dal simulatore
- Si possono definire dei puntatori a **files** (in processi) per poter caricare gli stimoli o salvare i risultati

# Testbench (2/2)

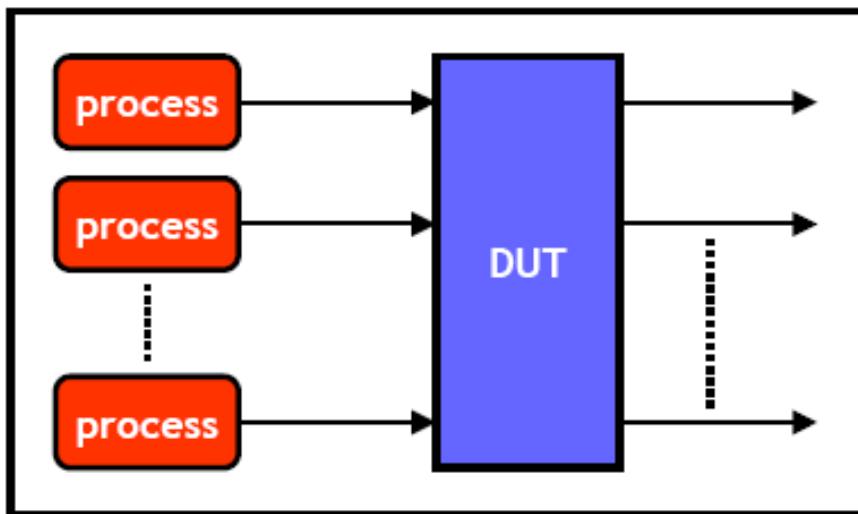
---

- Ha lo scopo di verificare la funzionalità di un circuito
  - ▶ In altre parole, modella gli aspetti salienti dell'ambiente in cui il circuito si troverà ad operare
- Per ottenere questo
  - ▶ Deve generare ingressi significativi
    - Valori significativi
    - Temporizzazione
  - ▶ Deve mostrare i valori delle uscite
  - ▶ Eventualmente, eseguire controlli di correttezza
    - Rispetto a valori noti

# Testbench: Struttura di base (1/3)

---

- Il testbench si limita a generare gli ingressi
- Il progettista controlla i valori di output analizzando le tracce ottenute in simulazione
  - ▶ Inapplicabile per problemi complessi



# Testbench: Struttura di base (2/3)

- I process per la generazione dei segnali di ingresso hanno la forma seguente

## Segnali aperiodici

```
GEN_X: process
begin

    X <= value1 ;
    wait for time1 ns;

    X <= value2 ;
    wait for time2 ns;

    ...

    X <= valueN ;
    wait;

end process
```

## Segnali periodici

```
GEN_X: process
begin

    X <= value1 ;
    wait for time1 ns;

    X <= value2 ;
    wait for time2 ns;

    ...

    X <= valueN ;
    wait for timeN ns;

end process
```

# Testbench: Struttura di base (3/3)

- Esempio
  - ▶ Segnali di clock e di reset

Reset

```
GEN_RESET: process
begin

    RESET <= '1';
    wait for 125 ns;

    RESET <= '0';
    wait;

end process
```

Clock

```
GEN_CLK: process
begin

    CLK <= '0';
    wait for 10 ns;

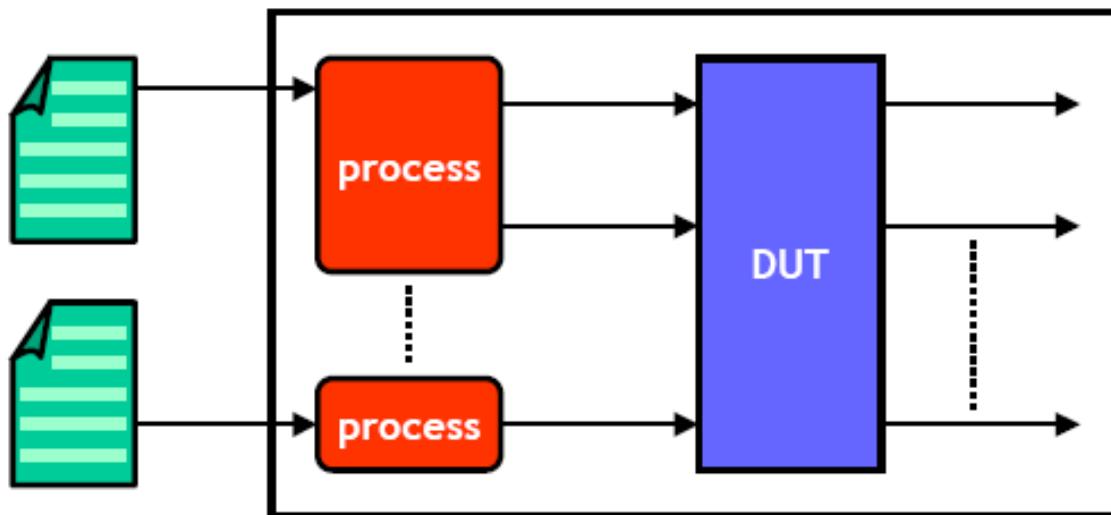
    CLK <= '1';
    wait for 10 ns;

end process
```

# Testbench: Lettura da file (1/4)

---

- Il testbench si limita a generare gli ingressi
  - ▶ I valori degli ingressi sono letti da file
- Il progettista controlla i valori di output analizzando le tracce ottenute in simulazione
  - ▶ Inapplicabile per problemi complessi



# Testbench: Lettura da file (2/4)

---

- Un process
  - ▶ Legge il/i file di dati in variabili temporanee
  - ▶ Converte tali valori in un tipo adatto
    - Solitamente std\_logic o std\_logic\_vector
  - ▶ Li assegna ai segnali d'ingresso del DUT
- Si hanno diverse possibilità
  - ▶ Un unico process genera clock e dati
  - ▶ Un process genera il clock, un altro i dati
  - ▶ Diversi process generano diversi dati
  - ▶ ...

# Testbench: Lettura da file (3/4)

- Esempio 1: temporizzazione esplicita
  - ▶ Lettura dei dati ogni 20 ns
  - ▶ **xb e yb sono segnali std\_logic\_vector a 8 bit**

```
READ_XY: process
    file      fpi:  text open read_mode is "in.dat";
    variable lni:  line;
    variable x, y: integer;
begin
    readline( fpi, lni );
    read( lni, x );
    xb <= conv_std_logic_vector( x, 8 );
    read( lni, y );
    yb <= conv_std_logic_vector( y, 8 );
    if endfile( fpi ) = true then
        wait;
    end if;
    wait for 20 ns;
end process;
```

in.dat

```
Xvalue1 Yvalue1
Xvalue2 Yvalue2
...
XvalueN YvalueN
```

# Testbench: Lettura da file (4/4)

- Esempio 2: temporizzazione implicita
  - ▶ Lettura dei dati in corrispondenza di un evento
  - ▶ **xb** e **yb** sono segnali `std_logic_vector` a 8 bit

```
READ_XY: process( clk )
  file      fpi:  text open read_mode is "in.dat";
  variable lni:  line;
  variable x, y: integer;
begin
  if clk'event and clk = '1' then
    readline( fpi, lni );

    read( lni, x );
    xb <= conv_std_logic_vector( x, 8 );

    read( lni, y );
    yb <= conv_std_logic_vector( y, 8 );

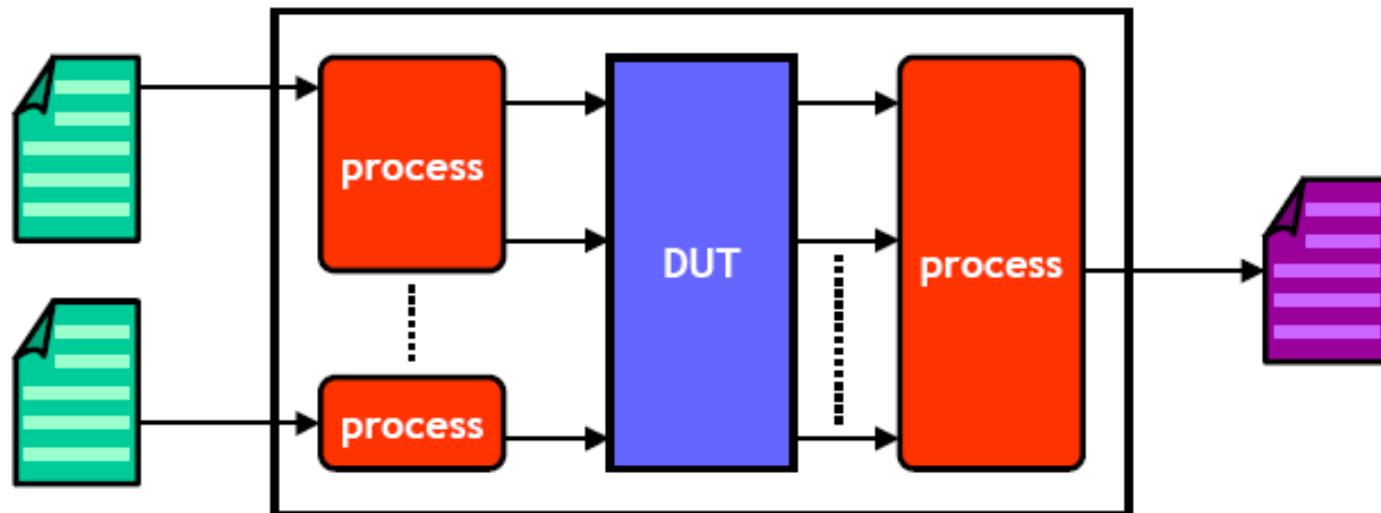
    if endfile( fpi ) = true then
      wait;
    end if;
  end if;
end process;
```

in.dat
Xvalue1 Yvalue1
Xvalue2 Yvalue2
...
XvalueN YvalueN

# Testbench: Lettura/Scrittura da file (1/2)

---

- Il testbench
  - ▶ Legge i valori degli ingressi da file
  - ▶ Scrive i valori delle uscite su file
- Il controllo è lasciato al progettista



# Testbench: Lettura/Scrittura da file (2/2)

- Esempio
  - ▶ Lettura dei dati secondo uno degli schemi visti
  - ▶ Scrittura dei dati in corrispondenza di un evento

```
WRITE_XY: process( clk )
    file      fpo:  text open write_mode is "out.dat";
    variable lno:  line;
    variable x, y: bit_vector(7 downto 0);
begin
    if clk'event and clk = '1' then
        x <= to_bitvector( xb );
        write( lno, x, "left", 10 );

        y <= to_bitvector( yb );
        write( lno, y, "left", 10 );

        writeline( fpo, ln0 );
    end if;
end process;
```

# Foreign Language Interface (1/3)

---

- I linguaggi di descrizione dell'hardware dispongono di un'interfaccia di programmazione per altri linguaggi
  - ▶ Tipicamente C/C++
  - ▶ Sono estensioni del linguaggio
  - ▶ Richiedono il supporto da parte del simulatore
- Per il linguaggio Verilog
  - ▶ PLI: Programming Language Interface
  - ▶ Schema semplice
- Per il linguaggio VHDL
  - ▶ FLI: Foreign Language Interface
  - ▶ Schema complesso, mutuato dal Verilog

# Foreign Language Interface (2/3)

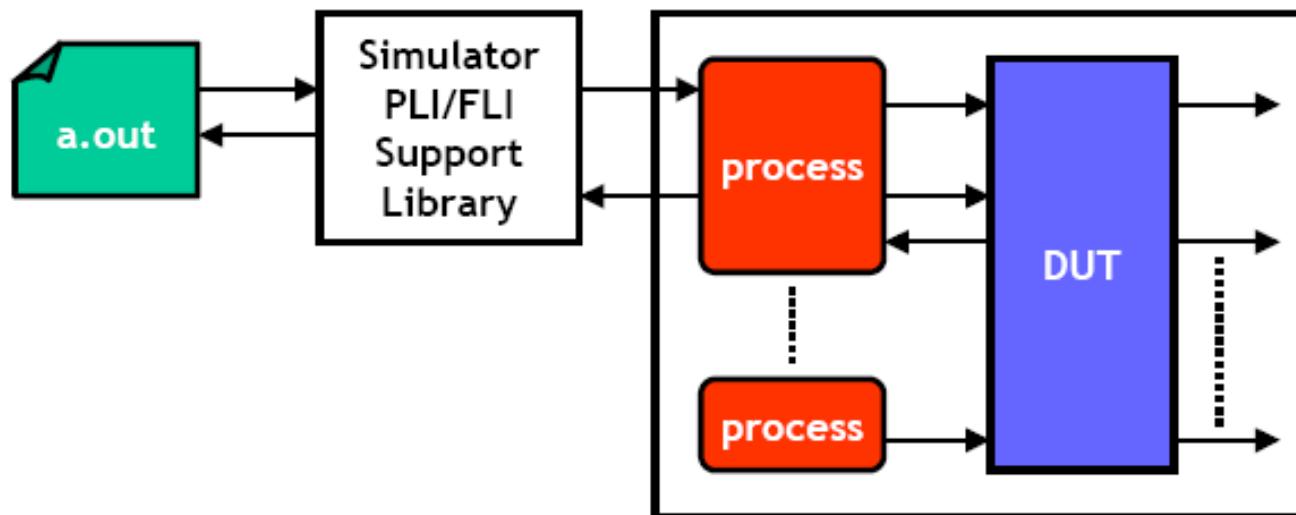
---

- Prescindendo dai dettagli specifici, il sistema PLI/FLI consente di
  - ▶ Dichiarare una interfaccia di funzione nel linguaggio HDL in uso (Verilog/VHDL)
  - ▶ Agganciare a tale interfaccia una funzione sviluppata in un altro linguaggio
    - Tipicamente C
  - ▶ Chiamare la funzione esterna
    - Il fatto che la funzione sia sviluppata in un linguaggio diverso è a questo punto indifferente
- Nella scrittura di test bench
  - ▶ Questo meccanismo è usato per generare dati

# Foreign Language Interface (3/3)

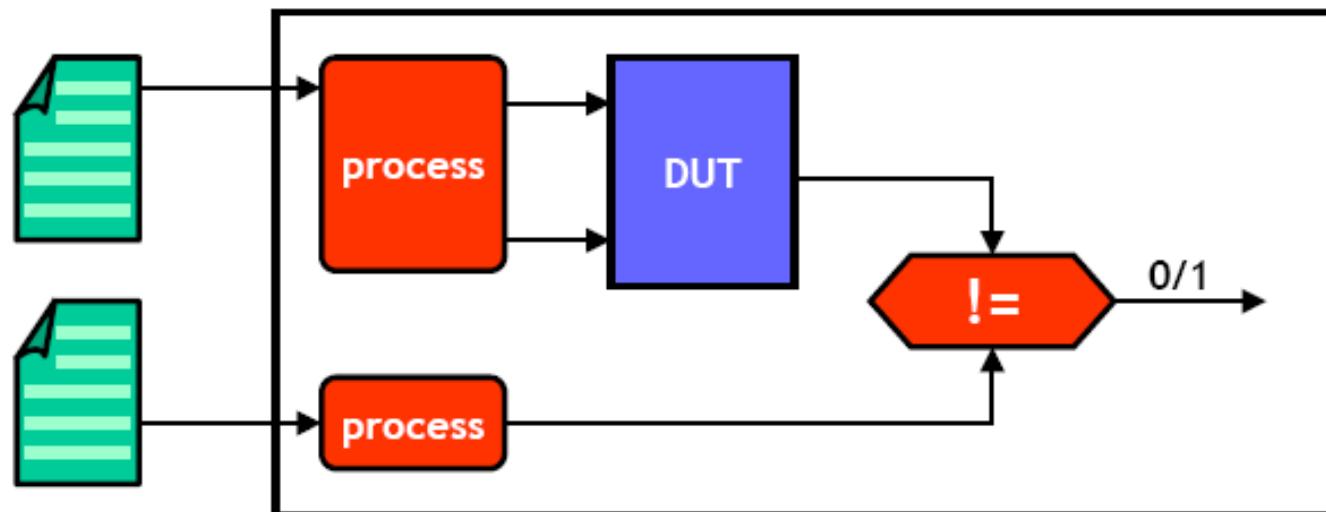
---

- Può sostituire la lettura da file
  - ▶ Quando i dati sono di dimensioni eccessive
  - ▶ Quando i dati in ingresso dipendono dalle uscite
- Lo schema di base è il seguente



# Testbench: Verifica automatica (1/3)

- Il testbench
  - ▶ Legge i valori degli ingressi da file
  - ▶ Legge i valori delle uscite attese da file
- Il controllo è effettuato automaticamente



# Testbench: Verifica automatica (2/3)

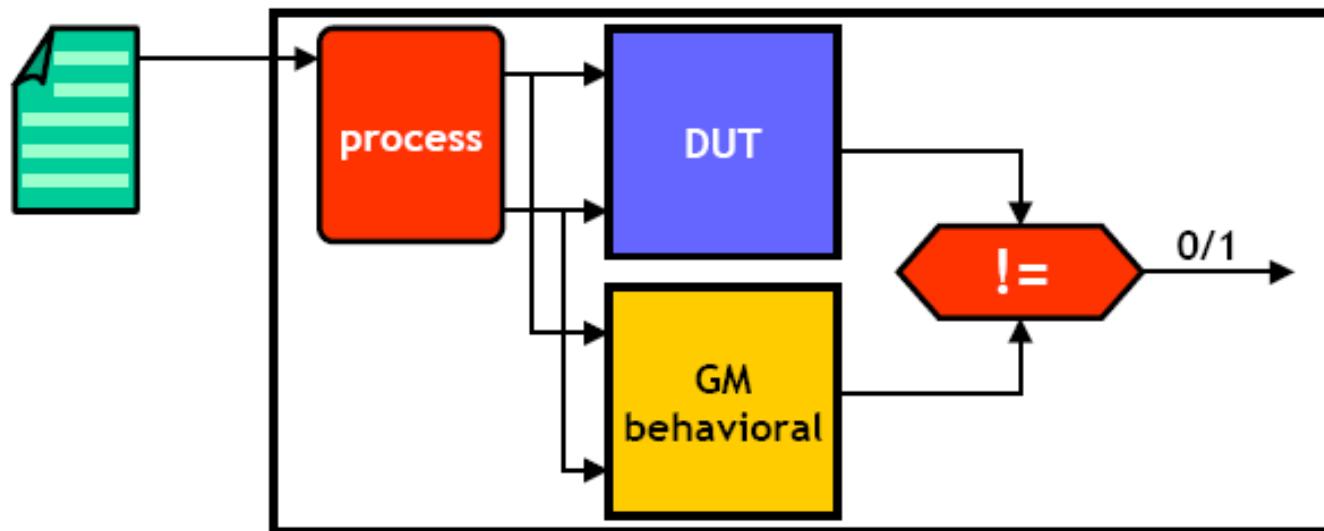
- Esempio

- ▶ Il controllo di correttezza avviene in un process sincronizzato da un opportuno evento
  - Tipicamente un clock
- ▶ In caso contrario il testbench può essere inusabile per la verifica del design dopo mapping e layout
  - La simulazione timing introduce ritardi di cui bisogna tenere conto

```
CHECK_XY: process( clk )
begin
    if clk'event and clk = '1' then
        if X /= Xexp or Y /= Yexp then
            XYerr <= '1';
        else
            XYerr <= '0';
        end if;
    end if;
end process;
```

# Testbench: Verifica automatica (3/3)

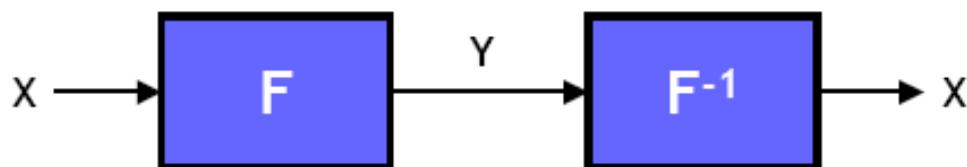
- Il testbench
  - ▶ Legge i valori degli ingressi da file
  - ▶ Fornisce gli ingressi a DUT e ad un golden model
- Il controllo è effettuato automaticamente



# Testbench: Verifica intrinseca (1/5)

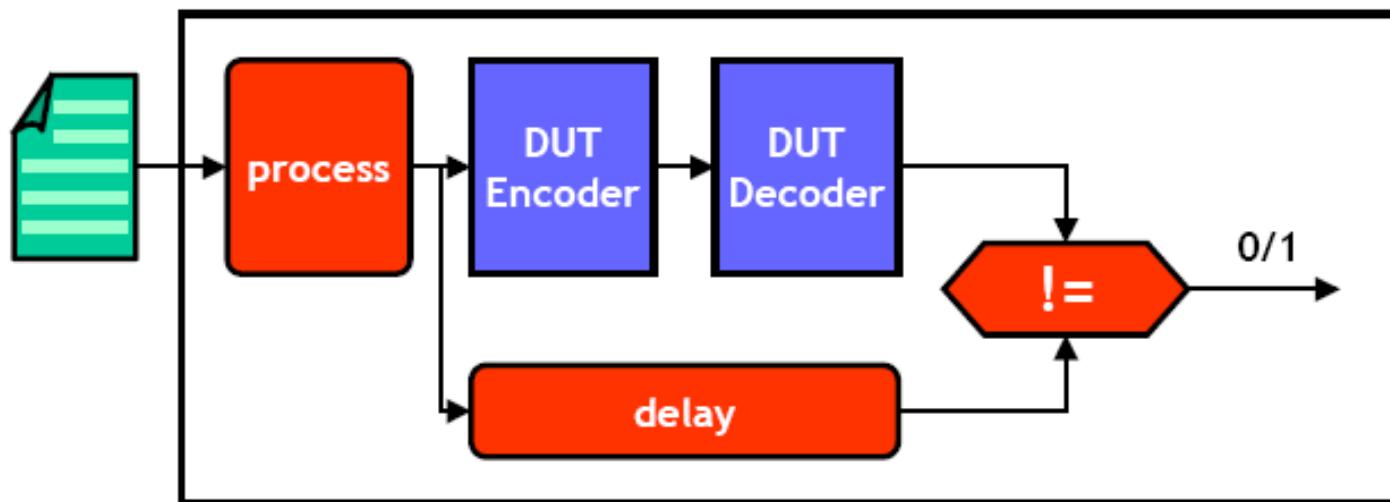
---

- Alcuni sistemi sono costituiti da due parti che svolgono funzionalità inverse
  - ▶ Codifica/decodifica
  - ▶ Compressione/decompressione
  - ▶ Modulazione/demodulazione
  - ▶ Trasformata/trasformata inversa
  - ▶ ...
- È possibile utilizzare una delle due sezioni per la verifica dell'altra e viceversa



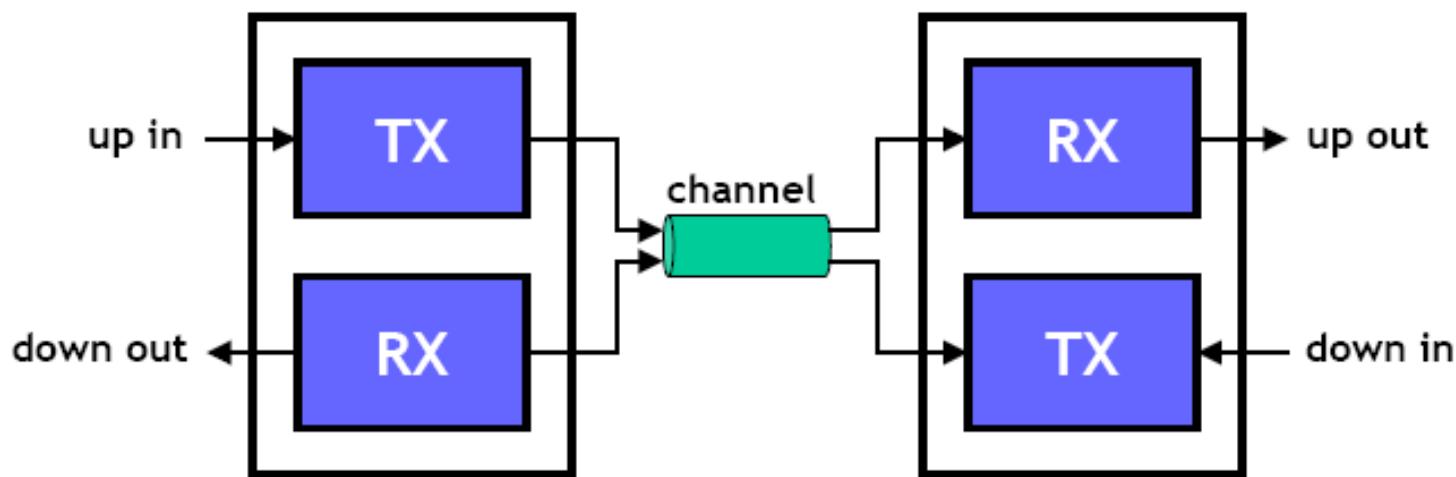
# Testbench: Verifica intrinseca (2/5)

- Esempio: encoder/decoder
  - ▶ Il testbench produce gli ingressi
  - ▶ Encoder e decoder sono connessi in cascata
- Il testbench confronta le uscite del decoder con gli ingressi, opportunamente ritardati



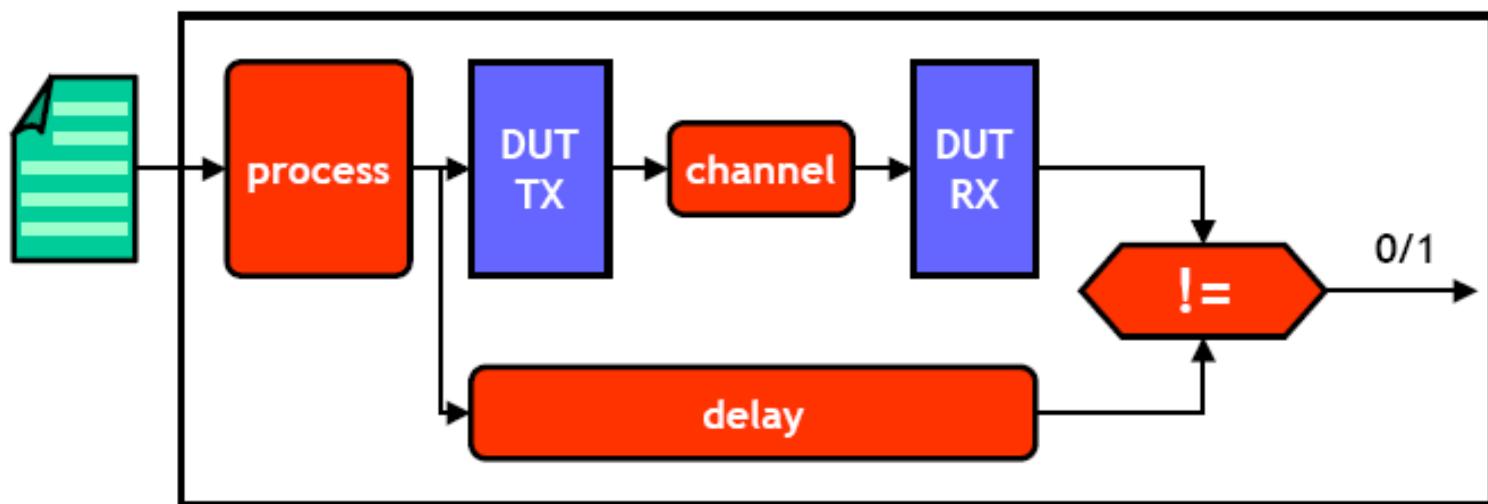
# Testbench: Verifica intrinseca (3/5)

- In alcuni casi è necessario tenere conto dell'ambiente nella verifica di tipo intrinseco
- Si consideri ad esempio un sistema di trasmissione
  - ▶ Volendo verificare la qualità del sistema è necessario modellizzare il canale
    - Tipicamente come sorgente di errori casuali



# Testbench: Verifica intrinseca (4/5)

- Esempio: trasmettitore/canale/ricevitore
  - ▶ Il testbench produce gli ingressi
  - ▶ Trasmettitore e ricevitore sono connessi in cascata mediante il modello del canale
- Il confronto avviene come già discusso



# Testbench: Verifica intrinseca (5/5)

---

- È possibile estendere il modello di testbench appena visto introducendo moduli (process) aggiuntivi
  - ▶ Calcolo del BER (Bit Error Rate)
    - Un process conta i bit ricevuti ed il numero di errori rilevati e calcola il BER ad ogni istante
  - ▶ Calcolo del MTBF (Mean Time Between Faults)
    - Un process conta i cicli di clock e memorizza i tempi a cui si verificano gli errori
- Questo tipo di misure non fa parte a rigore del processo di verifica
  - ▶ Più spesso si ricorre alla prototipazione e alla misurazione sperimentale