# Introduzione al VHDL

Giovanni De Luca

## Sommario



- Introduzione
- Sintassi
- Dichiarazione delle interfacce
- Descrizione delle architetture



## 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 livello di astrazione
  - Progressivo raffinamento della specifica



## 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

# SINTASSI



## Scrittura del codice sorgente



- 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, ...
```



## Tipi



- Il VHDL e' costituito da vari tipi (types) per consentire simulazione e sintesi a vari livelli
- Nel package STANDARD si trovano descritti quegli oggetti destinati alla descrizione COMPORTAMENTALE (non sempre sintetizzabile)
- Nel package IEEE1164 vi si trovano gli oggetti destinati alla sintesi ed alla simulazione logica
- 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



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

- È possibile definire nuovi tipi e sottotipi



## 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



- 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**



## std_logic



- Viene dichiarato racchiuso tra virgolette singole
  - Es: `'U' 'X' '1' '0'`
- In caso di equivoco si usi la dichiarazione esplicita
  - Es: `std_logic'('1')`
- E' il tipo più usato per la sintesi logica
- Per utilizzarlo va inclusa la libreria IEEE e specificato l'utilizzo del package std_logic_1164:

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



## std_logic



- Assume i valori:
  - U: unitialized
  - 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

## Real



- 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

## bit_vector



- 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")`



## string



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



## 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;
```

## Type e subtype



- 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
   campo1: bit; ...
END RECORD;
```

- E dei sottoinsiemi

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





# DICHIARAZIONE DELLE INTERFACCE



## Unità di progetto



- Può essere composto da più unità compilate e salvate in opportune librerie
- Queste unità sono:
  - Dichiarazione di Package (**package**)
  - Corpo del package (**package body**)
  - Dichiarazione di entità (**entity**)
  - Architettura di una entità (**architecture**)
  - Configurazione (**configuration**)

## Unità di progetto



- Il package ed il suo corpo sono usate per contenere funzioni e/o costanti di uso comune
- L'entità e l'architettura descrivono i componenti come interfaccia e come struttura interna
- Le configurazioni permettono di configurare i componenti specificati



## 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";
```



## Entity



- 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
  - Un componente
  - Un intero sistema

## Entity



- Sintassi:

```
entity ENTITY_NAME is
      generic(
            GENERIC_NAME: TYPE := DEFAULT_VALUE
      );
      port( PORT_NAME: DIRECTION TYPE;
      ...
            PORT_NAME: DIRECTION TYPE
      )
End entity [ENTITY_NAME];
```

- Specifica il nome dell'interfaccia
- Descrive la lista di porte
  - Per ogni porta è specificato il nome, il tipo e la direzione (IN, OUT, INOUT)



## Entity



- La lista dei generics permette di parametrizzare la specifica
  - Il valore attuale del parametro è stabilito durante l'istanziazione del componente
  - Si può specificare un valore di default nel caso in cui non venga specificato (o si compili la singola instanza)
- Sintassi:

```
generic(<nome> : <tipo> := <default>);
```



## Esempi di entity



```
entity CONSTANT_VALUE is
  generic(value : integer := 0);
  port (out1 : out std_logic);
end CONSTANT_VALUE ;

entity COMPARE is
  port (a, b : in bit;
        c : out bit
     );
end COMPARE
```

# DESCRIZIONE DELLE ARCHITETTURE



## Architecture



- L'architettura specifica il funzionamento di una entità
- Diverse architetture possono essere associate ad una stessa entità
  - Ogni architettura rappresenta una diversa realizzazione della funzionalità
    - Es: diverse architetture di moltiplicatore
  - La configurazione permette di stabilire quale implementazione usare in un progetto



## Architecture



- Sintassi:

```
architecture ARCH_NAME of ENTITY_NAME is
      istruzioni dichiarative
begin
      istruzioni funzionali
end ARCH_NAME
```

- Le istruzioni dichiarative permettono di dichiarare costanti, segnali e/o componenti che verranno utilizzate nel modello
- Le istruzioni funzionali specificano il funzionamento mediante istruzioni concorrenti

## 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);
```



## Assegnamento a segnale



- Sintassi:

```
segnale <= 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!



## 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 descrizione



- L'architettura può essere descritto 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



## Descrizione dataflow



- 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 (o porte)
  - Assegnamento condizionale di segnali
  - Assegnamento selettivo di segnali

## Descrizione dataflow



- 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



## Operatori



- Logici:
  `and, or, nand, nor, xor`
- Relazionali
  `=, /=, <, <=, >, >=`
- Concatenazione e aritmetici
  `&, +, -, *, /, mod, rem, **, abs`
- Logici unari:
  `not`

- NOTA: il precedente elenco è ordinato in base alla priorità



## Assegnamento a segnale



- 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');
```

## Assegnamento condizionale



- Sintassi

```
segnale <= valore1 when condizione1 else
           valore2 when condizione2 else
           ...
           valoren;
```

- Esempio:

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



## Assegnamento selettivo



- Sintassi

```
with espressione select
        segnale <= valore1 when caso1,
        segnale <= valore2 when caso2,
        ...
        [segnale <= valoren when others];
```

- 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;
```



## Esempio



```
...
begin

with sel select
  q<= i0 after 10 ns when 0;
  q<= i1 after 10 ns when 1;
  q<= i2 after 10 ns when 2;
  q<= i3 after 10 ns when 3;
  q<='U' after 10 ns when others;

sel <= 0 when a='0' and b='0' else
       1 when a='0' and b='1' else
       2 when a='1' and b='0' else
       3 when a='1' and b='1' else
       4;

end mux;
```

I due assegnamenti sono CONCORRENTI

## Descrizione strutturale



- 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



## Descrizione strutturale



- 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];
```



## Descrizione strutturale



- 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

## Descrizione strutturale



- 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



## Descrizione strutturale



- Esempio:

```
architecture STRUCTURAL of COMPARE is
signal I,L,M: std_logic;

component XR2 port (x,y: in BIT; z: out BIT);
end component;
component C0 generic(value : integer := 0);
      port (out BIT);
end component;
component INV port (x: in BIT; y: out BIT);
end component;

begin
      U0: XR2 port map (A,B,I);
      U1: INV port map (y=>L,x=>I);
      Const0 : C0 port map (out=>M);
      U2: XR2 port map (L, M, C);
end STRUCTURAL;
```



## Descrizione comportamentale



- Descrizione di un'architettura con una sintassi algoritmica mediante processi
  - risulta molto simile ad un algoritmo espresso secondo il 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

## Descrizione comportamentale



- 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



## 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;
```

Area sequenziale

Area concorrente

Area sequenziale



## Processi



- 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



- 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
- Il processo viene risvegliato alla variazione dei segnali della lista di sensibilità
- Il processo è eseguito interamente o interrotto nel caso si incontra un'istruzione wait
- I segnali solo aggiornati solo al termine dell'esecuzione con l'ultimo valore assegnato
- L'esecuzione al suo interno è strettamente sequenziale



## 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;
varaible MEMORY : bit_vector (0 to 7);
variable x,y,z : integer;
```

## 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 |



## Wait



- Sospende il processo in attesa di un evento
- `wait;`
  - Sospende il processo a tempo indefinito
- `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



- Le istruzioni di wait possono essere utilizzate al posto della lista di sensibilità
- `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;
```

## Costrutto condizionale if



- Sintassi:

```
if condizione2 then
   istruzioni sequenziali;
{elsif condizone2 then
   altre istruzioni sequenziali;}
[else
   ultime istruzioni sequenziali;]
end if;
```

- Esempio:

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



## Costrutto condizionale case



- Sintassi:

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

- Le scelte non possono sovrapporsi
- È possibile definire intervalli
  - e.g.: 1 to 4, 4 downto 1.



## Costrutto condizionale 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;
```

## Cicli for



```
[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

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



## È giusto?



```
...
signal muxval: integer range 0 to 4;
begin
mux: process (i0, i1, i2, i3, a, b)
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;
...
```

## Esempio corretto



```
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;
```



## Sottoprogrammi



- 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



- 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



- 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;
```



## Configurazioni



- Le configurazioni permettono di configurare i componenti utilizzati
- 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)
- Si può specificare in un'architettura per configurare i componenti utilizzati
- È possibile anche specificare un'unità di progetto chiamata configuration

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



## Package



- Sono dei "contenitori" in cui vengono raccolte le definizioni di tipi, costanti, segnali, procedure, funzioni, configurazioni...
- I package sono composti da una parte dichiarativa e un corpo
- Il corpo è strettamente necessario solo quando sono dichiarate delle funzioni o delle procedure

## Package



- Esempio di package:

```
package my_defs is
  constant unit_delay: time := 1;
  type giorni (lunedì, martedì);
  procedure myp(c:in integer; a:out integer);
end my_defs;
```

- Esempio di package body

```
package body my_defs is
  procedure myp(c:in integer; a:out integer) is
  begin
     a := c;
  end myp;
end my_defs;
```



## Librerie



- Le unità di progetto possono essere raggruppate in librerie
- In un dato istante c'è una sola libreria di lavoro e diverse librerie di risorse
- La libreria di lavoro è referenziata con il nome `work`
- Esempio di inclusione di una libreria e degli elementi contenuti

- Le librerie `work` e `std` e il package `std.standard.all` sono inclusi implicitamente

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
```



## Testbench



- I moduli di test, chiamati testbench, sono specificati direttamente in VHDL
- Sono connessi al componente da testare o lo o istanziano all'interno
- 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
- È utile usare istruzioni di asserzione
- Il codice del testbench può anche essere generato tramite un editor di forme d'onda