II VHDL 3° parte

Tipi

Tipi scalari
- **INTEGER;**
- **REAL;**
- **Tipo Fisici;**
- **BOOLEAN;**
- **CHARCTER;**
- **BIT**
- **ENUMERATO;**

Tipo composto
- **ARRAY**
- **RECORD**

FILE
ACCESS

Definizione di un nuovo tipo

```
TYPE Nome_tipo IS Definizione_tipo;
```

Tipo enumerato

```
TYPE quattrovalori IS ('0', '1', 'X','Z');

TYPE std_ulogic IS
    ('U','X','0','1','Z','W','L','H','-');

TYPE istruzioni IS (add, sub, lw, sw, mov, beq);

TYPE Stato IS
     (Fetch, Decode, Execute, MemAccess, WriteBack);
```

Array

Tipi predefiniti : bit_vector(array di bit) e string (array di character)

Definizione di un array monodimensionale

```
TYPE Nome_array IS ARRAY ( range_indici) OF
  Tipo_base;
```

Es.

```
TYPE data_bus IS array( 0 to 31) OF BIT;

TYPE byte IS ARRAY(7 downto 0) OF std_logic;

TYPE word IS ARRAY(31 downto 0) OF std_logic;
```

Riferimento agli elementi di un array monodimensionale

```
ARCHITECTURE Behavioral OF test IS
   SIGNAL w,x,y: word;
   SIGNAL b,c,f: byte;
   SIGNAL d,e: std_logic;
BEGIN
   d<= w(i);
   b<=y(7 downto 0);
   c<=b;
   x(15 downto 0) <=b&c;       -- concatenazione
   y(31 downto 16) <=b&b;
   y(15 downto 0) <= "1010101010101010";
   e <= '1';
   f<=('1','0','0','1','1','1','0','1');   --aggregazione

END Behavioral;
```

Array multidimensionale

```
TYPE Nome_array IS ARRAY(Range_1,.., Range_N) OF
  Tipo_base;

Es.
TYPE Memoria IS ARRAY (0 to memsize-1, 0 to dim-1) OF
  std_logic;

VARIABLE DataMem : Memoria;

DataMem(i,j) := A;
```

- **Nel caso in cui definiamo un array di array il riferimento agli elementi è differente.**

```
TYPE MemDati IS ARRAY (0 to memsize-1) OF word;

VARIABLE Mem: MemDati;

Mem(i)(j):=A;
```

Array con dimensione non specificata

- **Sono array la cui dimensione non è specificata al momento della sua dichiarazione.**

```
TYPE Nome_tipo IS ARRAY (NATURAL RANGE <>) OF Tipo_base
```

Es.

```
TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF
  std_logic;
```

- **Questo tipo di array vengono utilizzati per definire gli argomenti di sottoprogrammi o entity port.**

RECORD

```
TYPE Nome_record IS
  RECORD    campo1: tipo1;
            campo2: tipo2;
            ....
          campoN: tipoN;
  END RECORD;
```

Record: esempio

```
TYPE opcode_type IS (add, sub, lw, sw, mov, beq);

TYPE istruzione IS
   RECORD opcode:  opcode_type;
          src1:    integer range 0 to 31;
          src2: integer range 0 to 31;
          dst:  integer range 0 to 31;
   END RECORD;

VARIABLE ist: istruzione;
VARIABLE op :  opcode_type;

op:= ist.opcode;

ist := (add, 1,2,3);
```

SUBTYPE

La dichiarazione dei SUBTYPE è utilizzata per definire sottoinsiemi di un tipo.

Permette di evitare la definizione di un nuovo tipo.

```
type MY_WORD is array (15 downto 0) of std_logic;
subtype SUB_WORD is std_logic_vector (15 downto 0);

subtype MS_BYTE is integer range 15 downto 8;
subtype LS_BYTE is integer range 7 downto 0;
```

Subprograms

Consiste di funzioni e procedure

Function

- il nome di una funzione può essere un operatore;
- può avere un numero arbitrario di parametri di ingresso;
- restituisce un solo valore;

Procedure

- può avere numero arbitrario di parametri di ogni possibile direzione (IN,OUT,INOUT)
- istruzione RETURN opzionale (non restituisce alcun valore !)

E' possibile realizzare l'overloading dei subprogram

I parametri possono essere constants, signals, variables o files.

Esiste una versione sequenziale e concorrente delle function e delle procedure.

Functions

```
FUNCTION Nome_Function( par1: tipo1, ..., parN:
  tipoN)
  RETURN TipoValore_restituito
IS
--sezione_dichiarativa_function
BEGIN
-- sezione_esecutiva
 RETURN valore_restituito;
END  Nome_Function
```

Functions: esempi

```
FUNCTION   AndVect   (   op1:   std_logic_vector;   op2   :
   std_logic_vector ) RETURN std_logic_vector
    IS
  VARIABLE temp:
      std_logic_vector(op1'length-1 downto 0);
 BEGIN
   FOR i IN 0 to op1'length-1 LOOP
   temp(i) := op1(i) AND op2(i);
   END LOOP;
   RETURN temp;
 END AndVect;


FUNCTION rising_edge ( SIGNAL clk: std_logic) RETURN BOOLEAN
   IS
 BEGIN
   IF (clk'EVENT) AND (clk='1') THEN RETURN true;
    ELSE RETURN false;
   END IF;
 END rising_edge;
```

Procedure

```
PROCEDURE Nome_procedure
( par1: dir1 tipo1; ....; parN: dirN tipoN)
  IS
--sezione_dichiarativa_procedure
BEGIN
--sezione_esecutiva_procedure
END  Nome_procedure;
```

dove diri ∈{IN, OUT,INOUT}

Procedure: esempio 1

```
PROCEDURE And32
( op1: IN word; op2 : IN word; ris: OUT word)
    IS
  VARIABLE temp: word;
 BEGIN
   FOR i IN 0 to 31 LOOP
  temp(i) := op1(i) AND op2(i);
   END LOOP;
   ris := temp;
 END And32;
```

Procedure: esempio 2

```
TYPE bus_parita IS RECORD
          valori: word;
          pari: std_logic;
       END RECORD;

PROCEDURE calcoloparita (x : INOUT bus_parita)
   IS
  VARIABLE p: INTEGER;
BEGIN
   p:=0;
   FOR i IN 0 to 31 LOOP
      IF (x.valori(i)='1' THEN p:=p+1;
      END IF;
  END LOOP;
  IF (p MOD 2 = 1) x.pari := '1';
    ELSE x.pari := '0';
  END IF;
END calcoloparita;
```

PACKAGE

Contiene elementi che possono essere condivisi tra diverse entità.

Consiste di due parti:

- una sezione dichiarativa, che definisce l'interfaccia del package;
- il corpo del package, che definisce il comportamento del package.

Package declaration

Può contenere le seguenti dichiarazioni:

- dichiarazioni di subprogram, dichiarazioni di tipi e sottotipi
- constant o deferred constant, signal
- dichiarazione di file, dichiarazione di componenti

Package body

Può contenere le seguenti dichiarazioni:

- dichiarazioni di subprogram, definizione dei subprogram
- dichiarazione di di tpi e sottotipi, costanti

  dichiarazione di file

Package: esempio (1)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package dati is
 subtype w16 is std_logic_vector(15 downto 0);
 subtype w32 is std_logic_vector(31 downto 0);
constant z32 : w32
   :="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
 constant zero32 :w32 :=
  "00000000000000000000000000000000";

 type  vect is array ( natural range <>) of std_logic;
 type vect16 is array (natural range <>) of w16;
 type vect32 is array (natural range <>) of w32;
 function vect2int(vettore : std_logic_vector) return
  integer;
 end dati;
```

Package: esempio (2)

```
package body dati is
   function vect2int(vettore : std_logic_vector) return
    integer is
   variable risultato: integer:=0;
   variable dim: integer;

   begin
   dim:= vettore'length;
   for ind in dim-1 downto 0 loop
    risultato:=risultato*2;
    if vettore(ind)='1' then
    risultato:=risultato+1;
    end if;
   end loop;
   return risultato;
 end vect2int;

end dati;
```

File

- **`TYPE nome_tipo_file IS FILE OF Tipo_base`**

- **Esempio:**

```
TYPE integer_file IS FILE OF INTEGER;
```

- **Dichiarazione di un oggetto di tipo file**

```
FILE nomefile : nometipo
```

- **Esempio:**

```
FILE myfile: integer_file
```

File

- Apertura di un file

```
FILE_OPEN ( nome_file, "pathname", modalità
apertura);
```

dove modalità è
{WRITE_MODE, READ_MODE, APPEND_MODE}

- Chiusura di un file

```
FILE_CLOSE( nome_file);
```

Accesso ai file

- READ(file,data)
  Legge da file e restituisce in data il valore letto
- WRITE(file,data)
  Scrive su file il valore presente in data
- ENDFILE(file)
  Restituisce true se si è alla fine del file

File di testo

- Il tipo TEXT è definito nel package std.textio

  FILE nome_file: TEXT;

Un file di testo viene visto come un insieme di LINE

- readline(nome_file,nome_line);
  Legge una line da file
- writeline(nome_file,nome_line);
  Scrive una line su file

Per leggere da tastiera si usa *input* come nome del file, per visualizzare sullo schermo si usa *output*

Read e Write su line di std_logic

- Per leggere o scrivere su valori di tipo std_logic da o verso una line bisogna usare le funzioni
  - READ (nome_line, valore_std_logic)
  - WRITE(nome_line, valore_std_logic)
  - Il loro uso richiede il package ieee.std_logic_textio

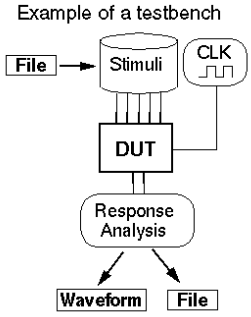
Esempio di lettura da file di testo e copia in std_logic_vector (1/2)

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

use std.textio.all;
use ieee.std_logic_textio.all;

entity prova_file is
  port ( clk: in std_logic;
         a: out std_logic_vector(3 downto 0) );
end prova_file;

architecture beh of prova_file is
begin
  process
    FILE  filedati: TEXT;
  variable tline: LINE;
  variable ta: std_logic_vector(3 downto 0);
  begin
```

Esempio di lettura da file di testo e copia in std_logic_vector (2/2)

```
FILE_OPEN(filedati,"dati.txt",READ_MODE);
  while not endfile(filedati) loop
   readline(filedati,tline);
   read(tline,ta);
   write(tline,ta);
   writeline(output,tline);
   a<= ta;
   wait until clk'event and clk='0';
  end loop;
  file_close(filedati);

end process;
end tbb;
```

Simulazioni: TestBench



Un TestBench è una entità che

- -fornisce gli stimoli (testvectors) per il Device Under Test (DUT) ;
- -non deve essere sintetizzabile;
- -non ha bisogno di porte per l'esterno;
- -contiene al suo interno come component il DUT;

TestBench: esempio (1)

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

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is

--. Sezione dichiarativa

component ADDER4 is
    port (
        a,b: in std_logic_vector(3 downto 0);
        ci: in std_logic;
        s: out std_logic_vector(3 downto 0);
        co: out std_logic);
end component;

signal a,b: std_logic_vector(3 downto 0);
signal ci: std_logic;
signal s: std_logic_vector(3 downto 0);
signal co: std_logic;
```

TestBench: esempio (2)

```
begin
    DUT: ADDER4 port map ( a, b, ci, s, co );

    StimuliA: process
      begin
        a <= "0010";
        wait for 20 ns;
       a <= "0110";
        wait for 20 ns;
        a <= "0110";
   wait for 20 ns;
end process StimuliA;

    b  <= "1001" AFTER 0ns, "0001" AFTER 30 ns,
    "1101" AFTER 50 ns ;
    ci<= '0' AFTER 0ns, '1' AFTER 60 ns;

end stimulus;
```