02_BASICS

LIBRARIES

```
library IEEE;
    use IEEE.STD_LOGIC_1664.ALL;
```

C'è una libreria standard, che contiene le informazioni di base.

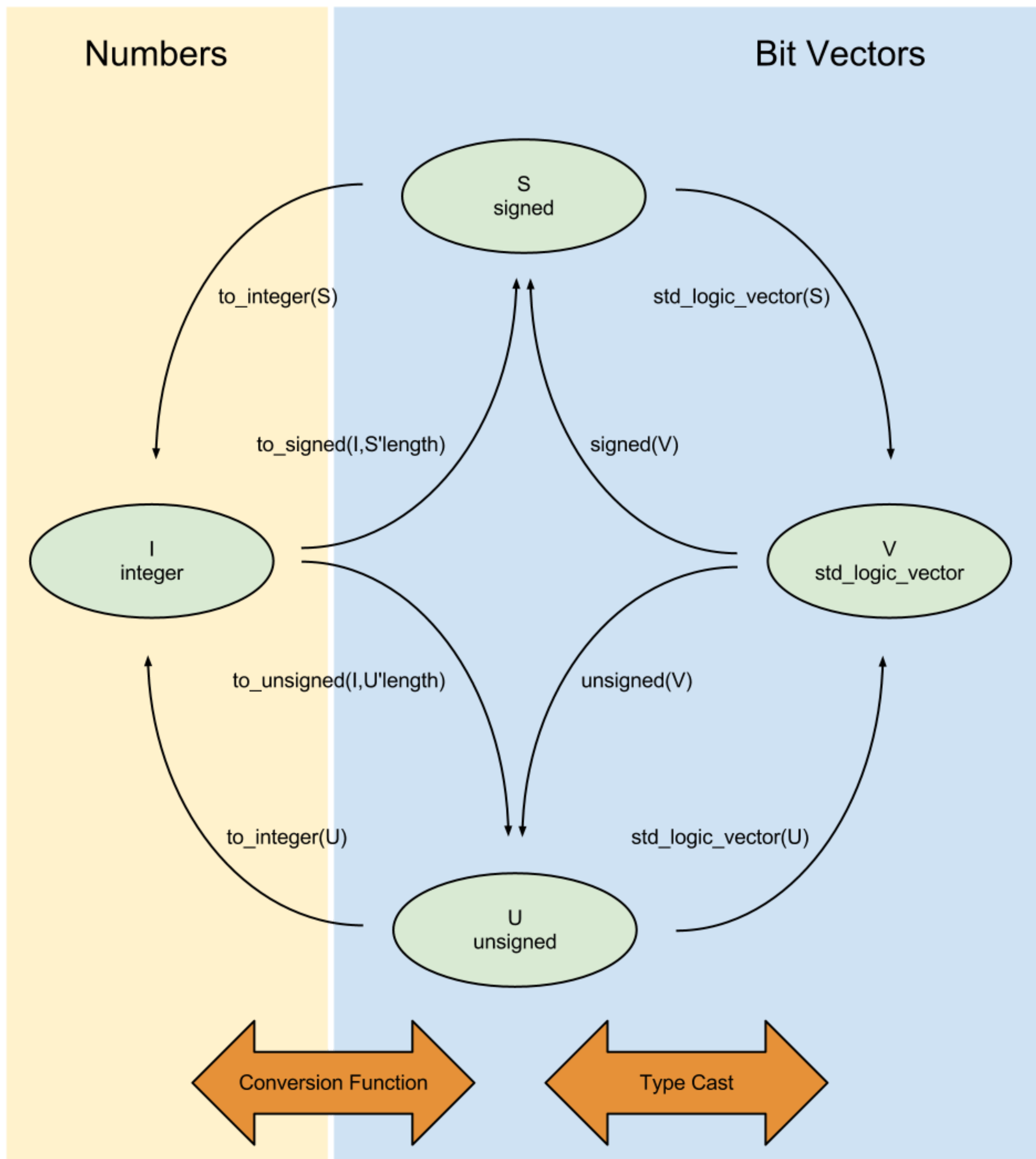
E' definito `integer` as `type INTEGER is range 2147483647 to 2147483647;`

In `STD_LOGIC_1664` sono definiti `std_logic` e `std_ulogic`.

Entrambi definiscono i valori di un segnale digitale su un filo.

`std_ulogic` non definisce però le combinazioni, l'altro sì.

CONVERSIONI



ENTITY DECLARATION

```
entity model is
    Generic (
        VECTOR_WIDTH : integer := 4;
    );
    Port (
        a : in std_logic;
        b : in integer;
        c : out std_logic_vector (VECTOR_WIDTH-1 DOWNTO 0)
        );
end model;
```

ARCHITECTURE DECLARATION WITH SIGNAL

```
architecture Behavioral of model is

    component model is
        Port (
            a : in std_logic;
            b : in integer;
            c : out std_logic_vector (VECTOR_WIDTH-1 DOWNTO 0)
        );
    end component;

    signal temp : std_logic_vector (VECTOR_WIDTH - 1 DOWNTO 0);
    signal temp2 : std_logic_vector (3 DOWNTO 0) := "0011";
    signal temp3 : std_logic_vector (temp'RANGE) := (Others => '0');

begin
    c <= a and b;

end Behavioral;
```

03_BASIC-STATEMENT

& OPERATOR

```
c <= "0010" & "0101"
```

BIT PADDING

Uso l’operatore & per estendere la lunghezza di un vettore.

Posso farlo anche se questo è un vettore di tipo signed/unsigned (ovvero contiene numeri al suo interno).

Per vettore di tipo signed devo però utilizzare un’accortezza in più, per evitare di perdere il segno del numero.

Devo copiare sulla sinistra la cifra più significativa del numero:

```
num : in std_logic_vector(3 DOWNTO 0);
num_extended : out std_logic_vector(4 DOWNTO 0);

num_extended <= num(3) & num;

--altrimenti posso usare gli attributes

num_extended <= num(num'HIGH) & num;
```

AGGREGATI

Sono un altro modo per comporre vettori. A differenza delle concatenazioni (&) non necessitano che la lunghezza dei vettori sia definita a priori

```
signal vect1 : std_logic_vector(7 DOWNTO 0);
signal vect2 : std_logic_vector(7 DOWNTO 0);

vect1 <= (7 => '0', 6 => vect2(3), 3|4|1 => '1', Others => '0');
vect1 <= (vect2(3 DOWNTO 0), vect2(7 DOWNTO 4);
vect1 <= (Others => '0');
vect1 <= (7 DOWNTO 4 => '0', 3 DOWNTO 0 => '1');
```

MULTIPLEXER

WITH/SELECT

Questo è il modo “ufficiale”

```
signal selector : std_logic_vector(1 DOWNTO 0);
signal output : std_logic_vector(3 DOWNTO 0);

with selector select output <= "0001" when "00",
                               "0010" when "01",
                               "1111" when "10",
                               "0000" when "11";

with selector select output <= "0001" when "00",
                               "0010" when "01",
                               "1111" when Others;
```

TUTTE LE POSSIBILI SCELTE DEVONO ESSERE ESPLICITATE

WHEN/ELSE

```
signal selector : std_logic_vector(1 DOWNTO 0);
signal output : std_logic_vector(3 DOWNTO 0);

output <= "0001" when selector = "00" else
          "0010" when selector = "01" else
          "1111" when selector = "10" else
          "0000" when selector = "11";

output <= "0001" when selector = "00" else
          "0010" when selector = "01" else "1111";
```

TUTTE LE POSSIBILI SCELTE DEVONO ESSERE ESPLICITATE

04_GENERATE-STATEMENT

I generate statement vengono elaborati in fase di pre-sintesi. Non possono quindi essere controllati in fasi successive.

If generate:

```
architecture Behavioral of test is
begin

    LABEL : if a = '0' generate
       c <= '1';
    end generate;

    LABEL : if a = '1' generate
       c <= '0';
    end generate;

end Behavioral;
```

Nested for and if generate:

```
architecture Behavioral of test is
begin
    REG_GEN : for I in data_out'RANGE generate

        FIRST_FF_GEN : if I = data_out'LEFT generate

            ff_d_inst :  ff_d
                Port Map(
                    reset    => reset,
                    clk      => clk,

                    d   => data_in,
                    q   => data_reg(I)
                );

        end generate;

        OTHERS_FF_GEN : if I /= data_out'LEFT generate

            other_ff : ff_d
                Port Map(
                    reset => reset,
                    clk => clk,

                    d => data_reg (I+1),
                    q => data_reg (I)
                );

        end generate;
```

```
    end generate;

end Behavioral;
```

Non lasciare eventualità non coperte

Posso annidare tutti i costrutti generate

NON esiste un costrutte `else` per gli `if generate`

05_CUSTOM-TYPES

Esempi di creazione tipo e sottotipo:

```
type TEST is ('0', '1'); --tipo numerato
type word is array (0 TO 31) of bit
subtype VALORE is integer range 0 TO 255

subtype sottotipo is std_logic_vector (10 DOWNTO 0);

signal collegamento : sottotipo;
signal collegamento_originale : std_logic_vector(10 DOWNTO 0);

--Posso anche assegnare un sottotipo all'originale
collegamento_originale <= collegamento;
```

Un tipo array è definibile così:

```
type ArraySemplice is array (3 DOWNTO 0) of std_logic;
```

Posso creare un **nested 1D array** in questo modo:

```
type NestedArray is array (10 DOWNTO 0) of std_logic_vector(7 DOWNTO 0);
signal esempio : NestedArray;

--se voglio inizzializzarlo
signal esempio: NestedArray := (Others => (Others => '0'));

esempio (0)(1) <= '1';
esempio (1) <= "01010101";
```

Questo tipo di array può essere utile per la creazione di piccole memorie.

Posso usare un tipo custom solo dopo averlo definito.

Per usarli in `architecture` basta averli definiti prima del `begin`

Per usarli nelle porte di una entity dovrei definirli in un package esterno. E' quindi meglio evitare.

06_PROCESSES

E' una metodologia ad alto livello, io descrivo il comportamento del circuito e Vivado cerca di crearlo.

Dentro i costrutti `process` vige una logica sequenziale.
Il resto del programma continua ad essere concurrent.

In simulazione il codice process è letto in modo sequenziale. Diventa un po' simile a C.

La simulazione viene implementata con un circuito logico/sequenziale. Non viene assolutamente creato un processore capace di leggere in sequenza il codice.
In sintesi viene creato un hardware con le stesse proprietà. In simulazione il codice viene letto ed eseguito riga per riga.

Quando arriva alla fine del process la simulazione ricomincia da capo. Se il loop non viene fermato abbiamo un "infinite loop process". Questi sono generalmente errori.

WAIT STATEMENT

```
wait;

wait for 10 ns;
wait until clk='1'
wait until A>B and S1 or S2; --qui possono mettere tutti gli statement logici che voglio
wait on sig1, sig2; --quando quei segnali cambiano il wait viene sbloccato (il programma va avanti)
```

Esempio1:

```
architecture Behavioral of wait1 is
    signal a : integer := 0;

begin

    process
```

```
    begin
        wait for 10 ns;
        a <= a + 1;
        wait;
    end process;

end Behavioral;

--qui in t=10ns assegno a=a+1 e rimane così per sempre
```

Esempio2:

```
architecture Behavioral of wait1 is
    signal a : integer := 0;

begin

    process
    begin
        wait for 10 ns;
        a <= a + 1;
        --wait;
    end process;

end Behavioral;

--qui a continua ad aumentare ogni 10ns, all'infinito.
```

SENSITIVITY LIST

E' IDENTICO ad inserire `wait on ...` alla fine del codice.

```
process (clk, reset)
begin
    if reset = '1' then
    ...
    elseif rising_edge(clk) then
    ...
    end if;
end process;

--il process parte solo quando clk o reset cambiano
```

NON bisogna fare affidamento ai wait nel file di sintesi.
  NON USARLI MAI nel file di sintesi. Bad-Practice.

Quali segnali aggiungere alla sensitivity list?
  Tutti i segnali con delle condizioni presenti nella parte destra delle assegnazioni.

In *VHDL 2008* si può usare il comando “all” per includere tutti i segnali utilizzati nel process.

Se il process è sincrono mi basta mettere il clock.

> **Uso i wait statement in simulazione e le sensitivity list in sintesi.**

> **In ogni process deve esserci una sensitivity list O un wait. Altrimenti ho un infinite loop**

Nel mondo reale non sono implementabili tutti i segnali di wait. Molti anzi vengono ignorati e basta (nella sintesi, in simulazione vanno).

DECLARATION

Il process vede tutte le dichiarazioni precedenti, quelle visibili nelle entity e nella architecture.

Posso dichiarare nuove risorse (variabili, ecc...) nell'area che si trova tra `process` e `begin`.

Non posso dichiarare un signal dentro i process.

A questo link posso trovare tutti gli statement che posso usare dentro un process.

Alcuni dei più importanti sono:

IF STATEMENT

```
if a=b then
    c:=a;
elsif b<c then
    d:=b;
    b:=c;
else
    do_it;
end if;
```

CASE STATEMENT

```
case my_val is
    when 1 =>
        a:=b;
    when 3 =>
        c:=d;
        do_it;
    when others =>
        null;
end case;
```

LOOP STATEMENT

Il loop continua a fare il loop finché non trova exit (break in C) tra i suoi statement.

Posso farne anche uno controllato.

```
loop --questo costrutto qui non lo usiamo quasi mai
    ...
    exit when end file;
end loop;

for I in 1 to 10 loop
    AA(I) := 0;
end loop;
```

FOR AND WHILE STATEMENT

Il loop continua a fare il loop finché non trova exit (break in C) tra i suoi statement.

Posso farne anche uno controllato.

```
for I in 1 to 10 loop
    AA(I) := 0;
  end loop;

while not end_file loop
  input_something;
end loop;
```

SIGNAL COMMIT - UNOFFICIAL NAME

Nei process i segnali si comportano in modo particolare.
Il valore dei segnali viene committato solo quando Vivado incontra un wait.

E' un comportamento voluto, ricalca li comportamento di un flip flop.

Questo codice produce il seguente output:

```
architecture Behavioral of SignalCommit is
    signal a : integer := 0;
    begin
        process
            begin
                a <= 2;
                wait for 10 ns;
                a <= 5;
                a <= a*5;
                wait for 10 ns;
                a <= a*5;
                wait;
        end process;
end Behavioral;
```

Questo tipo di cosa al di fuori da un process sarebbe un errore. Avrei un multiple assignment.

VARIABILI

Il commit di una variabile è istantaneo.

L’assegnamento di una variabile avviene con questa sintassi: `my_var := 5`

Codice e relativo output:

```
signal a : integer := 0;

process
    variable my_var : integer;
begin
    my_var := 2;
    a <= my_var;
    wait for 10 ns;
    my_var := 5;
    my_var := my_var*5;
    a <= my_var;
    wait for 10 ns;
    my_var := my_var*5;
    a <= my_var;
    wait;
end process;
```

Non posso fare il debug di variabili.

STANDARD STRUCTURES

Per implementare certi costrutti vengono utilizzate delle strutture standard.

Ad esempio `rising_edge` o `falling_edge` vengono implementati come flip_flop.

Se il circuito è molto grande certi buffer (tipo il reset) potrebbero avere difficoltà a cambiare il livello di una linea.
Questo potrebbe portare ad un fronte poco ripido.

Potrebbe succedere che il segnale arrivi prima ad una parte del circuito e poi ad un’altra.
Questo farebbe un macello.

Potrei aumentare il tempo di clock, ma se il circuito è troppo grande dovrei aumentarlo troppo e questo sarebbe impraticabile.

RICORDA:

E' una good practice quella di resettare tutti i segnali interni, e le uscite, quando ho un `reset = '1'`

CREDITS AND FINAL NOTE

Ho creato questi appunti durante il corso di SED, spero possano esserti utili.

Purtroppo questo notebook non è esente da errori, se ne trovi (o se hai modifiche da suggerire) ti prego di segnalarmeli via mail: **mario.calio@mail.polimi.it**