VHDL cheatsheet

Lorenzo Rossi

Anno Accademico 2019/2020

Email: lorenzo14.rossi@mail.polimi.it

GitHub: https://github.com/lorossi/appunti-vhdl

Indice

1 Sintassi base

1.1 Case sensitivity

Il VHDL è *case insensitive*, ovverosia non fa differenza tra lettere maiuscole e minuscole **in nessun caso.**

1.2 Assegnazioni

1.2.1 Assegnazioni di Signal

```
SIGNAL_NAME <= EXPRESSION;
```

1.2.2 Assegnazioni di Variable

```
VARIABLE NAME := EXPRESSION;
```

1.2.3 Slicing di vettori

SIGNAL_0 e *SIGNAL_1* sono segnali di tipo *std_logic_vector*

```
SIGNAL_1 <= SIGNAL_0(RANGE);
SIGNAL_1 <= SIGNAL_0(N downto M);
SIGNAL_1 <= SIGNAL_0(M to N);
```

1.3 Operatori

1.3.1 Uso degli operatori

```
SIGNAL_1 <= SIGNAL_2 operatore SIGNAL_3;
```

1.3.2 Operatori logici

- not *operazione di negazione*
- and *operazione di moltiplicazione logica*
- or *operazione di addizione logica*
- nor *operazione di somma negata*
- nand *operazione di moltiplicazione negata*
- xor *or esclusivo*
- xnor *or esclusivo negato*

1.3.3 Shift

- SLL *Shift left, i bit più a destra sono rimpiazzati da zeri*
- SRL *Shift right, i bit più a sinistra sono rimpiazzati da zeri*
- SLA *Shift left aritmetico*
- SRA *Shift right aritmetico*
- ROL *Rotate left*
- ROR *Rotate right*

1.3.4 Operatori relazionali

- $=$ *uguale*
- $/=$ *non uguale, diverso*
- $<$ *minore*
- $>$ *maggiore*
- $<=$ *minore uguale*
- $>=$ *maggiore uguale*

1.3.5 Operatori aritmetici

- $**$ *elevazione a potenza*
- rem *resto*
- mod *modulo*
- $/$ *divisione*
- $*$ *moltiplicazione*
- $+$ *somma*
- abs *valore assoluto*

Per incrementare di 1 una variabile non si può usare l'operatore ++ ma bisogna usare l'espressione

```
SIGNAL <= SIGNAL + 1;
```

E analogamente bisognerà comportarsi con sottrazione e altri operatori aritmetici.

1.3.6 Operatori miscellanei

- & *operazione di concatenazione*
- −− *commento in linea*
- Others *si riferisce a tutti gli elementi di un signal che non sono già stati menzionati*
- array(i) *accesso all'i-esimo elemento dell'array*
- array(i, j) *accesso al j-esimo elemento dell'i-esimo elemento dell'array - nested 2D array*
- array := (Others => (Others => '0')) *inizializzazione di un nested 1D array*

2 Modeling Styles

- Structural modeling
  - Implementazione come unione di porte/strutture
  - Technology dependent
- Dataflow Modeling
  - Implementazione come descrizione combinatoria tramite porte logiche di base
  - Technology independent
- Behavioral modeling
  - Implementazione come descrizione del comportamento che ha l'entity
  - Non riflette direttamente l'implementazione
- Mixed Modeling
  - Combinazione delle precedenti

3 Librerie

- Definiscono i tipi base o il comportamento delle funzioni elementari
- Tutte le librerie devono essere dichiarate manualmente, tranne la libreria *standard* che contiene informazione base e direttive date all'analizzatore (es. *boolean*)
- Librerie fondamentali:
  - *std_logic_1164.all* enhanced signal types
  - *numeric_std.all* numerical computation
  - *math_real.all* mathematical operations

4 Tipi di dato

Il linguaggio VHDL contiene diversi tipi di dato che possono caratterizzare *signal* e *variables*

4.1 Integer

- Definito nella libreria *std_logic_1164.all*
- Contiene gli interi da $-2^{31}-1$ a $2^{31}-1$
- Non ci si può affidare al *roll-up* o all'*overflow*
- Si può specificare la larghezza massima dell'intero. **Tuttavia in tal caso non ci si potrà affidare al *roll-over.***

```
my_num INTEGER range 0 to 64
```

4.2 Natural e Positive

- Sono *subtype* di *INTEGER*
- Non sono di 32 bit
  - Il *natural* contiene gli interi da 0 a $2^{31}-1$

    ```
    subtype NATURAL is INTEGER range 0 to INTEGER'HIGH
    ```

  - Il *positive* contiene gli interi da 1 a $2^{31}-1$

    ```
    subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH
    ```

4.3 Std_Ulogic

- Definito nella libreria *std_logic_1164*
- Definisce qualsiasi stato di un elemento ad 1 bit
  - 'U' *forcing uninitialized*
  - 'X' *forcing unknown*
  - '0' *forcing 0*
  - '1' *forcing 1*
  - 'Z' *high impedance*
  - 'W' *WEAK unknown*
  - 'L' *WEAK low*
  - 'H' *WEAL high*
  - '-' *don't care*

4.4 Std_Logic

- Definito nella libreria *standard*
- È un *subtype* del tipo *std_ulogic*
- Ovvia automaticamente al problema che sorge quando due signal diversi vengono connessi tra di loro (ad esempio, all'uscita di due buffer)
- Il segnale viene automaticamente *risolto* (assegnato) tramite una *tabella di risoluzione*

Tabella di risoluzione

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

4.5 Std_Logic_Vector

- Estensione di *std_logic* sotto forma di vettore

```
STD_LOGIC_VECTOR(15 downto 0)
STD_LOGIC_VECTOR(0 to 15)
```

4.6 Signed/Unsigned

- Definiti nella libreria *numeric_std*
- Array di *signed* or *unsigned* con tutte le loro operazioni definite
- Non c'è limite alla dimensione che possono assumere
- Si può fare affidamento al *roll-up* o all'*overflow*
- C'è controllo sulla rappresentazione numerica

```
type UNSIGNED is ARRAY (NATURAL range <>) OF std_logic
type SIGNED is ARRAY (NATURAL range <>) OF std_logic
```

4.7 Conversioni tra tipi di dato

- Categorizzazione dei tipi
  - *Integer* → numero
  - *Signed, Unsigned, Std_logic_vector* → vettori di bit
- Conversione dei tipi
  - Da *Integer* a:
    - *Signed* → *to_signed(integer, signed'LENGHT)*
    - *Unsigned* → *to_unsigned(integer, unsigned'LENGHT)*
  - Da *Signed* a:
    - *Integer* → *to_integer(signed)*
    - *Std_logic_vector* → *std_logic_vector(signed)*
  - Da *Unsigned* a:
    - *Integer*→ *to_integer(unsigned)*
    - *std_logic_vector*→ *std_logic_vector(unsigned)*
  - Da *Std_logic_vector* a:
    - *Signed*→ *signed(std_logic_vector)*
    - *Unsigned*→ *unsigned(std_logic_vector)*

5 Basic statements

5.1 Entity

- Una *entity* è la descrizione dell'interfaccia tra il design e l'ambiente esterno
- Può indicare le *declarations* e gli *statements* che sono parte del design
- Una *entity* può essere condivisa tra più elementi del *design*, ognuno con la sua differente architettura
- Struttura di una *entity*

```
entity ENTITY_NAME is
    port (
        PORT_NAME : PORT_MODE PORT_TYPE;
        PORT_NAME : PORT_MODE PORT_TYPE
        ...
    );
end ENTITY_NAME;
```

- **Non ci vuole il *punto e virgola* nella dichiarazione dell'ultima porta**
- Esempio di una *entity*

```
entity or2 is
    port (
        a : in std_logic;
        b : in std_logic;
        c : out std_logic
    );
end or;
```

- **Si dichiarano prima della *architecture***

5.2 Architecture

- Dentro la *architecture* viene descritto il comportamento di uno o più componenti istanziati
- Si possono usare più *architecture* per *entity*, scritte in forme diverse, per poi scegliere l'implementazione che si desidera
- Struttura di una *architecture*

```
architecture ARCHITECTURE_NAME of ENTITY_NAME is
Begin
    ...
End ARCHITECTURE_NAME
```

- Esempio di una *architecture*

```
architecture or2_a of or2 is
Begin
    c <= a or b;
End or2_a;
```

5.3 Modes

- Il modo di una porta definisce la direzione dei segnali che la attraversano
- Tipi di modi:
  - IN *ingresso, può solo essere letta*
  - OUT *uscita, può solo essere scritta*
  - INOUT *sia ingresso che uscita*
  - BUFFER *porta di uscita che può essere usata anche per leggere il valore scritto in precedenza*
  - LINKAGE *porta speciale, crea collegamento diretto senza buffer*

5.4 Signal

- Oggetti primari per la creazione di un sistema hardware
- Ad ogni segnale è associato un nome, un tipo ed un valore iniziale *(facoltativamente)*
  - Un segnale non inizializzato avrà valore indefinito *U*
- Possono essere utilizzati per memorizzare valori o per connettere entità
- Struttura di un *signal*

```
signal SIGNAL_NAME : SIGNAL_TYPE := INITIAL_VALUE;
signal SIGNAL_NAME : SIGNAL_TYPE;
```

- Esempio di un *signal*

```
signal s1 : std_logic := '1';
signal s2 : std_logic;
```

- **Il *signal* si dichiara all'interno dell'*architecture*, prima del *begin***

5.5 Component

- Per utilizzare i moduli all'interno del design, questi devono essere prima *dichiarati* e poi *istanziati*
  - La *dichiarazione* deve avvenire all'interno dell'*architecture*
  - L'*istanziamento* deve avvenire dopo il *begin*
- Struttura di una *dichiarazione* :

```
component ENTITY NAME is
port (
    PORT_NAME : PORT_MODE PORT_TYPE;
    PORT_NAME : PORT_MODE PORT_TYPE
    ...
    );
end component;
```

- **Non ci vuole il *punto e virgola* nella dichiarazione dell'ultima porta**
- La struttura delle porte del *componente dichiarato* deve essere identica a quella *dell'entity originale.*
- Struttura di un *istanziamento*:

```
INSTANTIATION_NAME : INSTANTIATED_NAME
port map (
    PORT_NAME => SIGNAL,
    ...
);
```

- **Non ci vuole la *virgola* nel map dell'ultima porta**
- Esempio di una *dichiarazione*:

```
component or2 is
port (
    a : in std_logic;
    b : in std_logic;
    c : out std_logic
);
end component;
```

- Esempio di un *istanziamento*:

```
or2_inst1 : or2
port map (
    a => a1,
    b => a0,
    c => n1
);
```

5.6 Generic

- I *generic* servono a passare informatzioni all'*entity*
- Non possono essere modificati duranete l'esecuzione
- Specificano parametri, ad esempio larghezza di vettori
- Si usano per rendere scalabile un sistema
- Struttura di un *generic*

```
Generic(
    GENRIC_NAME : GENERIC_TYPE := INITIAL_VALUE;
    ...
);
```

- Esempio di un *generic*

```
entity my_entity is
Generic (
    VECTOR_WIDTH : integer := 16;
    XOR_CHECK : boolean := false
);
Port (
    a : in std_logic_vector(VECTOR_WIDTH-1 downto 0);
    b : out std_logic_vector(VECTOR_WIDTH-1 downto 0)
);
end my_entity;
```

- **Vanno usati nella *entity* prima della funzione *port***

5.7 Attributes

- Permettono di ricavare informazioni su oggetti del nostro sistema
- *Attributes* più usati:
  - LEFT/RIGHT
  - HIGH/LOW
  - LENGTH
  - RANGE/REVERSE_RANGE
- Struttura di un *attribute*

```
object'attribute_name
```

- Esempi di *attribute*

```
signal vect : std_logic_vector(4 to 16);
vect'LEFT => 4
vect'RIGHT => 4
vect'HIGH => 16
vect'LOW => 4
vect'LENGTH => 13
vect'RANGE => (4 to 16)
vect'REVERSE_RANGE => (16 downto 4)
```

5.8 Aggregate

- Definisce un vettore tramite composizione
- A differenza della concatenazione, la lunghezza del vettore non è definita a priori
- Esempi di *aggregate*

```
signal vect_1 : std_logic_vector(7 downto 0);
signal vect_2 : std_logic_vector(7 downto 0);

vect1 <= (7 => '0', 6 => vect2(3), 3|4|2 => '1', Others => '0');
vect1 <= (vect2(3 downto 0), vect2(7 downto 4));
vect1 <= (Others => '0');
vect1 <= (7 downto 4 => '0', 3 downto 0 => '1');
```

5.9 When/Else

- Assegna il valore ad un *signal* condizionatamente ad un altro *signal*
- Tutte le possibili condizioni vanno esplicitate, usando la *keyword else*
- Non bisogna usare le virgole dopo gli else
- Struttura di un *when/else*

```
SIGNAL_1 <= VALUE when SIGNAL_2 0 VALUE else
            ...
            else = VALUE;
```

- Esempio di *when/else*

```
a, b signals
b <= "1000" when a = "00" else
     "0100" when a = "01" else
     "0010" when a = "10" else
     "0001" when a = "11";

b <= "1000" when a = "00" else
     "0100" when a = "01" else
     "0000";
```

5.10 With/Select

- Assegna il valore ad un *signal* condizionatamente ad un altro *signal*
- Tutte le possibili condizioni vanno esplicitate, usando la *keyword else*
- Servono le virgole a fine riga

- Struttura di un *with/select*

```
with SIGNAL_1 select SIGNAL_2 <= VALUE when VALUE;
                                          ...
```

- Esempio di *with/select*

```
a, b signals
with a select b <= "1000" when "00";
                   "0100" when "01";
                   "0010" when "10";
                   "0001" when "11";

with a select b <= "1000" when "00";
                   "0100" when "01";
                   "0000" when Others;
```

6 Generate statement

- Servono ad istanziare un numero di componenti non definito staticamente
- Il numero di *component* istanziati può essere in funzione dei *generic*
- I *generate statements* possono essere annidati
- **Vanno usati dentro l'*architecture*, dopo il *begin***

6.1 If ... Generate

- Tramite il costrutto *If ... Generate* è possibile includere determinate dichiarazioni in funzione del valore assunto da *generic* o *costanti*
- Il costrutto *If ... Generate* non ha il costrutto *else*
- Struttura di un *If ... Generate*

```
LABEL : if GENERIC == VALUE generate
    STATEMENT
end generate;
```

- Esempio di un *If ... Generate*

```
architecture Behavioral of test is
    begin
        NULL_GEN : if NULL_VALUE = true generate
            c <= (Others => '0');
        end generate;

        NO_NULL_GEN : if NULL_VALUE = false generate
            c <= a;
        end generate;
    end Behavioral;
```

6.2 For ... Generate

- Tramite il costrutto *For ... Generate* è possibile includere determinate dichiarazioni ricorsivamente, come in un ciclo *for*
- Struttura di un *For ... Generate*

```
LABEL : for VARIABLE in RANGE generate
    STATEMENT
end generate;
```

- Esempio di un *For ... Generate*

```
architecture Behavioral of test is
begin
    LOOP_GEN_1 : for I in 0 to 3 generate
        c(I) <= a(0);
    end generate;

    LOOP_GEN_2 : for I in 4 to 7 generate
        c(I) <= a(1);
    end generate;
end Behavioral;
```

7 Process

- Il *process* in VHDL è una struttura particolare usata per descrivere il comportamento di un aparte di circuito
- **Il comportamento è diverso da un linguaggio di programmazione classico**
- Durante la simulazione, un *process* viene letto ed eseguito sequenzialmente dal simulatore
- **I *process* vanno messi all'iterno del *begin***
- Differenza tra *sintesi* e *simulazione*
  - Sintesi: *il process viene sintetizzato per ottenere un Hardware con le stesse proprietà*
  - Simulazione: *il process viene letto da una macchina temporale che valuta riga per riga*
- Struttura di un *process*

```
PROCESS_NAME process (SENSITIVITY_LIST)
    TYPE_DECLARATIONS
    CONSTANT_DECLARATIONS
    VARIABLE_DECLARATIONS
    SUBPROGRAM_DECLARATIONS
begin
    SEQUENTIAL_STATEMENTS
end process PROCESS_NAME;
```

7.1 Struttura di un process

7.1.1 Infinite loop

- In un *process* il codice viene letto sequenzialmente.
- Quando il *process* raggiunge l'ultima riga, riparte dall'inizio
- **Senza nessun tipo di interruzione si crea un *infinite loop process***
  - **È normalmente un comportamento non voluto**

7.1.2 Wait statement

- Il *wait statement* temporizza il circuito, fermandone l'esecuzione in determinate condizioni
- Struttura di un *wait statement*

```
LABEL wait SENSITIVITY_CLAUSE CONDITION_CLAUSE;
```

- Esempio di un *wait statement*

```
wait for 10 ns;              -- Specifica un tempo di timeout
wait until clk='1';          -- Condizione booleana
wait until A>B and S1 or S2; -- Condizione booleana
wait on sig1, sig2;          -- Sensitivity clause, qualsiasi cambio
                             -- in uno dei segnali ferma la pausa
wait;                        -- Pausa per semprea
```

7.1.3 Sensitivity list

- La *sensitivity list*, insieme agli *wait statements*, serve a controllare il flusso del programma
- Funge da *lista di attivazione* del processo, poiché questo rimane dormiente finché un segnale non lo risveglia cambiando di valore
- Nella *sensitivity list* vanno aggiunti tutti i segnali presenti nella parte destra delle assegnazioni dei vari costrutti condizionali

7.1.4 Declaration region

- All'interno di un *process* sono visibili tutte le dichiarazioni presenti nella *entity* e *architecture*, come *signals* e *begin*
- La *declaration region* è nell'area compresa tra *process* e *begin*
- Dentro la *declaration region* è possibile dichiarare nuove risorse (le *variabili*). Non esistono *signals* locali
- Esempio di *declaration region*

```
process(clk, reset)
    variable local_count : count'base
begin
     ...
end process;
```

7.1.5 Sequential Statement Region

- Nella regione di *sequential statement* viene inserita tutta la descrizione del comportamento del *process*
- La lettura del codice avviene in maniera *sequenziale*

- Esempio di *sequential statement region*

```
process (...)
    ...
begin
    local_count := count;
    if reset = '1' then
        local_count <= 0;
    elsif rising_edge(clk) then
        local_count <= local_count + 1;
    end if;
    count <= local_count;
end process;
```

7.2 Sequential statements

- Questi *statements* sono usati nei *process*
- Lista di *statements*
  - wait statement
  - assertion statement
  - report statement
  - signal assignment statement
  - procedure call statement
  - if statement
  - case statement
  - loop statement
  - next statement
  - exit statement
  - return statement
  - null statement

7.2.1 If statement

- Struttura dell'*if statement*

```
LABEL if CONDITION1 then
    ...
elsif CONDITION2 then
    ...
else
    ...
end if LABEL;
```

- Esempio di *if statement*

```
if a=b then
    c := a;
elsif b<c then
    d := b;
    b := c;
else
    c := 0;
end if;
```

7.2.2 Case statement

- Struttura dell'*Case statement*

```
LABEL case EXPRESSION is
    when choice1 =>
        ...
    when choice2 =>
        ...
    when others =>
        ...
end case LABEL
```

- Esempio di *case statement*

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

7.2.3 Loop statement

- Struttura dell'*Loop statement*

```
LABEL loop
    ... --use exit statement to get out
end loop LABEL;

LABEL for VARIABLE in RANGE loop
    ...
end loop LABEL;

LABEL while CONDITION loop
    ...
end loop LABEL;
```

- Esempio di *loop statement*

```
loop
    input_something;
    exit when end_file;
end loop;

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

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

7.3 Commit

- Il *commit* indica quando il valore delli *signals* e delle *variables* vengono aggiornati nel sistema

7.3.1 Signal commit

Nei *process* i segnali hanno un comportamento particolare

- Il valore dei segnali rimane invariato tra due *wait statement*
- Il valore scritto nei segnali viene aggiornato solo quanto il flusso raggiunge uno *wait statement*

7.3.2 Variable commit

- Le *variables* hanno visibilità ristretta nel *process* che le ha dichiarate
- Le *variables* vengono aggiornate istantaneamente
- L'assegnazione delle *variables* avviene con =:

8 Custom types

- Nel VHDL sono disponibili molti tipi di dato base. Per creare nuovi tipi di dato personalizzati si usano le seguenti keywords
  - Type *tipi*
  - Subtype *sottotipi*
- **Si usano nell'architecture prima del *begin***

8.1 Sottotipi

- Un *subtype* è un sottoinsieme di un tipo già esistente *(per esempio, integer o std_logic_vector)*
- I nuovi tipi creati possono essere assegnati anche al tipo originale
- Struttura di un *subtype*

```
subtype SUBTYPE_NAME TYPE STATEMENTS;
```

- Esempi di *subtypes*

```
subtype short integer range 0 to 255;

subtype nib is std_logic_vector(3 downto 0);
subtype byte is std_logic_vector(7 downto 0);

signal myByte : byte;
signal myVect : std_logic_vector(7 downto 0);

myVect <= mybyte;
```

8.2 Tipi

8.2.1 Tipi enumerati

- *Tipi* di dato che rappresentano un set finito di stati diversi
- Struttura di un *tipo enumerato*

```
type TYPE_NAME is (ELEMENT, ELEMENT, ...);
```

- Esempi di *tipi enumerati*

```
type MyBit_type is (L, H);
type MyState_type is (init, waiting, working, done);
```

8.2.2 Tipi record

- *Tipi* di dato composto da diversi sotto oggetti
- Struttura di un *tipo record*

```
type TYPE_NAME is record
ELEMENT_NAME : element type;
...
end record TYPE_NAME;
```

- Esempi di *tipi record*

```
type Operation is record
OpCode : Bit_Vector(3 downto 0);
Op1, Op2, Res : RegName;
end record;
```

8.2.3 Tipi array

- *Tipi* di dato composto da multipli elementi dello stesso tipo
- Struttura di un *tipo array*

```
type TYPE_NAME is array (RANGE) of ELEMENT_TYPE;
```

- Esempi di *tipi record*

```
type nibble is array (3 downto 0) of std_ulogic;
type RAM is array (0 to 31) of integer range 0 to 255;
```

Array 1D

- *Array* in cui è definito un singolo range
- È possibile creare *nested 1D array* in cui ogni eleento di ogni cella è a sua volta un array
- Struttura di un *array 1D*

```
type TYPE_NAME is array (RANGE) of ANOTHER_TYPE;
```

- Esempi di *array 1D*

```
type MyNestedArray is array (0 to 10) of std_logic_vector(7 downto 0);
signal ciao : MyNestedArray := (Others => (Others => '0'));
ciao(0)(0) <= '1';
ciao(0) <= "01010101";
```

Array multidimensionali

- *Array* con più indici
- Utili quando la grandezza da rappresentare è a sua volta a più dimensioni *(es. immagini RGB)*
- Struttura di un *array multidimensionale*

```
type ARRAY_TYPE is array (RANGE, RANGE, ...) of ANOTHER TYPE
```

- Esempio di *array multidimensionale*

```
type RGB_Type is record
    r_ch : unsigned(7 downto 0);
    g_ch : unsigned(7 downto 0);
    b_ch : unsigned(7 downto 0);
end record;

type img_type is array (0 to 15, 0 to 15) of RGB_Type;

signal immagine : img_type := (Others => (Others => (
                                  r_ch => to_unsigned(100, 8),
                                  g_ch => to_unsigned(100, 8),
                                  b_ch => to_unsigned(100, 8)
                              )));

immagine(0, 0) <= (
                     r_ch => to_unsigned(100, 8),
                     g_ch => to_unsigned(100, 8),
                     b_ch => to_unsigned(100, 8)
                );
```

Constrained array

- *Tipo di array* in cui è definita la dimensione in fase di definizione
- Ogni segnale dichiarato con questo tipo sarà della stessa dimensione
- Può essere anche un *subtype*
- Struttura di un *constrained array*

```
type TYPE_NAME is array (RANGE) of ANOTHER_TYPE;
```

- Esempi di *constrained array*

```
type MySimpleArray is array (0 to 10) of Another_type;

type img_type is array (0 to 15, 0 to 15) of RGB_Type;
subtype byte_type is std_logic_vector(7 downto 0);
```

Unconstrained array

- *Tipo di array* in cui **non è** definita la dimensione in fase di definizione
- Ogni segnale dichiarato con questo tipo potrà avere una dimensione diversa
- Può essere anche un *subtype*
- Nel momento in cui si dichiara il segnale si deve specificare un *range* per definirne la grandezza. Il *range* può non essere un *integer* ma anche un *natural*

```
type TYPE_NAME is array (TYPE range<>, TYPE range<>, ...) of ANOTHER_TYPE;
```

- Esempi di *unconstrained array*

```
type my1DArray_type is array (integer range <>, integer range <>)
    of std_logic_vector(7 downto 0);

signal my1DArray : my1DArray_type(1 to 3);


type my2DArray_type is array (integer range<>, integer range<>)
     of std_logic_vector(7 downto 0);

signal my2DArray : my2DArray_type(1 to 3, 5 downto 0);
```