

VHDL: Syntax

Franco Fummi
Alessia Bozzini



UNIVERSITÀ
di VERONA
Dipartimento
di **INFORMATICA**

Contents

1 Built-in Types, Expressions and Operators

2 Architecture Declaration

2.1 Structural VHDL

- Signals
- Components
- Netlist

2.2 Dataflow VHDL

- Concurrent statements

2.3 Behavioral VHDL

- Variables
- Sequential Statements

3 Advanced Topics

- User Defined Types
- Subprograms
- Resolution Functions
- Attribute
- File

Built-in Types

- Types determine the values an object can assume and operations that can be performed on it.
- Packages STANDARD and IEEE provide several data types and operators.
 - A Scalar types
 - B Array types

A

Built-in Scalar Types

- real: -1.0E-38 to +1.0E+38.
- integer, positive and natural (32 bit). *VHDL is currently used to Model hw sys at RTL (Register Transfer Lvl)*
- boolean: false, true. *VHDL attualmente è utilizzato per*
- character: 'a', 'b', 'c', ...
- bit: '0', '1'. *Meglio poi spostarsi da std_logic a bit*
- time: number plus physical unit (fs, ps, ns, us, ms, sec, min, hr)

IL PIÙ
INTERESSANTE

std_logic: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'

- U: uninitialized
- X: forcing an unknown
- 0: forcing 0
- 1: forcing 1
- Z: high impedance
- W: weak unknown
- L: weak 0
- H: weak 1
- -: don't care

EISD

IEEE Standard Logic

- Objects defined in package IEEE are “visible” when:
Dalla libreria IEEE carica il modulo STD_LOGIC_1164 e usa tutte (all) le definizioni

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
```
- There are two versions std_logic and std_ulogic.
 std_ulogic is the unresolved version of std_logic.

B

Built-in Array Types

- String: “hold time error”.
- Bit_Vector: “00000100”.
- Std_Logic_Vector: “101Z”
- Sometime it can be necessary to explicitly provide the type name.

For example:

– `string'("10")`



Qualified expression

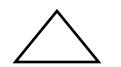
Names- Identifier

- All names must begin with an alphabetic letter (a-z), followed by a letter, an underscore, or a digit.
- VHDL is case insensitive ($xyz \equiv xYZ$).
- Two different objects cannot have the same name.
- Elements requiring unique names are:
 - Two entities in a library.
 - Two architectures of a single entity.
 - Two processes within the same architecture.

Range Constraint

- A range constraint declares the valid values for a particular type

```
integer range 1 to 10;
```



range_constraint

range *index_constraint*

index_constraint

{*low_val* **to** *high_val* | *high_val* **downto** *low_val*}

Scorre il vettore al contrario

Ordine di bit del + significativo al - => ho i segnali nell'ordine giusto!!

Expressions

- An expression is a formula that uses operators and defines how to compute or qualify a value.
- Operands must be of the same type. Type conversion can be done through conversion functions or by using qualified expressions:

integer(3.0)

signed' ("1010")

- There are four kinds of operators:
 - Logical
 - Relational
 - Arithmetic
 - Concatenation

Operators

L

Precedence

H

Logic operators	and	Logical And
	or	Logic Or
	nand	Complement of And
	nor	Complement of Or
	xor	Logical Exclusive Or
Relational operators	=	Equal
	/=	Not Equal
	<	Less Than
	<=	Less Than or Equal
	>=	Greater Than or Equal
Concatenation operator	&	Concatenation
Arithmetic operators	+	Addition
	-	Subtraction
Arithmetic operators	+	Unary Plus
	-	Unary Minus
Arithmetic operators	*	Multiplication
	/	Division
	mod	Modulus
	rem	Remainder
Arithmetic operators	**	Exponentiation
	abs	Absolute Value
Logical operator	not	Complement

New VHDL'92 Operators

- **sll**: shift left logical
- **sla**: shift left arithmetic
- **rol**: rotate left
- **xnor**: exclusive nor
- **ror**: rotate right
- **srl**: shift right logical
- **sra**: shift right arithmetic

Objects Declaration

- Each element of a VHDL description must be declared before its use.
- The only exceptions regard index of loops that can be an integer values only.
- Elements to be declared are:
 - constants
 - signals
 - entities
 - architectures
 - components
 - variables

*z mod:
per*

Signals → Valore che ha un'evoluzione nel tempo
Variables → Contenitore di un valore

Constant Declaration

- A constant is a name assigned to a fixed value
- Generally, constants increase readability

△*constant_declaration*

Scalar

```
constant name: type := expression;
```

Array

```
constant name:array_type[(index_constraint)] := expression;
```

- Examples:

```
constant Vdd: Real := -4.5;
```

```
constant FIVE: std_logic_vector(8 to 11) := "0101";
```

Signal Declaration

- Signals connect design entities and communicate changes in values between processes.
- The default initial value is the lowest value of the associated type, if not specified.

△ *signal_declaration*

```
signal names: type [range_constraint] [:= expression];
```

Assegno un valore iniziale

Scalar

```
signal names:array_type[(index_constraint)] [:=expression];
```

Array

Examples:

```
signal count: integer range 1 to 50;
signal SYS_BUS: std_logic_vector(7 downto 0);
signal bogus: bit_vector;
```



Entity Declaration -1

= to SystemC modules

NOV use I/O ports, o I o O (I/O now sono interfacciate)

- Design entities are used to represent VHDL models.
- They have a declaration part that defines the interface between the model and its environment.
- They have also a body that describes the relations between inputs and outputs of the model.
 - The model body must follow its entity declaration.
 - The entity declaration is mandatory but the architecture not.
 - The distinction between entity and architecture allows the definition of more than one architecture for the same entity.

Entity Declaration -2

△ *entity_declaration*

```

entity entity_name is
  [generic ({names: type[:= expression]} )];
  port ({names: direction type[:= expression]} );
end [entity_name];
  
```

DEFINE modifiable

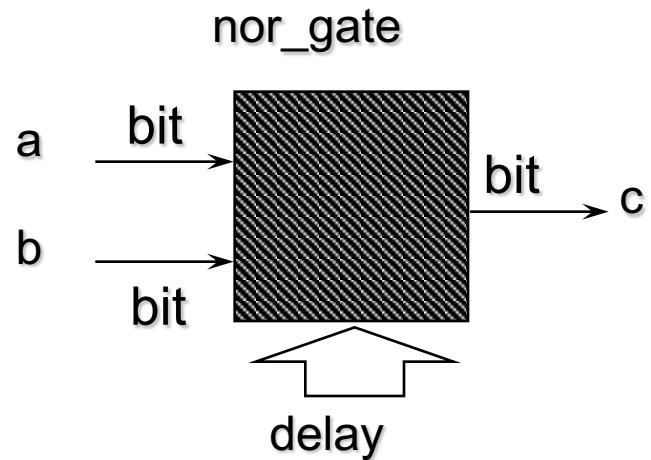
- Ports identifiers are used to interface the design entity with the environment:
 - IN: data enter into the design entity.
 - OUT: data come from the design entity.
 - INOUT: data enter into and come from the design entity.

Entity Declaration -3

- Example:

```
entity nor_gate is
  generic (delay: time := 5 ns);
  port (a,b: in bit;
        c:    out bit);
end nor_gate;
```

- a* and *b* can only receive values.
- c* produces the result of the computation.
- delay* is a constant value that can be used into the architecture body. It assumes the value of *5 ns* if it is not assigned.



Architecture Declaration-1

- An architecture design unit specifies the behavior, interconnections, and components of a previously compiled entity.
- It specifies the relationships between the inputs and outputs.



architecture_declaration

```
architecture architecture_name of entity_name is
    [declarations]
begin
    concurrent_statements
end [architecture_name];
```

Più di un'architettura per un'entità

Architecture Declaration-2

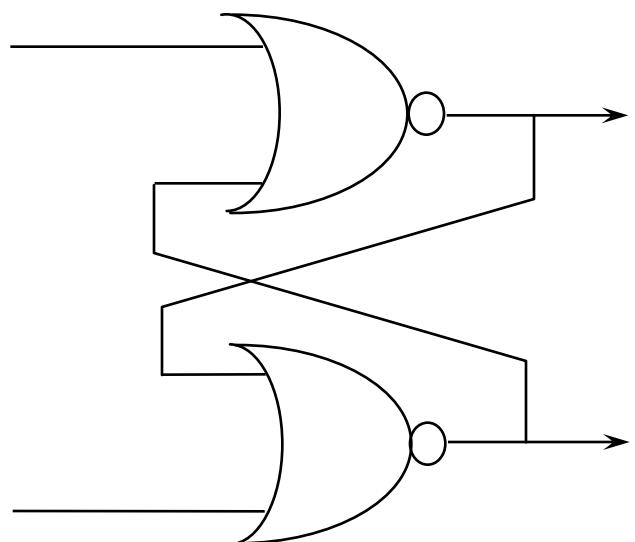
- Example:

```
architecture dataflow of nor_gate is
begin
    c <= a nor b after delay;
end dataflow;
```

- There are three architecture styles:
 - *Behavioral*: defines a sequentially described process.
 - *Dataflow*: implies a structure and a behavior.
 - *Structural*: defines interconnections of components.

Structural VHDL

- Structural style is similar to a netlisting language in other CAD systems



Components -1

- Component declaration and instantiation allow the structural kind of VHDL description
- Components must be *declared*, *specified* and *instantiated* for their use

△*component_declaration*

```
component component_name
  port({names: direction type[:= expression] })
end component;
```

Components -2

- Example:

```
entity rsflop is
    port(set,reset: in bit;
         q,qbar: inout bit);
end rsflop;
architecture netlist of rsflop is
component nor2
    generic(delay: time);
    port(a,b: in bit; c: out bit);
end component;
...
begin
...
end netlist;
```

Components -3

- The component configuration statement allows the specification of the selected architecture related to the declared component
- If no architecture is specified, the *default* architecture is selected

 *configuration_specification*

```
for names: comp_name use entity ent_name(arch_name); => collego un componente con un'entità
```

- Local names after the *FOR* statement specify the number of instantiated components.
- Example:

```
for u1,u2: nor2 use entity nor_gate(dataflow);
```

Components -4

△*component instantiation*

```
label: component_name port map( [named|positional]);
```

- Example:

```
U1: nor2 generic map (10.2 ns)
      port map(reset, qbar_int,q_int);
      port map(b => qbar_int, c => q_int, a => reset);
```

positional

named

→ Specifico il nome formale con quello attuale
 → CONSIGLIATO

- The local port names (reset, qbar_int, q_int) are put in relation with the formal names (a, b, c).
- The generic value 10.2 ns overrides the default value 5.0 ns.

Components -5

- Example:

```
architecture netlist of rsflop is
...
for u1,u2: nor2 use entity nor_gate(dataflow);
signal q_int, qbar_int: bit;
begin
  U1: nor2
    generic map (10.2 ns)
    port map(reset, qbar_int,q_int);
  U2: nor2
    generic map (10.3 ns)
    port map(q_int, set, qbar_int);
  q <= q_int;
  qbar <= qbar_int;
end netlist;
```

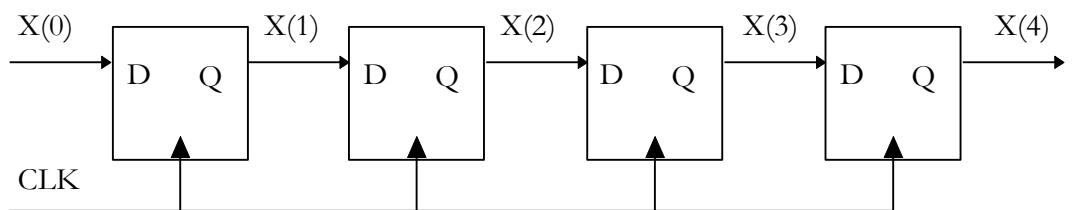
Generate Components

△ *generate_components*

```
label: for parameter in range generate
  component_instantiation
end generate;
```

- **Example**

```
gen1: for i in 0 to 3 generate
  U: dff port map (x(i), clk, x(i+1));
end generate;
```



- Nested *GENERATE* are allowed for bi-dimensional arrays
- Internal label must not be indexed
- Some customization can be done with the *if_generate* statement

Dataflow VHDL

- A set of VHDL statements is concurrently executed whenever they are placed into an architecture body.
- There are different versions of the same statement if it is executed concurrently or sequentially.
- Concurrent statements are:
 - *Signal assignment*
 - *Conditional signal assignment*
 - *Selected signal assignment*
 - *Instantiation statement*
 - *Block statement*
 - *Procedure call*
 - *Assertion statement*
 - *Process statement*

Signal Assignment -1

△*signal_assignment*

```
signal_name <= value;
```

DF: assegnamenti concorrenti (semplice, con delay,...)

- **Examples:**

architecture probe of halfadder is

With or without delays:

```

begin
  sum <= a xor b;
  carry <= a and b;
end probe;

```

sum <= a xor b after 5 ns; carry <= a and b after 10 ns;
--

- **Array values assignment:**

```

bus_out(4) <= data(5);
rotate_sig(7:0) <= sig(0:7);

```

Signal Assignment -2

- Aggregation

- Positional association:

```

SIGNAL z_bus : bit_vector (3 DOWNTO 0);
SIGNAL a_bit, b_bit, c_bit, d_bit : bit;
...
z_bus <= (a_bit, b_bit, c_bit, d_bit);

```

- Named association:

```

z_bus <= (2 => b_bit, 1 => c_bit, 0 => d_bit, 3 => a_bit);

```

- Others keyword:

```

z_bus <= (3 DOWNTO 2 => '1', OTHERS => '0'); → del bit 3 z/2, assegno '1'  

z_bus <= (OTHERS => '1' );
z_bus <= (2 => b_bit, 1 => c_bit, 0 => d_bit, 3 => a_bit);

```

Conditional Signal Assignment-1

 *conditional_assignment*

```
signal_name <= expression_1 WHEN condition_1 ELSE
    expression_2 WHEN condition_2 ELSE
    ...
    ...
expression_N;
```

- equivalent to *IF / THEN / ELSE / END IF*
- Each condition is a boolean expression.
- The expression of the *first* TRUE condition is assigned.
- There must be *always* an ELSE expression,
- The expression may be delayed.

a <= '1' AFTER 2 ns WHEN b = '0' **ELSE** *Quando ho un WHEN devo SEMPRE avere un ELSE*
 '0' AFTER 3 ns;

Conditional Signal Assignment-2

- Example:

```
ENTITY tri_state IS
    PORT(bit_1, en_1, en_2: IN std_logic;
        bus_1: IN std_logic_vector (0 TO 7);
        tri_bit: OUT std_logic;
        tri_bus: OUT std_logic_vector (0 TO 7) );
END tri_state;
ARCHITECTURE condition OF tri_state IS
BEGIN
    tri_bit <= bit_1 WHEN en_1 = '1' ELSE 'Z';
    tri_bus <= bus_1 WHEN en_2 = '1' ELSE (OTHERS => 'Z');
END condition;
```

Selected signal assignment

△ Selected signal assignment

```
with expression select
  signal_name <= expression_1 when choice_1,
  ...
  ...
  expression_n when choice_n;
```

- equivalent to *CASE / WHEN / END CASE*
- All choices must be included unless the *OTHERS* keyword is used
- A range may be used for a choice

- Example

```
with B select
  z <= '1' when "00" | "01",
  '0' when others;
```

- No overlapping in the choices is accepted

Block statement

- Conventional blocks represent a way to group any combination of concurrent statements that may appear into an architecture
- Blocks may contain further blocks thus implying an hierarchy
- Items declared within a block are only visible inside it

```

△ block_statement
  [label:] block [(guard_condition)]
  [declarations]
  begin
    concurrent_statements
  end block [label];

```

- Signals, constants, procedure, etc. may be declared into a block
- The guard condition must return a Boolean value:
it controls *guarded signal assignments* within the block

Guarded Block

- Whenever the guard condition evaluates to FALSE, the driver to any *guarded signal* is switched off.
- Example:

```
g_sig_es: block (clk = '1')
    sig_out <= guarded sig_in;
end block g_sig_es ;
```

```
no_g_proc:PROCESS(clk)
begin
    if(clk ='1) then
        sig_out <= sig_in;
    end if;
end process no_g_proc;
```

```
g_proc_es : block (clk = '1')
equiv : process
begin
    if guard then
        sig_out <= sig_in;
    end if;
    wait on guard;
end process equiv;
end block g_proc_es ;
```

Assertions -1

- Provide a method to communicate results, errors, ...

assertions_declaration

```
assert condition report string severity level;
```

assicurati che la condizione sia sempre verificata

- If the *condition* is *FALSE* the string is displayed on the simulator screen.
- Severity levels allow different kinds of simulation abort. They may be:
 - note, warning, error, failure, (*default is error*)
↳ con queste due opz la simulazione si ferma
- The concurrent ASSERT statement monitors the Boolean condition continuously.
- Assertions are used to debug the code or to provide information about the simulation.

Assertions -2

```

d_latch : PROCESS (clk,d)
  <-- process sensitive with these 2 signals
BEGIN
  c'è stato un evento su clock e adesso vale '1'
  IF (clk'EVENT and clk='1') THEN
    q <= d; assicurati che il tempo in cui d non sia modificato per almeno
    ASSERT d'STABLE (setup_time) "setup_time"
    REPORT "Setup violation ..."
    SEVERITY warning;
  END IF;
END PROCESS d_latch ;
  
```

Sequential assertion

```

d_latch : BLOCK (clk = '1')
  q <= GUARDED d;
  ASSERT clk'EVENT and clk = '1' and d'STABLE(setup_time)
  REPORT "Setup violation ..."
  SEVERITY warning;
END BLOCK d_latch;
  
```

Concurrent assertion

Behavioral VHDL

- Describe an architecture in a program-like style
 - Process statement
- A set of VHDL statements is sequentially executed whenever they are placed into process
 - Signal assignment
 - Variable assignment
 - Wait
 - Procedure call
 - Function call
 - Branches
 - Control flow
 - Assertion writes messages

NOTE: underlined statements are also concurrent statements

Variable Declaration

variable_declaration

```

variable names: type[range_constraint] [:= expression];
variable names:array_type[(index_constraint)] [:=expression];
  
```

Scalar

Array

usate solo dentro a processi

- Declaration examples:

```

variable sum : real;
variable voltage : integer := 0;
variable clock : bit := '1';
variable data : std_ulogic;
  
```

initial value?

*→ left-most significant value of the type
eg: per bit sarebbe '0'*

- Arrays:

```

variable data_bus : bit_vector (0 to 7) := "11111111";
variable inputs : std_ulogic_vector (15 downto 0);
  
```

Signals and Variables

- Declaration place:
 - Signals can be declared only between the *ARCHITECTURE* statement and its *BEGIN* (declarative part of the architecture)
 - Variables can be declared only between the *PROCESS* statement and its *BEGIN* (declarative part of the process)
- Default value:
 - Both objects assume the left-most or minimum value of the corresponding type

Signal vs. Variable Assignment-1

- Signal and variable assignments are performed by using different symbols to emphasize the different meaning of the two objects

△ *variable_assignment*

```
variable_name := value;
```

△ *signal_assignment*

```
signal_name <= value;
```

- Note that the assignment of the initial value to a signal uses the same symbol of variable assignment

Signal vs. Variable Assignment-2

- The main difference is the nature of the assignment:
 - Signal assignment → *concurrent statement*
 - Variable assignment → *sequential statement*
- Such a difference implies some other differences:
 - Variables can be assigned only in the sequential part of a VHDL description (into a process statement)
 - Signals can be assigned in the sequential or concurrent part
- A variable assignment takes effect *immediately!*
- A signal assignment may depend on a *delay*

Signal vs. Variable Assignment-3

```

var_ex: PROCESS
  VARIABLE num, sum: INTEGER:=0;
BEGIN
  WAIT FOR 20 ns;
  num := num + 1; @t=40ns:
  sum := sum + num; num=2 sum=3 (=1+2)
END PROCESS var_ex;
  
```

```

SIGNAL num, sum: INTEGER:=0;
sig_ex: PROCESS
BEGIN
  WAIT FOR 20 ns;
  num <= num + 1; @t=40ns:
  sum <= sum + num; num=2 sum=1
END PROCESS sig_ex;
  
```

L'assegnamento sarà effettuato a fine processo quindi 'sum' non vede 'num' già incrementato

Variabili: contatori intermedi per fare computazioni per poi assegnare il valore al segnale

- The two processes are apparently equal, but ...
- Do they produce the same result?
 - NO! Signals and variables are updated at different times.

Process -1

- It is a concurrent statement that delineates a set of sequentially executed statements

```
 process_statement
[label:] process [ (sensitivity_list) ]
[declarations]
begin
    sequential_statements
end process [label] ;
```

- The *sensitivity list* is a list of signals. The change of one or more of such signals causes the process to be activated
- Alternatively the WAIT statement may control the execution of a process
- The *sensitivity list* and the *WAIT* statement are mutually exclusive

Process -2

SET of activity to be performed before actual time starts

- Comparison

VHDL process = SystemC Thread
 Nothing is equals to SysC process

```
sens_list_style_proc:  

PROCESS (alarm_t, current_t)  

IF (alarm_t = current_t)  

THEN  

  sound <= '1';  

ELSE  

  sound <= '0';  

END PROCESS;
```

Sensitivity list \Rightarrow
 2 processes created if both signals change at the same time

```
wait_style_proc: PROCESS  

IF (alarm_t = current_t)  

THEN  

  sound <= '1';  

ELSE  

  sound <= '0';  

WAIT on alarm_t, current_t;  

END PROCESS wait_style_proc
```

- Both processes show the same behavior.
- Signals in the sensitivity list help the reader to understand the behavior of the process.
- Multiple WAIT statements may represent a more complex behavior.

Process -3

- Execution:
 - Every process is executed once in the initialization phase
 - a process based on the sensitivity list runs until its last instruction
 - a process based on the wait keyword runs until the first wait
 - A process is restarted when a signal in the sensitivity list or in the wait statement changes
- A process is considered as a *UNIQUE* concurrent operation
- Signals of a process are all updated at the end of the process execution
- All internal operations are sequentially executed, thus only sequential operators can be used

Wait -1

- Provides the control of the process execution

 `wait;`

- suspends a process indefinitely (useful in test benches)

 `wait for time; E.g.: wait for 20 seconds`

- suspends a process for time units (useful in test benches and behavioral models)

 `wait on signal_list;`

- suspends a process until a change occurs on one or more of the signals in the list (it is equivalent to the sensitivity list)

 `wait until condition;`

- suspends a process until a change occurs on one or more of the signals in the condition and it evaluates to TRUE.

Wait -2

- Examples:

```
stimuli : process
begin
  en_1 <= '0';
  en_2 <= '1';
  wait for 10 ns;
  en_1 <= '1';
  en_2 <= '0';
  wait for 10 ns;
  en_1 <= '0';
  wait;
end process stimuli ;
```

d_ff_1 : process
 begin
wait until clk'event and clk='1';
 q <= d;
 end process d_ff_1 ;

semantically equivalent to:
wait until clk='1';

d_ff_2 : process begin
 if clk='1' then
 q <= d;
 end if;
wait on clk;
 end process d_ff_2 ;

Branches -1

△ if_statement

```
if condition then sequential_statements
{elsif condition then sequential_statements}
[else sequential_statements]
end if;
```

- Example:

```
counter: process (clk, reset)
begin
    if reset = '1' then
        count <= '0';
    elsif clk'event and clk = '1' then
        if count >= 9 then
            count <= '0';
        else
            count <= count + 1;
        end if;
    end if;
end process counter;
```

Branches -2

△*case_statement*

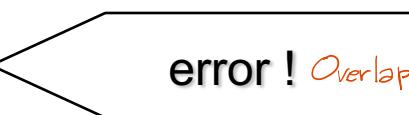
```

case expression is
  when choice-1 => sequential_statements
  ...
  ...
  when choice-n => sequential_statements
end case;

```

- Equivalent to *WITH /SELECT*
- *ALL* possible choices must be included, *RANGE* is allowed.
- Choices cannot overlap
- Example:

```

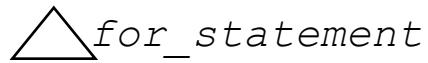
CASE int_a IS
  WHEN 0          => z <= a;
  WHEN 1 TO 3     => z <= b;
  WHEN 2 | 6 | 8  => z <= c;    
  WHEN OTHERS      => z <= 'X';
END CASE ;

```

error ! Overlapping conditions

Loops

!!
DON'T USE LOOPS



```
[label:] for index in range loop
    sequential_statements
end loop [label];
```

- *index* is automatically declared as integer and cannot be modified within the loop
- *range* may be an enumerative type



```
[label:] [while condition] loop
    sequential_statements
end loop [label];
```

- *condition* is tested before each iteration

Control Flow

△*exit_statement*

```
  exit [label:] [when condition];
```

- terminates the execution of a while, for, loop
- exit may be conditioned and it allows the exit from any loop even if it is not the innermost one

```
11: FOR i IN 0 TO 7 LOOP
    12: FOR j IN 0 TO 7 LOOP
        EXIT 11 WHEN quit_both_loops = '1'
```

△*next_statement*

```
  next [label:] [when condition];
```

- terminates of the current iteration of a while, for, loop
- it may be conditioned and it allows the termination of an iteration of any loop

Advanced Types -1

- The enumerated type declaration lists a set of names or values defining a new type

△*enumerated_type_declaration*

```
type identifier is (item, {item});
```

- Built-in scalar types: (standard packages)

```
type bit is ('0', '1') ;
type character is ('a', 'b', 'c', ... ;
type boolean is (false, true) ;
type std_ulogic is ('u', 'x', '0', '1', 'z', 'w', 'l', 'h', '-' );
```

- Built-in scalar physical types: (standard packages)

```
type time is range -922337036854775808 to 92...
units
  fs; ps = 1000 fs; ... hr = 60 min ;
end units
```

Advanced Types -2

△ array_type_declaration

```
type array_type_name is array (range) of type;
```

- **Example**

```
type string is array (positive range <>) of character;
```

```
type bit_vector is array (natural range <>) of bit;
```

- Subtypes are based upon existing type and is a restriction of that type in some way using a range constraint.

△ subtype_declaration

```
subtype subtype_name is type_name range range;
```

Don't abuse abstract datatypes=>THEY WILL EVENTUALLY BE TRANSFORMED INTO BITS

Advanced Types -3

- **Subtype examples:**

```
SUBTYPE natural IS integer RANGE 0 TO 2147483647;  
TYPE car IS (ford, buick, chevy, chrysler);  
SUBTYPE gm IS car RANGE buick TO chevy;  
TYPE data IS ARRAY (natural RANGE <>) OF bit;  
SUBTYPE low_range IS data range (0 TO 7);  
SUBTYPE high_range IS data range (8 TO 15);
```

- A subtype does not represent a new type

Advanced Types -4

- Record types allow the group of objects of different types into a single object

record_declaration

```
type record_type_name is
  record
    identifier : type;
    ...
  end record;
```

- Example:

```
type instruct is record
  source: integer range 0 to 7;
  det: integer range 0 to 15
end record;
```

Procedure call -1

- Represents a method to perform complex operation
- May produce multiple output values:
 - may affect input parameters (INOUT type);
 - may have OUT parameters
- Parameters may be signals, variables, constants

```
△ procedure_declaration
  procedure proc_name (parameters) is
    [declarations]
  begin
    sequential_statements
  end proc_name;
```

- Procedures are concurrently executed whenever any of their IN or INOUT parameters changes
- Procedure can contain **wait** statements
- Local variable are initialized each time the procedure is called

Procedure call -2

- Example:

```
procedure find_min (variable values : in int_array;
                    variable min_val : inout integer ;
                    variable old_min : out integer) is
    variable temp : integer;
begin
    temp := old_min := min_val;
    for i in values'range loop
        if values(i) < temp then
            temp := values(i)
        end if;
    end loop;
    min_val := temp;
end find_min;
...
find_min (my_array, minimum, old_value);
```

Functions -1

- Return the result of a computation. They may be used in any expression, in either a concurrent or sequential statement.
- Declaration:
 - it may be separated by its body part.
 - it must be placed before its body and before its first use.

```
function_declaration
  function fun_name (parameters) return type;
function_body
  function fun_name (parameters) return type is
    [declarations]
  begin
    sequential_statements and return
  end [fun_name];
```

Functions -2

- A function body can contain any sequential statement except *SIGNAL* assignments and *WAIT* statements.
- Local variables do not retain values between successive calls; they are re-initialized each time.
- Functions are described into the *package* body or *architecture* declarative part.

```
function bit_to_boolean (bit_in : in bit) return boolean is
begin
    if bit_in = '1' then return true;
    else return false;
    end if;
end bit_to_boolean;
```

Overloading -1

- The way of giving more than one meanings to the same item.
- Overloading possibilities:

- enumeration identifiers

```
type count_cnt is (load, clear, accumulate);  
type reg_cnt is (hold, clear, load);
```

- functions and procedures

```
function min (a, b : integer) return integer;  
function min (float_a, float_b : real) return real;
```

- operator symbols

```
function "+" (a : state; b : integer) return state;
```

Overloading -2

- Subprograms selection:
 - number of parameters
 - types of parameters
 - names of parameters (named association)
 - return type

```
SIGNAL res : real;  
SIGNAL in1, in2 : integer;  
res <= min (in1, in2);
```

- Overloaded operator call:
- function notation
- operator notation

```
x := "+" (y, z);
```

```
x := y + z ;
```

Signal Drivers

IMPORTANTE

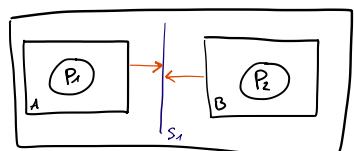
- They are containers for the assignments scheduled for a signal.
- A driver is created every time a signal assignment is made.
- Example:

```
clock <= '0', '1' after 5 ns, '0' after 10 ns;
```

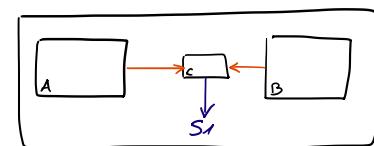
Time	Value
0	0
5	1
10	0

- Times must be in ascending order.
- Multiple executions of the same assignment modify the driver.
- Multiple concurrent statement assignments create multiple drivers which must be *resolved*.

LEZIONE 23/10



↓
error for
compiler



↓
OK ⇒ intermediate component
EISD

Resolution Functions -1

- Definition:
 - subprograms defining the single value that the signal should assume when there are multiple values concurrently assigned
 - Input: array that contains the current value of all drivers
 - Output: the selected single value

```

type s_state is ('x','0','1','z');      begin
architecture ds of exam is
  signal o : wired_or s_state;
  signal a, b, c : s_state;
  signal e1, e2, e3 : bit;
begin
  o <= a when e1 = '1' else 'z';
  o <= b when e2 = '1' else 'z';
  o <= b when e3 = '1' else 'z';
end
  
```

Resolution Functions -2

- Multiple drivers are created for the same signal o.
- Conflict assignments may occur.
- Es:
 - $e1 = '1' \text{ AND } e2 = '1'$
 - $e1 = '1' \text{ AND } e2 = '0' \text{ AND } e3 = '0'$
- Resolution function:

```
FUNCTION wired_or (dr_out: s_state) RETURN s_state IS
...
```

	1	X	0	Z
1	1	X	X	1
X	X	X	X	X
0	X	X	0	0
Z	1	X	0	Z

Attributes

- General attributes can be attached to variables also:

`x'high` `x'low` `x'left` `x'right`

- Attributes for array types:

`array'range` `array'reverse_range` `array'length`

- Example in a resolution function:

```
function wired_or (dr_out: s_state) return s_state is
begin
    for i in dr_out'range
```

- User defined attributes:

```
ATTRIBUTE clock_source OF ck: SIGNAL is TRUE;
```

Signal Attributes

- Attributes of a signal are automatically generated and can be obtained by using the ' symbol

only one which is synthesizable
 ↗

signal'EVENT	signal'QUIET(t)	signal'LAST_EVENT
boolean	boolean	boolean
signal'ACTIVE	signal' TRANSACTION	signal'LAST_ACTIVE
boolean	boolean	time
signal'STABLE(t)	signal'DELAYED(t)	signal'LAST_VALUE
boolean	signal	value

- Example:

```
if (clock = '1' and clock'active and clock'last_value = '0') then
  CLOCK_EVENT & CLOCK=1  ⇒ equivalent
```

File I/O -1

- Access to files is provided by the textio package specified by IEEE:

```
type line is access string;
type text is file of string;
procedure readline (logical_file_name, line_name);
```

- reads a line of strings from the file

```
procedure read (line_name, object_name);
```

- extracts an object from the line

```
procedure writeline (logical_file_name, line_name);
```

- writes a line of strings to the file

```
procedure write (line_name, object_name);
```

- writes an object to the line

```
function endfile (op : IN text) RETURN boolean;
```

File I/O -2

- File declaration:

```
file logical_name : type is mode "physical name";
```

- Mode may be IN or OUT

- Example:

```
file data_in : text is in "./input_file";
```

- File data analysis:

```
while not (endfile(data_in)) loop
```

- Data read:

```
variable in_line : line;  
readline (data_in, in_line);  
read (in_line, object1) ; read (in_line, object2) ... ... ...
```

File I/O -3

- **Data write:**

```
file data_out : text is out "./output_file" ;
variable out_line : line;
...
...
write (out_line, object1) ; write(out_line, object2) ... ... ...
writeln (data_out, out_line) ;
```

- **File open and close:**

- Files are automatically open at the beginning of the simulation and close at the end

- **Primary uses:**

- store simulation results
 - application of stimuli