Universidad
Carlos III de Madrid

Fundamentals of VHDL digital circuit design

INTEGRATED CIRCUITS AND MICROELECTRONICS

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Introduction to VHDL

- VHDL acronym from VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (HDL)
- It is different from software programming languages, like C or Java. It was devised to **describe hardware,** which implies **concurrence** (tasks are performed in parallel)
- It allows designers to develop circuits with higher complexity than graphical-based methods (based on schematics) and in a more productive manner.
- It is a IEEE (Institute of Electrical and Electronics Engineers) standard (last update in 2008, **IEEE Std 1076™-2008**)

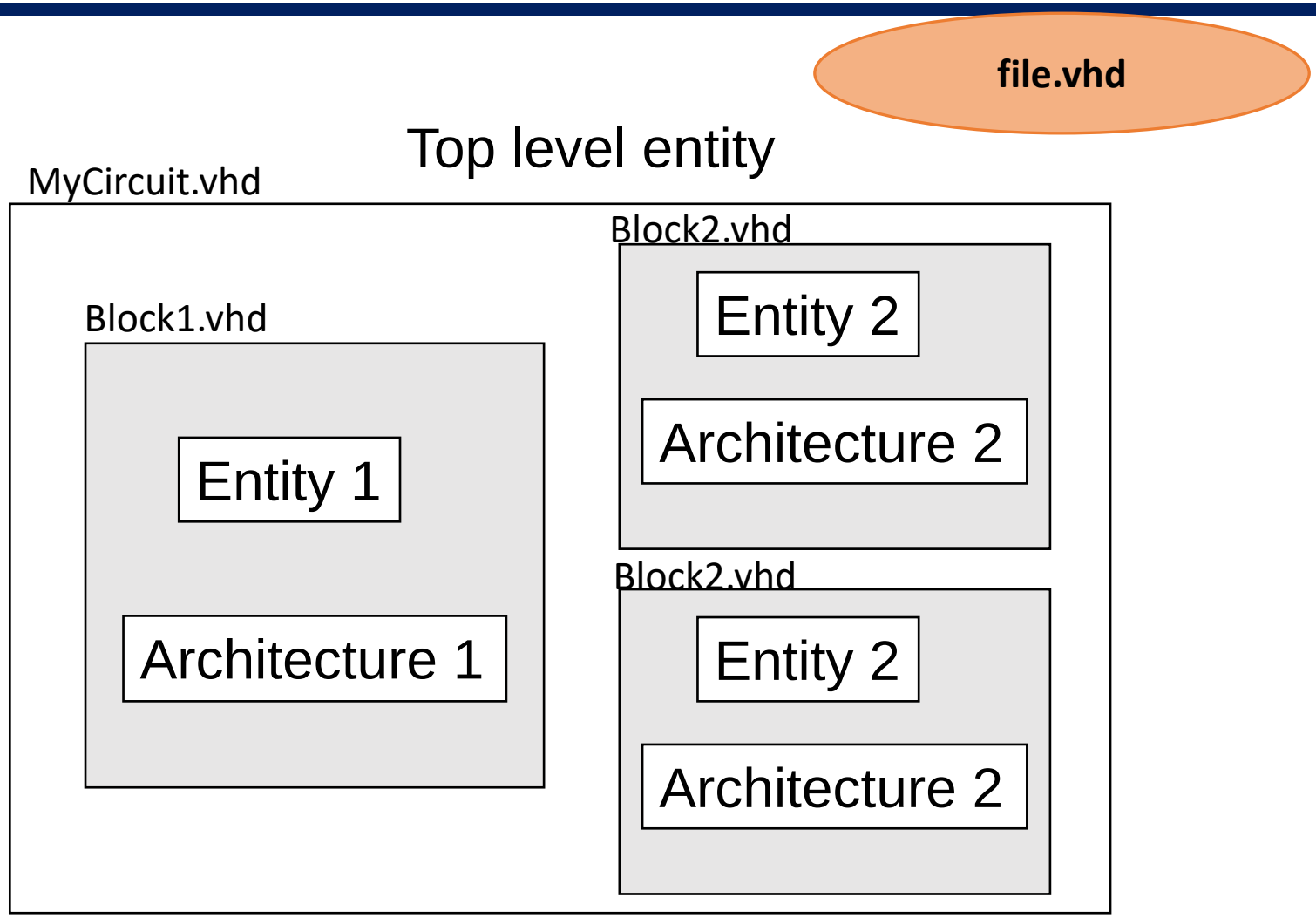
Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Hierarchy and basic blocks for design

- VHDL can be used to specify hierarchical designs by connecting several basic design units.
- In order to describe a circuit, VHDL provides the design units **entity** and **architecture**
  - The **entity** provides the information related with the circuit interface: input and output ports
  - The **architecture** describes the functionality implemented by the circuit and it is always linked to a given entity.

Hierarchy and basic blocks for design



Typical structure of a VHDL file

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY circuit_name IS
      PORT (
            input1     : IN    data_type;
            input2     : IN    data_type;
            output1    :OUT data_type;
            output2    :OUT data_type
       );
END circuit_name;

ARCHITECTURE arch_name OF circuit_name IS
      -- Declarations
BEGIN
      -- Operations
END arch_name ;
```

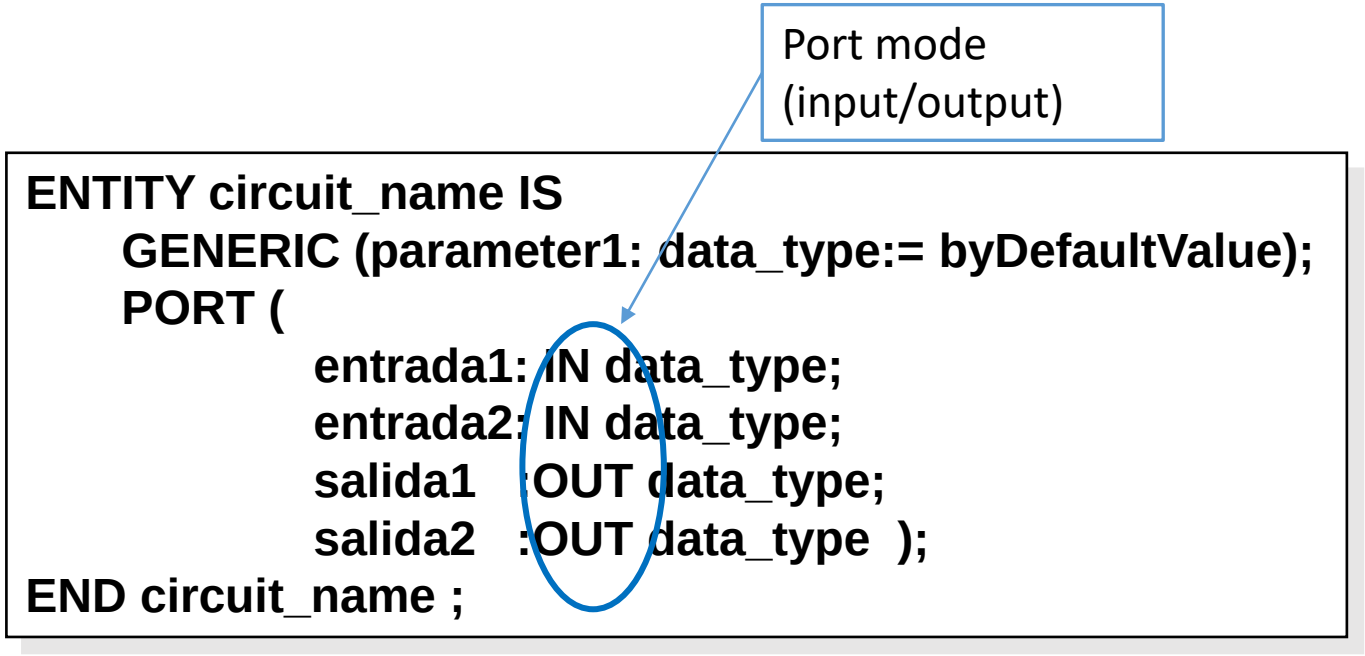
Libraries, packages

Entity: ports

Architecture: behavior and components

Hierarchy and basic blocks for design

- The entity declaration consists of
  - Generic parameters
  - Interface declaration (inputs and outputs): ports



Hierarchy and basic blocks for design

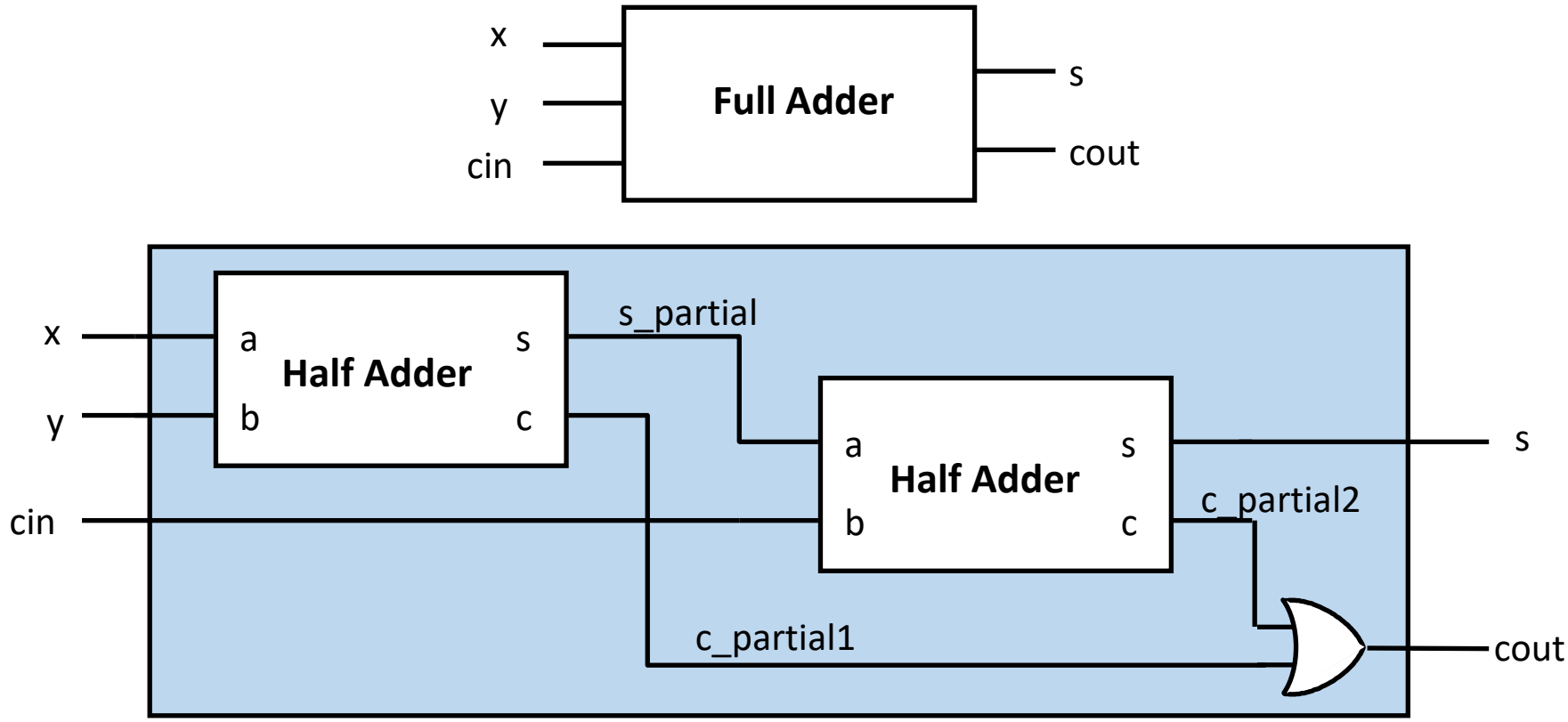
- The architecture contains
  - Declarations
  - Component instances (lower level entities) for hierarchical designs.
  - Statements to describe the behavior of the circuit

Comments

```
ARCHITECTURE arch_name OF circuit_name IS
    -- Declarations
BEGIN
    -- Statements and component instances

END arch_name;
```

Hierarchical example



Hierarchical example

```
ENTITY full_adder IS
      PORT ( x: IN BIT; y: IN BIT; cin: IN BIT;
                  s: OUT BIT; cout: OUT BIT);
END full_adder;

ARCHITECTURE structural OF full_adder IS
      COMPONENT half_adder
            PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
      END COMPONENT;
      SIGNAL s_partial: BIT;
      SIGNAL c_partial1, c_partial2: BIT;
BEGIN
      SS0: half_adder PORT MAP (x, y, s_partial, c_partial1);
      SS1: half_adder PORT MAP (s_partial, cin, s, c_partial2);
      cout <= c_partial1 OR c_partial2;
END structural;
```

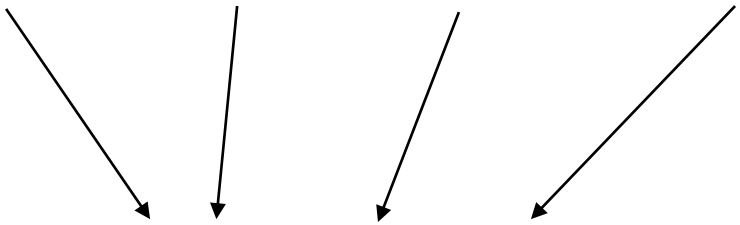
Port Association

- Port association has been made by position

```
COMPONENT semisumador
        PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END COMPONENT;

    ......

SS0: semisumador PORT MAP (x, y, s_parcial, c_parcial1);
```



Explicit port association

- Alternatively, port association can be made explicitly

```
COMPONENT semisumador
        PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END COMPONENT;

    ......

SS0: semisumador PORT MAP (a => x, b => y , s => s_parcial, c =>c_parcial1);
```

Packages

- A convenient way to group declarations

```
PACKAGE util IS
      COMPONENT half_adder
            PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
      END COMPONENT;
END util;
```

Packages

- A convenient way to group declarations

```
USE WORK.util.ALL;
ENTITY full_adder IS
        PORT ( x: IN BIT; y: IN BIT; cin: IN BIT;
                s: OUT BIT; cout: OUT BIT);
END full_adder;

ARCHITECTURE structural OF full_adder IS
        SIGNAL s_partial: BIT;
        SIGNAL c_partial1, c_partial2: BIT;
BEGIN
        SS0: half_adder PORT MAP (x, y, s_partial, c_partial1);
        SS1: half_adder PORT MAP (s_partial, cin, s, c_partial2);
        cout <= c_partial1 OR c_partial2;
END structural;
```

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Statements

- There are two types of VHDL statements depending on how they are executed during simulation:
  - Concurrent statements are run in parallel by the simulator.
  - Sequential statements are executed one after another, following the specified flow.
- All the statements must finish with the character *;*
- All the statements in an **architecture** are **concurrent**

```
ARCHITECTURE structural OF full_adder IS
…
BEGIN
        SS0: half_adder PORT MAP (x, y, s_partial, c_partial1);
        SS1: half_adder PORT MAP (s_partial, cin, s, c_partial2);
        cout <= c_partial1 OR c_partial2;
END structural;
```

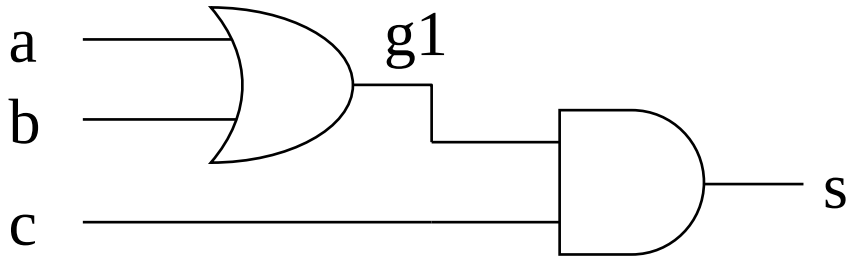
Concurrent Statements

- The order in which these statements are written is irrelevant. They are executed at the same time.
  - Event-driven simulation: an event on the inputs triggers the execution of the statement to update the outputs
- Two equivalent examples:

ASSIGNMENT

```
ARCHITECTURE a OF circuit IS
BEGIN
    g1 <= a OR b;
    s <= g1 AND c;
END a;
```

```
ARCHITECTURE a OF circuit IS
BEGIN
    s <= g1 AND c;
    g1 <= a OR b;
END a;
```



Sequential Statements

- Concurrence is difficult to handle:
  - Sequential statements can be used to describe a design
- Sequential statements:
  - They are executed one after another, as in software programming languages
  - They are always included in special containers: **processes** or **subprograms** (functions and procedures)
  - The execution of a sequential statement has not an immediate effect on the simulation waveform: only the final effect of the sequential statements included in the container is considered!
  - Time does not advance between the execution of one sequential statement and the next one.

Statements : Concurrent

- Process

```
PROCESS(sensitivity_list)
  --declaration of local objects
BEGIN
  --sequential statements
END PROCESS;
```

- Sensitivity list: list of signals that the simulator engine checks to know when the process must be executed. When one signal in the sensitivity list changes, then the simulator executes the statements within the process.
  - If there is no sensitivity list, the simulator executes the sequential statements until a WAIT statement is found. This statement is explained in the simulation lesson

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Objects

- Data objects
  - Constants
  - Variables
  - Signals

As in the case of programming languages

They represent hardware signals with values that change with time (wires)

Objects

- Constants
  - Represent a constant value
  - They can be declared in any part of a design

```
CONSTANT name1, name2, ..., nameN: type:= value;
```

  - Example

```
CONSTANT gnd : BIT := '0';
CONSTANT n, m : INTEGER := 3;
CONSTANT delay: TIME := 10 ns;
```

Objects

- Variables
  - They can change their value
  - The update of the value is performed just after the assignment
  - They have to be declared in sequential environments, like in processes or subprograms
  - They are local data, that is, variables are only visible in the process or subprogram where they are declared. There are no global variables
  - They cannot be graphically represented in simulation (waveform)

```
VARIABLE name1, name2, ..., nameN: type [ := value];
```

```
VARIABLE one_variable : BIT := '1';
VARIABLE i, j, k: INTEGER;
```

Objects

- Signals
  - Their values always have an associated time
  - Signal assignments do not cause a change in the signal value immediately, but after certain specified time. The pair (value, time) is called *transaction*
  - Signals can only be declared in concurrent environments. However they are global objects and are also visible in sequential environments, like processes or subprograms
    - **They are the only mechanism to communicate two processes**
  - They are visible in simulation (waveform)

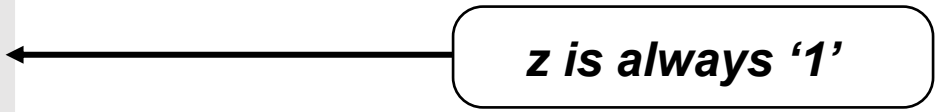
```
SIGNAL name1, name2, ..., nameN: type [ := value];
```

```
SIGNAL one_signal : BIT := '1';
SIGNAL i, j, k: INTEGER;
```

Objects

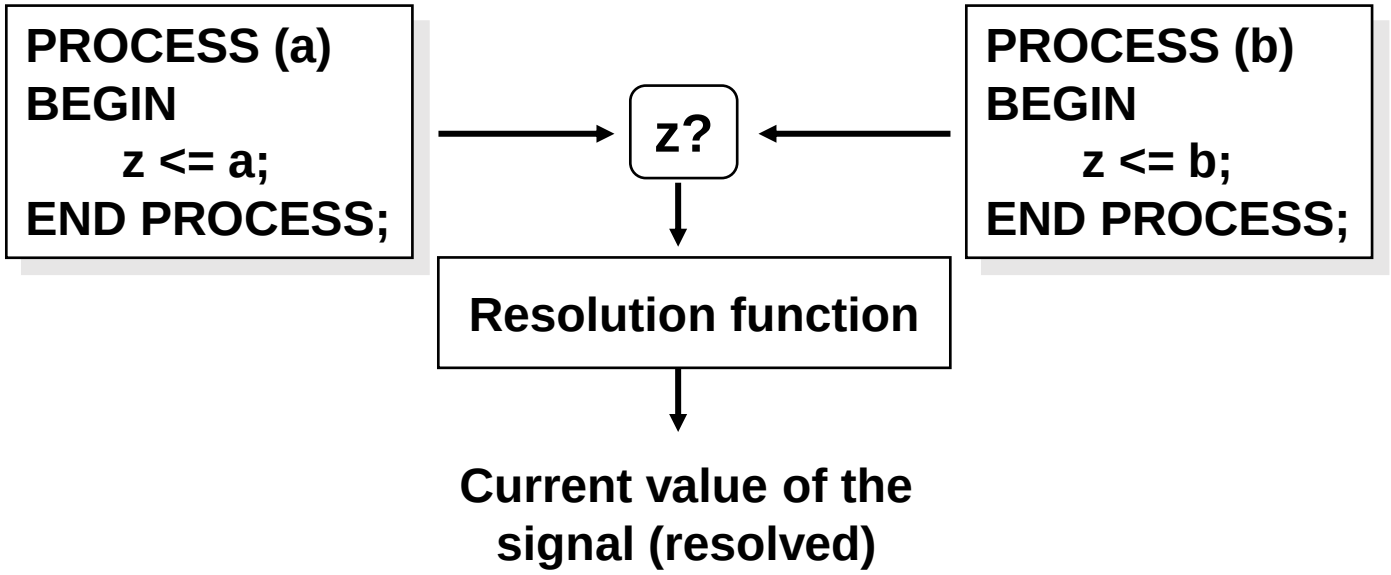
- Signal assignments
  - If a process contains several assignments for the same signal, there is only one driver, and then, the final value for the signal will be the last one assigned

```
PROCESS (a)
BEGIN
      z <= a;
      z <= '1';
END PROCESS;
```



Objects

- Signal assignments
  - If the same signal is assigned in different concurrent statements, there is a driver for each process (multiple drivers for the same signal). Therefore, the final value must be resolved.
  - The value of the signal must be determined by a resolution function



Objects : Initial values

- Simulation:
  - In order to simulate a design, every signal and variable should have an initial value. The initial value can be specified in the object declaration
  - If no initial value is specified, the simulator assumes a default value. In enumerated types, it is the first one in the list

```
TYPE state IS (S0, S1, S2, S3);
SIGNAL s: state := S3;          -- Initial value: S3
SIGNAL s: state;                -- Initial value: S0
SIGNAL a: BIT;                  -- Initial value: '0'
SIGNAL b: STD_LOGIC;            -- Initial value: 'U'
```

Objects : Initial values

- Synthesis
  - The synthesis tools do not take into account initial values
  - Initialization must be modeled in the design through the reset signal (as it actually occurs in hardware)

```
SIGNAL b: STD_LOGIC;
....
PROCESS (reset, ....)
BEGIN
        IF reset = '1' THEN
                b <= '0';
...
```

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Data types

- Scalars
  - Enumerated
  - Integer
  - Physical
  - Real
- Composite
  - Array
  - Record
- File

Data types: Scalars

- Enumerated
  - Values are identifiers or character literals
  - They can be pre-defined (by the language standard) or defined by the user
    - **Type declaration**

```
TYPE set_of_letters IS (‘A’, ‘B’, ‘C’, ‘D’);
TYPE traffic_light IS (green, yellow, red);
TYPE states IS (s0, s1, s2, s3, s4, s5);
```

    - **Use**

```
CONSTANT first_letter : set_of_letters := ‘A’;
SIGNAL traffic_light1: traffic_light ;
SIGNAL current_state: states ;
...
current_state <= s0;
traffic_light1 <= green;
```

Data types: Scalars

- Enumerated: Predefined enumerated types
  - BIT (‘0’, ‘1’)
  - BOOLEAN (FALSE, TRUE)
  - CHARACTER (NUL, …, ‘A’, ‘B’, …)
  - STD_LOGIC (‘U’, 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
    - **Defined by the IEEE 1164 standard**
    - **‘U’** **Uninitialized. This is the default value**
    - **‘X’** **Unknown (strong)**
    - **‘0’** **Logic Zero (strong). Gnd**
    - **‘1’** **Logic One (strong). Vdd**
    - **‘Z’** **High impedance**
    - **‘W’** **Unknown (weak)**
    - **‘L’** **Logic zero (weak). Pull-down resistors**
    - **‘H’** **Logic one (weak). Pull-up resistors**
    - **‘-’** **“don’t-care”. Used for synthesis**

Careful!

Data types: Scalars

- Enumerated: predefined
  - In order to use standard logic types we have to add the following lines before the design entity declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY...
```

Data types: Scalars

- Integer
  - It is advisable to fix a range
    - **It limits the number of bits inferred in the synthesis**
    - **The simulation sends an error message when the object has a value out of range**
  - Type declaration and usage

```
CONSTANT n: INTEGER := 8;
SIGNAL i, j, k: INTEGER;
...
i <= 3;
```

```
SIGNAL bcd_value: INTEGER RANGE 0 TO 9;
```

Data types: Scalars

- Real
  - It is not synthesizable. It is used in simulation: there are libraries that include components to operate with real objects
  - Type declaration and usage

```
SIGNAL d: REAL;
...
d <= 3.1416;
d <= 10E5;
d <= 2.01E-3;
```

Data types: Scalars

- Physical
  - Integer+ units of a physical magnitude
  - They are not synthezisable.
  - The most important one is the predefined type **TIME**

FS	femtosecond = 10^{-15} sec.
PS	picosecond = 10^{-12} sec.
NS	nanosecond = 10^{-9} sec.
US	microsecond = 10^{-6} sec.
MS	millisecond = 10^{-3} sec.
SEC	second
MIN	minute = 60 sec.
HR	hora = 60 minutes

```
SIGNAL t: TIME;
...
t  <= 10 NS;
```

Data types: Composite

- ARRAY
  - All the elements are of the same type, which can be any VHDL type.

```
TYPE byte IS ARRAY ( 7 DOWNTO 0 ) OF STD_LOGIC;
SIGNAL s: byte;
```

  - Unconstrained vectors (the size is specified when an object is declared)

```
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
SIGNAL s: bit_vector (7 DOWNTO 0);
```

  - It is possible to access to one element or to a subset of them

```
s <= “00000000”;
s(2) <= ‘1’;
s(4 downto 3) <= “00”;
```

  - The direction of the range can be ascending (TO) or descending (DOWNTO)
    - Subsets of elements must be accessed in the same direction used in the original vector declaration

Data types: Composite

- Array: vectors or memories
  - Vectors of data types BIT and STD_LOGIC are predefined

```
SIGNAL s: BIT_VECTOR (7 DOWNTO 0);
SIGNAL a: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

  - Multidimensional array and array of arrays

```
TYPE matrix IS ARRAY(0 TO 479, 0 TO 479) OF STD_LOGIC;
SIGNAL image: matrix ;
TYPE memory1 IS ARRAY(0 TO 1023) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL ram1: memory1;
TYPE memory2 IS ARRAY(0 TO 1023) OF INTEGER RANGE 0 TO 255;
SIGNAL ram2: memory2;
```

```
image(0,0) <= '1';
ram1(0)(0) <= '1';
ram1(0) <= "00000000";
image(0, 0 TO 7) <= "00000000";  -- ERROR
```

Data types: Interpretation of data types in synthesis

- INTEGER:
  - It is synthesized as a number coded in natural binary.
  - If the range contains negative numbers, it is synthesized as a number coded in **two's complement**
  - It is necessary to write the range in the declaration in order to make an efficient synthesis

```
SIGNAL a: INTEGER;                  -- 32 (64) bits
SIGNAL b: INTEGER RANGE 0 TO 7;     -- 3 bits
```

- Enumerated:
  - The are synthesized as a number coded in natural binary. A binary code is assigned to every value in the appearance order
- The following types are not synthezisable: REAL, physical, FILE

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Operands: Identifiers

- Identifiers are used to name objects. They consist of alphanumeric characters and '_'.
- It is advisable to choose names that point out what the object represent.
- VHDL is not case sensitive

Operands: Literals

- Symbols used for representing constant values in VHDL
- Characters: always between single quotation marks

```
'0'   '1'   'Z'
```

- Array of characters (string literals): between double quotation marks

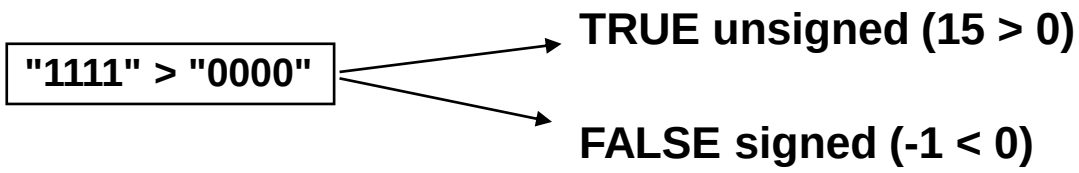
```
“This is a message"
"00110010"
```

- Bit string literals: a prefix points out the base

```
SIGNAL b: STD_LOGIC_VECTOR (7 DOWNTO 0);
b <= B"11111111";      -- Binary
b <= X"FF";            -- Hexadecimal
b <= O"377";           -- Octal
```

Signed and unsigned operands

- The result of some arithmetic operations with bit vectors can differ depending on the considered binary representation, signed or unsigned:



- Use different types of data:
  - UNSIGNED for arithmetic operations without sign
  - SIGNED for arithmetic operations with sign

Signed and unsigned operands

- Example:

```
USE IEEE.NUMERIC_STD.ALL;
...
SIGNAL u1, u2: UNSIGNED(3 DOWNTO 0);
SIGNAL s1, s2: SIGNED(3 DOWNTO 0);

-- signed
        u1 >u2
        UNSIGNED("1111") > UNSIGNED("0000")
-- unsigned
        s1 > s2
        SIGNED("1111") > SIGNED("0000")
```

Operand: Conversion functions

- Predefined functions allow the conversion between data types. They are included in the package NUMERIC_STD:
  - For conversion among vector data types (cast) :
    - **std_logic_vector (), unsigned(), signed()**
  - For conversion among integer and vectors
    - **to_integer**: from **unsigned** o **signed**
    - **to_unsigned**: from **integer**
    - **to_signed**: from **integer**

```
USE IEEE.NUMERIC_STD.ALL;
...
SIGNAL s : UNSIGNED (7 DOWNTO 0 );
SIGNAL i : INTEGER RANGE 0 TO 255;
...
i <= TO_INTEGER(s);
s <= TO_UNSIGNED(i, 8);
```

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Operators: predefined

Type of operation	Operator	Operand type	Result type
Logical	NOT, AND, OR, NAND, NOR, XOR	BOOLEAN, BIT, STD_LOGIC	BOOLEAN, BIT, STD_LOGIC
Relational	=, /=, <, <=, >, >=	Any type	BOOLEAN
Arithmetic	+, -, *, /, **, MOD, REM, ABS	INTEGER, REAL, Physical, UNSIGNED, SIGNED	INTEGER, REAL, Physical, UNSIGNED, SIGNED
Concatenation	&	ARRAY & ARRAY ARRAY & element	ARRAY

Operator: Concatenation and aggregates

- Concatenation is used to form vectors and to shift/rotate vectors

```
SIGNAL s1, s2 : STD_LOGIC_VECTOR ( 3 DOWNTO 0 );
SIGNAL x, z : STD_LOGIC_VECTOR ( 7 DOWNTO 0 );
...
z <= s1 & s2;
z <= x(6 DOWNTO 0) & '0';
```

- Aggregates are used to join elements in order to form data of a composite type

```
s1 <= ( '0', '1', '0', '0' );              -- Equivalent to s1 <= "0100";
s1 <= ( 2 => '1', OTHERS => '0');-- Equivalent to s1 <= "0010";
s1 <= ( 0 to 2 => '1', 3 => s2(0) );
S1 <= (others => '0');
```

Outline

- Introduction
- Hierarchy and basic design units
- Statements
- Data objects
- Data types
- Operands
- Operators
- Attributes

Attributes

- Provide information about values, ranges or types associated with VHDL objects and types
- They may be predefined or user-defined
- Predefined attributes
  - Array attributes: give the range, length or the limits of an array
  - Type attributes: used to access to the elements of a given type
  - Signal attributes: used to model some signal properties

Array related attributes

SIGNAL d: STD_LOGIC_VECTOR(15 DOWNTO 0)

Attribute	Description	Example	Result
‘LEFT	Left boundt	d’LEFT	15
‘RIGHT	Right bound	d’RIGHT	0
‘HIGH	Upper boundt	d’HIGH	15
‘LOW	Lower bound	d’LOW	0
‘RANGE	Range	d’RANGE	15 DOWNTO 0
‘REVERSE_RANGE	Reverse range	d’REVERSE_RANGE	0 TO 15
‘LENGTH	Length	d’LENGTH	16

Signal related attributes

Attribute	Type	Description
‘DELAYED(t)	Signal	It generates the same signal but delayed
‘STABLE(t)	Signal BOOLEAN	The result is *true* when the signal has not changed during time *t*
‘EVENT	Value BOOLEAN	The result is *true* when the signal has changed
‘LAST_EVENT	Value TIME	Spent time since the last signal event
‘LAST_VALUE	Value	The value of the signal before the last event
‘QUIET(t)	Signal BOOLEAN	The result is *true* if the signal has not received any transaction during time *t*
‘ACTIVE	Value BOOLEAN	The result is *true* when the signal has had a transaction
‘LAST_ACTIVE	Value TIME	Spent time since the last transaction
‘TRANSACTION	Signal BIT	It changes the value every time the signal receives a transaction