LANGUAGE VHDL FUNDAMENTALS

Authors: Luis Entrena Arrontes, Celia López, Mario García, Enrique San Millán, Marta Portela, Almudena Lindoso

History of VHDL

- ☞ **1980: The USA Department of Defense funded a project under the program VHSIC (Very High Speed Integrated Circuit) to create a standard hardware description language.**
- ☞ **1983: The development of VHDL began.**
- ☞ **1987: IEEE 1076-1987 standard**
- ☞ **1994: The standard is revised: IEEE 1076-1993**
- ☞ **1996: First automatic tools that support the new version of the standard**
    - ❑ **We will focus on the 1987 version since it is the most widely accepted standard.**
- ☞ **2002: New version with few modifications**

VHDL language

- ☞ **IEEE standard**
- ☞ **Widely used, mainly in Europe (Verilog in USA)**
- ☞ **Capabilities**
  - ❑ **VHDL is a very powerfu language. It can be used to design digital circuits at different abstract levels, from logic level to system level.**
  - ❑ **Model and simulate digital circuits at different abstract levels.**
  - ❑ **Logic synthesis, Register Transfer (RT) (automatic tools)**

General features

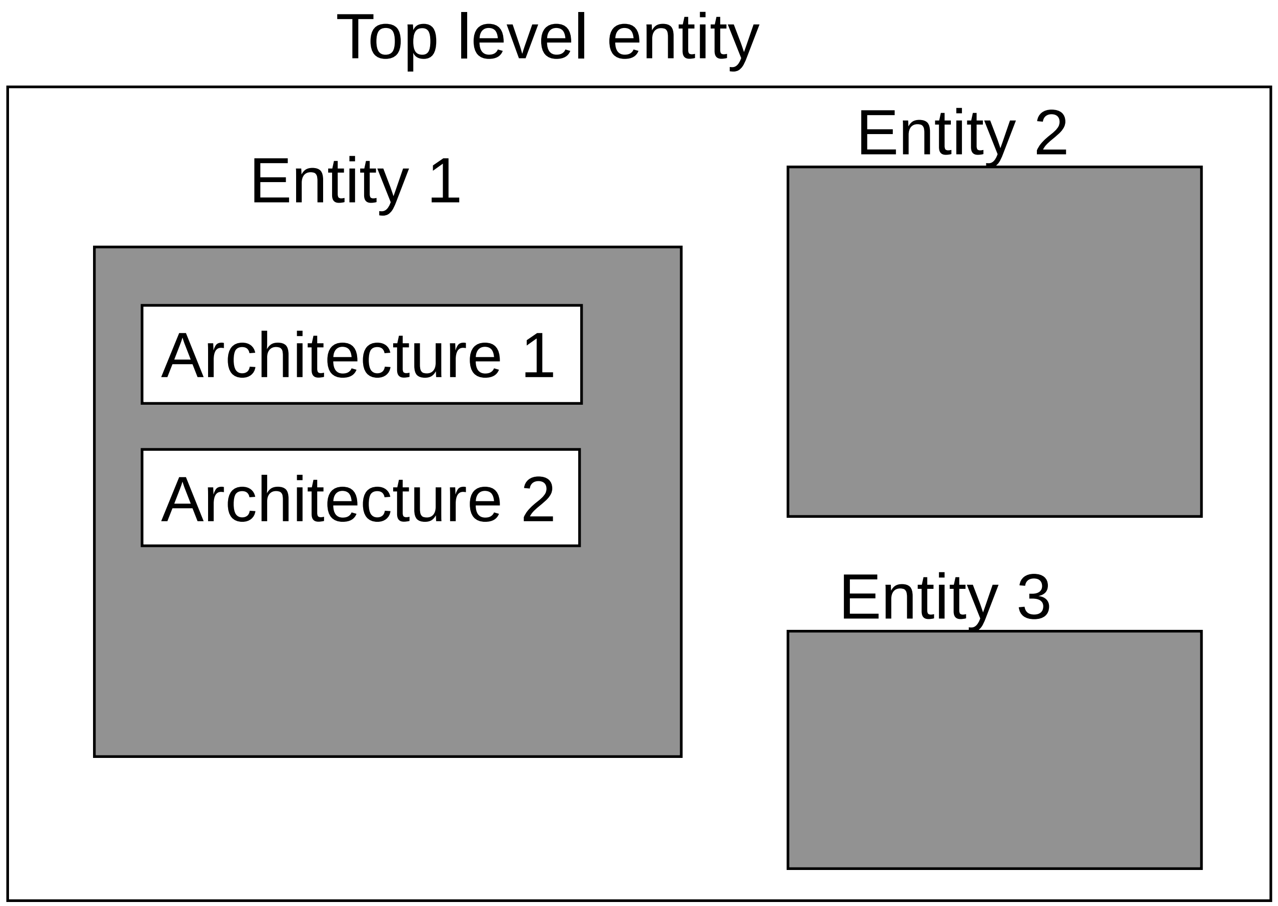
- Design hierarchy
- Allows the use of design libraries. Useful for managing large and multiple designs.
- Generic design.
- Execution in parallel (concurrence)
- Description style
  - Structural
  - Behavioral
- Simulation and synthesis
  - VHDL for synthesing hardware is just a subset of the VHDL for simulation.

Notation used

- ☞ Capital letters for VHDL keywords (reserved words), although VHDL is case insensitive.
- ☞ Brackets [ ] point out optional clauses in VHDL constructs.
- ☞ Vertical bar | point out alternative elements.
- ☞ Comments are pointed out with two dashes --, like in VHDL.
- ☞ VHDL code examples are in a rectangle ☐

LANGUAGE VHDL FUNDAMENTALS

Entities and architectures



Entities

☞ The entity is the basic design unit

☞ The entity declaration consists of

- ❑ Generic parameters declaration
- ❑ Interface declaration

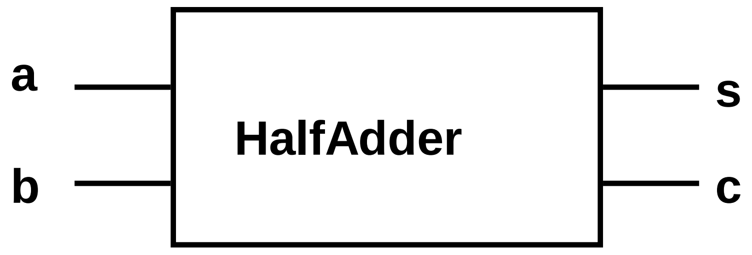
```
ENTITY not_cell IS
            GENERIC (delay_not: TIME := 5 NS);
            PORT (i1: IN BIT; o1: OUT BIT);
END not_cell;
```

Architectures

- The architecture is a secondary desing unit and it provides the functional description of a design.
- One entity can be matched with multiple architerctures that describe different versions for the same design.
- A VHDL architecture can instantatiate lower-level entities in which case they are known as component. This feature provides a hierarchical structure.

```
ARCHITECTURE example OF not_cell IS
            -- Declarations
BEGIN
            -- Sentences
            o1 <= NOT  i1 AFTER delay_not;
END example;
```

Example: a half-adder



☞ **Half-adder equations**

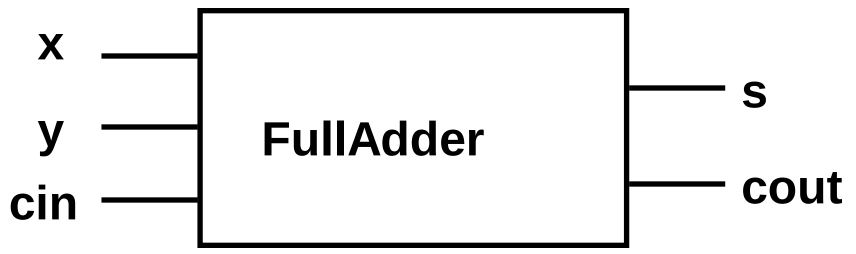
- ❑ s = a xor b
- ❑ c = a and b

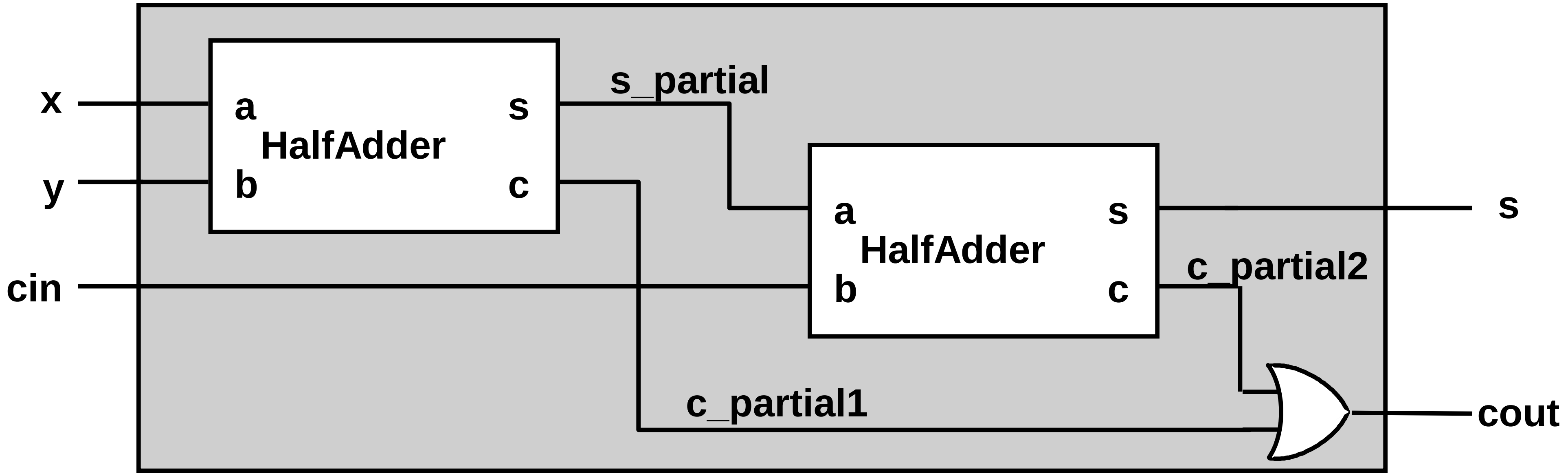
The half-adder in VHDL

```
ENTITY HalfAdder IS
        PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END HalfAdder;
```

```
ARCHITECTURE simple OF HalfAdder IS
BEGIN
        s <= a XOR b;
        c <= a AND b;
END simple;
```

Full-adder


x
y
cin
FullAdder
s
cout


x
y
cin
a
HalfAdder
b
s
c
s_partial
a
HalfAdder
b
s
c
c_partial2
c_partial1
s
cout

Full-adder in VHDL

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

ARCHITECTURE structural OF FullAdder IS
        COMPONENT HalfAdder
                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: HalfAdder PORT MAP (x, y, s_partial, c_partial1);
        SS1: HalfAdder PORT MAP (s_partial, cin, s, c_partial2);
        cout <= c_partial1 OR c_partial2;
END structural;
```

Port connection

- ☞ **Connecting ports by using positional assotiation**

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

          ......

SS0: HalfAdder PORT MAP (x, y, s_partial, c_partial1);
```

Port connection

☞ **Connecting ports by using named association**

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


               ......

SS0: HalfAdder PORT MAP (a => x, b => y , s => s_partial, c =>c_partial1);
```

LANGUAGE VHDL FUNDAMENTALS

Concurrent statements

- **These statements are executed in parallel!**
- **They operate independently and then they can be written in any order.**
- **The simulation tools detect when there is an event in the object values and it set when objects must be updated.**
- **Every statement in an architecture is concurrent with respect to the others.**

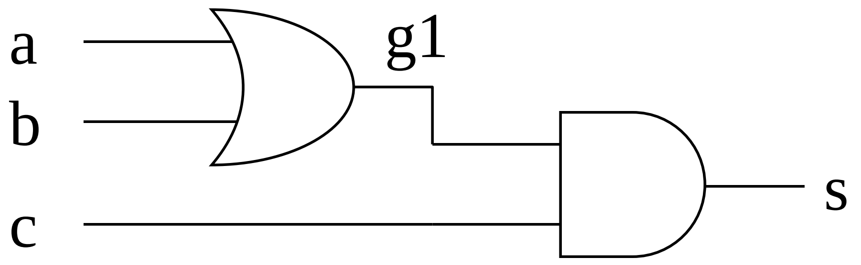
```
ARCHITECTURE structural OF FullAdder IS
...
BEGIN
        SS0: HalfAdder PORT MAP (x, y, s_partial, c_partial1);
        SS1: HalfAdder PORT MAP (s_partial, cin, s, c_partial2);
        cout <= c_partial1 OR c_patcial2;
END structural;
```

Concurrent statements

☞ Two equivalent examples

```
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;
```



Simulation of concurrent statements

```
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;
```

☞ **Simulation:**

- ❑ If there is a change in a or b -> Calculating the new value of g1
- ❑ If there is a change in g1 or c -> Calculation the new value of s
- ❑ Each statement is executed so many times as it is necessary
- ❑ The order in which concurrent statements are written is irrelevant!

Sequential statements

- **Concurrency can be difficult to manage by designers:**
  - **VHDL also allows the use of sequential statements to describe designs**
- **Sequential statements:**
  - **They are executed following a procedural flow, like in software languages.**
  - **Sequential statements are always included inside processes or procedures**
  - **Between the execution of two different sequential statements time does not pass.**

Processes

- Processes are used to describe hardware by means of sequential statements
- They consist of
  - Declarations
  - Sequential statements
- The process must have a sensitivity list or at least a WAIT clause.
- A process is activated when
  - A change (event) occurs in one or more signals contained in the sensitivity list
  - The condition in the WAIT statement becomes true
- Proceses are executed indefinitely over and over.

Example of process(I)

```
ARCHITECTURE one OF example IS
BEGIN
        PROCESS ( i1 )
                VARIABLE a: BIT;
        BEGIN
                a := NOT i1;        -- Sequential statement
                o1 <=  a;           -- Sequential statement
        END PROCESS;
END example;
```

Sensitivity list

☞ The process is activated when i1 changes the value

☞ The statements inside the process are executed sequentially

Example of process(II)

```
ARCHITECTURE two OF example IS
BEGIN
        PROCESS
                VARIABLE a: BIT;
        BEGIN
                a := NOT i1;    -- Sequential statement
                o1 <= a;        -- Sequential statement
                WAIT ON i1;     -- Sequential statement
        END PROCESS;
END example;
```

- Process execution is stopped in the WAIT statement
- The execution is resumed when there is a change in the value of i1
- When the end of the process is reached the execution starts again from the beginning.

Recommendations for synthesis

- **Do not use AFTER clauses in descriptions for synthesis**
  - ❑ **Synthesizer ignore the clause AFTER in the assignments, since the delay is a feature of the technology used to implement the circuit. This information is fixed at low abstraction levels (physical levels), while VHDL is used at high abstraction levels (mainly RT)**
- **Do use process with sensitivity list**
  - ❑ **WAIT statement is just synthesizable in very few cases**
  - ❑ **WAIT statement is not supported by many synthesizers**

LANGUAGE VHDL FUNDAMENTALS

VHDL Data Objects

- ☞Constants
- ☞Variables
- ☞Signals

Similar to constant and variables in software languages

Represent hardware signals. Their values change along the execution

Data Objects: Constants

- Hold one specific value that can not change
- A constant can be declared in any part of a design

```
CONSTANT name1, name2, ..., namen: type [ := value ];
```

- Examples

```
CONSTANT gnd : BIT := ‘0’;
CONSTANT n, m : INTEGER := 3;
CONSTANT delay: TIME := 10 NS;
```

Data 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 fields, 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 not global variables.**

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

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

Data objects: Signals

- **Their values have always associated a temporal factor**
- **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 fields. However they are global objects and are also visible in sequential environments, like processes or subprograms**

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

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

Signal simulation: Drivers

- Each process that contains a signal assignment contains a *driver* for that signal
- The driver for a given signal stores a list of transactions to represent the current and future signal values
- The first transaction in a driver is the current value of that driver
- The simulator is in charge of updating the values in every signal along the execution time: when the time in a transaction is equal to the current instant the signal is updated with the driver's value.

Signal assignments

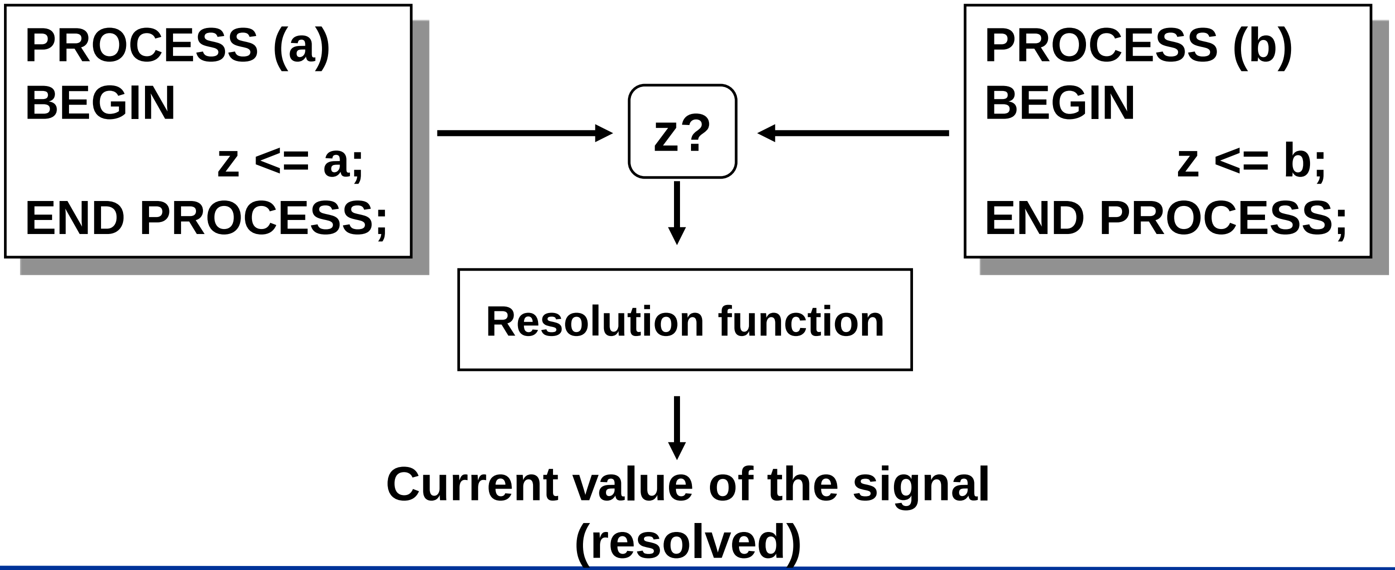
☞ If a process contain several assignments for the same signal, there is only one driver and then the final value for the signal will be the last one assigned

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

z always is '1'

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 is obtained by means of a *resolution function*



Using signals and variables

☞ **Variables:**

- ❑ **They are locals within the processes and subprograms. They are used for storing local values or temporary values**
- ❑ **They require less memory space and their use provides more efficiency at simulation, since they are updated immediately.**

☞ **Signals**

- ❑ **They are necessary for implementing concurrency. Using signals is the only way to communicate two different processes.**

☞ **Asignments**

❑ **For variables**	**:=**
❑ **For signals**	**<=**

LANGUAGE VHDL FUNDAMENTALS

Outline

- ☞ **Data types**
- ☞ **Operators and conversion functions**
- ☞ **Attributes**
- ☞ **Data types in synthesis**

Data types

- VHDL is a language with a wide set of data types
- Every data type limits the number of possible values that an object associated with that type can take.
- Types
  - Scalars:

- Enumerated	- Integer
- Real	- Physical

  - Composite

- Vector/Matrix	- Record

  - File

Scalars types

- **Enumerated**
  - **Values are identifiers or a character literal**
  - **Enumerated data types can be defined by the language of user defined**
- **Integer**
- **Real**
- **Physical**
  - **A numerical value and a physical unit**
  - **The predefined type TIME is the only physical type that we are going to use.**

Enumerated data types defined by the user

☞ **Type declaration**

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

☞ **Use**

```
CONSTANT first_letter: set_of_letters := ‘A’;
SIGNAL traffic_light1: traffic_lights ;
SIGNAL current_state: states ;
...
current_state <= s0;
Traffic_light1<= green;
```

Predefined enumerated types

❑ **BIT**	**('0', '1')**
❑ **BOOLEAN**	**(FALSE, TRUE)**
❑ **CHARACTER**	**(NUL, SOH, ..., 'A', 'B', ...)**

❑ **STD_LOGIC (defined in the IEEE 1164 standard)**
('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')

Standard logic type

- **This type is defined in the IEEE 1164 standard. It is use widely**
- **It is a enumerated typed for logic object. It provided multiple values**

❑ ‘U’	- uninitialized. This is the value of an object by default
❑ ‘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

Standard logic type

- **STD_LOGIC is a resolved tipo resuelto**
  - ❑ **A STD_LOGIC object may have got multiple drivers**
  - ❑ **The resolution function assign an ‘X’ when there is a collision between multiple drivers**
- **In order to use standard logic types we have to added the following lines before the design entity declaration:**

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

Integer type

☞ **Declaration and use**

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

☞ **It is advisable to fix a range**

- ❑ **It limits the number of bits inferred in the synthesis**
- ❑ **Simulation sends an error messages when the object has a value out of range**

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

Real type

☞ **Declaration and use**

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

☞ **It is not synthesizable**

- ❑ **However, there are libraries for operations with real numbers (simulation)**

Physical types

- **Example of declaration and use**

```
TYPE weight IS RANGE 0 TO 100.000.000
          UNITS
                    gram;
                    kilo = 1000 gram;
                    ton = 1000 kilo;
          END UNITS;
SIGNAL p: weight;
```

- **The most important physical type predefined in the VHDL standard is the 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	hour = 60 minutes

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

Composite types: ARRAY

- **All the elements are of the same type. All the VHDL types can be used to form arrays**

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

- **It is possible to access one element or 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):**
  - **The accessed subset of elements must have the same order as in the original vector.**

Composite types: ARRAY

☞ **Unconstrained vectors. The size of the vector is specified in a later declaration**

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

☞ **Vectors of data of types BIT and STD_LOGIC are predefined**

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

Multidimensional arrays

☞ Multidimensional matrix and vector of vectors

```
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;
```

☞ Use

```
image(0,0) <= ‘1’;
ram1(0)(0) <= ‘1’;
ram1(0) <= “00000000”;
image(0, 0 TO 7) <= “00000000”;  -- ERROR
```

Composite types: RECORD

☞ Collection of elements possibly of different types

```
TYPE type_opcode IS (sta, lda, add, sub, and, nop, jmp, jsr);
TYPE type_operand IS BIT_VECTOR (15 DOWNTO 0);
TYPE instruction_format IS RECORD
            opcode   : type_opcode;
            operand: type_operand;
END RECORD;


SIGNAL ins1: instruction_format := (nop, “0000000000000000”);


ins1.opcode <= lda;
ins1.operand <= “0011111100000000”;
```

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**

```
"Esto es un mensaje"
"00110010"
```

- **Bit string literals: A prefix points out the base required**

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

Aliases and subtypes

☞ Alias are used to reference an object by a new name

```
SIGNAL status_register: BIT_VECTOR (7 DOWNTO 0);
ALIAS carry_bit: BIT IS status_register(0);
ALIAS zero_bit: BIT IS status_register(1);
```

☞ Subtypes define a subset of a type previously defined

```
SUBTYPE ten_values IS INTEGER RANGE 0 TO 9;
```

Predefined operators

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, STD_LOGIC_VECTOR	INTEGER, REAL, physical, STD_LOGIC_VECTOR
Concatenation	&	ARRAY & ARRAY ARRAY & element	ARRAY

MOD and REM

☞ **Reminder of an integer division:**

- ❑ **Using REM, the sign of the result is equal to the first operand.**
- ❑ **Using MOD, the sign of the result is equal to the second operand.**

```
5 rem 3 = 2
5 mod 3 = 2
(–5) rem 3 = –2
(–5) mod 3 = 1
(–5) rem (–3) = –2
(–5) mod (–3) = –2
5 rem (–3) = 2
5 mod (–3) = –1
```

Signed and unsigned operands

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

$$\begin{tikzcd} & \text{TRUE without sign (15 > 0)} \\ \boxed{\text{"1111" > "0000"}} \arrow[ur] \arrow[dr] & \\ & \text{FALSE with sign (-1 < 0)} \end{tikzcd}$$

- There are two possible solutions:
  - Using two different set of operators:
    - ✓ STD_LOGIC_UNSIGNED for operations without sign
    - ✓ STD_LOGIC_SIGNED for operations without sign
  - Using two different data types:
    - ✓ UNSIGNED for operations without sign
    - ✓ SIGNED for operations without sign

Signed and unsigned operands

☞ **Example of the first possible solution:**

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
...
"1111" > "0000" -- without sign
```

```
USE IEEE.STD_LOGIC_SIGNED.ALL;
...
"1111" > "0000" -- with sign
```

☞ **The packages STD_LOGIC_UNSIGNED and STD_LOGIC_SIGNED define the same operations but taking into account different binary representation. Therefore, they cannot be used together in a same design unit.**

Signed and unsigned operands

☞ Example of the second possible solution:

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

-- without sign
            u1 >u2
           UNSIGNED("1111") > UNSIGNED("0000")
-- with sign
           s1 > s2
           SIGNED("1111") > SIGNED("0000")
```

Nowadays:
USE IEEE.NUMERIC_STD.ALL;

Conversion functions

☞ **Predefined function allows the conversion between data types**

- ❑ **CONV_INTEGER**
- ❑ **CONV_STD_LOGIC_VECTOR**

☞ **These functions are defined in the package STD_LOGIC_ARITH**

```
USE IEEE.STD_LOGIC_ARITH.ALL;
...
SIGNAL s : STD_LOGIC_VECTOR ( 7 DOWNTO 0 );
SIGNAL i : INTEGER RANGE 0 TO 255;
...
i <= CONV_INTEGER(s);
s <= CONV_STD_LOGIC_VECTOR(i, 8)
```

Concatenation and aggregates

- Concatenation is used to form vectors

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

- Aggregates are used to join elemnts in order to form data of a composite type (ARRAY o RECORD)

```
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) );
```

Attributes

- Pointing out values, functions, ranges or types associated with VHDL objects and types.
- They may be predefined or user-defined
- Predefined attributes
  - Array related: Are used to determine the range, length or the limits of an array
  - Type related: These attributes are used to access to the elements of a given type
  - Signal related: These attributes ares used to model some signal properties

Array related attributes

```
SIGNAL d: STD_LOGIC_VECTOR(15 DOWNTO 0)
```

Attribute	Description	Example	Result
‘LEFT	Left bound	d’LEFT	15
‘RIGHT	Right bound	d’RIGHT	0
‘HIGH	Upper bound	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

Type related attributes

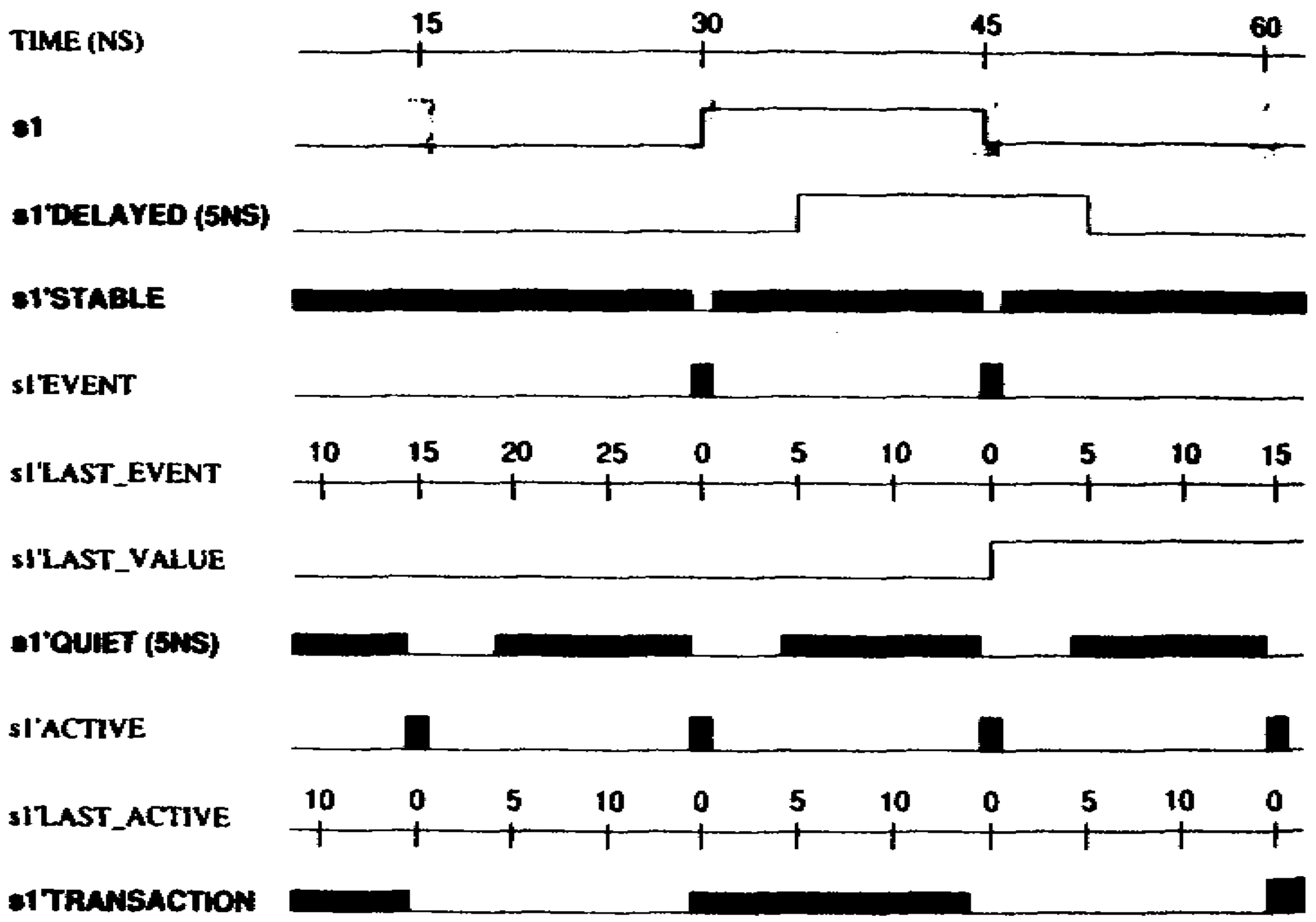
```
TYPE qit IS (‘0’, ‘1’, ‘Z’, ‘X’);
SUBTYPE tit IS qit RANGE ‘0’ TO ‘Z’;
```

Attribute	Description	Example	Result
‘BASE	Base of the type	tit’BASE	qit
‘LEFT	Left bound (subtype)	tit’LEFT	‘0’
‘RIGHT	Right bound (subtype)	tit’RIGHT	‘Z’
‘HIGH	Upper bound (subtype)	tit’HIGH	‘Z’
‘LOW	Lower bound (subtype)	tit’LOW	‘0’
‘POS(v)	Position of v (base type)	tit’POS(‘X’)	3
‘VAL(p)	Value in the p index (base type)	tit’VAL(3)	‘X’
‘SUCC(v)	Next value to v (base type)	tit’SUCC(‘Z’)	‘X’
‘PRED(v)	Previous value to (base type)	tit’PRED(‘1’)	‘0’
‘LEFTOF(v)	Value on the left of v (base type)	tit’LEFTOF(‘1’)	‘0’
‘RIGHTOF(v)	Value on the right of v (base type)	tit’RIGHTOF(‘1’)	‘Z’

Signal related attributes

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

Signal related attributes



User-defined attributes

- Attributes of entities, architectures, types and objects can be defined
- Firstly, we must to declare the attribute type

```
ATTRIBUTE <attribute_name> : <type>;
```

- Secondly, we must to specify its value

```
ATTRIBUTE < attribute_name > OF <item> : <class> IS <value> ;
```

- Example

Circuit is an entity

```
ATTRIBUTE technology: STRING;
ATTRIBUTE technology OF circuit: ENTITY IS “CMOS”;
...
              ... circuit’technology ...
```

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 effecient synthesis**

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

- **Enumerated:**
  - **They are synthesized as a number coded in natural binary. A binary code is assigned to every value in the appearence order**

Interpretation of data types in synthesis

- **The following types are not synthesizable**
  - ❑ **REAL**
  - ❑ **Physical**
  - ❑ **Files**
- **Using attributes**
  - ❑ **Array related attributes: They are useful and syntesizable**
  - ❑ **Types related attributes: They may be synthesizable although they are not ussualy used**
  - ❑ **Signal related attributes: EVENT is the most used. The rest of these kind of attributes are not synthesizable, except STABLE**

Initial values

☞ **Simulation:**

- ❑ **In order to simulate a signal, all the signal and variables must have an initial value. The initial value can be assigned in the object's declaration**
- ❑ **If the initial value is not specified, the VHDL language assigns a value by default. In enumerated types this value is the first one at the left**

```
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'
```

Initial values

☞ **Synthesis**

- ❑ **Synthesizers have not taking into account initial values**
- ❑ **The inizialization has to be made in hardware**

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