



**VietNam National University  
University of Engineering and Technology**

# **Modelling Hardware with VHDL**

TS. Nguyễn Kiêm Hùng  
Email: [kiemhung@vnu.edu.vn](mailto:kiemhung@vnu.edu.vn)

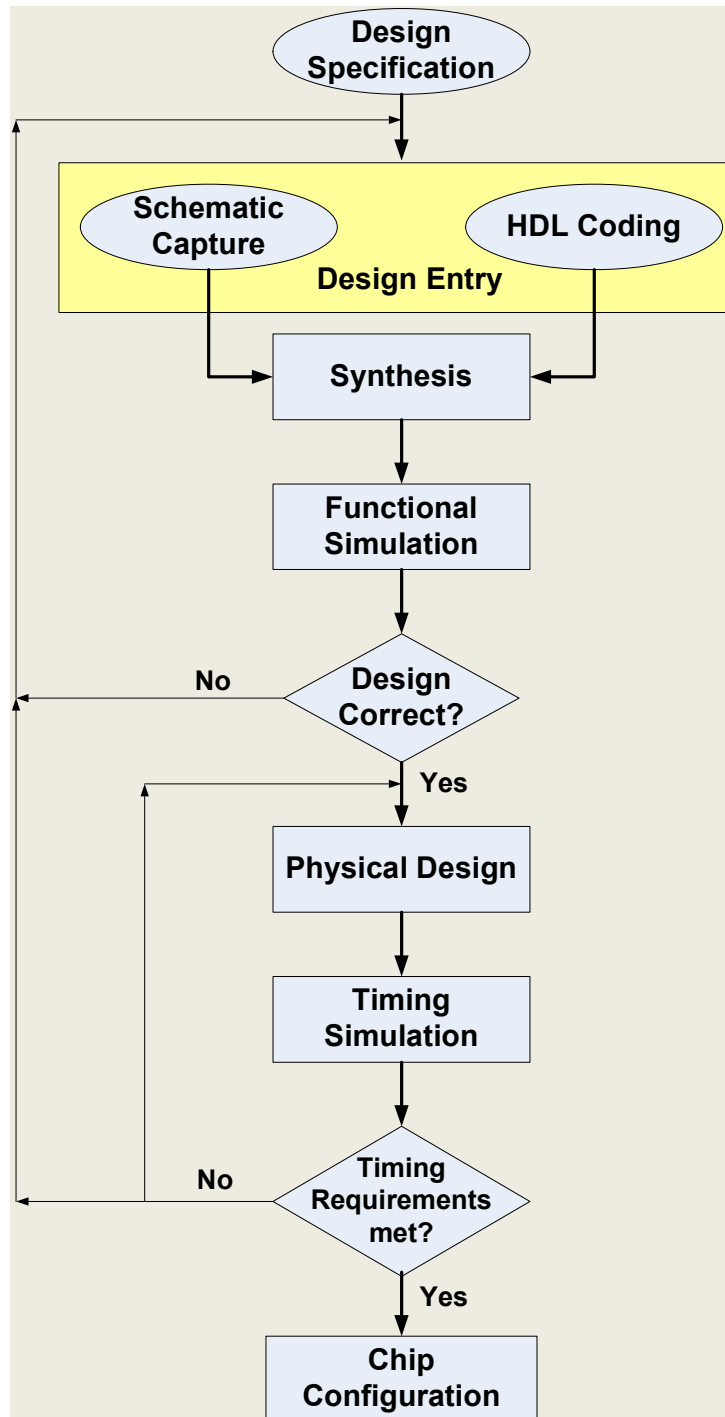
**Laboratory for Smart Integrated Systems**

# Objectives

## **In this lecture you will be introduced to:**

- Key concepts behind hardware description languages
  - What sets VHDL apart from a programming language
  - The most important VHDL language constructs
- Putting VHDL to service for hardware synthesis
  - VHDL subset for Synthesis
  - Patterns for registers and Finite state machines
  - How to establish a Register transfer level model
- Putting VHDL to service for hardware simulation
  - Writing a Test Bench
- Using EDA tools for circuit and synthesis such as ModelSim, Xilinx Vivado Design Suite, etc.

# Review:



# Hardware Description Languages (HDL)

- **Schematic-based Design:**
  - **Suitable for small design size (in the order of few thousand gates)**
    - High circuit entry time
    - The readability suffers while processing through hundreds of drawing sheets,
    - Difficult circuit optimization, etc
- **HDL-based Design:**
  - HDL is used to **describe hardware** rather than a program to be executed on a computer.
  - used in order to speed up design cycle times: **at least five times**
    - very concise representation of circuits
    - methods of optimization in HDL is easy to implement automatically
    - portable from one vendor platform to another
    - **Technology independent:** feature size has no effect on these HDL designs
    - Reusable
  - **Two most popular HDLs endorsed as IEEE standard: Verilog and VHDL**

# Brief History of VHDL

# VHDL

- VHDL is a language for describing digital hardware used by industry worldwide
  - **VHDL** is an acronym for **V**HSIC (**V**ery **H**igh **S**peed Integrated **C**ircuit) **H**ardware **D**escription **L**anguage

# Genesis of VHDL

## State of art circa 1980

- Multiple design entry methods and hardware description languages in use
- No or limited portability of designs between CAD tools from different vendors
- Objective: shortening the time from a design concept to implementation from *18 months to 6 months*

# A Brief History of VHDL

- **June 1981: Woods Hole Workshop**
- **July 1983: contract awarded to develop VHDL**
  - Intermetrics
  - IBM
  - Texas Instruments
- **August 1985: VHDL Version 7.2 released**
- **December 1987:**

VHDL became *IEEE Standard 1076-1987* and in 1988 an ANSI standard
- **In 1993 revised to IEEE 1164 and updated in 2000 and 2002.**



# Three Versions of VHDL

- VHDL-87 (IEEE 1076 )
- VHDL-93 (IEEE 1164)
- VHDL-01

# Verilog

# Verilog

- Essentially identical in function to VHDL
  - No *generate* statement
- Simpler and syntactically different
  - C-like
- Gateway Design Automation Co., 1983
- Early *de facto* standard for ASIC programming
- Open Verilog International standard
- Programming language interface to allow connection to non-Verilog code

# VHDL vs. Verilog

Government Developed	Commercially Developed
Ada based	C based
Strongly Type Cast	Mildly Type Cast
Difficult to learn	Easier to Learn
More Powerful	Less Powerful

# Examples

- **VHDL Example:**

```
process (clk, rstn)
begin
    if (rstn = '0') then
        q <= '0';
    elseif (clk'event and clk = '1') then
        q <= a + b;
    end if;
end process;
```

- **Verilog Example:**

```
always@(posedge clk or negedge rstn)
begin
    if (! rstn)
        q <= 1'b0;
    else
        q <= a + b;
end
```

# VHDL for Synthesis

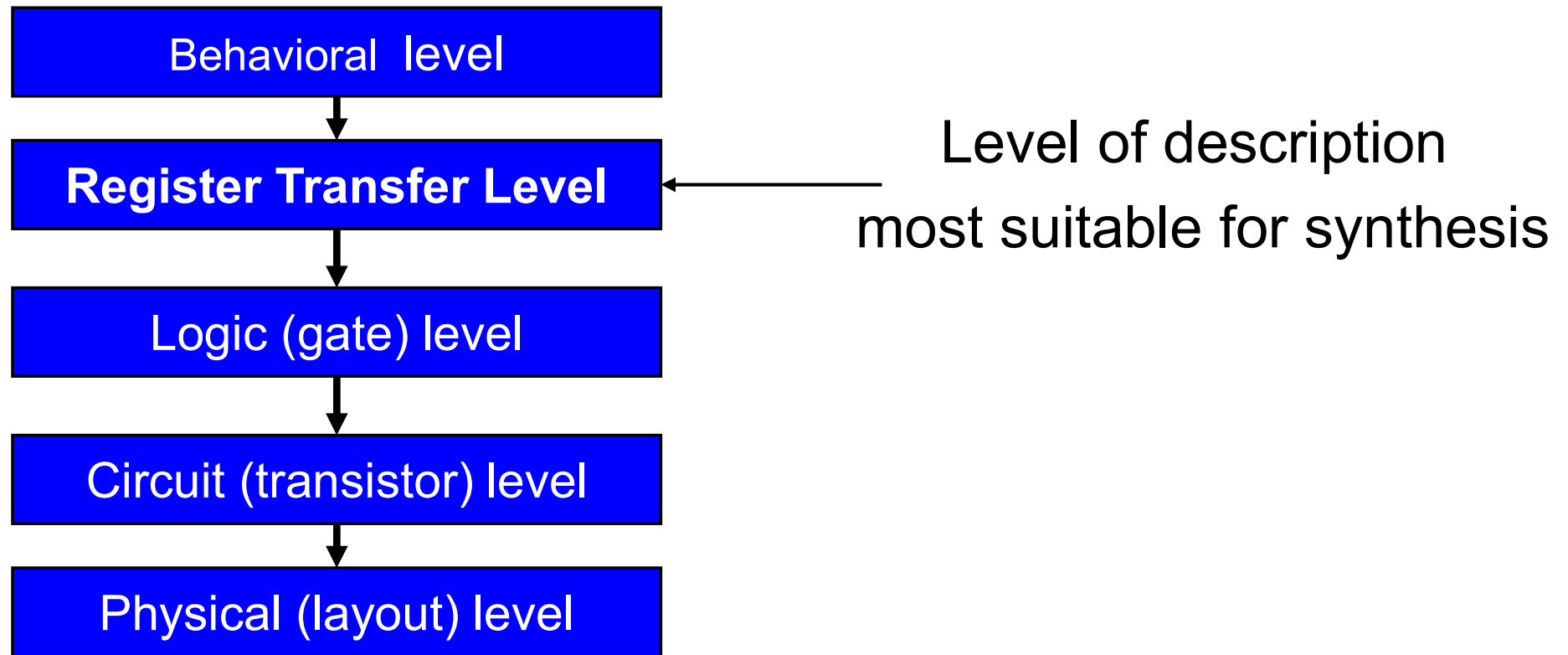
# Purposes of VHDL

**VHDL for Specification**

**VHDL for Simulation**

**VHDL for Synthesis**

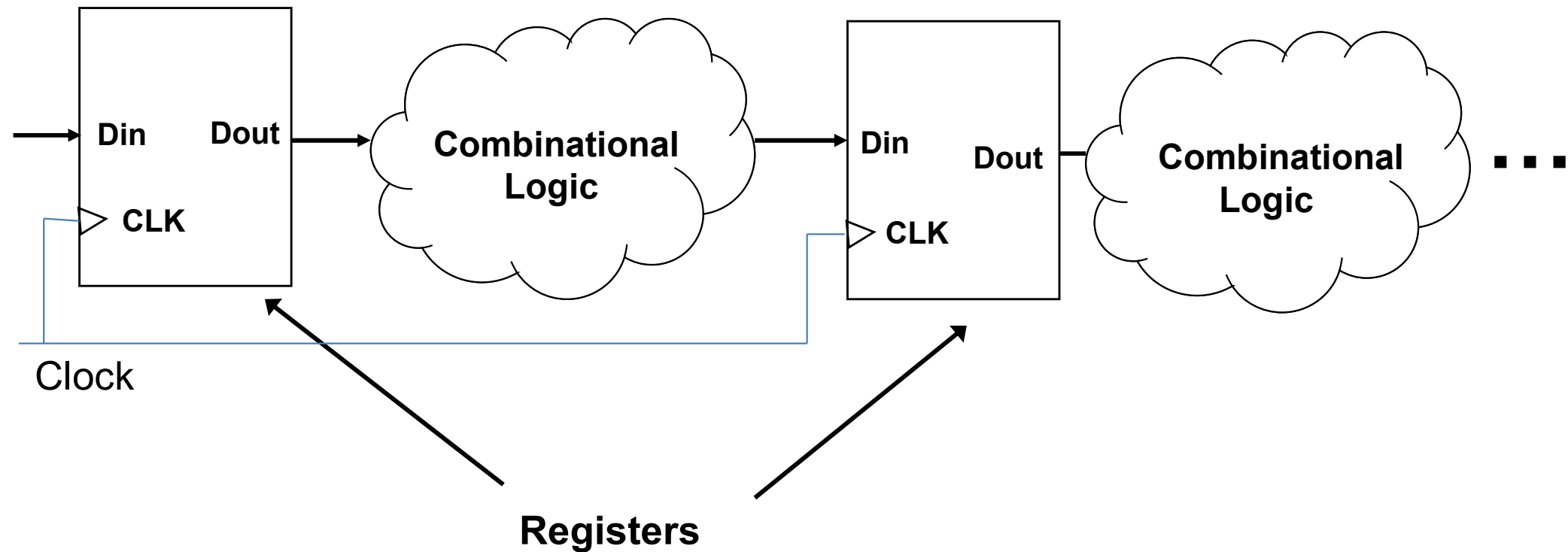
# Levels of design description





# RTL Design Description

## RTL: Register Transfer Level



# VHDL Fundamentals

# Naming and Labeling (1)

- VHDL is not case sensitive

Example:

Names or labels

**databus**

**Databus**

**DataBus**

**DATABUS**

are all equivalent

# Naming and Labeling (2)

## General rules of thumb (according to VHDL-87)

1. All names should start with an alphabet character (a-z or A-Z)
2. Use only alphabet characters (a-z or A-Z) digits (0-9) and underscore (\_)
3. Do not use any punctuation or reserved characters within a name (!, ?, ., &, +, -, etc.)
4. Do not use two or more consecutive underscore characters (\_\_) within a name (e.g., Sel\_\_A is invalid)
5. All names and labels in a given entity and architecture must be unique

# Free Format

- VHDL is a “free format” language

No formatting conventions, such as spacing or indentation imposed by VHDL compilers. Space and carriage return treated the same way.

Example:

```
if (a=b) then
```

*or*

```
if      (a=b)          then
```

*or*

```
if (a =
```

```
b) then
```

are all equivalent

# Comments

- Comments in VHDL are indicated with a “double dash”, i.e., “--”
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:

```
-- main subcircuit
```

```
Data_in <= Data_bus;    -- reading data from the input FIFO
```

# Comments

- Explain Function of Module to Other Designers
- Explanatory, Not Just Restatement of Code
- Locate Close to Code Described
  - Put near executable code, not just in a header

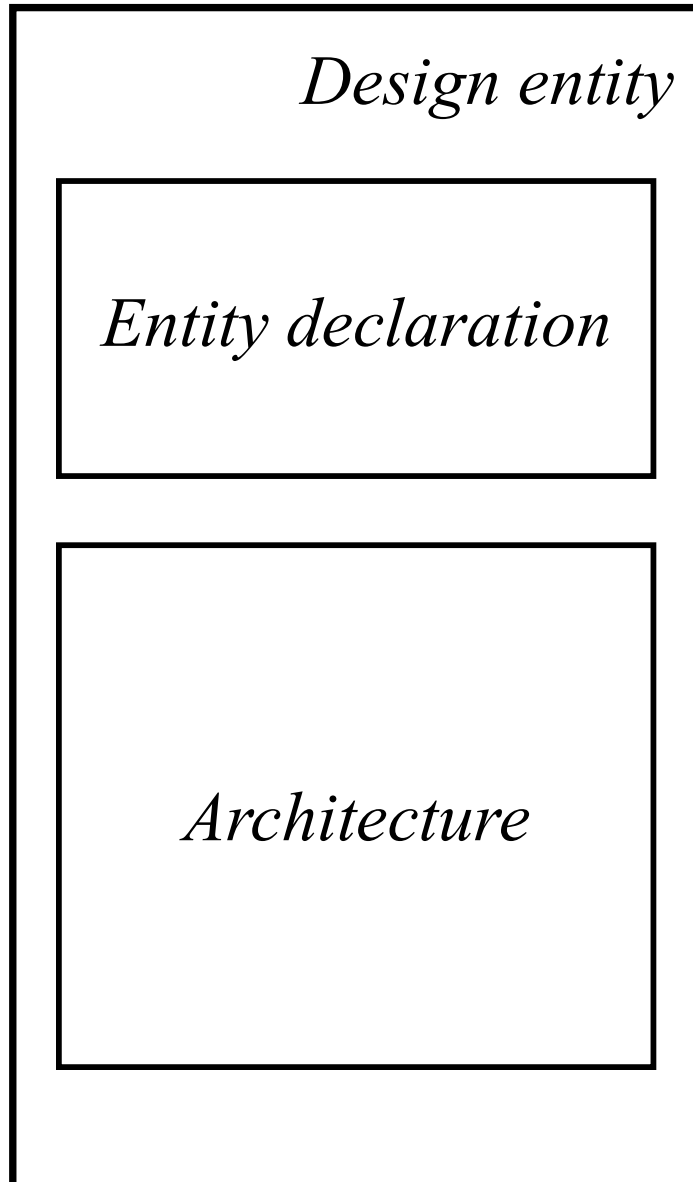
# Data Objects

- VHDL is a strongly **typed** language
  - has always to declare the type of every **object** that can have a value, such as **signals, constants and variables**.



# Design Entity

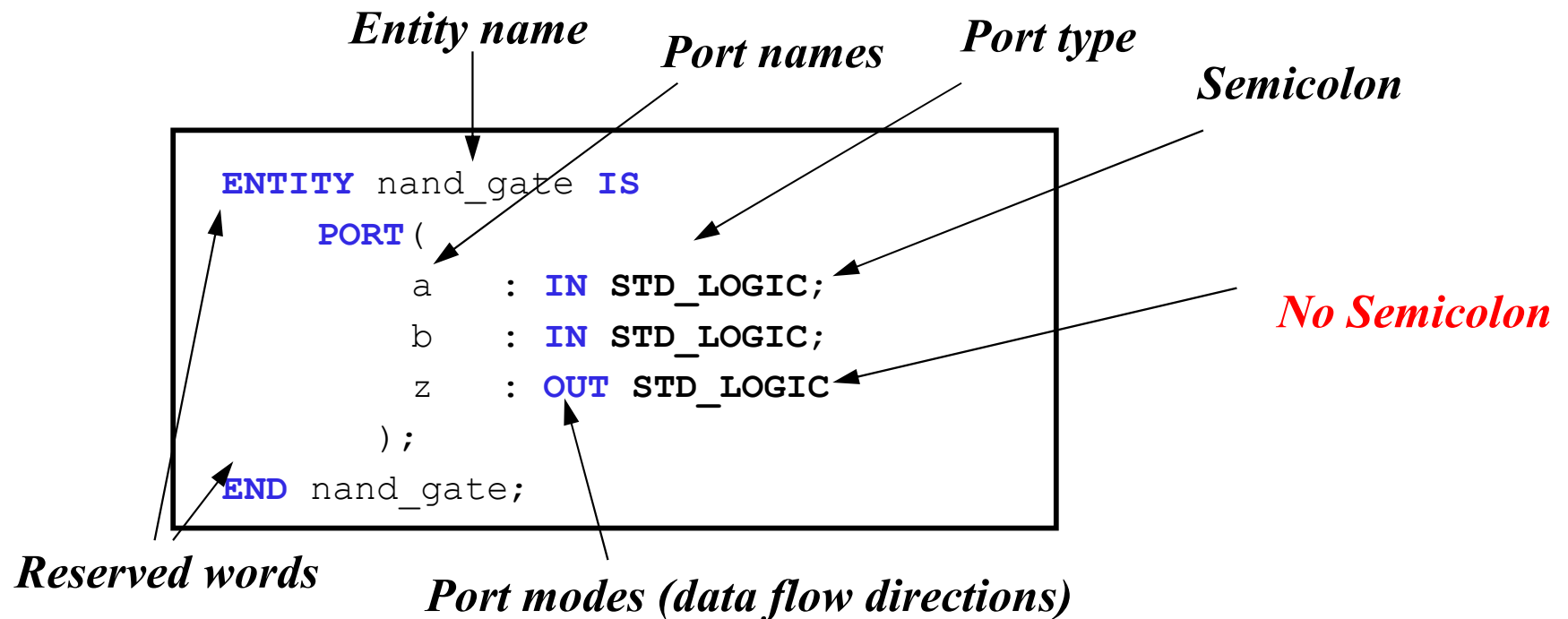
# Design Entity



- ✓ *Design Entity* - most basic building block of a design.
- ✓ A design entity that can contain other entities that are then considered components of the top-level entity

# *Entity Declaration*

- *Entity Declaration* describes the interface of the component, i.e. *input* and *output* ports.



# *Entity* declaration – simplified syntax

```
ENTITY entity_name IS  
  [ GENERIC generic_declarations;]  
  PORT (  
    port_name : signal_mode signal_type;  
    port_name : signal_mode signal_type;  
    .....  
    port_name : signal_mode signal_type);  
END [entity_name];
```

# *Generic* declaration

- Determine the local constants used for timing and sizing (e.g. bus widths) the entity.
- Can have a default value:

```
generic (  
    constant_name: type [:=value] ;  
    constant_name: type [:=value] ;  
    ...  
    constant_name: type [:=value] ) ;
```

# Architecture

- Describes how the circuit operates and how it is implemented

```
-- comments: An example of the architecture of 2-input Nand Gate
```

```
ARCHITECTURE model OF nand_gate IS
```

```
BEGIN
```

```
    z <= a NAND b;
```

```
END model;
```

Signal assignment operator assigns the value of the expression on the right to the signal on the left

# Architecture – simplified syntax

```
ARCHITECTURE architecture_name OF entity_name IS  
    -- components declarations ]  
    -- signals declarations ]  
    -- constants declarations ]  
    -- types declarations ]  
BEGIN  
    --Statements are here  
END architecture_name;
```

# Entity Declaration & Architecture

Complete VHDL file for describe NAND gate: nand\_gate.vhd

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
  
ENTITY nand_gate IS  
    PORT (  
        a      : IN STD_LOGIC;  
        b      : IN STD_LOGIC;  
        z      : OUT STD_LOGIC);  
END nand_gate;  
  
ARCHITECTURE model OF nand_gate IS  
BEGIN  
    z <= a NAND b;  
END model;
```



# QUIZ: *Concurrency?*

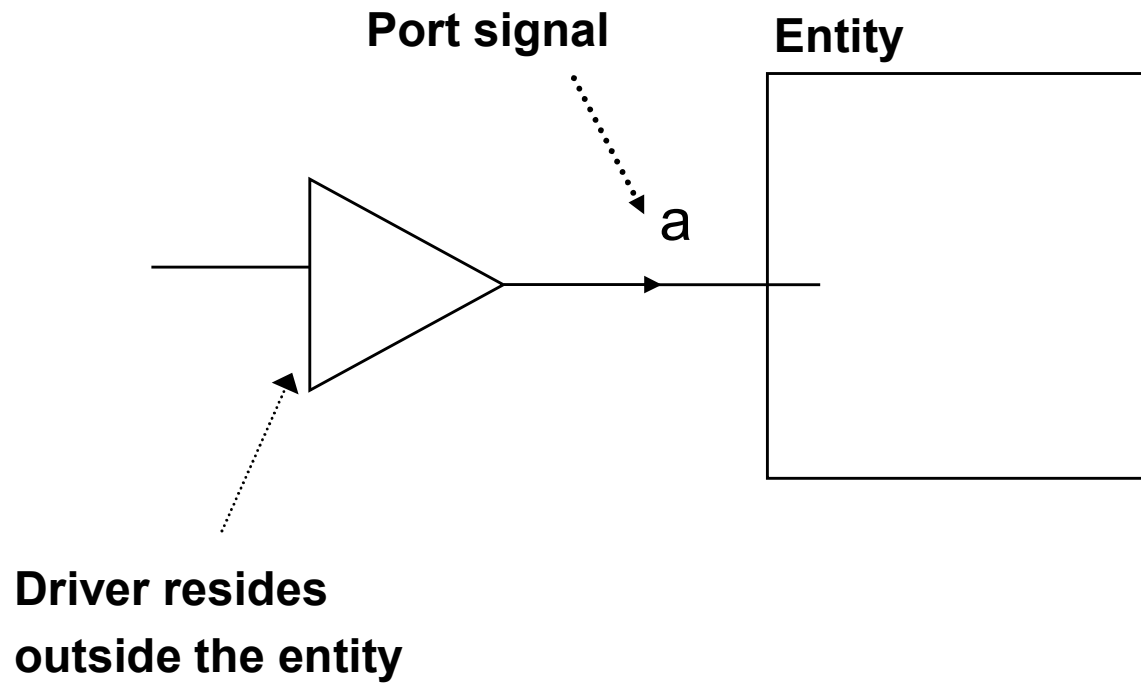
**nand\_gate.vhd:** various implementations of architecture body

```
ARCHITECTURE model OF nand_gate IS  
    -- signal declaration (of internal signals X)  
    signal X: std_logic;  
BEGIN  
    x <= a AND b;  
    z <= NOT X;  
END model;
```

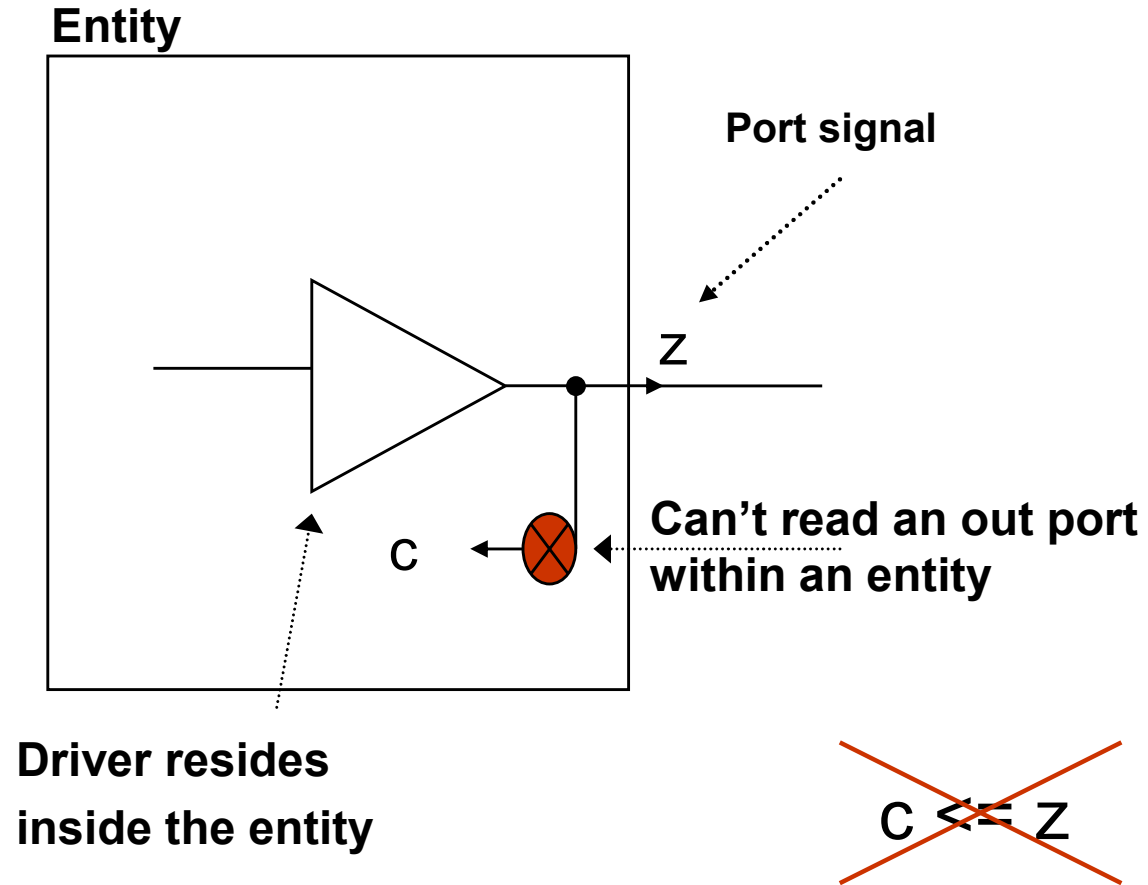
**Vs.**

```
ARCHITECTURE model OF nand_gate IS  
    -- signal declaration (of internal signals X)  
    signal X: std_logic;  
BEGIN  
    z <= NOT x;  
    x <= a AND b;  
END model;
```

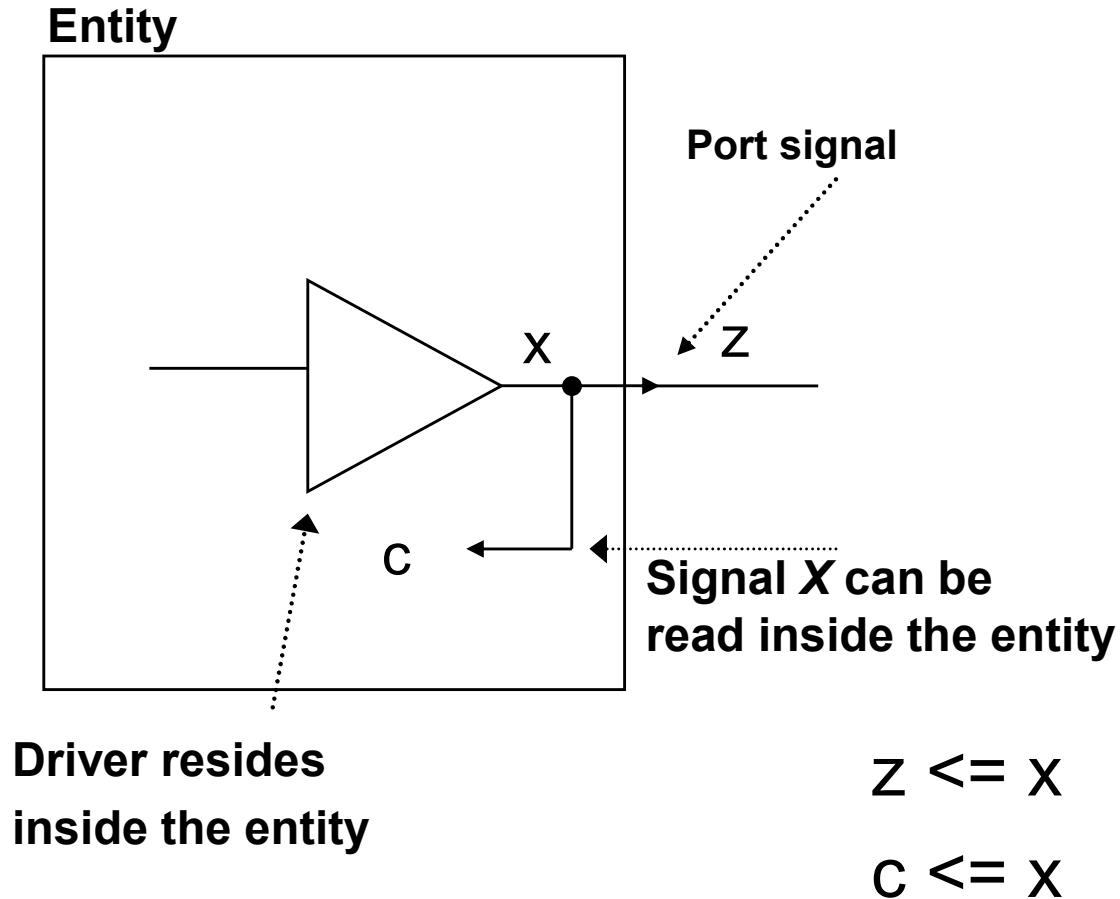
# Mode *In*



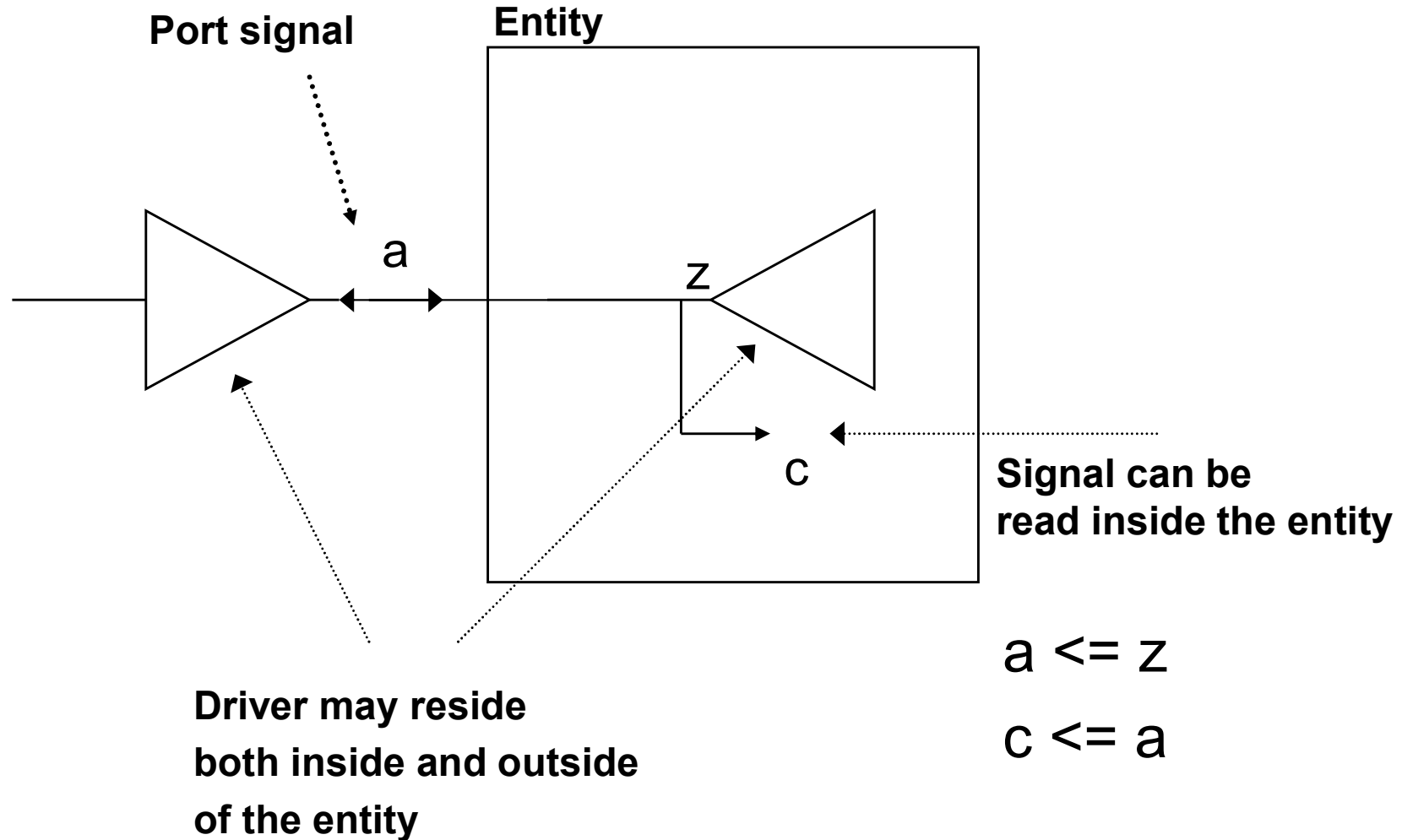
# Mode out



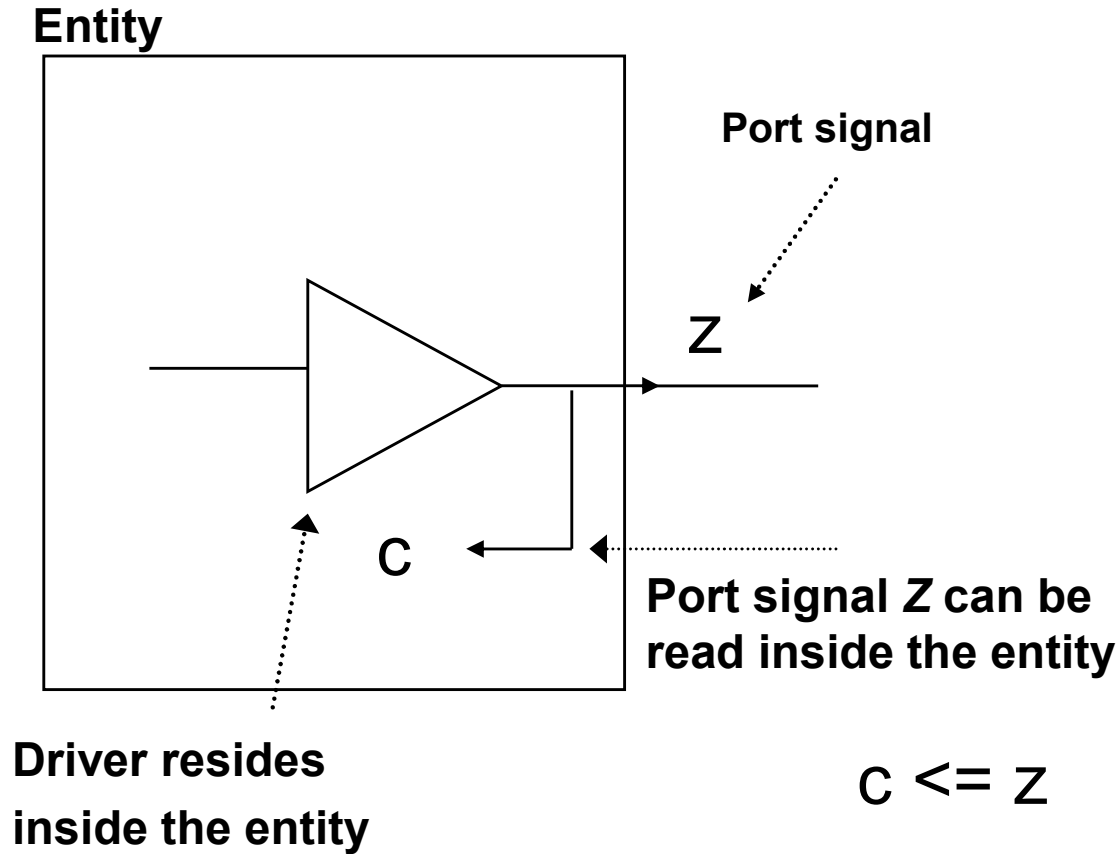
# Mode *out* with *signal*



# Mode *inout*



# Mode *buffer*



# Port Modes

The *Port Mode* of the interface describes the direction in which data travels with respect to the *component*

- **IN:** Data comes in this port and can only be read within the entity. It can appear **only on the right side** of a signal or variable assignment.
- **OUT:** The value of an output port can only be updated within the entity. **It cannot be read.** It can only appear **on the left side** of a signal assignment.
- **INOUT:** The value of a bi-directional port can be read and updated within the entity model. It can appear on **both sides** of a signal assignment.
- **BUFFER:** Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement the signal can appear on the left and right sides of the <= operator

# Libraries



# Libraries and Packages

- ✓ Library is the location in the computer file system where the package is stored
  - ✓ **System library** is a part of a **CAD** tool
  - ✓ User library is created by the user
- ✓ Package is a file or module that contains declarations of commonly used objects that can be shared among different:
  - data type,
  - component declarations,
  - signal,
  - procedures and functions,
  - etc.

# Library declarations

Library declaration

Use all definitions from the package  
std\_logic\_1164

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

ENTITY nand_gate IS
    PORT (
        a      : IN STD_LOGIC;
        b      : IN STD_LOGIC;
        z      : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
    z <= a NAND b;
END model;
```

Data type

# Library declarations - syntax

```
LIBRARY library_name;  
USE library_name.package_name.package_parts;
```

# Library declarations - syntax

```
-- Package declaration
PACKAGE name_of_package IS
    -- package declarations
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package name_of_package;

[-- Package body declarations
PACKAGE BODY name_of_package IS
    -- package body declarations
END PACKAGE BODY name_of_package;]
```

# Fundamental parts of a library

## LIBRARY

### PACKAGE 1

TYPES  
CONSTANTS  
FUNCTIONS  
PROCEDURES  
COMPONENTS

### PACKAGE 2

TYPES  
CONSTANTS  
FUNCTIONS  
PROCEDURES  
COMPONENTS

# Libraries

- **ieee**

Specifies multi-level logic system, including STD\_LOGIC, and STD\_LOGIC\_VECTOR data types

**Need to be explicitly declared**

---

- **std**

Specifies pre-defined data types (BIT, BOOLEAN, INTEGER, REAL, SIGNED, UNSIGNED, etc.), arithmetic operations, basic type conversion functions, basic text i/o functions, etc.

**Visible by default**

- **work**

Current designs after compilation

# DATA TYPES

- ✓ 10 pre-defined types:
  - ✓ BIT
  - ✓ BIT\_VECTOR,
  - ✓ **STD\_LOGIC,**
  - ✓ **STD\_LOGIC\_VECTOR,**
  - ✓ STD\_ULOGIC,
  - ✓ **SIGNED,**
  - ✓ **UNSIGNED,**
  - ✓ INTEGER,
  - ✓ ENUMERATION,
  - ✓ BOOLEAN

# **STD\_LOGIC Demystified**

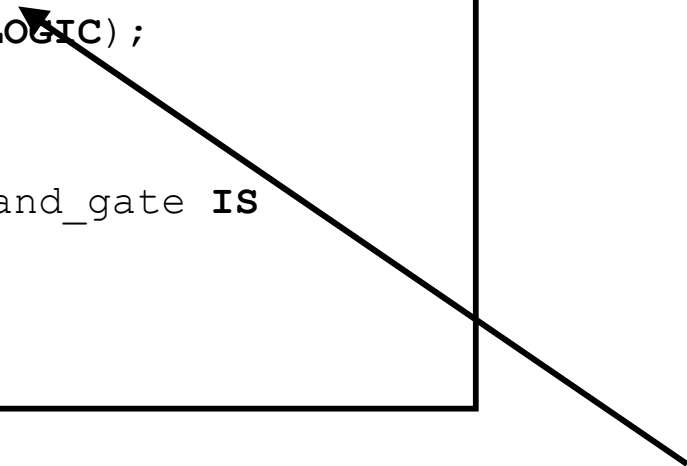


# STD\_LOGIC

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

ENTITY nand_gate IS
    PORT (
        a    : IN STD_LOGIC;
        b    : IN STD_LOGIC;
        z    : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
    z <= a NAND b;
END model;
```



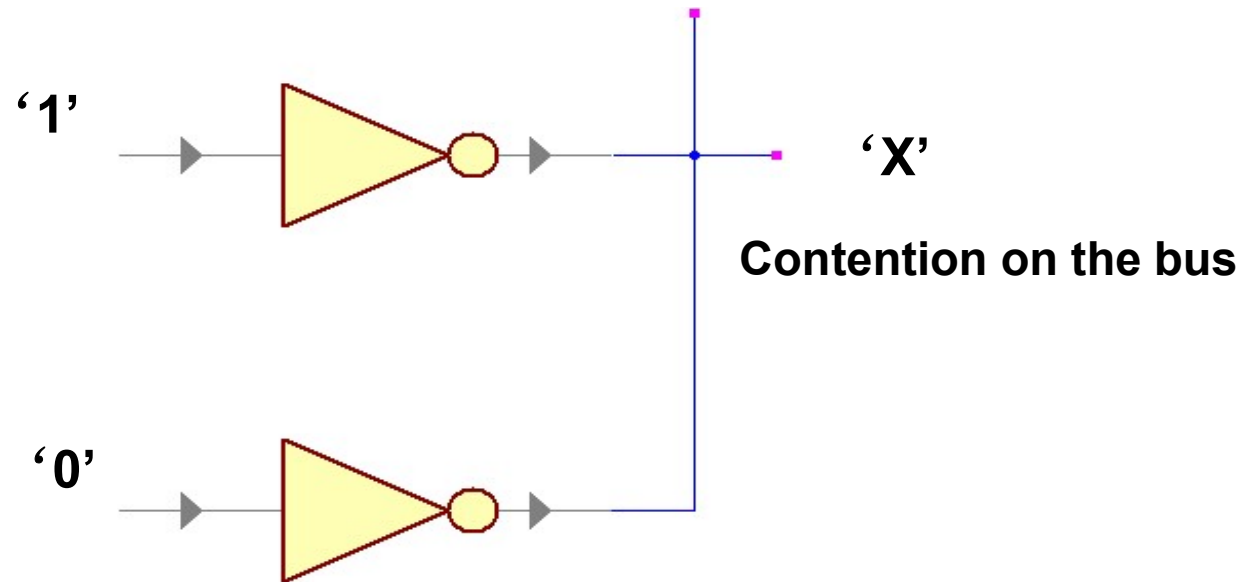
What is **STD\_LOGIC**?

# STD\_LOGIC *type* demystified

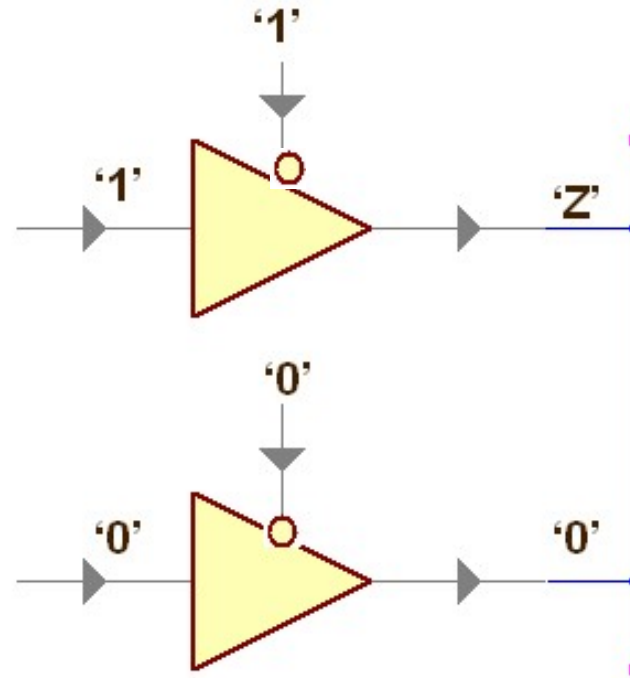
Value	Meaning
'_'	Don't Care
'0'	Forcing (Strong driven) 0
'1'	Forcing (Strong driven) 1
'Z'	High Impedance
'U'	Uninitialized value
'W'	Weak (Weakly driven) Unknown
'L'	Weak (Weakly driven) 0. Models a pull down.
'H'	Weak (Weakly driven) 1. Models a pull up.
'X'	Forcing (Strong driven) Unknown

# More on STD\_LOGIC Meanings (1)

**X**



# More on STD\_LOGIC Meanings (2)



# How to use *Std\_logic* type

- ✓ Must include the two statements:

`LIBRARY ieee ;`

`USE ieee.std_logic_1164.all ;`

- ✓ Should use only the `STD_LOGIC` data type in most cases.
  - ✓ VHDL is a strongly type-checked language

# *Std\_logic vs. Std\_logic\_vector*

- ✓ `STD_LOGIC_VECTOR` type represents an array of `STD_LOGIC` objects
  - ✓ `STD_LOGIC` objects are often used in logic expressions
  - ✓ `STD_LOGIC_VECTOR` signals can be used as binary numbers in arithmetic circuits
  - ✓ to use legally the `STD_LOGIC_VECTOR` signals with arithmetic operators, including in the code the statement:

`USE ieee.std_logic_signed.all ;`

Or: `USE ieee.std_logic_unsigned.all ;`

# *Signed vs. Unsigned*

- ✓ These types are identical to the STD\_LOGIC\_VECTOR
- ✓ These types allow the user to indicate in the VHDL code what kind of number representation is being used.
  - ✓ to use legally the SIGNED or UNSIGNED signals, including in the code the statement:  
`USE ieee.std_logic_arith.all ;`
- ✓ *std\_logic\_arith* package defines the type of circuit that should be used to implement the arithmetic operators

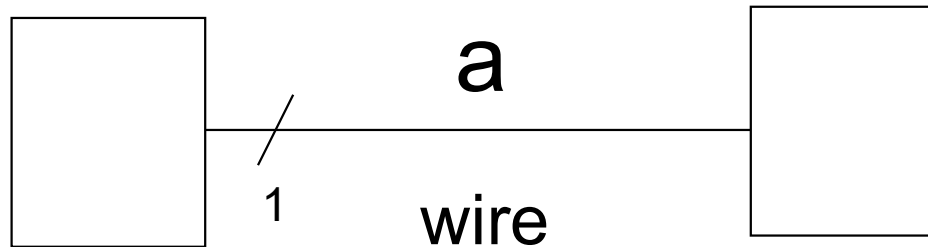
# Modeling Wires and Buses



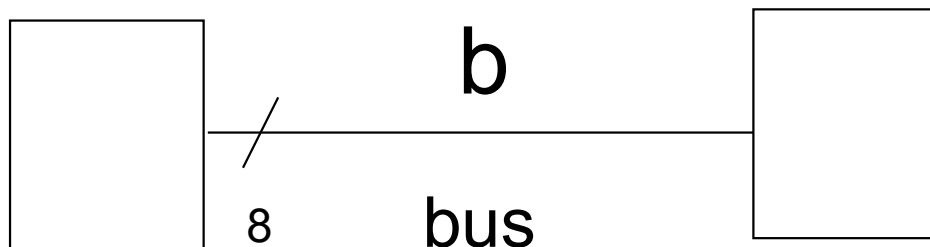
# Data Object: Signal

**SIGNAL** signal\_name1, signal\_name2 : data\_type;

SIGNAL a : STD\_LOGIC;



SIGNAL b : STD\_LOGIC\_VECTOR(7 DOWNT0 0);



# Standard Logic Vectors

```
SIGNAL a: STD_LOGIC;  
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNT0 0);  
SIGNAL c: STD_LOGIC_VECTOR(3 DOWNT0 0);  
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNT0 0);  
SIGNAL e: STD_LOGIC_VECTOR(15 DOWNT0 0);  
SIGNAL f: STD_LOGIC_VECTOR(8 DOWNT0 0);  
  
.....  
  
a <= '1';  
b <= "0000";           -- Binary base assumed by default  
c <= B"0000";          -- Binary base explicitly specified  
d <= "0110_0111";      -- You can use '_' to increase readability  
e <= X"AF67";          -- Hexadecimal base  
f <= O"723";           -- Octal base
```

# Vectors and Concatenation

```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);  
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);  
SIGNAL c, d, e: STD_LOGIC_VECTOR(7 DOWNTO 0);  
SIGNAL f: STD_LOGIC_VECTOR(6 DOWNTO 0);
```

```
a <= "0000";
```

```
b <= "1111";
```

```
c <= a & b; -- c = "00001111"
```

```
F <= "0001111";
```

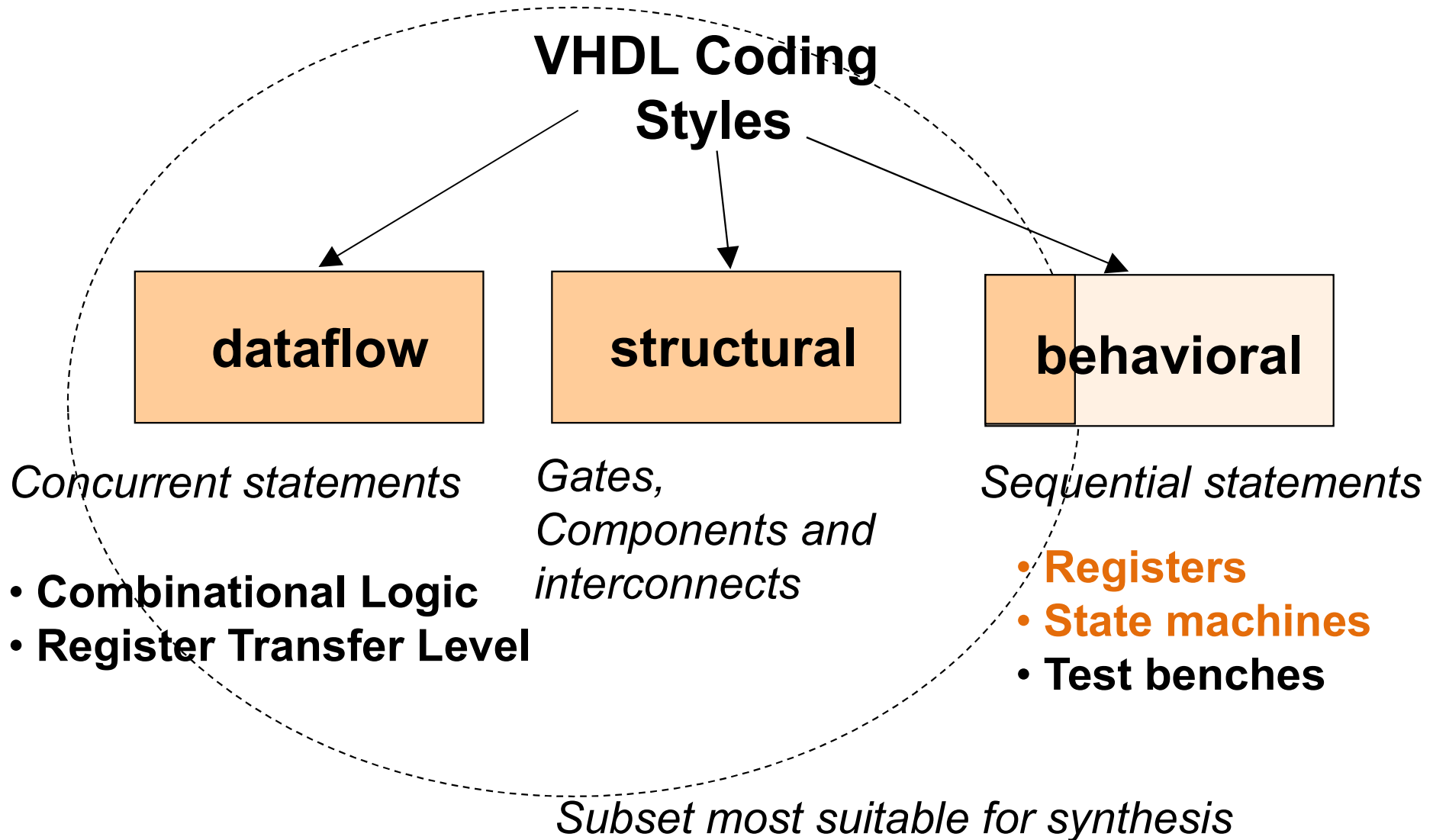
```
d <= '0' & f; -- d <= "00001111"
```

```
e <= '0' & '0' & '0' & '0' & '1' & '1' &  
    '1' & '1';
```

```
-- e <= "00001111"
```

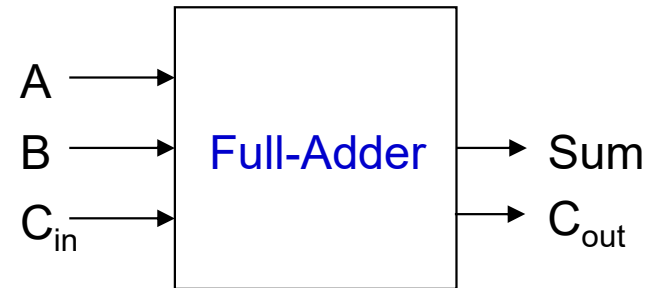
# VHDL Coding Styles

# VHDL Coding Styles

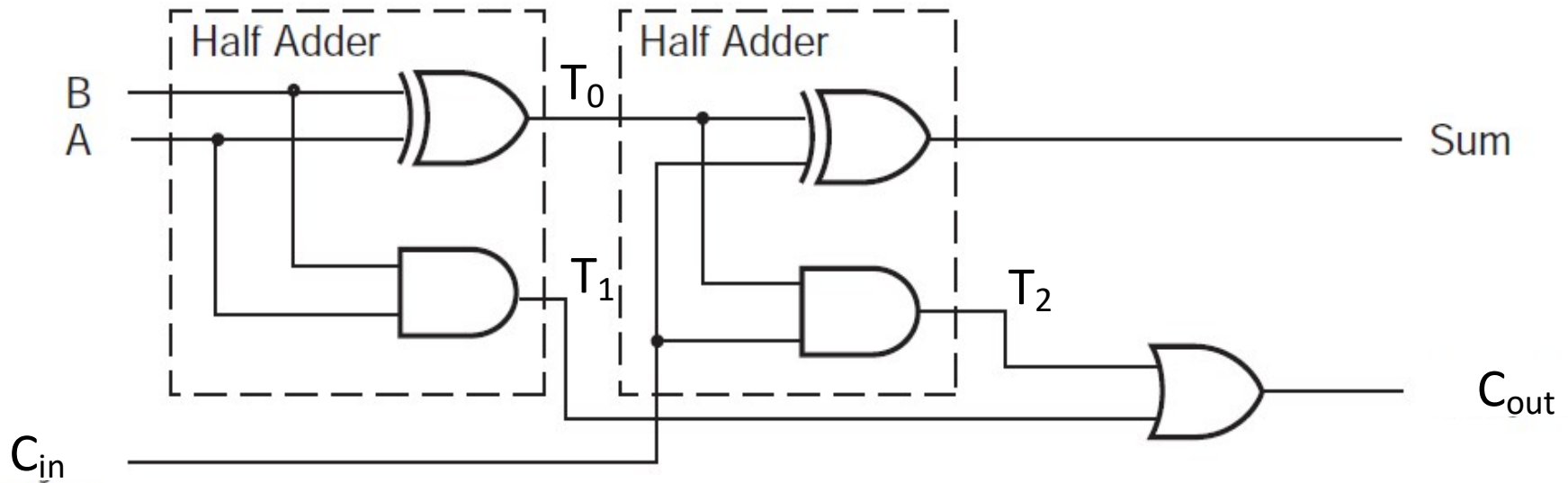


# Full-Adder Example

input			output	
a	b	$c_{in}$	sum	$c_{out}$
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1



# Dataflow Architecture (Full-adder)



# Dataflow Architecture (Full-adder)

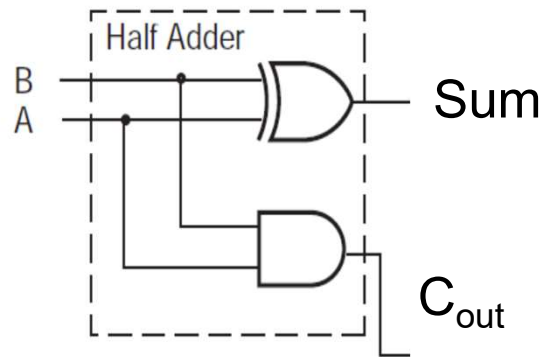
```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ENTITY fulladder_dfl IS
4      PORT (
5          a,b : IN  std_logic;
6          cin : IN  std_logic;
7          sum  : OUT std_logic;
8          cout : OUT std_logic);
9  END fulladder_dfl;
10
11 ARCHITECTURE architecture_fulladder_dfl OF fulladder_dfl IS
12     Signal T0, T1, T2 : STD_LOGIC;
13 BEGIN
14     --Statements are here
15         T0 <= A XOR B;
16         T1 <= A AND B;
17         SUM <= Cin XOR T0;
18         T2 <= Cin AND T0;
19         Cout <= T1 OR T2;
20 END architecture_fulladder_dfl;
```



# Dataflow Description

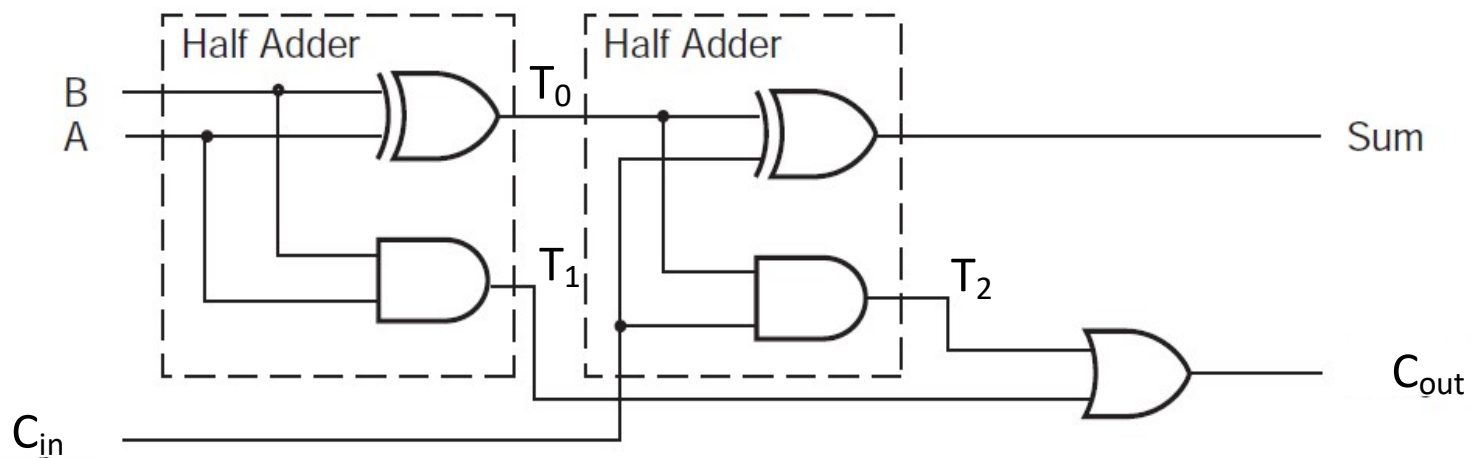
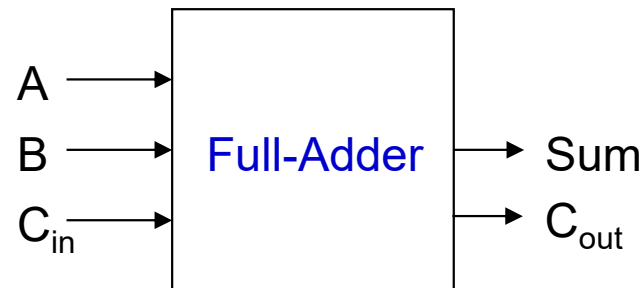
- Describes how data moves through the system and the various processing steps.
- Data Flow uses series of concurrent statements to realize logic. Concurrent statements are evaluated at the same time; thus, order of these statements doesn't matter.
- Data Flow is most useful style when series of Boolean equations can represent a logic.

# Structural Architecture (Full-adder)



**First, create a Half Adder (How?)**

**Next, use the half adder as a component to build Full Adder.**



# Half-Adder Design

```
1  -- halfadder.vhd
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4
5  ENTITY halfadder IS
6
7      PORT (
8          a,b : IN  std_logic;
9          sum  : OUT std_logic;
10         cout : OUT std_logic);
11  END halfadder;
12
13  ARCHITECTURE architecture_halfadder OF halfadder IS
14
15  BEGIN
16      --Statements are here
17          sum <= A XOR B;
18          cout <= A AND B;
19  END architecture_halfadder;
```

# Structural Architecture (Full-Adder)

```
1  -- fulladder_str.vhd
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4
5  ENTITY fulladder_str IS
6
7      PORT (
8          a,b : IN  std_logic;
9          cin : IN  std_logic;
10         sum  : OUT std_logic;
11         cout : OUT std_logic);
12  END fulladder_str;
13
14  ARCHITECTURE architecture_fulladder_str OF fulladder_str IS
15      COMPONENT halfadder
16      PORT (
17          a,b : IN  std_logic;
18          sum  : OUT std_logic;
19          cout : OUT std_logic);
20      END COMPONENT;
21      Signal T0, T1, T2 : STD_LOGIC;
22  BEGIN
23      --Statements are here
24      Hadder1: halfadder PORT MAP (a, b, T0, T1);
25      Hadder2: halfadder PORT MAP (T0, Cin, sum, T2);
26      Cout <= T1 OR T2;
27  END architecture_fulladder_str;
```

# Component declaration – simplified syntax

- Put either in the declaration region of an architecture or in a package declaration.

```
COMPONENT component_name
  [ GENERIC generic_declarations];
  PORT (
    port_name : signal_mode signal_type;
    port_name : signal_mode signal_type;
    .....
    port_name : signal_mode signal_type);
END COMPONENT;
```

# *Component Instantiation* – simplified syntax

```
instance_name: component_name  
[GENERIC MAP (formal_name => Value)]  
PORT MAP (formal_name => actual_name,  
          ...  
          formal_name => actual_name) ;
```

# Component and Instantiation (1)

- Named association connectivity (recommended)

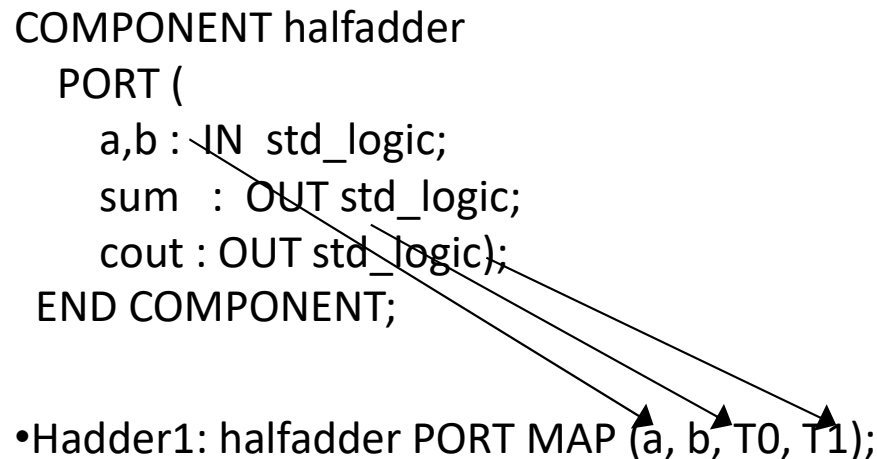
```
COMPONENT halfadder
  PORT (
    a,b : IN std_logic;
    sum  : OUT std_logic;
    cout : OUT std_logic);
END COMPONENT;
Hadder1: halfadder PORT MAP ( a => a,
                              b => b,
                              Sum => T0,
                              Cout => T1);
```

# Component and Instantiation (2)

- Positional association connectivity  
(not recommended)

```
COMPONENT halfadder
  PORT (
    a,b : IN std_logic;
    sum  : OUT std_logic;
    cout : OUT std_logic);
END COMPONENT;

•Hadder1: halfadder PORT MAP (a, b, T0, T1);
```





# Structural Description

- Structural design is the simplest to understand. This style is the closest to schematic capture and utilizes simple building blocks to compose logic functions.
- Components are interconnected in a hierarchical manner.
- Structural descriptions may connect simple gates or complex, abstract components.
- Structural style is useful when expressing a design that is naturally composed of sub-blocks.

# Behavioral Description

- It accurately models what happens on the inputs and outputs of the black box (no matter what is inside and how it works).
- This style uses PROCESS statements in *VHDL*.

# Behavioral Architecture (Full-Adder)

Input			Output	
a	b	c <sub>in</sub>	C <sub>out</sub>	Sum
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



a	b	C <sub>out</sub>	Sum
0	0	a	C <sub>in</sub>
0	1	C <sub>in</sub>	$\overline{C_{in}}$
1	0	C <sub>in</sub>	$\overline{C_{in}}$
1	1	a	C <sub>in</sub>

Look-ahead carry full-adder

# Behavioral Architecture (Full-Adder)

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ENTITY fulladder_bev IS
4      PORT (
5          a,b : IN std_logic;
6          cin : IN std_logic;
7          sum  : OUT std_logic;
8          cout : OUT std_logic);
9  END fulladder_bev;
10 ARCHITECTURE architecture_fulladder_bev OF fulladder_bev IS
11 BEGIN
12     --Statements are here
13     adding_proc: PROCESS (a, b, cin)
14         BEGIN
15             IF (a=b) then
16                 cout <= a;
17                 sum <= cin;
18             ELSE
19                 cout <= cin;
20                 sum <= not cin;
21             END IF;
22         END PROCESS;
23 END architecture_fulladder_bev;
```