

# Introduction to VHDL

# Objectives

## ■ Theory:

- Understand Basic Constructs of VHDL
- Understand Modeling Structures of VHDL

## ■ Lab:

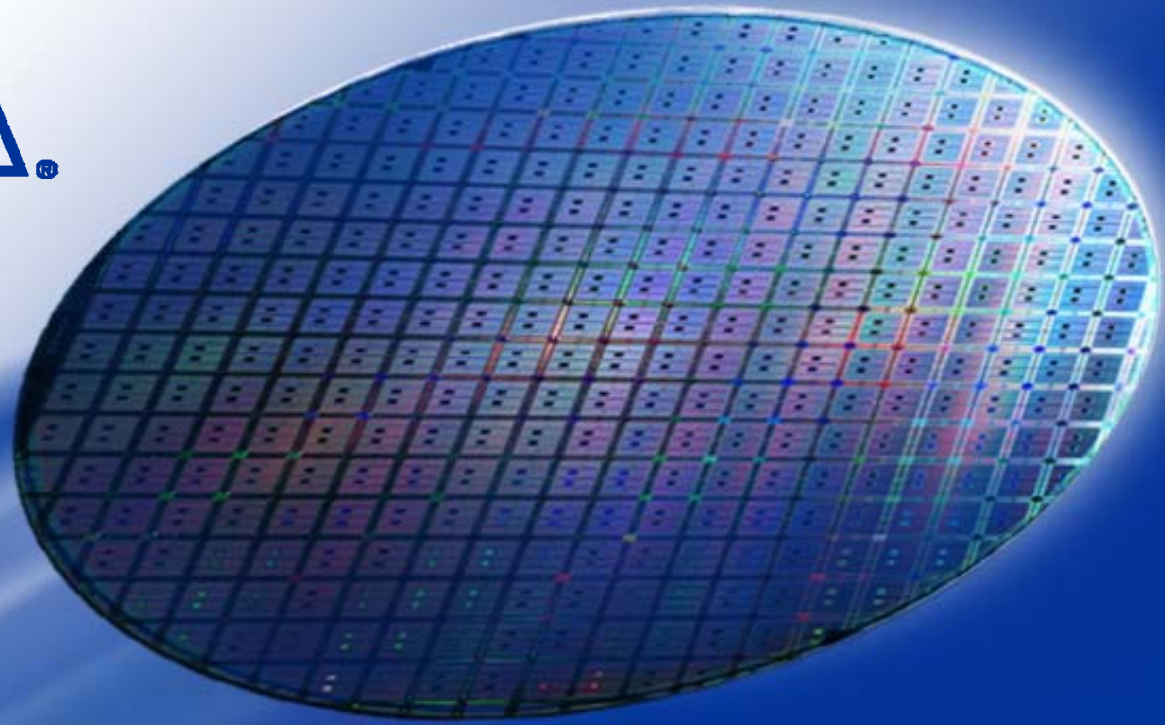
- Obtain an overview of Altera FPGA technology
- Create a New Quartus II Project
- Compile a Design into an FPGA
- Analyze the Design Environment

# Course Outline

- Introduction to Altera Devices & Altera Design Software
- VHDL Basics
  - Overview of Language
- Design Units
  - Entity
  - Architecture
  - Configurations
  - Packages (Libraries)
- Architecture Modeling Fundamentals
  - Signals
  - Processes

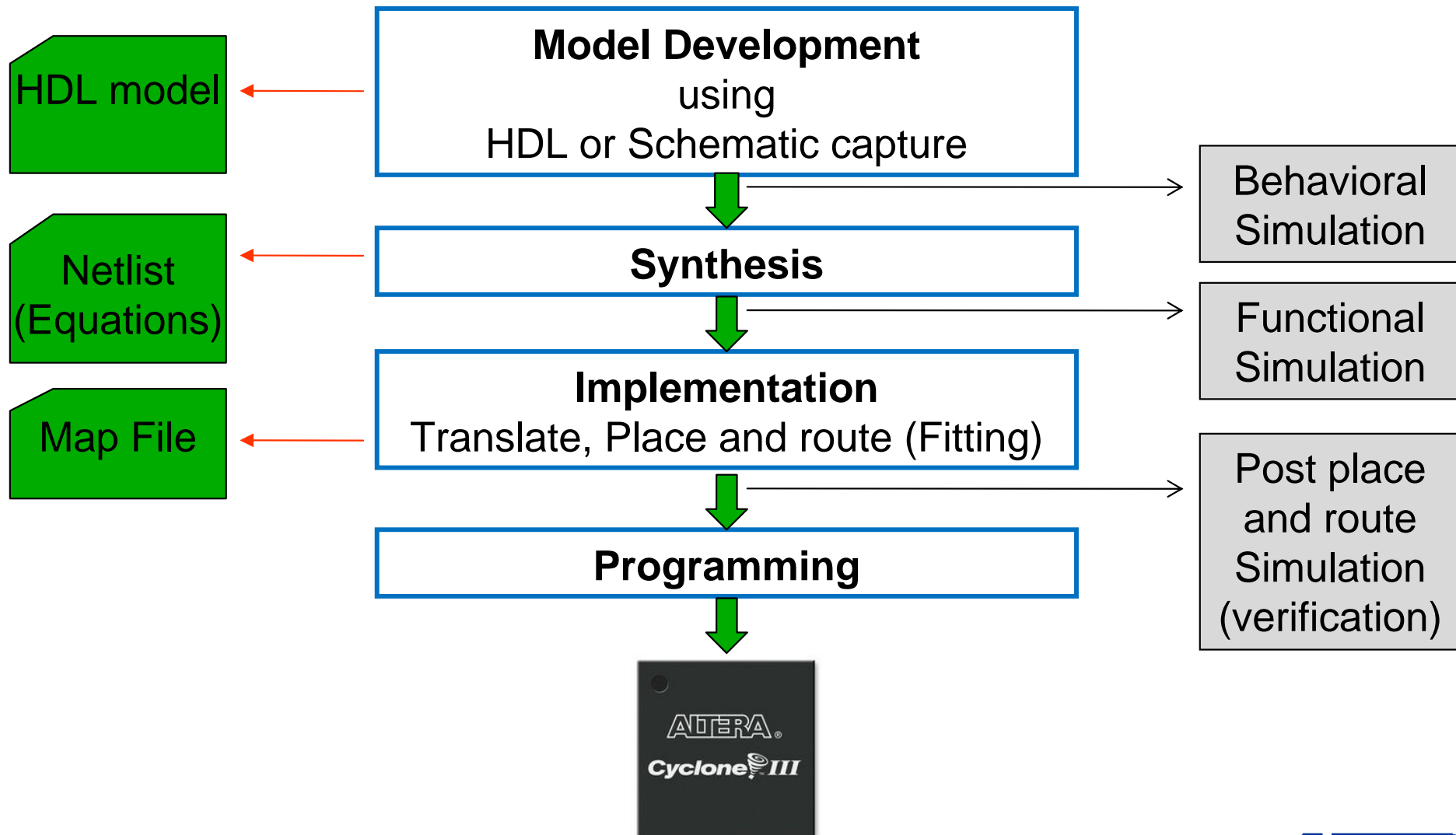
# Course Outline

- Understanding VHDL and Logic Synthesis
  - Process Statement
  - Inferring Logic
- Model Application
  - State Machine Coding
- Hierarchical Designing
  - Overview
  - Structural Modeling
  - Application of Library of Parameterized Modules (LPMs)



# Design Flow

# PLD Design Flow

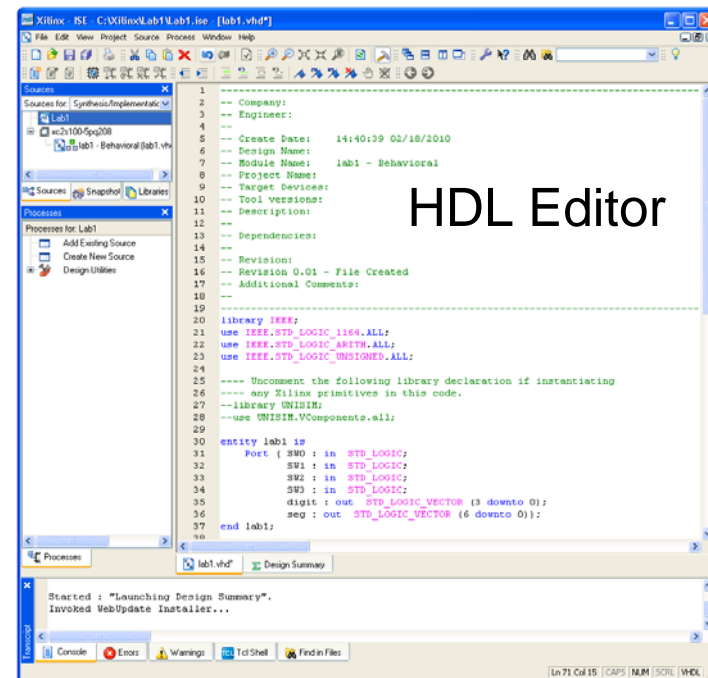
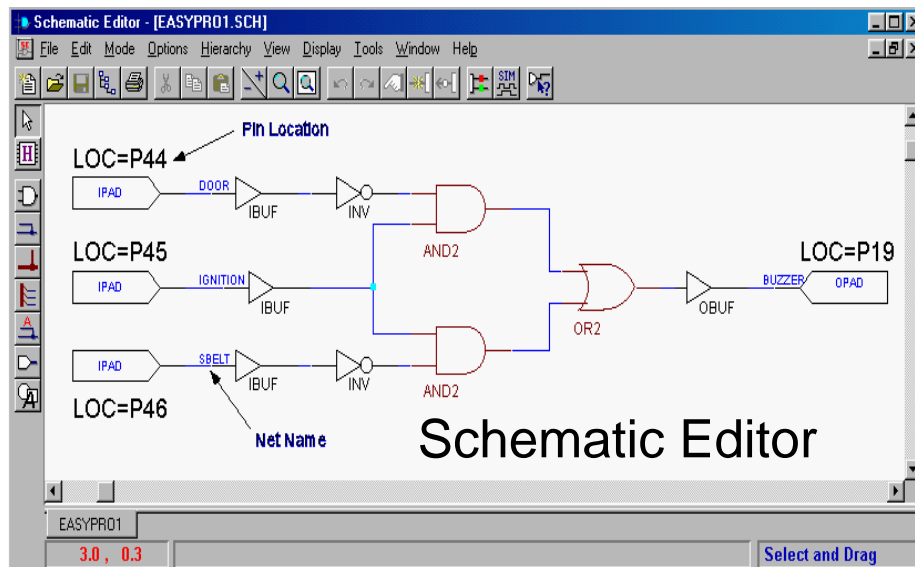




# PLD Design Flow (cont)

## ■ Model Development

- Logic design problems can be expressed in the form of graphical-based logic circuits (**schematic**) or text-based programs (hardware description language, **HDL**)
- The HDL one is more popular since the input method is less tedious



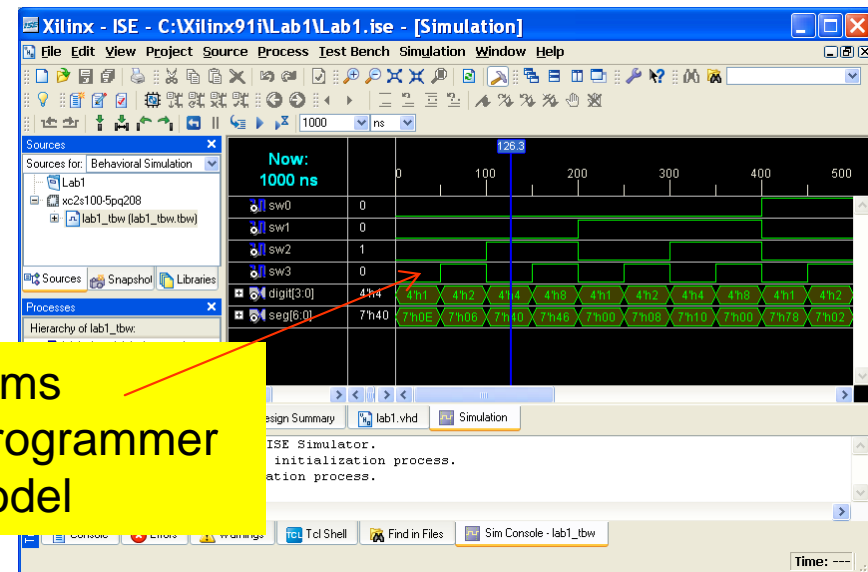
# PLD Design Flow (cont)

## ■ Behavioral Simulation

- The HDL model can be simulated before it is really mapped to the hardware constructs of the target FPGA
- The purpose of this **behavioral simulation** is usually to establish functional correctness
- It is usually much faster than the more detailed simulation (with timing consideration) after synthesis

- Testbench waveforms are generated to test the designed HDL model to verify its outputs

testbench waveforms  
designed by the programmer  
to test the HDL model





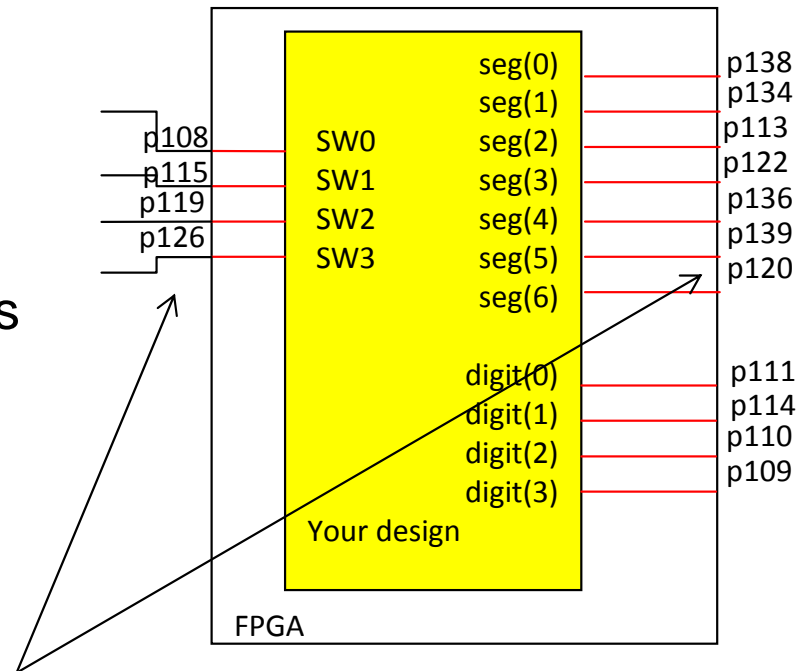
# PLD Design Flow (cont)

- **Synthesis**: logic synthesis is a process by which an abstract form of desired circuit behavior (HDL model in this case) is turned into a design implementation in terms of logic gates
- For FPGA, a **netlist** with format following an industrial standard will result
  - A netlist is just a simple text description of the logic gates and their connections used in a design
- For CPLD, an ensemble of logic **equations** will result
- Another **Functional Simulation** can be carried out after the synthesis process
  - Also mainly for verifying the functionality of the logic design
  - But now some timing information can be incorporated since the usage of logic gates in the design is known
  - However, the timing is not completely accurate since some exact details of the FPGA still are not known at this stage

# PLD Design Flow (cont)

## ■ Implementation

- **Translate** is the first step in the implementation process
  - The Translate process merges all of the input netlists and design constraint information (such as the pin assignment) and outputs a device (manufacturer) specific file
- For FPGA, the **place and route** process is then carried out
  - Should place the logic gates to different LEs (CLBs )
  - Then route the interconnections between them
- For CPLD, **fitting** of the project to available hardware resource is performed



pin assignment, one of the user constraints

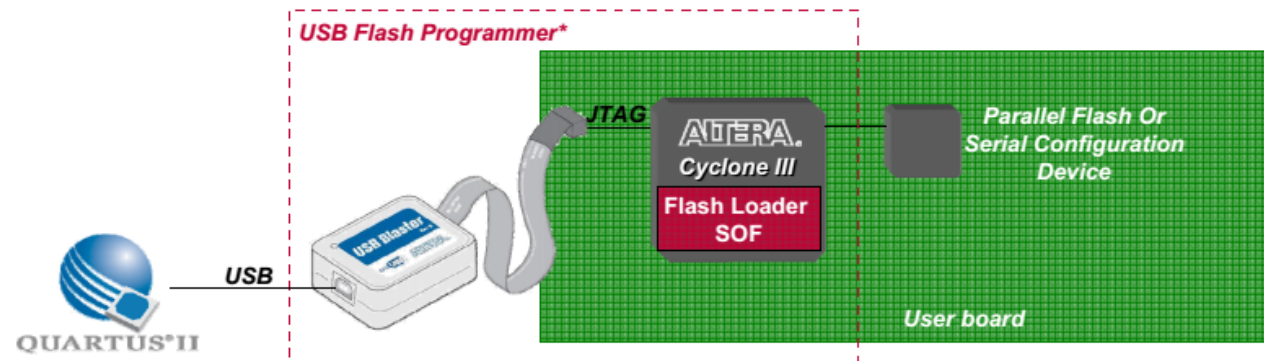
# PLD Design Flow (cont)

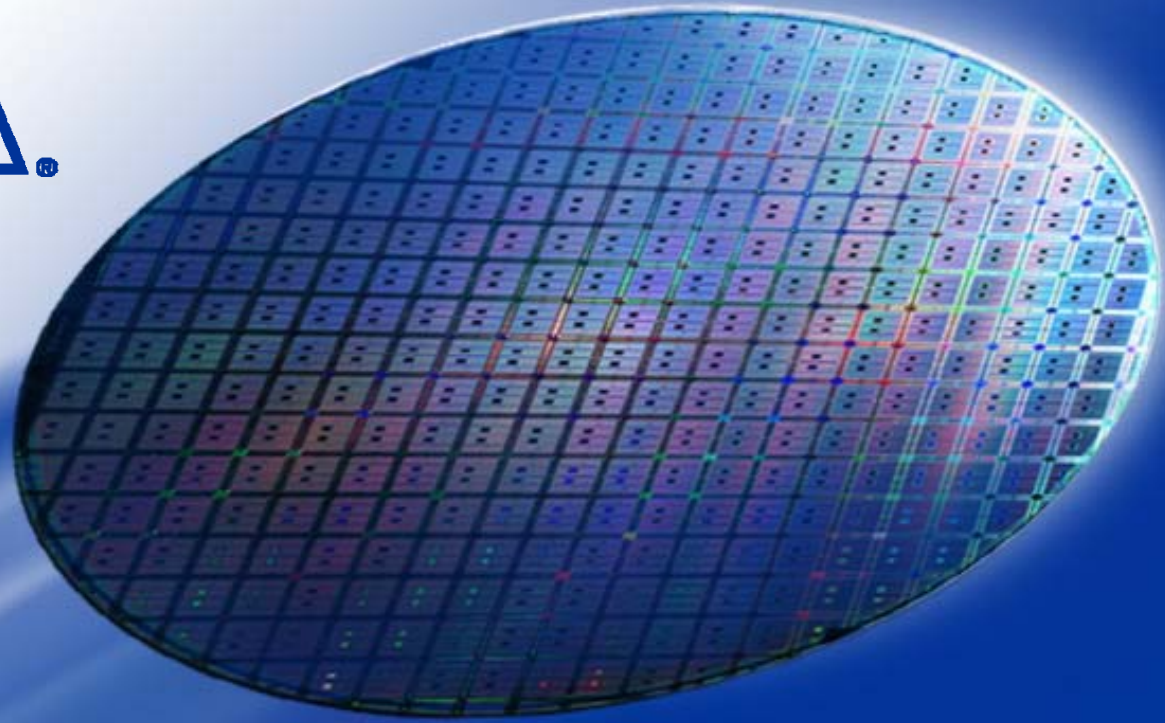
## ■ Post place and route (fitting) simulation (verification)

- The design is close to final. All interconnections and the LEs (CLBs) used in the design have been confirmed. Hence the actual timing can be determined
- A **post place and route (fitting) simulation** is often carried out at this moment to **verify the design** as a whole

## ■ Programming

- The implementation process will result in a vendor dependent file, which keeps the binary bitstream that can be sent to the FPGA for configuration – using PROM or download from computer





# **Introduction to Altera Devices & Design Software**

# Software & Development Tools



## ■ Quartus II

- Stratix II, Cyclone II, Cyclone III, Stratix GX, MAX II, Stratix HardCopy, Stratix, Cyclone, APEX II, APEX 20K/E/C, Excalibur, & Mercury Devices
- FLEX 10K/A/E, ACEX 1K, FLEX 6000, MAX 7000S/AE/B, MAX 3000A Devices

## ■ Quartus II Web Edition

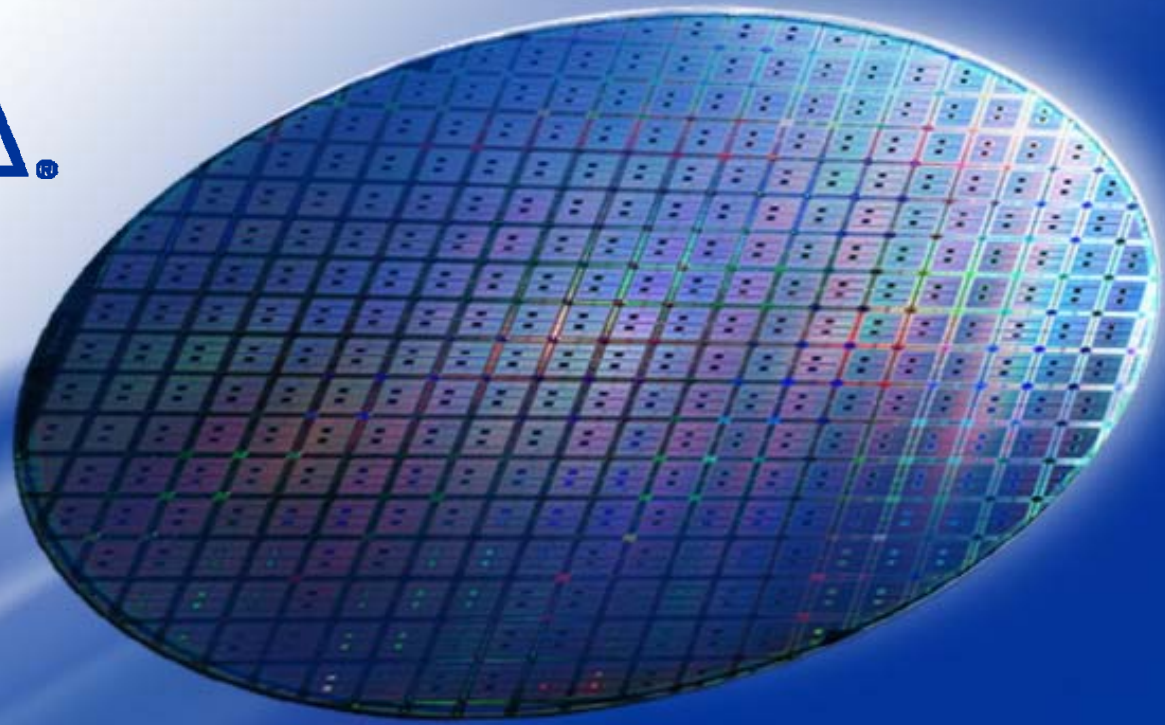
- Free Version
- Not All Features & Devices Included



## ■ MAX+PLUS® II

- All FLEX, ACEX, & MAX Devices





# VHDL Basics

# VHDL

**V**HSIC (Very High Speed Integrated Circuit)

**H**ardware

**D**escription

**L**anguage



# What is VHDL?

- IEEE Industry Standard Hardware Description Language
- High-level Description Language for Both Simulation & Synthesis

# VHDL History

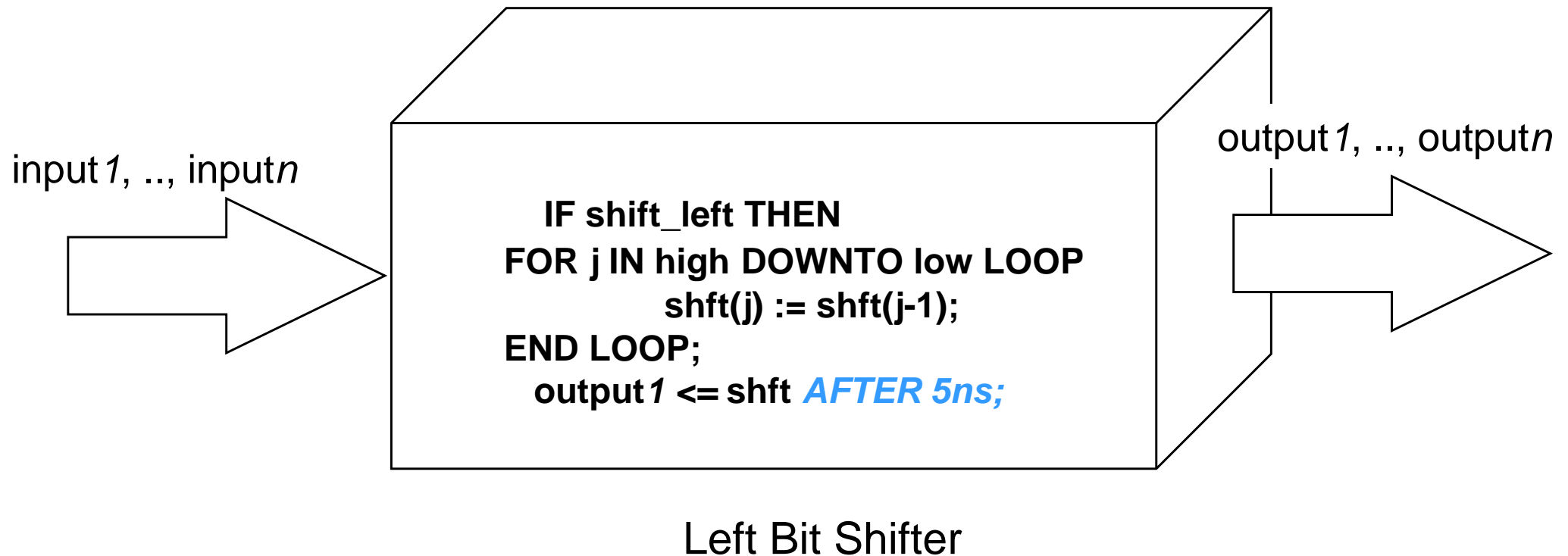
- 1980 - U.S. Department of Defense (DOD) Funded a Project to Create a Standard Hardware Description Language Under the Very High Speed Integrated Circuit (VHSIC) Program
- 1987 - the Institute of Electrical and Electronics Engineers (IEEE) Ratified As IEEE Standard 1076
- 1993 - the VHDL Language Was Revised and Updated to IEEE 1076 '93

# Terminology

- HDL - Hardware Description Language Is a Software Programming Language That Is Used to Model a Piece of Hardware
- Behavior Modeling - A Component Is Described by Its Input/Output Response
- Structural Modeling - A Component Is Described by Interconnecting Lower-level Components/Primitives

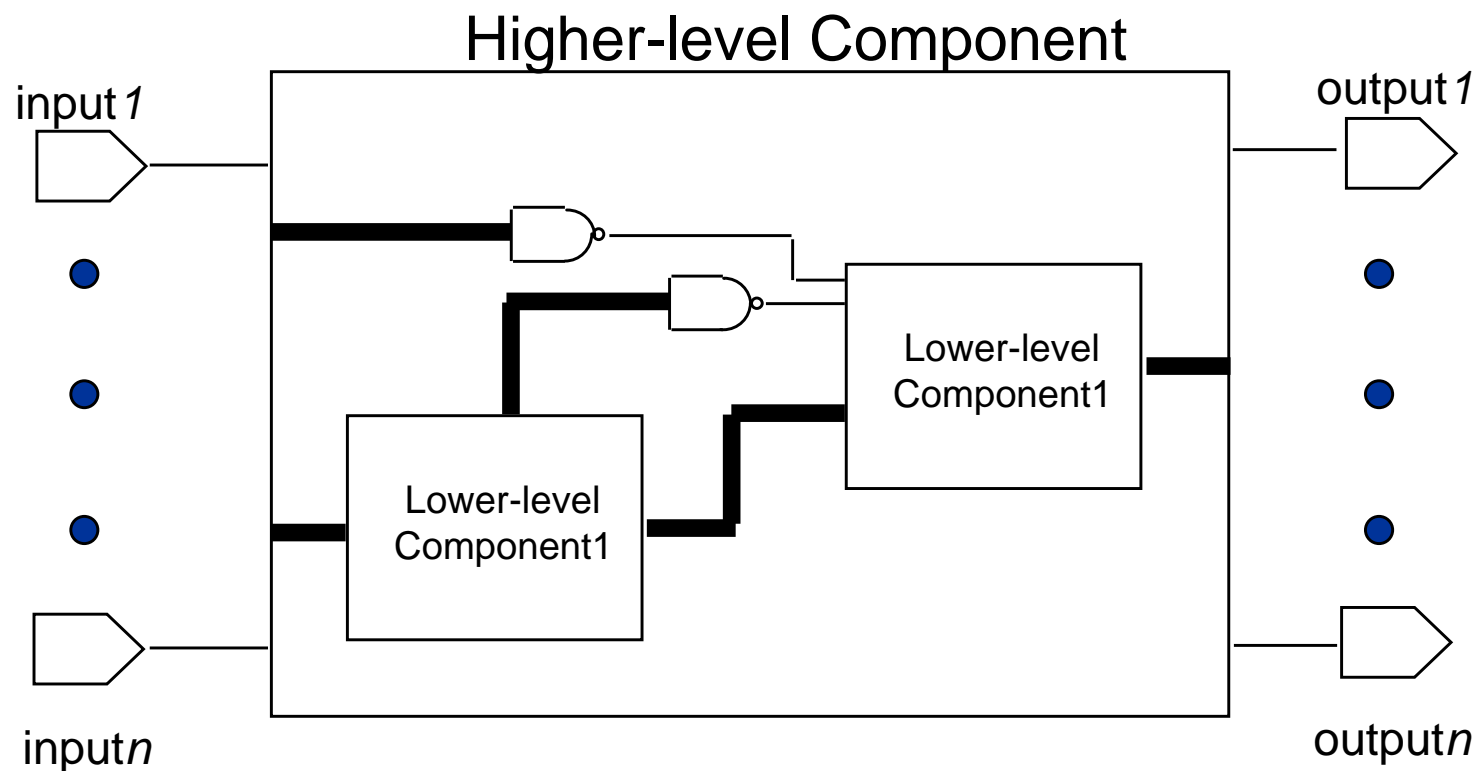
# Behavior Modeling

- Only the Functionality of the Circuit, No Structure
- No Specific Hardware Intent
- For the Purpose of Synthesis, As Well As *Simulation*



# Structural Modeling

- Functionality and Structure of the Circuit
- Call Out the Specific Hardware
- For the Purpose of Synthesis



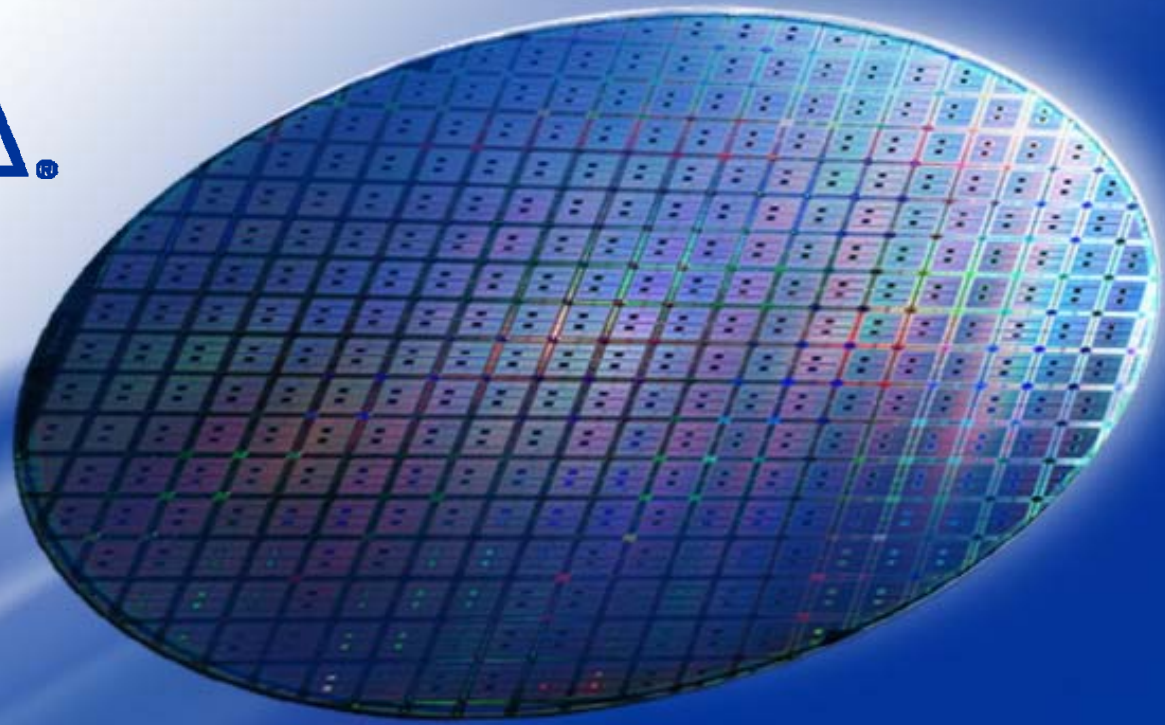
# More Terminology

- Register Transfer Level (RTL) - A Type of Behavioral Modeling, for the Purpose of Synthesis
  - An RTL description describes a circuit's registers and the sequence of transfers between these registers but does not describe the hardware used to carry out these operations
- Synthesis - Translating HDL to a Circuit and Then Optimizing the Represented Circuit
  - RTL Synthesis - The Process of Translating a RTL Model of Hardware Into an Optimized Technology Specific Gate Level Implementation

# VHDL Basics

- Two Sets of Constructs:
  - Synthesis
  - Simulation
- The VHDL Language Is Made up of Reserved Keywords
- The Language Is, for the Most Part, **Not** Case Sensitive
- VHDL Statements Are Terminated With a ;
- VHDL Is White Space Insensitive. Used for Readability.
- Comments in VHDL Begin With “--” to Eol
- VHDL Models Can Be Written:
  - Behavioral
  - Structural
  - Mixed





# VHDL Design Units

# VHDL Basics

## ■ VHDL Design Units

- Entity

- Used to Define External View of a Model.  
I.E. Symbol

- Architecture

- Used to Define the Function of the Model.  
I.E. Schematic

# VHDL Basics

## ■ VHDL Design Units (cont.)

### – Package

- Collection of Information That Can Be Referenced by VHDL Models. I.E. Library
- Consist of Two Parts Package Declaration and Package Body

# Entity Declaration

**ENTITY** *<entity\_name>* **IS**

Generic Declarations

Port Declarations

**END** *<entity\_name>;* (1076-1987 version)

**END ENTITY** *<entity\_name>* ; ( 1076-1993 version)

## ■ Analogy : Symbol

## ■ *<Entity\_name>* Can Be Any Alpha/Numerical Name

- Note: MAX+PLUS II Requires That the *<Entity\_name>* and *<File\_name>* Be the Same; Not Necessary in Quartus II

## ■ Generic Declarations

- Used to Pass Information Into a Model
- Quartus II & MAX+PLUS II Place Some Restriction on the Use of Generics

## ■ Port Declarations

- Used to Describe the Inputs and Outputs i.e. Pins

# Entity : Generic Declaration

```
ENTITY <entity_name> IS
```

```
    Generic ( constant tphl , tphl : time := 5 ns;  
              -- Note constant is assumed and is not required  
              tphz, tplz : time := 3 ns;  
              default_value : integer := 1;  
              cnt_dir : string := "up"  
            );
```

```
    Port Declarations
```

```
END <entity_name>; (1076-1987 version)
```

```
END ENTITY <entity_name> ; ( 1076-1993 version)
```

- New Values Can Be Passed During Compilation
- During Simulation/Synthesis a Generic Is Read Only

# Entity : Port Declarations

**ENTITY** *<entity\_name>* **IS**

Generic Declarations

**Port ( signal** clk : **in** bit;

--Note: **signal** is assumed and is not required

q : **out** bit

**);**

**END** *<entity\_name>; (1076-1987 version)*

**END ENTITY** *<entity\_name>; ( 1076-1993 version)*

■ Structure : **<Class> Object\_name : <Mode>**  
**<Type>;**

- **<Class>** : What Can Be Done to an Object

- **Object\_name** : Identifier

- **<Mode>** : Directional

  - **in** (Input)

  - Out** (Output)

  - **Inout** (Bidirectional)

  - Buffer** (Output W/ Internal Feedback)

- **<Type>** : What Can Be Contained in the Object

# Architecture

- Analogy : Schematic
- Describes the Functionality and Timing of a Model
- Must Be Associated With an **ENTITY**
- **ENTITY** Can Have Multiple Architectures
- Architecture Statements Execute Concurrently (Processes)



# Architecture (cont.)

## ■ Architecture Styles

- Behavioral : How Designs Operate
  - RTL : Designs Are Described in Terms of Registers
  - Functional : No Timing
- Structural : Netlist
  - Gate/Component Level
- Hybrid : Mixture of the Above

# Architecture

## **ARCHITECTURE** <Identifier> **OF** <Entity\_identifier> **IS**

--Architecture Declaration Section (List Does Not Include All)

**SIGNAL** Temp : Integer := 1; -- Signal Declarations :=1 Is Default Value Optional

**CONSTANT** Load : Boolean := True; --Constant Declarations

**TYPE** States **IS** ( S1, S2, S3, S4) ; --Type Declarations

--Component Declarations Discussed Later

--Subtype Declarations

--Attribute Declarations

--Attribute Specifications

--Subprogram Declarations

--Subprogram Body

## **BEGIN**

Process Statements

Concurrent Procedural Calls

Concurrent Signal Assignment

Component Instantiation Statements

Generate Statements

**END** <Architecture Identifier> ; *(1076-1987 Version)*

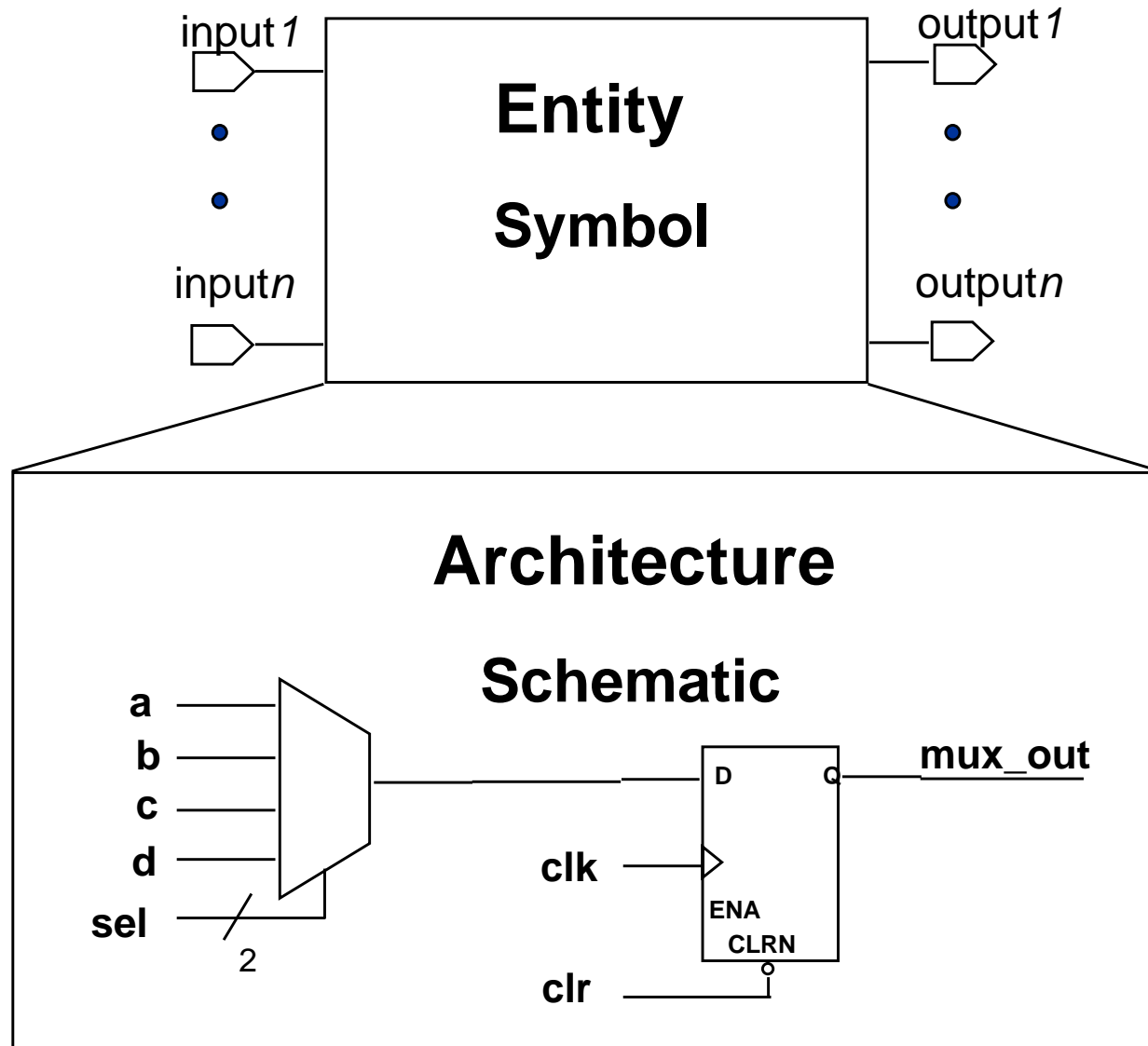
**End ARCHITECTURE;** *(1076-1993 Version)*

# VHDL - Basic Modeling Structure

```
ENTITY entity_name IS  
    generics  
    port declarations  
END entity_name;
```

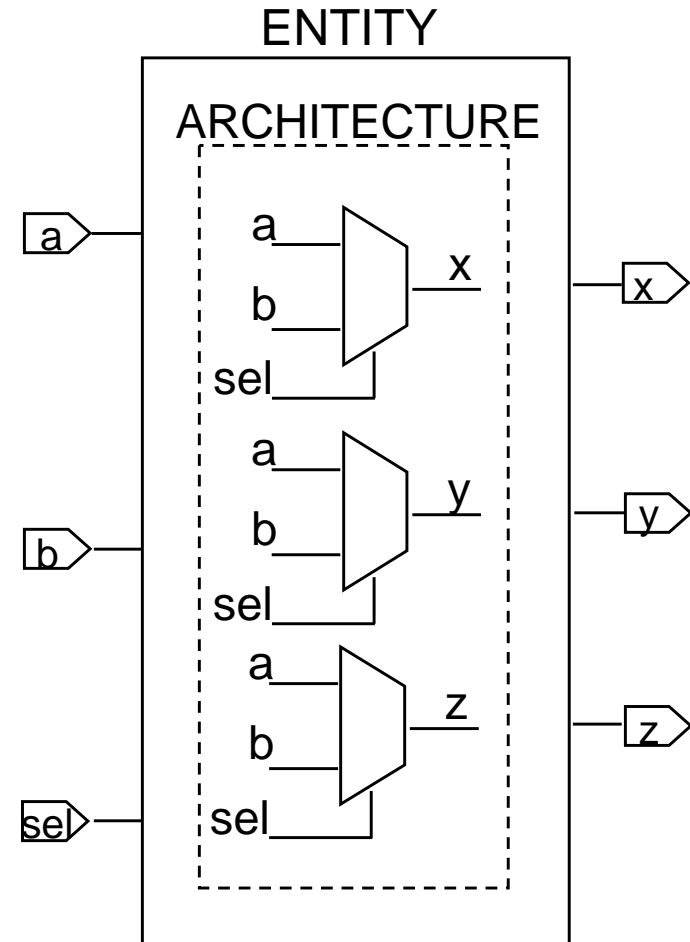
```
ARCHITECTURE arch_name OF entity_name IS  
    enumerated data types  
    internal signal declarations  
    component declarations  
BEGIN  
    signal assignment statements  
    process statements  
    component instantiations  
END arch_name;
```

# VHDL : Entity - Architecture



# Putting It All Together

```
ENTITY cmpl_sig IS  
PORT ( a, b, sel : IN bit;  
        x, y, z : OUT bit);  
END cmpl_sig;  
ARCHITECTURE logic OF cmpl_sig IS  
BEGIN  
    -- simple signal assignment  
    x <= (a AND NOT sel) OR (b AND sel);  
    -- conditional signal assignment  
    y <= a WHEN sel='0' ELSE  
        b;  
    -- selected signal assignment  
    WITH sel SELECT  
        z <= a WHEN '0',  
            b WHEN '1',  
            '0' WHEN OTHERS;  
END logic;
```



# Packages

- Packages Are a Convenient Way of Storing and Using Information Throughout an Entire Model
- Packages Consist Of:
  - Package Declaration (Required)
    - Type Declarations
    - Subprograms Declarations
  - Package Body (Optional)
    - Subprogram Definitions
- VHDL Has Two Built-in Packages
  - Standard
  - Textio

# Libraries

- Contains a Package or a Collection of Packages
- Resource Libraries
  - Standard Package
  - IEEE Developed Packages
  - Altera Component Packages
  - Any Library of Design Units That Are Referenced in a Design
- Working Library
  - Library Into Which the Unit Is Being Compiled



# Model Referencing of Library/Package

- All Packages Must Be Compiled
- Implicit Libraries
  - Work
  - Std
- ⇒ Note: Items in These Packages Do Not Need to Be Referenced, They Are Implied
- **LIBRARY** Clause
  - Defines the Library Name That Can Be Referenced
  - Is a Symbolic Name to Path/Directory
  - Defined by the Compiler Tool
- **USE** Clause
  - Specifies the Package and Object in the Library That You Have Specified in the Library Clause

# Example

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY cmpl_sig IS
PORT ( a, b, sel : IN std_logic;
       x, y, z : OUT std_logic);
END cmpl_sig;
ARCHITECTURE logic OF cmpl_sig IS
BEGIN
    -- simple signal assignment
    x <= (a AND NOT sel) OR (b AND sel);
    -- conditional signal assignment
    y <= a WHEN sel='0' ELSE
        b;
    -- selected signal assignment
    WITH sel SELECT
        z <= a WHEN '0',
            b WHEN '1',
            '0' WHEN OTHERS;

END logic;
CONFIGURATION cmpl_sig_conf OF cmpl_sig IS
    FOR logic
    END FOR;
END cmpl_sig_conf;
```

- **LIBRARY** <Name>, <Name> ;
  - Name Is Symbolic and Defined by Compiler Tool
  - ⇒ Note: Remember That WORK and STD Do Not Need to Be Defined.
- **Use**  
Lib\_name.Pack\_name.Object;
  - **All** Is a Reserved Word
- Placing the Library/Use Clause First Will Allow All Following Design Units to Access It

# Libraries

## ■ Library Std;

- Contains the Following Packages:
  - **Standard** (Types: Bit, Boolean, Integer, Real, and Time; All Operator Functions to Support Types)
  - **Textio** (File Operations)
- An Implicit Library (Built-in)
  - Does Not Need to Be Referenced in VHDL Design

# Types Defined in Standard Package

## ■ Type BIT

- 2 Logic Value System ('0', '1')

**Signal** A\_temp : Bit;

- Bit\_vector Array of Bits

**Signal** Temp : Bit\_vector(3 **Downto** 0);

**Signal** Temp : Bit\_vector(0 **to** 3) ;

## ■ Type Boolean

- (False, True)

## ■ Integer

- Positive and Negative Values in Decimal

**Signal** Int\_tmp : Integer; -- 32 Bit Number

**Signal** Int\_tmp1 : Integer **Range** 0 to 255; --8 Bit Number

⇒ Note: Standard Package Has Other Types

# Libraries

## ■ Library IEEE;

- Contains the Following Packages:
  - **Std\_logic\_1164** (Std\_logic Types & Related Functions)
  - **Std\_logic\_arith** (Arithmetic Functions)
  - **Std\_logic\_signed** (Signed Arithmetic Functions)
  - **Std\_logic\_unsigned** (Unsigned Arithmetic Functions)

# Types Defined in Std\_logic\_1164 Package

## ■ Type **STD\_LOGIC**

- 9 Logic Value System ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
  - 'W', 'L', 'H' Weak Values (Not Supported by Synthesis)
  - 'X' - Used for Unknown
  - 'Z' - (Not 'z') Used for Tri-state
  - '-' Don't Care
- Resolved Type: Supports Signals With Multiple Drives

## ■ Type **STD\_ULOGIC**

- Same 9 Value System As STD\_LOGIC
- Unresolved Type: Does Not Support Multiple Signal Drives; Error Will Occur

# User-defined Libraries/Packages

- User-defined Packages Can Be in the Same Directory As the Design

**Library Work; --Optional**

**USE WORK.<Package Name>.All;**

- Or Can Be in a Different Directory From the Design

**LIBRARY <Any\_name>;**

**Use <Any\_name>.<Package\_name>.All;**

# **Architecture Modeling Fundamentals**

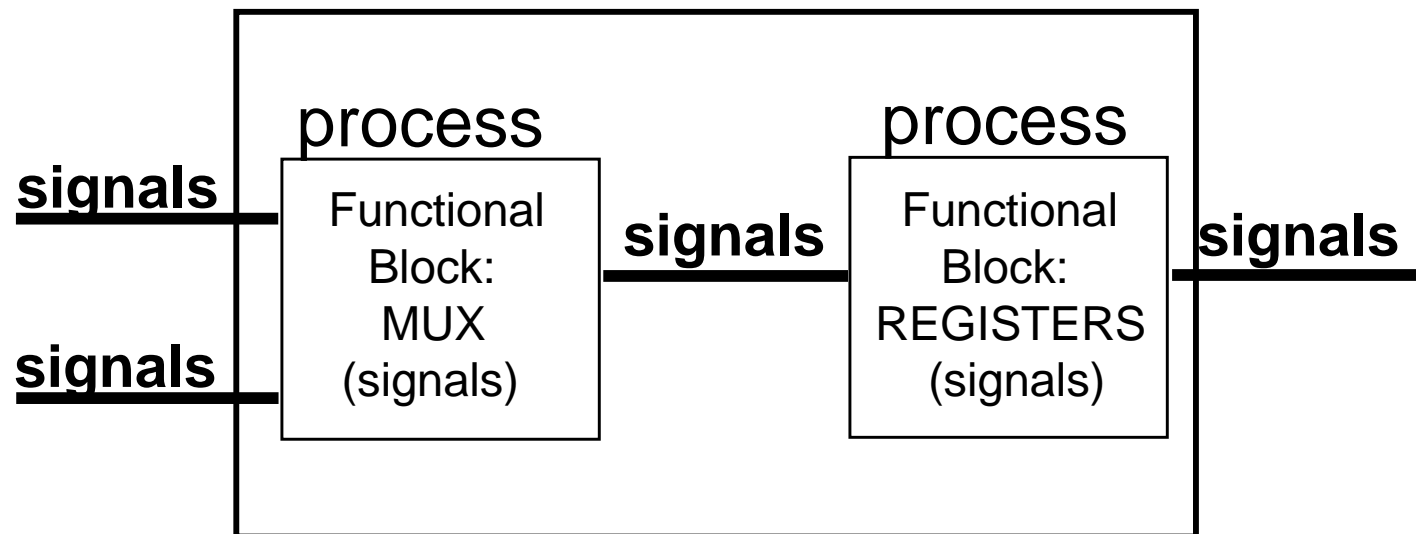


# Section Overview

- Understanding the Concept and Usage of Signals
  - Signal Assignments
  - Concurrent Signal Assignment Statements
  - Signal Delays
- Processes
  - Implied
  - Explicit
- Understanding the Concept and Usage of Variables
- Sequential Statement
  - If-then
  - Case

# Using Signals

- Signals Represent Physical Interconnect (Wire) That Communicate Between Processes (Functions)
- Signals Can Be Declared in **Packages**, **Entity** and **Architecture**



# Assigning Values to Signals

**SIGNAL** temp : **STD\_LOGIC\_VECTOR** (7 downto 0);

- All Bits:

Temp <= "10101010";

Temp <= X"aa" ; (1076-1993)

- Single Bit:

Temp(7) <= '1';

- Bit-slicing:

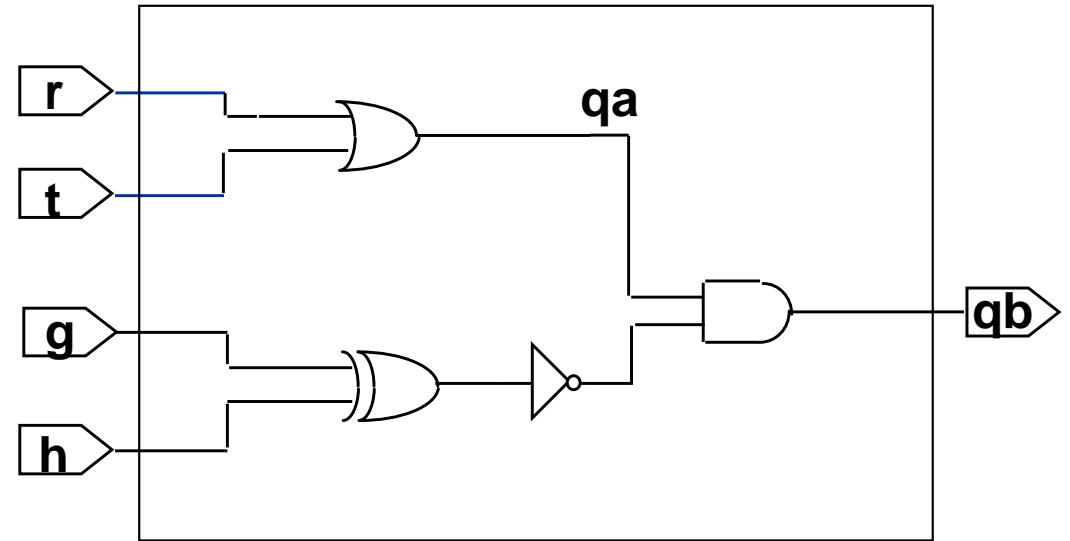
Temp (7 Downto 4) <= "1010";

- Single-bit: Single-quote (')

- Multi-bit: Double-quote (")

# Signal Used As an Interconnect

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp IS  
  PORT(r, t, g, h : IN STD_LOGIC;  
        qb : OUT STD_LOGIC);  
END simp;  
ARCHITECTURE logic OF simp IS  
  SIGNAL qa : STD_LOGIC;  
  
BEGIN  
  
  qa <= r or t;  
  qb <= (qa and not(g xor h));  
  
END logic;
```

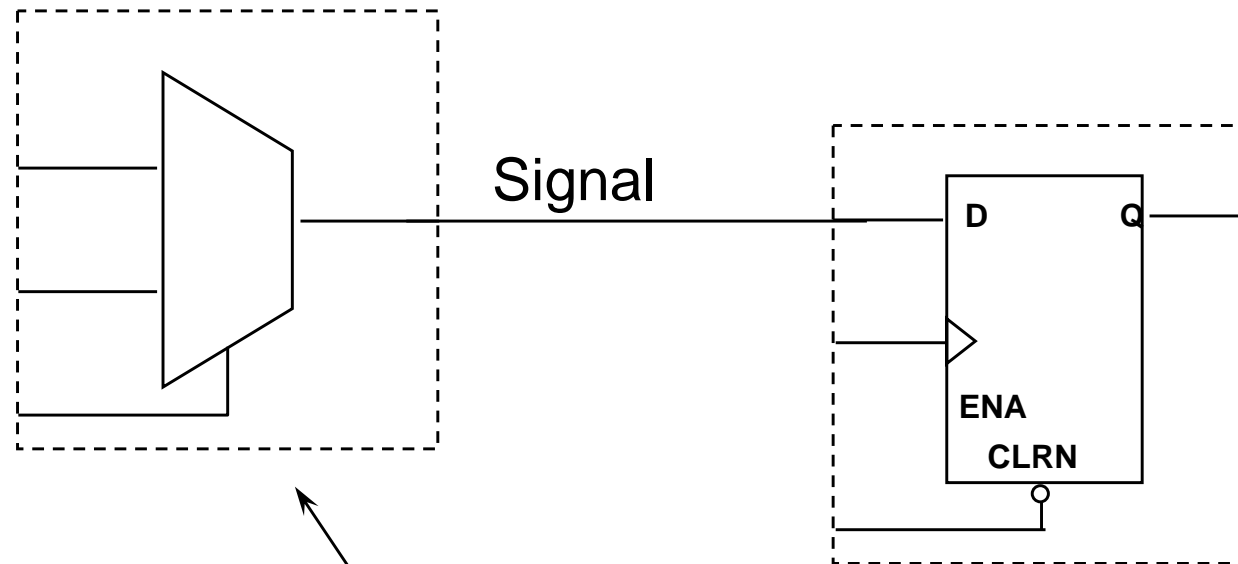


- **r, t, g, h,** and **qb** Are Signals (by Default)
- **qa** Is a Buried Signal and Needs to Be Declared

*Signal Declaration  
Inside Architecture*

# Signal Assignments

- Signal Assignments Are Represented By:  $\leq$
- Signal Assignments Have an *Implied* Process (Function) That Synthesizes to Hardware



Signal Assignment  $\leq$  Implied Process

# Concurrent Signal Assignments

- Three Concurrent Signal Assignments:
  - Simple Signal Assignment
  - Conditional Signal Assignment
  - Selected Signal Assignment

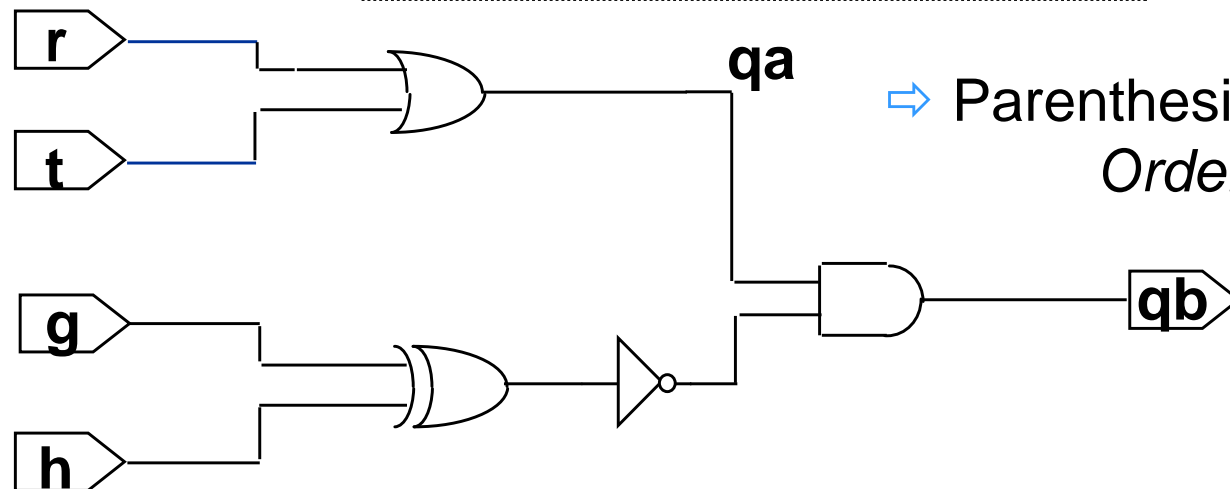
# Simple Signal Assignments

■ Format: `<signal_name> <= <expression>;`

■ Example:

```
qa <= r or t ;  
qb <= (qa and not(g xor h));
```

⇒ *Implied Processes*



■ VHDL Operators Are Used to Describe the Process

# VHDL Operators

Operator Type	Operator Name/Symbol
Logical	and or nand nor xor xnor <sup>(1)</sup>
Relational	= /= < <= > >=
Addition & Concatenation	+ - &
Signing	+ -
Multiplying	* / mod rem
Miscellaneous	** abs not

(1) Supported in VHDL '93 Only





# VHDL Operators

- VHDL Defines Arithmetic & Boolean Functions Only for Built-in Data Types (Defined in ***Standard*** Package)

- Arithmetic Operators Such As **+**, **-**, **<**, **>**, **<=**, **>=** Are Defined ***Only*** for **INTEGER** Type
- Boolean Operators Such As **AND**, **OR**, **NOT** Are Defined ***Only*** for **BIT** Type

- Recall: VHDL Implicit Library (Built-in)

- **Library STD**

- Types Defined in the **Standard** Package:

- **Bit, Boolean, Integer**

⇒ Note: Items in This Package Do Not Need to Be Referenced, They Are Implied

# Arithmetic Function

**ENTITY** opr **IS**

```
    PORT ( a  : IN  INTEGER RANGE 0 TO 16;  
          b  : IN  INTEGER RANGE 0 TO 16;  
          sum : OUT INTEGER RANGE 0 TO 32);
```

**END** opr;

**ARCHITECTURE** example **OF** opr **IS**  
**BEGIN**

```
    sum <= a + b;
```

**END** example;

*The VHDL Compiler Can Understand This Operation Because an Arithmetic Operation Is Defined for the Built-in Data Type **Integer***

⇒ Note: Remember the Library **STD** and the Package **Standard** Do Not Need to Be Referenced

# Operator Overloading

- How Do You Use Arithmetic & Boolean Functions With Other Data Types?
  - ***Operator Overloading*** - Defining Arithmetic & Boolean Functions With Other Data Types
- Operators Are Overloaded by Defining a Function Whose Name Is the Same As the Operator Itself
  - Because the Operator and Function Name Are the Same, the Function Name Must Be Enclosed Within Double Quotes to Distinguish It From the Actual VHDL Operator
  - The Function Is Normally Declared in a Package So That It Is Globally Visible for Any Design

# Operator Overloading Function/package

- Packages That Define These Operator Overloading Functions Can Be Found in the **LIBRARY IEEE**
- For Example, the Package ***std\_logic\_unsigned*** Defines Some of the Following Functions

package std\_logic\_unsigned is

```
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;  
function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;  
function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;  
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;  
function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

```
function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;  
function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;  
function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;  
function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;  
function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

# Use of Operator Overloading

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
USE IEEE.std_logic_unsigned.all;
```

*Include These Statements  
At the Beginning of a  
Design File*

**ENTITY** overload **IS**

```
    PORT ( a  : IN STD_LOGIC_VECTOR (4 downto 0);  
          b  : IN STD_LOGIC_VECTOR (4 downto 0);  
          sum : OUT STD_LOGIC_VECTOR (5 downto 0));
```

**END** overload;

**ARCHITECTURE** example **OF** overload **IS**  
**BEGIN**

```
    sum <= a + b;
```

**END** example;

*This Allows Us to Perform  
Arithmetic on Non-built-in  
Data Types*

# Conditional Signal Assignments

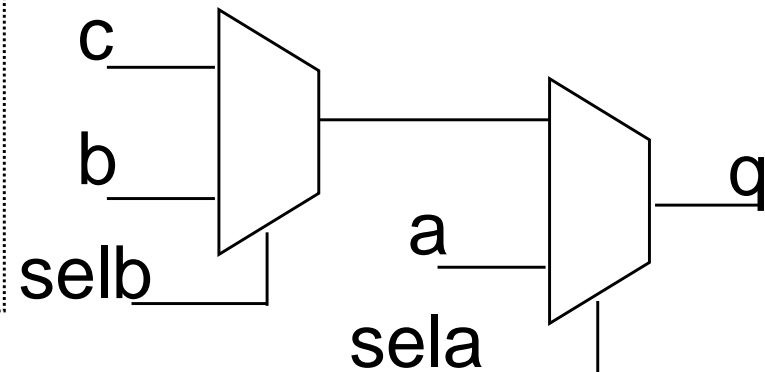
## ■ Format:

```
<signal_name> <= <signal/value> when <condition1> else  
    <signal/value> when <condition2> else  
    .  
    .  
    <signal/value> when <condition3> else  
    <signal/value>;
```

## ■ Example:

```
q <= a WHEN sela = '1' ELSE  
    b WHEN selb = '1' ELSE  
    c;
```

*Implied Process*



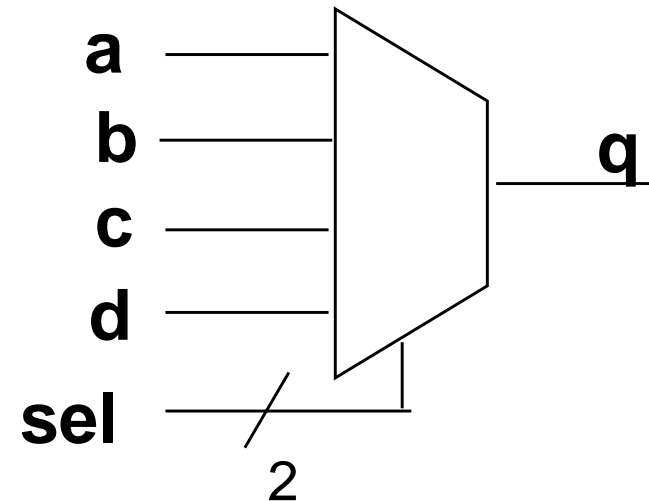
# Selected Signal Assignments

## ■ Format:

```
with <expression> select
<signal_name> <=      <signal/value> when <condition1>,
                        <signal/value> when <condition2>,
                        .
                        .
                        <signal/value> when others;
```

## ■ Example:

```
WITH      sel SELECT
  q <=      a WHEN "00",
            b WHEN "01",
            c WHEN "10",
            d WHEN OTHERS;
```



# Selected Signal Assignments

- **All Possible Conditions Must Be Considered**
- **WHEN OTHERS** Clause Evaluates All Other Possible Conditions That Are Not Specifically Stated

**SEE NEXT SLIDE** 



# Selected Signal Assignment

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;
```

```
ENTITY cmpl_sig IS  
PORT ( a, b, sel : IN STD_LOGIC;  
       z : OUT STD_LOGIC);  
END cmpl_sig;
```

```
ARCHITECTURE logic OF cmpl_sig IS  
BEGIN
```

```
    -- selected signal assignment  
    WITH sel SELECT  
        z <= a WHEN '0',  
          b  WHEN '1',  
          '0' WHEN OTHERS;
```

```
END logic;
```

**sel** Has a **STD\_LOGIC** Data Type

- What are the Values for a **STD\_LOGIC** Data Type
- Answer: {'0','1','X','Z'}
- Therefore, is the **WHEN OTHERS** Clause Necessary?
- Answer: YES

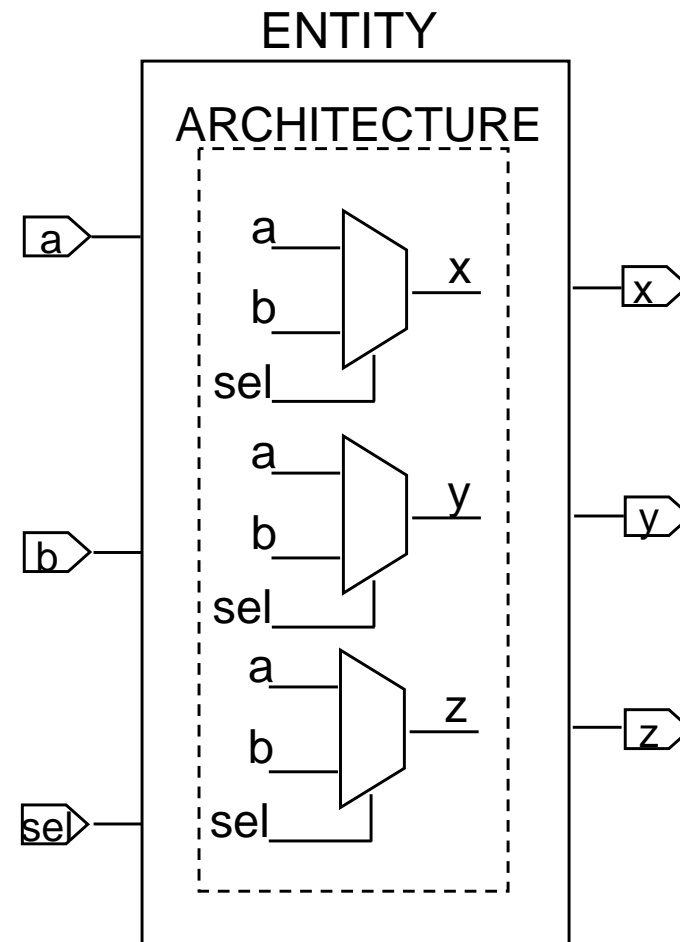
# VHDL Model - Concurrent Signal Assignments

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY cmpl_sig IS
PORT ( a, b, sel : IN STD_LOGIC;
      x, y, z : OUT STD_LOGIC);
END cmpl_sig;

ARCHITECTURE logic OF cmpl_sig IS
BEGIN
    -- simple signal assignment
    x <= (a AND NOT sel) OR (b AND sel);
    -- conditional signal assignment
    y <= a WHEN sel='0' ELSE
        b;
    -- selected signal assignment
    WITH sel SELECT
        z <= a WHEN '0',
            b WHEN '1',
            '0' WHEN OTHERS;
END logic;
```

- The Signal Assignments Execute in Parallel, and Therefore the Order We List the Statements Should Not Affect the Outcome



# Explicit Process Statement

- Process Can Be Thought of As
  - *Implied Processes*
  - *Explicit Processes*
- Implied Process Consist of
  - Concurrent Signal Assignment Statements
  - Component Statements
  - Processes' Sensitivity Is Read (Right) Side of Expression
- Explicit Process
  - Concurrent Statement
  - Consist of Sequential Statements Only

-- Explicit Process Statement

**PROCESS** (*sensitivity\_list*)

Constant Declarations

Type Declarations

Variable Declarations

**BEGIN**

-- Sequential statement #1;

-- .....

-- Sequential statement #N ;

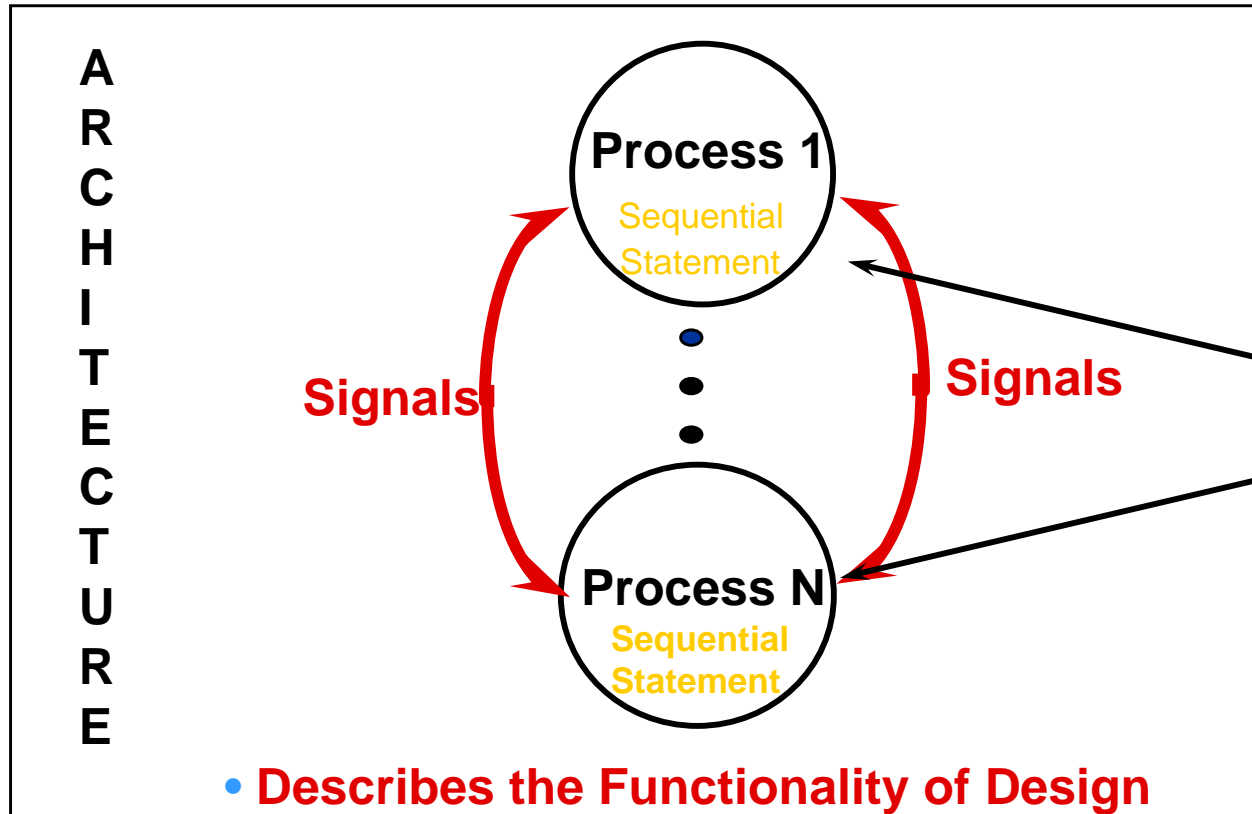
**END PROCESS;**

# Execution of Process Statement

- A process is like a circuit part, which can be
  - active (known activated)
  - inactive (known as suspended).
- A process is activated when a signal in the sensitivity list changes its value
- Its statements will be executed sequentially until the end of the process

```
PROCESS(a,b)  
BEGIN  
    -- sequential statements  
END PROCESS;
```

# Multi-Process Statements



- An Architecture Can Have Multiple Process Statements
- Each Process Executes in Parallel With Each Other
- However, Within a Process, the Statements Are Executed Sequentially

# VHDL Model - Multi-Process Architecture

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;
```

```
ENTITY if_case IS  
PORT ( a, b, c, d : IN STD_LOGIC;  
       sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);  
       y, z : OUT STD_LOGIC);  
END if_case;
```

```
ARCHITECTURE logic OF if_case IS  
BEGIN
```

```
if_label: PROCESS(a, b, c, d, sel)  
BEGIN  
    IF sel="00" THEN  
        y <= a;  
    ELSIF sel="01" THEN  
        y <= b;  
    ELSIF sel="10" THEN  
        y <= c;  
    ELSE  
        y <= d;  
    END IF;  
END PROCESS if_label;
```

- The Process Statements Execute in Parallel and Therefore, the Order in Which We List the Statements Should Have No Affect on the Outcome

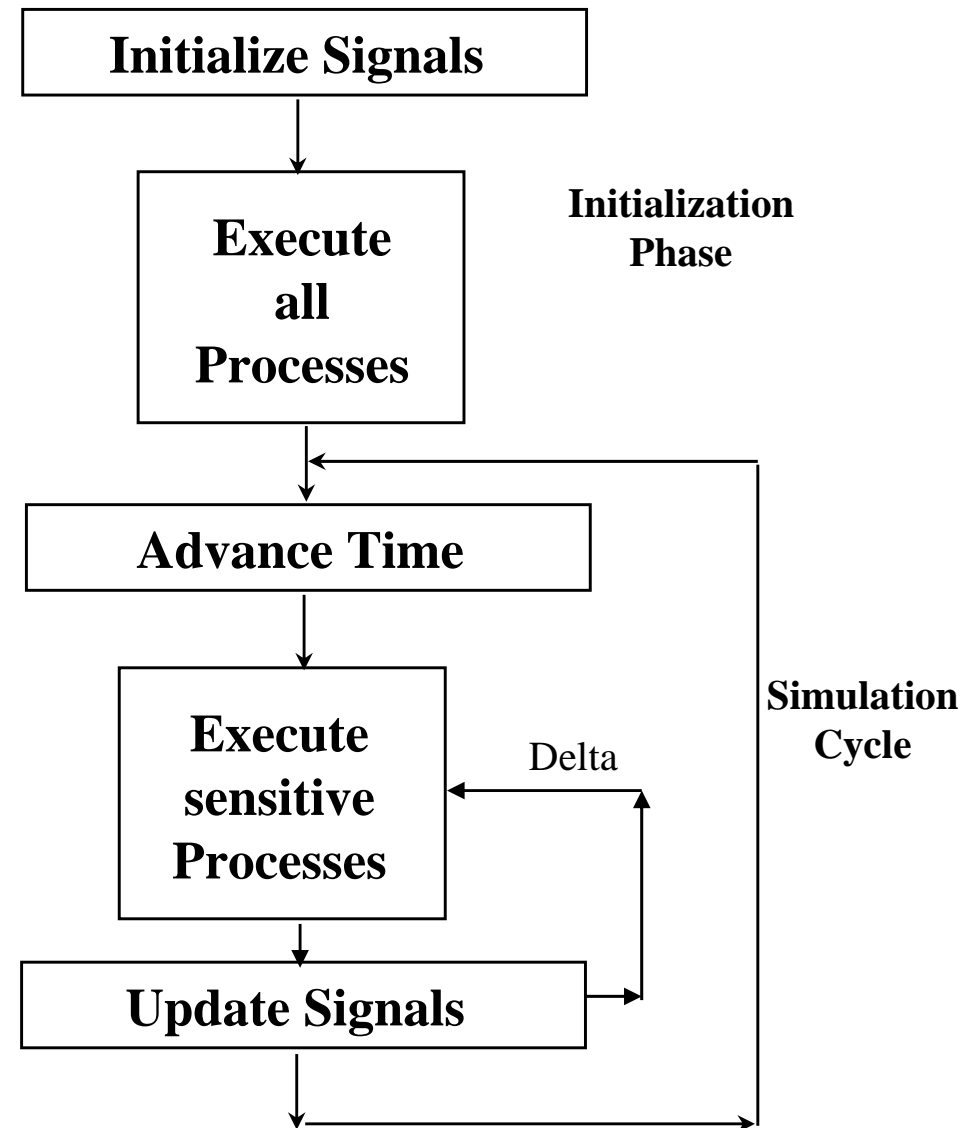
```
case_label: PROCESS(a, b, c, d, sel)  
BEGIN  
    CASE sel IS  
        WHEN "00" =>  
            z <= a;  
        WHEN "01" =>  
            z <= b;  
        WHEN "10" =>  
            z <= c;  
        WHEN "11" =>  
            z <= d;  
        WHEN OTHERS =>  
            z <= '0';  
    END CASE;  
END PROCESS case_label;  
END logic;
```

- Within a Process, the Statements Are Executed Sequentially

- Signal Assignments Can Also Be Inside Process Statements

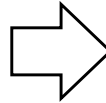
# VHDL Simulation

- Event - A Change in Value: From 0 to 1; Or From X to 1, Etc.
- Simulation Cycle
  - Wall Clock Time
  - Delta
    - Process Execution Phase
    - Signal Update Phase
- When Does a Simulation Cycle End and a New One Begin?
  - ⇒ **When:**
    - **All Processes Execute**
    - **Signals Are Updated**
- Signals Get Updated at the End of the Process



# Equivalent Functions

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp IS  
PORT(a, b : IN STD_LOGIC;  
      y : OUT STD_LOGIC);  
END simp;  
ARCHITECTURE logic OF simp IS  
SIGNAL c : STD_LOGIC;  
  
BEGIN  
  
c <= a and b;  
y <= c;  
  
END logic;
```



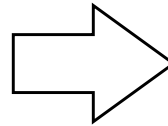
```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp_prc IS  
PORT(a,b : IN STD_LOGIC;  
      y : OUT STD_LOGIC);  
END simp_prc;  
ARCHITECTURE logic OF simp_prc IS  
SIGNAL c : STD_LOGIC;  
  
BEGIN  
process1: PROCESS(a, b)  
  BEGIN  
    c <= a and b;  
  END PROCESS process1;  
process2: PROCESS(c)  
  BEGIN  
    y <= c;  
  END PROCESS process2;  
END logic;
```

- **c** and **y** Get Executed and Updated in Parallel at the End of the Process Within One Simulation Cycle



# Equivalent Functions?

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp IS  
PORT(a, b : IN STD_LOGIC;  
      y : OUT STD_LOGIC);  
END simp;  
ARCHITECTURE logic OF simp IS  
SIGNAL c : STD_LOGIC;  
BEGIN  
c <= a and b;  
y <= c;  
END logic;
```

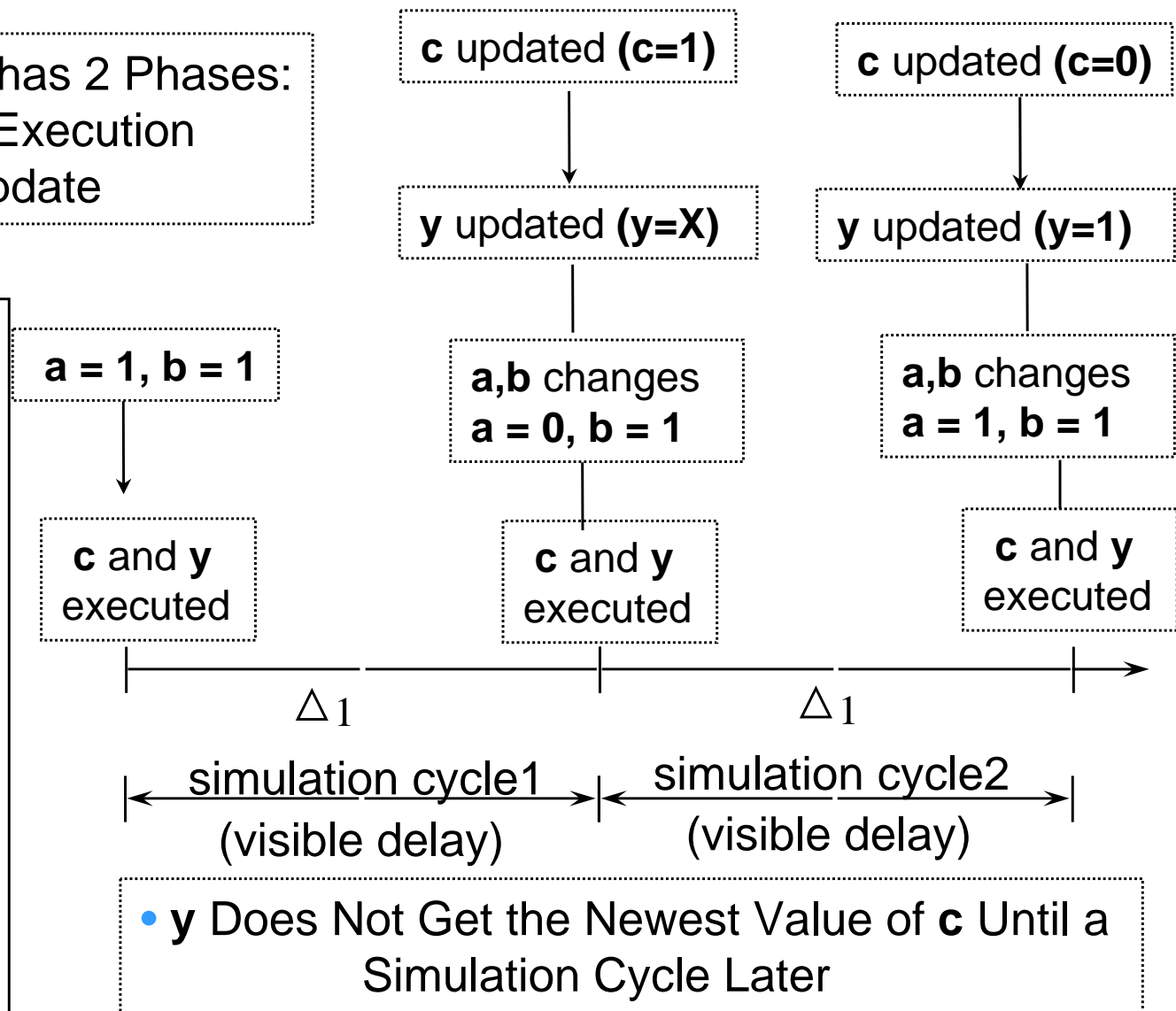


```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp_prc IS  
PORT(a, b : IN STD_LOGIC;  
      y: OUT STD_LOGIC);  
END simp_prc;  
ARCHITECTURE logic OF simp_prc IS  
SIGNAL c: STD_LOGIC;  
  
BEGIN  
PROCESS(a, b)  
BEGIN  
c <= a and b;  
y <= c;  
END PROCESS;  
END logic;
```

# Signal Assignment Inside a Process - Delay

- $\Delta$  Delta Cycle has 2 Phases:
  - Process Execution
  - Signal Update

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
ENTITY simp_prc IS  
  PORT(a, b : IN STD_LOGIC;  
        y : OUT STD_LOGIC);  
END simp_prc;  
ARCHITECTURE logic OF simp_prc IS  
  SIGNAL c: STD_LOGIC;  
  
  BEGIN  
    PROCESS(a, b)  
      BEGIN  
        c <= a and b;  
        y <= c;  
      END PROCESS;  
    END logic;
```



- $\Delta$  Delta Cycle is Non-Visible Delay (Very Small, Close to Zero)



## 2 Processes

Vs.

## 1 Process

process1: PROCESS(a, b)

BEGIN

c <= a and b;

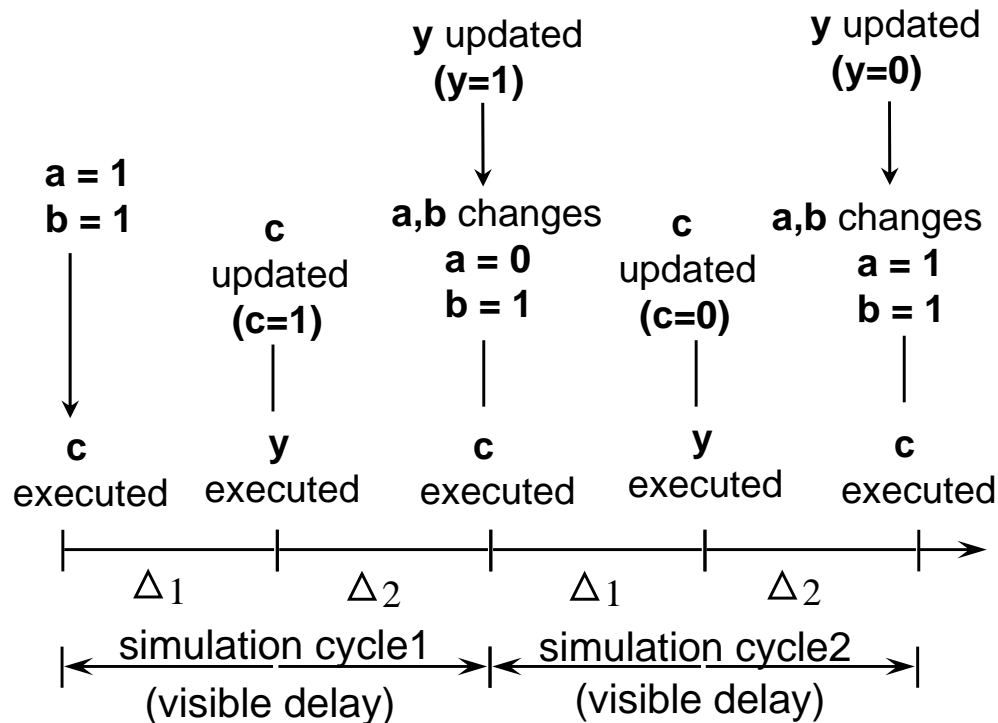
END PROCESS process1;

process2: PROCESS(c)

BEGIN

y <= c;

END PROCESS process2;



- **c** and **y** Gets Executed and Updated Within the Same Simulation Cycle

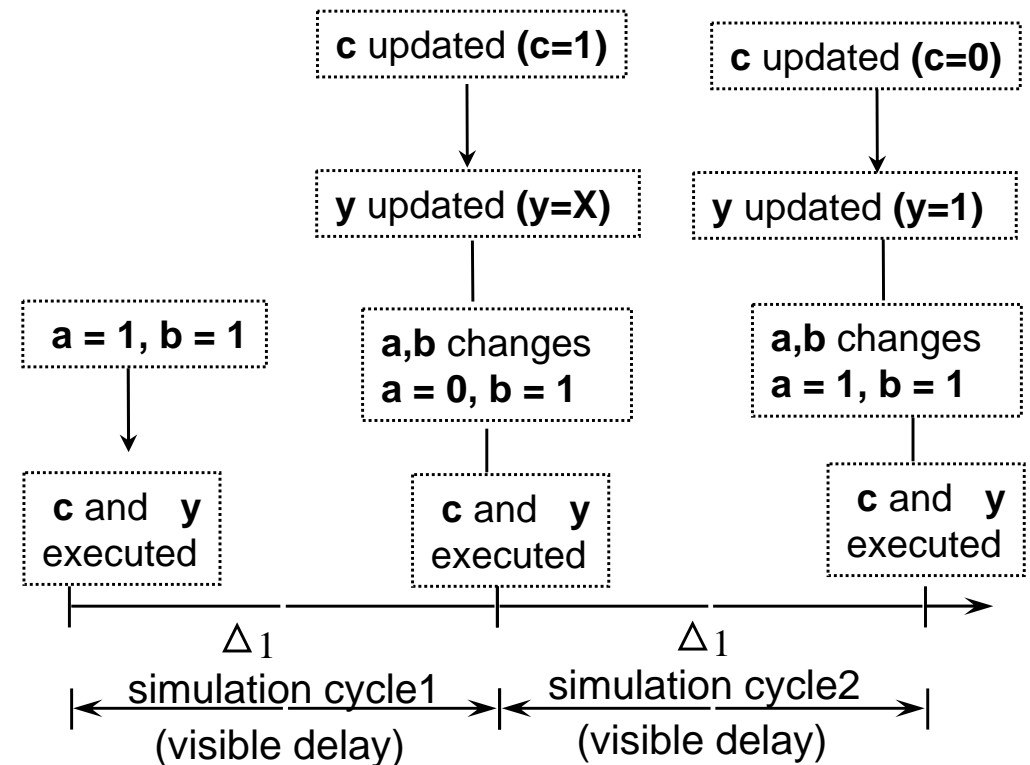
PROCESS(a, b)

BEGIN

c <= a and b;

y <= c;

END PROCESS;



- **y** Does Not Get the Newest Value of **c** Until a Simulation Cycle Later

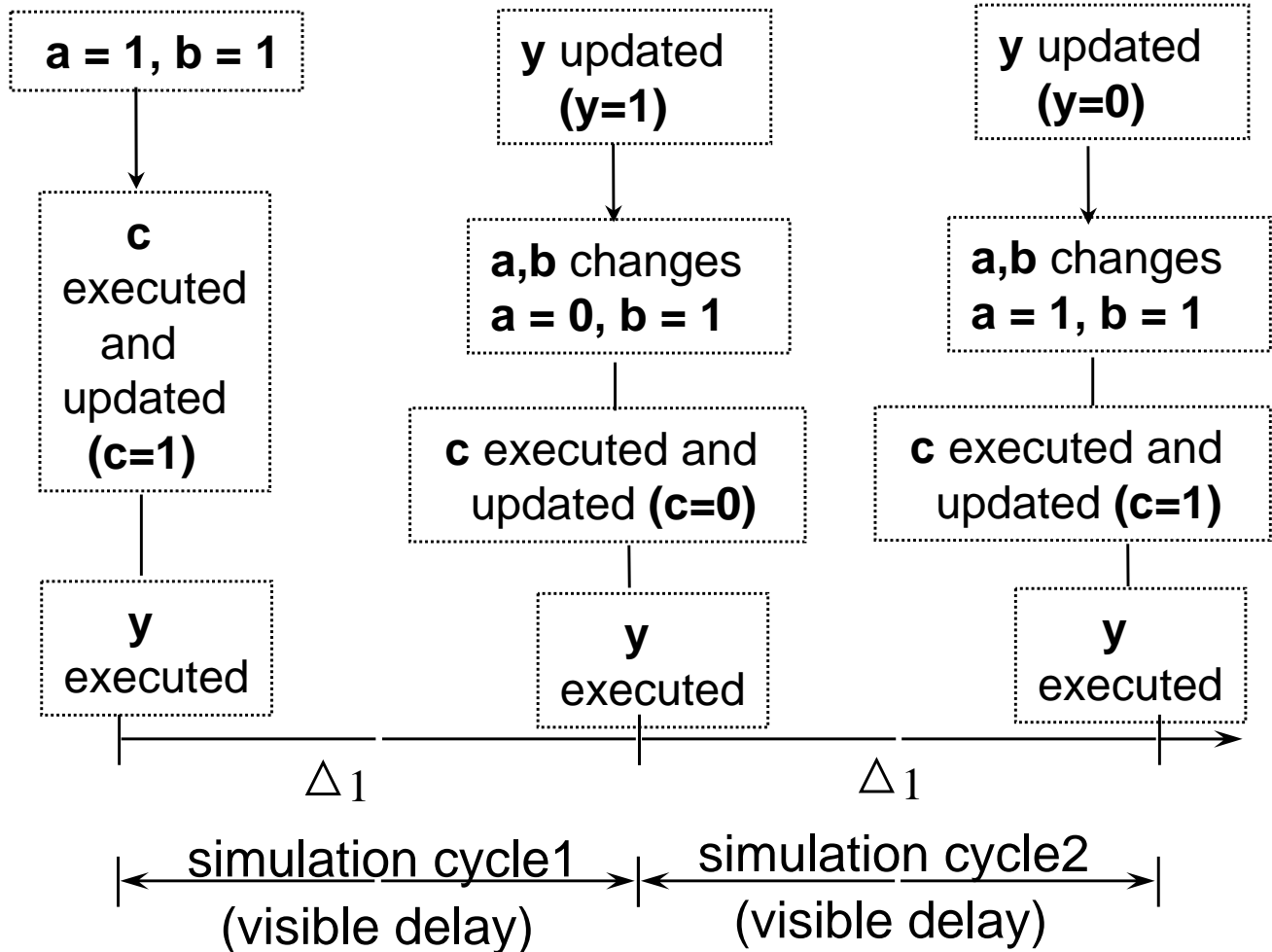
# Variable Assignment - No Delay

- $\Delta$  Delta Cycle has 2 Phases:
  - Process Execution
  - Signal Update

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY var IS
PORT    (a, b : IN  STD_LOGIC;
         y : OUT STD_LOGIC);
END var;
ARCHITECTURE logic OF var IS
BEGIN
PROCESS (a, b)
    VARIABLE c : STD_LOGIC;;
BEGIN
    c := a AND b;
    y <= c;

END PROCESS;
END logic;
    
```



- $c$  and  $y$  Gets Executed and Updated Within the Same Simulation Cycle (at the End of the Process)

- $\Delta$  Delta Cycle is Non-Visible Delay (Very Small, Close to Zero)



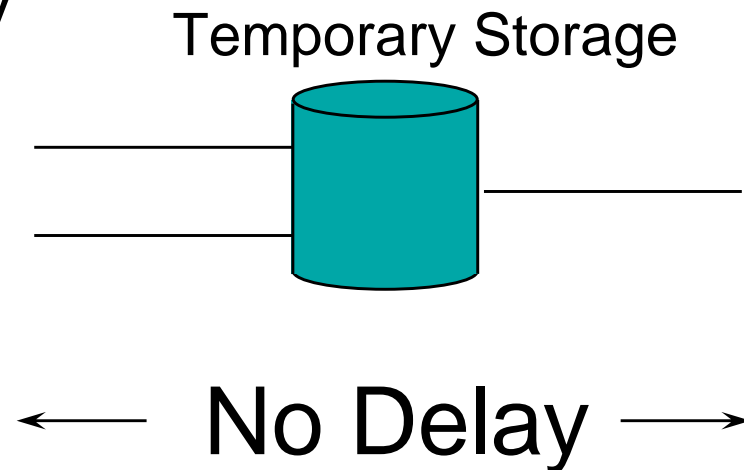
# Variable Declarations

- Variables Are Declared Inside a Process
- Variables Are Represented By: `:=`
- Variable Declaration

**VARIABLE** *<Name>* : *<DATA\_TYPE>* `:=` *<Value>*;

**Variable** Temp : **Std\_logic\_vector (7 Downto 0);**

- Variable Assignments Are Updated Immediately
  - Do Not Incur a Delay



# Assigning Values to Variables

**VARIABLE temp : STD\_LOGIC\_VECTOR (7 downto 0);**

- All Bits:

Temp := "10101010";

Temp := X"aa" ; (1076-1993)

- Single Bit:

Temp(7) := '1';

- Bit-slicing:

Temp (7 downto 4) := "1010";

- Single-bit: Single-quote (')

- Multi-bit: Double-quote (")

# Variable Assignment

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;
```

```
ENTITY var IS  
PORT    (a, b : IN  STD_LOGIC;  
          y : OUT STD_LOGIC);  
END var;
```

```
ARCHITECTURE logic OF var IS  
BEGIN
```

```
PROCESS (a, b)
```

```
VARIABLE c : STD_LOGIC;
```

```
BEGIN
```

```
c := a AND b;
```

```
y <= c;
```

```
END PROCESS;
```

```
END logic;
```

*Variable Declaration*

*Variable Assignment*

**Variable** is Assigned to a  
**Signal** to Synthesize to a  
Piece of Hardware

# Use of a Variable

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY cmb_var IS
PORT(i0, i1, a : IN BIT;
      q : OUT BIT);
END cmb_var;
ARCHITECTURE logic OF cmb_var IS
BEGIN
    PROCESS(i0, i1, a)
        VARIABLE val : INTEGER RANGE 0 TO 1;
    BEGIN
        val := 0;
        IF (a = '0') THEN
            val := val;
        ELSE
            val := val + 1;
        END IF;
        CASE val IS
            WHEN 0 =>
                q <= i0;

            WHEN 1 =>
                q <= i1;

        END CASE;
    END PROCESS;
END logic;
```

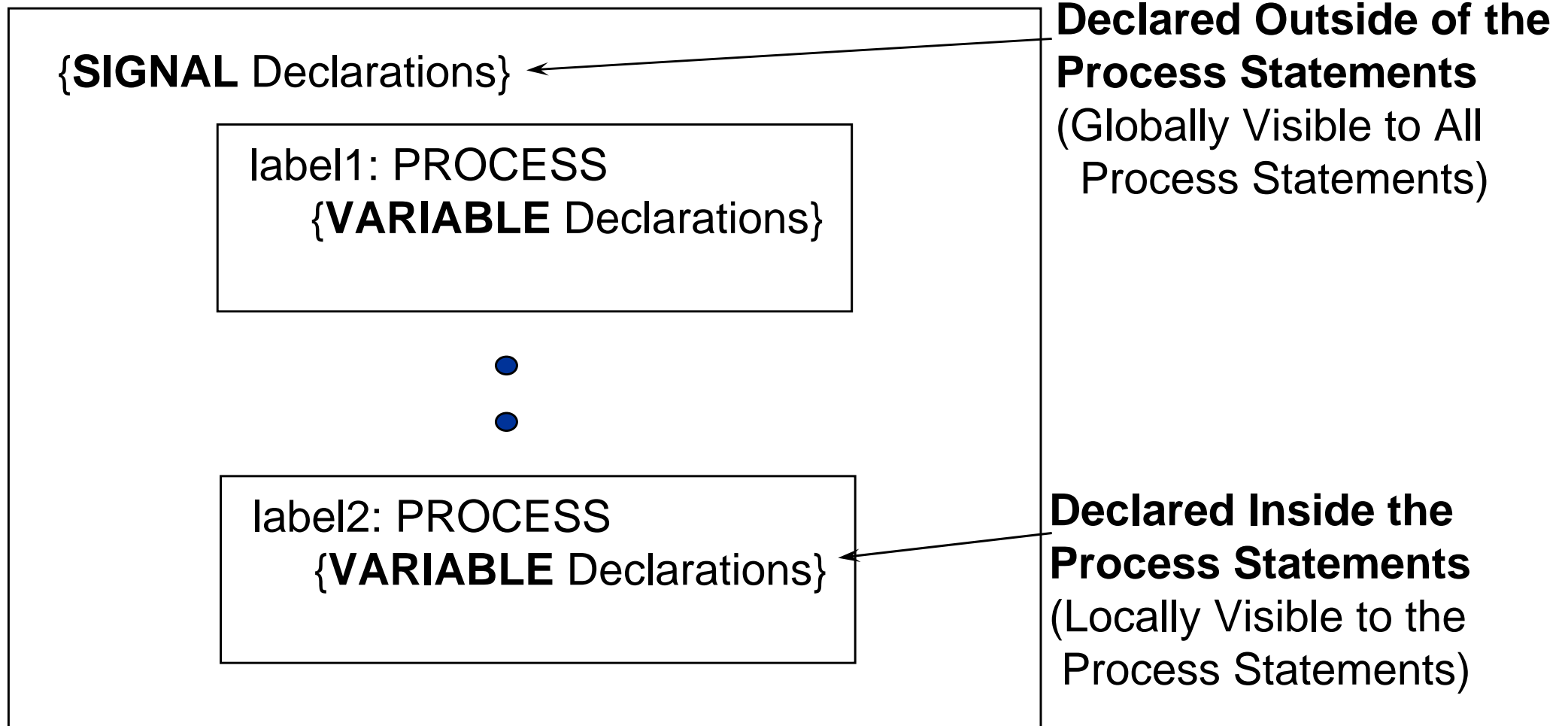
**val** Is a Variable That Is Updated at the Instant an Assignment Is Made to It

Therefore, the Updated Value of **val** Is Available for the CASE Statement



# Signal and Variable Scope

## ARCHITECTURE



# Review - Signals vs. Variables

	<b>SIGNALS ( &lt;= )</b>	<b>VARIABLES ( := )</b>
<b>ASSIGN</b>	assignee <= assignment	assignee := assignment
<b>UTILITY</b>	<b>Represent Circuit Interconnect</b>	<b>Represent Local Storage</b>
<b>SCOPE</b>	<b>Global Scope (Communicate Between PROCESSES)</b>	<b>Local Scope (Inside PROCESS)</b>
<b>BEHAVIOR</b>	<b>Updated at End of Process Statement (New Value Not Available)</b>	<b>Updated Immediately (New Value Available)</b>

# Sequential Statements

- Sequential Statements
  - IF-THEN Statement
  - CASE Statement

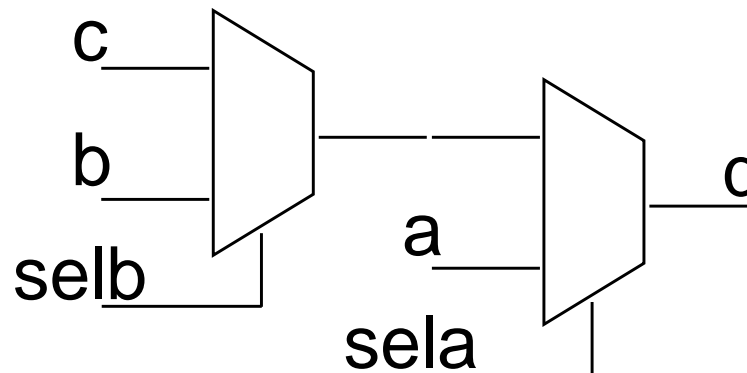
# If-then Statements

## ■ Format:

```
IF <condition1> THEN
    {sequence of statement(s)}
ELSIF <condition2> THEN
    {sequence of statement(s)}
    .
    .
ELSE
    {sequence of statement(s)}
END IF;
```

## ■ Example:

```
PROCESS(sela, selb, a, b, c)
BEGIN
    IF sela='1' THEN
        q <= a;
    ELSIF selb='1' THEN
        q <= b;
    ELSE
        q <= c;
    END IF;
END PROCESS;
```



# If-then Statements

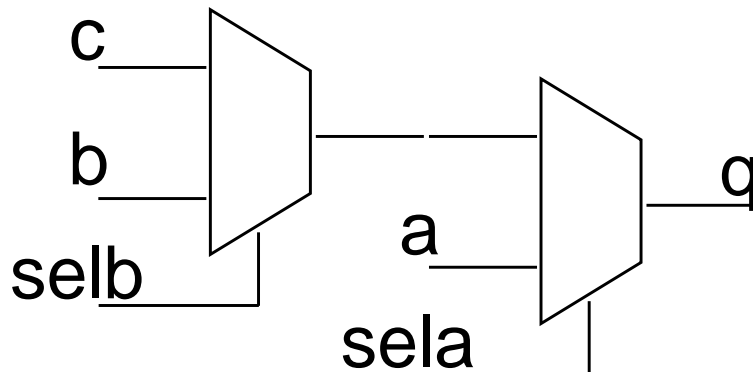
- Conditions Are Evaluated in Order From Top to Bottom
  - Prioritization
- The First Condition That Is True Causes the Corresponding Sequence of Statements to Be Executed
- If All Conditions Are False, Then the Sequence of Statements Associated With the “ELSE” Clause Is Evaluated

# If-then Statements

## ■ Similar to Conditional Signal Assignment

### Implied Process

```
q <= a WHEN sela = '1' ELSE  
  b WHEN selb = '1' ELSE  
  c;
```



### Explicit Process

```
PROCESS(sela, selb, a, b, c)  
BEGIN  
  IF sela='1' THEN  
    q <= a;  
  ELSIF selb='1' THEN  
    q <= b;  
  ELSE  
    q <= c;  
  END IF;  
END PROCESS;
```

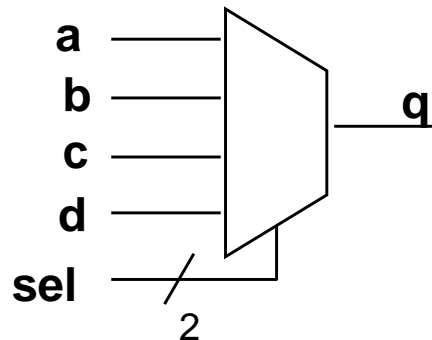
# Case Statement

## ■ Format:

```
CASE {expression} IS  
    WHEN <condition1> =>  
        {sequence of statements}  
    WHEN <condition2> =>  
        {sequence of statements}  
        .  
        .  
    WHEN OTHERS => -- (optional)  
        {sequence of statements}  
END CASE;
```

## ■ Example:

```
PROCESS(sel, a, b, c, d)  
BEGIN  
    CASE sel IS  
        WHEN "00" =>  
            q <= a;  
        WHEN "01" =>  
            q <= b;  
        WHEN "10" =>  
            q <= c;  
        WHEN OTHERS =>  
            q <= d;  
    END CASE;  
END PROCESS;
```



# Case Statement

- Conditions Are Evaluated at Once
  - No Prioritization
- **All** Possible Conditions Must Be Considered
- **WHEN OTHERS** Clause Evaluates All Other Possible Conditions That Are Not Specifically Stated

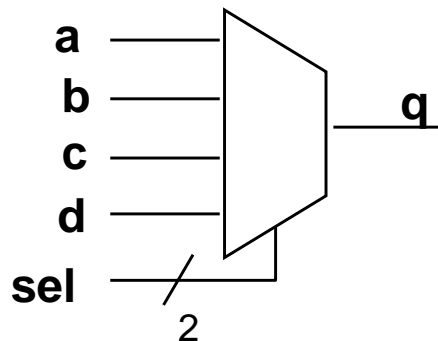


# Case Statement

## ■ Similar to Selected Signal Assignment

### Implied Process

```
WITH sel SELECT
    q <= a WHEN "00",
      b WHEN "01",
      c WHEN "10",
      d WHEN OTHERS;
```



### Explicit Process

```
PROCESS(sel, a, b, c, d)
BEGIN
    CASE sel IS
        WHEN "00" =>
            q <= a;
        WHEN "01" =>
            q <= b;
        WHEN "10" =>
            q <= c;
        WHEN OTHERS =>
            q <= d;
    END CASE;
END PROCESS;
```

# **Understanding VHDL and Logic Synthesis**

# Two Types of Process Statements

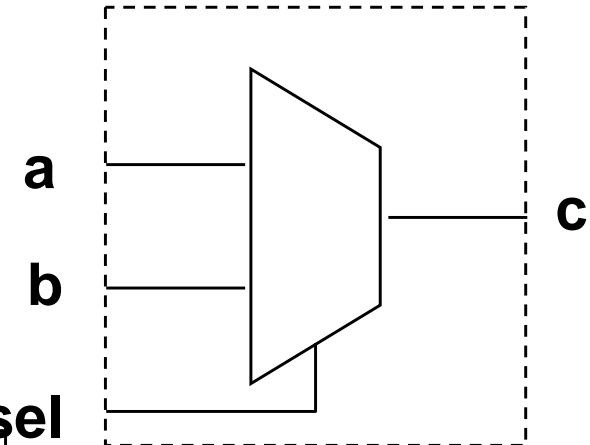
- **Combinatorial Process**

- Sensitive to All Inputs Used In the Combinatorial Logic

- **Example**

```
PROCESS(a, b, sel)
```

*Sensitivity List Includes All Inputs Used in the Combinatorial Logic*



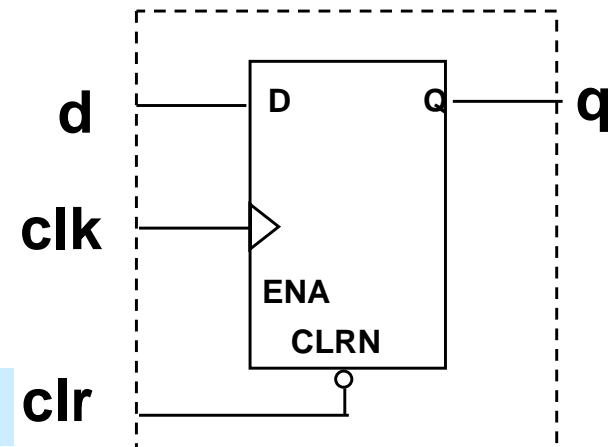
- **Sequential Process**

- Sensitive to a Clock or/and Control Signals

- **Example**

```
PROCESS(clr, clk)
```

*Sensitivity List Does Not Include the d Input, Only the Clock or/and Control Signals*

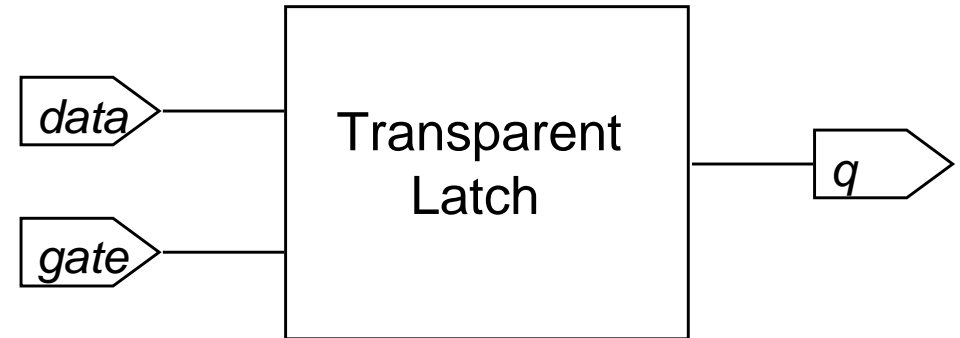


# Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY latch1 IS
PORT ( data : IN std_logic;
      gate : IN std_logic;
      q : OUT std_logic
    );
END latch1;

ARCHITECTURE behavior OF latch1 IS
BEGIN
  label_1: PROCESS (data, gate)
  BEGIN
    IF gate = '1' THEN
      q <= data;
    END IF;
  END PROCESS;
END behavior;
```



*Sensitivity List Includes Both Inputs*

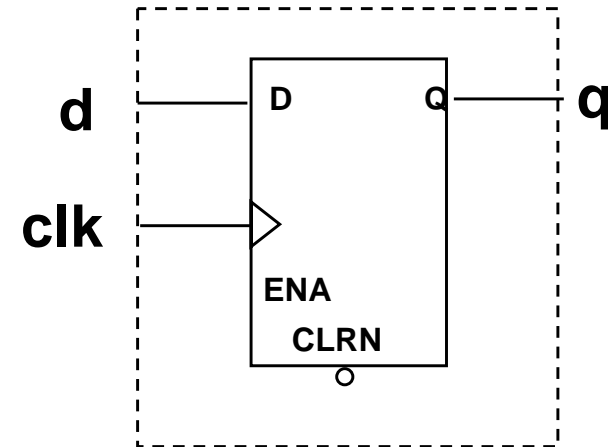
*What Happens if Gate = '0'?*  
⇒ **Implicit Memory**

# DFF - Clk'event and Clk='1'

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY dff_a IS
PORT ( d : in std_logic;
      clk : in std_logic;
      q : out std_logic
      );
END dff_a;

ARCHITECTURE behavior OF dff_a IS
BEGIN
PROCESS (clk)
BEGIN
    IF clk'event and clk = '1' THEN
        q <= d;
    END IF;
END PROCESS;
END behavior;
```



## ***clk'event and clk='1'***

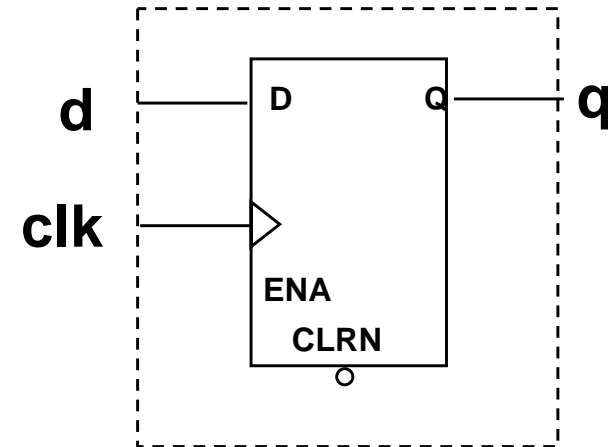
- ***clk*** Is the Signal Name (Any Name)
- ***'event*** Is a VHDL Attribute, Specifying That There Needs to Be a Change in Signal Value
- ***clk='1'*** Means Positive-Edge Triggered

# DFF - Rising\_edge

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY dff_b IS
PORT ( d : in std_logic;
      clk : in std_logic;
      q : out std_logic
      );
END dff_b;

ARCHITECTURE behavior OF dff_b IS
BEGIN
PROCESS(clk)
BEGIN
  IF rising_edge(clk) THEN
    q <= d;
  END IF;
END PROCESS;
END behavior;
```



## ***rising\_edge***

- *IEEE Function That is Defined in the std\_logic\_1164 Package*
- *Specifies That the Signal Value **must** be 0 to 1*
- *X, Z to 1 Transition Is Not Allowed*

# Rising\_edge vs Wait clk'event

- There is a small difference between them.

please see the snippet from the library  
std\_logic\_1164

- FUNCTION rising\_edge (SIGNAL s : std\_ulogic) RETURN BOOLEAN IS  
BEGIN

```
    RETURN (s'EVENT AND (To_X01(s) = '1') AND  
            (To_X01(s'LAST_VALUE) = '0'));
```

```
END;
```

- FUNCTION falling\_edge (SIGNAL s : std\_ulogic) RETURN BOOLEAN IS  
BEGIN

```
    RETURN (s'EVENT AND (To_X01(s) = '0') AND  
            (To_X01(s'LAST_VALUE) = '1'));
```

```
END;
```

- This function checks the rising or falling edge of the clock as well as the previous value of the clock.
- When you write (clk'event and clk='1') you check only the rising or falling edge not the previous value of the signal.

# DFF With Asynchronous Clear

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
USE IEEE.std_logic_unsigned.all;
```

```
ENTITY dff_clr IS  
  PORT ( clr : in bit;  
        d, clk : in std_logic;  
        q : out std_logic  
        );  
END dff_clr;
```

```
ARCHITECTURE behavior OF dff_clr IS  
  BEGIN
```

```
    PROCESS(clk, clr)  
      BEGIN
```

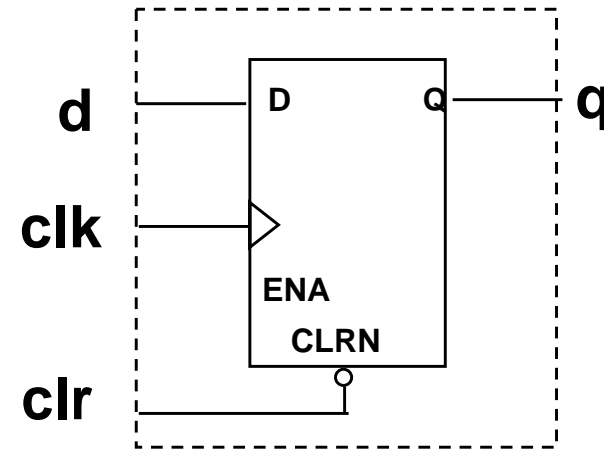
```
        IF clr = '0' THEN  
          q <= '0';
```

```
        ELSIF rising_edge(clk) THEN  
          q <= d;
```

```
        END IF;
```

```
    END PROCESS;
```

```
  END behavior;
```



- *This is How to Implement Asynchronous Control Signals for the Register*
- *Note: This IF-THEN Statement Is Outside the IF-THEN Statement that Checks the Condition **rising\_edge***
- *Therefore, **clr='1'** Does Not Depend on the Clock*



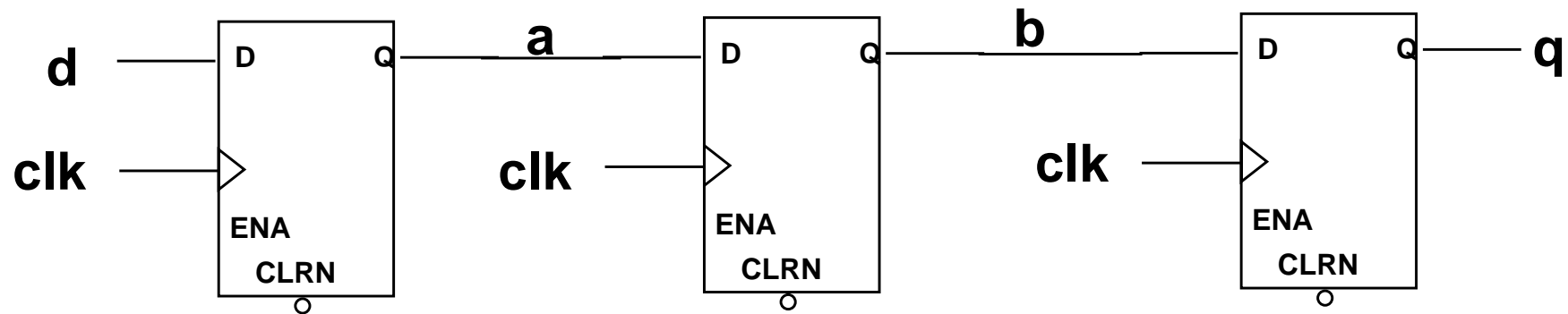
# How Many Registers?

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY reg1 IS
    PORT ( d      : in STD_LOGIC;
          clk     : in STD_LOGIC;
          q       : out STD_LOGIC);
END reg1;

ARCHITECTURE reg1 OF reg1 IS
    SIGNAL a, b : STD_LOGIC;
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            a <= d;
            b <= a;
            q <= b;
        END IF;
    END PROCESS;
END reg1;
```

# How Many Registers?

- Signal Assignments Inside the IF-THEN Statement That Checks the Clock Condition Infer Registers



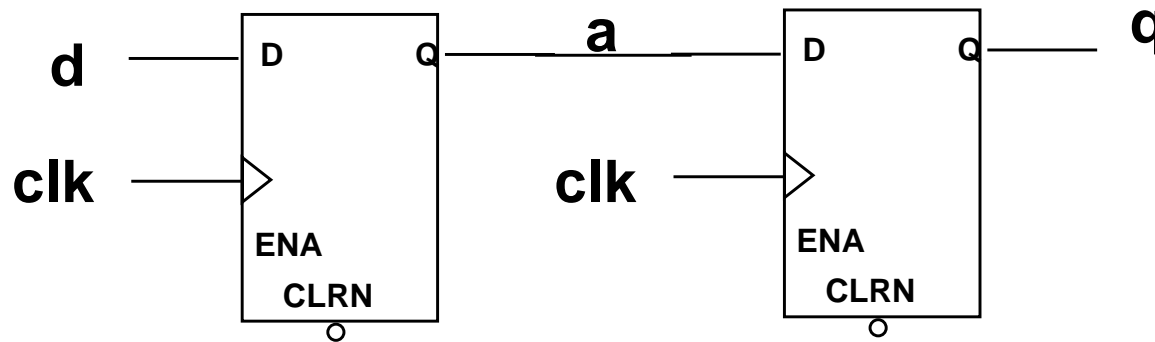
# How Many Registers?

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY reg1 IS
    PORT ( d      : in STD_LOGIC;
          clk     : in STD_LOGIC;
          q      : out STD_LOGIC);
END reg1;
ARCHITECTURE reg1 OF reg1 IS
    SIGNAL a, b : STD_LOGIC;
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            a <= d;
            b <= a;
        END IF;
        END PROCESS;
        q <= b;
    END reg1;
```

*Signal  
Assignment  
Moved*

# How Many Registers?

- B to Q Assignment Is No Longer Edge-sensitive Because It Is Not Inside the If-then Statement That Checks the Clock Condition



# How Many Registers?

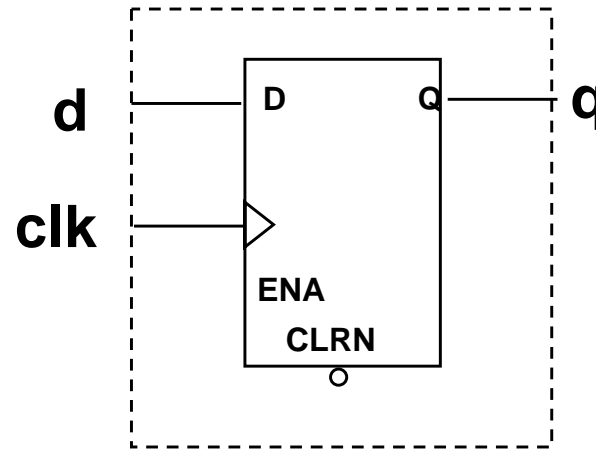
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY reg1 IS
    PORT ( d      : in STD_LOGIC;
          clk     : in STD_LOGIC;
          q      : out STD_LOGIC);
END reg1;

ARCHITECTURE reg1 OF reg1 IS
BEGIN
    PROCESS (clk)
        VARIABLE a, b : STD_LOGIC;
    BEGIN
        IF rising_edge(clk) THEN
            a := d;
            b := a;
            q <= b;
        END IF;
    END PROCESS;
END reg1;
```

*Signals Changed to Variables*

# How Many Registers?

- Variable Assignments Are Updated Immediately
- Signal Assignments Are Updated on Clock Edge



# Variable Assignments in Sequential Logic

- Variable Assignments Inside the IF-THEN Statement, That Checks the Clock Condition, Usually Don't Infer Registers
  - Exception: If the Variable Is on the Right Side of the Equation in a Clocked Process Prior to Being Assigned a Value, the Variable Will Infer a Register(s)
- Variable Assignments Are Temporary Storage and Have No Hardware Intent
- Variable Assignments Can Be Used in Expressions to Immediately Update a Value
  - Then the Variable Can Be Assigned to a Signal

# Example - Counter Using a Variable

- Counters Are Accumulators That Always Add a '1' or Subtract a '1'
- This Example Takes 17 LEs

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY count_a IS
PORT (clk, rst, updn : in std_logic;
      q : out std_logic_vector(15 downto 0));
END count_a;
ARCHITECTURE logic OF count_a IS
BEGIN
PROCESS(rst, clk)
VARIABLE tmp_q : std_logic_vector(15 downto 0);
BEGIN
    IF rst = '0' THEN
        tmp_q := (others => '0');
    ELSIF rising_edge(clk) THEN
        IF updn = '1' THEN
            tmp_q := tmp_q + 1;
        ELSE
            tmp_q := tmp_q - 1;
        END IF;
    END IF;
    q <= tmp_q;
END PROCESS;
END logic;
```

*Arithmetic Expression Assigned to a Variable*

*Variable Assigned to a Signal*

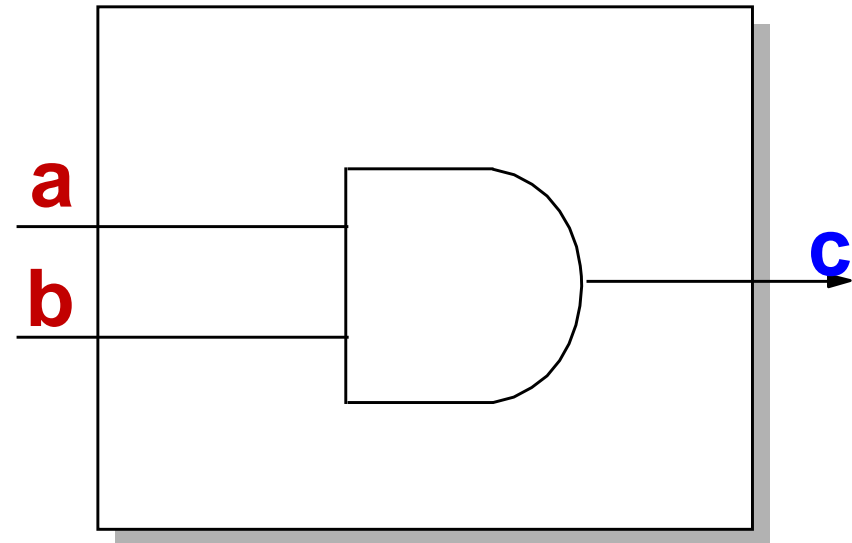


# Implicit memory

- Signals in VHDL have a current state and a future value
- In a process, if the future value of a signal cannot be determined, a latch will be synthesized to preserve its current state
- Advantages:
  - Simplifies the creation of memory in logic design
- Disadvantages:
  - Can generate unwanted latches, e.g., when all of the options in a conditional sequential statement are not specified

# Implicit memory: Example of complete specification

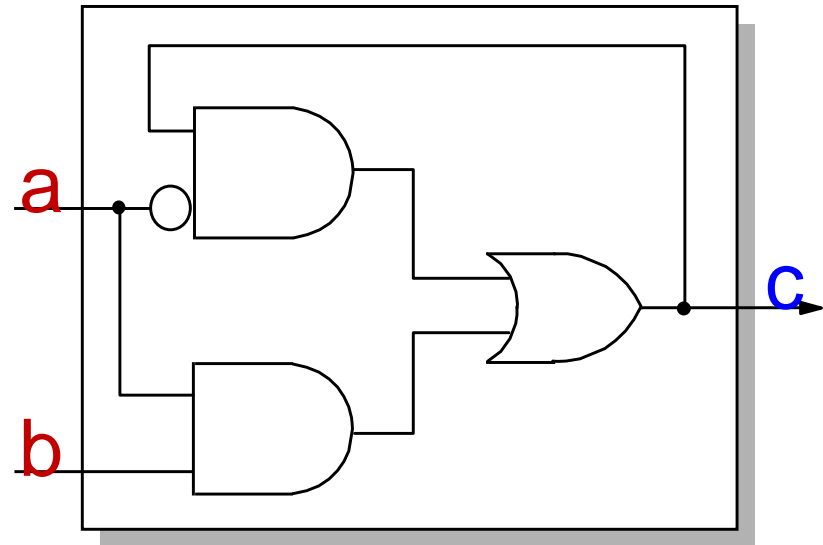
```
ARCHITECTURE archcomplete OF  
  complete IS  
BEGIN  
  PROCESS (a, b)  
  BEGIN  
    IF a = '1' THEN c <= b;  
    ELSE c <= '0';  
    END IF;  
  END PROCESS;  
END archcomplete;
```



- The conditional statement is fully specified, and this causes the process to synthesize to a single gate

# Implicit memory: Example of incomplete specification

```
ARCHITECTURE archincomplete OF
    incomplete IS
BEGIN
    PROCESS (a, b)
    BEGIN
        IF a = '1' THEN c <= b;
        END IF;
    END PROCESS;
END archincomplete;
```



- Here, the incomplete specification of the IF...THEN... statement causes a latch to be synthesized to store the previous state of 'c'

# The rules to avoid implicit memory

## ■ To avoid the generation of unexpected latches

- Always terminate an IF...THEN...ELSE... statement with an ELSE clause
- Cover all alternatives in a CASE statement
  - define every alternative individually, or
  - terminate the CASE statement with a WHEN OTHERS... clause, e.g.,

```
CASE decode IS
```

```
    WHEN b"100" => key <= first;
```

```
    WHEN b"010" => key <= second;
```

```
    WHEN b"001" => key <= third;
```

```
    WHEN OTHERS => key <= none;
```

```
END CASE;
```

# Model Application

# State machines

## ■ Moore Machines

- A finite state machine in which the outputs change due to a change of state

## ■ Mealy Machines

- A finite state machine in which the outputs can change asynchronously i.e., an input can cause an output to change immediately

# Enumerated Data Type

- Recall the Built-in Data Types:
  - Bit
  - Std\_logic
  - Integer
- What About User-defined Data Types?:
  - Enumerated Data Type:

**TYPE** *<your\_data\_type>* **IS**  
*(items or values for your data type separated by commas)*

# Writing VHDL Code for FSM

- State Machine States Must Be an Enumerated Data Type:

```
TYPE State_type IS (Idle, Tap1, Tap2, Tap3, Tap4 );
```

- Object Which Stores the Value of the Current State Must Be a ***Signal*** of the User-defined Type:

```
SIGNAL Filter : State_type;
```



# Writing VHDL Code for FSM

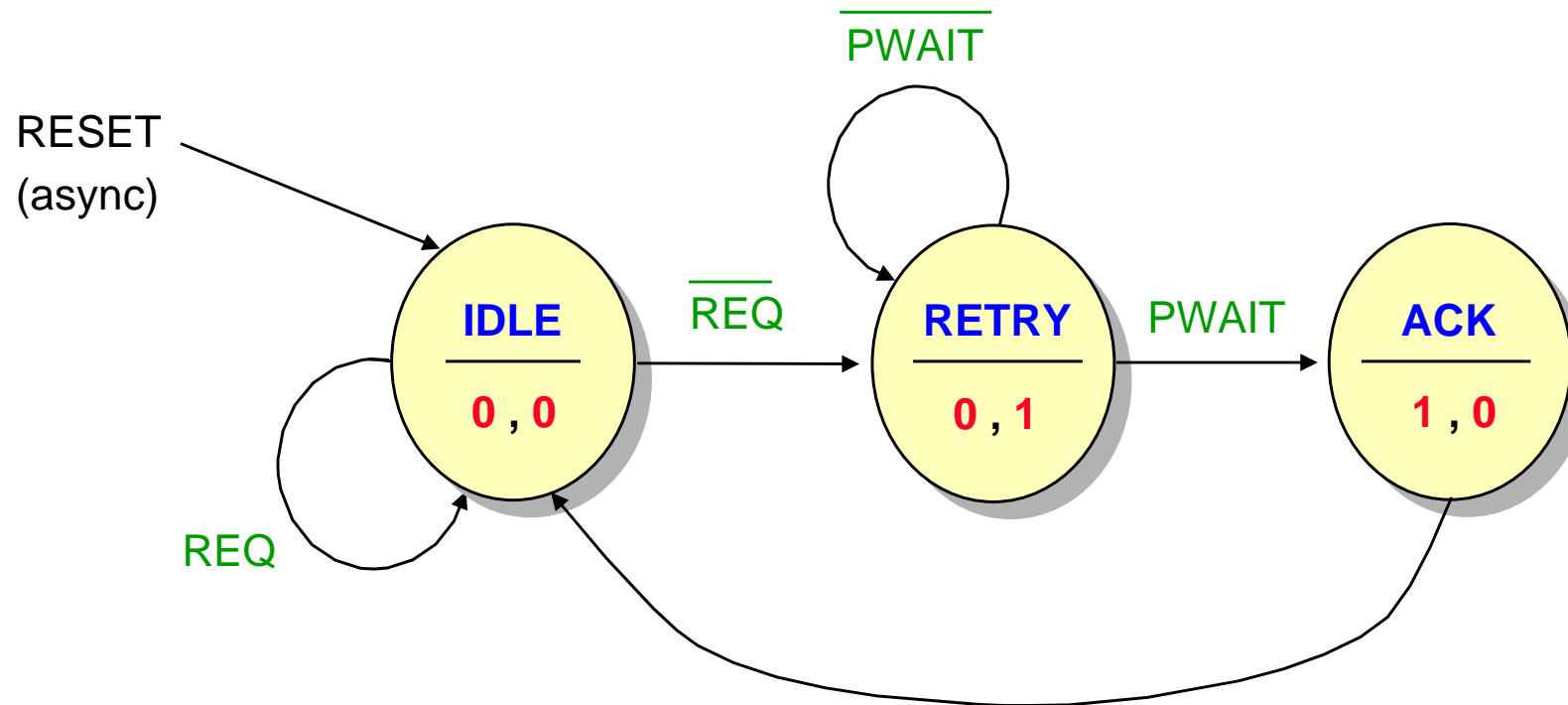
- One process only
  - Handles both state transitions and outputs
- Two processes
  - A synchronous process for updating the state register
  - A combinational process for conditionally deriving the next machine state and updating the outputs

# Moore machines

- Outputs may change only with a change of state
  - Automatic State Assignment
    - the compiler chooses the state encoding
    - outputs must be decoded from the state registers
      - can be a combinatorial decode
      - can be a registered decode
  - Specific State Assignment
    - you choose the state encoding
      - outputs may be encoded inside the state registers

# Example: A wait state generator

## ■ State diagram:



- Inputs: REQ, PWAIT + CLOCK + RESET
- Outputs: ACK\_OUT, RETRY\_OUT
- States: IDLE, RETRY, ACK

# Example: The entity declaration

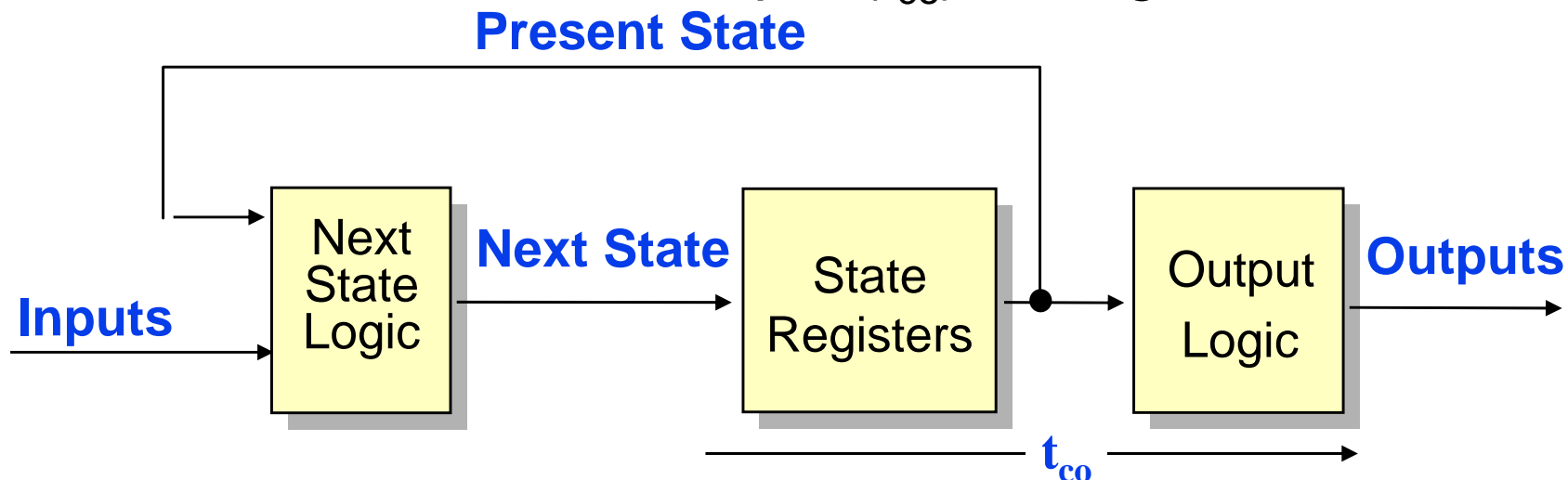
- The entity declaration remains the same for each of the following example implementations (except for the entity name)

e.g.,

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY moore1 IS PORT (  
    clock, reset: IN std_logic;  
    req, pwait: IN std_logic;  
    retry_out, ack_out: OUT std_logic);  
END moore1;
```

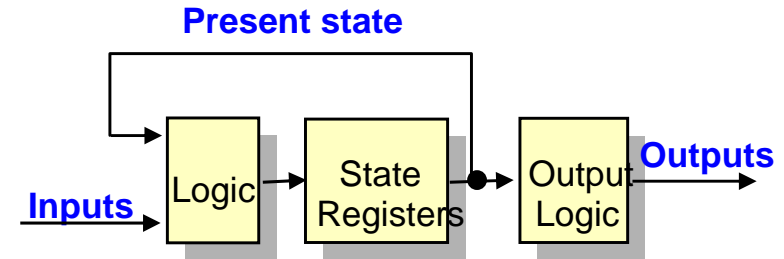
# Moore state machine implementations (1)

- Automatic State Assignment
- Outputs decoded from state bits COMBINATORIALLY
  - ❑ combinatorial output logic is ***in series*** with state registers
  - ❑ outputs are a function of the present state only
  - ❑ time from clock to output ( $t_{co}$ ) is long



# Example: Solution 1

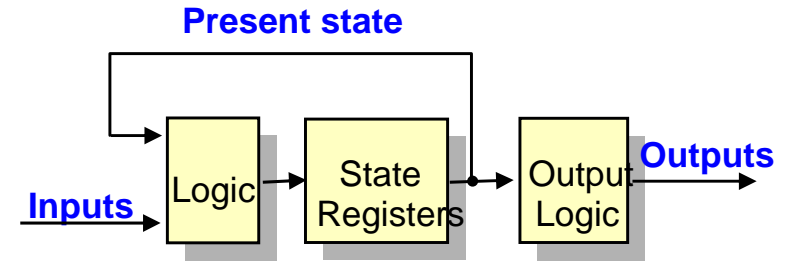
## ■ Combinatorial outputs decoded from the state bits



```
ARCHITECTURE archmoore1 OF moore1 IS
    TYPE fsm_states IS (idle, retry, ack);
    SIGNAL pres_state : fsm_states;
BEGIN
    fsm: PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            pres_state <= idle; -- asynchronous reset
        ELSIF clock'EVENT AND clock = '1' THEN
            CASE pres_state IS
                WHEN idle => IF req = '0' THEN pres_state <= retry;
                               ELSE pres_state <= idle;
                               END IF;

                WHEN retry => IF pwait='1' THEN pres_state <= ack;
                               ELSE pres_state <= retry;
                               END IF;
            END CASE;
        END IF;
    END PROCESS;
END
```

# Example: Solution 1 (contd.)



```
    WHEN ack => pres_state <= idle;  
    WHEN OTHERS => pres_state <= idle;  
  END CASE;  
  END IF;  
END PROCESS fsm;
```

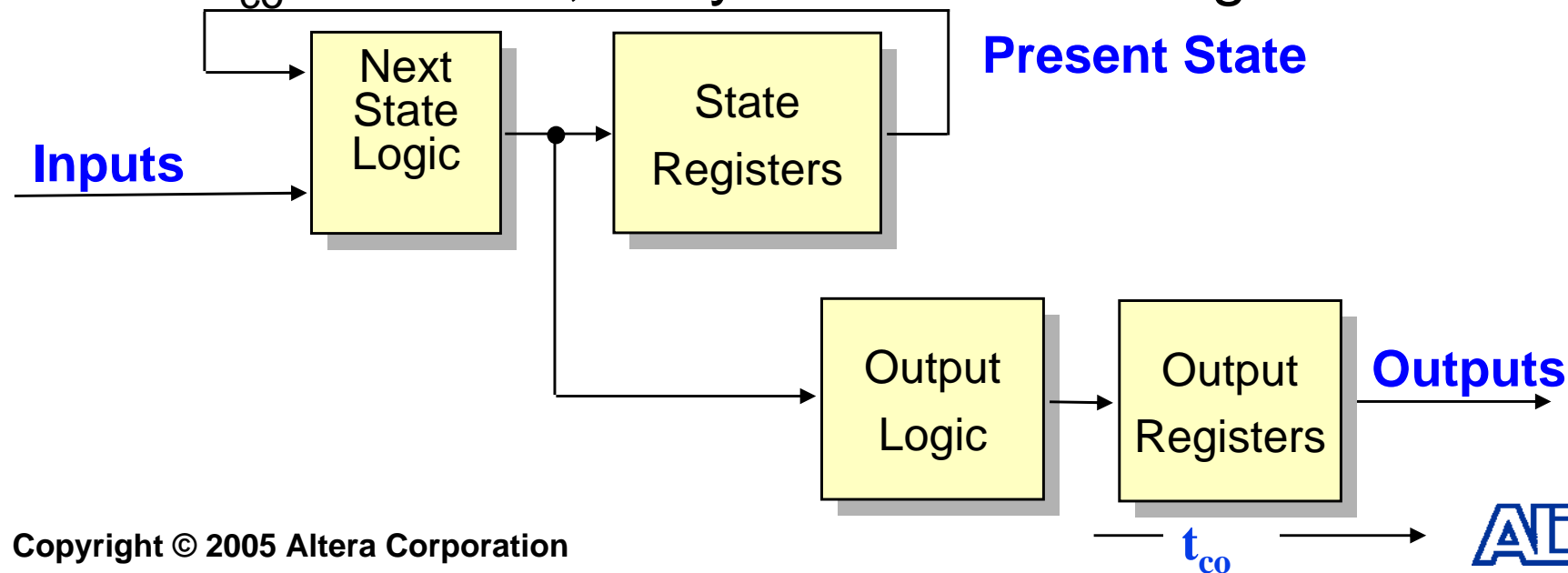
```
retry_out <= '1' WHEN (pres_state = retry) ELSE '0';  
ack_out   <= '1' WHEN (pres_state = ack)   ELSE '0';
```

```
END archmoore1;
```

# Moore state machine implementations (2)

## ■ Outputs decoded from state bits using REGISTERS

- registered output logic is *in parallel* with state registers
- outputs are a function of the previous state *and* the inputs
- $t_{co}$  is shorter, but you need more registers





# Example: Solution 2

## ■ Registered outputs decoded from the state bits

**ARCHITECTURE** archmoore2 OF moore2 IS

**TYPE** fsm\_states **IS** (idle, retry, ack);

**SIGNAL** pres\_state: fsm\_states;

**BEGIN**

fsm: **PROCESS** (clock, reset)

**BEGIN**

**IF** reset = '1' **THEN**

pres\_state <= idle;

retry\_out <= '0';

ack\_out <= '0';

**ELSIF** clock'EVENT AND clock = '1' **THEN**

retry\_out <= '0'; -- a default assignment

**CASE** pres\_state **IS**

**WHEN** idle =>

**IF** req = '0'

**THEN** pres\_state <= retry;

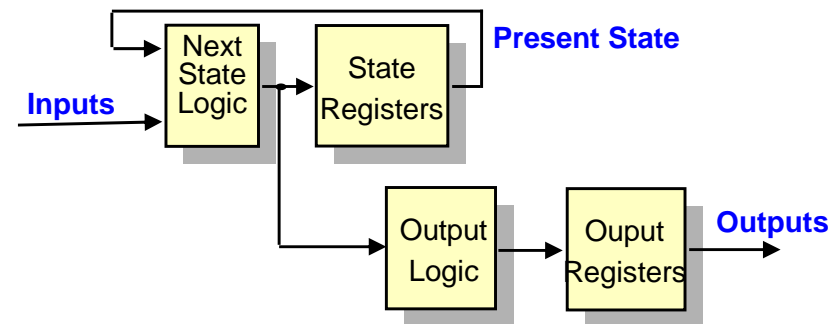
retry\_out <= '1';

ack\_out <= '0';

**ELSE** pres\_state <= idle;

ack\_out <= '0';

**END IF;**



# Example: Solution 2 (contd.)

```
    WHEN retry =>          IF pwait = '1'          THEN pres_state <= ack;
                           ack_out <= '1';
                           ELSE pres_state <= retry;
                               retry_out <= '1';
                           ack_out <= '0';
                           END IF;

    WHEN ack  =>            pres_state <= idle;
                           ack_out <= '0';

    WHEN OTHERS =>         pres_state <= idle;
                           ack_out <= '0'; -- note must define what
                                           -- happens to 'ack_out'
                                           -- here or a latch will
                                           -- be synthesized to
                                           -- preserve it's current state

    END CASE;
    END IF;
    END PROCESS fsm;
    END archmoore2;
```

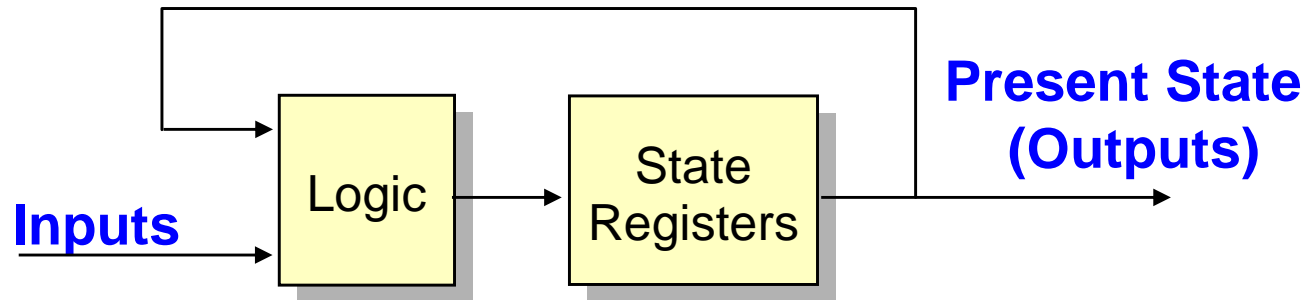
# Moore State Machine Implementations (3)

## ■ Outputs encoded within the state bits

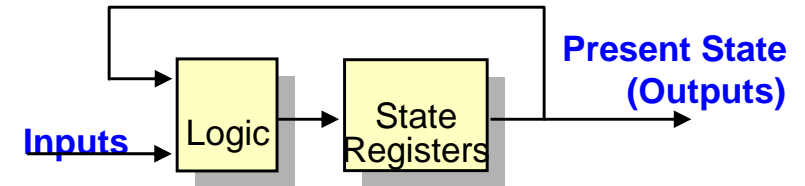
Example:

State	Output 1	Output 2	State Encoding
s1	0	0	00
s2	1	0	01
s3	0	1	10

**Note:** Both bits of the state encoding are used as outputs



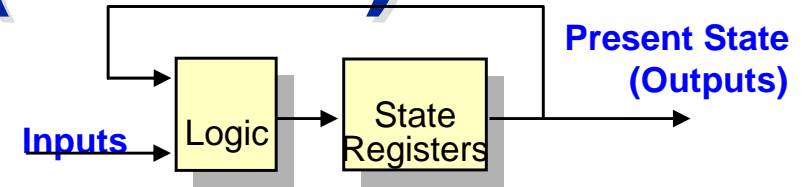
# Example: Solution 3



## ■ Outputs encoded within the state bits

```
ARCHITECTURE archmoore3 OF moore3 IS
  SIGNAL pres_state: std_logic_vector(1 DOWNTO 0);
  CONSTANT idle: std_logic_vector(1 DOWNTO 0) := "00";
  CONSTANT retry: std_logic_vector(1 DOWNTO 0) := "01";
  CONSTANT ack: std_logic_vector(1 DOWNTO 0) := "10";
BEGIN
  fsm: PROCESS (clock, reset)
  BEGIN
    IF reset = '1' THEN
      pres_state <= idle;
    ELSIF clock'EVENT AND clock = '1' THEN
      CASE pres_state IS
        WHEN idle => IF req = '0' THEN
          pres_state <= idle;
        ELSE
          pres_state <= retry;
        END IF;
      END CASE;
    END IF;
  END PROCESS;
END archmoore3;
```

# Example: Solution 3 (contd.)



```
WHEN retry =>          IF pwait = '1'          THEN pres_state <= ack;
                                ELSE pres_state <= retry;
                                END IF;
                                END IF;
                                WHEN ack  => pres_state <= idle;
                                WHEN OTHERS => pres_state <= idle;
                                END CASE;
                                END IF;
                                END PROCESS fsm;

                                retry_out <= pres_state(0);
                                ack_out  <= pres_state(1);

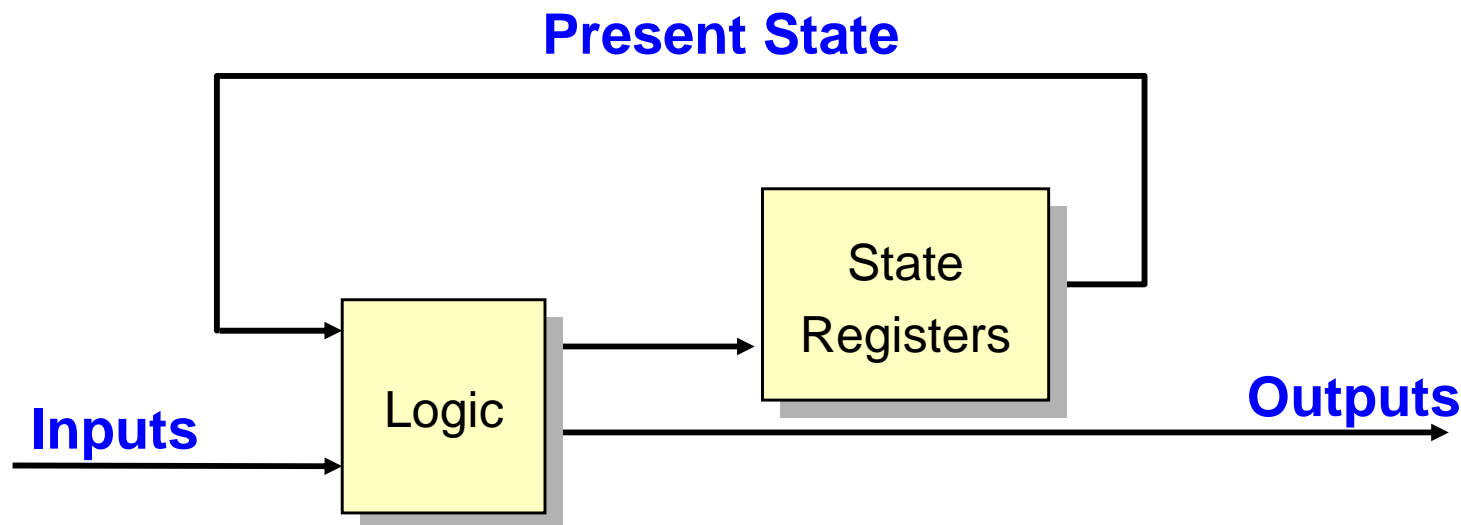
                                END archmoore3;
```

# Moore machines: Summary

- Outputs decoded from the state bits
  - flexibility during the design process
  - using enumerated types allows automatic state assignment during compilation
- Outputs encoded within the state bits
  - manual state assignment using constants
  - the state registers and the outputs are merged
  - reduces the number of registers
  - but, may require more product terms
- One-Hot encoding
  - reduces the number of product terms
  - high speed operation
  - but, uses more registers

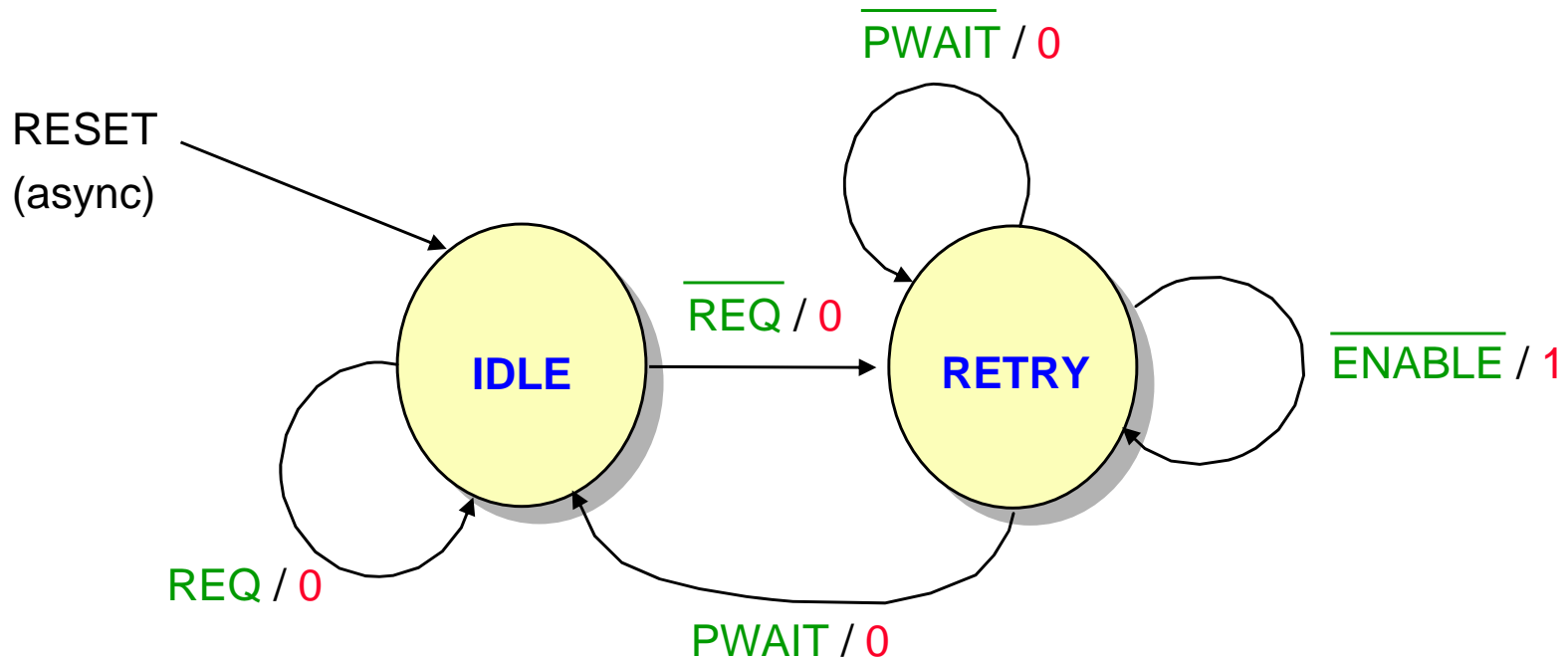
# Mealy machines

- Outputs may change with a change of state OR with a change of inputs
  - Mealy outputs are non-registered because they are functions of the present inputs



# Example: The Wait state generator

## ■ State diagram:



- Inputs: **REQ**, **PWAIT**, **ENABLE** + **CLOCK** + **RESET**
- Outputs: **RETRY\_OUT**
- States: **IDLE**, **RETRY**



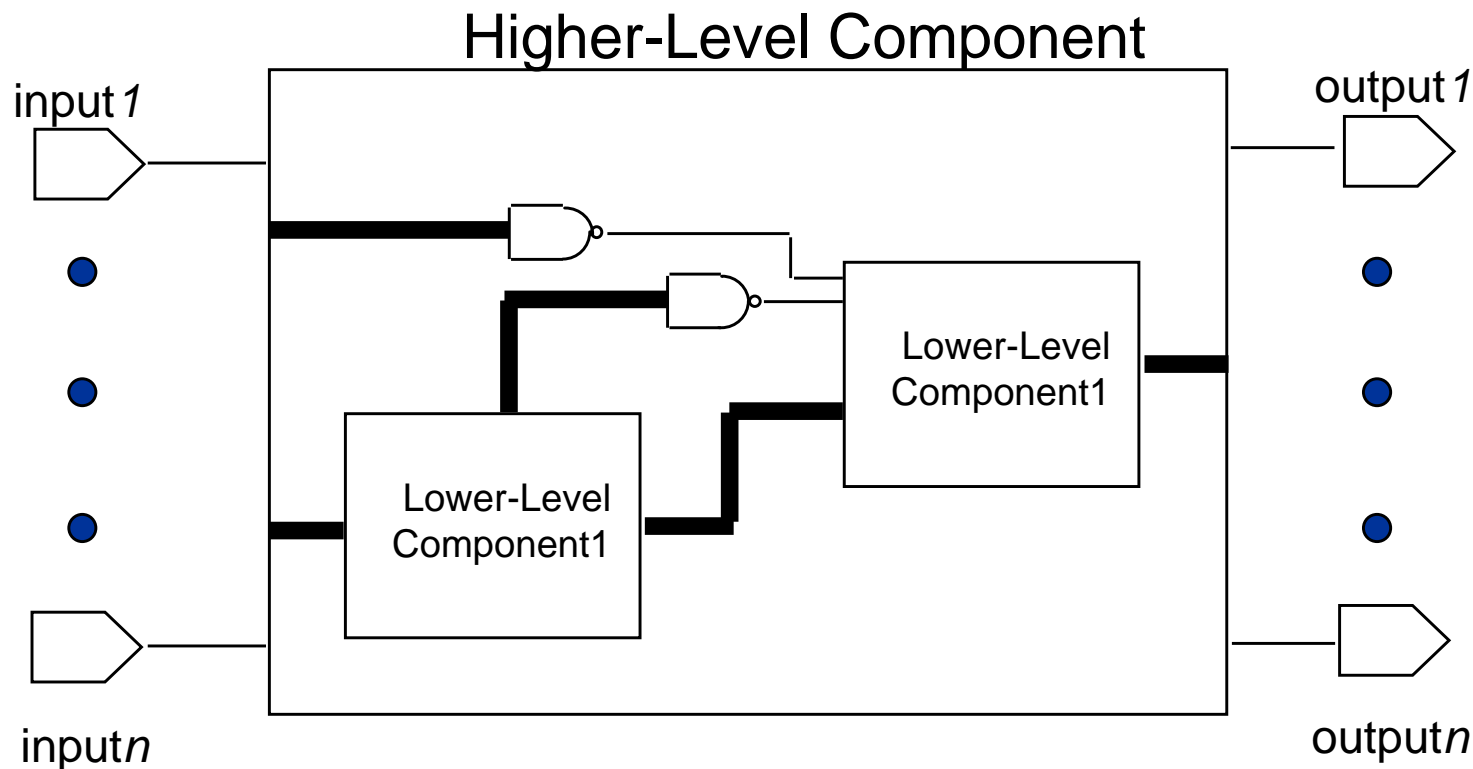
# Example: Mealy machine solution

```
ARCHITECTURE archmealy1 OF mealy1 IS
  TYPE fsm_states IS (idle, retry);
  SIGNAL pres_state: fsm_states;
BEGIN
  fsm: PROCESS (clock, reset)
  BEGIN
    IF reset = '1' THEN
      pres_state <= idle;
    ELSIF clock'EVENT AND clock = '1' THEN
      CASE pres_state IS
        WHEN idle          => IF req = '0'          THEN pres_state <= retry;
                                ELSE pres_state <= idle;
                                END IF;
        WHEN retry         => IF pwait = '1'        THEN pres_state <= idle;
                                ELSE pres_state <= retry;
                                END IF;
        WHEN OTHERS       => pres_state <= idle;
      END CASE;
    END IF;
  END PROCESS fsm;
  retry_out <= '1' WHEN (pres_state = retry AND enable='0') ELSE '0';
END archmealy1;
```

# Designing Hierarchically

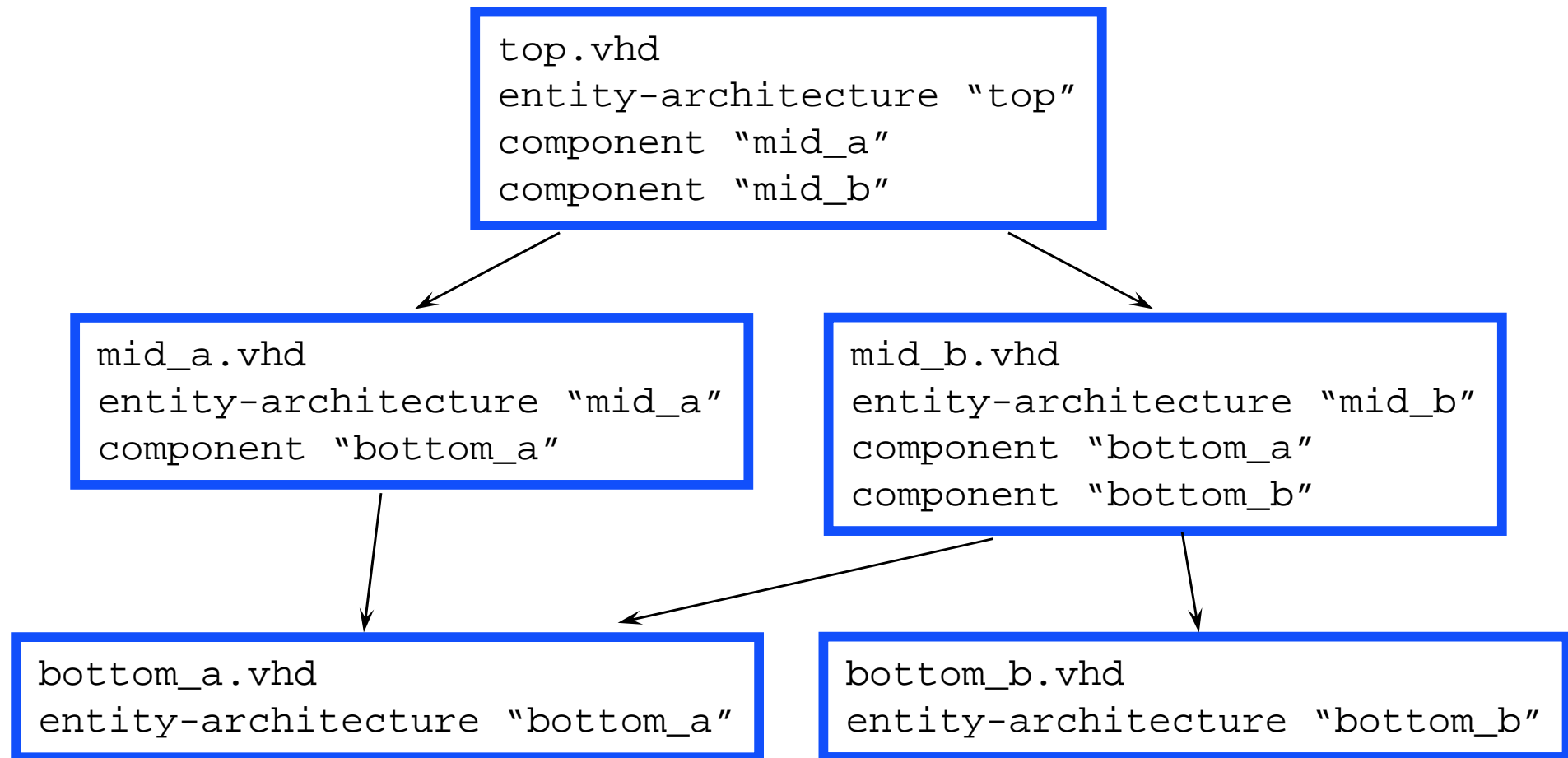
# Recall - Structural Modeling

- Functionality and Structure of the Circuit
- Call Out the Specific Hardware, Lower-Level Components
- For the Purpose of Synthesis



# Design Hierarchically - Multiple Design Files

- VHDL Hierarchical Design Requires Component Declarations and Component Instantiations



# Benefits of Hierarchical Designing

## Designing Hierarchically

- In a Design Group, Each Designer Can Create Separate Functions (Components) in Separate Design Files
- These Components Can Be Shared by Other Designers or Can Be Used for Future Projects
- Therefore, Designing Hierarchically Can Make Designs More Modular and Portable
- Designing Hierarchically Can Also Allow Easier and Faster Alternative Implementations
  - Example: Try Different Counter Implementations by Replacing Component Declaration and Component Instantiation

# Component Declaration and Instantiation

- Component Declaration - Used to Declare the *Port Types* and the *Data Types* of the Ports for a Lower-level Design

```
COMPONENT <Lower-level_design_name> IS  
  PORT ( <Port_name> : <Port_type> <Data_type>;  
          .  
          .  
          <Port_name> : <Port_type> <Data_type>);  
END COMPONENT;
```

- Component Instantiation - Used to Map the Ports of a Lower-level Design to That of the Current-level Design

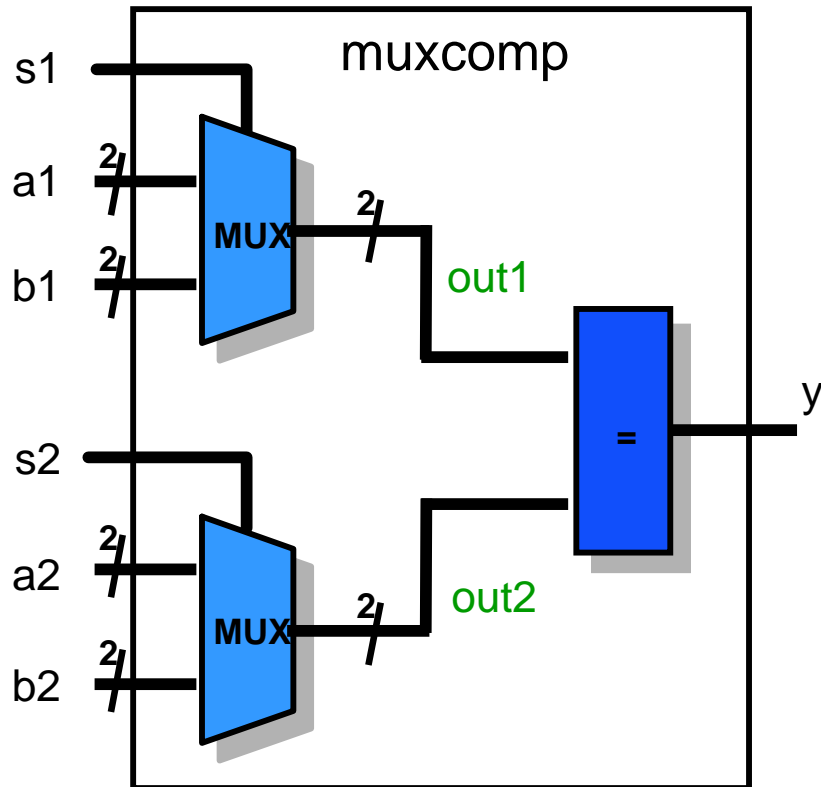
```
<Instance_name> : <Lower-level_design_name>  
PORT MAP(<lower-level_port_name> => <Current_level_port_name>,  
          ..., <Lower-level_port_name> => <Current_level_port_name>);
```

# Package and Component Declarations

- When you have created a working entity/architecture pair, you need to add a ***component declaration*** to make it a re-usable COMPONENT
- COMPONENTS need to be stored in PACKAGES, so you need to write a ***package declaration*** to store all your components
- When you compile your package with no errors, the components will be stored in the ***WORK*** “library”
- ***WORK*** is the ***current working directory*** where everything ***YOU*** compile gets stored. Because it is the current directory, you do NOT need to add it (even if a VHDL Design File should contain one Library Clause for each Use Clause.):

~~LIBRARY WORK;~~ -- not required

# COMPONENT: example



Without  
COMPONENT!

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY muxcomp IS PORT (
    a1,b1,a2,b2: IN std_logic_vector(1 DOWNTO 0);
    s1,s2:      IN std_logic;
    y:         OUT std_logic);
END muxcomp;

ARCHITECTURE archmuxcomp OF muxcomp IS
    SIGNAL out1,out2: std_logic_vector(1 DOWNTO 0);
BEGIN
    out1 <= a1 WHEN s1 = '1' ELSE b1;
    out2 <= a2 WHEN s2 = '1' ELSE b2;
    y <= '1' WHEN out1 = out2 ELSE '0';
END archmuxcomp;
```

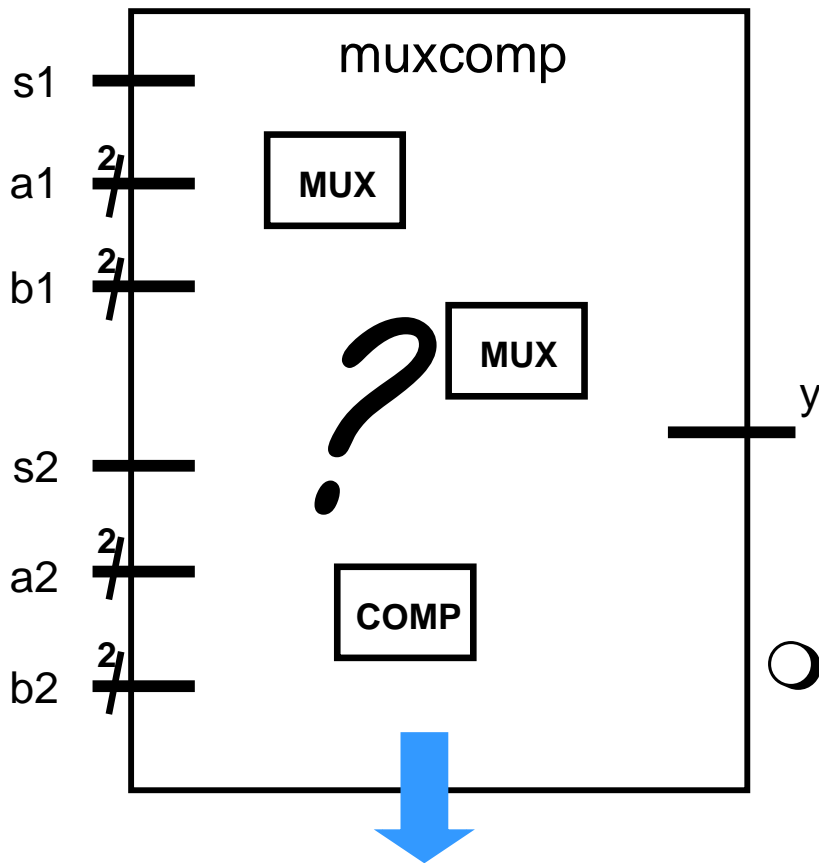


# COMPONENT: example - Top-Level

Modular solution

## Declaration Top-Level Entity

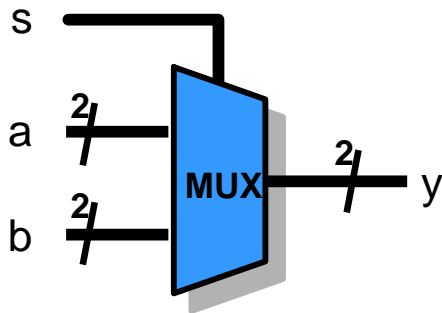
```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY muxcomp IS PORT (  
    a1,b1,a2,b2: IN std_logic_vector(1 DOWNT0 0);  
    s1,s2:      IN std_logic;  
    y:          OUT std_logic);  
END muxcomp;  
...
```



- Blocks
  - Library
  - User-defined

- The Top-Level Entity
  - Has the highest rank in the hierarchy
  - Contains and connects all the sub-blocks
  - Furnishes the path towards pins

# COMPONENT: example - definition

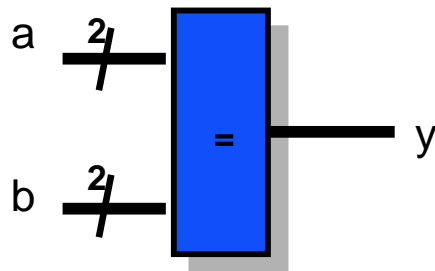
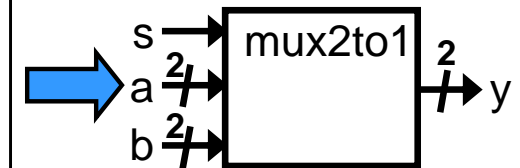


```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY mux2to1 IS PORT (
    a,b: IN std_logic_vector(1 DOWNTO 0);
    s: IN std_logic;
    y: OUT std_logic_vector(1 DOWNTO 0));
END mux2to1;
    
```

```

ARCHITECTURE archmux2to1 OF mux2to1 IS
BEGIN
    y <= a WHEN s = '1' ELSE b;
END archmux2to1;
    
```

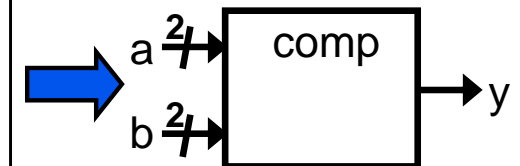


```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY comp IS PORT (
    a,b: IN std_logic_vector(1 DOWNTO 0);
    y: OUT std_logic);
END comp;
    
```

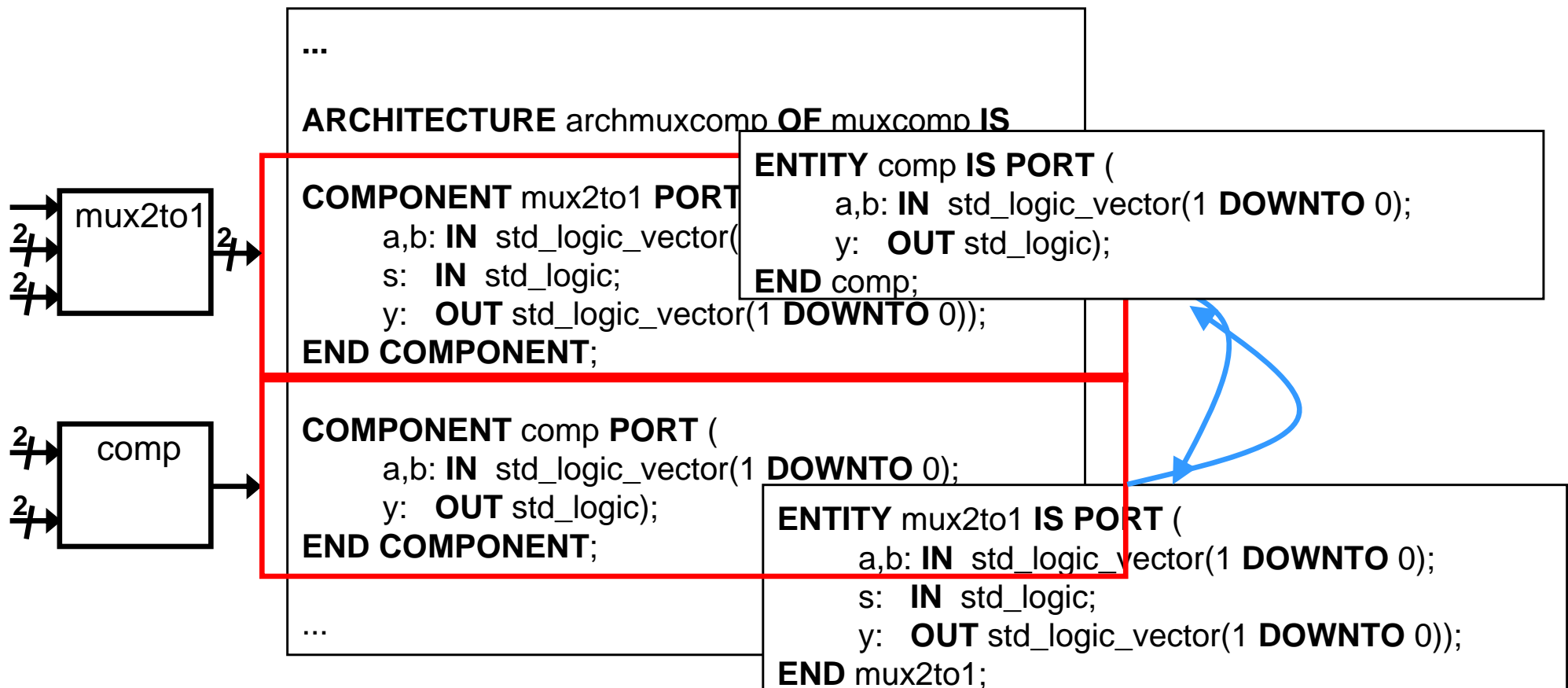
```

ARCHITECTURE archcomp OF comp IS
BEGIN
    y <= '1' WHEN a = b ELSE '0';
END archcomp;
    
```



# COMPONENT: example - declaration

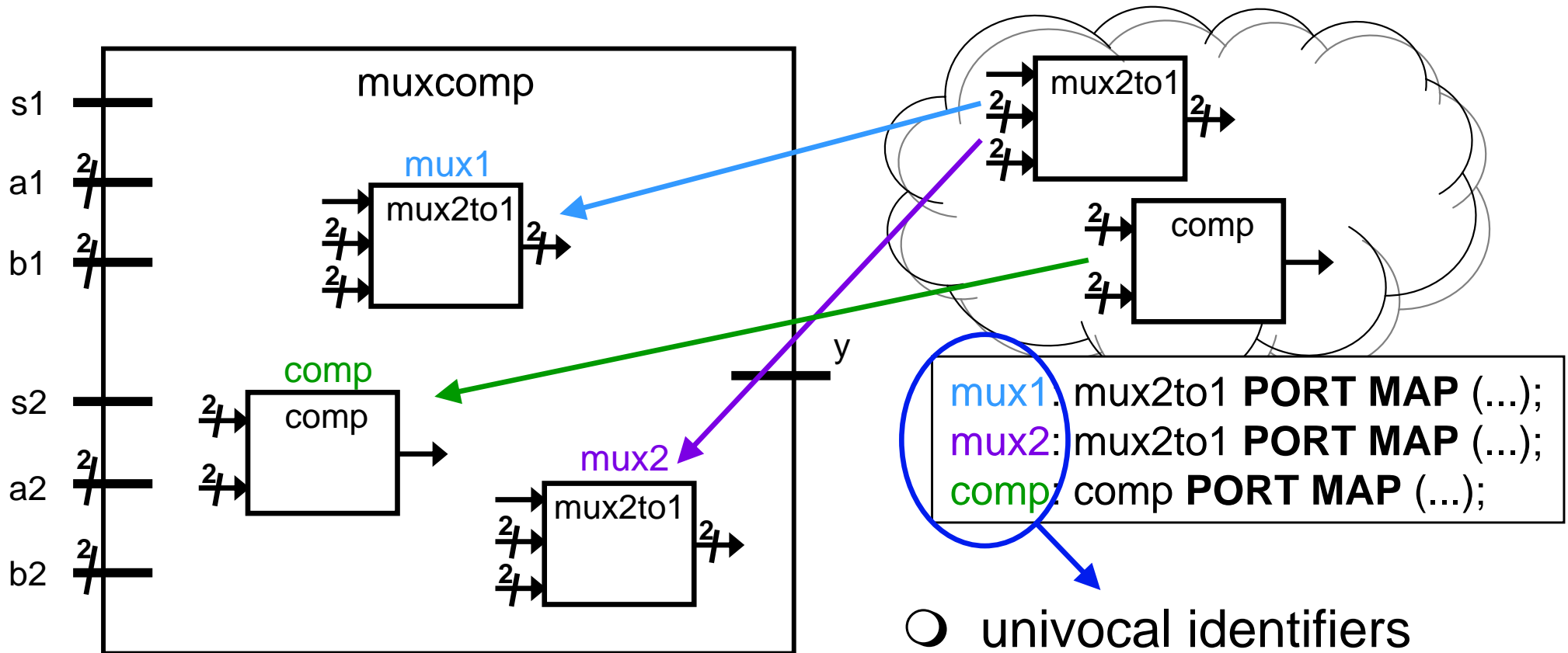
- A COMPONENT must be declared before it can be used in an architecture (as function prototypes in C...)



- COMPONENT declaration seems like its ENTITY

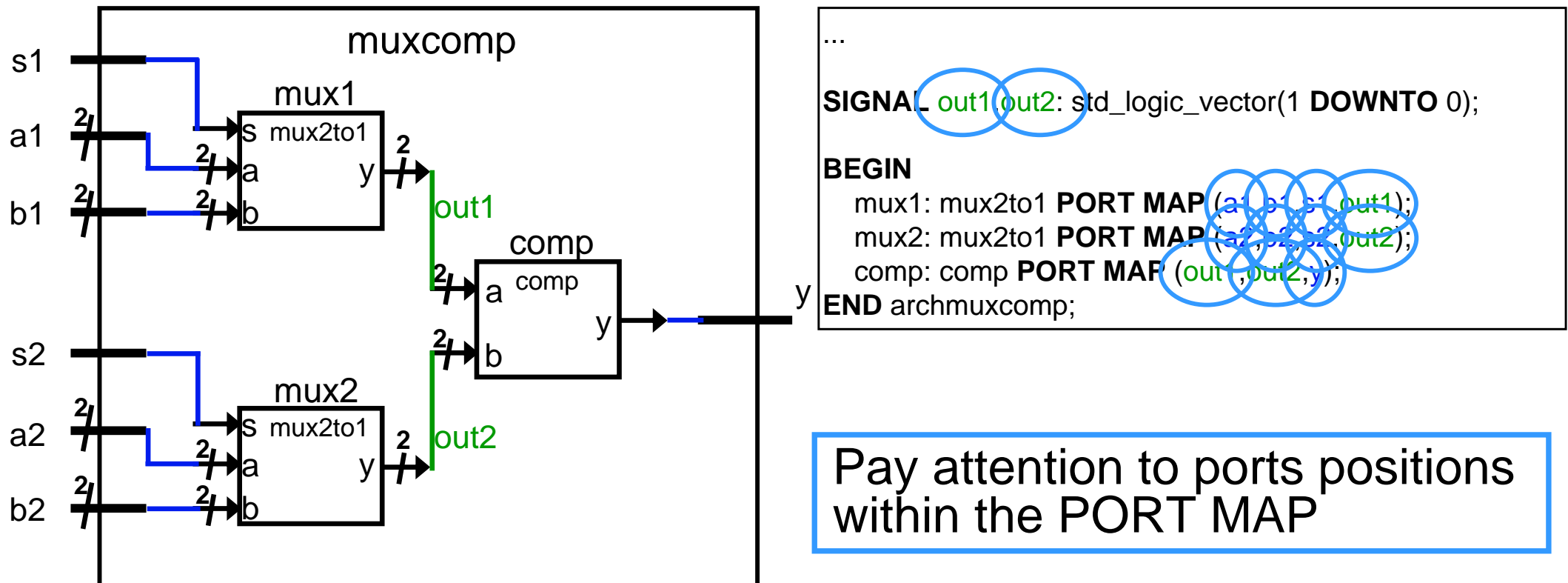
# COMPONENT: example - instance

- A declared and defined Component, can be instantiated several times within an Architecture
- Each instance has a univocal identifier



# COMPONENT: example - PORT MAP

- Components must be connected by means of PORT MAP



- Component Ports are connected with main Entity ports...
- ... Or to other components ports by means of signals

# COMPONENT: example - solution

- A modular project may be made of two files:
  - A File including component definitions
  - A Top-File with Top-Level Entity/Architecture and components definition/declaration

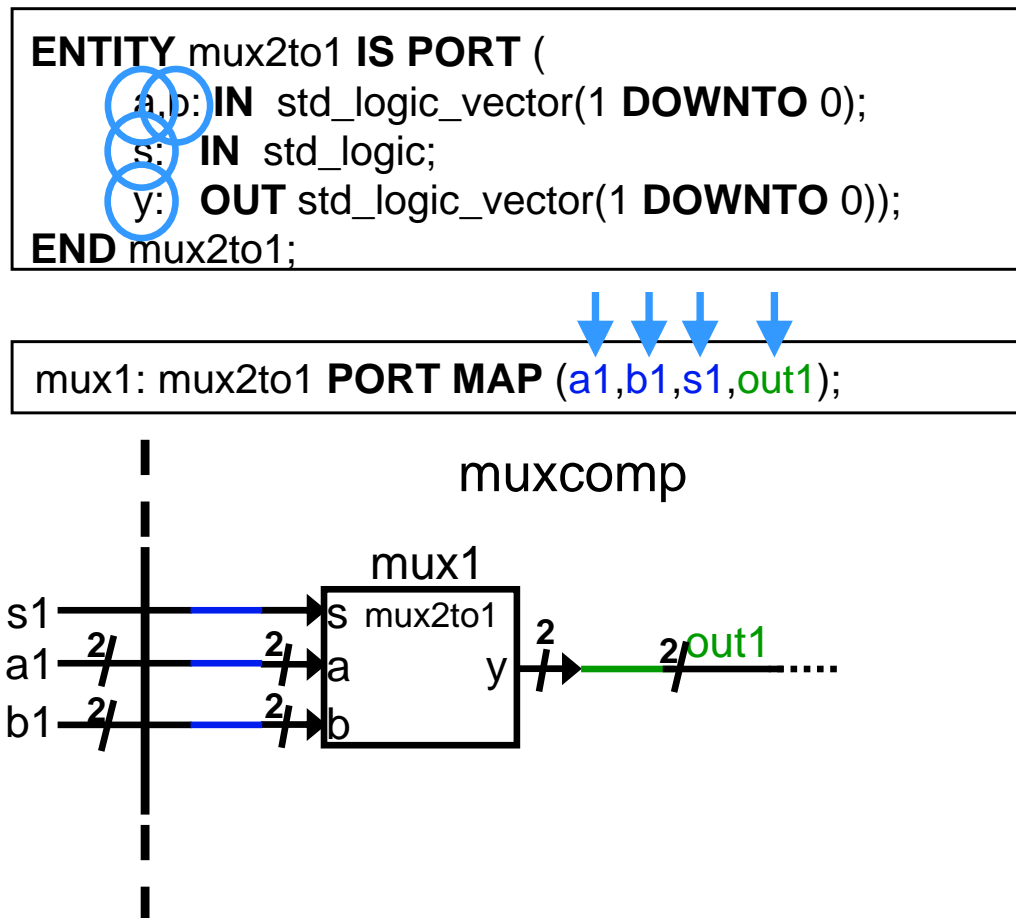
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY mux2to1 IS PORT (
  a,b: IN std_logic_vector(1 DOWNTO 0);
  s: IN std_logic;
  y: OUT std_logic_vector(1 DOWNTO 0));
END mux2to1;
ARCHITECTURE archmux2to1 OF mux2to1 IS
BEGIN
  y <= a WHEN s = '1' ELSE b;
END archmux2to1;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY comp IS PORT (
  a,b: IN std_logic_vector(1 DOWNTO 0);
  y: OUT std_logic);
END comp;
ARCHITECTURE archcomp OF comp IS
BEGIN
  y <= '1' WHEN a = b ELSE '0';
END archcomp;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY muxcomp IS PORT (
  a1,b1,a2,b2: IN std_logic_vector(1 DOWNTO 0);
  s1,s2: IN std_logic;
  y: OUT std_logic);
END muxcomp;
ARCHITECTURE archmuxcomp OF muxcomp IS
  COMPONENT mux2to1 PORT (
    a,b: IN std_logic_vector(1 DOWNTO 0);
    s: IN std_logic;
    y: OUT std_logic_vector(1 DOWNTO 0));
  END COMPONENT;
  COMPONENT comp PORT (
    a,b: IN std_logic_vector(1 DOWNTO 0);
    y: OUT std_logic);
  END COMPONENT;
  SIGNAL out1,out2: std_logic_vector(1 DOWNTO 0);
  BEGIN
    mux1: mux2to1 PORT MAP (a1,b1,s1,out1);
    mux2: mux2to1 PORT MAP (a2,b2,s2,out2);
    comp: comp PORT MAP (out1,out2,y);
  END archmuxcomp;
```

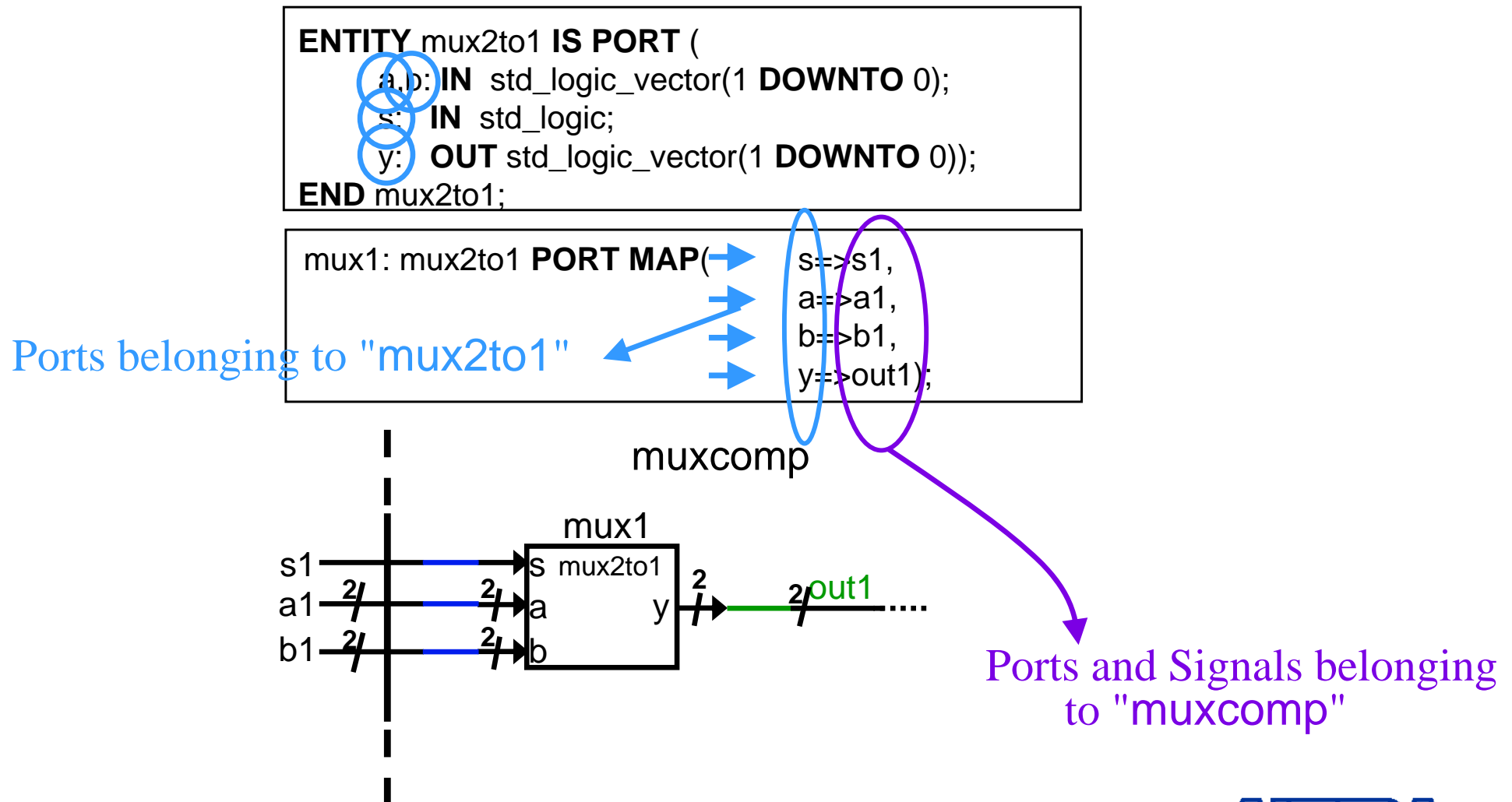
# PORT MAP

- In the previous example PORT MAP assignments were positional



# PORT MAP

- Named association explicitly identifies the connection between port identifiers and port map identifiers





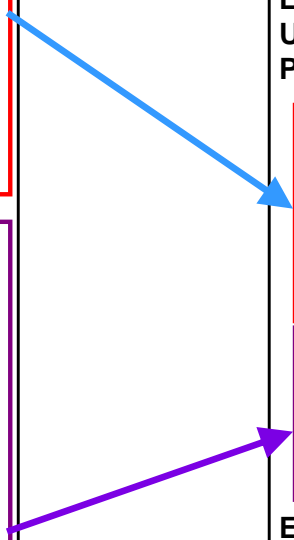
# PACKAGE: example

- At the beginning or at the end of the file containing the components definition, also the PACKAGE is defined, including COMPONENTs declarations

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY mux2to1 IS PORT (  
  a,b: IN std_logic_vector(1 DOWNTO 0);  
  s: IN std_logic;  
  y: OUT std_logic_vector(1 DOWNTO 0));  
END mux2to1;  
ARCHITECTURE archmux2to1 OF mux2to1 IS  
BEGIN  
  y <= a WHEN s = '1' ELSE b;  
END archmux2to1;
```

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY comp IS PORT (  
  a,b: IN std_logic_vector(1 DOWNTO 0);  
  y: OUT std_logic);  
END comp;  
ARCHITECTURE archcomp OF comp IS  
BEGIN  
  y <= '1' WHEN a = b ELSE '0';  
END archcomp;
```

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
PACKAGE mypkg IS  
  
  COMPONENT mux2to1 PORT (  
    a,b: IN std_logic_vector(1 DOWNTO 0);  
    s: IN std_logic;  
    y: OUT std_logic_vector(1 DOWNTO 0));  
  END COMPONENT;  
  
  COMPONENT comp PORT (  
    a,b: IN std_logic_vector(1 DOWNTO 0);  
    y: OUT std_logic);  
  END COMPONENT;  
  
END mypkg;
```



# PACKAGE: example - declaration

- In order to use COMPONENTs, it is no more needed to declare each one of them, but only to include (declaration) the PACKAGE in which they are stored

## Without PACKAGE

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY muxcomp IS PORT (
  a1,b1,a2,b2: IN std_logic_vector(1 DOWNTO 0);
  s1,s2: IN std_logic;
  y: OUT std_logic);
END muxcomp;
ARCHITECTURE archmuxcomp OF muxcomp IS
COMPONENT mux2to1 PORT (
  a,b: IN std_logic_vector(1 DOWNTO 0);
  s: IN std_logic;
  y: OUT std_logic_vector(1 DOWNTO 0));
END COMPONENT;
COMPONENT comp PORT (
  a,b: IN std_logic_vector(1 DOWNTO 0);
  y: OUT std_logic);
END COMPONENT;
SIGNAL out1,out2: std_logic_vector(1 DOWNTO 0);
BEGIN
  mux1: mux2to1 PORT MAP (a1,b1,s1,out1);
  mux2: mux2to1 PORT MAP (a2,b2,s2,out2);
  comp: comp PORT MAP (out1,out2,y);
END archmuxcomp;
```

## With PACKAGE

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY muxcomp IS PORT (
  a1,b1,a2,b2: IN std_logic_vector(1 DOWNTO 0);
  s1,s2: IN std_logic;
  y: OUT std_logic);
END muxcomp;
USE work.mypkg.ALL;
ARCHITECTURE archmuxcomp OF muxcomp IS
SIGNAL out1,out2: std_logic_vector(1 DOWNTO 0);

BEGIN
  mux1: mux2to1 PORT MAP (a1,b1,s1,out1);
  mux2: mux2to1 PORT MAP (a2,b2,s2,out2);
  comp: comp PORT MAP (out1,out2,y);
END archmuxcomp;
```

All the components...

... In the package

"mypkg"...

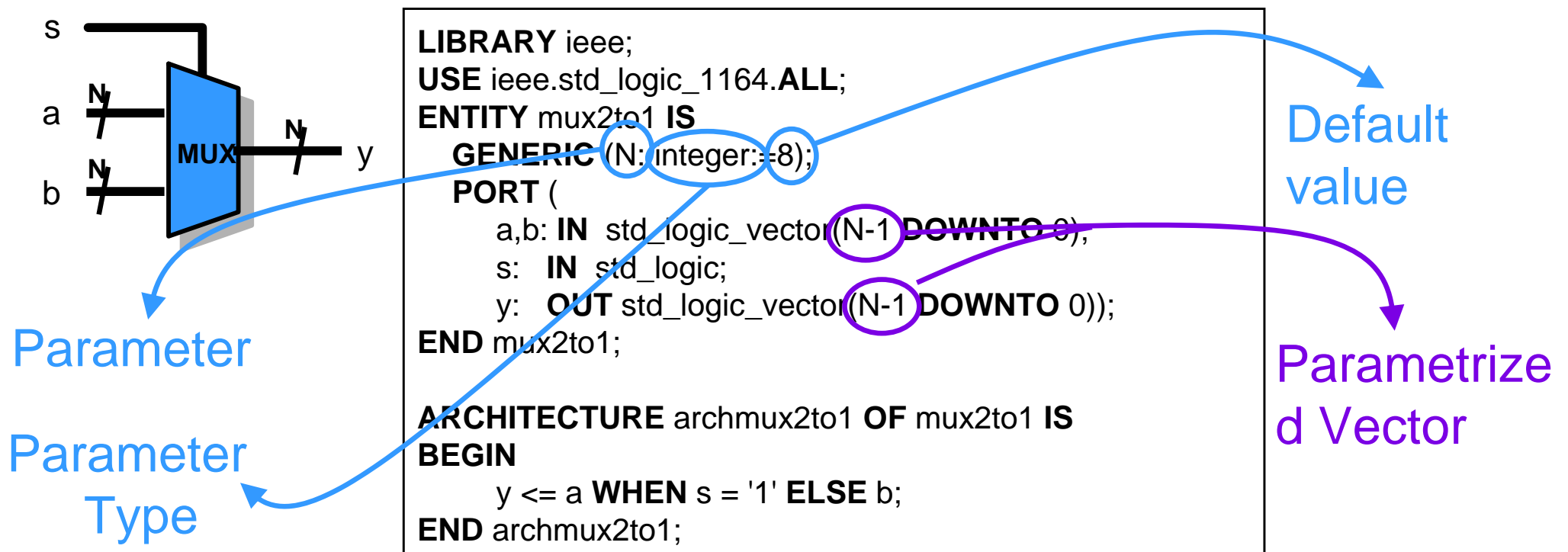
... In the library

"work"

**ALTERA**

# GENERIC

- Allows to define a “generic” components
- Example: PORT dimension, counter direction,...



# GENERIC - mapping

- Also GENERIC, as PORT, have to be mapped
- Named and positional notations are allowed

```
ARCHITECTURE archmuxcomp OF muxcomp IS
  COMPONENT mux2to1
    GENERIC (N: integer:=8);
    PORT (
      a,b: IN std_logic_vector(N-1 DOWNT0 0);
      s: IN std_logic;
      y: OUT std_logic_vector(N-1 DOWNT0 0));
  END COMPONENT;
  COMPONENT comp PORT (
    a,b: IN std_logic_vector(1 DOWNT0 0);
    y: OUT std_logic);
  END COMPONENT;
  SIGNAL out1,out2: std_logic_vector(1 DOWNT0 0);
  BEGIN
    mux1: mux2to1 GENERIC MAP (2)
      PORT MAP (a1,b1,s1,out1);
    mux2: mux2to1 GENERIC MAP (2)
      PORT MAP (a2,b2,s2,out2);
    comp: comp PORT MAP (out1,out2,y);
  END archmuxcomp;
```

Component declaration

Default value is  
overwritten!

Instances