OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Introduction to VHDL

Marta Bagatin, marta.bagatin@unipd.it

Degree Course in Information Engineering

Academic Year 2023-2024

Purpose of the Lesson

- Introduce the main concepts of **Hardware Description Languages**
- Introduce the **main parts of a VHDL code**
  - Entity
  - Architecture
  - Libraries and package
  - Components
- See some **examples of VHDL codes**
  - Description with architecture Structural
  - Description with architecture Dataflow
  - Description with architecture Behavioral
  - Simulation with Testbench

What is VHDL?
How will we study VHDL?

Hardware Description Language (HDL)

- **Computer-Aided Design (CAD)** tools are crucial for the design of digital systems with billions of transistors
  - Manual design/synthesis would be impractical (time, probability of error, cost, ...)
- **HDL** (**H**ardware **D**escription **L**anguage): allows a description at different levels of abstraction that can be simulated or synthesized (i.e. translated into hardware)
- **The most used HDLs are VHDL and Verilog**, both supported by the IEEE (Institute of Electrical and Electronics Engineers)
  - VHDL and Verilog are standardized, so they can be exchanged between different CAD tools

VHDL

- In HDL the **statements are concurrent, that is they are active in parallel, all at the same time** (as logic gates in a circuit work in parallel)
- An HDL is NOT a programming language (ex: Java, C ++, Python)
  - A programming language is translated into a sequence of instructions, which is executed sequentially (i.e. one after the other) by a microprocessor
  - An HDL language is translated into a digital circuit (logic synthesis or simulations)
- **VHDL: Very high speed integrated circuits Hardware Description Language**: was born in 1987 from a project of the US Department of Defense and was recognized as an IEEE standard (Institute of Electrical and Electronics Engineers)
  - Several revisions have been made over the years

VHDL: Purposes

- **Documentation** of a design
- **Simulation** of the behavior of a design: study the signals evolution, evaluation of correct functionality of the circuit
  - Apply a series of input signals to the circuit and observe the outputs
- **Automatic synthesis** of a design: transition from a behavioral description to a description at the level of logic gates, i.e. translation of the VHDL code in a network (netlist) of cells (logic cells), which can in turn be used to produce a real circuit
  - Supplied by special programs that rely on libraries where the available logic gates are described (supplied by the vendors)
- In this course we will NOT deal with circuit synthesis

Synthesizable and Non-synthesizable VHDL

- Synthesis is applicable only to a subset of the possible commands (synthesizable VHDL). **Not all the code written in VHDL is synthesizable!**
- Conversely, if the hardware description is meant to be simulated only, the entire set of VHDL commands can be used

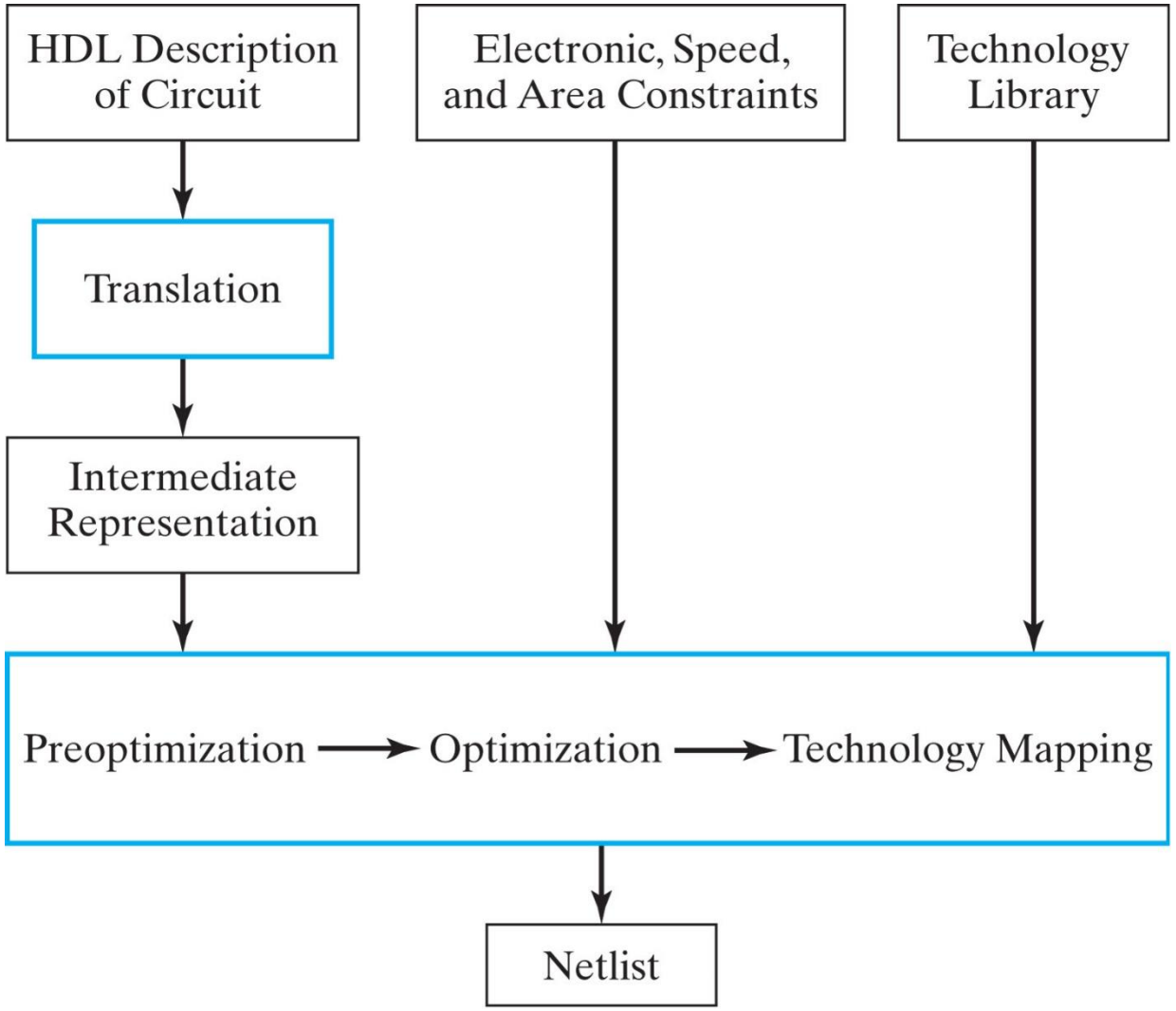
Levels of Abstraction

- VHDL allows us to describe a circuit at **different levels of abstraction**
  - **Structural**: describes the system in the form of interconnected components, it is equivalent to a schematic → **low** level of abstraction
  - **Behavioral**: describes the behavior and functionality of the system, regardless of the actual implementation at the component level → **high** level of abstraction
  - **Dataflow**: **intermediate** level of abstraction, between structural and behavioral, describes the system in the form of a flow of data

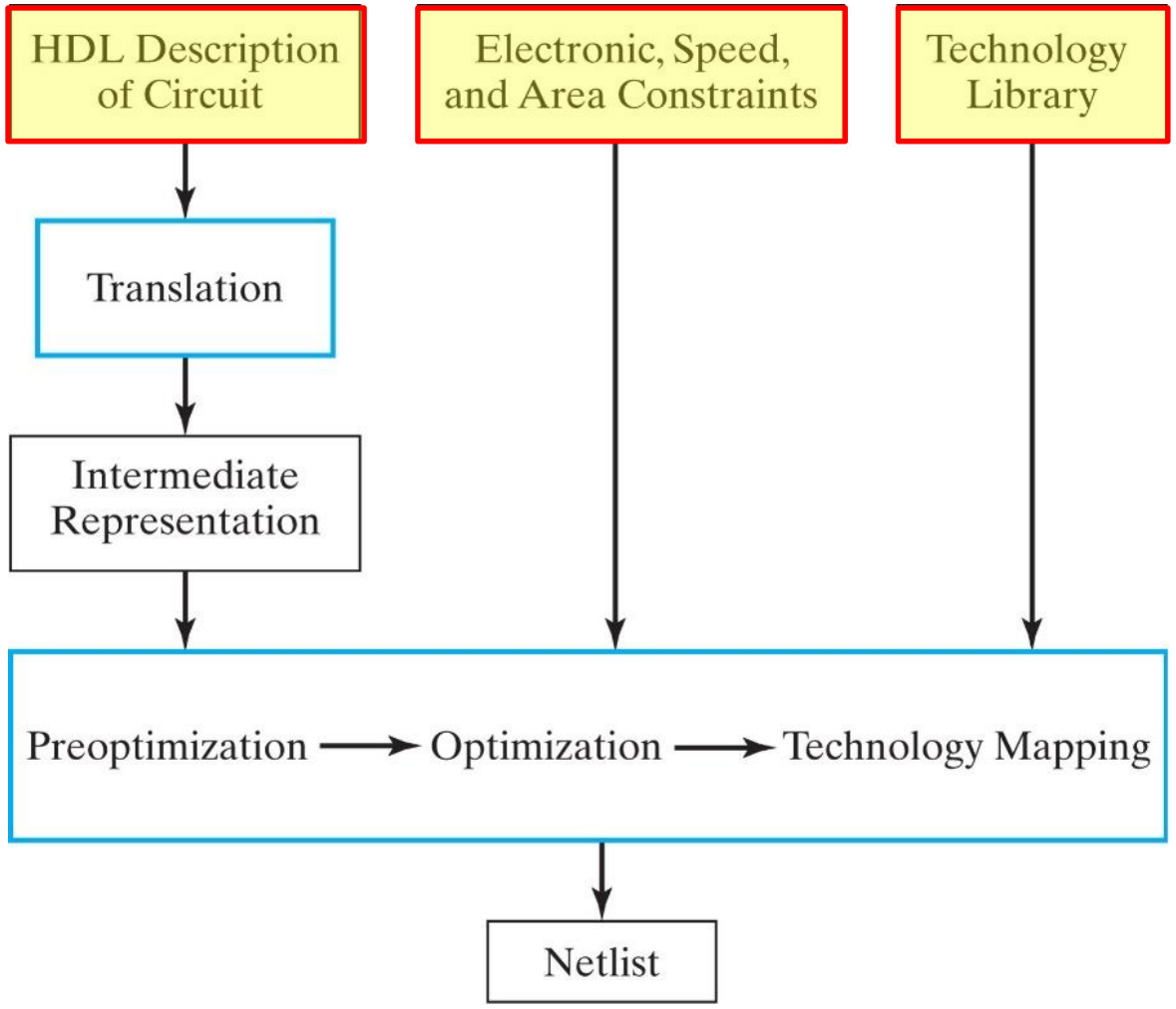
Example: Levels of Abstraction

- VHDL code of a circuit that adds two numbers A and B
- **Structural** description: presents a certain structure to the synthesizer, that is a specific set of interconnected blocks into which the system is divided
  - The designer has control on the circuit architecture and can suggest the most performing schematics to the synthesizer
- **Behavioral** description: tells the synthesizer to add A and B, without specifying the circuit structure: `F <= A + B;`
  - The designer leaves the choice of the circuit structure to the synthesizer

Circuit Synthesis Flow

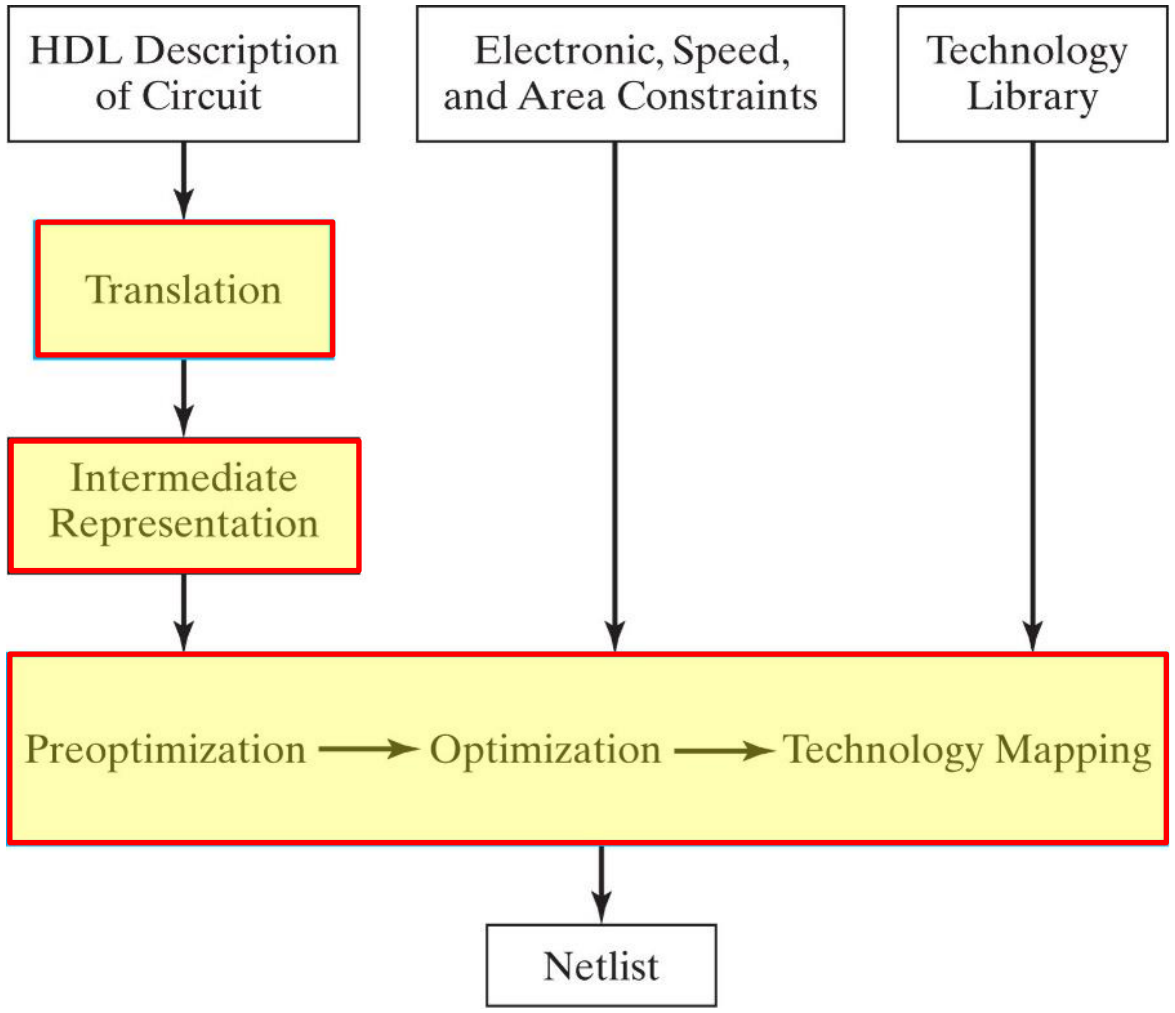


Circuit Synthesis Flow



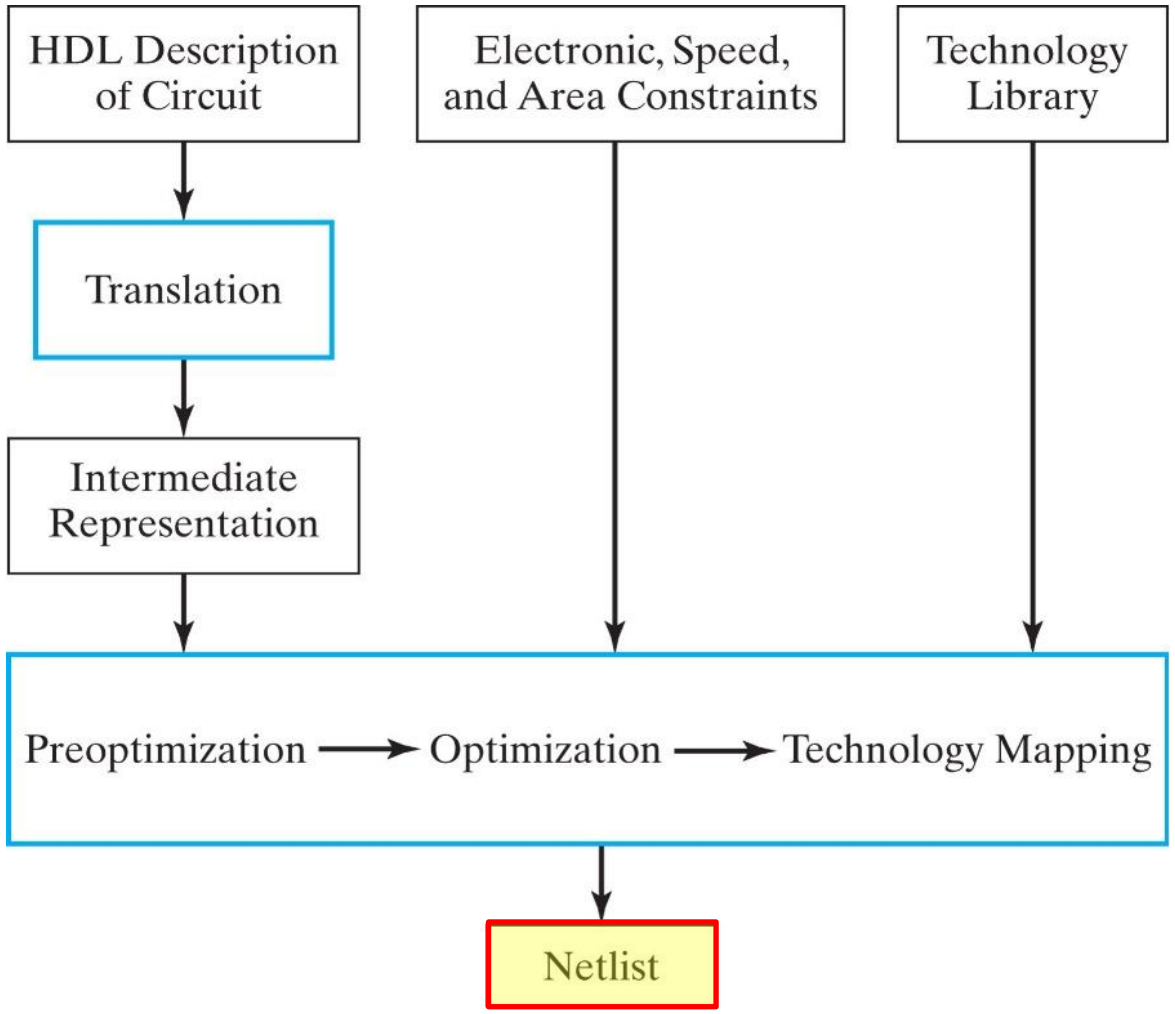
- **Circuit description** (HDL code)
- Project **constraints/ specifications** (e.g. area, speed, power, ...)
- Any additional information contained in **technological libraries** that describe the primitive blocks (ex: delays, area, ...)

Circuit Synthesis Flow



- Translation in an **intermediate form (RTL, Register Transfer Level)**: logic gates and generic memory elements connected to each other, not linked to a specific technology
- **Optimization** to satisfy the constraints
- **Technological mapping** replaces the primitive blocks with the gates available in the technology library
- Iterative process

Circuit Synthesis Flow



- The result is a **netlist** (**circuit diagram)**
- The netlist will be used by the design tools to realize the final layout for the manufacturing of the integrated circuit
- In case of implementation in programmable circuits (FPGA), it is used to produce the file that specifies how to program the logic inside the device

VHDL Basics

General Features of VHDL

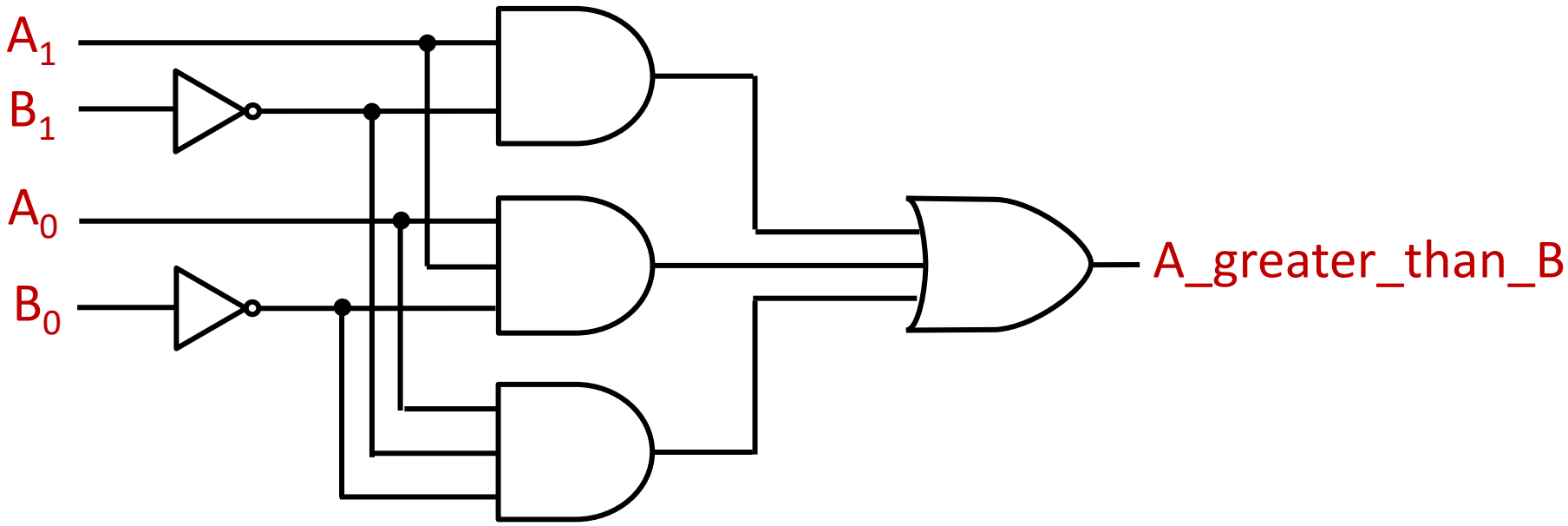
- VHDL is a **strongly-typed language**: data of different types cannot be assigned to each other without using proper conversion functions
- VHDL is **case insensitive** (it does not distinguish between lowercase and uppercase letters)
- Each line of the code can be written on several lines: spaces, tabs, and carriage returns are ignored
- It is important to **always keep in mind the hardware** of the circuit you want to describe!

Structure of the VHDL Code

- **First example of VHDL code: two-bit comparator**
- We will see the different types of circuit description and the basic blocks of VHDL code
  - Entity
  - Architecture
  - Data types
  - Component
  - Signal...
- We will start from a low-level description (structural) and then we will gradually increase the abstraction level (dataflow, behavioral). Finally, we will see how to simulate a design (testbench)

2-bit Comparator

Circuit diagram



Two 2-bit inputs: A: A_1A_0 , B: B_1B_0(0 is the LSB, 1 is the MSB)

One 1-bit output: A_greater_than_B is '1' if A > B

2-bit Comparator: VHDL

```
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
```

```
entity comparator_greater_than_structural is
  port (A: in std_logic_vector(1 downto 0);
    B: in std_logic_vector(1  downto 0);
    A_greater_than_B: out std_logic);
end comparator_greater_than_structural;
```

```
architecture structural of comparator_greater_than_structural is

 component NOT1
  port(in1: in std_logic;
    out1: out std_logic);
 end component;
 component AND2
  port(in1, in2: in std_logic;
   out1: out std_logic);
 end component;
 component AND3
  port(in1, in2, in3: in std_logic;
   out1: out std_logic);
 end component;
 component OR3
  port(in1, in2, in3 : in std_logic;
    out1: out std_logic);
 end component;
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 inv_0: NOT1 port map (in1 => B(0), out1 => B0_n);
 inv_1: NOT1 port map (B(1), B1_n);
 and_0: AND2 port map (A(1), B1_n, and0_out);
 and_1: AND3 port map (A(1), A(0), B0_n, and1_out);
 and_2: AND3 port map (A(0), B1_n, B0_n, and2_out);
 or0: OR3 port map (and0_out,and1_out,and2_out, A_greater_than_B);
end structural;
```

Libraries and packages

Entity

Architecture

Entity

- All designs must have an **entity**
- The entity is like the **identification plate** of the design: the entity specifies the its name and defines its interface with the outside world (how many, which, and what type are the **inputs and outputs**)
- The entity does not specify what is inside the block, it is analogous to a symbol in a circuit diagram

Entity

Syntax for the entity declaration:

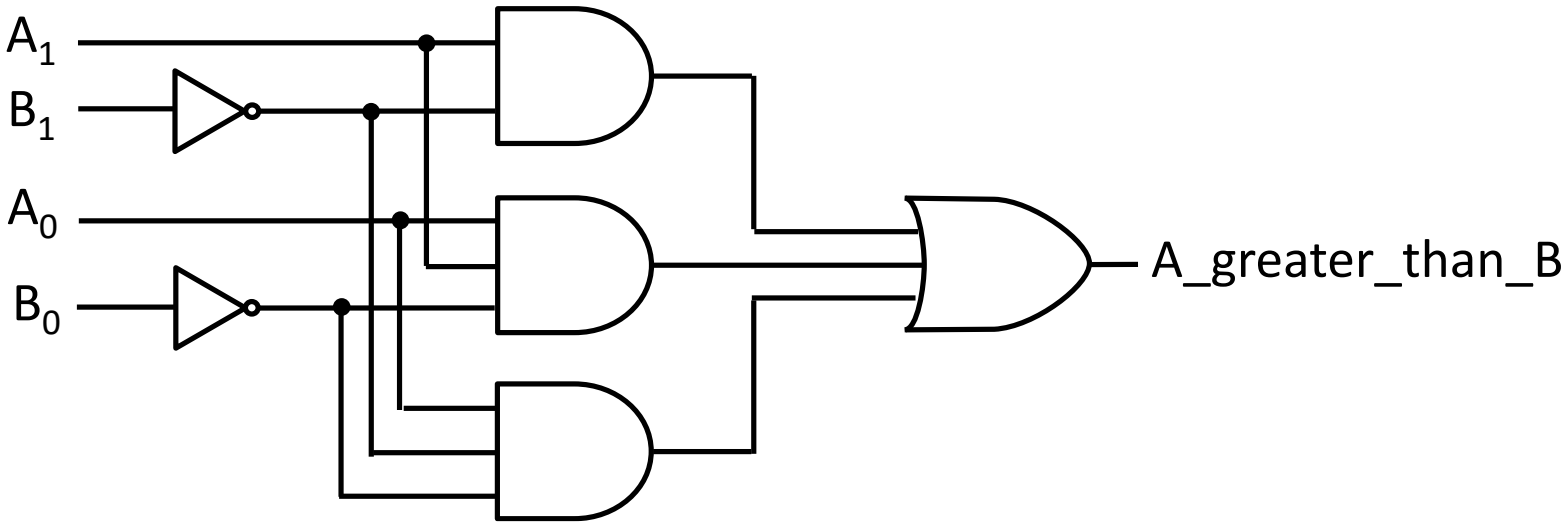
```
entity entityName is
-- Declaration of inputs (in) and outputs (out):
 port(port1Name, port2Name, ...: in typeName;
      portName11, portName12, ...: out typeName);
-- ";" goes after bracket closure
end entityName;
```

Note

- **Reserved words** must necessarily be in the indicated positions
- Possible comments can be written after --
- **in** / **out** is the mode (or direction) of the signals
- typeName is the data type

2-bit Comparator: Entity

```
entity comparator_greater_than_structural is
  port (A: in std_logic_vector(1 downto 0);
    B: in std_logic_vector(1  downto 0);
    A_greater_than_B: out std_logic);
end comparator_greater_than_structural;
```



Terminology

- **Assignment: X <= Y**: Y is assigned to X, that is we write the value of Y into X

  <= is to be understood as an arrow that goes from right to left
- We say that the statement X <= Y represents a driver for signal X
  - Whenever Y changes its value, X changes accordingly
- A **driver** is the source of a signal

Mode

- The mode indicates **the direction of signals in the entity** and it also indicates whether a gate can be read or assigned by the block
  - **in** (mode used for the **inputs**): data flow inside the entity. The value can be read but not assigned by the entity, that is, we can only read but not drive that gate. The driver stands outside of the entity
  - **out** (mode used for the **outputs**): data flow out of the entity. The value can be assigned (driven) but not read from the entity. The driver is inside the entity
  - **Inout**
  - **Buffer**

  } We will not study these modes!

Data Types: std_logic

- The data type **std_logic** is part of the **IEEE std_logic_1164** library
- It is used to describe a **1-bit logic signal** and can assume the following values

Of interest for our purposes

  - '0': logic 0
  - '1': logic 1
  - 'X': unknown or undefined value (can be logic '0' or logic '1')
  - '-': don't care (the synthesizer is free to optimize the function)
  - 'Z': high impedance (if it is an output, it is disconnected, floating)
  - 'U': not initialized (has no initial value), used for simulations
  - 'L': weak '0'
  - 'H': weak '1'

- The value of a **std_logic** is indicated **between single quotes**, for example '1'

Data Types: std_logic_vector

- The data type **Std_logic_vector** is used to describe a multi-bit signal, i.e. a sequence of **n bits** of **std_logic** type
- The indexes are indicated in ascending or descending order with the reserved words **downto** / **to**, the choice of which determines the index of the MSB

  Syntax for a sequence of n bits:

```
std_logic_vector (n-1 downto 0) -- n-1 is the MSB
std_logic_vector (0 to n-1) -- 0 is the MSB
```

Avoid mixing vectors defined with «downto» and «to» in the same code!

- Notation to extract a bit from a vector:

  `VectorName(indexBit)`

  Example: `x(2)` indicates the bit with index 2 in the vector x

- The value of a **std_logic_vector** is indicated **between double quotes**, e.g. "010"

Other Data Types

- **Bit**: **1 bit**, can only assume **'0' and '1' values**
- **Boolean**: **1 bit**, can only take the values **'TRUE' and 'FALSE'**
  - VHDL-2008 automatically converts the std_logic type to Boolean when a Boolean value is expected (ex: in the condition of an if statement)
- **Integer**: **32-bit** integers, from $-(2^{31} - 1)$ to $2^{31} - 1$
  - Typically used to specify parameters (e.g. length of a vector or index of a for cycle)
  - It is indicated in **decimal system** (digits 0 to 9) and **with no quotation marks** (ex: 100)
- These types of data are different from std_logic and std_logic_vector!

We will not use these data types to represent logic signals in digital systems! We will always use std_logic and std_logic_vector!

Architecture

- After the entity, **architecture** is the second fundamental part of a VHDL code
- Architecture provides a **particular description of the block defined in the entity**: architecture specifies what is inside the circuit, how it behaves, and how it works internally
- An **entity can be mapped to several architectures** in the same design. So a VHDL design/block must have a single entity, but it may have multiple architectures

Architecture

- The architecture can provide three main types of description, in increasing order of abstraction
  - **Structural**: makes use of **components**, is equivalent to the schematic of a circuit and it is used for hierarchical structures
  - **Dataflow**: describes the **signal flow** across the circuit, it is equivalent to a set of Boolean equations
  - **Behavioral**: **describes the behavior** without specifying the structure, it is used for complex circuits when it is not necessary to go into the details of the implementation
- Combinations between these types of descriptions are usually employed

Architecture

Syntax for the declaration of architecture:

```
architecture architectureName of entityName is
- Possible component declarations:
component componentName
      port ( in1: in typeName;
             out1: out typeName);
end component;
- any declaration of signals and constants:
signal signal1Name, signal2Name, ...: typeName;
constant constant1Name: typeName : = constant1Value;
- the body of architecture starts after 'begin':
      begin
             ...
end architectureName;
```

Component

- Components are used in **structural descriptions**: they can be connected to each other, even in nested structures (hierarchies) and reused several times in the design
- Components can be
  - **described by the user**, within the same file or in another file (each component must have its own entity and at least one architecture)
  - or taken from **predefined libraries** (e.g., provided by FPGA manufacturers)
- Creating an **instance** of a component means inserting a block (already defined elsewhere) by connecting its input and output ports (defined in entity) to the desired signals
- A structural, block-level description is obtained, equivalent to a **schematic**

Signal

- Whereas the input and output ports are defined in the entity and they communicate with the outside world, **signals represent wires within a circuit and they are defined in the architecture**
- Signals are used for **internal connections in the circuit,** for instance to connect the nodes of the circuit or connect different blocks together to form a larger design
- Signals must be defined before the keyword 'begin' inside architecture. It is good practice to use descriptive names for the signals
- Syntax:

```
signal signalName: signalType [: = signalValue];
```

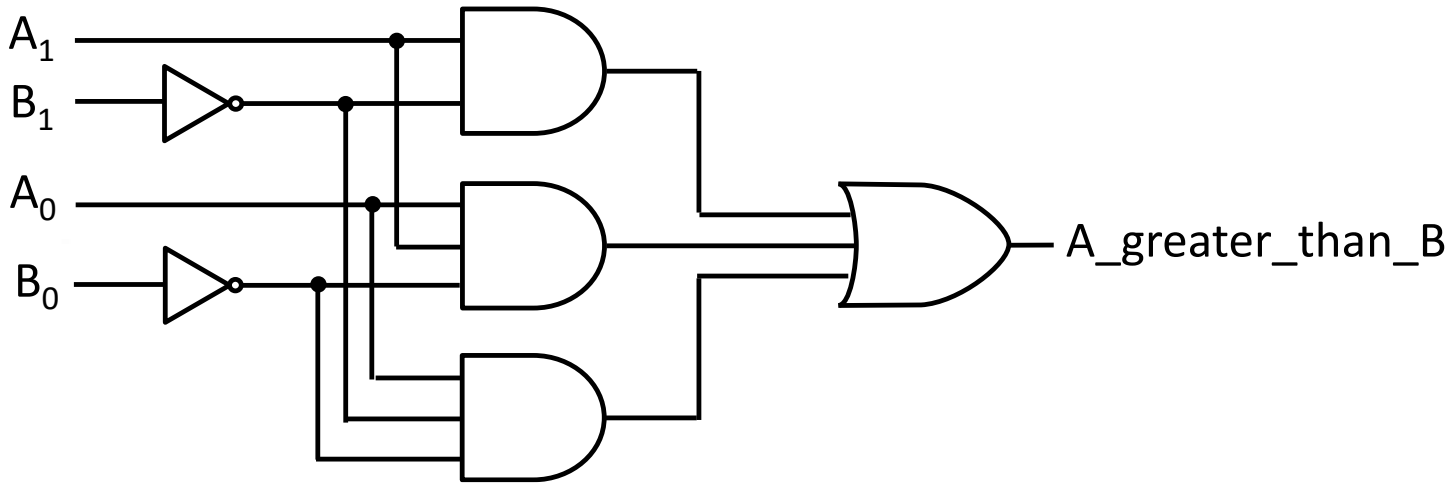
Constant

- Constants have **fixed values** and they are characterized by a specific data type
- They are used to **facilitate updating / modifications of the code** and improve its readability
- They must be defined before the keyword 'begin' inside architecture. It is good practice to use symbolic names for the constants
- Syntax:

```
constant constant1Name: typeName := constant1Value;
```

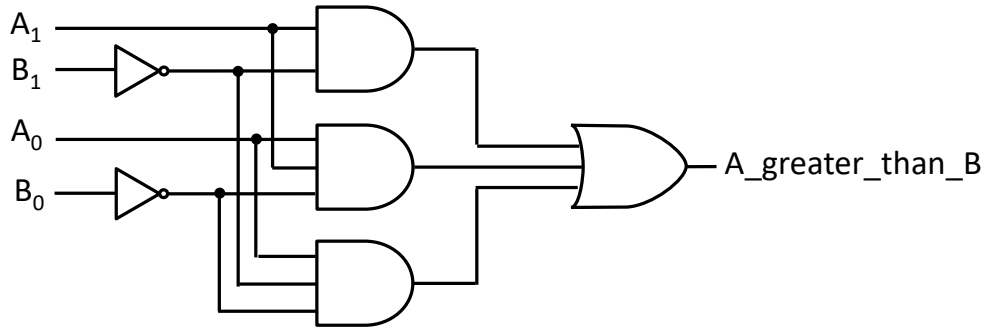
2-bit Comparator: Structural Architecture

```
architecture structural of comparator_greater_than_structural is

 component NOT1
  port(in1: in std_logic;
    out1: out std_logic);
 end component;
 component AND2
  port(in1, in2: in std_logic;
   out1: out std_logic);
 end component;
 component AND3
  port(in1, in2, in3: in std_logic;
   out1: out std_logic);
 end component;
 component OR3
  port(in1, in2, in3 : in std_logic;
    out1: out std_logic);
 end component;
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 inv_0: NOT1 port map (in1 => B(0), out1 => B0_n);
 inv_1: NOT1 port map (B(1), B1_n);
 and_0: AND2 port map (A(1), B1_n, and0_out);
 and_1: AND3 port map (A(1), A(0), B0_n, and1_out);
 and_2: AND3 port map (A(0), B1_n, B0_n, and2_out);
 or0: OR3 port map (and0_out,and1_out,and2_out, A_greater_than_B);
end structural;
```



2-bit Comparator: Structural Architecture

```
architecture structural of comparator_greater_than_structural is

component NOT1
 port(in1: in std_logic;
   out1: out std_logic);
end component;
component AND2
 port(in1, in2: in std_logic;
  out1: out std_logic);
end component;
component AND3
 port(in1, in2, in3: in std_logic;
  out1: out std_logic);
end component;
component OR3
 port(in1, in2, in3 : in std_logic;
   out1: out std_logic);
end component;
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 inv_0: NOT1 port map (in1 => B(0), out1 => B0_n);
 inv_1: NOT1 port map (B(1), B1_n);
 and_0: AND2 port map (A(1), B1_n, and0_out);
 and_1: AND3 port map (A(1), A(0), B0_n, and1_out);
 and_2: AND3 port map (A(0), B1_n, B0_n, and2_out);
 or0: OR3 port map (and0_out,and1_out,and2_out, A_greater_than_B);
end structural;
```

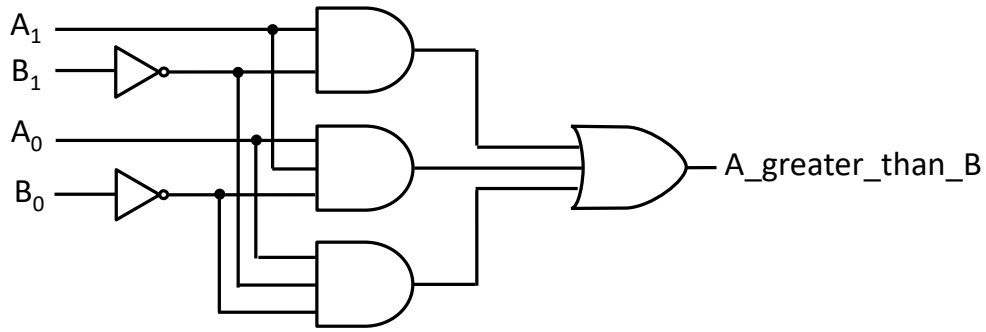


Declaration of components (logic gates)

The components will then be instantiated in the design: inverter, 2- or 3-input AND gate, 3-input OR gate

2-bit Comparator: Structural Architecture

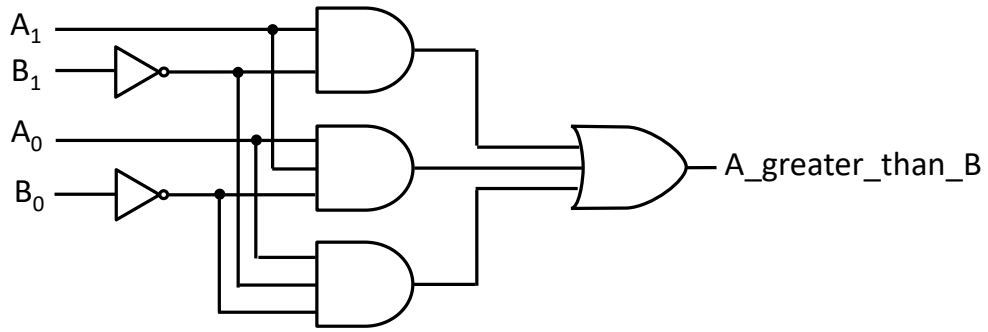
```
architecture structural of comparator_greater_than_structural is

 component NOT1
  port(in1: in std_logic;
    out1: out std_logic);
 end component;
 component AND2
  port(in1, in2: in std_logic;
   out1: out std_logic);
 end component;
 component AND3
  port(in1, in2, in3: in std_logic;
   out1: out std_logic);
 end component;
 component OR3
  port(in1, in2, in3 : in std_logic;
    out1: out std_logic);
 end component;
 signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 inv_0: NOT1 port map (in1 => B(0), out1 => B0_n);
 inv_1: NOT1 port map (B(1), B1_n);
 and_0: AND2 port map (A(1), B1_n, and0_out);
 and_1: AND3 port map (A(1), A(0), B0_n, and1_out);
 and_2: AND3 port map (A(0), B1_n, B0_n, and2_out);
 or0: OR3 port map (and0_out,and1_out,and2_out, A_greater_than_B);
end structural;
```



Declaration of signals, i.e. the internal nodes of the circuit (outputs of the 2 inverters, outputs of the 3 AND gates)

2-bit Comparator: Structural Architecture

```
architecture structural of comparator_greater_than_structural is

 component NOT1
  port(in1: in std_logic;
    out1: out std_logic);
 end component;
 component AND2
  port(in1, in2: in std_logic;
   out1: out std_logic);
 end component;
 component AND3
  port(in1, in2, in3: in std_logic;
   out1: out std_logic);
 end component;
 component OR3
  port(in1, in2, in3 : in std_logic;
    out1: out std_logic);
 end component;
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 inv_0: NOT1 port map (in1 => B(0), out1 => B0_n);
 inv_1: NOT1 port map (B(1), B1_n);
 and_0: AND2 port map (A(1), B1_n, and0_out);
 and_1: AND3 port map (A(1), A(0), B0_n, and1_out);
 and_2: AND3 port map (A(0), B1_n, B0_n, and2_out);
 or0: OR3 port map (and0_out,and1_out,and2_out, A_greater_than_B);
end structural;
```



These are concurrent statements, i.e. they are all evaluated at the same time, the order is not relevant!

Component instantiation and port mapping: signals are assigned to the component input and output ports

Libraries and Packages

- Libraries are collections of packages, which, in turn contain VHDL code (e.g. components, constants) that can be reused in different projects
- For example, there are IEEE standard packages for arithmetic functions , but also packages created by Synopsys, before IEEE standardization, are frequently used
  - Depending on the used package, the way in which data are handled in arithmetic operations changes. For example: std_logic_vector can be thought of as an array of signed or unsigned numbers

Package Synopsys	IEEE package	Features
`Std_logic_arith`	`Numeric_std`	Explicit declarations or specific conversions are required
`Std_logic_signed`	`Numeric_std_signed`	Vectors are considered as signed numbers (represented in 2's complement notation)
`Std_logic_unsigned`	`Numeric_std_unsigned`	Vectors are considered as unsigned numbers

Libraries and Packages

Syntax for using libraries and packages within a design:

```
library library1Name, library2Name, …;
use libraryName.packageName.all;
```

Note

- `.all` it allows you to use everything that is defined in that package
- The declaration of libraries and packages needs to be **repeated before each entity**!

2-bit Comparator: VHDL

```
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
```

- Declaration of libraries and packages used in the design
  - The package `std_logic_1164`, belonging to the IEEE library, contains the `std_logic` and `std_logic_vector` data types
  - The package `func_prims`, belonging to the `lcdf_vhdl` library (created by the textbook authors), contains the VHDL implementation of the basic logic gates, registers, and flip-flops (available on the textbook website)

Example: Creating a Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity NOT1 is
 port (in1 : in std_logic; out1 : out std_logic);
end NOT1;

architecture NOT1_impl of NOT1 is
begin
 out1 <= not in1;
end NOT1_impl;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
package MyPackage is
component NOT1 is
 port(in1 : in std_logic; out1 : out std_logic);
end component;
end;
```

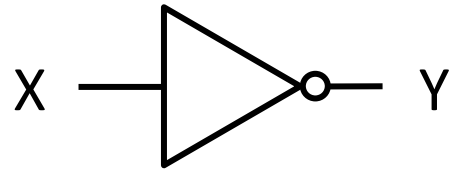
The names of the components and in, out ports must be the **same** as those appearing in the entity!

Component Instantiation

- A component can be instantiated in different ways inside a circuit in a VHDL design, we will study two of them
  1) **Component inside architecture** (already seen in the structural description of the comparator, see slide 36)
  2) **Direct instantiation** (more compact)

1) Component inside Architecture Instantiation

Inverter description:




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity inverter is
      port(X : in std_logic;
           Y : out std_logic);
end inverter;
architecture inverter_impl of inverter is
begin
      Y <= not X;
end inverter_impl;
```

1) Component inside Architecture Instantiation

Component is declared inside architecture (before `begin`), then the component is instantiated with port mapping:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TestCircuit is
    port ( input : in STD_LOGIC;
           output : out STD_LOGIC);
end TestCircuit;
architecture TestCircuit_impl of TestCircuit is
component inverter is
    port (X: in std_logic; Y: out std_logic);
end component;
begin
    INV1: inverter port map (input, output);
end TestCircuit_impl;
```

Same name and in / out ports that appear in the entity of the component

Positional association of the ports

1) Component inside Architecture Instantiation

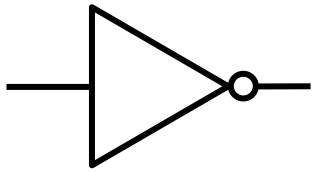
Component is declared inside architecture (before `begin`), then the component is instantiated with port mapping:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TestCircuit is
   port ( input : in STD_LOGIC;
          output : out STD_LOGIC);
end TestCircuit;
architecture TestCircuit_impl of TestCircuit is
component inverter is
   port (X: in std_logic; Y: out std_logic);
end component;
begin
   INV1: inverter port map (Y => output, X => input);
end TestCircuit_impl;
```

Nominal association of the ports: the order does not matter!

2) Direct Instantiation

Inverter description:



```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity inverter is
   port(X : in std_logic;
        Y : out std_logic);
end inverter;
architecture inverter_impl of inverter is
begin
   Y <= not X;
end inverter_impl;
```

2) Direct Instantiation

Instantiation is done specifying the entity and architecture, the component declaration is not required:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TestCircuit is
   port ( input : in STD_LOGIC;
          output : out STD_LOGIC);
end TestCircuit;

architecture TestCircuit_impl of TestCircuit is
begin
   INV1: entity work.inverter(inverter_impl) port map (input, output);
end TestCircuit_impl;
```

The reserved word work indicates that the component is defined in the current library

2-bit Comparator: Dataflow VHDL

```
library ieee;
use ieee.std_logic_1164.all;
```

Libraries / packages

```
entity comparator_greater_than_dataflow is
  port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end comparator_greater_than_dataflow;
```

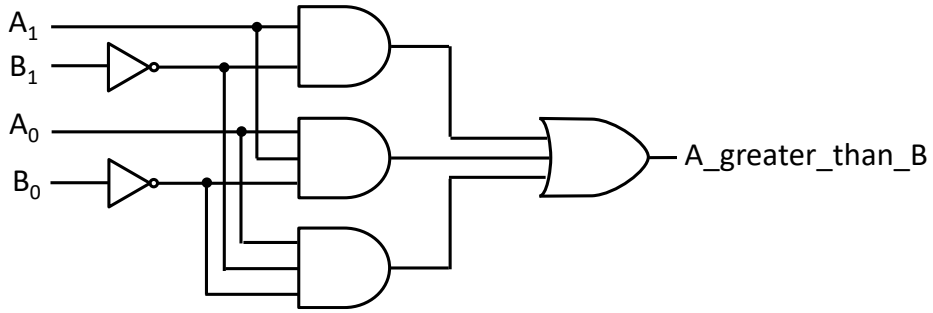
Entity

```
architecture dataflow of comparator_greater_than_dataflow is
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 B1_n <= not B(1);
 B0_n <= not B(0);
 and0_out <= A(1) and B1_n;
 and1_out <= A(1) and A(0) and B0_n;
 and2_out <= A(0) and B1_n and B0_n;
 A_greater_than_B <= and0_out or and1_out or and2_out;
end dataflow;
```

Architecture

Dataflow Architecture

```
architecture dataflow of comparator_greater_than_dataflow is
signal B1_n, B0_n, and0_out, and1_out, and2_out: std_logic;
begin
 B1_n <= not B(1);
 B0_n <= not B(0);
 and0_out <= A(1) and B1_n;
 and1_out <= A(1) and A(0) and B0_n;
 and2_out <= A(0) and B1_n and B0_n;
 A_greater_than_B <= and0_out or and1_out or and2_out;
end dataflow;
```



- The details of the circuit are not specified (there are no components), the circuit functionality is described through the flow of data
- The code is composed of **Boolean equations** expressed with **concurrent assignments** (i.e. that are performed all **in parallel**): whenever the right member of an assignment changes, the new value is assigned to the left member
- The order of execution of the statements does not depend on the order of their appearance on the VHDL code, but on the order of the changes of the signals on the right-hand of the statements. The **circuit remains the same if the order of the assignments is changed!**

2-bit Comparator: Behavioral (1) VHDL

```
library ieee;
use ieee.std_logic_1164.all;
```

Libraries / packages

```
entity comparator_greater_than_behavioral is
  port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end comparator_greater_than_behavioral;
```

Entity

```
architecture when_else of comparator_greater_than_behavioral is
begin
 A_greater_than_B <= '1' when (A > B) else
                     '0';
end when_else;
```

Architecture with **when-else statetement**

2-bit Comparator: Behavioral (2) VHDL

```
library ieee;
use ieee.std_logic_1164.all, ieee.std_logic_unsigned .all;
```

Libraries / packages

```
entity comparator_greater_than_behavioral2 is
  port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end comparator_greater_than_behavioral2;
```

Entity

```
architecture with_select of comparator_greater_than_behavioral2 is
begin
 with A select
 A_greater_than_B <= '0' when "00",
           B(0) nor B(1) when "01",
           not B(1) when "10",
           B(0) nand B(1) when "11",
           'X' when others;
end with_select;
```

Architecture with **with-select** statement

2-bit Comparator: Truth Table

A_1	A_0	B_1	B_0	A_greater_than_B
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**0**
1	0	1	1	**0**
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**0**

'When-else' and 'With-select' Statements

- 'When-else' and 'With-select' are concurrent constructs (statements are executed in parallel), mostly used in behavioral descriptions
  - **When-else**: used for conditional assignments of signals with priority logic, to evaluate **cascade priority**
  - **With-select**: used for assignment of selected signals with parallel logic, to evaluate different **choices with the same priority**
- Example: the PC data bus receives data from different devices (from the processor, memory, disk, I /O) - each of these devices controls the same bus, we decide which one is selected

'When-else' Statement

Syntax:

```
name <= expression1 when condition1 else
        { expression2 when condition2 else }
          expression3;
```

Priority logic:

- Condition1 has the priority, regardless of whether or not condition2 is met
- If condition1 is satisfied, the next conditions are not evaluated and the first choice is selected
- This statement allows decisions on multiple signals (multiple ‘when’ statements can be nested, each conditioned on a different signal)

'With-select' Statement

Syntax:

```
with expressionOfChoice select
      Z <= expression1 when choice 1,
         { expression2 when choice2,
           expressionN when choiceN, }
           expressionX when others;
```

Parallel logic:

- All conditions have the same priority
- All cases must be covered by one (and only one) choice
- All choices must be present (it is mandatory to enter the default condition 'when others')
- With-select statement allows decisions based on a single Boolean condition (typically corresponding to less complex structures than the when-else)

Testbench

- The testbench is a VHDL code to **simulate the behavior of a circuit (**DUT, Device Under Test) by applying proper signals to the inputs
  - It can be thought of as a testbench that applies stimuli to the DUT and (optionally) checks if the DUT behaves in the expected way
- The testbench has simulation only purposes. The code **will not be synthesized to hardware**, but it is used as a validation of the design

2-bit Comparator: Testbench

```
library ieee;
use ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
```

Libraries / packages

```
entity greater_testbench is
end greater_testbench;
```

Entity (empty: there are no input and output ports)

```
architecture testbench of greater_testbench is
signal A, B: std_logic_vector (1 downto 0);
signal struct_out: std_logic;
component comparator_greater_than_structural is
   port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end component;
begin
u1: comparator_greater_than_structural port map(A,B, struct_out);
tb: process
begin
 A <= "10";
 B <= "00";
 wait for 10 ns;
 B <= "01";
 wait for 10 ns;
 B <= "10";
 wait for 10 ns;
 B <= "11";
 wait; -- halt the process
end process;
end testbench;
```

Architecture

2-bit Comparator: Testbench

```
architecture testbench of greater_testbench is
signal A, B: std_logic_vector (1 downto 0);
signal struct_out: std_logic;
component comparator_greater_than_structural is
   port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end component;
begin
u1: comparator_greater_than_structural port map(A,B, struct_out);
tb: process
begin
 A <= "10";
 B <= "00";
 wait for 10 ns;
 B <= "01";
 wait for 10 ns;
 B <= "10";
 wait for 10 ns;
 B <= "11";
 wait; -- halt the process
end process;
end testbench;
```

Declaration of signals to stimulate DUT inputs and signals that will be connected to the output of the DUT

2-bit Comparator: Testbench

```
architecture testbench of greater_testbench is
signal A, B: std_logic_vector (1 downto 0);
signal struct_out: std_logic;
component comparator_greater_than_structural is
   port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end component;
begin
u1: comparator_greater_than_structural port map(A,B, struct_out);
tb: process
begin
 A <= "10";
 B <= "00";
 wait for 10 ns;
 B <= "01";
 wait for 10 ns;
 B <= "10";
 wait for 10 ns;
 B <= "11";
 wait; -- halt the process
end process;
end testbench;
```

Declaration of DUT component

Component instantiation (with positional association): A, B are connected to the inputs, struct_out is connected to the output

2-bit Comparator: Testbench

```
architecture testbench of greater_testbench is
signal A, B: std_logic_vector (1 downto 0);
signal struct_out: std_logic;
component comparator_greater_than_structural is
   port (A: in std_logic_vector(1 downto 0);
   B: in std_logic_vector(1 downto 0);
   A_greater_than_B: out std_logic);
end component;
begin
u1: comparator_greater_than_structural port map(A,B, struct_out);
tb: process
begin
 A <= "10";
 B <= "00";
 wait for 10 ns;
 B <= "01";
 wait for 10 ns;
 B <= "10";
 wait for 10 ns;
 B <= "11";
 wait; -- halt the process
end process;
end testbench;
```

Process: contains the **stimuli to the DUT**, i.e. combinations of values assigned to the inputs one after the other, waiting 10 ns, i.e. 10^{-9} s (used for simulation) between them.
Inside a process, the **statements are performed sequentially and non-concurrently**!
The reserved word `wait` suspends the process (for a certain time or indefinitely)

Process

- A process is a concurrent statement (in parallel to the others present in architecture), in which the **statements are executed sequentially** (one after the other) in the order in which they are written
- A process starts immediately if a sensitivity list is not specified (previous example) or when one of the signals in the sensitivity list changes (see next slide)
- The value of the signals assigned within a process is **updated only when the process is suspended** (either because all statements in the process have been executed or because there is a ‘wait’ instruction)
  - If there are multiple assignments to the same signal, the signal takes the last value that has been assigned to it in the process

Process

Syntax:

```
process [ ( <sensitivity_list> ) ]
    signal ...
    component ...
begin
    <sequential_statements>
end process;
```

Process: Sensitivity List

- In the testbench we have seen the use of a process statement without a sensitivity list
- The sensitivity list is optional. If present, it contains the signals to which change the process is sensitive
  - If the sensitivity list is empty, the process runs continuously, in absence of explicit statements to stop the process (e.g. ‘wait’)
- If present, the sensitivity list can only contain signals that can be read (inputs or signals, not outputs!)
- We will better study the use of process statement in future lectures

Disclaimer

Figures from *Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano | Kime| Martin

© 2016 Pearson Education, Ltd