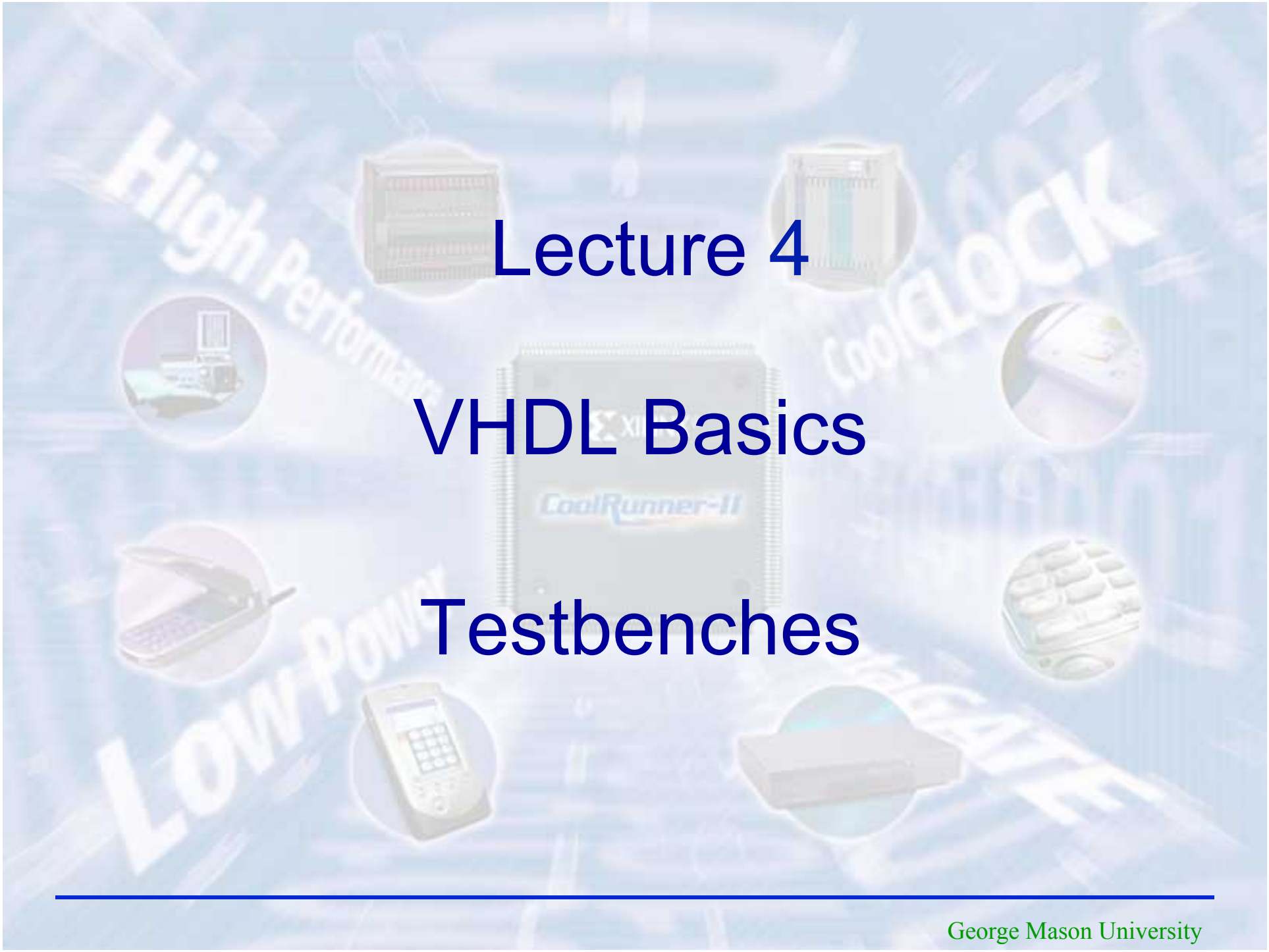


Lecture 4

VHDL Basics

Testbenches

Required reading

- P. Chu, *RTL Hardware Design using VHDL*

Chapter 2, Overview of Hardware Description Languages

Chapter 3, Basic Language Constructs of VHDL

Recommended reading

- Wikipedia – The Free On-line Encyclopedia

VHDL - ***http://en.wikipedia.org/wiki/VHDL***

Verilog - ***http://en.wikipedia.org/wiki/Verilog***

Accellera - ***http://en.wikipedia.org/wiki/Accellera***

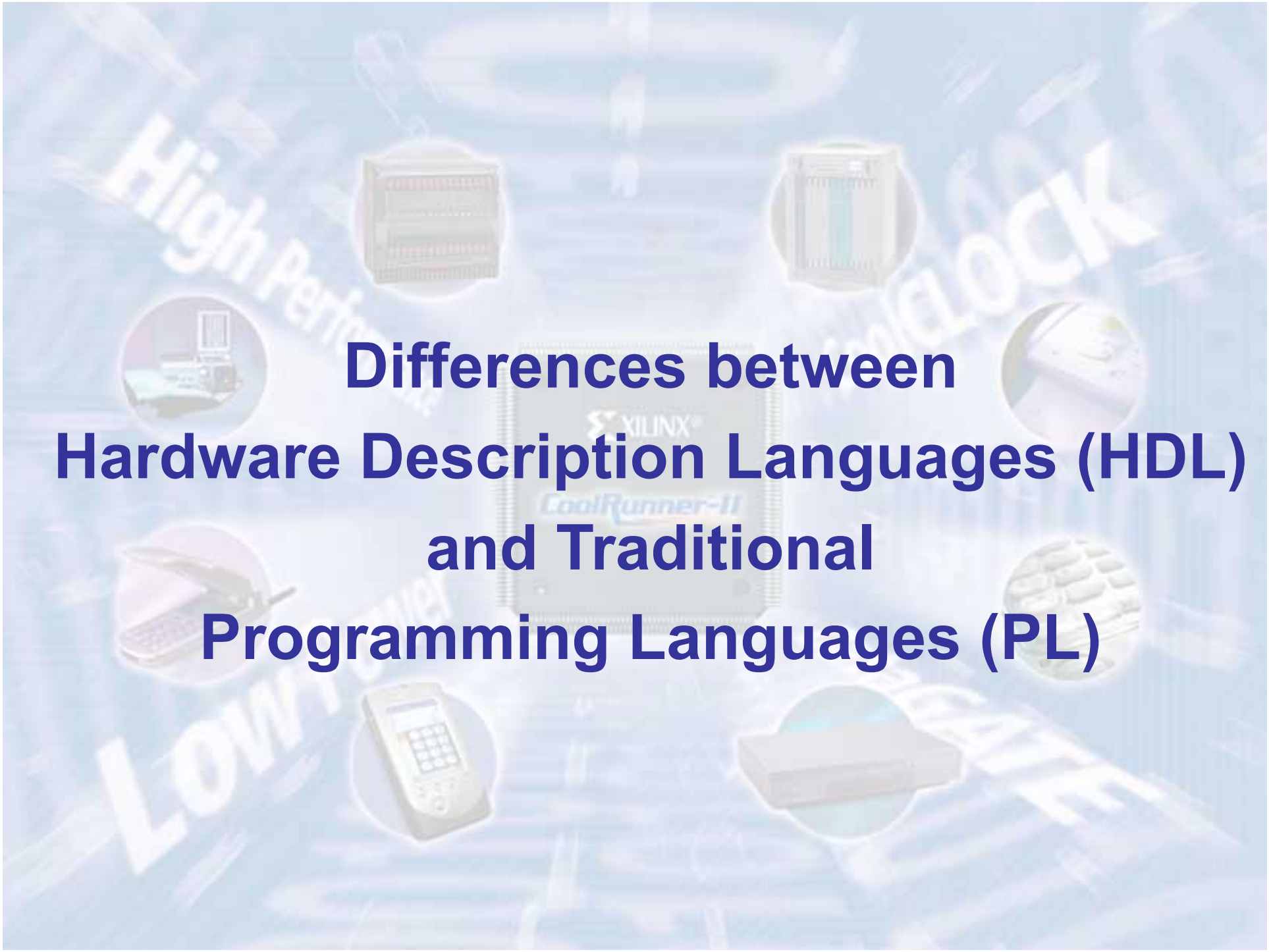
Required reading

- Sundar Rajan, *Essential VHDL: RTL Synthesis Done Right*

  Chapter 14, starting from "Design Verification"

Steps of the Design Process

1. Text description
2. Interface
3. Pseudocode
4. **Block diagram of the Datapath**
5. **Interface with the division into the Datapath and the Controller**
6. ASM chart of the Controller
7. RTL VHDL code of the Datapath, the Controller, and the Top Unit
8. **Testbench of the Datapath**, the Controller, **and the Top Unit**
9. **Functional simulation and debugging**
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing



Differences between Hardware Description Languages (HDL) and Traditional Programming Languages (PL)

Traditional PL

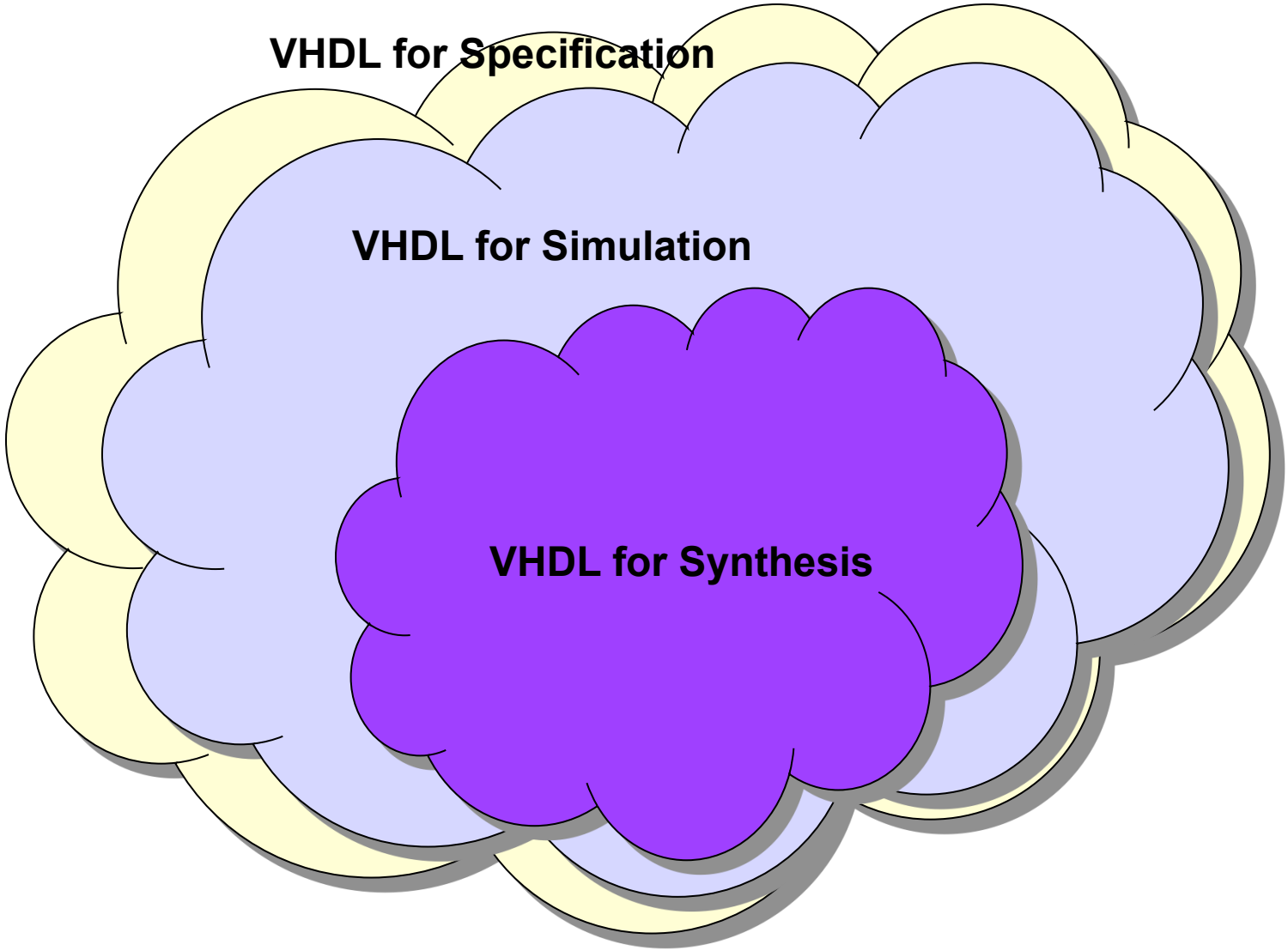
- Modeled after a sequential process
  - Operations performed in a sequential order
  - Help human's thinking process to develop an algorithm step by step
  - Resemble the operation of a basic computer model

HDL

- Characteristics of digital hardware
  - Connections of parts
  - Concurrent operations
  - Concept of propagation delay and timing
- Characteristics cannot be captured by traditional PLs
- Require new languages: HDL

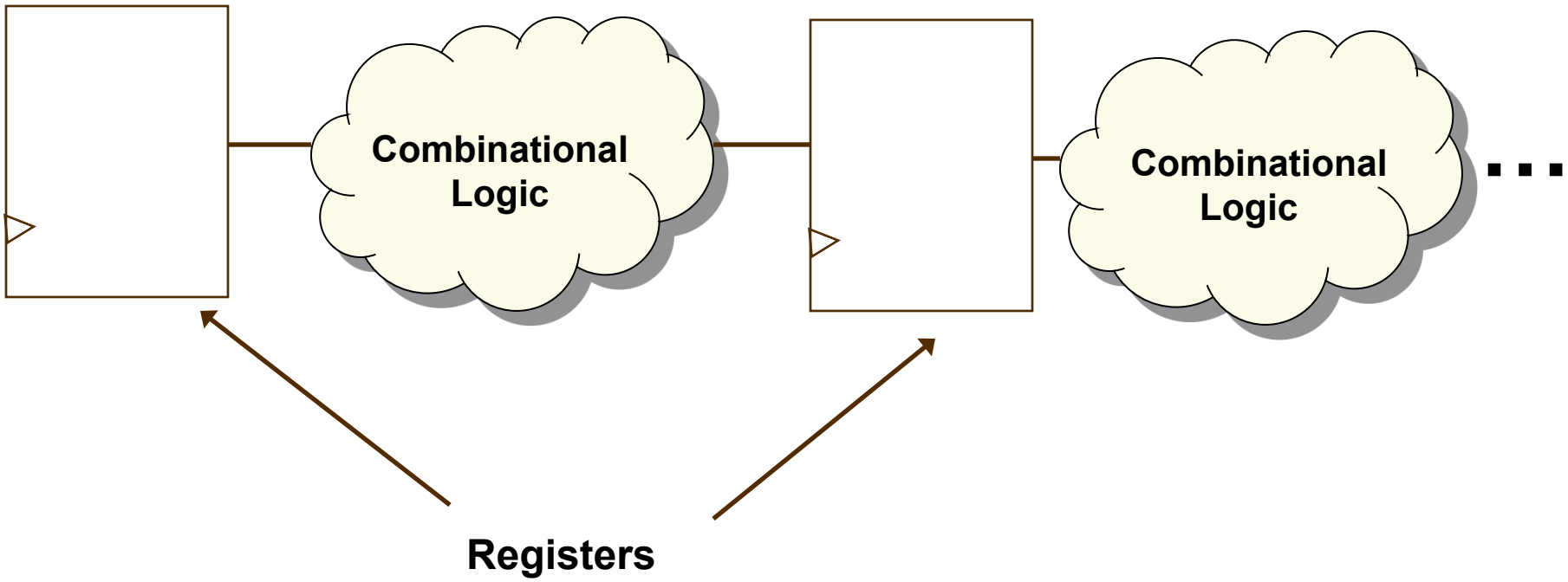
Use of an HDL program

- Formal documentation
- Input to a simulator
- Input to a synthesizer

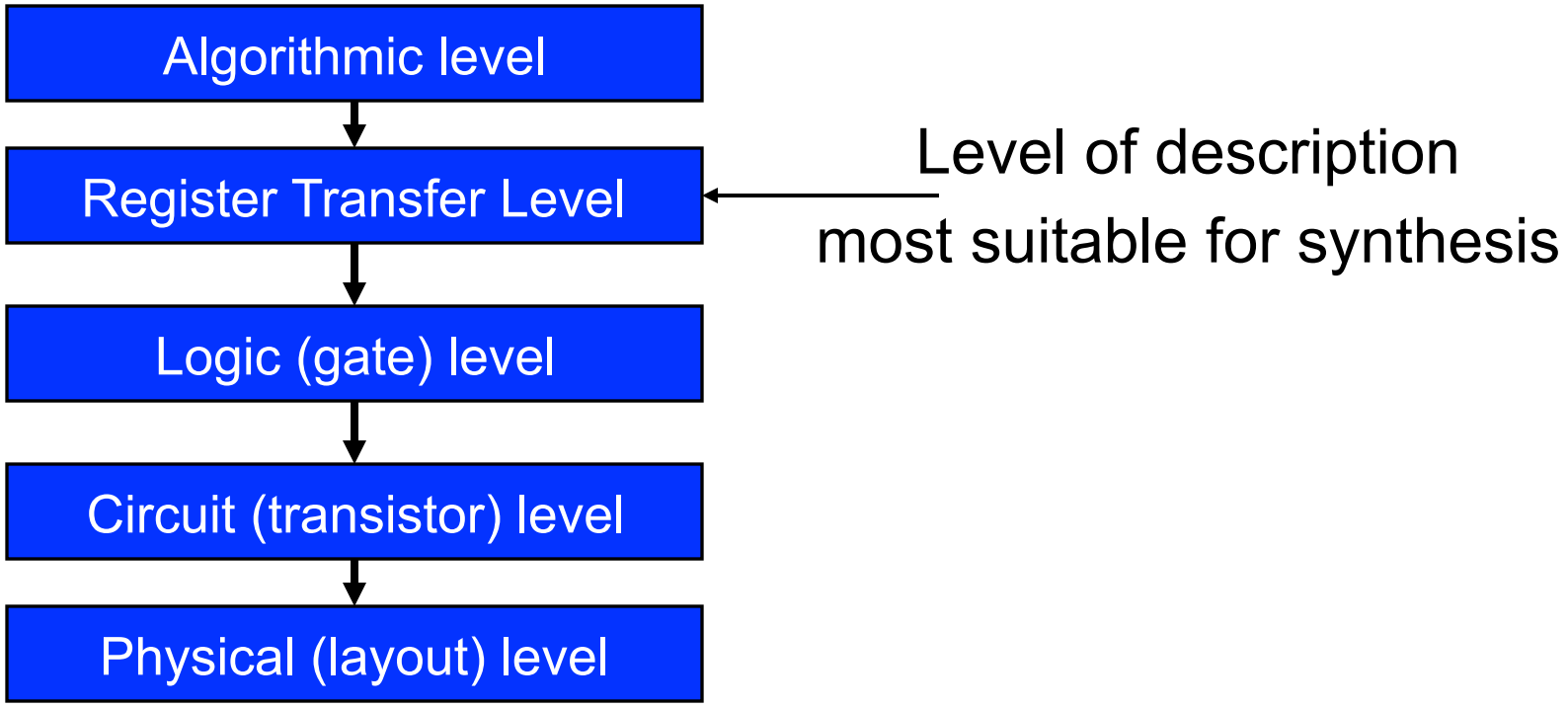

VHDL for Specification
VHDL for Simulation
VHDL for Synthesis

- Highlights of modern HDL:
  - Encapsulate the concepts of entity, connectivity, concurrency, and timing
  - Incorporate propagation delay and timing information
  - Consist of constructs for structural implementation
  - Incorporate constructs for behavioral description (sequential execution of traditional PL)
  - Describe the operations and structures in gate level and RT level.
  - Consist of constructs to support hierarchical design process

Register Transfer Level (RTL) Design Description



Levels of design description



Two HDLs used today

- VHDL and Verilog
- Syntax and ``appearance'' of the two languages are very different
- Capabilities and scopes are quite similar
- Both are industrial standards and are supported by most software tools

Brief History of VHDL

VHDL

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

Genesis of VHDL

State of art circa 1980

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

A Brief History of VHDL

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

  VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI standard

Four versions of VHDL

- Four versions of VHDL:
  - IEEE-1076 1987
  - IEEE-1076 1993 ← most commonly supported by CAD tools
  - IEEE-1076 2000 (minor changes)
  - IEEE-1076 2002 (minor changes)
  - IEEE-1076 2008

IEEE Extensions

- IEEE standard 1076.1 Analog and Mixed Signal Extensions (VHDL-AMS)
- IEEE standard 1076.2 VHDL Mathematical Packages
- IEEE standard 1076.3 Synthesis Packages
- IEEE standard 1076.4 VHDL Initiative Towards ASIC Libraries (VITAL)
- IEEE standard 1076.6 VHDL Register Transfer Level (RTL) Synthesis
- IEEE standard 1164 Multivalue Logic System for VHDL Model Interoperability
- IEEE standard 1029 VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES)


High Performance
Low Power
XILINX
Verilog

Verilog

- Simpler and syntactically different
  - C-like

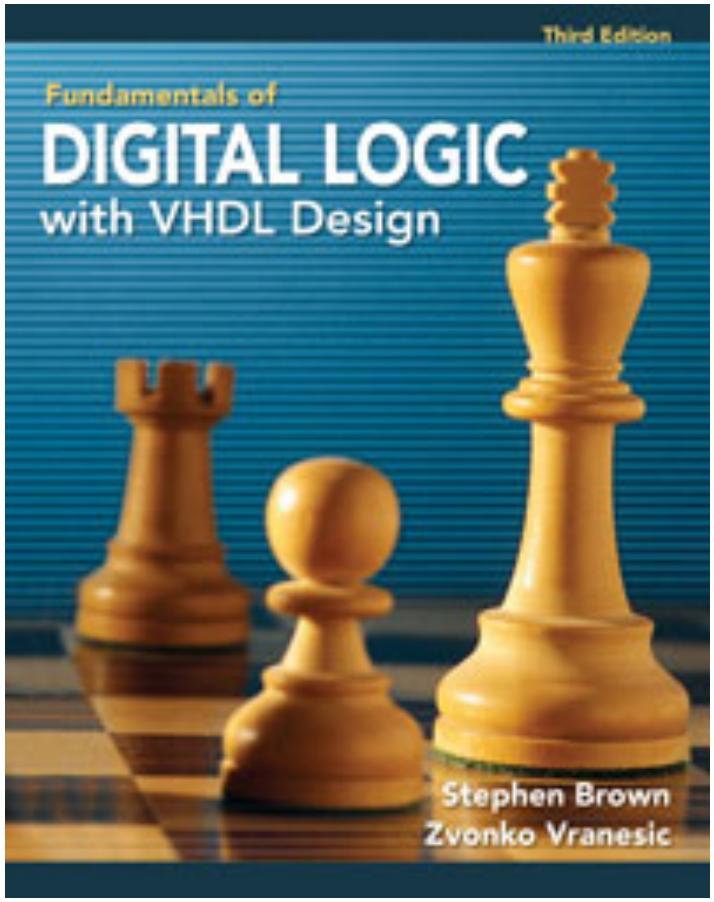
- Gateway Design Automation Co., 1985
- Gateway acquired by Cadence in 1990
- IEEE Standard 1364-1995
- Early *de facto* standard for ASIC programming

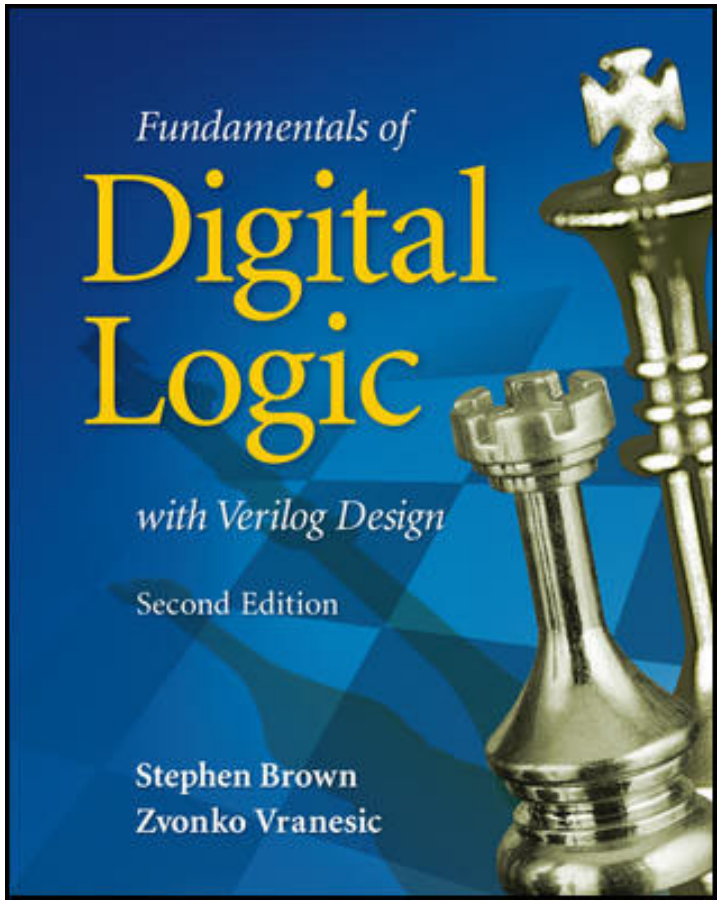
- Programming language interface to allow connection to non-Verilog code

VHDL vs. Verilog

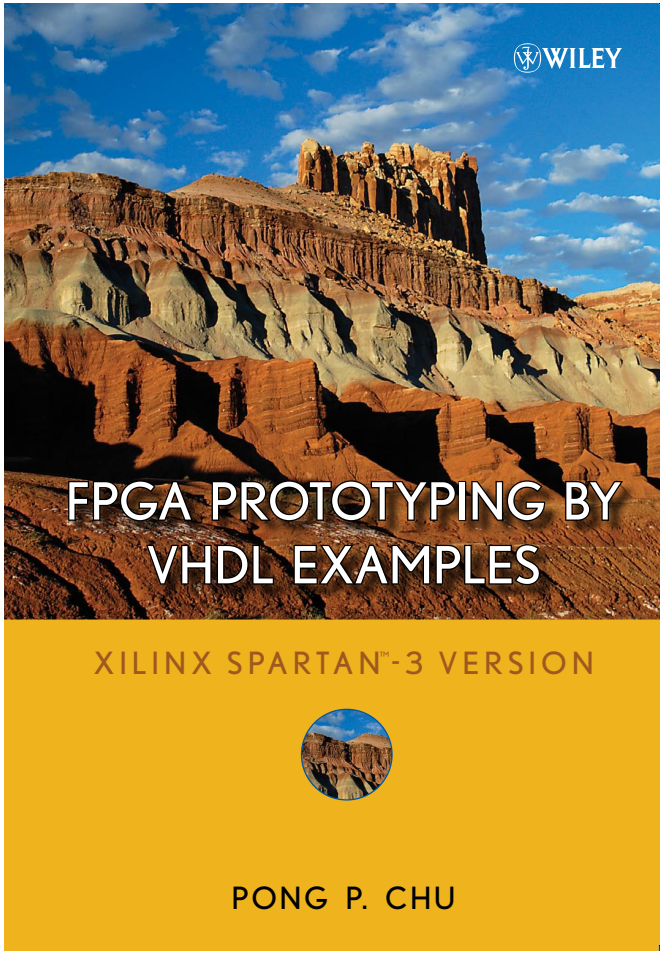
Government Developed	Commercially Developed
Ada based	C based
Strongly Type Cast	Mildly Type Cast
Case-insensitive	Case-sensitive
Difficult to learn	Easier to Learn
More Powerful	Less Powerful

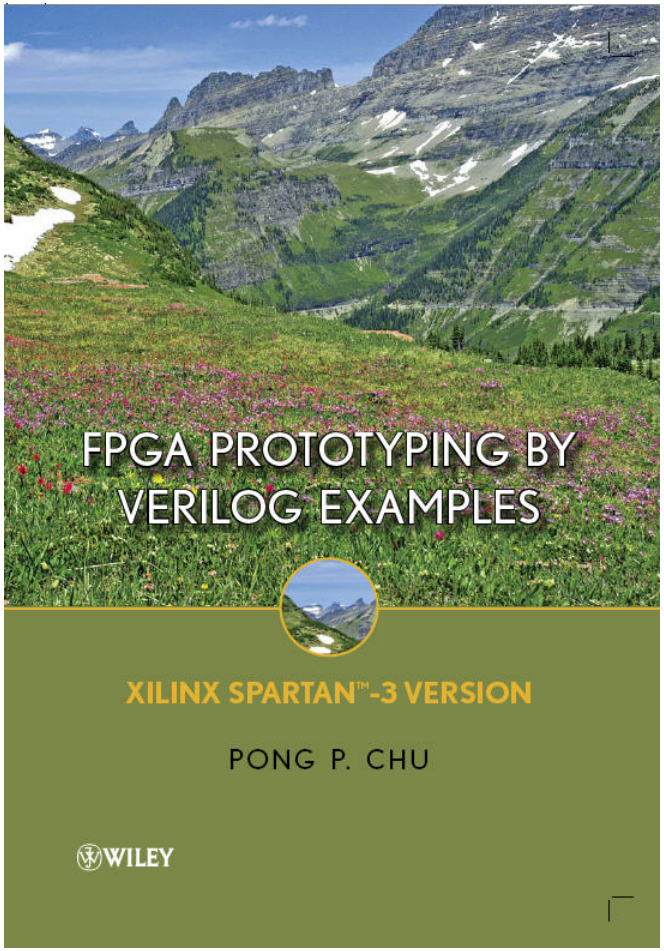
How to learn Verilog by yourself ?





How to learn Verilog by yourself ?





Features of VHDL and Verilog

- Technology/vendor independent
- Portable
- Reusable

VHDL Fundamentals

Naming and Labeling (1)

- VHDL is case insensitive

  Example:

  Names or labels

  `databus`

  `Databus`

  `DataBus`

  `DATABUS`

  are all equivalent

Naming and Labeling (2)

General rules of thumb (according to VHDL-87)

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

Valid or invalid?

7segment_display

A87372477424

Adder/Subtractor

/reset

And_or_gate

AND__OR__NOT

Kogge-Stone-Adder

Ripple&Carry_Adder

My adder

Extended Identifiers

Allowed only in VHDL-93 and higher:

1. Enclosed in backslashes
2. May contain spaces and consecutive underscores
3. May contain punctuation and reserved characters within a name (!, ?, ., &, +, -, etc.)
4. VHDL keywords allowed
5. Case sensitive

Examples:

/rdy/	/My design/	/!a/
/RDY/	/my design/	/-a/

Free Format

- VHDL is a “free format” language

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

  Example:

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

or

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

or

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

are all equivalent

Readability standards & coding style

Adopt readability standards based on one of the the two main textbooks:

Chu* or *Brown/Vranesic

Use coding style recommended in

OpenCores Coding Guidelines

linked from the course web page

Strictly enforced by the lab instructors and myself.
Penalty points may be enforced for not following these recommendations!!!

Comments

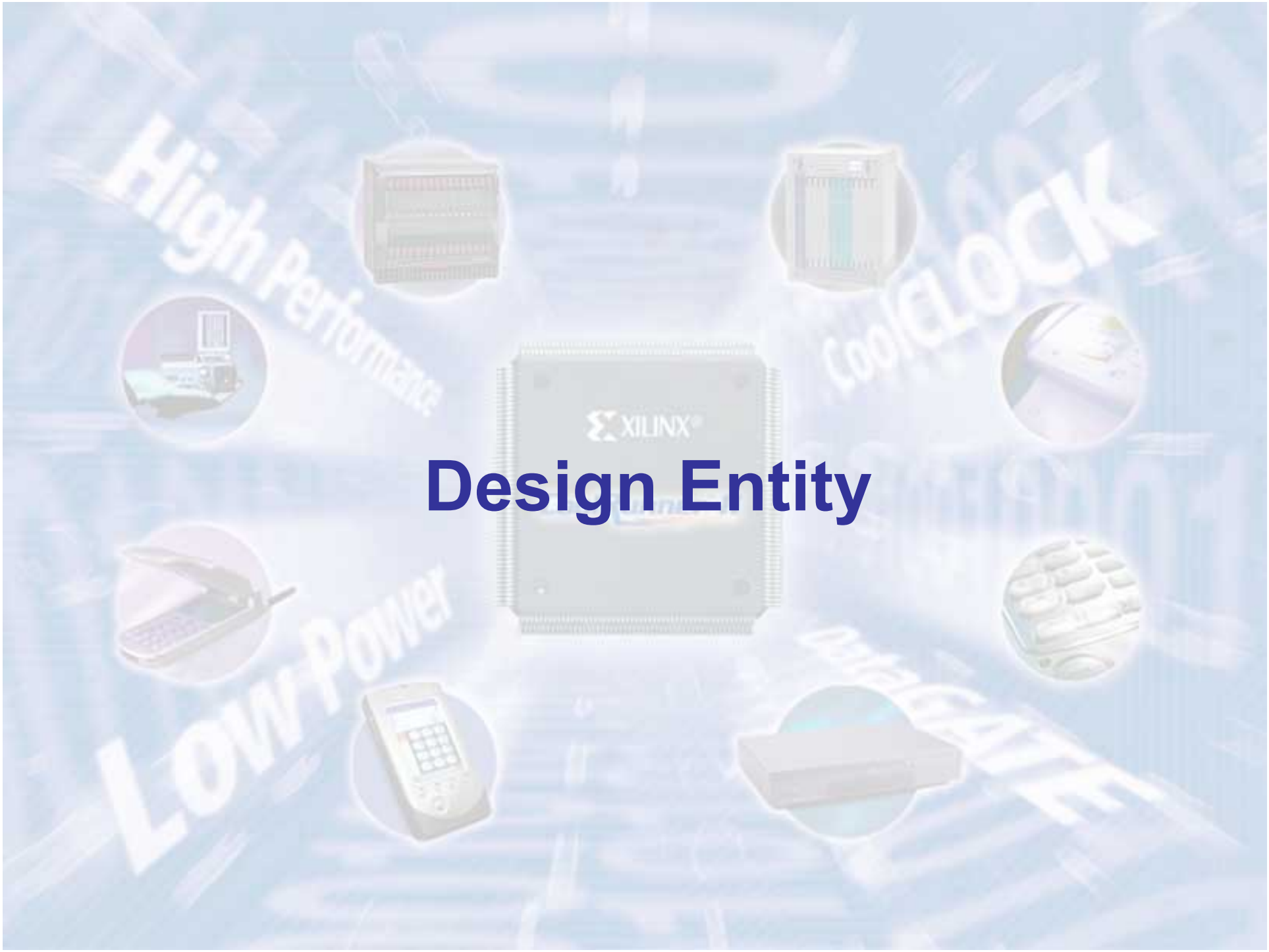
- Comments in VHDL are indicated with a “double dash”, i.e., “--”
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:

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

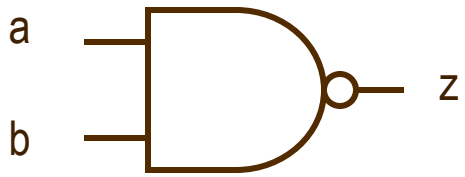
Comments

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



Design Entity

Example: NAND Gate



a	b	z
0	0	1
0	1	1
1	0	1
1	1	0

Example VHDL Code

- 3 sections to a piece of VHDL code
- File extension for a VHDL file is .vhd
- Name of the file should be the same as the entity name (nand_gate.vhd) [OpenCores Coding Guidelines]

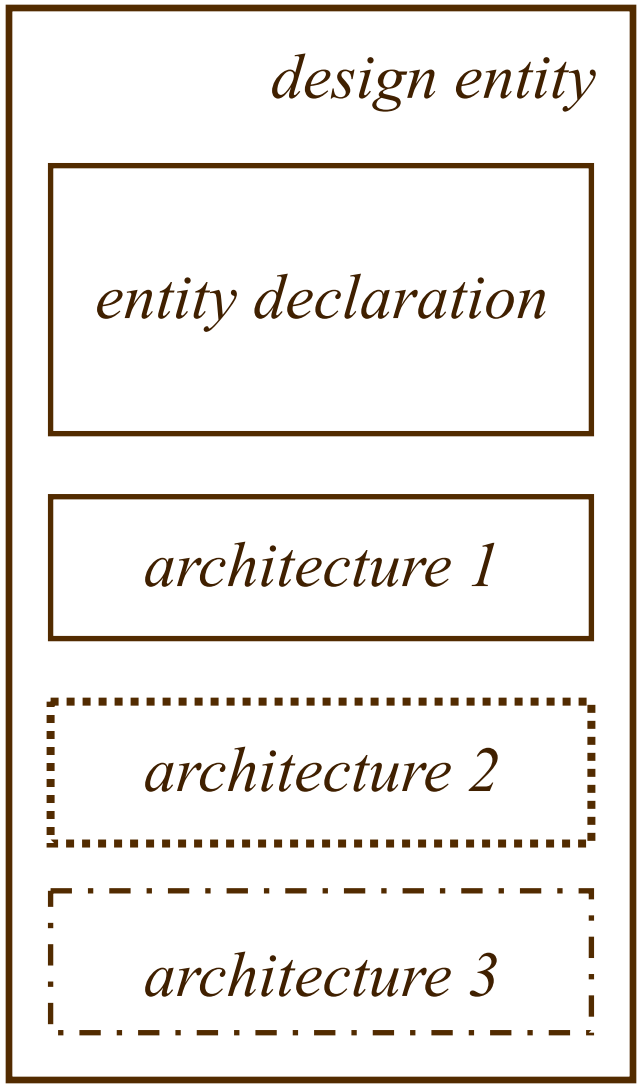
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

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

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

LIBRARY DECLARATION

ENTITY DECLARATION

ARCHITECTURE BODY

Design Entity

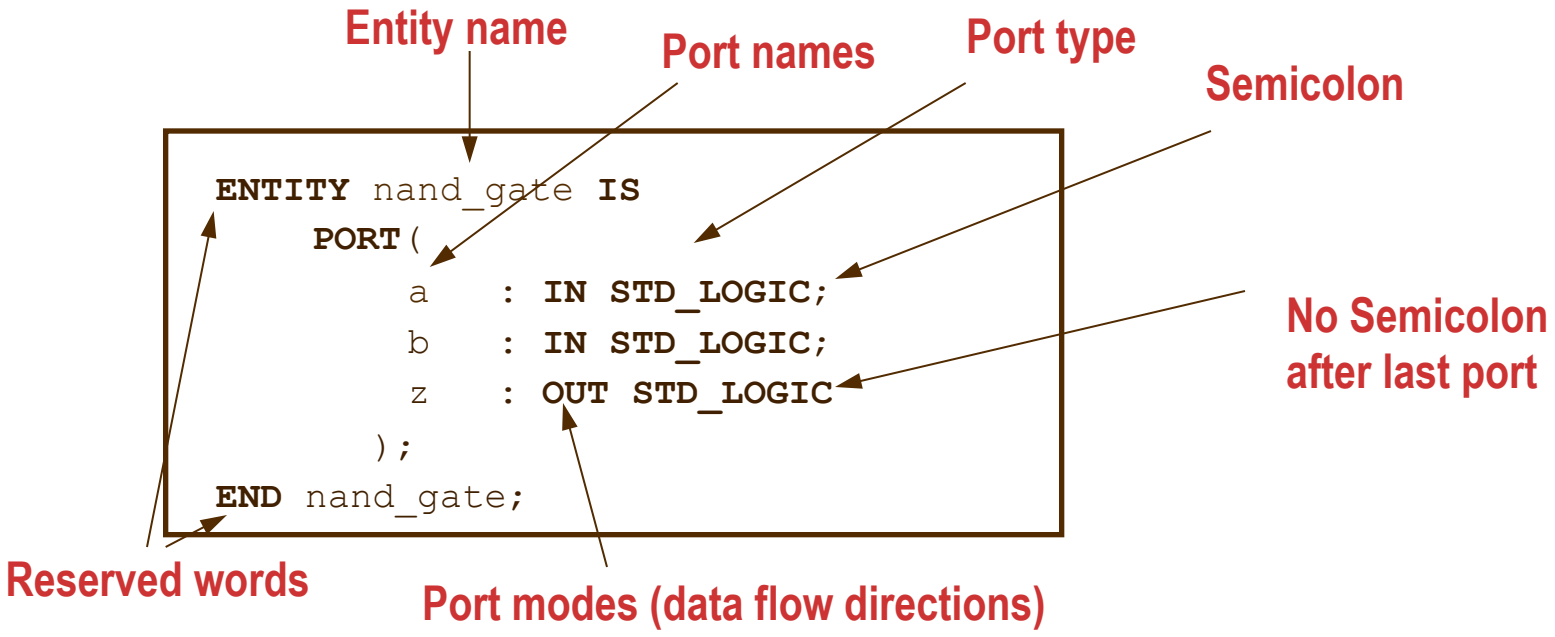


Design Entity - most basic building block of a design.

One *entity* can have many different *architectures.*

Entity Declaration

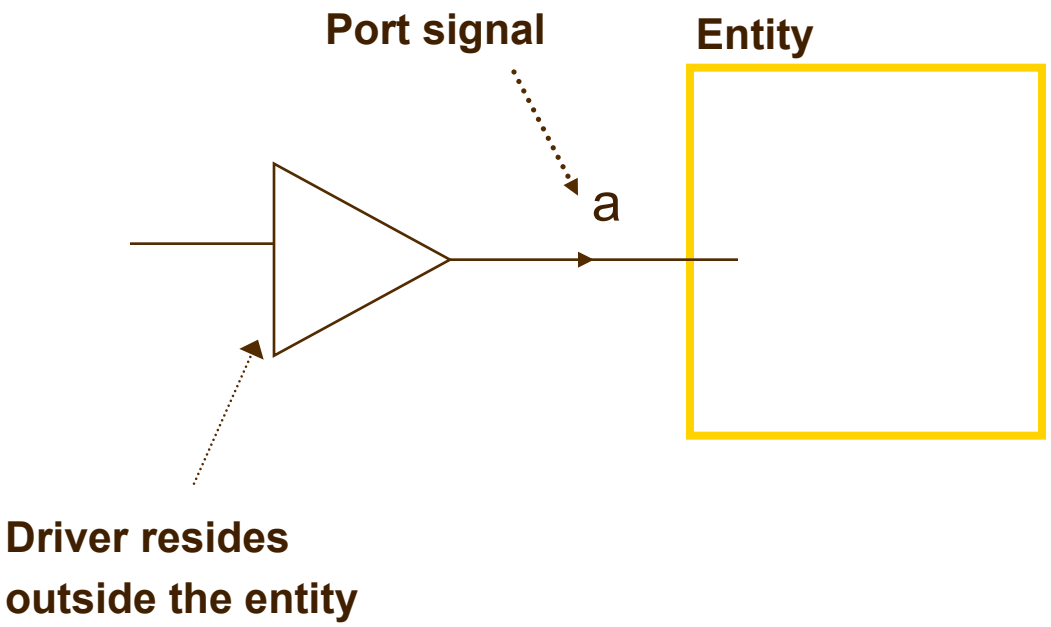
- Entity Declaration describes the interface of the component, i.e. input and output ports.



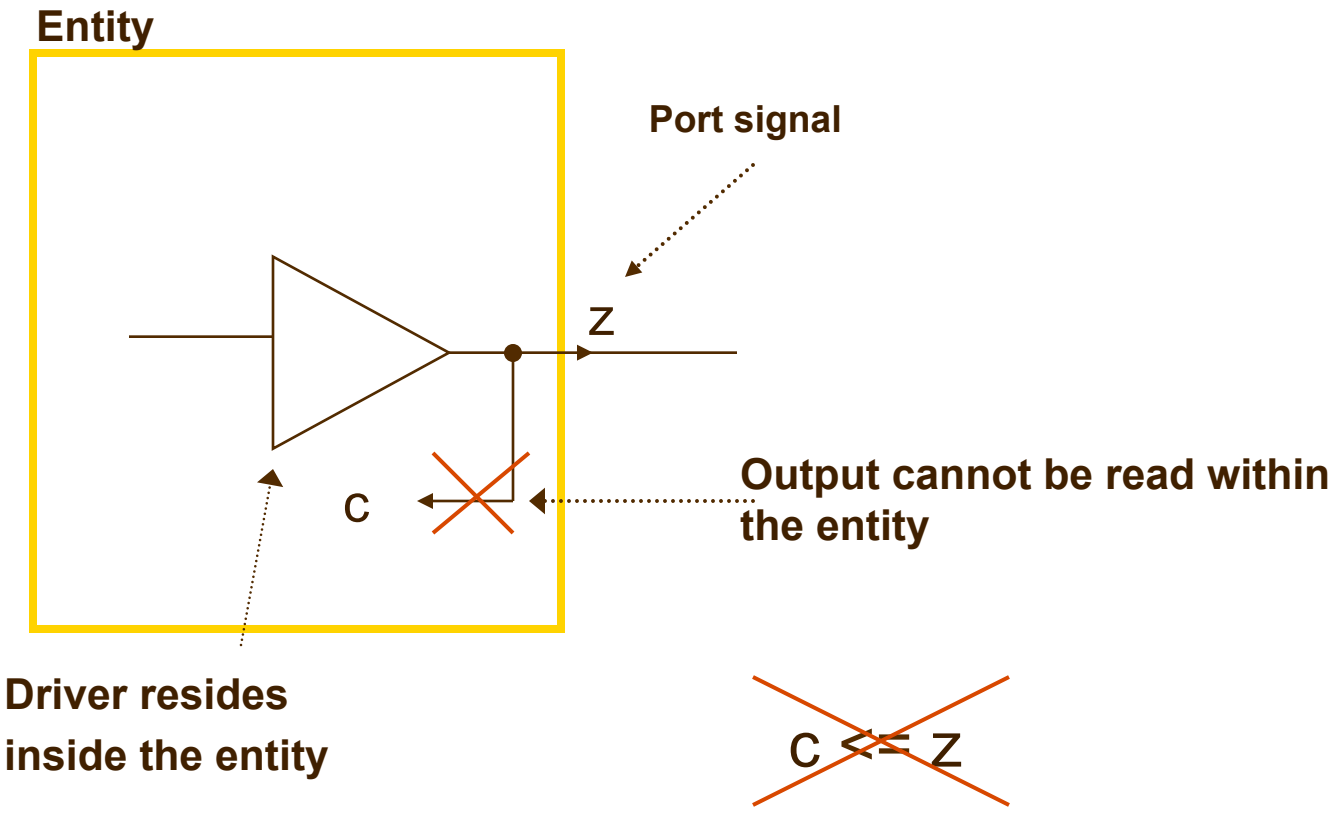
Entity declaration – simplified syntax

```
ENTITY entity_name IS
   PORT (
      port_name :  port_mode  signal_type;
      port_name :  port_mode  signal_type;
      ……..
      port_name :  port_mode  signal_type);
END entity_name;
```

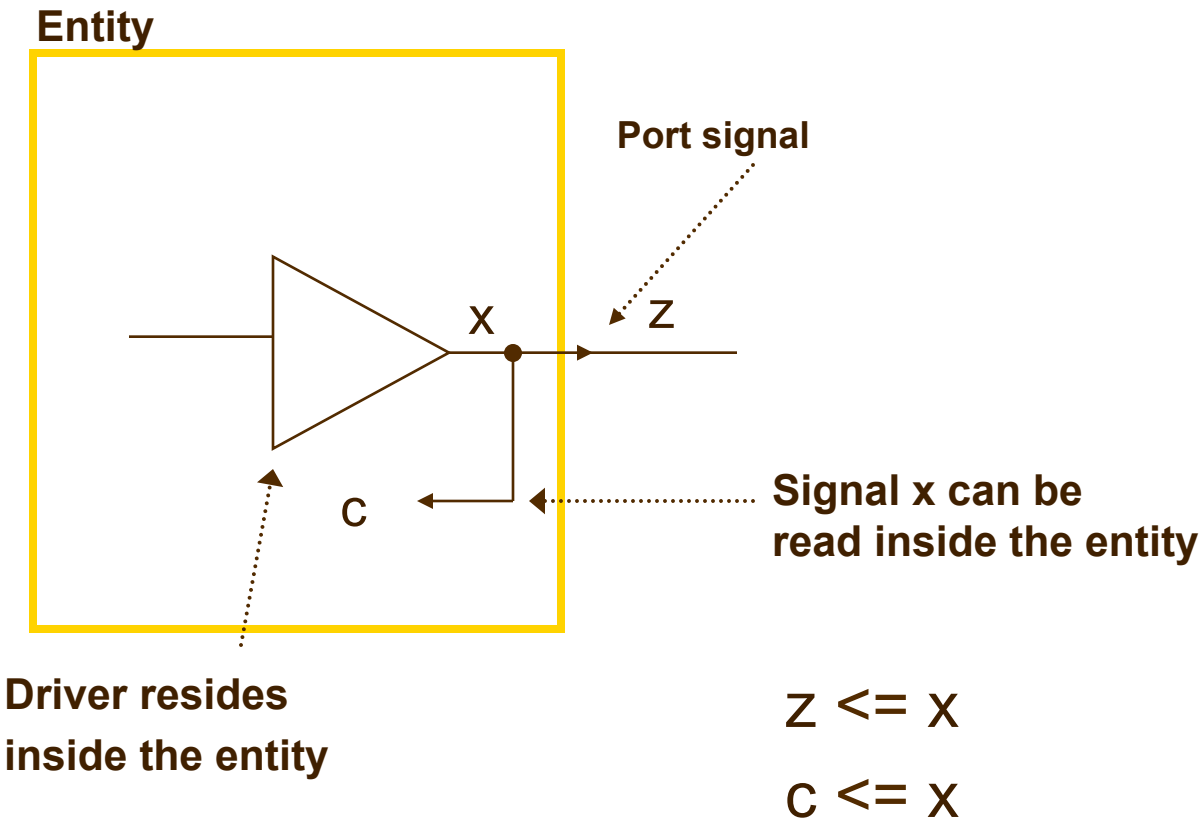
Port Mode IN



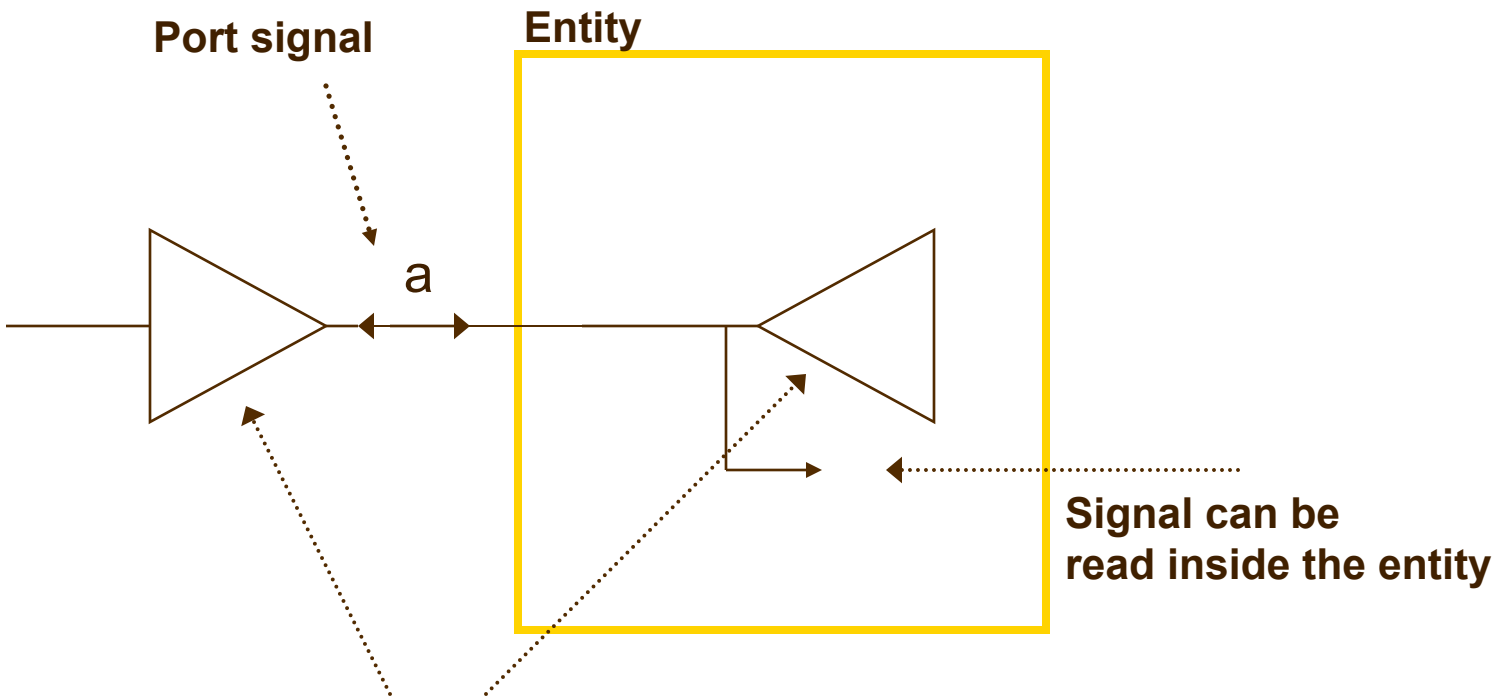
Port Mode OUT



Port Mode OUT (with extra signal)



Port Mode INOUT



Port Modes - Summary

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

- **In**: Data comes into this port and can only be read within the entity. It can appear **only on the right side** of a signal or variable assignment.

- **Out**: The value of an output port can only be updated within the entity. **It cannot be read**. It can only appear **on the left side** of a signal assignment.

- **Inout**: The value of a bi-directional port can be read and updated within the entity model. It can appear on **both sides** of a signal assignment.

Architecture (Architecture body)

- Describes an implementation of a design entity
- Architecture example:

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

Architecture – simplified syntax

```
ARCHITECTURE architecture_name OF entity_name IS
   [ declarations ]
BEGIN
    code
END architecture_name;
```

Entity Declaration & Architecture

nand_gate.vhd

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

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

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

Tips & Hints

Place each entity in a different file.

The name of each file should be exactly the same as the name of an entity it contains.

These rules are not enforced by all tools but are worth following in order to increase readability and portability of your designs

Tips & Hints

Place the declaration of each port, signal, constant, and variable in a separate line

These rules are not enforced by all tools but are worth following in order to increase readability and portability of your designs

Libraries

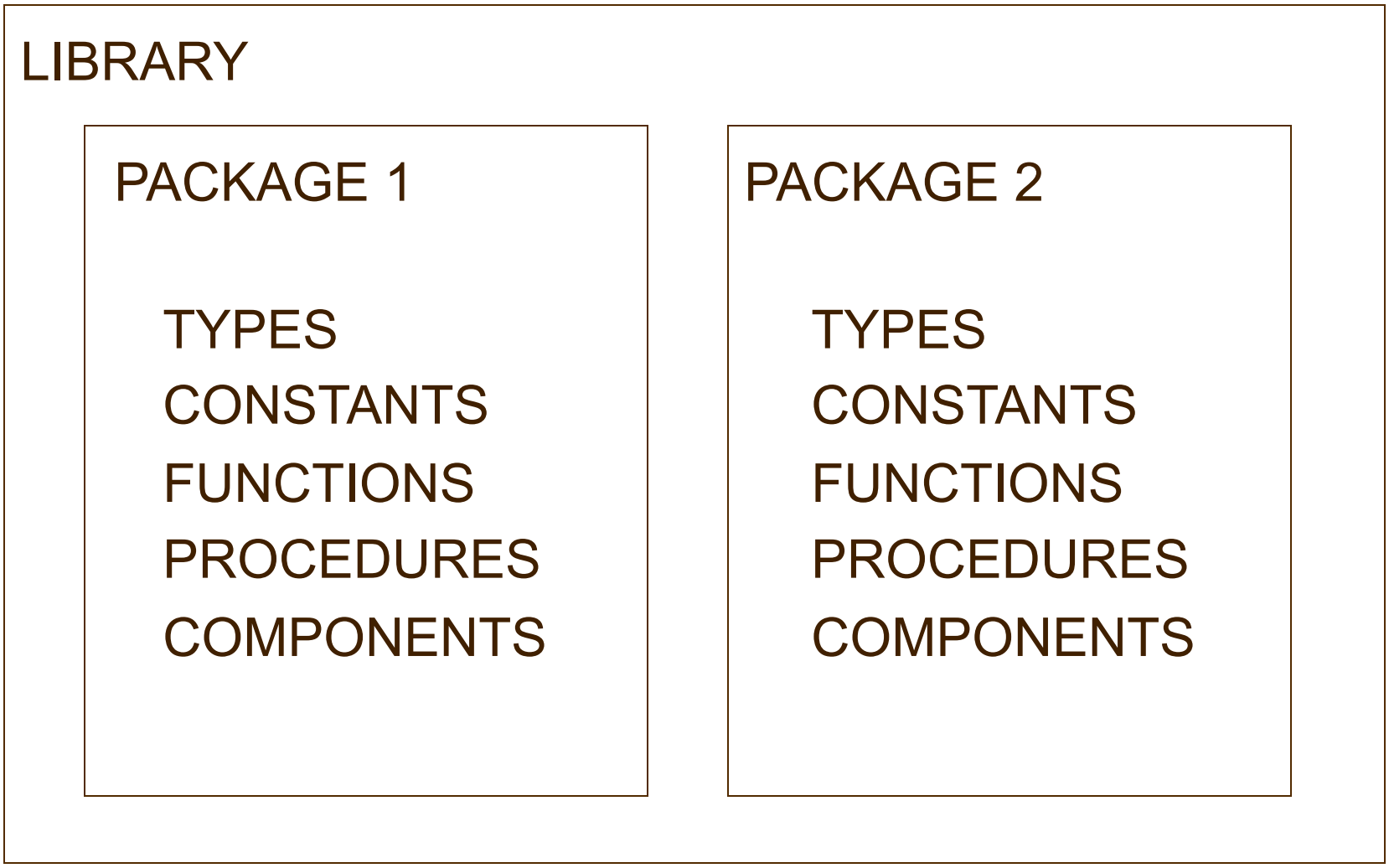
Library Declarations

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

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

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

Library declarations - syntax

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

Fundamental parts of a library



Libraries

- ieee

  Specifies multi-level logic system, including STD_LOGIC, and STD_LOGIC_VECTOR data types

Need to be explicitly declared

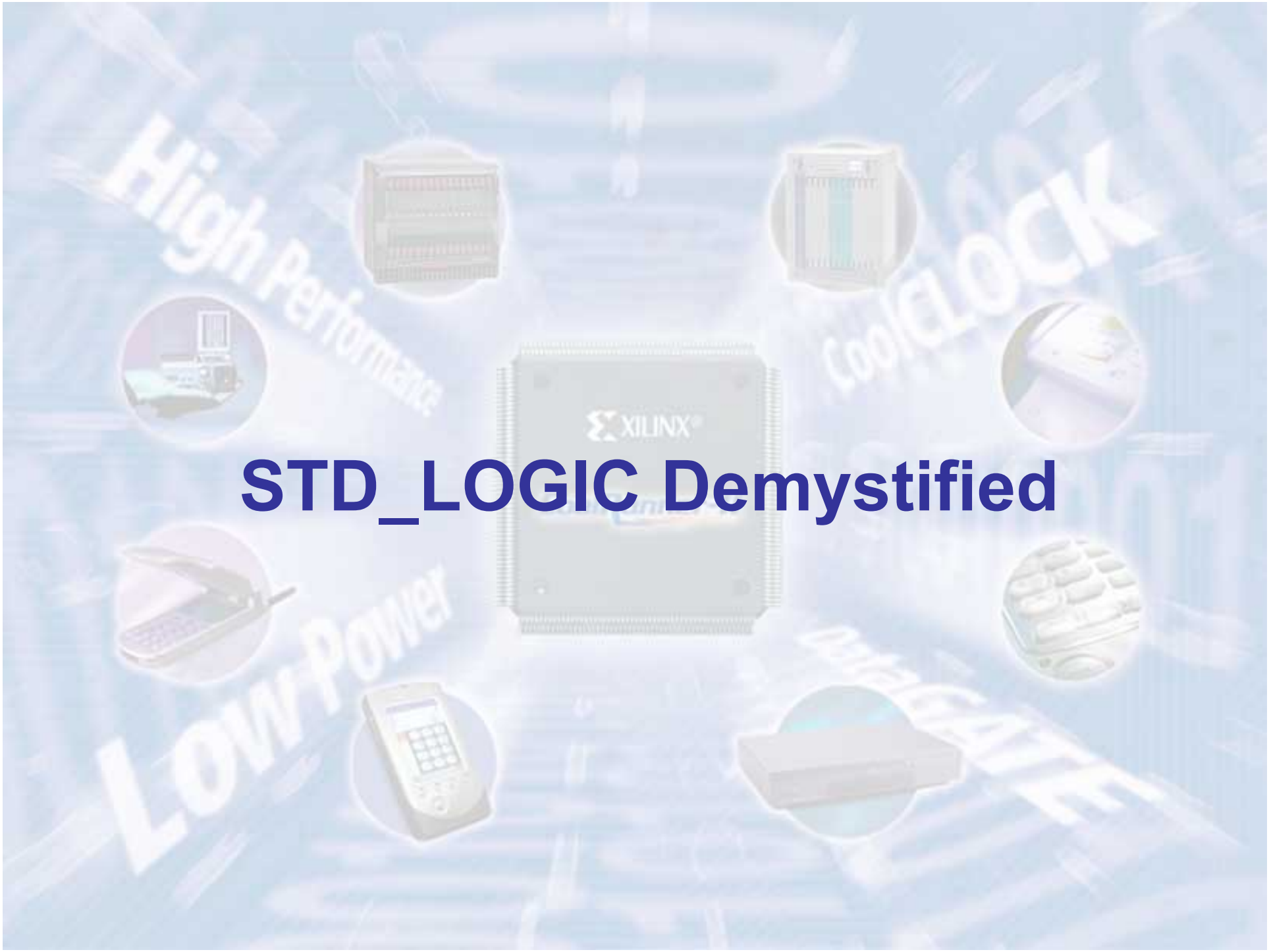
- std

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

- work

  Holds current designs after compilation

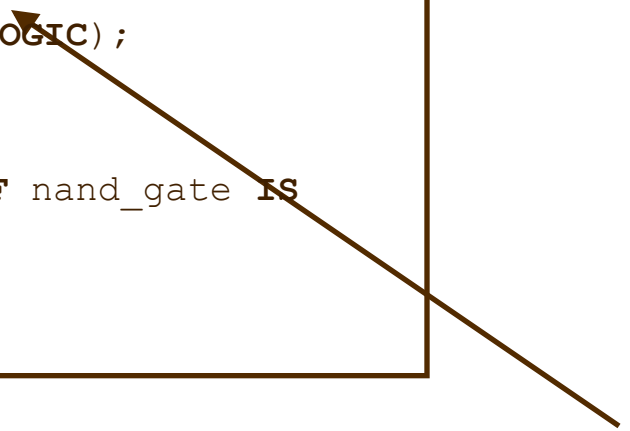
Visible by default



STD_LOGIC Demystified

STD_LOGIC

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

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

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



What is **STD_LOGIC** you ask?

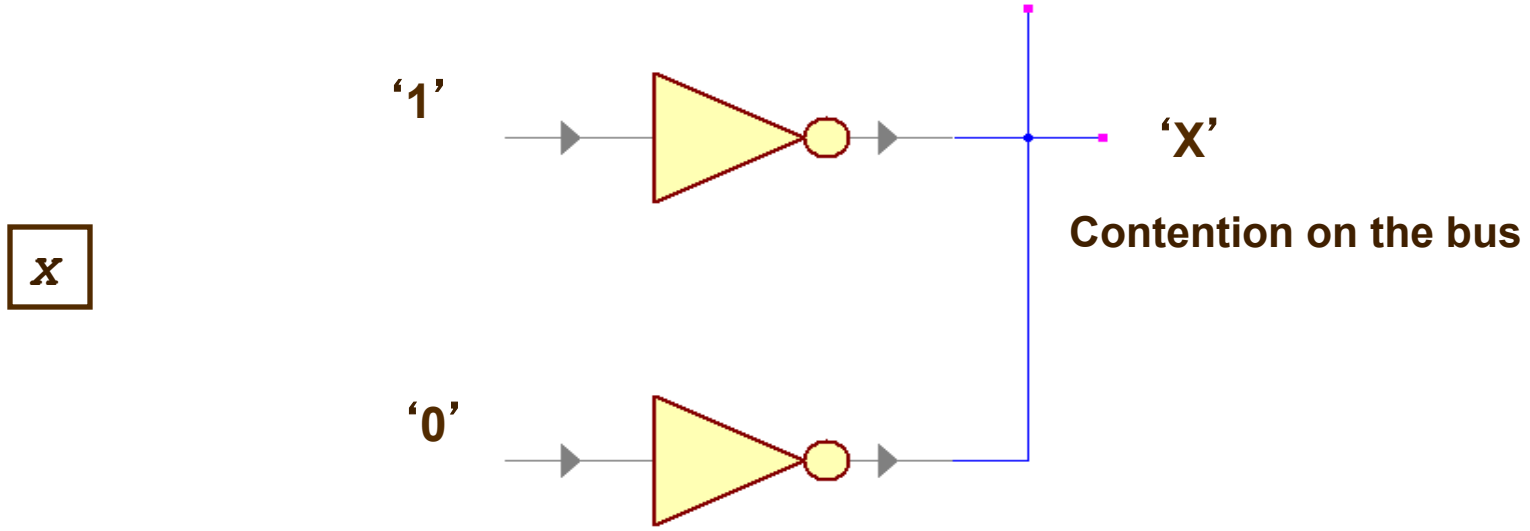
BIT versus STD_LOGIC

- BIT type can only have a value of ‘0’ or ‘1’
- STD_LOGIC can have nine values
  - ’U’,’X’,‘0’,’1’,’Z’,’W’,’L’,’H’,’-’
  - Useful mainly for simulation
  - ‘0’,’1’, and ‘Z’ are synthesizable (**your codes should contain only these three values**)

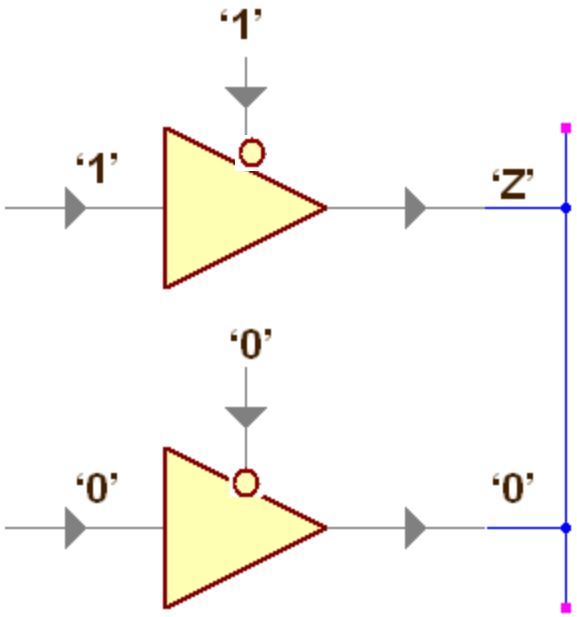
STD_LOGIC *type* demystified

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

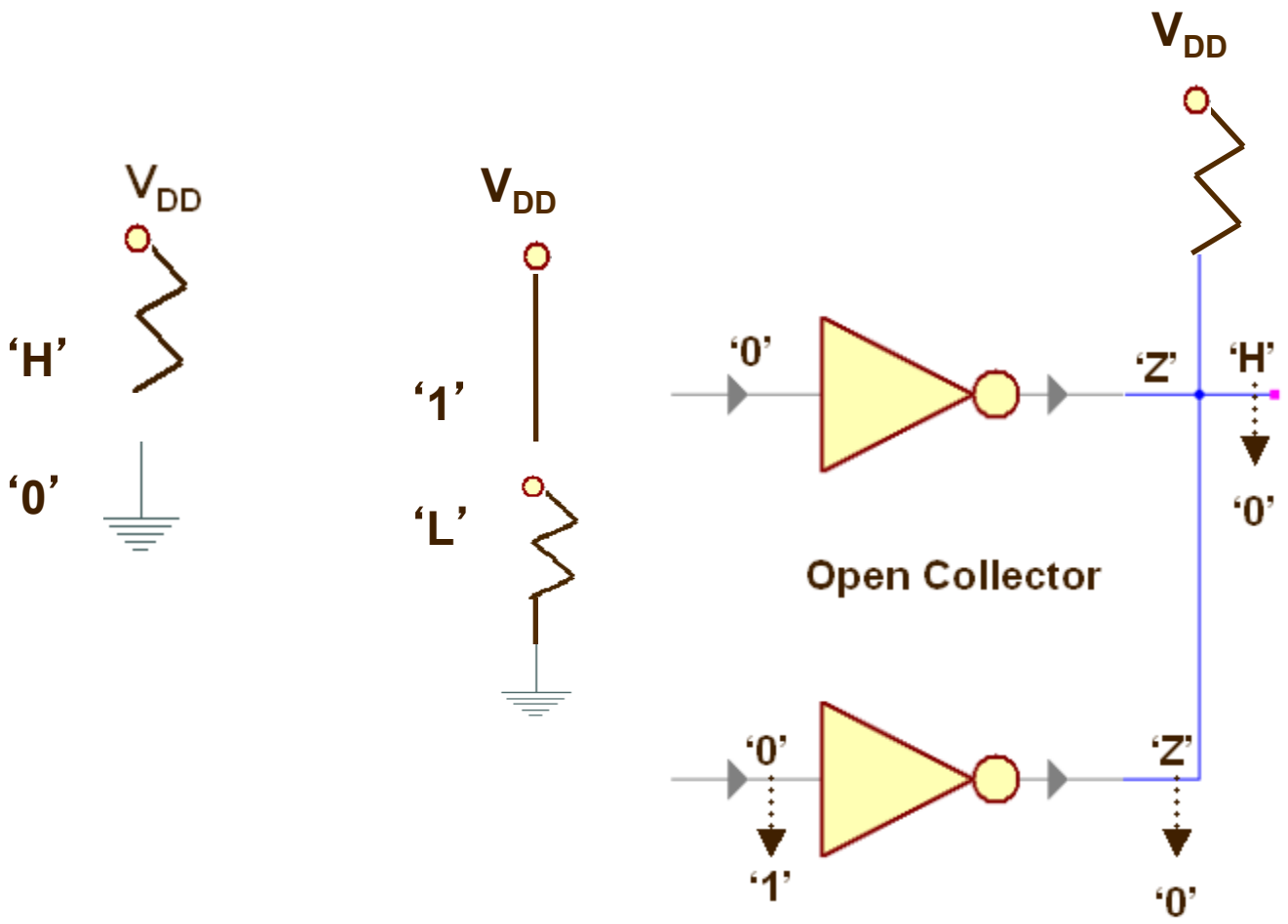
More on STD_LOGIC Meanings (1)



More on STD_LOGIC Meanings (2)



More on STD_LOGIC Meanings (3)



More on STD_LOGIC Meanings (4)

'-'

- *Do not care.*
- *Can be assigned to outputs for the case of invalid inputs (may produce significant improvement in resource utilization after synthesis).*
- *Must be used with great caution.*
  For example in VHDL, the direct comparison
  '1' = '-'
  gives FALSE.

The "std_match" functions defined in the numeric_std package must be used to make this value work as expected:

Example:

```
if (std_match(address, "-11---") then ...
elsif (std_match(address, "-01---") then ...
else ...
end if;
```

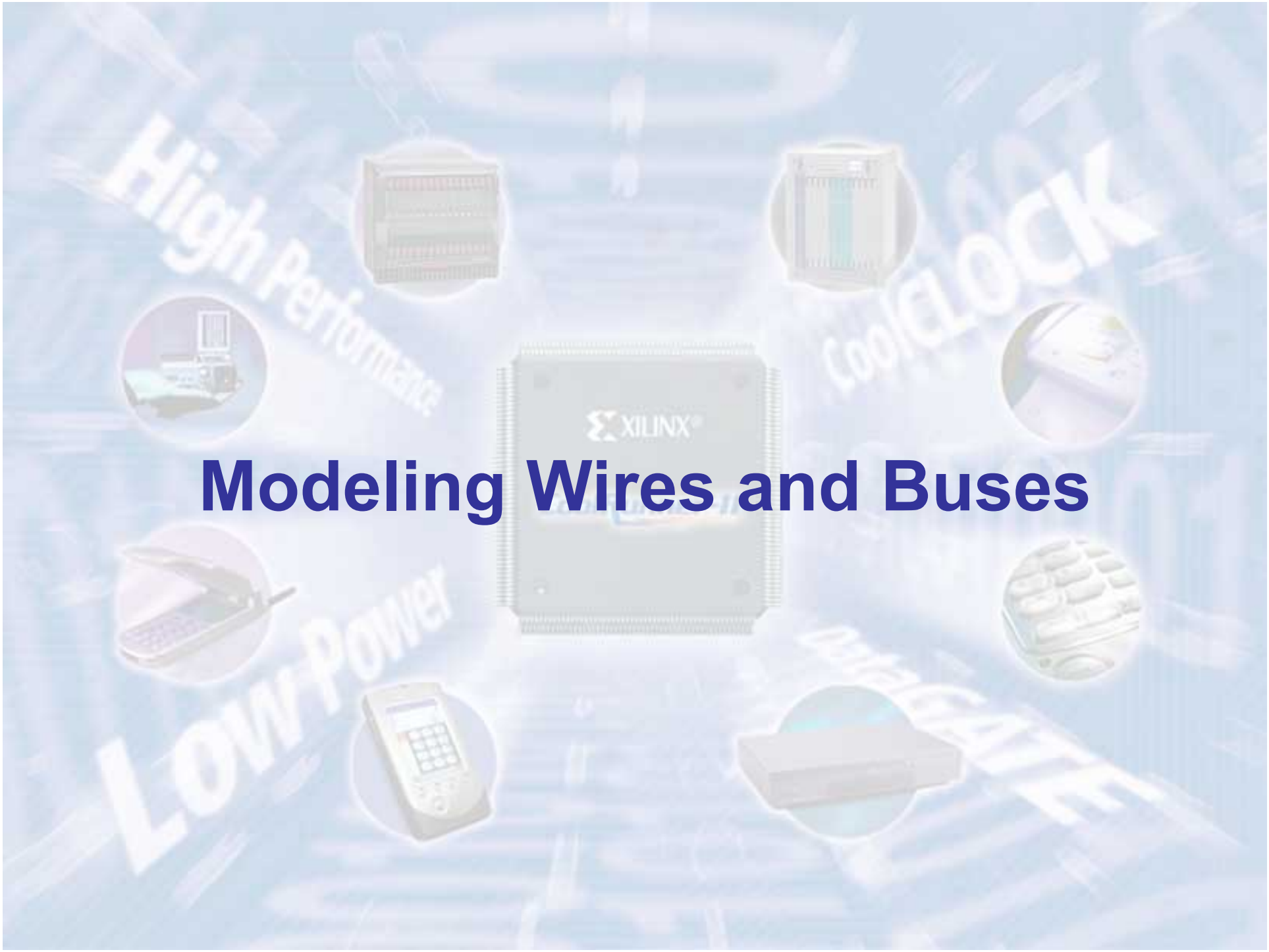
Resolving logic levels

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

STD_LOGIC Rules

- In ECE 545 use **std_logic** or **std_logic_vector** for all entity input or output ports
  - Do not use integer, unsigned, signed, bit for ports
  - You can use them inside of architectures if desired
  - You can use them in generics
- Instead use std_logic_vector and a conversion function inside of your architecture

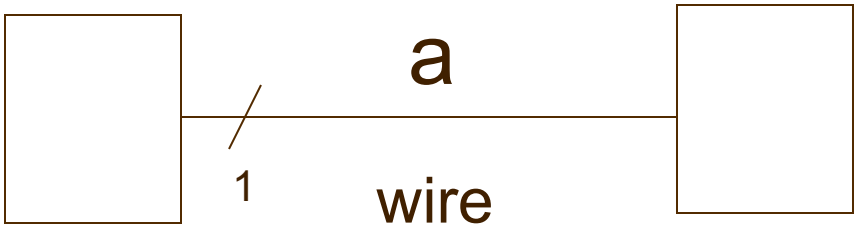
  [Consistent with OpenCores Coding Guidelines]



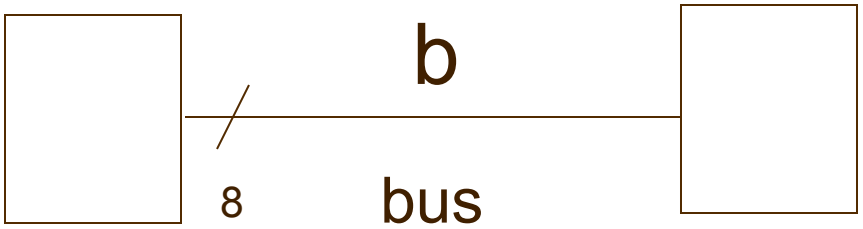
Modeling Wires and Buses

Signals

SIGNAL a : STD_LOGIC;



SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);



Standard Logic Vectors

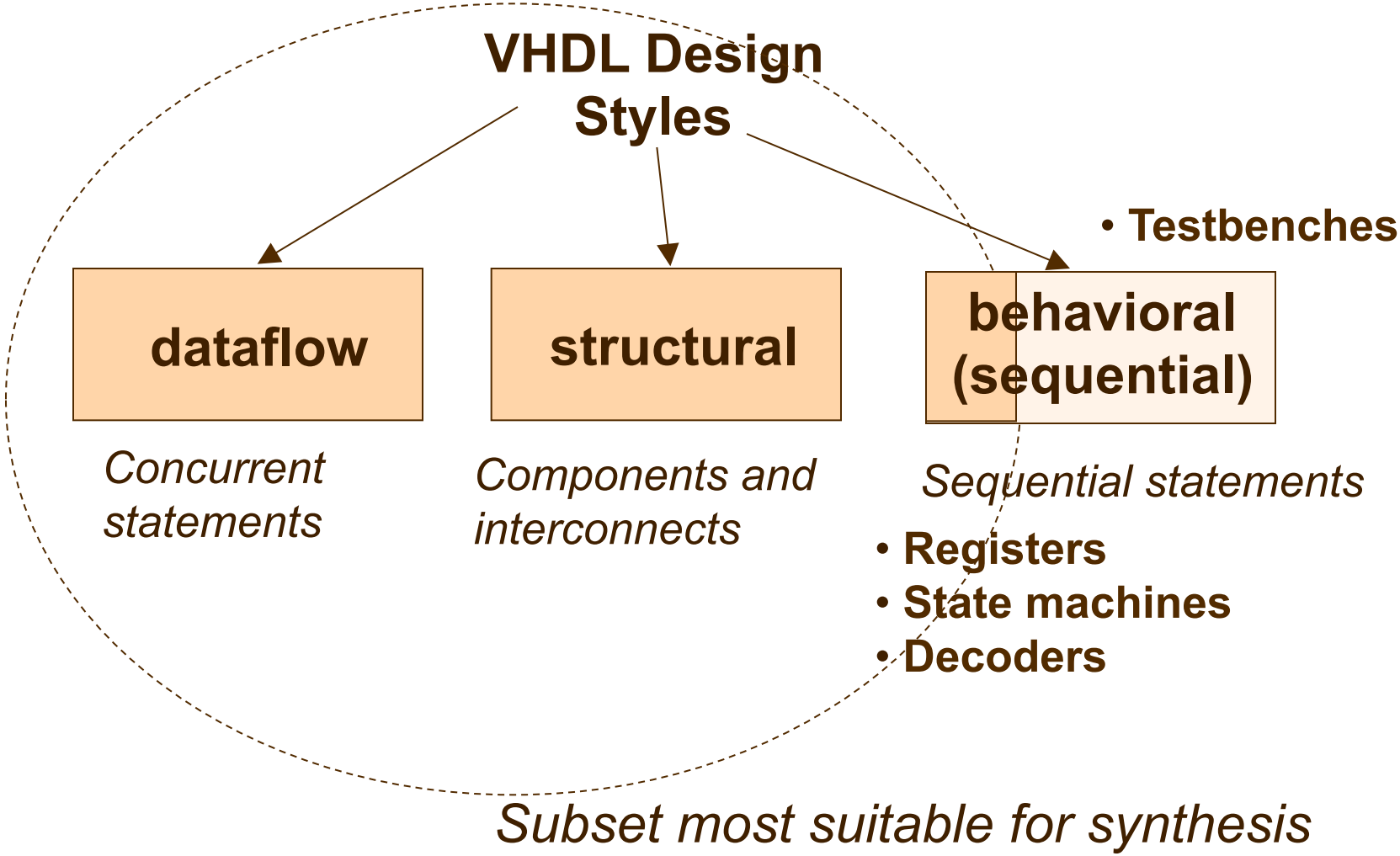
```
SIGNAL a: STD_LOGIC;
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL d: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL e: STD_LOGIC_VECTOR(8 DOWNTO 0);
                                .........
a <= '1';
b <= "0000";          -- Binary base assumed by default
c <= B"0000";         -- Binary base explicitly specified
d <= X"AF67";         -- Hexadecimal base
e <= O"723";          -- Octal base
```

Vectors and Concatenation

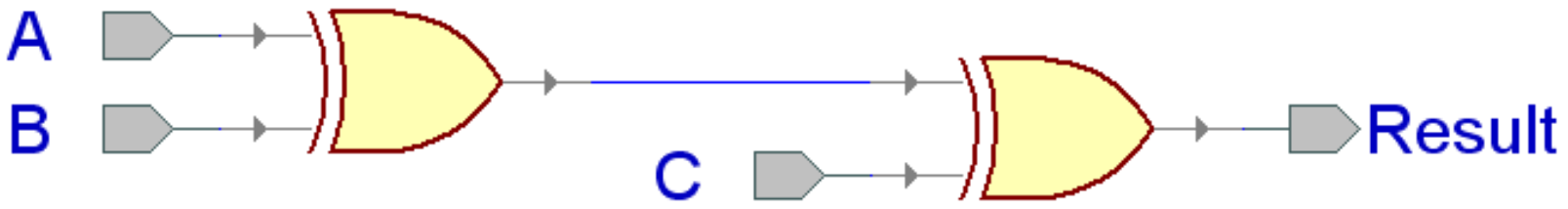
```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c, d, e: STD_LOGIC_VECTOR(7 DOWNTO 0);

a <= "0000";
b <= "1111";
c <= a & b;                    -- c = "00001111"

d <= '0' & "0001111";          -- d <= "00001111"

e <= '0' & '0' & '0' & '0' & '1' & '1' &
     '1' & '1';
                               -- e <= "00001111"
```

VHDL Design Styles

VHDL Design Styles



xor3 Example



Entity xor3_gate

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

ENTITY xor3_gate IS
    PORT(
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        C : IN STD_LOGIC;
        Result : OUT STD_LOGIC
    );
end xor3_gate;
```

Dataflow Architecture (xor3_gate)

```
ARCHITECTURE dataflow OF xor3_gate IS
SIGNAL U1_OUT: STD_LOGIC;
BEGIN
        U1_OUT <= A XOR B;
        Result  <= U1_OUT XOR C;
END dataflow;
```

Dataflow Description

- Describes how data moves through the system and the various processing steps.
  - Dataflow uses series of concurrent statements to realize logic.
  - Dataflow is the most useful style to describe combinational logic
  - Dataflow code also called "concurrent" code
- Concurrent statements are evaluated at the same time; thus, the order of these statements doesn't matter
  - This is not true for sequential/behavioral statements

This order…

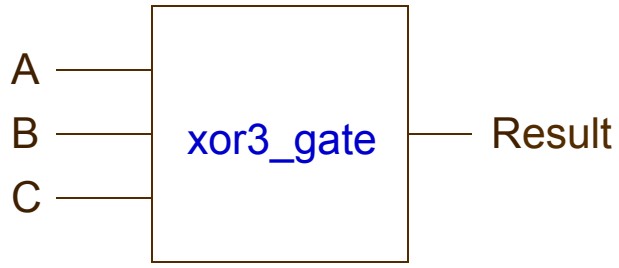
```
U1_out <= A XOR B;
Result <= U1_out XOR C;
```

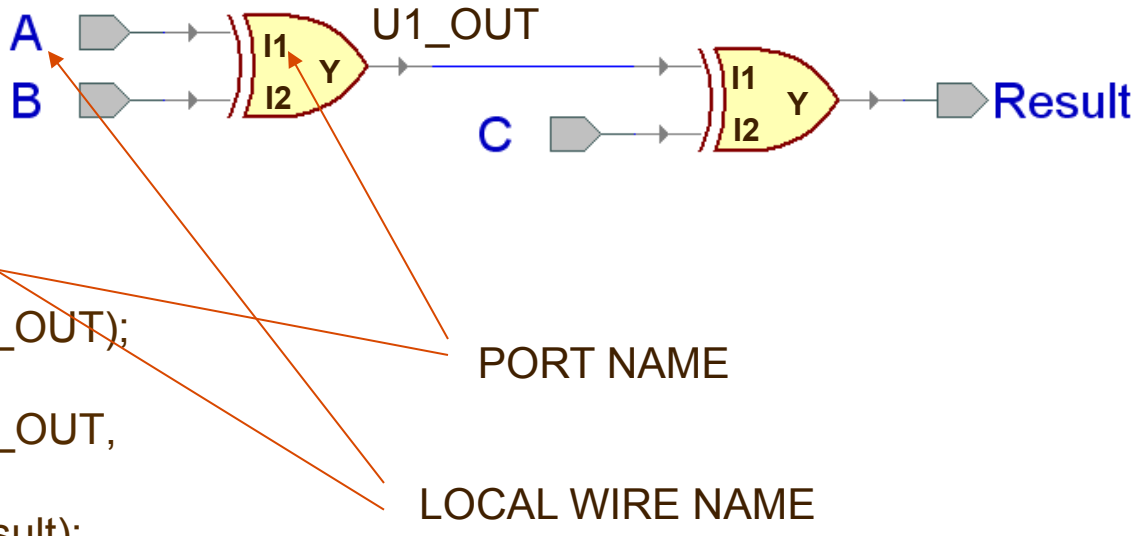
Is the same as this order…

```
Result <= U1_out XOR C;
U1_out <= A XOR B;
```

Structural Architecture in VHDL 87

```
ARCHITECTURE structural OF xor3_gate IS
SIGNAL U1_OUT: STD_LOGIC;

COMPONENT xor2
      PORT(
            I1 : IN STD_LOGIC;
            I2 : IN STD_LOGIC;
            Y : OUT STD_LOGIC
          );
END COMPONENT;

BEGIN
     U1: xor2 PORT MAP (I1 => A,
                        I2 => B,
                        Y  => U1_OUT);

     U2: xor2 PORT MAP (I1 => U1_OUT,
                        I2 => C,
                        Y  => Result);
END structural;
```





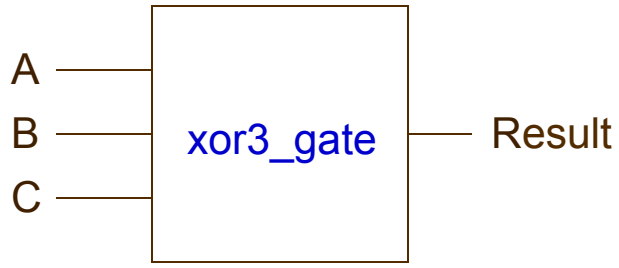
xor2

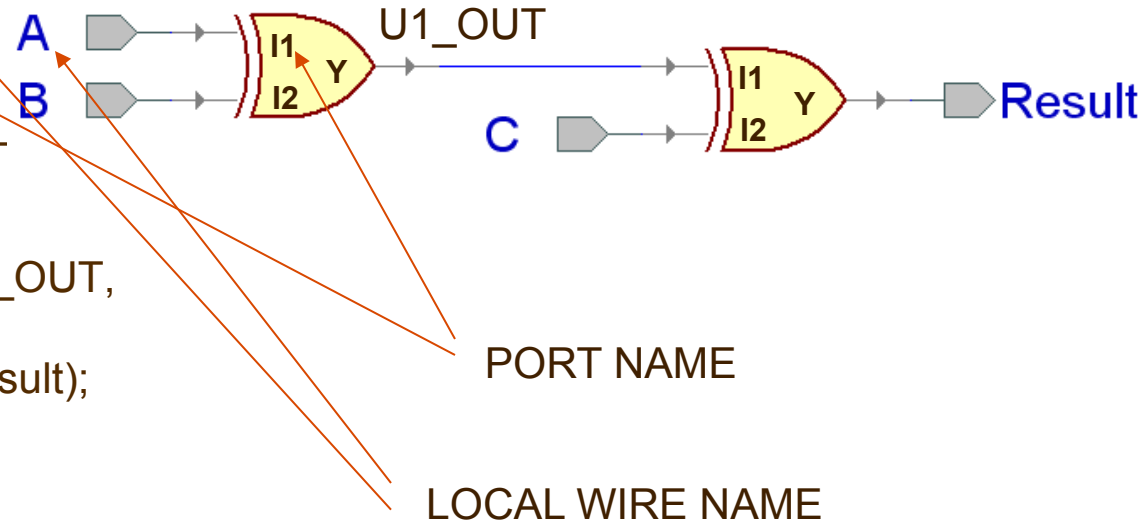
xor2.vhd

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

ENTITY xor2 IS
    PORT(
        I1   : IN STD_LOGIC;
        I2   : IN STD_LOGIC;
        Y    : OUT STD_LOGIC);
END xor2;

ARCHITECTURE dataflow OF xor2 IS
BEGIN
    Y <= I1 xor I2;
END dataflow;
```

Structural Architecture in VHDL 93

```
ARCHITECTURE structural OF xor3_gate IS
SIGNAL U1_OUT: STD_LOGIC;

BEGIN
    U1: entity work.xor2(dataflow)
            PORT MAP (I1 => A,
                      I2 => B,
                      Y  => U1_

    U2:  entity work.xor2(dataflow)
            PORT MAP (I1 => U1_OUT,
                      I2 => C,
                      Y  => Result);
END structural;
```





Structural Description

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

Behavioral Architecture (xor3 gate)

```
ARCHITECTURE behavioral OF xor3 IS
BEGIN
xor3_behave: PROCESS (A, B, C)
BEGIN
   IF ((A XOR B XOR C) = '1') THEN
        Result <= '1';
   ELSE
        Result <= '0';
   END IF;
END PROCESS xor3_behave;
END behavioral;
```

Behavioral Description

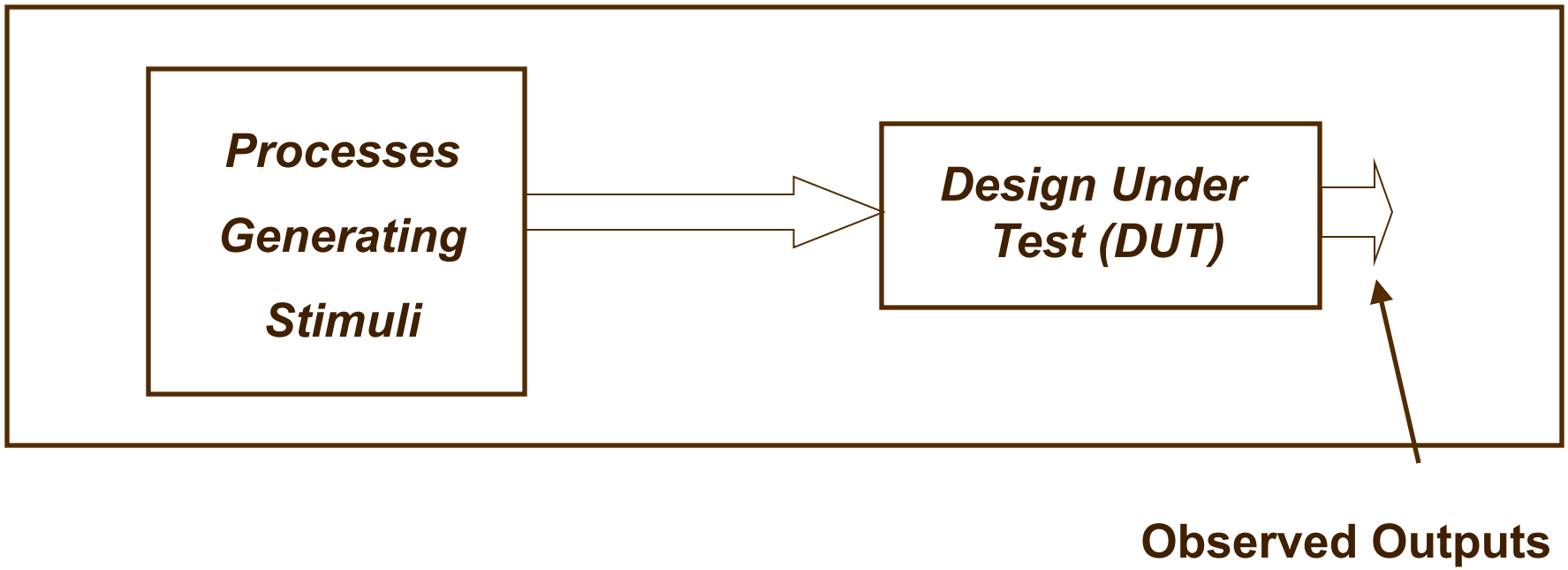
- It accurately models what happens on the inputs and outputs of the black box (no matter what is inside and how it works).
- This style uses PROCESS statements in *VHDL.*

Testbenches

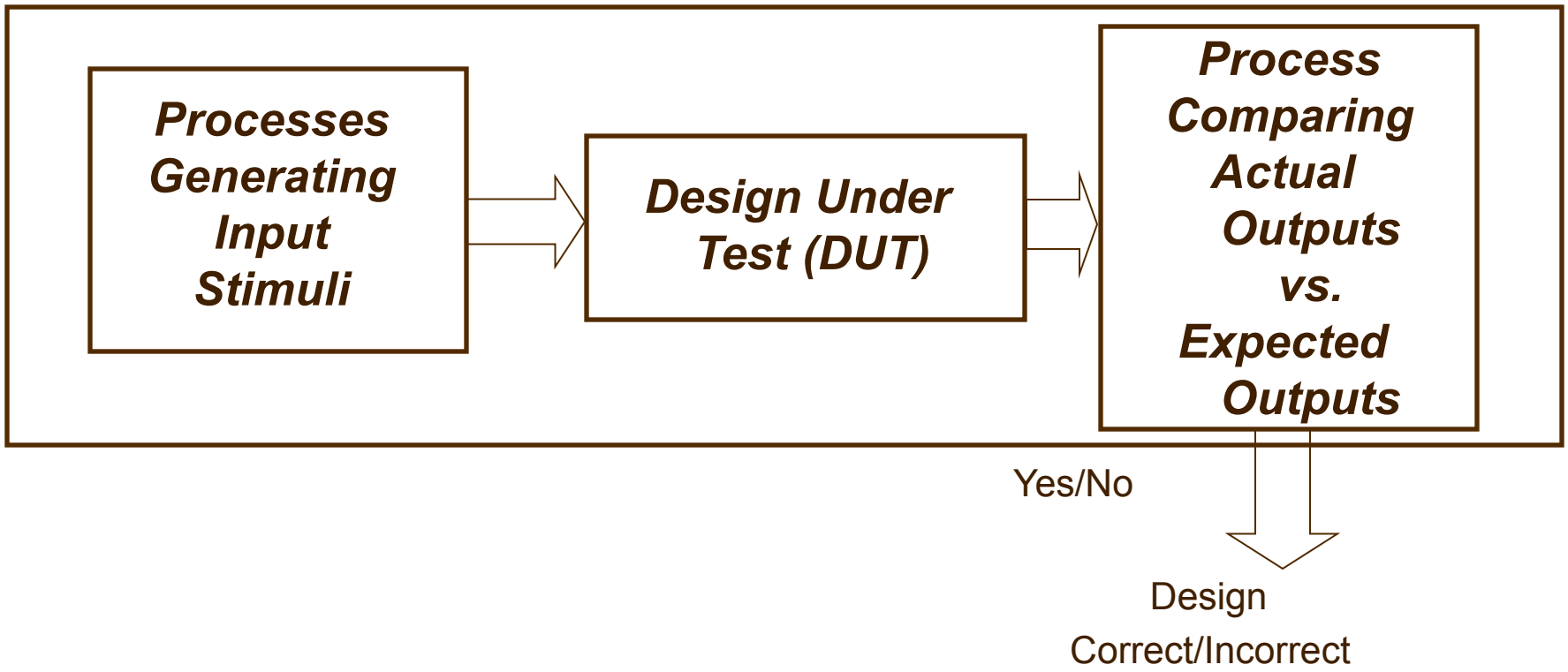
Testbench Defined

- *Testbench* = VHDL entity that applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
- The results can be viewed in a waveform window or written to a file.
- Since *Testbench* is written in VHDL, it is not restricted to a single simulation tool (portability).
- The same *Testbench* can be easily adapted to test different implementations (i.e. different *architectures*) of the same design.

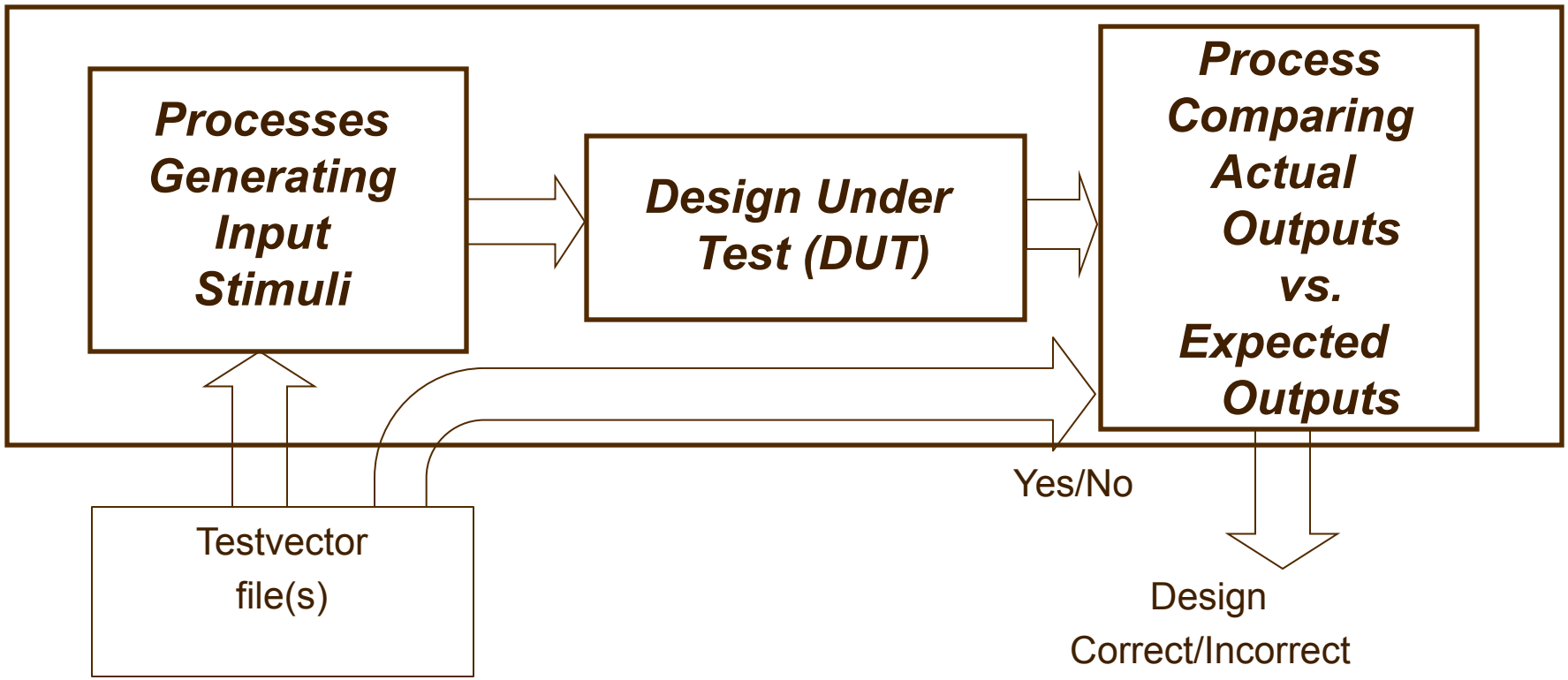
Simple Testbench



Advanced Testbench (1)



Advanced Testbench (2)



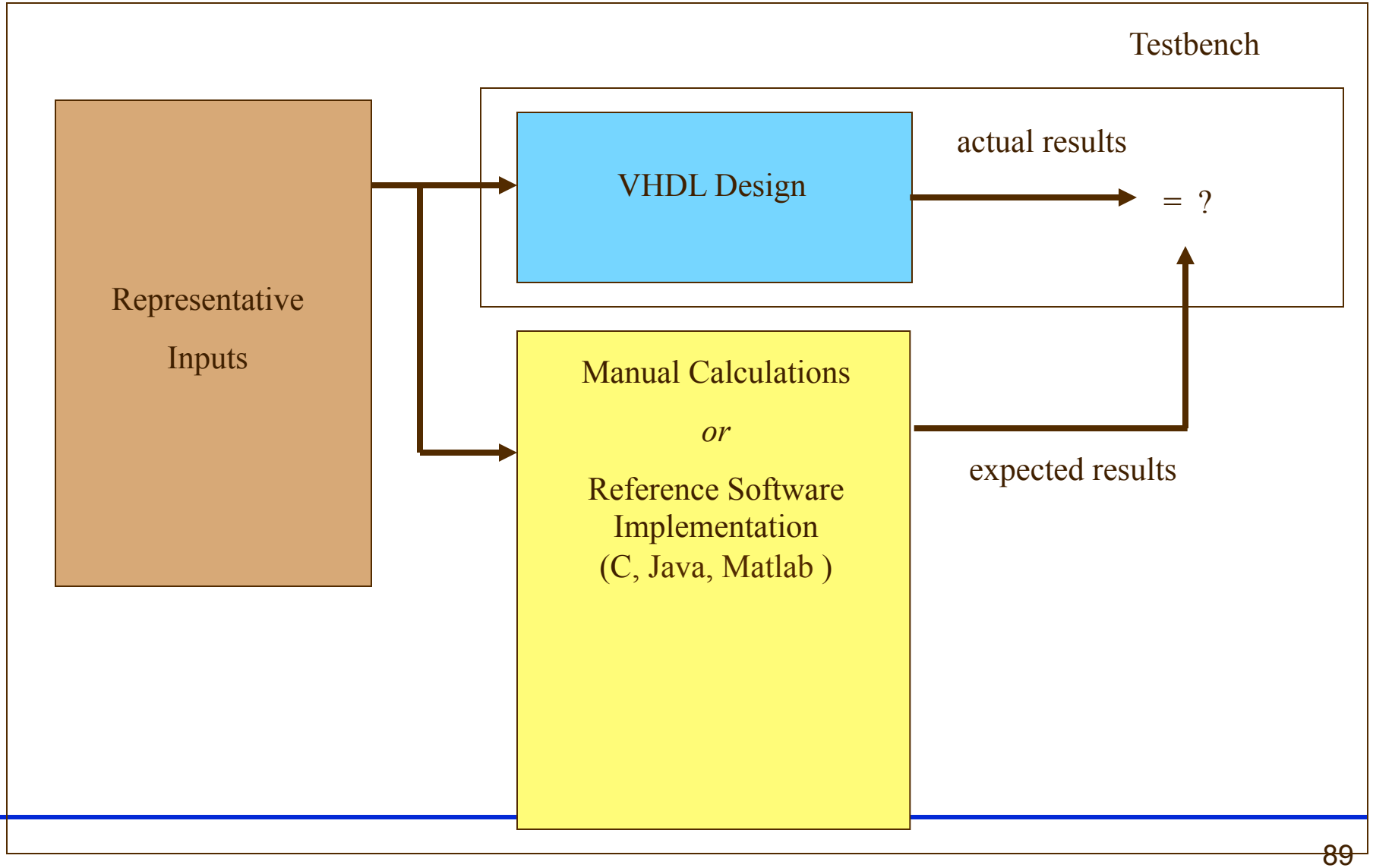
Test vectors

Set of pairs: {Input i, Expected Output i}

Input 1, Expected Output 1
Input 2, Expected Output 2
……
Input N, Expected Output N

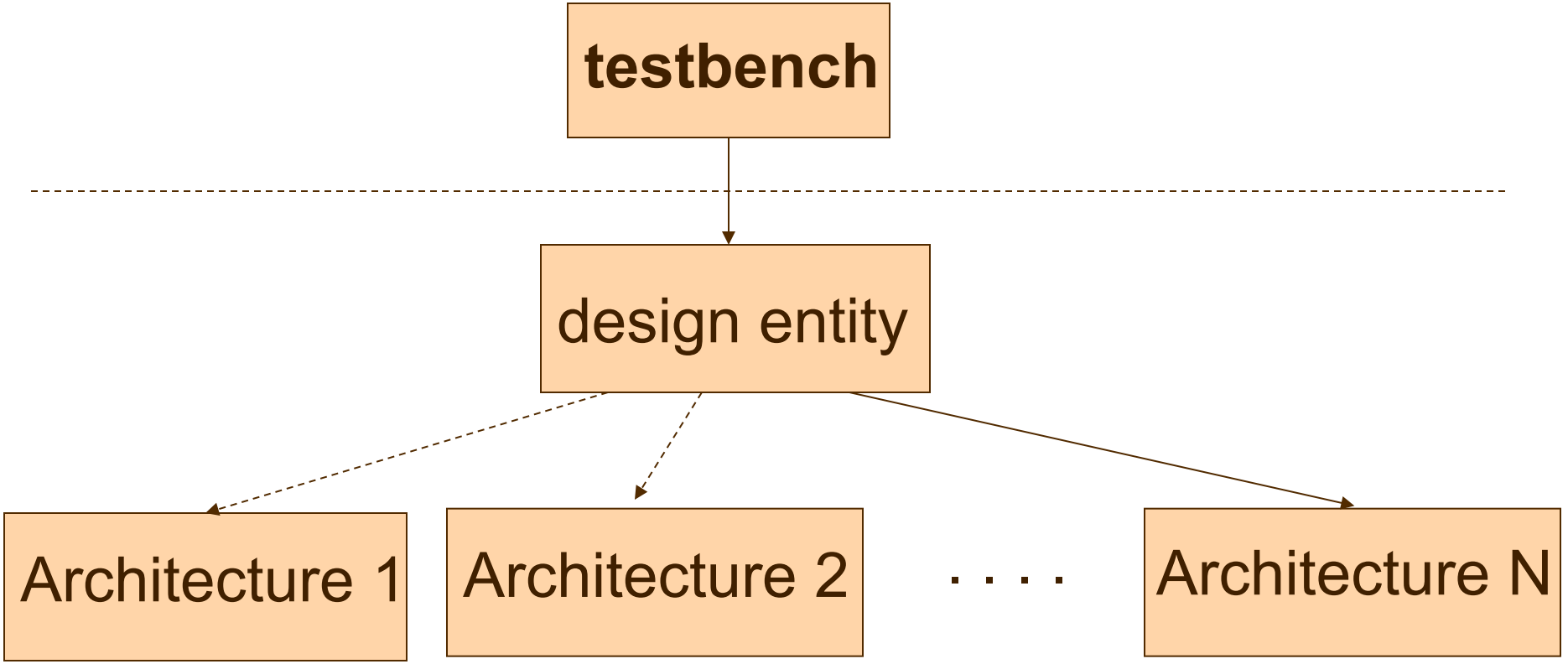
Test vectors can be:
- defined in the testbench source file
- stored in a data file

Possible sources of expected results used for comparison



Testbench

The same testbench can be used to test multiple implementations of the same circuit (multiple architectures)



Simple Testbench Anatomy

```
ENTITY my_entity_tb IS
              --TB entity has no ports
END my_entity_tb;

ARCHITECTURE behavioral OF tb IS

     --Local signals and constants

  COMPONENT TestComp --All Design Under Test component declarations
            PORT (  );
  END COMPONENT;
-------------------------------------------------------
BEGIN
  DUT:TestComp PORT MAP(                  -- Instantiations of DUTs
                       );

  testSequence: PROCESS
                                  -- Input stimuli
  END PROCESS;


END behavioral;
```

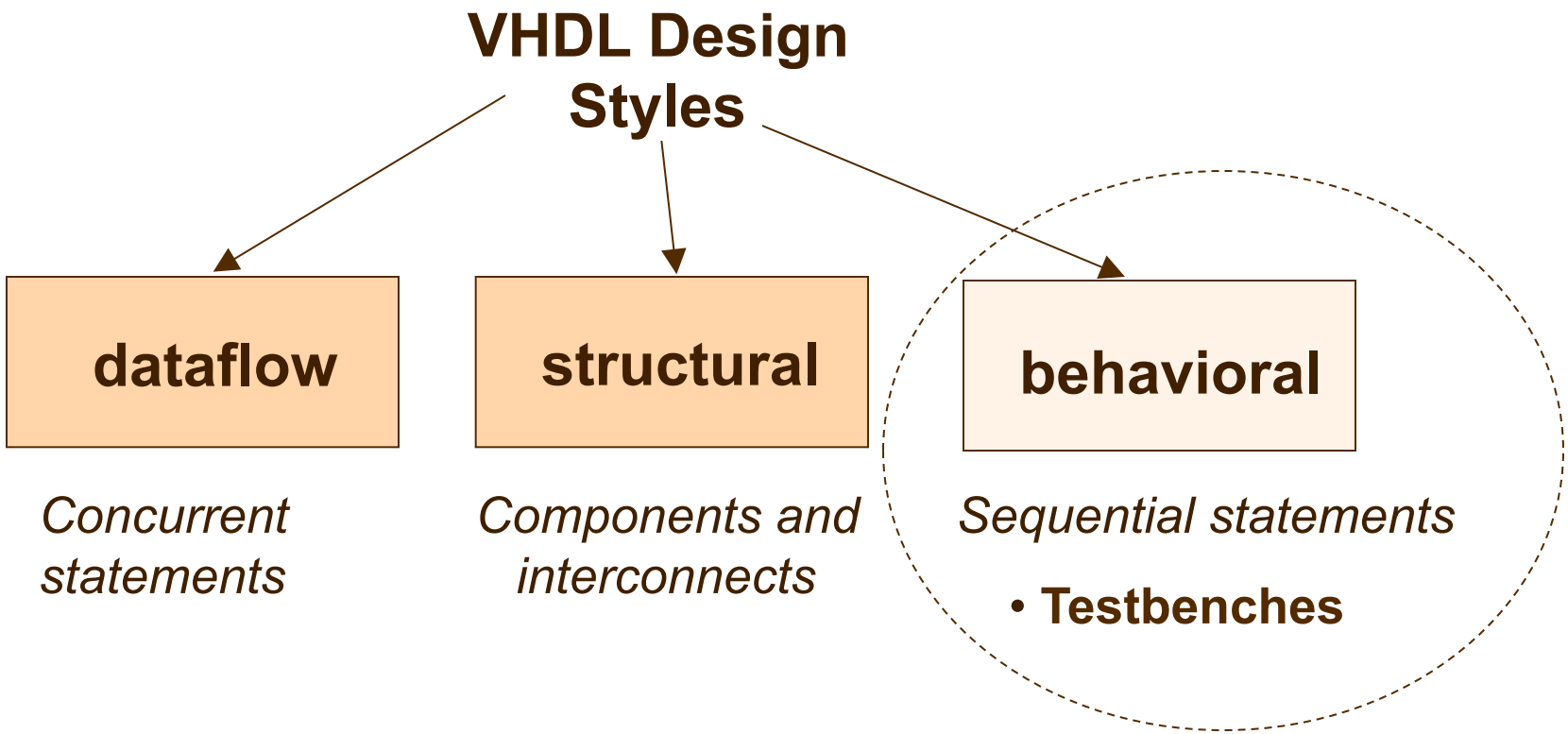
Testbench for XOR3 (1)

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

ENTITY xor3_tb IS
END xor3_tb;

ARCHITECTURE behavioral OF xor3_tb IS
 -- Component declaration of the tested unit
 COMPONENT xor3
 PORT(
     A : IN STD_LOGIC;
     B : IN STD_LOGIC;
     C : IN STD_LOGIC;
     Result : OUT STD_LOGIC );
 END COMPONENT;

 -- Stimulus signals - signals mapped to the input and inout ports of tested entity
 SIGNAL test_vector: STD_LOGIC_VECTOR(2 DOWNTO 0);
 SIGNAL test_result : STD_LOGIC;
```

Testbench for XOR3 (2)

```
BEGIN
 UUT : xor3
                PORT MAP (
                                A => test_vector(2),
                                B => test_vector(1),
                                C => test_vector(0),
                                Result => test_result);
                                        );
 Testing: PROCESS
 BEGIN
      test_vector <= "000";
     WAIT FOR 10 ns;
      test_vector <= "001";
      WAIT FOR 10 ns;
      test_vector <= "010";
      WAIT FOR 10 ns;
     test_vector <= "011";
      WAIT FOR 10 ns;
      test_vector <= "100";
      WAIT FOR 10 ns;
      test_vector <= "101";
      WAIT FOR 10 ns;
      test_vector <= "110";
      WAIT FOR 10 ns;
      test_vector <= "111";
     WAIT FOR 10 ns;
 END PROCESS;
END behavioral;
```

VHDL Design Styles

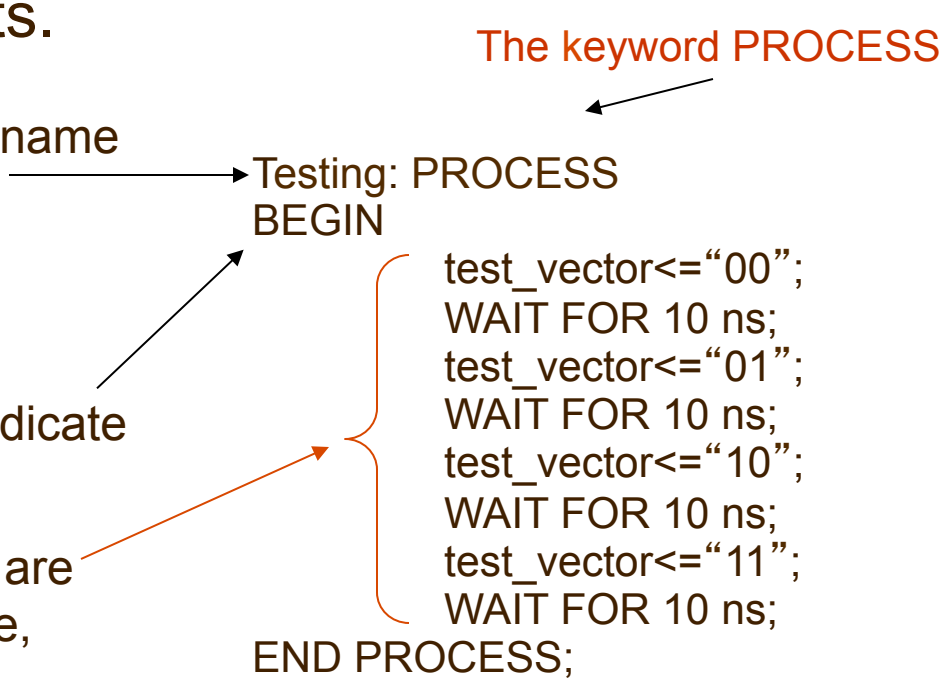


Process without Sensitivity List and its use in Testbenches

What is a PROCESS?

- A process is a sequence of instructions referred to as sequential statements.

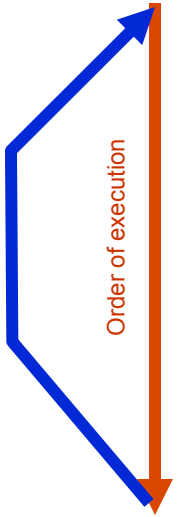
- A process can be given a unique name using an optional LABEL
- This is followed by the keyword PROCESS
- The keyword BEGIN is used to indicate the start of the process
- All statements within the process are executed SEQUENTIALLY. Hence, order of statements is important.
- A process must end with the keywords END PROCESS.



Execution of statements in a PROCESS

- The execution of statements continues sequentially till the last statement in the process.
- After execution of the last statement, the control is again passed to the beginning of the process.

```
Testing: PROCESS
BEGIN
        test_vector<=“00”;
        WAIT FOR 10 ns;
        test_vector<=“01”;
        WAIT FOR 10 ns;
        test_vector<=“10”;
        WAIT FOR 10 ns;
        test_vector<=“11”;
        WAIT FOR 10 ns;
END PROCESS;
```



Program control is passed to the first statement after BEGIN

PROCESS with a WAIT Statement

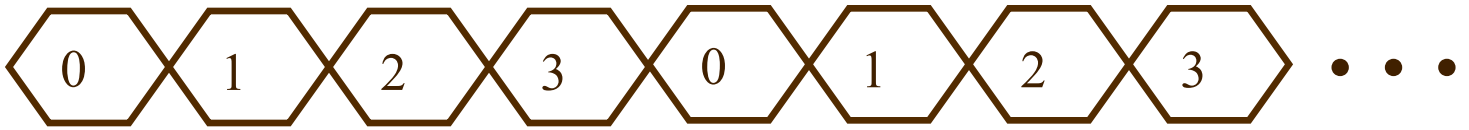
- The last statement in the PROCESS is a WAIT instead of WAIT FOR 10 ns.
- This will cause the PROCESS to suspend indefinitely when the WAIT statement is executed.
- This form of WAIT can be used in a process included in a testbench when all possible combinations of inputs have been tested or a non-periodical signal has to be generated.

```
Testing: PROCESS
BEGIN
        test_vector<=“00”;
        WAIT FOR 10 ns;
        test_vector<=“01”;
        WAIT FOR 10 ns;
        test_vector<=“10”;
        WAIT FOR 10 ns;
        test_vector<=“11”;
        WAIT;
END PROCESS;
```

Program execution stops here

WAIT FOR vs. WAIT

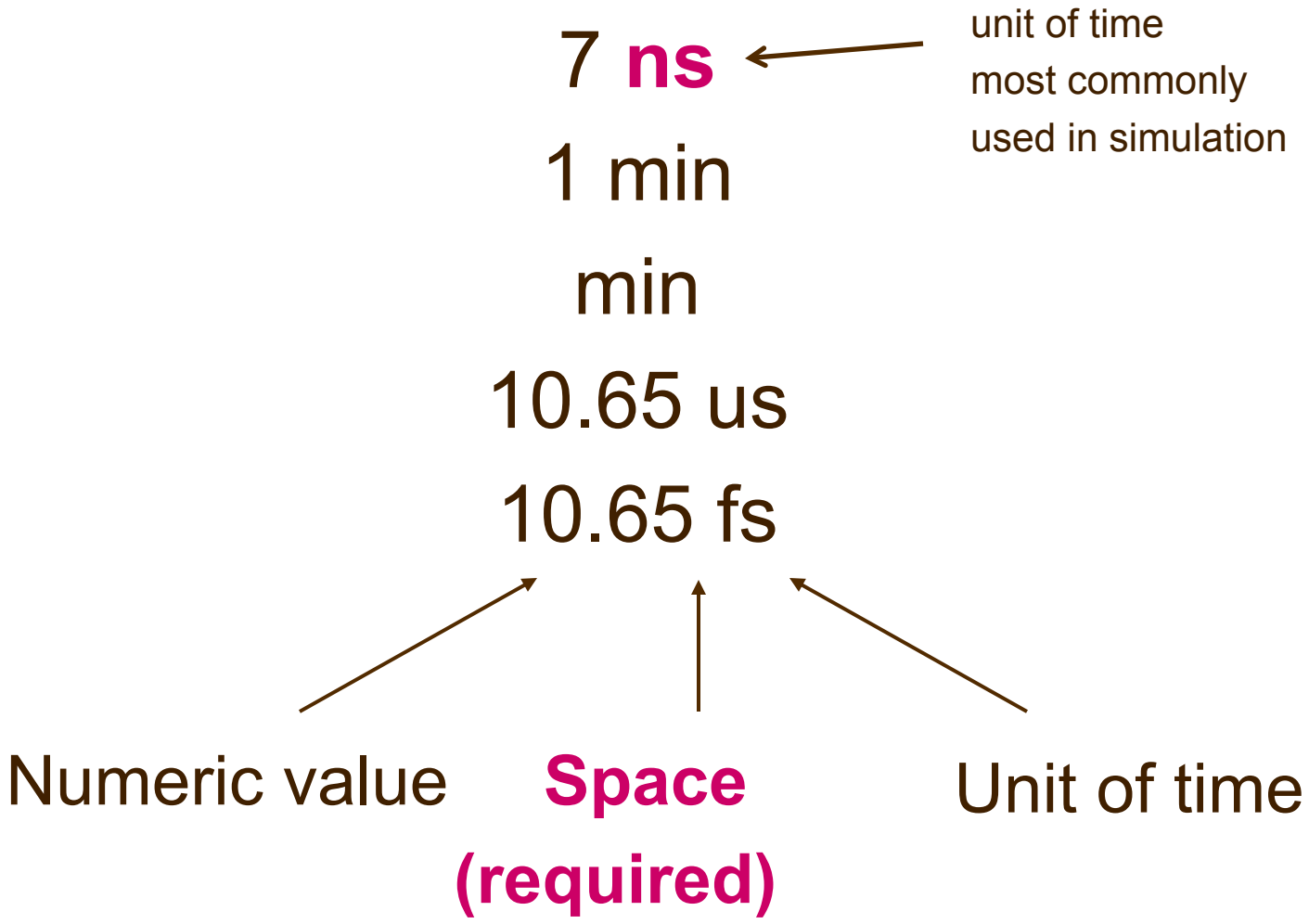
WAIT FOR: waveform will keep repeating itself forever



WAIT : waveform will keep its state after the last wait instruction.

Specifying time in VHDL

Time values (physical literals) - Examples



Units of time

Unit	Definition
Base Unit	
fs	femtoseconds (10^{-15} seconds)
Derived Units	
ps	picoseconds (10^{-12} seconds)
ns	**nanoseconds (10^{-9} seconds)**
us	microseconds (10^{-6} seconds)
ms	miliseconds (10^{-3} seconds)
sec	seconds
min	minutes (60 seconds)
hr	hours (3600 seconds)

Simple Testbenches

Generating selected values of one input

```
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);

BEGIN
   .......
        testing: PROCESS
        BEGIN
                test_vector <= "000";
                WAIT FOR 10 ns;
                test_vector <= "001";
                WAIT FOR 10 ns;
                test_vector <= "010";
                WAIT FOR 10 ns;
                test_vector <= "011";
                WAIT FOR 10 ns;
                test_vector <= "100";
                WAIT FOR 10 ns;
        END PROCESS;
 ........
END behavioral;
```

Generating all values of one input

```
SIGNAL test_vector : STD_LOGIC_VECTOR(3 downto 0):="0000";

BEGIN
    .......

    testing: PROCESS
    BEGIN
            WAIT FOR 10 ns;
            test_vector <= test_vector + 1;
    end process TESTING;

    ........
END behavioral;
```

Arithmetic Operators in VHDL (1)

To use basic arithmetic operations involving std_logic_vectors you need to include the following library packages:

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
or
USE ieee.std_logic_signed.all;

Arithmetic Operators in VHDL (2)

You can use standard +, - operators
to perform addition and subtraction:

```
signal A :  STD_LOGIC_VECTOR(3 downto 0);
signal B :  STD_LOGIC_VECTOR(3 downto 0);
signal C :  STD_LOGIC_VECTOR(3 downto 0);
……
C <= A + B;
```

Different ways of performing the same operation

signal count: std_logic_vector(7 downto 0);

You can use:

count <= count + “00000001”;

or

count <= count + 1;

or

count <= count + ‘1’;

Different declarations for the same operator

Declarations in the package ieee.std_logic_unsigned:

```
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: std_logic_vector;
                 R:std_logic)
          return std_logic_vector;
```

Operator overloading

- Operator overloading allows different argument types for a given operation (function)
- The VHDL tools resolve which of these functions to select based on the types of the inputs
- This selection is transparent to the user as long as the function has been defined for the given argument types.

Generating all possible values of two inputs

```
SIGNAL test_ab : STD_LOGIC_VECTOR(1 downto 0);
SIGNAL test_sel : STD_LOGIC_VECTOR(1 downto 0);

BEGIN
        .......

                double_loop: PROCESS
                BEGIN
                        test_ab  <="00";
                        test_sel  <="00";
                        for I in 0 to 3 loop
                            for J in 0 to 3 loop
                                    wait for 10 ns;
                                    test_ab <= test_ab + 1;
                            end loop;
                            test_sel <= test_sel + 1;
                        end loop;
                END PROCESS;

        ........
        END behavioral;
```

Generating periodical signals, such as clocks

```
    CONSTANT clk1_period : TIME := 20 ns;
    CONSTANT clk2_period : TIME := 200 ns;
    SIGNAL clk1 : STD_LOGIC;
    SIGNAL clk2 : STD_LOGIC := '0';

BEGIN
   .......
    clk1_generator: PROCESS
            clk1 <= '0';
            WAIT FOR clk1_period/2;
            clk1 <= '1';
            WAIT FOR clk1_period/2;
    END PROCESS;

    clk2 <= not clk2  after  clk2_period/2;
  .......
END behavioral;
```

Generating one-time signals, such as resets

```
    CONSTANT reset1_width : TIME := 100 ns;
    CONSTANT reset2_width : TIME := 150 ns;
    SIGNAL reset1 : STD_LOGIC;
    SIGNAL reset2 : STD_LOGIC := '1';

BEGIN
    .......
    reset1_generator: PROCESS
            reset1 <= '1';
            WAIT FOR reset1_width;
            reset1 <= '0';
            WAIT;
    END PROCESS;

    reset2_generator: PROCESS
            WAIT FOR reset2_width;
            reset2 <= '0';
            WAIT;
    END PROCESS;
    .......
END behavioral;
```

Typical error

```
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL reset : STD_LOGIC;

BEGIN
    .......
    generator1: PROCESS
            reset <= '1';
            WAIT FOR 100 ns
            reset <= '0';
            test_vector <="000";
            WAIT;
    END PROCESS;

    generator2: PROCESS
            WAIT FOR 200 ns
            test_vector <="001";
            WAIT FOR 600 ns
            test_vector <="011";
    END PROCESS;
    .......
END behavioral;
```

Asserts & Reports

Assert

Assert is a **non-synthesizable** statement whose purpose is to write out messages on the screen when problems are found during simulation.

Depending on the **severity of the problem**, The simulator is instructed to continue simulation or halt.

Assert - syntax

ASSERT condition
[REPORT "message“]
[SEVERITY severity_level];

The message is written when the condition is FALSE.

Severity_level can be:
Note, Warning, Error (default), or Failure.

Assert - Examples

```
assert initial_value <= max_value
   report "initial value too large"
   severity error;

assert packet_length /= 0
   report "empty network packet received"
   severity warning;

assert false
   report "Initialization complete"
   severity note;
```

Report - syntax

REPORT "message"
[SEVERITY severity_level];

The message is always written.

Severity_level can be:
Note (default), Warning, Error, or Failure.

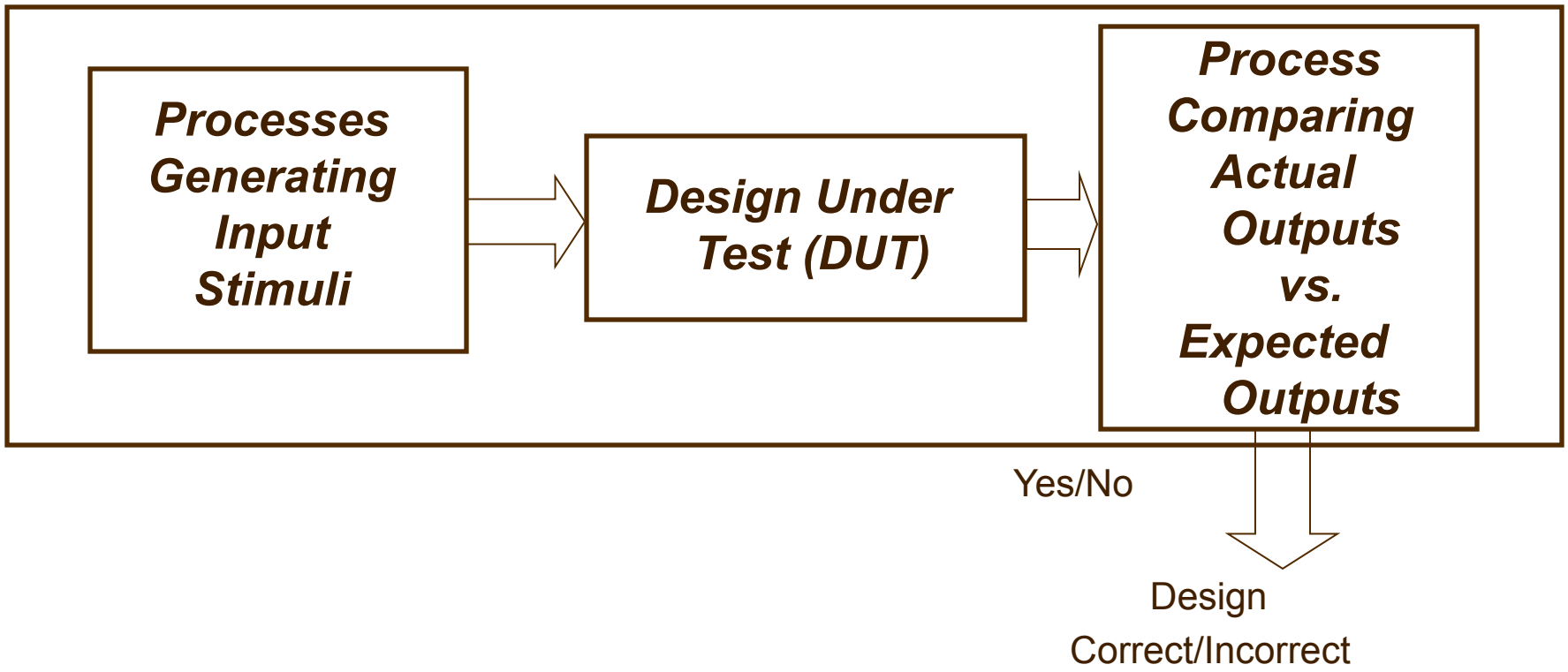
Report - Examples

```
report "Initialization complete";

report  "Current time = " & time'image(now);

report  "Incorrect branch"  severity error;
```

Report - Examples

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity example_1_tb is
end example_1_tb;

architecture behavioral of example_1_tb is
    signal clk : std_logic := '0';
begin
    clk <= not clk after 100 ns;
    process
        begin
            wait for 1000 ns;
            report "Initialization complete";
            report  "Current time = " & time'image(now);
            wait for 1000 ns;
            report "SIMULATION COMPLETED"  severity failure;
    end process;
end behavioral;
```

Advanced Testbench (1)



Records

Records

```
TYPE test_vector IS RECORD
  operation : STD_LOGIC_VECTOR(1 DOWNTO 0);
  a : STD_LOGIC;
  b: STD_LOGIC;
  y : STD_LOGIC;
END RECORD;

CONSTANT num_vectors : INTEGER := 16;

TYPE test_vectors IS ARRAY (0 TO num_vectors-1) OF test_vector;

CONSTANT and_op   : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";
CONSTANT or_op    : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01";
CONSTANT xor_op   : STD_LOGIC_VECTOR(1 DOWNTO 0) := "10";
CONSTANT xnor_op  : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11";
```

Records

```
CONSTANT test_vector_table: test_vectors :=(
 (operation => AND_OP, a=>'0', b=>'0', y=>'0'),
 (operation => AND_OP, a=>'0', b=>'1', y=>'0'),
 (operation => AND_OP, a=>'1', b=>'0', y=>'0'),
 (operation => AND_OP, a=>'1', b=>'1', y=>'1'),
 (operation => OR_OP, a=>'0', b=>'0', y=>'0'),
 (operation => OR_OP, a=>'0', b=>'1', y=>'1'),
 (operation => OR_OP, a=>'1', b=>'0', y=>'1'),
 (operation => OR_OP, a=>'1', b=>'1', y=>'1'),
 (operation => XOR_OP, a=>'0', b=>'0', y=>'0'),
 (operation => XOR_OP, a=>'0', b=>'1', y=>'1'),
 (operation => XOR_OP, a=>'1', b=>'0', y=>'1'),
 (operation => XOR_OP, a=>'1', b=>'1', y=>'0'),
 (operation => XNOR_OP, a=>'0', b=>'0', y=>'1'),
 (operation => XNOR_OP, a=>'0', b=>'1', y=>'0'),
 (operation => XNOR_OP, a=>'1', b=>'0', y=>'0'),
 (operation => XNOR_OP, a=>'1', b=>'1', y=>'1')
);
```

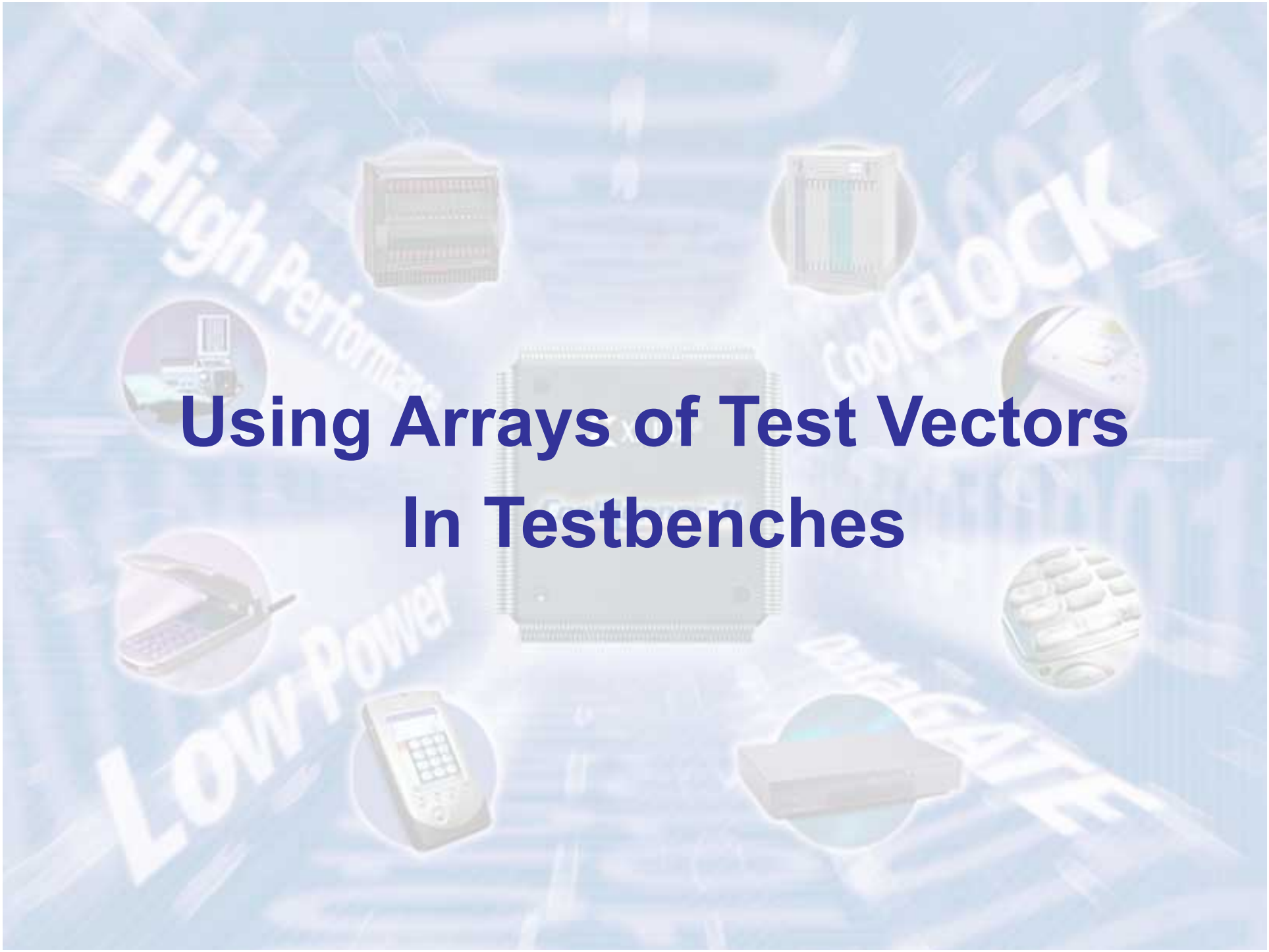
Variables

Variables - features

- Can only be declared within processes and subprograms (functions & procedures)
- Initial value can be explicitly specified in the declaration
- When assigned take an assigned value immediately
- Variable assignments represent the desired behavior, not the structure of the circuit
- Can be used freely in testbenches
- Should be avoided, or at least used with caution in a synthesizable code

Variables - Example

```
testing: PROCESS

VARIABLE error_cnt: INTEGER := 0;

BEGIN
    FOR i IN 0 to num_vectors-1 LOOP
        test_operation <= test_vector_table(i).operation;
        test_a <= test_vector_table(i).a;
        test_b <= test_vector_table(i).b;
        WAIT FOR 10 ns;
        IF test_y /= test_vector_table(i).y THEN
            error_cnt := error_cnt + 1;
        END IF;
    END LOOP;
END PROCESS testing;
```



Using Arrays of Test Vectors In Testbenches

Testbench (1)

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

ENTITY sevenSegmentTB IS
END sevenSegmentTB;

ARCHITECTURE testbench OF sevenSegmentTB IS

COMPONENTsevenSegment
PORT (
  bcdInputs              : IN     STD_LOGIC_VECTOR (3 DOWNTO 0);
  seven_seg_outputs      : OUT    STD_LOGIC_VECTOR(6 DOWNTO 0);
  );
end COMPONENT;

CONSTANT PropDelay: time := 40 ns;
CONSTANT SimLoopDelay: time := 10 ns;
```

Testbench (2)

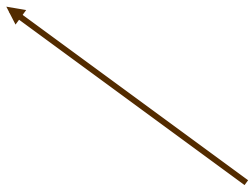
```
TYPE vector IS RECORD
  bcdStimulus:        STD_LOGIC_VECTOR(3 DOWNTO 0);
  sevSegOut:          STD_LOGIC_VECTOR(6 DOWNTO 0);
END RECORD;

CONSTANT NumVectors: INTEGER:= 10;

TYPE vectorArray is ARRAY (0 TO NumVectors - 1) OF vector;

CONSTANT vectorTable: vectorArray := (
  (bcdStimulus => "0000", sevSegOut => "0000001"),
  (bcdStimulus => "0001", sevSegOut => "1001111"),
  (bcdStimulus => "0010", sevSegOut => "0010010"),
  (bcdStimulus => "0011", sevSegOut => "0000110"),
  (bcdStimulus => "0100", sevSegOut => "1001100"),
  (bcdStimulus => "0101", sevSegOut => "0100100"),
  (bcdStimulus => "0110", sevSegOut => "0100000"),
  (bcdStimulus => "0111", sevSegOut => "0001111"),
  (bcdStimulus => "1000", sevSegOut => "0000000"),
  (bcdStimulus => "1001", sevSegOut => "0000100")
);
```

Testbench (3)

```
SIGNAL StimInputs:          STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL CaptureOutputs:      STD_LOGIC_VECTOR(6 DOWNTO 0);

BEGIN

  u1: sevenSegment PORT MAP (
          bcdInputs => StimInputs,
          seven_seg_outputs => CaptureOutputs);
```

Testbench (4)

```
LoopStim: PROCESS
BEGIN
   FOR i in 0 TO NumVectors-1 LOOP
     StimInputs <= vectorTable(i).bcdStimulus;
     WAIT FOR PropDelay;

     ASSERT CaptureOutputs == vectorTable(i).sevSegOut
        REPORT "Incorrect Output"
        SEVERITY error;

    WAIT FOR SimLoopDelay;

   END LOOP;
```

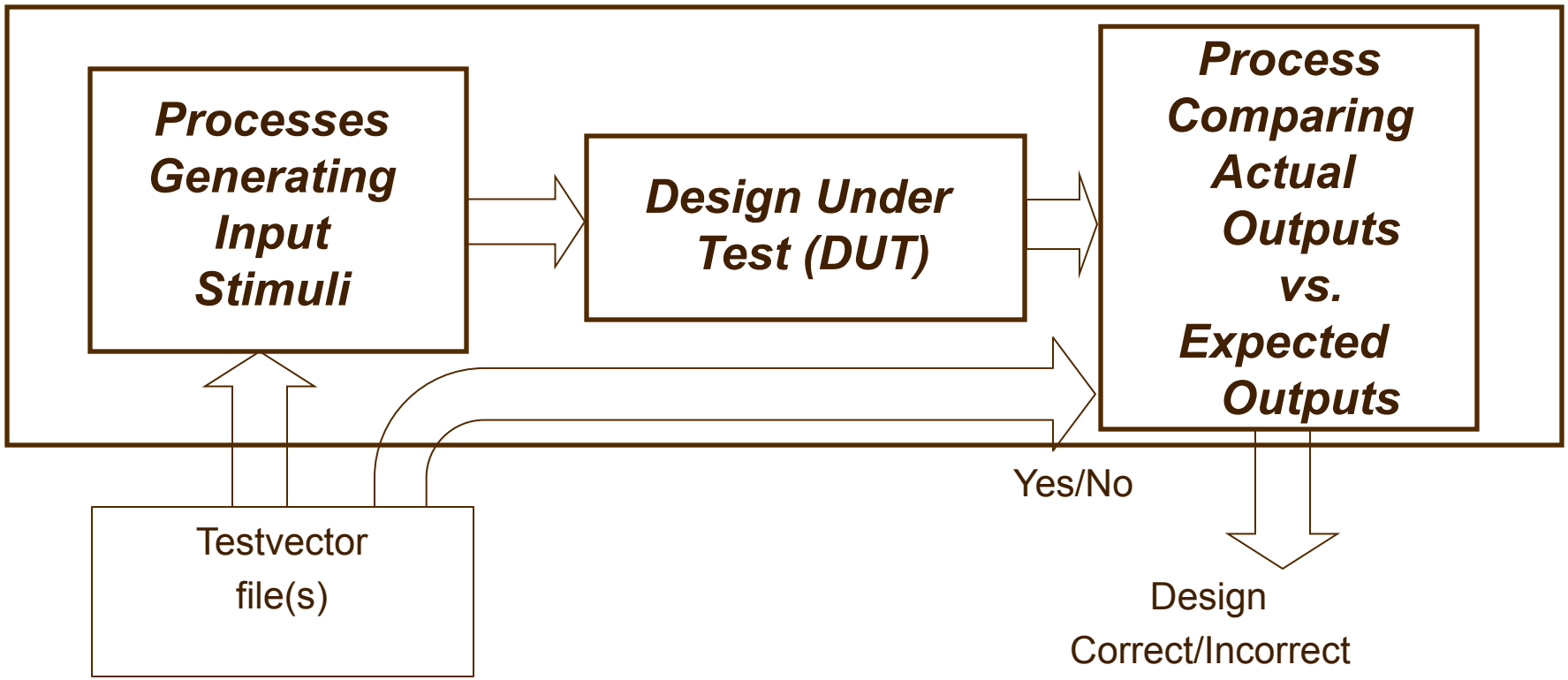


Verify outputs!

Testbench (5)

```
    WAIT;

    END PROCESS;

END testbench;
```

Advanced Testbench (2)




High Performance
Low Power
CoolClock
XILINX
CoolRunner-II

File I/O

File I/O Example

- Example of file input/output using a counter
- Text file is vectorfile.txt
  - Has both input data and EXPECTED output data
  - **Will compare VHDL output data with EXPECTED output data!**

Design Under Test (1)

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

ENTITY loadCnt IS
PORT (
  data:   IN STD_LOGIC_VECTOR (7 DOWNTO 0);
  load:   IN STD_LOGIC;
  clk:    IN STD_LOGIC;
  rst:    IN STD_LOGIC;
  q:      OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END loadCnt;
```

Design Under Test (2)

```
ARCHITECTURE rtl OF loadCnt IS

SIGNAL cnt: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
  counter: PROCESS (clk, rst)
BEGIN
    IF (rst = '1') THEN
      cnt <= (OTHERS => '0');
    ELSIF (clk'event AND clk = '1') THEN
      IF (load = '1') THEN
        cnt <= data;
      ELSE
        cnt <= cnt + 1;
      END IF;
    END IF;
  END PROCESS;
  q <= cnt;
END rtl;
```

Test vector file (1)

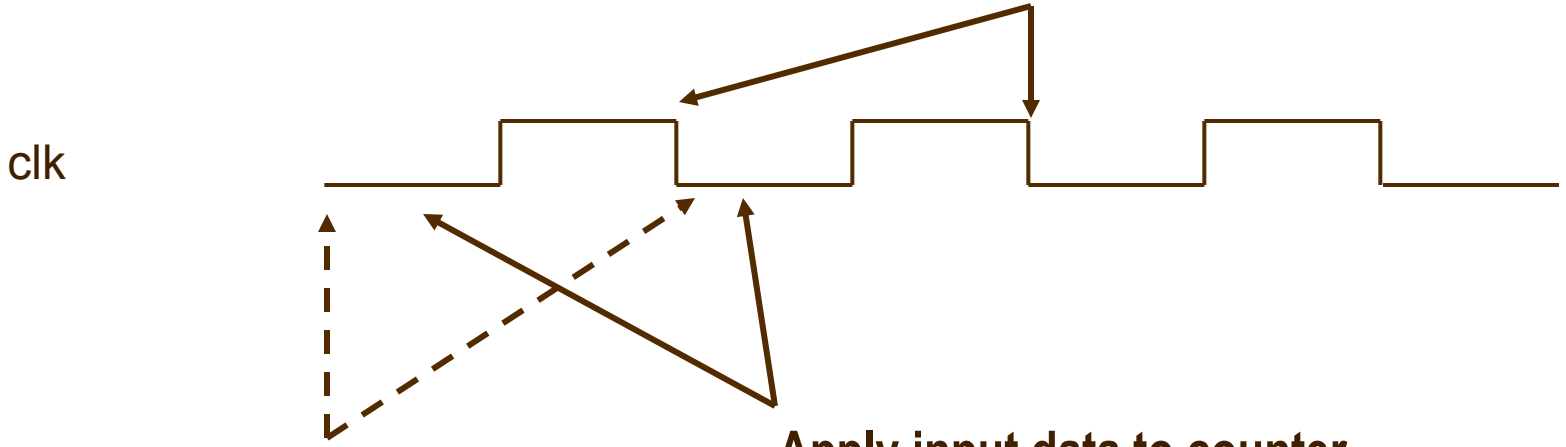
```
#Format is Rst, Load, Data, Q
#load the counter to all 1s
0 1 11111111 11111111
#reset the counter
1 0 10101010 00000000
#now perform load/increment for each bit
0 1 11111110 11111110
0 0 11111110 11111111
#
0 1 11111101 11111101
0 0 11111101 11111110
#
0 1 11111011 11111011
0 0 11111011 11111100
#
0 1 11110111 11110111
0 0 11110111 11111000
```

Test vector file (2)

```
#
0 1 11101111 11101111
0 0 11101111 11110000
#
0 1 11011111 11011111
0 0 11011111 11100000
#
0 1 10111111 10111111
0 0 10111111 11000000
#
0 1 01111111 01111111
0 0 01111111 10000000
#
#check roll-over case
0 1 11111111 11111111
0 0 11111111 00000000
#
# End vectors
```

Methodology to test vectors from file

Verify output is as expected:
compare Qout (the output of the VHDL counter)
with Qexpected (the expected value of Q from the test file)



read vector from text file into variables (vRst, vLoad, vData, vQ)

Apply input data to counter (i.e. rst <= vRst, load <= vLoad, reset <=vReset, data <= vData)

Testbench (1)

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

LIBRARY std;
USE std.textio.all;

ENTITY loadCntTB IS
END loadCntTB;
ARCHITECTURE testbench OF loadCntTB IS

COMPONENT loadCnt
  PORT (
     data:              IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
     load:              IN   STD_LOGIC;
     clk:               IN   STD_LOGIC;
     rst:               IN   STD_LOGIC;
     q:                 OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
END COMPONENT;
```

Testbench (2)

```
FILE vectorFile: TEXT OPEN READ_MODE is "vectorfile.txt";

SIGNAL Data:    STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL Load:    STD_LOGIC;
SIGNAL Rst:     STD_LOGIC;
SIGNAL Qout:    STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL Qexpected: STD_LOGIC_VECTOR(7 DOWNTO 0);

SIGNAL  TestClk:        STD_LOGIC := '0';
CONSTANT ClkPeriod:     TIME := 100 ns;
BEGIN

-- Free running test clock
  TestClk <= NOT TestClk AFTER ClkPeriod/2;

-- Instance of design being tested
  u1: loadCnt PORT MAP (Data => Data,
                        load => Load,
                        clk => TestClk,
                        rst => Rst,
                        q => Qout
                       );
```

Testbench (4)

```
-- File reading and stimulus application

  readVec: PROCESS
    VARIABLE   VectorLine:    LINE;
    VARIABLE   VectorValid:   BOOLEAN;
    VARIABLE   vRst:          STD_LOGIC;
    VARIABLE   vLoad:         STD_LOGIC;
    VARIABLE   vData:         STD_LOGIC_VECTOR(7 DOWNTO 0);
    VARIABLE   vQ:            STD_LOGIC_VECTOR(7 DOWNTO 0);
    VARIABLE   space:         CHARACTER;
```

Testbench (5)

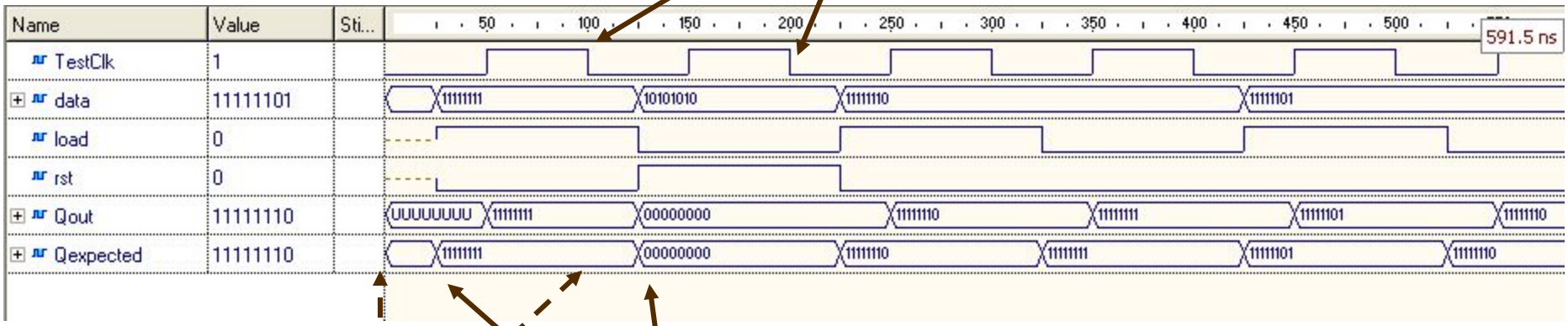
```
BEGIN
   WHILE NOT ENDFILE (vectorFile) LOOP
     readline(vectorFile, VectorLine); -- put file data into line

     read(VectorLine, vRst, good => VectorValid);
     NEXT WHEN NOT VectorValid;
     read(VectorLine, space);
     read(VectorLine, vLoad);
     read(VectorLine, space);
     read(VectorLine, vData);
     read(VectorLine, space);
     read(VectorLine, vQ);

     WAIT FOR ClkPeriod/4;
     Rst <= vRst;
     Load <= vLoad;
     Data <= vData;
     Qexpected <= vQ;

     WAIT FOR (ClkPeriod/4) * 3;
  END LOOP;
```

Testbench (6)

```
    ASSERT FALSE
      REPORT "Simulation complete"
      SEVERITY NOTE;
   WAIT;
END PROCESS;

-- Process to verify outputs
  verify: PROCESS (TestClk)
  variable ErrorMsg: LINE;
  BEGIN
    IF (TestClk'event AND TestClk = '0') THEN
      IF Qout /= Qexpected THEN
        write(ErrorMsg, STRING'("Vector failed "));
        write(ErrorMsg, now);
        writeline(output, ErrorMsg);
      END IF;
    END IF;
  END PROCESS;
END testbench;
```

Simulation Waveform

Verify output is as expected:
compare Q (the output of the VHDL counter)
with Qexpected
(the expected value of Q from the test file)



read vector from text file
into variables
(vRst, vLoad, vData, vQ)

Apply input data to counter
(i.e. rst <= vRst,
load <= vLoad,
reset <=vReset,
data <= vData)

Hex format

In order to read/write data in the hexadecimal notation, replace

read with hread, and

write with hwrite

Note on test file

- This example showed a test file that had both the control commands (i.e. load, reset), and the actual data itself
- Often the test file just has the input and output vectors (and no load, reset, etc.)