

ECE 331

Testbenches



High Performance

CoolClock

Low Power

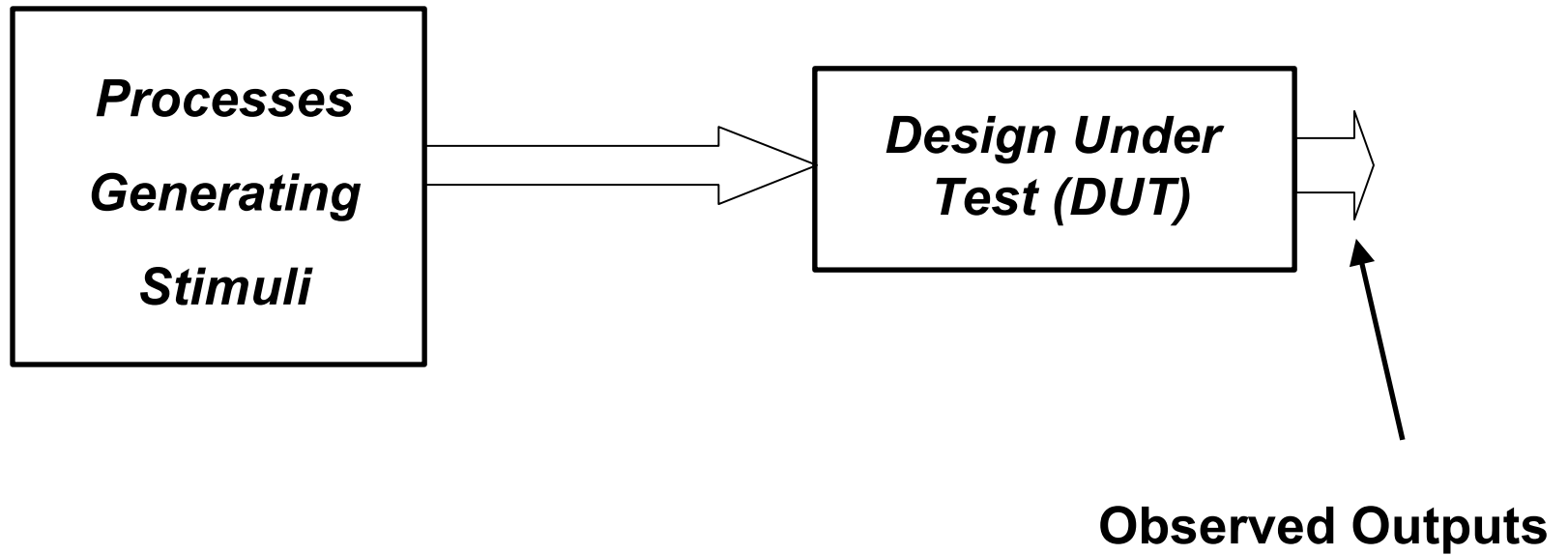
CoolCache



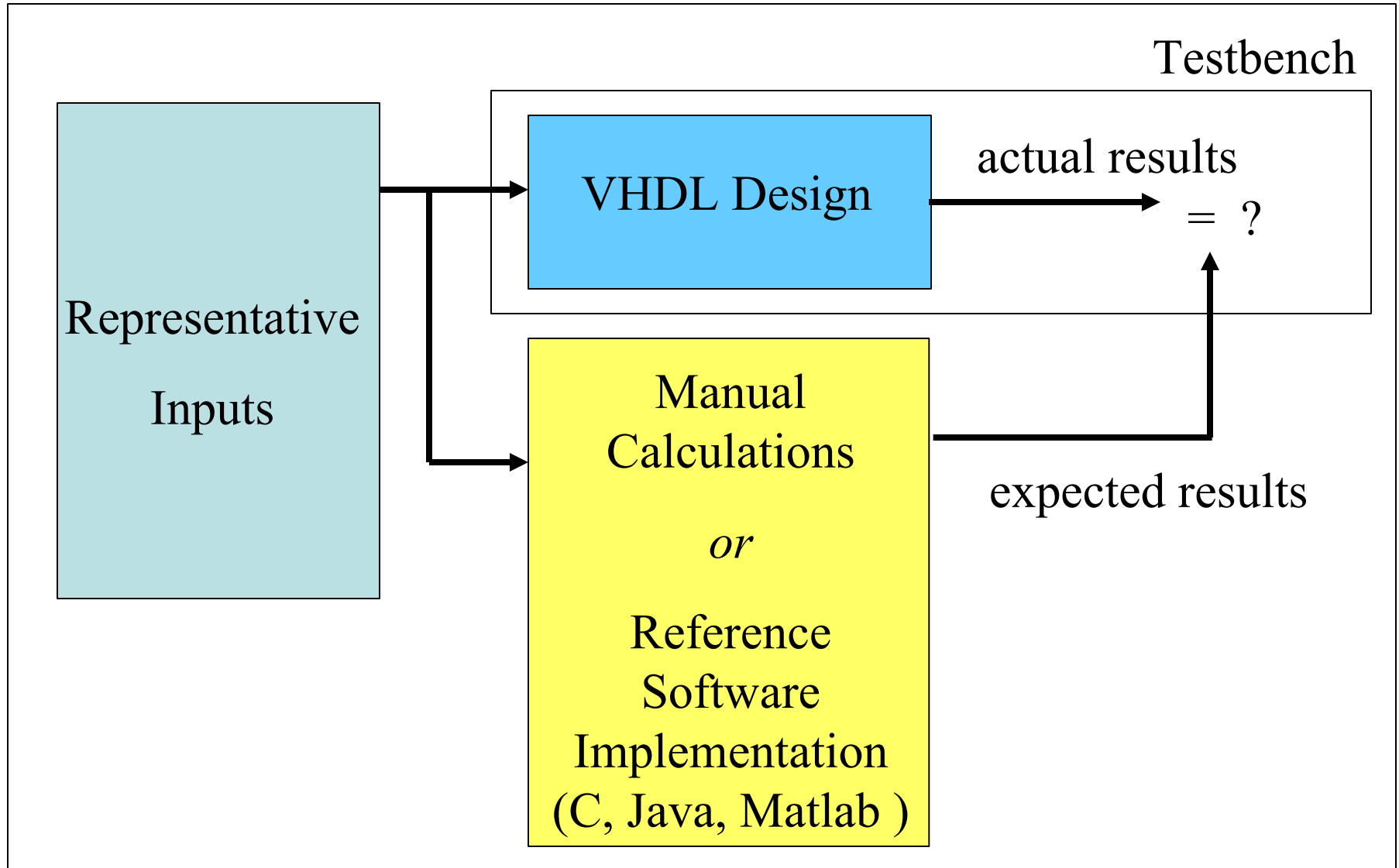
Testbench Defined

- *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.

Testbench

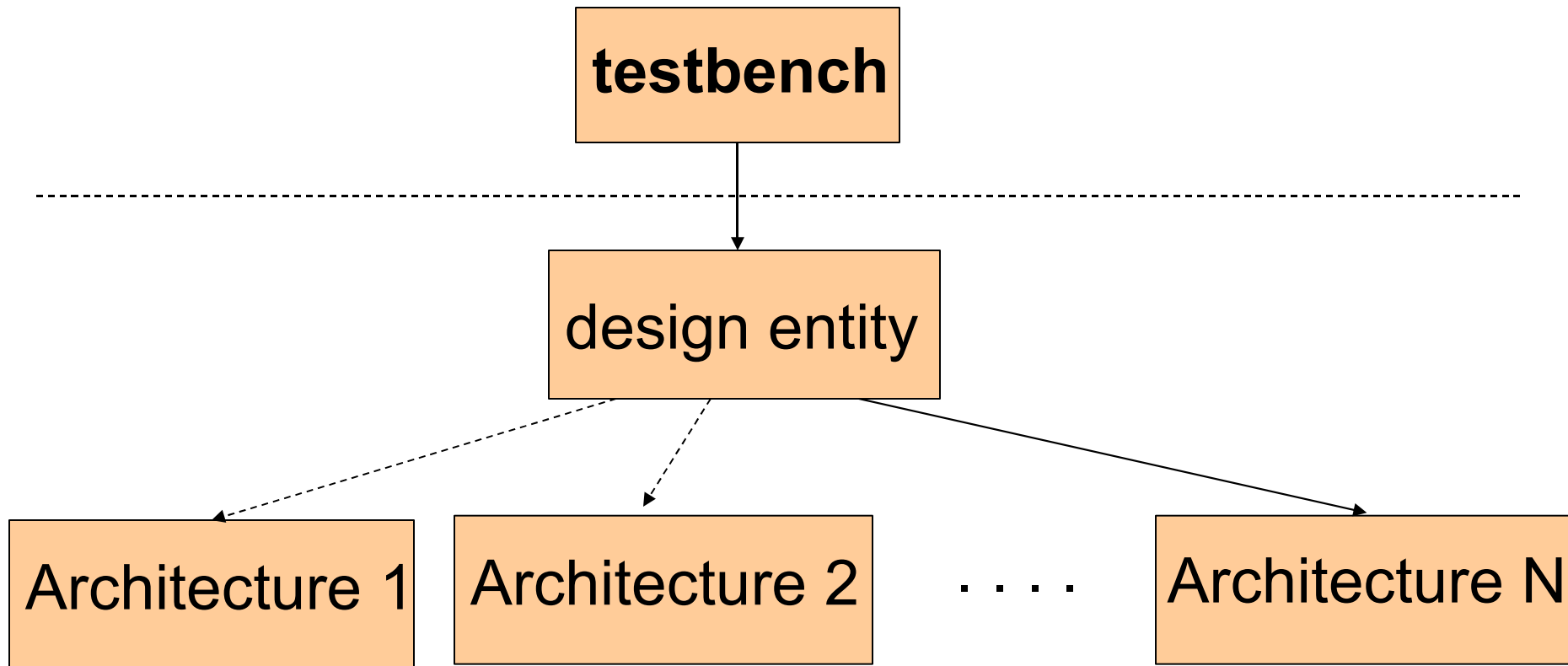


Possible sources of expected results used for comparison



Testbench

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



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)

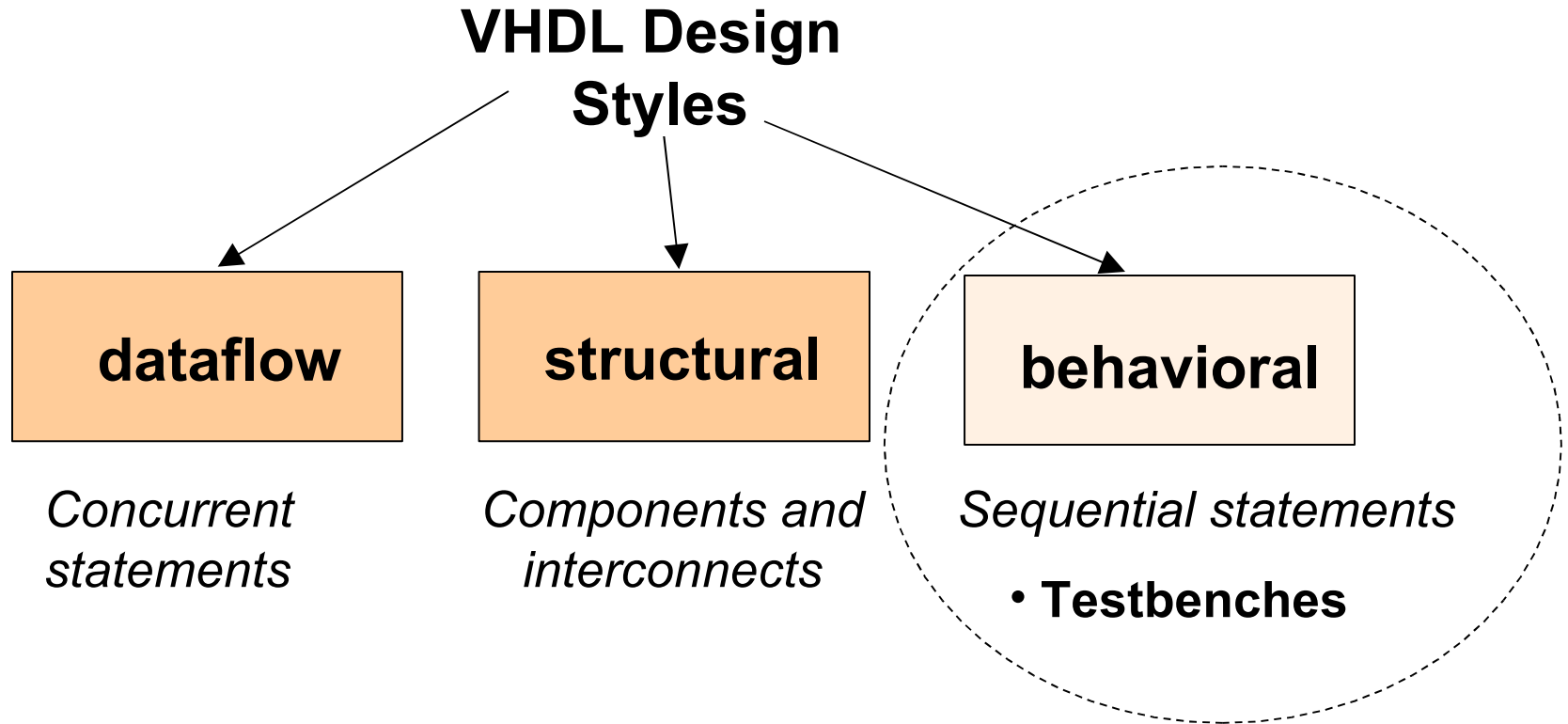
```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
  
ENTITY xor3_tb IS  
END xor3_tb;  
  
ARCHITECTURE behavioral OF xor3_tb IS  
  -- Component declaration of the tested unit  
  COMPONENT xor3  
  PORT(  
    A : IN STD_LOGIC;  
    B : IN STD_LOGIC;  
    C : IN STD_LOGIC;  
    Result : OUT STD_LOGIC );  
  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



What is a PROCESS?

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

The keyword PROCESS

- 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.

Testing: PROCESS
BEGIN

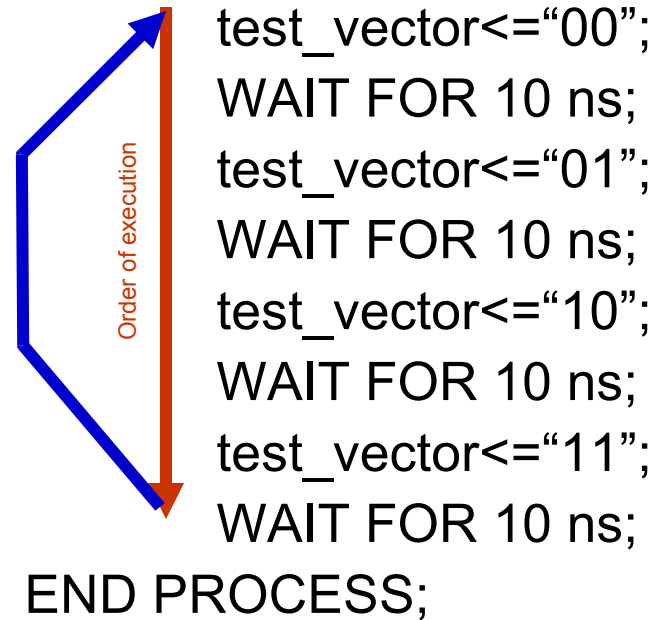
```
test_vector<="00";  
WAIT FOR 10 ns;  
test_vector<="01";  
WAIT FOR 10 ns;  
test_vector<="10";  
WAIT FOR 10 ns;  
test_vector<="11";  
WAIT FOR 10 ns;
```

END PROCESS;

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



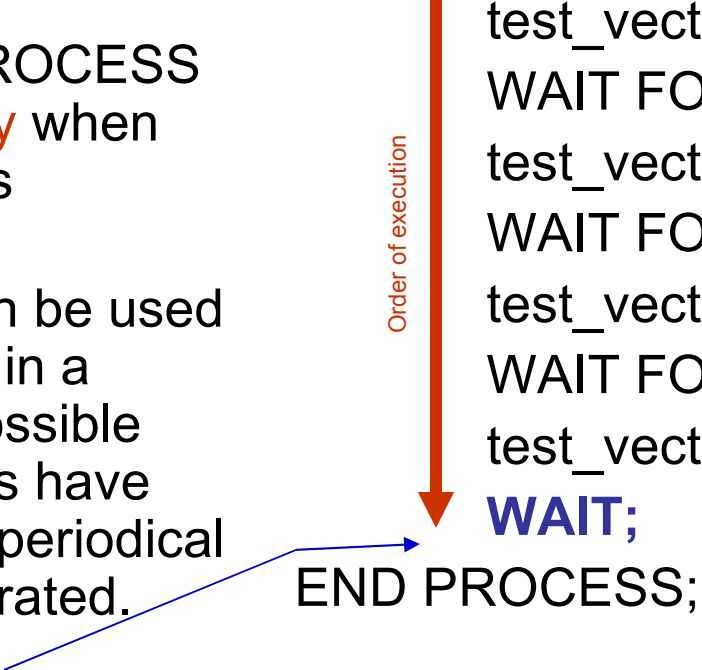
Program control is passed to the first statement after BEGIN

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

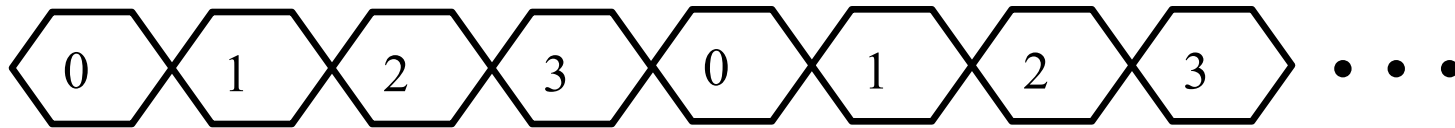
test_vector<="00";
WAIT FOR 10 ns;
test_vector<="01";
WAIT FOR 10 ns;
test_vector<="10";
WAIT FOR 10 ns;
test_vector<="11";
WAIT;
END PROCESS;



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.



The background is a light blue collage of various electronic components and text. At the top, there are two integrated circuits (ICs) in circular frames. Below them, a central IC is labeled 'CoolRunner-II'. To the left, a circular frame shows a circuit board. To the right, another circular frame shows a component. At the bottom left, a circular frame shows a mobile phone. At the bottom right, a circular frame shows a keyboard. The text 'High Performance' is written diagonally on the left, 'CoolClock' on the right, and 'Low Power' at the bottom left. The main title is centered in a large, bold, dark blue font.

**Simple
Testbenches
for Periodical
and Non-periodical
Control Signals
(Clock, Reset, etc.)**

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;
```

Specifying time in VHDL



High Performance

CoolClock

Low Power

Low Power



Physical data types

Types representing physical quantities, such as time, voltage, capacitance, etc. are referred in VHDL as **physical data types**.

TIME is the only **predefined** physical data type.

Value of the physical data type is called a **physical literal**.

Time values (physical literals) - Examples

7 ns

1 min

min

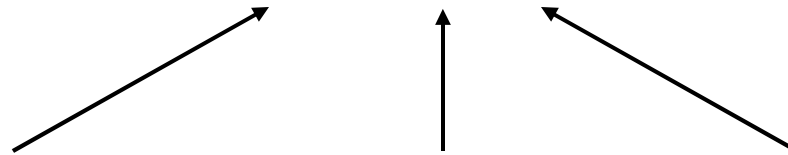
10.65 us

10.65 fs

Numeric value

Space

Unit of time
(dimension)



TIME values

Numeric value can be an **integer** or **a floating point number**.

Numeric value is **optional**. If not given, 1 is implied.

Numeric value and dimension **MUST** be separated by a **space**.

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 milliseconds (10^{-3} seconds)

sec seconds

min minutes (60 seconds)

hr hours (3600 seconds)

Simple Testbenches For Data Inputs



High Performance

CoolClock

Low Power

CoolCache

Loop Statement – Example (1)

```
Testing: PROCESS
BEGIN
    test_vector<="000";
    FOR i IN 0 TO 7 LOOP
        WAIT FOR 10 ns;
        test_vector<=test_vector+"001";
    END LOOP;
END PROCESS;
```

Loop Statement

- Loop Statement

```
FOR i IN range LOOP  
    statements  
END LOOP;
```

- Repeats a Section of VHDL Code
 - Example: process every element in an array in the same way

Loop Statement – Example (2)

```
Testing: 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+"01";
        END LOOP;
        test_sel<=test_sel+"01";
    END LOOP;
END PROCESS;
```

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;
```

Generating all possible values of two inputs

```
SIGNAL test_ab : STD_LOGIC_VECTOR(2 downto 0);  
SIGNAL test_sel : STD_LOGIC_VECTOR(2 downto 0);
```

```
BEGIN
```

```
.....
```

```
    double_loop: PROCESS  
    BEGIN  
        test_ab <="000";  
        test_sel <="000";  
        for I in 0 to 7 loop  
            for J in 0 to 7 loop  
                wait for 10 ns;  
                test_ab <= test_ab + 1;  
            end loop;  
            test_sel <= test_sel + 1;  
        end loop;  
    END PROCESS;
```

```
.....
```

```
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 - Example

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.