EDA321
Digital Design

Lecture 10:
Testbenches - VHDL

Ioannis Sourdis

Outline of Lecture 10

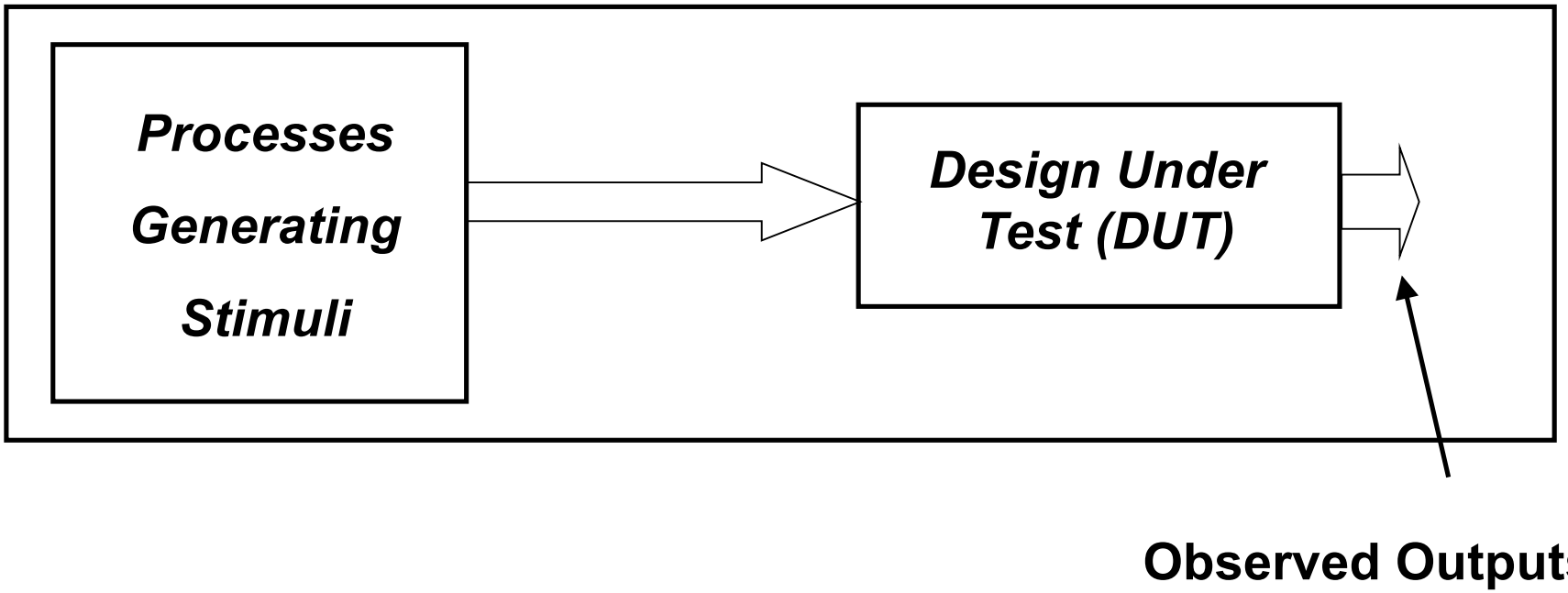
- Introduction to Testbenches
- Structure of a testbench
- Time, wait statements, Processes
- Simple testbenches
- Advanced testbenches
- Test-vectors, Asserts & Reports
- Records and variables
- File IO

Testbenches

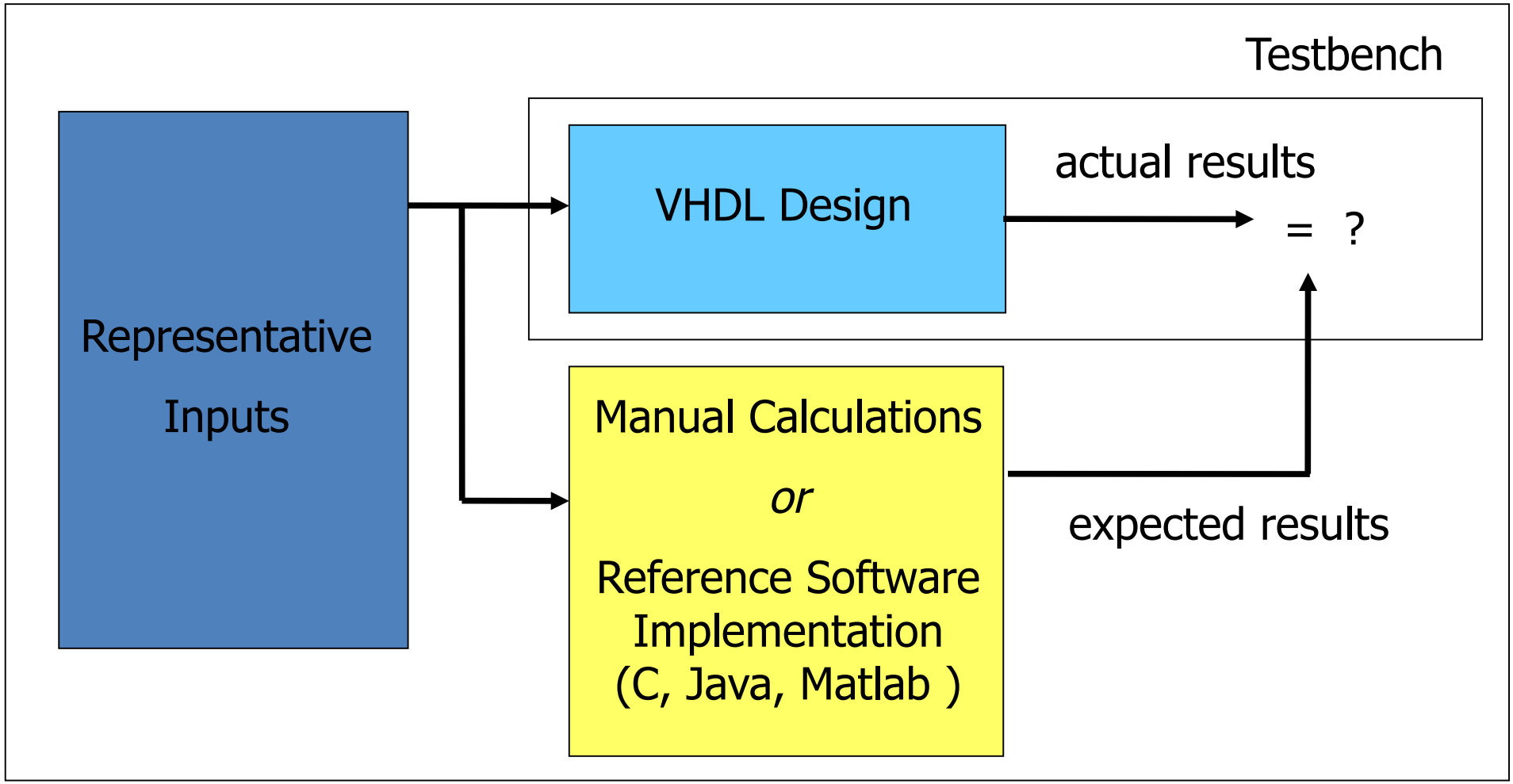
Testbench Defined

- *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

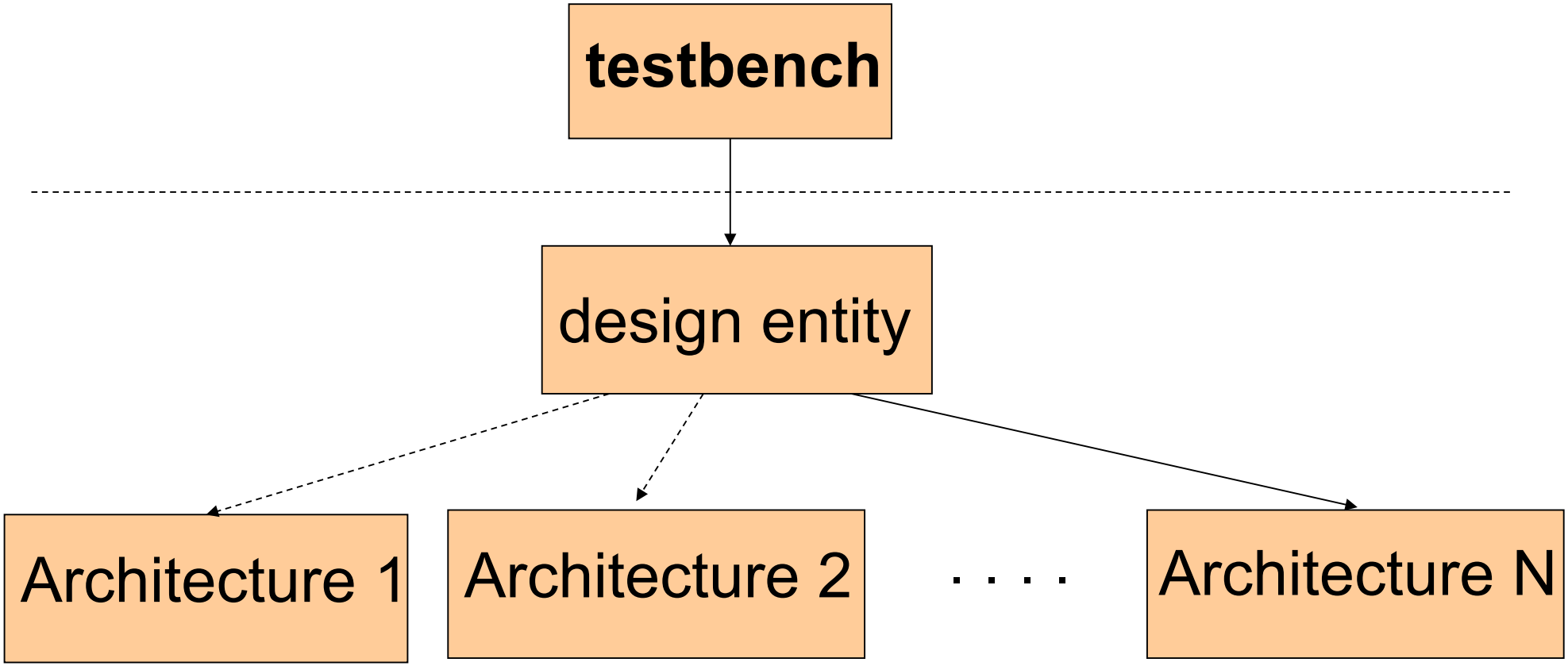


Possible sources of expected results used for comparison



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;
```

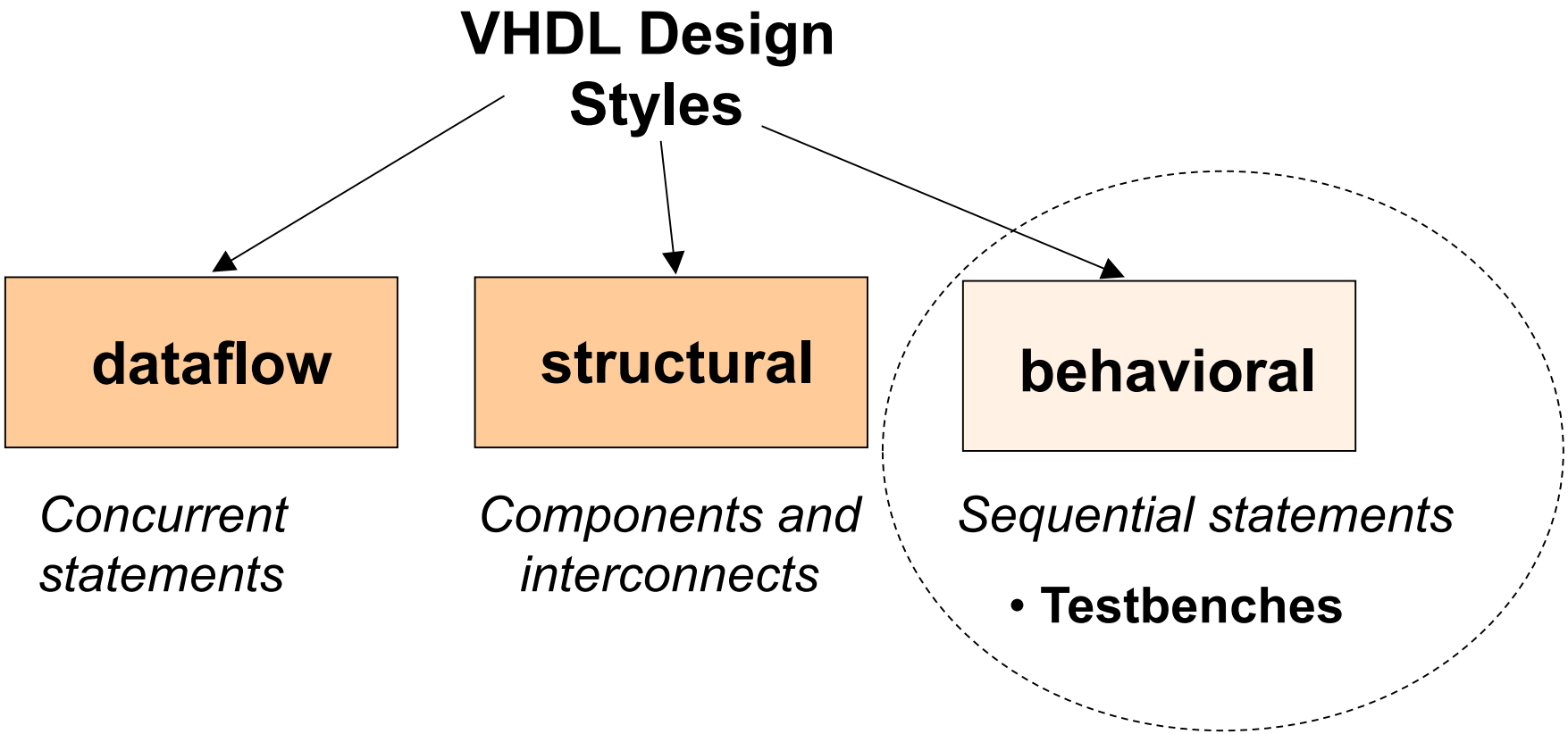
Testbench for XOR3 (1)

```
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

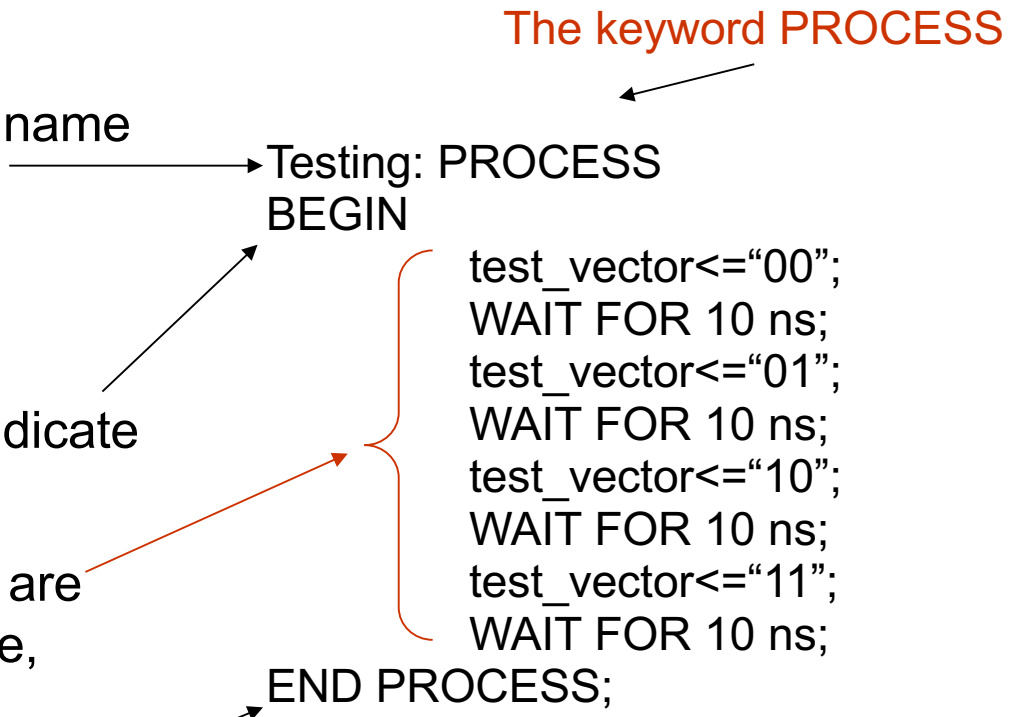


Process without Sensitivity List and its use in Testbenches

What is a PROCESS?

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

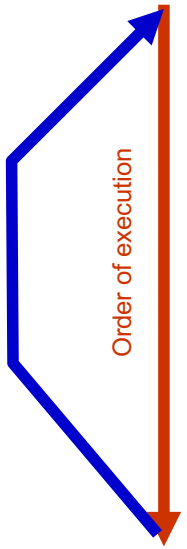
- A process can be given a unique name using an optional LABEL
- This is followed by the keyword PROCESS
- The keyword BEGIN is used to indicate the start of the process
- All statements within the process are executed SEQUENTIALLY. Hence, order of statements is important.
- A process must end with the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

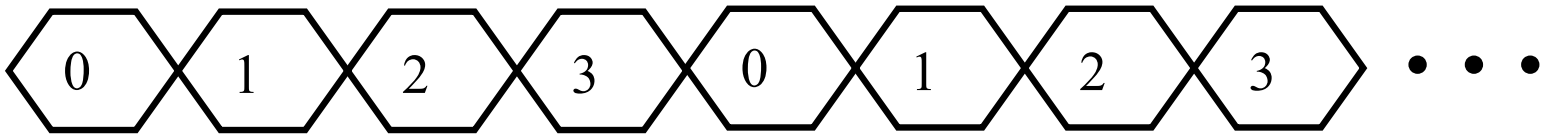
```
Testing: PROCESS
BEGIN
    test_vector<="00";
    WAIT FOR 10 ns;
    test_vector<="01";
    WAIT FOR 10 ns;
    test_vector<="10";
    WAIT FOR 10 ns;
    test_vector<="11";
    WAIT;
END PROCESS;
```

Order of execution

Program execution stops here

WAIT FOR vs. WAIT

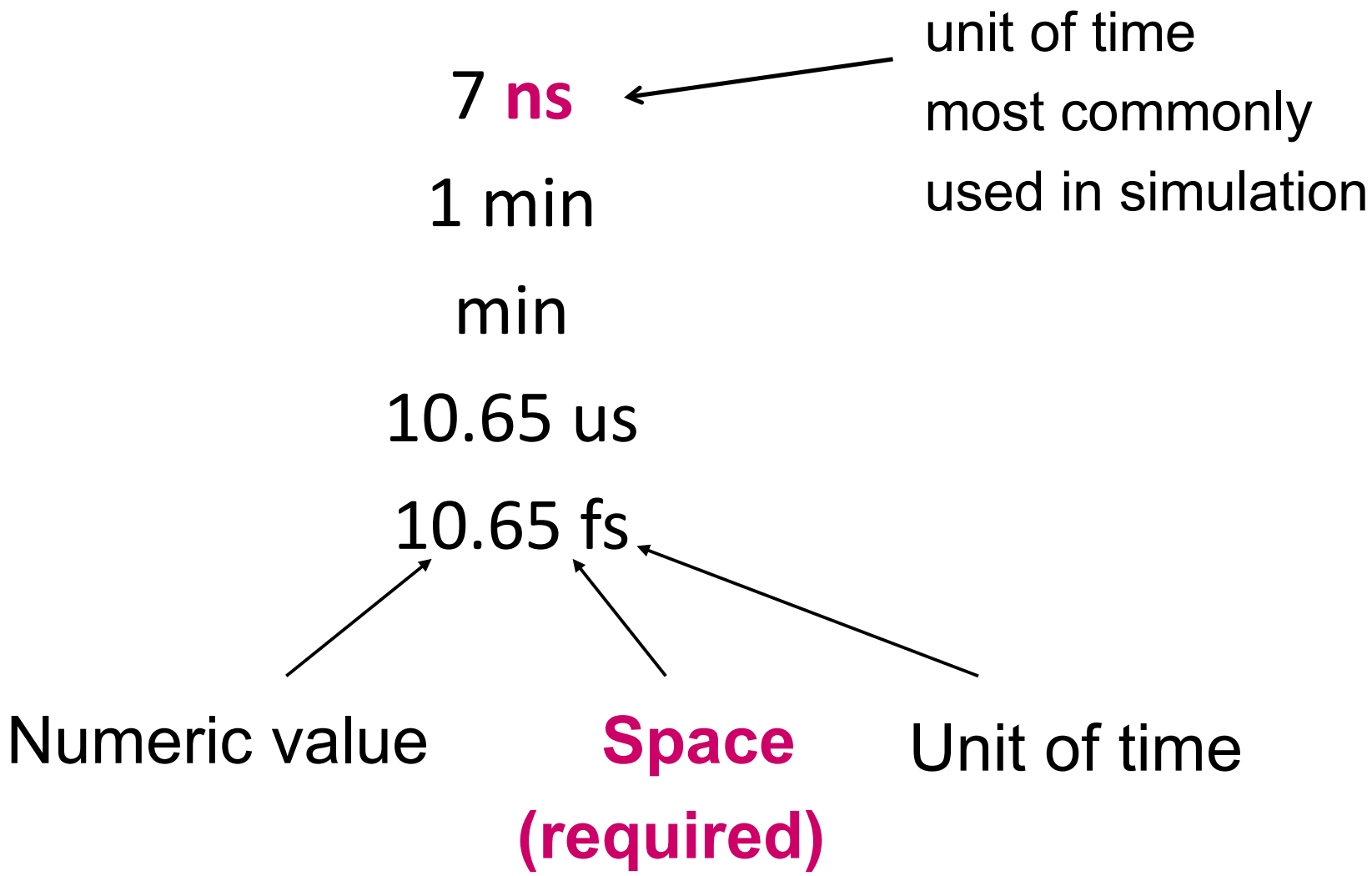
WAIT FOR: waveform will keep repeating itself forever



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

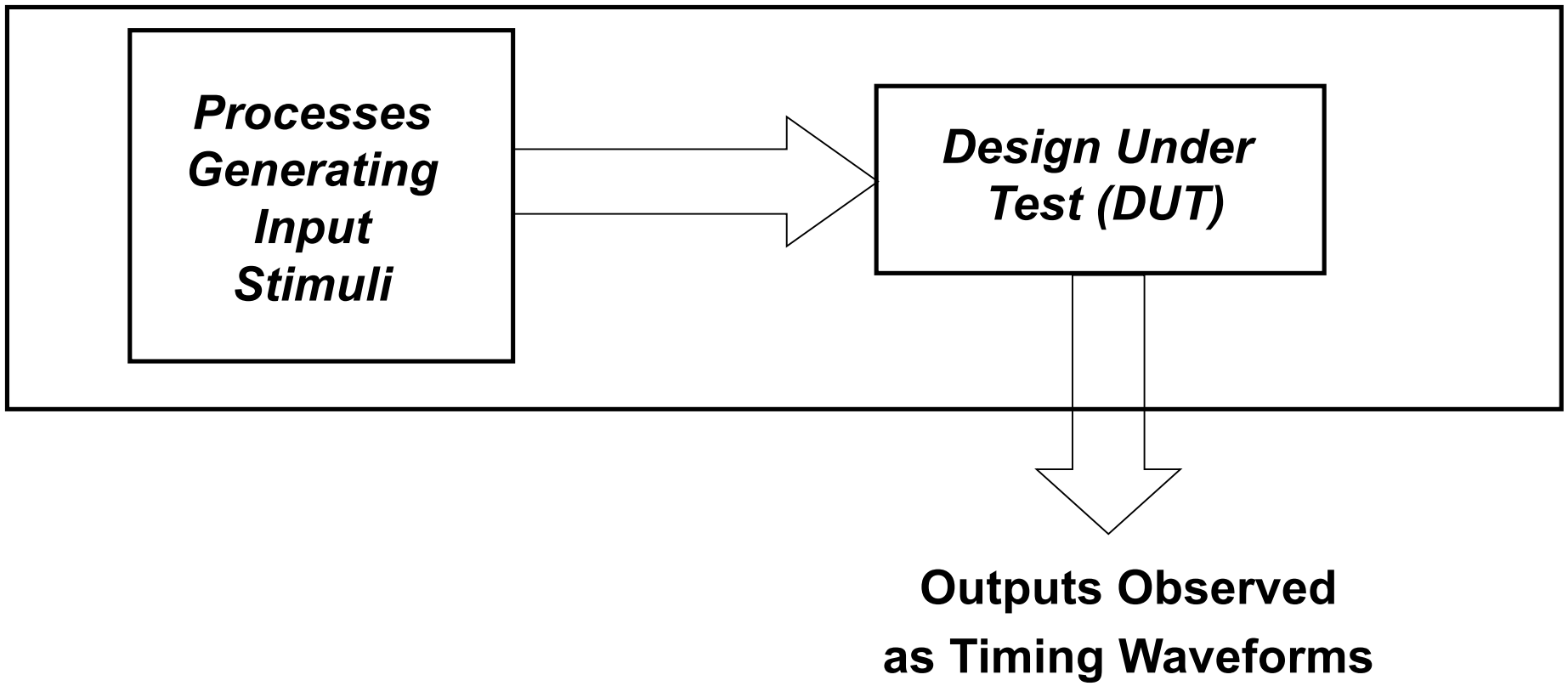
```
    CONSTANT reset1_width : TIME := 100 ns;
    CONSTANT reset2_width : TIME := 150 ns;
    SIGNAL reset1 : STD_LOGIC;
    SIGNAL reset2 : STD_LOGIC := '1';

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

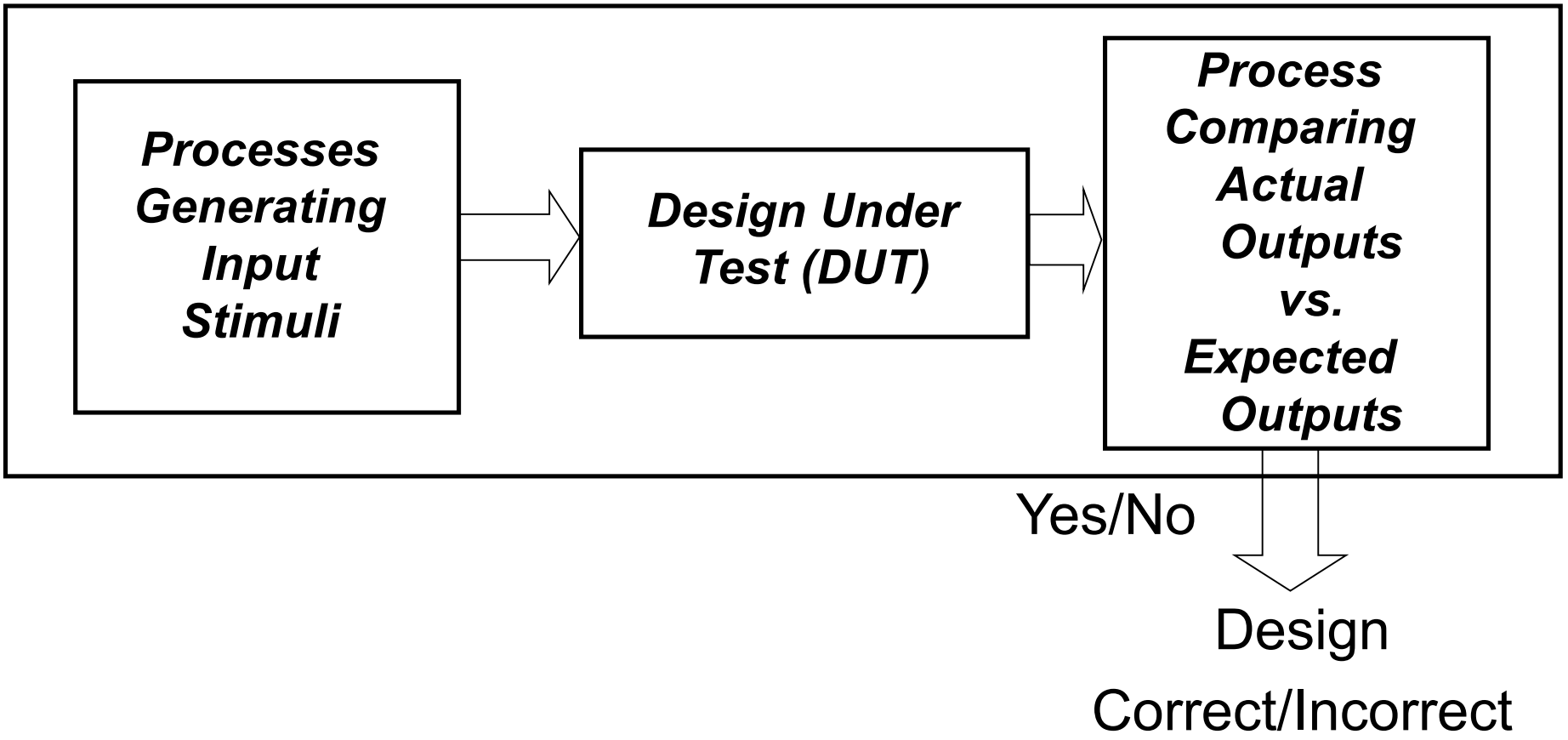
```
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL reset : STD_LOGIC;

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;
```

Advanced Testbenches

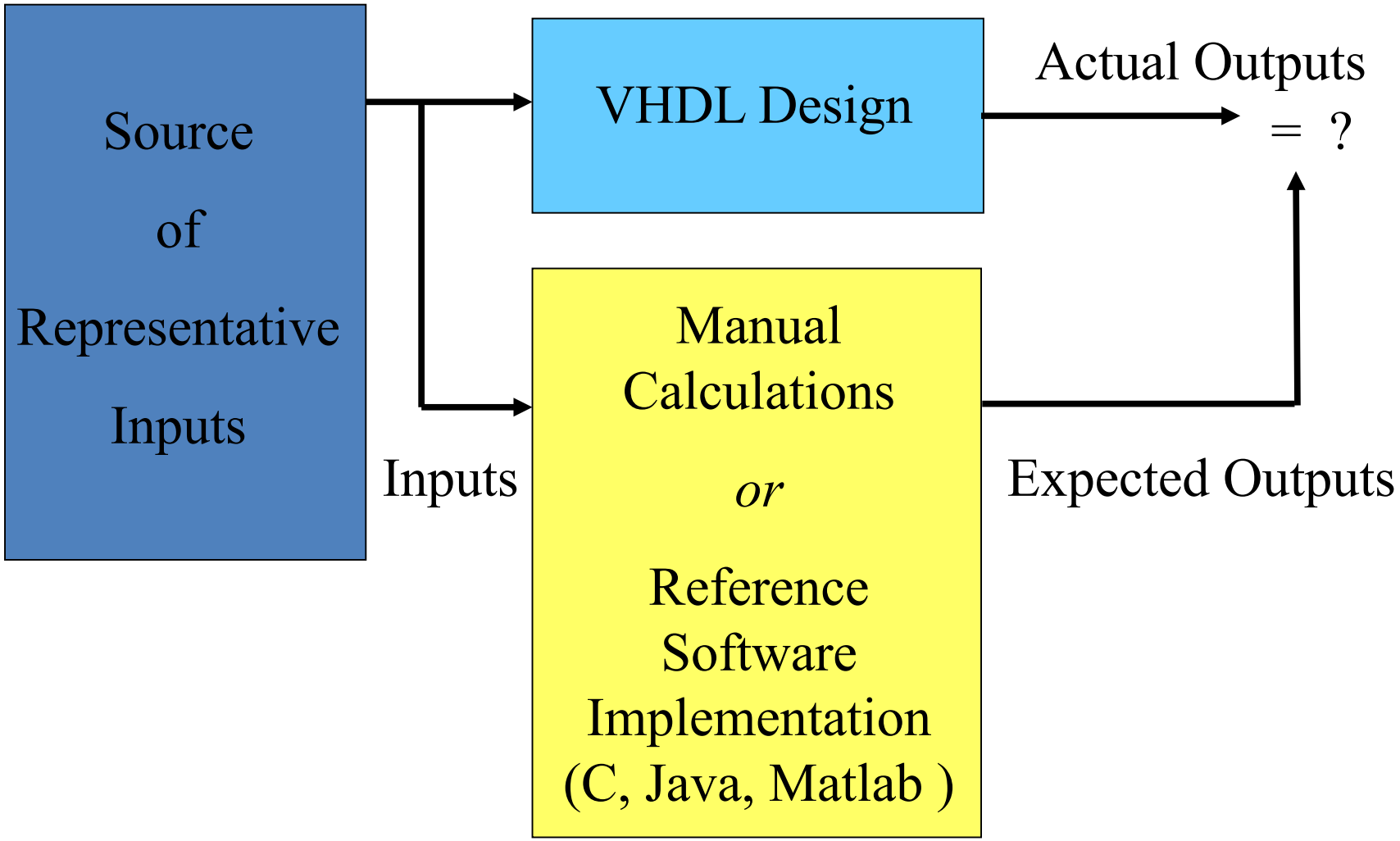
Simple Testbench



Advanced Testbench



Possible Sources of Expected Outputs



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

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;
```

Generating reports in the message window

```
reports: process(clk_trigger) begin
  if (clk_trigger = '0' and clk_trigger'EVENT) then
    case segments is
      when seg_0 => report time'image(now) & ": 0 is displayed" ;
      when seg_1 => report time'image(now) & ": 1 is displayed" ;
      when seg_2 => report time'image(now) & ": 2 is displayed" ;
      when seg_3 => report time'image(now) & ": 3 is displayed" ;
      when seg_4 => report time'image(now) & ": 4 is displayed" ;
      when seg_5 => report time'image(now) & ": 5 is displayed" ;
      when seg_6 => report time'image(now) & ": 6 is displayed" ;
      when seg_7 => report time'image(now) & ": 7 is displayed" ;
      when seg_8 => report time'image(now) & ": 8 is displayed" ;
      when seg_9 => report time'image(now) & ": 9 is displayed" ;
    end case;
  end if;
end process;
```

Records

Records

```
type opcodes is (add, sub, and, or);
type reg_number is range 0 to 8;

type instruction is record
    opcode        : opcodes;
    source_reg1   : reg_number;
    source_reg2   : reg_number;
    dest_reg      : reg_number;
end record instruction;

constant  add_instr_1_3 :  instruction:=
    (opcode => add,
     source_reg1 | dest_reg => 1,
    source_reg2 => 3);
```

Variables

Variable – Example (1)

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

ENTITY Numbits IS
    PORT ( X              : IN     STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                Count     : OUT  INTEGER RANGE 0 TO 16) ;
END Numbits ;
```

Variable – Example (2)

```
ARCHITECTURE Behavior OF Numbits IS

BEGIN

    PROCESS(X) – count the number of bits in X equal to 1
            VARIABLE Tmp: INTEGER;
    BEGIN
       Tmp := 0;
       FOR i IN 15 DOWNTO 0 LOOP
           IF X(i) = '1' THEN
                Tmp := Tmp + 1;
           END IF;
      END LOOP;
      Count <= Tmp;
    END PROCESS;

END Behavior ;
```

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
- Should be avoided, or at least used with caution in a synthesizable code

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

COMPONENT sevenSegment
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)

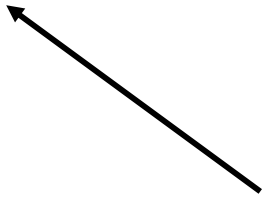
```
TYPE vector IS RECORD
  bcdStimulus:          STD_LOGIC_VECTOR(3 downto 0);
  sevSegOut:            STD_LOGIC_VECTOR(6 downto 0);
END RECORD;

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;
   WAIT;
END PROCESS;
END testbench;
```



Verify outputs!

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;
```

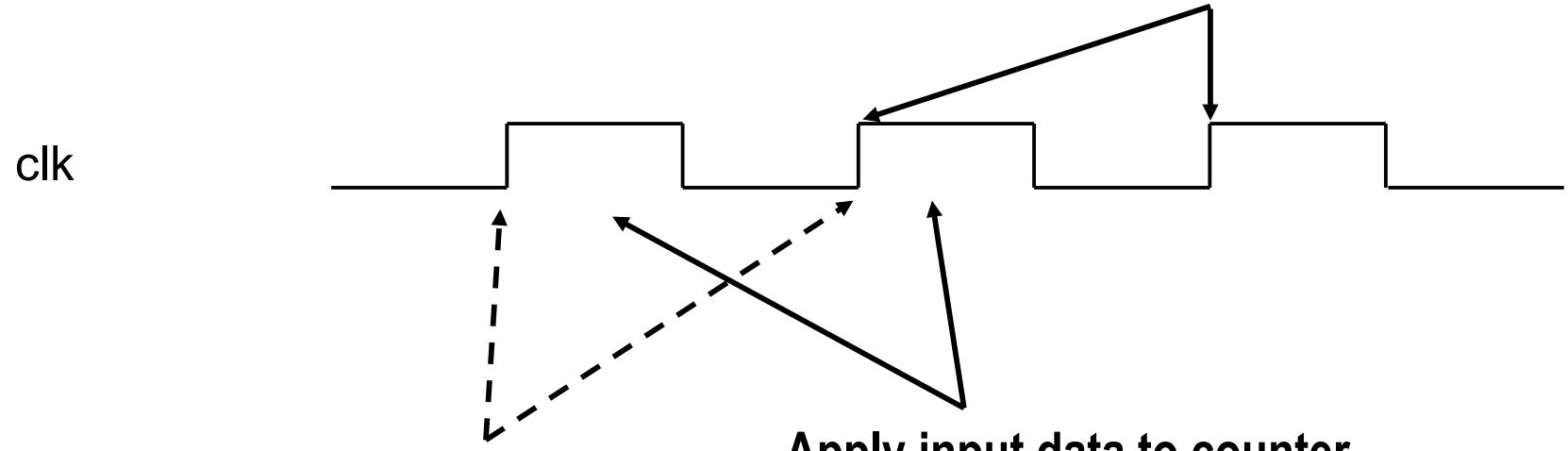
Test vector file (1)

```
#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



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 (3)

```
-- 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 (4)

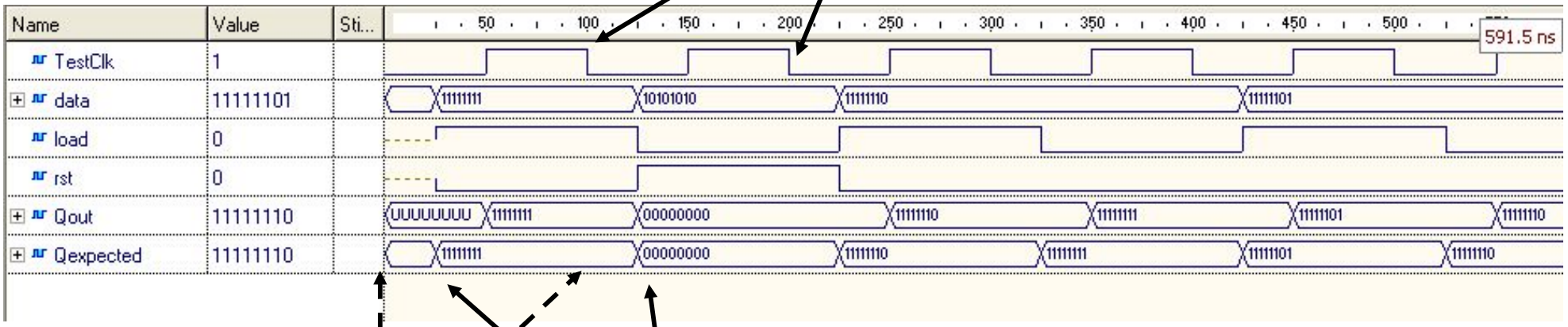
```
BEGIN
   WHILE NOT ENDFILE (vectorFile) LOOP
     readline(vectorFile, VectorLine); -- put file data into line

     read(VectorLine, vRst, good => VectorValid);
     NEXT WHEN NOT VectorValid;   -- "next" causes goes to the next
                                  --iteration of the while loop
     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 (5)

```
    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 vQ 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

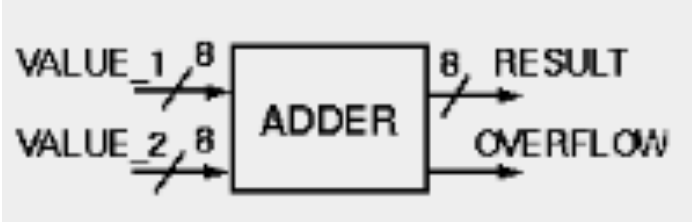
write with hwrite

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.)

Example2 for File I/O

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ADDER is
  port(VALUE_1     : in   std_logic_vector(7 downto 0);
       VALUE_2     : in   std_logic_vector(7 downto 0);
       OVERFLOW : out std_logic;
       RESULT      : out std_logic_vector(7 downto 0));
end ADDER;

architecture RTL of ADDER is
  signal INT_RES    : std_logic_vector(8 downto 0);
  signal INT_VAL_1 : std_logic_vector(8 downto 0);
  signal INT_VAL_2 : std_logic_vector(8 downto 0);
Begin
  INT_VAL_1   <= '0' & VALUE_1;
  INT_VAL_2   <= '0' & VALUE_2;
  INT_RES     <= INT_VAL_1 + INT_VAL_2;
  RESULT      <= INT_RES(7 downto 0);
  OVERFLOW <= INT_RES(8);
end RTL;
```



- ADDER module
  - Adds two 8 bit vectors and provides an 8 bit result vector
  - Generates an overflow signal
- Entity and architecture of the ADDER module
- std_logic_unsigned package
  - overloaded mathematical operators where std_logic_vector is treated as unsigned number

```
library IEEE;
        use IEEE.std_logic_1164.all;
        use IEEE.std_logic_unsigned.all;
        use IEEE.std_logic_textio.all;
        use STD.textio.all;


entity TB_ADDER is
        end TB_ADDER;

        architecture BEH of TB_ADDER is

          component ADDER
            port(VALUE_1     : in   std_logic_vector(7 downto 0);
                 VALUE_2     : in   std_logic_vector(7 downto 0);
                 OVERFLOW : out std_logic;
                 RESULT      : out std_logic_vector(7 downto 0));
          end component;

          signal W_VALUE_1     : std_logic_vector(7 downto 0);
          signal W_VALUE_2     : std_logic_vector(7 downto 0);
          signal W_OVERFLOW : std_logic;
          signal W_RESULT      : std_logic_vector(7 downto 0);

        begin
          DUT : ADDER
            port map(VALUE_1     => W_VALUE_1,
                     VALUE_2     => W_VALUE_2,
                     OVERFLOW => W_OVERFLOW,
                     RESULT      => W_RESULT);
```

Example2 for File I/O

- Add package std.textio and IEEE.std_logic_textio for file I/O functions and procedures
- Common testbench structure
  - Empty entity; no external interface
  - Component declaration and instantiation
  - Definition of internal signals to connect the input/output ports with the stimuli/response analysis processes

Example2 for File I/O

```
STIMULI : process
        variable L_IN : line;
        variable CHAR : character;
        variable DATA_1 : std_logic_vector(7 downto 0);
        variable DATA_2 : std_logic_vector(7 downto 0);
        file STIM_IN : text open read_mode is "stim_in.txt";
                begin
                        W_VALUE_1 <= (others => '0');
                        W_VALUE_2 <= (others => '0');
                        wait for PERIOD;
                        while not endfile (STIM_IN) loop
                                readline (STIM_IN, L_IN);
                                hread (L_IN, DATA_1);
                                 W_VALUE_1 <= DATA_1;
                                read (L_IN, CHAR);
                                hread (L_IN, DATA_2);
                                W_VALUE_2 <= DATA_2;
                                wait for PERIOD;
                        end loop;
                        wait;
    end process STIMULI;
```

```
==================
    FF 01
    FF 00
    11 55
    0F 01
    1F 05
```

- STIMULI process
  - File access is limited to only one line at a certain time
  - Only variables are allowed for the parameters of the read functions
  - The function hread(...) is defined in the IEEE.std_logic_textio package; it reads hex values and transforms them into a binary vector
- Stimuli file "stim_in.txt"
  - Each line contains two hex values to stimulate the inputs of the ADDER module

Example2 for File I/O

```
RESPONSE : process(W_RESULT)
        variable L_OUT : line;
        variable CHAR_SPACE : character := ' ';
        file STIM_OUT : text open write_mode is "stim_out.txt";
        begin
                write (L_OUT, now);
                write (L_OUT, CHAR_SPACE);
                write (L_OUT, W_RESULT);
                write (L_OUT, CHAR_SPACE);
                hwrite (L_OUT, W_RESULT);
                write (L_OUT, CHAR_SPACE);
                write (L_OUT, W_OVERFLOW);
                writeline (STIM_OUT, L_OUT);
      end process RESPONSE;
```

```
==================
        0 ns XXXXXXXX 00 X
        0 ns 00000000 00 0
        20 ns 10100001 A1 0
        40 ns 00000000 00 1
        60 ns 11111111 FF 0
        80 ns 01100110 66 0
        100 ns 00010000 10 0
        120 ns 00100100 24 0
        140 ns 10011101 9D 1
```

- Response process of the testbench
  - 'NOW' is a function returning the current simulation time
  - Several write commands assemble a line
  - Writeline saves this line in the file
  - The function hwrite(...) is defined in the IEEE.std_logic_textio package; it transforms a binary vector to a hex value and stores it in the line
- Response file "stim_out.txt"
  - 4 columns containing:
    - Simulation time
    - 8 bit result value (binary and hex)
    - Overflow bit

Summary of Lecture 10

- Introduction to Testbenches
- Structure of a testbench
- Time, wait statements, Processes
- Simple testbenches
- Advanced testbenches
- Test-vectors, Asserts & Reports
- Records and variables
- File IO

- The book does not cover any parts of this lecture

- Next Lecture 11:
  - FPGA Technology