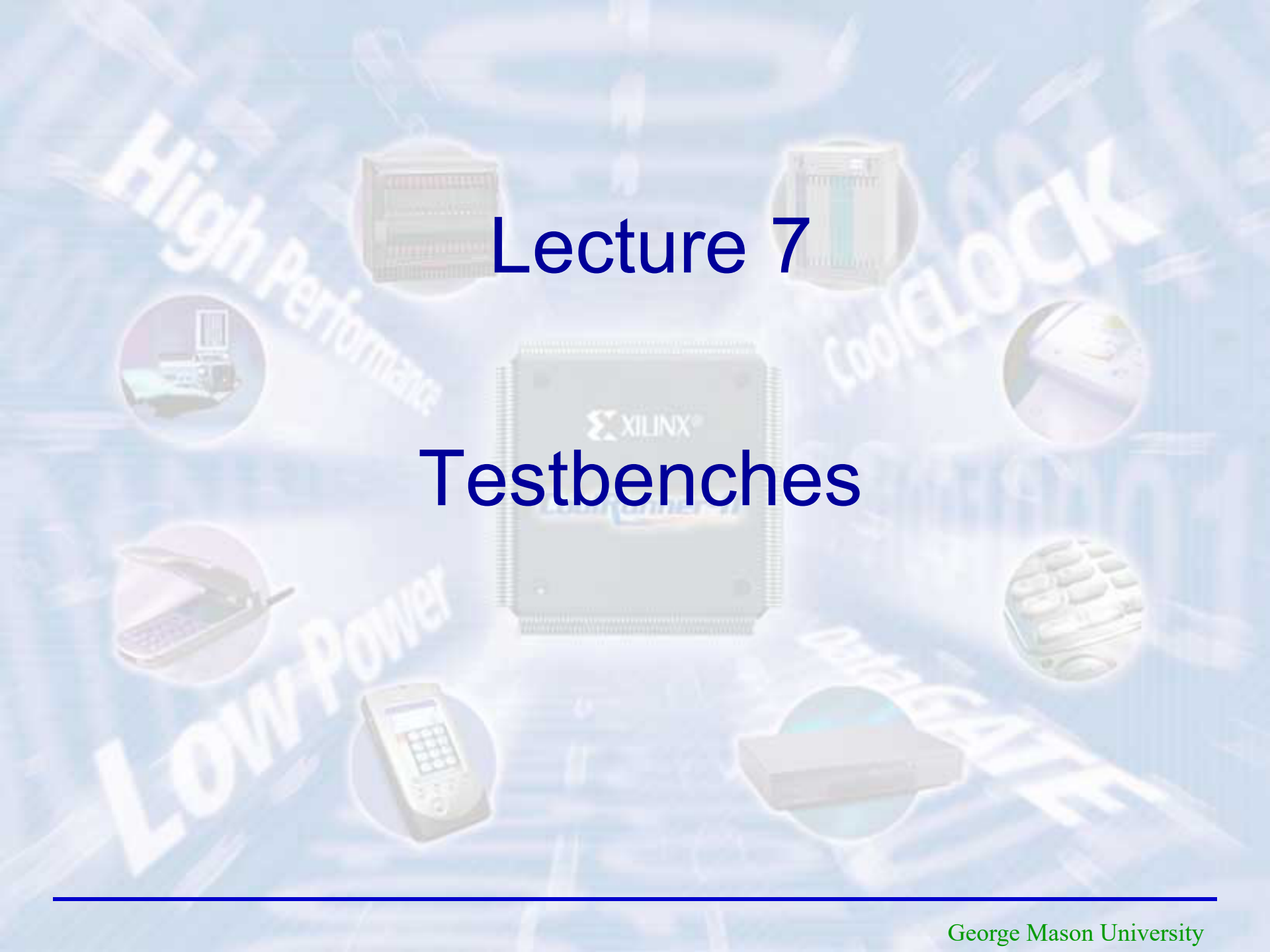


Lecture 7

Testbenches

Required reading

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

  Chapter 14, starting from "Design Verification"

  (posted on Piazza under

  Resources => Resources => Lecture Notes =>

  Reading_about_Testbenches.pdf)

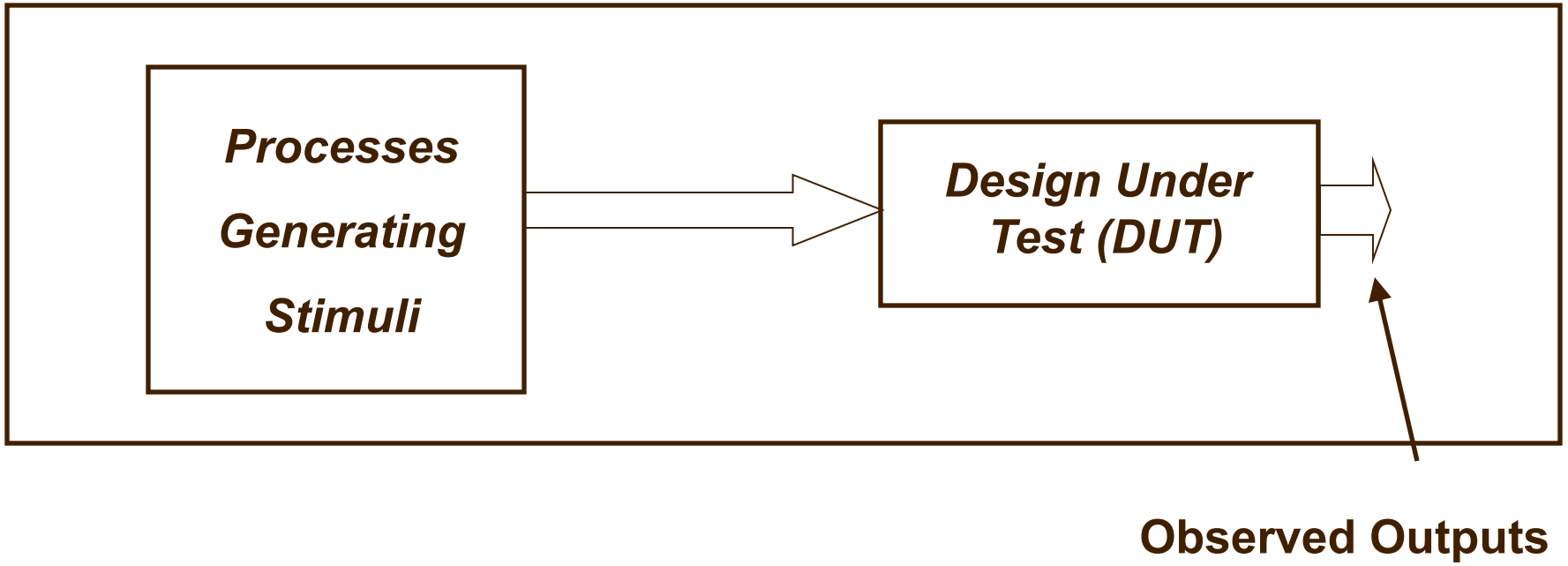
Steps of the Design Process

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

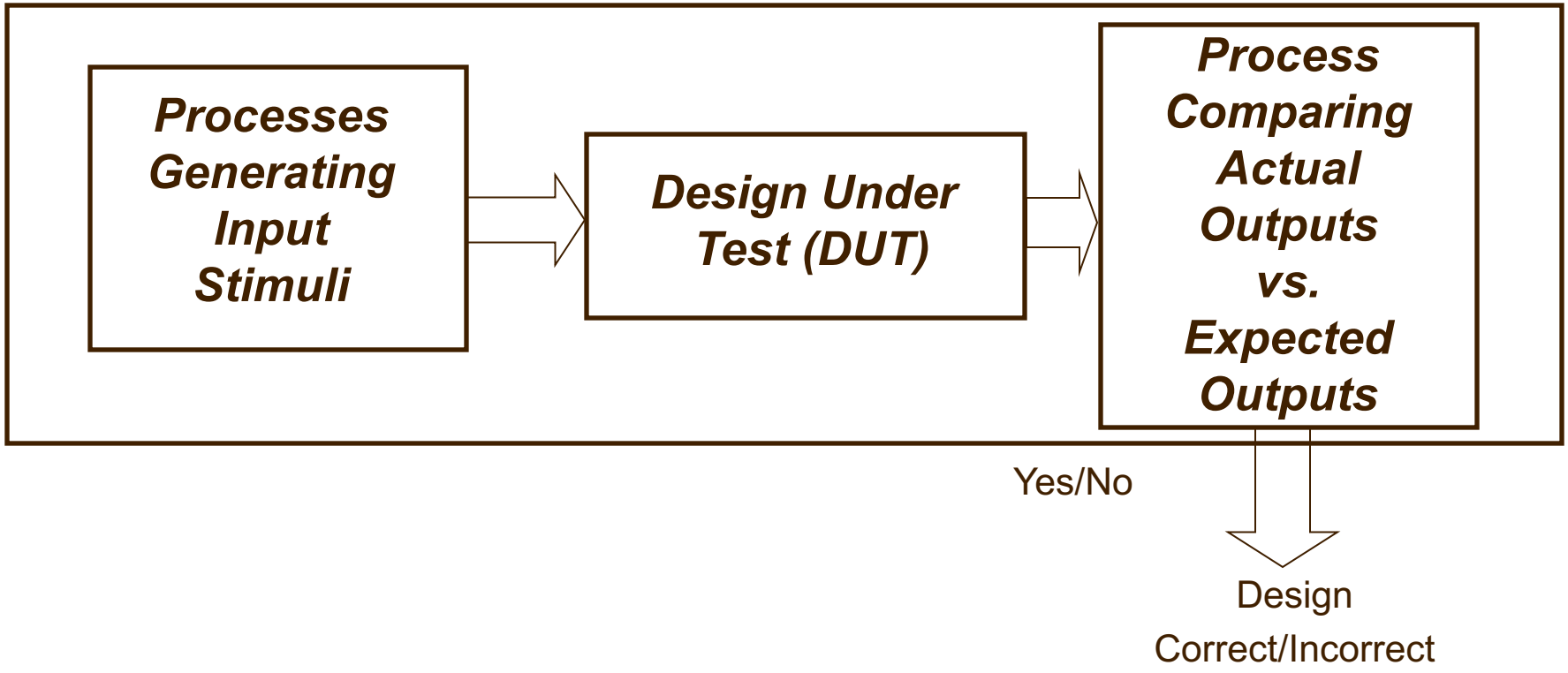
Testbench Defined

- *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.

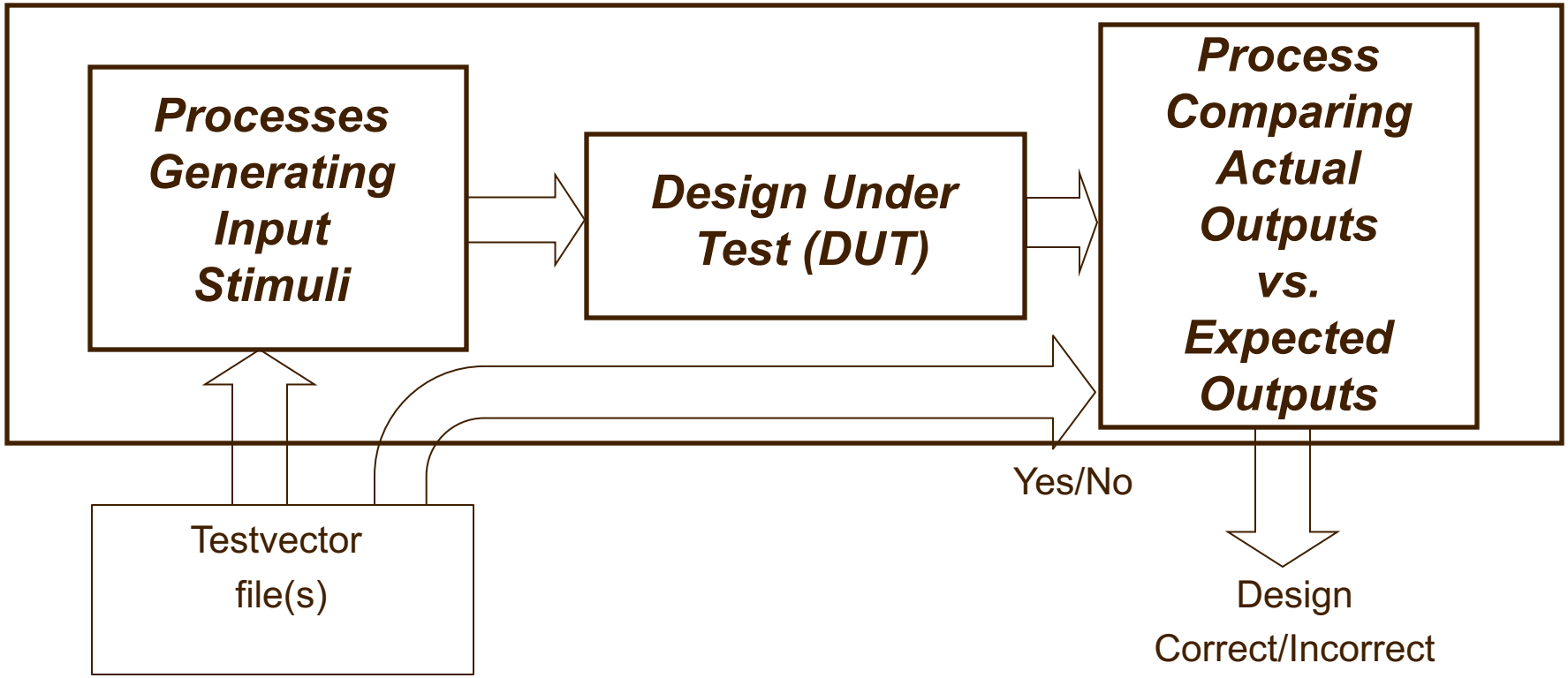
Simple Testbench



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

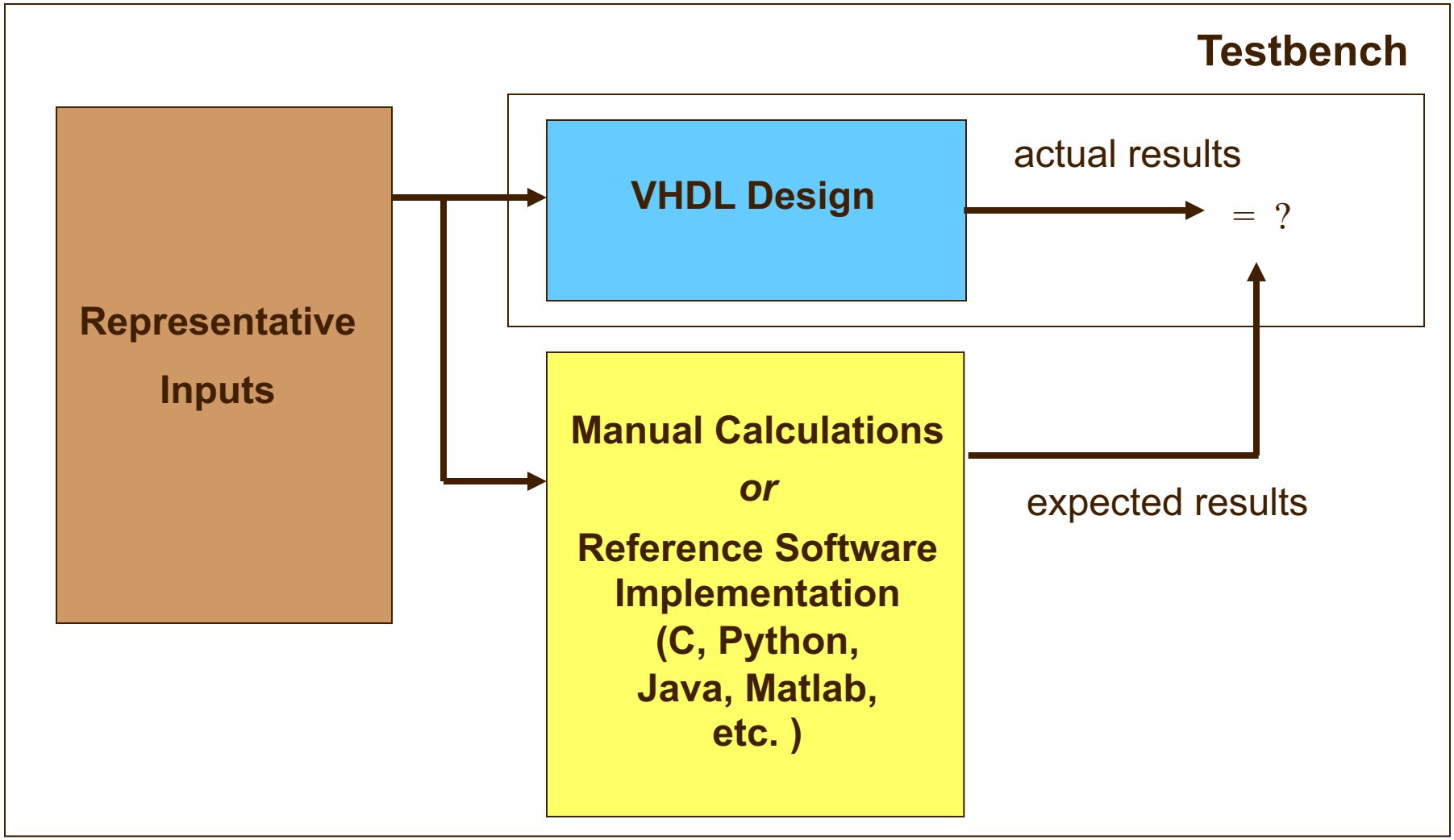
……

Input N, Expected Output N

Test vectors can be:

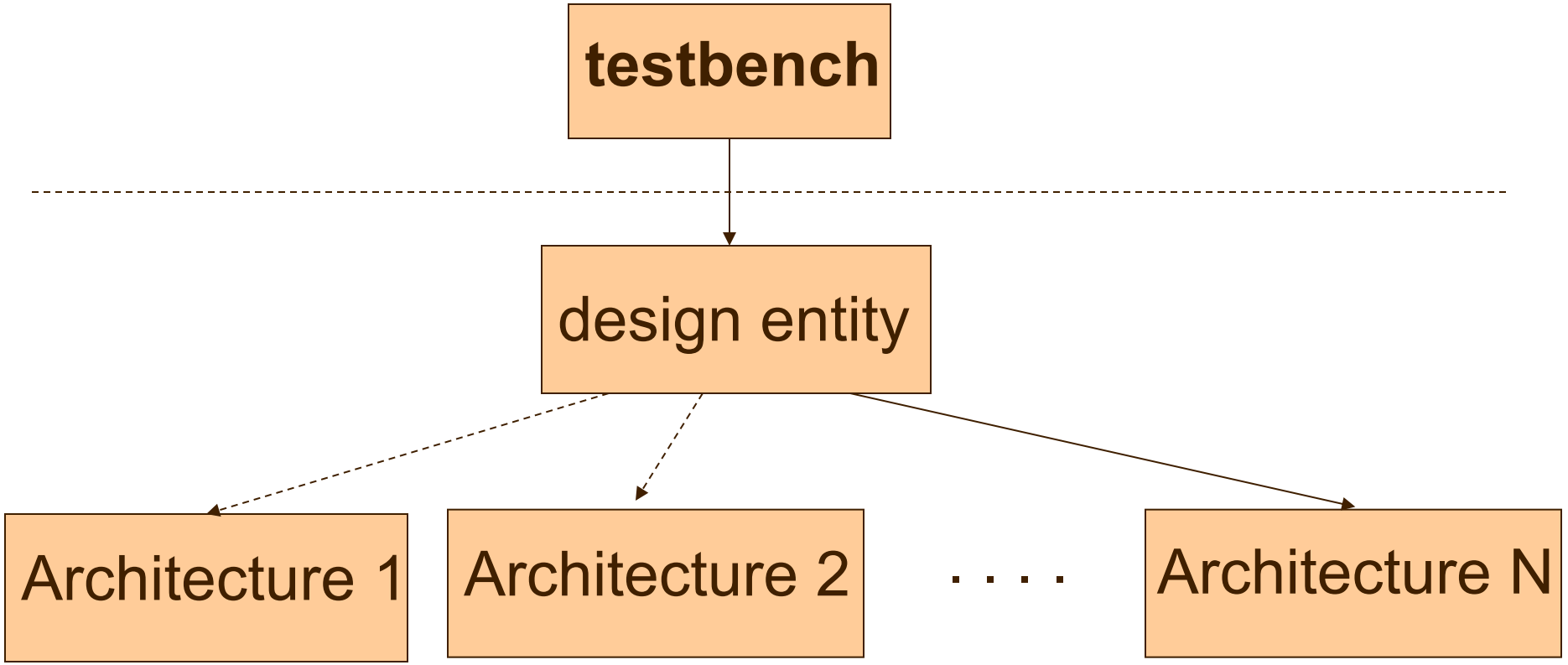
- defined in the testbench source file
- stored in a data file

Possible sources of expected results used for comparison



Testbench

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



Simple Testbench Anatomy

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

ARCHITECTURE behavioral OF TestComp_tb IS

     --Local signals and constants



-------------------------------------------------
BEGIN
  DUT: entity work.TestComp(dataflow) 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

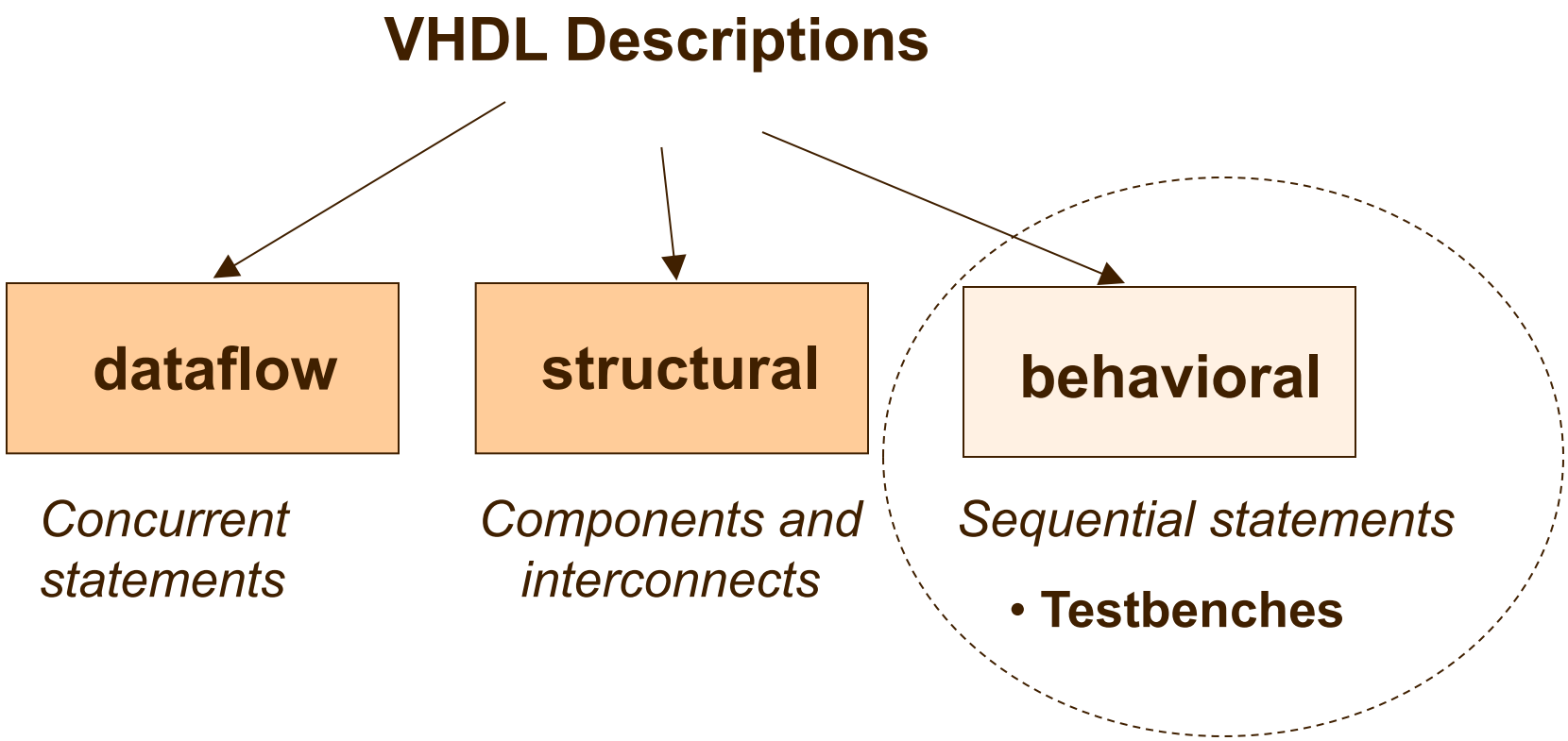
-- Stimulus signals - signals mapped to the input and inout ports of tested entity

SIGNAL test_vector: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL test_result : STD_LOGIC;
```

Testbench for XOR3 (2)

```
BEGIN
UUT : entity work.xor3(dataflow)
 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 Description Styles



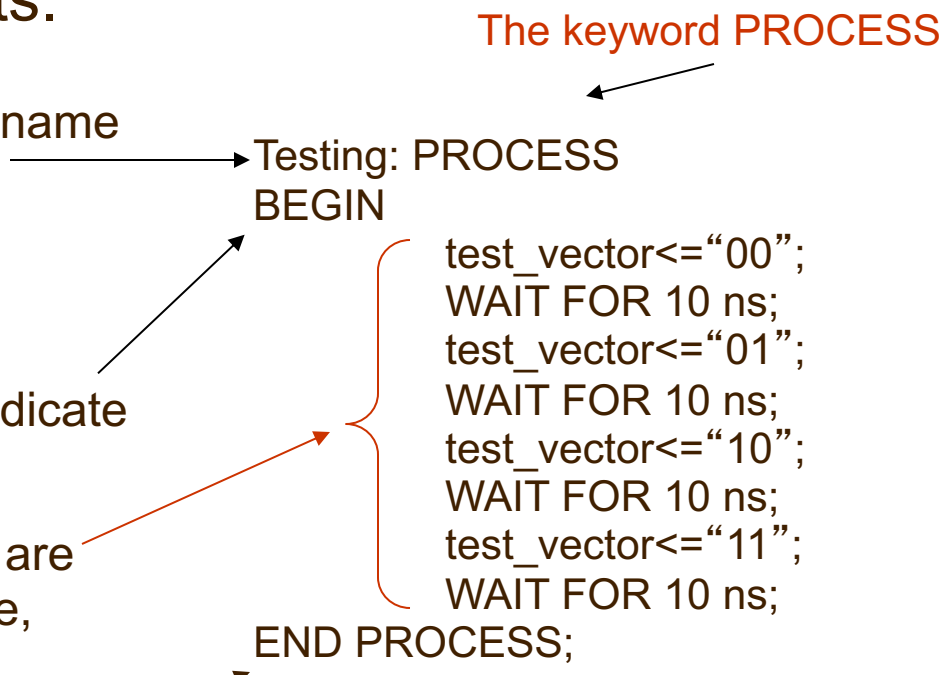


Process without Sensitivity List and its use in Testbenches

What is a PROCESS?

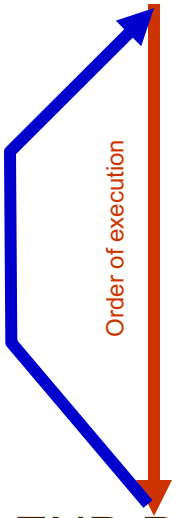
- A process is a sequence of instructions referred to as sequential statements.

- 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.

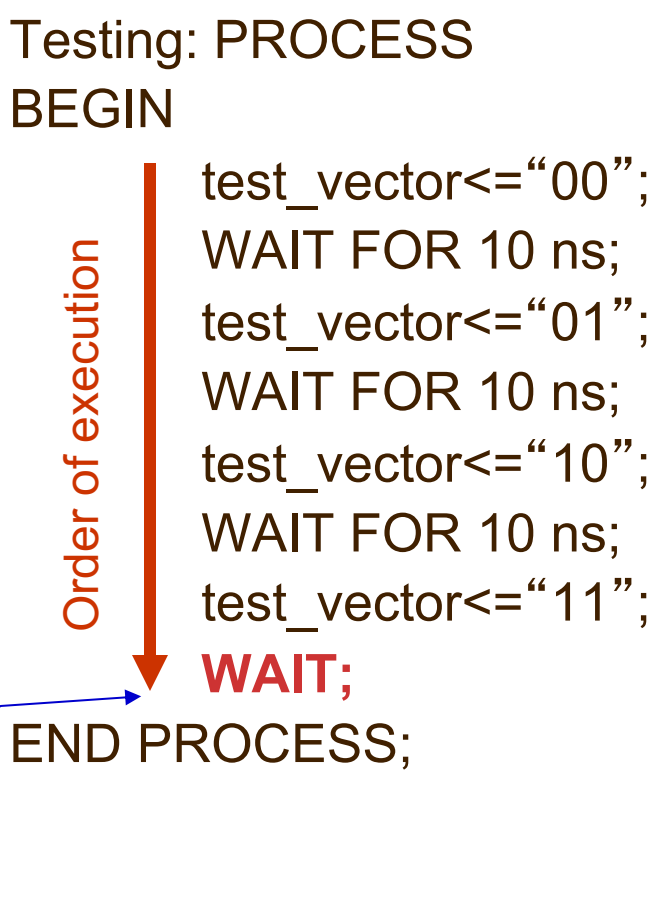



```
Testing: PROCESS
BEGIN
        test_vector<="00";
        WAIT FOR 10 ns;
        test_vector<="01";
        WAIT FOR 10 ns;
        test_vector<="10";
        WAIT FOR 10 ns;
        test_vector<="11";
        WAIT FOR 10 ns;
END PROCESS;
```

Program control is passed to the first statement after BEGIN

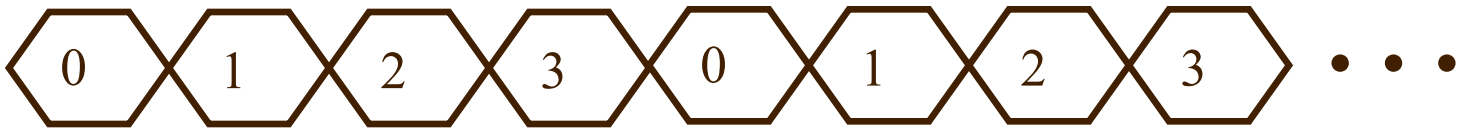
PROCESS with a WAIT Statement

- 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.



WAIT FOR vs. WAIT

WAIT FOR: waveform will keep repeating itself forever



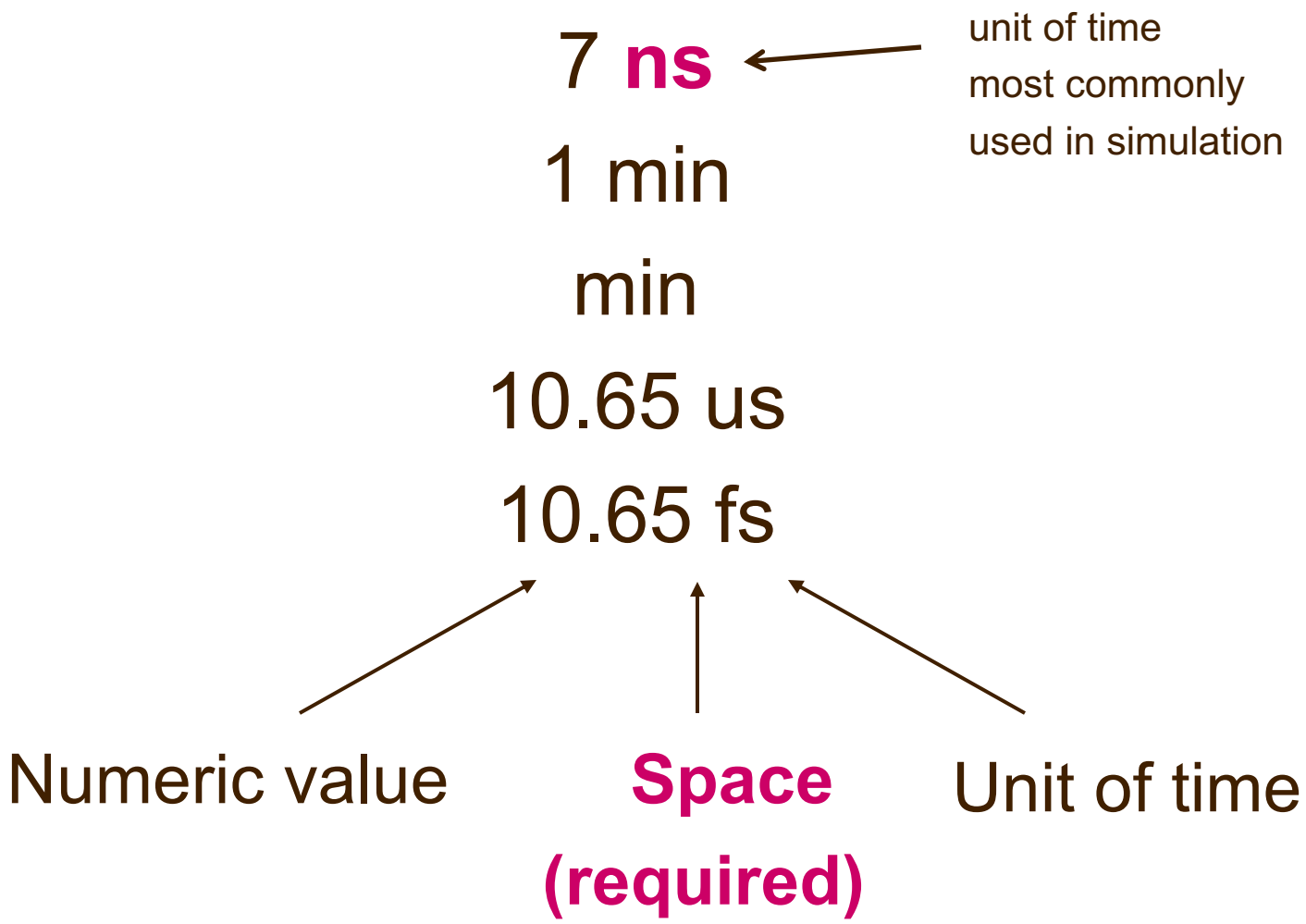
WAIT : waveform will keep its state after the last wait instruction





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

```
use IEEE.NUMERIC_STD.ALL;

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

BEGIN
    .......

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

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

Different ways of performing the same operation

signal count: unsigned(7 downto 0);

You can use:

count <= count + "00000001";

or

count <= count + 1;

Overloaded operators in IEEE numeric_std package

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

Different declarations for the same operator

Declarations in the package numeric_std:

function “+” (L: unsigned; R: unsigned)
return unsigned;

function “+” (L: unsigned; R: natural)
return unsigned;

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 <= std_logic_vector(unsigned(test_ab) + 1);
                            end loop;
                            test_sel <= std_logic_vector(unsigned(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;
    CONSTANT reset3_width : TIME := 200 ns;
    SIGNAL reset1 : STD_LOGIC;
    SIGNAL reset2 : STD_LOGIC := '1';
    SIGNAL reset3 : STD_LOGIC;

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;
    reset3 <= '1', '0' after reset3_width;
END behavioral;
```

Typical error

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

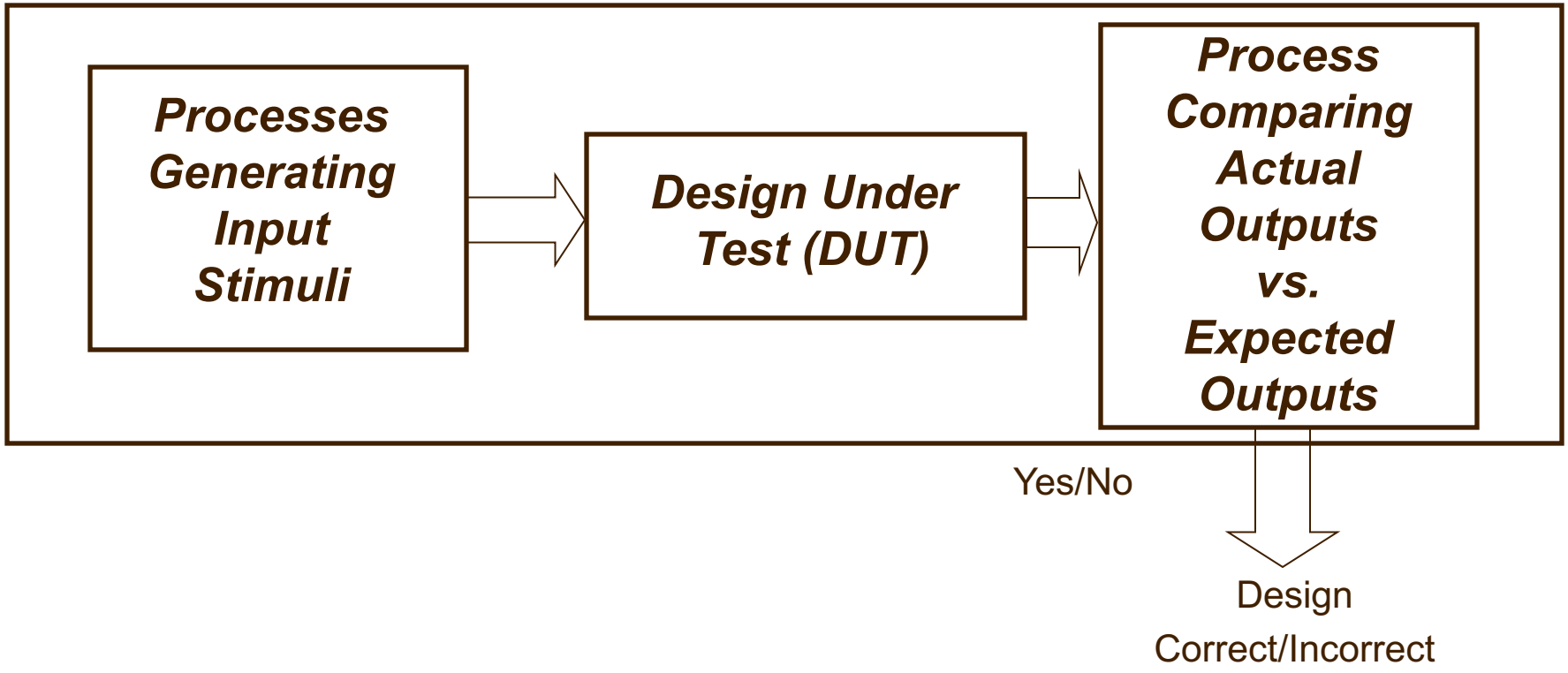
BEGIN
    .....
    generator1: PROCESS
            reset <= '1';
            WAIT FOR 100 ns
            reset <= '0';
            test_vector <="000";
            WAIT;
    END PROCESS;

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

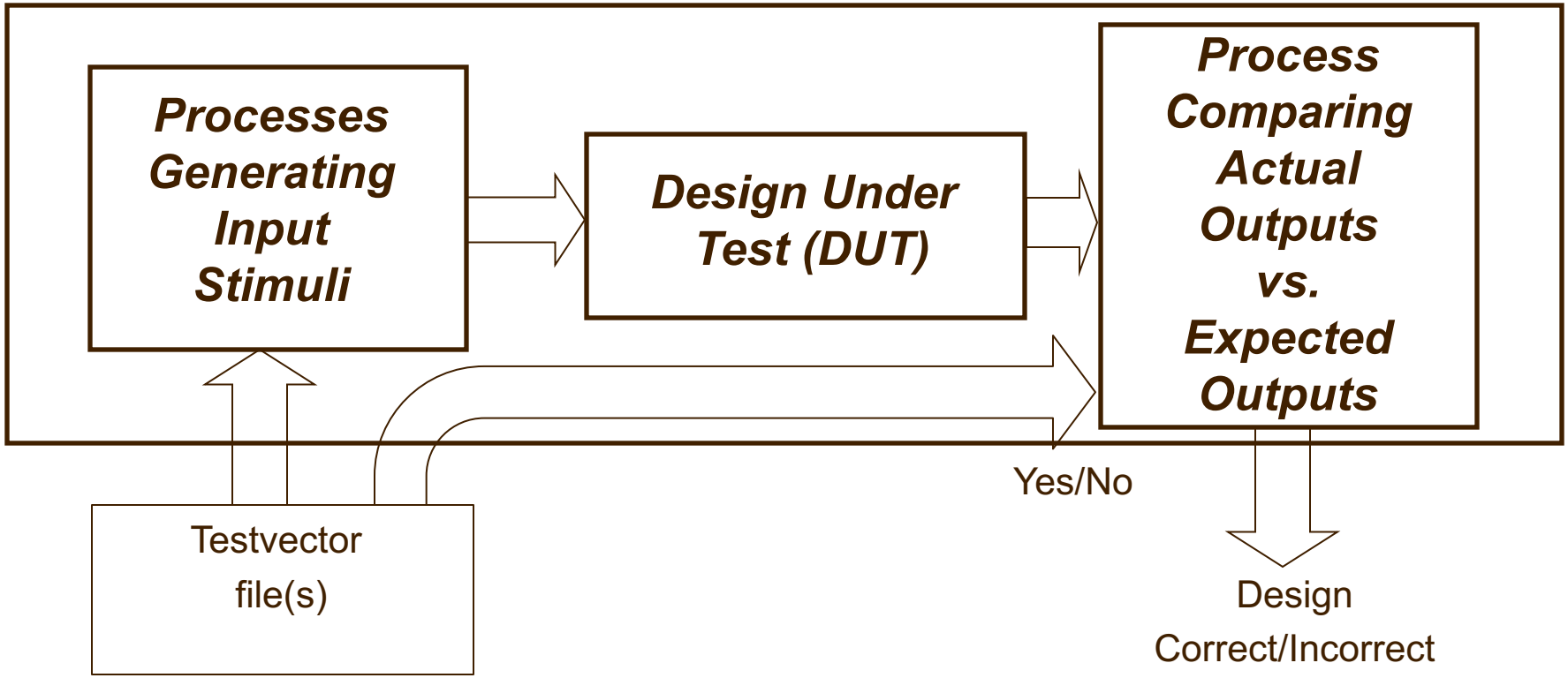


Advanced Testbenches

Advanced Testbench (1)



Advanced Testbench (2)





Asserts & Reports

Assert

Assert is a **non-synthesizable** statement which 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 (1)

```
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;
```

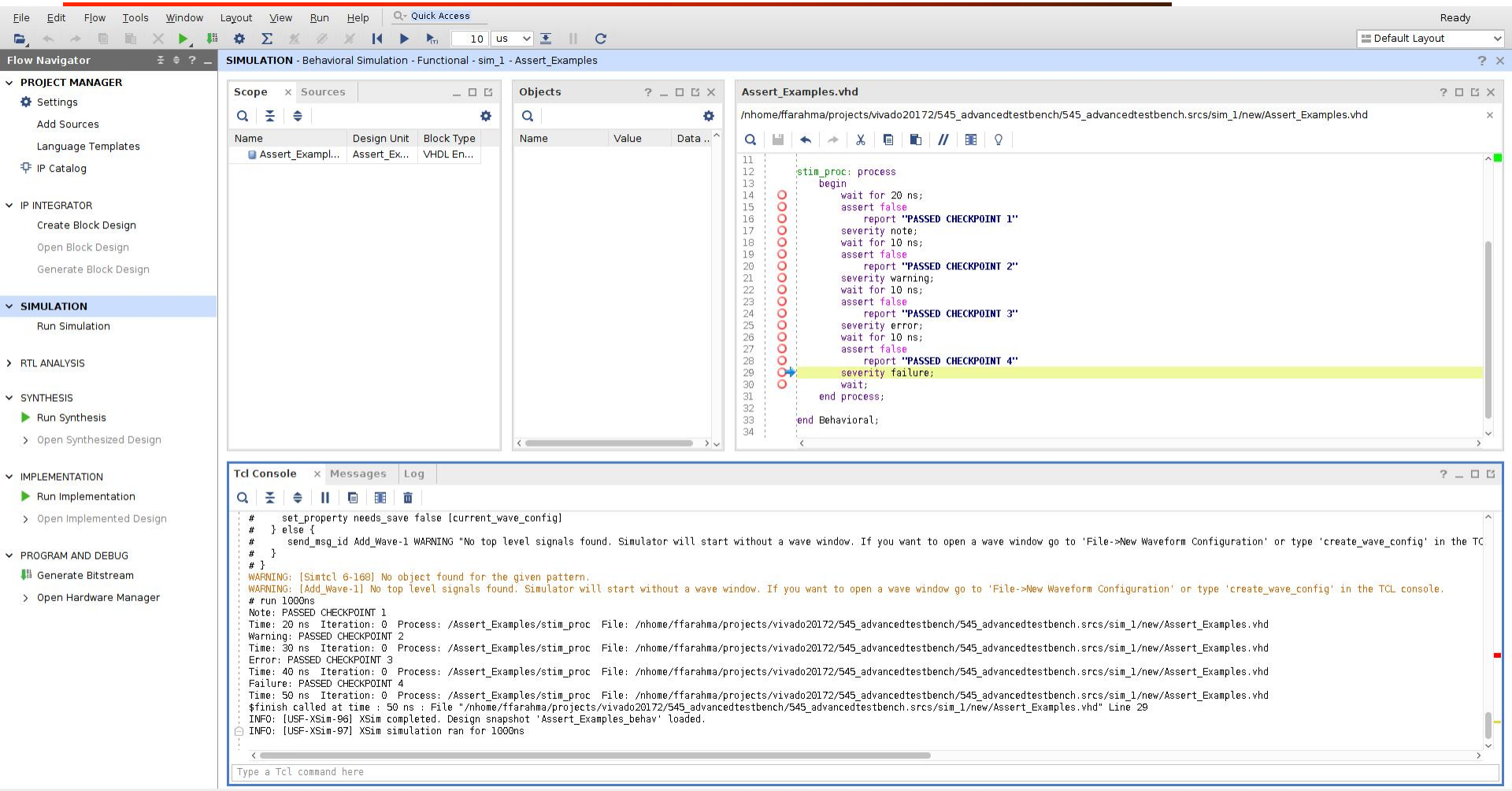
Assert – Examples (2)

```
stim_proc: process
  begin
        wait for 20 ns
        assert false
           report "PASSED CHECKPOINT 1"
           severity note;
        wait for 10 ns;
        assert false
           report "PASSED CHECKPOINT 2"
           severity warning;
```

Assert – Examples (3)

```
        wait for 10 ns;
        assert false
          report "PASSED CHECKPOINT 3"
          severity error;
        wait for 10 ns;
        assert false
          report "PASSED CHECKPOINT 4"
        severity failure;
        wait;
end process;
```

Format of messages in Vivado Simulator



Report - syntax

REPORT "message"
  [SEVERITY severity_level];

The message is always written.

Severity_level can be:

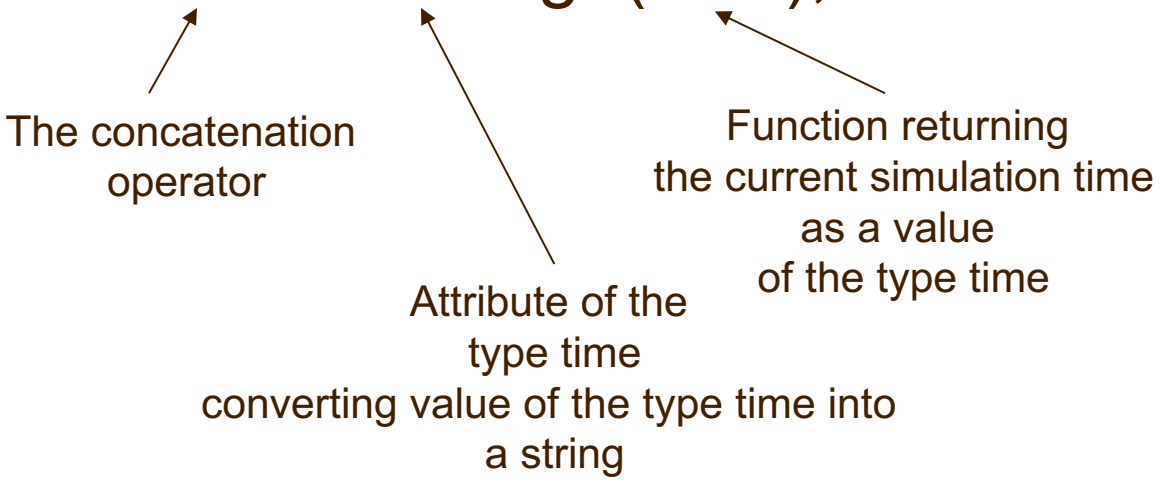
Note (default), Warning, Error, or Failure.

Report - Examples

```
report "Initialization complete";

report  "Incorrect branch"  severity error;

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



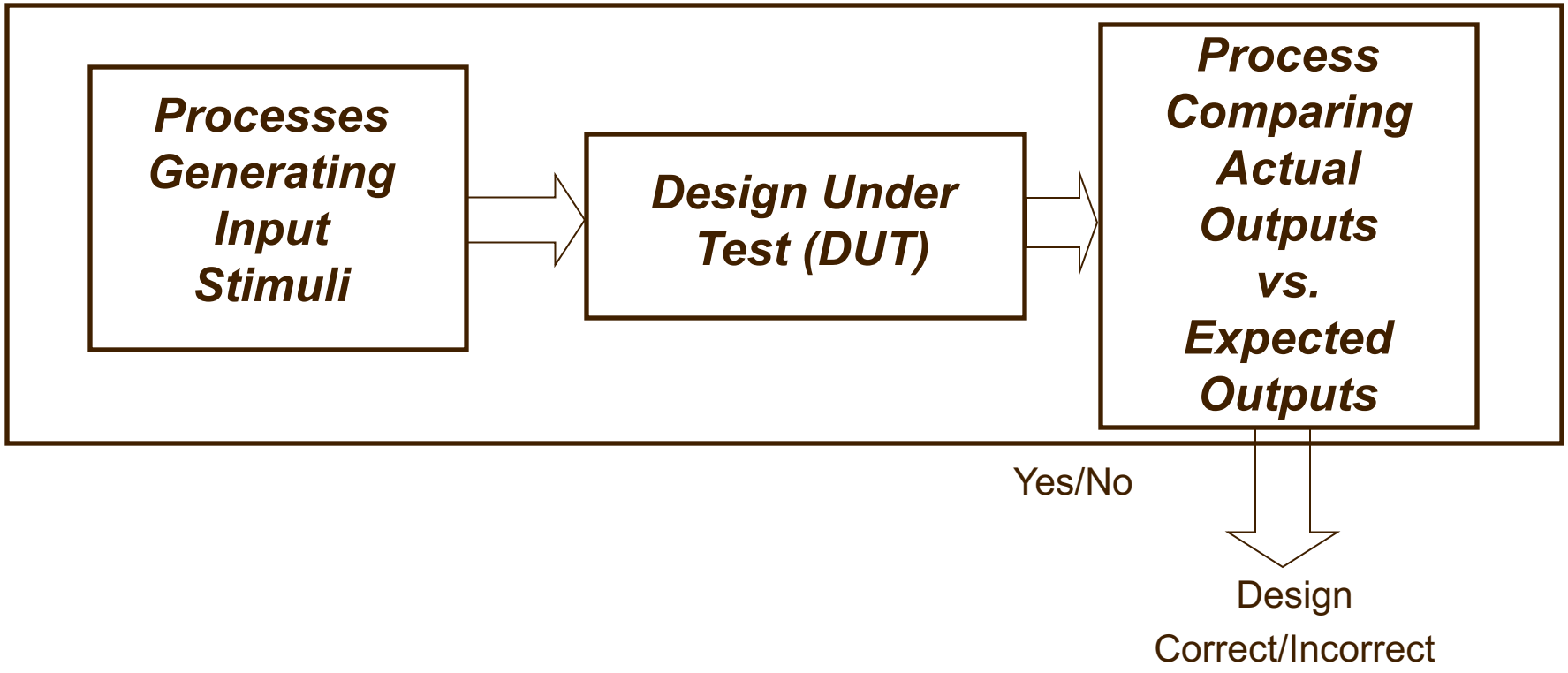
Report - Examples

```
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;
```

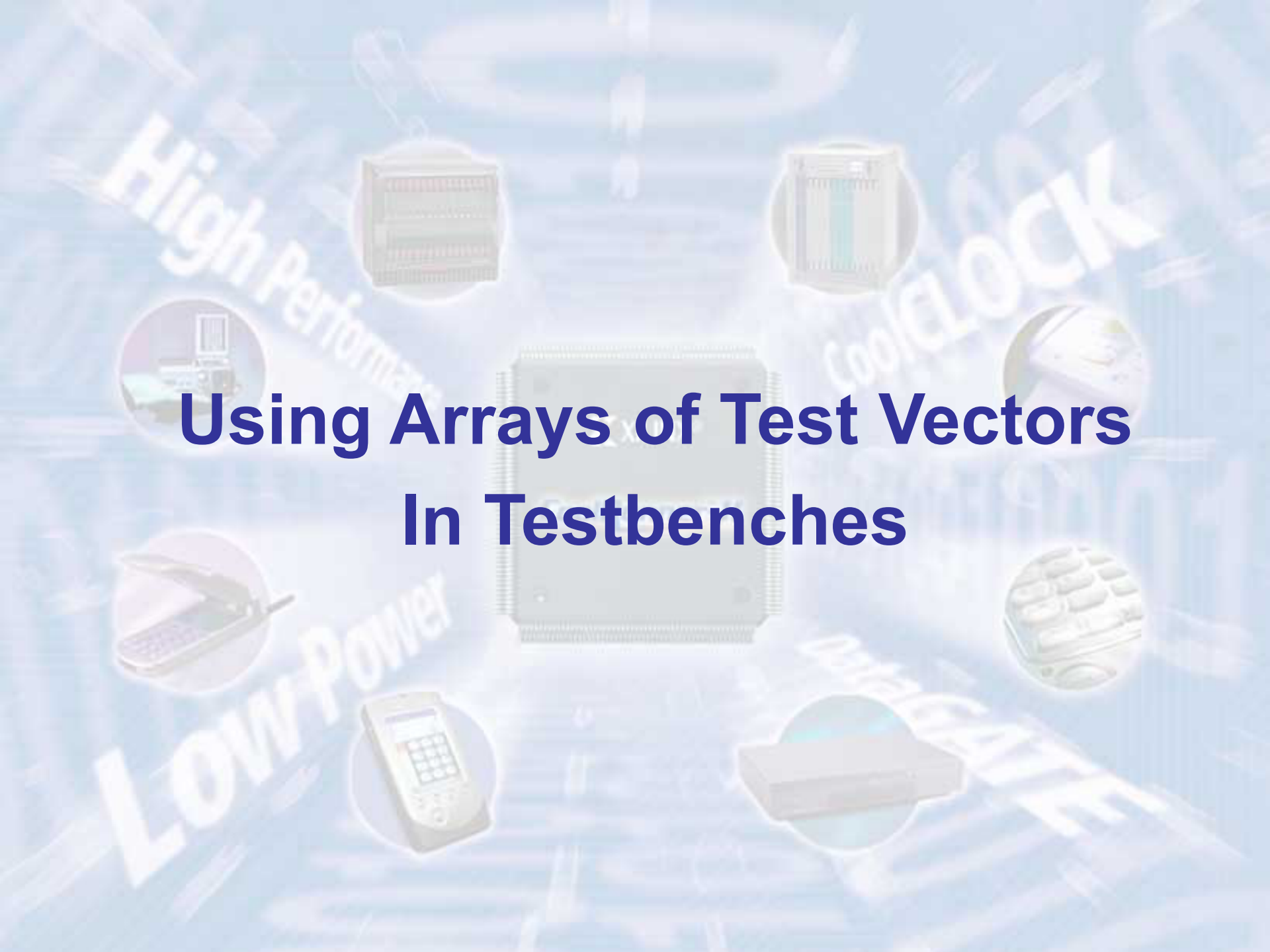
Variables - Example

In the declarative portion of the architecture:

```
SIGNAL test_operation : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL test_a : STD_LOGIC;
SIGNAL test_b : STD_LOGIC;
SIGNAL test_y : STD_LOGIC;
```

Between BEGIN and END

```
  DUT: entity work.ALU(dataflow)
            PORT MAP (operation => test_operation,
                      a => test_a,
                      b => test_b,
                      y  => test_y);
```



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

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

Testbench (2)

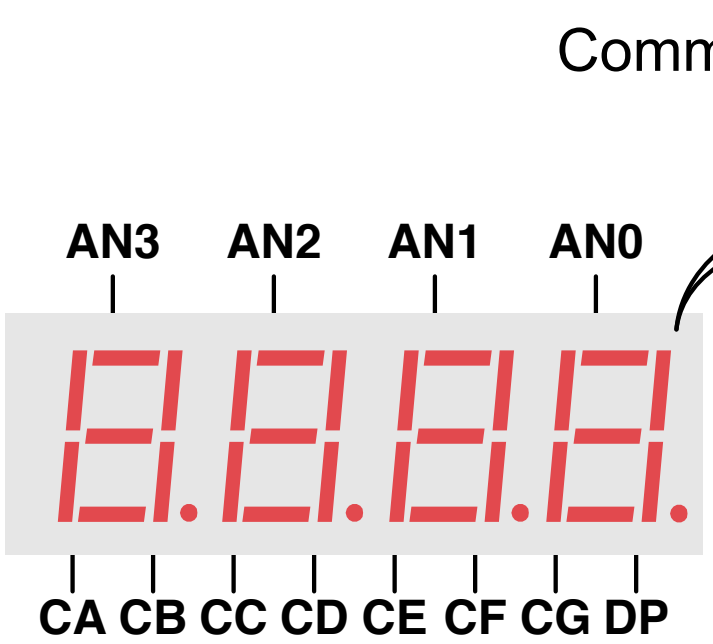
```
TYPE vector IS RECORD
  bcdStimulus:        STD_LOGIC_VECTOR(3 DOWNTO 0);
  sevSegOut:          STD_LOGIC_VECTOR(6 DOWNTO 0);
END RECORD;

CONSTANT NumVectors: INTEGER:= 10;

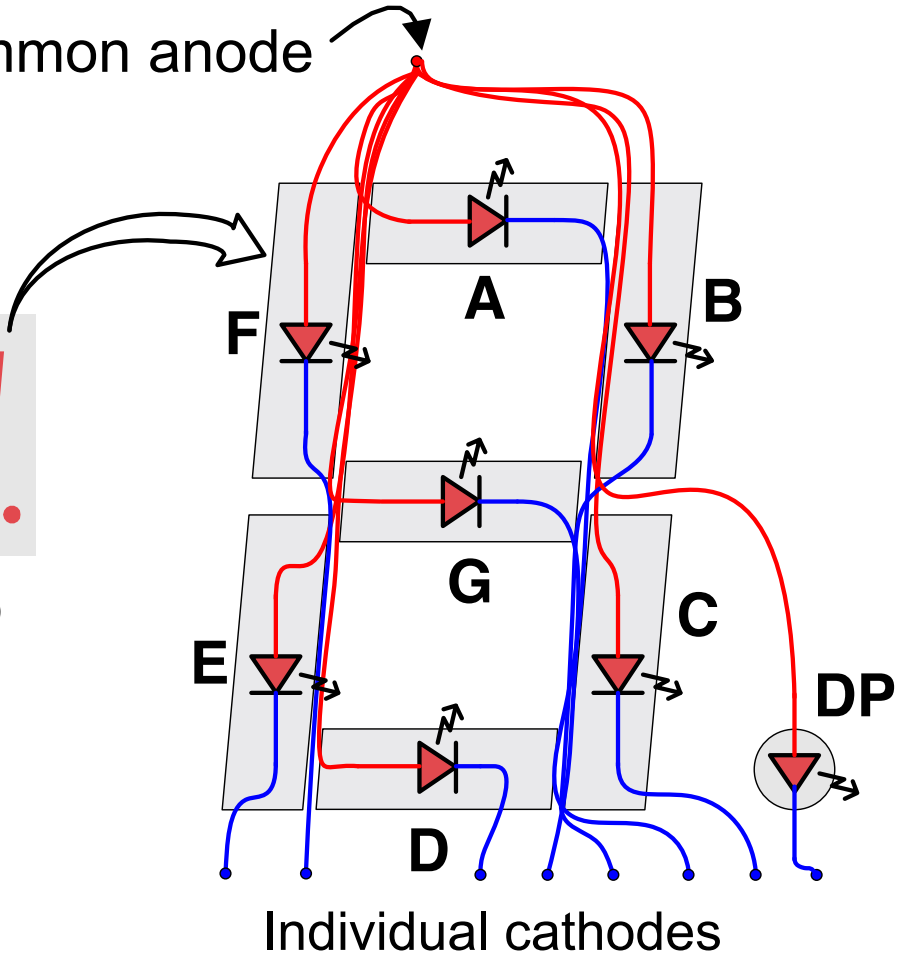
TYPE vectorArray is ARRAY (0 TO NumVectors - 1) OF vector;

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")
);
```

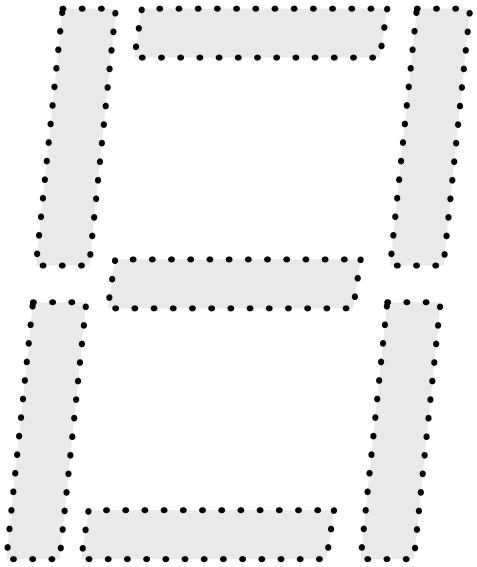
Seven Segment Displays

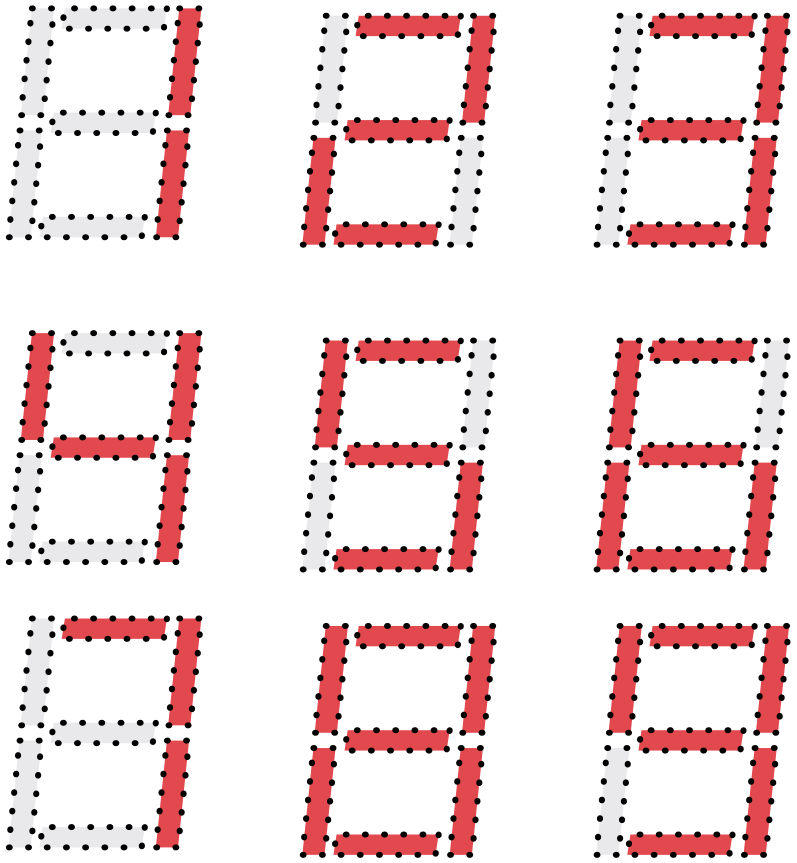


Four-digit Seven Segment Display



Patterns for Decimal Digits





Testbench (3)

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

BEGIN

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

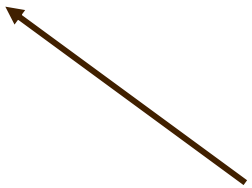
Testbench (4)

```
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!

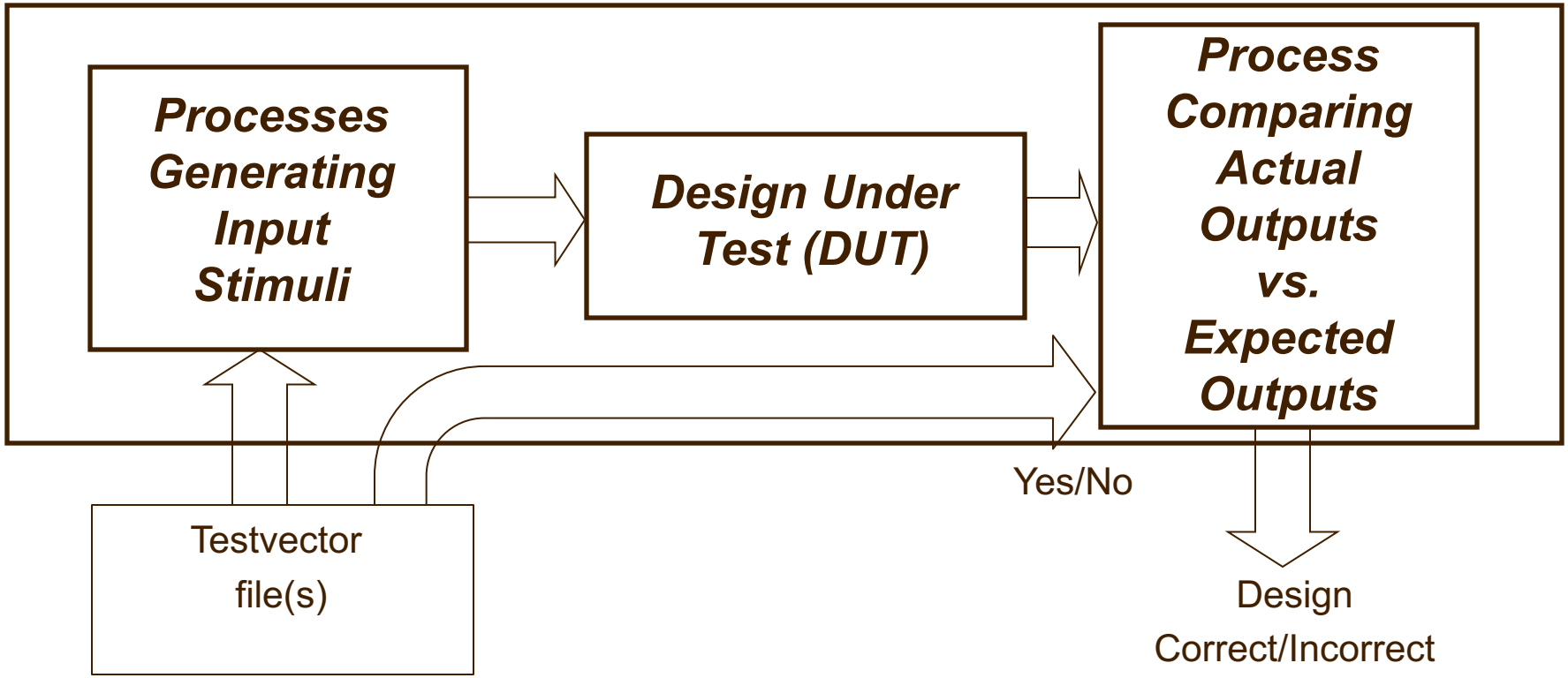
Testbench (5)

```
    WAIT;

    END PROCESS;

END testbench;
```

Advanced Testbench (2)




High Performance
Cool Clock
Low Power
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.numeric_std.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: unsigned (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 <= std_logic_vector(cnt);
END rtl;
```

Test vector file (1)

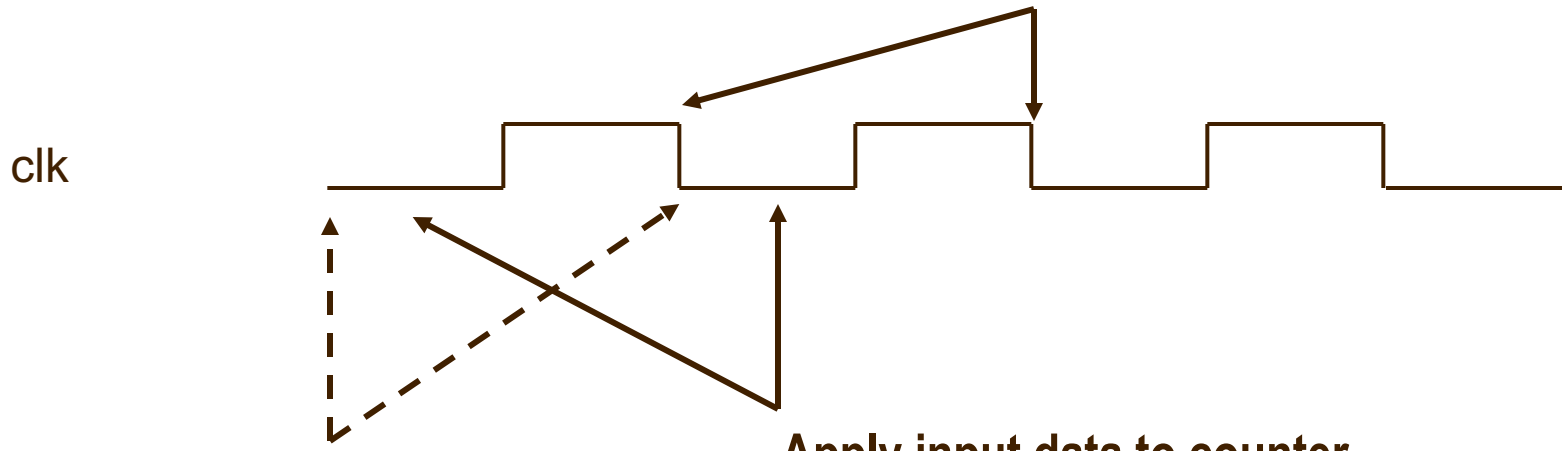
```
#Format is Rst, Load, Data, Q
#load the counter to all 1s
0 1 11111111 11111111
#reset the counter
1 0 10101010 00000000
#now perform load/increment for each bit
0 1 11111110 11111110
0 0 11111110 11111111
#
0 1 11111101 11111101
0 0 11111101 11111110
#
0 1 11111011 11111011
0 0 11111011 11111100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



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

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

Testbench (1)

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

LIBRARY std;
USE std.textio.all;

ENTITY loadCnt_tb IS
END loadCnt_tb;

ARCHITECTURE testbench OF loadCnt_tb IS
```

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: entity work.loadCnt(rtl) 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

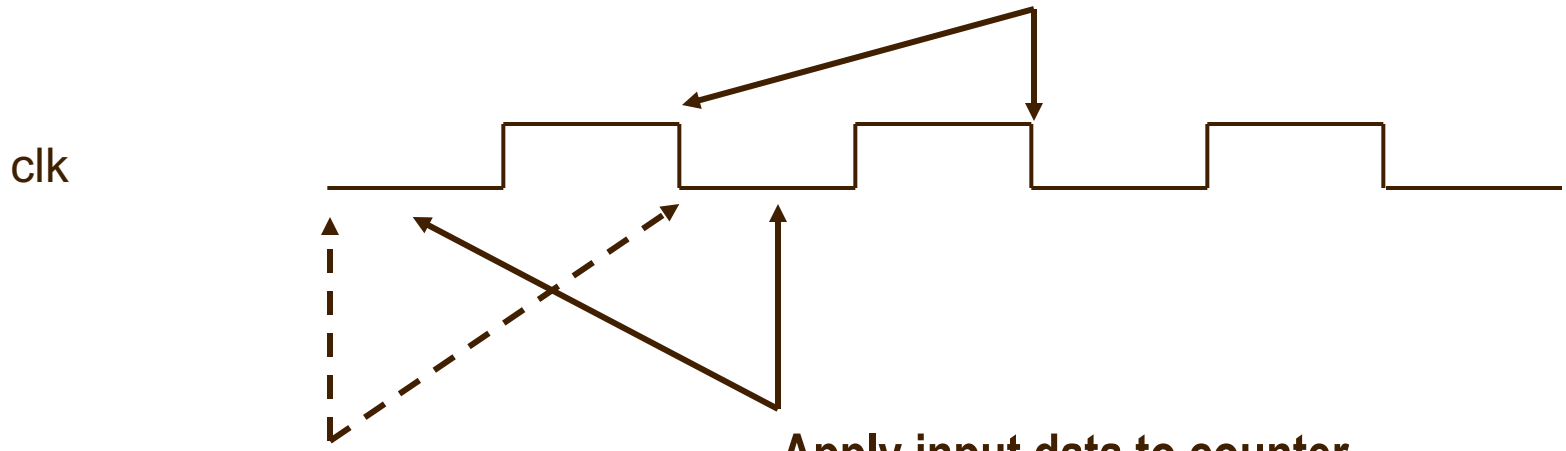
     read(VectorLine, vRst, good => VectorValid);
     NEXT WHEN NOT VectorValid;
     read(VectorLine, space);
     read(VectorLine, vLoad);
     read(VectorLine, space);
     read(VectorLine, vData);
     read(VectorLine, space);
     read(VectorLine, vQ);

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

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

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)



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

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

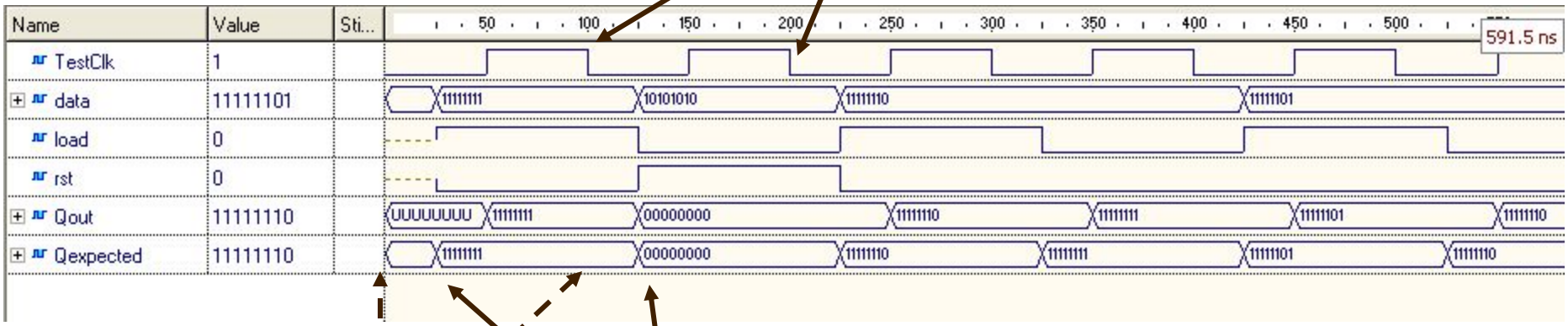
Testbench (6)

```
    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

Test vector file (1)

```
#Format is Rst, Load, Data, Q
#load the counter to all 1s
0 1 FF FF
#reset the counter
1 0 AA 00
#now perform load/increment for each bit
0 1 FE FE
0 0 FE FF
#
0 1 FD FD
0 0 FD FE
#
0 1 FB FB
0 0 FB FC
#
0 1 F7 F7
0 0 F7 F8
```

Test vector file (2)

```
#
0 1 EF  EF
0 0 EF  F0
#
0 1 DF  DF
0 0 DF  E0
#
0 1 BF  BF
0 0 BF  C0
#
0 1 7F  7F
0 0 7F  80
#
#check roll-over case
0 1 FF  FF
0 0 FF  00
#
# End vectors
```

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);
     hread(VectorLine, vData);
     read(VectorLine, space);
     hread(VectorLine, vQ);

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

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

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.)

?

