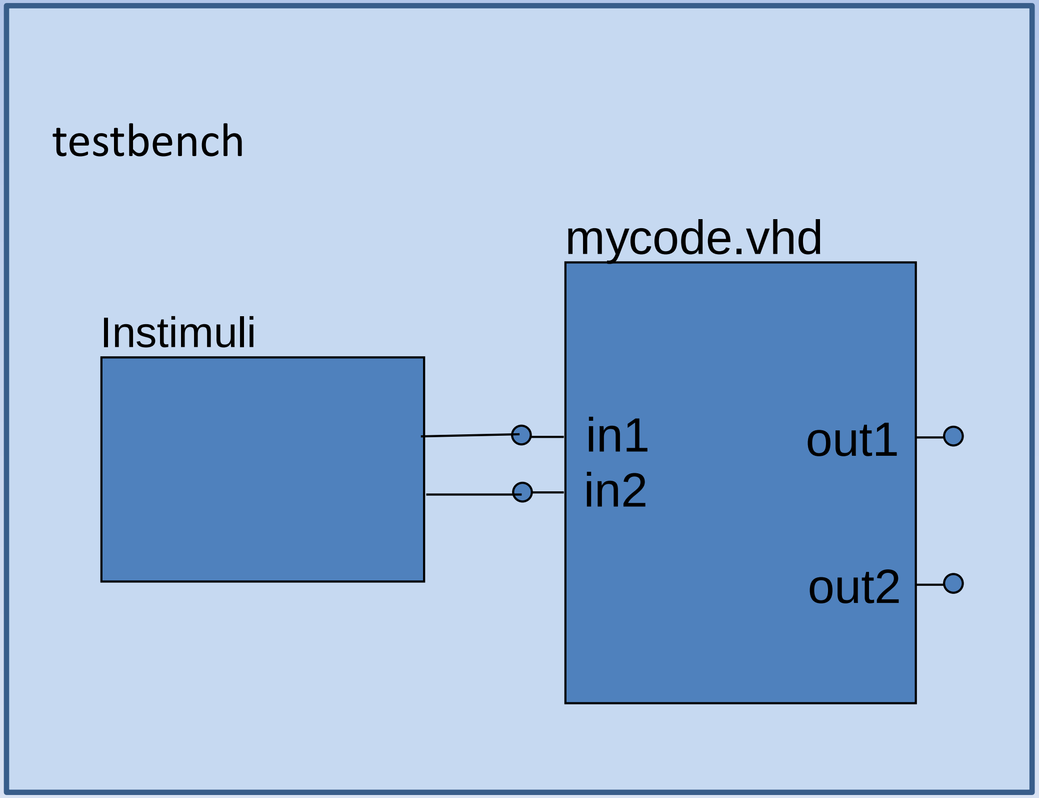
VHDL programmering för inbyggda system

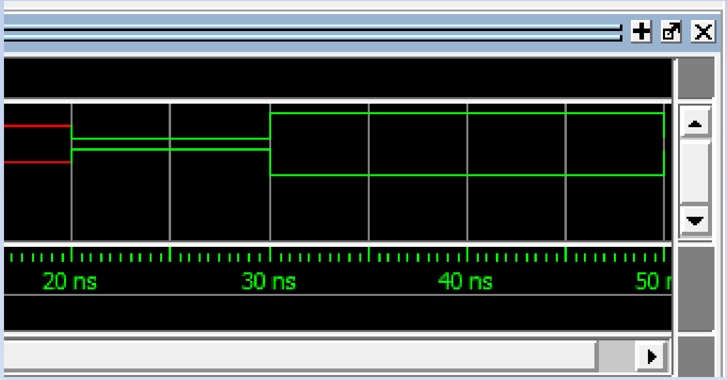
Välkommen

- Purpose of testbench
- Three classes of traditional testbenches
- General testbench methods
- Self verification methods
- Arrays for stimulus & results
- Assert Statements

General Testbench Methods

- Create in-stimulus signals and check the waveform
- Verifification of mycode.vhd





Sample Class I Testbench

```
ENTITY gatetest IS
END gatetest;

ARCHITECTURE stimulus OF gatetest IS

    COMPONENT simplegate
        PORT  (A,B: IN std_logic;
                Y: OUT std_logic);
    END COMPONENT;

    SIGNAL  A,B,Y: std_logic;

BEGIN

    NAND1: simplegate PORT MAP(A => A,B => B,Y => Y);

   PROCESS
      CONSTANT period : TIME := 40 ns;
   BEGIN
      A <= '1';
      B <= '1';
      WAIT FOR period;
      A <= '1';
      B <= '0';
      WAIT FOR period;
      A <= '0';
      B <= '1';
      WAIT FOR period;
      A <= '0';
      B <= '0';
      WAIT FOR period;
      WAIT;
   END PROCESS;
END stimulus;
```

Top-Level Entity Has No Ports

Declare Lower-Level Design Entity

Signals to Assign Values & Observe

Instantiate Lower-Level Entity

Process to Apply Stimulus

Final WAIT Keeps Process from Repeating Indefinitely

Purpose of Testbench

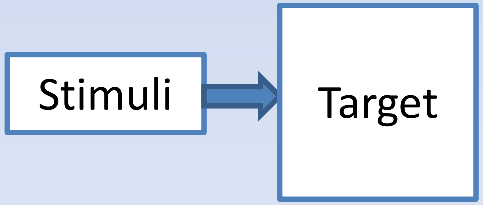
- Generate stimulus to test design
- Possible to automatically verify design to spec and log all errors
  - Regression tests
    - to ensure that a change, such as a bugfix, did not introduce new faults.
    - common methods include rerunning previously run tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged.
- Simulation = Real prototype

Three Classes of Traditional Test benches

I. Test bench applies stimulus to target code and outputs are manually reviewed

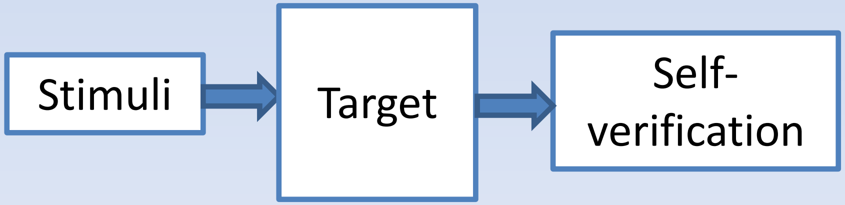
II. Test bench applies stimulus to target code and **verifies outputs** functionally

III. Test bench applies stimulus to target code and verifies outputs **with timing**



Stimuli

Target



Stimuli

Target

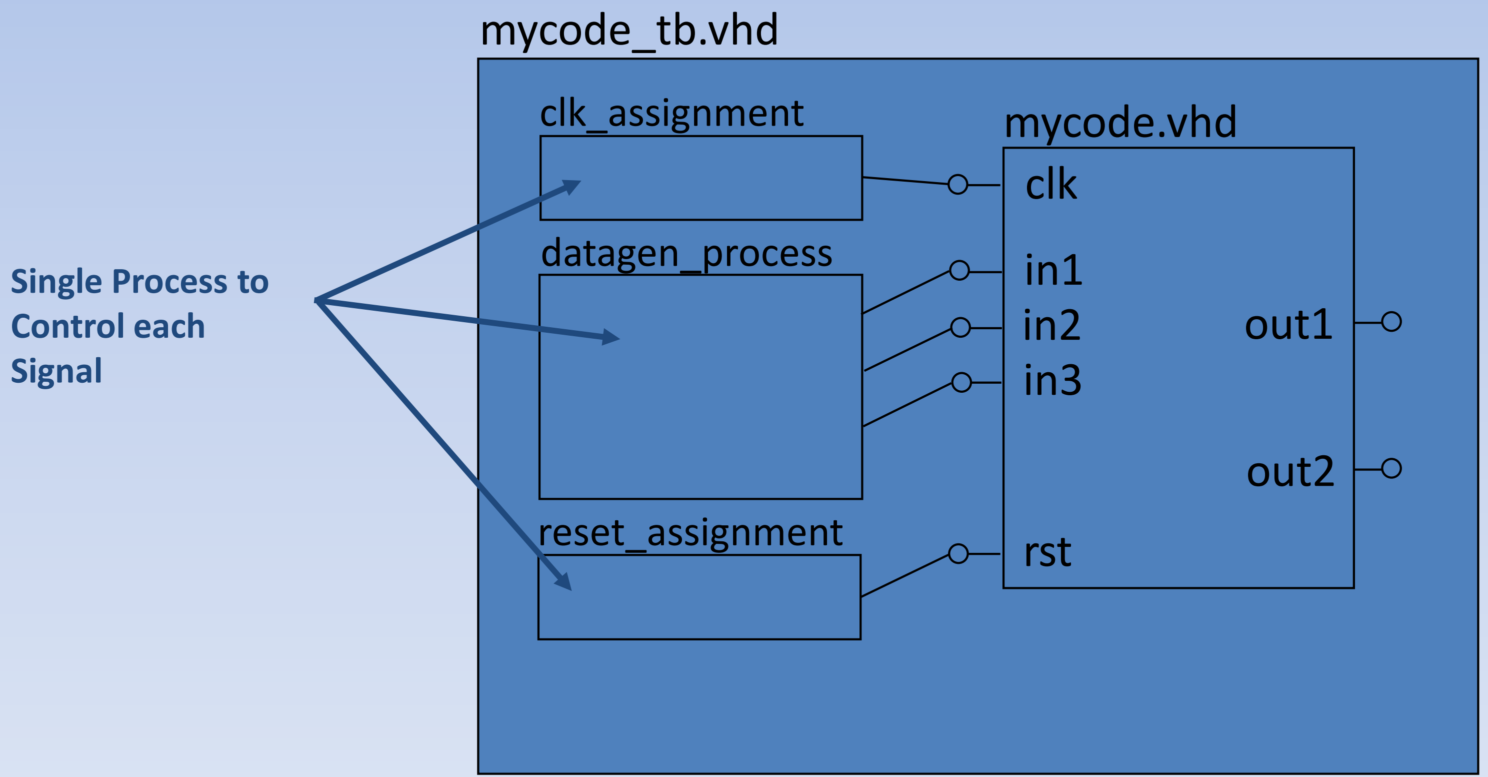
Self-verification

Advantages/Disadvantages

Testbench Type	Advantages	Disadvantages	Recommendation
Class I	• Simple to write	• Requires manual verification	• Great for verifying simple code • Not intended for re-use
Class II	• Easy to perform verification once complete • "Set and forget it"	• Takes longer to write • More difficult to debug initially	• Better for more complicated designs, designs with complicated stimulus/outputs and higher-level designs • Promotes re-usability • Regression tests
Class III	• Most in-depth • "Guarantees" design operation, if successful (subject to model accuracy)	• Takes longest to write • Most difficult to debug • Physical changes (i.e. target device, process) requires changing testbench	• Might be overkill for many FPGA designs • Required for non-Altera ASIC designs

General Testbench Methods

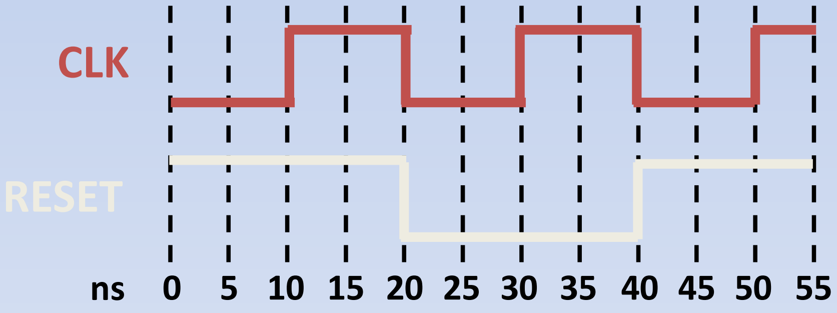
- Create stimulus signals to connect to DUT (device under test)



Concurrent Statements

- Signals with regular or limited transitions can be created with ***concurrent*** statements
- These statements can begin a testbench and reside outside any processes

```
ARCHITECTURE logic OF test_b IS

-- Use clkperiod constant to create 50 MHz clock
    CONSTANT clkperiod : TIME := 20 ns;

-- clk initialized to ‘0’
    SIGNAL clk : std_logic := ‘0’;

    SIGNAL reset : std_logic;

BEGIN

--clock must be initialized when declared to use
--      this notation
    clk <= NOT clk AFTER clkperiod/2;

    reset <= ‘1’,  ‘0’ AFTER 20 ns,  ‘1’ AFTER 40 ns;

END ARCHITECTURE logic;
```



Sequential Statements

```
clkgen: PROCESS  -- Another clock generation example
   CONSTANT clkperiod : TIME := 20 ns;
BEGIN
   clk <= ‘0’; -- Initialize clock
   WAIT FOR 500 ns;  -- Delay clock for 500 ns
   LOOP  -- Infinite loop to create free-running clock
      clk <= ‘1’;
      WAIT FOR clkperiod/2;
      clk <= ‘0’;
      WAIT FOR clkperiod/2;
   END LOOP;
END PROCESS clkgen;

buscount: PROCESS (clk) -- Generate counting pattern
BEGIN
      IF rising_edge (clk) THEN
         inbus <= count;
         count <= count + 1;
      END IF;
END PROCESS buscount;
```

- More complex combinations can be created using ***sequential*** statements (i.e. LOOP, WAIT, IF-THEN, CASE)
  - Statements dependent on clock edges

AGSTU
Arbete Genom STUdier
Utbildning

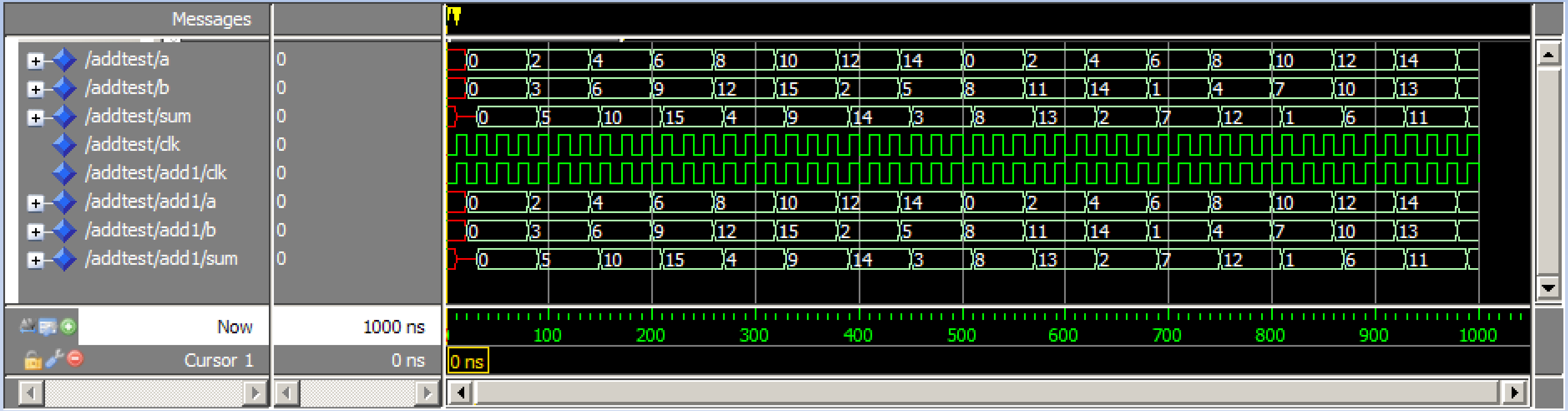
Sample VHDL Class I Testbench

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

ENTITY addtest IS  -- Top-level entity with no ports
END ENTITY addtest;

ARCHITECTURE stimulus OF addtest IS

    -- Declare design being tested
    COMPONENT adder
        PORT  (
            clk : IN std_logic;
            a, b: IN std_logic_vector(3 DOWNTO 0);
            sum : OUT std_logic_vector(3 DOWNTO 0)
            );
    END COMPONENT;

    -- Signals to assign values and observe results
    SIGNAL a, b, sum: std_logic_vector(3 DOWNTO 0);
    SIGNAL clk : std_logic := ‘0’;

    -- Constants for timing values
    CONSTANT clkperiod : TIME := 20 ns;

BEGIN

    -- Create clock to synchronize actions
    clk <= NOT clk AFTER clkperiod/2;

    -- Instantiate design being tested
    add1: adder PORT MAP (
        clk => clk, a => a, b => b, sum => sum);
```

```
    -- Process to generate stimulus; Note operations
    --    take place on inactive clock edge
    PROCESS
        CONSTANT period : TIME := 40 ns;
        VARIABLE ina, inb : std_logic_vector(3 DOWNTO 0);
    BEGIN
        WAIT UNTIL falling_edge (clk);
        ina := (OTHERS => ‘0’);
        inb := (OTHERS => ‘0’);

        stim_loop:  LOOP
            -- Apply generated stimulus to inputs
            a <= ina;
            b <= inb;
            WAIT FOR period;

            -- Exit loop once simulation reaches 1 us
            EXIT stim_loop WHEN NOW > 1 us ;

            -- Use equations below to generate new stimulus
            --      values
            WAIT UNTIL falling_edge (clk);
            ina := ina + 2;
            inb := inb + 3;
        END LOOP stim_loop;

        -- Final wait to keep process from repeating
        WAIT;
    END PROCESS;
END ARCHITECTURE stimulus;
```

AGSTU
Arbete Genom STUdier
Utbildning

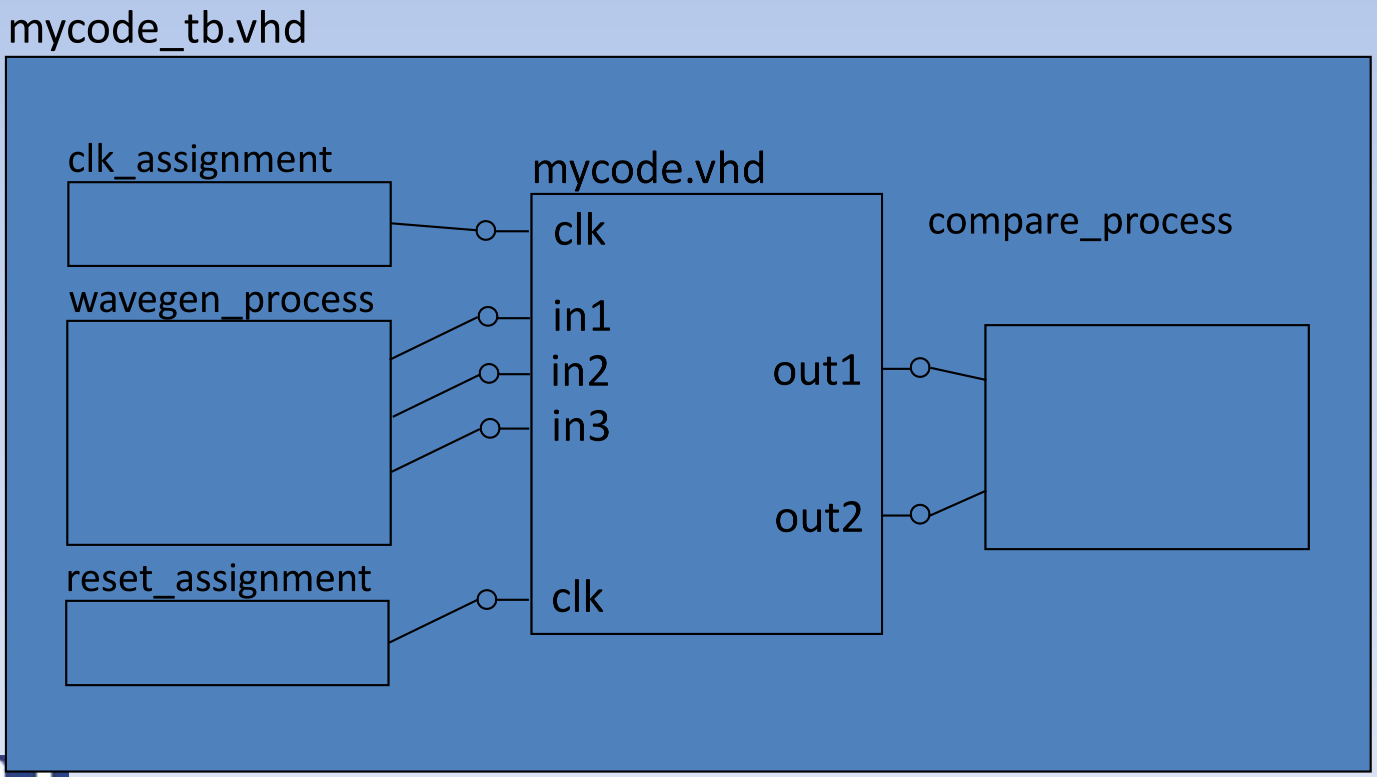
Example Results



Messages	
/addtest/a	0
/addtest/b	0
/addtest/sum	0
/addtest/clk	0
/addtest/add1/clk	0
/addtest/add1/a	0
/addtest/add1/b	0
/addtest/add1/sum	0
Now	1000 ns
Cursor 1	0 ns

Class II (& III) Methods

- Add a compare process so that DUT outputs can be monitored
  - Allows testbench to do “self-verification”



Self Verification Methods

- Use “compare_process” or equivalent to check results generated by design against expected results
- Single simulation can use one or multiple testbench files
    - Single testbench file containing all stimulus and all expected results
    - Multiple testbench files based on stimulus, expected results or functionality (e.g. data generator, control stimulus)

Simple Self Verifying Test Benches

```
clk <= NOT clk AFTER clkperiod/2;

add1 : adder PORT MAP (
      clk => clk, a => a, a => b, sum => sum);

stim: PROCESS
      VARIABLE error : BOOLEAN := FALSE;
BEGIN
      WAIT UNTIL falling_edge(clk);
      a <= (OTHERS => ‘0’);
      b <= (OTHERS => ‘0’);
      WAIT FOR  40 ns;
      IF (sum /= 0) THEN
            error := TRUE;
      END IF;

      WAIT UNTIL falling_edge(clk);
      a <= “0010”;
      b <= “0011”;
      WAIT FOR 40 ns;
      IF (sum /= 5) THEN
            error := TRUE;
      END IF;

      -- repeat above varying values of a and b

      WAIT;
END PROCESS stim;
```

- **Code repeated for each test case**
- **Result checked**

- *Simple self verifying test bench*
- *Each sub-block within process assigns values to a,b and waits to compare sum to its predetermined result*
- *Code not very efficient*
  - *Each test case may require a lot of repeated code*
- *Improve this code by introducing a procedure*

AGSTU
Arbete Genom STUdier
Utbildning

Simplifying Test Bench with Procedure

```
    PROCEDURE test (
        SIGNAL clk : IN std_logic;
        inval_a, inval_b, result : IN INTEGER RANGE 0 TO 15;
        SIGNAL in_a, in_b : OUT std_logic_vector(3 DOWNTO 0);
        SIGNAL sum_out :  IN std_logic_vector(3 DOWNTO 0);
        SIGNAL error : INOUT BOOLEAN) IS
    BEGIN
        WAIT UNTIL falling_edge(clk);
        in_a <= conv_std_logic_vector(inval_a,4);
        in_b <= conv_std_logic_vector(inval_b,4);
        WAIT FOR 40 ns;
        IF sum_out /= result THEN
            error <= TRUE;
        ELSE
            error <= FALSE;
        END IF;
    END PROCEDURE;

BEGIN – architecture begin

    clk <= NOT clk AFTER clkperiod/2;

    add1 : adder PORT MAP (clk => clk, a => a, a => b, sum => sum);

    PROCESS
    BEGIN
        test(clk, 0, 0, 0, a, b, sum, error);
        test(clk, 2, 3, 5, a, b, sum, error);
        test(clk, 4, 6, 10, a, b, sum, error);
        test(clk, 6, 9, 15, a, b, sum, error);
        test(clk, 8, 12, 4, a, b, sum, error);
        WAIT ;
    END PROCESS;
END ARCHITECTURE;
```

- Procedure used to simplify test bench
- Each procedure call passes in
  - clock
  - 3 integers representing input stimulus and expected result
  - ports connecting to adder
  - error flag

- *Procedure improves efficiency and readability of testbench*
- *Advantage: Easier to write*
- *Disadvantages*
  - *Each procedure call (like last example) assigns values to a, b then waits to compare sum to its predetermined result*
  - *Very difficult to do for complicated signaling*

Assert – test and report to the simulator

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 Statements

- Used to report to the simulator when a condition is **NOT** meet (false)
- Syntax

```
ASSERT <condition_expression>
       REPORT <text_string>
       SEVERITY <expression>;
```

- Report (optional)
  - Displays text in simulator window
  - Must be type string
    - Enclose character strings in “ “
    - Other data types must be converted (discussed later)
- Severity (optional)
  - Expression choices: NOTE, WARNING, ERROR, FAILURE
    - ERROR is the default
  - Results of severity depend on simulator
    - e.g. By default, ModelSim tool ends simulation on failure only

Sample Testbench Using Internal Array

```
test: PROCESS
      VARIABLE vector : test_record_type;
      VARIABLE found_error : BOOLEAN := FALSE;
BEGIN
      -- Loop through all the values in test_patterns
      FOR i IN test_patterns‘RANGE LOOP
            ….

            -- apply the stimulus on a falling edge clock
            WAIT UNTIL falling_edge(testclk);
            ……

            -- check result on next falling edge of c
            WAIT UNTIL falling_edge(testclk);
            IF (sum  /= vector.sum) THEN
                  …….
                  found_error := TRUE;
            END IF;

      END LOOP;

      ASSERT NOT found_error -- if false
            REPORT "---VECTORS FAILED---"
            SEVERITY FAILURE;
      ASSERT found_error -- if false
            REPORT "---VECTORS PASSED---"
            SEVERITY FAILURE;

END PROCESS;
END ARCHITECTURE;
```

```
** Note: 72 ns : Calc = 0100, Exp= 1001
   Time: 72 ns Iteration: 0 Instance: /record_add_tb
** Failure: ---VECTORS FAILED---
   Time:  288 ns Iteration: 0 Process:
/record_add_tb/test File: …
 Break in Process test at record_tb.vhd line 56
```

```
** Failure: ---VECTORS PASSED---
   Time:  288 ns Iteration: 0 Process: /record_add_tb/test File: …
Break in Process test at record_tb.vhd line 59
```

Assert – Examples

```
assert ( not (A and B) );
report " A and B is 1 at the same time "
severity failure;

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

assert not(In_signal = ’X’)
   report ”in_signal is not conected to entity"
   severity warning;

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

Component

Architecture

….

….

Assert

End;

Report - Examples

- report "Initialization complete";
- report "Current time = " & time'image(now);
- report "Incorrect branch" severity error;

```
Examples: 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;
```

Assert – time Examples

```
-- used in modeling flip-flops to check for timing problems
check: process
begin
    wait until clk'event and clk='1';
    assert D'stable(setup_time)
        report "Setup Time Violation"
        severity error;
    wait for hold_time;
    assert D'stable(hold_time)
        report "Hold Time Violation"
        severity error;
end process check;
```

TEXTIO/FILE Operations

- Please see the documentation

SLUT

AGSTU

Utbildning