

# Testbench, Assert, and Report

## Testbench in Combinational Circuit

To use a testbench in a combinational circuit, you need to follow these steps:

1. We must have a VHDL code that to be tested.

```
library ieee;
use ieee.std_logic_1164.all;

entity UUT is
    port (
        input_signal1 : in std_logic;
        input_signal2 : in std_logic;
        output_signal1 : out std_logic;
        output_signal2 : out std_logic
    );
end UUT;

architecture rtl of UUT is
begin
    output_signal1 <= input_signal1 and input_signal2;
    output_signal2 <= input_signal1 or input_signal2;
end rtl;
```

2. Create a testbench for the UUT.

```

library ieee;
use ieee.std_logic_1164.all;

entity testbench is
end testbench;

architecture tb_arch of testbench is
    signal input_signal1 : std_logic;
    signal input_signal2 : std_logic;
    signal output_signal1 : std_logic;
    signal output_signal2 : std_logic;

    component UUT
        port (
            input_signal1 : in std_logic;
            input_signal2 : in std_logic;
            output_signal1 : out std_logic;
            output_signal2 : out std_logic
        );
    end component;

begin
    UUT_inst : UUT
        port map (
            input_signal1 => input_signal1,
            input_signal2 => input_signal2,
            output_signal1 => output_signal1,
            output_signal2 => output_signal2
        );

    -- Apply stimulus to the UUT
    input_signal1 <= '0';
    input_signal2 <= '1';

    -- Monitor the output signals of the UUT
    process
    begin
        wait for 10 ns;
        assert output_signal1 = '0' and output_signal2 = '1'

```

```
        report "Test failed"
        severity error;
    wait;
end process;

end tb_arch;
```

## Things to note

- The testbench instantiates the UUT and connects the input and output signals.
- Entity block in testbench is empty because we are not using any ports.
- Entity block from UUT re-typed inside architecture block of testbench, entity keyword changed to component keyword.
- Signal input and output signals are declared in the architecture block of the testbench.
- Value changes are applied to the input signals with desired delays.

# Testbench Architecture Models, Assert, and Report

## Testbench Architecture Models

As mentioned in the previous section, there are three main testbench architecture models:

### Simple Testbench

Works for simple designs with a few inputs and outputs. Values are applied to the inputs, and the outputs are monitored. Each input value is applied with a delay to allow the UUT to process the input and generate the output. This resembles the data-flow style in VHDL, where input signals are directly assigned using `<=`, and changes are triggered after specific times using the `after` keyword.

```
begin
    -- Apply values to the input signals with delays
    input_signal1 <= '0', '1' after 10 ns, '0' after 20 ns;
    input_signal2 <= '1', '0' after 10 ns, '0' after 20 ns;
    -- Monitor the output signals
    assert output_signal1 = '1' and output_signal2 = '0'
        report "Test failed"
```

```
        severity error;  
    wait;  
end process;
```

## Process Statement Testbench

This resembles the behavioral style in VHDL, where a process statement is used, and each line within the process is executed sequentially.

```
begin  
    -- Stimulus process  
    stimulus : process  
    begin  
        input_signal1 <= '0';  
        input_signal2 <= '1';  
        wait for 10 ns;  
        input_signal1 <= '1';  
        input_signal2 <= '0';  
        wait for 10 ns;  
        input_signal1 <= '0';  
        input_signal2 <= '0';  
        wait for 10 ns;  
        wait;  
    end process stimulus;  
  
    -- Monitor the output signals  
    process  
    begin  
        wait for 10 ns;  
        assert output_signal1 = '1' and output_signal2 = '0'  
            report "Test failed"  
            severity error;  
        wait;  
    end process;  
  
end process;
```

## Look-up Table Testbench

This extends the process statement approach by storing input combinations in a lookup table (either signal or constant) and assigning values in a for-loop within the process statement.

```

begin
    -- Lookup table for input signals
    type input_table is array (natural range <>) of std_logic_vector(1 downto 0);
    constant input_values : input_table := (
        "00", "01", "10", "11"
    );

    -- Stimulus process
    stimulus : process
    begin
        for i in input_values'range loop
            input_signal1 <= input_values(i)(0);
            input_signal2 <= input_values(i)(1);
            wait for 10 ns;
        end loop;
        wait;
    end process stimulus;

    -- Monitor the output signals
    process
    begin
        wait for 10 ns;
        assert output_signal1 = '1' and output_signal2 = '0'
            report "Test failed"
            severity error;
        wait;
    end process;

end process;

```

## Assert and Report

Since testbench are for simulation purposes, it is important to include assertions and reports to verify the correctness of the design. The assert statement checks if a condition is true and reports an error if it is false. The report statement is used to display a message when the condition is false. Assert more likely `printf` in C language.

```

begin
    -- Monitor the output signals

```

```

process
begin
    wait for 10 ns;
    assert output_signal1 = '1' and output_signal2 = '0'
        report "Test failed"
        severity error;
    wait;
end process;

```

# Testbench for Sequential Circuit

Sequential circuit testbenches are similar to those for combinational circuits but include additional inputs like Clock and Reset. Clock signals require a separate process statement, while the reset signal can be configured as needed.

```

library ieee;
use ieee.std_logic_1164.all;

entity up_down_counter is
    port (
        clk : in std_logic;
        rst : in std_logic;
        up_down : in std_logic;
        count : out std_logic_vector(3 downto 0)
    );
end up_down_counter;

architecture rtl of up_down_counter is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            count <= "0000";
        elsif rising_edge(clk) then
            if up_down = '1' then
                count <= count + 1;
            else
                count <= count - 1;
            end if;
        end if;
    end process;
end architecture;

```

```

        end if;
    end process;
end rtl;

```

In this case, a synchronous up/down counter is tested with a testbench combining all three architecture models. The Clock uses a process statement, and Reset uses a simple assignment. Inputs are declared upfront as they do not change during the simulation.

```

library ieee;
use ieee.std_logic_1164.all;

entity testbench is
end testbench;

architecture tb_arch of testbench is
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal up_down : std_logic := '0';
    signal count : std_logic_vector(3 downto 0);

    component up_down_counter
        port (
            clk : in std_logic;
            rst : in std_logic;
            up_down : in std_logic;
            count : out std_logic_vector(3 downto 0)
        );
    end component;

begin
    UUT_inst : up_down_counter
        port map (
            clk => clk,
            rst => rst,
            up_down => up_down,
            count => count
        );

    -- Clock process
    clk_process : process
begin

```

```
    clk <= not clk;
    wait for 10 ns;
end process;

-- Apply stimulus to the UUT
up_down <= '1';
wait for 10 ns;
up_down <= '0';
wait for 10 ns;
rst <= '1';
wait for 10 ns;
rst <= '0';
wait for 10 ns;
wait;

end tb_arch;
```

---

Revision #1

Created 1 March 2025 14:58:21 by GI

Updated 1 March 2025 14:59:31 by GI