Module 4: Testbench

- Intro, Types, and Port Mapping
- Testbench, Assert, and Report
- File Operations

Intro, Types, and Port Mapping

Introduction

In VHDL, a testbench is a module that instantiates the unit under test (UUT) and applies stimulus to it. The stimulus can be a set of input signals, a clock signal, or a reset signal. The testbench also monitors the output signals of the UUT and compares them to the expected results. The testbench can be used to verify the functionality of the UUT and to debug any issues that arise during simulation. There're some benefits of using a testbench:

- It allows you to verify the functionality of your design before you synthesize it.
- It allows you to test your design under different conditions and edge cases.
- It allows you to debug your design by monitoring the signals in the simulation.
- It allows you to automate the testing process by running a set of test cases automatically.

Types of Testbenches

There are several types of testbenches that you can use to test your design:

Simple Testbench

A simple testbench is a basic testbench that applies stimulus to the UUT and monitors the output signals. It is useful for testing simple designs that do not require complex stimulus or verification.

Process Statement Testbench

A process statement testbench is a testbench that uses a process statement to generate stimulus for the UUT. It is useful for testing designs that require sequential stimulus or verification.

Look-up Table Testbench

A look-up table testbench is a testbench that uses a look-up table to generate stimulus for the UUT. It is useful for testing designs that require complex stimulus or verification.

Port Mapping

Port Map is a VHDL construct that allows you to connect the ports of a component to signals in the testbench. It is used to instantiate the UUT in the testbench and to connect the input and output signals of the UUT to the stimulus and monitor signals in the testbench. The syntax of the port map is as follows:

```
UUT_inst : entity work.UUT
    port map (
        input_signal1 => stimulus_signal1,
        input_signal2 => stimulus_signal2,
        output_signal1 => monitor_signal1,
        output_signal2 => monitor_signal2
    );
```

Code above shows an example of a port map for a UUT with two input signals and two output signals. The input signals are connected to the stimulus signals in the testbench, and the output signals are connected to the monitor signals in the testbench.

Example

Here is an example of a testbench that instantiates a UUT with two input signals and two output signals and connects them to the stimulus and monitor signals in the testbench:

```
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;
```

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 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;
```

File Operations

Read and Write File

In VHDL, you can read and write files using the `textio` package. The `textio` package provides procedures and functions for reading and writing text files. You can use the `textio` package to read data from a file into a variable or write data from a variable to a file.

Reading from a File

We can use the TextIO library to handle file operations in VHDL. This feature is useful for reading input from files during simulation. Here's how to read inputs from a file:

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

entity read_file is
end read_file;

architecture rtl of read_file is
    file input_file : text open read_mode is "input.txt";
    variable line : line;
    variable data : integer;
begin
    while not endfile(input_file) loop
        readline(input_file, line);
        read(line, data);
        report "Read data: " & integer'image(data);
    end loop;
    file_close(input_file);
    wait;
end rtl;
```

Code above shows an example of reading data from a file in VHDL. The file `input.txt` is opened in read mode, and data is read line by line using the `readline` procedure. The data is then converted to an integer using the `read` function and displayed using the `report` statement.

Writing to a File

We can also write testbench results to a file for further analysis. Use the TextIO library's `write` and `writeline` functions to save data to a file:

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

entity write_file is
end write_file;

architecture rtl of write_file is
    file output_file : text open write_mode is "output.txt";
    variable data : integer := 42;
begin
    write(output_file, data);
    writeline(output_file, "Data written to file: " & integer'image(data));
    file_close(output_file);
    wait;
end rtl;
```

> For more information on the `textio` package, refer to the IEEE Standard VHDL Language Reference Manual.