Digital System Design

Module 4: Testbench

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.

Module 5: Structural Style Programming In VHDL

Structural Style, Port Mapping, and Generic Map

Structural Style Programming

Structural Style Programming in VHDL allows designers to build digital circuits using basic components connected to form a more complex system. In this approach, circuits are represented as collections of entities connected in a specific way to achieve the desired function.

Port Mapping

Port mapping is the process of associating the ports of a VHDL component (entity) with signals in the architecture. This allows entities to be connected with the actual circuit in the design.

Important Points

- **Entity Definition**: An entity must be defined first, which includes its ports and data types.
- **Port-Map List**: Ports of the entity are mapped to corresponding signals in the architecture.
- **Port Mapping Order**: Mapping must follow the order defined in the entity.
- **Signal Declaration**: Signals used in port mapping must be declared in the architecture.

Example

```
entity AND2 is
    port (
        A, B: in std_logic;
        Y: out std_logic
    );
```

```
end entity;

-- Port mapping
architecture RTL of AND2 is
begin
    Y <= A and B;
end architecture;

-- Using the entity with port mapping
D1: AND2 port map (
    A => input_signal_A,
    B => input_signal_B,
    Y => output_signal
);
```

Generic Map

Generic map is the process of mapping generic values in an entity to corresponding values in the architecture. Generics are parameters that can be set for an entity to configure the behavior or characteristics of a component.

Important Points

- **Generic**: Parameters used to modify characteristics of an entity.
- **Generic Map**: Defines the values for generics when instantiating an entity.
- **Default Value**: Generics often have default values, but they can be overwritten during instantiation.

Example

```
entity Counter is
    generic (
        WIDTH: positive := 8; -- Default value for WIDTH is 8
        ENABLED: boolean := true -- Default value for ENABLED is true
    );
    port (
        clk: in std_logic;
        reset: in std_logic;
```

```
        count: out std_logic_vector(WIDTH-1 downto 0)
    );
end entity;

-- Generic map when instantiating the entity
architecture RTL of MyDesign is
    signal my_counter_output: std_logic_vector(7 downto 0);
begin
    my_counter_inst: Counter
        generic map (
            WIDTH => 8, -- Generic value WIDTH is reset to 8
            ENABLED => true -- Generic value ENABLED is reset to true
        )
        port map (
            clk => system_clock,
            reset => reset_signal,
            count => my_counter_output
        );
end architecture;
```

VHDL Modularity

We will build a 4-bit Ripple Carry Adder using 4 Full Adders in Structural Style Programming. Each Full Adder's carry-out serves as the carry-in for the next Full Adder, creating a ripple effect in addition.

Step 1 - Full Adder Entity

Inside full adder entity, we will declare the input and output ports. The input ports are A, B, and Cin, and the output ports are Sum and Cout.

```
entity full_adder is
    port(
        A, B, Cin : in std_logic;
        Sum, Cout : out std_logic
    );
end entity full_adder;

architecture structural of full_adder is
begin
    Sum <= A xor B xor Cin;
    Cout <= (A and B) or (B and Cin) or (A and Cin);
end architecture structural;
```

Step 2 - Ripple Carry Adder Architecture

Next step is to create the architecture for the 4-bit Ripple Carry Adder. We will instantiate 4 Full Adders and connect them in a way that the carry-out of one Full Adder is connected to the carry-in of the next Full Adder.

```
entity ripple_carry_adder is
    port(
        A, B : in std_logic_vector(3 downto 0);
        Sum : out std_logic_vector(3 downto 0);
        Cout : out std_logic
    );
end entity ripple_carry_adder;

architecture structural of ripple_carry_adder is
    component full_adder
        port(
            A, B, Cin : in std_logic;
            Sum, Cout : out std_logic
        );
    end component full_adder;

    signal C : std_logic_vector(3 downto 0);
begin
    FA0 : full_adder port map(A(0), B(0), '0', Sum(0), C(0));
    FA1 : full_adder port map(A(1), B(1), C(0), Sum(1), C(1));
    FA2 : full_adder port map(A(2), B(2), C(1), Sum(2), C(2));
    FA3 : full_adder port map(A(3), B(3), C(2), Sum(3), Cout);
end architecture structural;
```

Based on the above code, we can see that the carry-out of each Full Adder is connected to the carry-in of the next Full Adder. This creates a ripple effect in addition. Therefore the port map explanation is as follows:

FA0 - Full Adder 0

- A(0) and B(0) are the input bits for the first Full Adder.
- '0' is the carry-in for the first Full Adder.
- Sum(0) is the output sum of the first Full Adder.
- C(0) is the carry-out of the first Full Adder.

FA1 - Full Adder 1

- A(1) and B(1) are the input bits for the second Full Adder.
- C(0) is the carry-in for the second Full Adder.
- Sum(1) is the output sum of the second Full Adder.

- C(1) is the carry-out of the second Full Adder.

FA2 - Full Adder 2

- A(2) and B(2) are the input bits for the third Full Adder.
- C(1) is the carry-in for the third Full Adder.
- Sum(2) is the output sum of the third Full Adder.
- C(2) is the carry-out of the third Full Adder.

FA3 - Full Adder 3

- A(3) and B(3) are the input bits for the fourth Full Adder.
- C(2) is the carry-in for the fourth Full Adder.
- Sum(3) is the output sum of the fourth Full Adder.
- Cout is the carry-out of the fourth Full Adder.

With this architecture, we have successfully implemented a 4-bit Ripple Carry Adder using 4 Full Adders in Structural Style Programming.

Step 3 - Testbench

To test the functionality of the 4-bit Ripple Carry Adder, we will create a testbench that provides input values to the adder and checks the output values.

```
entity tb_ripple_carry_adder is
end entity tb_ripple_carry_adder;

architecture testbench of tb_ripple_carry_adder is
    signal A, B : std_logic_vector(3 downto 0);
    signal Sum : std_logic_vector(3 downto 0);
    signal Cout : std_logic;
    signal clk : std_logic := '0';

    component ripple_carry_adder
        port(
            A, B : in std_logic_vector(3 downto 0);
            Sum : out std_logic_vector(3 downto 0);
            Cout : out std_logic
        );
```

```
    end component ripple_carry_adder;

begin
    dut : ripple_carry_adder port map(A, B, Sum, Cout);

    process
    begin
        A <= "0000"; B <= "0000"; wait for 10 ns;
        A <= "0001"; B <= "0001"; wait for 10 ns;
        A <= "0010"; B <= "0010"; wait for 10 ns;
        A <= "0011"; B <= "0011"; wait for 10 ns;
        A <= "0100"; B <= "0100"; wait for 10 ns;
        A <= "0101"; B <= "0101"; wait for 10 ns;
        A <= "0110"; B <= "0110"; wait for 10 ns;
        A <= "0111"; B <= "0111"; wait for 10 ns;
        A <= "1000"; B <= "1000"; wait for 10 ns;
        A <= "1001"; B <= "1001"; wait for 10 ns;
        A <= "1010"; B <= "1010"; wait for 10 ns;
        A <= "1011"; B <= "1011"; wait for 10 ns;
        A <= "1100"; B <= "1100"; wait for 10 ns;
        A <= "1101"; B <= "1101"; wait for 10 ns;
        A <= "1110"; B <= "1110"; wait for 10 ns;
        A <= "1111"; B <= "1111"; wait for 10 ns;
        wait;
    end process;

end architecture testbench;
```

In the testbench, we provide different input values to the 4-bit Ripple Carry Adder and observe the output values. The testbench will simulate the addition process and verify the correctness of the adder. The output values can be checked against the expected results to ensure the adder is functioning correctly.

Array and Types in VHDL

Array

In VHDL, an array is a collection of elements that have the same data type. You can think of an array as a variable that has many elements with the same data type, and these elements are indexed for access. The index can be a number or another indexable type, such as integer, natural, or std_logic_vector. Arrays can have one dimension (one-dimensional array) or more (two-dimensional array, three-dimensional array, and so on). Two-dimensional arrays are often used to represent tables or matrices.

Types

In VHDL, a type is a definition used to declare a new data type. Types can be used to define complex data types, such as arrays or records, or as types used to declare variables, ports, or signals. Types can also be used to describe the properties and structure of data. VHDL has predefined data types, such as std_logic, std_logic_vector, integer, and others, but we can also create our own custom data types. Types that are predefined or embedded in the VHDL library are called "built-in types," while types that we define ourselves are called "derived types."

Example

```
-- Custom type definition
type color is (red, green, blue, yellow, black, white);

-- Variable declaration using custom type
signal primary_color: color;

-- Array declaration using custom type
type color_array is array (natural range <>) of color;

-- Array instantiation
signal color_table: color_array(0 to 3);
```

In the example above, we define a custom type `color` with specific values. We then declare a signal `primary_color` using this custom type. We also define an array type `color_array` that can hold elements of type `color`, and we instantiate an array `color_table` with a range of 0 to 3 using this array type.

Module 6: Looping

While Loop and For Loop

What is looping in VHDL?

A looping construct (looping statement) in VHDL is an instruction that allows a program to repeat the same block of code iteratively. In VHDL, there are two types of looping constructs: the while-loop and the for-loop.

While Loop

```
label_name: while (condition) loop
    -- Statements
end loop label_name;
```

For Loop

```
for index in range loop
    -- Statements
end loop;
```

The following are things that must be considered when using the looping construct:

- While-loop and for-loop can only be used inside a process statement.
- Unlike for-loop, for-generate can be used outside the process statement.
- Like process statements and structural assignments, looping constructs can be labeled. However, labeling is optional and only serves to help understand the code, especially in the nested-loop section.

While Loop

While loops operate by repeatedly checking the result of a condition. As long as this condition returns "true," the code within it will continue to execute. The looping process will end when the condition returns a "false" value. While loops are very useful when we don't have exact information

about how many times the code needs to be repeated.

However, it's important to remember that there must be a method in the program to change the result of the condition from "true" to "false" so that the loop can stop. Without such a method, the program will be stuck in an infinite loop.

Here is an example of VHDL code for a Shift Register that can be modified by implementing looping.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Shift_Register is
    port (
        clk, reset: in std_logic;
        data_in: in std_logic;
        data_out: out std_logic
    );
end entity;

architecture RTL of Shift_Register is
    signal reg: std_logic_vector(7 downto 0) := (others => '0');
begin
    process (clk, reset)
    begin
        if reset = '1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            reg(7) <= reg(6);
            reg(6) <= reg(5);
            reg(5) <= reg(4);
            reg(4) <= reg(3);
            reg(3) <= reg(2);
            reg(2) <= reg(1);
            reg(1) <= reg(0);
            reg(0) <= data_in;
        end if;
    end process;

    data_out <= reg(7);
```

```
end architecture;
```

The code above still uses the regular assignment method. To implement while loop, the process in the code above can be replaced with the following code.

```
process (clk, reset)
    variable i: integer := 0;
begin
    if reset = '1' then
        reg <= (others => '0');
    elsif rising_edge(clk) then
        while i < 8 loop
            reg(i) <= reg(i+1);
            i := i + 1;
        end loop;
    end if;
end process;
```

For Loop

A for loop repeats the code within a certain range using an index variable. The code inside is executed once on each iteration. This type of looping is suitable when the number of iterations required is known in advance.

The following are things to keep in mind when using for-loops: The range to be iterated over can be defined directly or taken from another previously declared variable in the form of a vector, array, or list.

- Index variables do not need to be declared beforehand; they will be created automatically.
- Index variables can be used in calculations inside the loop but cannot be manually changed.
- Index variables can only increase or decrease by 1 at each iteration.
- Index variables can be incremented or decremented by changing the range from “i0 to in” to “in downto i0”.

```
process (clk, reset)
begin
    if reset = '1' then
        reg <= (others => '0');
    elsif rising_edge(clk) then
```

```
        for i in 7 downto 1 loop
            reg(i) <= reg(i-1);
        end loop;
        reg(0) <= data_in;
    end if;
end process;
```

Loop Control: Next & Exit Statements

The following are two additional statements that can be used to control the looping construct:

Next

The `next` statement is used to skip the remaining code in the current iteration of the loop and proceed to the next iteration. In a for-loop, the index variable will be incremented or decremented automatically before the next iteration. In a while loop, this behavior depends on the logic applied. There are two ways to use the `next` statement.

```
next when (condition);
if (condition) then
    next;
end if;
```

Exit

The `exit` statement is used to stop the loop forcibly. When this statement is executed, the loop will terminate immediately, and the program will continue executing the code that follows the loop. There are two ways to use the `exit` statement.

```
exit when (condition);
if (condition) then
    exit;
end if;
```

Example

Below is an example of a for-loop that uses the `next` and `exit` statements.

```
process
    variable i: integer := 0;
begin
    for i in 0 to 7 loop
        if i = 3 then
            next; -- Skip the rest of the code in this iteration
        elsif i = 5 then
            exit; -- Exit the loop
        end if;
        reg(i) <= reg(i+1);
    end loop;
end process;
```

This VHDL code snippet demonstrates the use of a for-loop within a process statement. Here's the detailed explanation:

1. **Process Declaration**:
   - The process block is declared, and a variable `i` of type integer is initialized to 0.
2. **For-Loop**:
   - The for-loop iterates over the range from 0 to 7, inclusive. The loop variable `i` is automatically created and incremented with each iteration.
3. **Conditional Statements**:
   - Inside the loop, there are two conditional checks:
     - `if i = 3 then`: If `i` equals 3, the `next` statement is executed. This causes the loop to skip the remaining code in the current iteration and proceed to the next iteration.
     - `elsif i = 5 then`: If `i` equals 5, the `exit` statement is executed. This causes the loop to terminate immediately, and the process continues with the code following the loop.
4. **Register Assignment**:
   - If neither of the above conditions is met, the code `reg(i) <= reg(i+1);` is executed. This assigns the value of `reg(i+1)` to `reg(i)`.
5. **End Loop and Process**:
   - The loop ends after completing the iterations from 0 to 7 unless exited early by the `exit` statement.
   - The process block ends after the loop.

Summary

- The loop iterates from 0 to 7.
- When `i` is 3, the loop skips the current iteration.

- When `i` is 5, the loop exits.
- For other values of `i`, `reg(i)` is assigned the value of `reg(i+1)`.

Module 7: Procedure, Function, and Impure Function

Procedure and Function

Procedure in VHDL

In VHDL, a "procedure" is a language construct used to group multiple statements and specific tasks into a single block of code. Procedures help organize and understand complex VHDL designs.

Procedure Declaration

A procedure is defined using a procedure declaration. This declaration specifies the name of the procedure, the required parameters (if any), and the type of data returned (if any). Here is an example of a procedure declaration in VHDL:

```
procedure procedure_name(param1: in type1; param2: out type2) is
begin
    -- Procedure body
end procedure_name;
```

Procedures can accept parameters as arguments. These parameters are used to send data into the procedure for processing. The required parameters can be defined in the procedure declaration.

The code block in a procedure is where the tasks to be performed by the procedure are placed. You can write statements in the procedure code block to perform various operations. These statements can include calculations, tests, data manipulation, etc.

Procedure Call

To use a procedure, you can call it from the main part of the design or another procedure. Calling a procedure is done by providing arguments that match the parameters defined in the procedure declaration. Here is an example of using a procedure:

```
variable_name := procedure_name(arg1, arg2);
```

In this example, `arg1` and `arg2` are the arguments passed to the procedure, and the result of the procedure is assigned to `variable_name`. Here's an example of calling a procedure:

```
procedure add_numbers(a: in integer; b: in integer; sum: out integer) is
begin
    sum := a + b;
end add_numbers;

-- Calling the procedure
variable_name := add_numbers(5, 3);
```

Function in VHDL

In VHDL (VHSIC Hardware Description Language), "function" and "impure function" are two concepts used to create subprograms that can be used in hardware descriptions. The following is an explanation of both:

Function

Procedures in VHDL do not return values directly. Instead, they can modify the values of their parameters or variables within their scope. If a return value is needed, a function should be used instead.

Functions in VHDL are subprograms used to perform calculations or data processing that return a value as a result. You can think of them as mathematical functions in programming. Functions can have input arguments (parameters) that are used in the calculation. The result of the function will depend on the values of the input arguments provided. Here is an example of function usage in VHDL:

```
function function_name(param1: in type1; param2: in type2) return type3 is
begin
    -- Function body
    return result;
end function_name;
```

The code above just shows the basic structure of a function in VHDL. The actual implementation of the function will depend on the specific task it is designed to perform. Here's an example of a function that calculates the sum of two numbers:

```
function add_numbers(a: in integer; b: in integer) return integer is
begin
```

```
    return a + b;
end add_numbers;

-- Using the function
variable_name := add_numbers(5, 3);
```

Impure Function

An impure function is a function that can have properties that are unpredictable or changeable when executed. This means that the result of an impure function can depend on external factors unknown to the program, such as the time at which it is executed, random values, or global variables that can change.

An impure function is usually used when its result depends on values outside the input arguments and may change over time. Impure functions cannot be used in hardware descriptions that are deterministic or synchronous, as is commonly expected in VHDL.

```
function random_number return integer is
begin
    return integer'image(random(0, 100));
end random_number;
```

Procedure, Function, and Impure Function Synthesis

In VHDL, both "functions" and "procedures" can be used in the description of hardware. However, it should be understood that hardware synthesis is usually more suitable for implementations based on deterministic and synchronous behavior. Therefore, there are some restrictions on the use of functions and procedures in the context of synthesis:

Procedures

Procedures in VHDL perform tasks without returning values. They can also be used in hardware descriptions to organize operations and code. Hardware synthesis usually replaces a procedure call with a corresponding physical action in the target hardware. Therefore, deterministic procedures can be synthesized. However, there are some limitations in the use of procedures that depend on time streams or behaviors that are difficult to predict. Some VHDL compilers may not support the synthesis of such procedures.

Functions

VHDL functions that do not have impure properties (e.g., produce deterministic values based on input arguments alone) can usually be synthesized well.

Impure Functions

Impure functions, which produce results that are not predictable or depend on external factors, are usually not suitable for deterministic hardware synthesis. Impure functions that depend on random or non-deterministic behavior will not synthesize well because the resulting hardware must be deterministic and predictable.

So, while functions and procedures can be used in hardware descriptions and can be synthesized if they meet specific requirements, impure functions are not usually suitable for VHDL synthesis.

Difference Between It All

Criteria	Procedure	Function	Impure Function
Destination	Performing tasks without returning values	Returns the calculated values	Returning values with unpredictable properties
Arguments	Can have input and output arguments	Can have input arguments only	Can have input arguments only
Return value	No return value	Returns a value	Returns a value
Usage	Used for organizing code and operations	Used for calculations and data processing	Used for calculations and data processing with unpredictable properties
Example	Procedure to add two numbers	Function to add two numbers	Function to generate random numbers
Synthesis	Can be synthesized if deterministic	Can be synthesized if deterministic	Usually not suitable for synthesis

Example

Procedure

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Adder is
    port (
        A, B: in std_logic;
        Sum: out std_logic
    );
end entity;

architecture RTL of Adder is
    procedure add_numbers(a: in std_logic; b: in std_logic; sum: out std_logic) is
    begin
        sum <= a xor b;
    end add_numbers;
begin
    process (A, B)
    begin
        add_numbers(A, B, Sum);
    end process;
end architecture;
```

Function

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity Adder is
    port (
        A, B: in std_logic;
        Sum: out std_logic
    );
end entity;

architecture RTL of Adder is
    function add_numbers(a: in std_logic; b: in std_logic) return std_logic is
    begin
        return a xor b;
    end add_numbers;
begin
    process (A, B)
    begin
        Sum <= add_numbers(A, B);
    end process;
end architecture;
```

Impure Function

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.MATH_REAL.ALL;

entity Adder is
    port (
        Sum: out std_logic
    );
end entity;

architecture RTL of Adder is
    function random_number return std_logic is
    begin
        return REAL'(uniform(0.0, 1.0) > 0.5);
    end random_number;
begin
    process
    begin
```

```
        Sum <= random_number;
    end process;
end architecture;
```