Digital System Design

Module 4: Testbench

Testbench and Port Mapping

Testbench

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;
```

In this example, the testbench instantiates a UUT with two input signals and two output signals and connects them to the stimulus and monitor signals in the testbench. The testbench applies stimulus to the UUT by setting the input signals to '0' and '1' and monitors the output signals of the UUT by comparing them to the expected results.

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 threen 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;
```

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;
```

Module 8 : Finite State Machine

Finite State Machine

A Finite State Machine (FSM), or Finite Automata, is a mathematical model of a system whose state is capable of changing. These systems have characteristics or behaviors that vary depending on the current state. In general, FSMs are divided into two types: FSMs with output and FSMs without output.

However, in this module, only FSMs with output will be discussed. FSMs with output can be divided into 2 types, namely Mealy Machine and Moore Machine.

Mealy Machine

Mealy State Machine is an FSM in which the next state is determined by the current input and the present state. Different inputs, along with different present states, will result in different next states. For example, in digital circuits, Registers exhibit this characteristic. In everyday life, the change in state of a substance can be modeled as a Mealy State Machine.

Mealy State Machine Mealy State Machine Circuit

Moore Machine

Moore State Machine is a type of FSM where the next state is only determined by the current state, without being affected by inputs. This means that any input will not change the next state. The hallmark of a Moore State Machine is its one-way cycle structure. For example, digital circuits such as counters exhibit this property. In everyday life, the metamorphosis cycle in animals can be modeled as a Moore State Machine.

Moore State Machine Moore State Machine Circuit

Finite State Machine in VHDL

Basically, FSM serves to describe the workings of a sequential circuit. Therefore, the VHDL code of an FSM is not much different from the VHDL code of an ordinary sequential circuit, which uses a process statement (behavioral model). There are many methods that can be used to create an FSM using VHDL. However, we will only learn the most basic method. The method requires a minimum of two processes, commonly called Synchronous Process and Combinatorial Process.

- Synchronous Process in the context of a Finite State Machine (FSM) is responsible for arranging state transitions based on the current state and received inputs. This process is similar to how a D flip-flop works, controlling state transitions by utilizing clock signals and memory to store the current state. Its main functions include controlling state transitions, input management, and integration with the clock and memory, ensuring that state transitions take place in a synchronized and orderly manner.
- The Combinatorial Process is responsible for generating outputs based on the current state and the inputs received. This process is similar to how logic gates work, generating outputs based on the inputs received. Its main functions include output management, integration with inputs, and decision-making based on the current state, ensuring that outputs are generated in a synchronized and orderly manner.

Finite State Machine

FSM Implementation Example in VHDL

Moore Machine

This FSM has two states: ST0 and ST1. In the ST0 state, the FSM outputs '0', and in ST1, the output is '1'. This FSM also accepts two inputs: CLR and TOG_EN. The CLR input returns the FSM to ST0, while TOG_EN determines whether the FSM can switch states.

Moore Machine
Image not found or type unknown

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

entity FSM_Moore is
    Port ( CLK : in STD_LOGIC;
           CLR : in STD_LOGIC;
           TOG_EN : in STD_LOGIC;
           Z1 : out STD_LOGIC);
end FSM_Moore;

architecture Behavioral of FSM_Moore is
    type state_type is (ST0, ST1);
    signal state, next_state : state_type;
begin
    process (CLK, CLR)
    begin
        if CLR = '1' then
            state <= ST0;
        elsif rising_edge(CLK) then
            state <= next_state;
        end if;
    end process;
```

```
    process (state, TOG_EN)
    begin
        case state is
            when ST0 =>
                if TOG_EN = '1' then
                    next_state <= ST1;
                else
                    next_state <= ST0;
                end if;
            when ST1 =>
                if TOG_EN = '1' then
                    next_state <= ST0;
                else
                    next_state <= ST1;
                end if;
        end case;
    end process;

    process (state)
    begin
        case state is
            when ST0 =>
                Z1 <= '0';
            when ST1 =>
                Z1 <= '1';
        end case;
    end process;
end Behavioral;
```

Penjelasan Kode

- In the code above, we define the FSM_Moore entity with three inputs (CLK, CLR, TOG_EN) and one output (Z1).
- In the Behavioral architecture, we define a state_type data type that contains two states: ST0 and ST1. In addition, we define two signals state and next_state of type state_type.
- In the first process, we use the state and next_state signals to set the state transition based on the input received. If the input CLR = '1', the FSM will return to the ST0 state. If the CLK input changes from '0' to '1', the FSM will move to the next_state.
- In the second process, we use the state and TOG_EN signals to set the output based on the current state and the input received. If the FSM is in state ST0 and input TOG_EN = '1', then the FSM will move to state ST1. If the FSM is in the ST1 state and the TOG_EN input

= '1', then the FSM will move to the ST0 state.
- In the third process, we use state signals to set the output based on the current state. If the FSM is in the ST0 state, then the output is '0'. If the FSM is in the ST1 state, then the output is '1'.

Mealy Machine

This FSM has three states: “00”, ‘01’, and ‘11’. In addition, this FSM accepts two inputs: SET and X. The SET input returns the FSM to state “11”, while X determines the next output and state

Mealy Machine

Image not found or type unknown

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

entity FSM_Mealy is
    Port ( CLK : in STD_LOGIC;
           SET : in STD_LOGIC;
           X : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (1 downto 0);
           Z2 : out STD_LOGIC);
end FSM_Mealy;

architecture Behavioral of FSM_Mealy is
    type state_type is (ST00, ST01, ST11);
    signal state, next_state : state_type;
begin
    process (CLK, SET)
    begin
        if SET = '1' then
            state <= ST11;
        elsif rising_edge(CLK) then
            state <= next_state;
        end if;
    end process;

    process (state, X)
    begin
        case state is
            when ST00 =>
```

```
            if X = '0' then
                next_state <= ST00;
            else
                next_state <= ST01;
            end if;
        when ST01 =>
            if X = '0' then
                next_state <= ST00;
            else
                next_state <= ST11;
            end if;
        when ST11 =>
            next_state <= ST11;
    end case;
end process;

process (state, X)
begin
    case state is
        when ST00 =>
            Y <= "00";
            Z2 <= '0';
        when ST01 =>
            Y <= "01";
            Z2 <= '1';
        when ST11 =>
            Y <= "11";
            Z2 <= X;
    end case;
end process;
end Behavioral;
```

Penjelasan Kode

- In the code above, we define the FSM_Mealy entity with four inputs (CLK, SET, X) and two outputs (Y, Z2).
- In the Behavioral architecture, we define a state_type data type that contains three states: ST00, ST01, and ST11. In addition, we define two signals state and next_state of type state_type.
- In the first process, we use the state and next_state signals to set the state transition based on the input received. If the SET input = '1', the FSM will return to the ST11 state. If

the CLK input changes from '0' to '1', the FSM will move to the next_state.

- In the second process, we use the state and X signals to set the output and state transition based on the current state and the received input. If the FSM is in state ST00 and input X = '0', then the FSM will remain in state ST00. If the FSM is in state ST00 and input X = '1', then the FSM will move to state ST01. If the FSM is in state ST01 and input X = '0', the FSM will move to state ST00. If the FSM is in state ST01 and input X = '1', the FSM will move to state ST11. If the FSM is in state ST11, the FSM will remain in state ST11.
- In the third process, we use the state and X signals to set the output based on the current state and the received input. If the FSM is in state ST00, then output Y is “00” and output Z2 is '0'. If the FSM is in the ST01 state, then the Y output is “01” and the Z2 output is '1'. If the FSM is in the ST11 state, then the Y output is “11” and the Z2 output is X.

Module 9 : Microprogramming

Microprogramming in VHDL

Microprogramming in VHDL

Microprogramming is a technique in computer design that involves using microinstruction sets or small steps executed by a microprocessor's control unit. VHDL often implements this using state or finite state machines (FSMs).

Instruction Set

An instruction set is a collection of instructions that a computer's microprocessor or CPU (Central Processing Unit) understands. Each CPU has its own instruction set consisting of a series of basic operations that the CPU can perform. These instructions include basic operations such as addition, subtraction, multiplication, data transfer, and logical operations. With VHDL, we can simulate the running of a processor when executing its instructions.

The instruction set of a computer architecture is one of the critical elements in the design of a CPU. It determines the types of operations the CPU can perform, the format of the instructions, and how the instructions are executed. Instruction sets can be divided into several categories, namely:

Arithmetic and Logic

- Addition, subtraction, multiplication, division
- Logical operations (AND, OR, NOT, XOR)

Data Transfer

- Data transfer between registers and memory
- Data transfer between internal registers

Flow Control Program

- Branching instructions (conditional and unconditional)
- Looping instructions

Input/Output

- Special instructions related to specific architectures or application needs

Code below is an example of an instruction set for a simple CPU to perform arithmetic and logic operations.

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

entity Microprogram_ADD is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           start : in STD_LOGIC;
           operand1 : out STD_LOGIC_VECTOR(7 downto 0);
           operand2 : in STD_LOGIC_VECTOR(7 downto 0);
           operand3 : in STD_LOGIC_VECTOR(7 downto 0)
           );
end Microprogram_ADD;

architecture Behavioral of Microprogram_ADD is
    type State_Type is (FETCH, DECODE, EXECUTE, COMPLETE);
    signal state : State_Type := FETCH;
    signal counter : integer := 0;
    signal data_reg : STD_LOGIC_VECTOR(7 downto 0);
    signal add_result : STD_LOGIC_VECTOR(7 downto 0);
begin
    process(clk, reset)
    begin
        if reset = '1' then
            state <= FETCH;
            counter <= 0;
            data_reg <= (others => '0');
            add_result <= (others => '0');
        elsif rising_edge(clk) then
            case state is
                when FETCH =>
                    if start = '1' then
                        state <= DECODE;
                    end if;
```

```
        when DECODE => --Microinstruction for decoding phase
            data_reg <= "00000001"; --Assume control signal for ADD
            counter <= counter + 1;
            if counter = 2 then
                state <= EXECUTE;
            end if;
        when EXECUTE => --Microinstruction for execution phase
            data_reg <= "00000010"; --Assume control signal for addition operation
            add_result <= operand2 + operand3;
            counter <= counter + 1;
            if counter = 3 then
                state <= COMPLETE;
            end if;
        when COMPLETE => --Microinstruction for completion phase
            data_reg <= "00000000"; --Reset control signals
            state <= FETCH; --Reset to initial state
    end case;
  end if;
end process;
operand1 <= add_result;
end Behavioral;
```

Based on the code above, the instruction set consists of four states: FETCH, DECODE, EXECUTE, and COMPLETE. The instruction set is designed to perform an addition operation. The `operand2` and `operand3` are input operands, while `operand1` is the output operand. The `start` signal is used to initiate the operation. The `data_reg` signal is used to store the control signals for the microinstructions. The `add_result` signal stores the result of the addition operation.

Microprogram_ADD

Image not found or type unknown

`State_Type` is a user-defined type that defines the states of the CPU. The `state` signal is used to keep track of the current state of the CPU. The `counter` signal is used to count the number of clock cycles. The `data_reg` signal stores the control signals for the microinstructions. The `add_result` signal stores the result of the addition operation.

In this module, we will learn how to design a control unit using microprogramming in VHDL. We will also learn how to implement the control unit using a finite state machine (FSM) and microinstructions. To find out more about how to make a 16 bit CPU, see here

Central Processing Unit (CPU)

CPU has four main components:

1. The Control Unit (together with IR) interprets machine language instructions and issues control signals to make the CPU execute those instructions.
2. ALU (Arithmetic Logic Unit), which performs arithmetic and logic operations.
3. Set Register (File Register), which stores temporary results related to calculations. Special Registers are also used. The Control Unit also uses Special Registers.
4. Internal bus structure for communication.

The function of the control unit is to decode the binary machine words in the IR (Instruction Register) and issue appropriate control signals, primarily to the CPU. These control signals are what cause the computer to execute its programs.

Control Unit Design

There are two related issues when considering control unit design:

1. The complexity of the Instruction Set architecture and
2. The microarchitecture used to implement the control unit. A computer's ISA (Instruction Set Architecture) is the set of assembly language instructions that the computer can execute. It can be viewed as an interface between the software (expressed as assembly language) and the hardware. More complex ISAs require more complicated control units. At some point in the development of computers, the complexity of the control unit becomes a problem for designers.

How Control Unit Works

The binary form of the instruction now resides in the IR (Instruction Register). The control unit decodes the instruction and generates the control signals required for the CPU to act according to the machine language instructions. The two main design categories here are hardwired and microprogrammed:

Hardwired → Control signals are generated as outputs from a series of basic logic gates; the inputs are from binary bits in the Instruction Register. Microprogram → Control signals are generated by a microprogram stored in the Control Read Only Memory.

Control Unit Microprogram

In the microprogrammed control unit, control signals correspond to bits in the micro-memory (CROM for Control Read Only Memory), which are read into the micro-MBR. This register is simply a set of D flip-flops, the contents of which are transmitted as signals.

Microprogrammed Control Unit

Image not found or type unknown

The micro-control unit (CU) performs the following steps:

1. Place the address into the micro-memory Address Register (μMAR),
2. The control word is read from the Control Read-Only Memory,
3. Place the microcode word into the micro-memory Buffer Register,
4. The control signal is output.