Functions, Procedures, and Testbenches

Introduction

VHDL lets you define sub-programs using procedures and functions. They are used to improve the readability and to exploit re-usability of VHDL code. Functions are equivalent to combinatorial logic and cannot be used to replace code that contains event or delay control operators (as used in a sequential logic). Procedures are more general than functions, and may contain timing controls. A testbench is a program or model written in HDL for the purposes of exercising and verifying the functional correctness of a hardware model via simulation. VHDL is primarily a means for hardware modeling (simulation), the language contains various resources for formatting, reading, storing, allocating dynamically, comparing, and writing simulation data, including input stimulus and output results.

In this lab, you will learn how to write functions, procedures, and testbenches. You will learn about the components of a testbench, and language constructs available to verify the correctness of the underlying hardware model. *Please refer to the Vivado tutorial on how to use the Vivado tool for creating projects and verifying digital circuits*.

Objectives

After completing this lab, you will be able to:
• Develop procedures for modeling a combinatorial circuit
• Develop functions for modeling a combinatorial circuit
• Develop a testbench to test and validate a design under test

Procedures   Part 1

A procedure provides the ability to execute common pieces of code from several different places in a model. A procedure can contain timing controls, and it can call other procedures and functions (described in next part). A procedure is defined, within a module definition, as:

```
procedure identifier [input/output port declarations] is
      [variable declarations]
begin
      procedure statements
end identifier
```

A procedure can have zero, one, or more arguments. Values are passed to and from a procedure through arguments. The arguments can be input, output, or inout. Here is an example of a procedure definition and usage.

```
procedure HAS_PROCEDURE (
      DIN : in STD_LOGIC_VECTOR(7 downto 0);
      DOUT : out STD_LOGIC_VECTOR(7 downto 0)) is
variable k : integer := 0;
variable count : integer := 0;
begin
      for k in 0 to 7 loop
            count <= count + 1;
            DOUT(7 - count) <= DIN(7);
      end loop;
end HAS_PROCEDURE;
```

1-1. Write a procedure called add_two_values which will take two 4-bit parameters, add them, and output a 4-bit sum and a carry. Write a module, called add_two_values_procedure, that calls the procedure with the operands received via input ports and outputs the result. Simulate the design with the provided testbench, add_two_values_procedure_tb.vhd, and verify the functionality.

1-1-1. Open Vivado and create a blank project called lab4_1_1**.**

1-1-2. Create and add the VHDL module, named add_two_values_procedure, that defines a procedure called add_two_values. The procedure will take two 4-bit parameters, add them, and output a 4-bit sum and a carry. The module will call the procedure with the operands received via input ports and outputs the result.

1-1-3. Simulate the design with the provided testbench: add_two_values_procedure_tb.vhd, and verify that the design works. Look for the messages displayed in the simulator console window.

1-2. Write a procedure called calc_even_parity which will take an 8-bit number, and compute and return parity. Write a module, called calc_even_parity_procedure, that calls the procedure with the operand received via the input port and outputs the result. Use the provided testbench, calc_even_parity_procedure_tb.vhd. Simulate the design and verify the functionality.

Functions   Part 2

Functions primarily differ from procedures in that a single value is returned from executing the code. Functions are used if all of the following conditions are true:

- There are no delay, timing, or event control constructs that are present
- It returns a single value
- There is at least one input argument
- There are no output or inout arguments
- There are no non-blocking assignments

In short, functions may implement only combinatorial behavior, i.e. they compute a value on the basis of the present value of the input arguments and return a single value. Each line of code in the function is executed sequentially (they are *not non-blocking*). They are used in the right hand side of an assignment statement. Here is an example of a function definition and call. An example of a function definition in VHDL is as follows:

```
function identifier [input port declarations] return type is
      [variable declarations]
begin
      function statements
end identifier
```

To call a function, one needs to use the function identifier (with input(s) defined) as an assignment operand in a process block:

```
Func_Out <= identifier (input);
```

Here is code snippet showing how functions are used:

```
function HAS_FUNCTION (
      DIN : STD_LOGIC_VECTOR(7 downto 0);)
return STD_LOGIC_VECTOR is
      variable k : integer := 0;
      variable count : integer := 0;
      variable reverse_bits : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
begin
      for k in 0 to 7 loop
            reverse_bits(7 – count) := DIN(count);
            count := count + 1;
      end loop;
return reverse_bits;
end HAS_TASK;
```

2-1.   Write a function called add_two_values which will take two 4-bit parameters, add them, and return a 5-bit sum. Write a module, called add_two_values_function, with two 4-bit input ports and one 5-bit output port and calls the function. Simulate the design with the provided testbench, add_two_values_function_tb.vhd, and verify the functionality.

2-1-1. Open Vivado and create a blank project called lab4_2_1**.**

2-1-2. Create and add a VHDL module, called add_two_values_function, that defines a function called add_two_values that takes two 4-bit parameters, add them, and return a 5-bit sum. The module will have two 4-bit input ports and one 5-bit output port. It will call the function.

2-1-3. Simulate the design with the provided add_two_values_function_tb.vhd for 50 ns and verify that the design works.

2-2.   Write a function called calc_ones which will take an 8-bit number, and calculate and return number of ones. Write a module, called calc_ones_function, with one 8-bit input port and one 3-bit output port and calls the function. Simulate the design with the provided testbench, calc_ones_function_tb.vhd, and verify the functionality.

Testbench   Part 3

The major components of a testbench are:
  - o An empty entity, which defines the testbench top-level structure
    - ▪ A testbench usually does not have input or output ports
  - o Internal signals, which will drive the stimuli into the UUT and monitor the response from the UUT
    - ▪ Signal to drive and monitor
  - o Component(s) declaration of UUT and its lower-level components
  - o UUT instantiation
  - o Stimuli generation
    - ▪ Write statements to create stimulus and procedural block
  - o Response monitoring and comparing
    - ▪ Self-testing statements that will report values, error, and warnings

Here is an example of the testbench we used in the Vivado Tutorial lab.

```
14    use STD.textio.all;
15    use IEEE.std_logic_textio.all;
16    Entity tutorial_tb Is
17    end tutorial_tb;
18
19    Architecture behavior of tutorial_tb Is
20        Component tutorial
21        port (
22            swt : in STD_LOGIC_VECTOR(7 downto 0);
23            led : out STD_LOGIC_VECTOR(7 downto 0)
24            );
25        End Component;
26
27        Signal switch : STD_LOGIC_VECTOR(7 downto 0) := X"00";
28        Signal led_out : STD_LOGIC_VECTOR(7 downto 0) := X"00";
29        Signal led_exp_out : STD_LOGIC_VECTOR(7 downto 0) := X"00";
30        Signal count_int_2 : STD_LOGIC_VECTOR(7 downto 0) := X"00";
31
32        procedure expected_led (
33            swt_in : in std_logic_vector(7 downto 0);
34            led_expected : out std_logic_vector(7 downto 0)
35            ) is
36
37        Variable led_expected_int : std_logic_vector(7 downto 0) := "00000000";
38
39        begin
40            led_expected_int(0) := not(swt_in(0));
41            led_expected_int(1) := swt_in(1) and not(swt_in(2));
42            led_expected_int(2) := swt_in(2) and swt_in(3);
43            led_expected_int(3) := led_expected_int(1) or led_expected_int(3);
44            led_expected_int(7 downto 4) := swt_in(7 downto 4);
45
46            led_expected := led_expected_int;
47        end expected_led;
48
49    begin
50        uut:  tutorial PORT MAP (
51                swt => switch,
52                led => led_out
53            );
54
55        process
56            variable s : line;
57            variable i : integer := 0;
58            variable count : integer := 0;
59            variable proc_out : STD_LOGIC_VECTOR(7 downto 0);
60
61        begin
62            for i in 0 to 127 loop
63                count := count + 1;
64
65                wait for 50 ns; switch <= count_int_2;
66
67                wait for 10 ns; expected_led (switch, proc_out);
68                led_exp_out <= proc_out;
69
70                -- If the outputs match, then announce it to the simulator console.
71                if (led_out = proc_out) then
72                    write (s, string'("LED output MATCHED at ")); write (s, count ); write (s, string'(". Expected: ")); write (s, pro
73                    writeline (output, s);
74                else
75                    write (s, string'("LED output mis-matched at ")); write (s, count); write (s, string'(". Expected: ")); write (s, pr
76                    writeline (output, s);
77                end if;
78
79                -- Increment the switch value counters.
80                count_int_2 <= count_int_2 + 2;
81            end loop;
82
83        end process;
84    end behavior;
```

ΣΧILINX.

Skipping the first few lines of the code, Line 14 shows the TEXTIO package being used. Lines 16 and 17 define the testbench module name. Line 19 starts the architectural description of the code. Lines 19 to 25 defines the port mapping of the UUT VHDL module. Lines 27 to 30 specify the signals and vectors used in the testbench. Lines 32 to 47 define the procedure used to compute the expected UUT output. Lines 50 to 53 instantiates the UUT, mapping the input and output to signals defined earlier in the module. Line 55 begins the process where the behavioral code resides. Lines 56 to 59 defines local variables used in the process only. Lines 62 to 81 is the for-loop that will run 128 times. Lines 65 and 67 show how waits can be inserted to simulate delays. Lines 72 and 73 outputs to the simulation console window. The same applies to lines 75 and 76.

VHDL supports two types of delay modeling: (i) inertial and (ii) transport. The inertial delay is the delay that a gate or circuit may experience due to the physical nature of the gate or circuit. The inertial delay is also used to determine if the input has an effect on the gate or circuit. If the input does not remain changed at least for the initial delay then the input change is ignored. For example, an inertial delay of 5 ns means whenever input changes it should remain changed at least for 5 ns to have it considered as changed otherwise the change is ignored (considered a transient spike). The transport delay is the time-of-flight of a signal travelling a wire of a circuit.

In VHDL, inertial and transport delays are specified as part of assignment statements. Here are some examples of transport and inertial delays:

```
bit_result <= input_bit xor bit_mask after 10ns;
```

This is an inertial delay. The `after` keyword in the assignment statement indicates that the output will only change after 10 ns elapsed from changes in both of the input arguments. If one of the inputs reverts to its original value during the 10 ns delay, then the initial change will not be registered.

```
bit_result <= transport input_bit xor bit_mask after 10ns;
```

The example above shows how transport delay is specified. Notice the only difference with the previous example is the keyword `transport`. This delays the output by the time specified. In this case, the transport delay only delays the output by the 10 ns specified.

It is sometimes important to output results to the simulator console window. This may be necessary when comparing simulation outputs to determine if the code has passed or failed simulation. VHDL supports writing to either a text file or to the simulator console window. This is done via the TEXTIO package that can be added to a VHDL module via **use std.textio.all**. Packages allow for referencing functions and components in a design. In the case of simulation testbenches, it is often essential to output values and messages to the screen in addition to the waveform, depending on certain conditions to mark specific events. The following example shows how to output one line to the simulator console:

```
process
      variable s : line;
      variable n : integer := 5;
begin
      write (s, string'("The quick brown fox jumps over the lazy dog. ")); --
Writing a text string.
      write (s, n); --Writing a variable value.
      writeline (output, s); --Outputs the above to one line.
end process;
```
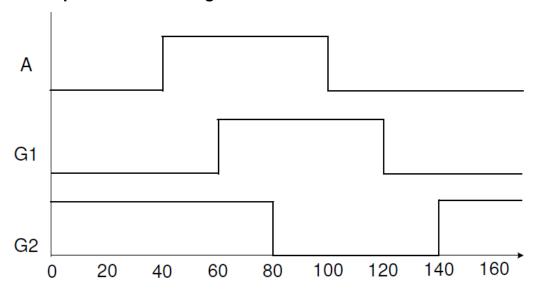
Where *s* is the line variable for writing both the string and the variable "n" to. The expected output of the example above will be:

The quick brown fox jumps over the lazy dog. 5

The write command is not synthesizable and should be used in testbenches only.

3-1. **Develop a testbench to verify the 4-bit ripple carry adder functionality. Use the ripple carry design (and its associated fulladder_dataflow module) you developed in Part 3-1 of Lab 2. Modify the design to include 2 units of inertial delay for every assignment statement in the design. Develop a testbench such that it computes expected output and compares the result.**

3-1-1. Open Vivado and create a blank project called lab4_3_1.

3-1-2. Add the VHDL module you used in part 3-1 of Lab 2. Modify the design to include 2 units of inertial delay for every assignment statement in the design. Develop a testbench such that it computes expected output, compares the result, and displays the result as "Test Passed" or "Test Failed".

3-1-3. Simulate the design for the desired time and verify that the design works. You should be able to verify just by looking in the console window of the simulator.

3-2. **Develop a testbench that generates the waveform shown below.**

3-2-1. Open Vivado and create a blank project called lab4_3_2.

3-2-2. Create and add the VHDL module that outputs the waveform shown above.

3-2-3. Simulate the design for 150 ns and verify that the correct output is generated.

Conclusion

In this lab, you learned how to write functions, procedures, and testbenches. You also learned the differences between functions and procedures, both in their definitions and in usage. You saw how function can be used in a testbench to compute the expected output.

XILINX