Tasks, Functions, and Testbench

Introduction

Verilog lets you define sub-programs using tasks and functions. They are used to improve the readability and to exploit re-usability 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). Tasks are more general than functions, and may contain timing controls. Testbench is a program or model written in any language for the purposes of exercising and verifying the functional correctness of a hardware model during the simulation. Verilog 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 tasks, functions, 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 tasks for modeling a combinatorial circuit
- Develop functions for modeling a combinatorial circuit
- Develop a testbench to test and validate a design under test

Tasks Part 1

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

```
task task_id;
   [declarations]
   procedure statements
endtask
```

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

```
module HAS_TASK;
  parameter MAXBITS = 8;
  task REVERSE_BITS;                // task definition starts here
    input [MAXBITS - 1 : 0] DIN;
    output [MAXBITS - 1 : 0] DOUT;
    integer k;

    begin
        for (k=0; k < MAXBITS; k = k +1)
           DOUT[MAXBITS-k] = DIN[k];
    end
  endtask                     // task definition ends here

  reg [MAXBITS - 1] REG_X, NEW_REG;
  always @ REG_X)
      REVERSE_BITS(REG_X,NEX_REG);        // task being called
endmodule
```


© copyright 2013 Xilinx

Verilog HDL also provides few system tasks. The system task name is preceded with a $. For example,

`$display` –Print the specified information to standard output with an end-of-line character.
e.g. `$display("At Simulation time %t, the x_var is %d");` will print x_var value in decimal format and time in current time format.
`$write` –Similar to the display task except it does not print an end-of-line character.
`$monitor` –Monitors the argument continuously. Whenever there is a change of value in an argument list, the entire argument list is displayed.
e.g.

```
initial
  $monitor("At %t, D= %d, CLK = %d", $time, D, CLK, "and Q is %b", Q);
```

Note that the argument list values are printed in `$display` task whenever it is called, whereas in the `$monitor` task it gets printed whenever value of one of the arguments change.

The system tasks are not synthesizable, i.e. they cannot be realized in real hardware.

1-1. Write a task 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_task, that calls the task with the operands received via input ports and outputs the result. Simulate the design with the provided testbench, add_two_values_tb.v, 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 Verilog module, named add_two_values_task, that defines a task called add_two_values. The task will take two 4-bit parameters, add them, and output a 4-bit sum and a carry. The module will call the task with the operands received via input ports and outputs the result.

1-1-3. Simulate the design with the provided testbench, add_two_values_task_tb.v, and verify that the design works. Look for the messages displayed by the $display task in the simulator console window.

1-2. Write a task called calc_even_parity which will take an 8-bit number, and compute and return parity. Write a module, called calc_even_parity_task, that calls the task with the operand received via the input port and outputs the result. Use the provided testbench, calc_even_parity_task_tb.v, that displays the result using $display system task. Simulate the design and verify the functionality.

Functions Part 2

Functions are declared within a parent module with the keywords **function** and **endfunction**. 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 argument
- There are no non-blocking assignments

© copyright 2013 Xilinx

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. They are used in the right hand side of an assignment statement. Here is an example of a function definition and call.

```
module HAS_FUNCTION(X_IN, REV_X);
  parameter MAXBITS = 8;
    output reg [MAXBITS – 1 : 0] REV_X;
    input [MAXBITS – 1 : 0] X_IN;

  function [MAXBITS – 1 : 0] REVERSE_BITS; // function definition starts here
    input [MAXBITS – 1 : 0] DIN;
    integer k;

    begin
        for (k=0; k < MAXBITS; k = k +1)
           REVERSE_BITS[MAXBITS-k] = DIN[k];
    end
  endfunction                       // function definition ends here


  always @ (X_IN)
    REV_X = REVERSE_BITS(X_IN);     // function being called

endmodule
```

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.v, 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 Verilog 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.v for 50 ns and verify that the design works.

2-2. Write a task 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.v, and verify the functionality.

© copyright 2013 Xilinx

Testbench Part 3

The major components of a testbench are:

- `*timescale* declaration
  - Specify the time unit for all delays
- Module, which defines the testbench top-level structure
  - A testbench usually does not have ports
- Internal signals, which will drive the stimuli into the UUT and monitor the response from the UUT
  - Signal to drive and monitor
- UUT instantiation
- Stimuli generation
  - Write statements to create stimulus and procedural block
- Response monitoring and comparing
  - Self-testing statements that will report values, error, and warnings
  - $display, $write, $strobe, and/or $monitor system tasks

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

```
 1 `timescale 1ns / 1ps
 2 module tutorial_tb(
 3     );
 4 
 5     reg [7:0] switches;
 6     wire [7:0] leds;
 7     reg [7:0] e_led;
 8 
 9     integer i;
10 
11     tutorial tut1(.led(leds),.swt(switches));
12 
13     function [7:0] expected_led;
14        input [7:0] swt;
15     begin
16        expected_led[0] = ~swt[0];
17        expected_led[1] = swt[1] & ~swt[2];
18        expected_led[3] = swt[2] & swt[3];
19        expected_led[2] = expected_led[1] | expected_led[3];
20        expected_led[7:4] = swt[7:4];
21     end
22     endfunction
23 
24     initial
25     begin
26         for (i=0; i < 255; i=i+2)
27         begin
28             #50 switches=i;
29             #10 e_led = expected_led(switches);
30             if(leds == e_led)
31                 $display("LED output matched at", $time);
32             else
33                 $display("LED output mis-matched at ",$time,": expected: %b, actual: %b", e_led, leds);
34         end
35     end
36 endmodule
```

Line 1 defines the `*timescale* directive. Lines 2 and 3 define the testbench module name. Note that typically testbench modules don't have ports listed in their port listing. Line 5 defines switches as a *reg* data type since it will be used to provide stimulus. It is connected to the instantiated device under test (tutorial). Line 11 instantiates design under test (tutorial) with instance name tut1 and input/output ports. Lines 13 through 22 define a function to compute expected output. Lines 24 through 35 define stimuli using a `initial` procedural statement. You will see messages generated by lines 31 and 33 using system task, $display, in the simulator console window. Line 29 calls the function expected_led by passing switches parameter and assigns returned (computed) output to e_led. The e_led is defined as

xup@xilinx.com
© copyright 2013 Xilinx

reg type on line 7 as it is receiving the output from the function call in the procedural statement (initial). Lines 28 and 29 also define inertial delays, 50 and 10 respectively, to model delays.

Verilog 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. Depending on the technology used, it can be in ps or ns. 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, 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 noise spike). The transport delay is the time-of-flight of a signal travelling a wire of a circuit. Here are some examples of transport and inertial delays:

```
wire #2 a_long_wire;    // transport delay of 2 units is modeled
xor #1 M1(sum, a, b);   // inertial delay of 1 unit exerted by xor gate
```

The `initial` statements are used in testbench to generate stimuli and control simulation execution. Here is an example of it:

```
initial begin
  #100 $finish; // run simulation for 100 units
end

initial begin
  #10 a=0; b=0; // a, b zero after 10 units delay.  Between 0 and 10, it is x
  #10 b=1;      // At 20, make b=1
  #10 a=1;      // at 30, make a=1
  #10 b=0;      // at 40, make b=0
end
```

Here is another example of the `initial` statement usage to generate a periodic signal called clock. It will generate a clock signal of 50% duty cycle with a period of 20 units.

```
reg clock;
parameter half_cycle = 10;
initial
  begin
    clock = 0;
    forever
       begin
         #half_cycle clock = 1;
         #half_cycle clock = 0;
       end
   end
```

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, compares the result, and displays the result as "Test Passed" or "Test Failed"

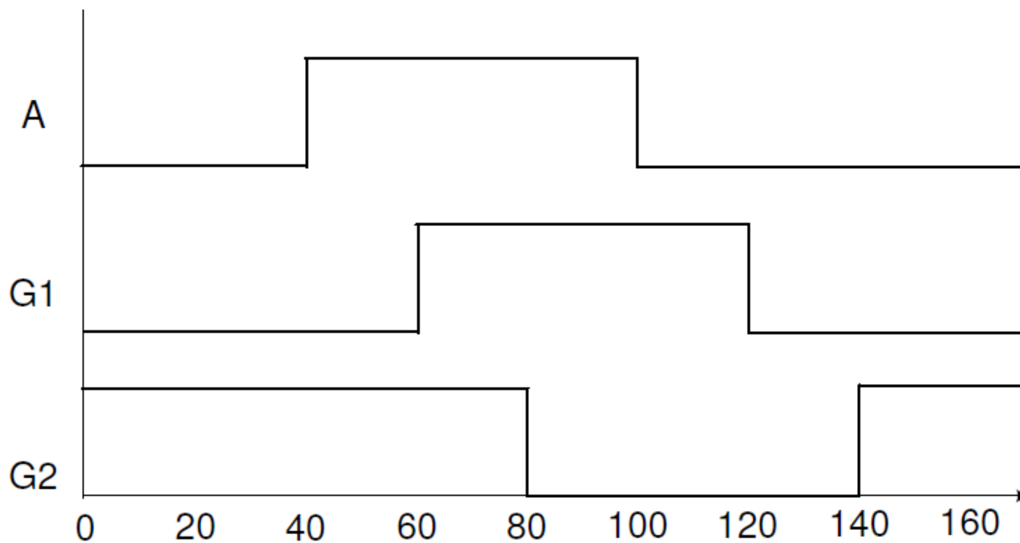
3-1-1. Open Vivado and create a blank project called lab4_3_1**.**

3-1-2. Add the Verilog 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".

© copyright 2013 Xilinx

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_2_1**.**

3-2-2. Create and add the Verilog 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, tasks, and testbenches. You also learned the differences between functions and tasks, both in their definitions and in usage. You saw how function can be used in a testbench to compute expected output.

© copyright 2013 Xilinx