A Verilog HDL Test Bench Primer

Application Note

Table of Contents

Introduction..1
Overview..1
The Device Under Test (D.U.T.)..1
The Test Bench..1
Instantiations..2
Figure 1- DUT Instantiation..2
Reg and Wire Declarations..2
Figure 2 – Reg and Wire Declarations..3
Initial and Always Blocks..3
Figure 3 – An Initial Block Example..3
Figure 4 – An Always Block Example..4
Initialization..4
Delays..4
Clocks and Resets..4
Assign Statements..4
Figure 5- An Assign Example..5
Printing during Simulations..5
$display..5
Figure 6- $display Example..5
$monitor..5
Figure 7- Using $monitor..5
Tasks..6
Figure 8- An Example of a Task – load_count..6
Count16 Simulation Example..6
Table 1- Simulation Steps..6
Figure 9 – The Transcript Window for the Count16 Simulation..8
Figure 10 – The Simulation Waveform Window for the Count16 Simulation..8
Gate Level Simulations..9
Appendix A- The count16.v Verilog Source File..9
Appendix B- The cnt16_tb.v Verilog Test Bench Source File..10
Reference Materials..12

Introduction

As digital systems become more complex, it becomes increasingly important to verify the functionality of a design before implementing it in a system. Hardware Descriptions Languages (HDL's) have become extremely popular because the same language can be used by engineers for both designing and testing CPLD's and FPGA's. The two most common HDL's are Verilog and VHDL. This document focuses on using Verilog HDL to test digital systems, by giving the designer a handful of simulation techniques that can be used on the majority of digital applications.

Overview

This applications note and the included Verilog source code describe how to apply stimulus to a behavioral or gate level description of a CPLD design. The designer should have access to a Verilog simulator and be familiar with its' basic functionality. In short, the Verilog code for each of the individual modules is compiled and the simulation is run. By applying stimulus and simulating the design, the designer can be sure the correct functionality of the design is achieved. This design uses a loadable 4-bit counter and test bench to illustrate the basic elements of a Verilog simulation. The design is instantiated in a test bench, stimulus is applied to the inputs, and the outputs are monitored for the desired results.

The Device Under Test (D.U.T.)

The Device Under Test can be a behavioral or gate level representation of a design. In this example, the DUT is behavioral Verilog code for a 4-bit counter found in Appendix A. This is also known as a Register Transfer Level or RTL description of the design. In the HDL source, all the input and output signals are declared in the port list. There is one always block that is sensitive to the positive edge of the clock and the negative edge of reset. The reset (rst_l) is asynchronous and active low. The count enable (enable_l) and count load (load_l) are both active low and are only checked at the positive edge of a clock. The output enable (oe_l) is also active low. When it's asserted the current count is output on the count_tri bus, otherwise the outputs are tri-stated. This 4-bit counter will be implemented in a Lattice/Vantis CPLD, and its functionality needs to be confirmed before implementation.

The Test Bench

The goal of this design is to implement a loadable 4-bit counter with an asynchronous reset, and count enable, into a Lattice/Vantis CPLD. Before design time is spent synthesizing and fitting the design, the RTL description is simulated to assure correct functionality. Each feature of the design should be tested to ensure that unexpected bugs have not been introduced into the design. This entails testing the specific features designed into the DUT, one at a time, as they would be used in the system. Does the counter reset properly? Will it increment properly and when expected? Does it roll when reaching the maximum count, and can it be loaded with an initial count using load_l? Does the output enable (oe_l) function correctly?

The Verilog test bench module cnt16_tb.v is found in Appendix B. Notice that there are no ports listed in the module. This is because all stimulus applied to the DUT is

generated in this module. The DUT is instantiated into the test bench, and always and initial blocks apply the stimulus to the inputs to the design. The outputs of the design are printed to the screen, and can be captured in a waveform viewer as the simulation runs to monitor the results.

The following sections go into detail on each part of the test bench and it's function.

Instantiations

The test bench applies stimulus to the DUT. To do this the DUT must be instantiated in the test bench, which is the equivalent to placing a component on a schematic. Figure 1 shows how count16 is instantiated in cnt16_tb of Appendix B.

Figure 1- DUT Instantiation

```
//-----------------------------------------------------------
// instantiate the Device Under Test (DUT)
// using named instantiation
count16 U1 (  .count(cnt_out),
              .count_tri(count_tri),
              .clk(clk_50),
              .rst_l(rst_l),
              .load_l(load_l),
              .cnt_in(count_in),
              .enable_l(enable_l),
              .oe_l(oe_l)
              );
//-----------------------------------------------------------
```

Comments in Verilog use a // for one line or a /* and */ when spanning multiple lines, just like C. Notice in Appendix A, the module name of the counter is called count16. So, Count16 is used to place an instance of the counter in the test bench with the instance name U1. The name U1 is arbitrary, and the instance can be given any descriptive name. In between the outer parenthesis are the signals connecting up to U1. The signals with a dot in front of them are the names for the signals inside the count16 module, while the wire or reg they connect to in the test bench is next to the signal in parenthesis. For example, the clock to the counter is called clk in count16, but in the test bench a more descriptive clock name clk_50 is used, which now connects to clk of count16. This allows a signal to be called different names in the test bench and the DUT. This type of instantiation is called “named instantiation” and allows the signals to be listed in any order, or even omitted when a module is instantiated.

Reg and Wire Declarations

There are two signal types in the test bench used to drive and monitor signals during the simulation of the counter. These two types of signals are reg and wire types. The reg data type holds a value until a new value is driven onto it in an initial or always block. The reg type can only be assigned a value in an always or initial block, and is used to apply stimulus to the inputs of the DUT. The wire type is a passive data type that holds a value driven on it by a port, assign statement or reg type. Wires can not be assigned values inside always and initial blocks. The wires declared in cnt16_tb.v of Appendix B are used to hold the values of the outputs of the DUT. Figure 2 is an example of the declaration of reg and wire types in the test bench.

Figure 2 – Reg and Wire Declarations

```
//---------------------------------------------------------
// inputs to the DUT are reg type
   reg clk_50;
   reg rst_l, load_l, enable_l;
   reg [3:0] count_in;
   reg oe_l;

//---------------------------------------------------------
// outputs from the DUT are wire type
   wire  [3:0]  cnt_out;
   wire  [3:0]  count_tri;
```

Initial and Always Blocks

Always and initial blocks are two sequential control blocks that operate on reg types in a Verilog simulation. Each initial and always block executes concurrently in every module at the start of simulation. An example of an initial block is found in Figure 3.

Figure 3 – An Initial Block Example

```
reg clk_50, rst_l;

initial
begin
    $display($time, " << Starting the Simulation >>");
    clk_50 = 1'b0;         // at time 0
    rst_l = 0;             // reset is active
    #20 rst_l = 1'b1;      // at time 20 release reset
end
```

Initial blocks start executing sequentially at simulation time 0. Starting with the first line between the "begin end pair" each line executes from top to bottom until a delay is reached. When a delay is reached, the execution of this block waits until the delay time has passed and then picks up execution again. Each initial and always block executes concurrently, so while this block is stalled between time 0 and 20, other blocks in the design are executing. The initial block in Figure 3, starts by printing << Starting the Simulation >> to the screen, and initializes the reg types clk_50 and rst_l to 0 at time 0. The simulation time wheel then advances to time index 20, and the value on rst_l changes to a 1. This simple block of code initializes the clk_50 and rst_l reg types at the beginning of simulation and causes a reset pulse from low to high for 20 ns in a simulation. An initial block very similar to Figure 3 is found in cnt16_tb of Appendix B.

An example of an always block is found in Figure 4.

Figure 4 – An Always Block Example

```
always
    #10 clk_50 = ~clk_50;    // every ten nanoseconds invert
```

This always block executes every 10 ns starting at time index 0. Hence, the value of clk_50 will invert from the initialized value in Figure 3 every 10ns. This causes a clock pulse to be generated on clk_50 with a period of 20ns or a frequency of 50 Mhz.

The always block in Appendix A does not continuously execute like Figure 4, but instead only executes on a change in the items in the sensitivity list (posedge clk or negedge rst_l). This means that on a "low to high" on the clk signal or a "high to low" of rst_l the always block will execute.

Initialization

When the simulation starts it's important to initialize any reg types in the design to a known value. Signals are undefined at startup, and initialize to the defaults of their data type. Wire types initialize to a Z (high Z) value, and reg types initialize to an X (unknown) value. This is why the initial block in Figure 3 initializes clk_50 to a 0. Without it, the simulator would be inverting an X to another X every 10 ns with the always block in Figure 4. In the same sense, a reset pulse is needed for Appendix A in the count16 module. If the counter was not initialized to a known value before the count was enabled, count = count + 1 would be X's in the simulation.

Delays

At the top of Appendix A and B is a line with a compiler directive `timescale:

```
'timescale 1 ns /  100 ps
```

This line is important in a Verilog simulation, because it sets up the time scale and operating precision for a module. It causes the unit delays to be in nanoseconds (ns) and the precision at which the simulator will round the events down to at 100 ps. This causes a #5 or #1 in a Verilog assignment to be a 5 ns or 1 ns delay respectively. The rounding of the events will be to .1ns or 100 pico seconds.

Clocks and Resets

Two important elements used in almost all simulations are clocks and resets. Using the code segments from Figure 3 and Figure 4 a clock and reset circuit can be incorporated into any test bench. In Appendix B, the test bench cnt16_tb utilizes these circuits to provide a clock and reset to the count16 instance U1. The clock is running at 50 Mhz and the reset pulse is 20 ns long.

Assign Statements

As mentioned in the wire and reg section, a wire can only have a value driven onto it by a port, reg or assign statement. An assign statement, drives a wire type with input from another wire or reg type. An example of a typical assign statement is found in Figure 5, where a slice of a data bus is assigned to a wire called upper byte.

Figure 5- An Assign Example

```
reg [15:0] data_bus;
wire [7:0] upper_byte;

assign upper_byte = data_bus[15:8];
```

With an assign statement, a continuous assignment is made where the value of data_bus[15:8] is constantly driven onto the upper_byte wire. A conditional assignment statement can also be used as in Appendix A, where the value of count is only driving count_tri when oe_l is low, otherwise the output will be high-Z or tri-stated. This is the conventional method for implementing tri-state buffers in a design or test bench.

Printing during Simulations

As a simulation runs, it's important to include a printout to the screen to inform the designer on the status of the current simulation. The value a net or register holds at a certain time in the simulation may be important in debugging a function, so signals can also be printed. Two of the most common commands to print to the screen are described below.

$display

$display is used to print to a line, and enter a carriage return at the end. Variables can also be added to the display, and the format for the variables can be binary using %b, hex using %h, or decimal using %d. Another common element used in $display is $time, which prints the current simulation time. A typical example of how $display is used in a test bench is found in Figure 6.

Figure 6- $display Example

```
$display($time, "<< count = %d - Turning OFF count enable >>",cnt_out);
```

The characters found between the quotes will be printed to the screen followed by a carriage return. The value of cnt_out will replace %d in decimal format, while the current simulation time is printed at the front of the line. At time index 211 the output of this $display is printed to the screen in Figure 9.

$monitor

To monitor specific variables or signals in a simulation every time one of the signals changes value, a $monitor can be used. Only one $monitor can be active at a time in a simulation, but it can prove to be a valuable debugging tool. An example of using $monitor is found in Figure 7.

Figure 7- Using $monitor

```
 initial
     begin
         $monitor($time, " clk_50=%b, rst_l=%b, enable_l=%b, load_l=%b,
count_in=%h, cnt_out=%h, oe_l=%b, count_tri=%h", clk_50, rst_l,
enable_l, load_l, count_in, cnt_out, oe_l, count_tri);
     end
```

The $monitor system task has the same layout as the $display task. Figure 9 has the output of this task during the cnt16_tb simulation.

Tasks

Tasks are a used to group a set of repetitive or related commands that would normally be contained in an initial or always block. A task can have inputs, outputs, and inouts, and can contain timing or delay elements. An example of a task is found in Figure 8.

Figure 8- An Example of a Task – load_count

```
task load_count;
     input [3:0] load_value;
     begin
      @(negedge clk_50);
      $display($time, " << Loading the counter with %h >>", load_value);
      load_l =  1'b0;
      count_in =  load_value;
      @(negedge clk_50);
      load_l =  1'b1;
     end
  endtask //of load_count
```

This task takes one 4-bit input vector, and at the negative edge of the next clk_50, it starts executing. It first prints to the screen, drives load_l low, and drives the count_in of the counter with the load_value passed to the task. At the negative edge of clk_50, the load_l signal is released. The task must be called from an initial or always block, as done in Appendix B. If the simulation was extended and multiple loads were done to the counter, this task could be called multiple times with different load values.

Count16 Simulation Example

The following example requires the use of a Verilog simulator and the Verilog HDL code from Appendix A and B. The HDL code is compiled and the simulation is run. As the simulation executes the $display and $monitor commands will print the results of the simulation out to the screen. Figure 9 shows the results of the cnt16_tb simulation when run on ModelSim 4.7. Signals may be added to a waveform viewer to monitor their activity, and check for the desired results.

In the beginning of the simulation, the time index is set to 0, and all the initial blocks and always blocks will start to execute. Each block will execute concurrently, including the always block in the DUT. Table 1, walks through the steps the simulator takes while executing the simulation.

Table 1- Simulation Steps

- The always block from Figure 4 is used in cnt16_tb to generate a clock for the counter.
- One of the initial blocks in the test bench simply starts the $monitor task as described in Figure 7.
- Another initial block initializes all the variables, and applies the stimulus to the counter. It starts by printing to the screen using a $display. Then the signals clk_50, rst_l, and oe_l are all initialized to 0 at time 0. The enable_l and load_l signals are disabled by driving a 1 to the reg, and count_in is assigned a 0.
- Concurrently, the always block in the count16 module of Appendix A will get a positive edge of the clock, while reset is low, and initialize the counter to 0.

- The next line in the initial block of the test bench has a #20 in front of it. This causes the simulation to wait 20 ns before continuing.
- After 20 ns, the rst_l is released by driving a 1 to the reg.
- Next, the @(negedge clk_50); instructs this initial block to wait until the negative edge of clk_50 before continuing.
- The counter is next loaded with the hex value "A" by using the load_count task of Figure 8.
- At the next negative edge of clk_50 the reg enable_l is asserted by driving a 0.
- At the next positive edge of clk_50, the counter starts incrementing the count.
- The initial block waits for the counter to roll, and the count to increase to 1 by using the "wait" command.
- After the count equals 1 the count is disabled by driving enable_l to 1. This causes the counter to stop counting.
- Next, #40 is used to wait 40 ns before disabling the output enable of the counter by driving oe_l to a 1. This causes count_tri to go to the high impedance state.
- The simulation waits 20 ns before stopping using the $stop system task.

Looking at the Transcript Window results of Figure 9 and the signals in the Waveform Window of Figure 10, the proper function of the counter can be confirmed. The counter resets correctly, can be loaded using load_l, and only counts when enable_l is active. The outputs of the counter are only driven onto count_tri when the output enable oe_l is active.

Using cnt16_tb, the basic functionality of count16 has been simulated. The design can now be synthesized and fit into a Lattice/Vantis CPLD.

Figure 9 – The Transcript Window for the Count16 Simulation

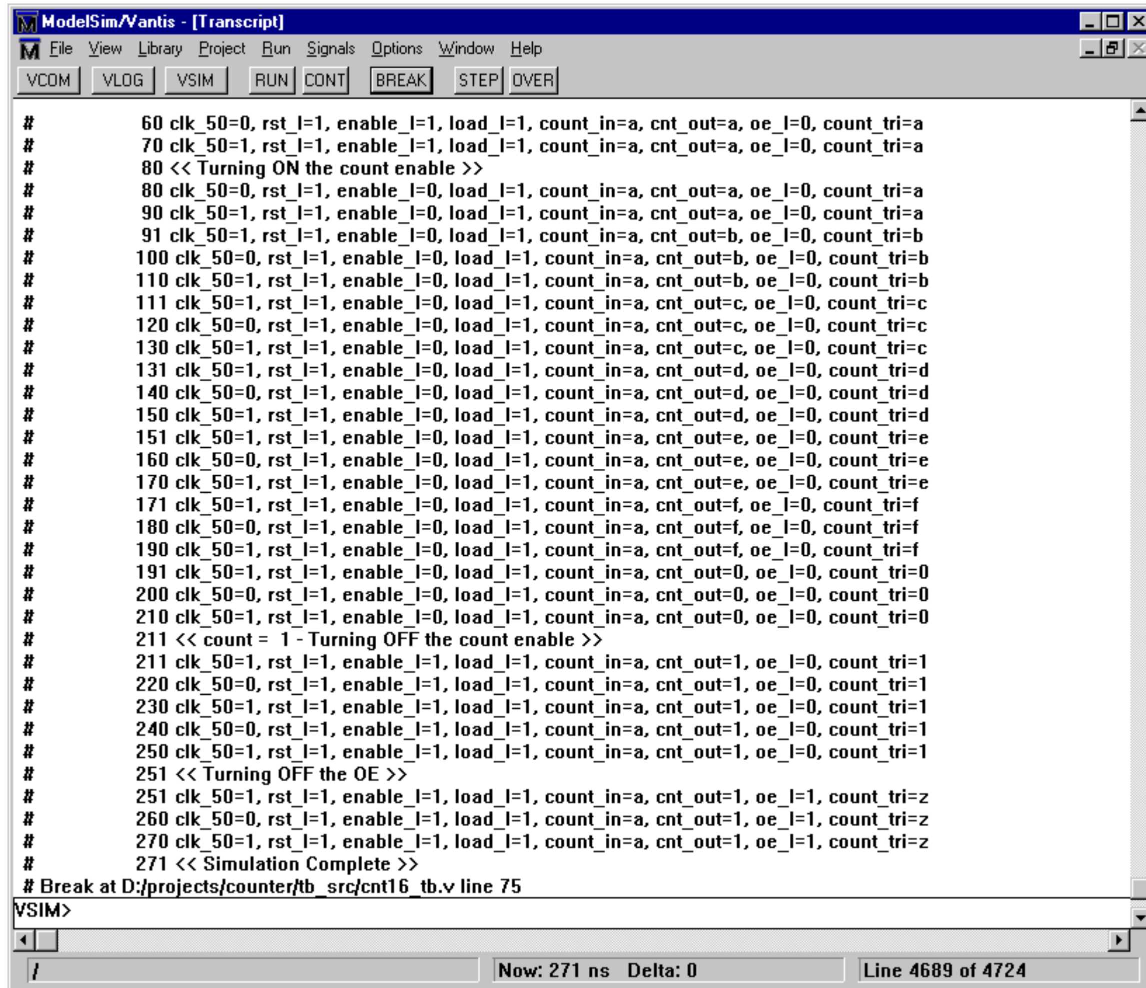
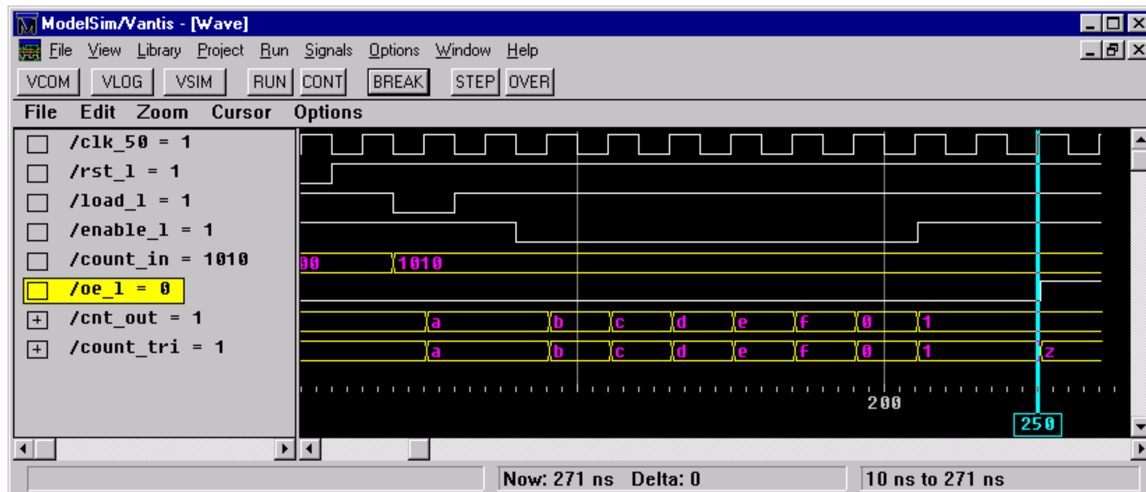


Figure 10 – The Simulation Waveform Window for the Count16 Simulation



Gate Level Simulations

Gate Level Simulations are performed after a design has fit into a Lattice/Vantis CPLD. Design Direct and ispEXPERT both export Structural Verilog and VHDL. They also export an SDF (Standard Delay File) which contains the particular delays for each gate in the design. To perform a gate level simulation, compile the gate level netlist in the simulation instead of the RTL code. Follow the instructions on your particular simulator to back annotate the SDF (Standard Delay File) to obtain gate level delays in the simulation. This will involve using the $sdf_annotate command. For ModelSim fill in the SDF options under VSIM. Then run the simulation as normal.

Appendix A- The count16.v Verilog Source File

```
//-------------------------------------------------
// File: count16.v
// Purpose: Verilog Simulation Example
//-------------------------------------------------
`timescale 1 ns /  100 ps

module count16 (count, count_tri, clk, rst_l, load_l, enable_l, cnt_in,
oe_l);
  output [3:0] count;
  output [3:0] count_tri;
  input clk;
  input rst_l;
  input load_l;
  input enable_l;
  input [3:0] cnt_in;
  input oe_l;

  reg  [3:0] count;

  // tri-state buffers
  assign count_tri = (!oe_l) ? count : 4'bZZZZ;

  // synchronous 4 bit counter
  always @ (posedge clk or negedge rst_l)
    begin
       if (!rst_l) begin
        count <= #1 4'b0000;
      end
      else if (!load_l) begin
        count <= #1 cnt_in;
      end
      else if (!enable_l) begin
          count <= #1 count + 1;
      end
    end

endmodule //of count16
```

Appendix B- The cnt16_tb.v Verilog Test Bench Source File

```
//---------------------------------------------------
// File: cnt16_tb.v
// Purpose: Verilog Simulation Example
//          Test Bench
//-----------------------------------------------------------

`timescale 1 ns /  100 ps

module cnt16_tb ();

  //---------------------------------------------------------
  // inputs to the DUT are reg type
     reg clk_50;
     reg rst_l, load_l, enable_l;
     reg [3:0] count_in;
     reg oe_l;

  //--------------------------------------------------------
  // outputs from the DUT are wire type
     wire  [3:0]  cnt_out;
     wire  [3:0]  count_tri;

  //---------------------------------------------------------
  // instantiate the Device Under Test (DUT)
  // using named instantiation
  count16 U1 ( .count(cnt_out),
                .count_tri(count_tri),
                .clk(clk_50),
                .rst_l(rst_l),
                .load_l(load_l),
                .cnt_in(count_in),
                .enable_l(enable_l),
                .oe_l(oe_l)
               );

  //----------------------------------------------------------
  // create a 50Mhz clock
  always
    #10 clk_50 = ~clk_50;   // every ten nanoseconds invert

  //----------------------------------------------------------
  // initial blocks are sequential and start at time 0

  initial
  begin
    $display($time, " << Starting the Simulation >>");
    clk_50 = 1'b0;
                                   // at time 0
    rst_l = 0;
                                   // reset is active
    enable_l = 1'b1;
                                   // disabled
    load_l = 1'b1;
                                   // disabled
    count_in = 4'h0;
    oe_l = 4'b0;
                                   // enabled
    #20 rst_l = 1'b1;
                                   // at time 20 release reset
    $display($time, " << Coming out of reset >>");
    @(negedge clk_50);             // wait till the negedge of
                                   // clk_50 then continue
    load_count(4'hA);
                                   // call the load_count task
```

```
                                    // and pass 4'hA
    @(negedge clk_50);
    $display($time, " << Turning ON the count enable >>");
    enable_l = 1'b0;
                                    // turn ON enable
                                    // let the simulation run,
                                    // the counter should roll
    wait (cnt_out == 4'b0001);      // wait until the count
                                    // equals 1 then continue
    $display($time, " << count = %d - Turning OFF the count enable >>",
cnt_out);
    enable_l = 1'b1;
    #40;
                                    // let the simulation run for 40ns
                                    // the counter shouldn't count
    $display($time, " << Turning OFF the OE >>");
    oe_l = 1'b1;
                                    // disable OE, the outputs of
                                    // count_tri should go high Z.
    #20;
    $display($time, " << Simulation Complete >>");
    $stop;
                                    // stop the simulation
  end

 //--------------------------------------------------------------
 // This initial block runs concurrently with the other
 // blocks in the design and starts at time 0
  initial begin
      // $monitor will print whenever a signal changes
      // in the design
    $monitor($time, " clk_50=%b, rst_l=%b, enable_l=%b, load_l=%b,
count_in=%h, cnt_out=%h, oe_l=%b, count_tri=%h", clk_50, rst_l,
enable_l, load_l, count_in, cnt_out, oe_l, count_tri);
   end

  //--------------------------------------------------------------
  // The load_count task loads the counter with the value passed

  task load_count;
     input [3:0] load_value;
     begin
      @(negedge clk_50);
      $display($time, " << Loading the counter with %h >>", load_value);
      load_l = 1'b0;
      count_in = load_value;
      @(negedge clk_50);
      load_l = 1'b1;
     end
  endtask //of load_count

endmodule //of cnt16_tb
```

Reference Materials

- *Simulating MACH Designs Using MTI ModelSim and DesignDirect-CPLD*
  Lattice/Vantis – Applications Group
- *The Verilog IEEE 1364-1995 Standard Language Reference Manual*
- *Verilog HDL – A Guide to Digital Design and Synthesis*
  Samir Palnitkar ISBN: 0-13-451675-3
- *The Verilog Hardware Description Language*
  Thomas & Moorby ISBN: 0-7923-9723-1
- *HDL Chip Design*
  Smith ISBN: 0-9651934-3-8

Trademarks

Copyright ©1999 Lattice Semiconductor Corporation. All rights reserved.

Vantis, the Vantis logo, and combinations thereof, DesignDirect, and Performance Analyst are trademarks, and MACH is a registered trademark of Lattice Semiconductor Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.