Creating Testbench

YK

September 10, 2024

Chapter 1

Introduction

In digital design, a testbench is a simulation environment that allows you to verify the functionality of your design before implementing it on hardware. Vivado Testbench is a powerful tool provided by Xilinx for simulating and testing FPGA designs.

The importance of a testbench cannot be overstated. It allows you to catch design errors, validate the functionality of your design, and ensure that it meets the desired specifications. By simulating your design, you can identify and fix issues early in the development process, saving time and effort in the long run.

This document serves as a guide to creating a testbench using Vivado. It covers the general format and rules for writing a testbench, signal generation techniques, timing considerations, and provides useful examples to help you get started.

The topics that will be covered in this document are as follows:

1. Testbench Fundamentals
2. Writing Stimulus for Simulation
3. Design Verification Techniques
4. Advanced Testbench Techniques

Chapter 2

Testbench Fundamentals

This chapter introduces the basic structure and key elements of a testbench, providing a foundation for developing effective and reliable test environments in Vivado. We will explore the essential constructs that make up a testbench, including the use of initial, always, and assign statements to control the flow of simulation, as well as methods for generating clocks and resets. You will also learn how to instantiate your design module within the testbench, which is a crucial step in applying stimuli and observing the module's responses during simulation. By mastering these basics, you will be well-equipped to create testbenches that not only verify the correctness of your design but also enhance your confidence in its performance under real-world conditions.

2.1 Testbench Structure

A testbench is a Verilog module that provides stimulus to the design under test (DUT) or any other name and monitors its outputs. It is written in the same language as the DUT and is used to simulate the behavior of the design in a controlled environment. The testbench is responsible for generating input signals, applying them to the DUT, and checking the output signals to verify that the design is functioning correctly.

Like synthesizable Verilog code, a testbench consists of modules, ports, signals, and procedural blocks. However, the testbench is not intended to be synthesized into hardware. It allows us to use advanced functionality that is not supported by synthesis tools, such as delays, file I/O, and random number generation.

To create a testbench in Vivado, you need to follow these steps:

1. Create a new Verilog file and define a module for the testbench.
2. Instantiate the DUT module within the testbench.
3. Define input and output signals for the DUT.

4. Write stimulus generation code to apply inputs to the DUT.

5. Write code to monitor the DUT outputs and check for correctness.

6. Compile the testbench and run the simulation.

The following example demonstrates a simple testbench structure for a DUT with two input ports (A and B) and one output port (Y).

```
module tb;
    // Define DUT ports
    reg A, B;
    wire Y;

    // Instantiate DUT
    DUT dut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    // Stimulus generation will go here
    // Testbench Contol codes will go here
```

2.2 Initial Blocks

An initial block executes only once, beginning at time zero and proceeding sequentially through the statements contained within it. Initial blocks are particularly useful in testbenches for tasks such as signal initialization, generating input stimuli, and setting up the simulation environment.

```
    initial begin
    clk = 0;
    forever #5 clk = ~clk; // Generates a 100 MHz clock
    #10 data_in = 8'hAA; // Apply first stimulus after 10 time units
    #20 data_in = 8'hFF; // Apply second stimulus after another 20 time units
    #30 data_in = 8'h55; // Apply third stimulus after another 30 time units
end
```

2.2.1 Multiple Initial Blocks

You can have multiple initial blocks in a testbench, each with its own set of statements. This allows you to organize your code into logical sections and control the flow of simulation more effectively. For example, you can use one initial block to initialize signals and another to generate input stimuli.

```
    initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    reset = 1;
    #10 reset = 0;
end

initial begin
    #10 data_in = 8'hAA;
    #20 data_in = 8'hFF;
end
```

2.3 Generating Clocks

If the design has clock inputs, clock generation is a critical aspect of testbench design, as it determines the timing of signal transitions and the overall behavior of the design under test. In digital systems, clocks are used to synchronize the operation of different components and ensure that signals are sampled at the correct time. "forever" loop is used to generate a continuous clock signal with a specified period. The " " operator is used to invert the clock signal, creating a square wave with a 50

```
    initial begin
    clk = 0;
    forever #5 clk = ~clk; // Generates a 100 MHz clock
end
```

#5 specifies a delay of 5 time units, which determines the clock period. You take half of the period to set delay value, which will create 50% duty cycle.

2.3.1 Shifted Clocks

To create a Shifted Clock, delay the second clock in a seperate initial block:

```
reg clk_0;
reg clk_90;

initial begin
    clk_0 = 0;
    forever #5 clk_0 = ~clk_0; // 100 MHz clock with 0-degree phase shift
end

initial begin
```

```
    #2.5; // 90-degree phase shift for a 100 MHz clock
    clk_90 = 0;
    forever #5 clk_90 = ~clk_90; // 100 MHz clock with 90-degree phase shift
end
```

2.3.2 Custom Duty Cycle

To create a clock with a custom duty cycle, you can use a counter to control the clock signal transitions. To create a "forever" block that is not single line statement, you can use "begin" and "end" block will be used:

```
reg clk;

initial begin
    clk = 0;
    forever begin
        #3 clk = 1;  // High for 3 time units (30% of a 10-time unit period)
        #7 clk = 0;  // Low for 7 time units (70% of the period)
    end
end
```

2.4 Generating Resets

Resets are essential for initializing the design and ensuring that it starts in a known state. In a testbench, you can generate reset signals using initial blocks and control their timing to match the design requirements. The following example demonstrates how to generate a reset signal that is active for a specified duration before deasserting.**For sending a reset signal that will comply with both the asynchronous and synchronous resets, you should set the reset signal to 1 for at least your maximum clock period**:

```
    reg reset;

    initial begin
        reset = 1;       // Assert reset
        #5 reset = 0;    // Deassert reset after 5 time units
    end
```

2.5 Simulation Control

You control the overall simulation, including when to start and stop it, and any final checks or logging.

```
    initial begin
    // Apply stimulus
    #100 $finish; // End simulation after 100 time units
```

```
end
```

Chapter 3

Writing Stimulus for Simulation

This chapter explores the various techniques for generating stimulus in a testbench, including using initial blocks, always blocks, and assign statements. You will learn how to create complex input patterns, apply random values, and generate repetitive sequences to test the design under different conditions. By mastering these stimulus generation techniques, you will be able to create comprehensive testbenches that thoroughly exercise your design and ensure its correctness and robustness.

3.1 Static Stimulus

Static stimulus refers to fixed input values that are applied to the design during simulation. These values are typically constants or predefined patterns that are used to verify the basic functionality of the design. Static stimulus is useful for testing specific corner cases, boundary conditions, and known scenarios that are critical for the correct operation of the design.

```
reg [7:0] input_signal;

initial begin
    input_signal = 8'hFF; // Set input_signal to 0xFF
    #10; // Wait for 10 time units
    input_signal = 8'h00; // Change input_signal to 0x00
end
```

3.2 Sequence Stimulus

Sequence stimulus involves applying a series of input values in a specific order to test the design under different conditions. Sequences can be used to simulate

real-world scenarios, complex data patterns, and dynamic behaviors that are difficult to capture with static stimulus alone. By creating sequences of input values, you can thoroughly exercise the design and uncover potential issues that may arise in practical use cases.

"for" loops are used to generate repetitive sequences of input values. The loop iterates over a range of values and applies them to the input signal at each iteration. This allows you to create complex patterns.

```
reg [7:0] input_signal;
integer i;

initial begin
    for (i = 0; i < 256; i = i + 1) begin
        input_signal = i;
        #10; // Wait for 10 time units
    end
end
```

3.3 Random Stimulus

Random stimulus involves applying random input values to the design to test its behavior under unpredictable conditions. Random stimulus is useful for stress testing the design, identifying corner cases, and verifying its robustness against unexpected inputs. By generating random values, you can simulate a wide range of scenarios and ensure that the design can handle any input it may encounter.

Define a seed variable and $random function to generate random values. The seed variable is used to initialize the random number generator, ensuring that the same sequence of random values is generated each time the simulation is run. The $random function returns a random 32-bit value, which can be used to generate random input values for the design.

```
reg [7:0] input_signal;
integer seed;

initial begin
// Seed the random number generator
$random(seed);

// Generate random values
forever begin
    input_signal = $random; // Assign a random value to input_signal
    #10; // Wait for 10 time units
end
```

The 32-bit is automatically truncated to 8-bit by the simulator.

3.4 File-Based Stimulus

File-based stimulus involves reading input values from a file and applying them to the design during simulation. File-based stimulus is useful for testing the design with large datasets, complex patterns, or real-world inputs that are difficult to generate manually. By reading input values from a file, you can simulate a wide range of scenarios and ensure that the design behaves correctly under different conditions.

A txt file like this:

```
8'hFF
8'h00
8'hAA
8'h55
```

```
reg [7:0] data;
integer file;
integer r;

initial begin
    // Open the file
    file = $fopen("input_data.txt", "r");

    if (file == 0) begin
        $display("Error opening file.");
        $finish;
    end

    // Read data from the file
    while (!$feof(file)) begin
        r = $fscanf(file, "%h\n", data);  // Read hexadecimal data
        if (r > 0) begin
            $display("Read data: %h", data);
            // Apply stimulus to the DUT
            // Example: dut_input <= data;
            // Wait for a specified time
        end
    end

    // Close the file
    $fclose(file);
end
```

Chapter 4

Design Verification Techniques

This chapter explores the various techniques for verifying the correctness of the design under test (DUT) in a testbench. You will learn how to compare the DUT outputs with expected values, check for errors and violations, and monitor the design behavior during simulation. By applying these verification techniques, you can ensure that the design meets the desired specifications, performs as expected, and is free from defects that may impact its functionality.

Some important built-in functions in this chapter are:

- $time: Returns the current simulation time.
- $display: Displays a message in the console.
- $monitor: Monitors a signal and displays its value when it changes.
- $assert: Checks a condition and stops the simulation if it is false.
- $fatal: Displays an error message and stops the simulation.
- $finish: Ends the simulation.

4.1 Checking Outputs

Checking outputs involves comparing the DUT outputs with expected values to verify that the design is functioning correctly. By monitoring the output signals during simulation and checking for errors or violations, you can identify issues early in the development process and ensure that the design meets the desired specifications. Checking outputs is an essential part of design verification and helps you gain confidence in the correctness and reliability of the design.

4.1.1 $assert

In an assert block you give both expected and actual value from module output. If the condition is false, else block will be executed and simulation will be stopped if $fatal is used there.

```
initial begin
    #20; // Wait for some time
    assert (out_signal == expected_value) else $fatal("Test failed at time %0t: ...
    ... out_signal = %0h, expected = %0h", $time, out_signal, expected_value);
end
```

4.2 Logging Outputs

Logging outputs involves recording the DUT outputs and other relevant information during simulation for debugging and analysis. By logging the output signals, you can track the design behavior, identify patterns, and diagnose issues that may arise during simulation. Logging outputs is a useful technique for gaining insights into the design performance and ensuring that it meets the desired requirements.

4.2.1 $display

A single report can be generated using $display function. It can be used to display the value of a signal, a message, or a combination of both. The %0t format specifier is used to display the current simulation time, and %0h is used to display the hexadecimal value of a signal.

```
initial begin
    $display("Simulation started at time %0t", $time);
end
```

4.2.2 $monitor

When used with a signal, $monitor will display the signal value whenever it changes automatically.

```
initial begin
    $monitor("Time = %0t, out_signal = %0h, expected_value = %0h", $time, out_signal, ex
end
```

4.2.3 File I/O Logging

You can save the output of the simulation to a file using the $fopen, $fwrite, and $fclose functions. This is useful for capturing large amounts of data, analyzing the design behavior, and generating reports for further analysis.

```
integer logfile;

initial begin
    logfile = $fopen("simulation_log.txt", "w");
    $fwrite(logfile, "Simulation started at time %0t\n", $time);
end

always @(posedge clk) begin
    $fwrite(logfile, "Time = %0t, out_signal = %0h\n", $time, out_signal);
end

final begin
    $fclose(logfile);
end
```

4.3 Using Always Blocks and Combine Techniques

To continuously monitor the DUT outputs and check for errors or violations, you can use always blocks in the testbench. Always blocks are triggered by specific events, such as signal changes or clock edges, and execute continuously during simulation. By using always blocks, you can create real-time monitoring and verification mechanisms that ensure the design is functioning correctly and meets the desired specifications.

For example using always block and file input/output we can give input from file and check output from an expected values file for the given inputs:

```
reg [7:0] data;
reg [7:0] out_data_expected;
integer file;
integer file_out;
integer r;
integer r_out;
reg clk;

/// GENERATE CLOCK SOMEWHERE ELSE


///INITIALIZE FILES FOR INPUT AND OUTPUT
initial begin
    // Open the files
    file = $fopen("input_data.txt", "r");

    if (file == 0) begin
        $display("Error opening file.");
```

```
        $finish;
    end
    file_out = $fopen("output_data.txt", "r");

    if (file_out == 0) begin
        $display("Error opening file.");
        $finish;
    end
end

// Read data from the file and apply it to the DUT each clock cycle
always @(posedge clk)
begin
    // Read data from the file
    if (!$feof(file)) begin
        r = $fscanf(file, "%h\n", data);  // Read hexadecimal data
        if (r > 0) begin
            $display("Read data: %h", data);
            // Apply stimulus to the DUT
            dut_input <= data;
        end
    end
    else begin
        // Close the file
        $fclose(file);
    end
end

// Check the output data against the expected values
always @(posedge clk)
begin
    // Read data from the file
    if (!$feof(file_out)) begin
        r_out = $fscanf(file_out, "%h\n", out_data_expected);  // Read hexadecimal data
        if (r_out > 0) begin
            $display("Read expected data: %h", out_data_expected);
            // Check the output data
            assert (dut_output == out_data_expected) else $fatal("Output mismatch");
        end
    end
    else begin
        // Close the file
        $fclose(file_out);
        $finish;
    end
end
```

4.4 Format Specifiers

Format specifiers are used to control the display format of values in Verilog. They allow you to specify how values should be formatted when displayed using functions like $display, $monitor, and $fwrite. Format specifiers are useful for customizing the appearance of output messages, aligning values, and controlling the number of digits displayed. **Also, when reading file the expected data is determined these formats.** Some common format specifiers are:

4.4.1 For File I/O

- %d: Signed decimal integer Example: %d reads an integer value from the file.

  Usage: $fscanf(file, "%d", var);

- %u: Unsigned decimal integer Example: %u reads an unsigned integer value.

  Usage: $fscanf(file, "%u", var);

- %h: Hexadecimal integer (lowercase letters a-f) Example: %h reads a hexadecimal value.

  Usage: $fscanf(file, "%h", var);

- %H: Hexadecimal integer (uppercase letters A-F) Example: %H reads a hexadecimal value with uppercase letters.

  Usage: $fscanf(file, "%H", var);

- %b: Binary integer Example: %b reads a binary value.

  Usage: $fscanf(file, "%b", var);

- %o: Octal integer Example: %o reads an octal value.

  Usage: $fscanf(file, "%o", var);

4.4.2 For Displaying

- %d: Signed decimal integer

  Usage: $display("Value: %d", var);

- %u: Unsigned decimal integer

  Usage: $display("Value: %u", var);

- %h: Hexadecimal integer (lowercase letters a-f)

  Usage: $display("Value: %h", var);

- %H: Hexadecimal integer (uppercase letters A-F)

  Usage: $display("Value: %H", var);

- %b: Binary integer
  Usage: $display("Value: %b", var);
- %o: Octal integer
  Usage: $display("Value: %o", var);
- %s: String
  Usage: $display("String: %s", str_var);
- %c: Single character
  Usage: $display("Character: %c", char_var);
- %t: Time (current simulation time)
  Usage: $display("Time: %t", $time);

Chapter 5

Advanced Testbench Techniques

This chapter explores advanced testbench techniques that can help you create more sophisticated and effective test environments in Vivado.

5.1 Tasks/Functions

Tasks are similar to the functions in high level programming languages. They are used to group a set of statements into a single block that can be called from other parts of the testbench. Tasks can take input and output arguments, and they can be used to encapsulate complex functionality, improve code readability, and reduce redundancy. Functions are similar to tasks, but they return a value and can be used in expressions. Tasks and functions are useful for creating reusable code blocks, modularizing the testbench, and simplifying the verification process.

Task Example (Generating Clock):

```
//ANSI Style
task generate_clock(output reg clk, input integer period);
    begin
        clk = 0;
        forever #(period/2) clk = ~clk;
    end
endtask

//OR ANOTHER FORMAT (Legacy Usage)
task generate_clock;
    input integer period;
    output reg clk;
    begin
```

```
        clk = 0;
        forever #(period/2) clk = ~clk;
    end
endtask

//TO CALL
initial begin
    generate_clock(clk, 10); // Generate a 10-time unit clock
end
```

Function Example (Can't use delay and @ events in functions):

```
// Function to return the maximum of two 8-bit numbers
function [7:0] max;
    input [7:0] a, b;

    begin
        if (a > b)
            max = a;
        else
            max = b;
    end
endfunction

//TO CALL
reg [7:0] result;
reg [7:0] x, y;

initial begin
    x = 8'hA5;  // 0xA5 = 165 in decimal
    y = 8'h3C;  // 0x3C = 60 in decimal

    result = max(x, y);  // Calling the function

    $display("Max of %h and %h is %h", x, y, result);
end
```

5.2 Asynchronous Calls (Fork-Join)

In Verilog to run multiple tasks concurrently, fork-join block is used. **The fork block is used to start multiple tasks concurrently, and the join block is used to wait for all the tasks to complete** before continuing. Fork-join blocks are useful for running tasks in parallel, improving simulation performance, and modeling asynchronous behavior in the testbench.

```
fork
```

```
   // Block 1
   begin
      // Statements to execute in parallel
   end

   // Block 2
   begin
      // Statements to execute in parallel
   end

   // Additional parallel blocks can be added here
join
```

5.3 Regression Testing

Regression testing is a critical process in design verification that ensures changes or updates to a design do not introduce new bugs or cause existing functionality to break. In the context of Verilog and digital design verification, regression testing involves running a comprehensive suite of testbenches on the design under test (DUT) to validate that all aspects of the design function as expected after modifications.

Different modules are tested using different testbenches. Also, a module can be tested using different testbenches to cover all possible scenarios. This is called regression testing. The following example demonstrates how to create a regression testbench that runs multiple test cases on a DUT and checks the outputs for correctness.

5.3.1 Using Tasks

1. Create a task for each test case
2. Call the tasks in main testbench module

Example:

```
module tb_adder;

    // Declare signals
    reg [3:0] a;
    reg [3:0] b;
    wire [4:0] sum;

    // Instantiate the adder module
    adder uut (
        .a(a),
        .b(b),
```

```
        .sum(sum)
    );

    // Task to run a single test
    task run_test;
        input [3:0] test_a;
        input [3:0] test_b;
        input [4:0] expected_sum;
        begin
            // Apply test inputs
            a = test_a;
            b = test_b;
            #10; // Wait for the output to stabilize

            // Check the result
            if (sum !== expected_sum) begin
                $display("Error: Test failed for inputs %d and %d. Expected %d but got %
            end else begin
                $display("Test passed for inputs %d and %d. Output is %d", test_a, test_
            end
        end
    endtask

    // Task to run all regression tests
    task run_all_tests;
    begin
        // Run test cases
        run_test(4'd1, 4'd1, 5'd2);    // 1 + 1 = 2
        run_test(4'd3, 4'd5, 5'd8);    // 3 + 5 = 8
        run_test(4'd7, 4'd8, 5'd15);   // 7 + 8 = 15
        run_test(4'd15, 4'd15, 5'd30); // 15 + 15 = 30
        // Add more test cases as needed
    end
    endtask

    // Initial block to start the regression testing
    initial begin
        // Run all regression tests
        run_all_tests();
        // End simulation
        $finish;
    end

endmodule
```

5.3.2 Using Shell Script

1. Create a tb files for each test case
2. Include console logs in each testbench (create self-checking testbenches)
3. Create a shell script to run all testbenches

A shell script will catch "PASS" and "FAIL" messages you have written using $display in your testbenches. It will run all testbenches and give a summary at the end.

```
#!/bin/bash

# Example Bash script to run regression tests

# List of testbenches Change this list according to your testbenches
testbenches=("tb_adder" "tb_multiplier" "tb_fsm")

# Loop through each testbench and run the simulation
for tb in "${testbenches[@]}"
do
   echo "Running $tb..."
   # Run the simulation tool (replace with your specific tool command)
   vsim -c -do "run -all; exit;" $tb > logs/${tb}_log.txt

   # Check if the test passed or failed.
   # You can customize this based on your testbench display messages.
   if grep -q "ERROR" logs/${tb}_log.txt; then
      echo "$tb failed. Check log file for details."
      exit 1
   else
      echo "$tb passed."
   fi
done

echo "All tests passed successfully."
```

This example is for ModelSim, you can replace the command with your simulator's command. Also, you can change the log file name and location as you wish.