ESE 382 Digital System Design Using VHDL and PLDs

Spring 20, Ken Short
March 6, 2020 4:26 pm

Laboratory 6: Testbenches for Combinational Designs
This laboratory is to be performed the week starting March 8th.

Prerequisite Reading

1. Chapter 7 of the text.
2. Exam 1 Spring 2020 Exam Problems and Solutions (posted on Blackboard).

Purpose
In this laboratory you will write several testbenches for combinational designs.

When writing a testbench for a combinational design you usually want the testbench to provide exhaustive verification. Exhaustive verification applies every possible input combination. Accordingly, the time required for the simulation increases exponentially with the number of inputs. Fortunately, for most combinational designs an exhaustive verification is easily achievable.

With a non-exhaustive testbench, the goal is to create a comprehensive verification, one that provides a high degree of certainty that the design is correct. In this case, careful thought must be given to what input combinations need to be applied to achieve a comprehensive verification.

Since all possible input values are applied to a combinational design during an exhaustive verification, the system's inputs are often treated as an aggregate and binary values from 0 to 2^n - 1 are applied to the aggregate in a counting order. This may make the waveforms generated a little more difficult to interpret because there may not be a relationship between the order of the application of input combinations and the functionality of the UUT. Another approach is to make the ordering of the application of input values functionally meaningful. That is, each group of input combinations checks a particular aspect of the system's functionality. In addition, giving some thought to the order of the inputs in the aggregate can make it much simpler to interpret and verify the output waveforms in terms of the design's required functionality.

Selfchecking testbenches have the advantage that simulation waveforms do not have to be examined in detail to verify that the UTT is functionally correct. The testbench compares the actual UUT output values to the expected output values for each input combination applied to the UUT. A message is displayed indicating for which input combination(s) the actual output and expected output differ.

A self-checking testbench must know the expected output values for each input combination and compare the UUT's actual output for each input combination against the expected output. Self-checking testbenches differ in how they determine the expected outputs. There are two basic

approaches, either include the expected output values as constants in the testbench (or in a file the testbench reads) or have the testbench compute the expected output for each input combination,

Design Tasks

Design Task 1: Exhaustive Self-Checking Testbench for BCD Code to Aiken Decimal Code Converter.

In Problem 1 of Exam 1, you were asked to design a system, described by a truth table, that converts from BCD code to Aiken code. The system was named `bcd2aiken`. You were asked to write two architectures for the system. Architecture `csop_cpos` used canonical sum-of-products and canonical product-of-sums concurrent signal assignment statements. Architecture `select_arch` used a selected signal assignment statement.

The first approach does not allow the specification of don't care outputs for input combinations that would not occur in an actual application or for which the output value is a don't care. The second approach did allow for specification of don't cares.

Write an exhaustive self-checking testbench that can be used to verify either of these designs. The testbench needs to include a boolean constant named `dont_care`. When this constant is `true`, the testbench includes checking that the UUT actually produces literal don't care output values when appropriate.

When `dont_care` is `false`, the testbench does not check the UUT's output for input combinations where we don't care what the output values are.

So, when using the testbench to functionally verify designs using the first architecture, `dont_care` is defined as `false`. And, when functionally verifying designs using the second architecture, `dont_care` is defined as `true`. For timing simulations, `dont_care` must be defined as `false` for both architectures.

Use table lookup in your testbench to compute the expected output. For each input combination where the converter fails, your testbench must write a message to the console indicating the input combination, the expected output, and the actual output.

Submit your self-checking testbench listing as part of your prelab.

Design Task 2: Exhaustive Self-Checking Testbench for Demultiplexer.

In Problem 2 of Exam 1, you were asked to design a demultiplexer that had a 4-bit vector data input and four 4-bit vector data outputs. Two select inputs determined on which output vector a copy of the input vector data would appear. The other, unselected, output vectors had to be high-impedance. The system was named `demux`.

Write an exhaustive self-checking testbench that can be used to verify this design. Determine whether the testbench needs to include a boolean constant to change its operation for functional simulation and timing simulations. If so, include it in your testbench. For each input combination where the design fails, your testbench must write a message to the console indicating the input combination, the expected output, and the actual output.

Submit your self-checking testbench listing as part of your prelab.

Design Task 3: Exhaustive Non-Self-Checking Testbench for Pattern Match System.
In Problem 4 of Exam 1, you were asked to design a pattern match system.

Write an exhaustive non-self-checking testbench that can be used to verify the pattern match system. Use the design description provided in the exam solutions posted on Blackboard. Since an exhaustive testbench for this design requires the application of 4096 input combinations, careful review of the output waveforms in the waveform editor is tedious. This would be a great application for a self-checking testbench. However, determining a mathematical relationship to compute the expected output is difficult. However, you could try to write a second design description that computes the expected output in a way that is different from the UUT's design description. You could then include the second description in the testbench code or instantiate it as an entity in the testbench code. The testbench compares the UUT's output with that of the second description. The second description does not have to be synthesizable, since it is part of a testbench. Although it would have to use a different style or different algorithm.

Submit your non-self-checking testbench listing as part of your prelab.

Design Task 4: Modified Pattern Match System
Modify the pattern match design description given in the Exam 1 solutions to create a new system named `pattern_match_cnt`. This modified system contains an additional output, named `match_count`. For each input `data` value and `pattern` value, `match_count` specifies the maximum number of matches for that combination. This output, is a 3-bit standard logic vector. However, internally it is computed using an integer variable.

Use the non-self-checking testbench from Task 1 to verify your results.

Submit your design description listing as part of your prelab.

Laboratory Tasks
One student in each group will be required to give an oral explanation to a TA as described in two of the lab tasks and the other student must do so for the other two lab tasks. **Note that it is up to the TA to select which student gives the explanation for each task.**

Lab Task 1: Exhaustive Self-Checking Testbench for BCD Code to Aiken Decimal Code Converter.
Demonstrate the functional simulation of the UUT using your selfchecking testbench. This can be done for either of the architectures specified in the exam. One student in your design team will be asked to explain to the TA the difference between and independence of the way the outputs are computed in the design description and in the self-checking testbench. This person must also explain how the testbench works and how the information displayed in the waveform editor verifies the correct operation of the UUT.

Have the TA evaluate the correctness of the simulation and the clarity of the explanation and provide a signature.

Lab Task 2: Exhaustive Self-Checking Testbench for Demultiplexer.
Demonstrate a functional simulation of the UUT using your selfchecking testbench. One student in your design team will be asked to explain to the TA the difference between and independence in the way the outputs are computed in the design description and in the self-checking testbench. This person must also explain how the testbench works and how it verifies the correct operation of the UUT.

Have the TA evaluate the correctness of the simulation and the clarity of the explanation and provide a signature.

Lab Task 3: Exhaustive Non-Self-Checking Testbench for Pattern Match System.
Demonstrate a functional simulation of the UUT using your non-self-checking testbench. One student in your design team will be asked to explain how the design description works and use the testbench and waveform editor to verity that design is correct.

Have the TA evaluate the correctness of the simulation and the clarity of the explanation and provide a signature.

Lab Task 4: Modified Pattern Match System
Demonstrate a functional simulation of the UUT using your non-selfchecking testbench from Task 3. One student in your design team will be asked to explain to the TA how the design description works and use the testbench and waveform editor to verity that design is correct.

Have the TA evaluate the correctness of the simulation and the clarity of the explanation and provide a signature.

Lab Task 5: Place and Route and Timing for Modified Pattern Match System
Use Synplify Pro to synthesize your design description from Design Task 4 to target an ispMACH4A5-64/32-10JC CPLD. After a successful synthesis, use **HDL Analyst** to view the hierarchical representation of your synthesized system. Use the information from the flattened to gates view to estimate the amount of logic required by your design.

Use ispLEVER Classic to place and route the EDIF file for an ispMACH4A5-64/32-10JC CPLD. Perform a timing simulation of your design targeted to the ispMACH4A5-64/32-10JC CPLD. Use the place and route tool's Chip Report to determine the amount of logic required

Have a TA verify your timing simulation.

Program an ispMACH4A5-64/32-10JC and place it in the test station to verify the functionality of your design.

Have a TA sign off whether your programmed CPLD meets the design specification.

No Questions