Module-2 Training - Lab Manual: Verilog, SystemVerilog and Random Testbenches

This lab manual introduces Verilog, a Hardware Description Language (HDL), and explores the concept of Random Testbenches for efficient verification.

# Verilog

Verilog is a Hardware Description Language used to describe the behavior and structure of digital circuits. It allows engineers to model and simulate digital systems before physical implementation.

## Learning Resources:

* Verilog Video Tutorial: ([Lecture 2a](https://www.youtube.com/watch?v=VjMxf2o8my0&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=6), [Lecture 2b](https://www.youtube.com/watch?v=hgzuxRc7n6M&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=7), [Lecture 2c](https://www.youtube.com/watch?v=nI8tFPhe0uk&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=8), [Lecture 2d,](https://www.youtube.com/watch?v=cufH5IEM98g&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=9) [Lecture 2e](https://www.youtube.com/watch?v=b9BowTQX3Es&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=10))
* Vivado coding Guideline: ([click here](https://www.youtube.com/watch?v=BIqLk23hE90&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=11))
* Vivado Video Tutorial: ([click here](https://www.youtube.com/watch?v=AEVnPWM1ipg&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G&index=12))

## Exercises:

1. **Verilog Syntax:**
   * Learn the basic syntax of Verilog, including modules, ports (inputs/outputs), wires, registers, and data types (logic, reg).
2. **Combinational Logic:**
   * Practice building combinational circuits like adders, multiplexers, and decoders using Verilog operators (and, or, not) and assignment statements.
3. **Sequential Logic:**
   * Understand sequential circuits with flip-flops (registers) and how to model their behavior using always blocks with clock signals.
4. **Finite State Machines (FSMs):**
   * Learn to design FSMs in Verilog using state registers and transition logic to represent control flow in digital systems.

**Tip:** Start with simple Verilog constructs and gradually build complexity as you gain confidence. Utilize online simulators and practice exercises to solidify your understanding.

# System Verilog

System Verilog is a superset of Verilog, offering additional features for complex digital system design and verification. It builds upon Verilog's foundation and provides powerful constructs for modularity, reusability, and advanced verification techniques.

## Learning Resources:

* SystemVerilog Tutorial: [[click here](https://fpgatutorial.com/sv/)]
* SystemVerilog for Verification: [[click here](https://www.chipverify.com/tutorials/systemverilog)]
* SystemVerilog for Verification: [[BOOK](https://pern-my.sharepoint.com/:b:/g/personal/meds_uet_edu_pk/EYkASFcEBUNDthMFB2drfD8Bu7NjoYYPyGLj4LZa6LWw1w?e=Q4qLxs)]

## Exercises:

1. **SystemVerilog Classes and Interfaces:**
   * Learn how to define classes in SystemVerilog to encapsulate design functionality and behavior. Understand interfaces for defining abstract contracts between modules.
2. **Advanced Verification Techniques:**
   * Explore features like assertions (checking design properties), functional coverage (tracking code execution), and Universal Verification Methodology (UVM) for structured testbench development.
3. **Random Verification with SystemVerilog:**
   * Leverage SystemVerilog's advanced features like constrained random variables and distributions for even more powerful random testbench generation.

**Tip:** While SystemVerilog builds upon Verilog knowledge, it introduces new complexities. Start with basic concepts and gradually progress to advanced techniques like UVM. Utilize online resources and practice exercises specifically focused on SystemVerilog.

**Benefits of SystemVerilog:**

* Improved modularity and reusability of code through classes and interfaces.
* Enhanced verification capabilities with assertions, coverage groups, and UVM methodology.
* Increased productivity through object-oriented programming features.

# Random Testbenches

Percentage: 30%

Random testbenches are a powerful technique for verifying digital designs. They generate random stimuli (inputs) to the design under test (DUT) and check for expected outputs, improving the chances of catching corner-case bugs.

## Learning Resources:

* A Beginner's Guide to Random Verification: [[click here](https://verificationguide.com/)]
* Constrained-Random Verification with SystemVerilog: [[click here](https://www.chipverify.com/verification/constraint-random-verification)]
* Self testing Testbenches: [[click here](https://pern-my.sharepoint.com/:b:/g/personal/meds_uet_edu_pk/EZCWVJ1iFRZJnqD5lG2YDxYBnX6RIYGFvZk5GK_um_sv-w?e=4fUwcE)]

## Exercises:

1. **Basic Testbench Structure:**
   * Understand the components of a testbench, including the DUT instance, clock generation, stimulus generation, and output verification logic.
2. **Verilog System Functions:**
   * Learn about Verilog system functions like $random for generating random numbers and $time for time-based control within the testbench.
3. **Constrained Random Verification:**
   * Explore techniques for applying constraints to the random stimuli generated, ensuring it remains within realistic operational ranges for the DUT.
4. **Coverage Analysis:**
   * Learn how to use coverage tools to track which parts of the design have been exercised by the random testbench, helping you identify areas potentially lacking verification.

**Tip:** Begin with simple testbenches generating unconstrained random stimuli. Gradually introduce constraints and explore coverage analysis tools as you progress.

### **Practice Tasks using Verilog and Random Testbenches**

1. **Verilog Combinational Circuit (Unit Testing):**
   * Design a Verilog module for a 4-bit adder that takes two 4-bit unsigned integers as inputs and outputs their sum.
   * Write a testbench using Verilog procedural blocks to exhaustively test all possible input combinations (16 test cases).
   * Verify the adder's functionality by comparing the generated sum with the expected output for each test case.
2. **7-Segment Display Decoder**
   * Implement a Verilog module that converts a 4-bit binary number to a 7-segment display code. The module should take a 4-bit input and output 7 bits representing the segments (a-g) of the display.
   * Write a testbench to test the 7-segment display decoder. Check the output for all 16 possible input combinations and verify that each input correctly lights up the appropriate segments on the display.
3. **16-bit ALU (Arithmetic Logic Unit)**
   * Create a 16-bit ALU in Verilog that supports addition, subtraction, AND, OR, and XOR operations. The ALU should have two 16-bit inputs, a 3-bit control signal to select the operation, and a 16-bit output.
   * Develop a testbench to verify the ALU’s functionality. Test all supported operations with various input values and ensure that the output matches the expected results for each operation.
4. **Parameterized Counter**
   * Design a parameterized counter in SystemVerilog with up, down, and reset controls. The counter should have an input to set the initial value and a parameter to specify the width of the counter.
   * Write a testbench to verify the counter's functionality. Test the up, down, and reset operations with different initial values and ensure the counter behaves as expected.
5. **FIFO (First-In-First-Out) Queue**
   * Implement a FIFO queue in SystemVerilog with parameterized depth. The module should support enqueue and dequeue operations, and indicate whether it is full or empty.
   * Write a testbench to verify the FIFO queue. Test enqueue and dequeue operations with different data sequences, and check the full and empty signals to ensure the queue operates correctly.
6. **Randomized Testbench for ALU**
   * Develop a randomized testbench for a 16-bit ALU in SystemVerilog. The testbench should generate random inputs for the ALU and compare the outputs with expected results using an assertion-based approach.
   * Write a testbench to generate random test vectors for the ALU. Use assertions to automatically check the correctness of the ALU outputs for each operation.
7. **AXI4-Lite Slave Interface**
   * Create a simple AXI4-Lite slave interface in SystemVerilog for reading and writing registers. The interface should support basic read and write transactions. The additional information is [available here](https://pern-my.sharepoint.com/:w:/g/personal/meds_uet_edu_pk/EeJLYaEqsdZCgZW7QDUBaOQBjZV-a42rI9wfFEjFHo7Iyg?e=mH7Aua).
   * Write a testbench to simulate AXI4-Lite transactions. Verify that the slave interface correctly handles read and write operations by comparing the data read from and written to the registers.
8. **Scoreboard for Verification**
   * Create a scoreboard module in SystemVerilog to compare expected and actual results for a DUT (Device Under Test) in a testbench. The scoreboard should track the expected outcomes and raise an error if there is a mismatch. The additional information is [available here](https://pern-my.sharepoint.com/:w:/g/personal/meds_uet_edu_pk/ETxr26LvnrFEmQwQ627dI_sB3-nsPo2MWyEE5qQnrgry6w?e=afCIXS).
   * Write a testbench that uses the scoreboard to verify the functionality of a DUT. Generate test cases, capture the results, and compare them with the expected values using the scoreboard.