Lab 5
Memory Layered Testbench

Module ID : CX-301

Design Verification

Instructor : Dr. Abid Rafique

Version 1.1

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Jan, 2024	Qamar Moavia
1.1	Modified with new exercises	26 Jan, 2024	Qamar Moavia

Table of Contents

Objectives

By the end of this lab, students will be able:

- To use a layered testbench for verifying memory features.
- To construct and integrate transaction classes, generators, drivers, monitors, and scoreboards into a verification environment.
- To implement and manage a wrapper environment that manages the verification process effectively.
- To execute the integrated test environment, analyze outputs, and validate the memory behavior against expected outcomes.
- To enhance troubleshooting skills by resolving errors and refining the verification process based on simulation feedback.

Tools

- SystemVerilog
- Synopsys VCS

Instructions for Lab Tasks

This lab does not require any file, you will be creating all the files from scratch with some help from your previous labs as well. The submission must follow the hierarchy below, with the folder name and the file names exactly as listed below.

```
./Lab5/
      ├── mem_interafce.sv
      ├── transaction.sv
      ├── generator.sv
      ├── driver.sv
      ├── monitor.sv
      ├── scoreboard.sv
      ├── env.sv
      ├── test.sv
      ├── testbench.sv
```

Along with that you also need to upload your solution on the github as well, and share the link.

Task 1: Building the Transaction Class

To create a transaction class encapsulating memory interface signals and methods for display and randomization.

Create the Transaction class

Working in the Task1 directory, create the transaction.sv file as follows:

1. Add memory interface signals as class properties.
2. Create a class constructor to initialize class properties.
3. Implement a display method to print out all properties of a transaction class.
4. Add constraints to control whether the operation is a write or read, ensuring that each transaction is correctly classified and executed as intended.

Task 2: Creating the Generator and Driver classes

Develop a generator to create different transactions and a driver to drive those transactions to the DUT.

Creating Generator Class

Create the generator class in the file generator.sv as follows:

1. Add class properties to the generator class. The properties should include a transaction object, two mailboxes (one for sending transactions to the driver and one for the scoreboard), an int repeat count to specify the number of transactions to be generated by driver, and an event to signal the completion of transaction generation.

   Note: The mailbox between generator and scoreboard is optional.

2. Write the constructor for the generator class. It should initialize the mailboxes and set the repeat count based on an input argument. Also, instantiate the transaction object.

3. Implement the run task as a class method. In this task, generate random transactions up to the specified repeat count using randomize() method. You can control the type of transaction (write or read) using constraints. After randomizing, send different copies of transaction to both the driver and the scoreboard through mailboxes. Once a required number of transactions are generated, trigger the completion event.

Remember to include error handling for the randomize method if the transaction fails to randomize due to conflicting constraints.

Creating Driver Class

Create the driver class in the file driver.sv as follows:

4. Define class properties including a transaction object, a mailbox for incoming transactions, a virtual interface to the DUT, and an integer for tracking the number of transactions received from the generator.

5. Create the driver class constructor to initialize the virtual interface and transaction mailbox.

6. Implement a run task that uses a forever loop to continuously receive transactions from the mailbox and drive them to the DUT via the virtual interface, ensuring that transactions are driven on the inactive edge of the clock. At the end of each iteration, increment the int that stores the number of transactions received.

Task 3: Creating the Monitor and Scoreboard Classes

Develop a monitor to observe the DUT responses and a scoreboard to compare those responses against expected results.

Creating Monitor Class

Create the monitor class in the file monitor.sv as follows:

1. Define class properties for the monitor class. These should include a virtual interface to the DUT and a mailbox to send monitored transactions to the scoreboard.
2. Write the constructor for the monitor class to initialize the virtual interface.
3. Implement the run method. This method should continuously monitor the DUT's output using the virtual interface. For each observed response, package the data into a transaction object and send it to the scoreboard mailbox for verification.

Ensure the monitor captures the DUT responses at the correct times for accurate results, avoiding transient states that could lead to verification errors.

Creating Scoreboard Class

Create the scoreboard class in the file scoreboard.sv as follows:

4. Define class properties for the scoreboard class. The properties should include two mailboxes: one to receive transactions from the generator and one to receive transactions from the monitor. Also add an int for tracking number of transactions received, associative array to model the memory behavior and an int to count the number of errors.

5. Write the constructor for the scoreboard class that initializes both mailboxes.

6. Implement a run task that uses a forever loop to continuously receive transactions from the generator and monitor through corresponding mailboxes. If a write transaction is received from the generator then write corresponding data to the associative array. If a read transaction is received from the monitor then compare the read data with the data stored in associative array at corresponding address, increment error count if the data doesn’t match. At the end of each iteration, increment the int that stores the number of transactions received.

Task 4: Building the Environment and Testbench Program

Develop a wrapper environment class that integrates the generator, driver, monitor, and scoreboard classes, and construct a testbench program to initiate and control the verification process.

Creating Environment Class

Create the environment class in the file env.sv as follows:

1. Add class properties including a virtual memory interface (virtual mem_intf), three mailbox handles for generator to driver (gen2driv), generator to scoreboard(gen2scb) and monitor to scoreboard (mon2scb), instances of generator, driver, monitor, and scoreboard, and an event (gen_ended) to signal the end of transaction generation.
2. Write the constructor, The constructor should take two inputs: virtual interface and repeat count. In this constructor first create instances of mailboxes and then construct objects of generator, driver, monitor and

scoreboard using the virtual interface, repeat count, end of generator event and the mailboxes.

3. Create the test task to start the generator, driver, monitor, and scoreboard using a fork...join_any construct, allowing them to execute in parallel.
4. Create the post_test task to first wait until the generator has finished producing all transactions, and that both the driver and the scoreboard have received the correct number of transactions equal to the repeat count. At the end, use the error count of the scoreboard object to display the pass/failed status of the test.
5. Implement the run method. This method should sequentially execute test(), post_test() and then finish the simulation using $finish.

Creating the Testbench Program

Compose the testbench program in the file random_test.sv as follows:

6. Declare an environment class instance within the test program. Instantiate the environment class, passing it the virtual interface and the repeat count.
7. Within an initial block, simply call the run task of the environment instance.

Running the Simulation

8. Run the simulation. Monitor the process through waveforms and console outputs to observe and verify the DUT's responses.
9. If errors occur, troubleshoot by examining the waveforms and log files. Adjust the testbench components if necessary to resolve any issues.

Good Luck 🙂