System Verilog for Design Verification

Project Manual

10x Engineers (Pvt.) Ltd.

Project Design Verification Environment

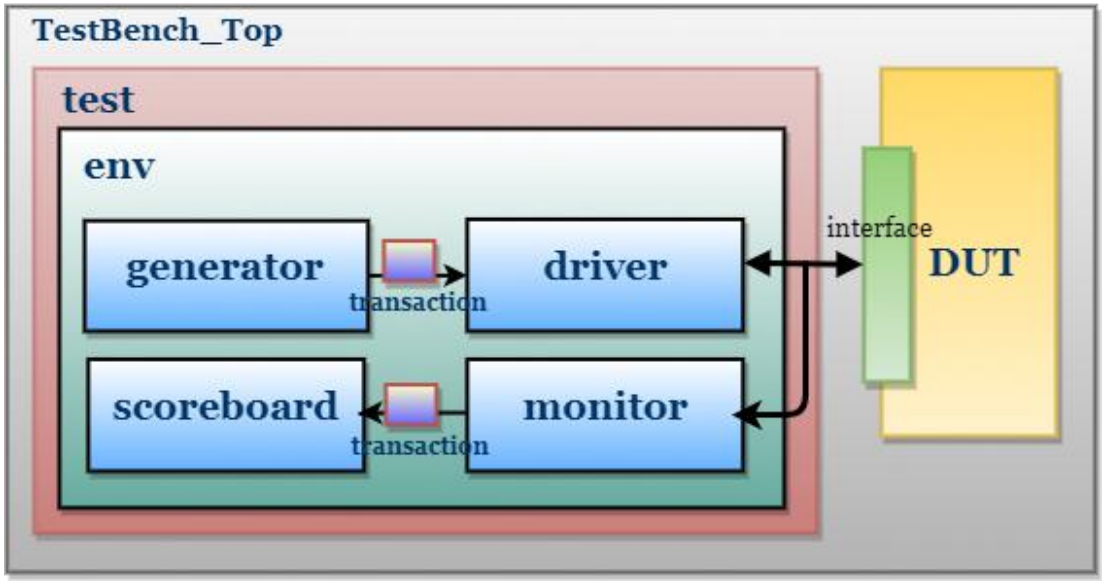
Objective: To create a layered testbench architecture design in System Verilog for AHB-Lite Slave Protocol

---

Design Verification Environment is used to check the functional correctness of the **D**esign **U**nder **T**est (**DUT**) by generating and driving a predefined input sequence to a design, capturing the design output and comparing with-respect-to expected output.

Verification environment is a group of classes or components. where each component is performing a specific operation. i.e, generating stimulus, driving, monitoring, etc. and those classes will be named based on the operation.

Architecture



Following are the key components of a design verification environment:

Transaction

- The Transaction class is used as a way of communication between Generator-Driver and Monitor-Scoreboard. Fields/Signals required to generate the stimulus are declared in this class.

Interface

- It contains design signals that can be driven or monitored.

Generator

- Generates the stimulus (create and randomize the transaction class) and send it to Driver

Driver

- Receives the stimulus (transaction) from a generator and drives the packet level data inside the transaction into the DUT through the interface.

Monitor

- Observes the activity on interface signals and converts into packet level data which is sent to the scoreboard.

Scoreboard

- Receives data items from monitors and compares them with expected values. Expected values can be either golden reference values or generated from the reference model.

Environment

- The environment is a container class for grouping all components like generator, driver, monitor and scoreboard.

Test

- The test is responsible for creating the environment and initiating the stimulus driving.

Test bench Top

- This is the topmost file, which connects the DUT and Test. It consists of DUT, Test and interface instances. The interface connects the DUT and Test.

Deliverable

A word document containing answers to the questions mentioned below in section “Modify the test bench”.

A test plan containing possible test cases to verify the features of the DUT.

Link of a project with fully functional verification environment. Your environment should come with some smoke tests / sanity checks. The environment should be capable of running user defined sequences for test cases. The environment

should give the user capability of defining and running their own sequences.

Important note:

Please note that you will be presenting your test bench to SME in the training room, so make sure you understand all aspects of your verification environment and are able to answer to all of the questions given above in a satisfactory manner.

Creating a Design Verification Environment

Work in the *proj-ahblite* directory. Design file (design.v) along with the constant definitions(amba_ahb_defines.v) and DVE files are already present in the directory.

Design (DUT)

1. Design under test a memory based on AHB-Lite Memory interface. Please read through the specification of the memory interface before implementing the verification environment.

   Spec:https://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf

Interface

2. Add design signals in the interface (interface.sv) along with modports and clocking blocks for synchronization.

Transaction

3. Add signals in the transaction class (transaction.sv) along with constraints.
4. Add the following constraints:
   a. Single burst, 4-beat wrapping burst and 4-beat increment burst
   b. Address aligned w.r.t. Size
   c. Protection control for *Data Access* only
   d. Transfer sizes of byte, half word and word only
5. Add a *print_trans* method to print the transaction item values for debug purposes.

Generator

6. Declare a transaction class, a mailbox (to send transaction packets to the driver), an event (which will be triggered when all the packets are sent to the driver) and a *main* method in the generator class (generator.sv).

7. In the *main* method, randomize the transaction packet and put it into the mailbox. method should fail if randomization fails.

8. Trigger the event at the end of the task.

Driver

9. Create a virtual interface handle and a mailbox (to receive the transaction packets from the generator) in the driver class (driver.sv).

10. Create a *reset* method to reset the interface signals to default/initial values.

11. Create a *drive* method to drive the transaction items to interface signals.

12. Create a *main* method to wait until the reset comes and then call the *drive* method.

Monitor

13. Create a virtual interface handle and a mailbox (to send the transaction packets to scoreboard) in the monitor class (monitor.sv).

14. Create a *main* method to sample the interface signals and put the sampled transaction packet in the mailbox.

Scoreboard

15. Create a mailbox (to receive the transaction packets from monitor) and a local memory to save the stored data (a copy of the data written to the DUT memory) in the scoreboard class (scoreboard.sv).

16. Create a *main* method to sample the transaction packet from the monitor. For a write operation, store the data in the local memory and for a read operation compare the data coming from the monitor with the already stored data in the local memory of the scoreboard. Generate an error if the comparison fails.

Environment

17. Create handles for all 4 components of the environment i.e. Generator, Driver, Monitor and Scoreboard in the environment class (environment.sv).

18. Create mailbox handles for transactions between Generator to Driver and Monitor to Scoreboard.

19. Create a virtual interface handle and an event for synchronization between generator and test.

20. Initialize all the components in the constructor.

21. Add a *pre_test* method for reset.

22. Add a *test* method to run the *main* tasks/functions of all components.

23. Add a *post_test* method to check for the triggered event.

24. Add a *run* method to combine above three methods in order.

25. Call *$finish* after *post_test* to end the simulation.

Test

26. Create an environment handle in the test program (test.sv).

27. Instantiate the environment and run the stimulus.

Testbench top

28. Declare and generate the clock and reset signals.

29. Instantiate the interface in order to connect DUT and test case.

30. Instantiate the test case and pass the interface handle.

31. Instantiate the DUT and connect to the interface ports.

Modify the test bench

Transaction

- Can I run predefined sequences? (e.g. reset sequence, random write sequence, random read/write bursts, a directed test). Can I debug easily if my test fails? Do I need one or multiple transactions for bursts?

Generator

- How to control how many transactions get generated? Sometimes random transactions are not needed. How do I generate non-random transactions when required?

Driver

- Are the interface signals driven according to the spec? Does the transaction have proper address/data Phases? Do I need to sample inputs to decide whether to drive outputs or not on the next clocking event?

Monitor

- Are the interface signals sampled according to the spec? Does the transaction have proper address/data Phases?

Scoreboard

- Does the scoreboard implement proper endianness? How to change endianness if required? How to not compare reset values and to compare only those memory locations which have already been written? Should scoreboard memory be static or dynamic?

Bonus tasks

Update the design verification components to support *n* number of transactions.

Add a test case for alternate read/write transactions (for same address) using the *pre_randomize* method.

Add a test case for all read transactions using *pre_randomize* method for checking reset state of design memory.

Notes

Don’t forget to include sv files w.r.t to hierarchy.

Make sure the reset state of design memory and scoreboard memory are the same.

Make sure all the transactions are completed before the test finishes.