Functional Verification Project 2023

Introduction & how to:

This project is meant to test the understanding of the notions presented during at the functional verification course. The grading of this assignment will be divided in 2:

- midterm (last week before X-MAS holiday): 20p
- final (last week before exam session): 30p

In order to pass this project a minimum of 25 points are required.

The development of this project will be done on the fallowing server: ***sonic.dcae.pub.ro*** . Please choose one of the available student users and stick to it. (ex: user:student1 pass:student1)
For developing this assignment a script to run the simulator is provided but some changes may be required if the student decides to modify the directory structure/directory name/ testbench name.

For midterm evaluation all students should be able to present an environment capable of driving a transaction without any monitors (task 17 from task list)

Template description

The aces_fv_project contains the fallowing directories and files:

- DOC:
  - ref_cards: contains auxiliary materials such as SystemVerilog Reference Manual, UVM Reference Manual, Linux cheat sheet.
  - Uarc: contains module specification and a copy of this document
- SIM:
  - sv_sim: contains the execution script for simulator
- SV:
  - transmitter_tb:
    - include:
      - env: contains verification environment source files.
        - Env_pkg.sv: verification env package file
        - Transmitter_agent.svh: source file for the uvc agent
        - Transmitter_driver.svh: source file for the drive used by the uvc agent.
        - Transmitter_in_mon.svh: source file for the input monitor used by the uvc agent
        - Transmitter_out_mon.svh source file for the output monitor used by the uvc agent
        - Transmitter_model.svh: source file for the model class
        - Transmitter_scoreboard.svh: source file for the scoreboard model
        - Transmitter_seq_item.svh: source file containing the sequence item for the input sequence
        - Transmitter_sequence.svh: source file containing all the sequences available for use in tests.
        - Uart_seq_item.svh: source file for the collected output item
      - Tests:
        - Transmitter_test_pkg.sv: package file containing all the tests that can be run by the env
    - Tb:
      - Sys_if.sv: interface for all the system signals
      - Transmitter_tb: testbench

Tasks:

1. After studying the module specification, write a verification plan which contains the functionality you want to verify and the way to approach verification (assertions, checkers, coverage, testcases).
2. Implement a clock generator. The implementation can be done either inside the testbench or as a separate module instantiated by the testbench.
3. Instantiate the DUT without assigning anything to its ports.
4. Implement a task capable of driving reset upon request. The reset should be asynchronous and its length should be configurable at call time.
5. Iniside the sys_itf declare the necessary wires and then at testbench level declare a variable of type sys_itf and connect the interface signals to their corresponding DUT ports. The inputs of the DUT should be controlled from the interface while the outputs should be observable from the interface.
6. Set the interface previously declared to the uvm_config_db
7. At testbench level include and import the test package and the interface.
8. Make the env package available at test level (hint inside the test_pkg)
9. Write a simple hello world test to check that all the connections were properly made (make sure no signal has X, the clock is ticking, etc.). Do not forget to include the test to the test package so that it can be compiled.
10. Inside the provided test_base.svh, declare a variable of the interface type and connect it to the interface previously set inside the uvm_config_db. After connecting the interface instantiate the verification env. Pay attention to the env_pkg and the test_pkg.
11. Inside the env declare and instantiate an object of transmitter_agent type.
12. Inside the agent class declare and instantiate variables for the driver, input monitor and output monitor.
13. Extend the previous hello world test to be a child of test base and make it run for enough time as to see the how the env is built. Add messages so that at run time you can see how the phases progress for each component. (build -> connect -> run)
14. Implement the required field (of appropriate size) for transmitter_seq_item so that a successful transaction can take place on the DUT input bus. Make this fields randomizable.
15. Inside the env_pkg declare a sequencer capable of accepting objects of type transmitter_seq_item. Next, declare and instantiate the sequencer at agent level.
16. Implement the driver run_phase so that it collects items sent from driver and triggers the drive_transfer function. The run_phase should kill any ongoing transfer if a reset is issued, wait for reset to end and then wait for a new valid transer. When a reset is issued the reset_signals function should be called and reset values should be put on bus. Do not forget to connect the sequence_port to the sequencer.

17. Extend the previous hello world test and instantiate an object of type transmitter_seq. The test should start 1 transaction, wait for it to complete, and then finish. Add signals to the waveform and check that the transactions occur.
18. Implement the required field (of appropriate size) for uart_seq_item so that a successful transaction can be recovered from the DUT output bus.
19. Implement the input_monitor run_phase so that it collects transactions from bus. The collection of data should be done by calling the task recover data and should only take place if no reset is active on bus. If a reset is active on the bus, collection is stopped, collected values are discarded and a new "collect" call happens only after reset signals goes inactive. The recover data task should display its collected item upon completion.
20. Repeat task 19 for the output monitor. Inside the monitor implement a function that can automatically determine the length of a baud and wait it out.
21. Rerun the test from task 17 and using the waveform and console outputs determine if the monitors work properly.
22. Inside the scoreboard class declare and instantiate an object of type transmitter_model and an object of type transmitter_coverage. Attention, transmitter coverage class isn't provided with the template.
23. Inside the transmitter_env class connect the available analysis ports to the analysis ports available inside the monitors. For the write_input_bus_ap model the way input data gets inside the internal FIFO. The write_tx_bus_ap should model the way data gets out of FIFO.
24. Implement the envisioned cover groups and checkers as you have detailed in the verification plan. Add new testcases as required and stated inside the verification plan.

Disclaimer:

The above task list is meant only as a guideline and may require additional steps.

The usage of the provided template isn't mandatory.