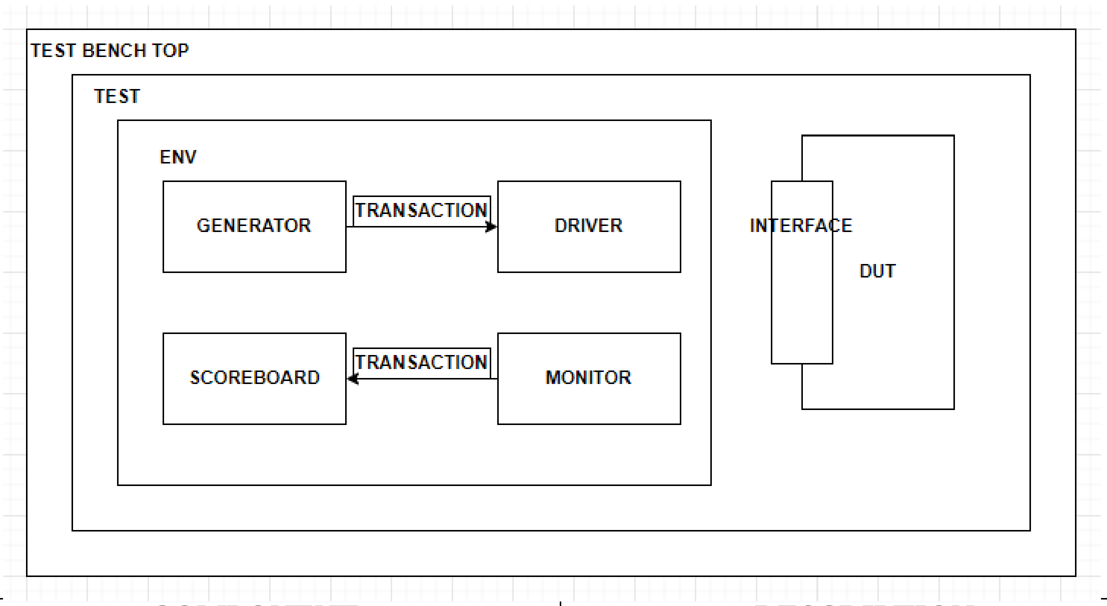
Class Based Testbench Hierarchy Overview

Verification Strategy:

Functional Verification of Asynchronous FIFO is a tedious task as two different clock domains are involved namely the write clock and the read clock. This milestone and the next milestone deal with us as a team coming with a system Verilog based OOP verification environment and provide deterministic testcases. We have come up with the idea of using Gray box verification, gray box verification typically means input signals are triggered accordingly to check the functional output of DUT and use it for comparison with the reference model.

Components of System Verilog Test Environment:



COMPONENT	DESCRIPTION
Generator	Generates different input stimulus to be drive to DUT (Design Under Test).
Interface	Contains Design Signals that can be driven or monitored
Driver	Drives the generated stimulus to the design.
Monitor	Monitor the design input-output ports to capture design activity.
Scoreboard	Checks output from the design with expected behavior.
Environment	Contains all the verification components mentioned below.
Test	Contains the environment that can be tweaked with different configuration settings.

Table:1 Test Bench Environment and Components (Source: Chip Verify)

- **Interface** – If the design contains hundred of port signals it would be difficult to connect, maintain and re-use those signals. Interface contains all design input-output ports into a container and the design can be driven with values through this interface.
- **Driver –** Driver as name suggests drives the values to the DUT through a pre-defined task in the driver class. This is the level of abstraction required to make test benches more flexible and scalable.
- **Scoreboard –** Scoreboard can have a reference model that behaves the same way as DUT, this model reflects the expected behavior of the DUT. Input sent to the DUT is also sent to the reference model.
- **Environment –** It makes the verification more flexible and scalable because more components can be plugged into the same environment for a future project. Environment class encapsulates major components of testbench – generator, driver, scoreboard, monitor.

Test Top: The test will instantiate an object of the environment and configure the way the test wants to. Testbench top module orchestrates the test environment, manages clock generation (using clocking blocks), reset generation (through driver class), interface connectivity. All the above-mentioned components collectively manage data-exchange, randomize input generations using randomize methods on the inputs which have been declared as rand or randc (here in Asynchronous FIFO, input data is randomized), transfer data and result analysis.

Required Tools: Siemens EDA Questa tool is used for OOP System Verilog simulation environment and debugging of Asynchronous FIFO, another tool used by our team is edaplayground.

Functions and Test Case Suites:

TEST CASES	DESCRIPTIONS
FIFO Reset	Reset task in driver class checks the reset condition to see if the FIFO has arrived to a known state.
FIFO Full	Drive the FIFO to completely fill the depth and check if the full flag is triggered or not.
FIFO Empty	Check to see if the empty flag is triggered when there is no data being driven to the FIFO and check if the empty flag is triggered when whole of the FIFO is being read.

Currently, out System Verilog Verification environment works fine for 7 write and read transactions and failing for transaction 8 and above, this environment creation is a ongoing process for milestone3.

Transcript1 – Shows the transactions for 5 reads and writes

Transcript2 – shows transdactions for 8 reads and writes.

Functional Coverage:

Functional coverage assesses whether the design functionality meets specified requirements. For our asynchronous FIFO design, functional coverage was evaluated by verifying the correct operation of read and write operations, as well as the behavior of winc and read_inc signals. Through rigorous testing with 1000 samples, the functional coverage achieved a commendable 75%.

Code Coverage:

Code coverage measures the extent to which the design code has been exercised during testing. Our assessment considered the lines of code executed during read and write operations, including the utilization of write_inc and read_inc signals. With thorough testing using 1000 samples, the code coverage attained a satisfactory level of 59%.

Testing Approach:

Functional Testing: Tested the functionality of read and write operations, ensuring correct data transfer and signal behavior.

Code Testing: Executed read and write operations with varying scenarios to exercise different parts of the code, including write_inc and read_inc signals.

Results and Evaluation:

Functional Coverage: Achieved 69% functional coverage by verifying the correct operation of read and write operations, as well as the behavior of write_inc and read_inc signals.

Code Coverage: Attained 59% code coverage by exercising different parts of the code during read and write operations with 5000 samples.

The code coverage is 69% because the testing focused on a wide range of functionalities, including read and write operations, as well as the behavior of additional signals such as write_inc and read_inc. However, it's likely that some corner cases or less frequently occurring scenarios were not fully covered in the testing process, leading to partial coverage.

While test cases were designed to cover various operational scenarios, there might have been limitations in the breadth and depth of the test cases. Certain edge cases or boundary conditions may not have been adequately addressed, contributing to the incomplete coverage.

The design of asynchronous FIFOs can involve intricate logic and multiple pathways within the code. Achieving 59% code coverage becomes increasingly challenging as the complexity of the codebase grows. Some sections of the code may have been less accessible or difficult to exercise fully during testing, resulting in partial code coverage.

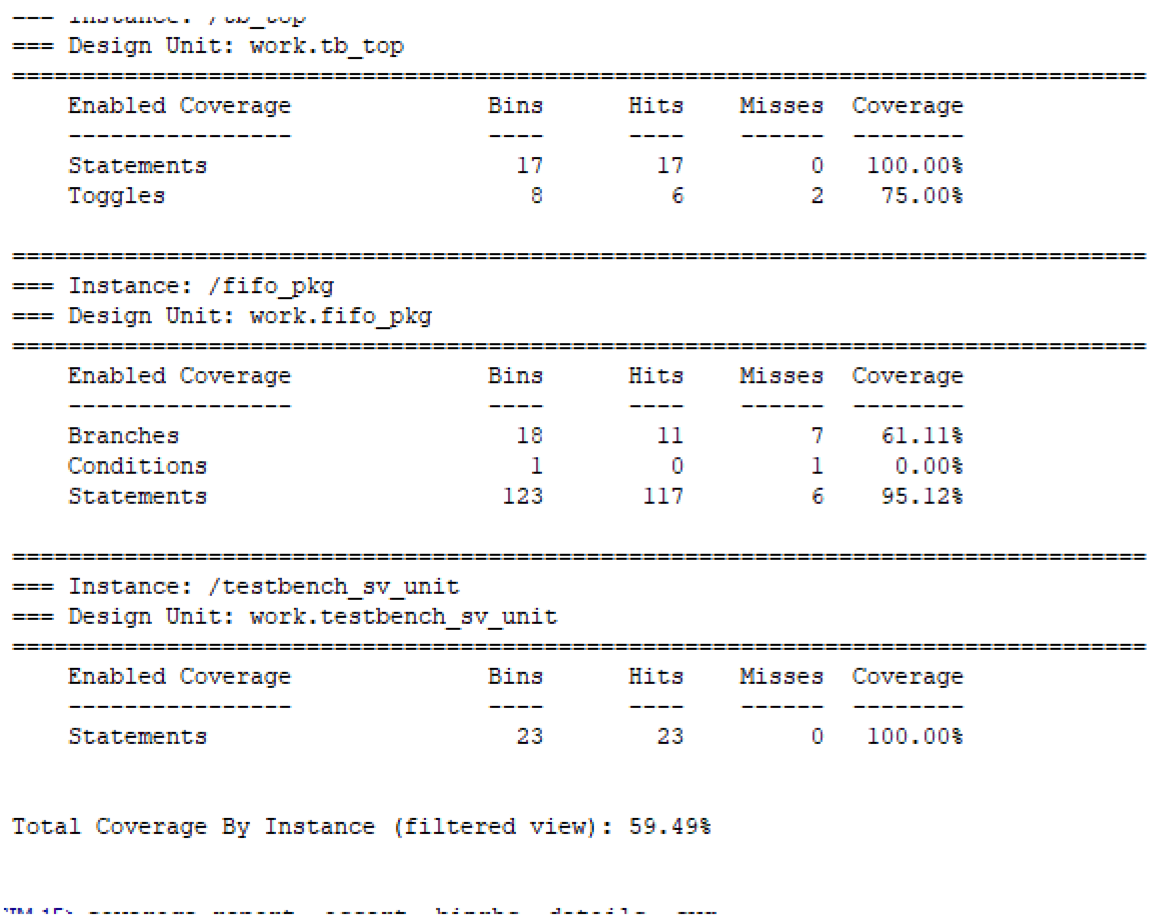
Statement Coverage: This metric measures the percentage of executable statements that have been executed during testing. It indicates how much of the code has been traversed at least once. Each line of code is considered a statement, and statement coverage ensures that every line has been executed at least once.

Branch Coverage: Branch coverage evaluates the decision points or branches in the code, such as if statements, switch statements, and loops. It measures the percentage of decision points

that have been exercised during testing. This metric ensures that both true and false branches of conditional statements are tested.

Toggle Coverage: Toggle coverage measures the utilization of flip-flops or toggles within the design. It evaluates whether each flip-flop has been toggled (changed its state) at least once during testing. Toggle coverage is particularly relevant in digital designs to ensure that all flip-flops are functioning as expected and that there are no unused or redundant elements in the design.

Code coverage :



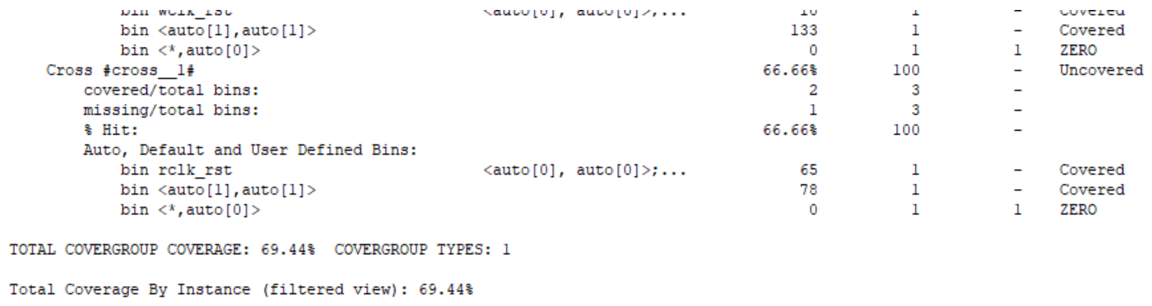
```
=== Instance: /tb_top
=== Design Unit: work.tb_top
=====================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      17        17         0   100.00%
    Toggles                          8         6         2    75.00%

=====================================================================================
=== Instance: /fifo_pkg
=== Design Unit: work.fifo_pkg
=====================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        18        11         7    61.11%
    Conditions                       1         0         1     0.00%
    Statements                     123       117         6    95.12%

=====================================================================================
=== Instance: /testbench_sv_unit
=== Design Unit: work.testbench_sv_unit
=====================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                      23        23         0   100.00%


Total Coverage By Instance (filtered view): 59.49%
```

Functional Coverage:



```
            bin wclk_rst                  <auto[0], auto[0]>;...        10          1          -    Covered
            bin <auto[1],auto[1]>                                      133          1          -    Covered
            bin <*,auto[0]>                                              0          1          1    ZERO
    Cross #cross__1#                                                66.66%        100          -    Uncovered
        covered/total bins:                                              2          3          -
        missing/total bins:                                              1          3          -
        % Hit:                                                      66.66%        100          -
        Auto, Default and User Defined Bins:
            bin rclk_rst                  <auto[0], auto[0]>;...        65          1          -    Covered
            bin <auto[1],auto[1]>                                       78          1          -    Covered
            bin <*,auto[0]>                                              0          1          1    ZERO

TOTAL COVERGROUP COVERAGE: 69.44%  COVERGROUP TYPES: 1

Total Coverage By Instance (filtered view): 69.44%
```

Resources:

- Professor Venkatesh Patil slides
- Chipverify (https://www.chipverify.com/systemverilog/systemverilog-simple-testbench)
- Verification Academy (https://verificationacademy.com/forums/t/verification-asynchronous-fifo-cummings/41199)