VERIFICATION TEST PLAN

For

Asynchronous FIFO

Fundamentals of Pre-Silicon Validation  
Winter -2024

By

**Abram Fouts & Yunus Syed**

**ECE-593: Fundamentals of Pre-Silicon Validation – Venkatesh Patil**

**February 1, 2024**

[**Introduction: 4**](#_gjdgxs)

[1.1 Objective 4](#_30j0zll)

[1.2 Top Level block diagram 4](#)

[1.3 Specifications for the design 4](#_prc6ghfmls6f)

[1.3.1 Producer Specification 4](#_3znysh7)

[**2 Verification Requirements 4**](#_2et92p0)

[2.1 Verification Levels 4](#_tyjcwt)

[2.1.1 Hierarchy 4](#_3dy6vkm)

[2.1.2 Control and Observation 5](#_1t3h5sf)

[2.1.3 Comparison Tagging 5](#_4d34og8)

[**3 Required Tools 5**](#_2s8eyo1)

[3.1 Simulation Software 5](#_17dp8vu)

[3.2 Directory structure 5](#_3rdcrjn)

[**4 Risks and Dependencies 8**](#_26in1rg)

[**5 Functions to be Verified. 8**](#_35nkun2)

[5.1 Write and Read Operation 8](#_if6piysvpc58)

[5.2 Empty and Full Status Handling 8](#_fhz69akoax9v)

[5.3 Idle Cycle Handling 8](#_rv3zq83ntz7)

[5.4 Burst Mode 9](#_u88tdhrmrlsi)

[5.5 Duty Cycle Handling 9](#_8xtv75xyay7p)

[**6 Tests and Methods 9**](#_3j2qqm3)

[6.1 Method 9](#_6wntpmg7gbne)

[6.2 Tests 9](#_jxjrrlfxk40)

[6.2.1 FIFO Full Test 9](#_iabw3evl9zks)

[6.2.2 FIFO Empty Test 9](#_uw31jp75hve)

[6.2.3 Reset Test 9](#_dhd0a12am7wx)

[6.2.4 Back To Back Operations 10](#_yqurcwcnrf94)

[6.3 Advantages and Disadvantages 10](#_4i7ojhp)

[6.4 Architecture 10](#_2xcytpi)

[6.5 Strategy 10](#_1ci93xb)

[6.6 Driving methodology 10](#_3whwml4)

[6.7 Checking methodology 10](#_2bn6wsx)

[6.8 Detailed Test Descriptions 11](#_qsh70q)

[**7 UVM 11**](#_5jezp5z8xxbu)

[7.1 Overview 11](#_dwzaiot8ah0a)

[7.2 The UVM Testbench Architecture 11](#_mb23qslm7gpw)

[7.3 UVM Testbench 12](#_zf0i1tzf9ure)

[7.4 UVM Test 13](#_f4q0u8ea1pt9)

[7.5 UVM Environment 13](#_2l84r9wawocl)

[7.6 UVM Scoreboard 13](#_68inojaf25y4)

[7.7 UVM Agent 13](#_n8khxcan5rr)

[7.8 UVM Sequence Item 14](#_mwr0m9r2kg7d)

[7.9 UVM Sequence 15](#_bgai0jyboetu)

[7.10 UVM Sequencer 15](#_etyh15icxswq)

[7.11 UVM Driver 16](#_i96sukl1jxwv)

[7.12 UVM Monitor 16](#_191ubwxw54hp)

[**8 Coverage Requirements 17**](#_3as4poj)

[8.1 Functional Coverage 17](#_qxl5m8wz6w7s)

[8.2 Code Coverage 17](#_2tjtxwjvbvnh)

[8.3 Cover Groups 17](#_gf2ml39295k4)

[8.3.1 Input Data 17](#_8unk7ilf4z36)

[8.3.2 Output Data 18](#_ecs0j1a8dnws)

[8.3.3 Control Signals 18](#_lwrbf29x2aeq)

[**9 Contributions & Expertise 18**](#_49x2ik5)

[9.1 Abram Fouts Expertise 18](#_2p2csry)

[9.2 Abram Fouts Contributions 18](#_st1v634z5rq8)

[9.3 Yunus Syed 19](#_pcb3ycn87ghd)

[**10 Schedule 19**](#_147n2zr)

[**11 References Uses / Citations/Acknowledgements 19**](#_23ckvvd)

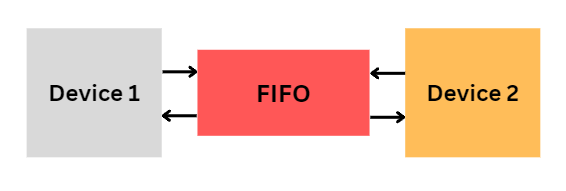
# 

# Introduction:

## **Objective**

The following sections will describe how the asynchronous FIFO will be verified using strategies learned from ECE-593. The goal of this section is to describe what type of testbench style will be used, what verification strategies will be implemented, and what type of stimulus will be used.

## **Top Level block diagram**



## **Specifications for the design**

## Producer Specification

* Frequency: 500MHz
* Idle Cycles: 2 clock cycles
* Burst Length: 1024\
* Duty cycle 50:50
  + 1. **Consumer Specifications**
* Frequency: 225MHz
* Idle Cycles: 1 clock cycles
* Duty cycle 50:50

# Verification Requirements

## **Verification Levels**

### **Hierarchy**

The testing will be performed at a module / block level environment that instantiates the FIFO as a DUT, and where a model of the external interfaces that handle all of the writing and reading from the FIFO. This allows there to be no limitations from any chip level testing. This will allow many corner cases or unique tests to be run against the block.

### **Control and Observation**

The control the user has is within the top level test. Using OOP the test has been simplified to a single start task. This task access protected tasks that read and write from the FIFO, so the controllability within the will be restricted to the scope of the number of data writes and the data vector supplied. Modifications can be made if necessary, but the single action task will perform all the needed transactions.

There will be multithreaded tasks that check all the necessary requirements provided to make sure that no requirement is being violated. This will report a pass or fail monitor/display into the transcript that can be extracted and manually verified to prove that no requirement is being violated.

### **Comparison Tagging**

Any comparison made to verify the asynchronous FIFO is working correctly must have an associated test tag with it and it must be displayed as a pass or failure inside of the transcript. This will be achievable by logging the stimulus imputed into the DUT and by comparing the outputs of the DUT to the inputs since it’s a FIFO the data should not change.

# **Required Tools**

## **Simulation Software**

We are using Mentor Questa as of now. In the future, we are also planning to use formal verification(FPV) so we might end up using the VCS.

## **Directory structure**

The top level directory for this project is structured as follows:

1. **README.md:** Lists the directories and directory descriptions along with how to run the simulation.
2. **docs/:** Any supporting documentation will be stored here for the project e.g. HLDS.
3. **sim/:** Contains simulation files.
   1. work/ directory for simulation files.
   2. run.do file.
   3. wave.do file containing the tests signals for the waveform.
   4. makefile to run the simulation
   5. manifest.f, a RTL and simulation files used in the project.
4. **src/:** Contains the RTL files.
5. **verify/:** Contains the testbench and device model system verilog code.

A manifest.f file will contain all of the system verilog simulation files. A rtl\_manifest.f will contain the source RTL code.

| manifest.f  *# File list for Simulation*  *# Sim*  +incdir+../verify/  ../verify/ccd\_if.sv    *#OOP portion of the project*  *##../verify/oop/sim\_ccd.sv*  *##../verify/oop/sim\_async\_fifo\_tb.sv*    *#UVM portion of the project*  ../verify/uvm/ccd\_agent\_pkg.sv  ../verify/uvm/top.sv   rtl\_manifest.f  *# File list for RTL and Simulation*    *# RTL*  +incdir+../src/  ../src/async\_fifo.sv |
| --- |

A wave.do file exists to auto populate the waveform signal names.

A makefile is used to compile the system verilog and run the simulation.

| *# Define Environment Variable*    PROJ\_NAME = 'ASYNC\_FIFO'  PROJ\_HOME = ../    make:  rm -rf work  vlib work  vlog -sv -64 -cover bcse -f rtl\_manifest.f  vlog -sv -64 -cover bcse -f manifest.f  vsim -gui -do run.do -classdebug -coverage -voptargs="+acc" -uvmcontrol=all -msg    UVM\_LOW:  rm -rf work  vlib work  vlog -sv -64 -cover bcse -f rtl\_manifest.f  vlog -sv -64 -cover bcse -f manifest.f  vsim -gui -do run.do -classdebug -coverage -voptargs="+acc" +UVM\_VERBOSITY=UVM\_L    *#Clean is ran automatically performed on 'make' to limit simulator issues*  clean:  rm -rf work |
| --- |

For the run we are using run.do file

| set StdArithNoWarnings 1 set NumericStdNoWarnings 1 do wave.do run 0 log -r /\* run -all |
| --- |

To run the simulation all the user needs to do is run *make* inside of the *sim/* directory.



The simulation can be run with the UVM verbosity set to UVM LOW to reduce how busy the transcript is. The down side to this is the tag log does not report comparisons when running in *make UVM\_LOW* mode.



# Risks and Dependencies

As Design RTL and Verification is progressing we might get a condition called metastability and rollover issue. It's an issue on hardware, not a simulation issue. As for now we haven't received any issues since the rollover is handled by a rollover or wrapped flag bit that is asserted when the write pointer rolls over or wraps memory, but is deasserted when the read pointer is rolling over or wrapping.

But we can handle using dependencies like Strategies such as effective clock domain crossover procedures, the use of gray-code counters, and the implementation of strong synchronizers can all help to reduce the risks associated with metastability and roll-over difficulties in hardware designs.

# Functions to be Verified.

## Write and Read Operation

The write operation will be verified through the use of an input and output mailbox that the test adds input write data to, and captures the output read data in the other mailbox. A test will be performed when both mailboxes contain an item. This item will be the input and the output of the FIFO, which should contain the same item. A comparison will be made and a test tag will be generated documenting the successful or failure comparison.

## Empty and Full Status Handling

The empty and full status flags are determined by the values of the read and write pointers. On a write, or a read the pointers are incremented, but the result of the increment is modulus to the size of memory to prevent overflow. The conditions to write are that the memory must not be full. This can be determined when the wrapped / rollover bit is asserted and the read and write pointers are at the same location. The empty signal is asserted when the wrapped / rollover bit is deasserted and the write and read pointers are at the same location. The wrapped bit is asserted when the write pointer reaches the end of memory and wraps to the beginning, and is deasserted when the read pointer reaches the end of memory and wraps to the beginning.

## Idle Cycle Handling

The idle cycle is handled in the external device models and it uses a parameter to determine how many loops must be done of *@(posedge clk)* must be done for the associated clock. This allows the idles to occur before performing a read or a write. A test tag needs to be generated confirming the idle cycles exist and this can be done by having a multithreaded task fork join a time tracker for the associated number of clock cycles before asserting a write enable or read enable.

## Burst Mode

Burst writes of 1024 will be tested as it is a maximum burst write requirement. This will also be tested through robustness and many different sizes will be tested. Every single size can be tested as it is a very quick simulation, but in the future if the tests become longer, less tests in the middle of 1 to 1024 range will be used less. This function will have corner cases that include exceeding the max, and performing below the minimum transactions and observing the behavior. During comparisons of the data values and the burst sizes test tags will be generated reporting a pass or a failure.

## Duty Cycle Handling

Duty cycle can be determined by kicking off a multithreaded fork join that expects the associated CLK to be asserted for half a period (time), and low for the exact same amount of time. This will be done by using the $time function at the rising edge and falling edge to verify the cycle.

# Tests and Methods

## Method

White box testing is being used because it is known how the internal operation of the FIFO is done. The FIFO is not very complex so it is very easy to perform predictions using two simple input and output mailboxes.

## Tests

### FIFO Full Test

The fifo will be filled and the FIFO\_FULL signal will confirm that the FIFO is full. More data will be attempted to be written, but it should wait until data is read from the FIFO. The FIFO full boundary will be tested.

### FIFO Empty Test

The fifo will initialize in an empty state, and will be filled to make the FIFO in a non-empty state, and then the FIFO will read until it no longer has contents. The FIFO will not read when the FIFO is empty. At the max burst of 1024, the FIFO will eventually become fully written to, and the FIFO will read the remaining data until it is empty.

### Reset Test

The reset test will reset the FIFO in multiple boundaries.

1. Reset After a burst write, and perform another burst write.
2. Reset during a burst write then perform another burst write to show the model properly recovers from a reset state.

### Back To Back Operations

The FIFO will be buffered with data to continuously perform write and read operations. The ideal operation is for the FIFO to wait for the FIFO to be emptied before writing another burst up to 1024 after receiving a burst of 1024. The FIFO will receive back to back 1024 burst writes, and multiple others such as 512 back to back writes. As long as there is a valid consumer and data in the FIFO, it will be read. During a full burst write the producer will finish writing to the FIFO, then the reader will continuously read until the FIFO is empty.

## Advantages and Disadvantages

We already know the internals of the design, therefore we use the white Box method.

Provides detailed testing of internal logic, code paths, and branching, resulting in excellent test coverage. enables early detection and correction of issues during the development process. Provides insights on optimizing code and boosting overall software efficiency.

## Architecture

Object oriented programming is used by making a model of the external interfaces, an interface is used to connect the model to the instantiated asynchronous FIFO (DUT). Methods inside of the model class will trigger single writes or burst writes depending on the stimulus provided.

## Strategy

Dynamic simulation for now to verify at block level and Sub system level. We also plan on using formal verification in the future to test the system

## Driving methodology

Exhaustive testing through constrained random testing using UVM. A sequence item is generated in a sequence that either is a reset, random value, max value, or minimum value test.

## Checking methodology

Checking is done through OOP before the implementation of UVM. After the implementation of UVM the UVM monitor class monitors the signals on the DUT interface. The monitor is modeled after the DUT to be able to know when to sample the interface to accurately fill in a sequence item with the correct input and output data that will be sent to the scoreboard for evaluation. The scoreboard will check the size of the dynamic arrays created in the monitor's sequence item for the input and output to verify they are the same size and that the contents match. Test tags described below will be produced in a test tag log and will state whether or not the stimulus is passing / the test is passing.

## Detailed Test Descriptions

| **Test Name** | **Test Tag** | **Description** |
| --- | --- | --- |
| Compare Burst Size | TT\_00 | Measures the number of burst items in the sequence sent to the scoreboard. |
| Compare Individual Element | TT\_01 | Measures the individual burst elements and compares the input to the output. |

# UVM

## Overview

This documentation gives a quick overview of the explanation of UVM testbench architecture and how we implemented the architecture.

The UVM code for the asynchronous fifo can be found here:

<https://github.com/abfouts/ece_593_asynchronous_fifo/tree/main/verify/uvm>

## The UVM Testbench Architecture

The generic utilities provided by the UVM class library are transaction library mode(TLM), configuration database, component hierarchy, etc.., The Below block diagram shows the terminology and typical block diagram of UVM testbench architecture.

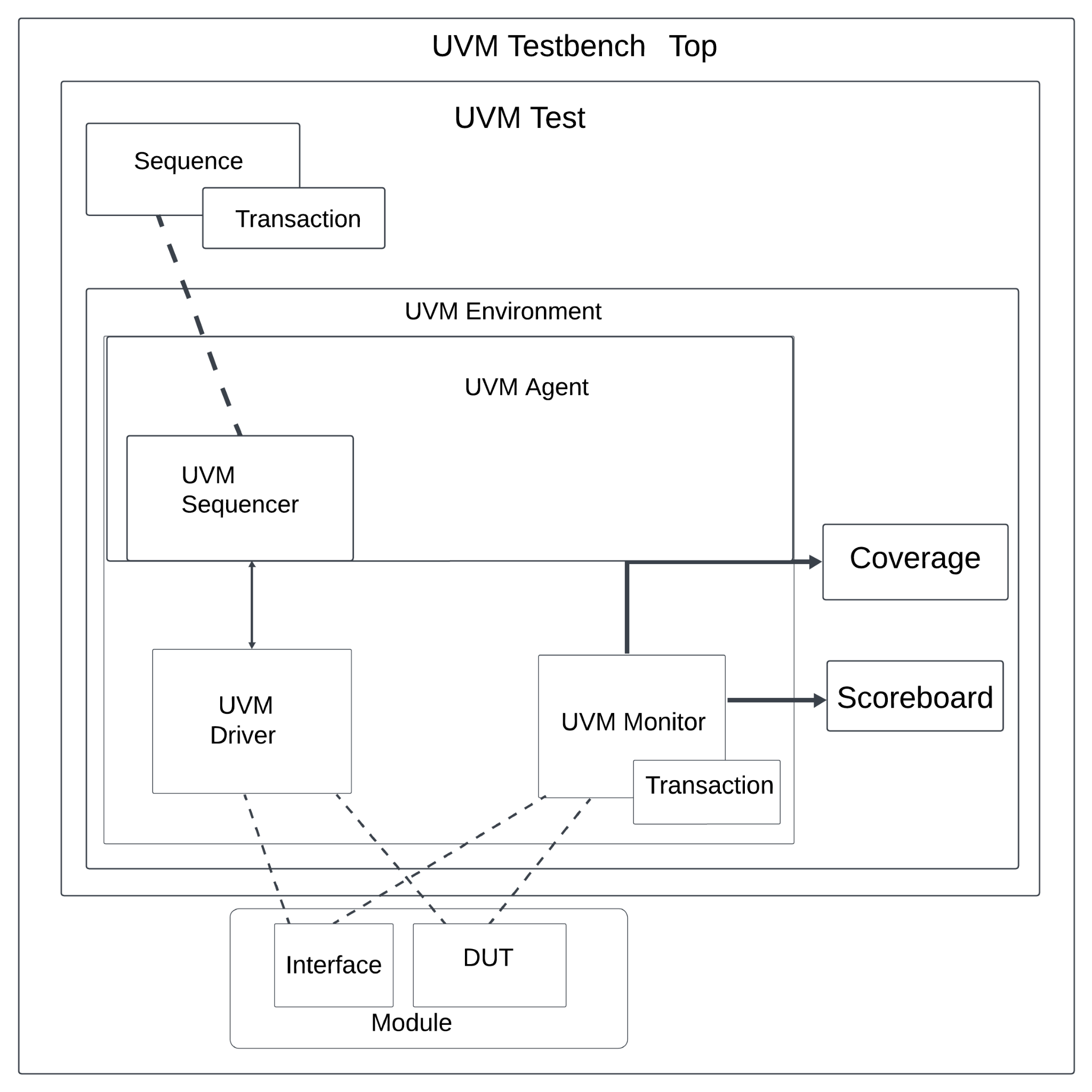


Fig 1: UVM Testbench Architechture.

## UVM Testbench

The UVM testbench instantiates the DUT (Design Under Test) module and test class, configuring the connection between them. UVM test is dynamically instantiated at runtime, allowing testbench to be compiled and run with different tests.

The testbench or top modules will generate the producer and consumer clocks, instantiate the dut and create the random test that extends from uvm test and register it with the factory. The top-level module creates the interface and places it on the uvm\_config\_db for all uvm components to receive.

## UVM Test

The UVM Test is the top-level component in the UVM testbench, it performs the main functions like instantiating the top environment, configuring the environment using factory overrides, and applying stimulus by UVM sequences through environment DUT.

The UVM test being used is specific for a random value test that will be run many times. The test will send many random burst transactions back to back with random resets produced throughout.

## UVM Environment

It is a hierarchical component that groups other verification components that are connected. Components that are instantiated inside the UVM environment are UVM Agents, UVM Scoreboard, or other environments. The top-level environment target remains similar to DUT.

The UVM environment extends from the built in UVM ENV, and it instantiates the agent and the scoreboard. In the future (next milestone), the environment will also instantiate the coverage. In the connect phase the environment will connect the agent analysis port to the scoreboard analysis port.

## UVM Scoreboard

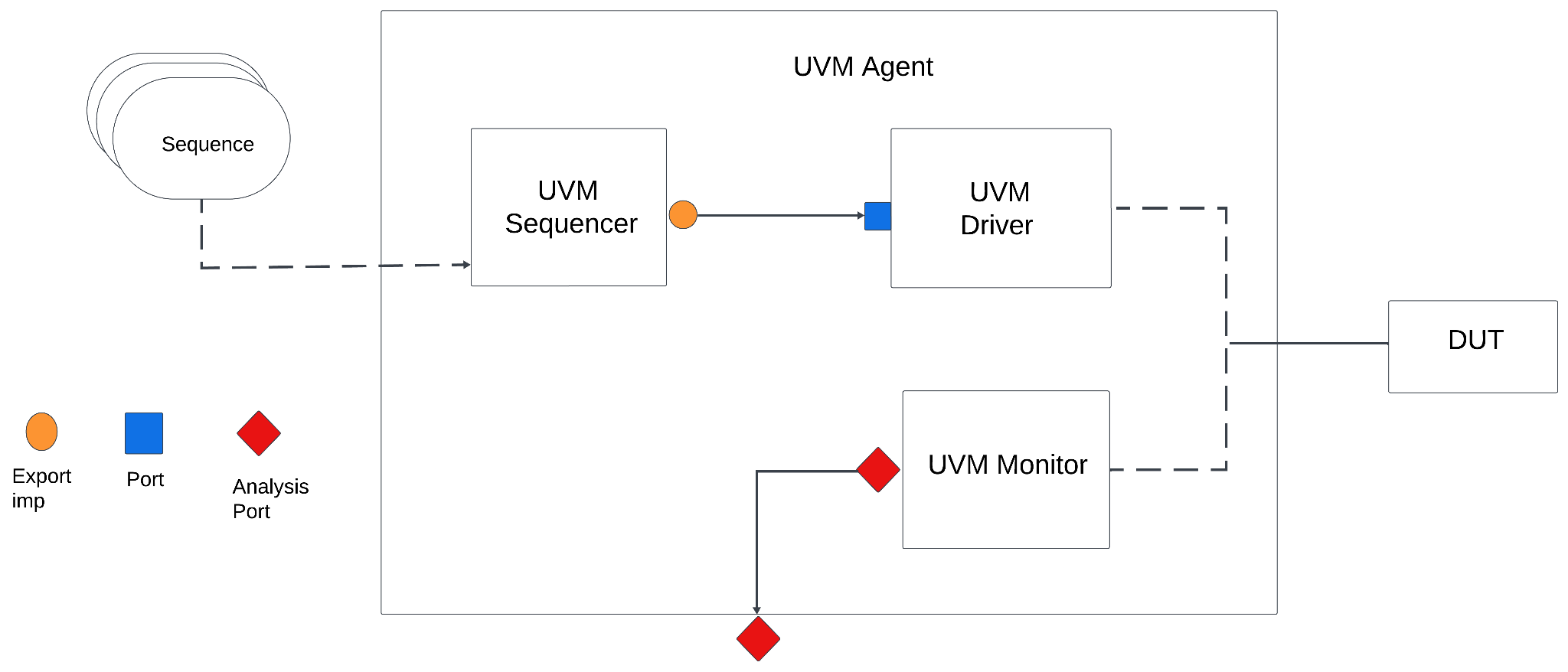
The Scoreboard's main function is to check the behavior of DUT, in our condition we are writing the mailboxes for the scoreboard, the data is generated where the mailboxes are used for communication between parts of the UVM testbench, and the random constraint makes sure it is in the specified range of burst entries.

The scoreboard will extend from the UVM scoreboard, and it uses the macro “uvm\_analysis\_imp\_decl” to create a custom port name for the write function. The comparison will be performed inside of the write function as it is not streaming individual transactions, but instead receiving the entire burst packet. It can be designed in other ways, but this allows the scoreboard to know exactly how many items were written and read from the asynchronous fifo.

## UVM Agent

UVM Agent is a hierarchical component that contains other verification components which are dealing with specific DUT interfaces. A UVM agent contains a UVM sequencer to manage stimulus flow, a UVM Driver to apply stimulus on the DUT interface, and a UVM monitor to observe the DUT interface. It might include other components like Checkers and TLM models, etc.

There is one agent and it extends from UVM agent and it instantiates the driver, monitor, sequencer, and config. The config sets if the UVM agent is active or not, but is not necessary. For the case of this class the agent will be active, and then creates an analysis port, and gets the ccd vif from the uvm\_config\_db. In the connect phase the monitor analysis port is connected to the agent analysis port to simplify the connection to the scoreboard, and the sequencer is connected to the driver seq\_item\_port.



**Fig 2: UVM Agent**

## UVM Sequence Item

The sequence item is used to define the type of item transaction that will take place in the sequence, sequencer and most of the times the analysis ports. It acts as a packet of data that can contain the inputs and outputs of the DUT.

The sequence item extends from the UVM sequence item and contains the input data, output data, reset and all random variables that are going to be used for randomizing the data. A single sequence item will act as an entire randomly generated write vector or read vector for the monitor. This contains the number of idle write and read cycles and any other information a sequence may need to successfully write to the asynchronous fifo.

## UVM Sequence

It is an object that contains the behavior of generating stimulus and is not a part of the component hierarchy. It can come from a single transaction and may drive stimulus for the time of simulation. It can invoke another sequence.

There are two sequences. The base sequence for reset that extends from UVM sequence and expects the sequence item defined above, and a random sequence that extends from base sequence. The base sequence randomizes the data but asserts the reset, so nothing should be inputted or outputted to/from the DUT except for the memory being reset. The random sequence contains the method to generate a random number of data writes, and dynamically allocate random bytes to the vectors.

## UVM Sequencer

The UVM Sequencer serves as an arbiter controlling transaction from multiple stimulus sequences, in more detail it controls the flow of sequence items transactions generated by one or more UVM sequences. In our code sequencer generates sequences of random transactions for testing DUT.

The sequencer extends from the UVM sequence.

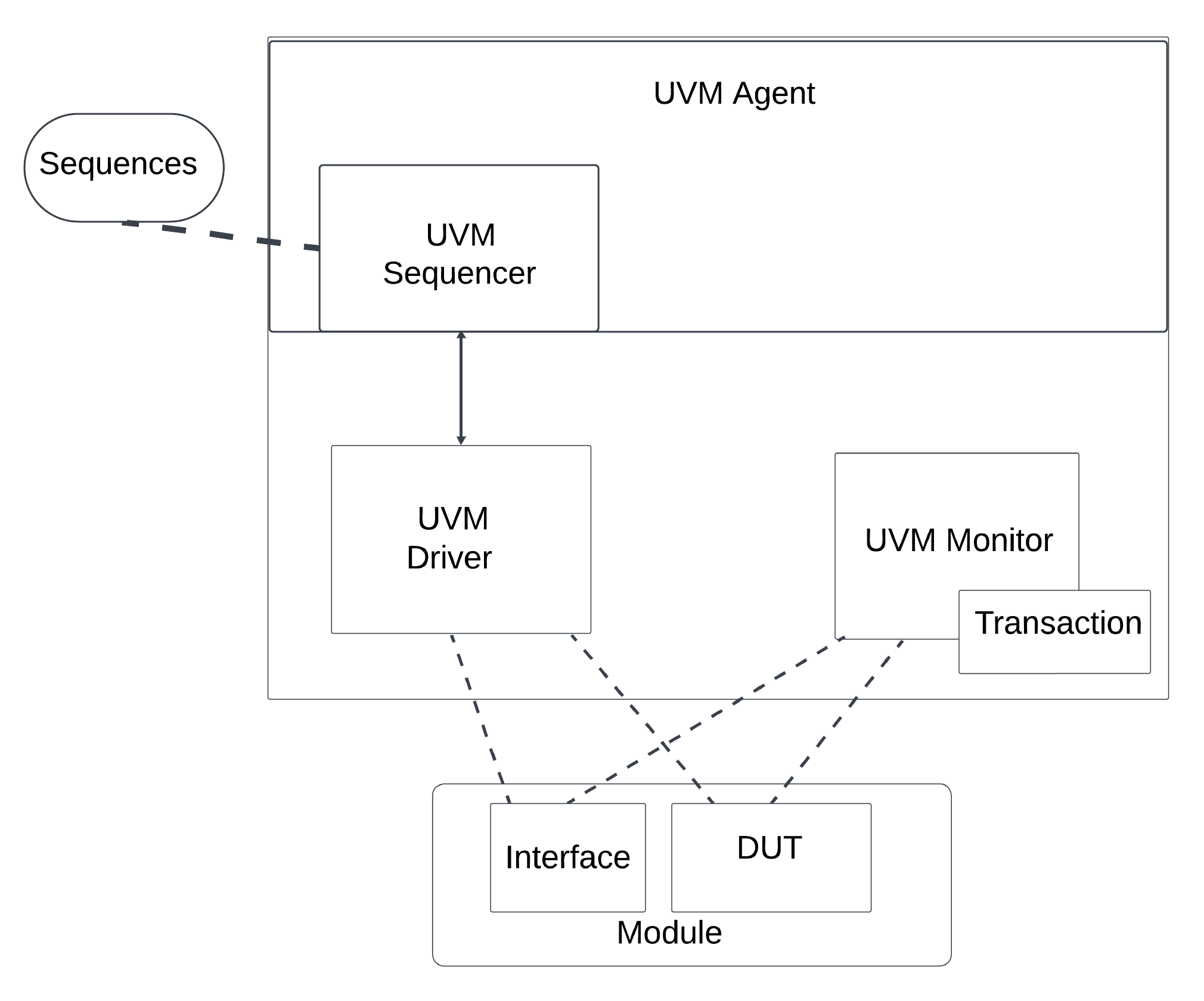


Fig 3: Sequencer

## UVM Driver

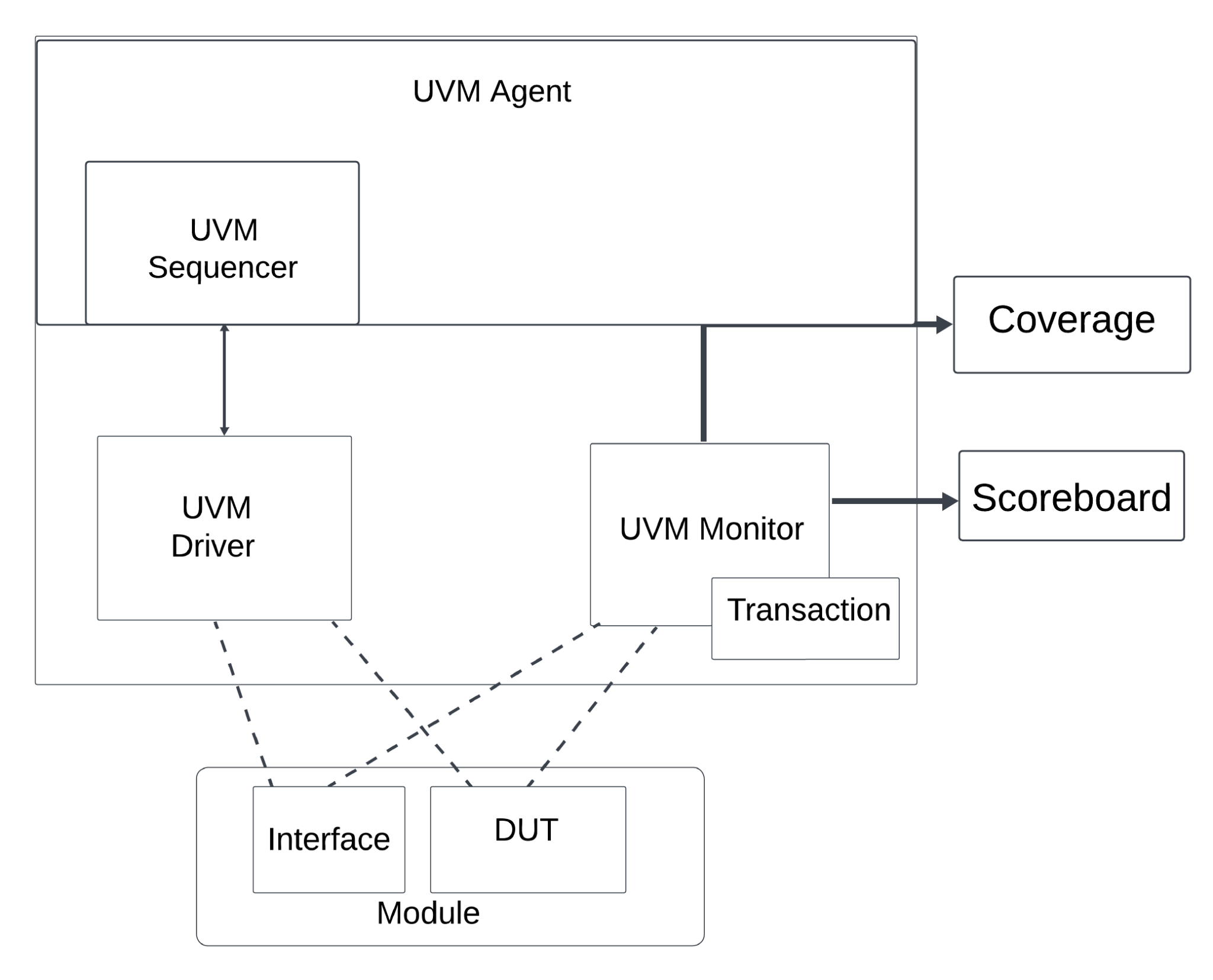
It receives individual UVM sequence items transactions from the UVM sequencer from the sequencer and drives to DUT. Driver converts transaction level stimulus to pin level stimulus. It can also have a TLM port to receive transactions as a burst from a sequencer to access DUT to drive the signals. In our code “start “ can be used from a test sequence to initiate transactions where it specifies the number of write operations and data to write. During simulation tasks will perform write and read operations on DUT.

The driver extends from the UVM driver class and contains the method to write the sequence item data to the DUT. The sequence item contains the entire byte vectors used. This method controls both read enable and write enable signals. The read and writes are forked off to run in parallel. Both loop for the expected number of writes, and on the nth +1 cycle it asserts write enable and read enable signals, and the write provides the expected byte onto the data in the bus. At the last write and expected read the respective enable signals are deasserted.

## UVM Monitor

UVM Monitor samples the DUT interface and captures the information there in transactions which are sent out to the rest of the UVm testbench for further analysis. It is similar to Driver as it converts the pin-level activity to transactions.

The monitor extends from the UVM monitor and provides the monitored sequence item to the analysis port of the scoreboard for comparison. The monitor keeps track of when a transaction occurs, so when it receives a write when it has not received any transactions previously and it is not in reset mode starts the transactions. The monitor forks off the read and write operations separately and performs while loops that use the positive edges of the respective clocks and following the design specifications for idle periods to accurately sample the data in and data out bus lines. Once the read and write enable signals are deasserted the sequence item that contains the entire transaction is sent to the scoreboard via the analysis port.



**Fig 4: UVM Monitor having Transaction, Coverage, and Scoreboard**

# **Coverage Requirements**

## Functional Coverage

Functional coverage is obtained by measuring every possible value provided to and produced by the asynchronous FIFO.

Data: The input and output data is measured by three different categories: 8’h00, 8’hFF, and 8’h01 : 8’hFE. This allows the edge cases to be obviously hit, and the middle cases being less important cases to test, so any value in the provided range will allow for functional coverage to be hit. This is so there is no need for 2^8 different data test cases.

## Code Coverage

The goal is to achieve 100% code coverage. Fortunately the design is simple enough that this should be achievable through robustness testing and forcing resets.

## Cover Groups

### Input Data

The input data cover group will contain a single coverpoint that has 3 bins.

1. Ones: The value is expected to be all ones.
2. Zeroes: The value is expected to be all zeroes.
3. Others: The values are between all ones and all zeroes.

### Output Data

The output data cover group will contain a single coverpoint that has 3 bins.

1. Ones: The value is expected to be all ones.
2. Zeroes: The value is expected to be all zeroes.
3. Others: The values are between all ones and all zeroes.

### Control Signals

Each single bit control signal is in the control signal covergroup, as an individual coverpoint. Each one contains a bin for 1’b1 and 1’b0, determining whether or not the signal is asserted or deasserted based on the configurations.

The signals covered in this section are: full, empty, write enable, read enable, and active low reset.

# Contributions & Expertise

## Abram Fouts Expertise

Working professional since graduating undergrad with a degree in Computer Engineering from Portland State. Worked in Aerospace since graduation working with FPGA design and FPGA verification. Been in verification for the last two years writing system verilog and UVM, and prior to graduation interned at Intel on the PHED team doing board design for 6 months, and then a 6 month internship at Micron writing software remotely for the board design team.

## Abram Fouts Contributions

1. Created the simulation directory environment.
2. Wrote a version of the DUT code.
3. Created outline of the object oriented programming section.
4. Filled in the OOP code.
5. Converted the OOP code to UVM.
6. Created UVM components, agent, env, scoreboard, and tests.
7. Participated in documentation.

## Yunus Syed

1. Verification of DUT code and added some elements.
2. created some sequences, and agent and testing of OOP code.
3. Helped in filling the OOP code and UVM testbench.
4. Updated versions of DUT and tested them.
5. Participated in doing documentation.

# Schedule

| Milestone | Completed in Verification | Progress | Date of Completion |
| --- | --- | --- | --- |
| 1 | conventional test bench to check RTL, run.do files | Done | Feb 3, 2024 |
| 2 | introduced - test using OOP | Done | Feb 1, 2024 |
| 2 | Introduced - Generators, transactions, Drivers | Done | Feb 6, 2024 |
| 2 | Updated verification plan and got all transcripts matching burst | Done | Feb 11, 2024 |
| 3 | Scoreboards and coverage | Done | Feb 14, 2024 |
| 3 | Revised verification plan & coverage report, do file for simulations | Done | Feb 15, 2024 |
| 4 | UVM test bench and test plan document | Done | Feb 25, 2024 |
| 5 | Agents, scoreboard, coverage, env tests | Done | March 3, 2024 |
| Final Project Submission | Completion | Done | Mar 7, 2024 |

# References Uses / Citations/Acknowledgements

Chip Verify, Verification Academy, and Doulos are the sites used most for learning syntax and code. These were paired with class lectures for successful completion.

<https://www.chipverify.com/>

<https://www.doulos.com/>

<https://verificationacademy.com/>