UVM Testbench Hierarchy Overview

Project Report:
UVM Changes:

The team was not able to complete the project by milestone 5 due date (5/27) because of some bugs that were difficult to debug, but have since made changes which has made the UVM testbench more functional. The remainder of this section will describe what occurred and what we were thinking during our troubleshooting.

We originally started seeing issues during the class based testbench development. During this time we were starting to see a mismatch between our testbench model and our DUT, but due to running on our class-based testbench development we decided to move on and focus on the UVM testbench version.

When the initial UVM testbench was developed, it was under the assumption that the Flags changed as soon as the writes or reads occurred. Since this was not explicitly specified in the design / verification document, it led to a mismatch in the Flag timing between the DUT and Model. In addition, there was a bug in our monitor class causing there to be a timing issue leading to a mismatch on read cycles, that we initially thought was caused by the DUT.

We initially believed the write and read pointers were potentially wrapping before reaching the last entry in the array caused by an error in the flag calculations since we were seeing many flag mismatches. We started looking into the DUT really closely and revisited our conventional testbench and expanded it to get a better view of the DUT's functionality. This is where we were at when the initial M5 deadline occurred. This also meant that we were unable to program the UVM File Logging functionality and were unable to fully run the test sequences / cases that we initially planned on.

Between the initial deadline (5/27) and our demo time (5/29), we realized that the flag timing issue was not an incorrect calculation in the DUT, but a misinterpretation of how our Asynchronous FIFO was going to function. Once that issue was discovered, we realized that our 'early wrapping' problem was not actually a bug with the DUT and must be a bug with the testbench itself.

When we adjusted the read and write clocks to be matching, the missed reads were suddenly being read, indicating that there was a timing issue with the driver or monitor. Since the monitor is what connected the DUT to the scoreboard model, we checked there first and discovered an error in how we were approaching the monitoring of the asynchronous nature of the design. We initially had the code below:

```
forever begin
        pkt = async_fifo_pkt::type_id::create("pkt");

        fork
          begin : Reset
            pkt.ainit           = vif.ainit;
          end
          begin : Write
```

```
            @(posedge vif.clk_wr);
            pkt.wr_en             = vif.req_wr;
            pkt.din               = vif.data_wr;
            pkt.fifo_full         = vif.fifo_full;
            pkt.fifo_almost_full  = vif.fifo_almost_full;
        end
        begin : Read
            @(posedge vif.clk_rd);
            pkt.rd_en             = vif.req_rd;
            pkt.dout              = vif.data_rd;
            pkt.fifo_empty        = vif.fifo_empty;
            pkt.fifo_almost_empty = vif.fifo_almost_empty;
        end
    join

    monitor_port.write(pkt);
end
```

This code was initially made in this way to simplify the packet communication between the scoreboard and monitor and we initially believe that the fork-join within the forever was sufficient enough to catch both the reads and writes. This was not correct and lead to the occasional scenario where 2 read clocks occur before the write clock causing the DUT to perform 2 reads but the model to only perform 1, causing our ‘missing reads’. We adjusted it to what is seen below the morning of the demo day.

```
fork
    forever begin
        w_pkt = async_fifo_pkt::type_id::create("w_pkt");
        w_pkt.w_pkt_f = 1;

        w_pkt.ainit             = vif.ainit;
        @(negedge vif.clk_wr);
        w_pkt.wr_en             = vif.req_wr;
        w_pkt.din               = vif.data_wr;
        w_pkt.fifo_full         = vif.fifo_full;
        w_pkt.fifo_almost_full  = vif.fifo_almost_full;
```

```
        // w_pkt.rd_en         = 0;
        // w_pkt.dout          = 0;
        r_pkt.rd_en          = vif.req_rd;
        r_pkt.dout           = vif.data_rd;
        w_pkt.fifo_empty     = vif.fifo_empty;
        w_pkt.fifo_almost_empty = vif.fifo_almost_empty;

        monitor_port.write(w_pkt);
      end
      forever begin
        r_pkt = async_fifo_pkt::type_id::create("r_pkt");
        r_pkt.r_pkt_f = 1;

        @(negedge vif.clk_rd);
        r_pkt.ainit          = vif.ainit;
        // w_pkt.wr_en         = 0;
        // w_pkt.din           = 0;
        w_pkt.wr_en          = vif.req_wr;
        w_pkt.din            = vif.data_wr;
        w_pkt.fifo_full      = vif.fifo_full;
        w_pkt.fifo_almost_full  = vif.fifo_almost_full;
        r_pkt.rd_en          = vif.req_rd;
        r_pkt.dout           = vif.data_rd;
        r_pkt.fifo_empty     = vif.fifo_empty;
        r_pkt.fifo_almost_empty = vif.fifo_almost_empty;

        monitor_port.write(r_pkt);
      end
    join_none
```

Separating the Monitor (and modifying the scoreboard) to handle separate read and write monitor packets ensured that the scoreboard model saw the same inputs as the DUT allowing us to achieve 100% transaction passing, though the flag mismatches still remained.

After the demo, the timing of the scoreboard was changed to match the functionality of the DUT with the flags changing on the next clock cycle after their respective read or write except for

when the flag would change due to an opposite operation occurring, which would need to propagate through a synchronizer adding further clock delays. This timing was added to the testbench and the flag warnings were cleared. With this done, UVM file Logging was added, but we did not have enough time to verify all of the test sequences / cases that we had initially planned on in our Verification Plan. **The current version of the UVM testbench can be found in the M6 folder**, as we wanted to make changes without changing the original M5 submission. This allows us to compare the two and measure the improvement from the initial submission date like can be seen here with the different monitor functions.

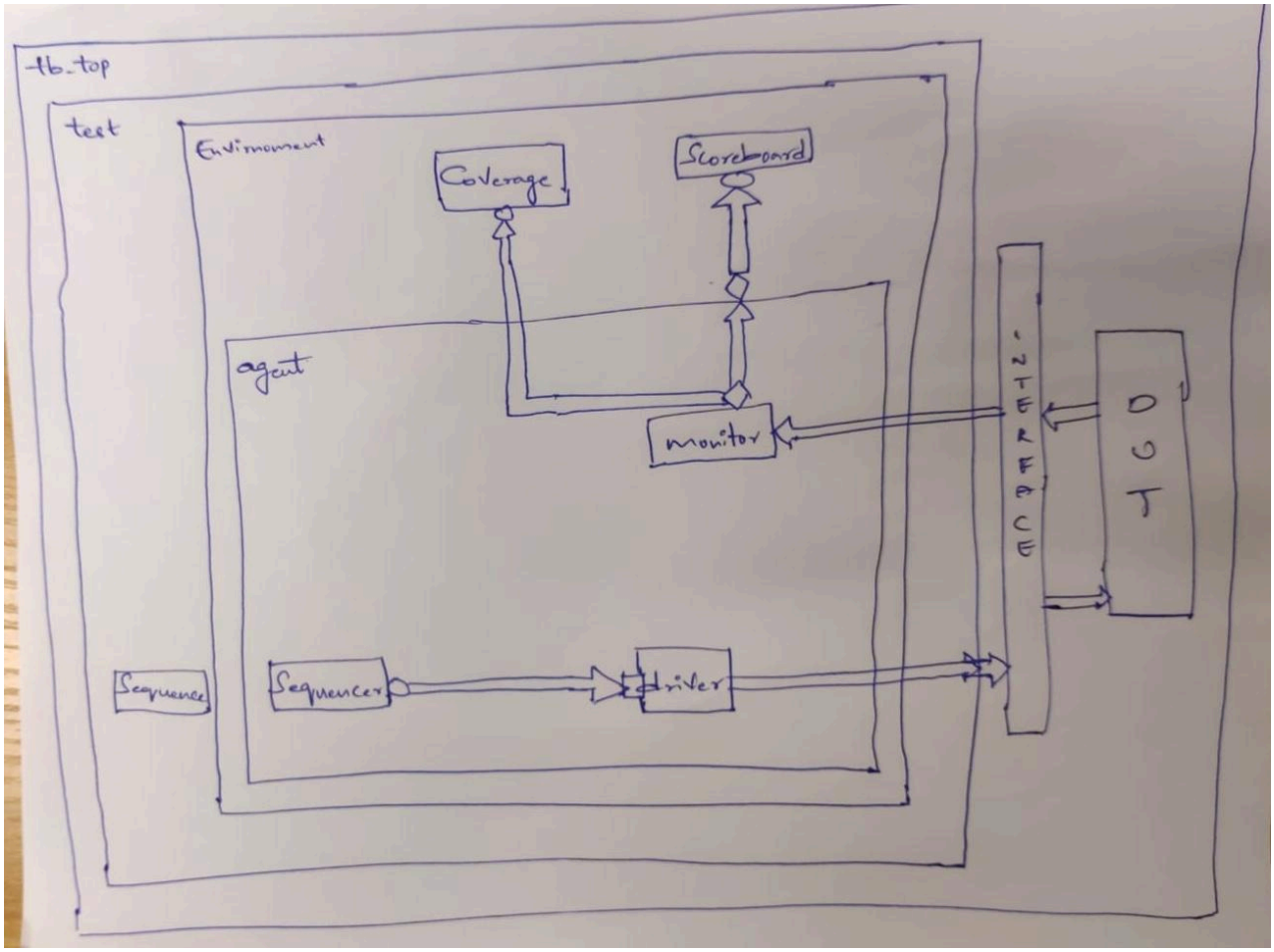
Class Based TB Changes:

In addition to the UVM changes, the team was able to complete the class based test bench that was basically abandoned so everyone could focus on the UVM testbench. In the M6 folder which holds all changes from 5/27 to 6/03, the class based test bench is now able to correctly keep track of the flags from the DUT since at the time of creation, the synchronization timing was not known. As for testing, we were able to add a few different test cases in the short amount of time available. Also, the class based tb will now report both code and functional coverage of the DUT.

Verification Strategy:

Verifying an asynchronous FIFO is challenging due to the involvement of two independent clock domains: one for writing and another for reading. The key goal of this milestone is to develop a test environment using the UVM (Universal Verification Methodology) framework. The verification strategy focuses on rigorously testing the FIFO's behavior during asynchronous data writing, storage, and reading. Additionally, it ensures proper handling of edge cases and error conditions, confirming the design's robustness and reliability.

Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.)



Testbench architecture used will include:

- Testbench Top
  - o Highest level of the testbench Hierarchy
  - o Contains Environment instantiation and configuration
  - o instantiates the DUT and the test sequence
- Environment
  - o Contains the agent and components for driving stimulus and checking DUT responses
  - o Instantiates the agent
- Agent
  - o Responsible for interfacing with DUT interface
  - o Contains components to drive stimulus and collect responses
    - ▪ Agent Sequencer
      - • Generates sequence of transactions
    - ▪ Agent Driver
      - • Drives the stimulus to the DUT
    - ▪ Agent Monitor
      - • Monitors signals on the interface of the DUT

- Collects data for analysis and scoreboard

- Scoreboard
  - Compares expected results with actual results from DUT
  - Raises error flags on data mismatch
- Sequences
  - Contains sequence of transactions that represent specific test scenarios
  - Sequence Item represents a single traction
    - Contains information for driving and checking
    - Each test contains separate sequence

Gray Code : Using gray code is a standard technique to ensure reliable and error-free data transfer between asynchronous domains. This minimizes the risk of metastability and sampling errors, thereby enhancing the reliability of the FIFO operation.

Write Pointer : A binary write pointer keeps track of the location where the next data should be written.. The write pointer, which operates in the write clock domain, is converted to Gray code before being passed to the read clock domain. This ensures that the read logic gets a stable and correct pointer value

Read Pointer : A binary read pointer keeps track of the location from where the next data should be read. The read pointer, which operates in the read clock domain, is converted to Gray code before being passed to the write clock domain

Memory Array : The FIFO buffer uses a memory array to store data. Data is written to the memory in the write clock domain and read from the memory array in the read clock domain.

Implementation: Expected Data Flow

- **Testbench Top-Level (TB Top)**
  Begins the testing process by triggering the test sequence.

- **Environment Setup**
  Responsible for creating and setting up the agent with necessary configurations.

- **Agent Responsibilities**

  - Produces and manages test sequences.

  - Sends transaction data to the DUT (Design Under Test) interface.

  - Observes and records the DUT's output behavior.

- **Scoreboard Function**
  Compares the input data with the DUT’s output and reports any mismatches or unexpected behavior

Test Scenarios:

TEST CASES	DESCRIPTION
FIFO Reset	The reset functionality in the driver is used to verify whether the FIFO returns to its default or initial state.
FIFO Full	Data is written into the FIFO until it reaches maximum capacity, then it is verified whether the 'full' flag gets asserted.
FIFO Empty	It is verified that the 'empty' flag gets asserted when no new data is written to the FIFO and the existing data is completely read.

FIFO Depth Calculation:

Double the frequency of write and half of the read

120 MHz is the sender clock frequency.

Three idle cycles separate two consecutive writes, and the receiver clock frequency is 50 MHz.

There are two idle cycles in between two consecutive reads.

Given that the sender clock frequency is less than the receiver clock frequency, write burst = 1024.

Three clock cycles separate two consecutive writes, indicating that after writing one piece of data, the

write module waits three clock cycles before starting the next write. This means that one piece of data is

written every four clock cycles.

Writing one data item takes 4 * (1/120MHz) = 33.33 ns.

It took 34,129.92 = 34,129 ns to write the data in the burst.

One data item's reading time is equal to 3 * (1/50MHz) = 60 ns.
Thus, the read module will read one data item from the burst every 60 ns.
In 34,129 ns, the number of data pieces that can be read is 34,129/60 = 568.81 = 568 items. 1024 – 568 = 456 is the remaining number of bytes that must be kept in the FIFO.

Transcript:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  563
# UVM_WARNING :    0
# UVM_ERROR :  144
# UVM_FATAL :    0
# ** Report counts by id
# [COMPARE]   348
# [Questa UVM]     2
# [RNTST]     1
# [SEQ_BODY]     6
# [TEST_DONE]     1
# [UVMTOP]     1
# [async_fifo_cov]   174
# [uvm_test_top.env.cov]   174
# ** Note: $finish    : /pkgs/mentor/questa/10.6b/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 2431 ns  Iteration: 54  Instance: /custom_async_fifo_tb
# Break in Task uvm_pkg/uvm_root::run_test at /pkgs/mentor/questa/10.6b/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
# Stopped at /pkgs/mentor/questa/10.6b/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
# End time: 22:58:17 on May 27,2025, Elapsed time: 0:00:05
# Errors: 0, Warnings: 0
```

Transcript attached in zip file.

References:

1. Professor examples from slides
2. Verification academy