Asynchronous FIFO: Verification Plan

Session 1 Group 4

Nick Allmeyer, Alexander Maso, Ahliah Nordstrom

Version 2.0 - May 27, 2024

Table of Contents

0: Document Revision History

Version	Date	Description
v 1.1	4/22/2024	For Milestone 1 deliverable
v 1.2	4/28/2024	For Milestone 2 deliverable, no feedback from M1 yet
v 1.3	5/14/2024	For Milestone 3 deliverable, feedback on M1&2 provided and not related to the verification plan
v 2.0	5/27/2024	For Milestone 4 deliverable, adding UVM information to verification plan

1: Function Intent, Design Description, and Specification

The asynchronous FIFO acts as a buffer to manage data flow between two parts of a system operating on different clock domains. The FIFO ensures data integrity and timing without the need for clock synchronization between sender and receiver blocks, operating at 80 Mhz and 50 Mhz respectively.

A memory buffer with a depth of at least 45 items is used to store data elements. Operations are consistent and predictable for both READ and WRITE, as they are fully synchronous to their respective clocks. Data is written to the FIFO at a rate of 12.5 ns per operation and is determined by the rising edge of the WRITE clock. Data is read at a rate of 20 ns per operation and is determined by the rising edge of the READ clock. The ratio of WRITEs to READs is 8:5, simplified down from 120:75. This ratio defines the operational balance between input and output operations so that the FIFO does not overflow/underflow under normal operating conditions.

A binary counter is used to track READ and WRITE pointers that then indicate the next positions for READ and WRITE operations. These pointers will be initialized to zero upon a system reset and then increment as data is written and read to and from the FIFO. If an invalid READ operation is performed, due to an empty FIFO, the RD_ERR signal will be asserted. If an invalid WRITE operation is performed, due to a full FIFO, the WR_ERR signal will be asserted. The HALF_EMPTY and HALF_FULL flags are used as a way to indicate nearing their respective states. The FULL flag is generated in the write-clock domain when no more data can be written to the FIFO without overwriting existing data. The EMPTY flag is generated in the read-clock domain when there are no data items to read from the FIFO and both pointers are equal.

2: Verification Levels

There will be two verification levels for the DUT: Unit-level and System-level.

- **Unit-Level Verification**: Each component of the DUT will be verified independently
    - **FIFO Memory Unit**: This FIFO memory buffer component is accessed by the write and ready clock domains. It uses separate write and read clocks (CLK_WR and CLK_RD) and enable signals (WR_EN and RD_EN) for data transfers. The memory is implemented as an array where the size and data width is parameterized.
    - **Sender Unit (WRITE control)**: This is where the WPTR module mirrors the functionality of the RPTR but for writing data into the FIFO. It uses gray code for cross-domain synchronization and manages the full condition of the FIFO by comparing WPTR and RPTR. It then updates the WADDR based on the increment (INC) signal to prevent overflow.
    - **Receiver Unit (READ Control)**: This is where the RPTR module keeps track of where the read from within the FIFO. It uses a gray code counter to manage cross-domain communication, with the use of a binary
    - **Control Unit (Synchronization)**: This unit facilitates cross-domain communication between RPTR and WPTR. It uses two stages of flip-flops to safely transfer signals between different clock domains.
- **System-Level Verification**: All components that make up the DUT are verified together, cohesively. It is to test the interaction rather than particular functions within the DUT. After unit-level testing, this is when system-level verification is performed for the FIFO and focuses on interfacing the FIFO units together. The FIFO will be tested for managing the flow of data between the two different clock domains.

3: Required Tools

- Questasim simulator for compiling, debugging, simulating, and coverage analysis
- Choice of HDL language is SystemVerilog (SV)
- Assertion-based verification will be in SystemVerilog Assertions (SVA)
- Github and Google Drive for version control and collaboration
    - https://github.com/anordstrom21/team_4_Async_FIFO

4: Risks, Dependendencies and Mitigation Plan

Here are the expected risks and dependencies throughout the verification process:

- Creating/interpreting the Clifford Cummings Asynchronous FIFO Design will be time consuming and may delay the verification process, as we will need a functional design in order to move forward and verify. There is also a possibility of underestimating the amount of time to debug RTL, as it proves to be more difficult/complex than our team initially anticipated.
    - Mitigation Plan: Clearly lay out the specifications of the design and ensure all team members are on the same page with how to interpret the design. Committing time to research informative resources to fulfill our team's understanding.
- There is a possibility that our team will have difficulties connecting to the remote lab and/or facing delays, freezes when connecting remotely.
    - Mitigation Plan: Starting deliverables early and avoiding peak hours so when we experience delays, we have time to wait it out or try to connect at a different time.
- Finding bugs late into the verification process that may force our team to make design changes when deadlines are tight (architecture closure).
    - Mitigation Plan: Our team needs to begin debugging the RTL as early as possible and make clear specifications/plans within the verification plan will help to try and get ahead of bugs that could be found later in the schedule.

5: Functions to be Verified

5.1 Critical Functions

These functions are to be verified before all others, as these provide the base set of tasks and behaviors of the DUT.

- **FULL and EMPTY Signals**: The FIFO signals when it is FULL or EMPTY in order to prevent overflow and underflow conditions. These signals depend on the WPTR and RPTR which look at their respective gray code counters to track the positions for writing and reading. FULL should be asserted when WPTR catches up to the RPTR. Empty should be asserted when RPTR catches up to WPTR
    - Write data until the FIFO is full, then attempt to write again.
    - Read data until the FIFO is empty, then attempt to read again
- **Handling of W/R Ratios**: When the FIFO is in situations where writing operations occur much faster than reading or vice versa, the FIFO will look to the EMPTY and FULL signals to manage W/R ratios. The FIFO should do this without losing data or reaching overflow/underflow conditions.
    - Simulate different W/R and R/W operations such as writing more frequently and reading more frequently.
- **FIFO Depth and Overflow/Underflow**: When the FIFO is halfway near or at its storage capacity limits, management of data needs to prevent too many reads from happening without enough data present and too many writes from happening without enough space for new data.

- **Cross Domain Data Transfer**: The FIFO performs operations asynchronously, with independent clocks for both READ and WRITE operations. The data read from the FIFO should match the data that was written.
    - Create an imitation FIFO within the scoreboard class to track results and known patterns and compare results with those of the main FIFO.
- **Reset Behavior**: The FIFO clears all pointers and flags to their default, stable states before the system begins to function. The FIFO should be empty and pointers should be at their default positions.
    - Apply a reset signal while data is being written or read, after different FIFO operations.
- **Pointer Synchronization**: The RPTR and WPTR are synchronized across different clock domains using dual flip-flops to avoid any metastability issues. The pointers need to be consistent in order to prevent READ and WRITE operations from interfering with each other.
- **Idle Cycle Management**: The FIFO should not enter erroneous states or lose data during idle periods so that it can effectively transition back to active operation when new READ or WRITE requests are received. The FIFO should maintain its current state during idle periods and resume normal operations without issue.

5.2 Secondary Functions

These functions are not critical to the next level of verification. If one of these functions are broken, the design is not “dead”.

- Operation robustness like invalid and error conditions

5.3 Non-Verified Functions

There are no functions in the DUT that may not be applicable at any level of verification or they have already been verified.

6: Tests and Methods

6.1 Type of Verification

Gray box verification will provide the ideal level of observation and controllability of the DUV. Observation points will be used to track any interesting behaviors during simulation and ensure that the functional specifications are being met/respected within the DUV structures. Instances where counter-initiated events need to be overwritten also fall under this verification category.

6.2 Verification Strategy

- **Deterministic**: Since our team is creating the RTL from scratch, this approach will be employed to initially verify basic functionality such as correct signaling of FULL/EMPTY states, R/W ratios, and data integrity. This allows for controlled testing of specific scenarios

- **Random-Based Simulation**: Constrained random testing to generate test inputs that are constrained by the FIFO specifications but random with those constraints for the breadth of testing.

- **Formal Verification**: This will be a little too extensive of testing for our DUV, as full on random testing for all permutations does not fit within our schedule/timeline.

6.3 Abstraction Level

The abstraction level that will be used at the start of the verification process is Register Transfer Level (RTL) based, since we want to ensure the FIFO functions correctly post-synthesis. Creating the FIFO from scratch might be the main driving point for choosing this abstraction level, and with consideration of our three-man team and limited timeframe for completion.

Transaction level abstraction should also be expected to be used in order to concentrate on R/W operations, control signals, and proper synchronization as they are important indicators of the current state and need to be monitored within the context of larger operations, for example. The focus is on the behavior and functionality of the design to provide a comprehensive view of the DUVs behavior and is applied through stimulus generation, scoreboard, monitors, etc.

6.4 Coverage Requirements

The intended stimulus goals are as follows:

- Including metrics for functional coverage, code coverage, and assertion coverage to get a sense of our verification quality and minimize coverage gaps
- Checking all critical data paths that affect data flow from WRITE to READ
- Checking that FIFO meets required throughput rates under various test conditions

6.5 Test Scenarios: Matrix

Test Reference #	Test Description	Function	Cross-Reference
T001	Validate signaling of FULL and EMPTY states	Signaling of FULL and EMPTY states	Functional Req, Coverage Goal for state signaling
T002	Test FIFO at various depth levels	FIFO Depth, overflow/underflow	Functional Req, Coverage Goal for depth handling
T003	Validate data integrity across transfers	Data integrity during transfers	Functional Req, Coverage Goal for data integrity
T004	Observe FIFO behavior following reset operations	Reset Behavior	Functional Req, Coverage Goal for reset behavior
T005	Test synchronization of READ and WRITE pointers	Pointer Synchronization	Functional Req, Coverage Goal for pointer sync
T006	Monitor FIFO during idle cycles	Idle cycle management	Functional Req, Coverage Goal for idle management
T007	Handling different WRITE/READ operation ratios	Handling of W/R ratios	Functional Req, Coverage Goal for W/R ratio handling
T008	Operation under invalid and error conditions	Operation robustness	Functional Req, Coverage Goal or error condition handling
T009	Boundary condition testing one element from full/empty	Critical boundary condition testing	Functional Req, Coverage Goal or boundary conditions
T010	High-frequency write/read stress test	FIFO performance under stress	Functional Req, Coverage Goal for performance testing
T011	Random reset during operation	Robustness to resets	Functional Req, Coverage Goal for random reset handling
T012	Abrupt changes in READ/WRITE rates	Pointer synchronization under stress	Functional Req, Coverage Goal for dynamic sync testing
T013	Throughput verification under varied conditions	Throughput and performance metrics	Functional Req, Coverage Goal for throughput testing

Table 1: Test Case Scenarios Matrix outlining test description and cross-reference to function and coverage list

6.6 Resources

Our team is made up of three graduate students. Work will be divided equally and tasks will be assigned early in order to coordinate best with due dates. Each student will need a computer in order to run simulations and track progress through our decided tools. One of these tools, QuestaSim, requires a license that is offered to students through PSU. By logging into the remote lab, access will be given.

7: Class-based Testbench and UVM

7.1 Introduction into Testbench Conversions

We started by converting our initial conventional testbench into a class-based stricture to improve modularity and reusability. This initial testbench was divided into driver, generator, monitor, scoreboard, testbench, and transaction classes in order to create reusable components that would then be incorporated into UVM. Here is how each class operates:

- **Top Level (module)**: Instantiates the verification environment
- **Test (module)**: Instantiates the environment and controls the overall verification process by configuring and managing the execution of different test scenarios
- **Environment (module)**: Encapsulates the set-up of the scoreboard, monitor, driver, and monitor
- **Driver**: This class retrieves transactions from the generator, applies them to the DUT, then sends the transactions to the monitor.
- **Generator**: This class creates and randomizes sequence items, sets the appropriate read and write enable signals, then sends them to the driver.
- **Monitor**: This class captures read and write transactions, monitors DUT signals, and sends observed transactions to the scoreboard for comparison. This is also where read operations are tracked and handled correctly.
- **Scoreboard**: This class compares observed DUT outputs with expected results in order to validate FIFO behavior by logging mismatches and errors.
- **Testbench**: This class creates handles for generator, driver, monitor, coverage, and scoreboard classes and passes the constructor for each so they can be executed.
- **Transaction**: This class encapsulates the data and control signals used in FIFO operations.

The architecture of our class-based testbench can be demonstrated by Figure 7.1 below:

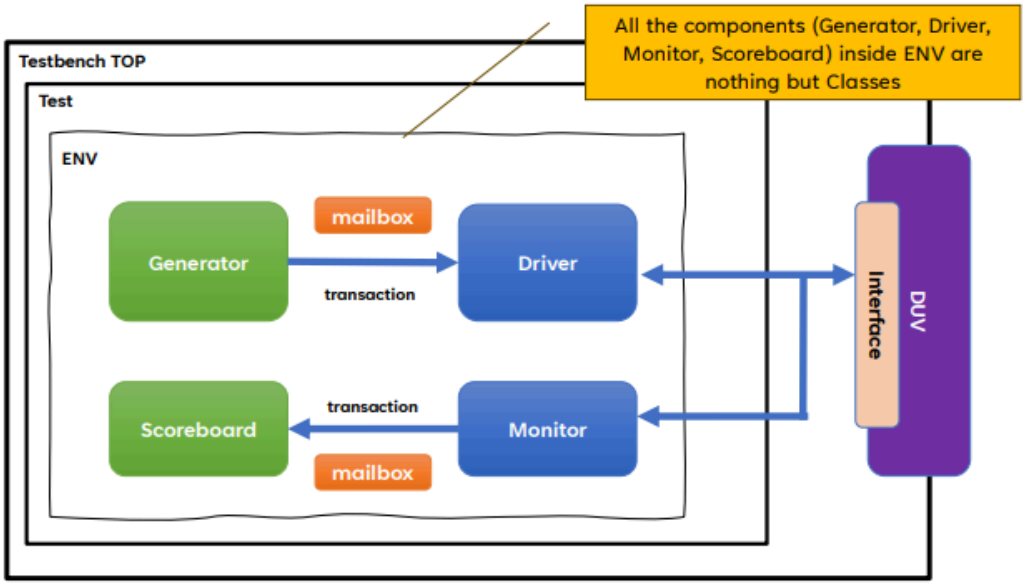

Figure 7.1: Class-based testbench architecture.

After creating our Class-based testbench architecture, we will convert it into its respective UVM structure. Most classes can be derived/inherited with the help of the '*extends*' keyword; these include driver, generator, environment, monitor, scoreboard, test, and transaction.

7.2 Architecture

UVM is made up of universal verification components and provides base library classes of each using the '*uvm_XXX*' designator; the testbench structure is well defined. Here are the core UVM components:

- **Top Level (module)**: Instantiates the verification environment.
- **Test (class)**: Instantiates the environment and controls the overall verification process by configuring and managing the execution of different test scenarios.
- **Environment (class)**: Encapsulates the set-up of the agent and thus the sequencer, monitor, and driver, as well as the scoreboard and coverage.
- **Agent (class)**: We use an active agent containing a sequencer, driver and monitor. The agent is used to hold and instantiate the above classes. We only have one interface and thus, only one agent.
- **Sequencer (class)**: Controls the execution of sequences and manages flow of sequence items to the driver.
- **Driver (class)**: Drives wr/rd enable and data in stimulus received by the sequencer by applying transactions to the DUT via the interface.
- **Monitor (class)**: Observes FIFO status signals by capturing results of read operations on the virtual interface.
- **Scoreboard (class)**: Receives the transaction from the monitor and validates the FIFO's functionality, (the data read from the FIFO is the data written to it), under various conditions.
- **Sequence (class)**: Create and configure transactions while also specifying the order in which each transaction should be sent. Transactions are stimulus then sent to driver

The architecture of our UVM testbench can be demonstrated by Figure 7.2 below:

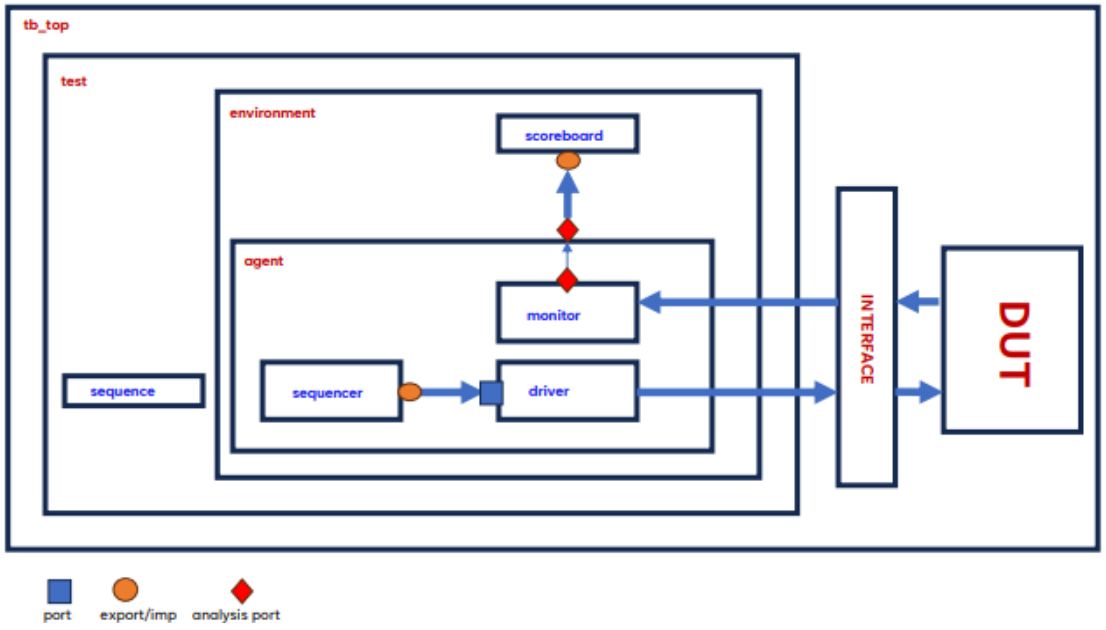

Figure 7.2: UVM testbench architecture.

7.3 Hierarchy

At the top of the UVM class hierarchy is '*uvm_object*' and it provides basic functionalities like copying, comparing, packing and unpacking, that are used for manipulating UVM objects. An extension of this class is '*uvm_sequence_item*'. This is a class that is used for creating transaction-level objects, like the control signals and data of single transactions in the FIFO. Next is the base class for all UVM components, grandchild of '*uvm_object*', '*uvm_component*'. This includes methods for managing the build, connect, run, and final phases of components. The core architectural components of our FIFO can be derived from this base class. Then there's the '*uvm_sequence*' class that is derived from '*uvm_sequence_item*'. These classes generate and manage the flow of transactions and will be creating the necessary read and write transactions to test the FIFO functionality. UVM provides four service mechanisms that can be used to help with the verification effort. The '*uvm_report_object*' and '*uvm_report_handler*' manage messages and reports within the environment and handle logs, warnings, and errors. The '*uvm_config_db*' manages the centralized configuration parameters for consistency and allows objects to be accessible to other components. Then, '*uvm_factory*' serves as a registry/ creation and override mechanism for UVM components and objects.

At the top level of our FIFO, UVM test instantiates the environment and controls the overall verification process, while the environment itself encapsulates key components. The agent, which includes a sequencer, driver, and monitor, manages the transaction-level operations, generating and applying read and write transactions to the DUT via virtual interface. The sequencer orchestrates the sequences of stimuli, the driver converts transactions into pin-level signals on the interface, and the monitor observes and captures the DUT's signals through the interface. The scoreboard validates these outputs against expected results. The

service mechanisms like the configuration database and the factory enhance flexibility and reusability, while the reporting mechanisms provide detailed logs for debugging.

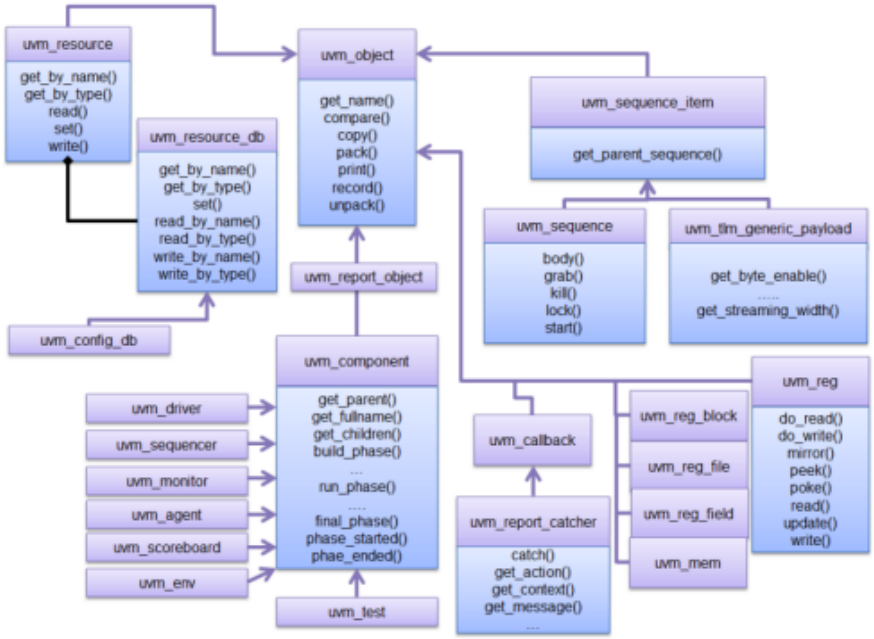

Figure 7.3: UVM hierarchy of classes diagram.

7.4 Components

The **uvm_test** component configures the environment and controls the verification process. It applies stimulus through UVM sequences and manages the overall execution of the testbench.
The test methods include:

- **Constructor '*new()*'**: Initializes the test by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Constructs the test components and creates an instance of the environment so it will be attached to the test.
- **End of Elaboration Phase '*end_of_elaboration_phase*'**: Finalizes testbench topology and prints the UVM topology to confirm this was done correctly.
- **Run Phase '*run_phase*'**: Serves as the main execution task for the test and is continuously running during simulation. An objection will be raised in order to keep the phase active. The sequence component is created and randomized and is started on the sequencer within the environment. A debugging message will be displayed to confirm successful execution and then the objection is dropped to signal the end of the phase.

The **uvm_environment** component encapsulates the agent and scoreboard and coordinates their interactions.
The environment methods include:

- **Constructor '*new()*'**: Initializes the environment by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Constructs the environment's subcomponents. The agent and scoreboard components are created and connected with the environment using the UVM factory.
- **Connect Phase '*connect_phase*'**: Establishes connections between the environment's components. The monitor's analysis port is connected to the scoreboards analysis export in order to allow for the transfer of observed transactions for comparison.
- **Run Phase '*run_phase*'**: Serves as the main execution tasks and runs continuously during simulation. A debugging message will be added to confirm operation.

The singular **uvm_agent** component will extend its corresponding class and encapsulate the sequencer, driver, and monitor. This agent coordinates the generation of stimulus, the application of that stimulus to the DUT, and observes the DUTs outputs.
The agent methods include:

- **Constructor '*new()*'**: Initialize the agent by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Construct the agent's subcomponents: sequencer, monitor, and driver. It creates an instance of these subcomponents and attaches them to the agent.
- **Connect Phase '*connect_phase*'**: Establishes connections between the components and displays a debugging message if unsuccessful. It connects the sequencer's export to the driver's port to allow the sequencer to send transactions to the driver.

The **uvm_sequencer** component is responsible for managing sequences of transactions. The sequencer will make sure the transactions from sequence cover all possible status states correctly and then provide them to the driver.
The sequencer methods include:

- **Constructor '*new()*'**: Initialize the sequencer by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Constructs the sequencer and confirms it was done correctly through a debugging message.
- **Connect Phase '*connect_phase*'**: Establishes connections between the components and confirms it was done correctly through a debugging message.

The **uvm_driver** extends its corresponding class and manages w/r enable and data in signals to ensure that they are correctly timed and applied to the DUT. It is executing the read and write commands generated by the sequencer.
The driver methods include:

- **Constructor '*new()*'**: Initialize the driver by setting its name and parent component and a debugging message to confirm it is correctly constructed.

- **Build Phase '*build_phase*'**: Constructs the driver and ensures the virtual interface is retrieved from the UVM configuration database (an error or debugging message will be generated if otherwise).
- **Connect Phase '*connect_phase*'**: Establishes connections between the driver's components and displays a debugging message if unsuccessful.
- **Run Phase '*run_phase*'**: Serves as the main execution tasks for the driver and is continuously running during simulation. It retrieves the next transactions from the sequencer, then on the positive edge of the write clock, it drives the '*data_in*' and '*wr_en*' signals to the DUT. On the negative edge of the write clock, it updates the status flags corresponding to the transaction. Once a transaction is finished, '*seq_item_port.item_done()*' is set and a debugging message is applied.

The **uvm_monitor** passively observes the DUT interface and captures transactions to be forwarded to the scoreboard. It will track read transactions and make sure they are correctly captured and analyzed.
The monitor methods include:

- **Constructor '*new()*'**: Initialize the monitor by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Constructs the monitor and ensures the virtual interface is retrieved from the UVM configuration database (an error or debugging message will be generated if otherwise). This is also where the *'monitor_port'* will be instantiated in order to allow the monitor to send captured transactions to other components.
- **Connect Phase '*connect_phase*'**: Establishes connections between the monitor's components and displays a debugging message if unsuccessful.
- **Run Phase '*run_phase*'**: Serves as the main execution task for the monitor and is continuously running during simulation. The monitor will wait for the driver to reset and for data to be written to the FIFO. Then capture read transactions by observing the '*rd_en*' and '*data_out*' signals and follow this up with updating '*empty*', '*full*', and half signals based on the FIFO's status. Finally, the captured transaction will be sent to the '*monitor_port*' so it can be forwarded to the scoreboard.

The **uvm_scoreboard** method is responsible for checking the correctness of the DUT's behavior by comparing its outputs with the expected results with a "fake/mimic" FIFO. The scoreboard will check if data read from the FIFO matches the data written to it.
The scoreboard methods include:

- **Constructor '*new()*'**: Initialize the scoreboard by setting its name and parent component and a debugging message to confirm it is correctly constructed.
- **Build Phase '*build_phase*'**: Constructs the scoreboard and retrieves the virtual interface from the UVM configuration database (an error or debugging message will be generated if otherwise).

The **uvm_sequence** component is responsible for generating sequences of transactions, both read and write transactions, to test the FIFO functionality. It defines the behavior and order of these sequence items that are to be sent to the DUT.
A task will be used for generating and sending transactions. It will check if 'starting_pagse' is null, and if it is not, an objection is raised to keep it active until the sequence completes. Write transactions, '*tx_wr*' are created, randomized, and sent to the driver. Read transactions, '*tx_rd*' are created and sent to the driver. If the 's*tarting_phase*' is still not null, the objection is dropped and signals the end of the sequence.
The sequence methods include:

- **Constructor '*new()*'**: Initializes the sequence and calls the superclass constructor to ensure it was done properly.

The **uvm_sequence_item** represents a single transaction in the UVM environment. This transaction is used to transfer data between the sequencer, driver, and monitor. It holds the details of the transaction, like control signals, data, and status flags.
The sequence item methods include:

- **Constructor '*new()*'**: Initializes the transaction and calls the superclass constructor to ensure it was done properly.

8: Schedule and Responsibilities

8.1 Week of 4/15-4/21/2024

This week's focus is on getting together as a team and going through project documents in order to determine what our final project will be. This is important as Milestone #1 will be due. Here are the team responsibilities for the week:

Nick Allmeyer:

- Read Clifford E. Cummings FIFO1 and FIFO2 papers
- Meet with team on 4/17 to discuss FIFO choice
- Meet with team on 4/19 to discuss FIFO implementation

Alexander Maso:

- Read Clifford E. Cummings FIFO1 and FIFO2 papers
- Meet with team on 4/17 to discuss FIFO choice
- Meet with team on 4/19 to discuss FIFO implementation

Ahliah Nordstrom:

- Read Clifford E. Cummings FIFO1 and FIFO2 papers
- Meet with team on 4/17 to discuss FIFO choice
- Meet with team on 4/19 to discuss FIFO implementation

8.2 Week of 4/22-4/28/2024

This week's focus is on completing Milestone #1 and starting Milestone #2 deliverables for submission on 4/24 and 4/30, respectively. Here are the team responsibilities for the week:

Nick Allmeyer:

- Perform FIFO calculations with our teams specifications
- Create conventional testbench for DUT

Alexander Maso:

- Create initial DUT files
- Ensure DUT files compile and debug if necessary

Ahliah Nordstrom:

- Create Github repository and Google Drive for team
- Put together and Verification Plan and Specifications documents
- Update documents later in week if there are any nuances
- Support others when needed

8.3 Week of 4/29-5/5/2024

This week's focus is on completing Milestone #2 deliverables for submission on 4/30. Here are the team responsibilities for the week:

Nick Allmeyer:

- Test FIFO top module within class based testbench
- Test FIFO read/write pointers within class based testbench
- Support others when needed

Alexander Maso:

- Test FIFO read/write pointers within class based testbench
- Polish the Class Based testbench and ensure all interfaces are tested
- Support others when needed

Ahliah Nordstrom:

- Update version control sites and Verification Plan
- Test control and memory unit interface within class based testbench
- Support others when needed

8.4 Week of 5/6-5/12/2024

This week's focus is on completing Milestone #3 deliverables for submission on 5/14. Here are the team responsibilities for the week:

Nick Allmeyer:

- Create a working driver and monitor for complete class based verification testing
- Polish off complete class-based testbench and ensure all components are defined and working
- Support others when needed

Alexander Maso:

- Create a working transaction tester for complete class based verification testing
- Finalize any RTL changes, ensure coverage goal is as close to 100% as possible
- Support others when needed

Ahliah Nordstrom:

- Update version control sites and Verification Plan
- Create a working scoreboard for complete class based verification testing
- Support others when needed, particularly with ensuring all components are defined and working in testbench

8.5 Week of 5/13-5/19/2024

This week's focus is on completing Milestone #3 and starting Milestone #4 deliverables for submission on 5/14 and 5/28 respectively. Here are the team responsibilities for the week:

Nick Allmeyer:

- Create a working driver and monitor for complete class based verification testing
- Polish off complete class-based testbench and ensure all components are defined and working
- Support others when needed

Alexander Maso:

- Create a working transaction tester for complete class based verification testing
- Finalize any RTL changes, ensure coverage goal is as close to 100% as possible
- Support others when needed

Ahliah Nordstrom:

- Update version control sites and Verification Plan in light of any nuances
- Finish working scoreboard for complete class based verification testing
- Support others when needed, particularly with ensuring all components are defined and working in testbench

8.6 Week of 5/20-5/26/2024

This week's focus is on completing Milestone #4 deliverables for submission on 5/28. Here are the team responsibilities for the week:

Nick Allmeyer:

- Ensure that UMV_MESSAGING and UMV_LOGGING mechanisms are utilized
- Collect log and reports data from UVM mechanisms
- Ensure testbench respects UVM testbench architecture and create monitor in testbench

Alexander Maso:

- Create skeleton for UVM testbench architecture for the FIFO
- Incorporate UVM sequencer and driver to testbench
- Ensure testbench respects UVM testbench architecture

Ahliah Nordstrom:

- Update version control sites
- Incorporate UVM scoreboard and FIFO interfaces in the testbench
- Add UVM section in Verification Plan and add related details

8.7 Week of 5/27-6/2/2024

This week's focus is on completing the final Milestone #5 deliverables and final project presentation for submission on 5/30 AND/OR 6/4. Here are the team responsibilities for the week:

Nick Allmeyer:

- Create UVM environment
- Ensure all testcases are completed and reflected in the coverage reports
- Support others when needed

Alexander Maso:

- Polish UVM testbench and ensure UVM architecture is respected
- Create scenarios of bug-injection and verify results
- Support others when needed

Ahliah Nordstrom:

- Update version control sites, Verification Plan, and specification documents
- Begin final report and presentation formatting
- Create scenarios of bug-injection and verify results
- Support others when needed

8.8 Week of 6/3-6/9/2024

This week's focus is on preparing for our final presentation on 6/4 OR 6/6. Here are the team responsibilities for the week:

Nick Allmeyer:

- Finish final report and presentation formatting
- Support others when needed

Alexander Maso:

- Finish final report and presentation formatting
- Support others when needed

Ahliah Nordstrom:

- Update version control sites, Verification Plan, and specification documents
- Finish final report and presentation formatting
- Support others when needed

9: References

[1] “Crossing clock domains with an Asynchronous FIFO,” zipcpu.com. https://zipcpu.com/blog/2018/07/06/afifo.html. [Accessed April 20, 2024]

[2] C. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers.* March 2002, Vol. 281.

[3] R. Salemi, “The UVM Primer: An Introduction to the Universal Verification Methodology”. Boston: Boston Light Press, 2013. Accessed: Apr. 1, 2024. [Online].

[3] B. Wile and J. Gross and W. Roesner, “Comprehensive Functional Verification: The Complete Industry Cycle”. San Francisco, CA, USA: Morgan Kaufmann, 2005. Accessed: Apr. 1, 2024. [Online].

[4] J. Yu, “Dual-Clock Asynchronous FIFO in SystemVerilog,” verilogpro.com. https://zipcpu.com/blog/2018/07/06/afifo.html. [Accessed April 30, 2024]

[5] P. Venkatesh. (2024). Lec-10-Essential_UVM_Components-Factory_Part1 [PDF]. Available: https://canvas.pdx.edu/courses/84550/files/11036314?module_item_id=4022467