Asynchronous FIFO Design and Verification using UVM

Nick Allmeyer, Alexander Maso, Ahliah Nordstrom
Department of Electrical and Computer Engineering, Portland State University

Abstract. The design and verification of Asynchronous FIFO memory systems are crucial for ensuring reliable data transmission between clock domains, in particular. This paper presents a comprehensive study of the implementation and verification of an asynchronous FIFO, incorporating minimal enhancements to Clifford E. Cummings design. These simple enhancements and inspirations include improved pointer management, synchronization logic, and flexible memory buffer configurations. We utilized both class-based and Universal Verification methodology (UVM) approaches for the verification process. The class-based approach provided initial simplicity and direct communication between components, while the UVM approach offered a structured, modular framework that facilitated effective debugging and scalability. Through extensive testing, including bug injections and coverage analysis, we demonstrated the effectiveness of UVM in managing complex verification tasks. The paper concludes with a comparative analysis of the two verification methodologies in order to emphasize the superior capabilities of UVM for comprehensive and intricate verification of an asynchronous FIFO design.

1. Introduction

The design and verification of digital systems has become increasingly complex with the rapid advancement of integrated circuits. Among the critical components in such systems, there is one that stands out for its reliability of data transmission between clock domains: an asynchronous First-In-First-Out (FIFO) memory system. These systems are often used to address metastability and data integrity issues that arise from signals crossing clock domains. Clifford E. Cummings lays out an asynchronous FIFO design that safely passes data from one clock domain to another asynchronous clock domain. This paper details the implementation and verification of his fundamental design in SystemVerilog, with additional specifications and adjustments to these specifications using class-based and Universal Verification Methodology (UVM) techniques.

Our asynchronous FIFO design includes memory buffer configurations, improved pointer management for handling increments and flag conditions, and a unified synchronization logic. To verify the correctness of our design, our class-based and UVM approach allows for a scalable and reusable verification process that helps to facilitate thorough testing of the FIFOs functionality. This paper will discuss the implementation and verification of our asynchronous FIFO design while also highlighting our team's contributions and results of our verification efforts.

2. FIFO Design and Implementation

Our FIFO design builds upon Clifford E. Cummings foundational FIFO design, integrating his framework while also introducing enhancements, including a half-full flag.

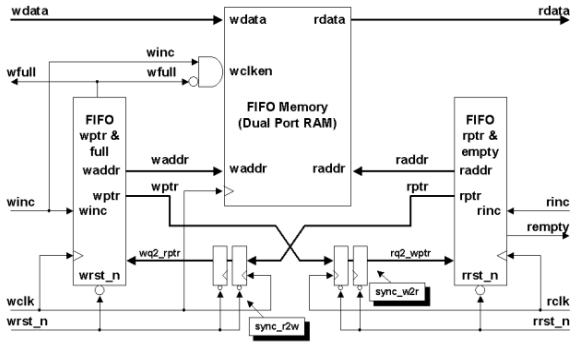

Figure 1: C. Cummings high-level asynchronous FIFO design

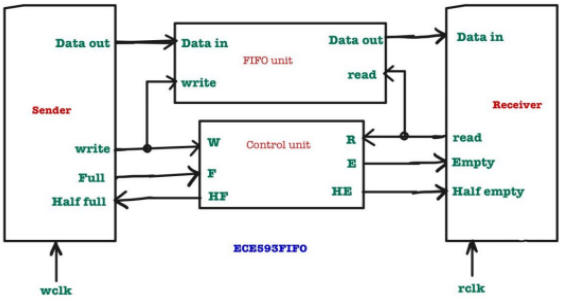

Figure 2: Our high-level asynchronous FIFO design

A. Memory Buffer

We utilized dual-port RAM so simultaneous read and write operations from different clock domains can occur. This architecture is needed for asynchronous operation and enables the FIFO to manage data transfer efficiently without causing contention. This approach follows Cummings' design principles, where dual-port RAM is used to facilitate concurrent access.

The dual-port RAM is implemented using a two-dimensional array to allow reliable data storage and retrieval. The memory array is parameterized so data width and depth is flexible. We optimized memory access latency from Cummings FIFO by reducing wait states.

B. Pointer Management

Effective management of read and write pointers ensures correct operation of the FIFO. Our design incorporates Gray code encoding for pointer representation to minimize metastability issues during clock domain crossing, as suggested by Cummings.

The write pointer module manages the address for write operations. Data is written sequentially and accurately, with additional logic added to detect when the FIFO is full. This prevents further writes to the FIFO and avoids any overflow.

The read pointer module handles the address for read operations. Data is read sequentially and there is additional logic for detecting when the FIFO is empty. This prevents underflow from occurring.

C. Pointer Management

Synchronization between the read and write clock domains is crucial for the FIFO. Our design, taking inspiration from Cummings, employs double-flop synchronizers to safely transfer pointer values between clock domains.

The synchronization technique involves passing data through two consecutive flip-flops that help to mitigate the risk of metastability. This is done by allowing any metastable state to resolve before the data is used in the receiving clock domain.

3. Basic Testbench

A conventional SystemVerilog testbench was developed to verify the functionality of our asynchronous FIFO early in our verification process. The creation of this initial testbench was to test our design at the most basic level with components and processes for clock generation, reset initialization, randomized data generation, and coverage checking.

The test process involved random toggling of write and read enable signals with corresponding data inputs, allowing the FIFOs behavior under various conditions to be evaluated. A coverage group monitored the FIFOs operation so that test conditions could be checked if they were exercised. A scoreboard mechanism verified data integrity by comparing

the FIFOs output data with expected values stored in local memory.

The setup of this basic testbench provided a solid foundation for more advanced verification methodologies. After successfully compiling our design and testbench code and verifying our work, we moved on to creating a class based testbench.

4. Class Based Verification

Class-based verification leverages object-oriented programming (OOP) to create modular, reusable, and scalable verification environments. This methodology enhances the ability to handle complex verification scenarios by separating different aspects of the testbench into distinct classes.

A. Testbench Architecture

The class-based testbench for our asynchronous FIFO design includes these key components: generator, driver, monitor, scoreboard, and coverage. These components are organized within an environment that facilitates communication and interaction among them. They collectively work together to simulate, observe, and verify the DUT.

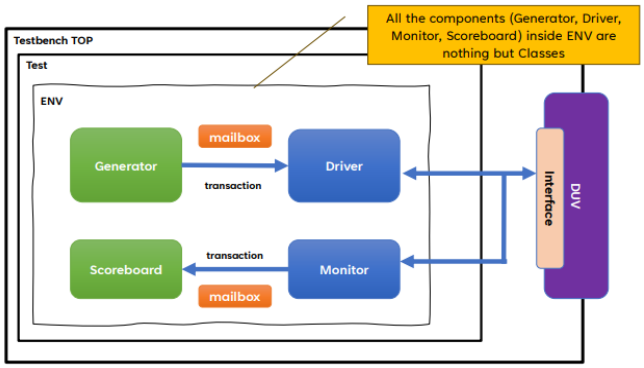

Figure 3: Class-based testbench architecture.

B. Testbench Components

Top Level is a module that instantiates the verification environment.

Test is a module that instantiates the environment and controls the overall verification process by configuring and managing the execution of different test scenarios.

Environment serves as the central structure that contains the major testbench components. Each component is implemented as a class, interacting and synchronizing through transactions and mailboxes.

Generator creates randomized transactions and sends them to the driver. It ensures that a wide range of inputs are tested.

Driver converts high-level transactions into signal-level activity for the DUT. It retrieves transactions from the generator and applies them to the DUT.

Monitor observes the DUT outputs and collects data for analysis. It captures the relevant signals and sends observed transactions to the scoreboard for comparison.

Scoreboard compares observed DUT outputs with expected results in order to validate FIFO behavior by logging mismatches and errors.

Transaction encapsulates the data and control signals used for FIFO operations.

Interface provides a communication channel between the testbench components and the DUT. It defines the signals and ports that connect the DUT to the testbench for consistent and organized interaction.

4.1 Challenges and Lessons Learned

After implementing and simulating the testbench, there was an issue observed where the scoreboard would be off by one cycle. This issue showed itself to be a result of the read pointer pointing to stale data. To address this issue, it was necessary to check that the read pointer updates accurately when the empty signal is asserted. From here, we decided to add logic to check the empty flags state and delay updating the read pointer until the data is valid. This would allow for the pointer to point to correct data and maintain synchronization between the read operations in the scoreboard.

There was an additional issue that persisted while implementing our class-based testbench, regarding the order of file inclusion. Simulation errors were stemming from dependencies not being correctly resolved in the '*run.do*' file. This simple error was resolved by reordering the file inclusions, using a "bottom-up" approach. By structuring lower-level dependencies before high-level modules, the testbench ran without any errors.

Another challenging aspect of the class-based testbench were mailboxes. Transactions sent between mailboxes required separate paths for read and write domains. Initially, using a shared path led to contention and synchronization issues that ultimately affected the performance and accuracy of transactions. We were able to address this by creating separate mailbox paths for read and write transactions. This made it so each domain operated independently.

5. UVM Based Verification

After creating our Class-based testbench architecture, we converted it into its respective UVM structure. Most classes can be derived/inherited from pre-existing UVM classes with the help of the '*extends*' keyword; these include driver, environment, monitor, scoreboard, test, and transaction (sequence item). As well as a few new classes, agent, sequence and sequencer.

A. Testbench Architecture

The UVM based testbench for our asynchronous FIFO design consists of several layers and components organized within a structured environment. The primary components include sequence, sequencer, driver, monitor, scoreboard, and interface. These components are all coordinated by the environment and agent.

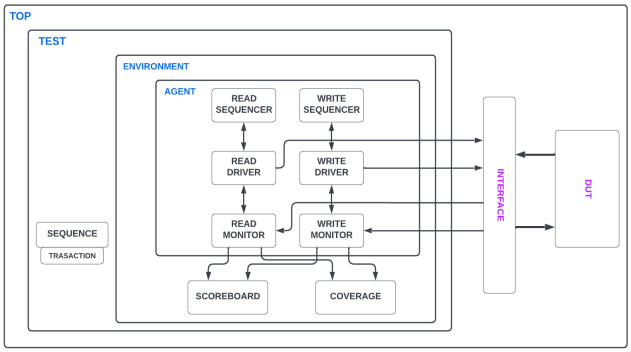

Figure 4: UVM testbench architecture.

B. Testbench Components

Top instantiates the verification environment.

Test sets up and runs the test by coordinating various phases of the UVM simulation, extending '*uvm_test*'. It is the top-level of the verification environment, making sure the FIFO is thoroughly exercised by generating write and read transactions and verifying their respective handling.

Environment encapsulates the set-up of the agent and thus the sequencer, monitor, and driver, as well as the scoreboard and coverage.

Agent contains the sequencers, drivers, and monitors for both read and write domains. It coordinates their activities to generate stimulus and captures responses from the DUT. During its build phase, instances of these components are

created and configured. In the connect phase, the driver's sequence item ports are connected to the corresponding sequencer export ports. This facilitates the flow of transactions from the sequencer to the driver.

Read Driver handles read transactions separately from write transactions by extending the '*uvm_driver*'. The run phase fetches read transactions and drives the read enable signal to the DUT at the positive edge of the read clock. The build phase obtains the interface handle from the UVM configuration database. The connect phase logs the connection status for debugging purposes.

Write Driver handles write transactions separately from read transactions by extending the '*uvm_driver*'. The run phase continuously fetches write transactions from the sequencer and drives them to the DUT. It waits for the positive edge of the write clock to update the data in and write enable signals of the interface, completing the transaction. The build phase and the connect phase mirror their respective read domain counterparts.

Read Monitor handles monitoring of read operations separately from write operations by extending the '*uvm_monitor*'. The run phase continuously creates and populates read transactions based on the DUTs read signals. It then creates the analysis port and writes these transactions to it in order to allow the scoreboard access. The build phase obtains the interface handle from the UVM configuration database. The connect phase logs the connection status for debugging purposes.

Write Monitor handles monitoring of write operations separately from read operations by extending the '*uvm_monitor*'. The run phase continuously creates and populates write transactions based on the DUTs write signals. It then writes these transactions to the respective analysis port to allow the scoreboard to access. The build phase and the connect phase mirror their respective read phase counterparts.

Sequencer manages the flow of transactions from the sequence to the driver by extending the '*uvm_sequencer*'. It acts as the intermediary that sequences transactions and ensures they are correctly managed and forwarded to the driver.

Read Sequence generates sequences of read transactions separately from write transactions by extending the '*uvm_sequence*'. It begins by raising an objection, then creates and randomizes read transactions with the operation set to READ. They are stated and finished, driving read enable signals to the DUT. The objection is dropped at the end of a sequence to signal completion.

Write Sequence generates sequences of write transactions separately from read transactions by extending the '*uvm_sequence*'. It begins by raising an objection, then creates and randomizes read transactions with the operation set to READ. They are stated and finished, driving read enable signals to the DUT. The objection is dropped at the end of a sequence to signal completion.

Transaction extends the '*uvm_sequence_item*' and represents the basic unit of data transfer in the UVM environment. All necessary signals and data for a single transaction are encapsulated here.

Coverage collects and reports coverage data by extending the '*uvm_subscriber*'. The coverage class is used to measure and record the features/functional properties of the DUT. The stimuli from the tests are packaged in transactions and sent from the monitor class to the coverage class via an analysis broadcast port.

Scoreboard compares the expected results with actual results of all operations. It verifies that the data read from the FIFO matches the data written to it.

Interface provides a communication channel between the testbench components and the DUT. It defines the signals and ports that connect the DUT to the testbench for consistent and organized interaction.

C. Hierarchy

At the top of the UVM testbench hierarchy is top.sv, which encompasses the entire verification environment. Here is where our FIFO environment acts as a container for the agent, scoreboard, and coverage components. Then lies the FIFO agent to generate stimulus for the DUT and collect responses while facilitating a controlled data flow through the sequencer. Then there's the scoreboard to verify data comparisons through analysis ports from the monitor. This comparison checks for data mismatches and correct operation. Then there's the coverage component that collects coverage data, checking if relevant scenarios are tested and identifying any untested areas.

Name	Type	Size	Value
uvm_test_top	my_first_test	-	@471
environment_h	fifo_environment	-	@478
agent_h	fifo_agent	-	@485
driver_rd_h	fifo_read_driver	-	@774
rsp_port	uvm_analysis_port	-	@789
seq_item_port	uvm_seq_item_pull_port	-	@781
driver_wr_h	fifo_write_driver	-	@751
rsp_port	uvm_analysis_port	-	@766
seq_item_port	uvm_seq_item_pull_port	-	@758
monitor_rd_h	fifo_read_monitor	-	@744
monitor_port_rd	uvm_analysis_port	-	@800
monitor_wr_h	fifo_write_monitor	-	@737
monitor_port_wr	uvm_analysis_port	-	@809
sequencer_rd_h	fifo_sequencer	-	@628
rsp_export	uvm_analysis_export	-	@635
seq_item_export	uvm_seq_item_pull_imp	-	@729
arbitration_queue	array	0	-
lock_queue	array	0	-
num_last_reqs	integral	32	'd1
num_last_rsps	integral	32	'd1
sequencer_wr_h	fifo_sequencer	-	@519
rsp_export	uvm_analysis_export	-	@526
seq_item_export	uvm_seq_item_pull_imp	-	@620
arbitration_queue	array	0	-
lock_queue	array	0	-
num_last_reqs	integral	32	'd1
num_last_rsps	integral	32	'd1
coverage_h	fifo_coverage	-	@499
analysis_imp	uvm_analysis_imp	-	@506
scoreboard_h	fifo_scoreboard	-	@492
scoreboard_port_rd	uvm_analysis_imp_port_b	-	@835
scoreboard_port_wr	uvm_analysis_imp_port_a	-	@827

Figure 5: UVM testbench topology.

5.1 Challenges and Lessons Learned

Similar to the class-based implementation, the UVM testbench presented synchronization challenges. One involved the management of clock domains. Data and transactions were lost due to inconsistent fluctuations in the timing of the read and write clocks. To resolve this, we created an additional Driver, Monitor, and Sequence for each clock domain (read and write). This separation allowed for each domain to be managed independently and correctly handle persisting timing discrepancies.

6. Test Scenarios and Coverage

6.1 Bug Injections and Detections

A. Bug Injection

In our design, we introduced a bug to test the effectiveness of our UVM testbench. The specific bug involved incorrectly specifying the size of our write pointer in the FIFO write pointer module to intentionally simulate an accidental typo, a common scenario that can occur in any FIFO implementation.

In a correctly functioning FIFO, the write pointer increments with each write operation until it hits the maximum depth of the FIFO, at which point it wraps around to 0. This is so that each new data entry is written to a uniquely new location in the buffer. However, in our bug injection scenario, the write pointer only has a width of 5 bits and therefore wraps back around and writes over data prematurely.

In addition to writing over data prematurely, when the read pointer increments past the 32nd address it points to uninitialized data. This causes the data out on the next reads (and the following 32 reads) to be unknown or undefined causing a mismatch in the scoreboard.

B. Bug Detection

Our testbench successfully identified the intentional write pointer bug. During simulation, it became evident from the transcript that the bug was present, both during initial compilation and by means of the scoreboard. The detailed logs and coverage reports pinpointed the issue to the write pointer, specifically the incorrect port sizing between the write pointer module and the rest of the DUT.. By analyzing the output data, we confirmed that the fault was isolated to the failure to increment the write pointer.

6.2 Test Matrix and Scenarios

A test scenario matrix was created in the beginning of the verification process, attached to the Verification Plan. This matrix details twelve testing scenarios that are designed to validate specific aspects of the FIFO's functionality and robustness.

Test #	Description
T001	Validate Full and Empty
T002	Test fifo at various memory depths
T003	Validate a range of data values
T004	Observe behavior after reset
T005	Test the Synchronization of the read and write pointers
T006	Observe behavior during idle cycles
T007	Observe different ratios of reading and writing
T008	Operation under errors
T009	Validation testing between the full and empty flags
T010	High frequency Read and Write tests
T011	Random reset during operation
T012	Verifying the throughput

Table 1: Test Case Scenarios Matrix outlining test description

6.3 Implicit and Functional Coverage

A. Implicit Coverage

We achieved a final implicit coverage percentage of 100%. This implicit coverage is specifically with respect to branches, statements, and expressions with respect to all modules, packages, and interfaces used to create the asynchronous fifo. The only notable hiccup in achieving this coverage number came with respect to the task in the fifo bus functional model that reset the fifo. Despite the reset task being called in the driver class, none of the statements in the actual task were being covered. We eventually resolved this by calling the reset fifo task in the fifo bus function model file.

B. Functional Coverage

We achieved a final functional coverage percentage of 100%. We split our coverage into four different covergroups. The first covergroup is cg_fifo, which includes coverpoints for the read and write clocks, read and write enable signals, and the full, empty, and half flags. Two of our covergroups, cg_data_range and cg_data_patterns related to covering different possible values for the data being written into the fifo. The covergroup cg_data_range splits the possible 256 bit values the data could take on into four mutually exclusive values whose aggregate is the set of all possible values a byte can have. This covergroup thus verified that we were getting the full range of possible data values. The covergroup cg_data_patterns was influenced by Ray Salemi's code in the UVM Primer. Data in is covered for bit values of all zeros, all ones and alternating ones and zeros. Finally the covergroup cg_abrupt_change includes all the signals featured in cg_fifo in transition bins from both values of high to low, and low to high. This allows us to see the signal transitions as well as how many times they

occur in the coverage report. We faced one major issue with respect to the functional coverage and that was with respect to getting the low signals to appear for both clocks as well as the read and write enable signals. We eventually realized this was because everything in our design responds to positive edges, thus the clock signals and enable signals would never be low when handling transactions. The solution to this problem was to pass both the read and write signals to both the read and write monitors. By doing this in conjecture with the specifications and volume of our tests, we were guaranteed to see the coverage values we desired.

7. Overall Findings

7.1 Bug Injection

In the bug injection, the first 32 write and read operations to the FIFO were correctly executed and subsequent write operations overwrote the initial data entries. This behavior led to only the first 32 entries being correctly stored in the FIFO, with all following writes overwriting these initial entries. This made the FIFO unable to store more than a handful of unique pieces of data at a time.

7.2 Verification Approaches

The class-based verification approach provided simplicity and direct communication between components, by means of mailboxes. This was beneficial for basic transaction management, however, synchronization issues and the complexity of shared mailbox paths for read and write transactions persisted with our implementation. In contrast, the UVM approach offered a more structured and modular framework. UVM's hierarchical nature made it easy for debugging and scalability thus making it more suitable for complex verification tasks.

8. Conclusion

The implementation and verification of the asynchronous FIFO using both class-based and UVM methodologies proved to be an insightful contrast between the two approaches. This FIFO project revealed that while class-based verification is simpler and more straightforward, it struggles with managing complex synchronization issues inherent in asynchronous designs. The use of mailboxes, although effective for basic communication, encountered difficulties in handling separate read and write transactions without contention and synchronization problems. Despite these challenges, class-based verification was beneficial for understanding the fundamental behavior of the FIFO design.

As for UVM, there are significant advantages in managing complexity and enhancing scalability. UVM's structured and modular framework allowed for effective isolation and resolution of synchronization issues, promoting accurate and reliable verification. The hierarchical nature of UVM helped in debugging and maintaining the testbench, making it more suitable for comprehensive and intricate verification tasks. Additionally, UVM's advanced features for coverage collections provided a thorough validation of the FIFO design. The UVM verification approach proved to be more efficient for verifying this complex asynchronous FIFO design.

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

[6] Doulos Training, *Easier UVM Video Tutorial.* (May 14, 2012). Accessed: May 27, 2024. [Online Video]. Available: https://www.youtube.com/playlist?list=PLBIILfL2t1lnvzw7vF0arlvu36Wj4--D7