Asynchronous FIFO Design and Verification using UVM

Anurag Ranga, Yashaswi Katne, Naveen Kumar Reddy Thummala, Maheshwar Reddy Kamalapuram
Department of Electrical and Computer Engineering, Portland State University

1. Introduction

The design and verification of digital systems have grown increasingly complex due to the rapid advancements in integrated circuits. Among the critical components in these systems, one stands out for its reliability in data transmission between clock domains: an asynchronous First-In-First-Out (FIFO) memory system. These systems are essential for addressing metastability and data integrity issues that occur when signals cross clock domains. Clifford E. Cummings presents an asynchronous FIFO design that securely transfers data from one clock domain to another asynchronous clock domain. This paper details the implementation and verification of his foundational design in SystemVerilog, including additional specifications and modifications using class-based techniques and Universal Verification Methodology (UVM).

Our asynchronous FIFO design features memory buffer configurations, enhanced pointer management for handling increments and flag conditions, and unified synchronization logic. To verify the accuracy of our design, we employ a class-based and UVM approach, allowing for a scalable and reusable verification process that ensures comprehensive testing of the FIFO's functionality. This paper will explore the implementation and verification of our asynchronous FIFO design.

2. FIFO Design and Implementation

Our FIFO design builds upon Clifford E. Cummings' foundational FIFO framework, incorporating his design while also introducing several enhancements.

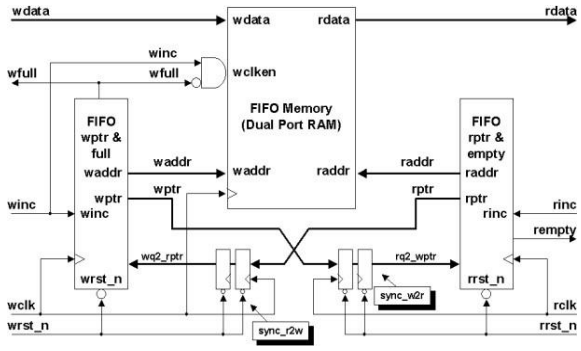

Figure 1: C. Cummings high-level asynchronous FIFO design

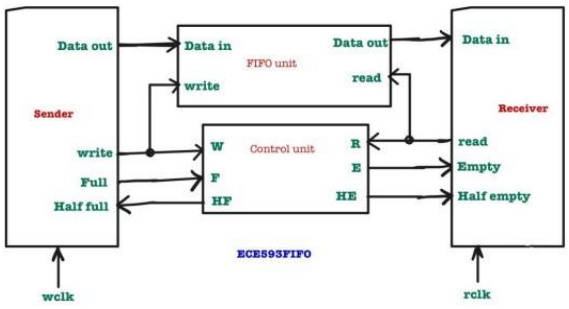

Figure 2: Our high-level asynchronous FIFO design

A. Memory Buffer

We utilize dual-port RAM to enable simultaneous read and write operations from different clock domains. This architecture is essential for asynchronous operation, allowing the FIFO to manage data transfer efficiently without contention.

This approach aligns with Cummings' design principles, where dual-port RAM facilitates concurrent access.

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

Pointer Management

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

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

The read pointer module handles the address for read operations. Data is read sequentially, with additional logic to detect when the FIFO is empty, preventing underflow.

Synchronization

Synchronization between the read and write clock domains is critical for FIFO functionality. Our design, inspired by Cummings, employs double-flop synchronizers to safely transfer pointer values between clock domains.

This synchronization technique involves passing data through two consecutive flip-flops, which helps mitigate the risk of metastability 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 initially developed to verify the functionality of our asynchronous FIFO. This testbench was designed to evaluate our design at the most basic level, incorporating components and processes for clock generation, reset initialization, randomized data generation, and coverage checking.

The test process involved random toggling of the write and read enable signals with corresponding data inputs, allowing us to evaluate the FIFO's behavior under various conditions. A coverage group monitored the FIFO's operation to ensure that all test conditions were exercised. Additionally, a scoreboard mechanism verified data integrity by comparing the FIFO's 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 proceeded to create 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 improves the ability to handle complex verification scenarios by separating different aspects of the testbench into distinct classes.

Testbench Architecture

The class-based testbench for our asynchronous FIFO design includes the following key components: generator, driver, monitor, scoreboard, and coverage. These components are organized within an environment that facilitates communication and interaction among them. Together, they simulate, observe, and verify the design under test (DUT).

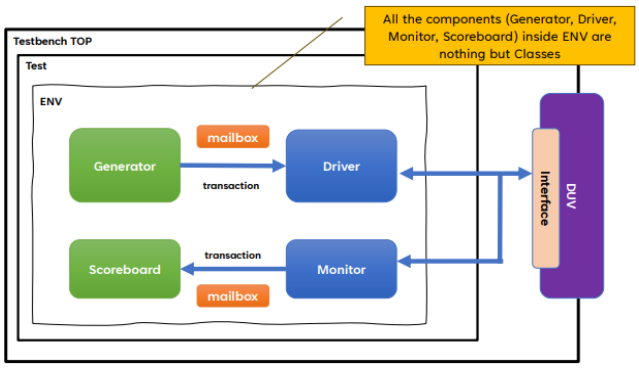

Figure 3: Class-based testbench architecture.

A. Testbench Components

- Top **Level:** A module that instantiates the verification environment.
- Test**:** 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 containing 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, ensuring a wide range of input scenarios are tested.
- Driver**:** Converts high-level transactions into signal-level activity for the DUT. It retrieves transactions from the generator, applies them to the DUT, and then sends the transactions to the monitor.
- Monitor**:** Observes the DUT outputs and collects data for analysis. It captures read and write transactions, sends observed transactions to the scoreboard for comparison, and ensures read operations are tracked and handled correctly.
- **Scoreboard:** Compares observed DUT outputs with expected results to validate FIFO behavior, 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.

5. UVM Based Verification

After developing our class-based testbench architecture, we converted it into its respective UVM structure. Most classes were derived or inherited using the 'extends' keyword, including the driver, generator, environment, monitor, scoreboard, test, and transaction.

Testbench Architecture

The UVM-based testbench for our asynchronous FIFO design comprises 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.

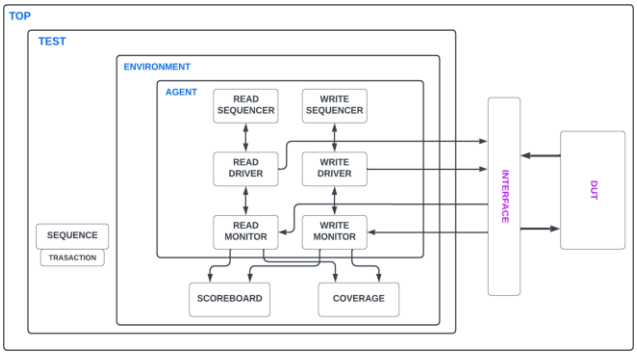

Figure 4: UVM testbench architecture.

A. 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 component of the verification environment, ensuring the FIFO is thoroughly exercised by generating write and read transactions and verifying their handling.

- Environment**:** Encapsulates the setup of the agent, which includes the sequencer, monitor, driver, scoreboard, and coverage.
- Agent**:** Contains the sequencers, drivers, and monitors for both read and write operations. It coordinates their activities to generate stimuli and capture 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, facilitating the flow of transactions from the sequencer to the driver.
- Read **Driver:** Handles read transactions separately from write transactions by extending 'uvm_driver'. In the run phase, it 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, and the connect phase logs the connection status for debugging purposes.
- Write **Driver:** Handles write transactions separately from read transactions by extending 'uvm_driver'. In the run phase, it continuously fetches write transactions from the sequencer and drives them to the DUT, waiting for the positive edge of the write clock to update the data and write enable signals of the interface. The build and connect phases mirror those of the read driver.
- Read **Monitor:** Monitors read operations separately from write operations by extending 'uvm_monitor'. The run phase continuously creates and populates read transactions.transactions based on the DUTs read signals. It then writes these transactions to the respective analysis port to allow the scoreboard to 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 'uvm_sequencer'. It acts as an intermediary that sequences transactions and ensures they are correctly managed and forwarded to the driver. The build phase calls the base class's build phase for any necessary setup and logs it. The connect phase calls the base class's connect phase to establish connections between the ports and exports and logs it.

Read Sequence: Generates sequences of read transactions separately from write transactions by extending 'uvm_sequence'. It begins by raising an objection, then creates and randomizes read transactions with the operation set to READ. These transactions are started and completed, driving read enable signals to the DUT. The objection is dropped at the end of the sequence to signal completion.

Write Sequence: Generates sequences of write transactions separately from read transactions by extending 'uvm_sequence'. It begins by raising an objection, then creates and randomizes write transactions with the operation set to WRITE. These transactions are started and completed, driving write enable signals to the DUT. The objection is dropped at the end of the sequence to signal completion.

```
# UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
# ------------------------------------------------------------------------------
# Name                        Type                        Size  Value
# ------------------------------------------------------------------------------
# uvm_test_top                fifo_random_test            -     @471
#   env                       fifo_env                    -     @478
#     ra                      read_agent                  -     @493
#       rd                    read_driver                 -     @618
#         rsp_port            uvm_analysis_port           -     @633
#         seq_item_port       uvm_seq_item_pull_port      -     @625
#       rm                    read_monitor                -     @641
#         port_read           uvm_analysis_port           -     @649
#       rs                    read_sequencer              -     @509
#         rsp_export          uvm_analysis_export         -     @516
#         seq_item_export     uvm_seq_item_pull_imp       -     @610
#         arbitration_queue   array                       0     -
#         lock_queue          array                       0     -
#         num_last_reqs       integral                    32    'd1
#         num_last_rsps       integral                    32    'd1
#     scb                     fifo_scoreboard             -     @500
#       read_port             uvm_analysis_imp_port_b     -     @671
#       write_port            uvm_analysis_imp_port_a     -     @663
#     wa                      write_agent                 -     @486
#       wd                    write_driver                -     @789
#         rsp_port            uvm_analysis_port           -     @804
#         seq_item_port       uvm_seq_item_pull_port      -     @796
#       wm                    write_monitor               -     @812
#         port_write          uvm_analysis_port           -     @820
#       ws                    write_sequencer             -     @680
#         rsp_export          uvm_analysis_export         -     @687
#         seq_item_export     uvm_seq_item_pull_imp       -     @781
#         arbitration_queue   array                       0     -
#         lock_queue          array                       0     -
#         num_last_reqs       integral                    32    'd1
#         num_last_rsps       integral                    32    'd1
# ------------------------------------------------------------------------------
```

- **Sequencer:** Manages the flow of transactions from the sequence to the driver by extending 'uvm_sequencer'. It acts as an intermediary that sequences transactions and ensures they are correctly managed and forwarded to the driver. The build phase calls the base class's build phase for any necessary setup and logs it. The connect phase calls the base class's connect phase to establish connections between the ports and exports and logs it.
- **Read Sequence:** Generates sequences of read transactions separately from write transactions by extending 'uvm_sequence'. It begins by raising an objection, then creates and randomizes read transactions with the operation set to READ. These transactions are initiated and completed, driving read enable signals to the DUT. The objection is dropped at the end of the sequence to signal completion.
- **Write Sequence:** Generates sequences of write transactions separately from read transactions by extending 'uvm_sequence'. It begins by raising an objection, then creates and randomizes write transactions with the operation set to WRITE. These transactions are initiated and completed, driving write enable signals to the DUT. The objection is dropped at the end of the sequence to signal completion.
- **Transaction:** Extends 'uvm_sequence_item' and represents the basic unit of data transfer in the UVM environment. It encapsulates all necessary signals and data for a single transaction.
- **Coverage:** Collects and reports coverage data by extending 'uvm_subscriber'.
- **Scoreboard:** Compares the expected results with the actual results of all operations, verifying 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, ensuring consistent and organized interaction.

Hierarchy

At the top of the UVM testbench hierarchy is the 'uvm_test_top' class, which instantiates the verification environment. This environment acts as a container for the agent, scoreboard, and coverage components.

The FIFO agent generates stimuli for the DUT and collects responses, facilitating a controlled data flow through the sequencer. The scoreboard verifies data comparisons through analysis ports from the monitor, checking for data mismatches and correct operation. Finally, the coverage component collects coverage data, ensuring that relevant scenarios are tested and identifying any untested areas.

6. Test Scenarios and Coverage

6.1 Bug Injection

In our design, we deliberately introduced a bug to assess the effectiveness of our UVM testbench. This bug involved incorrectly specifying the size of the write pointer in the FIFO write pointer module, simulating a common scenario of accidental typos that can occur in any FIFO implementation.

In a properly functioning FIFO, the write pointer increments with each write operation until it reaches the maximum depth of the FIFO, where it wraps around to 0. This ensures that each new data entry is written to a new location in the buffer. However, in our bug injection scenario, the write pointer only has a width of 2 bits, causing it to prematurely wrap around and overwrite data.

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].