Asynchronous FIFO Design and Verification using UVM

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

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 several enhancements.

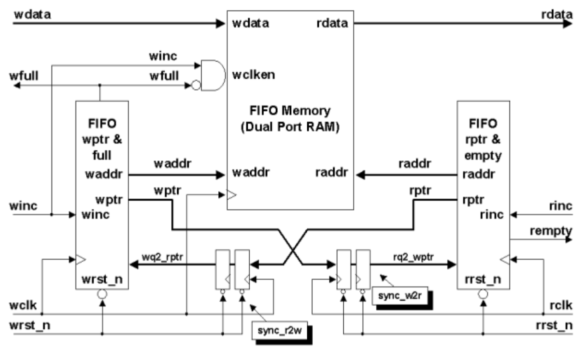


Figure 1: C. Cummings high-level asynchronous FIFO design

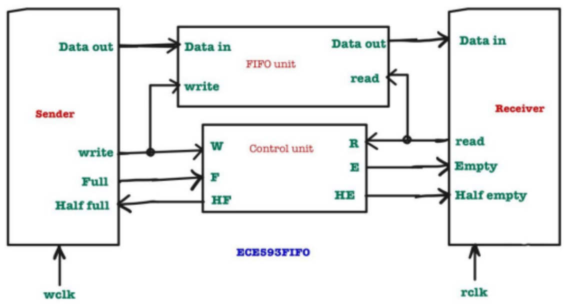


Figure 2: Our high-level asynchronous FIFO design

A. Memory Buffer

We utilize dual-port RAM 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 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.

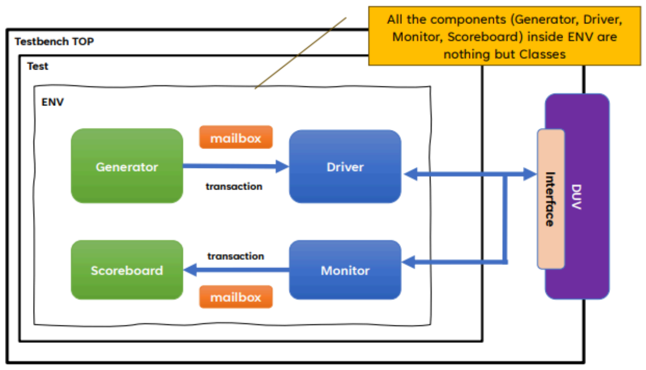


Figure 3: Class-based testbench architecture.

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 input scenarios 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, then sends the transactions to the monitor.

Monitor observes the DUT outputs and collects data for analysis. It captures read and write transactions and sends observed transactions to the scoreboard for comparison. This is also where read operations are tracked and handled correctly.

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.

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 with the help of the '*extends*' keyword; these include driver, generator, environment, monitor, scoreboard, test, and transaction.

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.

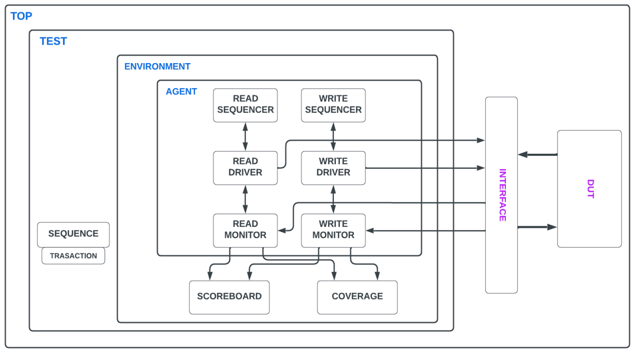


Figure 4: UVM testbench architecture.

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 operations. It coordinates their activities to generate stimulus and couture 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 phase 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 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 the '*uvm_sequencer*'. It acts as the intermediary that sequences transactions and ensures they are correctly managed and forwarded to the driver. The build phase calls the base classes build phase to perform any necessary setup and logs it. The connect phase calls the base classes 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 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.

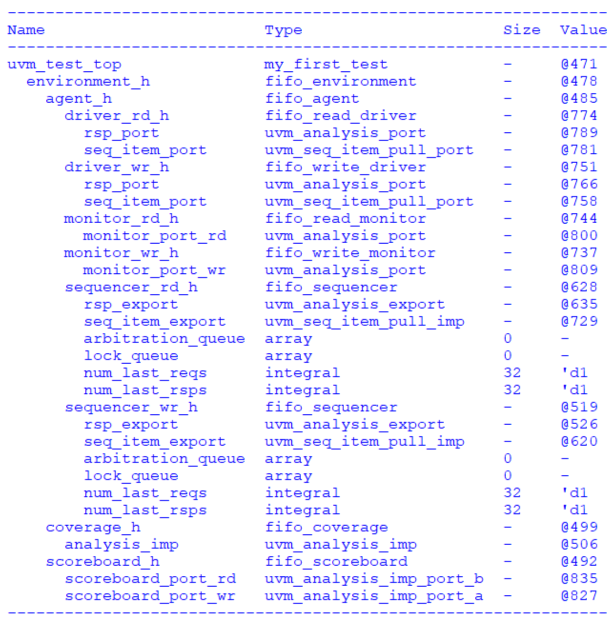
Coverage collects and reports coverage data by extending the '*uvm_subscriber*'.

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 '*uvm_test_top*' class; it instantiates the 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.

6. Test Scenarios and Coverage

6.1 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 2 bits and therefore wraps back around and write over data prematurely.

6.2 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, and specifically 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.

7. Overall Findings

7.1 Bug Injection Results

The first three 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 three 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.

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