UVM-Based Verification of FIFO

Introduction:

This report aims to provide an in-depth explanation of the Verilog code for a First-In-First-Out (FIFO) module and the techniques used to verify its functionality through the Universal Verification Methodology (UVM). FIFOs are crucial components in digital systems for managing data storage and retrieval in a systematic manner. The correct operation of a FIFO module is essential for the proper functioning of many digital systems.

Importance of Verification: The development of complex digital systems demands rigorous testing and verification processes to ensure the reliability and correctness of the design. UVM, a widely adopted methodology for verification, plays a pivotal role in achieving this goal by providing a standardized framework for verifying digital designs. In this report, we will explore the structure of the FIFO module and delve into the techniques and tools employed for its thorough verification.

The FIFO (First-In-First-Out) module is a crucial component in digital systems designed to manage data storage and retrieval in a specific manner. The primary function of the FIFO is to store a sequence of data elements and provide access to them in the order in which they were stored. This is achieved by employing two pointers, a write pointer (wr_ptr) for data insertion and a read pointer (rd_ptr) for data retrieval.

Module Declaration and Parameters

The module FIFO is declared with input and output ports.
The module includes three parameters: DEPTH, WIDTH, and POINTER_WIDTH, which are used to configure the FIFO's characteristics.

Input and Output Ports

The input and output ports for the FIFO module, includes clock (clk), reset (reset), control signals (put and get), data input (data_in), and status signals (empty_bar, full_bar, and data_out).

Registers and Wires

The module contains various registers (reg) and wires (wire) to store and control data flow. wr_ptr and rd_ptr are pointers for write and read operations, while empty_bar and full_bar represent whether the FIFO is empty or full. FIFO_3 is an array of registers to store data elements. Wires like wr_ptr_int, rd_ptr_int, full, empty, wr_rd, put_e, and get_e are used to perform operations on the data and status of the FIFO.

Data Flow

```
assign wr_rd = wr_ptr - rd_ptr;
assign full  = (wr_rd == 4'b1000);
assign empty = (wr_rd == 4'b0000);
assign put_e = (put && full == 1'b0);
assign get_e = (get && empty == 1'b0);
assign wr_ptr_int = wr_ptr[POINTER_WIDTH-1:0];
assign rd_ptr_int = rd_ptr[POINTER_WIDTH-1:0];
```

These assign statements are used to calculate various control signals such as wr_rd, full, empty, put_e, get_e, wr_ptr_int, and rd_ptr_int.
These signals help manage data flow and control the FIFO's operation.

Logic

```
always @(posedge clk)
begin
      if (reset)
      begin
      wr_ptr <= 3'b000;
      rd_ptr <= 3'b000;
      end
      else
      begin
      if (put_e)
      begin
            FIFO_3[wr_ptr_int] <= data_in;
            wr_ptr <= wr_ptr + 3'b001;
      end
      if (get_e)
      begin
            rd_ptr <= rd_ptr + 3'b001;
      end
      end
end
```

This block describes clocked logic that operates on the rising edge of the clock signal (clk). It manages the write (put) and read (get) operations on the FIFO. When reset is active, both pointers are reset to zero. Depending on the conditions, data can be written to and read from the FIFO.

Status Signals

```
always @(empty or full)
begin
      empty_bar <= ~empty;
      full_bar <= ~full;
end
```

The empty_bar and full_bar signals are updated based on the status of the FIFO, specifically whether it's empty or full.

UVM Flow Overview

The Universal Verification Methodology (UVM) is a standardized methodology for verifying digital designs. It provides a systematic and efficient approach to verify complex digital systems, ensuring their correctness and reliability. The UVM flow typically consists of the following key components:

Testbench Architecture:
UVM promotes the use of a structured testbench architecture. The testbench includes components such as **agents**, **sequences**, **drivers**, **monitors**, and **scoreboards**. This architecture facilitates modularity and reusability, making it easier to verify various design blocks and configurations.
Testbench Components:

Agents: Agents are responsible for interfacing with the design under test (DUT). They include the driver, responsible for driving input stimuli to the DUT, and the monitor, which captures the DUT's output and checks for errors.

Sequences: Sequences define the stimulus scenarios and sequences of transactions that need to be applied to the DUT. Sequences are organized into sequences items.
Drivers: Drivers execute sequences by driving the stimulus onto the DUT's inputs, simulating the behavior of the actual design.
Monitors: Monitors capture the DUT's output signals and perform checks to verify correctness.
Scoreboards: Scoreboards collect and compare expected results with actual results from the DUT to determine if the verification was successful.
UVM Test: A UVM test case is created by assembling sequences, specifying test scenarios, and configuring the test environment. The test case defines the stimulus and the expected responses, which are then executed against the DUT.
Phases: UVM introduces a predefined sequence of phases in the verification process, including build, connect, run, and report. These phases allow for a structured and synchronized execution of the testbench components.
Configuration and Reporting: UVM promotes the use of configuration objects to manage parameters and settings. Additionally, it provides built-in mechanisms for logging and reporting verification results, making it easy to identify issues and debug failures.
Coverage and Analysis: UVM allows for the collection and analysis of coverage data, helping to assess the comprehensiveness of verification tests and identify untested corner cases.
Reusability: UVM encourages the reuse of verification components across different projects, making it a powerful methodology for long-term productivity and maintainability.

Interface Definition

The dut_if interface defines the communication between the design under test (DUT) and its surrounding environment. It includes signals for clocking (clock), resetting (reset), control (put and get), status indicators (full_bar and empty_bar), and data input/output (data_in and data_out). This

interface enables the DUT to interact with testbench components and facilitates the verification process

Transaction Classes:

In the provided code, two transaction classes, **data_transaction** and **rst_transaction**, are defined. These classes play a crucial role in UVM-based verification, particularly in modeling the stimulus and response interactions between the testbench and the design under test (DUT).

data_transaction Class

Purpose: The data_transaction class represents a transaction that carries data-related information for the verification of the DUT. It includes the following fields:

data_in: A 16-bit data input field that simulates the data that can be written into the FIFO.

put: A control field that models the "put" operation, which instructs the DUT to write data into the FIFO.

get: A control field that models the "get" operation, which instructs the DUT to read data from the FIFO.

Inheritance: This class is derived from uvm_sequence_item, which is a base class in UVM used for modeling individual transactions.

Randomization: The rand keyword in front of the field declarations indicates that these fields can be randomized during testing, allowing for different data and control sequences to be applied to the DUT.

rst_transaction Class

Purpose: The **rst_transaction** class models a reset transaction. It is used to simulate the reset operation on the DUT. The class includes a single field:rst: A logic field that represents the reset signal.Inheritance: Similar to the data_transaction class, this class is derived from uvm_sequence_item.

Randomization: The **rst** field is not declared as rand because reset operations typically follow a specific sequence and are not randomized in the same way as data transactions.

Sequences

1. `FIFO_test_1_sequence` Class:

- Purpose: Alternates between putting data and getting data from the FIFO. Generates 40 transactions.
- Randomization: Allows for various test cases.

2. `FIFO_test_2_sequence` Class:
- Purpose: Focuses on writing data and performing simultaneous read and write operations. Generates 40 transactions.
- Randomization: Transactions are randomized.

3. `FIFO_rst_sequence` Class:
- Purpose: Handles reset operations. Creates reset and deassert reset transactions.

Sequencers

1. `rst_sequencer` Class:
- Purpose: Drives reset transactions into the test environment.
- Usage: Controls and manages reset operations during verification.

2. `data_sequencer` Class:
- Purpose: Drives data transactions into the test environment.
- Usage: Facilitates data and control signal delivery to the DUT.

Virtual Sequence

1. `top_vseq_base` Class:
- Purpose: Base class for virtual sequences with instances of `rst_sequencer` and `data_sequencer`.
- Usage: Common base for building virtual sequences that combine data and reset operations.

2. `vseq_rst_data` Class:
- Purpose: Combines data and reset sequences for comprehensive test scenarios.
- Usage: Orchestrates parallel execution of `FIFO_test_1_sequence` and `FIFO_rst_sequence` for FIFO module verification.

Driver

1. `data_driver` Class:
- Purpose: Drives data transactions to the DUT.

   - Tasks: `reset_check()` handles reset, and `send_data()` sends data to the DUT.
2. `rst_driver` Class:
   - Purpose: Drives reset transactions to the DUT.
   - Usage: Initiates and desserts reset.

Monitor

1. `my_monitor_1` Class:
   - Purpose: Monitors data transactions and forwards them to the scoreboard.
   - Checks: Monitors data transactions and looks for specific conditions to forward to the scoreboard.
2. `my_monitor_2` Class:
   - Purpose: Monitors data transactions and forwards them to the scoreboard.
   - Checks: Monitors data transactions and looks for specific conditions to forward to the scoreboard.

Agent

1. `my_agent_1` Class:
   - Purpose: Manages data-related verification components.
   - Components: Includes data driver, data sequencer, and monitor.
2. `my_agent_2` Class:
   - Purpose: Manages reset-related verification components.
   - Components: Includes reset driver and reset sequencer.
3. `my_agent_3` Class:
   - Purpose: Monitors data transactions and forwards them to the scoreboard.
   - Components: Includes monitor for data transactions.

Scoreboard Class

1. `my_scoreboard` Class:
   - Purpose: Compares expected and actual data transactions.
   - Comparison: Compares data transactions and reports success or failure.

Environment

1. `my_environment` Class:
   - Purpose: Orchestrates verification components.
   - Components: Includes agents, scoreboard, and analysis FIFOs.

These classes collectively form the testbench environment for verifying the FIFO module using UVM-based methodologies.

Test

top_test_base Class

- Purpose: A base class for test cases.
- Components: Contains the test environment (`env_h`) for connecting testbench components.
- Functions: Provides a common base for building test cases.

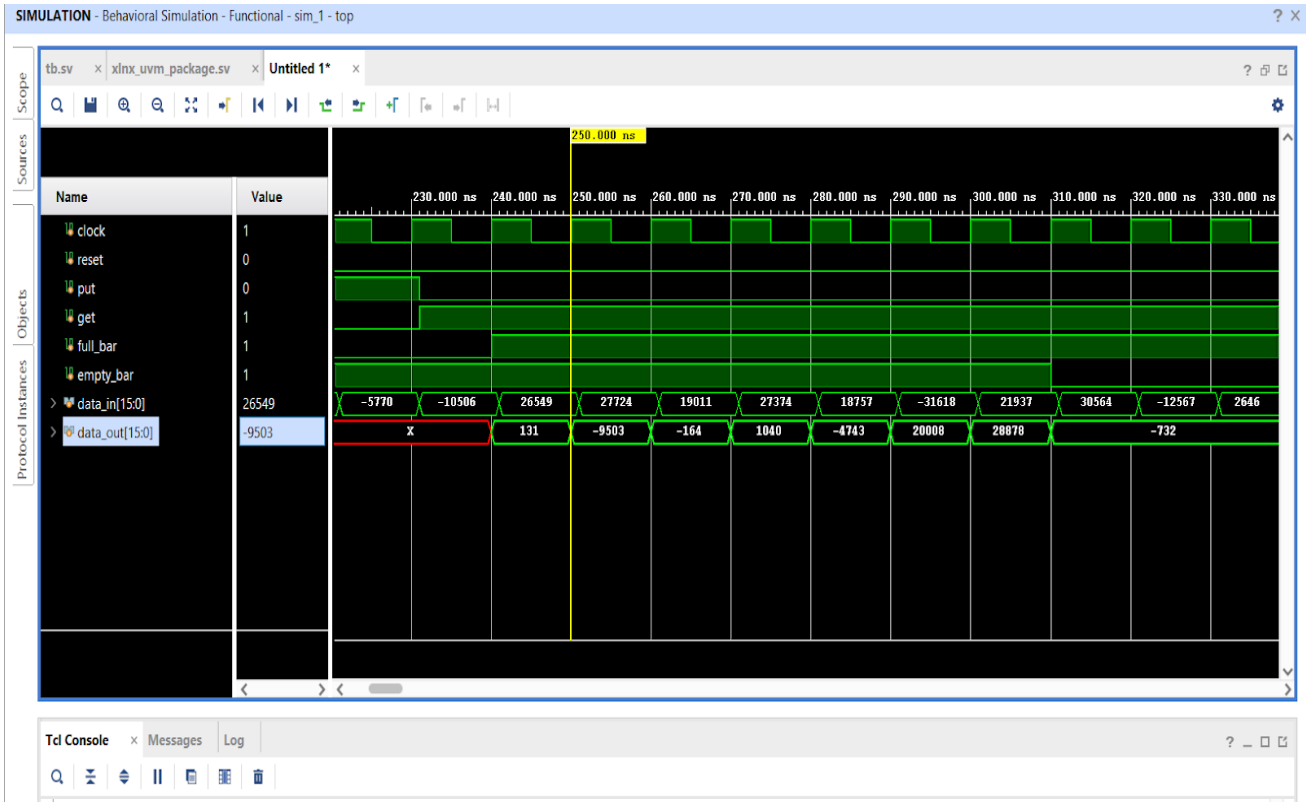
test_1 Class

- Purpose: A specific test case derived from `top_test_base`.
- Components: Includes a virtual sequence `vseq_h` of type `vseq_rst_data`.
- `build_phase`: Creates and initializes the virtual sequence.
- `run_phase`: Runs the test case and raises/drops objections during the test.

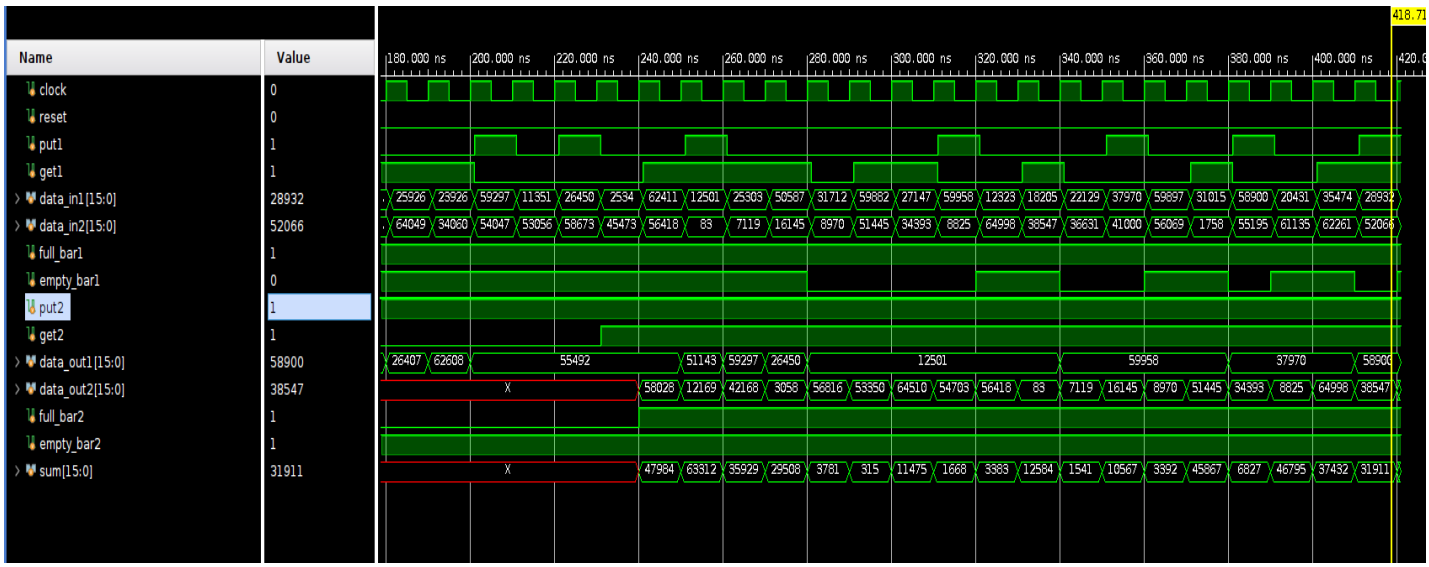
In the test case `test_1`, the virtual sequence `vseq_h` is created and initialized, and the test case is executed in the `run_phase`. It serves as a specific test scenario based on the `vseq_rst_data` sequence, allowing you to verify the functionality of the DUT under specific test conditions.

Simulation Results

Task A


SIMULATION - Behavioral Simulation - Functional - sim_1 - top
tb.sv
xlnx_uvm_package.sv
Untitled 1*
Scope
Sources
Objects
Protocol Instances
Name
Value
clock
reset
put
get
full_bar
empty_bar
data_in[15:0]
data_out[15:0]
250.000 ns
230.000 ns
240.000 ns
250.000 ns
260.000 ns
270.000 ns
280.000 ns
290.000 ns
300.000 ns
310.000 ns
320.000 ns
330.000 ns
-5770
-10506
26549
27724
19011
27374
18757
-31618
21937
30564
-12567
2646
X
131
-9503
-164
1040
-4743
20008
28078
-732
Tcl Console
Messages
Log

Task B


Name
Value
clock
reset
put1
get1
data_in1[15:0]
data_in2[15:0]
full_bar1
empty_bar1
put2
get2
data_out1[15:0]
data_out2[15:0]
full_bar2
empty_bar2
sum[15:0]
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
400.000 ns
418.71
25926 23926 59297 11351 26450 2534 62411 12501 25303 50587 31712 59862 27147 59958 12323 18205 22129 37970 59897 31015 58900 20431 35474 28932
64049 34060 54047 53056 58673 45473 56418 83 7119 16145 8970 51445 34393 8825 64998 38547 36631 41000 56069 1758 55195 61135 62261 52066
26407 62608 55492 51143 59297 26450 12501 59958 37970 58900
X
58028 12169 42168 3058 56816 53350 64510 54703 56418 83 7119 16145 8970 51445 34393 8825 64998 38547
X
47984 63312 35929 29508 3781 315 11475 1668 3383 12584 1541 10567 3392 45867 6827 46795 37432 31911