Capstone Project

UVM Course Capstone Project – Final Testbench Implementation

Objective

The capstone project will require students to design and implement a **complete UVM testbench** for a **simple AXI-like memory interface**. The project will **exercise all major UVM concepts**, including **UVM components, sequences, transaction randomization, scoreboarding, coverage, and debugging**.

Project Overview

Design Under Test (DUT)

- A simple **AXI-like memory module** with:
  - **Read & Write transactions**
  - **32-bit address and data width**
  - **FIFO for storing memory transactions**
  - **Basic handshaking mechanism (ready/valid)**

DUT Specification: AXI-Like Memory Module

The DUT for the UVM capstone project is a simple AXI-like memory module that supports read and write transactions. This module acts as a memory controller with a FIFO-based transaction queue and a simple ready/valid handshake mechanism.

Design Overview

- **Memory Interface**
  - **32-bit address bus**
  - **32-bit data bus**
  - **Read and Write support**
- **FIFO-Based Transaction Queue**
  - **Stores up to 16 outstanding transactions**
  - **FIFO depth: 16 entries**
- **AXI-Like Handshaking Mechanism**
  - **write_valid / write_ready**
  - **read_valid / read_ready**
- **Register Map**
  - **Configurable Memory Size Register**
  - **Status Register for FIFO Full/Empty Flags**

Interface Signals

Signal Name	Direction	Width	Description
clk	Input	1	System clock
rst_n	Input	1	Active-low reset
Write Interface			
write_addr	Input	32	Write transaction address
write_data	Input	32	Write data
write_valid	Input	1	Write request valid
write_ready	Output	1	Write request accepted
Read Interface			
read_addr	Input	32	Read transaction address
read_valid	Input	1	Read request valid
read_ready	Output	1	Read request accepted
read_data	Output	32	Read response data
Status Signals			
fifo_full	Output	1	FIFO full flag
fifo_empty	Output	1	FIFO empty flag

Register Map

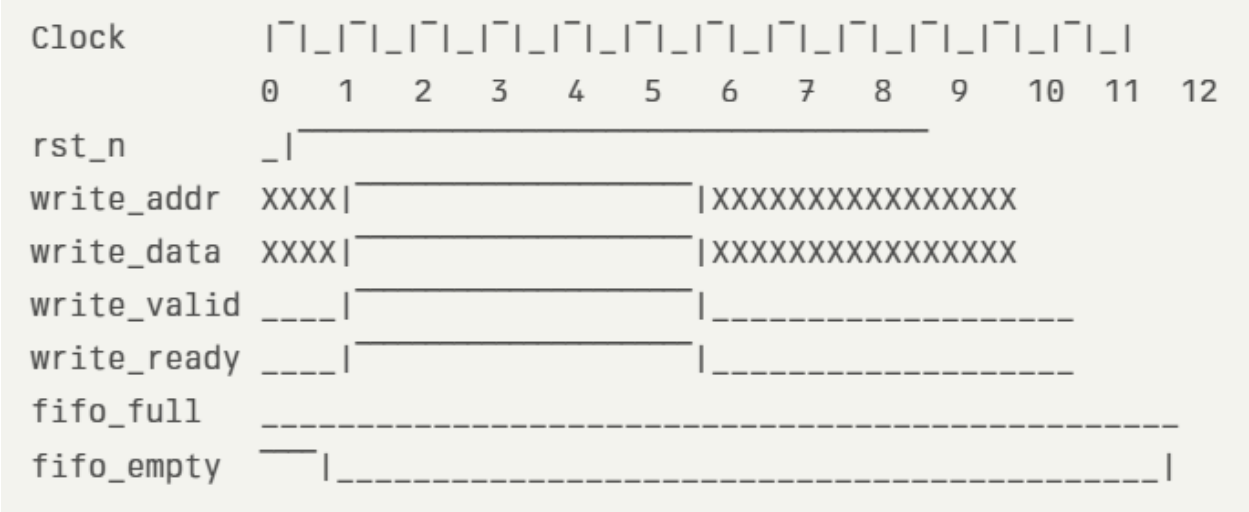
Register Name	Address	Width	Access	Description
MEM_SIZE_REG	0x0000	32	RW	Defines the total memory size
FIFO_STATUS_REG	0x0004	32	RO	Bit[0]: FIFO Full, Bit[1]: FIFO Empty
WRITE_COUNT_REG	0x0008	32	RO	Number of write transactions
READ_COUNT_REG	0x000C	32	RO	Number of read transactions

Functional Description

1. **Write Operation**
   - Testbench asserts `write_valid` with `write_addr` and `write_data`.
   - If FIFO is not full (`fifo_full = 0`), DUT asserts `write_ready` and stores the transaction.
   - Write transaction is committed to the internal memory after a **2-cycle delay**.
2. **Read Operation**
   - Testbench asserts read_valid with read_addr.
   - If FIFO is not empty (fifo_empty = 0), DUT asserts read_ready.
   - DUT outputs read_data after a 2-cycle delay.
3. **FIFO Management**
   - FIFO depth is 16 transactions.
   - FIFO is full when 16 transactions are queued.
   - FIFO is empty when all queued transactions are processed.

Timing Diagrams

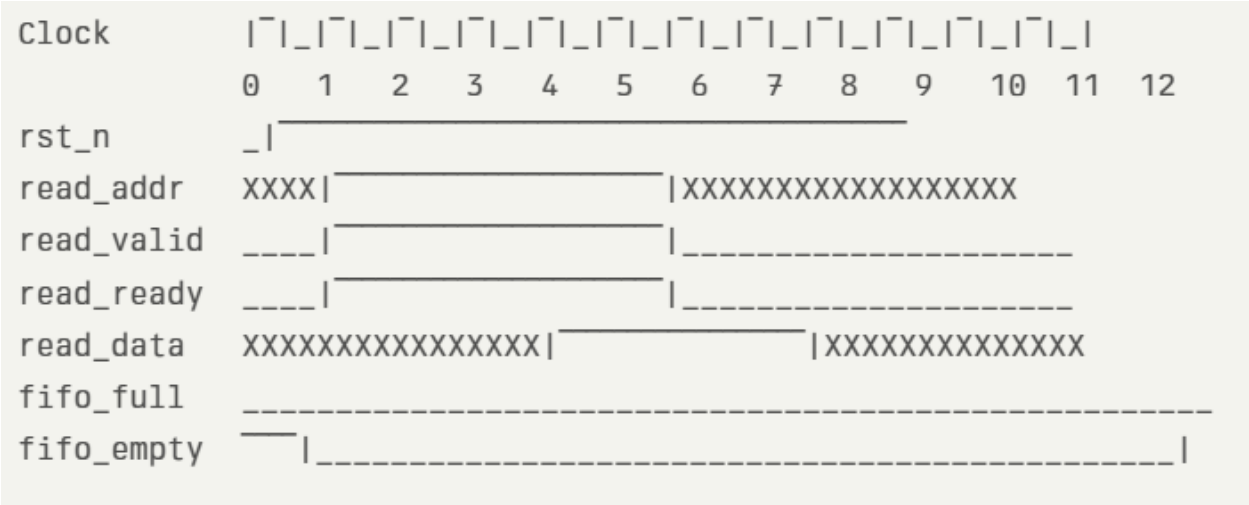
Write Operation Timing Diagram



Description:

1. At clock cycle 1, the testbench asserts `write_valid` and provides `write_addr` and `write_data`.
2. The DUT immediately asserts `write_ready` as the FIFO is not full.
3. The write transaction is held for 2 clock cycles (cycles 2 and 3).
4. At the end of cycle 3, the transaction is committed to internal memory.
5. At cycle 4, all signals return to their idle state.
6. The `fifo_empty` signal goes low as soon as the write is accepted and stays low until the FIFO becomes empty again.

Read Transaction Timing



Description:

1. At clock cycle 1, the testbench asserts `read_valid` and provides `read_addr`.
2. The DUT immediately asserts `read_ready` as the FIFO is not empty.
3. There's a 2-cycle delay (cycles 2 and 3) before the data is available.
4. At cycle 4, the DUT outputs the `read_data`.
5. The `read_data` remains valid for one clock cycle.
6. At cycle 5, all signals return to their idle state.
7. The `fifo_empty` signal goes low when the read is accepted and stays low until the FIFO becomes empty again.

Testbench Requirements

Students will build a **complete UVM environment**, including the following components:

UVM Transaction (`uvm_sequence_item`)

- Defines **read/write transactions**.
- Contains **address, data, and control signals**.
- Uses **constraints to randomize data patterns**.

UVM Sequences (`uvm_sequence`)

- Generates **stimulus for read and write transactions**.
- Includes a **directed sequence** for basic testing.
- Implements a **random sequence** with constraints.

UVM Driver (`uvm_driver`)

- Fetches transaction from the sequence.
- Converts it into **pin-level signal activity**.

UVM Monitor (`uvm_monitor`)

- Passively observes DUT transactions.
- Sends captured transactions to the scoreboard.

UVM Agent (`uvm_agent`)

- Encapsulates **Driver, Monitor, and Sequencer**.
- Configurable for **active/passive mode**.

UVM Scoreboard (`uvm_scoreboard`)

- Implements **predictive checking**.
- Compares **expected vs. actual results**.
- Uses **transaction-level checking**.

UVM Environment (`uvm_env`)

- Instantiates and connects **agents and scoreboard**.
- Uses **configuration database (uvm_config_db)**.

UVM Test (`uvm_test`)

- Runs different sequences and checks DUT behavior.
- Includes:
    - **Basic functional test** (directed sequence)
    - **Randomized stress test**
    - **Corner case tests**

Recommended Project Phases

This project can be implemented in **stages**, allowing you to build and verify each component before integrating the full testbench.

Phase 1: Setting Up the DUT & Basic Interface

- Students can use the **provided DUT**.
- Define **signal interface** using `virtual interface`.

Phase 2: Implementing UVM Components

- Students implement **transaction, driver, monitor, sequences**.
- Focus on **basic directed sequences**.

Phase 3: Integrating Agent & Environment

- Connect **driver, monitor, and sequencer** in an agent.
- Instantiate **agents & scoreboard** in the environment.

Phase 4: Functional Verification

- Implement **scoreboarding** with transaction checking.
- Add **coverage collection** (functional coverage).

Phase 5: Advanced Stimulus & Debugging

- Introduce **randomized sequences**.
- Add **debugging & UVM reporting best practices**.

Phase 6: Running Tests & Analyzing Results

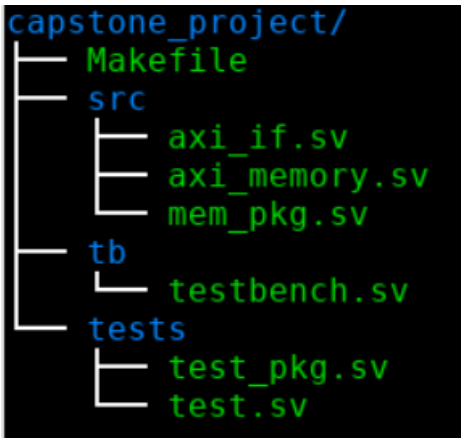
- Students execute tests with **log analysis**.
- Measure **coverage & debugging efficiency**.

Project Deliverables

Each student will submit:

1. **Complete UVM Testbench Source Code**
2. **Test Results & Functional Coverage Report**
3. **Debugging & Optimization Log** (e.g., performance/other bottlenecks found)
4. **Summary Report** (How they implemented the testbench & key challenges faced)

DUT Details (“src” directory)



axi_memory.sv // DUT Implementation

- **Key Features of This DUT**
  - **FIFO for handling outstanding transactions** (FIFO Depth = 16).
  - **Write transactions are stored & committed after a delay**.
  - **Read transactions fetch from memory after validation**.
  - **Uses an AXI-like ready/valid handshake mechanism**.
  - **FIFO management ensures valid reads & writes**.
  - **Students will verify this design using UVM testbench**.

axi_if.sv // Interface for connecting Testbench & DUT

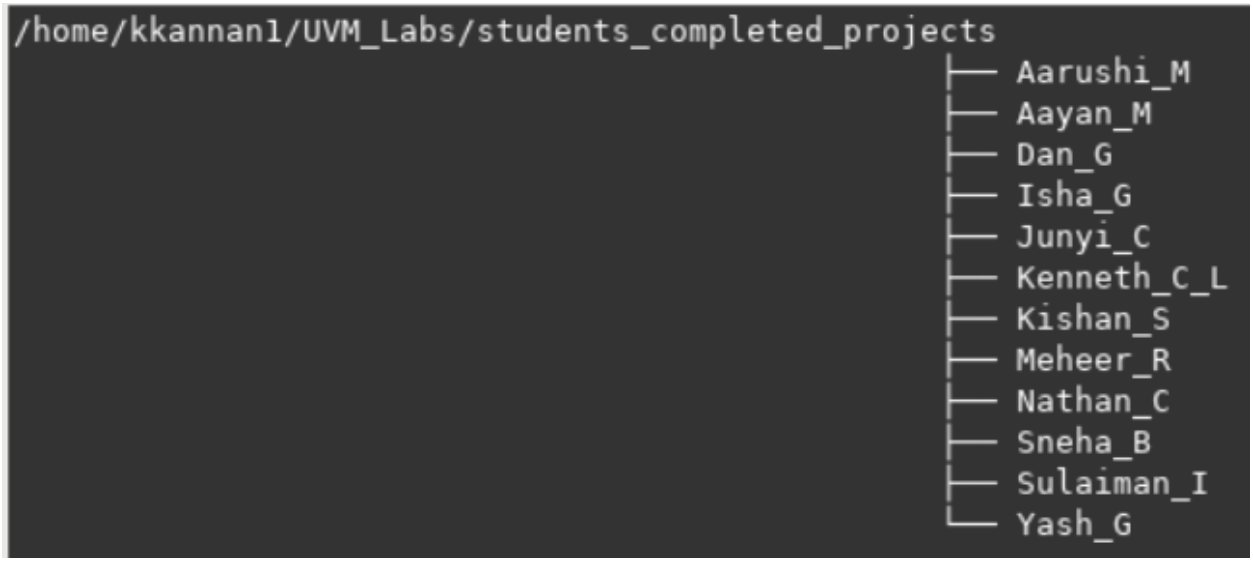
- Defines a structured interface for **AXI-like communication**.
- **Modports** cleanly separate DUT & Testbench connections.
- Provides a **single connection point** for UVM.

mem_pkg.sv // Package for common parameters

- Centralized memory & FIFO depth parameters for easy tuning.

Instructions:

- Please copy the entire capstone_project and build the UVM testbench around it.
  /home/kkannan1/UVM_Labs/capstone_project
- Once you are done, please copy your completed project back into this folder:



- Please do not hesitate to reach out to me in case of any questions or doubts; but please expect delays in responses.
- Feel free to collaborate amongst each other; and I encourage you to use discussions on canvas.