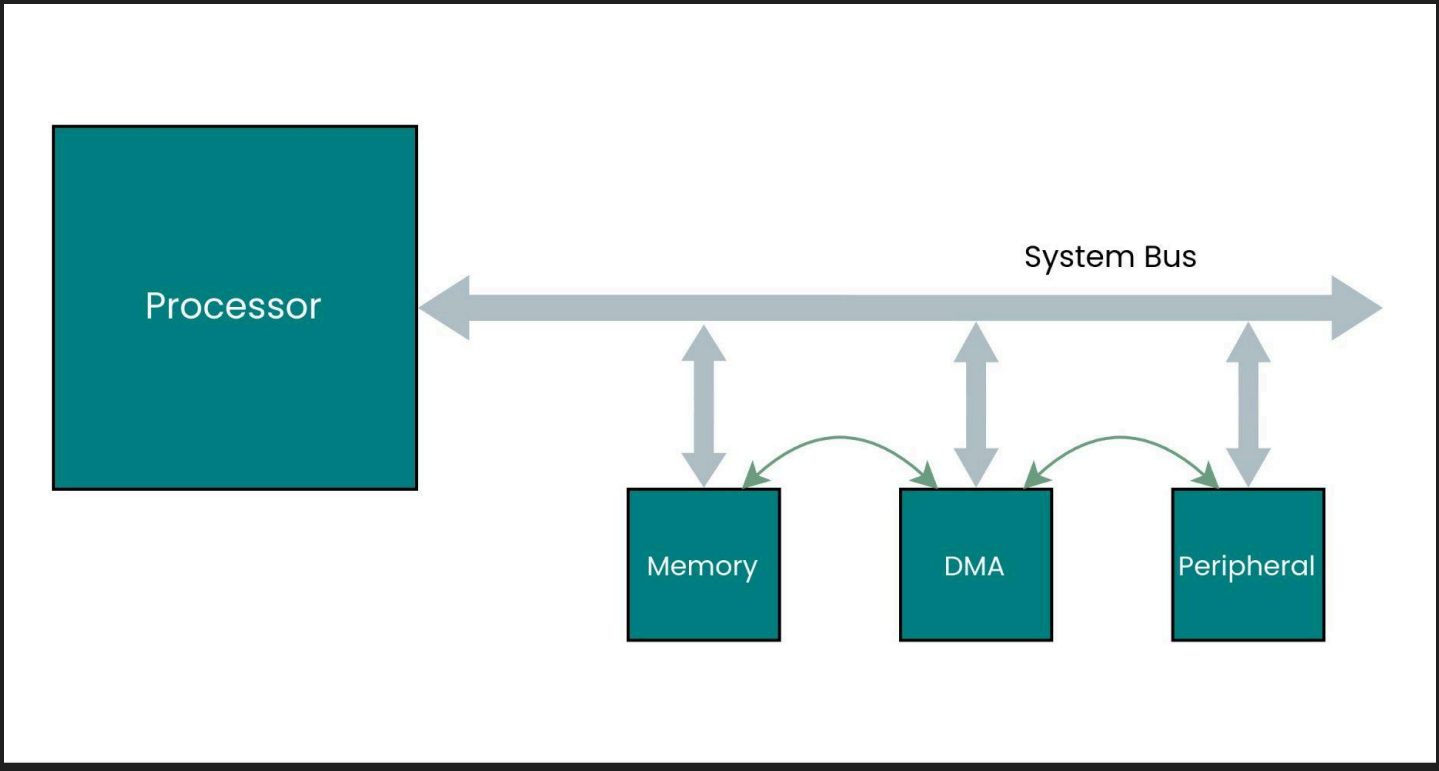
# AXI DMA Verification

**Status** Completed

**Timing** Dec 12, 2024 to Jan 24, 2025

**Owners** Noman Rafiq

**Tech Lead:** Hassan Ashraf



# Table of Contents

**AXI DMA Verification**
**Verification Architecture Document**

# UVM Verification Microarchitecture
# AXI DMA

---

## 1. Overview:

This project focuses on the verification of the AXI DMA IP designed in Vivado, with a specific emphasis on the **Direct Register Mode** functionality. The **Scatter/Gather Engine** is intentionally disabled to streamline the verification scope and center efforts on validating the core functionality of the DMA in this configuration.

The primary objective of the project is to ensure that the AXI DMA operates correctly and adheres to the defined specifications. This involves comprehensive testing to identify and resolve potential design issues, thereby achieving functional correctness, protocol compliance, and verification closure through coverage metrics.

**High-level goals include:**

1.1. **Functionality Verification**: Ensuring the DMA performs data movement between memory and AXI-Stream interfaces as expected.
1.2. **Protocol Compliance**: Validating adherence to the AMBA AXI and AXI-Stream specifications.

## 2. Features to Verify:

The key features to be verified include:

2.1. **Read/Write Operations for AXI-Lite Interface**
Verifying the AXI DMA's ability to correctly configure and respond to control and status register read/write operations using the AXI-Lite interface.
2.2. **Data Transfer Using AXI-Stream Interface**
Ensuring seamless data movement between AXI-Stream interfaces and system memory, including packetized and continuous streaming modes.

2.3. **Handling Packet Boundaries**
Validating proper recognition, processing, and termination of packets during data transfer, ensuring alignment with stream-based data protocols.

2.4. **Interrupt Generation and Handling**
Testing the DMA's ability to generate interrupts for events such as transfer completion, errors, and other status updates. Verifying the correct handling and acknowledgment of these interrupts in the design.

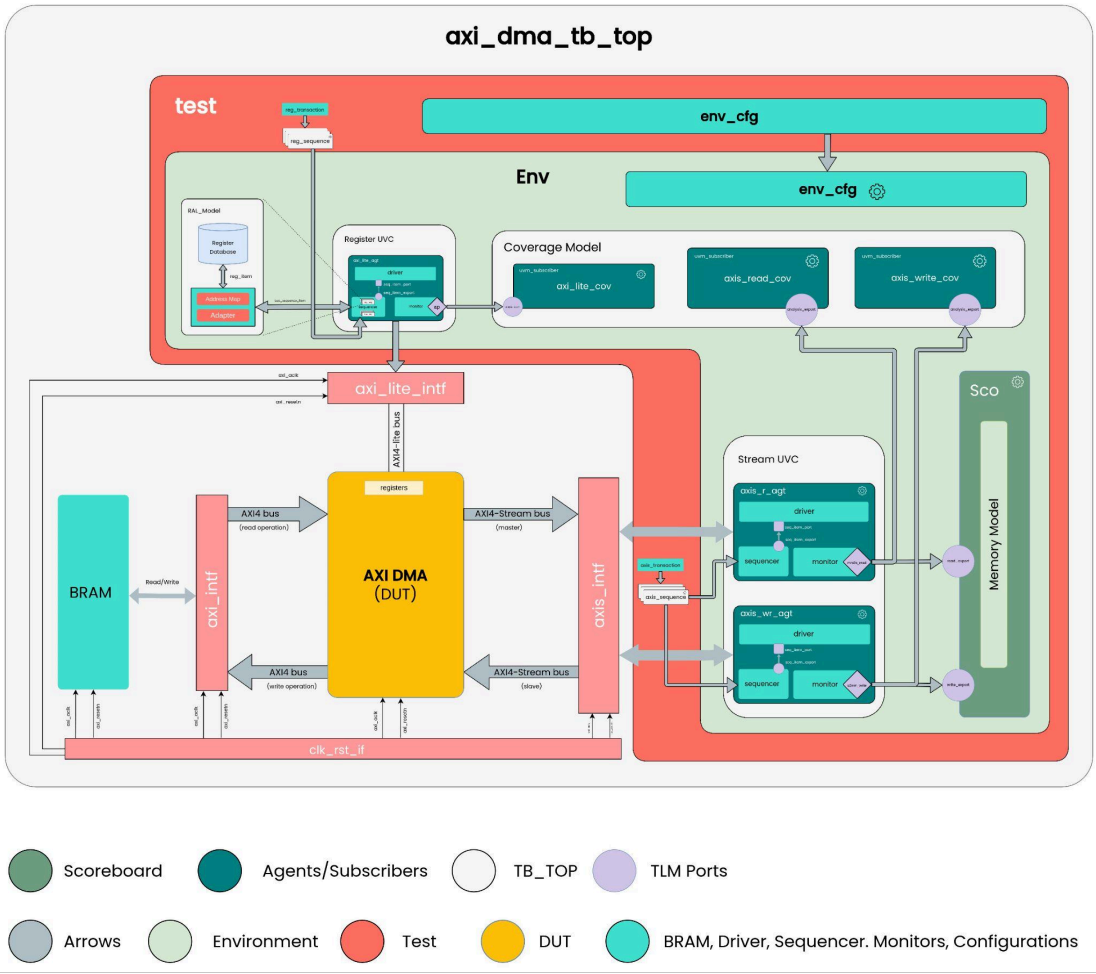
2.5. **Error Scenarios**
Evaluating the DMA's behavior under error conditions such as buffer overflows.

## 3. Testbench Architecture:

### 3.1 Overview

The testbench architecture is designed to validate the functionality and performance of the AXI DMA IP comprehensively. The top-level module, **`axi_dma_tb_top`**, orchestrates all testbench components and interfaces, ensuring a streamlined interaction between the DUT and verification components.

An image illustrating the architectural flow and component interaction is attached below:



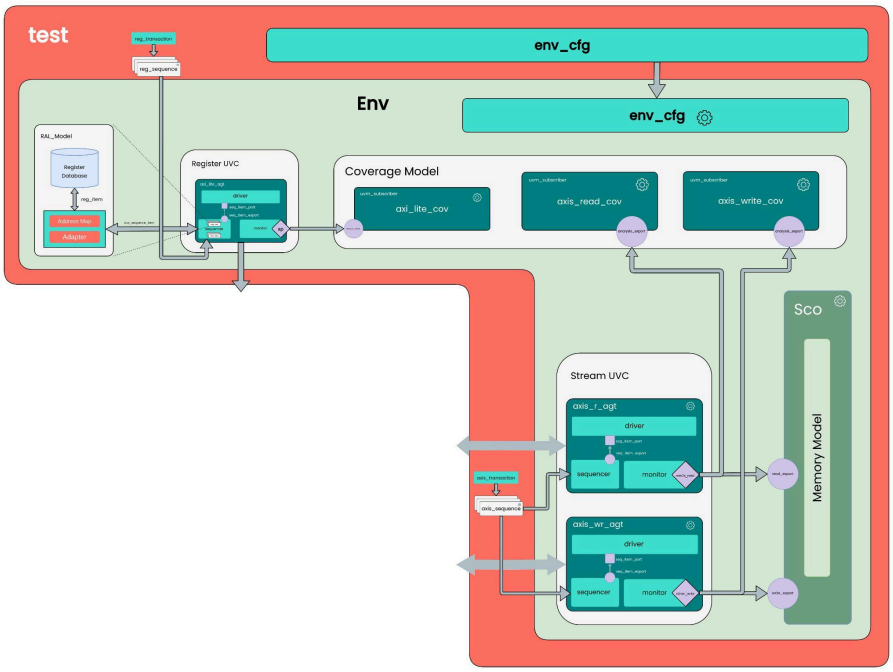

---

### 3.2 Components

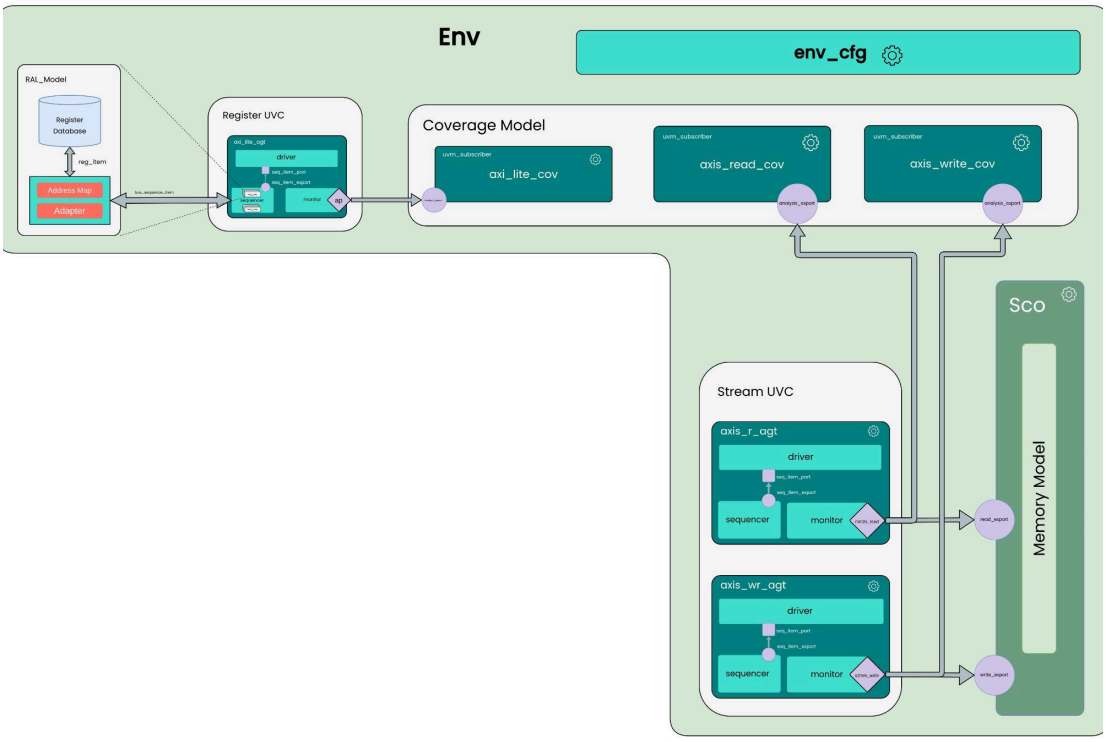
1. **Interfaces**
   - **`clk_rst_if`** (Clock and Reset Interface): Used for clock and reset generation.
   - **`axi_lite_intf`** (AXI4-Lite): Used for control and status register configuration.
   - **`axi_intf`** (AXI4): Handles memory-mapped data transactions for the DMA.
   - **`axis_intf`** (AXI-Stream): Facilitates data transfer between the DUT and external systems using AXI-Stream protocol.
2. **Block RAM (BRAM)**
   - Serves as system memory for memory-mapped read/write operations.

- Acts as the primary data repository for testing the DMA's memory interfaces.

3. **Test Class**
   - Encapsulates the environment class and supports multiple test scenarios.
   - Customized for specific test configurations based on verification objectives.




4. **Environment Class**
   - Houses the following sub-components:
     - Register_UVC
     - RAL_Model
     - Scoreboard
     - Stream_UVC
     - Coverage Model




---

### 3.3 Detailed Components

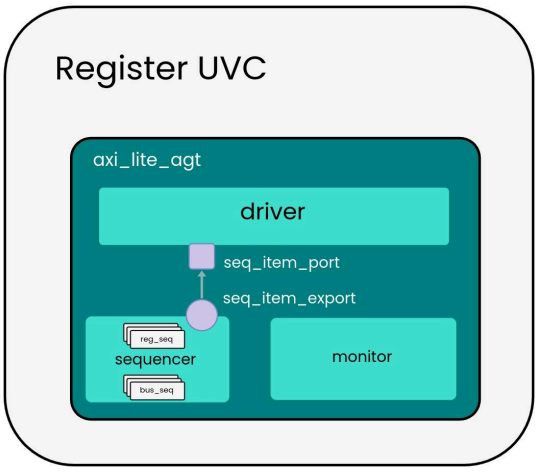
#### 3.1.1. Register_UVC

Encapsulates the **`axi_lite_agt`** responsible for configuring the DMA core. The agent consists of:

**Driver:** Drives transactions from the sequencer onto the AXI-Lite interface to read/write registers in the DUT.

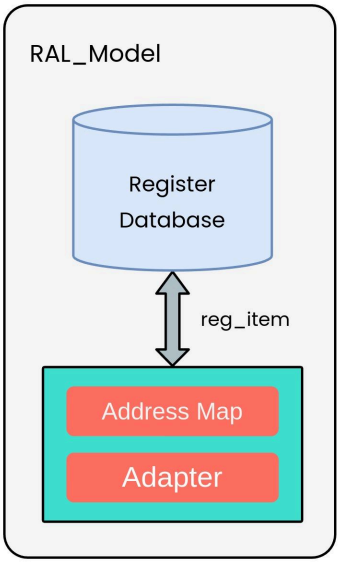
**Sequencer:** Fetches transactions from the sequence library or RAL_Model.

**Monitor:** Observes AXI-Lite transactions for validation and coverage collection.



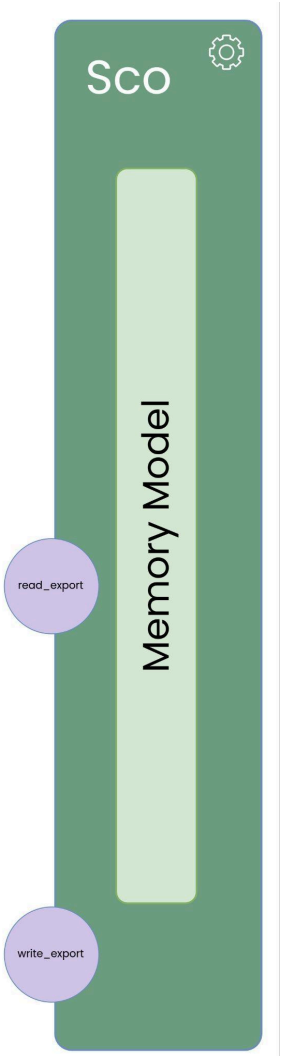

### 3.1.2. RAL Model

Mimics the memory-mapped registers of the DMA core, providing a high-level abstraction for register access.




### 3.1.3. Scoreboard

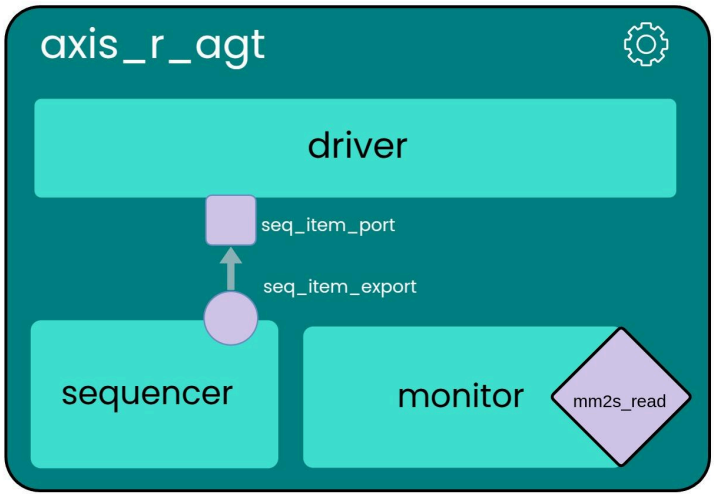
- Maintains a local memory model for read/write operations.
- Compares data read from local memory to actual data on the AXI-Stream read channel.
- Mirrors transactions written to the BRAM in its local memory for validation.
- Ensures data integrity by comparing expected and actual results, determining test pass/fail status.



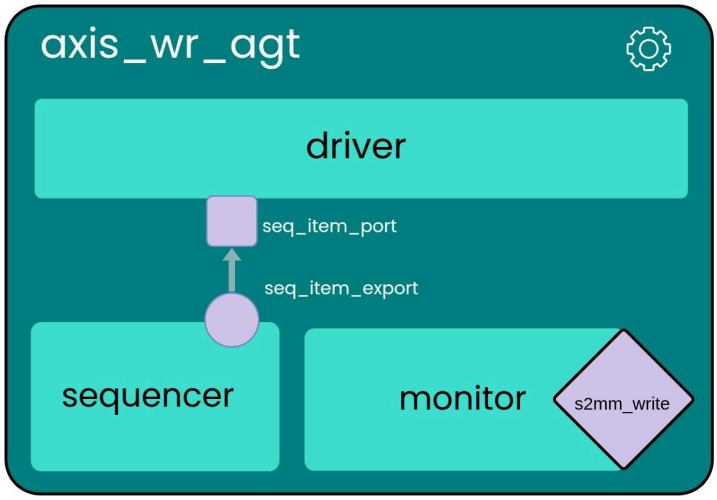

### 3.1.4. Stream_UVC

Contains two agents to verify AXI-Stream channels:

**`axis_read_agent`**: Handles transactions for the MM2S (Memory-Mapped to Stream) Read Channel.




**`axis_write_agent`**: Manages transactions for the S2MM (Stream to Memory-Mapped) Write Channel.




## 4. Folder Structure Diagram:

The verification environment has the following folder structure:

```
. Parent Directory
```

```
├── docs
│   ├── jpgs
│   └── pdfs
├── resources
│   └── misc
├── rtl
│   └── axi_dma_verification_10xe_tcp
└── verif
    ├── configurations
    ├── environment
    ├── includes
    ├── interfaces
    ├── ral_model
    ├── register_uvc
    ├── stream_uvc
    ├── tests
    └── top

18 directories
```

# 5. Test Workflow

## 5.1. High-Level Overview

The test workflow begins in the **tb_top** module by invoking the **run_test("test")** method. This initiates the execution of the **base_test** class, which orchestrates key setup and configuration tasks required for the simulation.

The workflow ensures the proper configuration of interfaces, sequence execution, and result validation. Interfaces such as AXI, AXI-Lite, and AXI-Stream are configured using the **uvm_config_db**, enabling communication between components like drivers, monitors, and the DUT.

---

## 5.2. Detailed Workflow

### 5.2.1. Test Initialization

- The test begins with the invocation of **run_test("base_test")** in the **tb_top** module. This triggers the execution of the **build_phase** across all hierarchical components.

- During the build_phase of the test, all components are created and connected through their connect_phase (bottom-up).
- A dynamic configuration object (**env_cfg**) is modified to test specific scenarios.

  For instance: To enable **MM2S_Read** operation, the **SRC_ADDR** and the number of **bytes** to read must be specified. The scoreboard's write operation is disabled to avoid entering an infinite loop.

  The number of transactions (**num_trans**) is calculated as **num_trans = env_cfg.calculate_txns()** within the build phase. This calculation ensures that the testbench operates without any infinite loops.

  The testbench configuration is derived from this env_cfg file, ensuring all components are tailored for the scenario being verified.

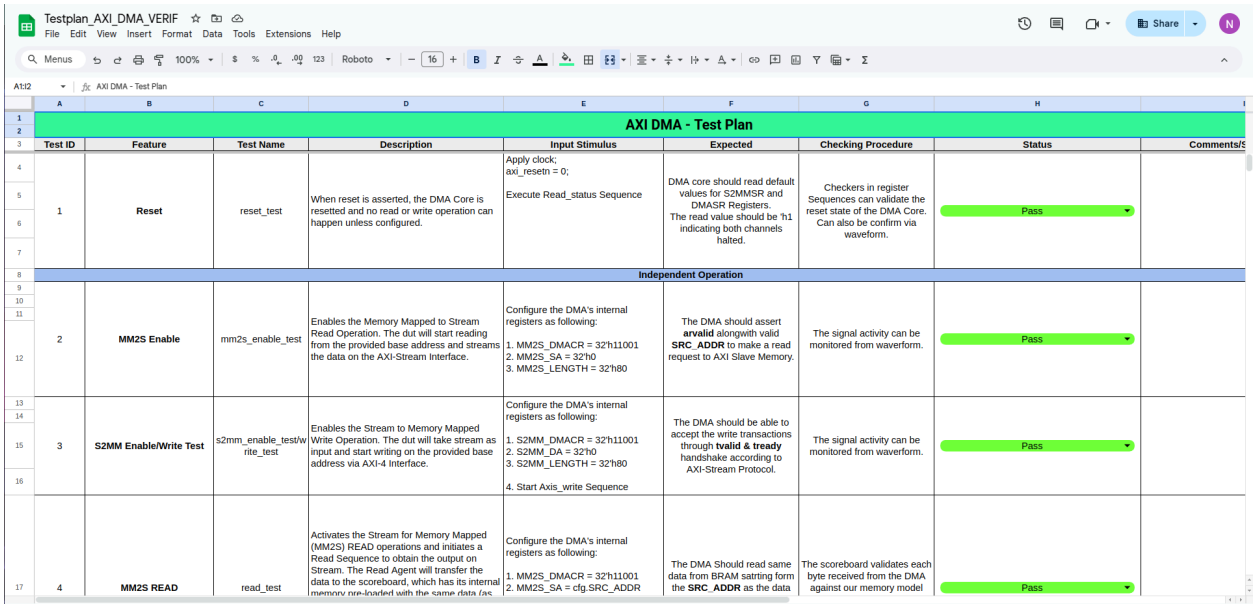
#### 5.2.2. Run Phase

- **DUT Configuration:**
  1. A simulation objection is raised at the start of the **run_phase**.
  2. DUT control and status registers are configured to enable desired operations such as **MM2S** (Memory-Mapped to Stream) Read or **S2MM** (Stream to Memory-Mapped) Write.
- **Sequence Execution:**
  1. Sequences such as **axis_read** and **axis_write** are started on their respective sequencers.
  2. These sequences drive transactions onto the AXI-Stream interfaces for data transfer operations.
- **Monitoring:**
  1. Monitors in each agent observe activity on their respective interfaces.
  2. Valid transactions are written to the analysis port by invoking the **write()** method, making them available for further analysis by other components like the scoreboard.
- **Scoreboarding:**
  1. The scoreboard captures transactions broadcasted via TLM implementation ports.

2. It performs a comparison between captured transactions and expected results, identifying any mismatches.
3. Errors, if any, are reported, and the test pass/fail status is determined during the **report_phase**.

- **Coverage Model:**
  1. The coverage model subscribes to transactions via TLM analysis ports by implementing the write() method.
  2. These transactions drive the sampling of functional coverage groups defined in the coverage model.
  3. Coverage groups capture key protocol behaviors and interface activities, including:
  4. AXI-Stream protocol events: Handshake patterns, burst lengths, and packet boundaries.
  5. Control signals: Activity of tvalid, tready, and interrupt signals (mm2s_introut, s2mm_introut).
  6. This systematic sampling ensures that all critical scenarios and edge cases are exercised during the verification process.
- **Termination:**
  1. Once all sequence items are driven and monitored on the interfaces, the test workflow transitions to termination.
  2. The simulation objection is dropped, signaling the end of the test.

## 6. Testplan



Testplan_AXI_DMA_VERIF

File Edit View Insert Format Data Tools Extensions Help

Share

AXI DMA - Test Plan

| Test ID | Feature | Test Name | Description | Input Stimulus | Expected | Checking Procedure | Status | Comments/S |
|---|---|---|---|---|---|---|---|---|
| 1 | **Reset** | reset_test | When reset is asserted, the DMA Core is resetted and no read or write operation can happen unless configured. | Apply clock;<br>axi_resetn = 0;<br><br>Execute Read_status Sequence | DMA core should read default values for S2MMSR and DMASR Registers.<br>The read value should be 'h1 indicating both channels halted. | Checkers in register Sequences can validate the reset state of the DMA Core. Can also be confirm via waveform. | Pass | |
| | **Independent Operation** | | | | | | | |
| 2 | **MM2S Enable** | mm2s_enable_test | Enables the Memory Mapped to Stream Read Operation. The dut will start reading from the provided base address and streams the data on the AXI-Stream Interface. | Configure the DMA's internal registers as following:<br>1. MM2S_DMACR = 32'h11001<br>2. MM2S_SA = 32'h0<br>3. MM2S_LENGTH = 32'h80 | The DMA should assert **arvalid** alongwith valid **SRC_ADDR** to make a read request to AXI Slave Memory. | The signal activity can be monitored from waverform. | Pass | |
| 3 | **S2MM Enable/Write Test** | s2mm_enable_test/write_test | Enables the Stream to Memory Mapped Write Operation. The dut will take stream as input and start writing on the provided base address via AXI-4 Interface. | Configure the DMA's internal registers as following:<br>1. S2MM_DMACR = 32'h11001<br>2. S2MM_DA = 32'h0<br>3. S2MM_LENGTH = 32'h80<br>4. Start Axis_write Sequence | The DMA should be able to accept the write transactions through **tvalid & tready** handshake according to AXI-Stream Protocol. | The signal activity can be monitored from waverform. | Pass | |
| 4 | **MM2S READ** | read_test | Activates the Stream for Memory Mapped (MM2S) READ operations and initiates a Read Sequence to obtain the output on Stream. The Read Agent will transfer the data to the scoreboard, which has its internal memory pre-loaded with the same data (as | Configure the DMA's internal registers as following:<br>1. MM2S_DMACR = 32'h11001<br>2. MM2S_SA = cfg.SRC_ADDR | The DMA Should read same data from BRAM sattring form the **SRC_ADDR** as the data | The scoreboard validates each byte received from the DMA against our memory model | Pass | |

## 7. Coverage Scores

XILINX

**Groups Coverage Summary**

| Score | Inst Score |
|---|---|
| 100 | 100 |

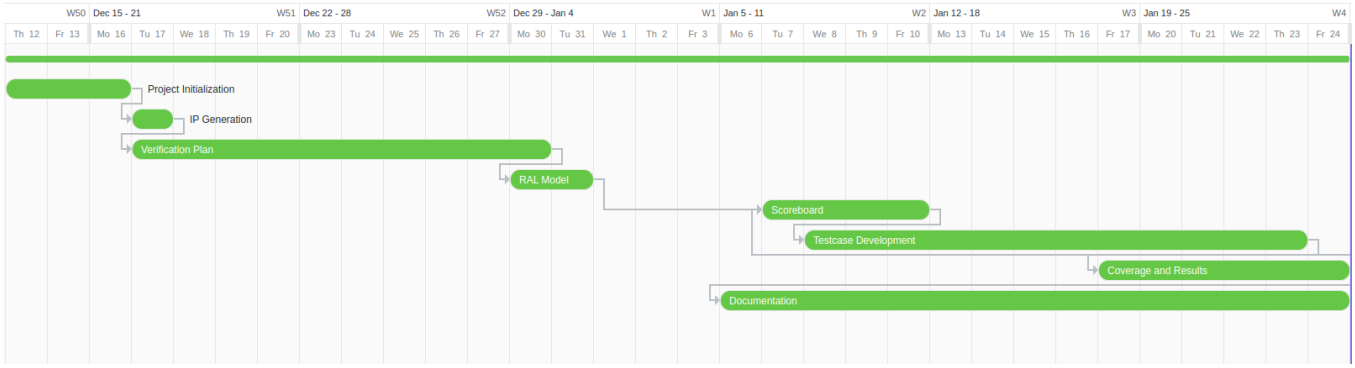
**Total groups in report: 4**

| Name | Score | Num Instances | Avg Instances Score | Weight | Goal | Merge Instances | Get Inst Coverage | Per Instance | Auto Bin Max | Comment |
|---|---|---|---|---|---|---|---|---|---|---|
| $unit_params_pkg_sv_2076944976::axi_lite_coverage::cg_read_channel | 100 | 1 | 100 | 1 | 100 | 0 | 0 | 0 | 64 | |
| $unit_params_pkg_sv_2076944976::axi_lite_coverage::cg_write_channel | 100 | 1 | 100 | 1 | 100 | 0 | 0 | 0 | 64 | |
| $unit_params_pkg_sv_2076944976::axis_read_coverage::cg_axis_read | 100 | 1 | 100 | 1 | 100 | 0 | 0 | 0 | 64 | |
| $unit_params_pkg_sv_2076944976::axis_write_coverage::cg_axis_write | 100 | 1 | 100 | 1 | 100 | 0 | 0 | 0 | 64 | |

## 8. Verification Challenges

- Synchronization between AXI and AXI-Stream interfaces.
- Read after Write Test (RAW).
- Scoreboard Checkers
- Writing Assertions
- Debugging random failures.

## 9. Timeline




## 10. References:

10.1. Git Repository
10.2. AXI Stream Specification
10.3. AXI4 and AXI-Lite Specification
10.4. AXI DMA Specification