AXI VIP

By

Shravya Samala

shravyasamala89@gmail.com

https://www.linkedin.com/in/samalashravya

https://github.com/ShravyaSamala/AXI_VIP

Introduction

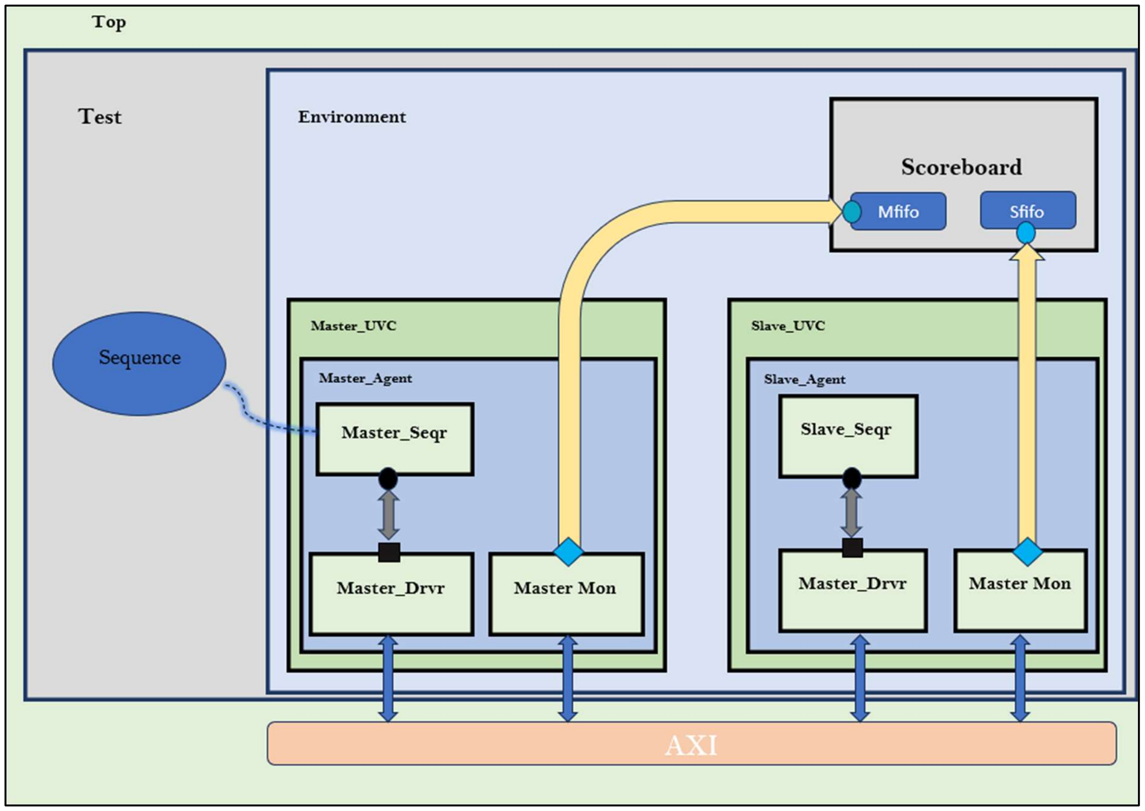
AXI, or Advanced eXtensible Interface, is a high-performance interface protocol developed by ARM for connecting components in System-on-Chip (SoC) designs. It is part of the AMBA (Advanced Microcontroller Bus Architecture) specification, which aims to standardize communication between various blocks in a chip.

AXI provides several key features that enhance performance and flexibility:

- **High Bandwidth**: AXI supports multiple outstanding transactions, allowing for high data throughput, crucial for memory-intensive applications.
- **Burst Transactions**: It enables burst transactions, where multiple data transfers occur in a single transaction, reducing overhead and improving efficiency.
- **Separate Read/Write Channels**: AXI utilizes distinct channels for read and write operations, enhancing parallelism and reducing latency.
- **Out-of-Order Transactions**: The protocol allows for out-of-order completion of transactions, further optimizing performance by allowing flexible data handling.
- **Master-Slave Architecture**: AXI employs a master-slave design, facilitating easy connection between different components, such as processors, memory, and peripherals.

These features make AXI suitable for a wide range of applications, from embedded systems to complex high-performance computing environments, ensuring scalability and adaptability in modern chip design.

Testbench Architecture



Verification Goals

The following verification goals have to met

a. **Protocol Compliance**: Ensure that the AXI interface adheres to the AXI protocol specifications by validating the correctness of signal transitions and timing relationships.
b. **Transaction Generation**: Create and verify various AXI transaction types (read, write, burst) to ensure that the interface can handle different scenarios.
c. **Signal Integrity**: Verify that all AXI signals (AW, AR, W, R, B) are driven correctly according to protocol specifications, including valid/ready handshakes.
d. **Response Handling**: Validate that the interface correctly handles responses, including read data and write response signals, ensuring they conform to expected behavior.
e. **Error Conditions**: Test the interface's ability to handle error conditions, such as misaligned addresses or unexpected response states, ensuring proper signaling and behavior.

Verification Strategies

To achieve the above verification goals we have to follow this verification strategies which are

a. **UVM Components**: Use UVM components like sequences, sequence items, and drivers to create robust transaction generation. You can define various sequences for read, write, and burst transactions.
b. **Scoreboarding**: Implement a scoreboard to track expected versus actual outputs, helping verify that the interface behaves correctly across different transactions.
c. **Randomization**: Utilize UVM's built-in randomization features to generate diverse transaction patterns and corner cases to stress-test the interface.
d. **Functional Coverage**: Implement functional coverage to track which aspects of the AXI protocol have been exercised, helping to identify untested scenarios.
e. **Assertions**: Use SystemVerilog assertions (SVAs) to enforce protocol rules and check for violations in real-time during simulation.

Testbench Setup

Setting up a testbench for an AXI interface using UVM (Universal Verification Methodology) involves several key components and steps. Here’s a structured approach to creating a robust testbench:

1. **Testbench Structure**
   - **Top-Level Module**: This module instantiates the AXI interface and connects the UVM environment.
   - **UVM Environment**: Create a UVM environment that includes the necessary components for stimulus generation, monitoring, and checking.

2. **UVM Components**

   a. **Agent**

   Agent is a configurable UVC(Universal Verification Component), It can be configured as Active Agent or Passive Agent. A Active Agent consists of Driver, Monitor and Sequencer which involves in driving and monitoring signals to/from DUT, where a Passive Agent consist only Monitor, this type agent involves only in Monitoring Signals from DUT.

- **Components:**
  - **Driver**: Drives the AXI signals based on sequences.
  - **Monitor**: Monitors the AXI signals to collect data and check protocol compliance.
  - **Sequencer**: Manages the sequences for generating AXI transactions.

b. **Sequence**

Purpose: Defines the types of transactions (e.g., read, write, burst).

Components:

- **Sequence Item**: Represents a single AXI transaction.
- **Sequences**: Create different sequences for varied transaction patterns.

c. **Scoreboard**

Purpose: Compares expected results with actual results from the AXI interface, verifying functionality. And collects coverage metrics to ensure thorough testing of the protocol

3. **Testbench Implementation Steps**

a. **Define the AXI Interface**: Create an AXI interface in SystemVerilog that includes all necessary signals (e.g., AW, AR, W, R, B).

b. **Create the UVM** :
- Implement the agent with its driver, monitor, and sequencer.
- Define sequences and sequence items to represent various AXI transactions.
- Develop the scoreboard to track and compare expected vs. actual results.

c. **Connect Components:**
- Connect the driver to the AXI interface signals.
- Connect the monitor to the AXI interface for observation.
- Ensure the sequencer can interact with the driver to send transactions.

d. **Test Sequence Execution:**

Create a top-level UVM test that instantiates the environment and runs the desired sequences.

e. **Run Simulations**: Execute simulations to test different scenarios and collect coverage.

f. **Analyze Results**: Use the scoreboard and coverage results to evaluate the correctness and completeness of the testing.

Coverage Model

Functional Coverage : Ensure that all type of burst transfers, burst lengths are covered.

The following coverpoints and crosses have to be included in coverage

- AWSIZE { bins{0,1,2}}
- AWBURST {bins{0,1,2}, illegal_bins {3}}
- AWADDR {bins addr {['h0:'hffffffff]}}
- AWLEN{bins len = {[0:15]}}
- Cross AWSIZE,AWBURST,AWLEN
- ARSIZE {bins{0,1,2}}
- ARBURST {bins{0,1,2}, illegal_bins {3}}
- ARADDR {bins addr {['h0:'hffffffff]}}
- ARLEN{bins len = {[0:15]}}
- Cross ARSIZE,ARBURST,ARLEN
- WDATA{bins wdata={['h0:'hffffffff]}}
- WSTRB{bins wstrb[]={4'b1111,4'b0001,4'b0010,4'b0100,
  4'b1000,4'b1100,4'b0110,4'b0011}}
- RDATA{bins wdata={['h0:'hffffffff]}}

Assertions

The following Assertions should be applied to find any black box bugs inside DUTs which are can't be monitored from testbench

- VALID should not go low once asserted until READY is high (in all channels)
- Every signal should be stable once VALID is asserted until READY is asserted (in all channels)
- If BURST mode is wrap then the AxLEN should be inside 1,3,7,15
- If AxSIZE is 1 and wrap BURST mode then address should be multiples of 2
- If AxSIZE is 2 and wrap BURST mode then address should be multiples of 4
- If VALID is high then SIZE should be inside 0,1,2 because here we are considering 32 bit data bus
- If VALID is high then BURST should not be equal to 3 because burst mode 3 is reserved

Objective

The objective of an AXI (Advanced eXtensible Interface) VIP (Verification IP) project is to verify and validate the functionality and compliance of the AXI protocol used in System-on-Chip (SoC) designs. AXI is a part of the AMBA (Advanced Microcontroller Bus Architecture) standard developed by ARM, commonly used for high-performance and low-latency communication between components in a system. The project typically involves the following objectives:

1. **Protocol Compliance:** Ensure that the AXI protocol implementation adheres strictly to the AXI specifications, including various AXI versions like AXI3, AXI4, and AXI4-Lite.
2. **Functional Verification:** Validate the functional correctness of the AXI interfaces and transactions, such as read and write operations, burst types, and data transfers, under various conditions.
3. **Stress Testing and Edge Cases:** Test the AXI interface behavior under different scenarios, including edge cases, corner cases, and stress conditions like simultaneous read/write bursts, burst lengths, and varying data sizes.
4. **Performance Metrics:** Measure and verify the performance aspects like latency, throughput, and bandwidth to ensure the AXI interface meets the required specifications.
5. **Reusability:** Develop a reusable VIP that can be integrated into different verification environments and adapted to multiple AXI-based designs with minimal modification.
6. **Integration Testing**: Verify the integration of AXI-based components within a larger SoC environment to ensure interoperability and proper communication between different IP blocks.

Advanced eXtensible Interface

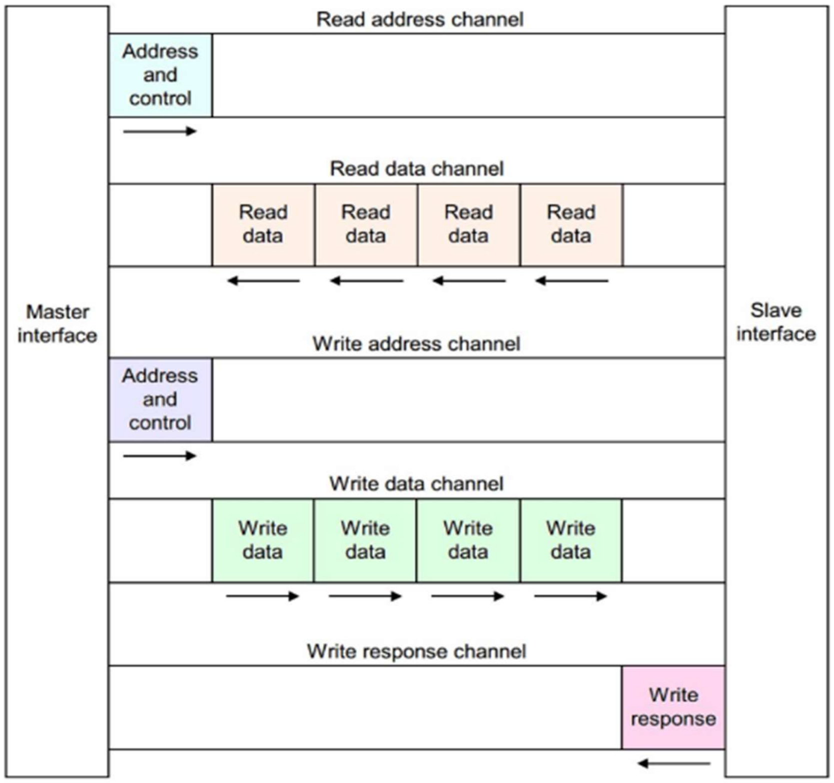
- The Advanced eXtensible Interface (AXI) protocol, part of the AMBA (Advanced Microcontroller Bus Architecture) specification from ARM, plays a critical role in high-performance System-on-Chip (SoC) design.
- AXI is renowned for its flexibility, efficiency, and ability to support high-speed and high-frequency data transactions, making it a fundamental component in modern SoC architectures.
- It builds on the earlier AXI3 protocol by enhancing performance, flexibility, scalability, and capability, particularly suited for high-bandwidth applications.

- AXI also includes a number of new features including out-of-order transactions, unaligned data transfers, cache support signals, and a low-power interface.

AXI Channels

AXI has 5 channel architecture, which are

- Write Address Channel
- Write Data Channel
- Write Response Channel
- Read Address Channel
- Read Data/Response Channel



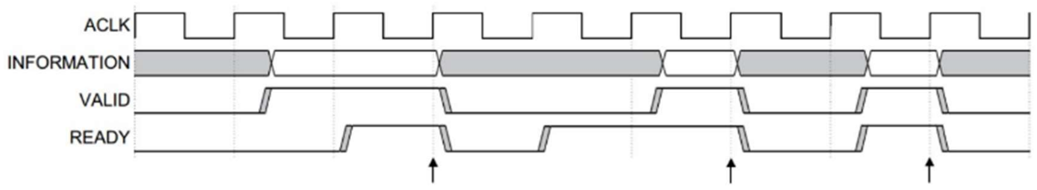
- The address channels are used to send address and control information while performing a basic handshake between master and slave. The data channels are where the information to be exchanged is placed.
- A master reads data from and writes data to a slave. Read response information is placed on the read data channel, while write response information has a dedicated channel. This way the master can verify a write transaction has been completed. Figure shows an AXI master and slave connected via the five AXI channels.
- Every exchange of data is called a transaction. A transaction includes the address and control information, the data sent, as well as any response information. The actual data is sent in bursts which contain multiple transfers.

AXI Transactions :

An AXI data transfer is called a transaction. Transactions can take the form of reads or writes and include address/control information, data, and a response. The data is sent in the form of bursts, which include multiple data items called beats. To synchronise the sending and receiving of data, an AXI master and slave perform a handshake at the beginning of a transaction using the READY and VALID signals.

Handshaking Mechanism :

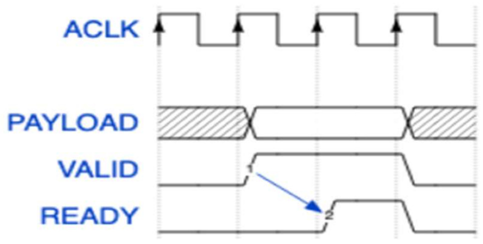
In AXI4, the handshaking mechanism is crucial for ensuring that data is transferred accurately between the master and slave devices. The protocol uses a valid/ready handshake on all its channels (read/write address, data, and response).



Valid/Ready Handshake in AXI :

The handshaking mechanism revolves around two signals on each channel:

- **VALID**: Indicates that the sender (master or slave) has valid data or an address to send.
- **READY**: Indicates that the receiver (slave or master) is ready to accept data or the address.
- The data transfer happens only when both the VALID and READY signals are asserted (set to high).



Key Points:

- The master asserts the VALID signal when it has data or an address ready to send.
- The slave asserts the READY signal when it is ready to receive that data or address.
- A transaction is completed when both VALID and READY are asserted at the same time.

Handshake Process Steps :

The handshake process follows a straightforward sequence:

1. The master asserts the VALID signal to indicate the availability of valid data.
2. The slave monitors the VALID signal and, if it is asserted and it is ready to receive data, asserts its READY signal.
3. When both the VALID and READY signals are high during the same clock cycle, data transfer occurs.
4. After data transfer, the master can de-assert the VALID signal, indicating that the data is sent, and the slave can also de-assert the READY signal if it no longer wishes to accept data.

Handshaking On Different Channels :

AXI4 has five independent channels, and each channel uses the VALID/READY

handshake for communication.

1. **Write Address Channel**
   - Master sends the write address on the address bus when AWVALID is high.
   - Slave acknowledges the address when AWREADY is high, completing the address handshake.
2. **Write Data Channel**
   - Master sends data on the data bus when WVALID is high.
   - Slave asserts WREADY when it's ready to accept the data, completing the data transfer handshake.
3. **Write Response Channel**
   - Slave sends a response after completing the write transaction using the BVALID signal.
   - Master acknowledges the response with BREADY, completing the handshake for write response.
4. **Read Address Channel**
   - Master sends the read address using ARVALID.
   - Slave acknowledges the read address by asserting ARREADY, completing the address transfer handshake.
5. **Read Data Channel**
   - Slave sends the requested data using RVALID when data is ready.
   - Master asserts RREADY when it is ready to accept the data, completing the handshake for the read data.

Scoreboard

A scoreboard in a verification environment is a component used to validate the correctness of the design under test (DUT) by comparing expected results with actual results. It plays a critical role in functional verification, ensuring that the DUT's output matches the expected behavior according to the design specifications. The scoreboard is typically part of a UVM (Universal Verification Methodology) environment and helps identify functional mismatches, thereby improving verification efficiency and accuracy.

Key Features of a Scoreboard:

1. **Data Monitoring**:
   - It captures data or transactions from various points of the verification environment, such as interface monitors, to keep track of what is being sent to and received from the DUT.
2. **Reference Model**:
   - The scoreboard may use a reference model or golden model, which predicts the expected output based on the input transactions. This model serves as a standard for comparison with the DUT's output.
3. **Comparison and Checking**:
   - The scoreboard compares the actual outputs of the DUT with the expected values generated by the reference model. If there is any discrepancy or mismatch, it logs an error or assertion failure, indicating a bug in the DUT.
4. **Transaction Tracking**:
   - It keeps a record of transactions and their status to ensure that they are completed as expected. This can include tracking responses for read/write operations, ensuring correct sequence, and verifying timing relationships.
5. **Coverage Collection**:
   - The scoreboard may also collect coverage metrics to provide information on the scenarios and corner cases that have been tested, ensuring the completeness of the verification.

Example Use Case:

In an AXI protocol verification environment, the scoreboard would track read and write transactions initiated by the master and ensure that the responses from the slave match the expected values. It checks aspects such as data integrity, burst length, address alignment, and protocol compliance.

Benefits:

- **Automation**: Automates the checking process, making it efficient to identify errors and reducing manual debugging efforts.
- **Reusability**: In a UVM environment, the scoreboard is designed to be reusable across different projects, making it a valuable component for consistent verification practices.

- **Comprehensive Validation**: Ensures that the DUT not only functions correctly but also meets performance and compliance criteria, enhancing the overall quality of the design.

A well-designed scoreboard is essential for effective verification, ensuring that any deviation from the expected behavior is quickly and accurately detected.

Take Away from The Project

The takeaway from a project, particularly in a verification environment like an AXI VIP project, involves summarizing the key learnings, achievements, and skills gained.

1. In-depth Understanding of the AXI Protocol:

- Gained a deep knowledge of the AXI protocol, including different transaction types, burst modes, and other aspects like handshaking, data transfer, and timing requirements.
- Understood the nuances of AXI variants like AXI3, AXI4, and AXI4-Lite and their applications in SoC designs.

2. Hands-on Experience with UVM (Universal Verification Methodology):

- Developed proficiency in building verification environments using UVM, including the creation of components such as agents, monitors, drivers, scoreboards, and sequence items.
- Learned how to structure and manage a scalable, reusable, and modular testbench using UVM best practices.

3. Building a Reusable and Scalable VIP:

- Gained experience in developing a Verification IP (VIP) that is both reusable and scalable, capable of being adapted to different verification environments with minimal modification.
- Focused on coding efficiency and modularity to ensure that the VIP can be integrated easily into larger verification environments.

4. Debugging and Problem-Solving Skills:

- Enhanced debugging skills by identifying and resolving issues related to protocol compliance, functional mismatches, and timing violations.
- Gained experience using debugging tools like Synopsys VCS or Siemens QuestaSim, which are crucial for identifying and fixing design and testbench issues.

5. Working with Scoreboards and Reference Models:

- Built and utilized scoreboards to automate the comparison of expected and actual outputs, ensuring comprehensive and accurate validation of the DUT.
- Implemented reference models to predict expected behavior, helping to enhance the verification process's accuracy and efficiency.

6. Coverage-Driven Verification:

- Learned the importance of functional and code coverage metrics in ensuring that all aspects of the design were tested, including edge cases and corner scenarios.

- Implemented coverage models and analyzed reports to identify coverage gaps, leading to the development of additional tests to close those gaps.

7. Performance Validation and Optimization:

- Gained experience in measuring and validating performance metrics such as latency, throughput, and bandwidth, ensuring that the design met specifications.
- Developed skills in optimizing the verification environment to reduce simulation time and enhance efficiency.

8. Exposure to EDA Tools:

- Acquired hands-on experience with industry-standard EDA tools like Synopsys VCS and Siemens QuestaSim for simulation and debugging, further enhancing practical skills**.**

Difficulties faced during Project Time

1. **Understanding the AXI Protocol Details:**

- **Difficulty:** Initially, grasping the complexities of the AXI protocol (including burst types, handshaking mechanisms, and timing requirements) can be challenging.
- **Solution:** Overcoming this required intensive study of the AXI specification, going through protocol training sessions, and analyzing reference designs to understand the protocol behavior practically.

2. **Debugging and Resolving Protocol Violations:**

- **Difficulty:** Debugging issues such as protocol violations or mismatches between the DUT and the expected behavior was challenging, particularly in complex scenarios like burst transfers or simultaneous read and write operations.
- **Solution:** Developing strong debugging skills by learning how to use waveform viewers, log files, and EDA tool features (e.g., assertions in Synopsys VCS or Siemens QuestaSim) proved essential. It also helped to work closely with teammates to brainstorm and resolve issues.

3. **Ensuring Full Coverage:**

- **Difficulty:** Achieving comprehensive coverage (functional and code) required identifying and testing numerous scenarios, including edge and corner cases. This was particularly difficult when dealing with different burst types, transaction sizes, and sequence variations.
- **Solution:** Building a detailed test plan and using coverage-driven verification techniques helped identify gaps in the test scenarios. Analyzing coverage reports and iteratively adding new test cases ensured full protocol coverage.

4. **Performance Optimization of the Testbench:**

- **Difficulty:** As the testbench became more complex, simulation time increased significantly, making it challenging to run multiple tests efficiently.
- **Solution:** Optimization techniques like reusing components, running tests in parallel where possible, and minimizing unnecessary log generation were implemented to reduce simulation time and improve efficiency.