Design and Verification of AXI Memory Interface

by

Hari Preetham Reddy Abbasani

A creative component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Electrical Engineering

Program of Study Committee:
Dr. Santosh Pandey, Major Professor

Iowa State University
Ames, Iowa
2025

DEDICATION

I dedicate this to my parents, my brother, whose endless love, sacrifices, and support have been the reason behind all my achievements. They showed me how important education and hard work are and always believed in me—even when I did not believe in myself. I also thank my friends, who stayed with me during tough times and shared my happiness in every success. Their support and friendship made this journey special and unforgettable.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to extend my gratitude to my major professor, Dr. Santosh Pandey, for his constant guidance, considerate mentoring, and ever being supportive in the whole process of the research. His profound knowledge, forbearing patience, and encouragement were the foremost elements in developing the project. His innovative ideas and constructive criticism helped my work as good as it has turned out to be. My greatest indebtedness is to the time and effort he devoted in assisting me in achieving the target.

I would also like to express my deep gratitude to Mr.Kumar Khandagle, a tutor on the Udemy website, whose verification as well as designing courses enabled me to acquire a base base as well as practical experience. The experience thus obtained through the courses proved greatly helpful in grasping complicated concepts as well as in implementing the same efficiently in this work.

Apart from that, I would like to thank my team member, Pavan Sai Kiran, for their constant support, teamwork, and inputs. Their effort and team support were of great help in the completion of the project. Their team support made it a happy and successful experience.

Abstract

In today's digital technology, communication among the components of a chip is of utmost importance. ARM's AMBA (Advanced Microcontroller Bus Architecture) in the chip's communication is widely employed. Being the most important of all the protocols of this type, AXI (Advanced eXtensible Interface) possesses the usual features of high speed of data transfer and high flexibility.

It being the verification as well as design of the AXI memory interface, the origin of AXI communication, data are moved between a slave as well as a master in the interface using five diverse channels. Each of them, in other words, executes a predefined operation such as the movement of the addresses, the data, or the responses. It tries to move as well as read data in the correct shape even in portions or in burst dealings. To verify if the design works right, we make use of System Verilog and UVM (Universal Verification Methodology). These are software tools that help in creating a test environment that supplies the AXI memory with random data and checks the results. It provides help in the identification of the communication errors that are corrected.

Overall, the goal of this dissertation is to create an AXI memory interface and prove that the interface functions effectively by using standard industry methods. The proper verification makes the digital systems less error-prone, better performing, and more reliable.

Chapter 1

Introduction

In the modern era, electronic systems are employed in all sorts of electronic appliances, be they mobile phones and computers or cars and home appliances. These systems are developed through the use of integrated circuits (ICs), consisting of numerous tiny units known as modules. To function well, the entire system requires all these modules, situated at different places, to converse among themselves promptly and accurately.

One of the most popular communication standards of an IC is AMBA, or Advanced Microcontroller Bus Architecture. It has been designed by ARM and is employed by a massive amount of companies all over the world. AMBA supports various types of protocols depending upon various needs. Among them, one of the most popular ones is AXI (Advanced eXtensible Interface). It supports high-speed data transport, reduced latencies, and can handle multiple data transfers simultaneously, known as a burst transaction.

AXI protocol possesses five different communication channels: read address, read data, write address, write data, and write response. These are the parallel communication that facilitate the rapid data transfer between a slave (the receiver device) and a master (the requesting device). AXI, therefore, is a general but robust protocol in the designing of today's systems-on-chip (SoC).

It is, however, not enough in creating the AXI interface. One must ensure that, in all scenarios, it behaves the way one desires. Verification involves the confirmation of whether the design performs fine, if at all, it can process any kind of input without being defective. Verification in real life is money-, time- and resource-consuming because, in the end product, a minor miscommunication results in huge issues.

In order to verify the AXI interface, we use System Verilog as well as UVM (Universal Verification Methodology). System Verilog is a verification and a hardware description language, while UVM is an open-source standard that can be used to develop reusable verification environments that scale. These utilities can be used to create a testbench that can generate random as well as directed test cases to the AXI interface and check if the result obtained is correct. It enables designers to find as well as fix design bugs early, thus saving time as well as money in the further development phases.

It involves the design, as well as the verification, of the AXI memory interface. Its function involves the creation of a workable model of the interface that will be verified by using the latest verification techniques in various states. In doing so, the system will be improved in terms of quality, as well as functionality.

In the later chapters, we will detail the AXI protocol in its entirety, after which we will give the memory interface design. Then, we will detail the verification environment thus developed using UVM, give the test results, and conclude by defining what we have learned through this project and how we can improve the project in the future.

1.1 Organization

This paper is organized as follows:

- Chapter 2 talks about all input and output ports of AXI Memory.
- Chapter 3 is about the design and implementation of AXI Memory Slave.
- Chapter 4 discusses the design and implementation of SystemVerilog verification.
- Chapter 5 is the test results.
- Chapter 6 is the conclusion and possible future work.

1.2 Background

As System-on-Chip (SoC) designs grow more complex, fast and reliable communication between internal blocks becomes very important. The AMBA (Advanced Microcontroller Bus Architecture) protocol by ARM, especially AXI (Advanced eXtensible Interface), is widely used for this purpose. AXI supports high-speed data transfer, burst transactions, and separate read/write channels, which help improve system performance.

In [3], AXI interface modules were designed using Verilog HDL on FPGA, supporting multiple masters and burst transfers. The design included separate read/write channels and a round-robin arbiter. It achieved speeds up to 298.958 MHz, showing AXI's suitability for high-speed embedded systems.

Another design in [4] focused on an AXI-4 master module that connects 16 masters and 16 slaves. It included a decoder and arbiter to manage communication. Simulations confirmed correct read and write functions, proving AXI-4's value in SoC designs.

In [5], an improved AXI model was proposed to reduce delay. A counter allowed the master to send new transactions without waiting for a response, increasing throughput by 23.2%. However, this also increased power usage by 73%, highlighting a trade-off between speed and power.

Verification is also critical. In [4], System Verilog and QuestaSim were used to verify AXI3, achieving 80% functional coverage and 100% assertion success. This shows AXI's reliability for connecting IP blocks.

In [6], a UVM-based verification of an AHB-Lite to AXI bridge was done using Xilinx Vivado. With 16 test cases, the project achieved 100% functional and 98.75% code coverage, proving UVM's strength in verifying complex protocols.

These research efforts show that AXI is a strong solution for connecting multiple functional blocks in SoCs. With proper design and verification, AXI can provide high throughput, scalable architecture, and reliable communication-making it a foundation for modern digital system designs.

Chapter 2
Input and Output ports of AXI

The AXI (Advanced eXtensible Interface) is one of the commonly employed communication protocols in digital circuits. It falls in the AMBA standard of the ARM. AXI permits high-speed data communication between chip blocks. One of the primary features of AXI is the employment of five separate communication channels. These are:

Write Address Channel, Write Data Channel, Write Response Channel, Read Address Channel, Read Data Channel.

Every channel has an input port and an output port. Both ports are used in a manner to transfer various kinds of data like addresses, actual values of the data, and response signals. The channels can be operated in parallel because of this isolation, and the data is transferred faster and more efficiently.

In the standard chip, the data communication of the slave to the master occurs. It is started by the master by the release of read or write requests, and the requests are replied by the slave. For the project, the verification environment functions as the master, generating random data, and the 128-depth, 32-bit memory functions as the slave. It can be used to verify if the slave memory device functions well and can accommodate various types of input data.

Another characteristic of the AXI protocol worth mentioning is the burst mode. With the use of the burst mode, the interface will either transmit or receive several data transfers within a single transfer. It facilitates the increase in speed, aside from the communication performance. Without the values of the data being constantly transferred, the master can transmit several values simultaneously, something that comes in handy when processing data in groups.

AXI supports three types of burst transfers:

- Fixed: In this mode, the address remains the same for every transaction in the burst.
- Increment: In this mode, the address increases w.r.t number of bytes stored in last

transaction.

- Wrap: In this mode, the address increases until a certain limit, then wraps around to the starting address.

The type of burst used decides how the address for the next transaction is calculated. For example, in the increment mode, the address of each new transaction is calculated by adding a fixed value to the previous address. This allows efficient memory access and avoids sending the same address again and again.

In our AXI memory interface design, we implement all five channels and support burst transactions. The verification environment randomly generates the burst type and the data. It sends this data to the memory interface to test how the slave responds under different conditions. It should correctly interpret the nature of the burst needed and find the next address so that the data in the memory may be stored or read accordingly.

2.1 Write Address Channel

Write Address Channel is one of the five AXI protocol channels. The master employs it to send the address and the control information prior to writing data into the slave. It informs the slave where data must be written and how the data must be treated.

Write Address Channel contains some key signals:

- awvalid (Input): It shows that the master has a valid address and control data to provide to the slave. When the awvalid signal has been asserted, the slave will verify the address.

- awid (Input [3:0]): It will be the 4-bit signal that will assign a special ID to every transaction. It enables the slave to keep records of all write operations.

- awlen (Input [3:0]): This signal tells the burst length. The total number of transfers in a burst is calculated as awlen + 1.

- awsize (Input [2:0]): This signal tells the size of each data transfer. It helps the slave know how many bytes of data will be written at each step.

- awaddr (Input [31:0]): This is the 32-bit address where the master wants to write data. It is the starting address of the transaction.

- awburst (Input [1:0]): This signal defines the type of burst. The three types supported are:

  - Fixed: Same address is used for all transfers.

  - Increment: Address increases after each transfer.

  - Wrap: Address wraps around after a certain point.

- awready (Output): This signal is sent by the slave to tell the master that it is ready to accept the address and control signals.

2.2 Write Data Channel

The Write Data Channel in AXI is used to send actual data from the master to the slave after the address has been sent through the Write Address Channel. This channel is responsible for delivering the data that needs to be written into the memory or any other slave device.

The Write Data Channel includes the following important signals:

- wvalid (Input): This signal is sent by the master to show that the data on the channel is valid. When wvalid is high, it means the master is ready to send data to the slave.

- wid (Input [3:0]): This 4-bit signal gives a unique ID for the data transfer. It helps the slave match the data with the correct write address transaction sent earlier.

- wdata (Input [31:0]): This is a 32-bit signal that carries the actual data being written to the slave. It is the main data path used for write operations.

- wstrb (Input [1:0]): This signal tells which lanes (or bytes) in the data are valid. It helps the slave know which parts of the 32-bit wdata are useful. For example, if only the lower byte is valid, the slave will write only that part to memory.

- wready (Output): This signal is sent by the slave to show that it is ready to accept the data. Data transfer happens only when both wvalid and wready are high.

2.3 Write Response Channel

The Write Response Channel in the AXI protocol is used by the slave to send a response back to the master after the write data transfer is completed. This response helps the master know whether the write transaction was successful or if there was any error. In our design, the master is the verification environment, and the slave is a memory device.

The Write Response Channel contains the following important signals:

- bready (Input): This signal is sent by the master to tell the slave that it is ready to receive the write response. If bready is low, the slave must wait before sending the response.

- bid (Output [3:0]): This 4-bit signal gives the unique ID for the completed write transaction. It matches the awid and wid sent earlier so that the master knows which transaction this response belongs to.

- bresp (Output [1:0]): This 2-bit signal provides the status of the write operation. It tells whether the write was successful, or if there was an error. Common response codes include:

  - 00: OKAY – Write completed successfully

  - 01: EXOKAY – Exclusive access okay

  - 10: SLVERR – Slave error

  - 11: DECERR – Decode error

- bvalid (Output): This signal tells the master that the response is valid and ready to be read. The master should only read the response when both bvalid and bready are high.

The Write Response Channel ensures that every write operation is confirmed, making the communication reliable. It helps in debugging and verifying if the data sent by the master was properly written to the slave.

2.4 Read Address Channel

The Read Address Channel in the AXI protocol is used by the master to send the read request to the slave. It has the source location where data will be read from, with details of how the data will be transferred. After the slave has received this address, it prepares the data for return through the Read Data Channel.

The Read Address Channel includes the following signals:

- arready (Output): This signal is sent by the slave to tell the master that it is ready to receive the read address and control information. Data transfer does not begin unless arready and arvalid are high.

- arid (Input [3:0]): It holds the 32-bit address from which the master will read the data. It will be the starting address of the read operation.

- araddr (Input [31:0]): This is the 32-bit address from where the master wants to read data. It is the starting address of the read operation.

- arlen (Input [3:0]): This signal tells the length of the burst. The number of data transfers is equal to arlen + 1.

- arsize (Input [2:0]): This signal defines the size of each data transfer, in bytes. It tells the slave how much data to send in each step of the burst.

- arburst (Input [1:0]): This signal shows the burst type. The burst type can be:

    - Fixed – address remains the same

    - Incremental – address increases with each transfer

    - Wrapping – address wraps around after a point

- arvalid (Input): This signal is sent by the master to show that the read address and control signals are valid.

2.5 Read Data Channel

The Read Data Channel in the AXI protocol is responsible for sending data from the slave to the master after a read request has been made through the Read Address Channel. Once the slave receives the read address and control signals, it uses this channel to return the requested data to the master.

The Read Data Channel includes the following important signals:

- rid (Output [3:0]): This 4-bit signal provides the ID of the read transaction. It matches the arid sent earlier in the Read Address Channel. This helps the master connect the received data to the correct request.
- rdata (Output [31:0]): This is the 32-bit data value returned by the slave. It is the actual data stored at the memory location requested by the master.
- rresp (Output reg [1:0]): This 2-bit signal shows the read response status. It tells the master whether the read operation was successful or if there was an error. Common values include:
    - 00: OKAY – Read successful
    - 10: SLVERR – Slave error
    - 11: DECERR – Decode error
- rlast (Output reg): This signal shows the end of the burst. In a burst read, rlast will have a high value in the final data of the series alone. It lets the master know that no more data will be arriving in this transaction.
- rvalid (Output reg): It will be pulled high by the slave when the data in rdata is valid and ready to be read by the master.
- rready (Input): It is the signal provided by the master that the master is ready to accept the data. The data will be transferred when the rvalid as well as the rready will be high.

Chapter 3
Implementation of AXI Interface

In this chapter, we describe how the AXI memory interface is implemented using finite state machines (FSMs) and how the memory block is built. The memory used in this project is 32-bit wide, has a depth of 128 locations, and each memory location is 8-bit addressable. This setup is used to test and verify the read and write operations through the AXI protocol.

Write Channel Overview

The write operation in AXI starts with the master sending the write address and control information through the Write Address Channel and then sending the actual data through the Write Data Channel. The Write Response Channel is used to confirm whether the write operation was successful or not.

One of the key parameters in a write transaction is awlen, which represents the burst length. The actual number of transactions is awlen + 1. For example, if awlen = 7, it means the master will perform 8 write transactions in one burst.

Another important signal is awsize, which defines the size of each transaction in bytes. If awsize = 4 (which means 4 bytes per transaction), then the total data transferred in the burst would be 8 transactions * 4 bytes = 32 bytes. This helps in efficient data transfers with fewer control signals.

Once the write data is received, the Write Response Channel sends a response to the master using the bresp signal. This response indicates if the write was successful, if the address was valid, or if there was an error (like out-of-bound access).

Read Channel Overview

The read operation is similar to the write process but in reverse. The master first sends the read address and control information using the Read Address Channel. Then the slave responds with the requested data through the Read Data Channel.

The actual data is sent using the rdata signal. This data was previously stored in memory using the write process. If the master tries to read from an address that was not written to earlier, the memory cannot return valid data. In this case, the slave sends an error status using the rresp signal.

The rresp signal plays a key role in indicating the status of the read transaction. For example, if data was never written to a certain address, reading from it will cause an error which is reported through rresp.

Finite State Machine (FSM) Control

To implement the AXI interface, multiple finite state machines are used to handle different channels. Each FSM manages the handshaking signals like valid and ready, controls the burst logic using awlen and awsize, and ensures proper data movement in and out of memory. The memory is implemented as a 2D array: reg [31:0] memory [0:127];

The FSMs also check address ranges and trigger error responses if the master tries to access memory outside the allowed 128-depth space.

Through this implementation, the memory interface supports full AXI read and write operations, including burst transfers, response checking, and error handling.

3.1 Write Address FSM

The Write Address Channel in the AXI interface is controlled by a finite state machine (FSM) with three main states: awidle, awstart, and awreadys. This FSM manages the process of receiving the write address and control signals from the master before the actual data write begins. Initially, the FSM is in the awidle state. In this state, all signals are set to their default values, and the system waits for a reset signal to be deactivated. The FSM stays in this state as long as the reset is asserted or no write address transaction is demanded. This state stabilizes the system and prepares it to receive new transactions. When reset is disabled, the FSM goes to the awstart state. In this state, the write address-related signals are prepared and set to values that will be utilized for writing data to memory. The write transaction itself does not begin in this state, however. Rather, the FSM waits for the master to assert the awvalid signal, which means that the master is transmitting a valid write address and control information. If awvalid is high, the FSM goes to the awreadys state. This state indicates that the slave is ready to receive the write address and control signals. The actual write transaction begins in this state. It is important to note that the FSM can only enter the awreadys state if awvalid is asserted; otherwise, it remains in the awstart state, continuously checking for a valid address. Once the transaction is accepted in the awreadys state, the FSM returns to the awidle state to wait for the next write address transaction. This cycle ensures proper handshaking between the master and slave, allowing reliable and synchronized data transfer.

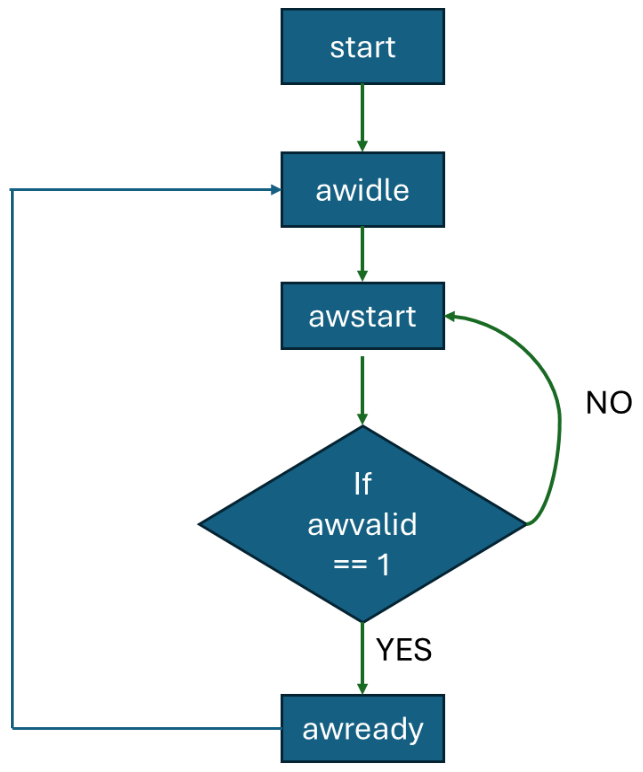


Figure 3-1 Write Address Channel FSM

3.2 Burst Modes

Before implementing the Write Data Channel FSM, it is important to understand the three types of burst modes supported by the AXI protocol: fixed, increment, and wrap. Each burst mode defines how the address changes during multiple data transfers in a single burst.

Fixed Burst Mode: In fixed burst mode, the address remains the same for all transactions within the burst. For example, if there are 8 transactions in one burst, the address used for all 8 transactions will be identical. Typically, the first transaction writes data to the memory location, and subsequent transactions may overwrite or update the same address repeatedly until the burst completes. This mode is often used for sensor data, where data is continuously updated but always read from or written to the same device or port address. If the data size is 8 bits, it is stored in a single memory address. For data larger than 8 bits, the data is stored in consecutive memory addresses.

The wstrb signal plays a key role in fixed mode. It specifies which byte lanes of the 32-bit data are valid. Since the memory is 8 bits wide per address, wstrb determines how many consecutive memory locations are used to store the data. For example, if wstrb is 4'b0001, only the least significant 8 bits are valid and stored in one address. If wstrb is 4'b0111, three lanes are valid, so data is stored in three consecutive addresses.

Increment Burst mode: In increment burst mode, the address used for each data transaction increases after every transfer. This means that the data is written to or read from consecutive memory locations, one after another. The amount by which the address increases depends on how many bytes of data are valid in each transaction, which is indicated by the wstrb signal.

For example, consider a 32-bit data bus divided into four 8-bit lanes. The wstrb signal is 4 bits wide, where each bit corresponds to one 8-bit lane. If a bit in wstrb is set to 1, it means that the corresponding lane contains valid data that should be stored in memory.

Suppose wstrb is 4'b0011. This means the two least significant lanes (each 8 bits) have valid data. Since each memory location can store 8 bits, the data will occupy two consecutive memory addresses. First 8 bits of data (wdata[7:0]) are written at the present address, and the next 8 bits (wdata[15:8]) are written at the next address (present address + 1). After write, the address is increased by 2 to write the next accessible location of memory for the next transaction. It will ensure that the next write of data will not override the present data and the memory will be efficiently being used. It will be carried out for every transaction of the burst, the address being increased by the number of valid lanes of every transfer. It is most effective while writing or reading blocks of sequential data, such as in arrays or buffers.

Wrap Burst Mode: Wrap burst mode of the AXI protocol involves wrapping the address crossing a given boundary. While the fixed or increment burst mode is not supported using this feature, the wrap burst mode is supported using the address alternating among a given set of memory locations repeatedly in a burst transaction. Three significant values need to be specified to support the wrap burst mode: the boundary, the wrap boundary, and the wrap address.

Boundary is the total burst size in bytes. It is determined by the product of the length of the burst plus one (awlen +1) and the size of the transfer (awsize). Algebraically, we can compute it as:

$$boundary = (awlen + 1) * awsize \qquad (1)$$

Then, the wrap boundary is calculated by rounding down the start address to the next multiple of the boundary. It is accomplished by dividing the start address by the boundary, getting the integer part of the result, and multiplying the result by the boundary again:

$$wrap\ boundary = \left(\frac{starting\ address}{boundary}\right) * boundary \qquad (2)$$

Lastly, the wrap address is resolved by adding the boundary to the wrap boundary:

$$wrap\ address = wrap\ boundary + boundary \qquad (3)$$

During the burst, if the next address that must be read reaches the wrap address, the address wraps around the wrap boundary. For instance, the wrap address being 32, the next address that must be stored being 32, the address wraps around to 0. Similarly, if the next address being 33, wraps around to 1; if 34, wraps around to 2, and so on.

This wrapping behavior allows the burst to cycle through a fixed block of memory repeatedly, which is useful in applications like circular buffers or streaming data where continuous access to a limited memory region is required.

3.3 Write Data and Write Response FSM

Write Data FSM:

The Write Data FSM controls the process of receiving write data from the master and storing it in the memory. It has five main states: WIDLE, WSTART, WADDR_DEC, WREADY, and WVALID.

WIDLE (Write Idle): This is the initial state after a reset. In this state, all temporary variables are initialized. The signal wready is set to 0, indicating that the slave is not yet ready to accept data. The write length counter (wlen_count) is reset to zero. The FSM then moves unconditionally to the next state, WSTART, on the next clock cycle.

WSTART (Write Start): In this state, the FSM waits for the master to provide valid write data. It monitors the wvalid signal. If wvalid is high, meaning the master is sending valid data, the FSM samples the data (wdata) and moves to the WADDR_DEC state to decode the address. If wvalid is low, the FSM remains in the WSTART state, waiting for valid data.

WADDR_DEC (Write Address Decode): This is a crucial state where the FSM calculates the next write address. For the first transaction in a burst, the next address is set to the address provided by the master (awaddr). For subsequent transactions, the next address is determined by the burst mode (fixed, increment, or wrap) using a function that calculates the correct address. After decoding the address, the FSM moves to the WREADY state.

WREADY (Write Ready): In this state, the slave signals that it has successfully processed the current data by setting wready high. The FSM checks if the current data is the last in the burst by examining the wlast signal. If wlast is high, the burst is complete. The FSM resets internal counters and returns to the WIDLE state. If wlast is low, the burst continues. The FSM keeps wready high to accept the next data and transitions to the WVALID state.

WVALID: This state manages the continuation of the burst. It writes wready low and increments the write length counter (wlen_count). Then the FSM returns to the WSTART state to wait for the next valid data from the master.

It is executed for every transaction within the burst, and the address is incremented by the number of valid lanes in the transfer. It is most beneficial in writing or reading large blocks of data sequentially, for instance, in buffers or arrays.

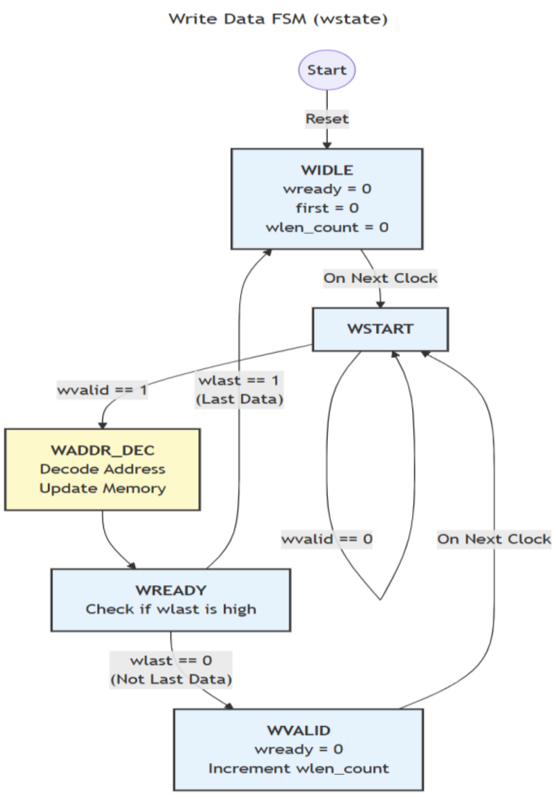


Figure 3-2 Write Data Channel FSM

Write Response FSM: The Write Response FSM governs the write response issued by the slave to the master upon acceptance of the write data. It guarantees that the master will receive an indication of whether the write transaction has succeeded or errors are being experienced.

It begins in the bidle state, waiting for termination of a write transaction. After termination of the write data burst, the FSM moves to the bdetect_last state to determine the last data transfer. It moves to the bstart state, starting to assemble the write response. The response holds the result of the write transaction, e.g., error or successful. After transmitting the response, the FSM will be in the bwait state, waiting for the response being recognized by the master. After that recognition, the FSM moves to the bidle state, waiting for the next write transaction.

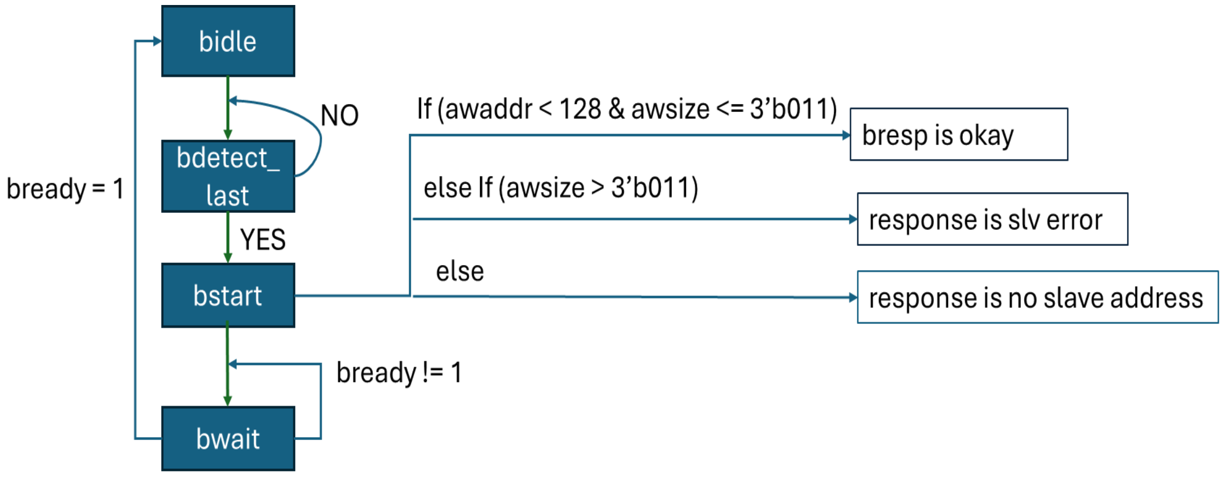


Figure 3-3 Write Response Channel

3.4 Read Data and Read Response FSM

In the AXI memory interface, the read operation is managed by two key finite state machines (FSMs): the Read Address FSM and the Read Data FSM. The two FSMs work together to accept read requests from the master, read data from memory, and transfer the data back to the master in a efficient and correct way.

Read Address FSM: The Read Address FSM controls the receipt of read address requests from the master. The three main states are ARIDLE, ARSTART, and ARREADY.

ARIDLE (Read Address Idle): This is the reset state. Here, the signal arready is 0, indicating that the slave is not ready to accept a read address. The FSM idles here until it is ready to process a new read request.

ARSTART (Read Address Start): The FSM moves to this state in the following clock cycle. In this state, it is waiting for the master to assert the arvalid signal, indicating that a valid read address is available on the bus. If arvalid is 0, the FSM remains in this state waiting for a valid address.

ARREADY (Read Address Ready): When arvalid is 1, the FSM transitions to the ARREADY state. In this state, the slave asserts arready to indicate that it has accepted the read address. The FSM also latches the read address for processing in the future. Then the FSM returns to the ARSTART state to wait for the next read address.

.

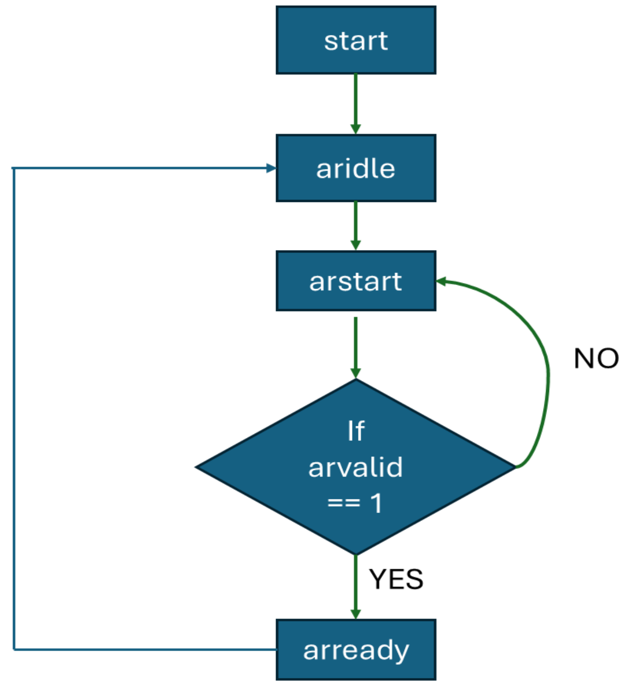


Figure 3-4 Read Address Channel FSM

Read Data FSM: The Read Data FSM manages the transfer of the required data to the master. It has several states: RIDLE, RSTART, RWAIT, RVALID, and RERROR.

RIDLE (Read Idle): This is the reset state. The FSM resets all the output signals and waits for a read request from the master.

RSTART (Read Start): The FSM goes into this state when the master asserts arvalid. It checks the read address and transfer size validity. If both are OK, it sets the response signal (rresp) to

OK (2'b00) and goes to the RWAIT state. If the address or size is invalid, it sets an error response (rresp = 2'b10 or 2'b11) and goes to the RERROR state.

RWAIT (Read Wait): The FSM in this state waits for the master to be ready to accept data, indicated through the rready signal. The FSM keeps rvalid low and increments the length counter (len_count) to count the number of data beats sent.

RVALID (Read Valid): Once ready, the master asserts rvalid and sends the data. It also checks if the present data beat is the last one in the burst by comparing len_count with the burst length (arlen). If it is the last transfer, the FSM sets to return to the idle state. Otherwise, it loops back to RSTART to process the next data beat.

RERROR (Read Error): The FSM enters this state when the address or size is invalid. It asserts the read response signal to indicate an error and also increments the length counter. Having sent the error response, it takes the same path as RVALID to restart for the next data beat or to finish the transaction.

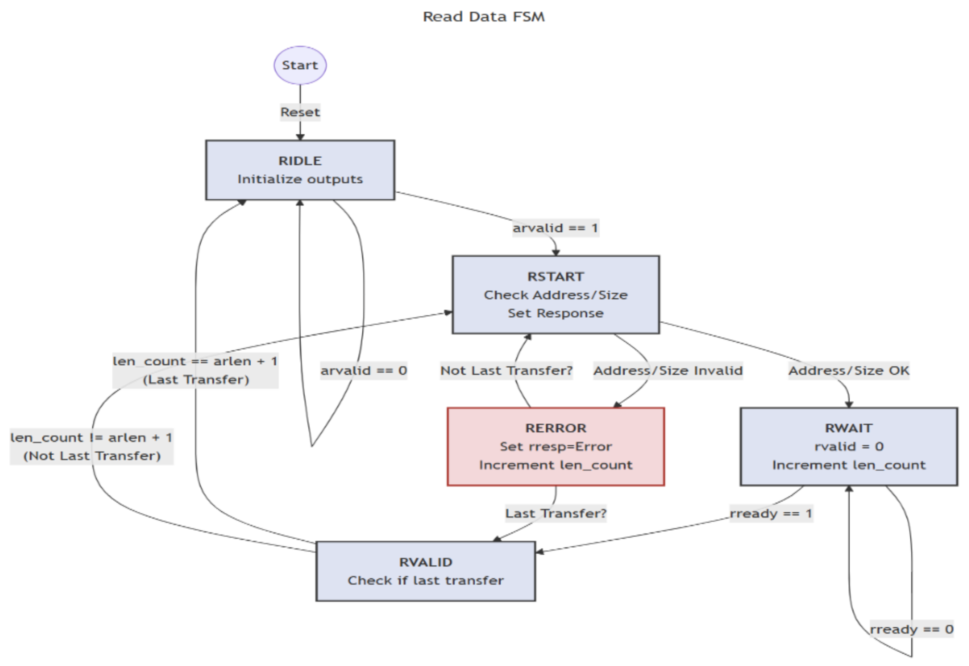


Figure 3-5 Read Data Channel FSM

Chapter 4

SystemVerilog and UVM Verification Design

As digital designs become more complex, verifying them quickly and accurately is very important. Verification takes a large part of the total time in a project because it ensures the design works correctly before manufacturing. For complex protocols like AXI, manual testing is not enough, so automated verification methods are needed.

SystemVerilog is a hardware description and verification language that helps engineers create advanced testbenches. It supports object-oriented programming, where reusable and flexible verification building blocks can be composed. Engineers can write tests that generate random inputs with some constraints, auto-verify outputs, and measure how much of the design is hit. Universal Verification Methodology (UVM) is a popular framework derived from SystemVerilog. It provides a set of base classes and guidelines to build structured and reusable verification environments. UVM allows structuring the verification process in layers for ease of management and growth.

In this AXI project, the testbench is written in SystemVerilog and UVM to check the memory interface. The testbench includes constructs like generators to provide test data, drivers to drive signals to the design, monitors to observe outputs, and scoreboards to compare actual and actual outputs. This hierarchical approach allows efficient testing of any AXI features, for example burst transfers and multiple channels.

Using SystemVerilog and UVM makes the verification environment reusable and modular. It saves verification time and allows bugs to be found early. It also allows for easier reuse of the testbench for other projects in the future or other AXI configurations.

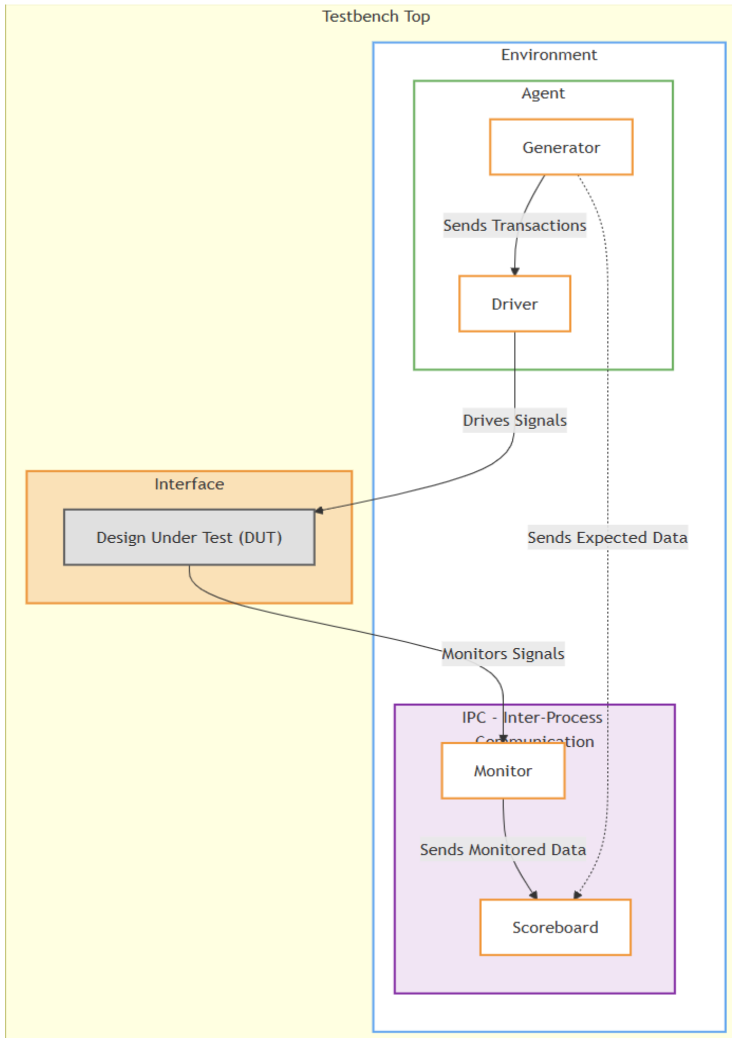


Figure 4-1 SystemVerilog Environment

Testbench Top

The Testbench Top is the top-level module of the verification environment. It is the top-level container that contains the verification environment and the Design Under Test (DUT) inside it. Its major task is to create instances of the Environment class and also of the DUT. It wire-connects the DUT to the verification environment using a virtual interface. Using this wiring, verification blocks to communicate with the DUT signals without connecting wires. The Testbench Top organizes the entire verification flow and ensures the DUT and environment are expected to interact during simulation.

4.1 Environment

Environment is a top-level container that holds the main verification blocks. It manages the Agent, Scoreboard, and other verification elements. The Environment synchronizes the generation of stimulus, monitors the activity of DUT, and checks if the DUT operates normally. It is also the central point where different verification components congregate to perform their work. By organizing these fragments, the Environment helps create a reusable and scalable verification environment.

4.2 Generator

Generator is part of the stimulus layer. Its main function is to create sequences of transactions that represent different test cases. It has the ability to generate random or guided transactions, which are class-based data objects that specify the stimulus. The Generator sends these transactions to the Driver so that they can be applied on the DUT. It also sends expected results to the Scoreboard, which uses them as a reference when checking the output of the DUT. This helps in ensuring that the DUT is behaving as anticipated.

4.3 Agent

Agent is a reusable verification block that drives one interface of the DUT. It contains the Generator and Driver that generate and drive transactions onto the DUT. Agent offers a connection between high-level test scenario and low-level signal driving. It receives generated transactions and drives them onto the DUT appropriately interface. The Agent can then be reused for other projects or interfaces.

4.4 Driver

The Driver is the closest module to the hardware. It receives transactions from the Generator and translates them into pin-level signals. The Driver stimulates these signals onto the input ports of the DUT, based on the interface's timing and protocol requirements. It aligns signal transitions with the system clock to enable successful communication. The Driver plays a critical role in the application of the test stimulus to the DUT.

4.5 Monitor

The Monitor is a passive component which observes the DUT's output signals through the Interface. It captures signal activity without affecting the functioning of the DUT. The Monitor converts raw signal information to transaction objects, the real output of the DUT. The observed transactions are then passed on to the Scoreboard to be compared against expected output. The Monitor helps to ensure that the DUT acts appropriately by providing proper data to be processed.

4.6 Interface

The Interface is a package with all the signals that connect the verification environment and the DUT. It defines the physical pins and timing of the communication protocol. In contrast to other modules, the Interface is not class-based but rather a hardware description package. The Driver actively drives signals on the Interface, since the Monitor merely observes these signals. The Interface is a pure separation of the DUT and verification components.

4.7 Inter Process Communication (IPC)

IPC block represents communication paths among class-based components in the verification context. Communication is typically handled by UVM's Transaction-Level Modeling (TLM) ports. The Monitor sends trapped transactions to the Scoreboard, and expected transactions are sent by the Generator. This enables the Scoreboard to receive actual and expected data and compare and validate the DUT's behavior exactly.

4.8 Scoreboard

The Scoreboard is the main checker of the verification environment. It receives expected results from the Generator and actual results from the Monitor. The Scoreboard cross-checks these two streams of data to see if there are any mismatches or discrepancies. It keeps verification statistics, such as the number of transactions verified, passes, and failures. If it ever detects a mismatch, it reports an error, indicating a possible bug in the DUT. The Scoreboard is invaluable in checking the design's correctness.

Chapter 5
Tests and Results

In this project, the AXI memory slave was designed in SystemVerilog and verified using a testbench written in UVM and SystemVerilog. The design and verification environment were all executed on EDA Playground, an open-source HDL simulation platform. This allowed us to execute the AXI interface with different test cases and check how it performs, thereby confirming the design meets the AXI protocol specifications.

5.1 Simulation

5.1.1 Write Channel Results

The waveform shown in the figure is a standard write transaction on the AXI interface. It includes prominent signals such as the write address (awaddr), burst length (awlen), burst type (awburst), write data (wdata), write strobes (wstrb), and control signals such as awvalid, wvalid, wready, and wlast.

From figure 5-1, a few observations can be made. The write address channel starts the transaction with awvalid being asserted and the address being 0x5. The burst length is 7, i.e., there are 8 data transfers in this burst.The burst type is set to increment mode (awburst = 1), so the address increases after each data transfer. The write data channel shows valid data (wvalid) being sent one after another, with the data values changing as expected for each transfer. The wstrb signal is set to 4'b1111, indicating that all four bytes of the 32-bit data are valid for every transfer. The slave asserts the wready signal to acknowledge the acceptance of each data beat. Finally, the wlast signal is asserted on the last data transfer, marking the completion of the burst.

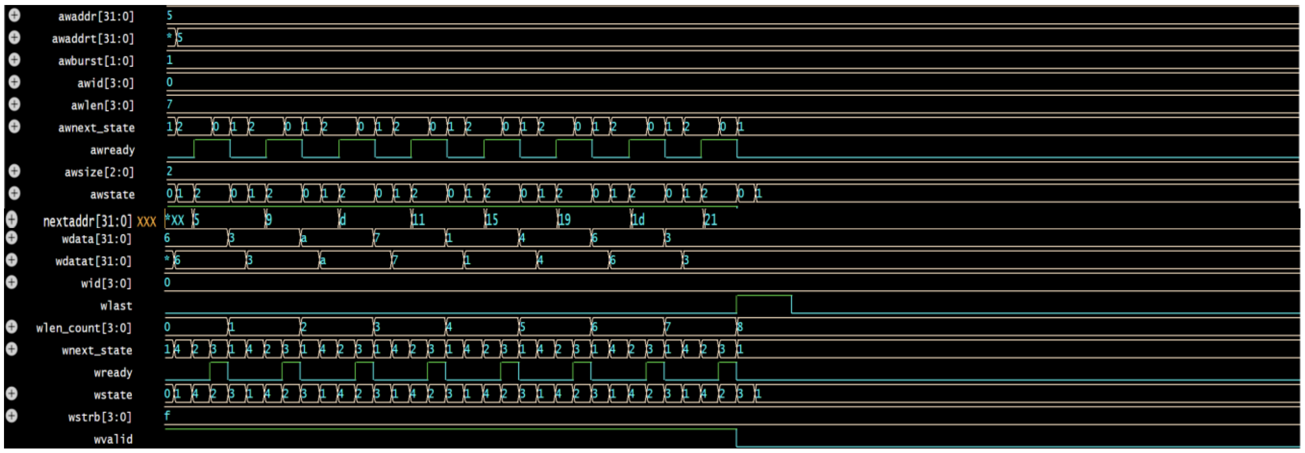


Figure 5-1 AXI Write Address and Data Channels Simulation Results

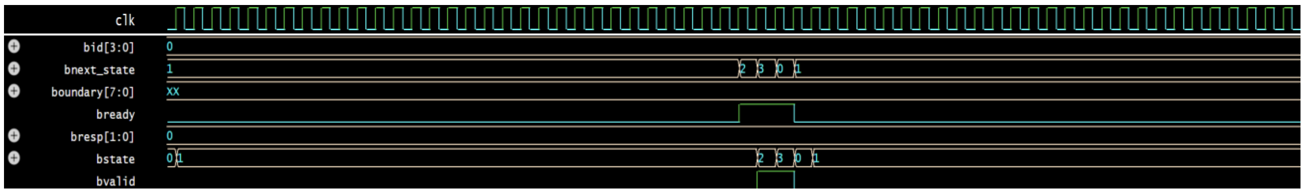


Figure 5-2 AXI Write Response Channel Results

Figure 5-2 The write response channel signals, bvalid and bresp, confirm that the write transaction has completed successfully by sending an OKAY response (bresp = 2'b00). The FSM states for the write address (awstate), write data (wstate), and write response (bstate) channels transition smoothly through their expected sequences, demonstrating correct handshaking and proper handling of burst transactions.

5.1.2 Read Channel Results

The Figures 5-3 & 5-4 represents a typical read transaction on the AXI interface, showing key signals such as the read address (araddr), burst length (arlen), burst type (arburst), read data (rdata), read response (rresp), and control signals including arvalid, arready, rvalid, rready, and rlast. The transaction begins with the read address channel asserting arvalid and setting the address to 0x5. The burst length is 7, indicating that eight data transfers will occur in this burst. The burst type is set to increment mode (arburst = 1), so the address increases after each data transfer. The slave acknowledges the read address by asserting arready. On the read data channel, valid data (rvalid) is sent sequentially, with data values changing as expected for each transfer. The master signals its readiness to accept data by asserting rready. The rlast signal is asserted on the final data transfer, marking the end of the burst. The read response (rresp) remains zero throughout, indicating an OKAY response with no errors. The FSM states for both the read address (arstate) and read data (rstate) channels transition correctly through their expected sequences, demonstrating proper handshaking and burst handling. Additionally, the nextaddr and retaddr signals show the address progression during the burst, confirming that the address increments correctly for each data beat. The len_count signal tracks the number of data transfers, reaching the burst length before the transaction completes.

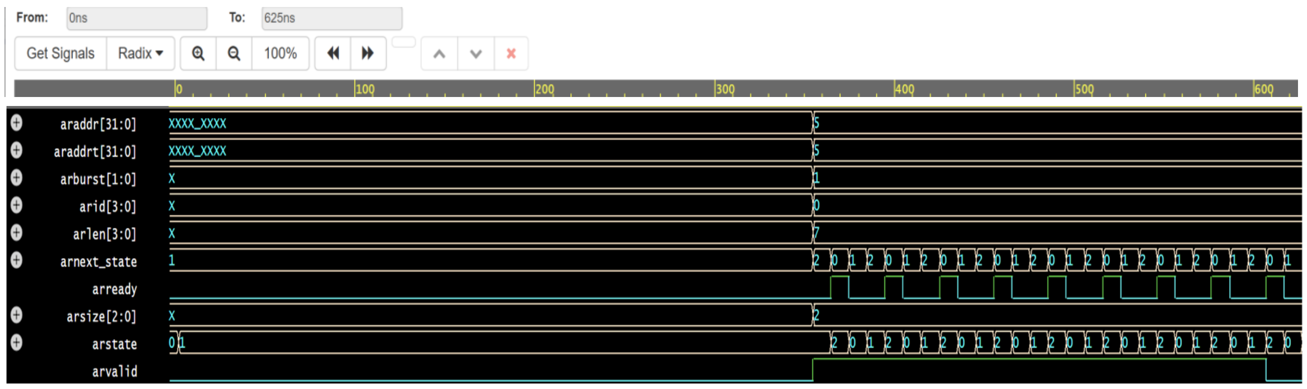


Figure 5-3 Read Address Channel

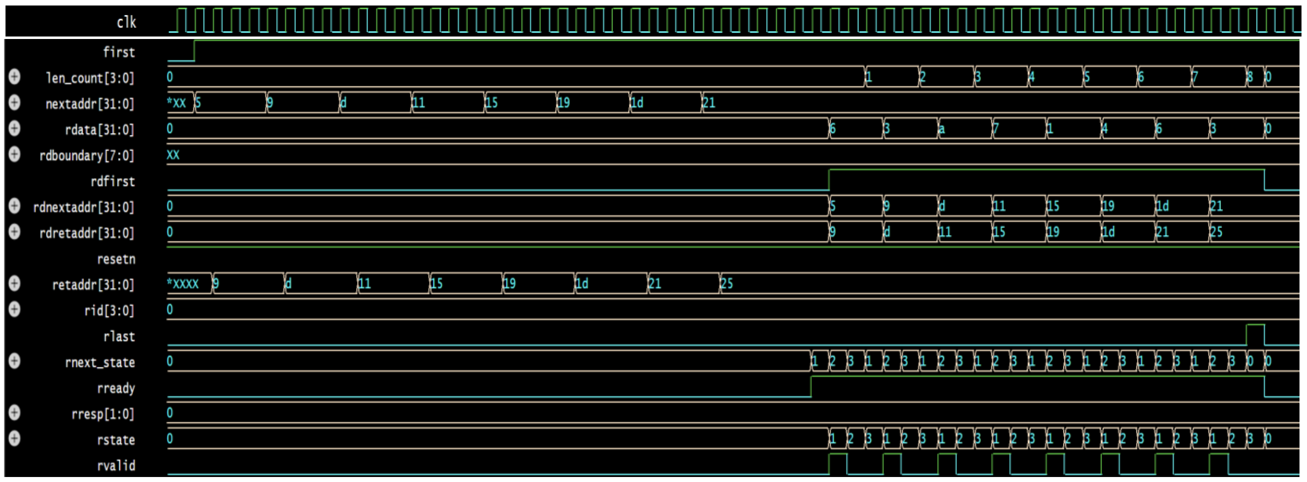
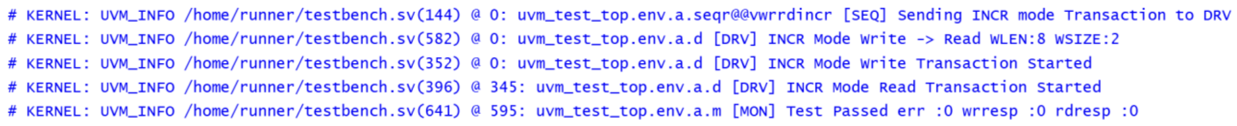


Figure 5-2 Read Data Channel

5.2 UVM Verification Results

These log messages are informational outputs from the UVM testbench during simulation. They show the progress of an increment (INCR) mode transaction being sent to the driver (DRV), including the start of write and read transactions with specific parameters like burst length (WLEN) and size (WSIZE). The final message from the monitor (MON) indicates that the test passed successfully, with no errors reported in the write or read responses (wrresp and rdresp are zero). This confirms that the AXI interface handled the transaction correctly.



```
# KERNEL: UVM_INFO /home/runner/testbench.sv(144) @ 0: uvm_test_top.env.a.seqr@@vwrrdincr [SEQ] Sending INCR mode Transaction to DRV
# KERNEL: UVM_INFO /home/runner/testbench.sv(582) @ 0: uvm_test_top.env.a.d [DRV] INCR Mode Write -> Read WLEN:8 WSIZE:2
# KERNEL: UVM_INFO /home/runner/testbench.sv(352) @ 0: uvm_test_top.env.a.d [DRV] INCR Mode Write Transaction Started
# KERNEL: UVM_INFO /home/runner/testbench.sv(396) @ 345: uvm_test_top.env.a.d [DRV] INCR Mode Read Transaction Started
# KERNEL: UVM_INFO /home/runner/testbench.sv(641) @ 595: uvm_test_top.env.a.m [MON] Test Passed err :0 wrresp :0 rdresp :0
```

Figure 5-5 UVM INFO Results

Chapter 6
Conclusions

This work introduced a complete AXI memory slave interface design and verification using SystemVerilog and Universal Verification Methodology (UVM). The design includes all significant AXI features like multiple independent channels, burst transactions, and different burst modes like fixed, increment, and wrap. The verification environment was built to heavily test the functionality of the interface, including correct data transfer, address management, and response signaling.

The use of SystemVerilog and UVM provided a structured and reusable methodology for verification. Testbench had components such as agents, drivers, monitors, and scoreboards that worked together to generate stimulus, observe DUT behavior, and check for correctness. Randomized and directed test cases were employed to cover lots of test scenarios, including burst transactions and edge cases. The verification result, as defined by simulation on the EDA Playground platform, guaranteed that the design meets the requirements of AXI protocol and handles all the tested cases appropriately.

The finite state machines (FSMs) employed for write address, write data, write response, read address, and read data channels performed well in managing the complex handshaking and data transfer of AXI. Waveforms and state transitions demonstrated good master-slave synchronization, proper management of burst length, and proper response signaling.

In short, the project demonstrates the worth of combining a good design with a good verification environment. The modularity and reusability of the testbench that used UVM were not only beneficial to increase verification efficiency, but also made it easy to add or adjust the environment to accommodate future AXI-based designs.

6.1 Future Work

This project effectively implemented and validated a minimal AXI memory interface. But there are several ways that it can be improved and augmented in the future. The first key step would be to add support for the full AXI4 protocol. This means implementing such things as out-of-order transactions, exclusive access, and quality of service signals. They are needed for advanced chip designs and will enrich the interface. Improvement must also be made in the verification environment. More coverage checks and testing of hard cases will enable finding sneaky bugs. Formal verification or hardware emulation will also accelerate testing and make it more reliable. The model for memory can be made more robust to support different data sizes and allow simultaneous multiple access. Also, providing support for other types of AXI such as AXI-lite or AXI-stream will increase the flexibility of the design.

Finally, the AXI memory interface can be verified in a larger system containing many masters and slaves. This will enable one to see how the interface really works in real systems and develop skills in system-level verification. The future advancements will strengthen the AXI memory interface, making it more adaptable and ready for real use.

References

[1] Learn the architecture - An introduction to AMBA AXI

[2] Verification Series Part 1 : SystemVerilog Fundamentals | Udemy – Lectures by Kumar khandagle

[3] R. Bhaktavatchalu, B. S. Rekha, G. A. Divya and V. U. S. Jyothi, "Design of AXI bus interface modules on FPGA," 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), Ramanathapuram, India, 2016, pp. 141-146, doi: 10.1109/ICACCCT.2016.7831617.

[4] Guruprasad, S., & Sudharshan, K. (2011). Design and Analysis of master module for AMBA AXI-4. *NCECS-2011) at Siliguri Institute for technology*.

[5] A. Gollapudi, A. K. Kumar and M. L. N. Charyulu, "Implementation of an Optimized AXI using Verilog," *2024 7th International Conference on Circuit Power and Computing Technologies (ICCPCT)*, Kollam, India, 2024, pp. 684-690, doi: 10.1109/ICCPCT61902.2024.10673101.

[6] N. K. P, D. V, A. M, S. K. R and E. S, "Design and Verification of AMBA AXI3 Protocol for High Speed Communication," *2022 Smart Technologies, Communication and Robotics (STCR)*, Sathyamangalam, India, 2022, pp. 1-5, doi: 10.1109/STCR55312.2022.10009388.

[7] Pradeep, S., & Laxmi, C. (2014). Design and verification environment for AMBA AXI protocol for SoC integration. *International Journal of Research in Engineering and Technology*, *3*(05), 2014.

[8] Toe, Aung, "Design and Verification of a Round-Robin Arbiter" (2018). Thesis. Rochester Institute of Technology.

[9] K. C. Lee, "A variable round-robin arbiter for high speed buses and statistical multiplexers," in *[1991 Proceedings] Tenth Annual International Phoenix Conference on Computers and Communications*, Mar 1991, pp. 23–29.

[10] Pradeep, S., & Laxmi, C. (2014). Design and verification environment for AMBA AXI protocol for SoC integration. *International Journal of Research in Engineering and Technology*, *3*(05), 2014.

[11] Goossens, K., Dielissen, J., Gangwal, O. P., Pestana, S. G., Radulescu, A., & Rijpkema, E. (2005, March). A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *Design, Automation and Test in Europe* (pp. 1182-1187). IEEE.

[12] Bamford, N., Bangalore, R. K., Chapman, E., Chavez, H., Dasari, R., Lin, Y., & Jimenez, E. (2006, December). Challenges in system on chip verification. In Seventh International Workshop on Microprocessor Test and Verification (MTV'06) (pp. 52-60). IEEE.

[13] Chen, M., Zhang, Z., & Ren, H. (2021, October). Design and verification of high performance memory interface based on axi bus. In 2021 IEEE 21st International Conference on Communication Technology (ICCT) (pp. 695-699). IEEE.

[14] Patel, S. (2022). Functional Verification of Universal Memory Controller (Doctoral dissertation, Institute of Technology).

[15] Patell, N. (2021). Verification Challenges and Solutions in Verifying High Speed Interfaces Like PCIe or DDR.

[16] Chen, C. Z., Liu, X., & Wu, H. (2022, June). An FPGA-Based Verification Platform for High-Speed Interface IPs. In 2022 China Semiconductor Technology International Conference (CSTIC) (pp. 1-3). IEEE.

[17] Melikyan, V., Harutyunyan, S., Kirakosyan, A., & Kaplanyan, T. (2021, September). Uvm verification ip for axi. In 2021 IEEE East-West Design & Test Symposium (EWDTS) (pp. 1-4). IEEE.

[18] Nakkala, S., Vaddavalli, T., & Arja, S. S. (2024, May). Design and Verification of AMBA AXI Protocol. In 2024 International Conference on Electronics, Computing, Communication and Control Technology (ICECCC) (pp. 1-6). IEEE.

Appendix I

Arbiter Source Code

```
/////////////////////////////////////////////
module axi_slave(
  ///////////////global control signals
  input clk,
  input resetn,

  //////////////////write address channel

  input  awvalid,  /// master is sending new address
  output reg awready,  /// slave is ready to accept request
  input [3:0] awid, ////// unique ID for each transaction
  input [3:0] awlen, ////// burst length AXI3 : 1 to 16, AXI4 : 1 to 256
  input [2:0] awsize, ////unique transaction size : 1,2,4,8,16 ...128 bytes
  input [31:0] awaddr, ////write adress of transaction
  input [1:0] awburst, ////burst type : fixed , INCR , WRAP

  ///////////////////write data channel

  input wvalid, //// master is sending new data
  output reg wready, //// slave is ready to accept new data
  input [3:0] wid, /// unique id for transaction
  input [31:0] wdata, //// data
  input [3:0] wstrb, //// lane having valid data
  input wlast, //// last transfer in write burst

  //////////////write response channel

  input bready, ///master is ready to accept response
  output reg bvalid, //// slave has valid response
  output reg [3:0] bid, ////unique id for transaction
  output reg [1:0] bresp, /// status of write transaction

  ///////////// read address channel

   output reg     arready,  //read address ready signal from slave
   input [3:0]    arid,     //read address id
   input [31:0]   araddr,           //read address signal
   input [3:0]    arlen,    //length of the burst
   input [2:0]    arsize,           //number of bytes in a transfer
   input [1:0]    arburst, //burst type - fixed, incremental, wrapping
   inputarvalid, //address read valid signal

 //////////////////read data channel
        output reg [3:0] rid,              //read data id
        output reg [31:0]rdata,     //read data from slave
        output reg [1:0] rresp,            //read response signal
        output reg rlast,           //read data last signal
        output reg rvalid,                 //read data valid signal
        input rready

);



  //axi_if vif();
  typedef enum bit [1:0] {awidle = 2'b00, awstart = 2'b01, awreadys = 2'b10} awstate_type;
  awstate_type awstate, awnext_state;

  reg [31:0] awaddrt;

  //////reset decoder
  always_ff@(posedge clk , negedge resetn)
    begin
      if(!resetn) begin
        awstate <= awidle;  ///idle state for write address FSM
      wstate  <= widle;   ///idle state for write data fsm
      bstate  <= bidle;  ///// idle state for write response fsm
      end
    else
      begin
      awstate <= awnext_state;
      wstate  <= wnext_state;
      bstate  <= bnext_state;
      end
  end


////////////fsm for write address channel
always_comb
  begin
    case(awstate)
    awidle:
    begin
       awready  = 1'b0;
       awnext_state = awstart;
    end

    awstart:
    begin
      if(awvalid)
        begin
        awnext_state = awreadys;
        awaddrt      = awaddr;  ////storing address
        end
      else
        awnext_state = awstart;
    end

    awreadys:
    begin
```

```
      awready  = 1'b1;

      if(wstate == wreadys)
      awnext_state  = awidle;
      else
      awnext_state =  awreadys;
    end
   endcase
  end



 //////////////////fsm for write data channel


 reg [31:0] wdatat;
 reg [7:0] mem[128] = '{default:12};
 reg [31:0] retaddr;
 reg [31:0] nextaddr;
 reg first; /// check operation executed first time



/////////////////////////function to compute next address during FIXED burst type
  function bit[31:0] data_wr_fixed (input [3:0] wstrb, input [31:0] awaddrt);
  unique case (wstrb)
    4'b0001: begin
      mem[awaddrt] = wdatat[7:0];
    end

    4'b0010: begin
      mem[awaddrt] = wdatat[15:8];
    end

    4'b0011: begin
      mem[awaddrt] = wdatat[7:0];
  mem[awaddrt + 1] = wdatat[15:8];
end

 4'b0100: begin
   mem[awaddrt] = wdatat[23:16];
end

 4'b0101: begin
  mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[23:16];
end



 4'b0110: begin
  mem[awaddrt] = wdatat[15:8];
   mem[awaddrt + 1] = wdatat[23:16];
end

 4'b0111: begin
   mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[15:8];
   mem[awaddrt + 2] = wdatat[23:16];
end

 4'b1000: begin
   mem[awaddrt] = wdatat[31:24];
end

 4'b1001: begin
   mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[31:24];
end



 4'b1010: begin
   mem[awaddrt] = wdatat[15:8];
```

```
       mem[awaddrt + 1] = wdatat[31:24];
    end



     4'b1011: begin
       mem[awaddrt] = wdatat[7:0];
       mem[awaddrt + 1] = wdatat[15:8];
       mem[awaddrt + 2] = wdatat[31:24];
    end

    4'b1100: begin
       mem[awaddrt] = wdatat[23:16];
       mem[awaddrt + 1] = wdatat[31:24];
    end

    4'b1101: begin
      mem[awaddrt] = wdatat[7:0];
      mem[awaddrt + 1] = wdatat[23:16];
      mem[awaddrt + 2] = wdatat[31:24];
    end

    4'b1110: begin
      mem[awaddrt] = wdatat[15:8];
      mem[awaddrt + 1] = wdatat[23:16];
      mem[awaddrt + 2] = wdatat[31:24];
    end

    4'b1111: begin
      mem[awaddrt] = wdatat[7:0];
      mem[awaddrt + 1] = wdatat[15:8];
      mem[awaddrt + 2] = wdatat[23:16];
      mem[awaddrt + 3] = wdatat[31:24];
    end
   endcase
  return awaddrt;
endfunction
```

```
//////////////////////////function to compute next address during INCR burst type

    function bit[31:0] data_wr_incr (input [3:0] wstrb, input [31:0] awaddrt);

  bit [31:0] addr;

   unique case (wstrb)
     4'b0001: begin
       mem[awaddrt] = wdatat[7:0];
       addr = awaddrt + 1;
     end

     4'b0010: begin
       mem[awaddrt] = wdatat[15:8];
       addr = awaddrt + 1;
     end

     4'b0011: begin
       mem[awaddrt] = wdatat[7:0];
       mem[awaddrt + 1] = wdatat[15:8];
       addr = awaddrt + 2;
     end

      4'b0100: begin
        mem[awaddrt] = wdatat[23:16];
        addr = awaddrt + 1;
     end

      4'b0101: begin
       mem[awaddrt] = wdatat[7:0];
        mem[awaddrt + 1] = wdatat[23:16];
        addr = awaddrt + 2;
     end
```

```
 4'b0110: begin
   mem[awaddrt] = wdatat[15:8];
   mem[awaddrt + 1] = wdatat[23:16];
   addr = awaddrt + 2;
end

 4'b0111: begin
   mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[15:8];
   mem[awaddrt + 2] = wdatat[23:16];
   addr = awaddrt + 3;
end

 4'b1000: begin
   mem[awaddrt] = wdatat[31:24];
   addr = awaddrt + 1;
end

 4'b1001: begin
   mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[31:24];
   addr = awaddrt + 2;
end


 4'b1010: begin
   mem[awaddrt] = wdatat[15:8];
   mem[awaddrt + 1] = wdatat[31:24];
   addr = awaddrt + 2;
end


 4'b1011: begin
   mem[awaddrt] = wdatat[7:0];
   mem[awaddrt + 1] = wdatat[15:8];
```

```
       mem[awaddrt + 2] = wdatat[31:24];
       addr = awaddrt + 3;
    end

    4'b1100: begin
       mem[awaddrt] = wdatat[23:16];
       mem[awaddrt + 1] = wdatat[31:24];
       addr = awaddrt + 2;
    end

    4'b1101: begin
      mem[awaddrt] = wdatat[7:0];
      mem[awaddrt + 1] = wdatat[23:16];
      mem[awaddrt + 2] = wdatat[31:24];
      addr = awaddrt + 3;
    end

    4'b1110: begin
      mem[awaddrt] = wdatat[15:8];
      mem[awaddrt + 1] = wdatat[23:16];
      mem[awaddrt + 2] = wdatat[31:24];
      addr = awaddrt + 3;
    end

    4'b1111: begin
      mem[awaddrt] = wdatat[7:0];
      mem[awaddrt + 1] = wdatat[15:8];
      mem[awaddrt + 2] = wdatat[23:16];
      mem[awaddrt + 3] = wdatat[31:24];
      addr = awaddrt + 4;
    end
   endcase
  return addr;
endfunction

/////////////////////Function to compute Wrapping boundary
```

```
function bit [7:0] wrap_boundary (input bit [3:0] awlen,input bit[2:0] awsize);
 bit [7:0] boundary;

unique case(awlen)
 4'b0001:
 begin
        unique case(awsize)
                3'b000: begin
                boundary = 2 * 1;
               end
                3'b001: begin
                boundary = 2 * 2;

                end
                3'b010: begin
                boundary = 2 * 4;

                end
         endcase
   end
 4'b0011:
 begin
        unique case(awsize)
                3'b000: begin
                boundary = 4 * 1;
               end
                3'b001: begin
                boundary = 4 * 2;

                end
                3'b010: begin
                boundary = 4 * 4;

                end
         endcase
      end

   4'b0111:
     begin
           unique case(awsize)
                3'b000: begin
                boundary = 8 * 1;
               end
                3'b001: begin
                boundary = 8 * 2;

                end
                3'b010: begin
                boundary = 8 * 4;

                end
         endcase
      end



     4'b1111:
     begin
           unique case(awsize)
                3'b000: begin
                boundary = 16 * 1;
               end
                3'b001: begin
                boundary = 16 * 2;

                end
                3'b010: begin
                boundary = 16 * 4;

                end
         endcase
         end

     endcase


     return boundary;
   endfunction
   ////////////////////////////////////////////////////////////

     function bit[31:0] data_wr_wrap (input [3:0] wstrb, input [31:0] awaddrt, input [7:0]
wboundary);

      bit [31:0] addr1, addr2, addr3, addr4;
      bit [31:0] nextaddr, nextaddr2;

    unique case (wstrb)

    ////////////////////////////////////////////////
      4'b0001: begin
        mem[awaddrt] = wdatat[7:0];

        if((awaddrt + 1) % wboundary == 0)
          addr1 = (awaddrt + 1) - wboundary;
        else
          addr1 = awaddrt + 1;

      return addr1;
      end

      ////////////////////////////////////////////////


      4'b0010: begin
        mem[awaddrt] = wdatat[15:8];

       if((awaddrt + 1) % wboundary == 0)
          addr1 = (awaddrt + 1) - wboundary;
```

```
  else
     addr1 = awaddrt + 1;

return addr1;
end

//////////////////////////////////////////////////

4'b0011: begin
  mem[awaddrt] = wdatat[7:0];

 if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
  else
     addr1 = awaddrt + 1;

 mem[addr1] = wdatat[15:8];

 if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  return addr2;

 end

//////////////////////////////////////////////

 4'b0100: begin
   mem[awaddrt] = wdatat[23:16];

  if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
  else
     addr1 = awaddrt + 1;
 return addr1;
end

//////////////////////////////////////////////

 4'b0101: begin
  mem[awaddrt] = wdatat[7:0];

    if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;


  mem[addr1] = wdatat[23:16];



    if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  return addr2;

end

///////////////////////////////////////////////////

 4'b0110: begin
  mem[awaddrt] = wdatat[15:8];

    if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;
    mem[addr1] = wdatat[23:16];

    if((addr1 + 1) % wboundary == 0)
      addr2 = (addr1 + 1) - wboundary;
   else
      addr2 = addr1 + 1;

   return addr2;

  end
///////////////////////////////////////////////////////////////////////


   4'b0111: begin
    mem[awaddrt] = wdatat[7:0];
     if((awaddrt + 1) % wboundary == 0)
      addr1 = (awaddrt + 1) - wboundary;
     else
      addr1 = awaddrt + 1;

    mem[addr1] = wdatat[15:8];

   if((addr1 + 1) % wboundary == 0)
      addr2 = (addr1 + 1) - wboundary;
   else
      addr2 = addr1 + 1;

    mem[addr2] = wdatat[23:16];

   if((addr2 + 1) % wboundary == 0)
      addr3 = (addr2 + 1) - wboundary;
   else
      addr3 = addr2 + 1;

     return addr3;
 end
 4'b1000: begin
   mem[awaddrt] = wdatat[31:24];

   if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

     return addr1;
end

 4'b1001: begin
   mem[awaddrt] = wdatat[7:0];

   if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;


   mem[addr1] = wdatat[31:24];

   if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  return addr2;
end


 4'b1010: begin
   mem[awaddrt] = wdatat[15:8];

   if((awaddrt + 1) % wboundary == 0)
```

```
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

   mem[addr1] = wdatat[31:24];

  if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  return addr2;
end



 4'b1011: begin
   mem[awaddrt] = wdatat[7:0];

    if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;



   mem[addr1] = wdatat[15:8];

   if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

   mem[addr2] = wdatat[31:24];

  if((addr2 + 1) % wboundary == 0)
     addr3 = (addr2 + 1) - wboundary;
  else
```

```
     addr3 = addr2 + 1;

 return addr3;

end

4'b1100: begin
   mem[awaddrt] = wdatat[23:16];

     if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

   mem[addr1] = wdatat[31:24];

   if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

     return addr2;
end

4'b1101: begin
  mem[awaddrt] = wdatat[7:0];

     if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

  mem[addr1] = wdatat[23:16];

   if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  mem[addr2] = wdatat[31:24];

   if((addr2 + 1) % wboundary == 0)
     addr3 = (addr2 + 1) - wboundary;
  else
     addr3 = addr2 + 1;

 return addr3;

end

4'b1110: begin
  mem[awaddrt] = wdatat[15:8];

   if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

  mem[addr1] = wdatat[23:16];

   if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
  else
     addr2 = addr1 + 1;

  mem[addr2] = wdatat[31:24];

  if((addr2 + 1) % wboundary == 0)
     addr3 = (addr2 + 1) - wboundary;
  else
     addr3 = addr2 + 1;
     return addr3;
 end

 4'b1111: begin
   mem[awaddrt] = wdatat[7:0];

     if((awaddrt + 1) % wboundary == 0)
     addr1 = (awaddrt + 1) - wboundary;
    else
     addr1 = awaddrt + 1;

   mem[addr1] = wdatat[15:8];

    if((addr1 + 1) % wboundary == 0)
     addr2 = (addr1 + 1) - wboundary;
   else
     addr2 = addr1 + 1;

   mem[addr2] = wdatat[23:16];

    if((addr2 + 1) % wboundary == 0)
     addr3 = (addr2 + 1) - wboundary;
   else
     addr3 = addr2 + 1;



   mem[addr3] = wdatat[31:24];

  if((addr3 + 1) % wboundary == 0)
     addr4 = (addr3 + 1) - wboundary;
   else
     addr4 = addr3 + 1;

    return addr4;
 end
endcase
```

```
  endfunction










  reg [7:0] boundary;  ////storing boundary
  reg [3:0] wlen_count;

  typedef enum bit [2:0] {widle = 0, wstart = 1, wreadys = 2, wvalids = 3, waddr_dec = 4}
wstate_type;
  wstate_type wstate, wnext_state;

  always_comb
    begin
      case(wstate)

      widle: begin
        wready = 1'b0;
        wnext_state = wstart;
        first = 1'b0;
        wlen_count = 0;
      end

      wstart: begin
        if(wvalid)
          begin
            wnext_state = waddr_dec;
            wdatat      = wdata;
          end
        else
    begin
      wnext_state = wstart;
    end
end

waddr_dec: begin
       wnext_state = wreadys;


       if(first == 0) begin
         nextaddr  = awaddr;
         first = 1'b1;
         wlen_count = 0;
         end
        else if (wlen_count < (awlen + 1 ))
         begin
         nextaddr = retaddr;
         end
        else
          begin
         nextaddr  = awaddr;
          end
end



wreadys: begin

    if(wlast == 1'b1) begin
     wnext_state = widle;
     wready = 1'b0;
     wlen_count = 0;
     first = 0;
     end
    else if(wlen_count < (awlen + 1))
    begin
    wnext_state = wvalids;
    wready      = 1'b1;
   end
   else
    wnext_state = wreadys;


 case(awburst)
   2'b00:  ////Fixed Mode
   begin
   retaddr = data_wr_fixed(wstrb, awaddr);  ///fixed
   end

   2'b01:  ////Incr mode
   begin
   retaddr =  data_wr_incr(wstrb,nextaddr);
   end

   2'b10:  //// wrapping
   begin
        boundary = wrap_boundary(awlen, awsize);   /////calculate wrapping boundary
        retaddr = data_wr_wrap(wstrb, nextaddr, boundary); ///////generate next addr
     end
 endcase
end


wvalids: begin
wready      = 1'b0;
wnext_state = wstart;

if(wlen_count < (awlen + 1))
wlen_count = wlen_count + 1;
else
wlen_count = wlen_count;
```

```
     end
     endcase
end



//////////////////////fsm for write response

typedef enum bit [1:0] {bidle = 0, bdetect_last = 1, bstart = 2, bwait = 3} bstate_type;
bstate_type bstate,bnext_state;


always_comb
begin
  case(bstate)
    bidle: begin
        bid = 1'b0;
        bresp = 1'b0;
        bvalid = 1'b0;
        bnext_state = bdetect_last;
    end

    bdetect_last: begin
        if(wlast)
         bnext_state = bstart;
         else
         bnext_state = bdetect_last;
    end

    bstart: begin
     bid = awid;
     bvalid = 1'b1;
     bnext_state = bwait;
      if( (awaddr < 128 ) && (awsize <= 3'b010) )
        bresp = 2'b00;  ///okay
      else if (awsize > 3'b010)
```

```
        bresp = 2'b10; /////slverr
      else
         bresp = 2'b11;  ///no slave address
    end

  bwait: begin
     if(bready == 1'b1)
        bnext_state = bidle;
    else
        bnext_state = bwait;
  end

  endcase
end

///////////////////////fsm for read address

always_ff @(posedge clk, negedge resetn)
begin
   if(!resetn)
      begin
      arstate <= aridle;
      rstate  <= ridle;
      end
      else
      begin
      arstate <= arnext_state;
      rstate  <= rnext_state;
      end
end


typedef enum bit [1:0] {aridle = 0, arstart = 1, arreadys = 2} arstate_type;
arstate_type arstate, arnext_state;

reg [31:0] araddrt; ///register address
```

```
always_comb
begin
case(arstate)
  aridle: begin
     arready = 1'b0;
     arnext_state = arstart;
  end

  arstart: begin
     if(arvalid == 1'b1) begin
        arnext_state = arreadys;
        araddrt = araddr;
        end
      else
        arnext_state = arstart;
  end

  arreadys: begin
         arnext_state = aridle;
         arready = 1'b1;
  end
endcase
end

//////////////////////////read data in FIxed Mode

   function void read_data_fixed (input [31:0] addr, input [2:0] arsize);
               unique case(arsize)
               3'b000: begin
                rdata[7:0] = mem[addr];
               end

               3'b001: begin
                rdata[7:0]  = mem[addr];
                rdata[15:8] = mem[addr + 1];
```

```
              end

              3'b010: begin
               rdata[7:0]    = mem[addr];
               rdata[15:8]   = mem[addr + 1];
               rdata[23:16]  = mem[addr + 2];
               rdata[31:24]  = mem[addr + 3];
              end
              endcase
  endfunction
/////////////////////////end of function
/////////////////////////// read data in INCR Mode

    function bit [31:0] read_data_incr (input [31:0] addr, input [2:0] arsize);
     bit [31:0] nextaddr;

     unique case(arsize)
       3'b000: begin
         rdata[7:0] = mem[addr];
         nextaddr = addr + 1;
      end

      3'b001: begin
      rdata[7:0]  = mem[addr];
      rdata[15:8] = mem[addr + 1];
      nextaddr = addr + 2;
      end

      3'b010: begin
      rdata[7:0]    = mem[addr];
      rdata[15:8]   = mem[addr + 1];
      rdata[23:16]  = mem[addr + 2];
      rdata[31:24]  = mem[addr + 3];
      nextaddr = addr + 4;
      end
     endcase

     return nextaddr;

    endfunction

 ////////////////////////////////////////end of function
  function bit [31:0] read_data_wrap (input bit [31:0] addr, input bit [2:0] rsize, input [7:0]
rboundary);
    bit [31:0] addr1,addr2,addr3,addr4;

    unique case (rsize)
     3'b000: begin
       rdata[7:0] = mem[addr];

       if(((addr + 1) % rboundary ) == 0)
             addr1 = (addr + 1) - rboundary;
       else
             addr1 = (addr + 1);

       return addr1;
     end

     3'b001: begin
       rdata[7:0] = mem[addr];

        if(((addr + 1) % rboundary ) == 0)
             addr1 = (addr + 1) - rboundary;
       else
             addr1 = (addr + 1);

        rdata[15:8] = mem[addr1];

        if(((addr1 + 1) % rboundary ) == 0)
             addr2 = (addr1 + 1) - rboundary;
       else
```

```
          addr2 = (addr1 + 1);

   return addr2;
end

3'b010:  begin

    rdata[7:0] = mem[addr];

    if(((addr + 1) % rboundary ) == 0)
          addr1 = (addr + 1) - rboundary;
   else
          addr1 = (addr + 1);

    rdata[15:8] = mem[addr1];

    if(((addr1 + 1) % rboundary ) == 0)
          addr2 = (addr1 + 1) - rboundary;
   else
          addr2 = (addr1 + 1);

    rdata[23:16]  = mem[addr2];

   if(((addr2 + 1) % rboundary ) == 0)
          addr3 = (addr2 + 1) - rboundary;
   else
          addr3 = (addr2 + 1);

     rdata[31:24] = mem[addr3];

    if(((addr3 + 1) % rboundary ) == 0)
          addr4 = (addr3 + 1) - rboundary;
   else
          addr4 = (addr3 + 1);

  return addr4;
```

```
    end

  endcase

 endfunction

///////////////////////////////////////
 reg rdfirst;
 bit [31:0] rdnextaddr, rdretaddr;
 reg [3:0] len_count;
 reg [7:0] rdboundary;

 typedef enum bit [2:0] {ridle = 0, rstart = 1, rwait = 2, rvalids = 3, rerror = 4} rstate_type;
 rstate_type rstate, rnext_state;

 /////////////////////////////////
 always_comb
 begin
 case(rstate)
    ridle: begin

    rid = 0;
    rdfirst = 0;
    rdata = 0;
    rresp = 0;
    rlast = 0;
    rvalid = 0;
    len_count = 0;

    if(arvalid)
    rnext_state = rstart;
    else
    rnext_state = ridle;
   end

   rstart: begin
```

```
if ((araddrt < 128) && (arsize <= 3'b010) ) begin
  rid = arid;
  rvalid = 1'b1;
  rnext_state = rwait;
  rresp = 2'b00;
  unique case(arburst)

   ////////////////////fixed
    2'b00: begin
        if(rdfirst == 0) begin
         rdnextaddr  = araddr;
         rdfirst = 1'b1;
         len_count = 0;
         end
       else if (len_count != (arlen + 1))
         begin
         rdnextaddr  = araddr;
         end

        read_data_fixed(araddrt, arsize);
      end
/////////////////////end of fixed

///////////////////start of incr
2'b01: begin
        if(rdfirst == 0) begin
         rdnextaddr  = araddr;
         rdfirst = 1'b1;
         len_count = 0;
         end
       else if (len_count != (arlen + 1))
         begin
         rdnextaddr = rdretaddr;
         end

       rdretaddr = read_data_incr(rdnextaddr, arsize);
```

```
          end
  //////////////////////end of incr
  2'b10: begin
           if(rdfirst == 0)
           begin
           rdnextaddr  = araddr;
           rdfirst = 1'b1;
           len_count = 0;
           end
          else if (len_count != (arlen + 1))
           begin
           rdnextaddr = rdretaddr;
           end
      rdboundary = wrap_boundary(arlen, arsize);
      rdretaddr  = read_data_wrap(rdnextaddr, arsize, rdboundary);
      end
 endcase
   end
   else if ( (araddr >= 128) && ( arsize <= 3'b010) ) begin
     rresp = 2'b11;
     rvalid = 1'b1;
     rnext_state = rerror;
   end
   else if (arsize > 3'b010) begin
     rresp = 2'b10;
     rvalid = 1'b1;
     rnext_state = rerror;
    end

end

 rwait: begin
   rvalid = 1'b0;
   if(rready == 1'b1)
        rnext_state = rvalids;
      else
      rnext_state = rwait;
end

rvalids: begin
    len_count = len_count + 1;
    if(len_count == (arlen + 1))
    begin
     rnext_state = ridle;
     rlast       = 1'b1;
    end
    else
      begin
     rnext_state = rstart;
     rlast       = 1'b0;
      end
end

rerror : begin
    rvalid = 1'b0;

      if(len_count < (arlen))
         begin
         if(arready)
           begin
           rnext_state = rstart;
           len_count = len_count + 1;
           end
         end
      else
         begin
         rlast = 1'b1;
         rnext_state = ridle;
         len_count   = 0;
         end
     end
```

```
   default : rnext_state = ridle;
 endcase
 end
endmodule


////////////////////////////////////////////////

interface axi_if();

  ////////write address channel (aw)

  logic awvalid;  /// master is sending new address
  logic awready;  /// slave is ready to accept request
  logic [3:0] awid; ////// unique ID for each transaction
  logic [3:0] awlen; ////// burst length AXI3 : 1 to 16, AXI4 : 1 to 256
  logic [2:0] awsize; ////unique transaction size : 1,2,4,8,16 ...128 bytes
  logic [31:0] awaddr; ////write adress of transaction
  logic [1:0] awburst; ////burst type : fixed , INCR , WRAP


  //////////write data channel (w)
  logic wvalid; //// master is sending new data
  logic wready; //// slave is ready to accept new data
  logic [3:0] wid; /// unique id for transaction
  logic [31:0] wdata; //// data
  logic [3:0] wstrb; //// lane having valid data
  logic wlast; //// last transfer in write burst


  //////////write response channel (b)
  logic bready; ///master is ready to accept response
  logic bvalid; //// slave has valid response
  logic [3:0] bid; ////unique id for transaction
  logic [1:0] bresp; /// status of write transaction
  //////////////read address channel (ar)

  logic arvalid;  /// master is sending new address
  logic arready;  /// slave is ready to accept request
  logic [3:0] arid; ////// unique ID for each transaction
  logic [3:0] arlen; ////// burst length AXI3 : 1 to 16, AXI4 : 1 to 256
  logic [2:0] arsize; ////unique transaction size : 1,2,4,8,16 ...128 bytes
  logic [31:0] araddr; ////write adress of transaction
  logic [1:0] arburst; ////burst type : fixed , INCR , WRAP

  /////////// read data channel (r)

  logic rvalid; //// master is sending new data
  logic rready; //// slave is ready to accept new data
  logic [3:0] rid; /// unique id for transaction
  logic [31:0] rdata; //// data
  logic [3:0] rstrb; //// lane having valid data
  logic rlast; //// last transfer in write burst
  logic [1:0] rresp; ///status of read transfer

  ///////////////

  logic clk;
  logic resetn;

  /////////////////
  logic [31:0] next_addrwr;
  logic [31:0] next_addrrd;




endinterface
```

Appendix II

Test Bench Source Code

```
`include "uvm_macros.svh"
 import uvm_pkg::*;


typedef enum bit [2:0] {wrrdfixed = 0, wrrdincr = 1, wrrdwrap = 2, wrrderrfix = 3,
rstdut = 4 } oper_mode;

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)



  function new(string name = "transaction");
    super.new(name);
  endfunction


  int len = 0;
  rand bit [3:0] id;
  oper_mode op;
  rand bit awvalid;
  bit awready;
  bit [3:0] awid;
  rand bit [3:0] awlen;
  rand bit [2:0] awsize; //4byte =010
  rand bit [31:0] awaddr;
  rand bit [1:0] awburst;

  bit wvalid;
  bit wready;
  bit [3:0] wid;
  rand bit [31:0] wdata;
  rand bit [3:0] wstrb;
  bit wlast;

  bit bready;
  bit bvalid;
  bit [3:0] bid;
  bit [1:0] bresp;


  rand bit arvalid;  /// master is sending new address
  bit arready;  /// slave is ready to accept request
  bit [3:0] arid; ////// unique ID for each transaction
  rand bit [3:0] arlen; ////// burst length AXI3 : 1 to 16, AXI4 : 1 to 256
  bit [2:0] arsize; ////unique transaction size : 1,2,4,8,16 ...128 bytes
  rand bit [31:0] araddr; ////write adress of transaction
  rand bit [1:0] arburst; ////burst type : fixed , INCR , WRAP

  ////////// read data channel (r)

  bit rvalid; //// master is sending new data
  bit rready; //// slave is ready to accept new data
  bit [3:0] rid; /// unique id for transaction
  bit [31:0] rdata; //// data
  bit [3:0] rstrb; //// lane having valid data
  bit rlast; //// last transfer in write burst
  bit [1:0] rresp; ///status of read transfer

  //constraint size { awsize == 3'b010; arsize == 3'b010;}
  constraint txid { awid == id; wid == id; bid == id; arid == id; rid == id;  }
  constraint burst {awburst inside {0,1,2}; arburst inside {0,1,2};}
  constraint valid {awvalid != arvalid;}
  constraint length {awlen == arlen;}

endclass : transaction



////////////////////////////////////////////////////////////////////////////

class rst_dut extends uvm_sequence#(transaction);
  `uvm_object_utils(rst_dut)

  transaction tr;

  function new(string name = "rst_dut");
    super.new(name);
  endfunction


  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
         $display("------------------------------");
        `uvm_info("SEQ", "Sending RST Transaction to DRV", UVM_NONE);
        start_item(tr);
        assert(tr.randomize);
        tr.op      = rstdut;
        finish_item(tr);
      end
  endtask


endclass
```

```
//////////////////////////////////////////////////////////////////////

class valid_wrrd_fixed extends uvm_sequence#(transaction);
  `uvm_object_utils(valid_wrrd_fixed)

  transaction tr;

  function new(string name = "valid_wrrd_fixed");
    super.new(name);
  endfunction


  virtual task body();

      tr = transaction::type_id::create("tr");
      $display("------------------------------");
      `uvm_info("SEQ", "Sending Fixed mode Transaction to DRV", UVM_NONE);
      start_item(tr);
      assert(tr.randomize);
        tr.op      = wrrdfixed;
        tr.awlen   = 7;
        tr.awburst = 0;
        tr.awsize  = 2;

      finish_item(tr);
  endtask


endclass
////////////////////////////////////////////////////////////

class valid_wrrd_incr extends uvm_sequence#(transaction);
  `uvm_object_utils(valid_wrrd_incr)

  transaction tr;

  function new(string name = "valid_wrrd_incr");
    super.new(name);
  endfunction


  virtual task body();
      tr = transaction::type_id::create("tr");
      $display("------------------------------");
      `uvm_info("SEQ", "Sending INCR mode Transaction to DRV", UVM_NONE);
      start_item(tr);
      assert(tr.randomize);
        tr.op      = wrrdincr;
        tr.awlen   = 7;
        tr.awburst = 1;
```

```
          tr.awsize  = 2;

        finish_item(tr);
  endtask


endclass

/////////////////////////////////////////////////////////

class valid_wrrd_wrap extends uvm_sequence#(transaction);
  `uvm_object_utils(valid_wrrd_wrap)

  transaction tr;

  function new(string name = "valid_wrrd_wrap");
    super.new(name);
  endfunction


  virtual task body();
        tr = transaction::type_id::create("tr");
         $display("------------------------------");
        `uvm_info("SEQ", "Sending WRAP mode Transaction to DRV", UVM_NONE);
        start_item(tr);
        assert(tr.randomize);
          tr.op      = wrrdwrap;
          tr.awlen   = 7;
          tr.awburst = 2;
          tr.awsize  = 2;

        finish_item(tr);
  endtask


endclass

////////////////////////////////////////////////////////////////////////////////

class err_wrrd_fix extends uvm_sequence#(transaction);
  `uvm_object_utils(err_wrrd_fix)

  transaction tr;

  function new(string name = "err_wrrd_fix");
    super.new(name);
  endfunction


  virtual task body();
        tr = transaction::type_id::create("tr");
        $display("------------------------------");
        `uvm_info("SEQ", "Sending Error Transaction to DRV", UVM_NONE);
        start_item(tr);
        assert(tr.randomize);
          tr.op      = wrrderrfix;
          tr.awlen   = 7;
          tr.awburst = 0;
          tr.awsize  = 2;
        finish_item(tr);
  endtask


endclass









////////////////////////////////////////////////////////////
class driver extends uvm_driver #(transaction);
  `uvm_component_utils(driver)

  virtual axi_if vif;
  transaction tr;


  function new(input string path = "drv", uvm_component parent = null);
    super.new(path,parent);
  endfunction

 virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     tr = transaction::type_id::create("tr");

   if(!uvm_config_db#(virtual axi_if)::get(this,"","vif",vif))
      `uvm_error("drv","Unable to access Interface");
  endfunction



  task reset_dut();
    begin
    `uvm_info("DRV", "System Reset : Start of Simulation", UVM_MEDIUM);
    vif.resetn      <= 1'b0;  ///active high reset
    vif.awvalid     <= 1'b0;
    vif.awid        <= 1'b0;
    vif.awlen       <= 0;
    vif.awsize      <= 0;
    vif.awaddr      <= 0;
    vif.awburst     <= 0;

    vif.wvalid      <= 0;
    vif.wid        <= 0;
    vif.wdata       <= 0;
    vif.wstrb      <= 0;
    vif.wlast      <= 0;

    vif.bready      <= 0;

    vif.arvalid    <= 1'b0;
    vif.arid       <= 1'b0;
    vif.arlen      <= 0;
    vif.arsize     <= 0;
    vif.araddr      <= 0;
    vif.arburst    <= 0;

    vif.rready      <= 0;
     @(posedge vif.clk);
      end
  endtask


  /////////////////////////write read in fixed mode

  task wrrd_fixed_wr();
           `uvm_info("DRV", "Fixed Mode Write Transaction Started", UVM_NONE);
    /////////////////////////write logic
           vif.resetn      <= 1'b1;
           vif.awvalid     <= 1'b1;
           vif.awid        <= tr.id;
           vif.awlen       <= 7;
           vif.awsize      <= 2;
           vif.awaddr      <= 5;
           vif.awburst     <= 0;


           vif.wvalid      <= 1'b1;
           vif.wid        <= tr.id;
           vif.wdata       <= $urandom_range(0,10);
           vif.wstrb       <= 4'b1111;
           vif.wlast       <= 0;

           vif.arvalid     <= 1'b0;  ///turn off read
           vif.rready      <= 1'b0;
           vif.bready      <= 1'b0;
            @(posedge vif.clk);

            @(posedge vif.wready);
            @(posedge vif.clk);

     for(int i = 0; i < (vif.awlen); i++)//0 - 6 -> 7
        begin
           vif.wdata       <= $urandom_range(0,10);
           vif.wstrb       <= 4'b1111;
           @(posedge vif.wready);
```

```
        @(posedge vif.clk);
      end
      vif.awvalid     <= 1'b0;
      vif.wvalid      <= 1'b0;
      vif.wlast       <= 1'b1;
      vif.bready      <= 1'b1;
      @(negedge vif.bvalid);
      vif.wlast       <= 1'b0;
      vif.bready      <= 1'b0;
     ////////////////////////////////////// read logic
  endtask

  ///////////////////////////////////////////////////////// read transaction in fixed mode

     task  wrrd_fixed_rd();
     `uvm_info("DRV", "Fixed Mode Read Transaction Started", UVM_NONE);
     @(posedge vif.clk);

     vif.arid        <= tr.id;
     vif.arlen       <= 7;
     vif.arsize      <= 2;
     vif.araddr      <= 5;
     vif.arburst     <= 0;
     vif.arvalid     <= 1'b1;
     vif.rready      <= 1'b1;


   for(int i = 0; i < (vif.arlen + 1); i++) begin // 0 1  2 3 4 5 6 7
     @(posedge vif.arready);
     @(posedge vif.clk);
    end

   @(negedge vif.rlast);
   vif.arvalid <= 1'b0;
   vif.rready  <= 1'b0;

 endtask

//////////////////////////////////////////////////////////////////////


  task wrrd_incr_wr();
  ////////////////////////write logic
   `uvm_info("DRV", "INCR Mode Write Transaction Started", UVM_NONE);
        vif.resetn      <= 1'b1;
        vif.awvalid     <= 1'b1;
        vif.awid        <= tr.id;
        vif.awlen       <= 7;
        vif.awsize      <= 2;
        vif.awaddr      <= 5;
        vif.awburst     <= 1;
```

```
       vif.wvalid      <= 1'b1;
       vif.wid         <= tr.id;
       vif.wdata       <= $urandom_range(0,10);
       vif.wstrb       <= 4'b1111;
       vif.wlast       <= 0;

       vif.arvalid     <= 1'b0;  ///turn off read
       vif.rready      <= 1'b0;
       vif.bready      <= 1'b0;


        @(posedge vif.wready);
        @(posedge vif.clk);

for(int i = 0; i < (vif.awlen); i++)
    begin
       vif.wdata       <= $urandom_range(0,10);
       vif.wstrb       <= 4'b1111;
       @(posedge vif.wready);
       @(posedge vif.clk);
    end

    vif.wlast       <= 1'b1;
    vif.bready      <= 1'b1;
    vif.awvalid     <= 1'b0;
    vif.wvalid      <= 1'b0;
    @(negedge vif.bvalid);
    vif.bready      <= 1'b0;
     vif.wlast      <= 1'b0;

 endtask

   ////////////////////////////////////// read logic
task wrrd_incr_rd();
  `uvm_info("DRV", "INCR Mode Read Transaction Started", UVM_NONE);
  @(posedge vif.clk);



   vif.arid        <= tr.id;
   vif.arlen       <= 7;
   vif.arsize      <= 2;
   vif.araddr      <= 5;
   vif.arburst     <= 1;
   vif.arvalid     <= 1'b1;
   vif.rready      <= 1'b1;

for(int i = 0; i < (vif.arlen + 1); i++) begin // 0 1  2 3 4 5 6 7
  @(posedge vif.arready);
  @(posedge vif.clk);
 end

@(negedge vif.rlast);
```

```
    vif.arvalid <= 1'b0;
    vif.rready  <= 1'b0;
  endtask

//////////////////////////////////////////////////////////////////////////////

   task wrrd_wrap_wr();
    `uvm_info("DRV", "WRAP Mode Write Transaction Started", UVM_NONE);
   /////////////////////////write logic
          vif.resetn      <= 1'b1;
          vif.awvalid     <= 1'b1;
          vif.awid        <= tr.id;
          vif.awlen       <= 7;
          vif.awsize      <= 2;
          vif.awaddr      <= 5;
          vif.awburst     <= 2;


          vif.wvalid      <= 1'b1;
          vif.wid         <= tr.id;
          vif.wdata       <= $urandom_range(0,10);
          vif.wstrb       <= 4'b1111;
          vif.wlast       <= 0;

          vif.arvalid     <= 1'b0;  ///turn off read
          vif.rready      <= 1'b0;
          vif.bready      <= 1'b0;


           @(posedge vif.wready);
           @(posedge vif.clk);

      for(int i = 0; i < (vif.awlen); i++)
        begin
          vif.wdata       <= $urandom_range(0,10);
          vif.wstrb       <= 4'b1111;
          @(posedge vif.wready);
          @(posedge vif.clk);
        end

        vif.wlast       <= 1'b1;
        vif.bready      <= 1'b1;
        vif.awvalid     <= 1'b0;
        vif.wvalid      <= 1'b0;

        @(negedge vif.bvalid);
        vif.bready      <= 1'b0;
        vif.wlast       <= 1'b0;


       endtask
```

```
     ///////////////////////////////////// read logic
    task wrrd_wrap_rd();
   `uvm_info("DRV", "WRAP Mode Read Transaction Started", UVM_NONE);
    @(posedge vif.clk);
    vif.arvalid    <= 1'b1;
    vif.rready     <= 1'b1;


     vif.arid       <= tr.id;
     vif.arlen      <= 7;
     vif.arsize     <= 2;
     vif.araddr     <= 5;
     vif.arburst    <= 2;

  for(int i = 0; i < (vif.arlen + 1); i++) begin // 0 1  2 3 4 5 6 7
    @(posedge vif.arready);
    @(posedge vif.clk);
   end

  @(negedge vif.rlast);
  vif.arvalid <= 1'b0;
  vif.rready  <= 1'b0;
endtask

//////////////////////////////////////////////////////////////////////

   task err_wr();
   `uvm_info("DRV", "Error Write Transaction Started", UVM_NONE);

 ////////////////////////write logic
        vif.resetn      <= 1'b1;
        vif.awvalid     <= 1'b1;
        vif.awid        <= tr.id;
        vif.awlen       <= 7;
        vif.awsize      <= 2;
        vif.awaddr      <= 128;
        vif.awburst     <= 0;


        vif.wvalid      <= 1'b1;
        vif.wid         <= tr.id;
        vif.wdata       <= $urandom_range(0,10);
        vif.wstrb       <= 4'b1111;
        vif.wlast       <= 0;

        vif.arvalid     <= 1'b0;  ///turn off read
        vif.rready      <= 1'b0;
        vif.bready      <= 1'b0;


         @(posedge vif.wready);
         @(posedge vif.clk);
   for(int i = 0; i < (vif.awlen); i++)
      begin
        vif.wdata       <= $urandom_range(0,10);
        vif.wstrb       <= 4'b1111;
        @(posedge vif.wready);
        @(posedge vif.clk);
      end

      vif.wlast       <= 1'b1;
      vif.bready      <= 1'b1;
      vif.awvalid     <= 1'b0;
      vif.wvalid      <= 1'b0;

      @(negedge vif.bvalid);
      vif.bready      <= 1'b0;
      vif.wlast       <= 1'b0;
     endtask


     /////////////////////////////////////// read logic
    task err_rd();
     `uvm_info("DRV", "Error Read Transaction Started", UVM_NONE);
    @(posedge vif.clk);
    vif.arvalid    <= 1'b1;
    vif.rready     <= 1'b1;
    //vif.bready      <= 1'b1;

     vif.arid        <= tr.id;
     vif.arlen       <= 7;
     vif.arsize      <= 2;
     vif.araddr      <= 128;
     vif.arburst     <= 0;

   for(int i = 0; i < (vif.arlen + 1); i++) begin // 0 1  2 3 4 5 6 7
    @(posedge vif.arready);
    @(posedge vif.clk);
   end

   @(negedge vif.rlast);
   vif.arvalid <= 1'b0;
   vif.rready  <= 1'b0;

  endtask
////////////////////////////////////////////////////////////////////////


virtual task run_phase(uvm_phase phase);
    forever begin

      seq_item_port.get_next_item(tr);
        if(tr.op == rstdut)
         reset_dut();
```

```
          else if (tr.op == wrrdfixed)
            begin
           `uvm_info("DRV", $sformatf("Fixed Mode Write -> Read WLEN:%0d
WSIZE:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
            wrrd_fixed_wr();
            wrrd_fixed_rd();
            end
          else if (tr.op == wrrdincr)
            begin
           `uvm_info("DRV", $sformatf("INCR Mode Write -> Read WLEN:%0d
WSIZE:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
            wrrd_incr_wr();
            wrrd_incr_rd();
            end
          else if (tr.op == wrrdwrap)
            begin
           `uvm_info("DRV", $sformatf("WRAP Mode Write -> Read WLEN:%0d
WSIZE:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
            wrrd_wrap_wr();
            wrrd_wrap_rd();
            end
          else if (tr.op == wrrderrfix)
            begin
           `uvm_info("DRV", $sformatf("Error Transaction Mode WLEN:%0d
WSIZE:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
            err_wr();
            err_rd();
            end

        seq_item_port.item_done();
      end
  endtask


endclass
//////////////////////////////////////////////////////////////////////



class mon extends uvm_monitor;
`uvm_component_utils(mon)

transaction tr;
virtual axi_if vif;

  logic [31:0] arr[128];

  logic [1:0] rdresp;
  logic [1:0] wrresp;

  logic       resp;

  int err = 0;
```

```
    function new(input string inst = "mon", uvm_component parent = null);
    super.new(inst,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tr = transaction::type_id::create("tr");
      if(!uvm_config_db#(virtual
axi_if)::get(this,"","vif",vif))//uvm_test_top.env.agent.drv.aif
        `uvm_error("MON","Unable to access Interface");
    endfunction


  ///////////////////////////////////////////////////////////////////////


  task compare();
    if(err == 0 && rdresp == 0 && wrresp == 0 )
        begin
          `uvm_info("MON", $sformatf("Test Passed err :%0d wrresp :%0d rdresp :%0d ",
err, rdresp, wrresp), UVM_MEDIUM);
          err = 0;
        end
    else
       begin
         `uvm_info("MON", $sformatf("Test Failed err :%0d wrresp :%0d rdresp :%0d ",
err, rdresp, wrresp), UVM_MEDIUM);
         err = 0;
       end
  endtask


  //////////////////////////////////////////////////////////////////////
    virtual task run_phase(uvm_phase phase);
    forever begin

      @(posedge vif.clk);
      if(!vif.resetn)
       begin
         `uvm_info("MON", "System Reset Detected", UVM_MEDIUM);
       end

      else if(vif.resetn && vif.awaddr < 128)
       begin

         wait(vif.awvalid == 1'b1);

         for(int i =0; i < (vif.awlen + 1); i++) begin
         @(posedge vif.wready);
         arr[vif.next_addrwr] = vif.wdata;
         end

        // @(negedge vif.wlast);
         @(posedge vif.bvalid);
         wrresp = vif.bresp;///0
  ///////////////////////////////////////////////////
         wait(vif.arvalid == 1'b1);

         for(int i =0; i < (vif.arlen + 1); i++) begin
           @(posedge vif.rvalid);
           if(vif.rdata != arr[vif.next_addrrd])
              begin
              err++;
              end
         end

         @(posedge vif.rlast);
         rdresp = vif.rresp;

         compare();
          $display("------------------------------");
         end

         else if (vif.resetn && vif.awaddr >= 128)
         begin
         wait(vif.awvalid == 1'b1);

         for(int i =0; i < (vif.awlen + 1); i++) begin
         @(negedge vif.wready);
         end

         @(posedge vif.bvalid);
         wrresp = vif.bresp;

         wait(vif.arvalid == 1'b1);

         for(int i =0; i < (vif.arlen + 1); i++) begin
           @(posedge vif.arready);
           if(vif.rresp != 2'b00)
              begin
              err++;
              end
         end

         @(posedge vif.rlast);
          rdresp = vif.rresp;

             compare();
          $display("------------------------------");
        end

  end
endtask

endclass
```

```
//////////////////////////////////////////////////////////
class agent extends uvm_agent;
`uvm_component_utils(agent)



function new(input string inst = "agent", uvm_component parent = null);
super.new(inst,parent);
endfunction

 driver d;
 uvm_sequencer#(transaction) seqr;
 mon m;


virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
   m = mon::type_id::create("m",this);
   d = driver::type_id::create("d",this);
   seqr = uvm_sequencer#(transaction)::type_id::create("seqr", this);


endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
    d.seq_item_port.connect(seqr.seq_item_export);
endfunction

endclass

//////////////////////////////////////////////////////////////////////////////

class env extends uvm_env;
`uvm_component_utils(env)

function new(input string inst = "env", uvm_component c);
super.new(inst,c);
endfunction

agent a;


virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  a = agent::type_id::create("a",this);

endfunction
```

```
endclass

////////////////////////////////////////////////
class test extends uvm_test;
`uvm_component_utils(test)

function new(input string inst = "test", uvm_component c);
super.new(inst,c);
endfunction


env e;
valid_wrrd_fixed vwrrdfx;
valid_wrrd_incr  vwrrdincr;
valid_wrrd_wrap  vwrrdwrap;
err_wrrd_fix     errwrrdfix;
rst_dut rdut;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
   e       = env::type_id::create("env",this);
  vwrrdfx = valid_wrrd_fixed::type_id::create("vwrrdfx");
  vwrrdincr = valid_wrrd_incr::type_id::create("vwrrdincr");
  vwrrdwrap = valid_wrrd_wrap::type_id::create("vwrrdwrap");
  errwrrdfix = err_wrrd_fix::type_id::create("errwrrdfix");
  rdut       = rst_dut::type_id::create("rdut");
endfunction

virtual task run_phase(uvm_phase phase);
phase.raise_objection(this);
//rdut.start(e.a.seqr);
//#20;
//vwrrdfx.start(e.a.seqr);
//#20;
vwrrdincr.start(e.a.seqr);
#20;
//vwrrdwrap.start(e.a.seqr);
//#20;
//errwrrdfix.start(e.a.seqr);
//#20;

phase.drop_objection(this);
endtask
endclass

//////////////////////////////////////////////////////////////////

module tb;

 axi_if vif();
 axi_slave  dut  (vif.clk,  vif.resetn,  vif.awvalid,  vif.awready,   vif.awid,  vif.awlen,
vif.awsize, vif.awaddr,  vif.awburst, vif.wvalid, vif.wready, vif.wid, vif.wdata, vif.wstrb,
vif.wlast,  vif.bready,  vif.bvalid,  vif.bid,  vif.bresp  ,  vif.arready,  vif.arid,  vif.araddr,
vif.arlen, vif.arsize, vif.arburst, vif.arvalid, vif.rid, vif.rdata, vif.rresp,vif.rlast,  vif.rvalid,
vif.rready);

  initial begin
    vif.clk <= 0;
  end

  always #5 vif.clk <= ~vif.clk;

    initial begin
    uvm_config_db#(virtual axi_if)::set(null, "*", "vif", vif);
    run_test("test");
   end
   initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end

  assign vif.next_addrwr = dut.nextaddr;
  assign vif.next_addrrd = dut.rdnextaddr;
endmodule
```