

Design of AXI Verification IP



COVERIFY

13/10/24

Coverify Systems Technology LLP • Plot #54,
Sector 38 • IN-122005 Gurugram • India

Training and Placement Officer,
Sri Ramdeobaba College of Engineering
and Management, Nagpur

Your letter of

Your reference

Our reference

Date

RCOEM-01

7th October 2024

Subject: Internship Completion Certificate

Dear Sir/Madam,

Ms Anoushka Sanjeev Tripathi, daughter of Sri Sanjeev Tripathi, and a resident of Orbital Empire, monarch 405, Ekatmata Nagar Road, Jaitala, Nagpur, has undergone a six-week internship program starting August 25, 2024 and completing on October 7, 2024.

During the internship program, Anoushka has worked on "Design of AXI Protocol Verification IP". Anoushka is a fast learner and her conduct and performance have been excellent.



Puneet Goel, CTO

Table of Contents

1. Introduction

- 1.1 Background of AXI Protocol
- 1.2 Importance of Verification in SoC Design
- 1.3 Overview of Verification Methodologies (UVM, eUVM, Vlang)
- 1.4 Objective of the Thesis

2. AXI Protocol Overview

- 2.1 Features of AXI Protocol
- 2.2 AXI Transaction Types
- 2.3 AXI Read and Write Operations
- 2.4 Burst Transactions in AXI

3. UVM-Based AXI Verification IP Design

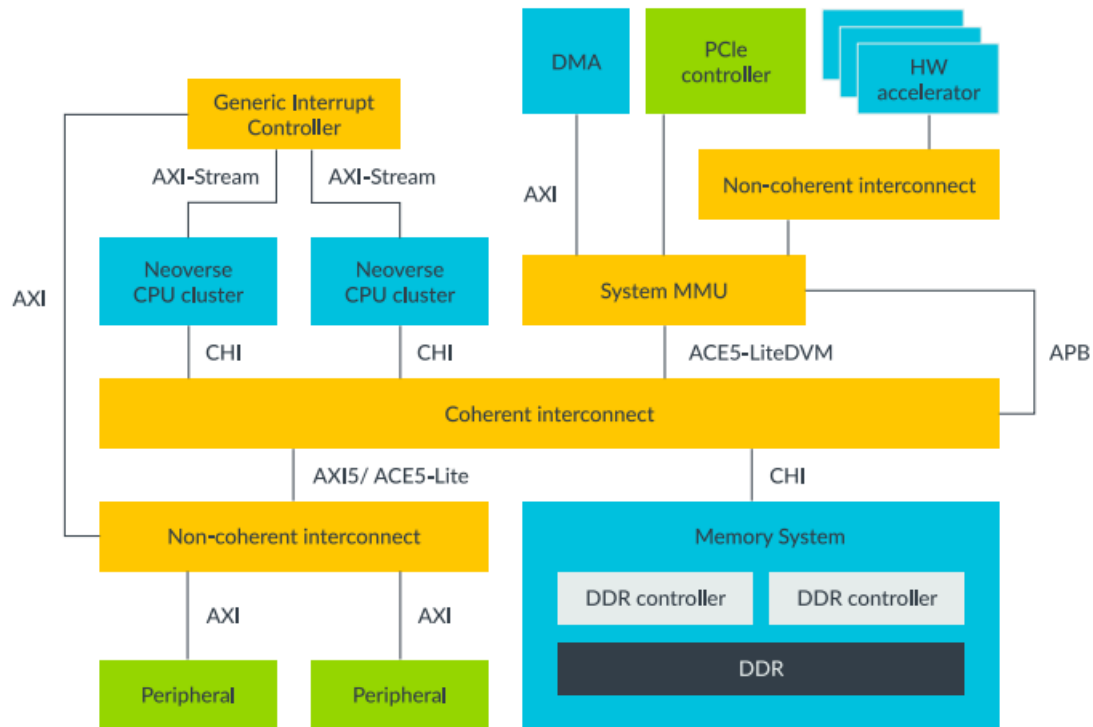
- 3.1 Overview of Universal Verification Methodology (UVM)
- 3.2 Components of AXI Verification IP in UVM
 - 3.2.1 AXI Master and Slave Models
 - 3.2.2 Driver and Monitor
 - 3.2.3 Sequence and Sequence Items
 - 3.2.4 Scoreboard and Protocol Checker
- 3.3 Implementation of AXI Verification IP in SystemVerilog UVM
- 3.4 Testbench Architecture and Structure
- 3.5 Constraints and Randomization for AXI Protocol

4 . Appendices

- A. UVM AXI Testbench Code

AXI Protocol in SoC Design

The Advanced eXtensible Interface (AXI) protocol is a part of ARM's AMBA specification, widely adopted in modern System-on-Chip (SoC) designs. AXI provides a high-performance, scalable, and flexible bus architecture that allows for the efficient communication between various components of an SoC, including processors, memories, and peripherals. AXI supports multiple independent transactions, burst transfers, and out-of-order processing, making it ideal for high-speed data transfers and multiprocessing environments.

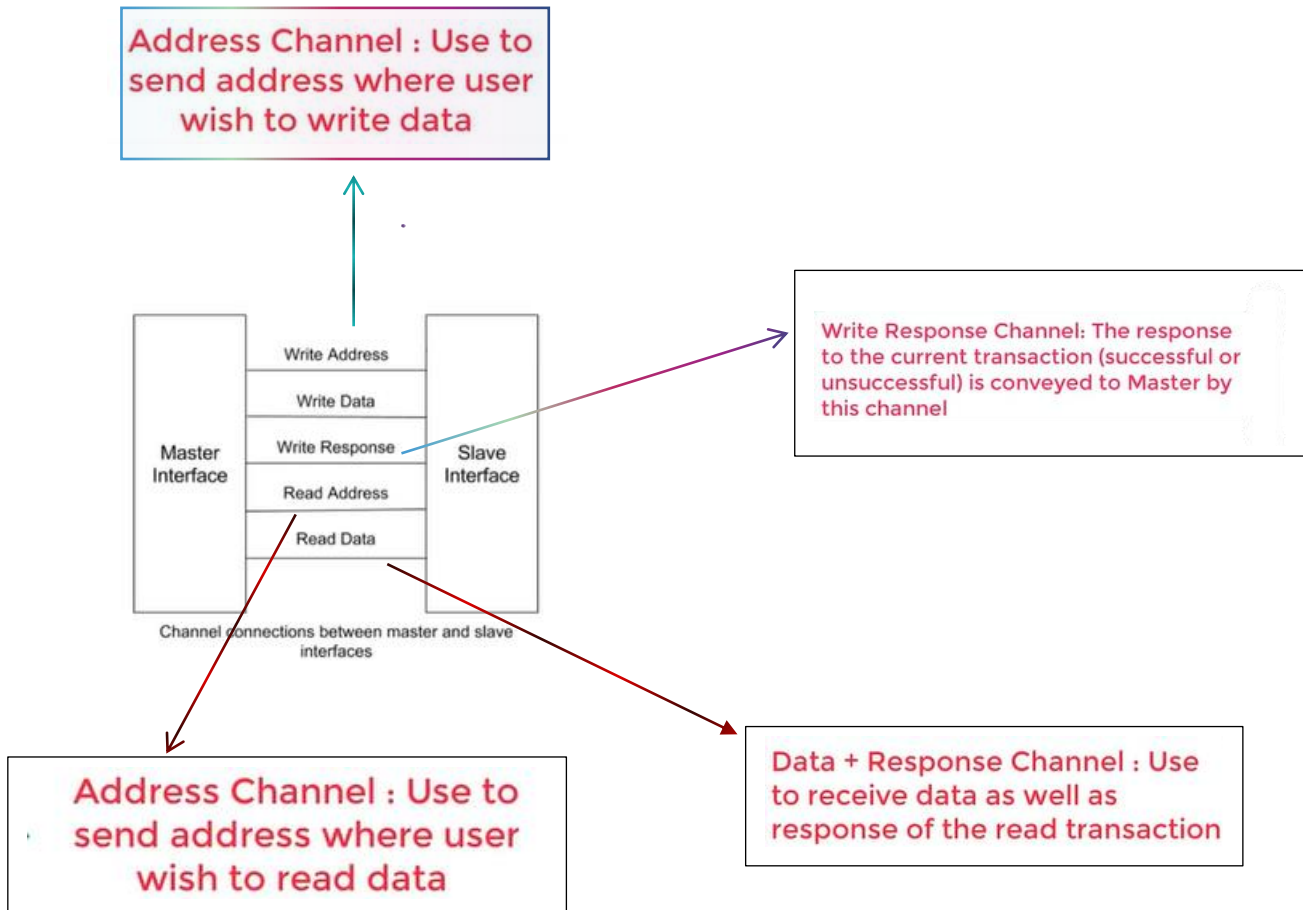


Verification Importance

Verification plays a critical role in the SoC design lifecycle. As systems grow more complex, ensuring the correct functionality of all components becomes essential to avoid costly design failures. AXI, being at the heart of many communication processes in SoCs, must be thoroughly verified to ensure that it adheres to the protocol specifications, handles corner cases, and operates efficiently under different conditions. Proper verification ensures reliability, performance, and compliance with the expected design functionality.

Objective

The objective of this project is to design an AXI Verification IP (VIP) using the Universal Verification Methodology (UVM). The AXI Verification IP will be used to create a robust and reusable testbench for verifying the behavior of AXI-compliant designs. Through constrained randomization, functional coverage, and error checking, this VIP aims to validate AXI transactions, burst modes, and protocol adherence, ensuring that the system functions as intended.



AXI interface have 5 channels.

Types of Responses(Write response channel):

- 1) Successful Transaction : OKAY / 2'b00
- 2) Specified burst size greater than supported burst size : SLV ERR / 2'b10
- 3) Out of range address : no such Slave Address / 2'b11

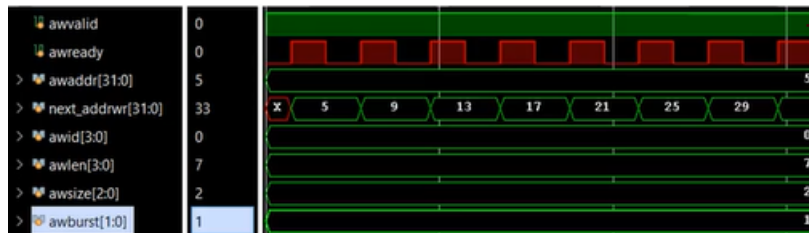
Different signals in various channels and there Usage:

```

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

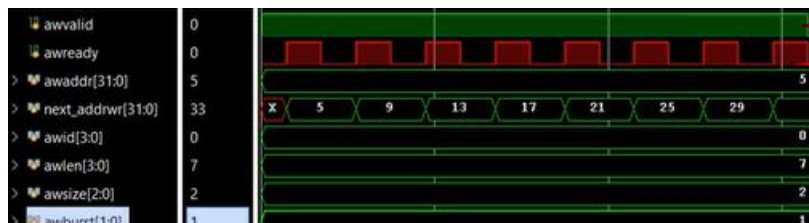
```

1. awvalid : It is use to signify a slave that we want to start a transaction. Whenever master wants to start a transaction it will convey to slave with the help of awvalid.



Till the time we do not complete all transactions, awvalid is high.

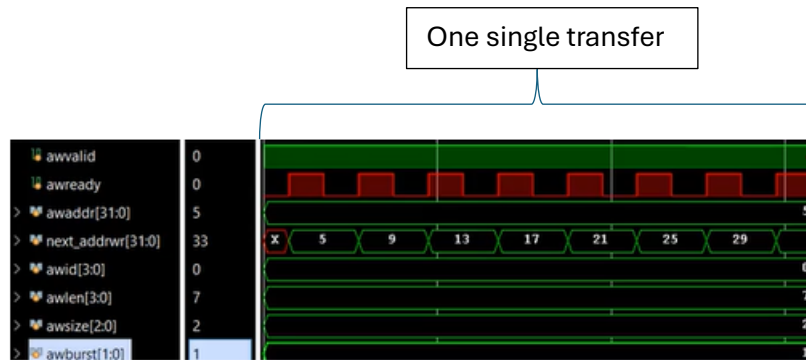
2. awready : Output from slave and input to master. Master will send awvalid after that we will wait till the time slave send us acknowledgement that it is ready to accept the transfer. Our transfer will not transfer till the time we don't get awready.
3. awid : For each transaction we assign a unique id.



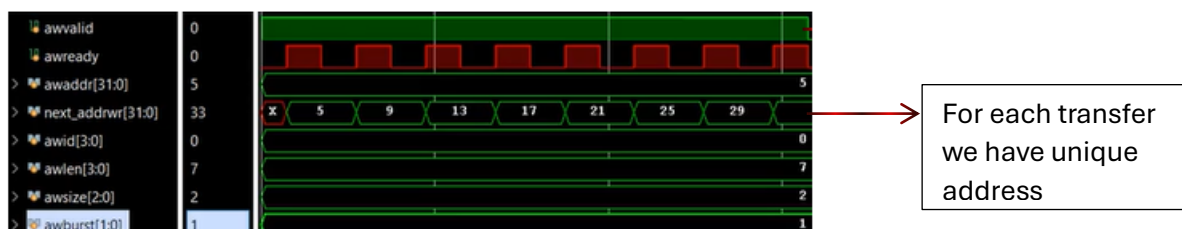
awready is high 8 times, 8 total transactions happened in one single transfer

As we did not assign any id to our transaction i.e why awid is 0.

4. awlen : Number of transaction in one single transfer.
 - burst length AXI3 : 1 to 16
 - burst length AXI4 ; 1 to 256



Burst = awlen + 1 → this is valid for read transfer as well, but for read we have arlen
 since awlen is 7 , our burst length will be 8, burst will consist of 8 transactions.



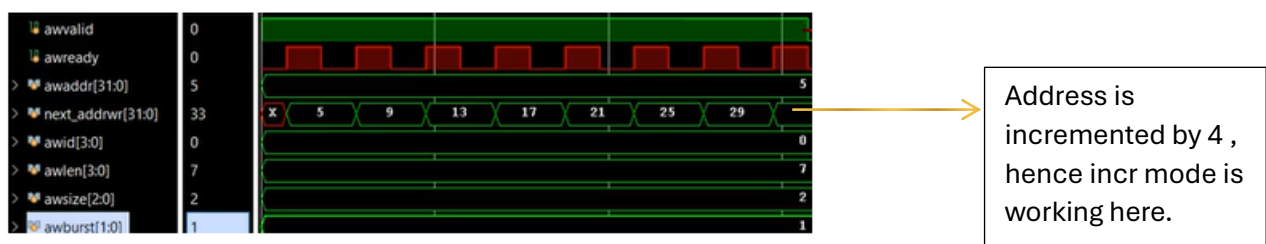
5. awsize : Maximum size of a single transaction. It helps us get control over size of data we are sending.

awdata [31 : 0] i.e 32 bits which is 4 bytes, so awsize will be 4

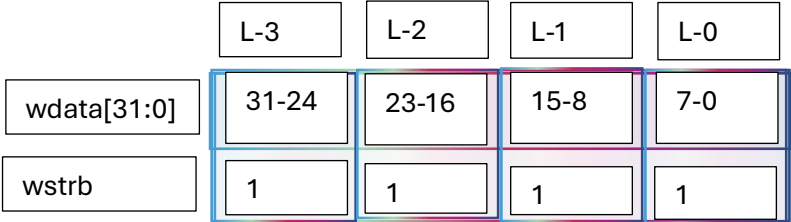
6. awburst : Helps us specify burst type. (awburst [1 : 0] = 0 , 1, 2)

Different types of burst :

- ➔ Fixed : useful when we want to read/write from a single address.
- ➔ INCR : next address is decided on basis of valid byte
 $\text{next_addr} = \text{curr addr} + \text{no. of bytes stored} = 5 + 4 = 9$
 no of bytes stored depends on how much size can be stored in memory
- ➔ WRAP :



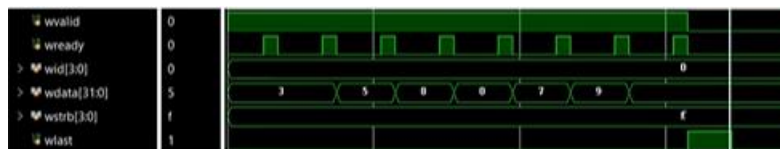
WRITE DATA CHANNEL :




```

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

```



awsize → max byte size of transaction (4 bytes).

WRITE RESPONSE CHANNEL:

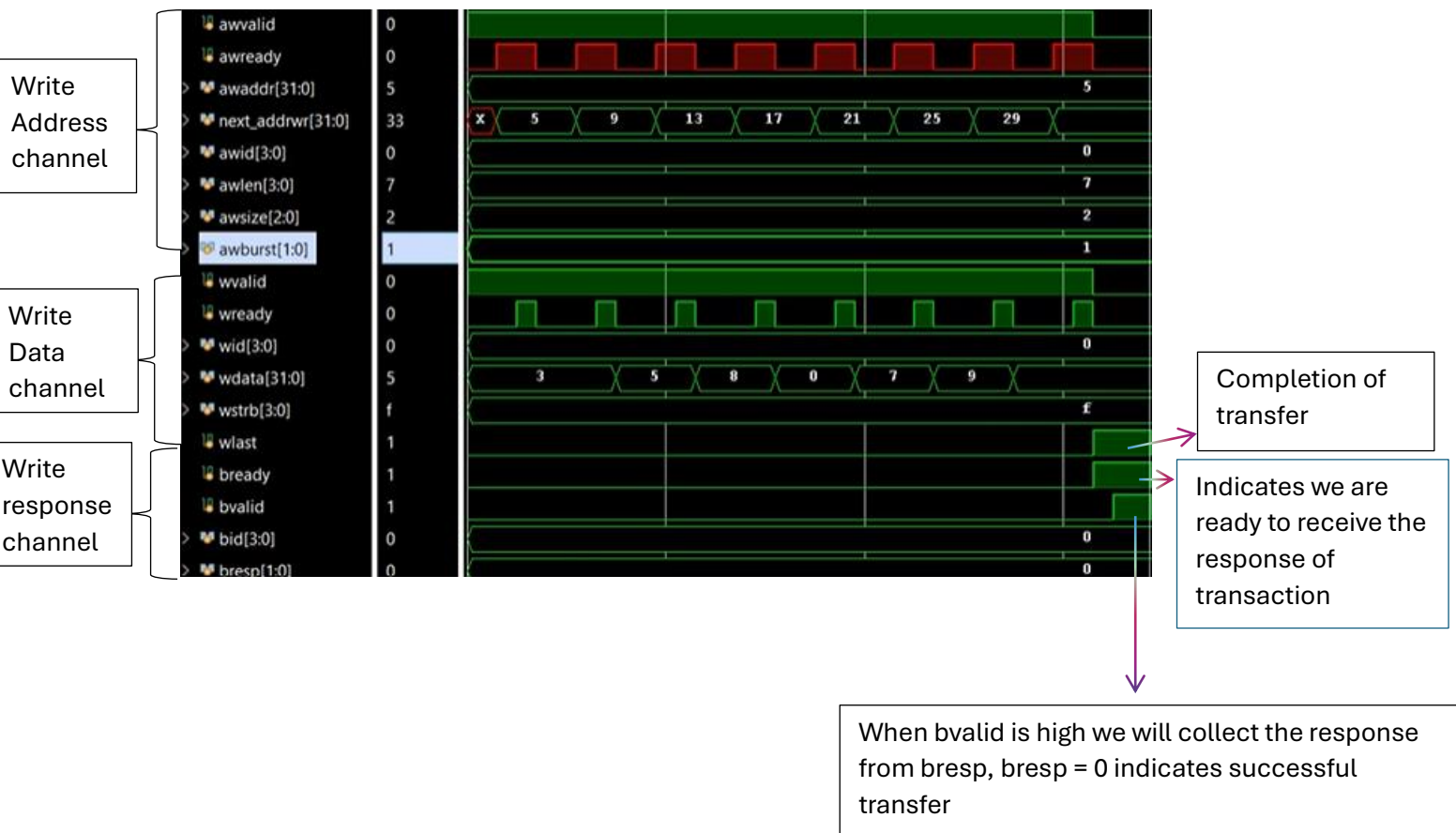
```

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

```

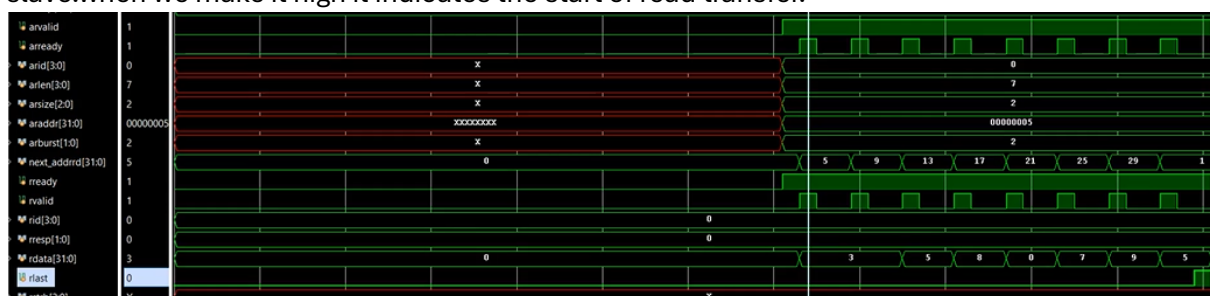


SINGLE WRITE TRANSACTION :



READ CHANNEL

1. `arvalid` : whenever master wants to read data from slave it will convey through `arvalid` to slave. when we make it high it indicates the start of read transfer.



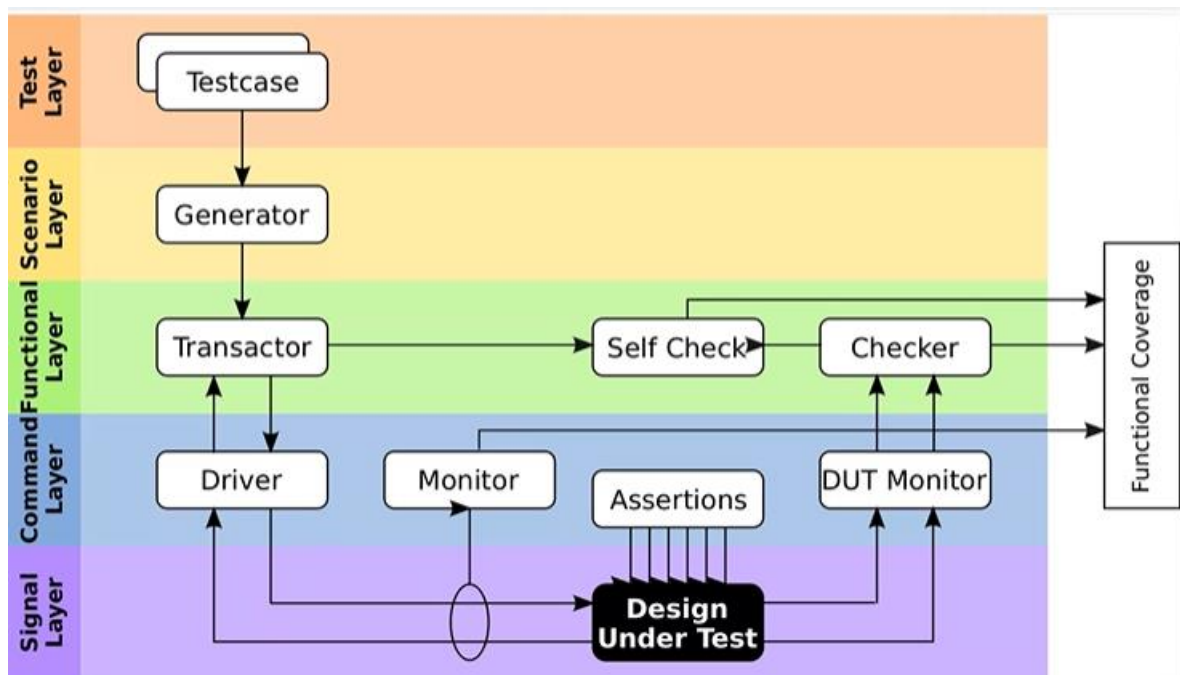
2. arlen : It is use to indicate the burst length.
burst length =arlen +1
3. arsize : It indicates maximum valid byte size in a transaction.

```

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
input  arvalid,      //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

```



3.1 Overview of Universal Verification Methodology (UVM)

Universal Verification Methodology (UVM) is a standardized methodology used to build reusable and scalable testbenches for verifying digital designs, such as AXI protocols. UVM facilitates the creation of modular components (like drivers, monitors, agents) and allows randomization, functional coverage, and detailed logging, enabling efficient design verification.

In UVM:

- **Components** like sequences, drivers, and monitors are highly reusable.
 - **Transaction-based modeling** is emphasized, where stimulus is applied using sequences and sequence items.
 - **Coverage and randomization** help achieve thorough design verification, making UVM suitable for verifying complex protocols like AXI.
-

3.2 Testbench Architecture and Structure

The testbench is structured around UVM components:

- **Test:** The test class starts the environment and sequences to simulate different transaction patterns.
- **Environment (env):** The env class instantiates the AXI agent, which contains the driver, monitor, and sequencer.
- **Agent:** The agent encapsulates the driver, monitor, and sequencer components.

In the code:

- The env class contains the UVM agent, which ties together the driver and monitor.
 - The test class controls the sequences and initiates the test.
-

3.3. Components of AXI Verification IP in UVM

This section outlines the key UVM components in the AXI Verification IP for stimulus generation and design monitoring.

3.2.1 AXI Master and Slave Models

- The AXI **master** initiates transactions like address requests, data write, or read operations.

- The **slave** responds to master's requests, handling data reception and responding with status.
- In UVM, both models can be represented using a combination of sequences (master transactions) and drivers that interface with the DUT (Device Under Test).

3.2.2 Driver and Monitor

- **Driver:** The driver class extends `uvm_driver` and is responsible for driving the AXI signals to the DUT. It receives a transaction from the sequencer and implements the write/read logic (e.g., `wrrd_fixed_wr()` and `wrrd_fixed_rd()` tasks).
- **Monitor:** The monitor passively observes the communication between the DUT and driver. It captures transactions, compares expected and actual results, and reports errors if there are mismatches (using `compare()` task).

3.2.3 Sequence and Sequence Items

- **Sequence:** The sequence class generates stimulus for the DUT. The sequence items (transactions) represent AXI operations such as write/read commands. Each sequence triggers different types of operations.
- **Sequence Item:** The transaction class extends `uvm_sequence_item` and defines the attributes of a typical AXI transaction like `awvalid`, `awaddr`, `wdata` for writes, and `arvalid`, `araddr`, `rdata` for reads.

3.2.4 Scoreboard and Protocol Checker

- **Scoreboard:** Collects the results from both the monitor and the DUT. It compares the DUT's output with expected results to ensure correctness.
- **Protocol Checker:** Ensures the AXI protocol's timing, signal integrity, and transaction order are correctly followed.

3.3 Implementation of AXI Verification IP in SystemVerilog UVM

This section describes the practical implementation of the AXI Verification IP using UVM in SystemVerilog.

- **Transactions** are modeled using the transaction class, where randomization is applied to parameters like `awvalid`, `awaddr`, and `wdata`.
- **Drivers** implement the actual protocol logic by assigning AXI signal values to the interface (`vif`).

AXI Burst Transaction Modes: Fixed, Increment, Wrap, and Error Handling

In the process of verifying AXI-based systems, different burst transaction modes are used to test data transfers. The AXI protocol defines several burst types, each with unique characteristics for addressing, data transfer, and error handling. These burst types include **Fixed**, **Increment**,

Wrap, and **Error Handling**. Each mode is critical for validating the robustness and versatility of AXI transactions. In this section, I will explain each mode and the tasks implemented to handle write and read transactions for each burst type.

1. Fixed Burst Mode

In **Fixed Burst Mode**, the address remains constant for the duration of the burst. This is useful when multiple data beats are to be written or read from a single address, commonly used in memory-mapped peripherals.

Fixed Mode Write Transaction


The task `wrrd_fixed_wr` initiates a fixed burst write operation. It configures the AXI write address and data channels to keep the address constant across multiple beats. The process involves the following steps:

- The address channel (`awvalid`, `awaddr`) is asserted with the required transaction ID and address (`awaddr`).
- The burst type is set to fixed (`awburst = 0`), ensuring the address remains constant.
- Multiple beats of data are sent over the write data channel (`wdata`), with the data being valid (`wvalid`) and strobes enabled (`wstrb = 4'b1111`).
- The process waits for the slave's `wready` signal before sending each data beat. The transaction completes when all beats are sent, and the `wlast` signal is asserted to indicate the end of the burst.

Fixed Mode Read Transaction

The task `wrrd_fixed_rd` initiates a fixed burst read operation. Like the write transaction, the address remains constant, and the task performs the following:

- The address channel (`arvalid`, `araddr`) is asserted with the read address and transaction ID (`arid`).
- The burst type is set to fixed (`arburst = 0`), ensuring the read address remains unchanged.
- Data beats are read sequentially, with the task waiting for the slave's `rready` signal to receive each beat. The process concludes once all beats are received, with the `rlast` signal indicating the end of the transaction.



```

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

```

2. Increment Burst Mode

In **Increment Burst Mode**, the address increments after each data beat. This mode is often used for accessing sequential memory locations.

Increment Mode Write Transaction

The task `wrrd_incr_wr` performs an increment burst write transaction. The key difference from fixed mode is the increment in address for each beat, configured by setting `awburst = 1`. The write address is sequentially updated after each beat, making this mode suitable for scenarios where data needs to be written to consecutive memory locations.

Increment Mode Read Transaction

Similarly, the task `wrrd_incr_rd` handles an increment burst read transaction. The read address increments after each data beat, configured by setting `arburst = 1`. This allows for sequential data reading from memory, with each beat corresponding to the next memory address.

3. Wrap Burst Mode

Wrap Burst Mode is useful for memory transactions that need to wrap around a specific boundary, ensuring data is accessed in a circular manner. The address wraps back to a starting point after reaching the boundary defined by the transaction length.

Wrap Mode Write Transaction

The task `wrrd_wrap_wr` initiates a wrap burst write transaction by configuring the `awburst` signal to 2, indicating wrap mode. During the transaction, the address wraps around a boundary, allowing the system to repeatedly access the same memory block without exceeding predefined limits.

Wrap Mode Read Transaction

The task `wrrd_wrap_rd` performs a wrap burst read transaction, with the address similarly configured to wrap around a boundary (`arburst = 2`). This mode is particularly useful in applications where continuous data streams need to be written or read within a specific address range.

4. Error Transaction Handling

Error handling is a critical aspect of verification, ensuring that the system responds correctly to erroneous transactions. Error conditions may arise when accessing out-of-bound addresses or when performing unsupported operations.

Error Write Transaction

The task `err_wr` simulates an error scenario by writing to an invalid or out-of-bound address (`awaddr = 128`). This task tests how the system responds to such conditions, verifying whether appropriate error responses are generated. It is essential in ensuring the robustness of the AXI protocol in handling erroneous writes.

Error Read Transaction

The task `err_rd` handles an error condition during a read transaction by reading from an invalid address (`araddr = 128`). Similar to the error write task, it ensures that the system can detect and respond to read errors appropriately.


```

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

```

```

  virtual axi_if vif;
  transaction tr;

```

virtual interface to the AXI interface, allowing the driver to manipulate the AXI signals.

```

  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

```

The vif (virtual interface) is obtained from the UVM configuration database (uvm_config_db), and an error

```

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 wrdd_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;

```

Assert reset, 1'b1 means deasserting the reset (system is active).

```

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

```

All the lanes have valid data

```

    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;

```

```

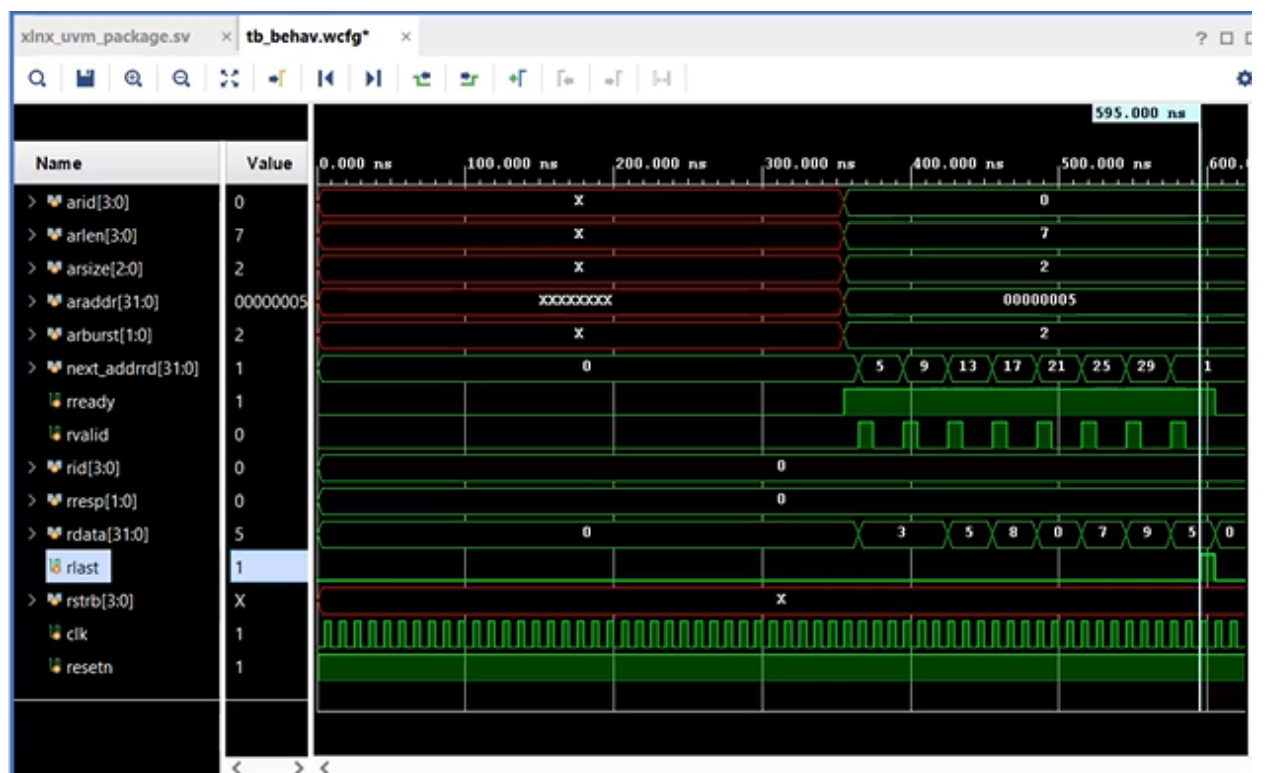
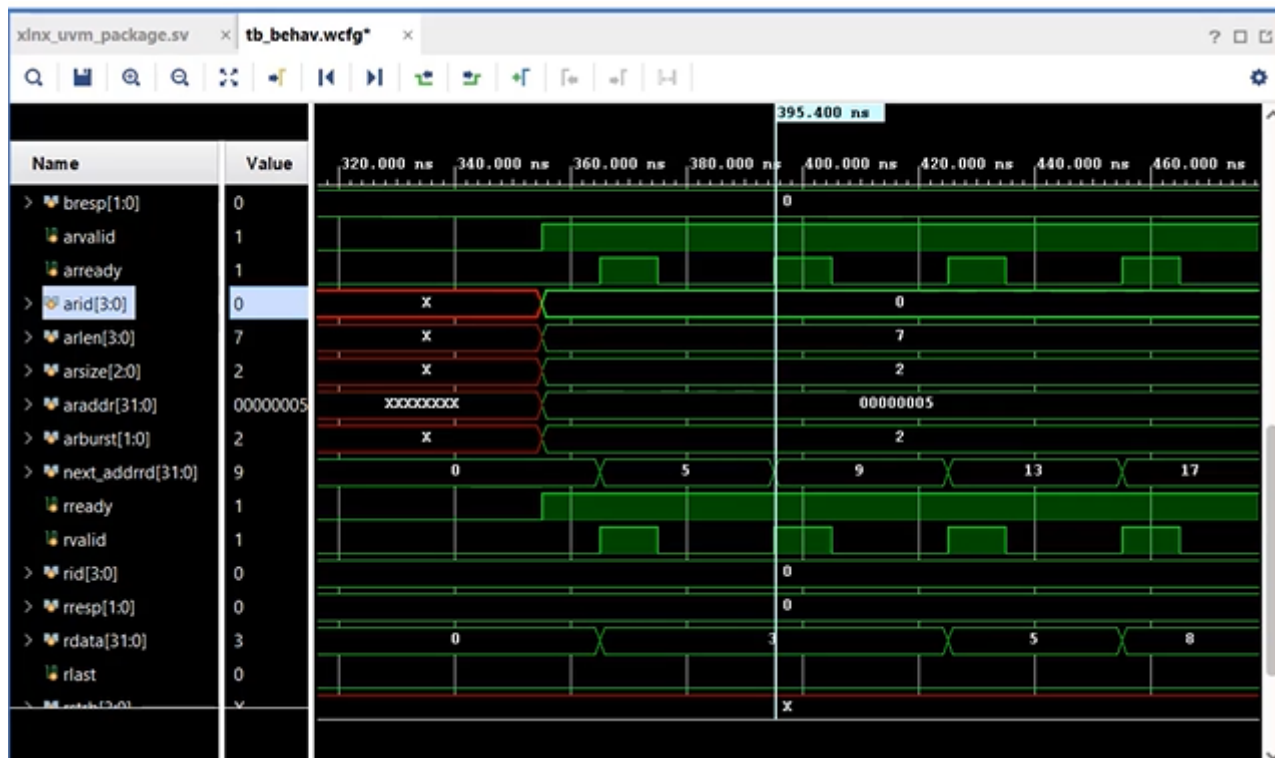
    ////////////////////// read logic

```

```

endtask

```



- **Sequences** such as valid_wrrd_fixed, valid_wrrd_incr, and valid_wrrd_wrap represent different transaction patterns for the AXI protocol. We will be using sequences to send important parameters that will help driver generate a specific sequence

```

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

```

It will be use to reset memory

We update operation to rstdut. op is a variable which is of enum type which will store the type of operation which we are verifying

Randomize data

```

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

```

Update mode to write read fixed

Burst mode type

Burst length of size 8

```

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

endclass

```

UVM Driver Code for AXI Burst Transactions

The code below describes a `run_phase` task that handles multiple AXI burst transaction modes (Fixed, Increment, Wrap, and Error) in a UVM environment. This task performs a sequence of write and read operations based on the transaction type and is part of the AXI Verification IP driver.

- The **run_phase** is a virtual task that defines what actions the UVM driver will take during the simulation's run phase. This task is continuously active during the main execution of the testbench.
- **seq_item_port.get_next_item(tr);** This function gets the next transaction item from the sequence (driven by the sequencer) and assigns it to `tr` (a transaction object). It indicates that the driver should now perform the operation specified in `tr.op`
- **if (tr.op == rstdut),** The first condition checks if the operation is to reset the DUT. If the operation is `rstdut`, the driver will call the `reset_dut()` task to reset the DUT.

Handling Different Transaction Modes:

The code then branches into different transaction types based on the value of `tr.op` (which specifies the operation mode).

Purpose of Each Mode:

- **Fixed Mode:** The address remains constant during the burst transaction, and data is written/read to/from the same address.
- **Increment Mode:** The address increases sequentially for each beat of the burst, meaning data is written/read to/from consecutive addresses.
- **Wrap Mode:** The address wraps around a pre-defined boundary, writing/reading data in a circular fashion.
- **Error Mode:** Simulates incorrect transactions (e.g., to invalid addresses), testing how the DUT responds to faulty conditions.



```
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 == wrdfixed)
      begin
        `uvm_info("DRV", $sformatf("Fixed Mode Write -> Read WLEN:%0d
WLEN:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
        wrd_fixed_wr();
        wrd_fixed_rd();
      end
    else if (tr.op == wrdincr)
      begin
        `uvm_info("DRV", $sformatf("INCR Mode Write -> Read WLEN:%0d
WLEN:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
        wrd_incr_wr();
        wrd_incr_rd();
      end
    else if (tr.op == wrdwrap)
      begin
        `uvm_info("DRV", $sformatf("WRAP Mode Write -> Read WLEN:%0d
WLEN:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
        wrd_wrap_wr();
        wrd_wrap_rd();
      end
    else if (tr.op == wrderrfix)
      begin
        `uvm_info("DRV", $sformatf("Error Transaction Mode WLEN:%0d
WLEN:%0d",tr.awlen+1,tr.awsize), UVM_MEDIUM);
        err_wr();
        err_rd();
      end

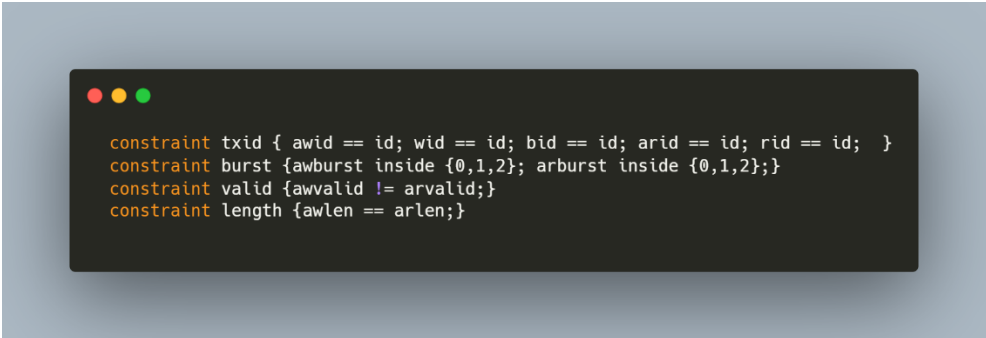
    seq_item_port.item_done();
  end
endtask

endclass
```

defines what actions the UVM driver will take during the simulation's run phase.

gets the next transaction item from the sequence

3.5 Constraints and Randomization for AXI Protocol



```
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;}
```

Randomization and constraints are critical to ensure all AXI protocol behaviors are tested comprehensively. Random values for awaddr, wdata, etc., allow for varying transaction patterns, while constraints ensure protocol-specific rules are followed.

- **Randomization:** The keyword rand is used to randomize fields like awaddr and wdata, which generate different stimulus for the DUT.
- **Constraints:** Constraints ensure valid AXI protocol operations, such as:
 - constraint txid { awid == id; wid == id; bid == id; arid == id; rid == id; } ensures that all ID fields remain consistent within a transaction.
 - constraint burst {awburst inside {0,1,2}; arburst inside {0,1,2};} restricts the burst types to valid AXI values.
 - constraint valid {awvalid !=arvalid} at a time we can either read or write, that is why at same time awvalid and arvalid should not be equal.

3.6 Monitor

```

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

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

```

Data container for storing response

Store data during write operation

Store the read and write responses

Count error in transaction

In this phase we get access of interface

```

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;
            end
    end

```

waits until the awvalid signal is asserted

stores the incoming write data (wdata) into an array (arr) at the index specified by next_addrwr.

Wait for bvalid to be asserted, bvalid marks the completion of write operation

This block is executed if the system is not in reset and the address being written to (awaddr) is less than 128 (presumably within a valid range). It waits until the awvalid signal is asserted, indicating that a write address is valid

WRITE OPERATION

monitor waits for the arvalid signal to go high, indicating that the DUT has a valid read address on the AXI bus.

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

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

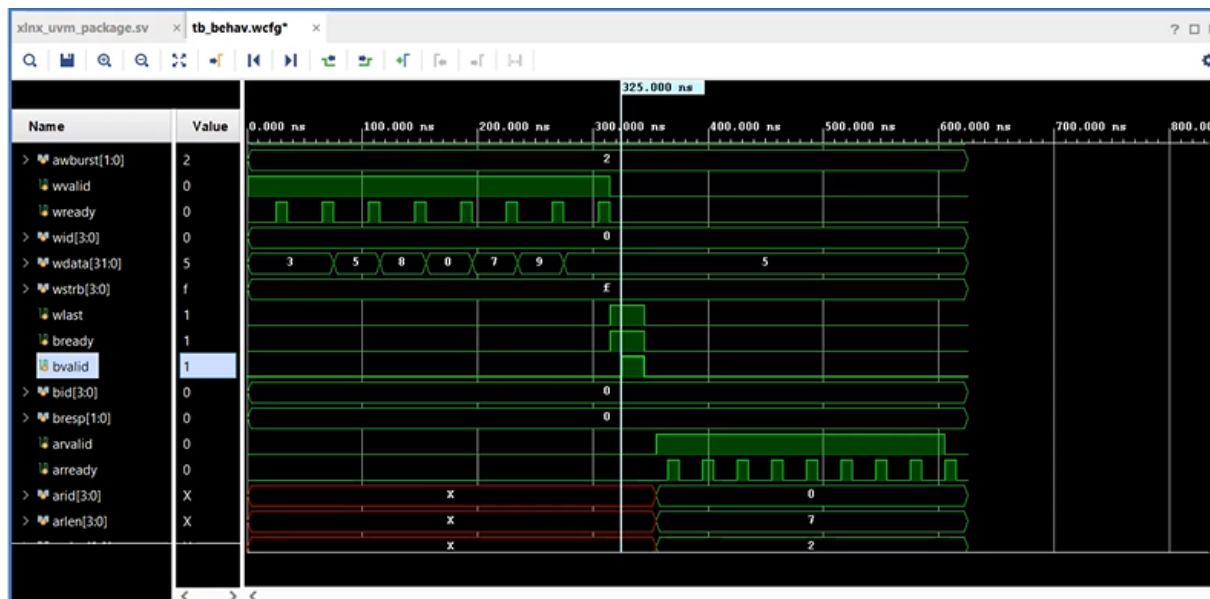
@(posedge vif.rlast);
rdresp = vif.rresp;

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

end
endtask

endclass
```

condition checks if the received data (vif.rdata) does not match the expected data from the memory array (arr[vif.next_addrd]). If there's a mismatch, it increments the error counter (err++)



Wwresp is the variable where we store the response of write transfer, we wait for bvalid to become high whatever value we have in bresp we store in wwresp for successful transfer it should be zero.

Error Increment Concept Explained

Monitoring Read Transaction (arvalid, rdata, and rresp):

1. Wait for Read Address Valid (arvalid):

- The monitor waits for the arvalid signal to go high, indicating that the DUT has a valid read address on the AXI bus.

2. Data Beat Loop:

- The loop iterates over the number of data beats specified by arlen + 1. This means it waits for the data transfers based on how many beats were specified in the read transaction.

3. Waiting for Read Valid (rvalid):

- Inside the loop, the code waits for rvalid to be asserted, indicating that the DUT has valid read data on the bus.

4. Error Detection (rdata mismatch):

- The condition checks if the received data (vif.rdata) does not match the expected data from the memory array (arr[vif.next_addrd]). If there's a mismatch, it increments the error counter (err++).
- This process ensures that any discrepancy between what was written and what was read back is counted as an error.

3.7 AGENT



```
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
```

Connect seq item port of
a driver to sequence item
export of a sequencer

3.8 ENV



```
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
```

‘3.9 TEST



```
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
```

Objects of
all
sequences

Raise objection
and one by one
call sequence

```

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_addrdd = dut.rdnxtaddr;

endmodule

```

We provide access to
interface to component

Appendices

A. UVM AXI Testbench 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 wrdd_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.awsz), 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.awsz), 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
```



```

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

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

```
endmodule
```