

Design of AXI Verification IP in a Nutshell

Overview and Implementation using UVM



by **Anoushka Tripathi**

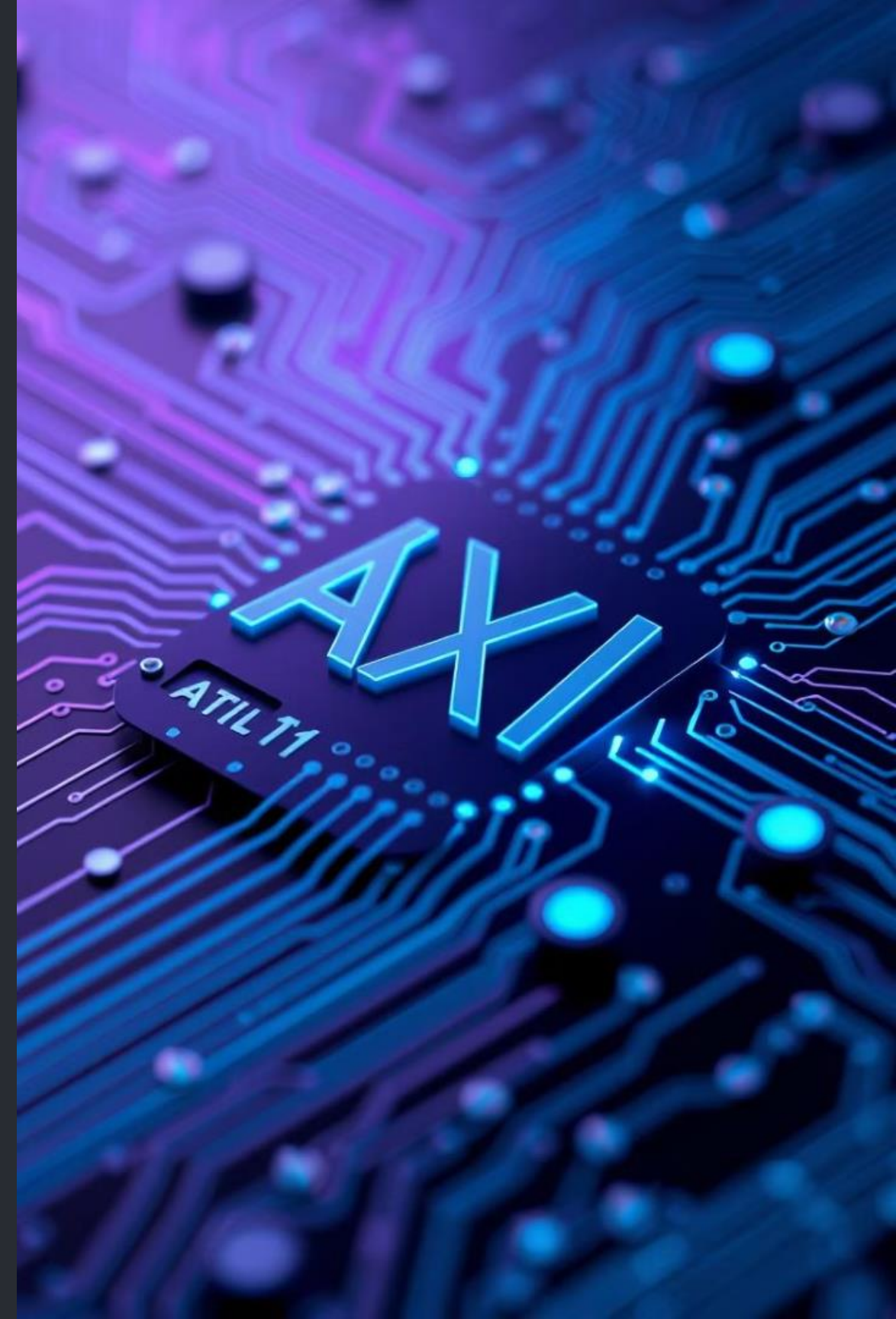


Table of Contents

Introduction

AXI Protocol overview

UVM Based AXI Verification IP Design

Components of AXI Verification IP

AXI Read/Write Operations

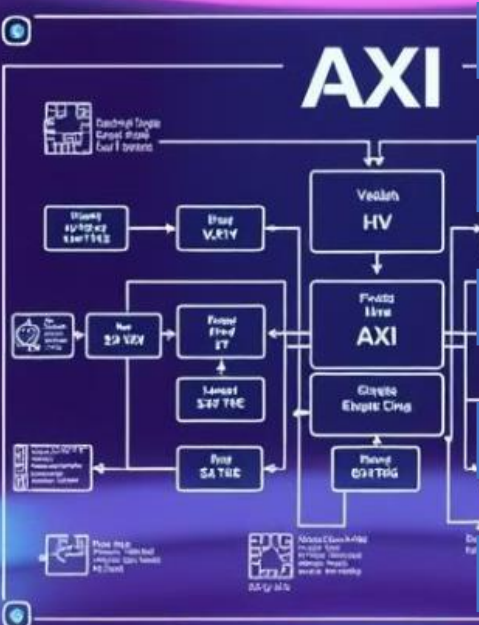
Burst transactions

AXI Testbench design

AXI Protocol Constraints and Randomization

Error Handling

Results and conclusion



Introduction

Background

The AXI protocol is a key component in modern System-on-Chip (SoC) designs, enabling high-performance data communication between processors, memory, and peripherals. Its support for burst transfers and parallel transactions makes it ideal for complex SoC architectures.

Importance

Verification ensures the correct functionality of SoCs by thoroughly testing the AXI protocol. It helps identify potential issues early, ensuring the system performs reliably under various conditions and meets design specifications.

Objective

The goal is to design an AXI Verification IP using UVM, which will validate AXI transactions, burst modes, and protocol compliance through a reusable, coverage-driven testbench.

AMBA Overview

Key AMBA specifications

AMBA CHI

AMBA Coherent Hub Interface

AMBA CHI C2C

AMBA Coherent Hub Interface Chip-to-Chip

AMBA AXI

AMBA Advanced Extensible Interface

AMBA AHB

AMBA Advanced High Performance Bus

AMBA APB

AMBA Advanced Peripheral Bus

Open standard for SoCs : Allows components in System-on-Chip (SoC) designs to communicate efficiently.

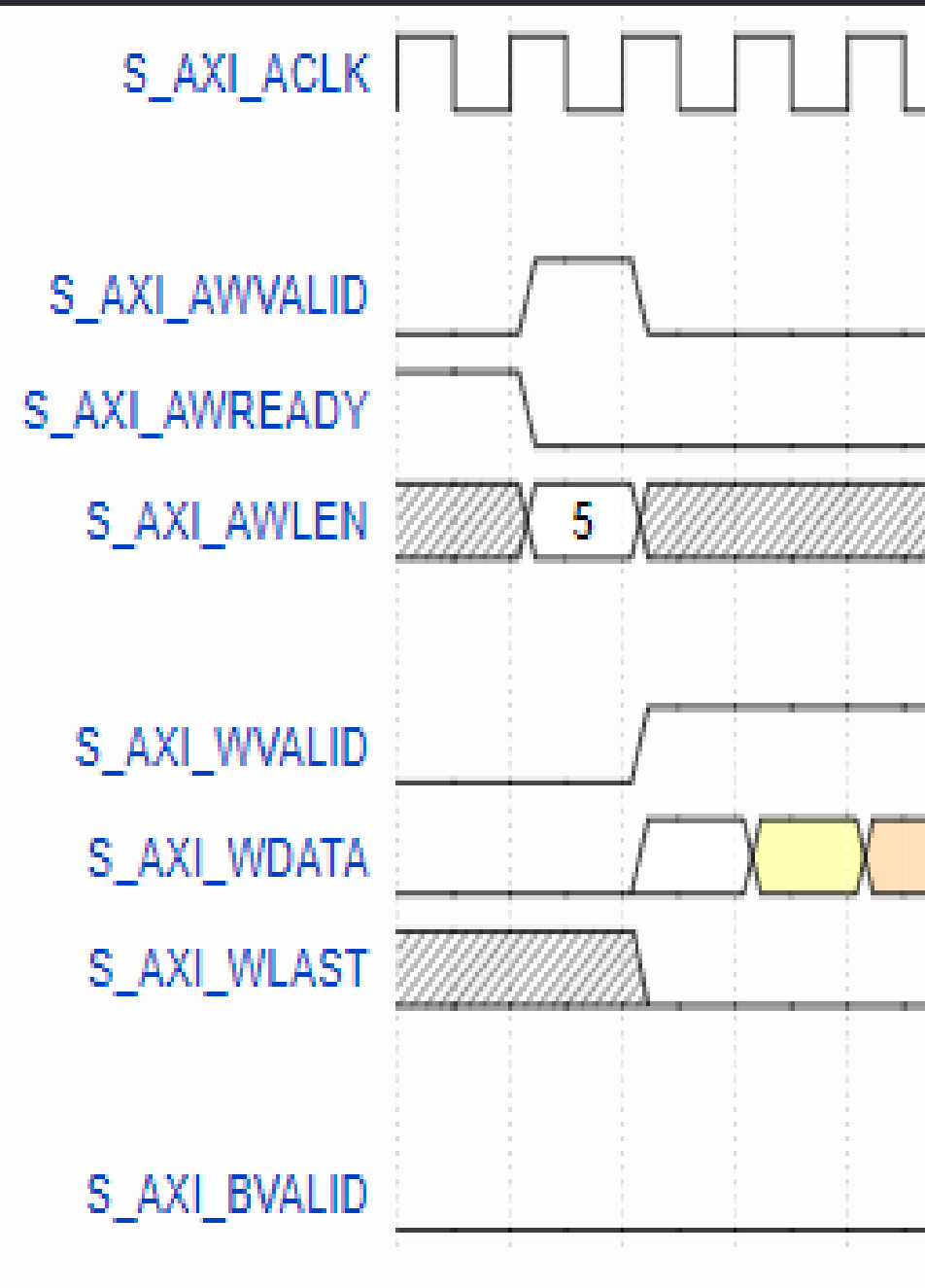
Efficient communication: AMBA enables smooth data transfers between different blocks in a chip, improving system performance.

Supports IP reuse: Designers can reuse verified intellectual property (IP) blocks, reducing development time and costs.

Compatibility and flexibility: AMBA supports various SoCs with different power, performance, and area requirements, ensuring compatibility across devices.

Widely used: AMBA is found in microcontrollers, smartphones, IoT devices, and more, due to its adaptability and efficiency.

AXI Overview



Part of AMBA protocol: AXI is a key protocol within the AMBA family, tailored for high-performance system interconnects.

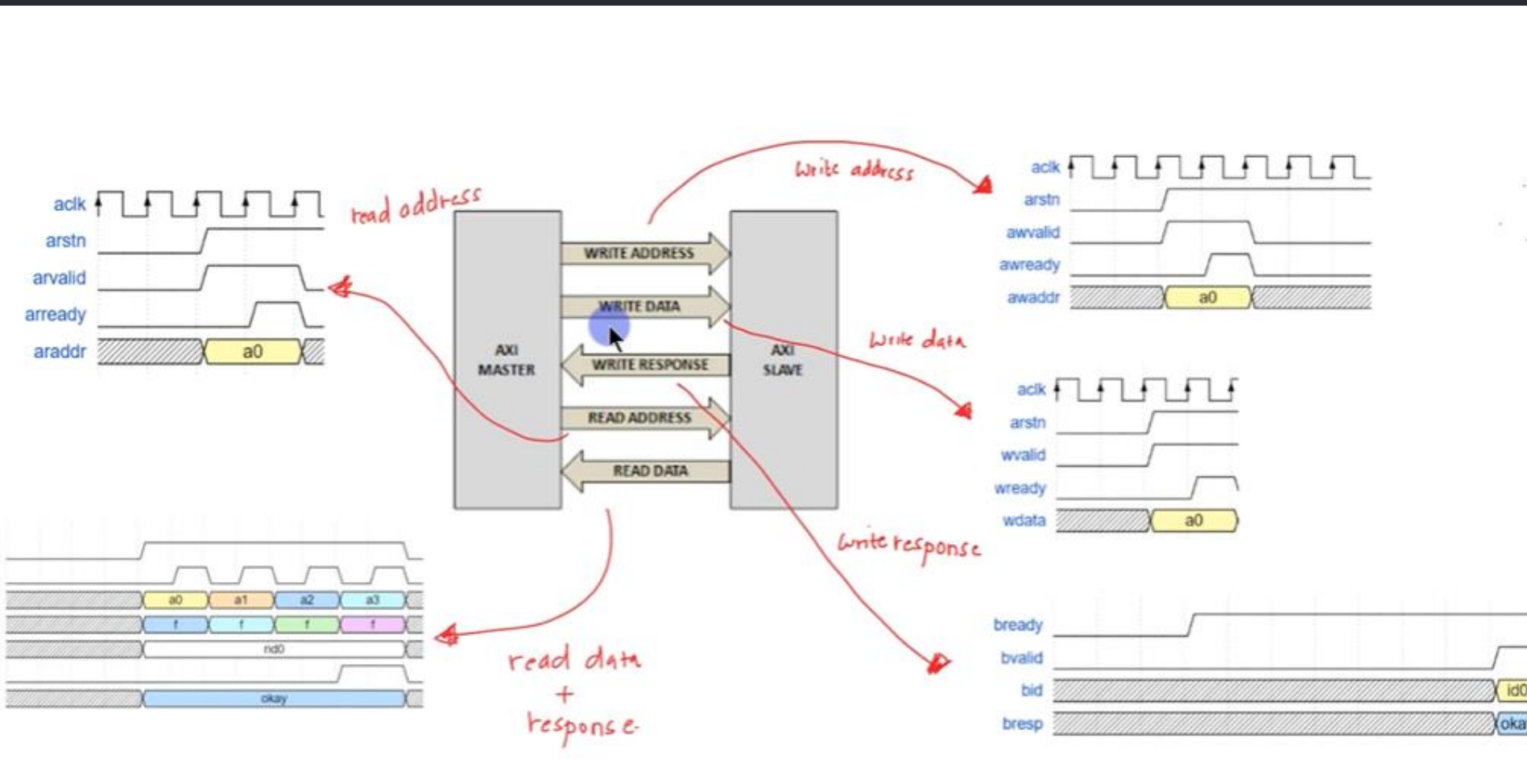
Independent read/write channels: AXI allows simultaneous data transfers, boosting efficiency by separating read and write channels.

Supports burst transactions: AXI can transfer data in bursts, which improves data flow for large transfers.

Handles multiple outstanding transactions: Multiple transactions can be processed concurrently without waiting for earlier ones to finish.

Optimizes bandwidth and latency: AXI's design minimizes delays and maximizes data throughput, critical for modern SoCs

AXI protocol – Master Slave Communication



Communication between the master and slave occurs over multiple channels:

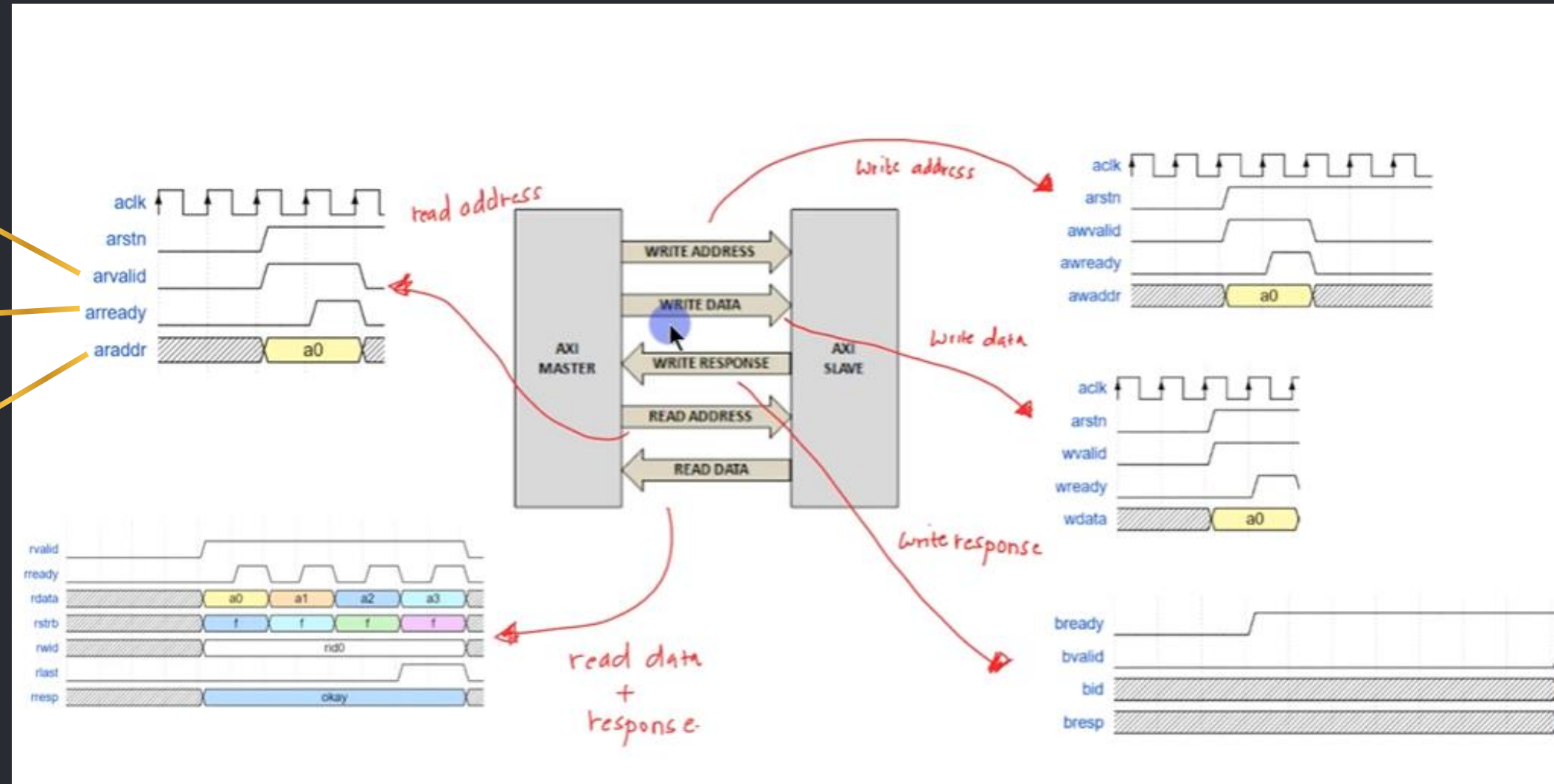
- Write Address Channel
- Write Data Channel
- Write Response Channel
- Read Address Channel
- Read Data Channel

Read Transaction Flow

arvalid to indicate master have applied valid address

arready indicates slave is ready to accept address

araddr contains the address from which data will be read.



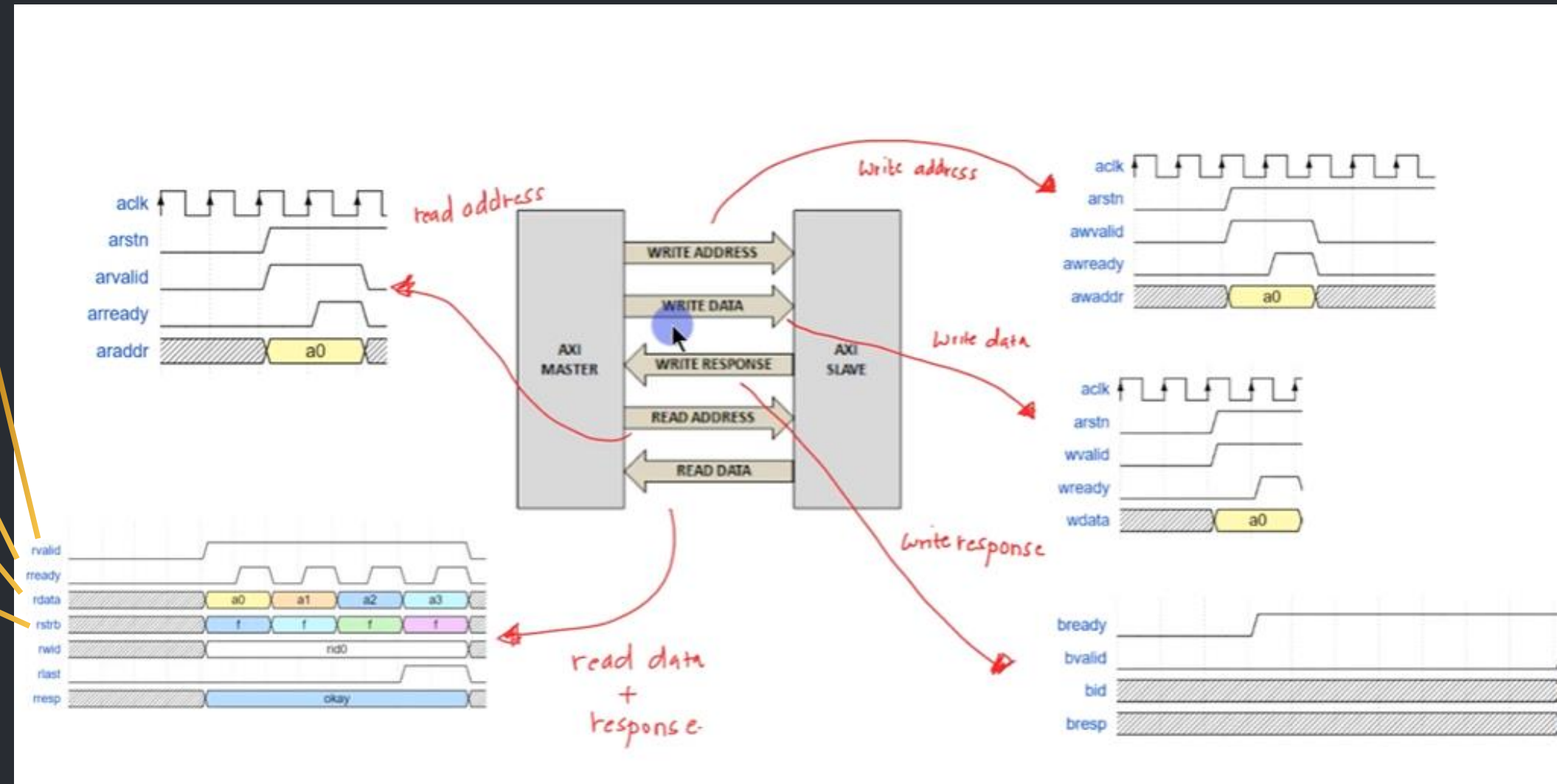
Read Transaction Flow

rvalid indicates slave is ready to accept data

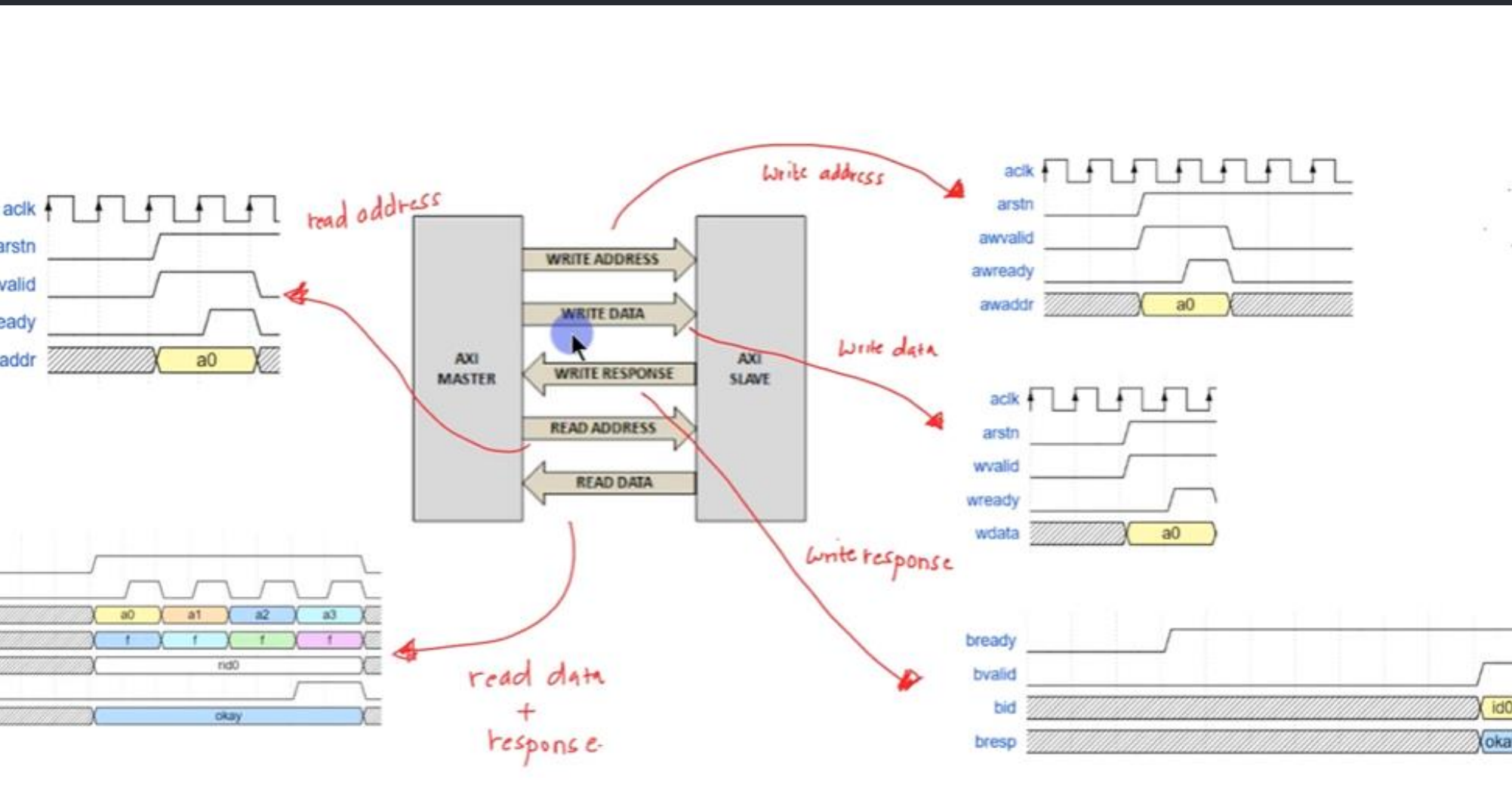
master asserts rready when it is ready to accept the data.

rdata contains the actual data read from the specified address.

Rstrb indicates lane have valid data.



Write Transaction Flow



Write Transaction Flow:

1. Write Address (awaddr):

- The AXI master sends the write address to the slave, which is indicated by the valid signal (`awvalid`) and acknowledged by the ready signal (`awready`).

2. Write Data (wdata):

- After the address is transferred, the data is sent over the write data channel. The data transfer is valid when `wvalid` is high and acknowledged by the slave with `wready`.

3. Write Response (bresp):

- Once the data is successfully written, the slave responds with a write response (`bresp`), confirming the completion of the write transaction. The response can indicate success or an error.

Importance of Burst Modes in AXI

Value	Burst type	Usage notes	Length (number of transfers)	Alignment
0x00	FIXED	Reads the same address repeatedly. Useful for FIFO s.	1-16	Fixed byte lanes only defined by start address and size.
0x01	INCR	Incrementing burst. The subordinate increments the address for each transfer in the burst from the address for the previous transfer. The incremental value depends on the size of the transfer, as defined by the AxSIZE attribute. Useful for block transfers.	AXI3: 1-16 AXI4: 1-256	Unaligned t ransfers are supported.
0x10	WRAP	Wrapping burst. Similar to an incrementing burst, except that if an upper address limit is reached, the address wraps around to a lower address. Commonly used for cache line accesses.	2, 4, 8, or 16	The start address must be aligned to the transfer size.
0x11	RESERVED	Not for use.	•	•

Universal Verification Methodology (UVM)

Overview



Modular Components

UVM utilizes reusable blocks such as drivers, monitors, and agents to streamline testbench design and improve verification efficiency.



Transaction-based

Focuses on verifying the flow of transactions between components, making it easier to validate complex data exchanges in a system.



Constrained random testing

Allows for generating randomized test scenarios within specified constraints, increasing the probability of catching edge-case bugs.

AXI Testbench Architecture

UVM Test: The test is at the top of the hierarchy that initiates the environment component construction. It is also responsible for the testbench configuration and stimulus generation process.

UVM Environment : An environment provides a well-mannered hierarchy and container for agents, scoreboards.

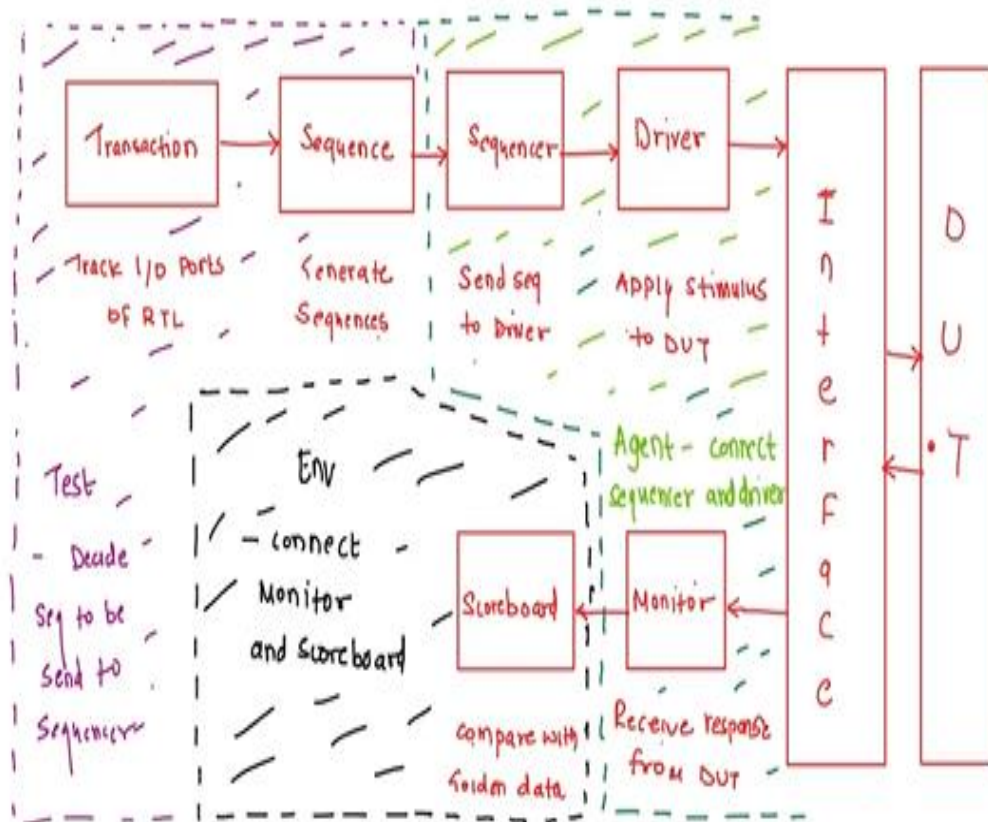
UVM Agent : An agent is a container that holds the driver, monitor, and sequencer. This is helpful to have a structured hierarchy based on the protocol or interface requirement

UVM Sequence Item : The transaction is a packet that is driven to the DUT or monitored by the monitor as a pin-level activity

UVM Driver : The driver interacts with DUT. It receives randomized transactions or sequence items and drives them to the DUT as a pin-level activity

UVM Sequencer : The sequencer is a mediator who establishes a connection between the sequence and the driver.

UVM Monitor : The monitor observes pin-level activity on the connected interface at the input and output of the design. This pin-level activity is converted into a transaction packet and sends to the scoreboard for checking purposes.



Enum for AXI Operation Modes

```
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
```

Different of Burst
Modes in AXI

Example usage

```
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
```

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

UVM Transaction class for AXI Transactions

Purpose of UVM Sequence Item

The sequence-item consist of data fields required for generating the stimulus. In order to generate the stimulus, the sequence items are randomized in sequences.

- 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

```
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
```

Sequence Class for Fixed Mode

```
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
```

a write transaction object.

Update mode to write read fixed

Burst mode type

Burst length of size 8

Sequence Class for Incrementing Mode

```
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
```

Error Transaction Sequence

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.

```
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
```


Error Transaction Sequence

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

Error Read Transaction Logic

- The `err_rd()` task initiates an error read transaction in AXI protocol simulation.
- Logs the start of the transaction using `uvm_info("DRV", "Error Read Transaction Started", UVM_NONE)`.
- The read transaction starts at the positive edge of the clock (`@posedge vif.clk`).
- **Control Signals:**
 - `vif.arvalid <= 1'b1`: Master initiates a valid read address transaction.
 - `vif.rready <= 1'b1`: Master is ready to accept data.
- Configures the read transaction's parameters (e.g., ID, address, burst type).
- After reading all data (`vif.arlen`), deasserts the control signals (`vif.arvalid <= 1'b0`, `vif.rready <= 1'b0`) to end the transaction.

UVM Driver for AXI Protocol

```
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
```

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

gets the next transaction item from the sequence

- 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.

UVM Driver for AXI Protocol

```
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
```

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

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

```
//////////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;

  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);

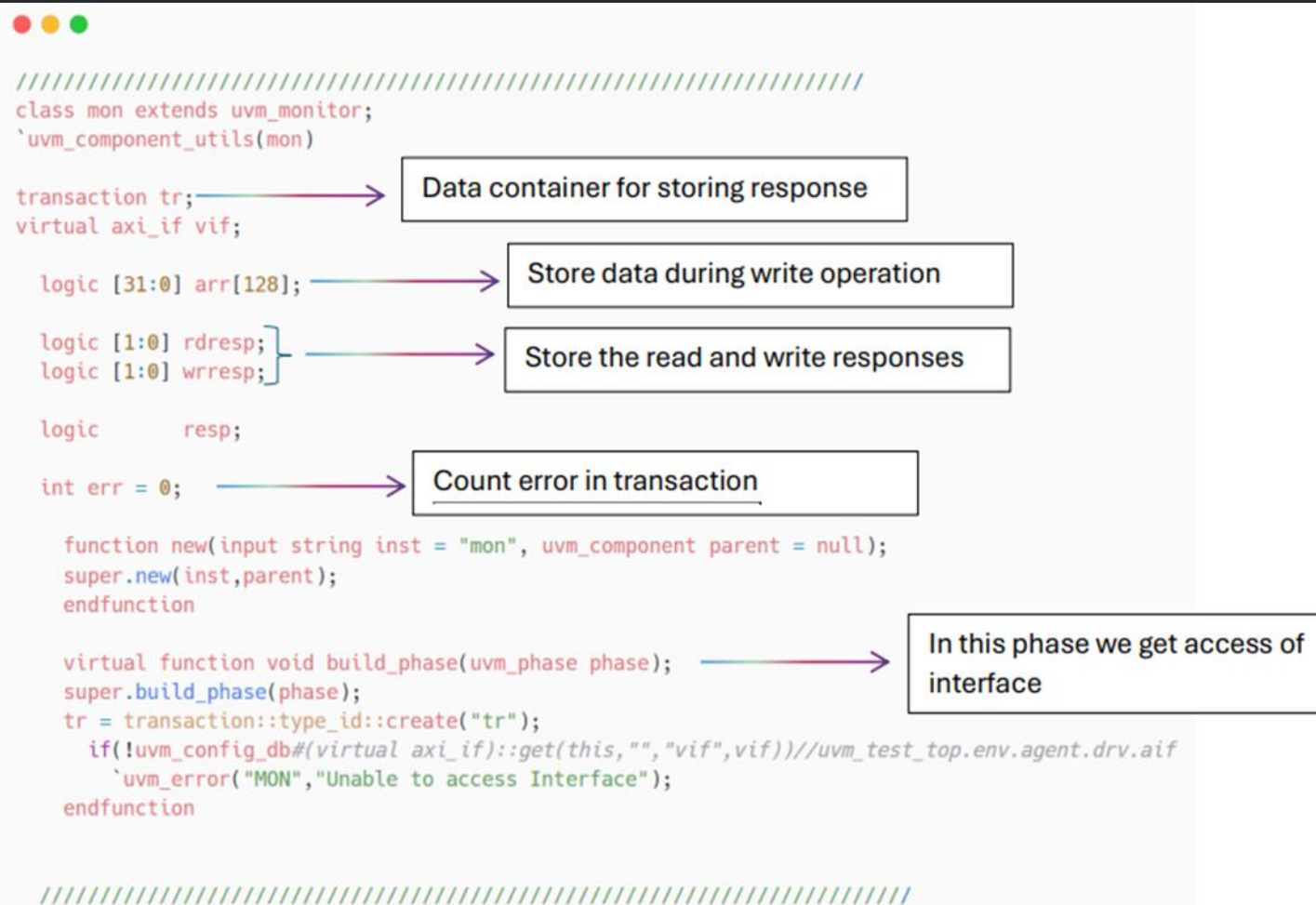
  for(int i = 0; i < (vif.awlen); i++)//0 - 6 -> 7
  begin
    vif.wdata     <= $urandom_range(0,10);
    vif.wstrb     <= 4'b1111;
    @(posedge vif.wready);
    @(posedge vif.clk);
  end
  vif.awvalid    <= 1'b0;
  vif.wvalid     <= 1'b0;
  vif.wlast      <= 1'b1;
  vif.bready     <= 1'b1;
  @(negedge vif.bvalid);
  vif.wlast      <= 1'b0;
  vif.bready     <= 1'b0;
  //////////// read logic
endtask
```

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

All the lanes have valid data

Monitor

Overview: The monitor class (mon) is responsible for observing signals in the AXI protocol and comparing expected vs actual behavior. It extends the `uvm_monitor` class and includes key methods for initialization, monitoring, and error-checking.

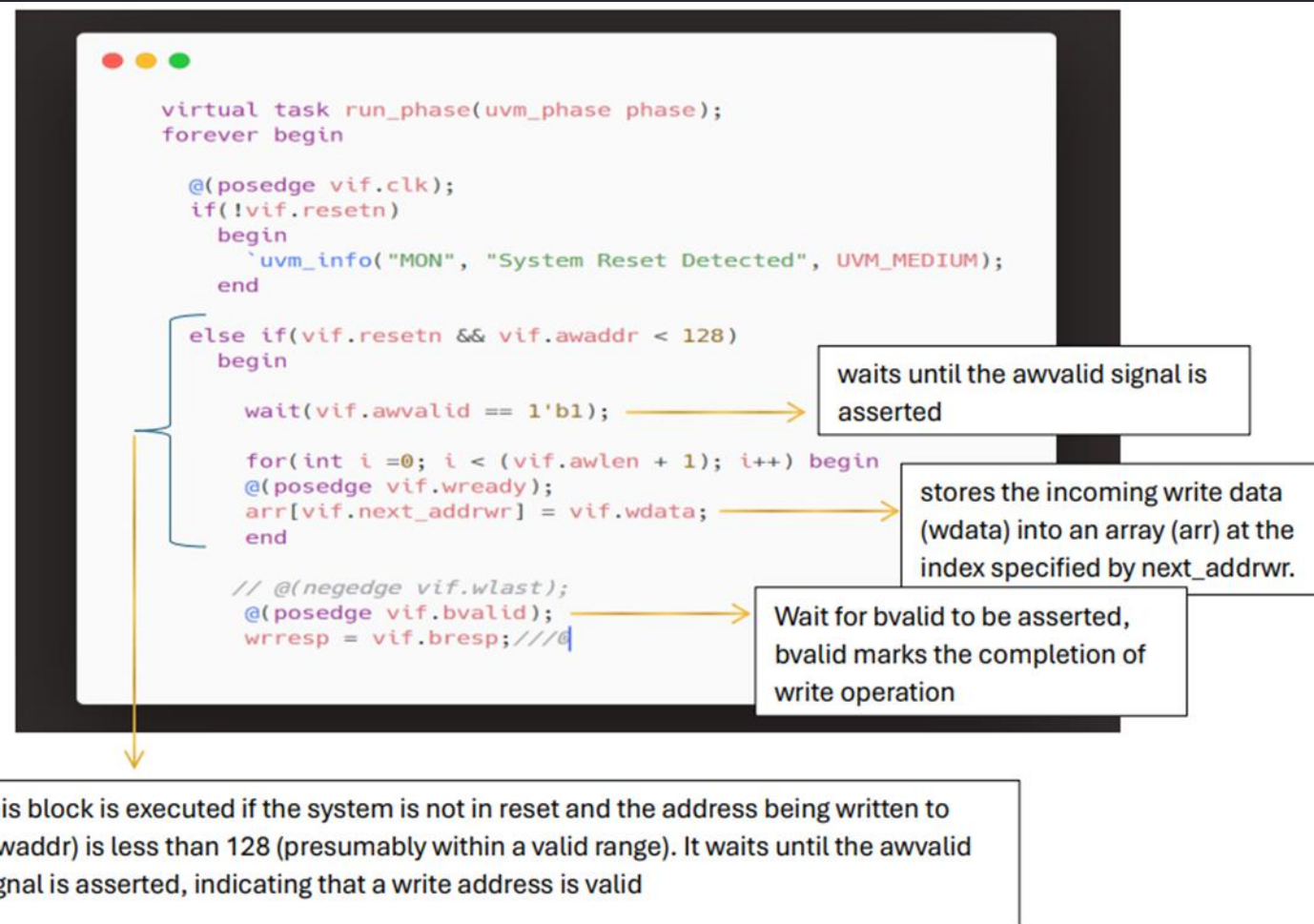


Key Components:

- Transaction Object: transaction tr: Used to capture the transactions observed during the simulation.
- Virtual Interface: virtual axi_if vif: Interface through which the monitor observes the signals (like clk, resetn, awaddr, etc.) in the AXI protocol.
- Handling and Responses: logic [1:0] rdresp, wrresp: Stores read and write responses. int err: Tracks the number of errors encountered during the read-write operations.

Monitor

Overview: The monitor class (mon) is responsible for observing signals in the AXI protocol and comparing expected vs actual behavior. It extends the uvm_monitor class and includes key methods for initialization, monitoring, and error-checking.



Phases:

- Build Phase: Creates the transaction object (tr=transaction::type_id::create("tr")). Sets up the interface to the vif using uvm_config_db and raises an error if it cannot be accessed.
- Run Phase: Monitors the AXI protocol transactions (awvalid, arvalid, rdata, wdata). Captures write data into an array arr[] and checks read data against it. Waits for valid write (awvalid) and read (arvalid) signals, ensuring data correctness.
- Error Checking and Comparison: The compare() task checks if errors occurred and logs the results (either success or failure) based on the read/write responses and error count. Displays appropriate messages using UVM info methods.

Monitor

Error Increment Concept Explained

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

1. Wait for Read Address Valid (arvalid):

- o 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:

- o 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):

- o 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):

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

- o This process ensures that any discrepancy between what was written and what was read back is counted as an error

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

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

Agent

The agent class is responsible for coordinating the driver, sequencer, and monitor.

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

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

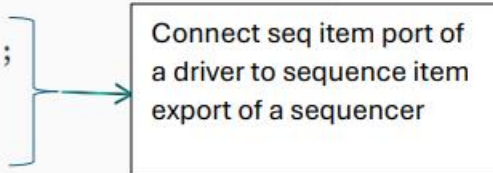
driver d;
uvm_sequencer#(transaction) seqr;
mon m;

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

Phases:

- Build Phase:** Creates the driver, sequencer, and monitor instances.
- Connect Phase:** Connects the sequencer's port to the driver's export for transaction flow.

Environment (env) Class:

The env class acts as a container for the agent.

```
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
```

Test Class:

The test class is the top-level control for UVM testing.

Environment (e): Holds an instance of the environment.

Sequences: Multiple sequences for different scenarios (e.g., valid_wrrd_fixed, valid_wrrd_incr, valid_wrrd_wrap, err_wrrd_fix).

Reset DUT Sequence: Handles the reset of the DUT.

Phases:

Build Phase: Creates instances of the environment and different sequences.

Run Phase: Manages the test execution, running sequences one by one.

```
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
```


Testbench (TB) Module:

The tb module instantiates the DUT and handles clock/reset logic.

```
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_addr rd = dut.rdnnextaddr;

endmodule
```

We provide access to
interface to component