AXI Verification IP

Developer Guide

1. Table of Contents

1. Preface

This document is intended for engineers who will maintain the AXI verification IP (VIP). This document describes how the AXI VIP is implemented and how it should be used. As a prerequisite, the reader should be knowledgeable in both AXI protocol and Universal Verification Methodology (UVM).

Related documents:

- AMBA® AXI™ and ACE™ Protocol Specification
- Universal Verification Methodology (UVM) 1.2 User's Guide

2. Introduction

The AXI VIP supports the verification of designs using AMBA AXI3 and AXI4 protocols. The VIP is written in SystemVerilog while following the Universal Verification Methodology (UVM).

3. Implementation Details

The AXI VIP supports operating as either an AXI master or an AXI slave. Furthermore, the AXI slave is configurable as either active (reactive agent) or passive (monitor only). The AXI interface is split into 5 interfaces, one for each channel of the AXI protocol. Interactions between these channels are handled at the sequence level to give users full control in creating various test scenarios. The interfaces are parameterized to allow signal widths to be configured. The sequence item, sequence, and other UVM components are also parametrized to support any signal width required by users.

3.1. Parameterization

3.1.1. Interface Parameterization

Interfaces have parameters to define the width of multi-bit signals. Users can override these parameters to match their requirements. Details on these parameters are found in section 3.7.

3.1.2. Class Parameterization

The AXI VIP uses data-type parameterization for its UVM object and component classes (e.g., sequence, agent, driver, etc.). Shown below is an example.

```
class axi_master_agent # (type T=axi_params) extends uvm_agent;
   `uvm_component_param_utils(axi_master_agent#(T))
.
```

By default, *T* is of type *axi_params*. The class *axi_params* is a virtual class which contains parameters for all the signal widths.

Table 1. Class Parameters

Parameter	Default Value	Description
AXI_ID_WIDTH	7	ID width
AXI_ADDR_WIDTH	64	ADDR width
AXI_LEN_WIDTH	8	LEN width
AXI_SIZE_WIDTH	3	SIZE width
AXI_BURST_WIDTH	2	BURST width
AXI_LOCK_WIDTH	2	LOCK width
AXI_CACHE_WIDTH	4	CACHE width
AXI_PROT_WIDTH	3	PROT width
AXI_QOS_WIDTH	4	QOS width
AXI_REGION_WIDTH	4	REGION width
AXI_USER_WIDTH	4	USER width
AXI_DATA_WIDTH	64	DATA width
AXI_STRB_WIDTH	8	STRB width
AXI_RESP_WIDTH	2	RESP width

To override these parameters, users can extend *axi_params* and create a child class. This child class can then be used during instantiation of agents and sequences to parameter override the default type. Declaration of this child class is done inside a package. Shown below is an example.

```
 1 package sys;
 2   import uvm_pkg::*;
 3   `include "uvm_macros.svh"
 4   import axi_pkg::*;
 5
 6   `include "axi_params.sv"
 7
 8   //extend axi_params class and override parameters
 9   class axi_port_32x32x4 extends axi_params;
10     parameter AXI_ADDR_WIDTH = 32;
11     parameter AXI_DATA_WIDTH = 32;
12     parameter AXI_STRB_WIDTH = AXI_DATA_WIDTH/8;
13     parameter AXI_ID_WIDTH = 4;
14
15    //Used by write and read address channels
16    typedef axi_addr_item #(
17      .AXI_ID_WIDTH(AXI_ID_WIDTH),
18      .AXI_ADDR_WIDTH(AXI_ADDR_WIDTH),
19      .AXI_LEN_WIDTH(AXI_LEN_WIDTH),
20      .AXI_SIZE_WIDTH(AXI_SIZE_WIDTH),
21      .AXI_BURST_WIDTH(AXI_BURST_WIDTH),
22      .AXI_LOCK_WIDTH(AXI_LOCK_WIDTH),
23      .AXI_CACHE_WIDTH(AXI_CACHE_WIDTH),
24      .AXI_PROT_WIDTH(AXI_PROT_WIDTH),
25      .AXI_QOS_WIDTH(AXI_QOS_WIDTH),
26      .AXI_REGION_WIDTH(AXI_REGION_WIDTH),
27      .AXI_USER_WIDTH(AXI_USER_WIDTH)
28    ) axi_addr_item_t;
29
30    //Used by write and read data channels
31    typedef axi_data_item #(
32      .AXI_ID_WIDTH(AXI_ID_WIDTH),
33      .AXI_DATA_WIDTH(AXI_DATA_WIDTH),
34      .AXI_STRB_WIDTH(AXI_STRB_WIDTH),
35      .AXI_RESP_WIDTH(AXI_RESP_WIDTH)
36    ) axi_data_item_t;
37
38    //Used by write response channel
39    typedef axi_resp_item #(
40      .AXI_ID_WIDTH(AXI_ID_WIDTH),
41      .AXI_RESP_WIDTH(AXI_RESP_WIDTH),
42      .AXI_USER_WIDTH(AXI_USER_WIDTH)
43    ) axi_resp_item_t;
```

```
44
45      typedef virtual axi_addr_inf #(
46        .ID_WIDTH     (AXI_ID_WIDTH),
47        .ADDR_WIDTH   (AXI_ADDR_WIDTH),
48        .LEN_WIDTH    (AXI_LEN_WIDTH),
49        .SIZE_WIDTH   (AXI_SIZE_WIDTH),
50        .BURST_WIDTH  (AXI_BURST_WIDTH),
51        .LOCK_WIDTH   (AXI_LOCK_WIDTH),
52        .CACHE_WIDTH  (AXI_CACHE_WIDTH),
53        .PROT_WIDTH   (AXI_PROT_WIDTH),
54        .QOS_WIDTH    (AXI_QOS_WIDTH),
55        .REGION_WIDTH (AXI_REGION_WIDTH),
56        .USER_WIDTH   (AXI_USER_WIDTH)
57      ) addr_if_t;
58
59      typedef virtual axi_data_inf #(
60        .ID_WIDTH     (AXI_ID_WIDTH),
61        .DATA_WIDTH   (AXI_DATA_WIDTH),
62        .STRB_WIDTH   (AXI_STRB_WIDTH),
63        .USER_WIDTH   (AXI_USER_WIDTH),
64        .RESP_WIDTH   (AXI_RESP_WIDTH)
65      ) data_if_t;
66
67      typedef virtual axi_resp_inf #(
68        .ID_WIDTH     (AXI_ID_WIDTH),
69        .RESP_WIDTH   (AXI_RESP_WIDTH),
70        .USER_WIDTH   (AXI_USER_WIDTH)
71      ) resp_if_t;
72
73    endclass
74
75  endpackage
```

In the example, *axi_params* is extended in line #9. In lines #10 to #13, 4 parameters were overridden to change their value. Lines #16 to #71 are needed so that the definition of the type defines are also overridden with the new values. These type defines are used internally in the code of the VIP. Users can create multiple child classes from *axi_params* if their testbench uses multiple AXI buses with different signal widths. Refer to section 3.8 Environment for example of agent instantiation with parameter override.

3.2. AXI Master Agent

The AXI master agent is used to issue AXI write and read transactions. Each AXI channel has its own independent driver and interface. The write address, write data, and read address channel drivers are each connected to their respective sequencers. The write response and read data channels are each assigned a ready signal driver. The ready signal drivers toggle the ready signal of the write response channel and read data channel based on the configuration set by the user. This allows the user to create back-pressure on the write response channel and read data channel.

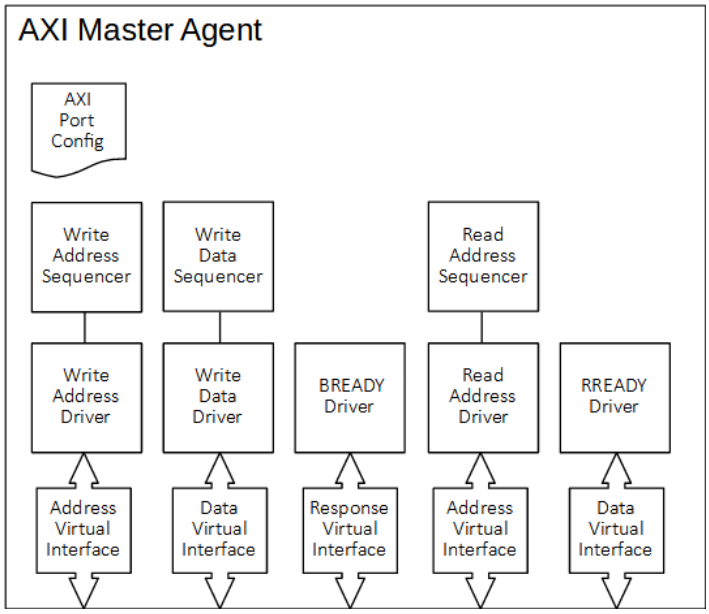


Figure 1. AXI Master Agent Diagram

3.2.1. Address Driver

The AXI master uses a common address driver for its write and read address channels. This driver controls the address valid (AxVALID) signal and other address channel signals that must be driven by the master. The common address driver toggles the pins of the address channel based on information it gets from the *axi_addr_item* sequence item passed by the address sequencer.

3.2.2. Data Driver

The AXI master uses a common data driver for its write data channel. This driver controls the data valid (xVALID) signal and other data channel signals that must be driven by the master. The common data driver toggles the pins of the data channel based on the information it gets from *axi_data_item* sequence item passed by the data sequencer.

3.2.3. Ready Driver

The AXI master uses a common ready signal driver for its write response and read data channel. Unlike the address and data drivers, the ready driver does not need a sequence item. The ready driver toggles the ready signal based on the settings found in *axi_port_cfg*. Listed below are the fields of *axi_port_cfg* that are used by ready driver.

Table 2. Ready driver configuration fields

Field Name	Default Value	Description
min_axready_delay	0	From the time valid signal is high, this field controls the **minimum** number of clock cycles the **address** channel ready (AxREADY) signal is low before the driver sets AxREADY to high.
max_axready_delay	0	From the time the valid signal is high, this field controls the **maximum** number of clock cycles the **address** channel ready (AxREADY) signal is low before the driver sets AxREADY to high.
axready_high_cycles	0	From the time AxREADY becomes high, this field specifies the number of clock cycles the **AxREADY** signal remains high.

Field Name	Default Value	Description
min_xready_delay	0	From the time the valid signal is high, this field controls the **minimum** number of clock cycles the **data** channel ready (xREADY) signal is low before the driver sets xREADY to high.
max_xready_delay	0	From the time the valid signal is high, this field controls the **maximum** number of clock cycles the **data** channel ready (xREADY) signal is low before the driver sets xREADY to high.
xready_high_cycles	0	From the time xREADY becomes high, this field specifies the number of clock cycles the **xREADY** signal remains high.
min_bxready_delay	0	From the time the valid signal is high, this field controls the **minimum** number of clock cycles the **write response** channel ready (BREADY) signal is low before the driver sets BREADY to high.
max_bxready_delay	0	From the time the valid signal is high, this field controls the **maximum** number of clock cycles the **write response** channel ready (BREADY) signal is low before the driver sets BREADY to high.
bxready_high_cycles	0	From the time BREADY becomes high, this field specifies the number of clock cycles the B**READY** signal remains high.

When both min and max delays are set to zero, the ready signal is always high when not in reset as illustrated in the next figure.

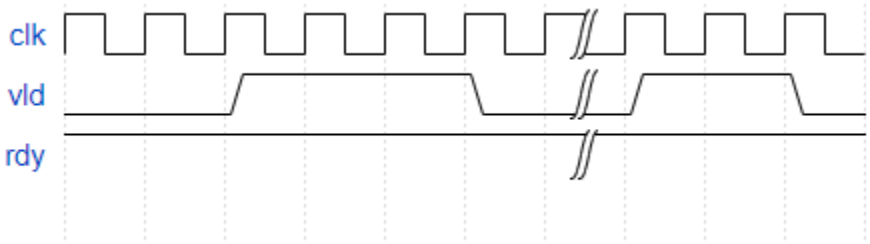


Figure 2. Both min and max ready delays are zero

When the max delay is non-zero, the driver will randomize the delay between the min and max values. The max delay value should always be greater than or equal the min delay value. In the Figure 3 below, min delay is zero, max delay is five, and high cycles is 2.

In the example below, delay is randomized to 0 which resulted in the ready signal being high immediately after reset. Since high cycles is set to 2, ready signal remains asserted for 2 cycles (clock cycles #3 & #4) from the time the valid signal asserted. After ready signal negation, delay is randomized to 3 cycles so ready signal remains negated for 3 cycles (#5 to #7). At cycle #8, ready signal is asserted again for 2 cycles until cycle #9. Delay is then randomized to 1 so ready signal waits for valid signal to become high for 1 cycle before becoming asserted at cycle #12.

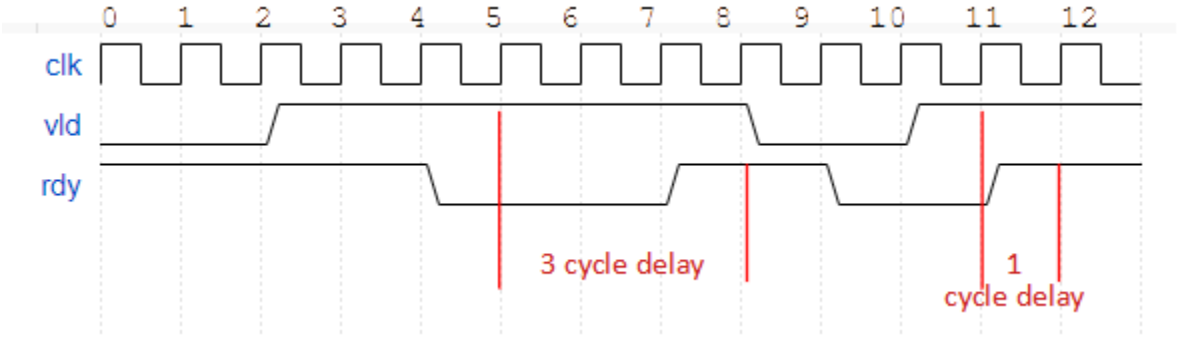


Figure 3. Non-zero max delay

3.2.4. Virtual Interface

The master agent has virtual interface handles for each of the 5 AXI channels. The master agent gets the interfaces from the UVM config_db. It uses the master agent name concatenated to the interface name as the *field_name* (3rd argument) to the config_db get() function, as shown below.

```
if (!uvm_config_db#(addr_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.mstr_waddr_inf", get_name()), awaddr_vif)) begin
   `uvm_fatal(get_name(), "No Virtual Master Interface specified for mstr_waddr_inf")
end
if (!uvm_config_db#(addr_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.mstr_raddr_inf", get_name()), araddr_vif)) begin
   `uvm_fatal(get_name(), "No Virtual Master Interface specified for mstr_raddr_if")
end
if (!uvm_config_db#(data_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.mstr_wdata_inf", get_name()), wdata_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for mstr_wdata_if")
end
if (!uvm_config_db#(data_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.mstr_rdata_inf", get_name()),rdata_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for mstr_rdata_if")
end
if (!uvm_config_db#(resp_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.mstr_resp_inf", get_name()),bresp_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for mstr_resp_if")
end
```

3.3. AXI Slave Agent

The AXI slave agent is used to receive write and read requests and issue corresponding responses back to the master. It has monitors on each channel which passes transaction information to its AXI model. The AXI model, inside the AXI slave agent, assembles the information it gathers from each channel and performs the necessary action. For write transactions, the model writes data into the system memory and assembles a write response that it puts into an outbound write response queue. For read transactions, it gets data from the memory and assembles a read response that it puts into an outbound read response queue. These outbound queues are accessed by the user inside a sequence to issue the responses in the appropriate channel.

Shown in Figure 4 is the slave configured as an active agent. The active slave agent has sequencers and drivers on its write response channel and read data channel that allows it to respond to AXI write and read requests. It also has ready drivers that allows it to control the ready signals on the write address, read address, and write data channels.

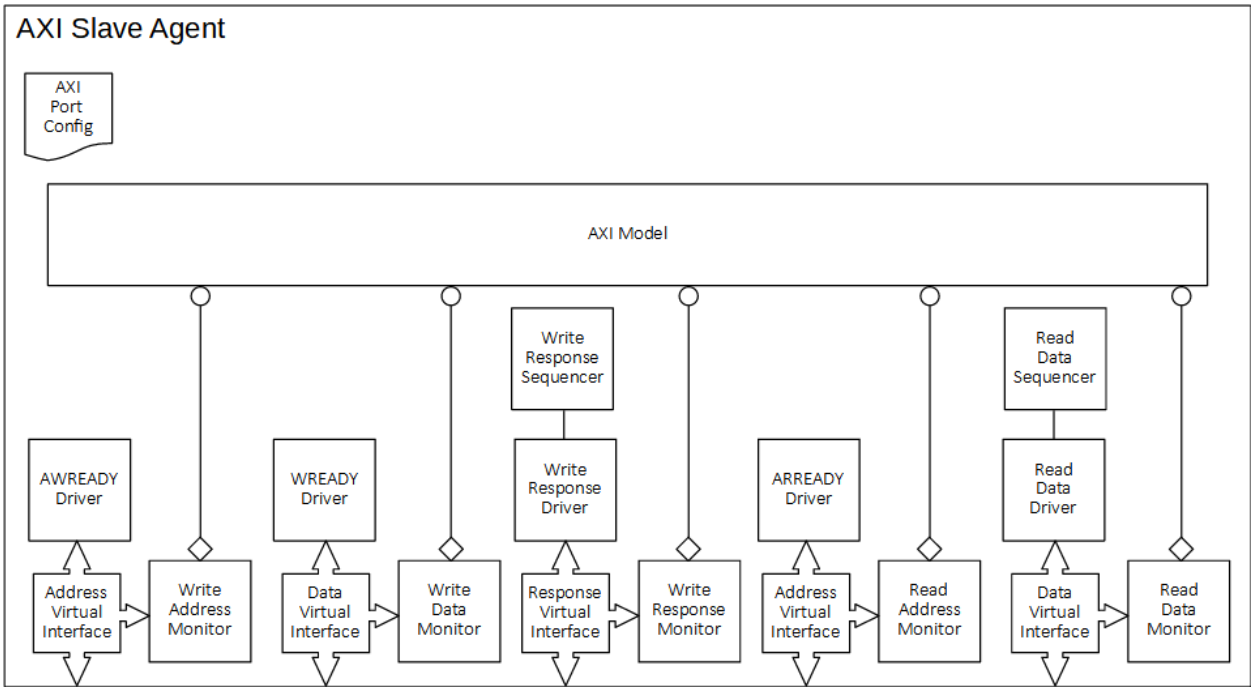


Figure 4. AXI Slave Active Agent Diagram

Shown in Figure 5 is the slave configured as a passive agent. The passive slave agent is only used to monitor and sample transactions on the AXI interface.

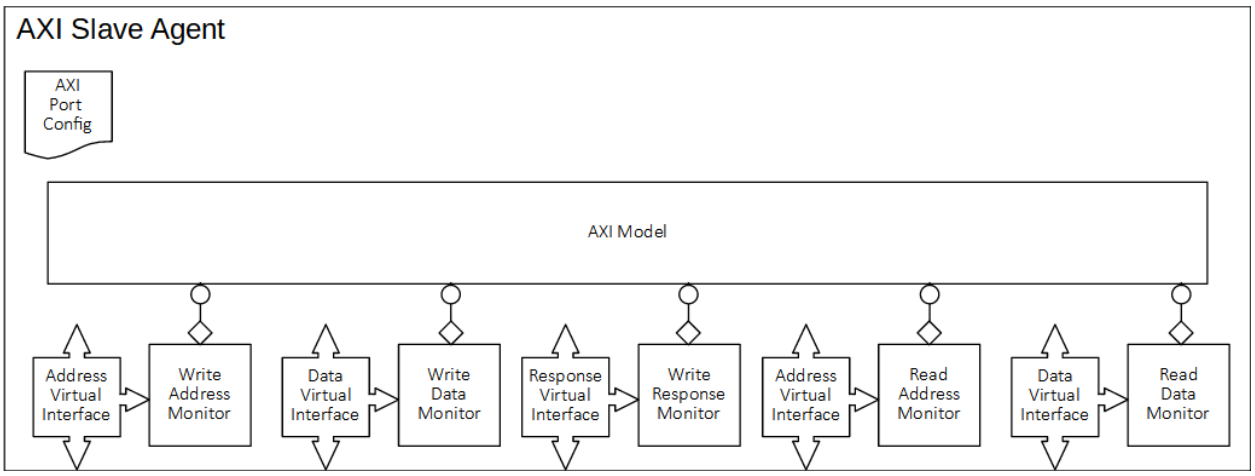


Figure 5. AXI Slave Passive Agent Diagram

3.3.1. Data Driver

The AXI slave agent uses the same data driver used by the master. See section 3.3.2 Data Driver for details. This driver is used by the slave to send read data(response).

3.3.2. Ready Driver

The AXI slave agent uses the same ready driver used by the master. See section 3.4.2 Ready Driver for details.

3.3.3. Write Response Driver

The AXI slave agent uses a write response driver to control the write response valid (BVALID) signal and other write response channel signals that must be driven by the slave. It toggles the pins of the write response channel based on the information it gets from *axi_resp_item* sequence item passed by write response sequencer.

3.3.4. Address Monitor

The AXI slave uses a common address monitor for both of its write and read address channels. This monitor samples the address channel whenever a valid-ready handshake occurs and creates an *axi_addr_item* sequence item. This sequence item is then broadcasted via its analysis port *axi_addr_aport*.

3.3.5. Data Monitor

The AXI slave uses a common data monitor for both of its write and read data channels. This monitor samples the data channel whenever a valid-ready handshake occurs and creates an axi_data_item sequence item. The data monitor waits for the entire burst to complete (i.e. last signal is high during valid-ready handshake) before the sequence item is broadcasted via its analysis port *axi_data_aport*. The data monitor can handle interleaved data by matching the id of each transfer. In the case of AXI4, wherein WID signal is not used, the xID signal of the write data channel interface should be fixed to 0 for the monitor to function correctly.

3.3.6. Write Response Monitor

The AXI slave uses a write response monitor to sample response on the write response channel. A response is sampled whenever a valid-ready handshake occurs. The monitor creates an *axi_resp_item* sequence item and broadcasts it via its analysis port *axi_resp_aport.*

3.3.7. Virtual Interface

The slave agent has virtual interface handles for each of the 5 AXI channels. The slave agent gets the interfaces from the UVM config_db. It uses the slave agent name concatenated to the interface name as the *field_name* (3rd argument) to the config_db get() function, as shown below.

```
if (!uvm_config_db#(addr_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.slv_waddr_inf", get_name()), awaddr_vif)) begin
   `uvm_fatal(get_name(), "No Virtual Master Interface specified for slv_waddr_inf")
end
if (!uvm_config_db#(addr_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.slv_raddr_inf", get_name()), araddr_vif)) begin
   `uvm_fatal(get_name(), "No Virtual Master Interface specified for slv_raddr_if")
end
if (!uvm_config_db#(data_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.slv_wdata_inf", get_name()), wdata_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for slv_wdata_if")
end
if (!uvm_config_db#(data_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.slv_rdata_inf", get_name()),rdata_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for slv_rdata_if")
end
if (!uvm_config_db#(resp_if_type)::get(null, "uvm_test_top",
    $sformatf("%s.slv_resp_inf", get_name()),bresp_vif)) begin
   `uvm_fatal(get_full_name(), "No Virtual Master Interface specified for slv_resp_if")
end
```

3.3.8. AXI Model

The AXI slave agent uses an AXI model to assemble the information gathered from all the channels into a complete AXI burst. The AXI model does this by subscribing to each of the channel monitors and matching the transaction IDs from these channels.

For write transactions, the model matches the write address with its corresponding write data. The model then writes the data into the appropriate address location in the system memory. After data is written, the model prepares a default write response of type *axi_resp_item*. The default response contains an OKAY write response, matching response id, user information equal to 0, and random delay. The model pushes this response into an outbound write response queue called *outbound_bresp_Q*. This queue is intended to be used by sequences that handle sending of write responses.

For read transactions, the model prepares a default read response of type *axi_data_item* whenever it receives a request from the read address channel. The default read response contains data from system memory, matching id, user information equal to 0, and random delay. The default response also contains strobe information that indicates which byte lanes contain valid read data; this information is helpful when the request is either a narrow data transfer or unaligned transfer. The model pushes this response into an outbound read response queue called *outbound_rresp_Q*. This queue is intended to be used by sequences that handle sending of read responses.

The AXI model also provides a third outbound queue that holds the complete AXI burst (address, data, and/or response). The user may use this queue in creating more complex response sequences. For example, users want to modify BRESP or RRESP values depending on the value of the address or some other condition.

3.3.9. System Memory

The system memory stores data sent to the slave. It uses an 8-bit associative array as storage. The depth of the array is determined by the ADDR_WIDTH parameter. It can hold 2^ADDR_WIDTH number of bytes. The system memory is instantiated outside the slave agent and a handle is simply passed to the slave model.

Writing and reading to the memory is done via the *writeMem()* and *readMem()* functions.

```
   function void writeMem(bit[ADDR_WIDTH-1:0] start_addr, logic[7:0] wdata[$], bit
strb[$]);
      foreach (wdata[i]) begin
        $display(" - wdata[%0d] = 0x%0x", i, wdata[i]);
        if (strb[i] == 1'b1) begin
            mem[start_addr+i] = wdata[i];
        end
      end
   endfunction

   function void readMem(bit[ADDR_WIDTH-1:0] start_addr, int byte_count, ref
logic[7:0] rdata [$]);
      rdata.delete();
      for (int i=0; i<byte_count; i++) begin
         if (mem.exists(start_addr+i)) begin
            rdata.push_back(mem[start_addr+i]);
         end else begin
            rdata.push_back('0);
         end
      end
   endfunction
```

3.4. AXI Port Configuration

The AXI slave agent uses the common AXI port configuration class. Listed below are the fields used by slave agent.

Table 3. AXI Port Configuration Class Fields

Field Name	Target	Default Value	Description
version	Master & Slave	AXI3	This field specifies the AXI version, either AXI3 or AXI4.
is_active	Master	UVM_ACTIVE	This field configures the slave agent as either active or passive. Active: The slave agent has both drivers and monitors on all channels. The slave agent is able send response as well as sample information from all channels Passive: The slave agent only has monitors on all channels. It is only used to sample information from all channels.
min_axready_delay	Slave	0	From the time valid signal is high, this field controls the minimum number of clock cycles the address channel ready (AxREADY) signal is low before the driver sets AxREADY to high.
max_axready_delay	Slave	0	From the time the valid signal is high, this field controls the maximum number of clock cycles the address channel ready (AxREADY) signal is low before the driver sets AxREADY to high.
axready_high_cycles	Slave	0	From the time AxREADY becomes high, this field specifies the number of clock cycles the AxREADY signal remains high.
min_xready_delay	Master & Slave	0	From the time the valid signal is high, this field controls the minimum number of clock cycles the data channel ready (xREADY) signal is low before the driver sets xREADY to high.
max_xready_delay	Master & Slave	0	From the time the valid signal is high, this field controls the maximum number of clock cycles the data channel ready (xREADY) signal is low before the driver sets xREADY to high.
xready_high_cycles	Master & Slave	0	From the time xREADY becomes high, this field specifies the number of clock cycles the xREADY signal remains high.
min_bxready_delay	Master	0	From the time the valid signal is high, this field controls the minimum number of clock cycles the write response channel ready (BREADY) signal is low before the driver sets BREADY to high.
max_bxready_delay	Master	0	From the time the valid signal is high, this field controls the maximum number of clock cycles the write response channel ready (BREADY) signal is low before the driver sets BREADY to high.

bxready_high_cycles	Master	0	From the time BREADY becomes high, this field specifies the number of clock cycles the BREADY signal remains high.
wr_resp_min_delay	Slave	0	Minimum number of clock cycles before the slave sends a write response.
wr_resp_max_delay	Slave	0	Maximum number of clock cycles before the slave sends a write response.
rd_resp_min_delay	Slave	0	Minimum number of clock cycles before the slave sends a read response.
rd_resp_max_delay	Slave	0	Maximum number of clock cycles before the slave sends a read response.

3.5. AXI System Config

The AXI system config holds all the AXI port configs. It has a function that takes in the number of masters and number slaves as input and creates default AXI port configs. Users can call this function in their base test to create the port configs. Users can then register the system config into the UVM config_db so it can be used in the environment. Refer to the example below.

```
class sig_test_base extends uvm_test;
   `uvm_component_utils(sig_test_base)

   axi_sys_cfg sys_cfg;

   function void build_phase(uvm_phase phase);
      sys_cfg = axi_sys_cfg::type_id::create("sys_cfg", this);
      sys_cfg.setDefaultPortCfg(2,4); //Create 2 port configs for 2 master agents
                                      //Create 3 port configs for 4 slave agents
      uvm_config_db#(axi_sys_cfg)::set(null, "*", "axi_system_cfg", sys_cfg);
   endfunction // build_phase
.
.
```

3.6. AXI Channel Interfaces

Each channel of AXI protocol is assigned a separate interface. These interfaces are parameterized to give users control over the widths of each signal. These interfaces also make use of clocking blocks to ensure the signals are synchronous with the AXI clock. These clocking blocks have an input (sampling) skew of 1 and an output (driving) skew of 1. This means input signals are sampled 1 time unit before the positive edge of the clock and output signals are driven 1 time unit after the positive edge of the clock. Another feature of these interface are the signal-level assertions implemented within the interface.

Figure 6 and Figure 7 shows which clocking blocks are used by drivers and monitors of the master and slave agents.

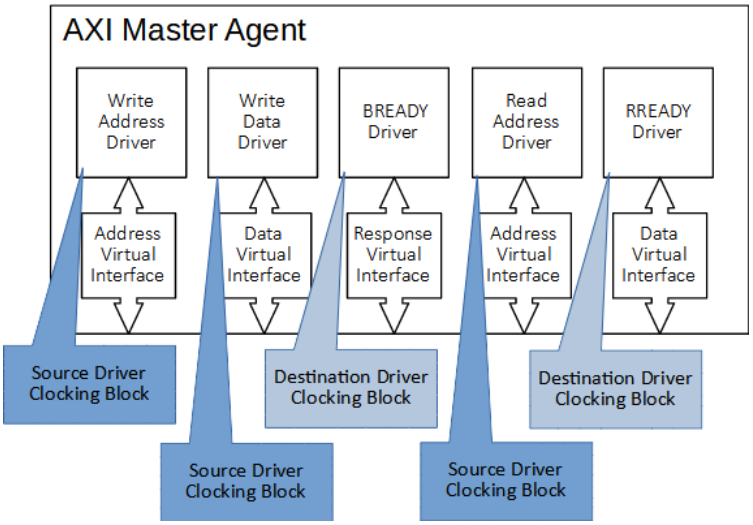


Figure 6. AXI Master Agent Clocking Blocks

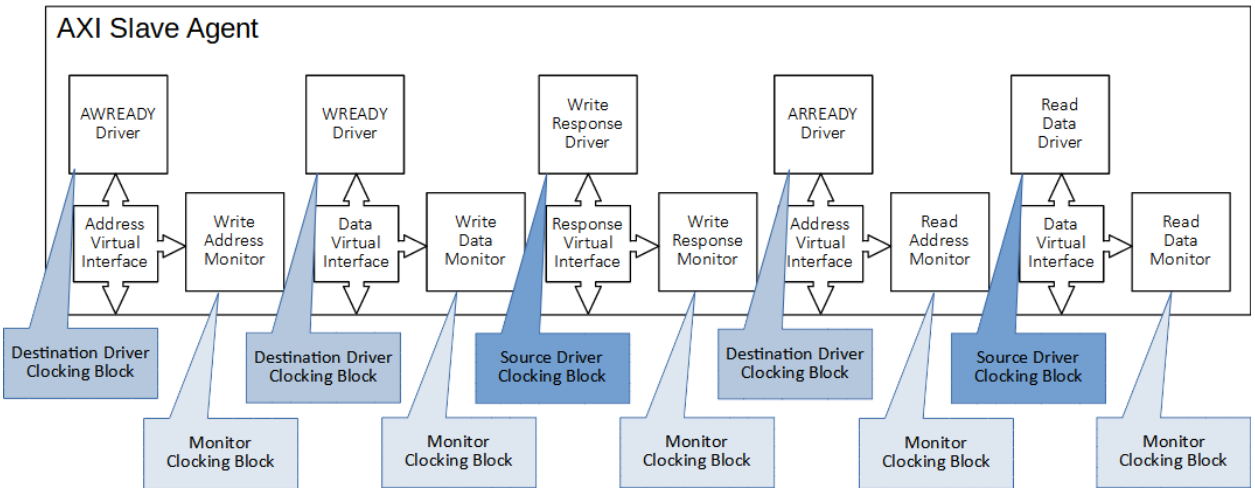


Figure 7. AXI Slave Agent Clocking Blocks

3.6.1. Address Channel Interface

The address channel interface is used for both write and read address channels. This interface contains signals used by both AXI3 and AXI4. For signals that are not intended to be used, user should fix these signals to 0. Listed below are the parameters of the address channel interface. The user can change the signal widths by overriding these parameters during instantiation.

Table 4. Address Channel Interface Parameters

Parameter	Default Value	Description
ID_WIDTH	7	AxID signal width
ADDR_WIDTH	64	AxADDR signal width
LEN_WIDTH	8	AxLEN signal width
SIZE_WIDTH	3	AxSIZE signal width
BURST_WIDTH	2	AxBURST signal width
LOCK_WIDTH	2	AxLOCK signal width
CACHE_WIDTH	4	AxCACHE signal width

Parameter	Default Value	Description
PROT_WIDTH	3	AxPROT signal width
QOS_WIDTH	4	AxQOS signal width
REGION_WIDTH	4	AxREGION signal width
USER_WIDTH	4	AxUSER signal width

Listed below are signals of the address channel interface and their respective directions in each of the clocking blocks.

Table 5. Address Channel Interface Signals and Clocking Blocks

Signal	Direction		
	Source Driver Clocking Block	Destination Driver Clocking Block	Monitor Clocking Block
clk	-	-	-
resetn	IN	IN	IN
AxREADY	IN	OUT	IN
AxVALID	OUT	IN	IN
AxID	OUT	IN	IN
AxADDR	OUT	IN	IN
AxLEN	OUT	IN	IN
AxSIZE	OUT	IN	IN
AxBURST	OUT	IN	IN
AxLOCK	OUT	IN	IN
AxCACHE	OUT	IN	IN
AxPROT	OUT	IN	IN
AxQOS	OUT	IN	IN
AxREGION	OUT	IN	IN
AxUSER	OUT	IN	IN

3.6.1.1. Source Driver Clocking Block

The source driver clocking block is used by address drivers of the master, which are the sources of address transactions.

3.6.1.2. Destination Driver Clocking Block

The destination driver clocking block is used by the address channel ready drivers of the slave agent, which are the recipient of address transactions.

3.6.1.3. Monitor Clocking Block

The monitor clocking block is used by the address monitors of the slave agent.

3.6.2. Data Channel Interface

The data channel interface is used for both write and read address channels. This interface contains signals used by both AXI3 and AXI4. For signals that are not intended to be used, user should fix these signals to 0. Listed below are the parameters of the data channel interface. The user can change the signal widths by overriding these parameters during instantiation.

Table 6. Data Channel Interface Parameters

Parameter	Default Value	Description
ID_WIDTH	7	xID signal width
DATA_WIDTH	64	xDATA signal width
STRB_WIDTH	8	xSTRB signal width
USER_WIDTH	4	xUSER signal width
RESP_WIDTH	2	xRESP signal width

Listed below are signals of the data channel interface and their respective directions in each of the clocking blocks.

Table 7. Data Channel Interface Signals and Clocking Blocks

Signal	Direction		
	Source Driver Clocking Block	Destination Driver Clocking Block	Monitor Clocking Block
clk	-	-	-
resetn	IN	IN	IN
xREADY	IN	OUT	IN
xVALID	OUT	IN	IN
xID	OUT	IN	IN
xDATA	OUT	IN	IN
xSTRB	OUT	IN	IN
xLAST	OUT	IN	IN
xRESP	OUT	IN	IN
xUSER	OUT	IN	IN

3.6.2.1. Source Driver Clocking Block

The source driver clocking block is used by write data driver of the master and read data driver of the slave.

3.6.2.2. Destination Driver Clocking Block

The destination driver clocking block is used by the write data channel ready driver of the slave agent and the read data channel ready driver of the master agent.

3.6.2.3. Monitor Clocking Block

The monitor clocking block is used by the data monitors of the slave agent.

3.6.3. Response Channel Interface

The response channel interface is only used for the write response channel. This interface contains signals used by both AXI3 and AXI4. For signals that are not intended to be used, user should fix these signals to 0. Listed below are the parameters of the response channel interface. The user can change the signal widths by overriding these parameters during instantiation.

Table 8. Response Channel Interface Parameters

Parameter	Default Value	Description
ID_WIDTH	7	xID signal width
USER_WIDTH	4	xUSER signal width
RESP_WIDTH	2	xRESP signal width

Listed below are signals of the response channel interface and their respective directions in each of the clocking blocks.

Table 9. Data Channel Interface Signals and Clocking Blocks

Signal	Direction		
	Source Driver Clocking Block	Destination Driver Clocking Block	Monitor Clocking Block
clk	-	-	-
resetn	IN	IN	IN
BREADY	IN	OUT	IN
BVALID	OUT	IN	IN
BID	OUT	IN	IN
BRESP	OUT	IN	IN
BUSER	OUT	IN	IN

3.6.3.1. Source Driver Clocking Block

The source driver clocking block is used by the write response driver of the slave agent.

3.6.3.2. Destination Driver Clocking Block

The destination driver clocking block is used by the write response channel ready driver of the master agent.

3.6.3.3. Monitor Clocking Block

The monitor clocking block is used by the write response monitor of the slave agent.

3.7. Environment

The AXI master and slave agents are intended to be instantiated directly into any UVM environment. However, there are some key details that users must add into their environment to make sure the agents function correctly. These key points are listed below.

1. Package containing *axi_params* and *axi_params* child class should be imported in the environment
2. Agents, virtual sequencers, and system memory should be instantiated in the environment with parameter overrides when needed
3. In the build phase, create the agents, virtual sequencers, and system memory.
4. In the build phase, get the system config from config_db and assign the individual port configs to their respective master and slave agents
5. In the connect phase, pass the handle of the system memory in the environment to the handle of the system memory in the AXI model inside the AXI slave agents.
6. In the connect phase, pass the handle of the sequencers in the agents to the handle of the sequencers inside the virtual sequencer

```
import sys::*;
class sig_axi_env extends uvm_env;
   `uvm_component_utils(sig_axi_env)


   axi_sys_cfg sys_cfg;
   axi_slave_agent#(sys::axi_port_32x32x4) slv_agents[];
   axi_master_agent#(sys::axi_port_32x32x4) mstr_agents[];
   axi_virtual_sequencer#(sys::axi_port_32x32x4) vsqrs[];
   axi_system_memory#(sys::axi_port_32x32x4::AXI_ADDR_WIDTH) mem;


   function new(string name="", uvm_component parent);
      super.new(name, parent);
   endfunction // new


   function void build_phase(uvm_phase phase);
      string str;
      super.build_phase(phase);
      if (!uvm_config_db#(axi_sys_cfg)::get(this, "*", "axi_system_cfg", sys_cfg)) begin
         `uvm_info(get_full_name(), "Cannot find system config setting, use defaults", UVM_LOW)
         sys_cfg = new("sys_cfg");
         sys_cfg.setDefaultPortCfg(1, 1);
      end
      mstr_agents = new[sys_cfg.num_mstrs];
      slv_agents = new[sys_cfg.num_slvs];
      vsqrs = new[sys_cfg.num_mstrs];
      foreach(mstr_agents[i]) begin
         str = $sformatf("mstr_agent_%0d", i);
         mstr_agents[i] = axi_master_agent#(sys::axi_port_32x32x4)::type_id::create(str, this);
         mstr_agents[i].port_cfg = sys_cfg.mstr_prt_cfg[i];
      end
      foreach(slv_agents[i]) begin
         str = $sformatf("slv_agent_%0d", i);
         slv_agents[i] = axi_slave_agent#(sys::axi_port_32x32x4)::type_id::create(str, this);
         slv_agents[i].port_cfg = sys_cfg.slv_prt_cfg[i];
      end
      foreach(vsqrs[i]) begin
         str = $sformatf("vsqr_%0d", i);
         vsqrs[i] = axi_virtual_sequencer#(sys::axi_port_32x32x4)::type_id::create(str, this);
      end
      mem = axi_system_memory#(sys::axi_port_32x32x4::AXI_ADDR_WIDTH)::type_id::create( "mem", this);
   endfunction // build_phase


   function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      foreach(slv_agents[i]) begin
         slv_agents[i].slv_model.mem = mem;
      end
      foreach(vsqrs[i]) begin
         vsqrs[i].waddr_sqr = mstr_agents[i].axi_waddr_sequencer;
         vsqrs[i].wdata_sqr = mstr_agents[i].axi_wdata_sequencer;
         vsqrs[i].bresp_sqr = slv_agents[i].axi_bresp_sequencer;
         vsqrs[i].raddr_sqr = mstr_agents[i].axi_raddr_sequencer;
         vsqrs[i].rdata_sqr = slv_agents[i].axi_rdata_sequencer;
      end
   endfunction // connect_phase
endclass // sig_axi_env
```

3.8. Sequence Items

3.8.1. AXI Address Item

The AXI address item (*axi_addr_item*) is extended from UVM sequence item. It is used by both AXI address driver and monitor. It is used for both write address channel and read address channel. The driver gets the *axi_addr_item* from the sequencer and the sequencer gets it from the sequence. The monitor creates the axi_addr_item whenever there is a valid-ready handshake on the address channel. Listed below are fields of *axi_addr_item.*

Table 10. AXI Address Item Fields

Field	Description
id	ID tag. Used for AxID signal.
addr	Start address. Used for AxADDR signal.
len	Burst length. Used for AxLEN signal.
size	Burst size. Used for AxSIZE signal.
burst	Burst type. Used for AxBURST signal.
lock	Lock type. Used for AxLOCK signal.
cache	Memory type. Used for AxCACHE signal.
prot	Protection type. Used for AxPROT signal.
qos	Quality of service. Used for AxQOS signal.
region	Region identifier. Used for AxREGION signal.
user	User signal. Used for AxUSER signal.
delay	Delay. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.
rwType	Read/Write Flag. Indicates whether read or write burst.

3.8.2. AXI Data Item

The AXI data item (axi_data_item) is extended from UVM sequence item. It is used by both AXI data driver and monitor. It is used for both write data channel and read data channel. The driver gets the *axi_data_item* from the sequencer and the sequencer gets it from the sequence. The monitor creates the *axi_data_item* whenever it observes a complete burst on the data channel. Listed below are the fields of the *axi_data_item.*

Table 11. AXI Data Item Fields

Field	Description
id	ID tag. Used for xID signal.
data[$]	Queue of data transfers. Used for xDATA signal.
strb[$]	Queue of strobes. Used for xSTRB signal. Only used by drivers for writes.
user[$]	Queue of user information. Used for xUSER signal.
resp[$]	Queue of responses. Used for xRESP signal. Used only for reads.
delay[$]	Queue of transfer delays. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.
data_len	Burst length. Used by driver to determine number of transfers.
not_last	Flag used by sequence to let driver know not to assert last signal; used for interleaving data.
rwType	Read/Write Flag. Indicates whether read or write burst.

3.8.3. AXI Response Item

The AXI response item (*axi_resp_item*) is extended from UVM sequence item. It is used by both AXI response driver and monitor. It is only used for write response channel. The driver gets the *axi_resp_item* from the sequencer and the sequencer gets it from the sequence. The monitor creates the *axi_resp_item* whenever there is a valid-ready handshake on the write response channel. Listed below are the fields of *axi_resp_item.*

Table 12. AXI Response Item Fields

Field	Description
id	ID tag. Used for BID signal.
resp	Start address. Used for BRESP signal.
user	User signal. Used for BUSER signal.
delay	Delay. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.

3.8.4. AXI Burst Item

The AXI burst item (*axi_item*) is extended from UVM sequence item. It is created by the AXI model when it assembles the address, data, and response items into a complete AXI burst. This allows for all burst information to be available inside a single object. Listed below are the fields of *axi_item.*

Table 13. AXI Address Item Fields

Field	Description
id	ID tag.
addr	Start address.
len	Burst length.
size	Burst size.
burst	Burst type.
lock	Lock type.
cache	Memory type.
prot	Protection type.
qos	Quality of service.
region	Region identifier.
axuser	Address user information.
data[$]	Queue of data transfers.
strb[$]	Queue of strobes.
xuser[$]	Queue of user information on data channel.
buser	User information from write response channel.
rresp[$]	Queue of responses from the read data channel.
bresp	Response from write data channel.
rwType	Read/Write Flag. Indicates whether burst is read or write.

3.9. Sequences

The AXI VIP provides 3 basic sequences. These sequences are used as building blocks to create more complex sequences.

3.9.1. AXI Address Sequence

The AXI address sequence (*sig_axi_addr_seq*) is the basic sequence for the write and read address channels. This sequence issues a single transaction, either read or write, on the address channel. Listed below are the fields of this sequence.

Table 14. AXI Address Sequence Fields

Field	Description
seq_direction	Read/Write Flag. Indicates whether read or write burst.
seq_id	ID tag. Used for AxID signal.
seq_addr	Start address. Used for AxADDR signal.
seq_len	Burst length. Used for AxLEN signal.
seq_size	Burst size. Used for AxSIZE signal.
seq_burst	Burst type. Used for AxBURST signal.
seq_lock	Lock type. Used for AxLOCK signal.
seq_cache	Memory type. Used for AxCACHE signal.
seq_prot	Protection type. Used for AxPROT signal.
seq_qos	Quality of service. Used for AxQOS signal.
seq_region	Region identifier. Used for AxREGION signal.
seq_auser	User signal. Used for AxUSER signal.
seq_delay	Delay. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.

3.9.2. AXI Data Sequence

The AXI data sequence (*sig_axi_data_seq*) is the basic sequence for the write and read data channels. This sequence issues a single or partial burst of data, either read or write. Partial means last signal is not asserted at the end of the sequence. Partial bursts are sued to create interleaved transfers. Listed below are the fields of this sequence.

Table 15. AXI Data Sequence Fields

Field	Description
seq_direction	Read/Write Flag. Indicates whether read or write burst.
seq_data_len	Burst length. Used by driver to determine number of transfers.
seq_id	ID tag. Used for xID signal.
seq_data[]	Array of data transfers. Used for xDATA signal.
seq_strb[]	Array of strobes. Used for xSTRB signal. Only used by drivers for writes.
seq_user[]	Array of user information. Used for xUSER signal.
seq_resp[]	Array of responses. Used for xRESP signal. Used only for reads.
seq_delay[]	Array of transfer delays. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.
not_last	Flag used by sequence to let driver know not to assert last signal; used for interleaving data.

3.9.3. AXI Response Sequence

The AXI response sequence (*sig_axi_resp_seq*) is the basic sequence for the write data channel. This sequence issues a single write response. Listed below are the fields of this sequence.

Table 16. AXI Response Sequence Fields

Field	Description
seq_id	ID tag. Used for BID signal.
seq_resp	Start address. Used for BRESP signal.
seq_user	User signal. Used for BUSER signal.
seq_delay	Delay. Number of clock cycles before the driver toggles the interface pins. Not used by the monitor.

3.10. Virtual Sequences

The AXI VIP provides 2 virtual sequences that makes use of the basic sequences discussed in the previous section. Users can use these virtual sequences as reference in building more complex scenarios.

3.10.1. AXI Request Virtual Sequence

The AXI request virtual sequence (*sig_axi_req_vseq*) is used with the AXI master agent to issue a single write or read transaction. For write transactions, this virtual sequence starts an address sequence on the write address channel and a data sequence on the write data channel. For read transactions, this virtual sequence starts an address sequence on the read data channel. Listed below are the fields of this virtual sequence.

Table 17. AXI Request Virtual Sequence Fields

Field	Description
direction	Read/Write Flag. Indicates whether read or write burst.
id	ID tag.
addr	Start address.
len	Burst length.
size	Burst size.
burst	Burst type.
lock	Lock type.
cache	Memory type.
prot	Protection type.
qos	Quality of service.
region	Region identifier.
auser	User signal.
addr_delay	Address Delay.
data[]	Array of data transfers.
strb[]	Array of strobes.
duser[]	Array of user information.
data_delay[]	Array of data transfer delays.
not_last	Flag used by sequence to let driver know not to assert last signal; used for interleaving data.
just_data	When set to 1, the virtual sequence will not start the write address sequence; only the write data sequence will be started.
data_len	Burst length. Used by driver to determine number of transfers.
version	Set to either AXI3 or AXI4
wait_done	When set to1, waits for sequence to be finished before unblocking progress.

Shown below is a sample usage of this virtual sequence to issue write and read transactions.

```
class sig_axi_wr_rd_test extends sig_test_base;
.
.
   task run_phase(uvm_phase phase);
      sig_axi_req_vseq#(sys::axi_port_32x32x4) axi_req;
      phase.raise_objection(this);

      //Create and randomize write vseq
      axi_req = sig_axi_req_vseq#(sys::axi_port_32x32x4)::type_id::create("axi_req", this);
      axi_req.randomize() with {
         direction == WRITE;
         addr == 32'h1234;
         len == 7;
         size == 0;
         burst == INCR;
         addr_delay == 0;
      };
      axi_req.wait_done = 1;           //Waits for vseq to finish before proceeding
      axi_req.start(axi_env.vsqrs[0]); //Start vseq using virtual sequencer instantiated in env

      //Reuse created vseq to use same address channel fields but change direction to read
      axi_req.direction = READ;
      axi_req.start(axi_env.vsqrs[0]);
.
```

The example below shows how to use this virtual sequence to issue interleaved write data.

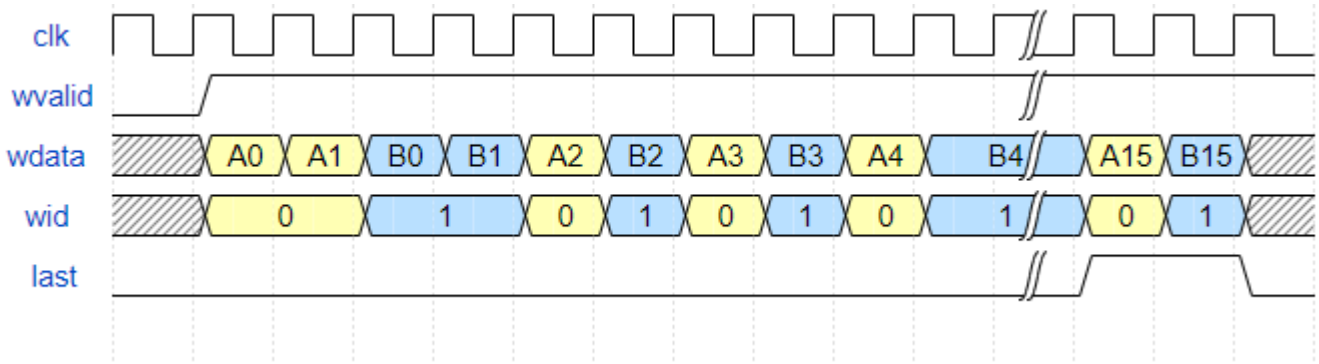
```
class sig_axi_write_interleaved_test extends sig_test_base;
.
.
   task run_phase(uvm_phase phase);
      sig_axi_req_vseq#(sys::axi_port_32x32x4) axi_req[];
      phase.raise_objection(this);

      //Issue 4 interleaved write bursts
      axi_req = new[2];
      foreach (axi_req[i]) begin
         axi_req[i] =
sig_axi_req_vseq#(sys::axi_port_32x32x4)::type_id::create($sformatf("axi_req_%0d",i), this);
         axi_req[i].wait_done = 0;
         axi_req[i].not_last = 1;
         axi_req[i].data_len = 2; //master will only issue 2 transfers per burst
         axi_req[i].randomize() with {
            direction == WRITE;
            addr == 32'h1000_0000 * (i + 1);
            len == 15; //16 transfers per burst
            id == i;
            size == $clog2(sys::axi_port_32x32x4::AXI_DATA_WIDTH/8);
            burst == INCR;
            addr_delay == 0;
         };
         axi_req[i].start(axi_env.vsqrs[0]);
      end
```

```
//Interleave remaining 14 transfers for each burst
for(int j=0; j<14; j++) begin
   foreach (axi_req[i]) begin
      axi_req[i].wait_done = 0;
      axi_req[i].not_last = (j < 13) ? 1 : 0;
      axi_req[i].data_len = 1; //master will only issue 1 transfer at a time
      axi_req[i].just_data = 1;
      axi_req[i].start(axi_env.vsqrs[0]);
   end
end
```

Below is the resulting interleaved data.

Table 18. Interleaved write data waveform



3.10.2. AXI Response Virtual Sequence

The AXI response virtual sequence (*sig_axi_resps_vseq*) is used with the AXI slave agent to automatically issue write and read responses. This is a free-running sequence that only needs to be started once, at the start of the test. Listed below are the fields of this virtual sequence.

Table 19. AXI Response Virtual Sequence Fields

Field	Description
user_rresp_q	Queue of type *axi_data_item*. This queue gives the user control over the read data/response. If this queue is empty, the sequence uses default read response prepared by the AXI model. If this queue is not empty, the sequence pops entries from this queue and overrides the default response from the AXI model with information provided by the user.
user_bresp_q	Queue of type *axi_resp_item*. This queue gives the user control over the write response. If this queue is empty, the sequence uses default write response prepared by the AXI model. If this queue is not empty, the sequence pops entries from this queue and overrides the default response from the AXI model with information provided by the user.
slv_model	Handle of AXI model. This should be set prior to starting the sequence.
enable_interleaved	Read data interleave enable. When set to 1, the sequence will return read data in interleaved manner. Interleaving will only occur when there are multiple pending read transactions.

Shown below is a sample usage of this virtual sequence to issue default write and read responses.

```
class sig_axi_wr_rd_test extends sig_test_base;
.
.
   task run_phase(uvm_phase phase);
      sig_axi_resps_vseq#(sys::axi_port_32x32x4) axi_resp;

      axi_resp = sig_axi_resps_vseq#(sys::axi_port_32x32x4)::type_id::create("axi_resp", this);
      fork
        axi_resp.slv_model = axi_env.slv_agents[0].slv_model; //Pass AXI model to vseq
        axi_resp.start(axi_env.vsqrs[0]); //Start free-running sequence
      join_none
.
.
```

Shown below is an example showing how to use user_rresp_q and user_bresp_q to control the responses.

```
class sig_axi_error_response_test extends sig_test_base;
.
.
   task run_phase(uvm_phase phase);
      sig_axi_resps_vseq#(sys::axi_port_32x32x4) axi_resp;

      axi_resp = sig_axi_resps_vseq#(sys::axi_port_32x32x4)::type_id::create("axi_resp", this);

      //Ex #1: User wants bresp for 1st transaction to be DECERR
      write_resp = axi_resp_item_type::type_id::create("write_resp", this);
      write_resp.randomize () with {
         resp == DECERR;
         user == 'hf;
         delay == 0;
      };
      axi_resp.user_bresp_q.push_back(write_resp); //push custom write response

      //Ex #2. User wants to change/corrupt the read data
      read_resp = axi_data_item_type::type_id::create("read_resp", this);
      read_resp.randomize () with {
         data.size() == 1;
         foreach (data[i]) { data[i] == 'hffff;}
      };
      axi_resp.user_rresp_q.push_back(read_resp); //push custom read response

      fork
        axi_resp.slv_model = axi_env.slv_agents[0].slv_model; //Pass AXI model to vseq
        axi_resp.start(axi_env.vsqrs[0]); //Start free-running sequence
      join_none
.
```

3.11. Test

The AXI VIP provides sample base test and sample test cases. Listed below are tests and their descriptions.

Table 20. Tests

Test Name	Description
sig_test_base	Sample base test. This shows how to instantiate the system config, create the port configs, and register the system config into the UVM config_db.
sig_axi_wr_rd_test	This test shows how to issue write and read transactions from the master and how to enable responses from the slave.
sig_axi_burst_test	This test demonstrates how to issue bursts with different lengths, types, and sizes.
sig_axi_write_interleaved_test	This test shows how to use the request virtual sequence to create interleaved write data.
sig_axi_read_interleaved_test	This test shows how to use the response virtual sequence to create interleaved read data.
sig_axi_error_response_test	This test shows how to modify the response from the response virtual sequence to create error scenarios.
sig_axi_mst_timing_test	This test shows how configure the port config to control the timing of the master agent.
sig_axi_slv_timing_test	This test shows how to configure the port config to control the timing of slave agent.

4. Testbench Integration

To integrate the AXI VIP, users need to perform the following:

1. Import the necessary packages
2. Instantiate interfaces, connect them to clock and reset sources, register them in UVM config_db
   - The *field_name* (3rd argument) used in config_db set() function should match the *field_name* used by the master and slave agents. Refer to sections 3.3.4 and 3.4.7 for more details.
3. Connect interface pins to DUT pins

See below for an example.

```
import sys::*;
module top;

   //Instantiate interfaces
   // Write master IFs
   axi_addr_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .ADDR_WIDTH   (sys::axi_port_32x32x4::AXI_ADDR_WIDTH)
   ) axi_mstr_waddr_inf[number_of_masters-1:0](
      .clk(clk), .resetn(rsn));
   axi_data_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .DATA_WIDTH   (sys::axi_port_32x32x4::AXI_DATA_WIDTH),
      .STRB_WIDTH   (sys::axi_port_32x32x4::AXI_STRB_WIDTH)
   ) axi_mstr_wdata_inf[number_of_masters-1:0](
      .clk(clk), .resetn(rsn));
   axi_resp_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH)
   ) axi_mstr_resp_inf[number_of_masters-1:0](
      .clk(clk), .resetn(rsn));
   // Read master IFs
   axi_addr_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .ADDR_WIDTH   (sys::axi_port_32x32x4::AXI_ADDR_WIDTH)
   ) axi_mstr_raddr_inf[number_of_masters-1:0](
      .clk(clk), .resetn(rsn));
   axi_data_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .DATA_WIDTH   (sys::axi_port_32x32x4::AXI_DATA_WIDTH),
      .STRB_WIDTH   (sys::axi_port_32x32x4::AXI_STRB_WIDTH)
   ) axi_mstr_rdata_inf[number_of_masters-1:0](
      .clk(clk), .resetn(rsn));

   //Write slave Ifs
   axi_addr_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .ADDR_WIDTH   (sys::axi_port_32x32x4::AXI_ADDR_WIDTH)
   )axi_slv_waddr_inf[number_of_slaves-1:0](.clk(clk), .resetn(rsn));
   axi_data_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .DATA_WIDTH   (sys::axi_port_32x32x4::AXI_DATA_WIDTH),
      .STRB_WIDTH   (sys::axi_port_32x32x4::AXI_STRB_WIDTH)
     ) axi_slv_wdata_inf[number_of_slaves-1:0](.clk(clk),.resetn(rsn));
   axi_resp_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH)
   ) axi_slv_resp_inf[number_of_slaves-1:0](.clk(clk),.resetn(rsn));
   //READ slave IFs
   axi_addr_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .ADDR_WIDTH   (sys::axi_port_32x32x4::AXI_ADDR_WIDTH)
   ) axi_slv_raddr_inf[number_of_slaves-1:0](.clk(clk),.resetn(rsn));
   axi_data_inf #(
      .ID_WIDTH     (sys::axi_port_32x32x4::AXI_ID_WIDTH),
      .DATA_WIDTH   (sys::axi_port_32x32x4::AXI_DATA_WIDTH),
      .STRB_WIDTH   (sys::axi_port_32x32x4::AXI_STRB_WIDTH)
   ) axi_slv_rdata_inf[number_of_slaves-1:0](.clk(clk),.resetn(rsn));
```

```
//Register the interfaces into the UVM config DB
genvar i;
generate
   for (i=0; i<number_of_masters; i++) begin
      initial begin
         uvm_config_db #(sys::axi_port_32x32x4::addr_if_t)::set(null, "uvm_test_top",
             $sformatf("mstr_agent_%0d.mstr_waddr_inf", i), axi_mstr_waddr_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::addr_if_t)::set(null, "uvm_test_top",
             $sformatf("mstr_agent_%0d.mstr_raddr_inf", i), axi_mstr_raddr_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::data_if_t)::set(null, "uvm_test_top",
             $sformatf("mstr_agent_%0d.mstr_wdata_inf", i), axi_mstr_wdata_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::data_if_t)::set(null, "uvm_test_top",
             $sformatf("mstr_agent_%0d.mstr_rdata_inf", i), axi_mstr_rdata_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::resp_if_t)::set(null, "uvm_test_top",
             $sformatf("mstr_agent_%0d.mstr_resp_inf", i), axi_mstr_resp_inf[i]);
      end // initial begin
   end // for (i=0; i<number_of_masters; i++)
endgenerate

generate
   for (i=0; i<number_of_slaves; i++) begin
      initial begin
         uvm_config_db #(sys::axi_port_32x32x4::addr_if_t)::set(null, "uvm_test_top",
             $sformatf("slv_agent_%0d.slv_waddr_inf", i), axi_slv_waddr_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::addr_if_t)::set(null, "uvm_test_top",
             $sformatf("slv_agent_%0d.slv_raddr_inf", i), axi_slv_raddr_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::data_if_t)::set(null, "uvm_test_top",
             $sformatf("slv_agent_%0d.slv_wdata_inf", i), axi_slv_wdata_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::data_if_t)::set(null, "uvm_test_top",
             $sformatf("slv_agent_%0d.slv_rdata_inf", i), axi_slv_rdata_inf[i]);
         uvm_config_db #(sys::axi_port_32x32x4::resp_if_t)::set(null, "uvm_test_top",
             $sformatf("slv_agent_%0d.slv_resp_inf", i), axi_slv_resp_inf[i]);
      end // initial begin
   end // for (i=0; i<number_of_slaves; i++)
endgenerate
```

4.1. Directory Structure

```
axi_vip
  tb ------------------------------ top files, sys package
  env ----------------------------- UVM components (agents, drivers, etc.), interfaces
  seqs ---------------------------- sequence and virtual sequence
  tests --------------------------- sample test cases
  sim ----------------------------- sample Makefile and file list
```