AXI Verification IP v1.1

LogiCORE IP Product Guide

Vivado Design Suite

PG267 October 30, 2019

Table of Contents

XILINX®

Introduction

The Xilinx® LogiCORE™ AXI Verification IP (VIP) core has been developed to support the simulation of customer designed AXI-based IP. The AXI VIP core supports three versions of the AXI protocol (AXI3, AXI4, and AXI4-Lite).

The AXI VIP is unencrypted SystemVerilog source that is comprised of a SystemVerilog class library and synthesizable RTL.

The embedded RTL interface is controlled by the AXI VIP through a virtual interface. AXI transactions are constructed in the customer's verification environment and passed to the AXI driver class. The driver class then manages the timing and drives the content on the interface.

Features

- Supports all protocol data widths, address widths, transfer types, and responses
- Transaction-level protocol checking (burst type, length, size, lock type, and cache type)
- Arm®-based protocol transaction level checker for tools that support assertion property [Ref 1]
- Behavioral SystemVerilog Syntax
- SystemVerilog class-based API
- Synthesizes to nets and constant tie-offs

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale+™, UltraScale™, Zynq®-7000 SoC, 7 series FPGAs
Supported User Interfaces	AXI4, AXI4-Lite, AXI3
Resources	N/A
Provided with Core	
Design Files	SystemVerilog
Example Design	SystemVerilog
Test Bench	N/A
Constraints File	N/A
Simulation Model	Unencrypted SystemVerilog
Supported S/W Driver	N/A
Tested Design Flows[2][3]	
Design Entry	Vivado® Design Suite
Simulation[4]	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis
Support	
Release Notes and Known Issues	Master Answer Record: 68234
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:
1. For a complete list of supported devices, see the Vivado IP catalog.
2. For the supported versions of third-party tools, see the Xilinx Design Tools: Release Notes Guide.
3. This IP does not deliver VIP for Zynq PS. It only delivers the VIP core for AXI3, AXI4, and AXI4-Lite interfaces.
4. To take advantage of the full features of this IP, it requires simulators supporting advanced simulation capabilities.
5. The AXI VIP can only act as a protocol checker when contained within a VHDL hierarchy.
6. To use the virtual part of the AXI Verification IP, it must be in a Verilog hierarchy.
7. Do not import two different revisions/versions of the `axi_vip` packages. This causes elaboration failures.
8. All AXI VIP and parents to the AXI VIP must be upgraded to the latest version.

Overview

The Xilinx® LogiCORE™ AXI Verification IP (VIP) core is used in the following manner:

- Generating master AXI commands and write payload
- Generating slave AXI read payload and write responses
- Checking protocol compliance of AXI transactions

The AXI VIP can be configured in three different modes:

- AXI master VIP
- AXI slave VIP
- AXI pass-through VIP

Figure 1-1 shows the AXI master VIP which generates AXI commands and write payload and sends it to the AXI system.
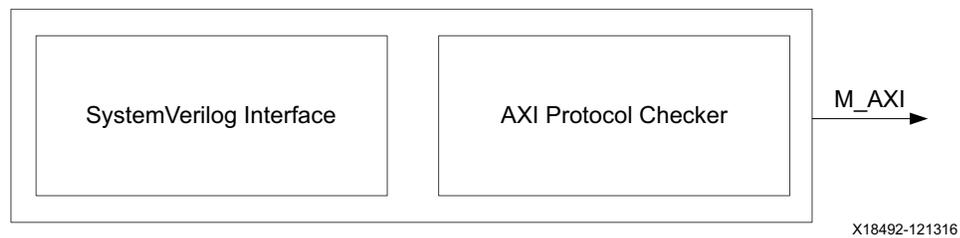
Figure 1-1: **AXI Master VIP**

Figure 1-2 shows the AXI slave VIP which responds to the AXI commands and generates read payload and write responses.
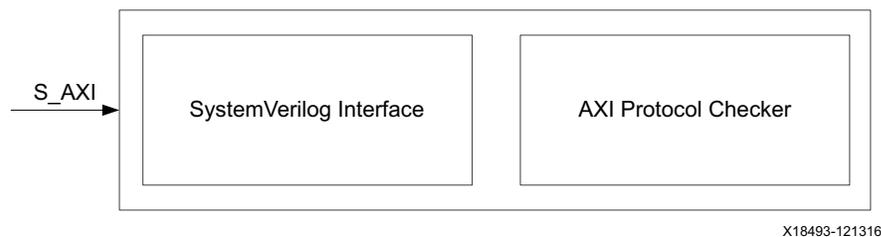
Figure 1-2: **AXI Slave VIP**

Figure 1-3 shows the AXI pass-through VIP which protocol checks all AXI transactions that pass through it. The IP can be configured to behave in the following modes:

* Monitor only

* Master

* Slave

The AXI protocol checker does not exist in the synthesized netlist.
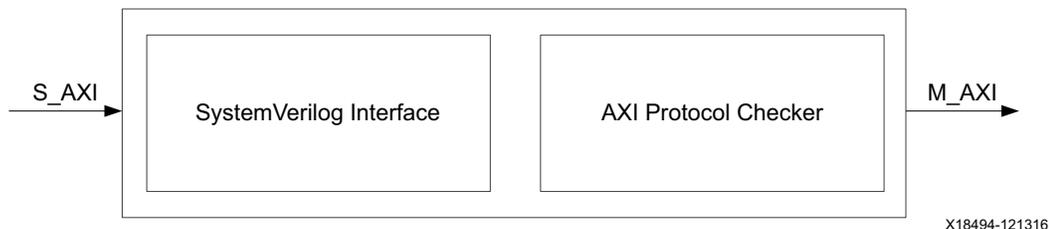
Figure 1-3: **AXI Pass-Through VIP**

IMPORTANT: *When using the Vivado*® *simulator, the AXI Protocol Checker IP [Ref 4] is used in place of the Arm AMBA Assertions.*

Feature Summary

* Supports AXI3, AXI4, or AXI4-Lite interface

* Configurable as an AXI master, AXI slave, and in pass-through mode

* Configurable simulation messaging

* Provides simulation AXI protocol checking

Applications

The AXI VIP is for verification and system engineers who want to:

* Monitor transactions between two AXI connections

* Generate AXI transactions

* Check for AXI protocol compliance

Licensing and Ordering

This Xilinx LogiCORE™ IP module is provided at no additional cost with the Xilinx Vivado Design Suite under the terms of the Xilinx End User License.

Information about other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

The AXI VIP core supports the AXI3, AXI4, and AXI4-Lite protocols.

Standards

The AXI interfaces conform to the Arm® Advanced Microcontroller Bus Architecture (AMBA®) AXI version 4 specification [Ref 2], including the AXI4-Lite control register interface subset.

Performance

The AXI VIP core synthesizes to wires and does not impact performance.

User Parameters

Table 2-1 shows the AXI VIP core user parameters.

Table 2-1: **AXI VIP User Parameters**

Parameter Name	Format/Range	Default Value	Description
PROTOCOL	Type: string Value range: AXI4, AXI3, AXI4LITE	AXI4	Used to enable protocol specific signals.
INTERFACE_MODE	Type: string Value range: PASS_THROUGH, MASTER, SLAVE	PASS_ THROUGH	Used to control the mode of protocol to be configured as master, slave, or pass-through.
READ_WRITE_MODE	Type: string Value range: READ_WRITE, READ_ONLY, WRITE_ONLY	READ_ WRITE	Used to enable the corresponding AXI read/ write signals.

Table 2-1: **AXI VIP User Parameters** *(Cont'd)*

Parameter Name	Format/Range	Default Value	Description
ADDR_WIDTH	1 ≤ integer ≤ 64 (AXI4LITE) 12 ≤ integer ≤ 64 (AXI4/AXI3)	32	Width of *_axi_{ar,aw}addr;
DATA_WIDTH	Type: long Value range: 32, 64, 128, 256, 512, 1024 AXI4LITE: 32, 64	32	Width of the *_axi_wdata and *_axi_rdata
ID_WIDTH	Type: long Value range: 0..32 AXI4LITE: 0	0	Width of the *_axi_{aw,ar,r,b}id. When the VIP is configured for AXI3 use, this parameter also controls the width of *_axi_wid.
AWUSER_WIDTH	Type: long Value range: 0..1024 AXI4LITE: 0 READ_ONLY: 0	0	Width of the *_axi_awuser
ARUSER_WIDTH	Type: long Value range: 0..1024 AXI4LITE: 0 WRITE_ONLY: 0	0	Width of the *_axi_aruser
WUSER_WIDTH	Type: long Value range: 0..1024 AXI4LITE: 0 READ_ONLY: 0	0	Width of the *_axi_wuser
BUSER_WIDTH	Type: long Value range: 0..1024 AXI4LITE: 0 READ_ONLY: 0	0	Width of the *_axi_buser
RUSER_WIDTH	Type: long Value range: 0..1024 AXI4LITE: 0 WRITE_ONLY: 0	0	Width of the *_axi_ruser
NARROW[1]	Type: long Value range: 0,1 AXI4LITE: 0	1	Specifies whether the VIP supports burst transactions with a size smaller than the native data width of the interface. When the Protocol is AXI4LITE or HAS_WSTRB is 0, SUPPORTS_NARROW must be 0.
HAS_BURST	Type: long Value range: 0,1 AXI4LITE: 0	2'b01	Used to control the enablement of the *_axi_arburst and *_axi_awburst. When it is 0, *_axi_arburst and *_axi_awburst values are 2'b01.

Table 2-1: **AXI VIP User Parameters** *(Cont'd)*

Parameter Name	Format/Range	Default Value	Description
HAS_LOCK	Type: long Value range: 0, 1 AXI4LITE: 0	1	Used to control the enablement of the *_axi_arlock and *_axi_awlock. When it is 0, *_axi_arlock and *_axi_awlock values are 0. When the Protocol is AXI4LITE, READ_WRITE_MODE is not in READ_WRITE, HAS_BRESP is 0, or HAS_RRESP is 0, HAS_LOCK must be 0.
HAS_CACHE	Type: long Value range: 0, 1 AXI4LITE: 0	1	Used to control the enablement of the *_axi_arcache and *_axi_awcache. When it is 0, *_axi_arcache and *_axi_awcache values are 0.
HAS_REGION	Type: long Value range: 0, 1 AXI4LITE/AXI3: 0	1	Used to control the enablement of the *_axi_arregion and *_axi_awregion. When it is 0, *_axi_arregion and *_axi_awregion values are 0.
HAS_PROT	Type: long Value range: 0, 1	1	Used to control the enablement of the *_axi_arprot and *_axi_awprot. When it is 0, *_axi_arprot and *_axi_awprot values are 0.
HAS_QOS	Type: long Value range: 0, 1 AXI4LITE/AXI3: 0	1	Used to control the enablement of the *_axi_arqos and *_axi_awqos. When it is 0, *_axi_arqos and *_axi_awqos values are 0.
HAS_WSTRB	Type: long Value range: 0, 1 READ_ONLY: 0	1	Used to control the enablement of the *_axi_wstrb. When HAS_WSTRB is 1, any bit in *_axi_wstrb can either be 1 or 0. When it is 0, all bits of *_axi_wstrb are 1.
HAS_BRESP	Type: long Value range: 0, 1 READ_ONLY: 0	1	Used to control the enablement of the *_axi_bresp. When it is 0, *_axi_bresp value is 0.
HAS_RRESP	Type: long Value range: {0, 1} WRITE_ONLY: 0	1	Used to control the enablement of the *_axi_rresp. When it is 0, *_axi_rresp value is 0.
HAS_ACLKEN	Type: long Value range: {0, 1}	0	Used to control the enablement of the ACLKEN port. User parameter only. When it is 0, it is treated as 1.
HAS_USER_BITS_PER_BYTE	Type: long Value range: {0, 1} AXI4LITE: 0	1	Used to control whether write/read user bits per byte is 1 or 0. 1: user bits per byte, this means ruser/wuser size are per byte based 0: user bits per transfer, this means ruser/wuser size are per transfer based

Table 2-1: **AXI VIP User Parameters** *(Cont'd)*

Parameter Name	Format/Range	Default Value	Description
WUSER_BITS_PER_BYTE	Type: long Value range: {0, 32} AXI4LITE: 0 READ_ONLY: 0	0	When HAS_USER_BITS_PER_BYTE is 1, then the value of WUSER is calculated. WUSER_WIDTH = WUSER_BITS_PER_BYTE*(DATA_WIDTH/8).
RUSER_BITS_PER_BYTE	Type: long Value range: {0, 32} AXI4LITE: 0 READ_ONLY: 0	0	When HAS_USER_BITS_PER_BYTE is 1, then the value of RUSER is calculated. RUSER_WIDTH = RUSER_BITS_PER_BYTE*(DATA_WIDTH/8).

Notes:
1. SUPPORTS_NARROW is treated as SUPPORTS_NARROW_BURST in Vivado IP integrator.

Port Descriptions

Table 2-2 shows the AXI VIP independent port descriptions.

Table 2-2: **AXI VIP Independent Port Descriptions**

Signal Name	I/O	Width	Description	Enablement
aclk	I	1	Interface clock input	
aresetn	I	1	Interface reset input (active-Low)	HAS_ARESETN == 1
aclken	I	1	Interface Clock enable signal. (active-High)	HAS_ACLKEN == 1

Table 2-3 lists the interface signals for the AXI VIP core in master or pass-through mode. The **m_axi_aw***, **m_axi_w***, and **m_axi_b*** signals are not shown on the port list when the READ_WRITE MODE parameter is READ_ONLY. The **m_axi_ar*** and **m_axi_r*** signals are not shown on the port list when the READ_WRITE MODE parameter is WRITE_ONLY. See the AXI specification for port definitions.

Table 2-3: **AXI Master or Pass-Through VIP Port Descriptions**

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
m_axi_awid	O	x	x		ID_WIDTH	Write Address Channel Transaction ID	ID_WIDTH > 0
m_axi_awaddr	O	x	x	x	ADDR_WIDTH	Write Address Channel Transaction Address (12-64)	
m_axi_awlen	O	x	x		8 for AXI4 4 for AXI3	Write Address Channel Transaction Burst Length (0-255)	
m_axi_awsize	O	x	x		3	Write Address Channel Transfer Size Code (0-7)	Always ON when it is in AXI3/AXI4

Send Feedback

Table 2-3: **AXI Master or Pass-Through VIP Port Descriptions** *(Cont'd)*

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
m_axi_awburst	O	x	x		2	Write Address Channel Burst Type Code (0-2)	HAS_BURST == 1
m_axi_awlock	O	x	x		2 for AXI4 1 for AXI3	Write Address Channel Atomic Access Type (0-1)	HAS_LOCK == 1
m_axi_awcache	O	x	x		4	Write Address Channel Cache Characteristics	HAS_CACHE == 1
m_axi_awprot	O	x	x	x	3	Write Address Channel Protection Characteristics	HAS_PROT == 1
m_axi_awqos	O	x			4	Write Address Channel Quality of Service	HAS_QOS == 1
m_axi_awregion	O	x			4	Write Address Channel Region Index	HAS_REGION== 1
m_axi_awuser	O	x			AWUSER_WIDTH	Write Address Channel User-defined signals	AWUSER_WIDTH > 0
m_axi_awvalid	O	x	x	x	1	Write Address Channel Valid	
m_axi_awready	I	x	x	x	1	Write Address Channel Ready	
m_axi_arid	O	x	x		ID_WIDTH	Read Address Channel Transaction ID	ID_WIDTH > 0
m_axi_araddr	O	x	x	x	ADDR_WIDTH	Read Address Channel Transaction Address (12-64)	
m_axi_arlen	O	x	x		8 for AXI4 4 for AXI3	Read Address Channel Transaction Burst Length (0-255)	Always ON when it is AXI3/AXI4
m_axi_arsize	O	x	x		3	Read Address Channel Transfer Size Code (0-7)	Always ON when it is in AXI3/AXI4
m_axi_arburst	O	x	x		2	Read Address Channel Burst Type Code (0-2)	HAS_BURST == 1
m_axi_arlock	O	x	x		2 for AXI4 1 for AXI3	Read Address Channel Atomic Access Type (0-1)	HAS_LOCK == 1
m_axi_arcache	O	x	x		4	Read Address Channel Cache Characteristics	HAS_CACHE == 1
m_axi_arprot	O	x	x	x	3	Read Address Channel Protection Characteristics	HAS_PROT == 1
m_axi_arqos	O	x			4	Read Address Channel Quality of Service	HAS_QOS == 1
m_axi_arregion	O	x			4	Read Address Channel Region Index	HAS_REGION == 1
m_axi_aruser	O	x			ARUSER_WIDTH	Read Address Channel User-defined signals.	AWUSER_WIDTH > 0
m_axi_arvalid	O	x	x	x	1	Read Address Channel Valid	

Table 2-3: **AXI Master or Pass-Through VIP Port Descriptions** *(Cont'd)*

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
m_axi_arready	I	x	x	x	1	Read Address Channel Ready	
m_axi_wid	O		x		ID_WIDTH		ID_WIDTH > 0
m_axi_wlast	O	x	x		1	Write Data Channel Last Data Beat	
m_axi_wdata	O	x	x	x	DATA_WIDTH	Write Data Channel Data	
m_axi_wstrb	O	x	x	x	DATA_WIDTH/8	Write Data Channel Byte Strobes	HAS_WSTRB == 1
m_axi_wuser	O	x	x		WUSER_WIDTH	Write Data Channel user-defined signal	WUSER_WIDTH > 0
m_axi_wvalid	O	x	x	x	1	Write Data Channel Valid	
m_axi_wready	I	x	x	x	1	Write Data Channel Ready	
m_axi_rid	O	x			ID_WIDTH	Read Data Channel Transaction ID	ID_WIDTH > 0
m_axi_rlast	O	x			1	Read Data Channel Last Data Beat	
m_axi_rdata	O	x		x	DATA_WIDTH	Read Data Channel Data	
m_axi_rresp	O	x		x	2	Read Data Channel Response Code (0-3)	HAS_RRESP == 1
m_axi_ruser	O	x			RUSER_WIDTH	Read Data Channel user-defined signal	RUSER_WIDTH> 0
m_axi_rvalid	O	x		x	1	Read Data Channel Valid	
m_axi_rready	I	x		x	1	Read Data Channel Ready	
m_axi_bid	O	x			ID_WIDTH	Write Response Channel Transaction ID	ID_WIDTH > 0
m_axi_bresp	O	x		x	2	Write Response Channel Response Code (0-3)	HAS_BRESP > 0
m_axi_buser	O	x			BUSER_WIDTH	Write Response Channel user-defined signal	BUSER_WIDTH> 0
m_axi_bvalid	O	x		x	1	Write Response Channel Valid	
m_axi_bready	I	x		x	1	Write Response Channel Ready	

Send Feedback

Table 2-4 lists the interface signals for the AXI VIP core when it has been configured to be in slave or pass-through mode.

Table 2-4: **AXI Slave or Pass-Through VIP Port Descriptions**

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
s_axi_awid	I	x	x		ID_WIDTH	Write Address Channel Transaction ID	ID_WIDTH > 0
s_axi_awaddr	I	x	x	x	ADDR_WIDTH	Write Address Channel Transaction Address (12-64)	
s_axi_awlen	I	x	x		8 for AXI4 4 for AXI3	Write Address Channel Transaction Burst Length (0-255)	
s_axi_awsize	I	x	x		3	Write Address Channel Transfer Size Code (0-7)	Always ON when it is in AXI3/AXI4
s_axi_awburst	I	x	x		2	Write Address Channel Burst Type Code (0-2)	HAS_BURST == 1
s_axi_awlock	I	x	x		2 for AXI4 1 for AXI3	Write Address Channel Atomic Access Type (0-1)	HAS_LOCK == 1
s_axi_awcache	I	x	x		4	Write Address Channel Cache Characteristics	HAS_CACHE == 1
s_axi_awprot	I	x	x	x	3	Write Address Channel Protection Characteristics	HAS_PROT == 1
s_axi_awqos	I	x			4	Write Address Channel Quality of Service	HAS_QOS == 1
s_axi_awregion	I	x			4	Write Address Channel Region Index	HAS_REGION== 1
s_axi_awuser	I	x			AWUSER_WIDTH	Write Address Channel User-defined signals	AWUSER_WIDTH> 0
s_axi_awvalid	O	x	x	x	1	Write Address Channel Valid	
s_axi_awready	I	x	x	x	1	Write Address Channel Ready	
s_axi_arid	I	x	x		ID_WIDTH	Read Address Channel Transaction ID	ID_WIDTH >0
s_axi_araddr	I	x	x	x	ADDR_WIDTH	Read Address Channel Transaction Address (12-64)	
s_axi_arlen	I	x	x		8 for AXI4 4 for AXI3	Read Address Channel Transaction Burst Length (0-255)	
s_axi_arsize	I	x	x		3	Read Address Channel Transfer Size Code (0-7)	Always ON when it is in AXI3/AXI4
s_axi_arburst	I	x	x		2	Read Address Channel Burst Type Code (0-2)	HAS_BURST == 1
s_axi_arlock	I	x	x		2 for AXI4 1 for AXI3	Read Address Channel Atomic Access Type (0-1)	HAS_LOCK == 1

Send Feedback

Table 2-4: **AXI Slave or Pass-Through VIP Port Descriptions** *(Cont'd)*

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
s_axi_arcache	I	x	x		4	Read Address Channel Cache Characteristics	HAS_CACHE == 1
s_axi_arprot	I	x	x	x	3	Read Address Channel Protection Characteristics	HAS_PROT == 1
s_axi_arqos	I	x			4	Read Address Channel Quality of Service	HAS_QOS == 1
s_axi_arregion	I	x			4	Read Address Channel Region Index	HAS_REGION == 1
s_axi_aruser	I	x			ARUSER_WIDTH	Read Address Channel User-defined signals	AWUSER_WIDTH> 0
s_axi_arvalid	O	x	x	x	1	Read Address Channel Valid	
s_axi_arready	I	x	x	x	1	Read Address Channel Ready	
s_axi_wid	I		x				ID_WIDTH
s_axi_wlast	I	x	x		1	Write Data Channel Last Data Beat	
s_axi_wdata	I	x	x	x	DATA_WIDTH	Write Data Channel Data	
s_axi_wstrb	I	x	x	x	DATA_WIDTH/8	Write Data Channel Byte Strobes	HAS_WSTRB == 1
s_axi_wuser	I	x	x		WUSER_WIDTH	Write Data Channel User-defined signal	WUSER_WIDTH > 0
s_axi_wvalid	O	x	x	x	1	Write Data Channel Valid	
s_axi_wready	I	x	x	x	1	Write Data Channel Ready	
s_axi_rid	I	x	x		ID_WIDTH	Read Data Channel Transaction ID	ID_WIDTH > 0
s_axi_rlast	I	x	x		1	Read Data Channel Last Data Beat	
s_axi_rdata	I	x	x	x	DATA_WIDTH	Read Data Channel Data	
s_axi_rresp	I	x	x	x	2	Read Data Channel Response Code (0-3)	HAS_RRESP == 1
s_axi_ruser	I	x	x		RUSER_WIDTH	Read Data Channel User-defined signal	RUSER_WIDTH > 0
s_axi_rvalid	O	x	x	x	1	Read Data Channel Valid	
s_axi_rready	I	x	x	x	1	Read Data Channel Ready	
s_axi_bid	I	x	x		ID_WIDTH	Write Response Channel Transaction ID	ID_WIDTH > 0
s_axi_bresp	I	x	x	x	2	Write Response Channel Response Code (0-3)	HAS_BRESP > 0

Send Feedback

Table 2-4: **AXI Slave or Pass-Through VIP Port Descriptions** *(Cont'd)*

Signal Name	I/O	AXI4	AXI3	AXI4-LITE	Width	Description	Enablement
s_axi_buser	I	x	x		BUSER_WIDTH	Write Response Channel User-defined signal	BUSER_WIDTH > 0
s_axi_bvalid	O	x	x	x	1	Write Response Channel Valid	
s_axi_bready	I	x	x	x	1	Write Response Channel Ready	

AXI Protocol Checks and Descriptions

Table 2-5 lists the AXI protocol checks and descriptions which are essentially the same as the assertions that are found in the *AXI Protocol Checker LogiCORE IP Product Guide* (PG101) [Ref 4].

Table 2-5: **AXI Protocol Checks and Descriptions**

Name of Protocol Check	Protocol Support	Description
AXI_ERRM_AWADDR_BOUNDARY	AXI4/AXI3	A write burst cannot cross a 4 KB boundary.
AXI_ERRM_AWADDR_WRAP_ALIGN	AXI4/AXI3	A write transaction with burst type WRAP has an aligned address.
AXI_ERRM_AWBURST	AXI4/AXI3	A value of 2'b11 on AWBURST is not permitted when AWVALID is High.
AXI_ERRM_AWLEN_LOCK	AXI4/AXI3	Exclusive access transactions cannot have a length greater than 16 beats.
AXI_ERRM_AWCACHE	AXI4/AXI3	If not cacheable (AWCACHE[1] == 1'b0), AWCACHE = 2'b00.
AXI_ERRM_AWLEN_FIXED	AXI4/AXI3	Transactions of burst type FIXED cannot have a length greater than 16 beats.
AXI_ERRM_AWLEN_WRAP	AXI4/AXI3	A write transaction with burst type WRAP has a length of 2, 4, 8, or 16.
AXI_ERRM_AWSIZE	AXI4/AXI3	The size of a write transfer does not exceed the width of the data interface.
AXI_ERRM_AWVALID_RESET	AXI4/AXI3/Lite	AWVALID is Low for the first cycle after ARESETn goes High.
AXI_ERRM_AWADDR_STABLE	AXI4/AXI3/Lite	Handshake Checks AWADDR must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWBURST_STABLE	AXI4/AXI3	Handshake Checks AWBURST must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWCACHE_STABLE	AXI4/AXI3	Handshake Checks AWCACHE must remain stable when AWVALID is asserted and AWREADY Low.

Table 2-5: **AXI Protocol Checks and Descriptions** *(Cont'd)*

Name of Protocol Check	Protocol Support	Description
AXI_ERRM_AWID_STABLE	AXI4/AXI3	Handshake Checks AWID must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWLEN_STABLE	AXI4/AXI3	Handshake Checks AWLEN must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWLOCK_STABLE	AXI4/AXI3	Handshake Checks AWLOCK must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWPROT_STABLE	AXI4/AXI3/Lite	Handshake Checks AWPROT must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWSIZE_STABLE	AXI4/AXI3	Handshake Checks AWSIZE must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWQOS_STABLE	AXI4/AXI3	Handshake Checks AWQOS must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_AWREGION_STABLE	AXI4	Handshake Checks AWREGION must remain stable when ARVALID is asserted and AWREADY Low.
AXI_ERRM_AWVALID_STABLE	AXI4/AXI3/Lite	Handshake Checks Once AWVALID is asserted, it must remain asserted until AWREADY is High.
AXI_RECS_AWREADY_MAX_WAIT	AXI4/AXI3/Lite	Recommended that AWREADY is asserted within MAXWAITS cycles of AWVALID being asserted.
AXI_ERRM_WDATA_NUM	AXI4/AXI3	The number of write data items matches AWLEN for the corresponding address. This is triggered when any of the following occurs: • Write data arrives and WLAST is set, and the WDATA count is not equal to AWLEN • Write data arrives and WLAST is not set, and the WDATA count is equal to AWLEN • ADDR arrives, WLAST is already received, and the WDATA count is not equal to AWLEN
AXI_ERRM_WSTRB	AXI4/AXI3/Lite	Write strobes must only be asserted for the correct byte lanes as determined from the: Start Address, Transfer Size, and Beat Number.
AXI_ERRM_WVALID_RESET	AXI4/AXI3/Lite	WVALID is Low for the first cycle after ARESETn goes High.
AXI_ERRM_WDATA_STABLE	AXI4/AXI3/Lite	Handshake Checks WDATA must remain stable when WVALID is asserted and WREADY Low.
AXI_ERRM_WLAST_STABLE	AXI4/AXI3	Handshake Checks WLAST must remain stable when WVALID is asserted and WREADY Low.
AXI_ERRM_WSTRB_STABLE	AXI4/AXI3/Lite	Handshake Checks WSTRB must remain stable when WVALID is asserted and WREADY Low.
AXI_ERRM_WVALID_STABLE	AXI4/AXI3/Lite	Handshake Checks Once WVALID is asserted, it must remain asserted until WREADY is High.
AXI_RECS_WREADY_MAX_WAIT	AXI4/AXI3/Lite	Recommended that WREADY is asserted within MAXWAITS cycles of WVALID being asserted.

Send Feedback

Table 2-5: **AXI Protocol Checks and Descriptions** *(Cont'd)*

Name of Protocol Check	Protocol Support	Description
AXI_ERRS_BRESP_WLAST	AXI4/AXI3	A slave must not take BVALID High until after the last write data is handshake is complete.
AXI_ERRS_BRESP_EXOKAY	AXI4/AXI3	An EXOKAY write response can only be given to an exclusive write access.
AXI_ERRS_BVALID_RESET	AXI4/AXI3/Lite	BVALID is Low for the first cycle after ARESETn goes High.
AXI_ERRS_BRESP_AW	AXI4/AXI3/Lite	A slave must not take BVALID High until after the write address is handshake is complete.
AXI_ERRS_BID_STABLE	AXI4/AXI3	Handshake Checks BID must remain stable when BVALID is asserted and BREADY Low.
AXI_ERRS_BRESP_STABLE	AXI4/AXI3/Lite	Checks BRESP must remain stable when BVALID is asserted and BREADY Low.
AXI_ERRS_BVALID_STABLE	AXI4/AXI3/Lite	Once BVALID is asserted, it must remain asserted until BREADY is High.
AXI_RECM_BREADY_MAX_WAIT	AXI4/AXI3/Lite	Recommended that BREADY is asserted within MAXWAITS cycles of BVALID being asserted.
AXI_ERRM_ARADDR_BOUNDARY	AXI4/AXI3	A read burst cannot cross a 4 KB boundary.
AXI_ERRM_ARADDR_WRAP_ALIGN	AXI4/AXI3	A read transaction with a burst type of WRAP must have an aligned address.
AXI_ERRM_ARBURST	AXI4/AXI3	A value of 2'b11 on ARBURST is not permitted when ARVALID is High.
AXI_ERRM_ARLEN_LOCK	AXI4/AXI3	Exclusive access transactions cannot have a length greater than 16 beats.
AXI_ERRM_ARCACHE	AXI4/AXI3	When ARVALID is High, if ARCACHE[1] is Low, then ARCACHE[3:2] must also be Low.
AXI_ERRM_ARLEN_FIXED	AXI4/AXI3	Transactions of burst type FIXED cannot have a length greater than 16 beats.
AXI_ERRM_ARLEN_WRAP	AXI4/AXI3	A read transaction with burst type of WRAP must have a length of 2, 4, 8, or 16.
AXI_ERRM_ARSIZE	AXI4/AXI3	The size of a read transfer must not exceed the width of the data interface.
AXI_ERRM_ARVALID_RESET	AXI4/AXI3/Lite	ARVALID is Low for the first cycle after ARESETn goes High.
AXI_ERRM_ARADDR_STABLE	AXI4/AXI3/Lite	ARADDR must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARBURST_STABLE	AXI4/AXI3	ARBURST must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARCACHE_STABLE	AXI4/AXI3	ARCACHE must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARID_STABLE	AXI4/AXI3	ARID must remain stable when ARVALID is asserted and ARREADY Low.

Table 2-5: **AXI Protocol Checks and Descriptions** *(Cont'd)*

Name of Protocol Check	Protocol Support	Description
AXI_ERRM_ARLEN_STABLE	AXI4/AXI3	ARLEN must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARLOCK_STABLE	AXI4/AXI3	ARLOCK must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARPROT_STABLE	AXI4/AXI3/Lite	ARPROT must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARSIZE_STABLE	AXI4/AXI3	ARSIZE must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARQOS_STABLE	AXI4/AXI3	ARQOS must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARREGION_STABLE	AXI4	ARREGION must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRM_ARVALID_STABLE	AXI4/AXI3/Lite	Once ARVALID is asserted, it must remain asserted until ARREADY is High.
AXI_RECS_ARREADY_MAX_WAIT	AXI4/AXI3/Lite	Recommended that ARREADY is asserted within MAXWAITS cycles of ARVALID being asserted.
AXI_ERRS_RDATA_NUM	AXI4/AXI3	The number of read data items must match the corresponding ARLEN.
AXI_ERRS_RID	AXI4/AXI3	The read data must always follow the address that it relates to. Therefore, a slave can only give read data with an ID to match an outstanding read transaction.
AXI_ERRS_RRESP_EXOKAY	AXI4/AXI3	An EXOKAY read response can only be given to an exclusive read access.
AXI_ERRS_RVALID_RESET	AXI4/AXI3/Lite	RVALID is Low for the first cycle after ARESETn goes High.
AXI_ERRS_RDATA_STABLE	AXI4/AXI3/Lite	RDATA must remain stable when RVALID is asserted and RREADY Low.
AXI_ERRS_RID_STABLE	AXI4/AXI3	RID must remain stable when RVALID is asserted and RREADY Low.
AXI_ERRS_RLAST_STABLE	AXI4/AXI3	RLAST must remain stable when RVALID is asserted and RREADY Low.
AXI_ERRS_RRESP_STABLE	AXI4/AXI3/Lite	RRESP must remain stable when RVALID is asserted and RREADY Low.
AXI_ERRS_RVALID_STABLE	AXI4/AXI3/Lite	Once RVALID is asserted, it must remain asserted until RREADY is High.
AXI_RECM_RREADY_MAX_WAIT	AXI4/AXI3/Lite	Recommended that RREADY is asserted within MAXWAITS cycles of RVALID being asserted.
AXI_ERRM_EXCL_ALIGN	AXI4/AXI3	The address of an exclusive access is aligned to the total number of bytes in the transaction.
AXI_ERRM_EXCL_LEN	AXI4/AXI3	The number of bytes to be transferred in an exclusive access burst is a power of 2, that is, 1, 2, 4, 8, 16, 32, 64, or 128 bytes.

Send Feedback

Table 2-5: **AXI Protocol Checks and Descriptions** *(Cont'd)*

Name of Protocol Check	Protocol Support	Description
AXI_RECM_EXCL_MATCH	AXI4/AXI3	Recommended that the address, size, and length of an exclusive write with a given ID is the same as the address, size, and length of the preceding exclusive read with the same ID.
AXI_ERRM_EXCL_MAX	AXI4/AXI3	128 is the maximum number of bytes that can be transferred in an exclusive burst.
AXI_RECM_EXCL_PAIR	AXI4/AXI3	Recommended that every exclusive write has an earlier outstanding exclusive read with the same ID.
AXI_ERRM_AWUSER_STABLE	AXI4/AXI3	AWUSER must remain stable when AWVALID is asserted and AWREADY Low.
AXI_ERRM_WUSER_STABLE	AXI4/AXI3	WUSER must remain stable when WVALID is asserted and WREADY Low.
AXI_ERRS_BUSER_STABLE	AXI4/AXI3	BUSER must remain stable when BVALID is asserted and BREADY Low.
AXI_ERRM_ARUSER_STABLE	AXI4/AXI3	ARUSER must remain stable when ARVALID is asserted and ARREADY Low.
AXI_ERRS_RUSER_STABLE	AXI4/AXI3	RUSER must remain stable when RVALID is asserted and RREADY Low.
AXI_AUXM_RCAM_OVERFLOW	AXI4/AXI3/Lite	Read CAM overflow, increase MAXRBURSTS parameter.
AXI_AUXM_RCAM_UNDERFLOW	AXI4/AXI3/Lite	Read CAM Underflow
AXI_AUXM_WCAM_OVERFLOW	AXI4/AXI3/Lite	Write CAM overflow, increase MAXWBURSTS parameter.
AXI_AUXM_WCAM_UNDERFLOW	AXI4/AXI3/Lite	Write CAM Underflow
AXI_AUXM_EXCL_OVERFLOW	AXI4/AXI3	Exclusive access monitor overflow, increase EXMON_WIDTH parameter.
AXI4LITE_ERRS_BRESP_EXOKAY	Lite	A slave must not give an EXOKAY response on an AXI4-Lite interface.
AXI4LITE_ERRS_RRESP_EXOKAY	Lite	A slave must not give an EXOKAY response on an AXI4-Lite interface.
AXI4LITE_AUXM_DATA_WIDTH	Lite	DATA_WIDTH parameter is 32 or 64.

Xilinx Configuration Checks and Descriptions

Table 2-6 lists the Xilinx configuration checks and descriptions.

Table 2-6: **Xilinx Configuration Checks and Descriptions**

Name of Protocol Check	Protocol Support	Description
XILINX_AW_SUPPORTS_NARROW_BURST	AXI4/AXI3	When the connection does not support narrow transfers, the AW Master cannot issue a transfer with AWLEN > 0 and AWSIZE less than the defined interface DATA_WIDTH.
XILINX_AR_SUPPORTS_NARROW_BURST	AXI4/AXI3	When the connection does not support narrow transfers, the AR Master cannot issue a transfer with ARLEN > 0 and ARSIZE less than the defined interface DATA_WIDTH.
XILINX_AW_SUPPORTS_NARROW_CACHE	AXI4/AXI3	When the connection does not support narrow transfers, the AW Master cannot issue a transfer with AWLEN > 0 and AWCACHE modifiable bit not asserted.
XILINX_AR_SUPPORTS_NARROW_CACHE	AXI4/AXI3	When the connection does not support narrow transfers, the AR Master cannot issue a transfer with ARLEN > 0 and ARCACHE modifiable bit not asserted.
XILINX_AW_MAX_BURST	AXI4/AXI3	AW Master cannot issue AWLEN greater than the configured maximum burst length.
XILINX_AR_MAX_BURST	AXI4/AXI3	AR Master cannot issue ARLEN greater than the configured maximum burst length.
XILINX_AWVALID_RESET	AXI4/AXI3/Lite	AWREADY is Low for the first cycle after ARESETn goes High.
XILINX _WVALID_RESET	AXI4/AXI3/Lite	WREADY is Low for the first cycle after ARESETn goes High.
XILINX _BVALID_RESET	AXI4/AXI3/Lite	BREADY is Low for the first cycle after ARESETn goes High.
XILINX _ARVALID_RESET	AXI4/AXI3/Lite	ARREADY is Low for the first cycle after ARESETn goes High.
XILINX _RVALID_RESET	AXI4/AXI3/Lite	RREADY is Low for the first cycle after ARESETn goes High.
ARESET_XCHECK	AXI4/AXI3/Lite	AREST_N cannot be X/Z after one clock cycle.
XILINX_AXI_ERRM_RESET_PULSE_WIDTH	AXI4/AXI3/Lite	ARESETN must stay at least 16 clock cycles long when it goes Low. For more information, see the *Vivado Design Suite User Guide: AXI Reference Guide* (UG1037) [Ref 3].

Designing with the Core

This chapter includes guidelines and additional information to facilitate designing with the core.

General Design Guidelines

The AXI VIP core should be inserted into a system as shown in Figure 3-1 for AXI master VIP, Figure 3-2 for AXI slave VIP, and Figure 3-3 for AXI pass-through VIP.
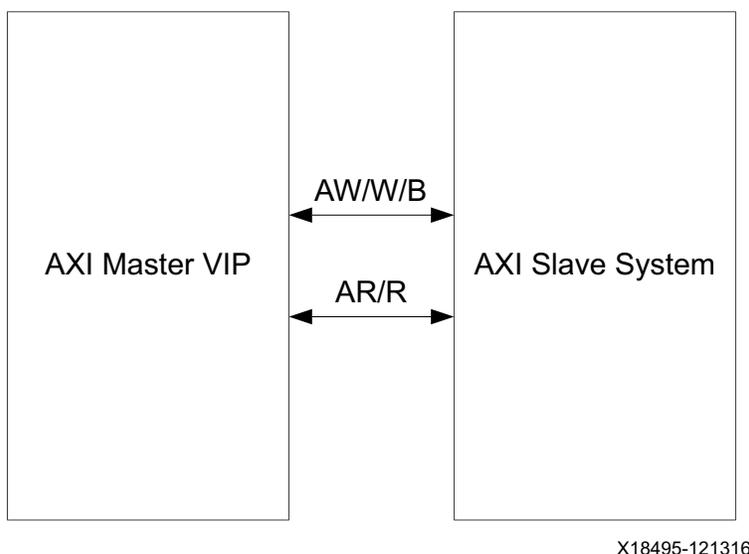

X18495-121316

Figure 3-1: **AXI Master VIP Example Topology**
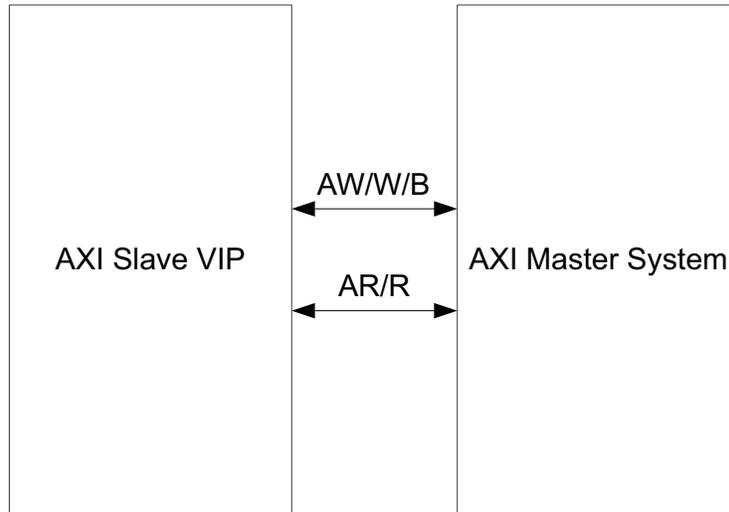
X18496-121316

Figure 3-2: **AXI Slave VIP Example Topology**
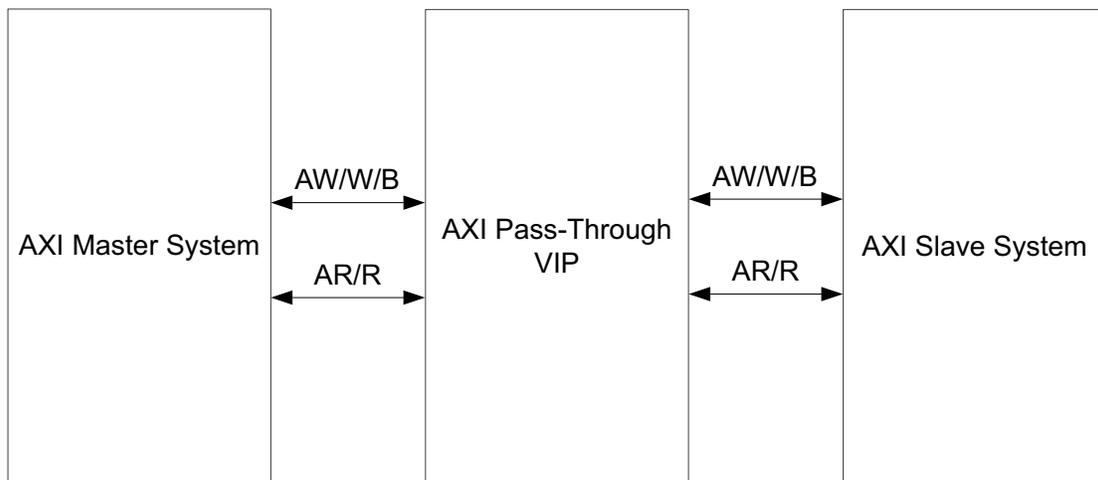
X18497-121316

Figure 3-3: **AXI Pass-Through VIP Example Topology**

Clocking

This section is not applicable for this IP core.

Resets

The AXI VIP requires one active-Low reset, `aresetn`. The reset is synchronous to `aclk`. The reset is optional based on `HAS_ARESETN`.

Send Feedback

Design Flow Steps

This chapter describes customizing and generating the core, constraining the core, and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

• *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5]

• *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6]

• *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7]

• *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8]

Customizing and Generating the Core

This section includes information about using Xilinx tools to customize and generate the core in the Vivado Design Suite.

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5] for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the Vivado IP catalog.

2. Double-click the selected IP or select the **Customize IP** command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7].

Note: Figures in this chapter are an illustration of the Vivado Integrated Design Environment (IDE). The layout depicted here might vary from the current version.

Figure 4-1 shows the AXI VIP Vivado IDE **Basic Settings** tab configuration screen.
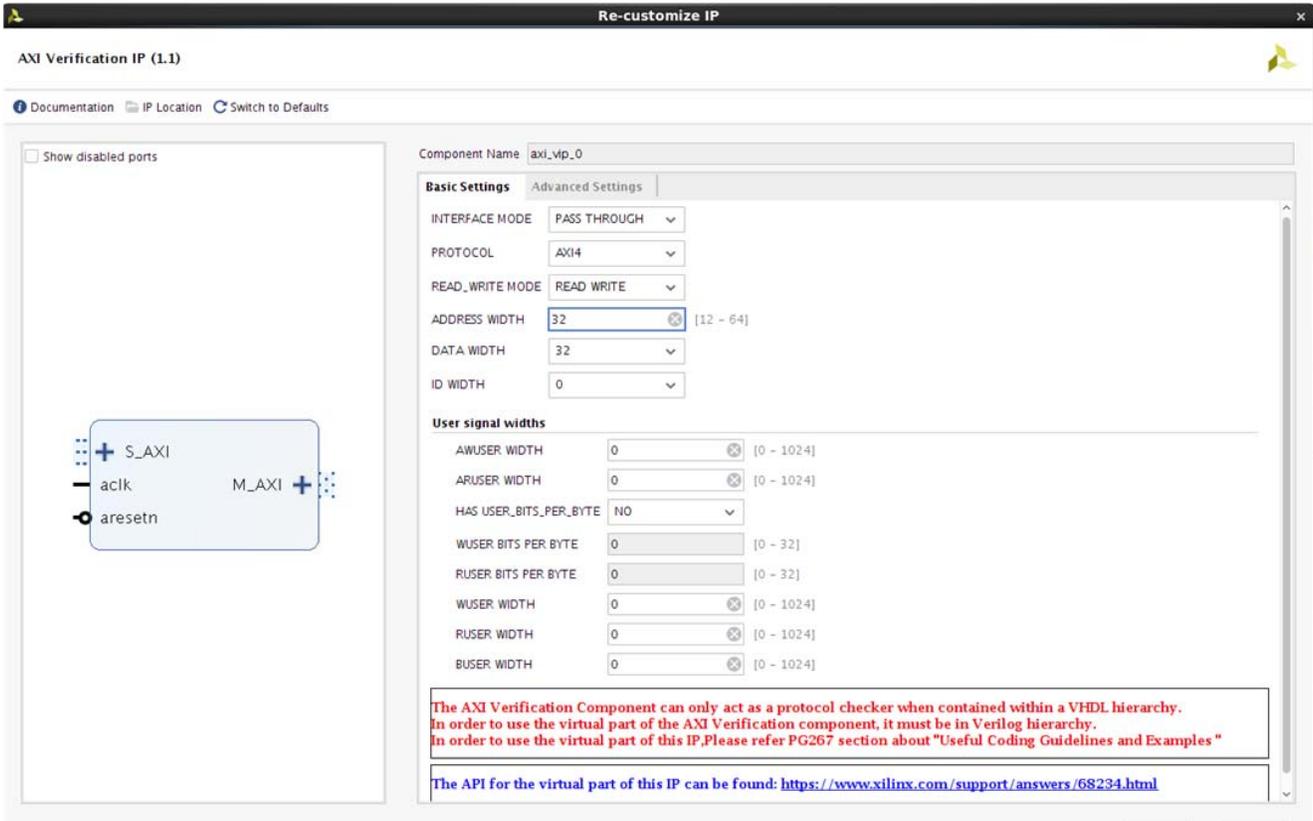
Figure 4-1: **AXI VIP Customize IP – Basic Settings Tab**

For the runtime parameter descriptions, see Table 2-1.

- **Component Name** – The component name is used as the base name of output files generated for the module. Names must begin with a letter and must be composed from characters: a to z, 0 to 9 and "_".

- **Interface Mode** – Controls the mode of protocol to be configured as master, slave, or pass-through.

- **Protocol** – Selects the specific AXI protocol.

- **Read or Write Mode** – Enables the AXI read or write mode.

- **Address Width** – Selects the address width. Default at 32.

- **Data Width** – Selects the data width. Default at 32.

- **ID Width** – Selects the ID width. Default at 0.

- **User Signal Widths** – Selects the width for each user signal. Default at 0.

Send Feedback

Figure 4-2 shows the AXI VIP Vivado IDE **Advanced Settings** tab configuration screen. For more information on the user parameters, see Table 2-1.
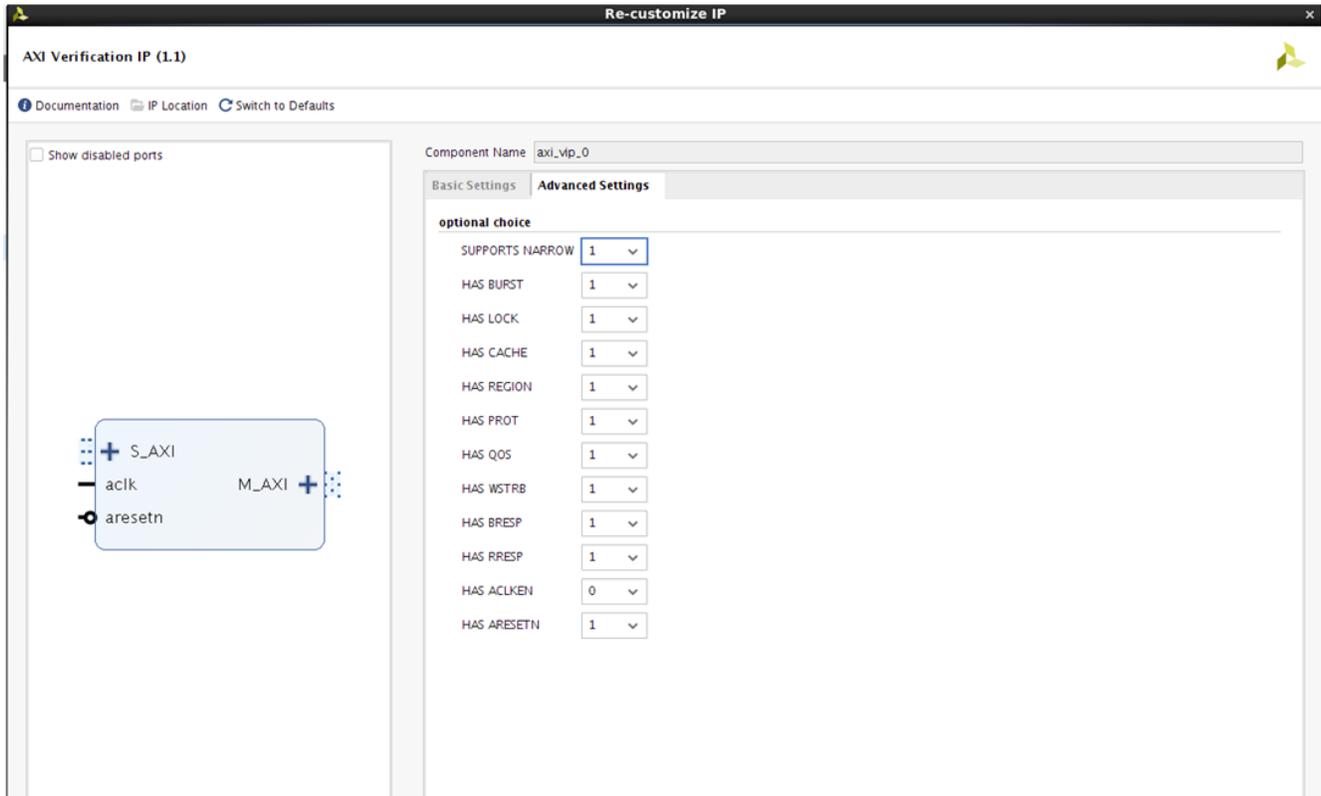
Figure 4-2:   **AXI VIP Customize IP – Advanced Settings Tab**

User Parameters

For the relationship between the fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl Console), see Table 2-1.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

The AXI VIP deliverables are organized in the directory `<project_name>/<project_name>.srcs/sources_1/ip/<component_name>` and are designated as the `<ip_source_dir>`. The relevant contents or directories are described in the following sections.

Vivado Design Tools Project Files

The Vivado design tools project files are located in the root of the `<ip_source_dir>`.

Table 4-1: **Vivado Design Tools Project Files**

Name	Description
<component_name>.xci	Vivado tools IP configuration options file. This file can be imported into any Vivado tools design and be used to generate all other IP source files.
<component_name>.{veo|vho}	AXI VIP instantiation template.

IP Sources

The IP sources are held in the subdirectories of the `<ip_source_dir>`.

Table 4-2: **IP Sources**

Name	Description
hdl/*.sv	AXI VIP source files.
synth/<component_name>.sv	AXI VIP generated top-level file for synthesis. Optional, generated if synthesis target selected.
sim/<component_name>.sv	AXI VIP generated top-level file for simulation. Optional, generated if simulation target selected.

AXI VIP in Vivado IP Integrator

This section contains information about how to use the AXI VIP in a design and test bench environment. Figure 4-3 shows a possible design with the AXI VIPs.

X18577-121316
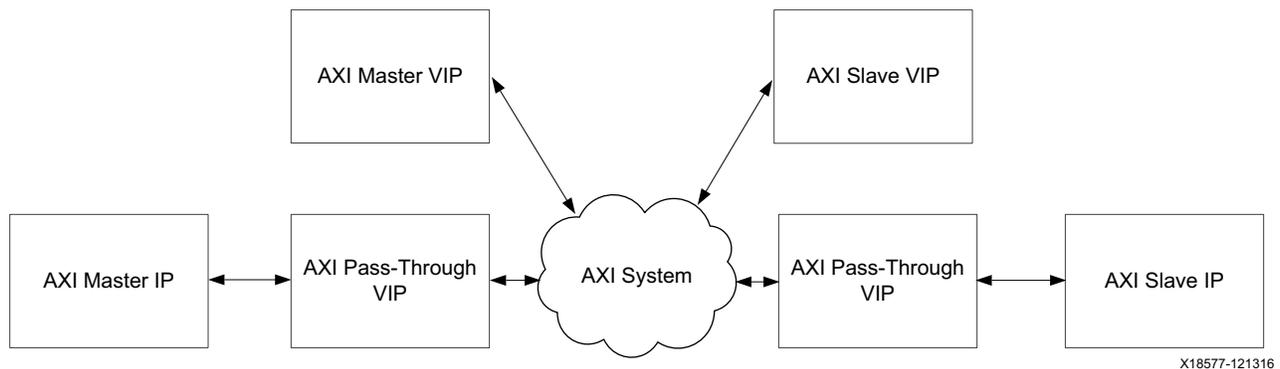
Figure 4-3: **AXI VIP Design**

The AXI VIP uses similar naming and structures as the Universal Verification Methodology (UVM) for core design. It is coded in SystemVerilog. The AXI VIP is comprised of two parts.

One is instanced like other traditional IP (modules in the static/physical world) and the second part is used in the dynamic world in your verification environment. The AXI VIP is an IP which has a static world connected to the dynamic world with a virtual interface. The virtual interface is the only mechanism that can bridge the dynamic world of objects with the static world of modules and interfaces.

AXI Master VIP

Figure 4-4 shows the AXI master VIP with its test bench. The test bench has three parts:

- User environment

- Master agent

- AXI master VIP

The user environment and master agent are in the dynamic world while the AXI master VIP is in the static world. The user environment communicates with the master agent and the master agent communicates with the AXI VIP interface though a virtual interface. The master agent has four class members:

- Master write driver

- Master read driver

- Monitor

- Virtual interface

For more information about the master agent, see Appendix D, AXI VIP Agent and Flow Methodology.

Send Feedback
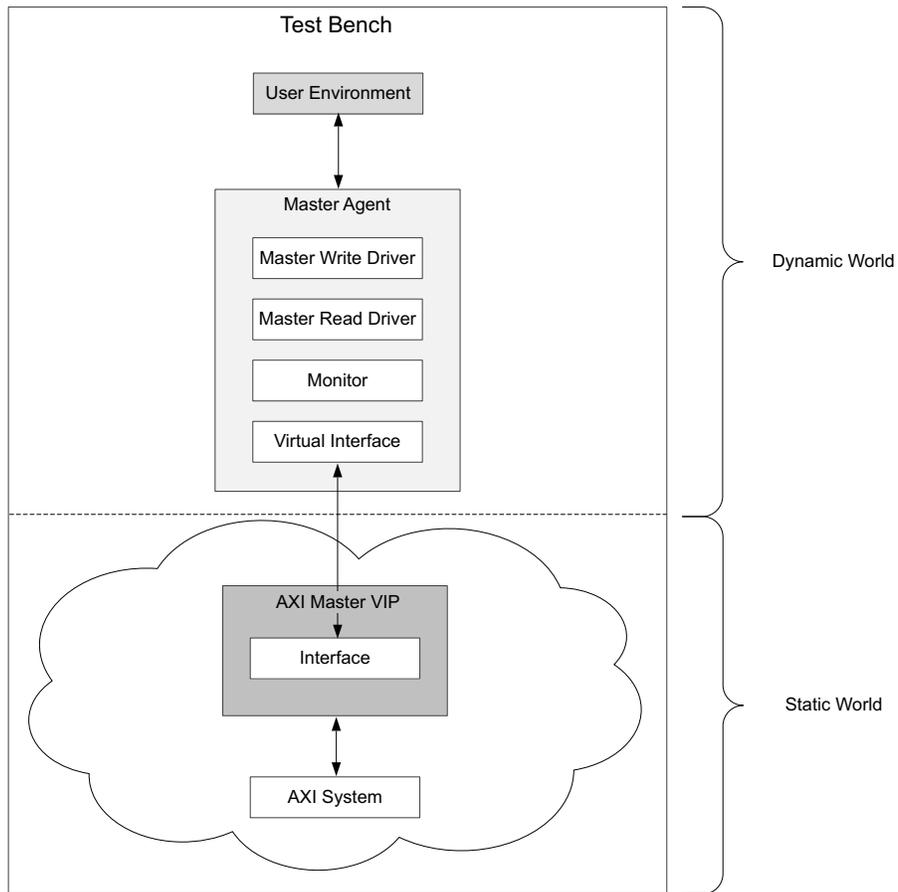
Figure 4-4: **AXI VIP Master Test Bench**

Figure 4-5 shows how a write transaction is constructed and sent to the AXI VIP interface. The user environment first declares a variable of the transaction type and then requests the Master Write Driver for a handle to a new transaction. During the `create_transaction`, the Master Write Driver sets properties on the transaction based on the physical configuration of the AXI VIP.

Using the handle, the user environment sets up the members of the transaction by either filling in using access methods or randomization. Once the transaction has been configured, the transaction is passed back to the Master Write Driver which sends it to the AXI VIP through a virtual interface and the AXI VIP interface pins start to wiggle. The read transaction flow follows a similar process, except using the Master Read Driver as the source of the handle to the transaction.

Send Feedback
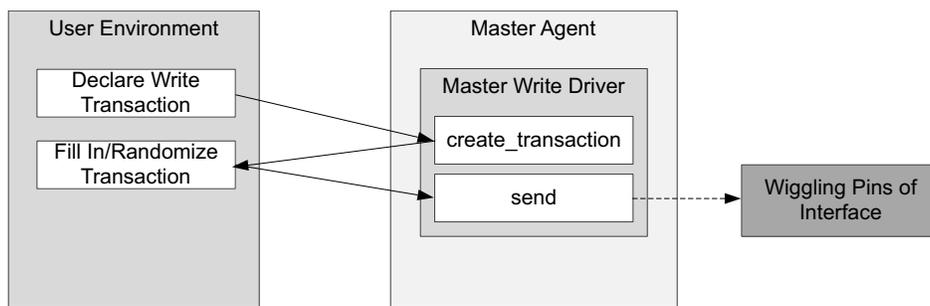
Figure 4-5:   **Write Transaction Flow**

AXI Slave VIP

Figure 4-6 shows the AXI slave VIP with its test bench. The test bench has three parts:

- User environment

- Slave agent without a memory model

- AXI slave VIP

The user environment and slave agent are in the dynamic world, while the AXI slave VIP is in the static world. The user environment communicates with the slave agent and the slave agent communicates with the AXI VIP interface though a virtual interface. The slave agent without a memory model has four class members.

- Slave write driver

- Slave read driver

- Monitor

- Virtual interface

For more information about the slave agent, see Appendix D, AXI VIP Agent and Flow Methodology.
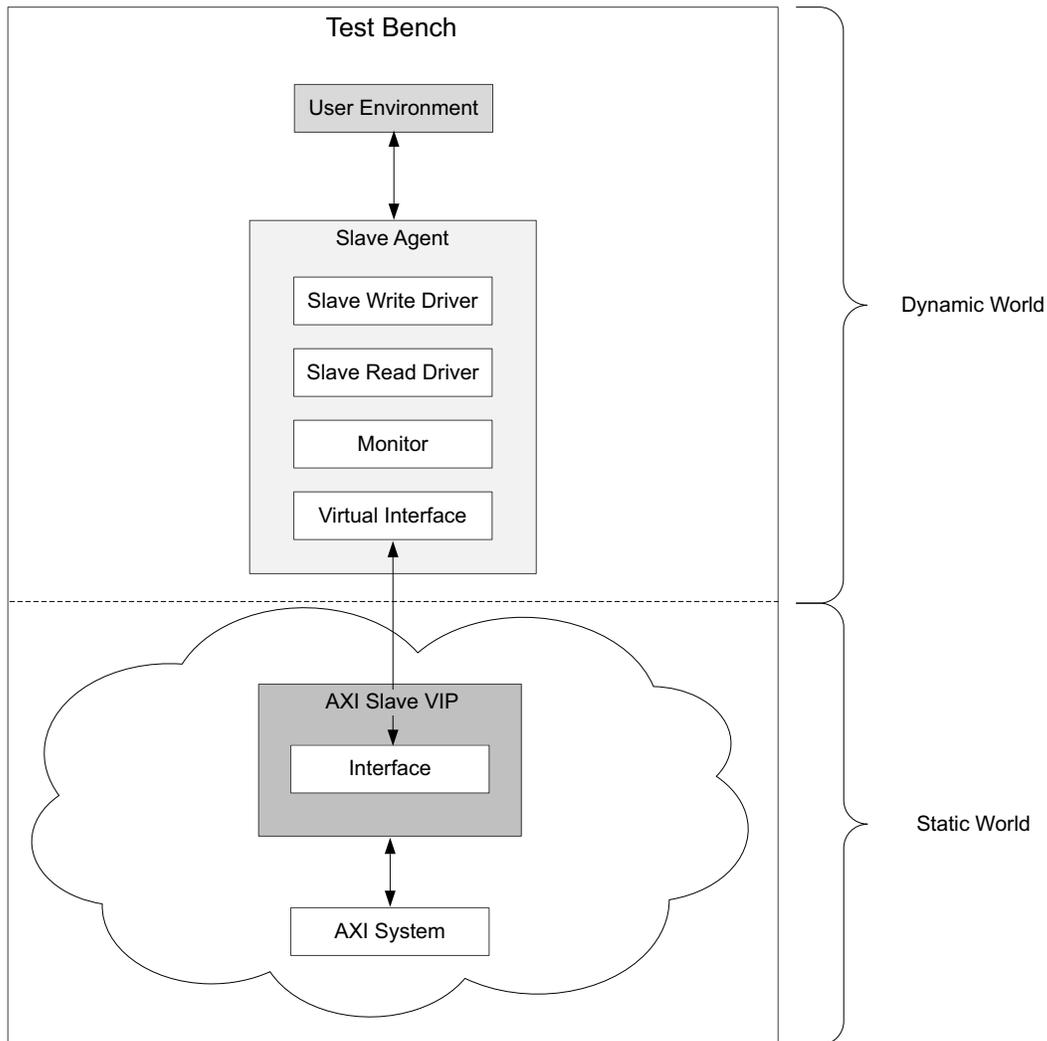
Figure 4-6: **AXI VIP Slave Test Bench**

Figure 4-7 shows how a write response transaction is constructed and sent to the AXI VIP interface. The user environment first declares a variable of the transaction type which is used by the user environment to manage the transaction. The user environment then calls **get_wr_reactive** which blocks until a write transaction has been received by the Slave Write Driver.

The Slave Write Driver waits until it receives a write command, and then it returns the handle to the transaction. The user environment fills in the response portion of the transaction by either randomization or member functions of the transaction class. The Slave Write Driver then sends it to the AXI VIP interface through a virtual interface and the AXI VIP interface related pins start to wiggle. The read response transaction flow follows a similar process.
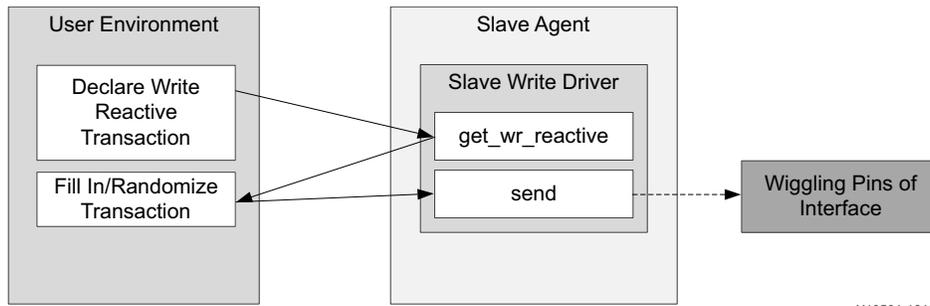
X18581-121316

Figure 4-7: **Write Transaction Flow**

Simple SRAM Memory Model

The AXI slave VIP has a simple memory model (see Figure 4-8) and it is an associative array of SystemVerilog. The write transaction can write to the memory model and the read transaction can read data from the memory (also called front door access to differ from the backdoor access APIs). These two features are implemented in the AXI slave VIP and AXI pass-through VIP in runtime slave mode. At the same time, the memory model has backdoor APIs for you to access memory directly, which are **backdoor_memory_write** and **backdoor_memory_read**. The **backdoor_memory_write** writes data to memory and **backdoor_memory_read** reads data from memory. For usage of memory model APIs, see the *AXI VIP API Documentation* [Ref 12].
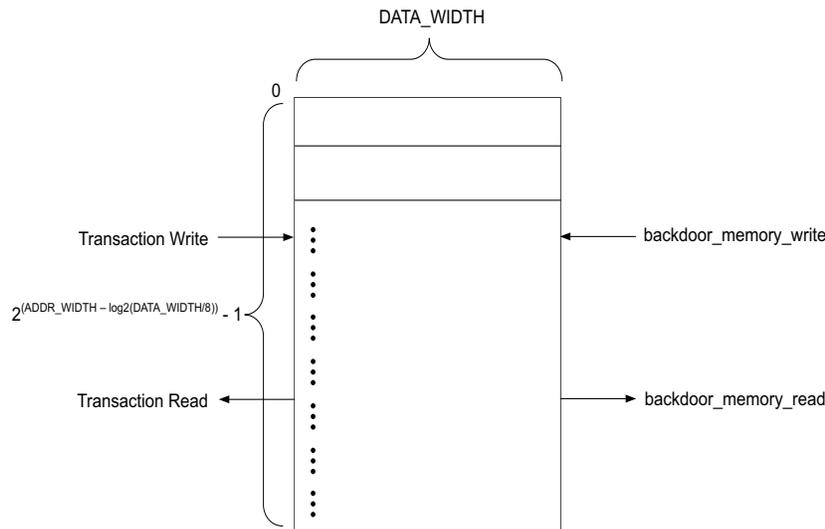
X18984-092718

Figure 4-8: **Memory Model**

Send Feedback

IMPORTANT: *This memory has no support for built-in system tasks such as* `readmemh`. *You can use the* `backdoor_memory_write` *to write all of the file information into the memory. Reset has no effect on memory content.*

AXI Slave Simple Memory VIP

As shown in Figure 4-6, the AXI slave simple memory VIP is similar to the AXI slave VIP. The only difference is that the AXI slave simple memory VIP has a simple memory model. The user environment does not need to fill in read/write reactive transaction information. Its test bench has three parts:

- User environment

- Slave agent with a memory model

- AXI slave VIP

The user environment and slave agent are in the dynamic world, while the AXI slave VIP is in the static world. The user environment communicates with the slave agent and the slave agent communicates with the AXI VIP interface though a virtual interface. The slave agent with a memory model has five class members:

- Slave write driver

- Slave read driver

- Monitor

- Virtual interface

- Memory model

For more information about the slave agent with memory model, see Appendix D, AXI VIP Agent and Flow Methodology.

Different from the AXI slave VIP, the AXI slave simple memory VIP does not need the user environment to create and fill in write/read reactive transaction since all these are being done in the slave memory agent. Refer to the Multiple Simulation Sets in Chapter 6 for its usage.

Multiple AXI VIP

Figure 4-9 shows multiple AXI VIPs in one design and its test bench. Similar to the single AXI VIP, it has dynamic and static worlds that are bridged through a virtual interface.

Send Feedback
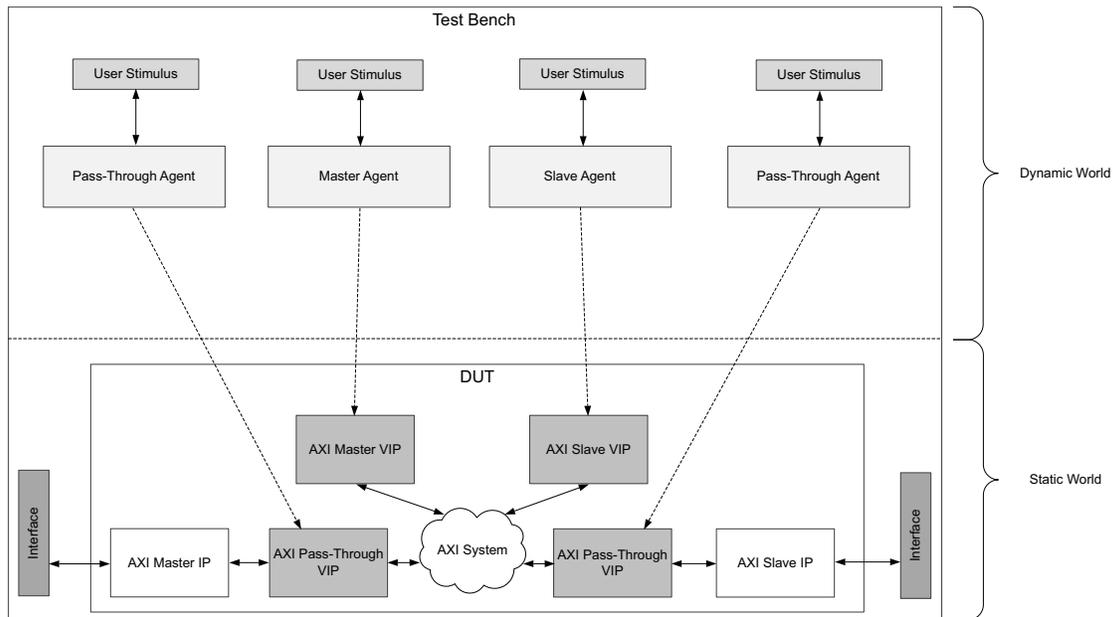
Figure 4-9: **Multiple AXI VIP Test Bench**

Finding the AXI VIP Hierarchy Path in IP Integrator

As mentioned earlier, the user environment has to declare the agent for the AXI VIP. Also, the AXI VIP interface has to be passed to the agent when the user environment constructs it to set it as a virtual interface. The following guidelines describe how to find the hierarchy path of the AXI VIP in the IP integrator.

1. Create a **bd design** and add the VIP like other IPs into the design.

2. After connection and validation checks for the IP integrator design, click the **Simulation Settings**, set up the tool, and then click **Run Simulation**. Figure 4-10 shows the Mentor Graphics Questa Advanced Simulator results. After the hierarchy is identified, it is used in the SystemVerilog test bench to drive the AXI VIP APIs.

Figure 4-10: **AXI VIP Instance in IP Integrator Design Hierarchy**

After the AXI VIP is instantiated in the IP integrator design and its hierarchy path found, the next step is using the AXI VIP in the test bench. See Chapter 5, Example Design.

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

This section is not applicable for this IP core.

Clock Management

This section is not applicable for this IP core.

Send Feedback

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8].

IMPORTANT: *For cores targeting 7 series or Zynq-7000 devices, UNIFAST libraries are not supported. Xilinx IP is tested and qualified with UNISIM libraries only.*

Synthesis and Implementation

The AXI VIP core is a verification IP set to synthesize as wires. The `axi_protocol_checker` contained in the AXI VIP is for simulation only and does not synthesize. There is no implementation for the AXI VIP.

Example Design

This chapter contains information about the example design provided in the Vivado®
Design Suite.

> **IMPORTANT:** *The example design of this IP is customized to the IP configuration. The intent of this
> example design is to demonstrate how to use the AXI VIP. The AXI VIP can only act as a protocol
> checker when contained within a VHDL hierarchy. Do not import two different revision/versions of the*
> `axi_vip` *packages. This will cause elaboration failures. All AXI VIP and parents to the AXI VIP must be
> upgraded to the latest version.*

Overview

Figure 5-1 shows the AXI VIP example design.
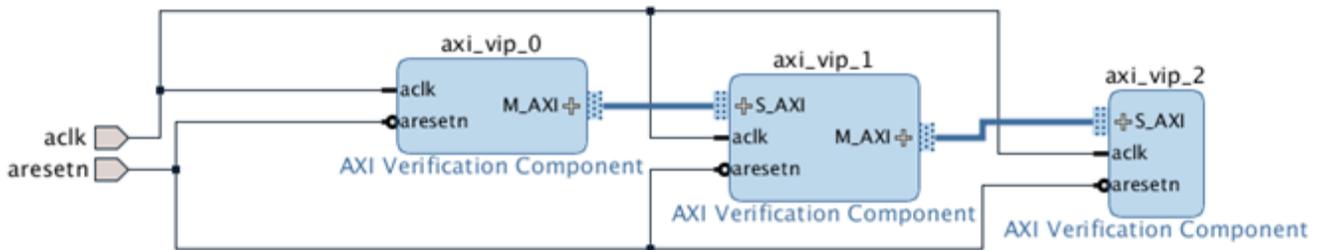
Figure 5-1: **AXI VIP Example Design**

This section describes the example tests used to demonstrate the abilities of the AXI VIP
core. Example tests are delivered in SystemVerilog.

When the core example design is open, the example files are delivered in a standard path
test bench and **bd** design are under directory imports. The packages are under the directory
example.srcs/sources_1/bd/ex_sim/ipshared.

The example design consists of three components:

* AXI VIP in master mode
* AXI VIP in pass-through mode

- AXI VIP in slave mode

The AXI master VIP creates write/read transactions and sends them to the AXI pass-through VIP. The AXI pass-through VIP receives transactions from the AXI master VIP and sends them to the AXI slave VIP. The AXI slave VIP receives transactions from the AXI pass-through VIP, generates write/read responses, and sends the responses back to the AXI pass-through VIP and then back to AXI master VIP.

Monitors for the AXI VIP (master, pass-through, and slave) are always on and collect all of the information from the interfaces. The monitors convert the interface information back to the transaction level and sends it to the scoreboard. Two scoreboards are built in the test bench which performs self-checking for the AXI master VIP against the AXI pass-through VIP and the AXI slave VIP against the AXI pass-through VIP.

The AXI VIP core is not fully autonomous. If tests are written using the APIs, there are different methods from the user environment to set up the transaction. It is possible that the AXI protocol can be accidentally violated. The member functions of the transaction class performs quick protocol and configuration checks. Xilinx recommends the use of transaction randomization and constraints for generating generic transactions. Furthermore, it is possible to further modify a transaction after it was originally generated through a randomization.

When the AXI VIP is configured in pass-through mode, it has the ability to be a passive monitor or it can take control of the interface. The AXI VIP can change to be a master driving a downstream slave or a slave responding to the master. This process can be done during a simulation at any time and then changed back to pass-through mode according to your preference.

When it is switched to runtime master mode, it behaves exactly as an AXI master VIP. When it is switched to runtime slave mode, it behaves exactly as an AXI slave VIP.

IMPORTANT: *Ensure all transactions have completed before switching modes. Examples of how to wait for the transactions to finish can be found in the example design.*

Test Bench

This chapter contains information about the test bench for the example design provided in the Vivado® Design Suite.

To open the example design either from the Vivado IP catalog or Vivado IP integrator design, follow these steps:

1. Open a new project and click **IP Catalog**.

2. Search for **AXI Verification IP**. Double-click it, configure the IP, and generate the IP.

3. Right-click the IP and choose **Open IP Example Design**.

   Note: If you have the AXI VIP as one component in the IP integrator design, right-click AXI VIP and click **Open IP Example Design...**

In both scenarios, a new project with the example design is created. The example design has the master, pass-through, and slave VIP connected directly to each other as shown in Figure 5-1. The configuration of the example design matches the original VIP configuration.

Multiple Simulation Sets

The Vivado Design Suite has a feature that each design can have multiple simulation sets according to your settings (AR: 64111). Especially, multiple test benches can be constructed for the same design. For example, one test bench provides stimulus for behavioral simulation of a complicated design and a different test bench provides stimulus for timing simulation of the implemented design.

The AXI VIP example design in the 2018.3 release has simulation sets listed in Table 6-1. The `sim_all_config` is a comprehensive simulation set. See the AXI VIP Example Test Bench and Test section for a list of features. It shows different examples of how to use the AXI VIP in a complex method.

For ease of use, 10 additional simulation sets with simple codes are also included in the example design. Naming of these 10 simulation sets:

> `sim_<basic` or `adv>_mst_<mode>__pt_<mode>__slv_<mode>`

Where `mode` = active, mem, passive, or combo.

Send Feedback

- For Master VIP, it can be in active or passive mode.

  ◦ In active mode, it generates transactions and sends it out.

  ◦ In passive mode, pass-through VIP is in run-time master mode while master VIP is not active.

- For Slave VIP, it can be in mem, passive, or combo mode.

  ◦ In mem mode, it means slave VIP has a memory model.

  ◦ In passive mode, pass-through VIP is in run-time slave mode while slave VIP is not active.

  ◦ In combo mode, it means slave VIP does not have a memory model.

- For Pass-through VIP, it can be in mst, slv, passive, or mem mode.

  ◦ In mst mode, pass-through VIP is in run-time master mode.

  ◦ In slv mode, pass-through VIP is in run-time slave mode without a memory model.

  ◦ In passive mode, pass-through VIP is not active.

  ◦ In mem mode, pass-through VIP is in run-time slave mode with a memory model.

For more information, see Table 6-1.

For example, `sim_basic_mst_passive__pt_mst__slv_comb` means this simulation set is performing a basic feature of the AXI VIP when the AXI master VIP is in passive mode. The AXI pass-through VIP is in the runtime master mode and communicates to the AXI slave VIP which does not include the memory model.

For the 10 simulation set test benches, it includes three files which are `generic_tb`, `master stimulus`, and `slave stimulus`.

- The `generic_tb` performs a simple self-checking of the master side (can be AXI master VIP or AXI pass-through VIP in runtime master mode) against the slave side (can be AXI slave VIP or AXI pass-through VIP in runtime slave mode).

- Master stimulus is generated by the AXI master VIP or AXI pass-through VIP in the runtime master mode.

- Slave stimulus is generated by the AXI slave VIP or AXI pass-through VIP in the runtime slave mode with/without the memory model. The slave mem stimulus file is included and you can access the file to check the AXI slave VIP with the memory model.

The difference between basic and advanced simulation sets is that the basic simulation set shows the code snippets which are needed in the test bench to use the AXI VIP. Advanced simulation set adds more APIs such as user-configured READY signals which are optional. For more information, see the example design in the Vivado Design Suite and for usage of APIs, see the *AXI VIP API Documentation* [Ref 12].

The following shows how to generate a transaction for each mode:

Send Feedback

1. To generate a transaction for the AXI master VIP, see the `mst_stimulus.sv` from any of the 10 simulation sets in Table 6-1.

2. To generate a transaction response for a basic AXI slave VIP, see the `slv_basic_stimulus.sv` from any of the 10 simulation sets in Table 6-1. For memory model requirements, see the `mem_basic_stimulus.sv`.

3. To generate a transaction response for an advanced AXI slave VIP, see the `slv_stimulus.sv` from any of the 10 simulation sets in Table 6-1. For memory model requirements, see the `mem_stimulus.sv`.

4. To generate a transaction for the AXI pass-through VIP, see the `passthrough_mst_stimulus.sv` from any of the simulation sets in Table 6-1.

5. To generate a transaction response for a basic AXI pass-through VIP, see the `passthrough_slv_basic_stimulus.sv` from any of the simulation sets in Table 6-1. For memory model requirements, see the `passthrough_mem_basic_stimulus.sv`.

6. To generate a transaction response for an advanced AXI pass-through VIP, see the `passthrough_slv_stimulus.sv` from any of the simulation sets in Table 6-1. For memory model requirements, see the `passthrough_mem_stimulus.sv`.

7. When you open the AXI VIP example design under the **Sources** window, it shows 11 simulation sets. You can choose any simulation sets and run simulation or view the source codes of each test bench.

Table 6-1: **Simulation Sets for AXI VIP**

| Simulation Set Name	Files Included	Description
sim_basic_mst_passive_pt_mst_slv_comb	passthrough_mst_stimulus.sv slv_basic_stimulus.sv generic_tb.sv	Basic feature of AXI VIP when AXI master VIP is in passive mode. AXI pass-through VIP is in runtime master mode and talks to AXI slave VIP which does not include memory model.
sim_basic_mst_active_pt_slv_slv_passive	mst_stimulus.sv passthrough_slv_basic_stimulus.sv generic_tb.sv	Basic feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in runtime slave mode without memory model. AXI slave VIP is in passive mode.
sim_basic_mst_active_pt_mem_slv_passive	mst_stimulus.sv passthrough_mem_basic_stimulus.sv generic_tb.sv	Basic feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in runtime slave mode with memory model. AXI slave VIP in passive mode.
sim_basic_mst_active_pt_passive_slv_mem	mst_stimulus.sv mem_basic_stimulus.sv generic_tb.sv	Basic feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in passive mode. AXI slave VIP is in active mode with memory model.

Table 6-1: **Simulation Sets for AXI VIP** *(Cont'd)*

Simulation Set Name	Files Included	Description
sim_basic_mst_active_pt_passive_slv_comb	mst_stimulus.sv slv_basic_stimulus.sv generic_tb.sv	Basic feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in passive mode AXI slave VIP is in active mode without memory model.
sim_adv_mst_passive_pt_mst_slv_comb	passthrough_mst_stimulus.sv slv_stimulus.sv generic_tb.sv	Advanced feature of AXI VIP when AXI master VIP is in passive mode. AXI pass-through VIP is in runtime master mode and talks to AXI slave VIP which does not include memory model.
sim_adv_mst_active_pt_slv_slv_passive	mst_stimulus.sv passthrough_slv_stimulus.sv generic_tb.sv	Advanced feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in runtime slave mode without memory model. AXI slave VIP is in passive mode.
sim_adv_mst_active_pt_mem_slv_passive	mst_stimulus.sv passthrough_mem_stimulus.sv generic_tb.sv	Advanced feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in runtime slave mode with memory model. AXI slave VIP in passive mode.
sim_adv_mst_active_pt_passive_slv_mem	mst_stimulus.sv mem_stimulus.sv generic_tb.sv	Advanced feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in passive mode. AXI slave VIP is in active mode with memory model.
sim_adv_mst_active_pt_passive_slv_comb	mst_stimulus.sv slv_stimulus.sv generic_tb.sv	Advanced feature of AXI VIP when AXI master VIP is in active mode. AXI pass-through VIP is in passive mode. AXI slave VIP is in active mode without memory model.
sim_all_config		Shows all of the configured examples.
sim_ready_gen		Usage about ready signal generation with different ready policy and randomization policy.
sim_memory		Usage about memory model such as backdoor_memory_write/read and pre_load_mem.

Send Feedback

Figure 6-1 to Figure 6-3 show the basic, advanced, and all configured simulation sets for AXI VIP.

sim_basic_mst_passive_pt_mst_slv_comb

Testbench

| Master VIP (Passive) | Pass-Through VIP (Runtime Master) | Slave VIP (Without Memory Model) |

passthrough_mst_stimulus.sv

slv_basic_stimulus.sv

sim_basic_mst_active_pt_slv_slv_passive

Testbench

| Master VIP | Pass-Through VIP (Runtime Slave without Memory Model) | Slave VIP (Passive) |

mst_stimulus.sv

passthrough_slv_basic_stimulus.sv

sim_basic_mst_active_pt_mem_slv_passive

Testbench

| Master VIP | Pass-Through VIP (Runtime Slave with Memory Model) | Slave VIP (Passive) |

mst_stimulus.sv

passthrough_mem_basic_stimulus.sv

sim_basic_mst_active_pt_passive_slv_mem

Testbench

| Master VIP | Pass-Through VIP (Passive) | Slave VIP (With Memory Model) |

mst_stimulus.sv

mem_basic_stimulus.sv

sim_basic_mst_active_pt_passive_slv_comb

Testbench

| Master VIP | Pass-Through VIP (Passive) | Slave VIP (Without Memory Model) |

mst_stimulus.sv

slv_basic_stimulus.sv

X18985-033017

Figure 6-1: **Basic Simulation Sets**

sim_adv_mst_passive_pt_mst_slv_comb

sim_adv_mst_active_pt_slv_slv_passive

sim_adv_mst_active_pt_mem_slv_passive

sim_adv_mst_active_pt_passive_slv_mem

sim_adv_mst_active_pt_passive_slv_comb
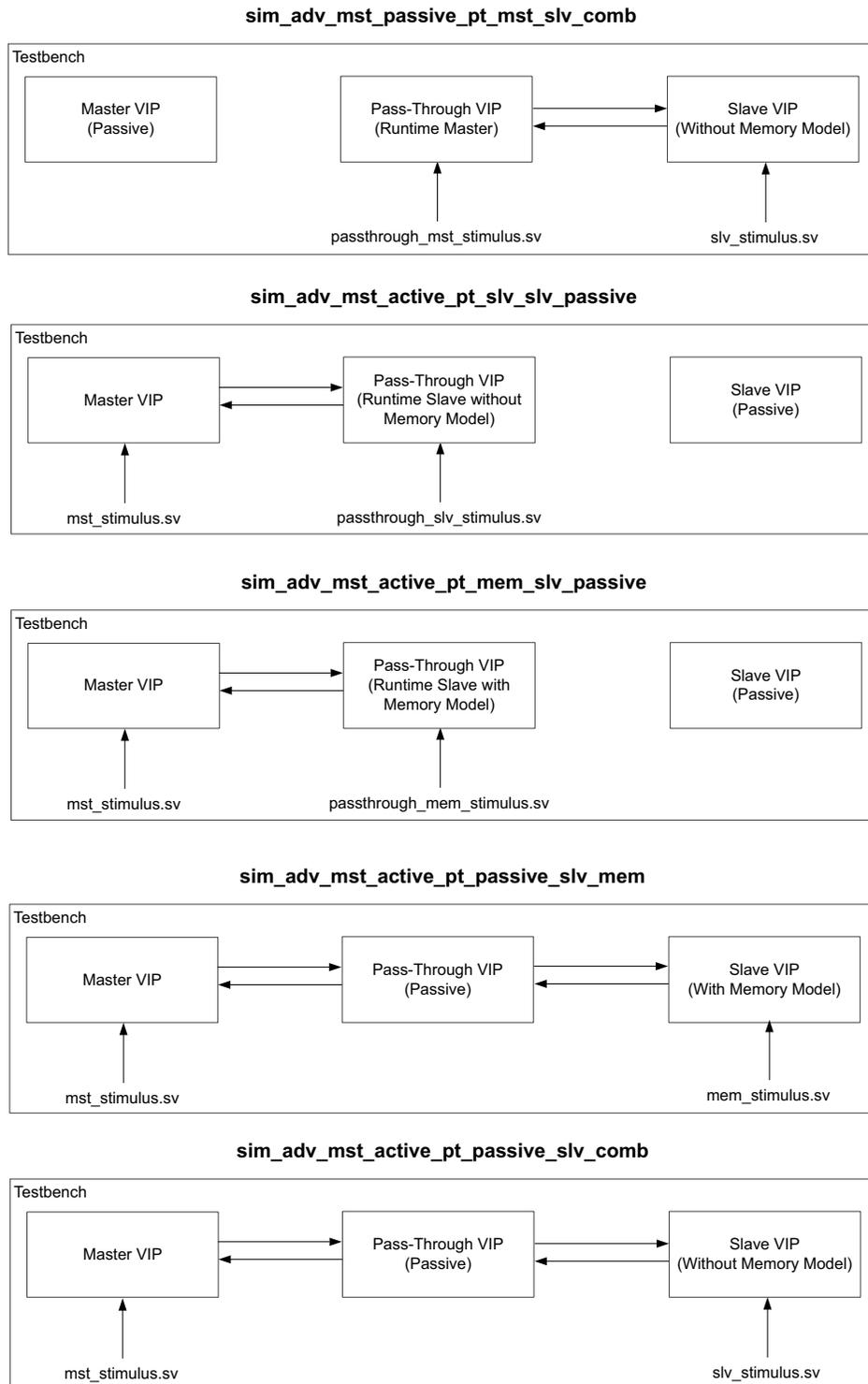
X18986-033017

Figure 6-2: **Advanced Simulation Sets**

sim_all_config, sim_ready_gen, sim_memory
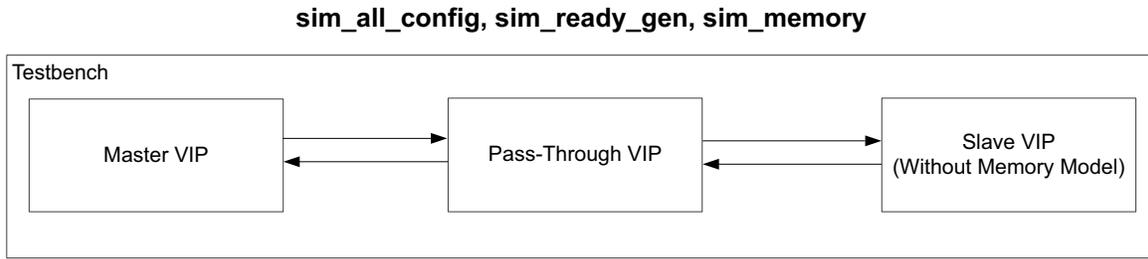
X18988-050719

Figure 6-3: **All Configured Simulation Set**

Figure 6-4 shows a simulation configuration example in Vivado IDE.
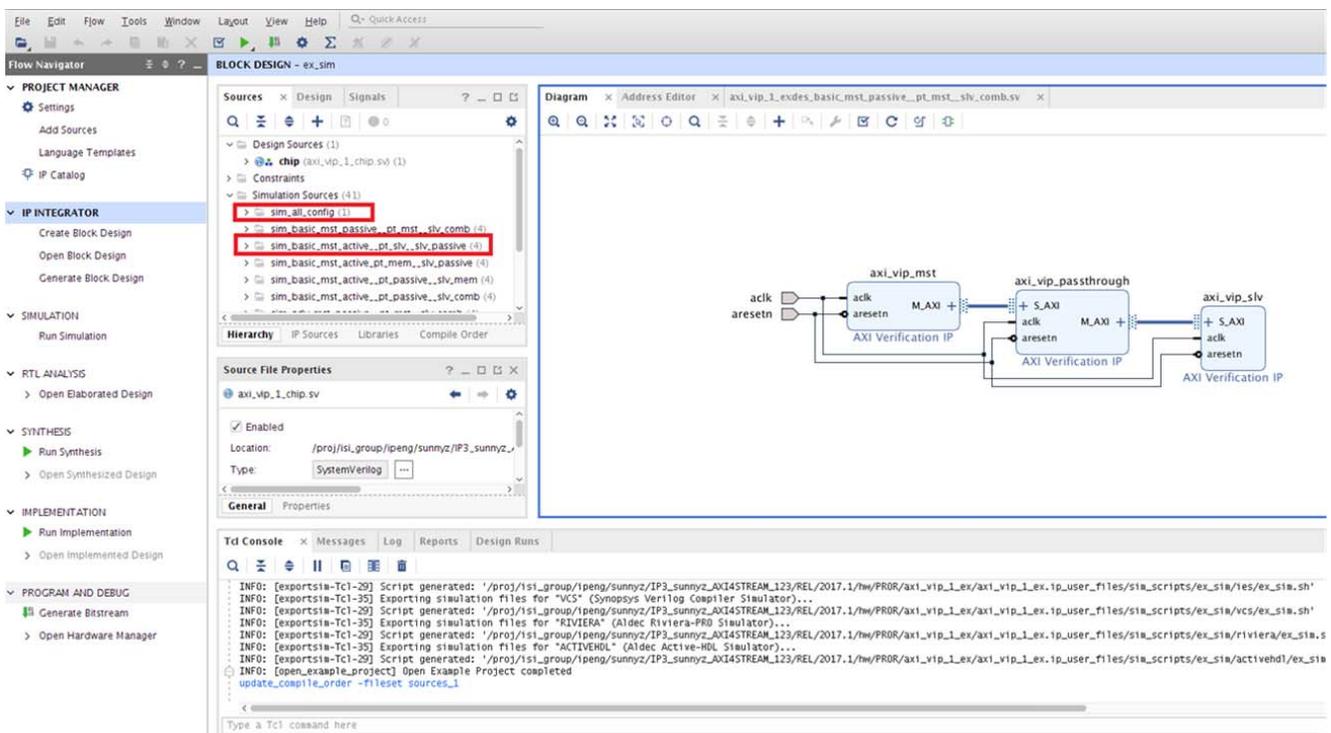
Figure 6-4: **Simulation Configuration in Vivado Design Suite**

AXI VIP Example Test Bench and Test

Because the example design is generated to match the VIP's configuration, the test bench is also configured to match the AXI VIP configuration. Eleven different test benches have been implemented for this VIP and each simulation set fits different needs described in the Multiple Simulation Sets section. For more information, see Chapter 5, Example Design.

Send Feedback

Useful Coding Guidelines and Examples

Must Haves in the Test Bench

While coding the test bench for the AXI VIP, the following requirements must be met. Otherwise, the AXI VIP does not work. These are the requirements for all VIPs.

1. Create module test bench as all other standard SystemVerilog test benches.

    ```
    module testbench();
    …
    endmodule
    ```

2. Import two required packages: `axi_vip_pkg` and `<component_name>_pkg`. The `<component_name>_pkg` includes agent classes and its subclasses for AXI VIP. For each VIP instance, it has a component package which is automatically generated when the outputs are created. This component package includes a `typedef` class of a parameterized agent. Xilinx recommends importing this package because reconfiguration of the VIP has no impact of the test bench. Figure 6-5 shows how to retrieve the `<component_name>_pkg from a` standalone AXI VIP.
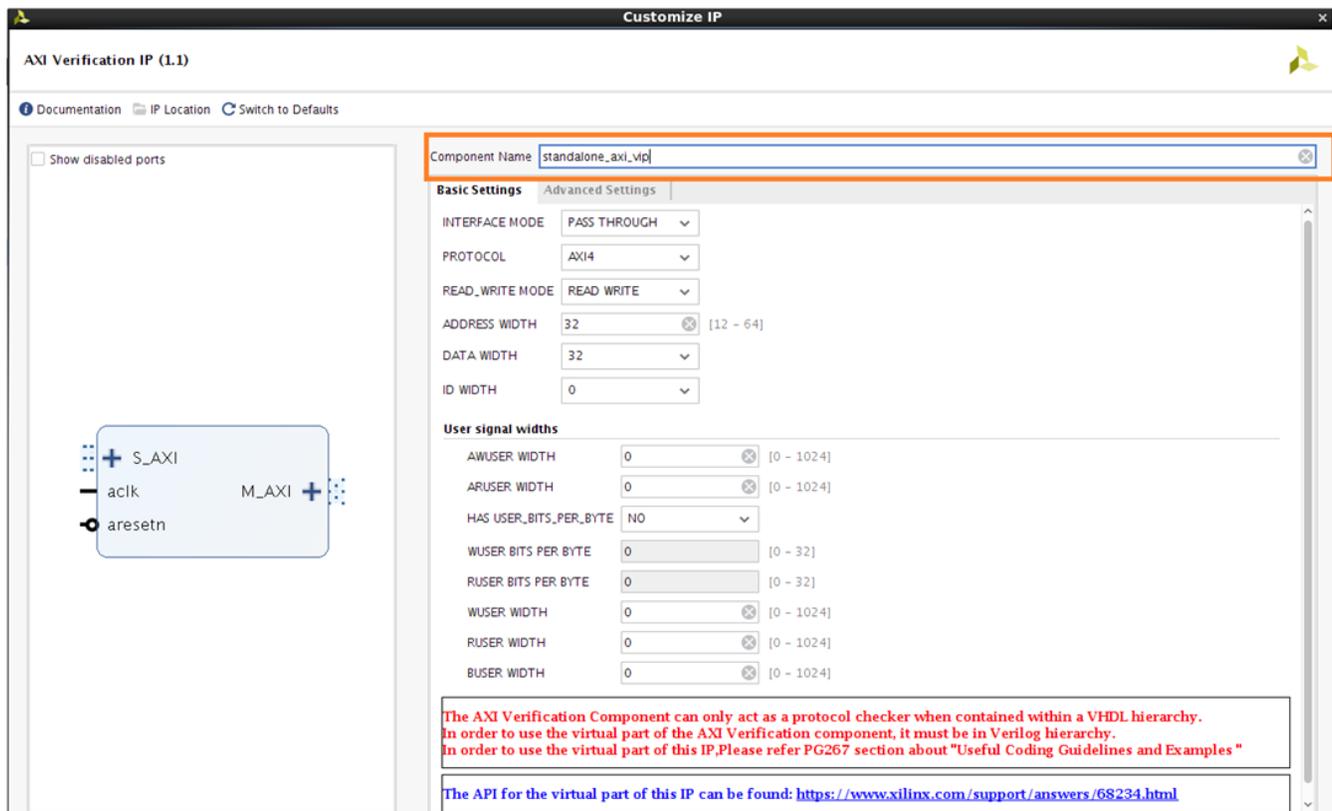
Figure 6-5: **Standalone <component_name>_pkg**

Figure 6-6 shows how to retrieve the `<component_name>_pkg` from an IP integrator design. First, select the VIP and in the **Block Properties** window click **Properties**. Then, under the **CONFIG** column select **Component_Name**.
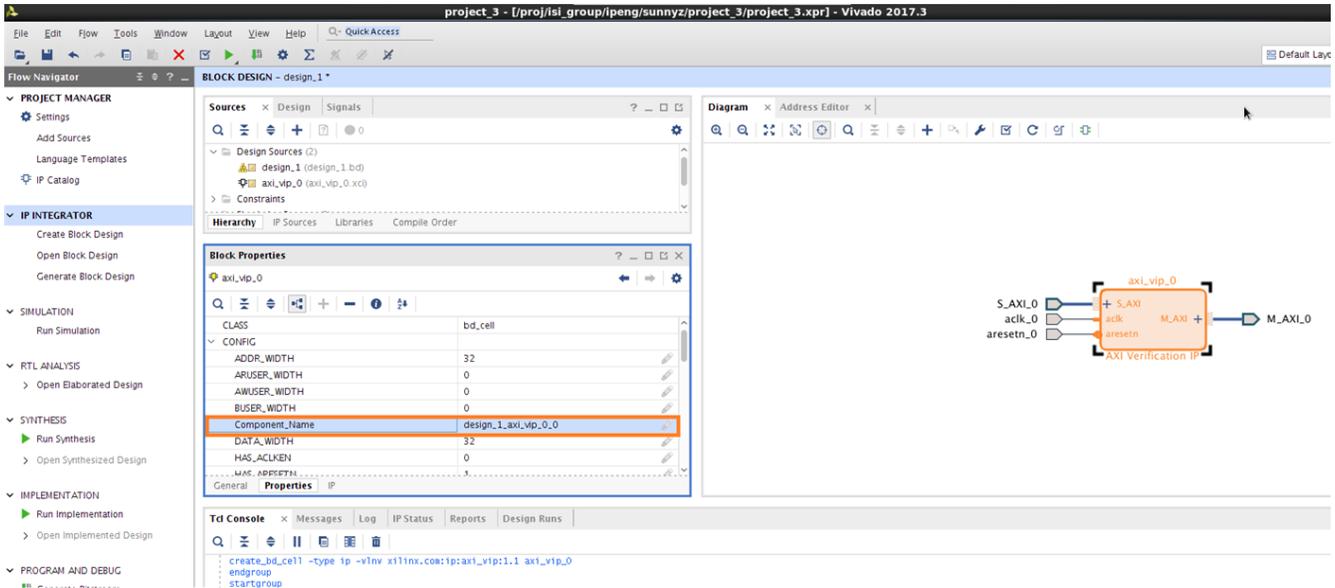

Figure 6-6: **IP Integrator <component_name>_pkg**

3. Declare agents. One agent for one AXI VIP has to be declared. Depending on the AXI VIP interface mode, different `typedef` class should be called for declaration of the agent.

Table 6-2: **Declare Agents**

Name	Description
<component_name>_mst_t	Master VIP
<component_name>_slv_t	Slave VIP without memory model
<component_name>_slv_mem_t	Slave VIP with memory model
<component_name>_passthrough_t	Pass-through VIP without memory model
<component_name>_passthrough_mem_t	Pass-through VIP with memory model

4. Create a `new` for the agent and pass the hierarchy path of IF correctly into the `new` function. Before the agent is set to `new`, run the simulation with an empty test bench to find the hierarchy path of the AXI VIP instance. A message like this shows up and then puts the path into a new function (see Figure 4-10).

```
agent = new("my VIP agent", <hierarchy_path>.IF);
```

Start Agent

For the VIP to start running, start agent has to be called. The following shows how the master, slave, and pass-through VIPs start to run.

AXI Master VIP

Start agent:

```
mst_agent.start_master();
```

Then, generate the transaction. For more information about how to generate transaction, see the Vivado example design simset and look for **mst_stimulus.sv**.

AXI Slave VIP

Start agent:

```
slv_agent.start_slave();
```

If **<component_name>_slv_t** is used, the user environment has to fill a write response if READ_WRITE_MODE is not READ_ONLY, and read response if READ_WRITE_MODE is not WRITE_ONLY. For more information, see the Vivado example design simset and look for **slv_basic_stimulus.sv/slv_stimulus.sv**.

If **<component_name>_slv_mem_t** is used, write response and read response are handled in the agent. The user environment does not need to do anything here. For more information, see the Vivado example design simset and look for **mem_basic_stimulus.sv/mem_stimulus.sv**.

AXI Pass-Through VIP

Runtime Slave Mode

Switch AXI pass-through VIP into the runtime slave mode. The **<hierarchy_path>** can be found in step 4 and then start the slave agent.

```
<hierarchy_path>.set_slave_mode();
passthrough_agent.start_slave();
```

If **<component_name>_passthrough_slv_t** is used, the write response/read response have to be filled in through the user environment.

For more information, see the Vivado example design simset and look for **passthrough_slv_basic_stimulus.sv/passthrough_slv_stimulus.sv**.

If **<component_name>_passthrough_mem_t** is used, the user environment does not need to do anything. For more information, see the Vivado example design simset and look for **passthrough_mem_basic_stimulus.sv/passthrough_mem_stimulus.sv**.

Send Feedback

Runtime Master Mode

Switch AXI pass-through VIP into the runtime master mode. The **<hierarchy_path>** can be found in step 4 and then start the master agent.

```
<hierarchy_path>.set_master_mode();
passthrough_agent.start_master();
```

Create transaction. For more information, see the Vivado example design simset and look for **passthrough_mst_stimulus.sv**.

Runtime Pass-Through Mode

Default is in pass-through mode. If you want to switch back to the pass-through mode from the master/slave mode, you have to call API **set_passthrough_mode**. The **<hierarchy_path>** can be found in step 4 and then start the pass-through agent.

```
<hierarchy_path>.set_passthrough_mode();.
```

The **start_monitor** is optional.

As described above, APIs used to switch pass-through VIP into the runtime master, runtime slave, and runtime pass-through modes are **set_master_mode**, **set_slave_mode**, and **set_passthrough_mode**.

The following shows a list of related parameters and type definitions used in the AXI VIP:

```
parameter XIL_AXI_MAX_DATA_WIDTH             = 1024;
parameter XIL_AXI_USER_BEAT_WIDTH            = 1024;
parameter XIL_AXI_USER_ELEMENT_WIDTH         = 32;
parameter XIL_AXI_VERBOSITY_NONE             = 0;
parameter XIL_AXI_VERBOSITY_FULL             = 400;
typedef integer                              xil_axi_int;
typedef longint                              xil_axi_long;
typedef integer unsigned                     xil_axi_uint;
typedef longint unsigned                     xil_axi_ulong;
typedef logic [7:0]                          xil_axi_payload_byte;
typedef logic                                xil_axi_strb_1byte;
typedef logic [XIL_AXI_USER_BEAT_WIDTH-1:0]  xil_axi_user_beat;
typedef logic [XIL_AXI_MAX_DATA_WIDTH-1:0]   xil_axi_data_beat;
typedef logic [XIL_AXI_MAX_DATA_WIDTH/8-1:0] xil_axi_strb_beat;
typedef integer unsigned                     xil_axi_user_element;
```

IMPORTANT: *You have to call* stop_master *when pass-through VIP switches from the runtime master mode to the other mode. Similarly, you have to call* stop_slave *when pass-through VIP switches from runtime slave mode to the other mode. For more information, see the example design in Vivado. The* start_master *and* start_slave *of the pass-through VIP agent cannot be called at the same time. When the pass-through VIP is switching from the runtime master mode to the runtime slave mode, the* stop_master *has to be called. Vice versa,* stop_slave *has to be called.*

Optional Test Bench Controls

Reactive Ports for the AXI Slave VIP

The AXI slave VIP has two agents which are **axi_slv_agent** and **axi_slv_mem_agent**. If you want to generate your own traffic, use the **axi_slv_agent** and **get_rd_reactive**. The **axi_slv_mem_agent** has its own method of generating traffic.

The best technique is to place the read response which is shown in the example design **simset sim_all_config**. The procedure is shown here:

1. In the read driver of the slave agent, use the **get_rd_reactive** to receive the read command,

2. Fill in the transaction read data information and send it back to the AXI slave VIP interface. Because both the **get_rd_reactive** and send are blocking, they have to be included in the initial and forever blocks without any blocking events. See the following code example:

```
//slave VIP agent gets read transaction cmd information, fill in data information and
send it back to Slave VIP interface
  initial begin
    forever begin
      slv_agent.rd_driver.get_rd_reactive(rd_reactive);
      fill_payload(rd_reactive);
      fill_ruser(rd_reactive);
      fill_beat_delay(rd_reactive);
      slv_agent.rd_driver.send(rd_reactive);
    end
  end
```

How to Receive Read Data

There are two techniques for obtaining the read data:

- Receiving it through the read driver of the master agent (see the **driver_rd_data_method_one** and **driver_rd_data_method_two** in the ***mst_stimulus.sv** file).

- Obtaining it through the monitor of the VIP (see the **monitor_rd_data_method_one** and **monitor_rd_data_method_two** in the ***exdes_generic.sv** file).

To receive data from the monitor, follow these steps:

1. Obtain the monitor transaction from the **item_collected_port**. In this example, it comes from the master agent.

2. If the **cmd** type is XIL_AXI_READ in the monitor transaction, use the **get_data_beat** and **get_data_block** to obtain the read data.

If the `cmd` type is XIL_AXI_WRITE in the monitor transaction, use the `get_data_beat` and `get_data_block` to obtain the write data. The `monitor_rd_data_method_one` shows how to receive the data beat through the monitor transaction. The `monitor_rd_data_method_two` shows how to receive the data block through the monitor transaction.

To receive data from the read driver, follow theses steps:

1. Use the read driver in the master agent to create a read transaction handle.

2. Randomize the read transaction. If you want to generate a specific read command, use the APIs in the `axi_transaction` class to set the address, burst length, etc. or randomize the transaction with specific values.

3. Set the `driver_return_item_policy` of the read transaction to be any of the following values:

   XIL_AXI_PAYLOAD_RETURN or XIL_AXI_CMD_PAYLOAD_RETURN

4. Use the read driver to send read transaction out.

5. Use the read driver to wait for the response.

6. Use the `get_data_beat/get_data_block` to inspect data from the response transaction. The `driver_rd_data_method_one` shows how to obtain a data beat from the read driver. The `driver_rd_data_method_two` shows how to obtain a data block from the read driver.

Note: The API `get_data_beat`: `get_data_beat` returns the value of the specified beat. It always returns 1024 bits. It aligns the significant bytes to the lower bytes and sets the unused bytes to zeros.

The is not always the RDATA representation. If the data width is 32 bits and the transaction is subsize burst (1B in this example), only the last byte of `get_data_beat` is valid. This is very different from the Physical Bus.

```
get_data_bit               Physical Bus
1024  ...      0            32         0
----------------           ----------
|            X|            |          X|
|            X|            |       X   |
|            X|            |    X      |
|            X|            | X         |
----------------           ----------
```

Note: The API `get_data_block`: `get_data_block` returns 4K bytes of the payload for the transaction. This is not always the RDATA representation. If the data width is 32 bits and the transaction is subsize burst (1B in this example), it aligns the significant bytes to the lower bytes and set the unused bytes to zeros.

```
     get_data_block              Physical Bus
     32      ...        0        32           0
  0 ----------------            -----------
    | D   C   B   A|            |         A|
    | 0   0   0   0|            |      B   |
    | 0   0   0   0|            |   C      |
    | 0   0   0   0|            | D        |
    | 0   0   0   0|            -----------
    | 0   0   0   0|
 1k----------------
```

```
   task monitor_rd_data_method_one(input axi_monitor_transaction updated);
     xil_axi_data_beat                       mtestDataBeat[];
     mtestDataBeat = new[updated.get_len()+1];
    for( xil_axi_uint beat=0; beat<updated.get_len()+1; beat++) begin
       mtestDataBeat[beat] = updated.get_data_beat(beat);
      //  $display(" Read data from Monitor: beat index %d, Data beat %h", beat,
mtestDataBeat[beat]);
     end
   endtask

   task monitor_rd_data_method_two(input axi_monitor_transaction updated);
     bit[8*4096-1:0]                         data_block;
     data_block = updated.get_data_block();
    // $display(" Read data from Monitor: Block Data %h ", data_block);
   endtask

    task driver_rd_data_method_one();
    axi_transaction                          rd_transaction;
    xil_axi_data_beat                        mtestDataBeat[];
    rd_transaction = agent.rd_driver.create_transaction("read transaction with
randomization");
    RD_TRANSACTION_FAIL_1a:assert(rd_transaction.randomize());
    rd_transaction.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN);
    agent.rd_driver.send(rd_transaction);
    agent.rd_driver.wait_rsp(rd_transaction);
    mtestDataBeat = new[rd_transaction.get_len()+1];
    for( xil_axi_uint beat=0; beat<rd_transaction.get_len()+1; beat++) begin
      mtestDataBeat[beat] = rd_transaction.get_data_beat(beat);
   //   $display("Read data from Driver: beat index %d, Data beat %h ", beat,
mtestDataBeat[beat]);
    end
   endtask


   task driver_rd_data_method_two();
    axi_transaction                          rd_transaction;
    bit[8*4096-1:0]                          data_block;
    rd_transaction = agent.rd_driver.create_transaction("read transaction with
     randomization");
    RD_TRANSACTION_FAIL_1a:assert(rd_transaction.randomize());
    rd_transaction.set_driver_return_item_policy(XIL_AXI_PAYLOAD_RETURN);
    agent.rd_driver.send(rd_transaction);
    agent.rd_driver.wait_rsp(rd_transaction);
    data_block = rd_transaction.get_data_block();
   // $display("Read data from Driver: Block Data %h ", data_block);
   endtask
```

For more information, see the `simset sim_adv_mst_active_*` in the Vivado example design.

Create Ready Signal

The following shows how to create a specified `rready` signal. Use the AXI master VIP or AXI pass-through VIP in runtime master mode. Use the `create` API to create `rready`, set the low and high pattern, and then send it to the VIP interface. For other `rready` signals, if nothing is done, the AXI VIP itself generates this `rready` signal. For more information on different modes of the READY handshake, see the READY Generation section.

```
// master agent create rready
rready_gen = mst_agent.rd_driver.create_ready("rready");
// set the feature of rready signal. If nothing is done here, default pattern of
ready will be generated
rready_gen.set_ready_policy(XIL_AXI_READY_GEN_OSC);
rready_gen.set_low_time(1);
rready_gen.set_high_time(2);
// send the rready to VIP interface
mst_agent.rd_driver.send_rready(rready_gen);
```

Other Optional Controls

Using APIs to set agents' tag and verbosity for debug purpose.

```
 //set tag for agents for easy debug
 mst_agent.set_agent_tag("Master VIP");
//verbosity level which specifies how much debug information to produce
// 0        - No information will be shown.
// 400      - All information will be shown.
mst_agent.set_verbosity(mst_agent_verbosity);
```

Transaction Examples

There are different methods of configuring the transaction after it is created. In the example design, `sim_all_config`, three methods are shown for write and read transactions. Method 1 is to fully randomize the transaction after it is being created. Method 2 is similar to AXI BFM WRITE_BURST and READ_BURST for migration purpose. Method 3 shows how to use the APIs to set the transaction.

Write Transaction Methods

Method 1 for Write Transaction

For a write transaction, follow these steps:

1. Create the transaction from the write driver of the master agent.

2. Randomize the transaction.

3. Send the transaction from the master agent write driver.

> **IMPORTANT:** *The API sent here is non-blocking. By default, the driver does not return transaction information. If you want to receive transaction information back, the API,* `set_driver_return_item`, *can be called and the related* `driver_return_item_policy` *can be called here.*

```
wr_transaction = mst_agent.wr_driver.create_transaction("write transaction");
WR_TRANSACTION_FAIL_1b: assert(wr_transaction.randomize());
mst_agent.wr_driver.send(wr_transaction);
```

Method 2 for Write Transaction

Note: This is a blocking write and it blocks until the BRESP is returned.

```
//The following shows the AXI4_WRITE_BURST. For other options. see the AXI
VIP API Documentation [Ref 12].
 mst_agent.AXI4_WRITE_BURST(
    mtestID,
    mtestADDR,
    mtestBurstLength,
    mtestDataSize,
    mtestBurstType,
    mtestLOCK,
    mtestCacheType,
    mtestProtectionType,
    mtestRegion,
    mtestQOS,
    mtestAWUSER,
    mtestWData,
    mtestWUSER,
    mtestBresp
  );
```

Method 3 for Write Transaction

This methods shows how to use the APIs to set the command and data of the write transaction.

```
wr_transaction = mst_agent.wr_driver.create_transaction("write transaction");
wr_transaction.set_write_cmd(mtestADDR,mtestBurstType,mtestID,
                                  mtestBurstLength,mtestDataSize);
wr_transaction.set_data_block(mtestWData);
for(int beat=0; beat<wr_transaction.get_len()+1; beat++) begin
    wr_transaction.set_data_beat(beat, dbeat);
end
mst_agent.wr_driver.send(wr_transaction);
```

Read Transaction Methods

Method 1 for Read Transaction

For a read transaction, follow these steps:

1. Create the transaction from the read driver of the master agent.

2. Randomize the transaction.

3. Send the transaction from the master agent read driver.

IMPORTANT: *The API sent here is non-blocking. If you want to receive transaction information back, the driver return item policy needs to be set here.*

```
rd_transaction = mst_agent.rd_driver.create_transaction("read transaction");
RD_TRANSACTION_FAIL_1a:assert(rd_transaction.randomize());
mst_agent.rd_driver.send(rd_transaction);
```

Method 2 for Read Transaction

Note: This is a blocking read and it blocks until the BRESP is returned.

```
    // The following shows the AXI4_READ_BURST. For other options. see the AXI
    VIP API Documentation [Ref 12].
 mst_agent.AXI4_READ_BURST (
        mtestID,
        mtestADDR,
        mtestBurstLength,
        mtestDataSize,
        mtestBurstType,
        mtestLOCK,
        mtestCacheType,
        mtestProtectionType,
        mtestRegion,
        mtestQOS,
        mtestARUSER,
        mtestRData,
        mtestRresp,
        mtestRUSER
    );
```

Method 3 for Read Transaction

This methods shows how to use the APIs to set the command and data of the read transaction.

```
rd_transaction = mst_agent.rd_driver.create_transaction("read transaction");
rd_transaction.set_read_cmd(mtestADDR,mtestBurstType,mtestID,
                            mtestBurstLength,mtestDataSize);
mst_agent.rd_driver.send(rd_transaction);
```

Issue Capability

For the **axi_vip** slave and **axi_vip** master, you have the capability to program how many threads of write/read transactions it can accept at the same time. By default, the maximum number of VIP is set at 25. To accept or send more transactions, use the following sample codes to set at 40 transactions. Also, ensure the **<your_slv_agent>** in the code is the right agent in your design.

For AXI slave VIP,

```
write transaction:
<your_slv_agent>.slv_wr_driver.seq_item_port.set_max_item_cnt(40);
read transaction:
<your_slv_agent>.slv_rd_driver.seq_item_port.set_max_item_cnt(40);
```

For AXI master VIP,

```
write transaction:
<your_slv_agent>.mst_wr_driver.seq_item_port.set_max_item_cnt(40);
read transaction:
<your_slv_agent>.mst_rd_driver.seq_item_port.set_max_item_cnt(40);
```

For AXI pass-through VIP in runtime slave mode,

```
write transaction:
<your_slv_agent>.slv_wr_driver.seq_item_port.set_max_item_cnt(40);
read transaction:
<your_slv_agent>.slv_rd_driver.seq_item_port.set_max_item_cnt(40);
```

For AXI pass-through VIP in runtime master mode,

```
write transaction:
<your_slv_agent>.mst_wr_driver.seq_item_port.set_max_item_cnt(40);
read transaction:
<your_slv_agent>.mst_rd_driver.seq_item_port.set_max_item_cnt(40);
```

Upgrading

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations between core versions.

Changes from v1.0 to v1.1

The `axi_vip` package changed from `axi_vip_v1_0` to `axi_vip_v1_1`.

Send Feedback

axi_vip_v1_1_top APIs

This appendix contains information about the `axi_vip_v1_1_top` APIs. These APIs can be called through the following code. The `set_passthrough_mode`, `set_master_mode`, and `set_slave_mode` are used to switch the pass-though VIP into different runtime modes. Other APIs are used for assertion purposes. An example would be `set_passthrough_mode`.

```
<hierarchy_path>.set_passthrough_mode()
```

For `<hierarchy_path>`, see Figure 4-10 in Chapter 4, Design Flow Steps and AXI Pass-Through VIP, page 48.

```
set_passthrough_mode
function void set_passthrough_mode()
Sets AXI VIP passthrough into run time passthrough mode

set_master_mode
function void set_master_mode()
Sets AXI VIP passthrough into run time master mode

set_slave_mode
function void set_slave_mode()
Sets AXI VIP passthrough into run time slave mode

set_xilinx_slave_ready_check
function void set_xilinx_slave_ready_check()
Sets xilinx_slave_ready_check_enable of IF to be 1

clr_xilinx_slave_ready_check
function void clr_xilinx_slave_ready_check()
Sets xilinx_slave_ready_check_enable of IF to be 0

set_max_aw_wait_cycles
function void set_max_aw_wait_cycles(
input integer unsigned new_num)
Sets max_aw_wait_cycles of PC(Arm Protocol Checker), not available in Vivado
Simulator.

set_max_ar_wait_cycles
function void set_max_ar_wait_cycles(
input integer unsigned new_num)
Sets max_ar_wait_cycles of PC(Arm Protocol Checker), not available in Vivado
Simulator.

set_max_r_wait_cycles
function void set_max_r_wait_cycles(
input integer unsigned new_num)
```

Send Feedback

Sets max_r_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_max_b_wait_cycles
function void set_max_b_wait_cycles(
input integer unsigned new_num)
Sets max_b_wait_cycles of PC (Arm Protocol Checker), not available in Vivado Simulator.

set_max_w_wait_cycles
function void set_max_w_wait_cycles(
input integer unsigned new_num)
Sets max_w_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_max_wlast_wait_cycles
function void set_max_wlast_wait_cycles(
input integer unsigned new_num)
Sets max_wlast_to_awvalid_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_max_rtransfer_wait_cycles
Sets max_rtransfer_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_max_wtransfer_wait_cycles
Sets max_wtransfer_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_max_wlcmd_wait_cycles
function void set_max_wlcmd_wait_cycles(
input integer unsigned new_num)
Sets max_wlcmd_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_aw_wait_cycles
function integer unsigned get_max_aw_wait_cycles()
Returns max_aw_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_ar_wait_cycles
function integer unsigned get_max_ar_wait_cycles()
Returns max_ar_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_r_wait_cycles
function integer unsigned get_max_r_wait_cycles()
Returns max_r_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_b_wait_cycles
function integer unsigned get_max_b_wait_cycles()
Returns max_b_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_w_wait_cycles
function integer unsigned get_max_w_wait_cycles()
Returns max_w_wait_cycles of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_wlast_wait_cycles
`function integer unsigned get_max_wlast_wait_cycles()`
Returns `max_wlast_to_awvalid_wait_cycles` of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_rtransfer_wait_cycles
Returns `max_rtransfer_wait_cycles` of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_wtransfer_wait_cycles
Returns `max_wtransfer_wait_cycles` of PC(Arm Protocol Checker), not available in Vivado Simulator.

get_max_wlcmd_wait_cycles
`function integer unsigned get_max_wlcmd_wait_cycles()`
Returns `max_wlcmd_wait_cycles` of PC(Arm Protocol Checker), not available in Vivado Simulator.

set_fatal_to_warnings
`function void set_fatal_to_warnings()`
Sets `fatal_to_warnings` of PC(Arm Protocol Checker) to be 1, not available in Vivado Simulator.

clr_fatal_to_warnings
`function void clr_fatal_to_warnings()`
Sets `fatal_to_warnings` of PC(Arm Protocol Checker) to be 0, not available in Vivado Simulator.

Send Feedback

Migrating from BFM to VIP

This appendix contains information about migrating a Vivado® design with AXI BFM to AXI VIP.

1. In Vivado IP integrator BD design, replace BFM with AXI VIP and configure the AXI VIP. If the old BFM is AXI4 in slave mode, for example, set up the AXI VIP protocol to AXI4 and the interface mode to slave.

2. In the test bench, remove all BFM related tasks and add the following codes:

   ◦ Import two packages, see Chapter 6, Test Bench on how to obtain
     `<component_name>_pkg`:

     ```
     import <component_name>_pkg::*
     import axi_vip_pkg::*;
     ```

   ◦ Declare agent, see Chapter 6, Test Bench for requirements.

     Because it is for migration purpose, no pass-through agent is declared since BFM did not support pass-through mode.

   ◦ Construct the agent.

     - For example, AXI VIP in master mode:

       ```
       mst_agent = new("master vip agent",<hierarchy path to AXI VIP instance>
       inst.IF);
       ```

     See Chapter 4, Design Flow Steps to find the hierarchy path.

   ◦ Start agent:

     - If AXI VIP is in master mode:

       ```
       mst_agent.start_master();
       ```

   ◦ Replace the existing BFM WRITE/READ_BURST with the VIP WRITE/READ_BURST.

     - If AXI VIP is AXI4:

       ```
       AXI4_WRITE/READ_BURST
       ```

     - If AXI VIP is AXI3:

       ```
       AXI3_WRITE/READ_BURST
       ```

     - If AXI VIP is AXI4LITE:

       ```
       AXI4LITE_WRITE/READ_BURST
       ```

AXI VIP Agent and Flow Methodology

This appendix contains information about the AXI VIP agents and flow methodologies. The AXI VIP has three agents:

- AXI Master Agent
- AXI Slave Agent
- AXI Pass-Through Agent

AXI Master Agent

When instantiating an AXI master VIP, a master agent has to be declared and constructed. Class `axi_mst_agent` contains other components that consist of the entire master Verification IP. These are the read driver, write driver, and monitor.
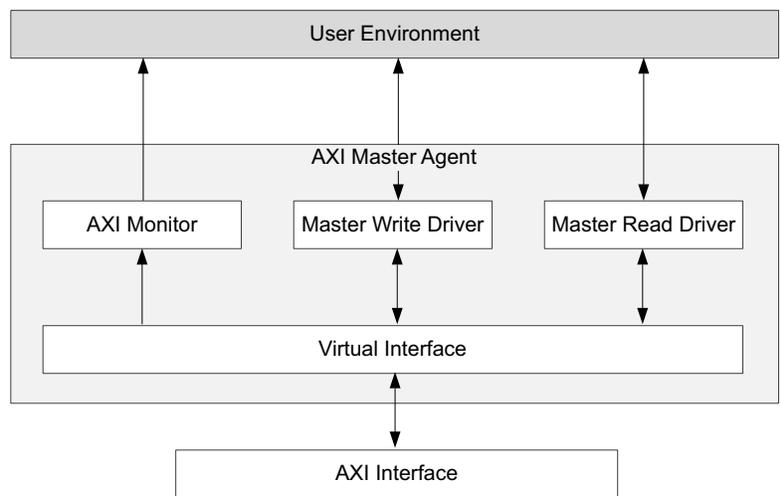
Figure D-1: **AXI Master VIP Agent**

Send Feedback

AXI Master Read Driver

• Receives READ transactions from the user environment and drives the AR channel.

• Triggers an event when the AR Command is accepted.

• Receives READY transactions from the user environment and drives the RREADY signal of the R channel.

• By default, transaction does not return until you set the driver return item policy. For usage, see the *AXI VIP API Documentation* [Ref 12].

AXI Master Write Driver

• Receives WRITE transactions from the user environment and drives the AW and W channels.

• Triggers an event when the AW Command is accepted.

• Triggers an event when the WLAST is accepted.

• Receives READY transactions from the user environment and drives the BREADY signal of the B channel.

• By default, transaction does not return until you set the driver return item policy. For usage, see the *AXI VIP API Documentation* [Ref 12].

AXI Monitor

• Monitors all five AXI channels: AW, AR, R, W, and B.

• It has seven analysis ports which you can select to use. By default, only `item_collect_port` is ON, all other ports are OFF. You can use the API, `set_enabled`, in each analysis port to switch ON the port. They include:

Table D-1: **Analysis Ports**

Name	Description
item_collect_port	Collects both write/read transaction information and converts to axi_monitor_transaction.
axi_cmd_port	Collects both write/read channel information and converts to xil_axi_cmd_beat.
axi_rd_cmd_port	Collects read address channel information and converts to xil_axi_cmd_beat.
axi_wr_cmd_port	Collects write address channel information and converts to xil_axi_cmd_beat.
axi_bresp_port	Collects write response channel information and coverts to xil_axi_resp_beat.
axi_wr_beat_port	Collects write data channel information and converts to xil_axi_wr_beat.
axi_rd_beat_port	Collects read data channel information and converts to xil_axi_read_beat.

• Collects and reorders R Channel beats and returns a completed transaction when the RLAST is accepted.

Send Feedback

- Collects and reorders B Channel response and returns a completed transaction when the B channel is accepted.

- Transaction based protocol checking.

Transaction Return Item Policy Implementation

Table D-2 shows the transaction return descriptions.

`last_handshake` = *LAST + *VALID + *READY

`cmd_handshake` = *VALID + *READY

`bresp_handshake` = BVALID + BREADY

Table D-2: **Transaction Return Descriptions**

Name	Description
NO_RETURN	Item is not returned by the driver.
CMD_RETURN	Item is returned when at cmd_handshake.
PAYLOAD_RETURN	For writes, the item is returned at bresp_handshake. For reads, the item is returned at last_handshake
CMD_PAYLOAD_RETURN	Item is returned twice. At cmd_handshake. At: For writes: bresp_handshake. For reads: last_handshake.
CMD_WLAST_RETURN	Item is returned twice: At cmd_handshake. At last_handshake. ***Note:*** This return type is not valid for read transactions.
CMD_WLAST_PAYLOAD_RETURN	Item is returned three times: At cmd_handshake. At last_handshake. At bresp_handshake. ***Note:*** This return type is not valid for read transactions.
WLAST_PAYLOAD_RETURN	Item is returned twice: At last_handshake. At bresp_handshake. ***Note:*** This return type is not valid for read transactions.
WLAST_RETURN	Item is returned at last_handshake. ***Note:*** This return type is not valid for read transactions.

Items Returned During Reset

When a reset is applied, transactions that are in flight are returned to the user environment (sequencer). There are cases where the same transaction exist in multiple queues, so in those cases the process modifies the transaction and changes it to NO_RETURN. In these situations, the user environment does not return the item multiple times.

Send Feedback

Write CMD and Data Flow Diagram
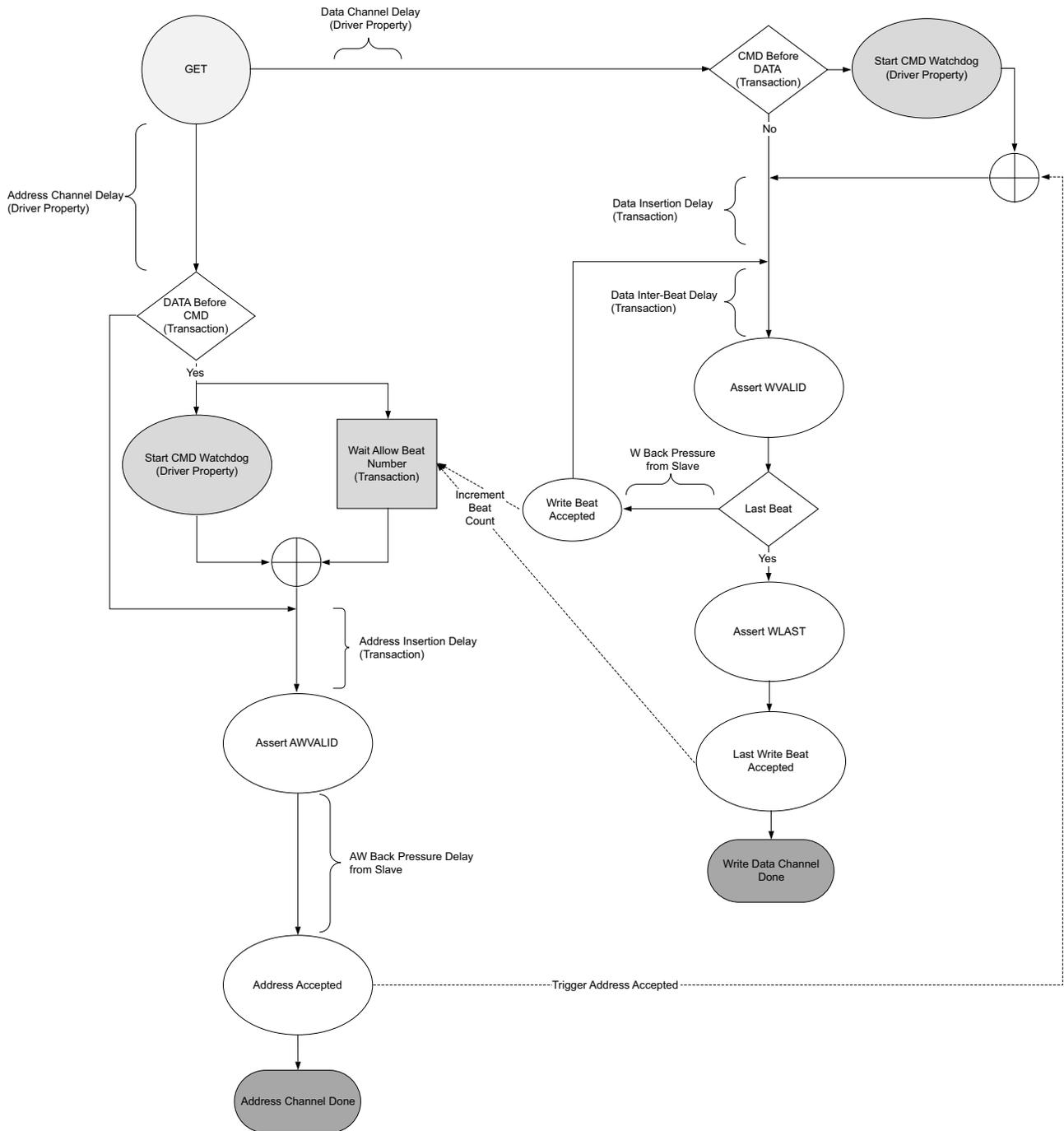
Figure D-2: **Write Command and Data Flow**

Send Feedback

Write Transaction Flow

The AXI master write transaction flows through the write agent in the following steps:

1. Driver asks for the next transaction through a `get_next_item`. This is a blocking get.

2. The user environment creates a single transaction. The transaction contains the following:

   ◦ **Command Information** – AWADDR, AWLEN, AWSIZE, AWID, etc.

   ◦ **Payload** – WDATA byte array and WSTRB array

   ◦ **Master Controlled Timing** – Inter-beat timing and address delay

3. The user environment pushes the transaction to the driver.

4. The driver pops the transaction from the REQUEST port and places it on a queue to be processed and driven onto the interface. If the transaction depth > 1, the driver continues to accept transactions until this value has been met.

5. When the interface receives the BRESP from the slave, the master driver returns a copy of the transaction to the user environment if the transaction driver return item policy is set to one of the following values:

   ◦ XIL_AXI_CMD_RETURN

   ◦ XIL_AXI_CMD_WLAST_RETURN

   ◦ XIL_AXI_WLAST_RETURN

   ◦ XIL_AXI_PAYLOAD_RETURN

   ◦ XIL_AXI_CMD_PAYLOAD_RETURN

   ◦ XIL_AXI_CMD_WLAST_PAYLOAD_RETURN

   ◦ XIL_AXI_WLAST_PAYLOAD_RETURN

6. The user environment receives the completed transaction if the driver return item policy is not XIL_NO_RETURN.
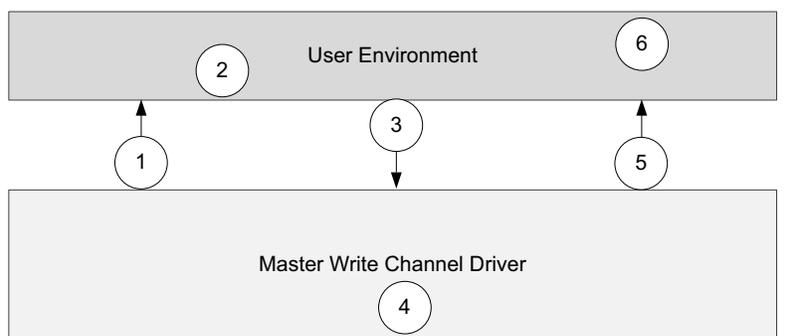
X18587-121316

Figure D-3: **Write Transaction Flow**

Send Feedback

Read Command Timing Diagram
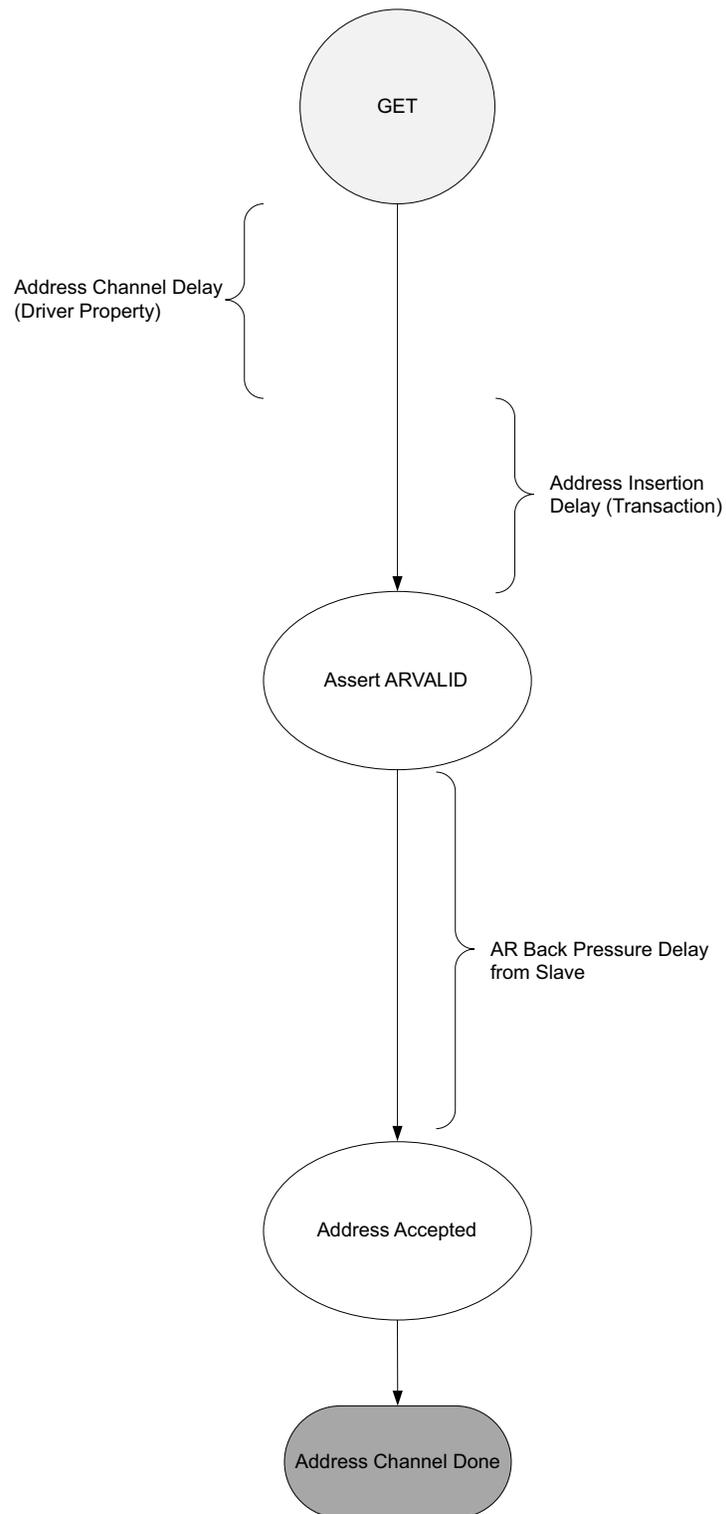
Figure D-4: **Read Command Diagram**

Read Transaction Flow

An AXI master read transaction flows through the read agent in the following steps:

1. Driver asks for the next transaction through a `get_next_item`. This is a blocking get.

2. The user environment creates a single transaction. The transaction contains the following:

   ◦ **Command Information** – AWADDR, AWLEN, AWSIZE, AWID, etc.

   ◦ **Master Controlled Timing** – Address delay

3. The user environment pushes the transaction to the driver.

4. The driver pops the transaction from the REQUEST port and places it on a queue to be processed and driven onto the interface. If the transaction depth > 1, the driver continues to accept transactions until this value has been met.

5. When the interface receives the RDATA from the slave for the given RID, it fills in the Payload field as RDATA byte array, RRESP.

6. The user environment receives the completed transaction when the transaction driver return item policy is set to either XIL_AXI_CMD_RETURN or XIL_AXI_CMD_PAYLOAD_RETURN.

X18589-121316
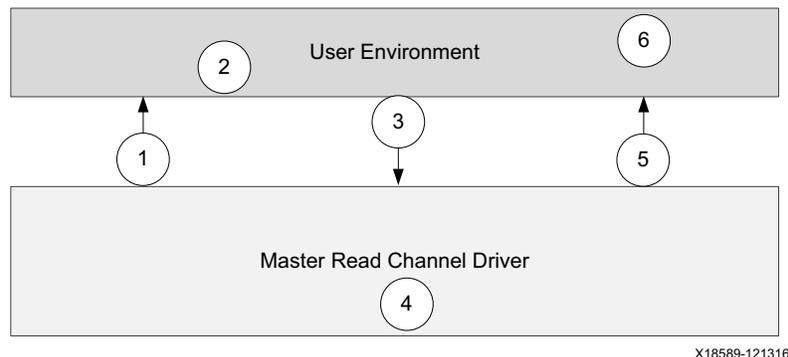
Figure D-5: **Read Transaction Flow**

BREADY Data Flow

- BREADY is generated independently of the AW and W channels.

- Configuration of the BREADY does not come from a transaction or analysis port as the transaction that carries the AW and W payload.

- BREADY configuration can change at different times than the AW/W channels.

- See the Configurable Ready Delays for different timing options.
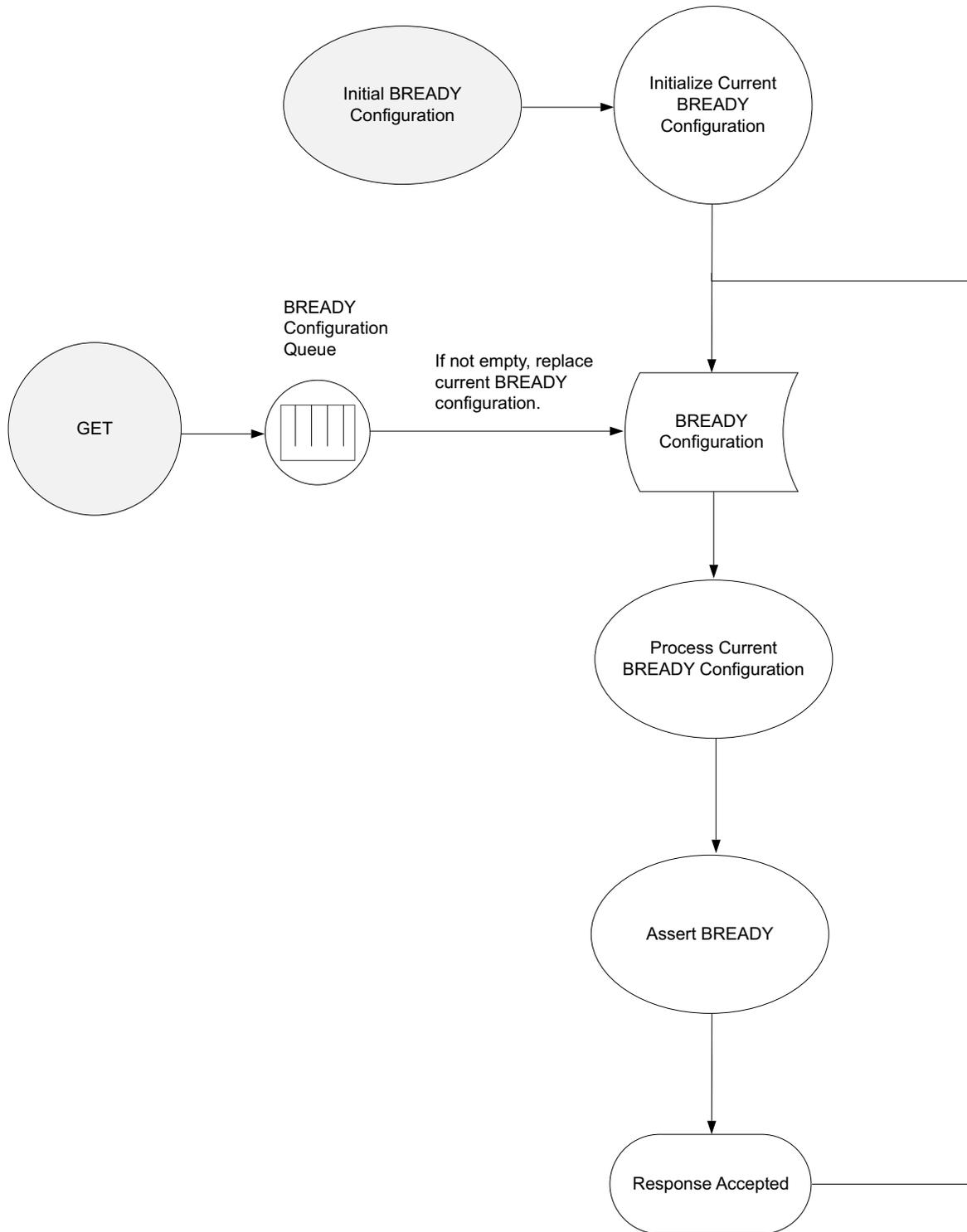
X18590-033017

Figure D-6: **BREADY Data Flow**

RREADY Data Flow

- RREADY is generated independently of the AR channel.

- Configuration of the RREADY is not from the same transaction or analysis port as the transaction that carries the AR payload.

- RREADY configuration can change asynchronously to the AR channels.

- See the Configurable Ready Delays for different timing options.
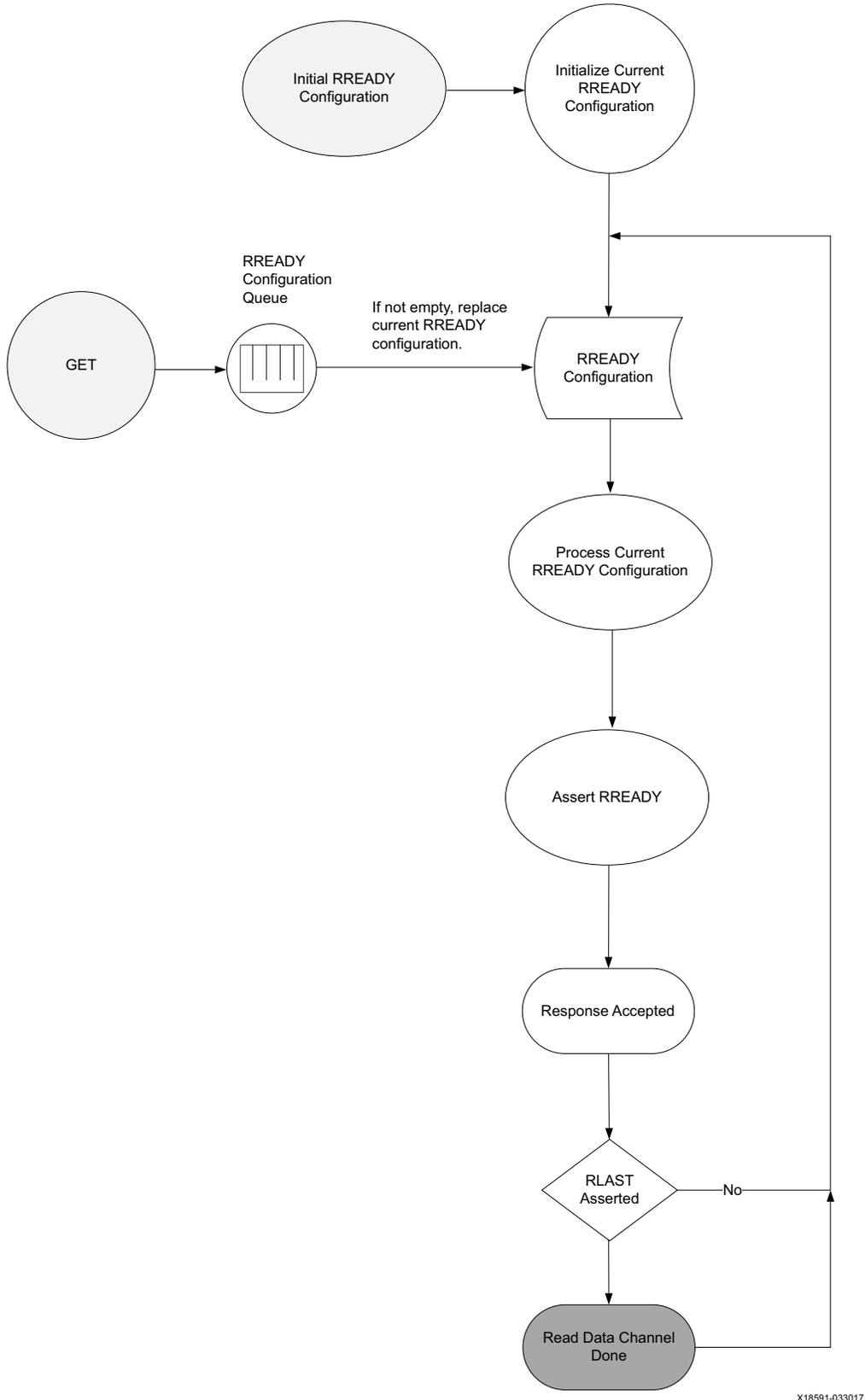
Figure D-7:   **RREADY Data Flow**

Send Feedback

AXI Slave Agent

When instantiating an AXI slave VIP, a slave agent has to be declared and constructed. Class **axi_slv_agent** contains other components that consist of the entire slave Verification IP. These are the read driver, write driver, and monitor.

```
User Environment
```

```
AXI Slave Agent
  AXI Monitor    CMD      BRESP      CMD      RDATA
                 Slave Write Driver   Slave Read Driver
                 Virtual Interface
```

```
AXI Interface
```

X18592-121316
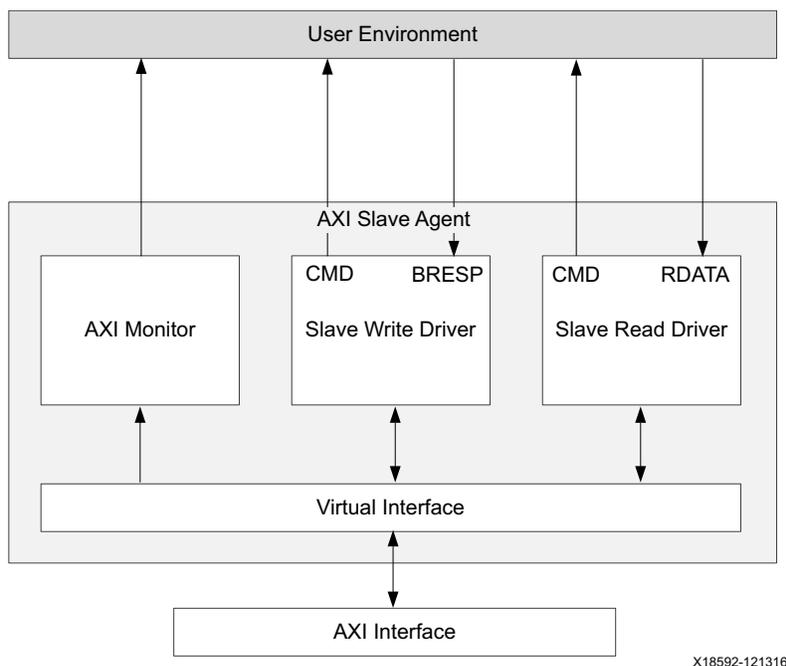
Figure D-8: **AXI Slave VIP Agent**

AXI Slave Read Driver

• Receives AR Command from the interface and then passes that command to the user environment. Create a READ transaction and pass it back to the driver to drive the R channel.

• Triggers an event when the AR Command is accepted.

• Triggers an event when the RLAST is accepted.

• Receives READY transactions from the user environment and drives the ARREADY signal of the AR channel.

• Drives the R channel.

• Configurable command reordering.

Send Feedback

AXI Slave Write Driver

- Receives AW Command and the WLAST from the interface and then passes that transaction to the user environment. The user environment creates a BRESP transaction and pass it back to the driver to drive the B channel.

- Receives WRITE transactions from the user environment and drives the AW and W channels.

- Triggers an event when the AW Command is accepted.

- Triggers an event when the WLAST is accepted.

- Receives READY transactions from the user environment and drives the AWREADY signal of the AW channel.

- Receives READY transactions from the user environment and drives the WREADY signal of the W channel.

- Generates reordered B Channel responses.

AXI Monitor

- Monitors all five AXI channels: AW, AR, R, W, and B.

- Collects and reorders R Channel beats and returns a completed transaction when the RLAST is accepted.

- Collects and reorders B Channel response and returns a completed transaction when the B channel is accepted.

- Transaction-based protocol checking.

Write Response/Reaction Data Flow Diagram

Wait for AW Command Accepted

Wait for WLAST Beat Accepted

awcmd_accept_event

AW Address Accepted

WLAST Beat Accepted

wlast_accept_event

FIFO

FIFO

Reactive Put to User Environment

User Environment Receives Transaction and Fills in BRESP

GET BRESP

Response Channel Delay (Driver Property)

Response Delay (Transaction Property)

Assert BVALID

B Back Pressure from Master

Response Accepted

Response Channel Done

X18593-040517

Figure D-9: **Write Response/Reaction Data Flow**

Send Feedback

Write Transaction Flow

An AXI slave write agent has three concurrent transaction flows:

- Address channel servicing

- Data channel serving

- Response channel generation

Both the address and data channels have their READY responses configured independently from the user environment. Only the response channel, B, relies on communication with the user environment to make forward progress. The transaction flows through the write agent in the following steps:

1. Master write response driver performs a blocking get to the user environment through a `get_next_item`. Because the command has not yet been received, the user environment must wait until the command has been received from the master.

2. The user environment performs a blocking get, `get_next_item`, on the reactive port of the driver.

3. The driver at this time can accept the incoming beats of WDATA and AWADDR and places them in a holding structure.

4. Only after the slave driver receives the complete AWADDR phase and the WDATA phase, it transfers the command object through the reactive port to the user environment.

5. The user environment determines the correct response to the request and puts the complete transaction on the REQUEST port of the driver. The transaction has:

   ○ **Command Information** – AWADDR, AWLEN, AWSIZE, AWID, etc. From the original command passed to the user environment.

   ○ **Response Controlled Timing and Response** – Response delay, BRESP.

6. The driver pops the transaction from the REQUEST port and places it on the queue to be processed on the response interface and starts the response channel delay timer. It is possible that before the response channel delay timer has expired more than one transaction can exist in the response list. When there is more than one item on the response list, the driver selects the next response to be sent.

7. When the transaction has been configured to be returned following a BRESP acceptance event, the slave driver fills in the beats WDATA and places it on the RESPONSE port of the driver.

8. The user environment receives the completed transaction.

Send Feedback
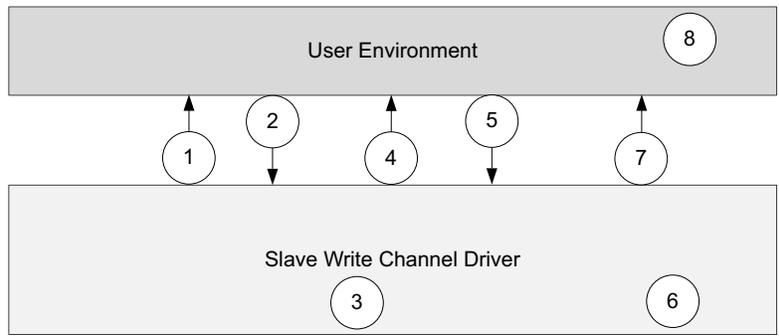
X18594-121316

Figure D-10: **Write Transaction Flow**

AWREADY Timing Flow

- AWREADY is generated independently of the AW and W channels.

- Configuration of the AWREADY is not from the same transaction or analysis port as the transaction that carries the AW and W payload.

- AWREADY configuration can change asynchronously to the AW/W channels.

- See the Configurable Ready Delays for different timing options.
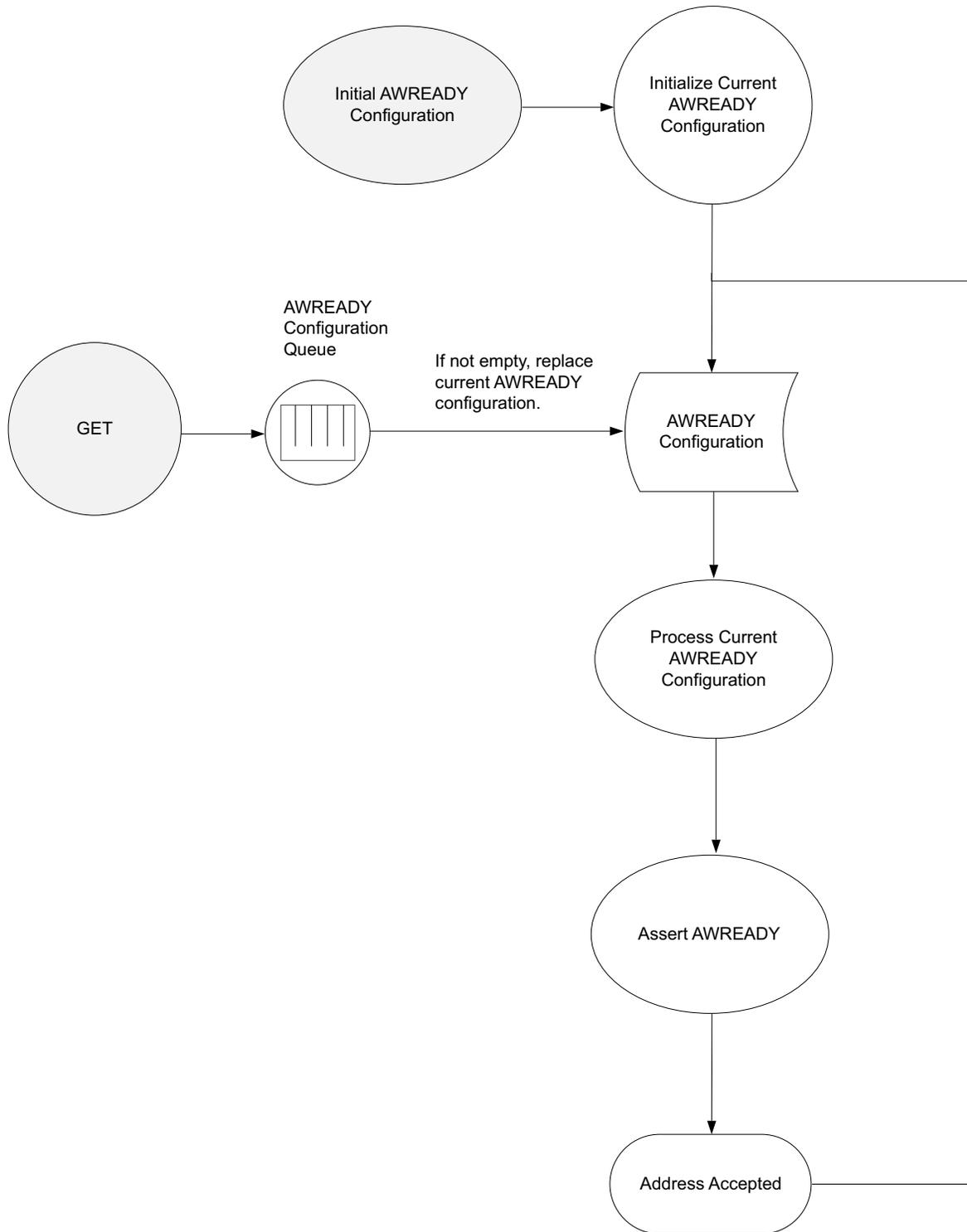
Figure D-11: **AWREADY Timing Flow**

www.xilinx.com

Send Feedback

WREADY Timing Flow

- WREADY is generated independently of the AW and W channels.

- Configuration of the WREADY is not from the same transaction or analysis port as the transaction that carries the AW and W payload.

- WREADY configuration can change asynchronously to the AW/W channels.

- See the Configurable Ready Delays for different timing options.
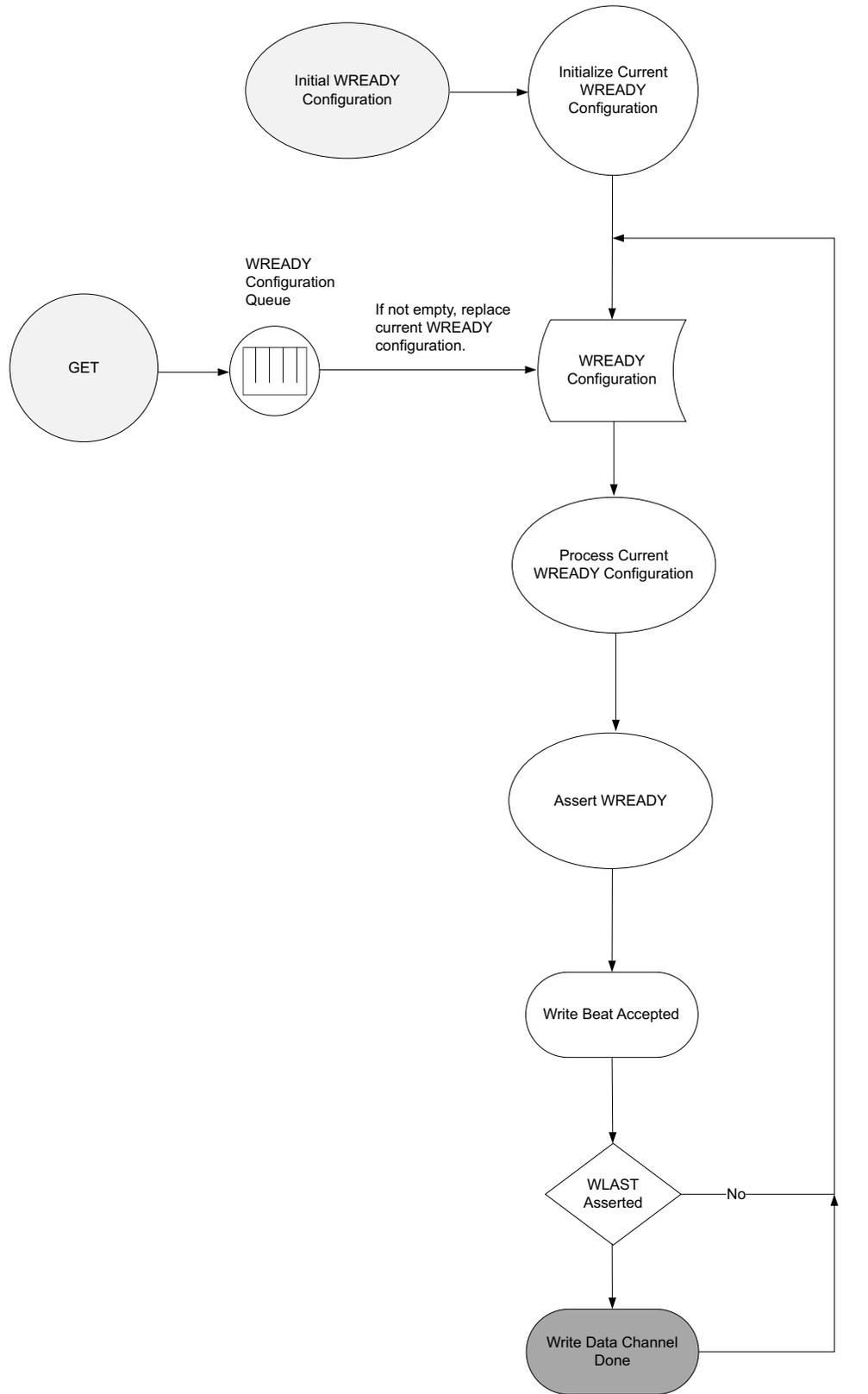
Figure D-12: **WREADY Timing Flow**

Read Data/Reaction Data Timing Flow Diagram
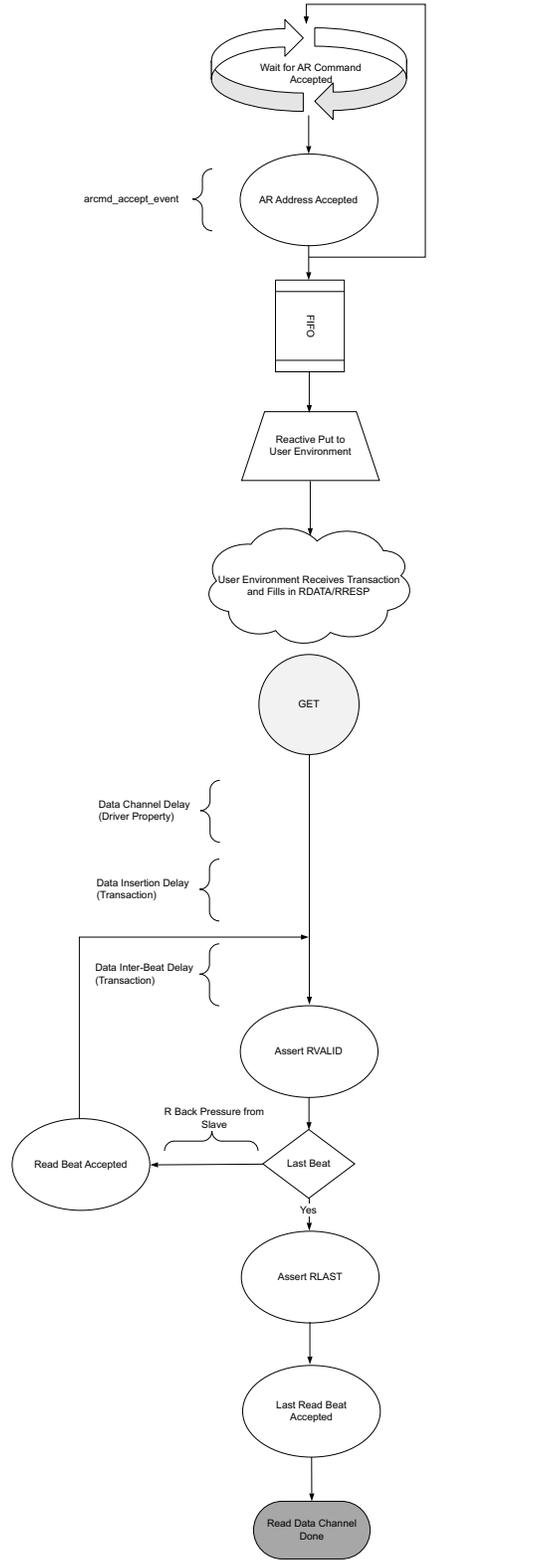
Figure D-13: **Read Data/Reaction Flow**

Send Feedback

Read Transaction Flow

An AXI slave read agent has two concurrent transaction flows:

- Address channel servicing

- Data channel serving

The address channel has its READY responses configured through the user environment. The data channel relies on the user environment for timing and payload generation. The transaction flows through the read agent in the following steps:

1. Master read response driver performs a blocking get to the user environment through a `get_next_item`. Because the command has not yet been received the user environment must wait until the command has been received from the master.

2. The user environment performs a blocking get, `get_next_item`, on the reactive port of the driver.

3. The slave driver waits for an ARADDR command.

4. Only after the slave driver receives the completion ARADDR phase, it transfers the command object through the reactive port to the sequencer. The command information consists of the Command Information field with ARADDR, ARLEN, ARSIZE, ARID, etc.

5. The user environment creates a single transaction. The transaction contains the following:

   ◦ **Payload** – RDATA byte array and RRESP array

   ◦ **Slave Controlled Timing** – Inter-beat timing and data insertion delay

6. The driver pops the transaction from the REQUEST port and places it on a queue to be processed and driven on the interface. If the system was idle at the time, the driver starts the slave data channel delay timer.

7. After the expiration of the timer, the driver processes the beats of data to be driven on the interface. In the case of multiple transactions pending and the slave being configured to support read, it interleaves the beats of the pending transactions.

8. Upon the acceptance of the last beat of a given command, if the transaction is configured to be returned to the user environment, the slave driver places it on the RESPONSE port to be sent.

9. The user environment receives the completed transaction.

ARREADY Timing Flow

- ARREADY is generated independently of the AR channel.

- Configuration of the ARREADY is not from the same transaction or analysis port as the transaction that caries the AR payload.

- ARREADY configuration can change asynchronously to the AR channels.

- See the Configurable Ready Delays for different timing options.
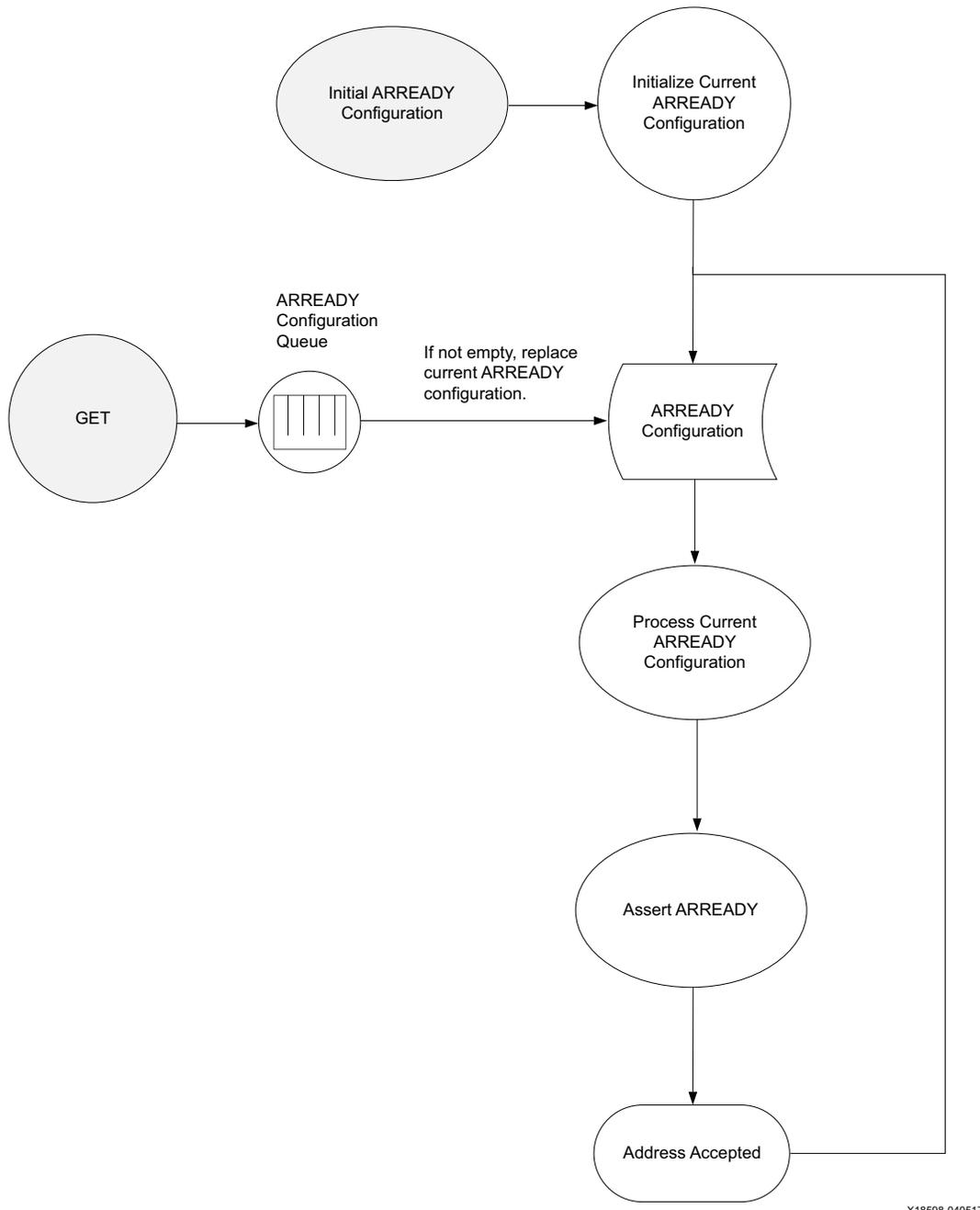
Figure D-14: **ARREADY Timing Flow**

Send Feedback

AXI Pass-Through Agent

When instantiating an AXI pass-through VIP, a pass-through agent has to be declared and constructed. Class `axi_passthrough_agent` contains other components that consist of the entire pass-through Verification IP. The pass-through VIP can be switched to runtime master or runtime slave modes. It includes master read driver, master write driver, slave read driver, slave write driver, and monitor.
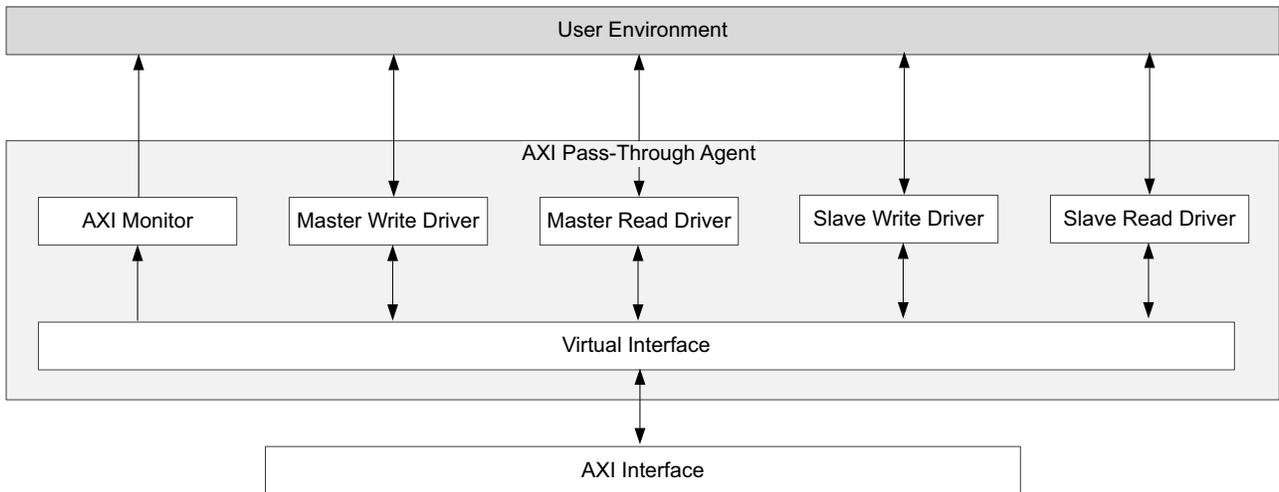
Figure D-15: **AXI Pass-Through VIP Agent**

AXI Master Read Driver

The same features as the AXI master read driver in `axi_mst_agent`.

AXI Master Write Driver

The same features as the AXI master write driver in `axi_mst_agent`.

AXI Slave Read Driver

The same features as the AXI slave read driver in `axi_slv_agent`.

AXI Slave Write Driver

The same features as the AXI slave write driver in `axi_slv_agent`.

AXI Monitor

The same features for both master/slave agent monitors.

Send Feedback

READY Generation

READY signals of write command channel, write data channel, write response channel, read command channel, and read data channel are generated independently from other attributes. The `axi_ready_gen` is the class used for READY generation.

Configurable Ready Delays

There is no one way that the READY signals on a channel are supposed to behave. There are no requirements for when READY should be asserted or how long READY should remain asserted, nor any that states that the READY must be asserted following a power up.

The control of the READY signal is set in the DRIVER of the given AGENT. For masters these are:

• RREADY

• BREADY

For slaves these are:

• AWREADY

• ARREADY

• WREADY

To control the generation of the READY signal there are two main configurations, however, to simplify the programming model these might be presented as different configurations.

Table D-3 shows the configurable READY delay description.

Table D-3: **Configurable Ready Delays**

Member Name	Default	Range	Description
use_variable_ranges	FALSE	0..1	When set TRUE, this property instructs the ready_gen class to generate a random value for high_time, low_time, and event_count based on the minimum/maximum ranges. When set FALSE, the ready_gen uses the programmed value of the high_time, low_time, and event_count.
max_low_time	5	$0..2^{32} - 1$	Used to constrain the low_time value. Indicates the maximum range of the low_time constraint.
min_low_time	0	$0..2^{32} - 1$	Used to constrain the low_time value. Indicates the minimum range of the low_time constraint.
low_time	2	$0..2^{32} - 1$	When used, indicates the number of cycles that *READY is driven Low.
max_high_time	5	$0..2^{32} - 1$	Used to constrain the high_time value. Indicates the maximum range of the high_time constraint.

Send Feedback

Table D-3: **Configurable Ready Delays** *(Cont'd)*

Member Name	Default	Range	Description
min_high_time	0	$0..2^{32} - 1$	Used to constrain the high_time value. Indicates the minimum range of the high_time constraint.
high_time	5	$0..2^{32} - 1$	When used, indicates the number of cycles that *READY is driven High.
max_event_count	1	$1..2^{32} - 1$	Used to constrain the event_count value. Indicates the maximum range of the event_count constraint.
min_event_count	1	$1..2^{32} - 1$	Used to constrain the event_count value. Indicates the minimum range of the event_count constraint.
event_count	1	$0..2^{32} - 1$	When used, indicates the number of handshakes that are sampled before the end of the policy.
event_count_reset	2000	$0..2^{32} - 1$	Watchdog wait time.

GEN_SINGLE/RAND_SINGLE (Default Policy)

While this policy is active, it drives the *READY signal 0 for `low_time` cycles and then drives 1 until one handshake occurs on this channel. The policy repeats until the channel is given a different policy.
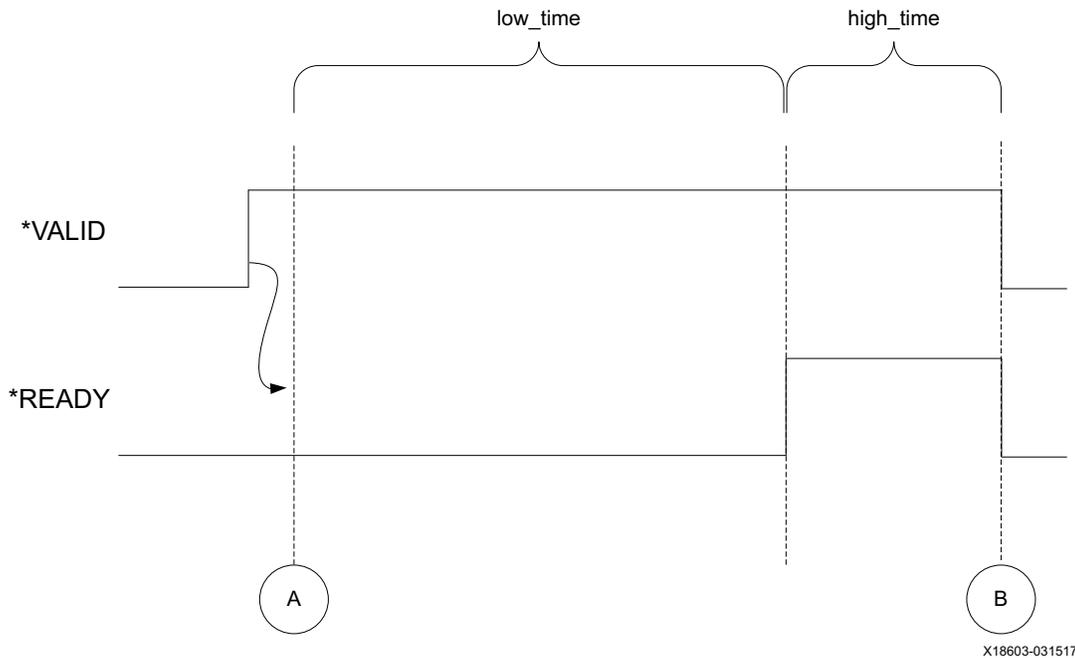

Figure D-16: **GEN_SINGLE/RAND_SINGLE**

GEN_OSC/RAND_OSC – Assert and Remain Asserted for a Number of Cycles

When this policy is active, it drives the *READY signal 0 for `low_time` cycles and then drives 1 for `high_time` cycles.

Note: The *READY does not drop until the specified number of cycles has occurred. The policy repeats until the channel is given a different policy.

Figure D-17 shows that following event A, there is a delay of `low_time` ACLKs, then READY is asserted. After `high_time` cycles of ACLK, READY is deasserted and the counter restarts at A.
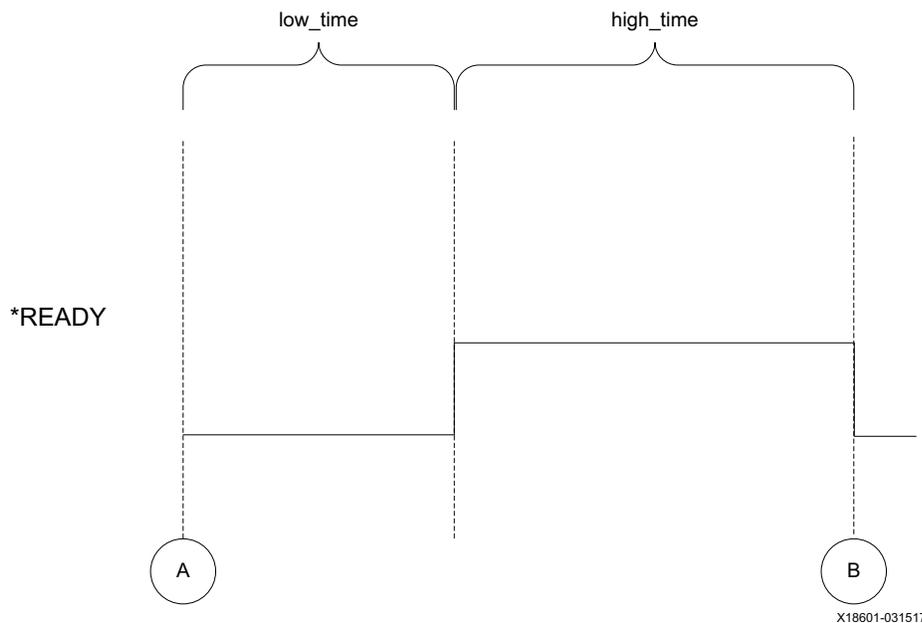
Figure D-17: **GEN_OSC/RAND_OSC**

GEN_EVENTS/RAND_EVENTS – Assert and Remain Asserted for a Number of Events

When this policy is active, it drives the *READY signal 0 for `low_time` cycles and then drives 1 until `event_count` handshakes occur.

Note: There is a built-in watchdog that triggers after the `event_cycle_count_reset` cycles and the programmed number of events has not been satisfied. This terminates that part of the policy. The policy repeats until the channel is given a different policy.

The value of `low_time` can range from 0 to 256 cycles. The READY remains asserted for N channel accept events, where N can be from 1 to N beats. This allows you to assert a READY after some number of cycles and keep it asserted indefinitely or for some number of events.

When attempting to model a self-draining FIFO, an event cycle count time reset is provided. This allows you to configure the READY to be deasserted after some number of events,

unless the event cycle count time has expired. In this case, the event count resets and the READY remains asserted for N more events.

Figure D-18 shows that following event A, there is a delay of `low_time` ACLKs, then the READY is asserted. It remains asserted for events E1 to E4 then deasserts since the event count is satisfied. The algorithm then restarts at A.
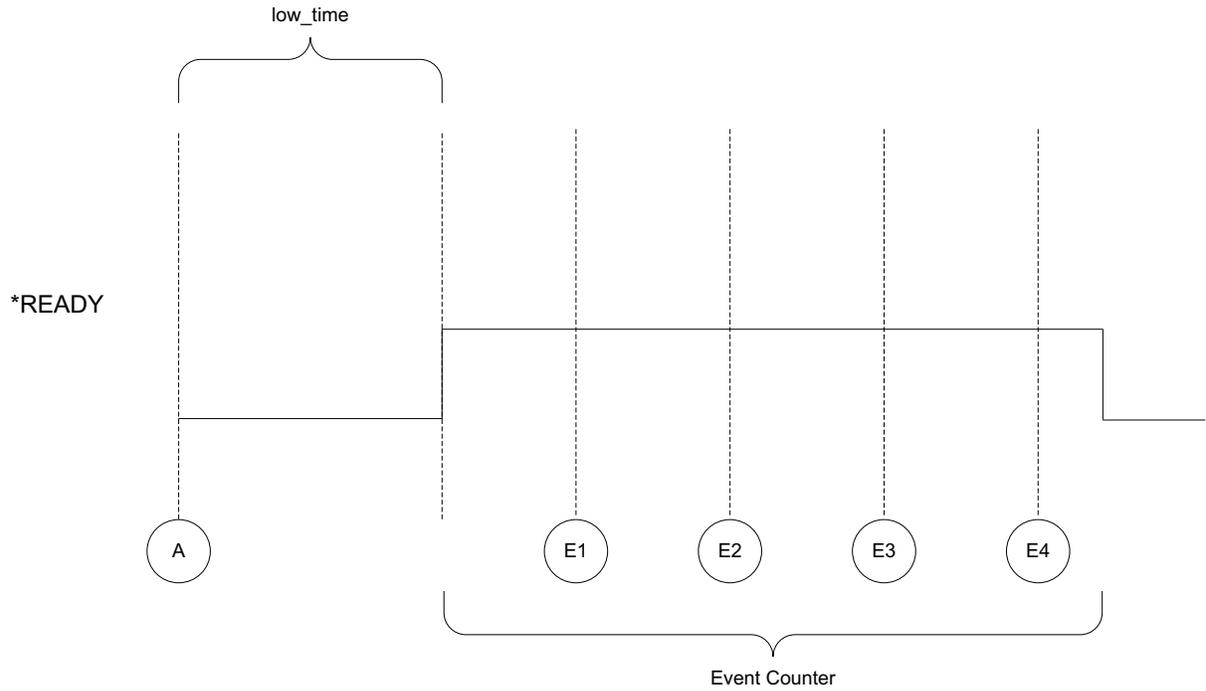
Figure D-18:   **GEN_EVENTS/RAND_EVENTS**

GEN_AFTER_VALID_SINGLE/RAND_AFTER_VALID_SINGLE

This policy is active when *VALID is detected to be asserted. When enabled, it drives the *READY Low for `low_time` and then asserts the *READY until one handshake has been detected. The policy repeats until the channel is given a different policy.
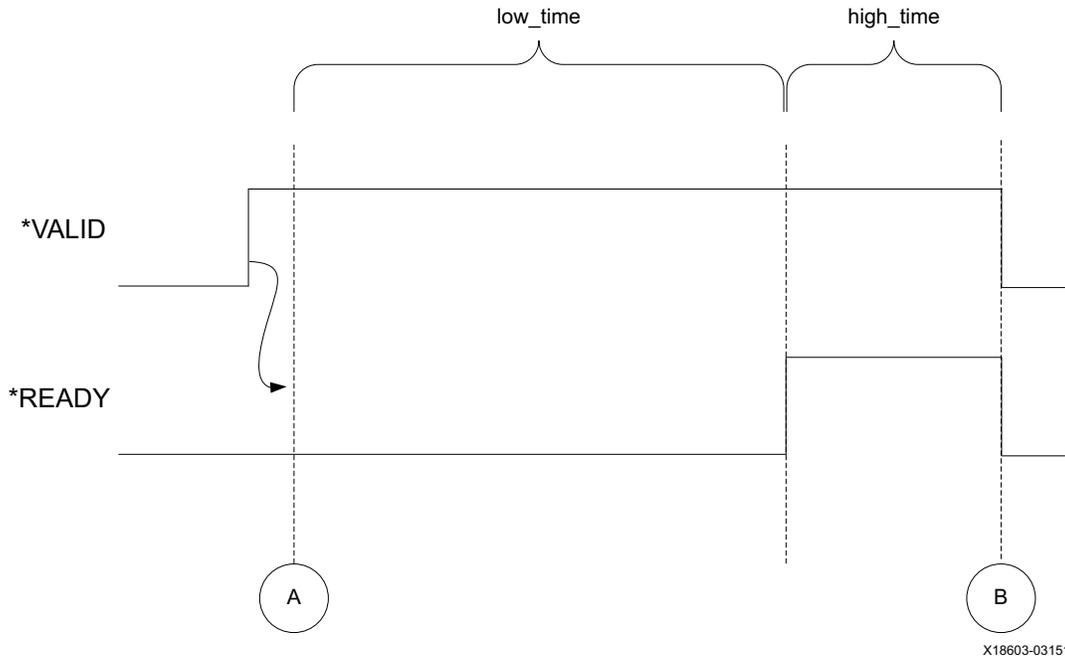
Figure D-19: **GEN_AFTER_VALID_SINGLE/RAND_AFTER_VALID_SINGLE**

GEN_AFTER_VALID_EVENTS/RAND_AFTER_VALID_EVENTS

This policy is active when *VALID is detected to be asserted. When enabled, it drives the *READY Low for `low_time` and then drives the *READY High until either `event_count` handshakes have been received OR `event_count_reset` number of cycles have passed. The policy repeats until the channel is given a different policy.
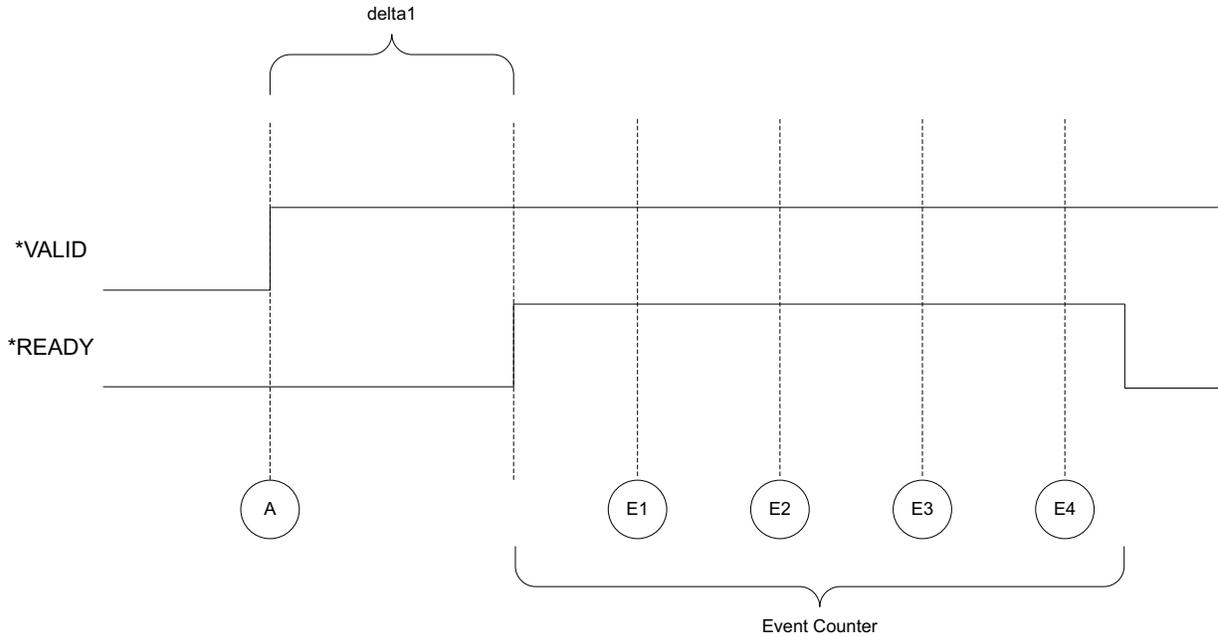
Figure D-20: **GEN_AFTER_VALID_EVENTS/RAND_AFTER_VALID_EVENTS**

GEN_AFTER_VALID_OSC/RAND_AFTER_VALID_OSC

This policy is active when the *VALID is detected to be asserted. When enabled, it drives the *READY Low for `low_time` and then drives the *READY High for `high_time`. The policy repeats until the channel is given a different policy.
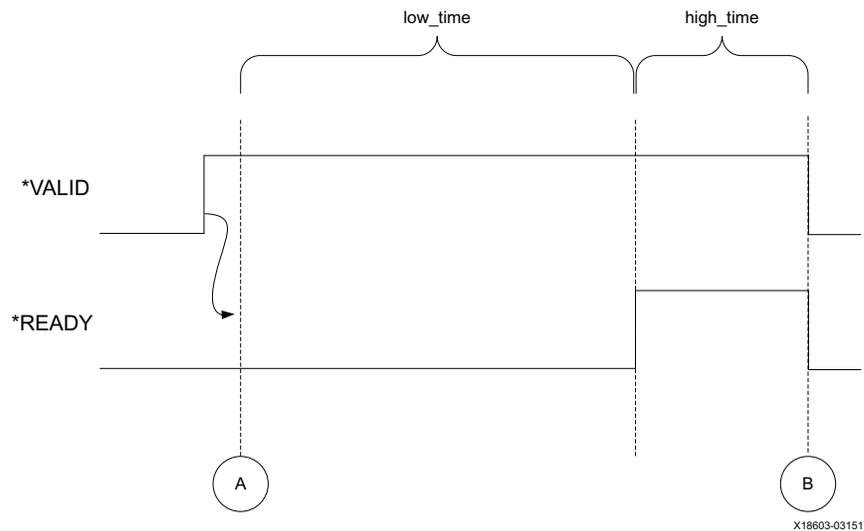
Figure D-21: **GEN_AFTER_VALID_OSC/RAND_AFTER_VALID_OSC**

Send Feedback

GEN_NO_BACKPRESSURE

This policy generates a `ready` signal staying asserted and does not change until the driver detects a policy change.

GEN_RANDOM

This policy randomly generates different policies including `low_time`, `high_time`, and `event_count`. When used, it randomly selects a new policy when the previous policy has completed.

This uses the minimum/maximum pairs for generating the value of the `low_time`, `high_time`, and `event_count` values.

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the AXI VIP, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the AXI VIP. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the AXI VIP

AR: 68234

Technical Support

Xilinx provides technical support at the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

•   Implement the solution in devices that are not defined in the documentation.

•   Customize the solution beyond that allowed in the product documentation.

•   Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see
Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support
resources, which you can filter and search to find information. To open the Xilinx
Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.

- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.

- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other
topics, which you can use to learn key concepts and address frequently asked questions. To
access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.

- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page
on the Xilinx website.

References

These documents provide supplemental material useful with this product guide:

1. Arm® AMBA® 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide (DUI0534B)

2. Instructions on how to download the Arm AMBA AXI specifications are at Arm AMBA Specifications. See the:

   ◦ AMBA AXI4-Stream Protocol Specification

   ◦ AMBA AXI Protocol v2.0 Specification

3. *Vivado Design Suite User Guide: AXI Reference Guide* (UG1037)

4. *AXI Protocol Checker LogiCORE™ IP Product Guide* (PG101)

5. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)

6. *Vivado Design Suite User Guide: Designing with IP* (UG896)

7. *Vivado Design Suite User Guide: Getting Started* (UG910)

8. *Vivado Design Suite User Guide: Logic Simulation* (UG900)

9. *ISE to Vivado Design Suite Migration Guide* (UG911)

10. *Vivado Design Suite User Guide: Implementation* (UG904)

11. *LogiCORE IP AXI Interconnect Product Guide* (PG059)

12. AXI VIP API Documentation

Note: VIP API documentation source codes are different from the Install area implementation codes, refer to the Install area for the source codes.

Send Feedback

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/30/2019	1.1	Updated API documentation link in References.
05/22/2019	1.1	• Added sim_ready_gen and sim_memory in Table 6-1. • Updated Figure 6-3. • Updated API documentation link in References.
12/05/2018	1.1	• Updated backdoor_memory_write and backdoor_memory_read arrows for Figure 4-8. • Updated Finding the AXI VIP Hierarchy Path in IP Integrator section. • Corrected generic_tb file name and added description to modes in Multiple Simulation Sets. • Updated GEN_RANDOM heading in AXI VIP Agent and Flow Methodology.
04/04/2018	1.1	Updated to latest Vivado Design Suite.
12/20/2017	1.1	• Updated NARROW description in Product Specification chapter.. • Added Issue Capability section in Test Bench chapter.
10/04/2017	1.1	• Added Note #6 in IP Facts table. • Updated SUPPORTS_NARROW, HAS_LOCK, HAS_WSTRB, HAS_BRESP, and HAS_RRESP descriptions in AXI VIP User Parameters table. • Added ARESET_XCHECK and XILINX_AXI_ERRM_RESET_PULSE_WIDTH in Xilinx Configuration Checks and Descriptions table. • Updated Arm reference 1. • Updated Overview section in Example Design chapter. • Updated code in Create Ready Signal section in Test Bench chapter. • Updated Must Haves section and added Reactive Ports for the AXI Slave VIP section in the Test Bench chapter. • Added GEN_NO_BACKPRESSURE section to Configurable Ready Delays section. • Added APIs to axi_vip_v1_1_top APIs appendix.
06/07/2017	1.0	• Added note #5-7 in IP Facts table. • Added description in Multiple Simulation Sets section in Test Bench chapter. • Added type definition description in AXI Pass-Through VIP section in Test Bench chapter.
04/05/2017	1.0	Initial Xilinx release.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.