AXI4-Stream Verification IP v1.0

LogiCORE IP Product Guide

Vivado Design Suite

PG277 June 7, 2017

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

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

The AXI4-Stream VIP is unencrypted SystemVerilog source that is comprised of a SystemVerilog class library and synthesizable RTL.

The embedded RTL interface is controlled by the AXI4-Stream VIP through a virtual interface. AXI4-Stream transactions are constructed in the customer's verification environment and passed to the AXI4-Stream driver class. The driver class then manages the timing and driving the content on the interface.

Features

- Supports the following widths:
  - Data widths up to 512 bytes
  - ID widths up to 32 bits
  - DEST widths up to 32 bits
- ARM-based transaction-level protocol checking for tools that support assertion property
- Behavioral SystemVerilog Syntax
- SystemVerilog class-based API

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family(1)	UltraScale+™, UltraScale™, Zynq®-7000 All Programmable SoC, 7 series FPGAs
Supported User Interfaces	AXI4-Stream
Resources	N/A
Provided with Core	
Design Files	N/A
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	N/A
Support	
Provided by Xilinx at the Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog.
2. For the supported versions of the 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 AXI4-Stream interfaces.
4. To take advantage of the full features of this IP, it requires simulators supporting advanced simulation capabilities.
5. The AXI4-Stream VIP can only act as a protocol checker when contained within a VHDL hierarchy.
6. Do not import two different revisions/versions of the `axi4stream_vip` packages. This causes elaboration failures.
7. All AXI4-Stream VIP and parents to the AXI4-Stream VIP must be upgraded to the latest version.

Overview

The Xilinx® LogiCORE™ AXI4-Stream Verification IP (VIP) core is used in the following manner:

- Generating master AXI4-Stream commands and write payload
- Generating slave AXI4-Stream read payload and write responses
- Checking protocol compliance of AXI4-Stream transactions

The AXI4-Stream VIP can be configured in three different modes:

- AXI4-Stream master VIP
- AXI4-Stream slave VIP
- AXI4-Stream pass-through VIP

Figure 1-1 shows the AXI4-Stream master VIP which generates AXI4-Stream payloads and sends it to the AXI4-Stream system.

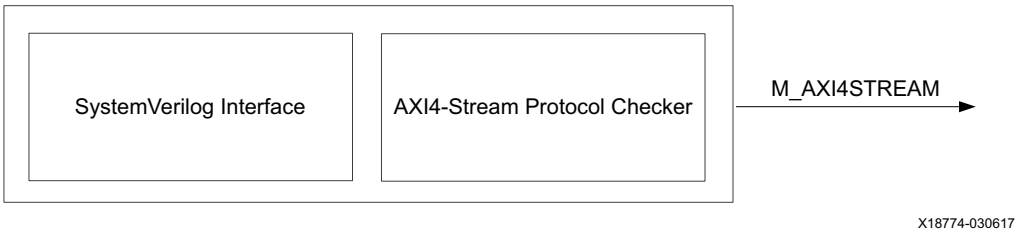

Figure 1-1: **AXI4-Stream Master VIP**

Figure 1-2 shows the AXI4-Stream slave VIP which responds to the AXI4-Stream and generates a `Ready` signal.

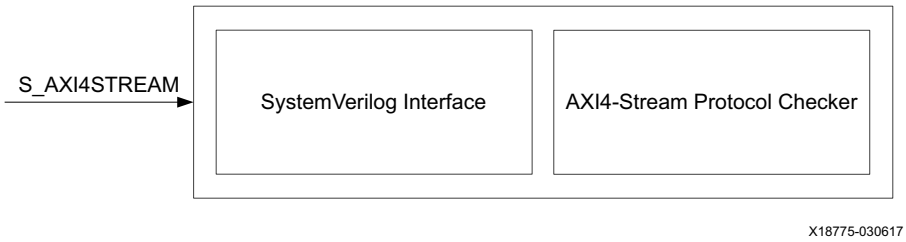

Figure 1-2: **AXI4-Stream Slave VIP**

Figure 1-3 shows the AXI4-Stream pass-through VIP which protocol checks all AXI4-Stream transactions that pass through it. The IP can be configured to behave in the following modes:

- Monitor only
- Master
- Slave

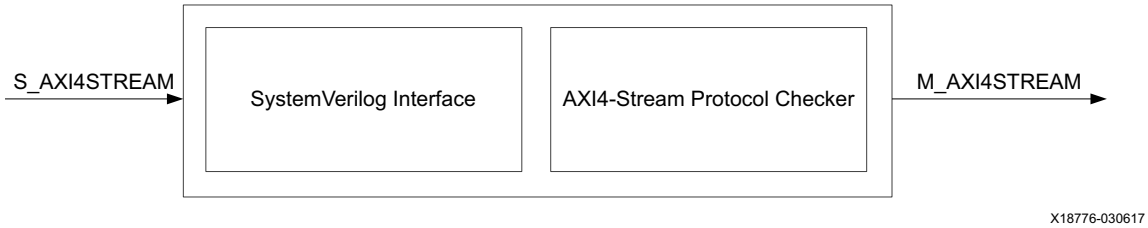

Figure 1-3: **AXI4-Stream Pass-Through VIP**

Feature Summary

- Supports AXI4-Stream interface
- Configurable as an AXI4-Stream master, AXI4-Stream slave, and in pass-through mode
- Configurable simulation messaging
- Provides simulation AXI4-Stream protocol checking
- An example design that demonstrates the usage of this VIP is available for reference

Applications

The AXI4-Stream VIP is for verification and system engineers who want to:

- Monitor transactions between two AXI4-Stream connections
- Generate AXI4-Stream transactions
- Check for AXI4-Stream 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

Standards

The AXI4-Stream interfaces conform to the ARM® Advanced Microcontroller Bus Architecture (AMBA®) AXI4-Stream version 4 specification [Ref 2].

Performance

The AXI4-Stream VIP core synthesizes to wires and does not impact performance.

User Parameters

Table 2-1 shows the AXI4-Stream VIP core user parameters.

Table 2-1: **AXI4-Stream VIP User Parameters**

Parameter Name	Format/Range	Default Value	Description
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.
HAS_TREADY	Type: long Value range: 0, 1	0	Used to control the enablement of the TREADY ports.
TDATA_WIDTH	Type: long Value range: 0..512	1	Width of the *_AXI4STREAM_tdata
HAS_TUSER_BITS_PER_BYTE	Type: long value range: {0, 1}	0	Used to control whether user bits per byte is ON or OFF. 1: user bits per byte, this means tuser size are per byte based 0: user bits per transfer, this means tuser size are per transfer based

Table 2-1: **AXI4-Stream VIP User Parameters *(Cont'd)***

Parameter Name	Format/Range	Default Value	Description
TUSER_BITS_PER_BYTE	Type: long value range: {0, 32}	0	When HAS_TUSER_BITS_PER_BYTE is 1, then the value of TUSER is calculated. TUSER_WIDTH = (TUSER_BITS_PER_BYTE × TDATA_WIDTH)
HAS_TSTRB	Type: long Value range: 0, 1	0	Used to control the enablement of the TSTRB ports.
HAS_TKEEP	Type: long Value range: 0, 1	0	Used to control the enablement of the TKEEP ports.
HAS_TLAST	Type: long Value range: 0, 1	0	Used to control the enablement of the TLAST ports.
TID_WIDTH	Type: long Value range: 0..32	0	Width of the *_AXI4STREAM_tid.
TDEST_WIDTH	Type: long Value range: 0..32	0	Width of the *_AXI4STREAM_tdest.
TUSER_WIDTH	Type: long Value range: 0..4096	0	Width of the *_AXI4STREAM_tuser
HAS_ACLKEN	Type: long Value range: {0, 1}	0	Used to control the enablement of the ACLKEN port. User parameter only (no corresponding model parameter).
HAS_ARESETN	Type: long Value range: {0, 1}	0	Used to control the enablement of the ARESETN port. User parameter only (no corresponding model parameter).

Port Descriptions

Table 2-2 shows the AXI4-Stream VIP independent port descriptions.

Table 2-2: **AXI4-Stream VIP Independent Port Descriptions**

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

Table 2-3 lists the interface signals for the AXI4-Stream VIP core in master or master pass-through mode.

Table 2-3: **AXI4-Stream Master or Pass-Through VIP Port Descriptions**

Signal Name	I/O	Default	Width	Description	Enablement
m_axi4stream_tdata	O		TDATA_WIDTH × 8	Streaming data	TDATA_WIDTH > 0
m_axi4stream_tdest	O		TDEST_WIDTH	Routing information for data stream	TDEST_WIDTH > 0
m_axi4stream_tid	O	0	TID_WIDTH	Stream data identifier	TID_WIDTH > 0
m_axi4stream_tkeep	I		TDATA_WIDTH	Byte qualifier (data byte or null byte)	HAS_KEEP
m_axi4stream_tlast	O	0b1	1	Last data beat of streaming packet	HAS_LAST
m_axi4stream_tready	I		1	Slave ready to accept stream data	HAS_TREADY
m_axi4stream_tstrb	O		TDATA_WIDTH	Byte qualifier of streaming data	HAS_STRB
m_axi4stream_tuser	O		TUSER_WIDTH	User defined sideband signals for stream	TUSER_WIDTH > 0
m_axi4stream_tvalid	O	Required	1	Streaming data valid	Always

Table 2-4 lists the interface signals for the AXI4-Stream VIP core when it has been configured to be in slave or slave pass-through mode.

Table 2-4: **AXI4-Stream Slave or Pass-Through VIP Port Descriptions**

Signal Name	I/O	Default	Width	Description	Enablement
s_axi4stream_tdata	O		TDATA_WIDTH × 8	Streaming data	TDATA_WIDTH > 0
s_axi4stream_tdest	O		TDEST_WIDTH	Routing information for data stream	TDEST_WIDTH >0
s_axi4stream_tid	O	0	TID_WIDTH	Stream data identifier	TID_WIDTH > 0
s_axi4stream_tkeep	I		TDATA_WIDTH	Byte qualifier (data byte or null byte)	HAS_KEEP
s_axi4stream_tlast	O	0b1	1	Last data beat of streaming packet	HAS_LAST
s_axi4stream_tready	I		1	Slave ready to accept stream data	HAS_TREADY
s_axi4stream_tstrb	O		TDATA_WIDTH	Byte qualifier of streaming data	HAS_STRB
s_axi4stream_tuser	O		TUSER_WIDTH	User defined sideband signals for stream	TUSER_WIDTH > 0
s_axi4stream_tvalid	O		1	Streaming data valid	Always

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 3].

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

Name of Protocol Check	Description
AXI4STREAM_ERRM_TVALID_RESET	TVALID is Low for the first cycle after aresetn goes High. This assertion is not available when the system_resetn port is not enabled.
AXI4STREAM_ERRM_TID_STABLE	TID remains stable when TVALID is asserted, and TREADY is Low. This assertion is only valid if both TREADY and TID are enabled on the interface.
AXI4STREAM_ERRM_TDEST_STABLE	TDEST remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if both TREADY and TDEST are enabled on the interface.
AXI4STREAM_ERRM_TKEEP_STABLE	TKEEP remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if TDATA, TREADY, and TKEEP are enabled on the interface.
AXI4STREAM_ERRM_TDATA_STABLE	TDATA remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if both TREADY and TDATA are enabled on the interface.
AXI4STREAM_ERRM_TLAST_STABLE	TLAST remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if both TREADY and TLAST are enabled on the interface.
AXI4STREAM_ERRM_TSTRB_STABLE	TSTRB remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if TDATA, TREADY, and TSTRB are enabled on the interface
AXI4STREAM_ERRM_TVALID_STABLE	When TVALID is asserted, it must remain asserted until TREADY is High. This assertion is only valid if TREADY is enabled on the interface.
AXI4STREAM_ERRM_TREADY_MAX_WAIT	Xilinx recommends that TREADY is asserted within MAXWAITS cycles of TVALID being asserted. This assertion is only valid if TREADY is enabled on the interface.
AXI4STREAM_ERRM_TUSER_STABLE	TUSER remains stable when TVALID is asserted and TREADY is Low. This assertion is only valid if both TREADY and TUSER are enabled on the interface.
AXI4STREAM_ERRM_TKEEP_STABLE	If TKEEP is deasserted, then TSTRB must also be deasserted. This assertion is only valid it TDATA, TSTRB, and TKEEP are enabled on the interface.

Designing with the Core

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

General Design Guidelines

The AXI4-Stream VIP core should be inserted into a system as shown in Figure 3-1 for AXI4-Stream master VIP, Figure 3-2 for AXI4-Stream slave VIP, and Figure 3-3 for AXI4-Stream pass-through VIP.

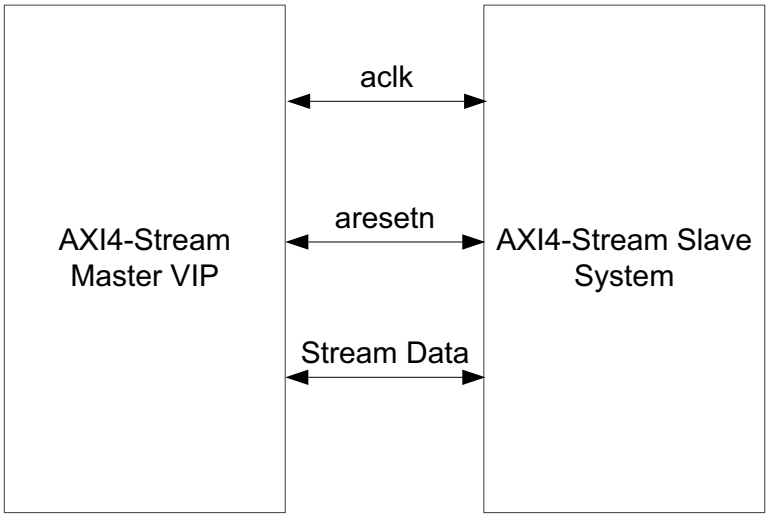

Figure 3-1: **AXI4-Stream Master VIP Example Topology**

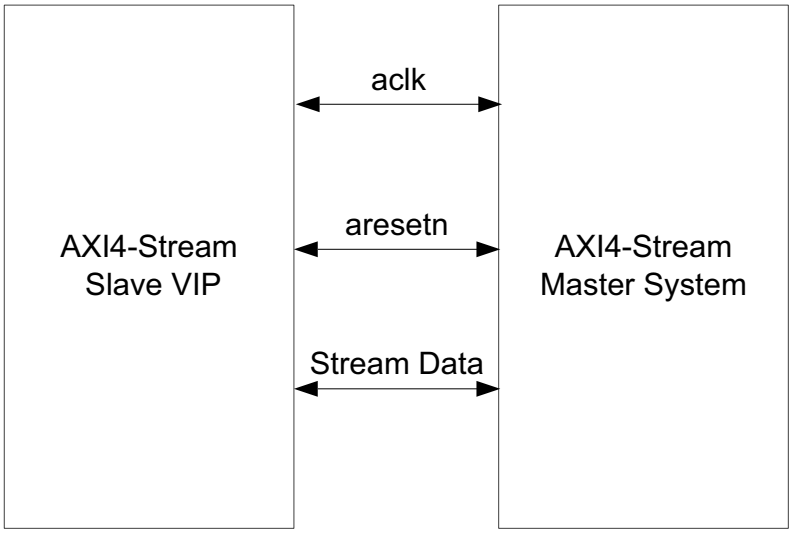

Figure 3-2: **AXI4-Stream Slave VIP Example Topology**

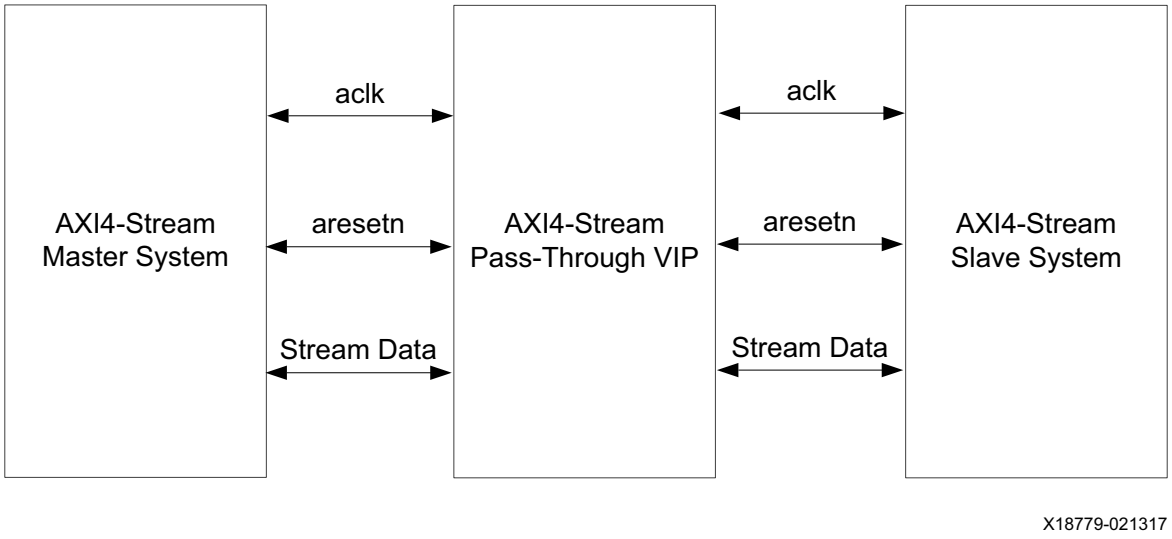

Figure 3-3: **AXI4-Stream Pass-Through VIP Example Topology**

Clocking

This section is not applicable for this IP core.

Resets

The AXI4-Stream VIP requires one active-Low reset, `aresetn`. The reset is synchronous to `aclk`.

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 3]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 5]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 6]

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 3] 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 4] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 5].

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

Figure 4-1 shows the AXI4-Stream VIP Vivado IDE **Component Name** tab configuration screen.

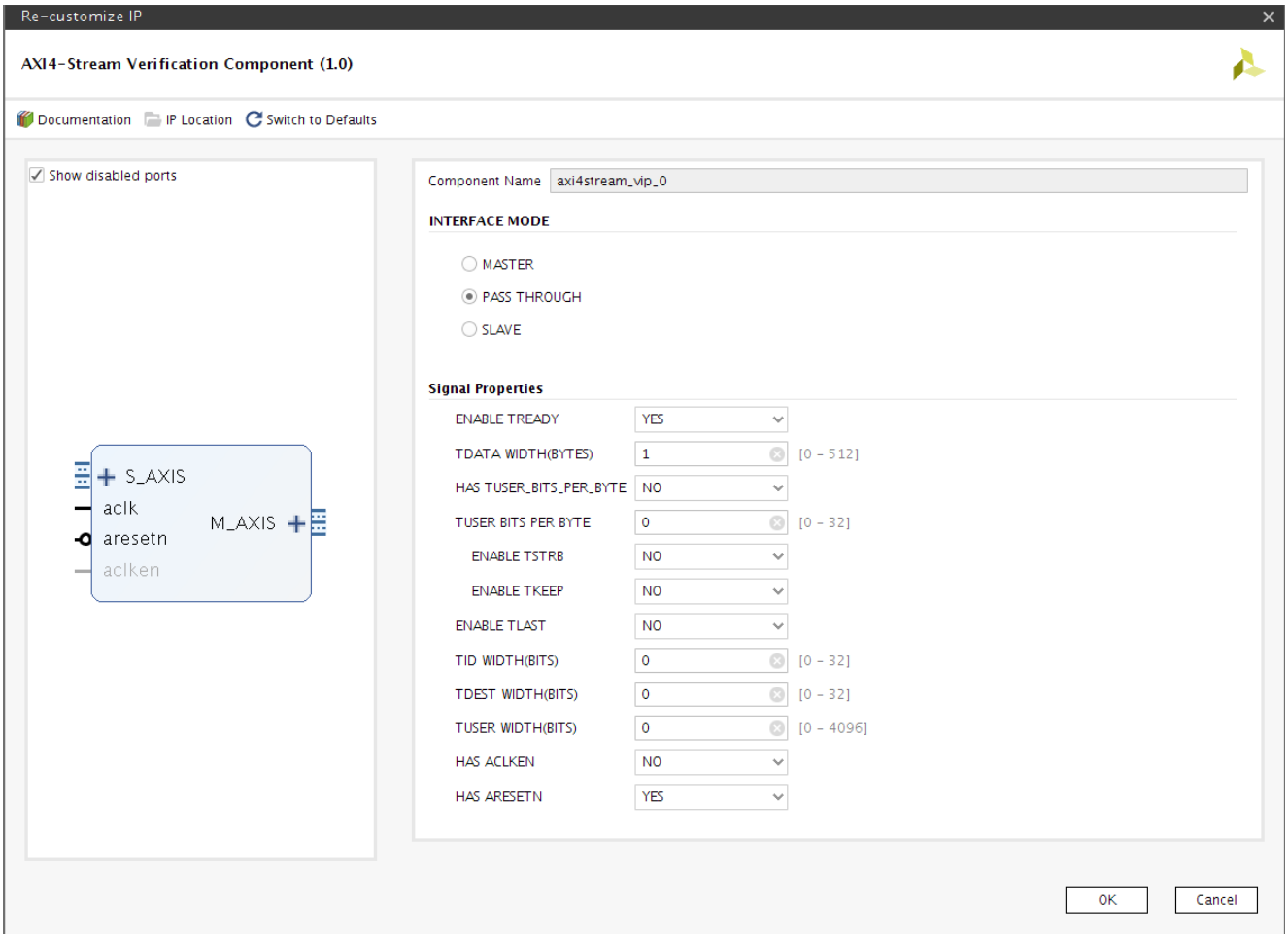

Figure 4-1: **AXI4-Stream VIP Customize IP – Component Name Tab**

Note: 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** – Control the mode of protocol to be configured as master, slave, or pass-through.
- **Signal Properties** – Select the specific signal properties.

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 4].

The AXI4-Stream VIP deliverables are organized in the directory `<project_name>/<project_name>.srcs/sources_1/ip/<component_name>` and are designed 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}	AXI4-Stream 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	AXI4-Stream VIP source files.
synth/<component_name>.sv	AXI4-Stream VIP generated top-level file for synthesis. Optional, generated if synthesis target selected.
sim/<component_name>.sv	AXI4-Stream VIP generated top-level file for simulation. Optional, generated if simulation target selected.

AXI4-Stream VIP in Vivado IP Integrator

This section contains information about how to use the AXI4-Stream VIP in a design and test bench environment. Figure 4-2 shows a possible design with the AXI4-Stream VIPs.

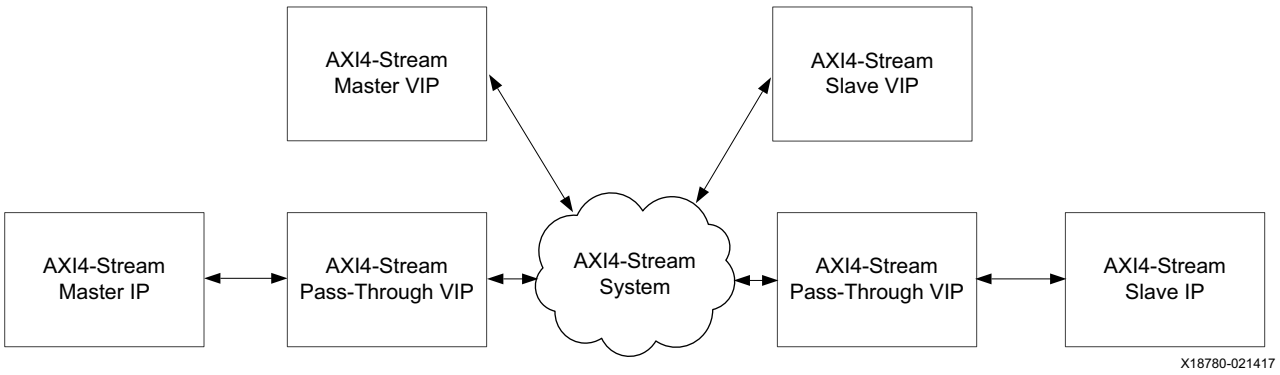

Figure 4-2: **AXI4-Stream VIP Design**

The AXI4-Stream VIP uses similar naming and structures as the Universal Verification Methodology (UVM) for core design. It is coded in SystemVerilog. The AXI4-Stream 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 AXI4-Stream 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.

AXI4-Stream Master VIP

Figure 4-3 shows the AXI4-Stream master VIP with its test bench. The test bench has three parts:

- User environment
- Master agent
- AXI4-Stream master VIP

The user environment and master agent are in the dynamic world while the AXI4-Stream master VIP is in the static world. The user environment communicates with the master agent and the master agent communicates with the AXI4-Stream VIP interface though a virtual interface.

The master agent has three class members:

- Master driver
- Monitor
- Virtual interface

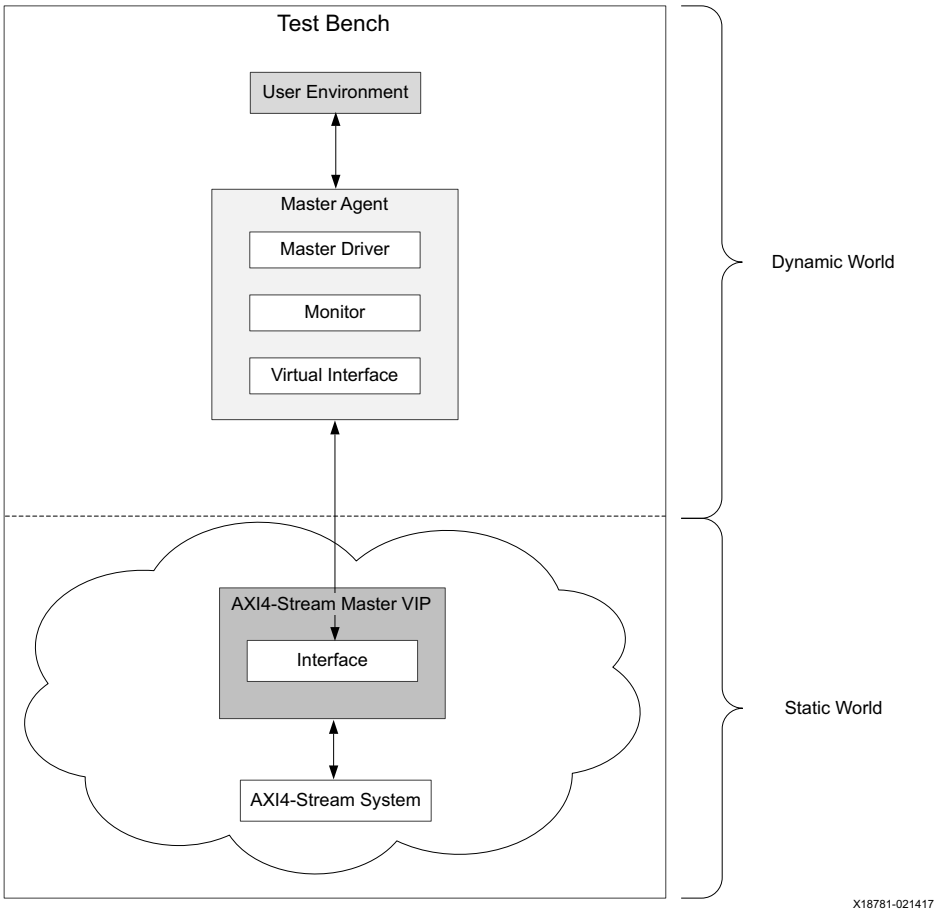

Figure 4-3: **AXI4-Stream VIP Master Test Bench**

Figure 4-4 shows how stream data is constructed and sent to the AXI4-Stream VIP interface. The user environment first declares a variable of stream transaction and the master driver constructs it with a `new()` function. The user environment then sets up the stream transaction members by either filling in or randomization. The master write then sends it to the AXI4-Stream VIP interface through a virtual interface and the AXI4-Stream VIP interface pins start to wiggle.

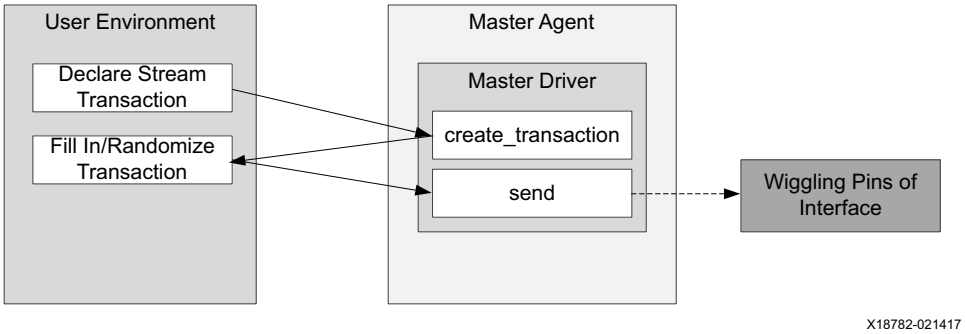

Figure 4-4: **Stream Transaction Flow**

AXI4-Stream Slave VIP

Figure 4-5 shows the AXI4-Stream slave VIP with its test bench. The test bench has three parts:

- User environment
- Slave agent
- AXI4-Stream slave VIP

The user environment and slave agent are in the dynamic world, while the AXI4-Stream slave VIP is in the static world. The user environment communicates with the slave agent and the slave agent communicates with the AXI4-Stream VIP interface though a virtual interface. The slave agent has three class members:

- Slave driver
- Monitor
- Virtual interface

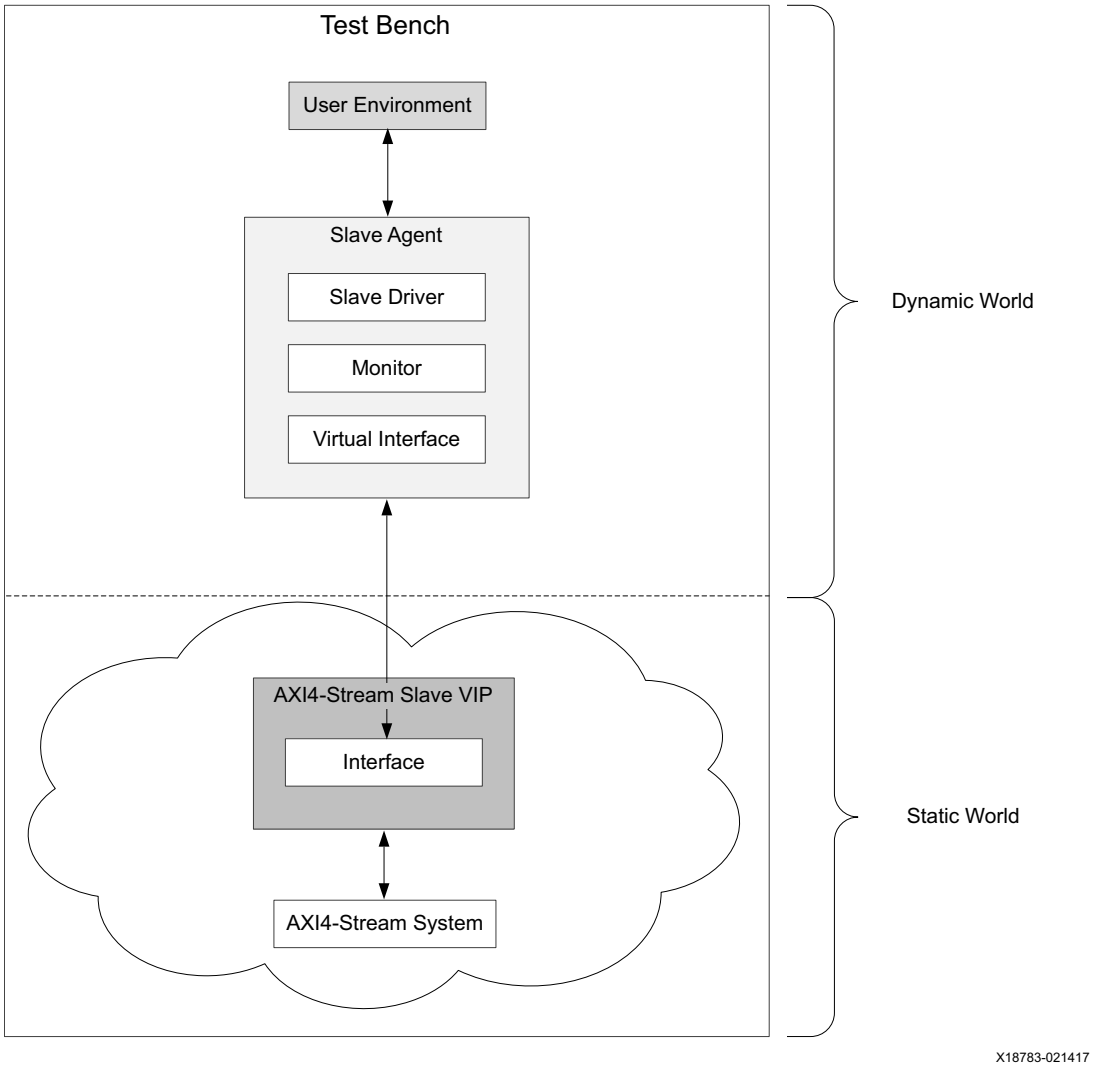

Figure 4-5: **AXI4-Stream VIP Slave Test Bench**

Figure 4-6 shows how a `ready` generation is constructed and sent to the AXI4-Stream VIP interface. The user environment first declares a variable of `ready`. The user environment then fills in the ready by either randomization or direct values.

The Slave Driver then sends it to the AXI4-Stream VIP interface through a virtual interface and the AXI4-Stream VIP interface related pins start to wiggle. If no `ready` is being created and generated in the user environment, the default randomized `ready` pattern is generated and sent to interface.

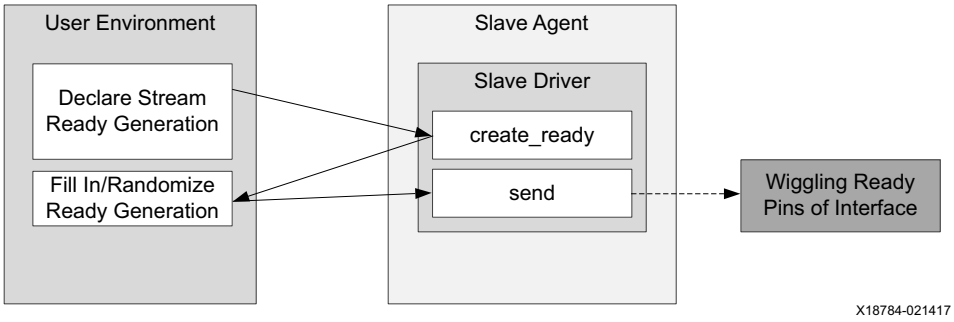

Figure 4-6: **Ready Generation Flow**

Multiple AXI4-Stream VIP

Figure 4-7 shows multiple AXI4-Stream VIPs in one design and its test bench. Similar to the single AXI4-Stream VIP, it has dynamic and static worlds that are bridged through a virtual interface.

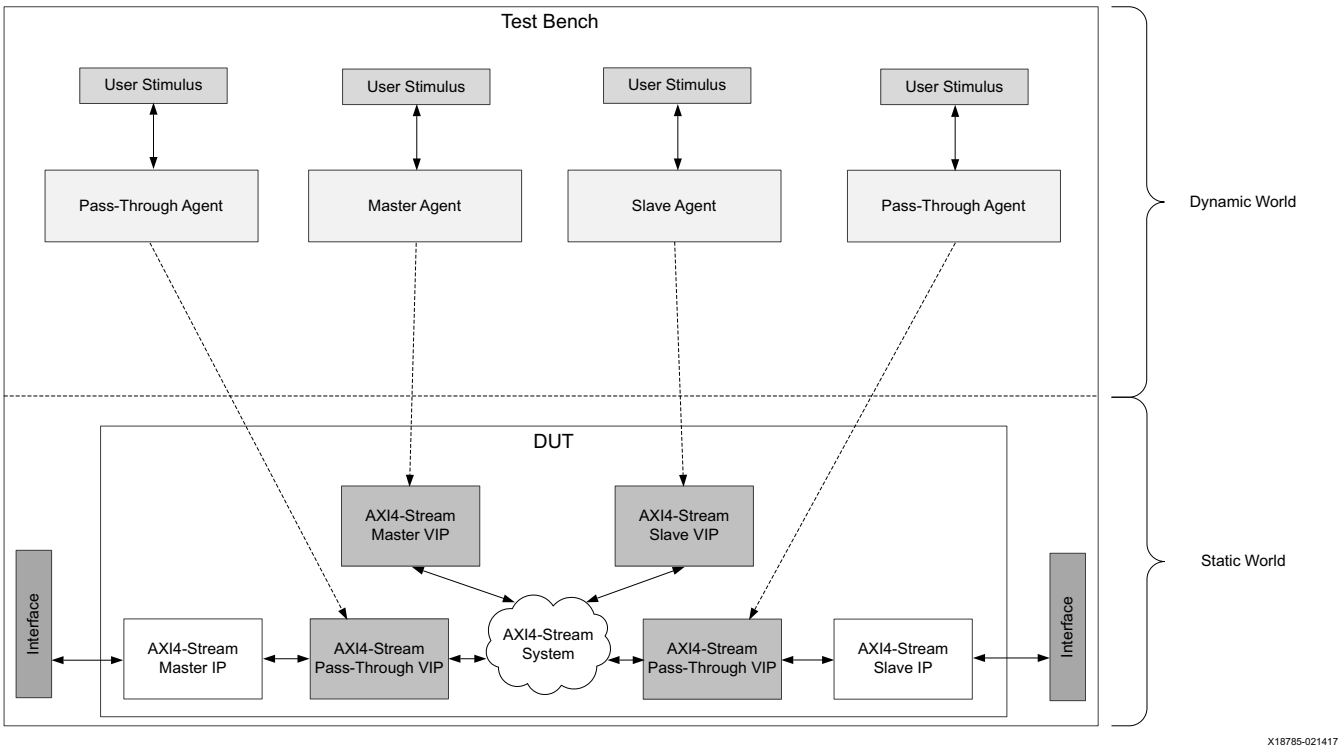

Figure 4-7: **Multiple AXI4-Stream VIP Test Bench**

Finding the AXI4-Stream VIP Hierarchy Path in IP Integrator

As mentioned earlier, the user environment has to declare the agent for the AXI4-Stream VIP. Also, the AXI4-Stream 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 AXI4-Stream VIP in the IP integrator.

1. Right-click **bd design** and select **Generate Output Products** (Figure 4-8).

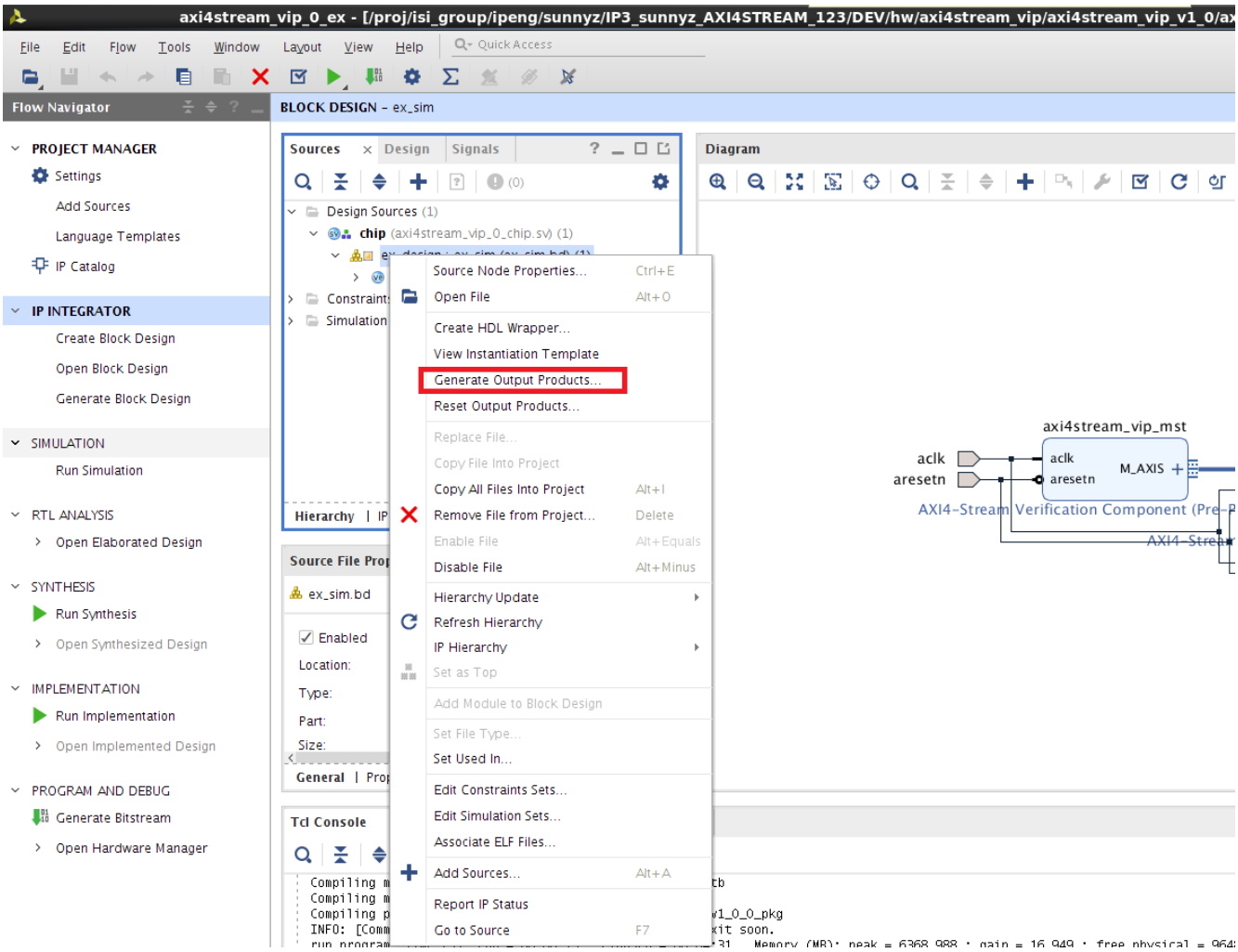

Figure 4-8: **Generate Output Products**

2. After the **General Output Products** is delivered, right-click **bd design** again and select **Create HDL Wrapper**.

   Note: AXI4-Stream VIP cores support only SystemVerilog language.

3. Figure 4-9 shows the complete hierarchy of the instances after the wrapper generates.

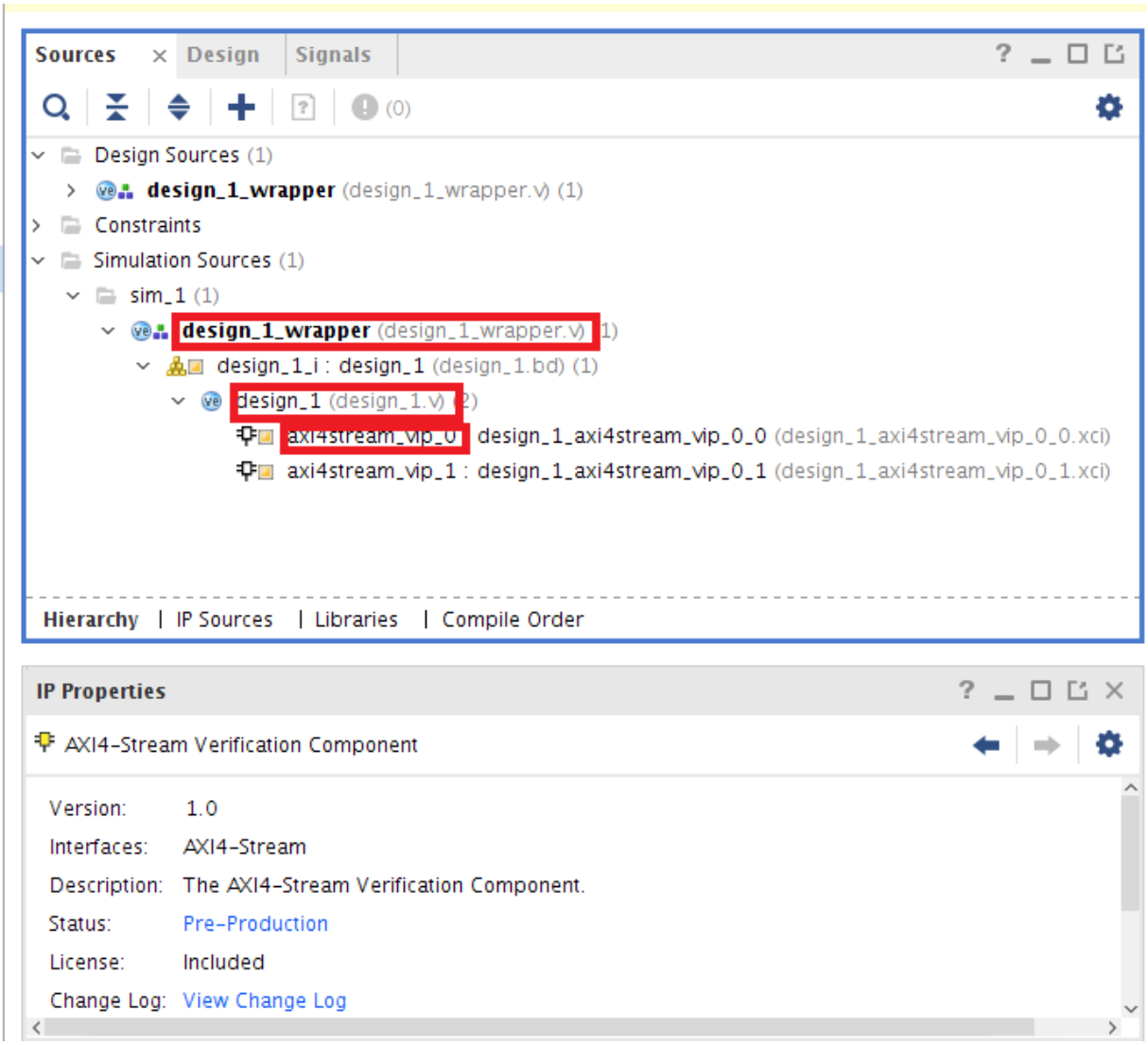

Figure 4-9: **Complete Design Hierarchy**

4. Use the generated wrapper as a DUT module in the test bench and the hierarchy path of these three AXI4-Stream VIPs are `DUT.design_1.axi4stream_vip_0.inst`, `DUT.design_1.axi4stream_vip_1.inst`, and `DUT.design_1.axi4stream_vip_2.inst`.
5. The best method to identify the VIP instance in the hierarchy is after the connection of all the IPs and the validation check. 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 AXI4-Stream VIP APIs.

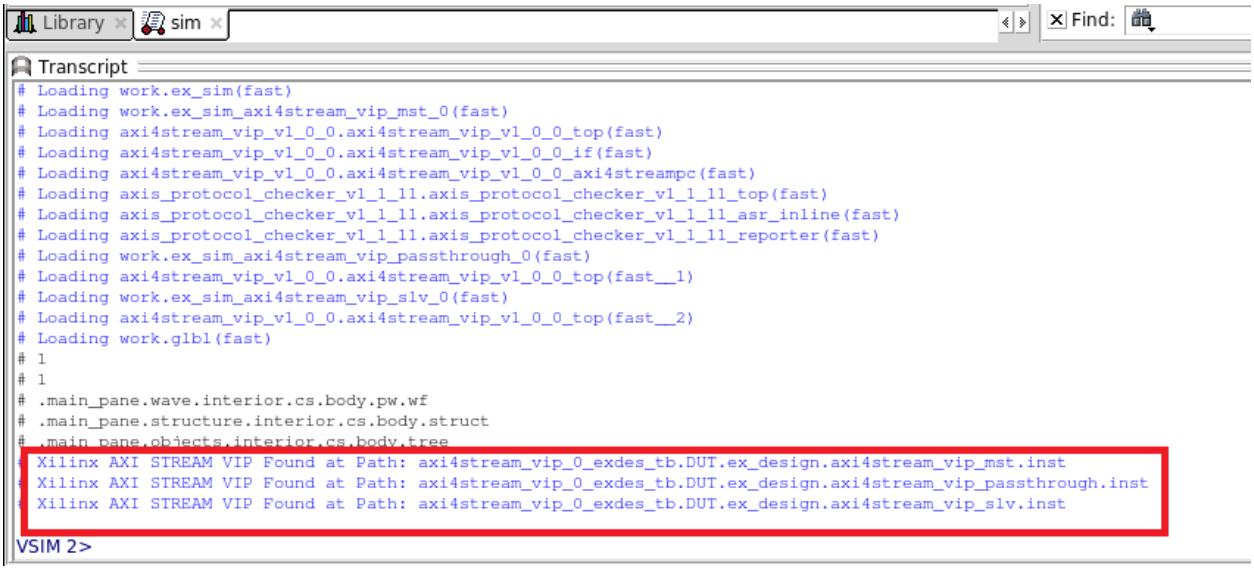

Figure 4-10: **AXI4-Stream VIP Instance in IP Integrator Design Hierarchy**

After the AXI4-Stream VIP is instantiated in the IP integrator design and its hierarchy path found, the next step is using the AXI4-Stream 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.

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 6].

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 AXI4-Stream VIP core is a verification IP set to synthesize as wires. There is no implementation for the AXI4-Stream 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 AXI4-Stream VIP. The AXI4-Stream VIP can only act as a protocol checker when contained within a VHDL hierarchy. Do not import two different revision/versions of the `axi4stream_vip` packages. All AXI4-Stream VIP and parents to the AXI4-Stream VIP must be upgraded to the latest version.*

Overview

Figure 5-1 shows the AXI4-Stream VIP example design.

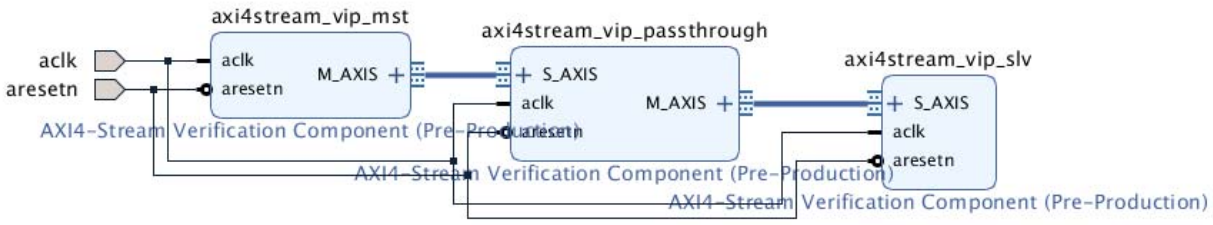

Figure 5-1: **AXI4-Stream VIP Example Design**

This section describes the example tests used to demonstrate the abilities of the AXI4-Stream VIP core. Example tests are delivered in SystemVerilog. The example design is available in the AXI4-Stream VIP installation area in the Tcl Console folder in an encrypted format.

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:

- AXI4-Stream VIP in master mode
- AXI4-Stream VIP in pass-through mode
- AXI4-Stream VIP in slave mode

The AXI4-Stream master VIP creates stream transactions and sends them to the AXI4-Stream pass-through VIP. The AXI4-Stream pass-through VIP receives stream transactions from the AXI4-Stream master VIP and sends them to the AXI4-Stream slave VIP. The AXI4-Stream slave VIP generates `ready` and sends the responses back to the AXI4-Stream pass-through VIP and then back to AXI4-Stream master VIP when it is configured to `ready`.

Monitors for the AXI4-Stream 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 AXI4-Stream master VIP against the AXI4-Stream pass-through VIP and the AXI4-Stream slave VIP against the AXI4-Stream pass-through VIP.

The AXI4-Stream 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 AXI4-Stream protocol can be accidentally violated. Xilinx recommends accessing all the members through the APIs instead of accessing them directly.

When the AXI4-Stream VIP is configured in pass-through mode, it can change to either master or slave mode in the runtime and then changed back to pass-through mode based on your requirements.

When it is switched to runtime master mode, it behaves exactly as an AXI4-Stream master VIP. When it is switched to runtime slave mode, it behaves exactly as an AXI4-Stream slave VIP.

IMPORTANT: *When the AXI4-Stream VIP is configured in pass-through mode, make sure all transactions has been cleaned out before switching modes. Examples of how to wait for the transactions finishing 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 the example from the Vivado IP catalog.
2. Open a new project and click **IP Catalog**.
3. Search for **AXI4-Stream Verification IP**. Double-click it, configure the IP, and generate the IP.
4. Right-click the IP and choose **Open IP Example Design**.

   Note: If you have the AXI4-Stream VIP as one component in the IP integrator design, right-click AXI4-Stream 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.

AXI4-Stream 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 AXI4-Stream VIP configuration. The following scenarios are covered in the example design.

- AXI4-Stream pass-through VIP in pass-through mode. The AXI4-Stream master VIP generates a simple sequential randomized stream transfer and passes them to the AXI4-Stream slave VIP.
- Switches AXI4-Stream pass-through VIP into the runtime master mode and generates a simple sequential randomized stream transfer and passes them to the AXI4-Stream slave VIP.
- Switches AXI4-Stream pass-through VIP into the runtime slave mode, The AXI4-Stream master VIP generates a simple sequential randomized stream transfer.

Useful Coding Guidelines and Examples

Must Haves in the Test Bench

While coding test bench for the AXI4-Stream VIP, the following requirements must be met. Otherwise, the AXI4-Stream VIP does not work.

1. Import the two required packages: `axi4stream_vip_v1_0_0_pkg` and `<component_name>_pkg`. The `axi4stream_vip_v1_0_0_pkg` includes agent classes and its subclasses for AXI4-Stream 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 on the test bench. Figure 6-1 shows the AXI4-Stream VIP with the specific packages to copy into your test bench.
2. Copy the code snippet into your test bench: see Figure 6-1

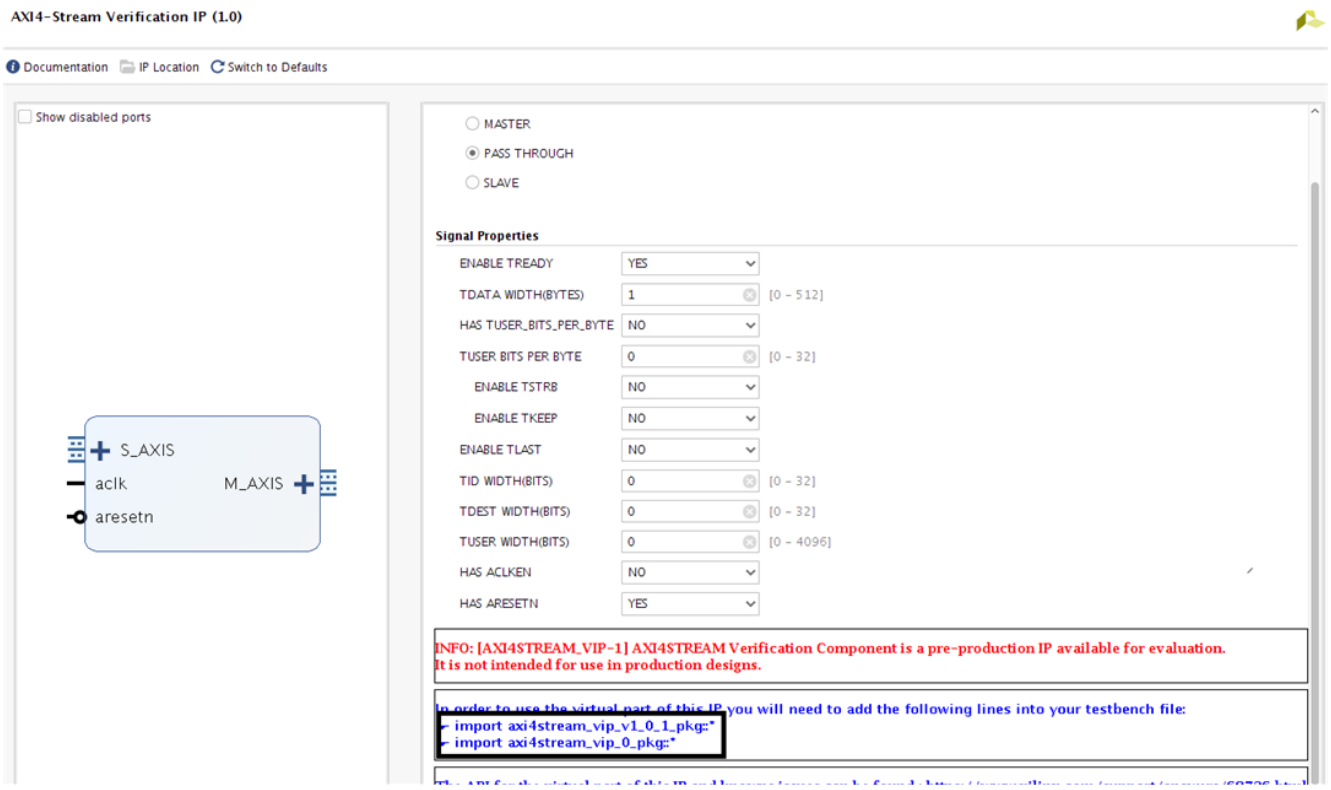

Figure 6-1: **Test Bench Package Codes**

In the example design, three AXI4-Stream VIPs are being instantiated so the total test bench has four packages to import which are listed below:

```
import axi4stream_vip_v1_0_0_pkg::*;
import ex_sim_axi4stream_vip_mst_0_pkg::*;
import ex_sim_axi4stream_vip_slv_0_pkg::*;
import ex_sim_axi4stream_vip_passthrough_0_pkg::*;
```

In the example design, the component name for each AXI4-Stream VIP are:

- AXI4-Stream master VIP is `ex_sim_axi4stream_vip_mst_0`
- AXI4-Stream slave VIP is `ex_sim_axi4stream_vip_slv_0`
- AXI4-Stream pass-through VIP is `ex_sim_axi4stream_vip_passthrough_0`

The corresponding packages are:

- `ex_sim_axi4stream_vip_mst_0_pkg`
- `ex_sim_axi4stream_vip_slv_0_pkg`
- `ex_sim_axi4stream_vip_passthrough_0_pkg`

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

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

4. Declare transaction class handles and ready class handles for use (when `tready` is High).

```
// write transaction created by master VIP
 axi4stream_transaction    wr_transaction;
// Ready signal created by slave VIP when TREADY is High
 axi4stream_ready_gen      ready_gen;
```

5. Declare agents. Normally, one agent for one AXI4-Stream VIP has to be declared. Even in the case when the AXI4-Stream VIP is being configured in pass-through mode and no monitor of it is needed. But, it might switch to master or slave mode in the runtime so it is better to have it ready.

   Therefore, in a design which has six AXI4-Stream VIPs, six component packages have to be imported, six agents have to be declared, constructed (new here), and the right interfaces have to be assigned. For master VIP, `component_name_mst_t` has to be declared, for slave VIP, `component_name_slv_t` has to be declared, and for pass-through VIP, `component_name_passthrough_t` has to be declared. When the agent is being constructed, special care has to be taken to assign the interface of the instance to the agent. See Chapter 4, Design Flow Steps to find the hierarchy path.

```
ex_sim_axi4stream_vip_mst_0_mst_t                                mst_agent;
  ex_sim_axi4stream_vip_slv_0_slv_t                                slv_agent;
 ex_sim_axi4stream_vip_passthrough_0_passthrough_t               passthrough_agent;
  //After declaration, new has to be done
  mst_agent = new("master vip agent",DUT.ex_design.axi4stream_vip_mst.inst.IF);
  slv_agent = new("slave vip agent",DUT.ex_design.axi4stream_vip_slv.inst.IF);
  passthrough_agent = new("passthrough vip agent",DUT.
ex_design.axi4stream_vip_passthrough.inst.IF);
```

6. When the agents are constructed, to start the agent, `start_master` has to be called for AXI4-Stream master VIP or AXI4-Stream pass-through VIP in the runtime master mode. The `start_slave` has to be called for AXI4-Stream slave VIP or AXI4-Stream pass-through VIP in the runtime slave mode.

   To stop the agent, `stop_master` has to be called for AXI4-Stream master VIP or AXI4-Stream pass-through VIP in the runtime master mode. The `stop_slave` has to be called for AXI4-Stream slave VIP or AXI4-Stream pass-through VIP in the runtime slave mode. Pass-through VIP can only be either in runtime master mode, runtime slave mode, or pass-through mode.

   Consequently, 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.

```
// start master/slave VIP agent
mst_agent.start_master();
slv_agent.start_slave();
…
// stop master/slave VIP agent
mst_agent.stop_master();
slv_agent.stop_slave();
…
 // passthrough VIP switch to run time master mode
 passthrough_agent.start_master();
   ….
 // passthrough VIP switch to run time slave mode
 passthrough_agent.stop_master();
 passthrough_agent.start_slave();
…
// passthrough VIP switch to run time master mode
 passthrough_agent.stop_slave();
 passthrough_agent.start_master();
```

7. When bus is in idle, it drives everything to zero. The cause is the AXI4-Stream protocol checker is being used in the AXI4-Stream VIP. Because the AXI4-Stream protocol checker is synthesizable, it does not have case equality which triggers false assertions. If this is not set, false assertions fire. For more information about the AXI4-Stream protocol checker, see the ARM® Advanced Microcontroller Bus Architecture (AMBA®) AXI version 4 specification [Ref 1].

```
mst_agent.vif_proxy.set_dummy_drive_type(XIL_AXI_VIF_DRIVE_NONE);
slv_agent.vif_proxy.set_dummy_drive_type(XIL_AXI_VIF_DRIVE_NONE);
passthrough_agent.vif_proxy.set_dummy_drive_type(XIL_AXI_VIF_DRIVE_NONE);
```

8. Create transaction. For the AXI4-Stream master VIP or AXI4-Stream pass-through VIP in runtime master mode, it has to create transaction, randomize it, and then send it to the VIP interface. The code shows how one transaction is created and sent to the VIP interface.

```
// Master agent create write transaction
 wr_transaction = mst_agent.wr_driver.create_transaction("write transaction");
// set_transaction_depth is used to setup maximum outstanding write transactions
 // for master VIP,if not set, default is 16 (optional)
 mst_agent.wr_driver.set_transaction_depth(2);
 // randomize the transaction
   WR_TRANSACTION_FAIL_1a: assert(wr_transaction.randomize());
 // send the transaction to VIP interface
   mst_agent.wr_driver.send(wr_transaction);
```

Optional Haves in the Test Bench

1. To create a `ready` signal, use the AXI4-Stream slave VIP or AXI4-Stream pass-through VIP in runtime slave mode. When `tready` is being configured to be enabled and if you want to create specific `ready` signals, use the `create ready`, set the low and high pattern, and send them to the VIP interface. Otherwise, default `ready` signals are generated by the AXI4-Stream VIP. The following codes show how to generate two `ready` with different policies of XIL_AXI4STREAM_READY_GEN_OSC and XIL_AXI4STREAM_READY_GEN_SINGLE. Also, you can call `set_use_variable_range()` to randomly generate the Low and High time of `ready`.

```
// slave agent driver create ready and set it up
    ready_gen = slv_agent.driver.create_ready("ready_gen");
    ready_gen.set_ready_policy(XIL_AXI4STREAM_READY_GEN_OSC);
      ready_gen.set_low(1);
      ready_gen.set_high(2);
      slv_agent.driver.send_tready(ready_gen);
    ready_gen = slv_agent.driver.create_ready("ready_gen 2");
    ready_gen.set_ready_policy(XIL_AXI4STREAM_READY_GEN_SINGLE);
    ready_gen.set_low_time(1);
    slv_agent.driver.send_tready(ready_gen);
```

2. While coding the test bench for the AXI4-Stream VIP, it is optional to enable the monitor. The codes are displayed for monitor and scoreboard purposes.

3. Declare monitor transaction and construct it for the scoreboard purpose. The following is a code snippet of a scoreboard which checks master VIP against pass-through VIP.

```
//monitor transaction from master VIP
  axi4stream_monitor_transaction           mst_monitor_transaction;
  //monitor transaction queue for master VIP
  axi4stream_monitor_transaction           master_moniter_transaction_queue[$];
  // size of master_moniter_transaction_queue
  xil_axi4stream_uint                     master_moniter_transaction_queue_size =0;
  //scoreboard transaction from master monitor transaction queue
  axi4stream_monitor_transaction          mst_scb_transaction;
  //monitor transaction from passthrough VIP
  axi4stream_monitor_transaction         passthrough_monitor_transaction;
  //monitor transaction queue for passthrough VIP for scoreboard 1
  axi4stream_monitor_transaction
passthrough_master_moniter_transaction_queue[$];
  //size of passthrough_master_moniter_transaction_queue;
  xil_axi4stream_uint
passthrough_master_moniter_transaction_queue_size =0;
  //scoreboard transaction from passthrough VIP monitor transaction queue
  axi4stream_monitor_transaction       passthrough_mst_scb_transaction;

// Master VIP monitor collect its interface information and put it into transaction
queue
  initial begin
    forever begin
      mst_agent.monitor.item_collected_port.get(mst_monitor_transaction);
      master_moniter_transaction_queue.push_back(mst_monitor_transaction);
      master_moniter_transaction_queue_size++;
    end
  end

// passthrough vip monitors all the transaction from interface and put then into
transaction queue
  initial begin
    forever begin
      passthrough_agent.monitor.item_collected_port.
get(passthrough_monitor_transaction);
      if (exdes_state != EXDES_PASSTHROUGH_SLAVE) begin

passthrough_master_moniter_transaction_queue.push_back(passthrough_monitor_transact
ion);
        passthrough_master_moniter_transaction_queue_size++;
      end
      if (exdes_state != EXDES_PASSTHROUGH_MASTER) begin

passthrough_slave_moniter_transaction_queue.push_back(passthrough_monitor_transacti
on);
        passthrough_slave_moniter_transaction_queue_size++;
      end
    end
  end

//simple scoreboard doing self checking
  //comparing transaction from master VIP monitor with transaction from passsthrough
VIP in slave side
  // if they are match, SUCCESS. else, ERROR
  initial begin
```

```
    forever begin
       wait (master_moniter_transaction_queue_size>0 ) begin
         mst_scb_transaction = master_moniter_transaction_queue.pop_front;
         master_moniter_transaction_queue_size--;
         wait( passthrough_slave_moniter_transaction_queue_size>0) begin
           passthrough_slv_scb_transaction =
passthrough_slave_moniter_transaction_queue.pop_front;
           passthrough_slave_moniter_transaction_queue_size--;
          if (passthrough_slv_scb_transaction.do_compare(mst_scb_transaction) == 0)
begin
            $display("Master VIP against passthrough VIP scoreboard : ERROR: Compare
failed");
             error_cnt++;
           end else begin
           $display("Master VIP against passthrough VIP scoreboard : SUCCESS: Compare
passed");
           end
           comparison_cnt++;
         end
       end
     end
   end
```

4. 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");
slv_agent.set_agent_tag("Slave VIP");
passthrough_agent.set_agent_tag("Passthrough 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);
slv_agent.set_verbosity(slv_agent_verbosity);
passthrough_agent.set_verbosity(passthrough_agent_verbosity);
```

Loop Construct Simple Example

While coding directed tests, for loops are typically employed to efficiently generate large volumes of stimulus for both the master and/or slave VIP. For example:

```
wr_transaction = mst_agent.driver. create_transaction("write transaction");
mst_agent.driver.set_transaction_depth(2);
for(int i = 0; i < 16; i++) begin
  WR_TRANSACTION_FAIL_1a: assert(wr_transaction.randomize());
  mst_agent.driver.send(wr_transaction);
end
```

In this for loop, the master VIP agent generates 16 transactions and sends them over to the VIP interface. The WR_TRANSACTION_FAIL_1a is added here for debugging randomization failure purpose.

Transaction Examples

There are different methods of configuring the transaction after it is created. In the example design, three methods are shown for write and read transactions. Method 1 is to fully randomize the transaction after it is being created. Method 2 shows how to use the APIs to set the transaction.

Method 1 for Write Transaction

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

Method 2 for Write Transaction

```
wr_transaction = mst_agent.driver.create_transaction("write transaction");
wr_transaction.set_id(mtest_id);
wr_transaction.set_data(mtest_data);
wr_transaction.set_dest(mtest_dest);
wr_transaction.set_last(mtest_last);
wr_transaction.set_strb(mtest_strb);
….
mst_agent.wr_driver.send(wr_transaction);
```

Upgrading

This appendix is not applicable for the first release of the core.

AXI4-Stream VIP Agent and Flow Methodology

This appendix contains information about the AXI4-Stream VIP agents and flow methodologies. The AXI4-Stream VIP has three agents:

- AXI4-Stream Master Agent
- AXI4-Stream Slave Agent
- AXI4-Stream Pass-Through Agent

AXI4-Stream Master Agent

When instantiating an AXI master VIP, a master agent has to be declared and constructed. Class `axi4stream_mst_agent` contains other components that consist of the entire master Verification IP. These are the monitor, driver, and `vif_proxy`.

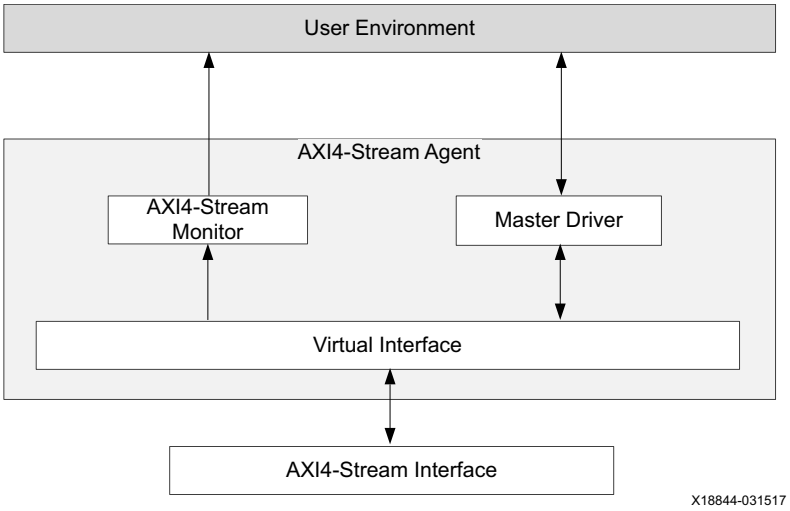

Figure B-1: **AXI4-Stream Master VIP Agent**

AXI4-Stream Master Driver

- Receives transactions from the user environment and drives stream interface.
- Returns a completed transaction when the transaction is accepted.

AXI4-Stream Monitor

- Monitors AXI4-Stream interface.
- Collects transaction into analysis port.

Write Transaction Flow

The AXI4-Stream master write transaction flows through the master agent in the following steps:

1. Driver asks for the next transaction through a `try_next_item`. This is a blocking get.
2. The user environment creates a single transaction. The transaction contains the following:
   - **Command Information** – TID, TDEST, TSTRB, TKEEP, TUSER, TLAST, and TDATA when available
   - **Master Controlled Timing** – Delay between two transfers
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.
5. When the interface receives the TREADY from the slave, the master driver returns a copy of the transaction (with the same `transaction_id`) to the user environment.
6. The user environment receives the completed transaction. Because the ID of the transaction is updated, the sequence knows that it has completed.

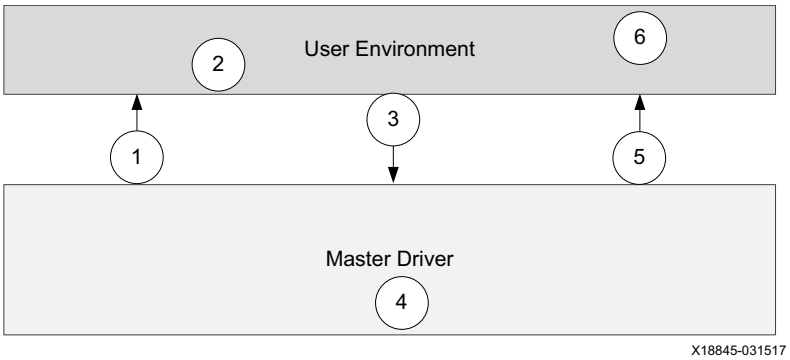

Figure B-2: **Write Transaction Flow**

AXI4-Stream Slave Agent

When instantiating an AXI4-Stream slave VIP, a slave agent has to be declared and constructed. Class `slv_agent` contains other components that consist of the entire slave Verification IP. These are monitor and driver.

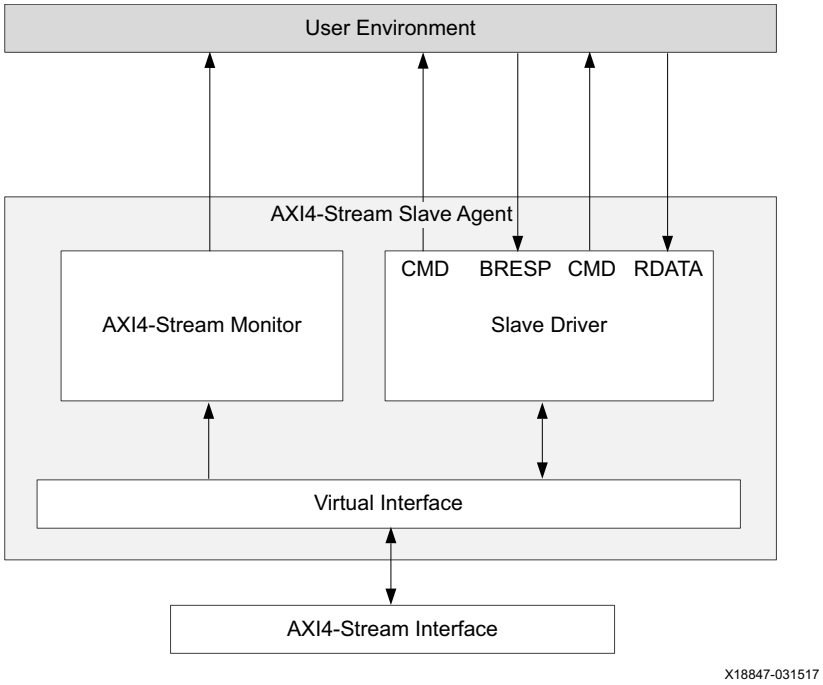

Figure B-3: **AXI4-Stream Slave VIP Agent**

AXI4-Stream Slave Driver

- Receives TREADY transactions from the user environment and drives the TREADY signal if HAS_TREADY is ON.
- If HAS_TREADY is OFF, TREADY is set to High all the time.

AXI4-Stream Monitor

- Monitors AXI4-Stream interface.
- Collects completed transaction into analysis port.

TREADY Timing Flow

- TREADY is generated independently of the stream transaction.
- Configuration of the TREADY is not from the same transaction.

- See the Configurable Ready Delays for different timing options.

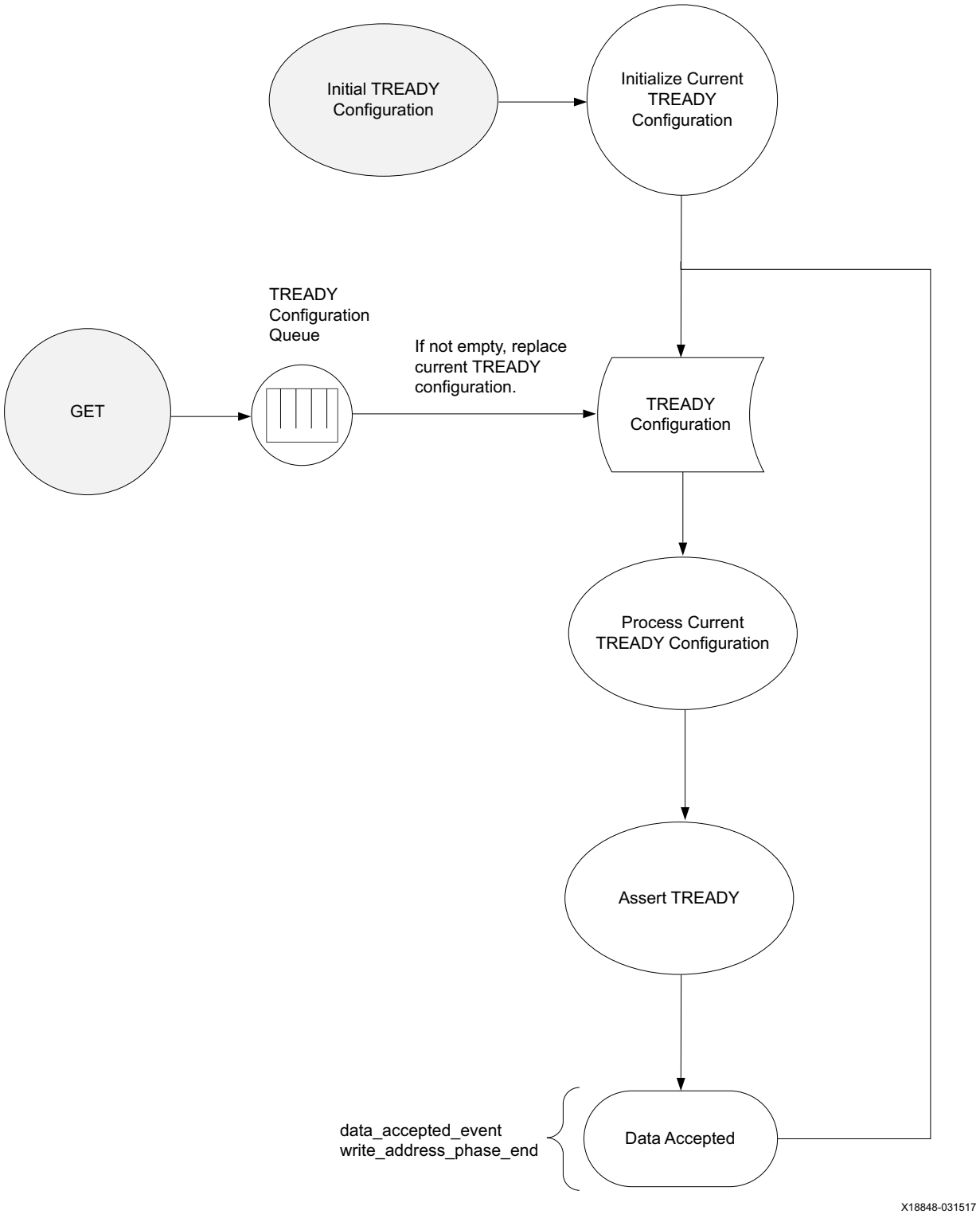

Figure B-4: **TREADY Timing Flow**

AXI4-Stream Pass-Through Agent

When instantiating an AXI4-Stream pass-through VIP, a pass-through agent has to be declared and constructed. Class `axi4stream_passthrough_agent` contains other components that consist of the entire pass-through Verification IP. The pass-through VIP has the options to be switched to runtime master or runtime slave modes. It includes monitor, master driver, and slave driver.

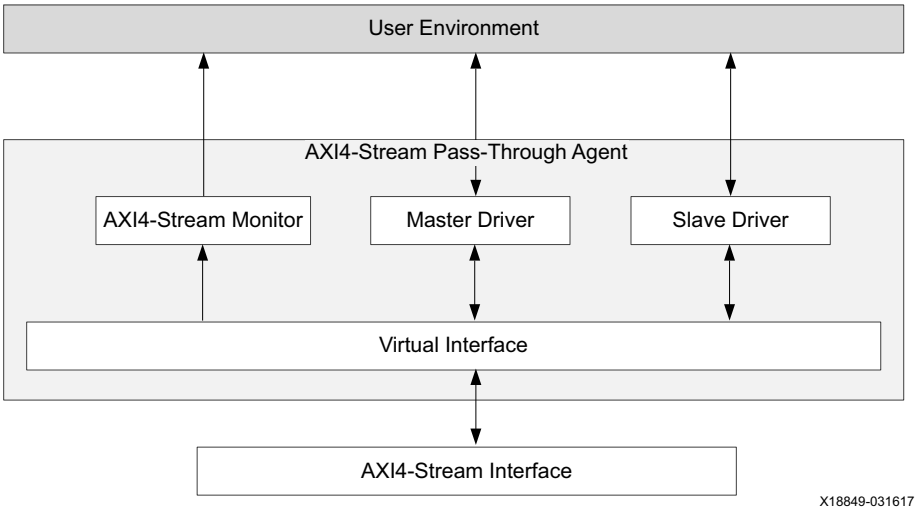

Figure B-5: **AXI4-Stream Pass-Through VIP Agent**

AXI4-Stream Master Driver

The same features as the AXI4-Stream master driver in `mst_agent`.

AXI4-Stream Slave Driver

The same features as the AXI4-Stream slave driver in `slv_agent`.

AXI4-Stream Monitor

The same features for both master/slave agent monitors.

READY Generation

READY signals are generated independently from other attributes. The `axi4stream_ready_gen` is the class used for READY generation.

Configurable Ready Delays

There is no one way that the READY signals 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 slave AGENT. 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 B-1 shows the configurable READY delay description.

Table B-1: **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.
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 (also called event in ready generation policy) occurs on this channel. The policy repeats until the channel is given a different policy.

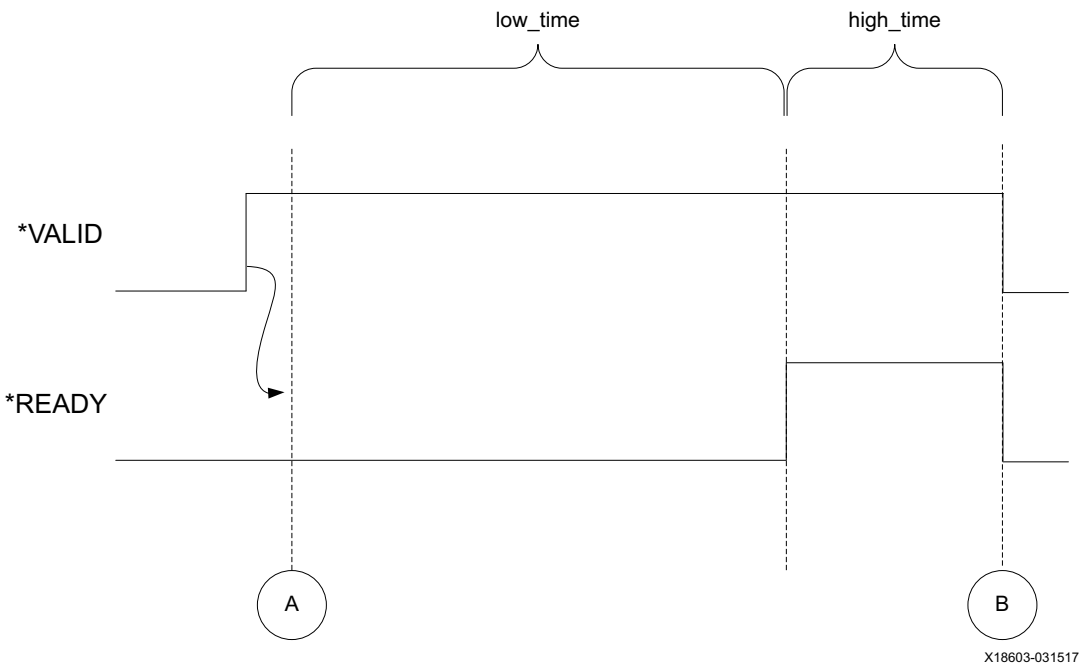

Figure B-6: **GEN_SINGLE/RAND_SINGLE**

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 B-7 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.

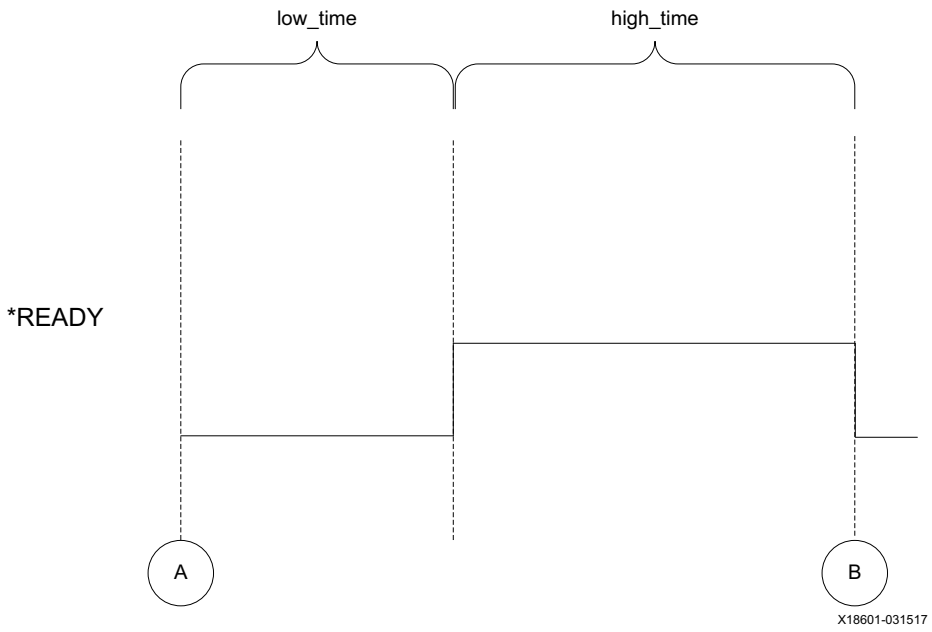

Figure B-7: **GEN_OSC/RAND_OSC**

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 B-8 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.

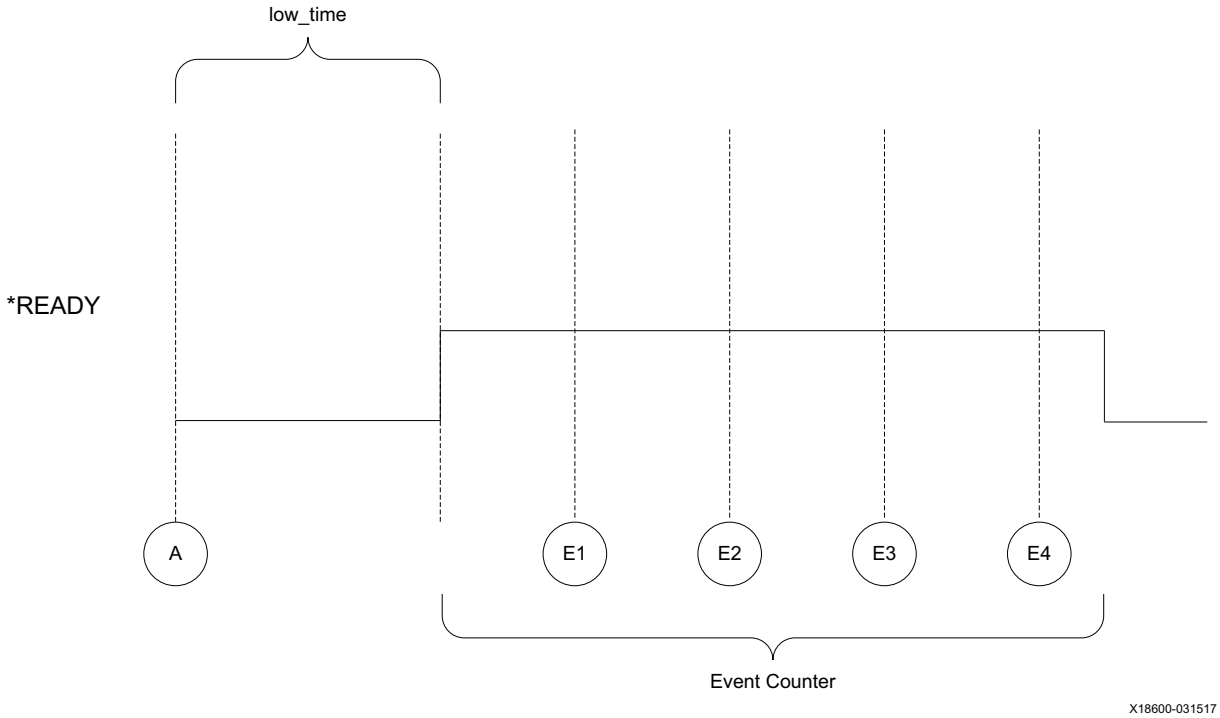

Figure B-8: **GEN_EVENTS/RAND_EVENTS**

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.

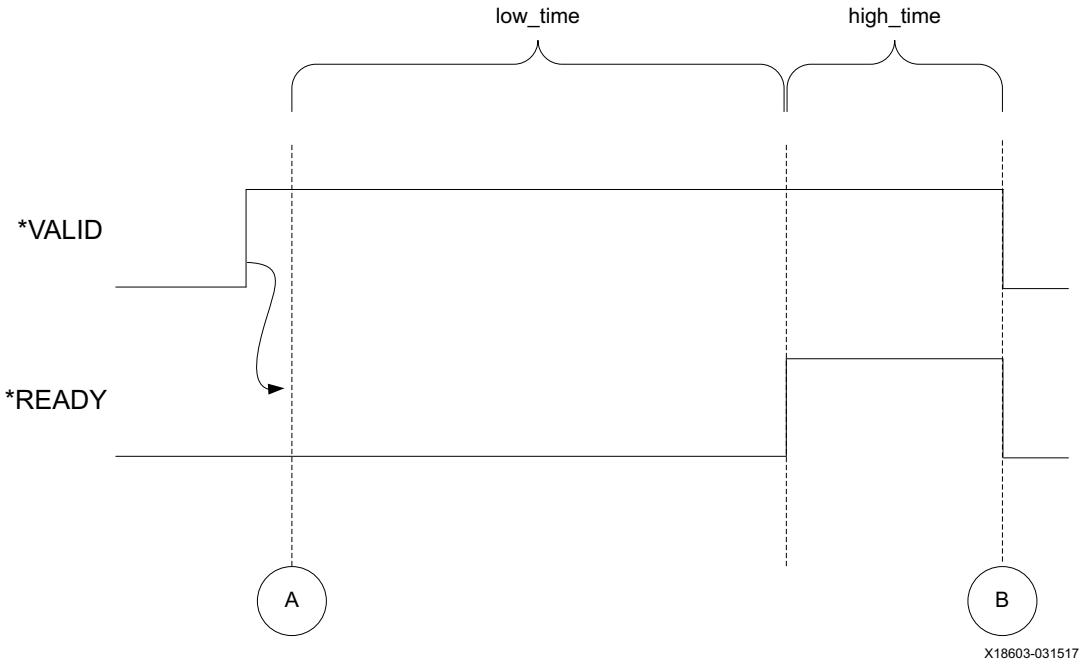

Figure B-9: **GEN_AFTER_VALID_SINGLE/RAND_AFTER_VALID_SINGLE**

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.

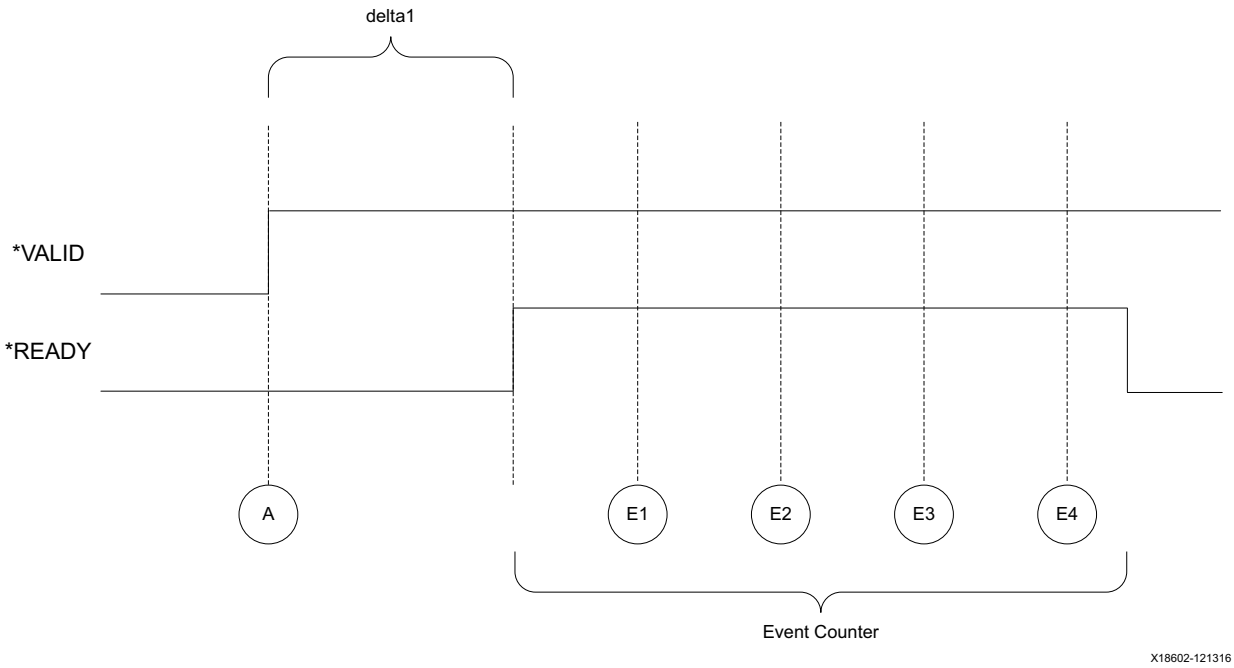

Figure B-10: **GEN_AFTER_VALID_EVENTS/RAND_AFTER_VALID_EVENTS**

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.

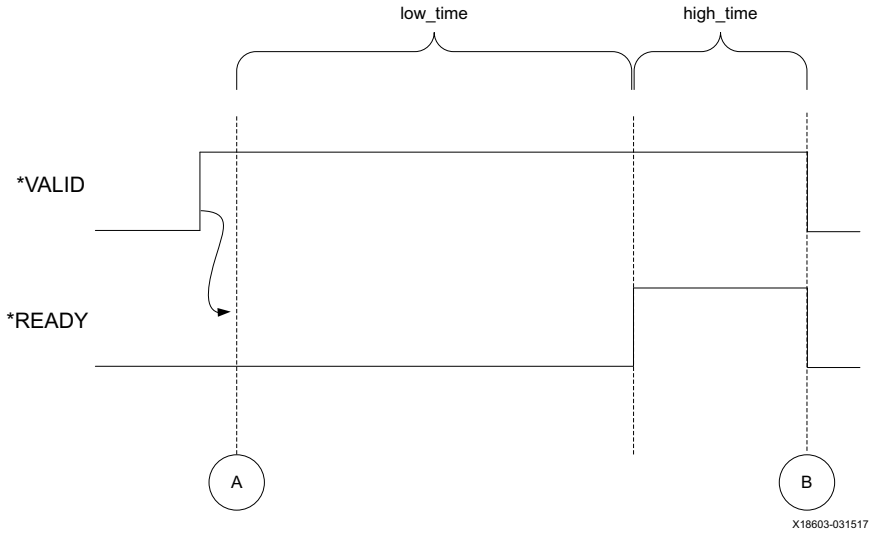

Figure B-11: **GEN_AFTER_VALID_OSC/RAND_AFTER_VALID_OSC**

GEN_AFTER_RANDOM

This policy is used to randomly generate 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 AXI4-Stream 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 AXI4-Stream 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 AXI4-Stream VIP

AR: 68726

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.

Appendix D

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. 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
2. *AXI Protocol Checker LogiCORE™ IP Product Guide* (PG101)
3. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
4. *Vivado Design Suite User Guide: Designing with IP* (UG896)
5. *Vivado Design Suite User Guide: Getting Started* (UG910)
6. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
7. *ISE to Vivado Design Suite Migration Guide* (UG911)
8. *Vivado Design Suite User Guide: Implementation* (UG904)
9. *LogiCORE IP AXI Interconnect Product Guide* (PG059)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/07/2017	1.0	• Added note #5-7 in IP Facts table. • Updated AXI Protocol Checks and Descriptions table. • Updated note in Example Design chapter.
04/05/2017	1.0	Initial Xilinx release.

Please Read: Important Legal Notices