Verification Continuum™

VC Verification IP
AMBA AXI
VMM User Guide

Version R-2021.03, March 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

Preface

About This Guide

This guide contains installation, setup, and usage material for SystemVerilog VMM users of the VC Verification IP for AMBA AXI, and is for design or verification engineers who want to verify AXI operation using a VMM testbench written in SystemVerilog. Readers are assumed to be familiar with AXI, Object Oriented Programming (OOP), SystemVerilog, and Verification Methodology Manual (VMM) techniques.

From the R-2021.03 release onwards, the documentation updates for the guides that are based on the SystemVerilog OVM designs will be suspended. For more information, see the UVM guides for the current updates on this VIP. Contact Synopsys support for any queries or clarifications.

Guide Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Introduction", introduces the Synopsys AXI VIP and its features
- Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using the Synopsys AXI VIP
- Chapter 3, "General Concepts", introduces the AXI VIP within a VMM environment and describes the data objects and components that comprise the VIP
- Chapter 5, "Verification Features", describes the verification features supported by AXI VIP such as, Verification Planner and Protocol Analyzer
- Chapter 6, "Verification Topologies", describes the topologies to verify master, slave and interconnect DUT
- Chapter 7, "Using AXI Verification IP", shows how to install and run a getting started example
- Chapter 8, "Troubleshooting" explains how to generate trace files and customize logging
- Appendix A, "Reporting Problems", outlines the process for working through and reporting AXI VIP transactor issues

Web Resources

- Documentation through SolvNet: https://solvnetplus.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): http://www.synopsys.com/keys

Customer Support

To obtain support for your product, choose one of the following:

1. Go to https://solvnetplus.synopsys.com and open a case.

   Enter the information according to your environment and your issue.
2. Send an e-mail message to support_center@synopsys.com.

   Include the Product name, Sub Product name, and Tool Version in your e-mail so it can be routed correctly.
3. Telephone your local support center.
   - North America:

     Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
   - All other countries:

     https://www.synopsys.com/support/global-support-centers.html

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1 Introduction

This chapter gives a basic introduction, overview and features of the AXI VMM Verification IP.

This chapter discusses the following topics:

1.1 Introduction

The AXI VIP supports verification of designs that include interfaces implementing the AXI Specification. This document describes the use of this VIP in testbenches that comply with the SystemVerilog Verification Methodology Manual (VMM). This approach leverages advanced verification technologies and tools that provide,

- Protocol functionality and abstraction
- Constrained random verification
- Functional coverage
- Rapid creation of complex tests
- Modular testbench architecture that provides maximum reuse, scalability and modularity
- Proven verification approach and methodology
- Transaction-level models
- Self-checking tests
- Object oriented interface that allows OOP techniques

1.2 Prerequisites

- Familiarize with AXI, object oriented programming, SystemVerilog, and the current version of VMM 1.0 (onwards).

1.3 References

For more information on AXI Verification IP, refer to the following documents:

- Class Reference for VC Verification IP for AMBA® AXI is available at:
  $DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_svt_vmm_class_reference/html/index.html

1.4 Product Overview

The AXI VIP is a suite of VMM-based verification components that are compatible for use with SystemVerilog-Compliant testbenches. The AXI VIP suite simulates AXI transactions through active components, as defined by the AXI specification.

VIP provides AXI System component that contain master components, slave components and multi-stream scenario generator. Master and slave components support all functionality normally associated with active and passive VMM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging and functional coverage. After instantiating the system component, you can select and combine active and passive components to create an environment that verifies AXI features in the DUT.

The Master and Slave components can also be used in standalone mode, that is, they can be instantiated in the testbench directly without the system component.

1.5 Language and Methodology Support

The current version of AXI VIP suite is qualified with the following languages and methodology:

- Languages
  - SystemVerilog
- Methodology
  - VMM 1.2

1.6 Features Supported

1.6.1 Protocol Features

AXI VIP currently supports the following protocol functions:

- AXI3 Channel handshake (valid, ready signaling)
- AXI3 Addressing options (all burst lengths, burst types, burst sizes)
- AXI3 Response Signaling (support for OKAY, DECERR and SLVERR)
- AXI3 Ordering Model (transaction IDs, read/write ordering, write data interleaving)
- AXI3 Data Buses (write strobes, narrow transfers)
- AXI3 Exclusive Access
- AXI3 Locked Access
- AXI3 Unaligned Transfers
- AXI3 Reset functionality
- AXI4 Read/Write
- AXI4 Interface categories (read only/write only)

- AXI4 Quality of Service
- AXI4 Region
- AXI4 AWCACHE and ARCACHE Attributes
- AXI4-Lite
- AXI4 Longer bursts
- AXI4 User signals
- ACE coherent and snoop transactions
- ACE Domain
- ACE-Lite
- ACE Barrier and DVM
- ACE Speculative Read
- ACE Snoop filtering
- ACE Exclusive Access

1.6.2 Verification Features

AXI VIP currently supports the following verification functions:

- Default functional coverage (transaction, state and toggle coverage)
- Protocol checking for ease of debugging protocol transactions
- Debug port
- Programmable value (X, Z, hold previous) on all channel signals, when valid signal is low
- Control on delays and timeouts
- Built-in slave memory
- Support for Verification Planner
- Support for Protocol Analyzer
- VC Auto Testbench
- AutoPerformance

1.6.3 Methodology Features

AXI VIP currently supports the following methodology functions:

- VIP organized as a system component, which includes set of master and slave components. Master and slave components can also be used in standalone mode.
- Analysis ports for connecting master/slave component to scoreboard, or any other component
- Callbacks for master/slave component
- Notifications to denote start and end of transactions

1.7 Features Not Supported

Refer to section “Known Issues and Limitations” present in Chapter “AXI Verification IP Notes” in the AMBA SVT VIP Release Notes.

AMBA SVT VIP Release Notes are present at:

```
$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/amba_svt_rel_notes.pdf
```

2
Installation and Setup

This chapter leads you through installing and setting up the Synopsys AMBA AXI VIP. When you complete this checklist, the provided example testbench will be operational and the Synopsys AXI VIP will be ready to use.

The checklist consists of the following major steps:

1. "Verifying the Hardware Requirements"
2. "Verifying Software Requirements"
3. "Preparing for Installation"
4. "Downloading and Installing"
5. "What's Next?"

If you encounter any problems with installing the Synopsys AXI VIP, see Customer Support.

2.1 Verifying the Hardware Requirements

The AXI Verification IP requires a Solaris or Linux workstation configured as follows:

- 1440 MB available disk space for installation
- 16 GB Virtual Memory recommended
- FTP anonymous access to ftp.synopsys.com (optional)

2.2 Verifying Software Requirements

The Synopsys AXI VIP is qualified for use with certain versions of platforms and simulators. This section lists software that the AXI VIP requires.

2.2.1 Platform/OS and Simulator Software

- **Platform/OS and VCS**: You need versions of your platform/OS and simulator that have been qualified for use. To see which platform/OS and simulator versions are qualified for use with the AXI VMM VIP, check the support matrix for "SVT-based" VIP in the following document:

  Support Matrix for SVT-Based AXI VIP is in:

  VC Verification IP for AMBA Release Notes

2.2.2 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the Synopsys AXI VIP. Acquiring the SCL software is covered here in the installation instructions in Licensing Information.

2.2.3 Other Third Party Software

- **Adobe Acrobat**: Synopsys AXI VIP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.
- **HTML browser:** Synopsys AXI VIP includes class reference documentation in HTML. The following browser/platform combinations are supported:
  - Microsoft Internet Explorer 6.0 or later (Windows)
  - Firefox 1.0 or later (Windows and Linux)
  - Netscape 7.x (Windows and Linux)

2.3 Preparing for Installation

1. Set the `DESIGNWARE_HOME` to the following absolute path where the AXI VIP is installed:

   ```
   setenv DESIGNWARE_HOME absolute_path_to_designware_home
   ```

2. Ensure that your environment and PATH variables are set correctly, including:
   - `DESIGNWARE_HOME/bin` – The absolute path as described in the previous step.
   - `LM_LICENSE_FILE` – The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.

     ```
     % setenv LM_LICENSE_FILE <my_license_file|port@host>
     ```

   - `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for Synopsys software or the port@host reference to this file.

     ```
     % setenv SNPSLMD_LICENSE_FILE <my_Synopsys_license_file|port@host>
     ```

   - `DW_LICENSE_FILE` – The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.

     ```
     % setenv DW_LICENSE_FILE <my_VIP_license_file|port@host>
     ```

2.4 Downloading and Installing

Attention The Electronic Software Transfer (EST) system only displays products your site is entitled to download. If the product you are looking for is not available, contact est-ext@synopsys.com.

Follow the instructions below for downloading the software from Synopsys. You can download from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If your Synopsys SolvNet password is unknown or forgotten, go to http://solvnet.synopsys.com.

Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, refer to the following web page:

https://www.synopsys.com/apps/protected/support/EST-FTP_Accelerator_Help_Page.html

2.4.1 Downloading From the Electronic Software Transfer (EST) System (Download Center)

a. Point your web browser to http://solvnet.synopsys.com.

b. Enter your Synopsys SolvNet Username and Password.

c. Click Sign In button.

d. Make the following selections on SolvNet to download the `.run` file of the VIP (See Figure 2-1).

   i. Downloads tab

   ii. VC VIP Library product releases

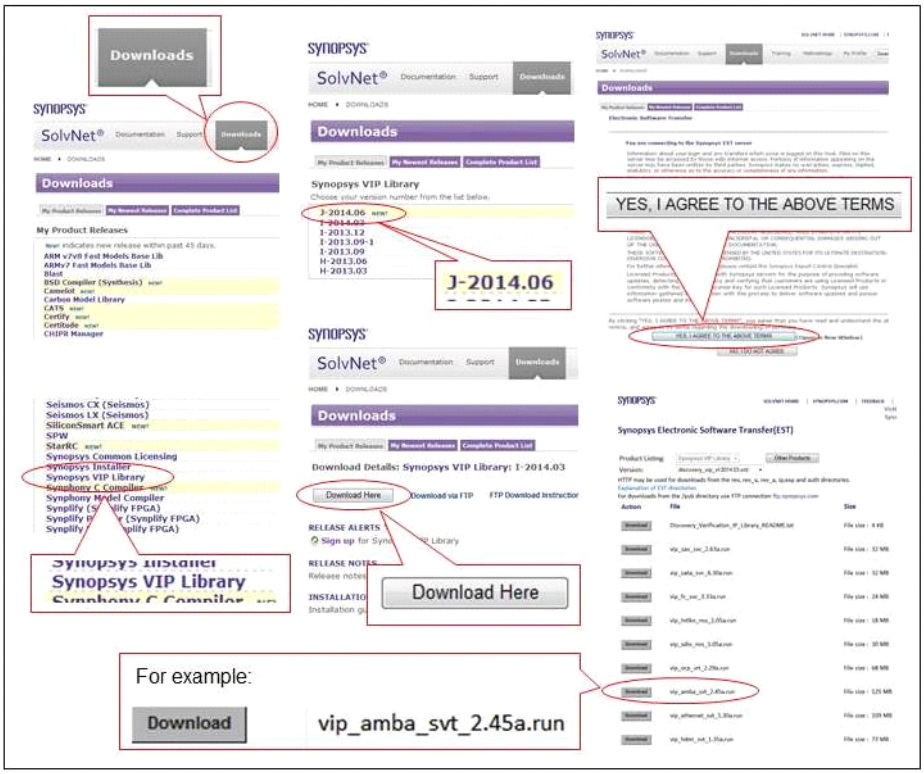
   iii. <release_version>

   iv. Download Here button

   v. Yes, I Agree to the Above Terms button

   vi. Download `.run` file for the VIP

Figure 2-1 SolvNet Selections for VIP Download



e. Set the `DESIGNWARE_HOME` environment variable to a path where you want to install the VIP.

```
% setenv DESIGNWARE_HOME VIP_installation_path
```

f. Execute the `.run` file by invoking its filename. The VIP is unpacked and all files and directories are installed under the path specified by the `DESIGNWARE_HOME` environment variable. The `.run` file can be executed from any directory. The important step is to set the `DESIGNWARE_HOME` environment variable before executing the `.run` file.

The Synopsys AMBA VIP suite includes VIP models for all AMBA interfaces (AHB, APB, AXI, and ATB). You must download the VC VIP for AMBA suite to access the VIP models for AHB, APB, AXI, and ATB.

2.4.2 Downloading Using FTP with a Web Browser

a. Follow the above instructions through the product version selection step.
b. Click the "Download via FTP" link instead of the "Download Here" button.
c. Click the "Click Here To Download" button.
d. Select the file(s) that you want to download.
e. Follow browser prompts to select a destination location.

If you are unable to download the Verification IP using above instructions, refer to "Customer Support" section to obtain support for download and installation.

2.5 What's Next?

The remainder of this chapter describes the details of the different steps you performed during installation and setup, and consists of the following sections:

- Licensing Information
- Environment Variable and Path Settings
- Determining Your Model Version
- Integrating the VIP into Your Testbench
- Include and Import Model Files into Your Testbench
- Compile and Run Time Options

2.5.1 Licensing Information

The AMBA VIP uses the Synopsys Common Licensing (SCL) software to control its usage.

You can find general SCL information in the following location:

http://www.synopsys.com/keys

For more information on the order in which licenses are checked out for each VIP, refer to VC VIP AMBA Release Notes.

The licensing key must reside in files that are indicated by specific environment variables. For more information about setting these licensing environment variables, see Environment Variable and Path Settings.

2.5.1.0.1 License Polling

If you request a license and none are available, license polling allows your request to exist until a license becomes available instead of exiting immediately.

To control license polling, you use the DW_WAIT_LICENSE environment variable as follows:

- To enable license polling, set the DW_WAIT_LICENSE environment variable to 1.
- To disable license polling, unset the DW_WAIT_LICENSE environment variable. By default, license polling is disabled.

2.5.1.0.2 Simulation License Suspension

All VIP products support license suspension. Simulators that support license suspension allow a model to check in its license token while the simulator is suspended, then check the license token back out when the simulation is resumed.

This capability is simulator-specific; not all simulators support license check-in during suspension.

2.5.2 Environment Variable and Path Settings

The following are environment variables and path settings required by the AXI VMM VIP verification models:

- `DESIGNWARE_HOME`: The absolute path to where the VIP is installed.
- `DW_LICENSE_FILE` - The absolute path to file that contains the license keys for the VIP product software or the port@host reference to this file.
- `SNPSLMD_LICENSE_FILE`: The absolute path to file(s) that contains the license keys for Synopsys software (VIP and/or other Synopsys Software tools) or the port@host reference to this file.

For faster license checkout of Synopsys VIP software please ensure to place the desired license files at the front of the list of arguments to `SNPSLMD_LICENSE_FILE`.

- `LM_LICENSE_FILE`: The absolute path to a file that contains the license keys for both Synopsys software and/or your third-party tools.

The Synopsys VIP License can be set in either of the 3 license variables mentioned above with the order of precedence for checking the variables being:

- `DW_LICENSE_FILE` -> `SNPSLMD_LICENSE_FILE` -> `LM_LICENSE_FILE`, but also note If DW_LICENSE_FILE environment variable is enabled, VIP will ignore SNPSLMD_LICENSE_FILE and `LM_LICENSE_FILE` settings.

Hence to get the most efficient Synopsys VIP license checkout performance, set the `DW_LICENSE_FILE` with only the License servers which contain Synopsys VIP licenses. Also, include the absolute path to the third party executable in your PATH variable.

2.5.2.1 Simulator-Specific Settings

Your simulation environment and PATH variables must be set as required to support your simulator.

2.5.3 Determining Your Model Version

The following steps tell you how to check the version of the models you are using.

Note Verification IP products are released and versioned by the suite and not by individual model. The version number of a model indicates the suite version.

- To determine the versions of VIP models installed in your $DESIGNWARE_HOME tree, use the setup utility as follows:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -i home
```

- To determine the versions of VIP models in your design directory, use the setup utility as follows:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir_path -i design
```

2.5.4 Integrating the VIP into Your Testbench

After installing the VIP, follow these procedures to set up the VIP for use in testbenches:

- "Creating a Testbench Design Directory"
- "Setting Up a New VIP"
- "Installing and Setting Up More than One VIP Protocol Suite"
- "Updating an Existing Model"
- "Removing VIP Models from a Design Directory"
- "Reporting Information About DESIGNWARE_HOME or a Design Directory"
- "The dw_vip_setup Utility"

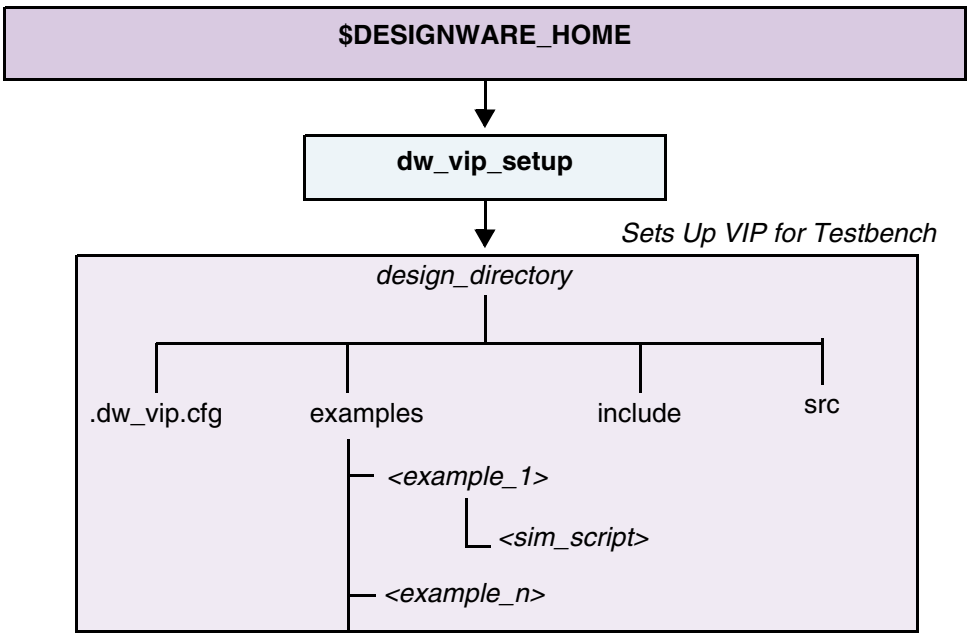
2.5.4.1 Creating a Testbench Design Directory

A *design directory* contains a version of the Synopsys VIP that is set up and ready for use in a testbench. You use the dw_vip_setup utility to create design directories. For the full description of dw_vip_setup, refer to The dw_vip_setup Utility.

Note If you move a design directory, the references in your testbenches to the include files will need to be revised to point to the new location. Also, any simulation scripts in the examples directory will need to be recreated.

A design directory gives you control over the version of VIP in your testbench because it is isolated from the DESIGNWARE_HOME installation. When you want, you can use dw_vip_setup to update VIP in your design directory. Figure 2-2 shows this process and the contents of a design directory.

Figure 2-2 Design Directory Created by dw_vip_setup



A design directory contains:

examples Each VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.

include Language-specific include files that contain critical information for Synopsys VIP models. This directory is specified in simulator command lines.

src Synopsys VIP-specific include files (not used by all Synopsys VIP). This directory may be specified in simulator command lines.

.dw_vip.cfg A database of all VIP models being used in the testbench. The dw_vip_setup program reads this file to rebuild or recreate a design setup

Note Do not modify this file because dw_vip_setup depends on the original contents.

Note When using a design_dir, you have to make sure that the DESIGNWARE_HOME that was used to setup the design_dir is the same one used in the shell when running the simulation.
In other words when using a design_dir, you have to make sure that the SVT version identified in the design_dir is available in the DESIGNWARE_HOME used in the shell when running the simulation.

2.5.4.2 Setting Up a New VIP

After you have installed the VIP, you must set up the VIP for project and testbench use. All VC VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on the protocol.

The setup process gathers together all the required component files you need to incorporate into your testbench required for simulation runs.

You have the choice to set up all of them, or only specific ones. For example, the AXI VIP contains the following components.

- axi_master_group_svt
- axi_slave_group_svt
- axi_system_group_svt

You can set up either an individual component, or the entire set of components within one protocol suite. Use the Synopsys provided tool called dw_vip_setup for these tasks. It resides in $DESIGNWARE_HOME/bin.

To get help on dw_vip_setup, invoke the following:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup --help
```

The following command adds a model <model_svt> to the directory `design_dir`.

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add axi_system_env_svt
-svlog
```

This command sets up all the required files in `/tmp/design_dir`.

The utility dw_vip_setup creates three directories under design_dir which contain all the necessary model files. Files for every VIP are included in these three directories.

- **examples**: Each VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.
- **include**: Language-specific include files that contain critical information for Synopsys models. This directory "include/sverilog" is specified in simulator commands to locate model files.
- **src**: Synopsys-specific include files This directory "src/sverilog/vcs" must be included in the simulator command to locate model files.

Note that some components are “top level” and will setup the entire suite. You have the choice to set up the entire suite, or just one component such as a monitor.

Attention There ***must*** be only one design_dir installation per simulation, regardless of the number of Verification and Implementation IPs you have in your project. Do create this directory in $DESIGNWARE_HOME.

2.5.4.3 Installing and Setting Up More than One VIP Protocol Suite

All VIPs for a particular project must be set up in a single common directory once you execute the *.run file. You may have different projects. In this case, the projects can use their own VIP setup directory. However, all the VIPs used by that specific project must reside in a common directory.

The examples in this chapter call that directory as `design_dir`, but you can use any name.

In this example, assume you have the AXI suite set up in the `design_dir` directory. In addition to the AXI VIP, you require the Ethernet and USB VIP suites.

First, follow the previous instructions on downloading and installing the Ethernet VIP and USB suites.

Once installed, the Ethernet and USB suites must be set up in and located in the same `design_dir` location as AMBA. Use the following commands:

```
// First install AXI
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add axi_system_env_svt -svlog

//Add Ethernet to the same design_dir as AXI
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add ethernet_system_env_svt -svlog

// Add USB to the same design_dir as AMBA and Ethernet
%unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
-add usb_system_env_svt -svlog
```

To specify other model names, consult the VIP documentation.

By default, all of the VIPs use the latest installed version of SVT. Synopsys maintains backward compatibility with previous versions of SVT. As a result, you may mix and match models using previous versions of SVT.

2.5.4.4 Updating an Existing Model

To add and update an existing model, do the following:

1. Install the model to the same location at which your other VIPs are present by setting the $DESIGNWARE_HOME environment variable.
2. Issue the following command using design_dir as the location for your project directory.

   ```
   %unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path  /tmp/design_dir
   -add axi_master_agentgroup_svt -svlog
   ```

3. You can also update your design_dir by specifying the version number of the model.

   ```
   %unix> dw_vip_setup -path design_dir -add axi_master_agentgroup_svt -v 3.50a -svlog
   ```

2.5.4.5 Removing VIP Models from a Design Directory

This example shows how to remove all listed models in the design directory at "/d/test2/daily" using the model list in the file "del_list" in the scratch directory under your home directory. The dw_vip_setup program command line is:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p /d/test2/daily -r -m ~/scratch/del_list
```

The models in the *del_list* file are removed, but object files and include files are not.

2.5.4.6 Reporting Information About DESIGNWARE_HOME or a Design Directory

In these examples, the setup program sends output to STDOUT.

The following example lists the Synopsys VIP libraries, models, example testbenches, and license version in a DESIGNWARE_HOME installation:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -i home
```

The following example lists the Synopsys VIP libraries, models, and license version in a testbench design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir -i design
```

2.5.4.7 Running the Example with +incdir+

In the current setup, you install the VIP under DESIGNWARE_HOME followed by creation of a design directory which contains the versioned VIP files. With every newer version of the already installed VIP requires the design directory to be updated. This results in:

- Consumption of additional disk space
- Increased complexity to apply patches

The new alternative approach of directly pulling in all the files from DESIGNWARE_HOME eliminates the need for design directory creation. VIP version control is now in the command line invocation.

The following code snippet shows how to run the basic example from a script:

```
cd <testbench_dir>/examples/sverilog/amba_svt/tb_amba_svt_uvm_basic_sys/
// To run the example using the generated run script with +incdir+
./run_amba_svt_uvm_basic_sys -verbose -incdir shared_memory_test vcsvlog
```

For example, the following compile log snippet shows the paths and defines set by the new flow to use VIP files right out of DESIGNWARE_HOME instead of design_dir.

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/amba_svt/<vip_version>/sverilog/include \
-ntb_opts uvm -full64 -sverilog +define+UVM_DISABLE_AUTO_ITEM_RECORDING \
+define+UVM_PACKER_MAX_BYTES=1500000 \
-timescale=1ns/1ps \
+define+SVT_UVM_TECHNOLOGY \
+incdir+<testbench_dir>/examples/sverilog/amba_svt/tb_amba_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm_basic_sys/
env \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm_basic_sys/
dut \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm_basic_sys/
hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm_basic_sys/
tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

For VIPs with dependency, include the +incdir+ for each dependent VIP.

2.5.4.8 Getting Help on Example Run/make Scripts

You can get help on the generated make/run scripts in the following ways:

1. Invoke the run script with no switches, as in:

   run_axi_svt_vmm_basic_sys --help

   usage: run_axi_svt_vmm_basic_sys [-32] [-incdir] [-verbose] [-debug_opts] [-waves] [-clean] [-nobuild] [-buildonly] [-norun] [-pa] <scenario> <simulator>

where <scenario> is one of: all axi_slave_mem_diff_data_width_response_test axi_unaligned_backdoor_write_read_test base_test config_creator_test directed_4kboundary_test directed_test directed_write_read_data_cehck_wysiwyg_enable_test random_wr_rd_test reorder_wr_rd_test

<simulator> is one of: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog

-32 forces 32-bit mode on 64-bit machines

-incdir use DESIGNWARE_HOME include files instead of design directory

-verbose enable verbose mode during compilation

-debug_opts enable debug mode for VIP technologies that support this option

-waves [fsdb | verdi | dve | dump] enables waves dump and optionally opens viewer (VCS only)

-seed run simulation with specified seed value

-clean clean simulator generated files

-nobuild skip simulator compilation

-buildonly exit after simulator build

-norun only echo commands (do not execute)

-pa invoke Verdi after execution

2. Invoke the make file with help switch as in:

gmake help

Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG_OPTS=1] [SEED=<value>] [FORCE_32BIT=1] [WAVES=fsdb | verdi | dve | dump] [NOBUILD=1] [BUILDONLY=1] [PA=1] [<scenario> ...]

Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog

Valid scenarios are: all axi_slave_mem_diff_data_width_response_test axi_unaligned_backdoor_write_read_test base_test config_creator_test directed_4kboundary_test directed_test directed_write_read_data_cehck_wysiwyg_enable_test random_wr_rd_test reorder_wr_rd_test

You must have PA installed if you use the -pa or PA=1 switches.

2.5.4.9 The dw_vip_setup Utility

The dw_vip_setup utility:

- Adds, removes, or updates the VIP models in a design directory
- Adds example testbenches to a design directory, the VIP models they use (if necessary), and creates a script for simulating the testbench using any of the supported simulators
- Restores (cleans) example testbench files to their original state
- Reports information about your installation or design directory, including version information

- Supports Protocol Analyzer (PA)
- Supports the FSDB wave format

2.5.4.9.1 Setting Environment Variables

Before running dw_vip_setup, the following environment variables must be set:

- DESIGNWARE_HOME – Points to where the Synopsys VIP is installed

2.5.4.9.2 The dw_vip_setup Command

You invoke dw_vip_setup from the command prompt. The dw_vip_setup program checks command line argument syntax and makes sure that the requested input files exist. The general form of the command is:

```
% dw_vip_setup [-p[ath] directory] switch (model [-v[ersion] latest | version_no] ) …
```

or

```
% dw_vip_setup [-p[ath] directory] switch -m[odel_list] filename
```

where

[**-p**[ath] *directory*] — The optional -path argument specifies the path to your design directory. When omitted, dw_vip_setup uses the current working directory.

switch — The *switch* argument defines dw_vip_setup operation. Table 2-1 lists the switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

Switch	Description
-a[dd] (*model* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or current working directory. If you do not specify a version, the latest version is assumed. The model names are: • axi_master_group_svt • axi_slave_group_svt • axi_system_group_svt The -add switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r[emove] *model*	Removes ***all versions*** of the specified model or models from the design. The dw_vip_setup program does not attempt to remove any include files used solely by the specified model or models. The model names are: • axi_master_group_svt • axi_slave_group_svt • axi_system_group_svt

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-u[pdate] (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The dw_vip_setup script updates to the latest models when you do not specify a version. The model names are: • axi_master_group_svt • axi_slave_group_svt • axi_system_group_svt The -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {*scenario* | *model*/*scenario*} [**-v**[ersion] *version*]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulator run program for all supported simulators. If you specify a *scenario* (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command. **Note:** Use the -info switch to list all available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog vmm. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {*scenario* | *model*/*scenario*}	Cleans the specified scenario/testbench in either the design directory (as specified by the *-path* switch) or the current working directory. This switch deletes *all files in the specified directory*, then restores all Synopsys created files to their original contents.
-i/nfo design | home[:<product>[:<version>[:<methodology>]]]	Generate an informational report on a design directory or VIP installation. `design`: If the '`-info design`' switch is specified, the tool displays product and version content within the specified design directory to standard output. This output can be captured and used as a modellist file for input to this tool to create another design directory with the same content. `home`: If the '`-info home`' switch is specified, the tool displays product, version, and example content within the VIP installation to standard output. Optional filter fields can also be specified such as <product>, <version>, and <methodology> delimited by colons (:). An error will be reported if a nonexistent or invalid filter field is specified. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	Synopsys AXI VIP models are: • axi_master_group_svt • axi_slave_group_svt • axi_system_group_svt The *model* argument defines the model or models that dw_vip_setup acts upon. This argument is not needed with the -info or -help switches. All switches that require the *model* argument may also use a model list. You may specify a version for each listed *model*, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores *model* version information.
-b/ridge	Updates the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.
-pa	Enables the run scripts and Makefiles generated by dw_vip_setup to support PA. If this switch is enabled, and the testbench example produces XML files, PA will be launched and the XML files will be read at the end of the example execution. For run scripts, specify `-pa`. For Makefiles, specify `-pa = 1`.
-waves	Enables the run scripts and Makefiles generated by dw_vip_setup to support the `fsdb` waves option . To support this capability, the testbench example must generate an FSDB file when compiled with the WAVES Verilog macro set to `fsdb`, that is, `+define+WAVES=\"fsdb\"`. If a .fsdb file is generated by the example, the Verdi nWave viewer will be launched. For run scripts, specify `-waves fsdb`. For Makefiles, specify `WAVES=fsdb`.
-doc	Creates a doc directory in the specified design directory which is populated with symbolic links to the `DESIGNWARE_HOME` installation for documents related to the given model or example being added or updated.
-methodology <name>	When specified with -doc, only documents associated with the specified methodology name are added to the design directory. Valid methodology names include: OVM, RVM, UVM, VMM, and VLOG.
-copy	When specified with -doc, documents are copied into the design directory, not linked.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-s/uite_list <filename>	Specifies a file name which contains a list of suite names to be added, updated or removed in the design directory. This switch is valid only when following an operation switch, such as, -add, -update, or -remove. Only one suite name per line and each suite may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. The lines in the file starting with the pound symbol (#) will be ignored.
-m/odel_list <filename>	Specifies a file name which contains a list of model names to be added, updated or removed in the design directory. This switch is valid only when following an operation switch, such as, -add, -update, or -remove. Only one model name per line and each model may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. The lines in the file starting with the pound symbol (#) will be ignored.
-simulator <vendor>	When used with the `-example` switch, only simulator flows associated with the specified vendor are supported with the generated run script and Makefile. **Note**: Currently the vendors VCS, MTI, and NCV are supported.

The dw_vip_setup program treats all lines beginning with “#” as comments.

For more information on installing and running SystemVerilog VMM example testbenches, refer to “SystemVerilog VMM Example Testbenches”and “Installing and Running the Examples” sections.

2.5.5 Include and Import Model Files into Your Testbench

After you set up the models, you must include and import various files into your top testbench files to use the VIP.

The following is a code list of the includes and imports for components within `svt_axi_system_env`

```
/* include AXI VIP interface */
`include "svt_axi_if.svi"

/** Include the AXI SVT VMM package */
`include "svt_axi.vmm.pkg"


/** Import the SVT VMM Package */
import svt_vmm_pkg::*;

/** Import the AXI VIP Package*/
 import svt_axi_vmm_pkg::*;
```

You must also include various VIP directories on the simulator command line. Add the following switches and directories to all compile scripts:

- +incdir+<design_dir>/include/verilog
- +incdir+<design_dir>/include/sverilog
- +incdir+<design_dir>/src/verilog/<vendor>
- +incdir+<design_dir>/src/sverilog/<vendor>

Supported vendors are VCS, MTI and NCV. For example:

```
+incdir+<design_dir>/src/sverilog/vcs
```

Using the previous examples, the directory `<design_dir>` would be /tmp/design_dir.

2.5.6 Compile and Run Time Options

Every Synopsys provided example has ASCII files containing compile and run time options. The examples for the model are located in:

$DESIGNWARE_HOME/vip/svt/<*model*>/latest/examples/sverilog/<*example_name*>

The files containing the options are:

- sim_build_options (contain compile time options)
- sim_run_options (contain run time options)

These files contain both optional and required switches. For AXI VIP, the following are the contents of each file, listing optional and required switches:

sim_build_options

Required: -ntb_opts rvm

Required:`+define+SVT_VMM_TECHNOLOGY`

Optional: -timescale=1ns/1ps

Required: `+define+SVT_AXI_INCLUDE_USER_DEFINES`

sim_run_options

Required: `+vmm_test=$scenario`

The “scenario” is the VMM test name you pass to VCS.

3 General Concepts

This chapter describes the usage of AXI VIP in a VMM environment, and it's user interface. This chapter discusses the following topics:

- Introduction to VMM
- AXI VIP in a VMM Environment
- AXI VMM User Interface
- Functional Coverage
- Protocol Checks
- Reset Functionality
- Support for ACE Protocol in AXI Master
- Support for ACE-Lite Protocol in AXI Slave
- AXI4 Region and Address Range Support in Slave

3.1 Introduction to VMM

VMM is an object-oriented approach. It provides a blueprint for building testbenches using a constrained random verification. The resulting structure also supports directed testing. This chapter describes the AXI VIP components and user interface.

Refer to the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter.

This chapter assumes that you are familiar with SystemVerilog and VMM. For more information,

- For the IEEE SystemVerilog standard, see:

  IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

3.2 AXI VIP in a VMM Environment

AXI VIP group components are derived from vmm_group. The following sub-sections describe the group components in more detail.

3.2.1 Master Group

The master group encapsulates atomic generator, scenario generator, master transactor, and port monitor. The master group can be configured to operate in active mode and passive mode. In active mode, you might use the internal generators to generate the stimulus. You can select between atomic and scenario generator, using port configuration parameter svt_axi_port_configuration::generator_type. In addition, you can directly drive the input channel of master transactor. The input channel of master transactor is specified by svt_axi_master::xact_chan.

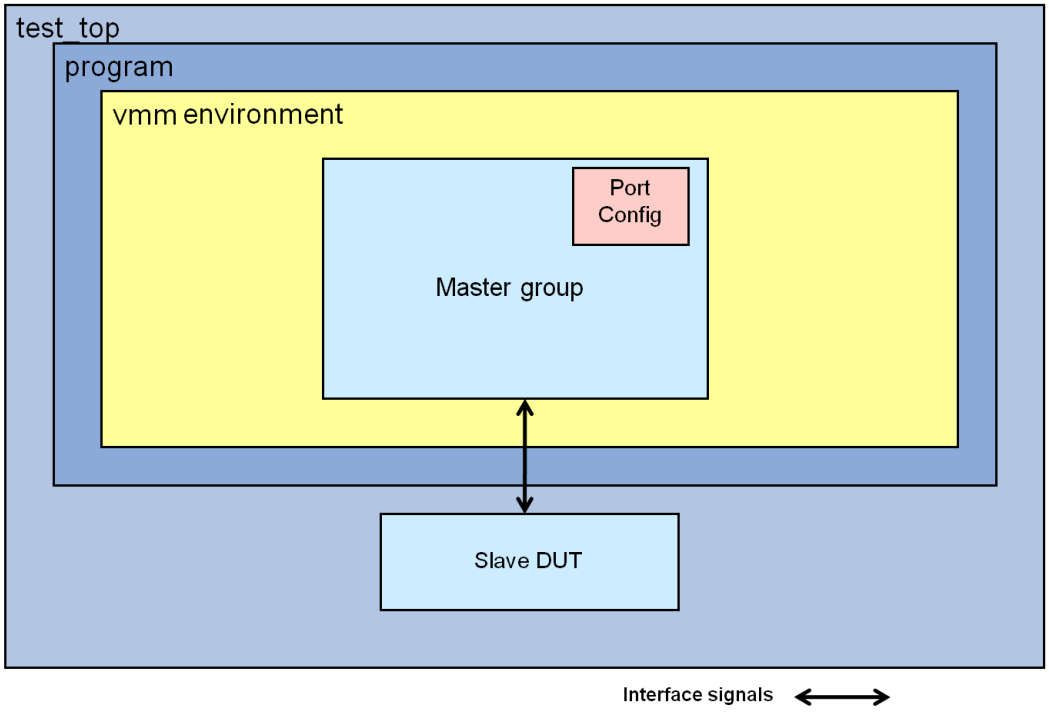
The master group is configured using port configuration, which is available in the system configuration. If the master group is used as a standalone, the port configuration should be provided to the master group in the environment. If the master group is used as part of the system group, the system group would provide the port configuration to the master group.

The master transactor within the master group drives the AXI transactions on the AXI port. The master transactor and monitor components call the callback methods at various phases of execution of the AXI transaction. The details of callbacks are covered in the later sections. After the AXI transaction on the bus is complete, the completed transaction object is provided to the analysis port of the port monitor, for use by the testbench.

The master group also creates vmm_voter objects which are registered with the internal 'vote' consensus object. These are used to consent to or to oppose the end of test conditions based on the state of the generator and the transactor components in the system.

The master component also contains a snoop response generator. If the master component's interface type is configured to support ACE protocol, the snoop response generator is used to provide responses to received snoop transactions.

Figure 3-1 Usage With Master Group



3.2.2 Slave Group

The slave group encapsulates slave transactor, port monitor, and response generator. The slave group can be configured to operate in active mode and passive mode. The response generator of the slave group contains built-in memory, which you can optionally use.

The slave response generator provides callbacks to allow user to update the slave response. For more details on callbacks, see Slave Component Callbacks.

Note

The slave transactor expects the slave response generator to:

- Return same handle of the slave response object as provided to the response generator by the port monitor
- Return the slave response object in zero time, that is, without any delay after response generator receives object from port monitor

If any of the above conditions is violated, the slave issues a FATAL message.

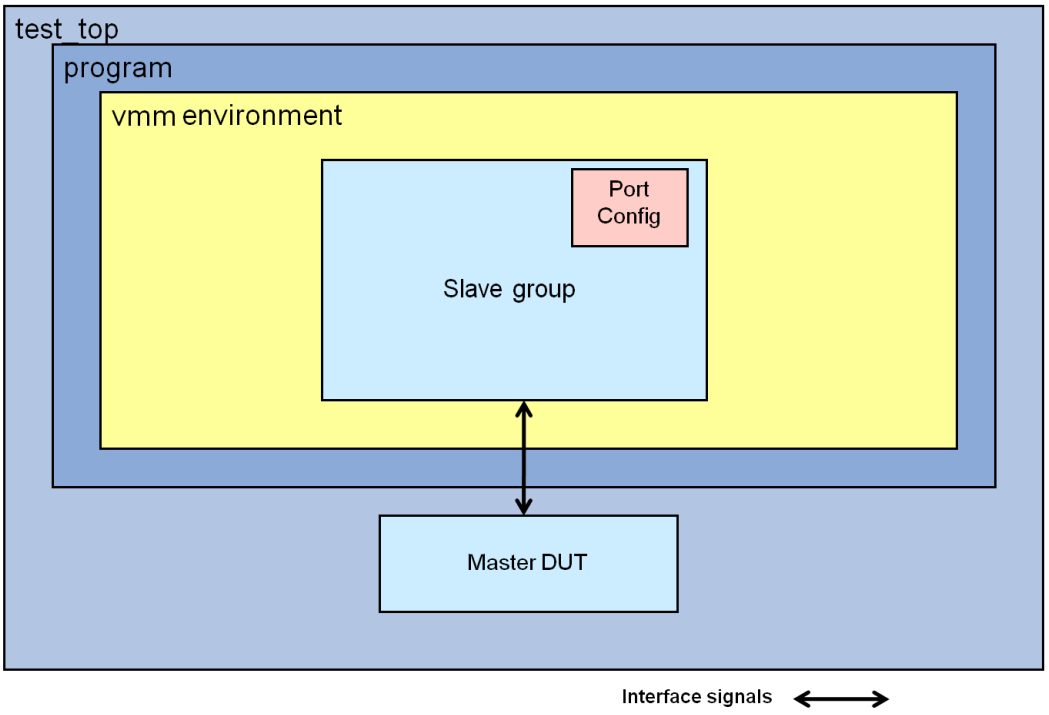
The slave group is configured using port configuration, which is available in the system configuration. If the slave group is used standalone, the port configuration should be provided to the slave group in the environment. If the slave group is used as part of the system group, the system group would provide the port configuration to the slave group.

The slave transactor and slave monitor within the slave group call the callback methods at various phases of execution of the AXI transaction.

For more details on callbacks, see Slave Component Callbacks. After the AXI transaction on the bus is complete, the completed transaction object is provided to the analysis port of the port monitor in the slave group, for use by the testbench.

The slave group also creates vmm_voter objects which are registered with the internal 'vote' consensus object. These are used to consent to or to oppose the end of test conditions based on the state of the generator and the transactor components in the system.

Figure 3-2 Usage With Slave Group



3.2.3 Slave Memory

AXI VIP provides slave memory represented by class svt_mem. Slave memory is instantiated in the slave response generator within the slave group. If the configuration parameter svt_axi_port_configuration::generator_type is set to MEMORY_RESPONSE_GEN, the slave response generator reads/writes data from/to the built-in memory.

Please refer to AXI SVT class reference HTML documentation for details on svt_mem, and svt_axi_port_configuration::generator_type.

3.2.4 FIFO Memory

AXI VIP provides FIFO memory represented by class `svt_axi_fifo_mem`. FIFO memory is instantiated in the slave group as follows:

```
svt_axi_fifo_mem fifo_mem[];
```

This FIFO is configured based on the port configuration parameters `num_fifo_mem` and `fifo_mem_addresses`.

Example configuration to create a single FIFO element is as follows:

```
this.slave_cfg[0].num_fifo_mem = 1;
this.slave_cfg[0].fifo_mem_addresses = new[1];
this.slave_cfg[0].fifo_mem_addresses[0] = 64'h100;
```

The depth of the FIFO is infinite. The width of the FIFO is same as the data width of the interface. These are not configurable.

The FIFO will be accessed by the slave memory sequence (`svt_axi_slave_memory_sequence`) only for the FIXED type bursts. You need to use the `svt_axi_slave_memory_sequence` to access the FIFO.

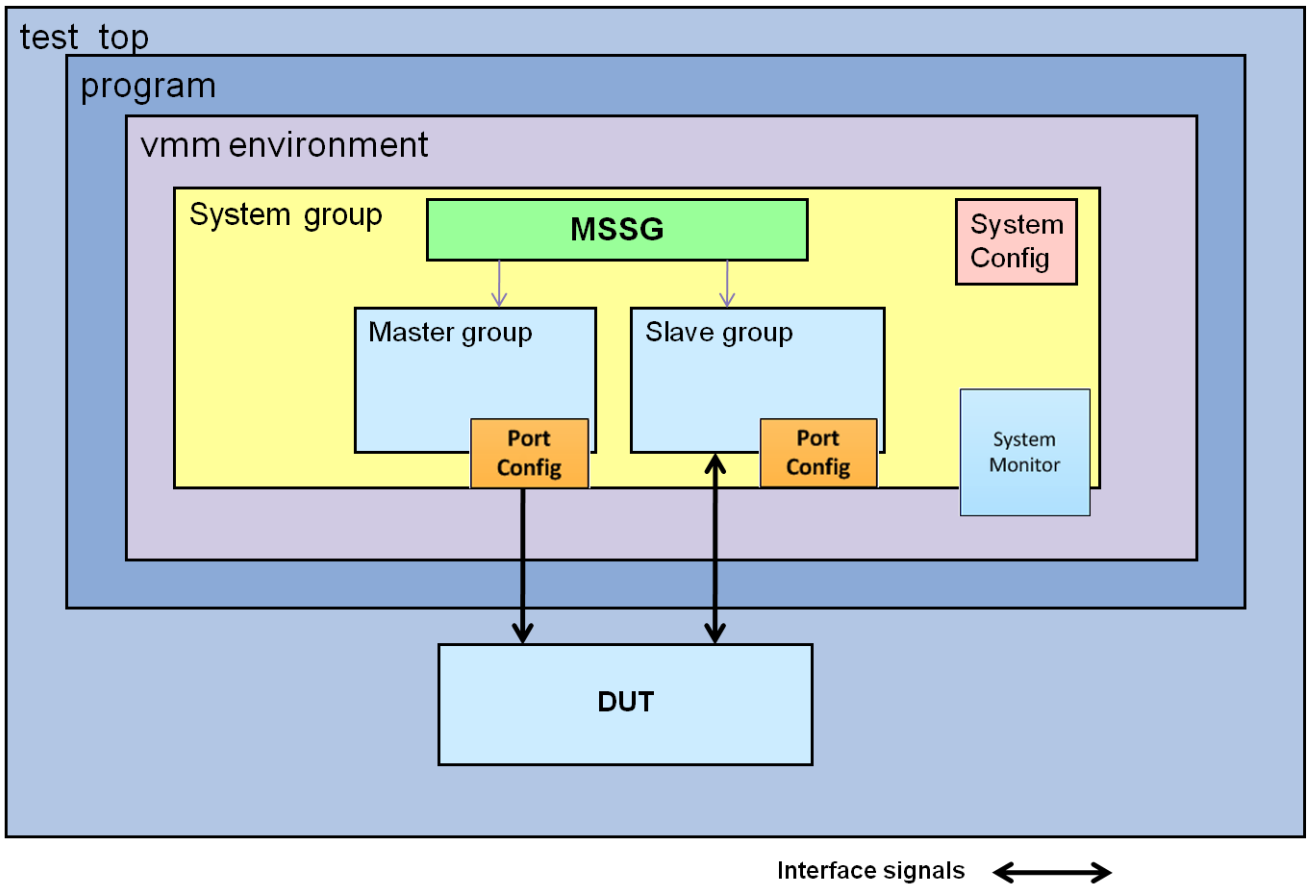
For more information on `svt_axi_fifo_mem`, see the AXI SVT class reference HTML documentation.

3.2.5 System Group

The AXI System group encapsulates the master groups, slave groups, multi-stream scenario generator (MSSG) and the system configuration. The number of master and slave groups are configured based on the system configuration provided by you. In the build_ph phase, the system group builds the Master and Slave groups. After the Master and Slave groups are built, the system group configures these groups using the port configuration information available in the system configuration.

The system group also creates vmm_voter object which are registered with the internal 'vote' consensus object. These are used to consent to or to oppose the end of test conditions based on the state of the generator and the group components in the system.

Figure 3-3 System Group



3.2.5.1 Multi-Stream Scenario Generator (MSSG)

Multi-stream scenario generator (MSG) is encapsulated within the system group. The input channels of master transactors in the system group are registered with the MSSG. The MSSG is created in the build phase of the system group. The MSSG can be used to synchronize stimulus generated by active master groups in the system.

3.2.6 System Monitor

The System Monitor component is instantiated within the AXI System Group component. The System Monitor performs system-level checks across the master and slave ports within the system. The system monitor is enabled by setting the system configuration class member svt_axi_system_configuration::system_monitor_enable.

3.2.6.1 System Checks

The System checks supported by System Monitor fall under the following categories:

- Checks for mapping between ACE coherent transaction and snoop transactions
- Checks for sequencing between ACE coherent and snoop transactions
- Checks for response to coherent transactions based on response to snoop transactions
- Data integrity between ACE coherent transaction data and snoop transaction data

- Data integrity checks for transactions that span across multiple cachelines such as READONCE and WRITEUNIQUE transactions
- Data integrity between cache of all masters
- Data integrity between cache of all masters and slave memory
- Data Integrity across Interconnect master and slave ports
- Transaction routing

There are certain checks which might be design specific. System Monitor provides hooks in the form of callbacks, which can be used by you to perform such design specific checks. For the list of system checks and callbacks, refer to the AXI VIP Class reference HTML documentation.

3.2.7 Active and Passive Mode

Table 3-1 lists the behavior of Master and Slave components in active and passive modes.

Table 3-1 Components in Active and Passive Mode

Component behavior in active mode	Component behavior in passive mode
The Master and Slave components generate transactions on the signal interface.	The Master and Slave components do not generate transactions on the signal interface. These components only sample the signal interface.
The Master and Slave components continue to perform passive functionality of coverage and protocol checking. You can enable/disable this functionality through configuration.	The Master and Slave components monitor the input and output signals, and perform passive functionality such as coverage and protocol checking. You can enable/disable this functionality through configuration options.
The port monitor within the component performs protocol checks only on sampled signals. That is, it does not perform checks on the signals that are driven by the component. This is because when the component is driving an exception (exceptions are not supported in this release) the monitor should not flag an error as it knows that it is driving an exception. Exception means error injection.	The port monitor within the component performs protocol checks on all signals. In passive mode, signals are considered as input signals.
The delay values reported in the AXI transaction provided by the Master and Slave component, are the values provided by you, and not the sampled delay values.	The delay values reported in the AXI transaction provided by the Master and Slave component, are the sampled delay values on the bus.

3.3 AXI VMM User Interface

The following sections give an overview of the user interface into the AXI VMM VIP.

3.3.1 Configuration Objects

Configuration data objects convey the system-level and port-level testbench configuration. The test cases can submit configuration to the VIP during gen_config_ph(), and the configuration will be applied during the build_ph(). The configuration data objects contain built-in constraints, which come into effect when the configuration objects are randomized. If the configuration needs to be changed later, it can be done through reconfigure() method of the master, slave or system component.

The configuration can be of the following two types:

- Static configuration properties

  Static configuration parameters specify configuration which cannot be changed when the system is running. The examples of static configuration parameters are the number of masters and slaves in the system, data bus width, address width.

- Dynamic configuration properties

  Dynamic configuration parameters specify the configuration which can be changed at any time, irrespective of whether the system is running or not. The example of dynamic configuration parameter is timeout values.

AXI VIP defines the following configuration classes:

- System configuration (svt_axi_system_configuration)

  System configuration class contains configuration information which is applicable across the entire AXI system. You can specify the system-level configuration parameters through this class. You should provide the system configuration to the system component from the environment or the test case. The system configuration mainly specifies,

  - Number of Master and Slave components in the system component
  - Port configurations for Master and Slave components
  - Virtual top-level AXI interface
  - Address map (for interconnect component - not currently supported)
  - Timeout values

- Port configuration (svt_axi_port_configuration)

  Port configuration class contains configuration information which is applicable to individual AXI Master or Slave components in the system component. Some of the important information provided by port configuration class are,

  - Active/Passive mode of the Master/Slave port component
  - Enable/disable protocol checks
  - Enable/disable port-level coverage
  - Interface type (AXI3/AXI4/AXI4-Lite/AXI ACE)
  - Port configuration contains the virtual interface for the port

The port configuration objects within the system configuration object are created in the constructor of the system configuration.

For more details on individual members of configuration classes, refer to the AXI VIP Class reference HTML documentation.

3.3.2 Transaction Objects

Transaction objects, which are extended from the vmm_data base class, define a unit of AXI protocol information that is passed across the bus. The attributes of transaction objects are public and are accessed directly for setting and getting values. Most transaction attributes can be randomized. The transaction object can represent the desired activity to be simulated on the bus, or the actual bus activity that was monitored.

AXI transaction data objects store data content and protocol execution information for AXI transactions in terms of timing details of the transactions.

These data objects extend from the vmm_data base class and implement all methods specified by VMM for that class.

AXI transaction data objects are used to:

- Generate random stimulus
- Report observed transactions
- Generate random responses to transaction requests
- Collect functional coverage statistics
- Support error injection

Class properties are public and accessed directly to set and read values. Transaction data objects support randomization and provide built-in constraints. Two set of constraints are provided: valid_ranges and reasonable constraints.

- valid_ranges constraints limit generated values to those acceptable to the drivers. These constraints ensure basic VIP operation and should never be disabled.
- reasonable_* constraints, which can be disabled individually or as a block, limit the simulation by:
  - Enforcing the protocol. These constraints are typically enabled unless errors are being injected into the simulation.
  - Setting simulation boundaries. Disabling these constraints may slow the simulation and introduce system memory issues.

VIP supports extending transaction data classes for customizing randomization constraints. This allows you to disable some reasonable_* constraints and replace them with constraints appropriate to your system.

Individual reasonable_* constraints map to independent fields, each of which can be disabled. The class provides the reasonable_constraint_mode() method to enable or disable blocks of reasonable_* constraints.

AXI VIP defines the following transaction classes:

- AXI Base transaction (svt_axi_transaction)

  This is the base transaction type which contains all the physical attributes of the transaction like address, data, burst type, burst length, etc. It also provides the timing information of the transaction to the Master and Slave drivers, that is, delays for valid and ready signals with respect to some reference events.

  The base transaction also contains a handle to configuration object of type svt_axi_port_configuration, which provides the configuration of the port on which this transaction would be applied. The port configuration is used during randomizing the transaction. The master component initializes the port configuration inside the master transaction and slave component initializes the port configuration inside the slave transaction. If the port configuration handle in the transaction is null at the time of randomization, the transaction will issue a fatal message.

- AXI Master transaction (svt_axi_master_transaction)

  The master transaction class extends from the AXI transaction base class svt_axi_transaction. The master transaction class contains the constraints for master specific members in the base transaction class. At the end of each transaction, the master component provides object of type svt_axi_master_transaction from its analysis ports in active and passive mode.

- AXI Slave transaction (svt_axi_slave_transaction)

The slave transaction class extends from the AXI transaction base class svt_axi_transaction. The slave transaction class contains the constraints for slave specific members in the base transaction class. At the end of each transaction, the slave component provides object of type svt_axi_slave_transaction from its analysis ports, in active and passive mode.

- AXI ACE Snoop Base transaction (svt_axi_snoop_transaction)

  This is the base class for snoop transaction type which contains all the physical attributes of the transaction like address, data, transaction type, etc. It also provides the timing information of the transaction to the master component, that is, delays for valid and ready signals with respect to some reference events.

  The svt_axi_snoop_transaction also contains a handle to configuration object of type svt_axi_port_configuration, which provides the configuration of the port on which this transaction would be applied. The port configuration is used during randomizing the transaction.

- AXI ACE Master Snoop transaction (svt_axi_master_snoop_transaction)

  The master snoop transaction class extends from the snoop transaction base class svt_axi_snoop_transaction. The master snoop transaction class contains the constraints for master specific members in the base transaction class. At the end of each transaction, the master VIP component provides object of type svt_axi_master_snoop_transaction from its analysis ports, in active and passive mode.

For more details on individual members of transaction classes, refer to the AXI VIP Class Reference HTML documentation.

3.3.3 Analysis Port

The port Monitor in the Master and Slave component provides an analysis port "item_observed_port". At the end of the transaction, the Master and Slave components respectively write the completed svt_axi_master_transaction and svt_axi_slave_transaction object to the analysis port. This holds true in active as well as passive mode of operation of the Master/Slave component. You can use this analysis port for connecting to scoreboard, or any other purpose, where a transaction object for the completed transaction is required.

The port monitor in Master component also provides an analysis port called "snoop_item_observed_port". This is used when the Master component is configured with svt_axi_port_configuration::axi_interface_type = AXI_ACE. At the end of the snoop transaction, the master component writes the completed svt_axi_master_snoop_transaction object to the snoop analysis port "snoop_item_observed_port".

3.3.4 Callbacks

Callbacks are an access mechanism that enable the insertion of user-defined code and allow access to objects for scoreboarding and functional coverage. Each Master and Slave component is associated with a callback class that contains a set of callback methods. These methods are called as part of the normal flow of procedural code. There are a few differences between callback methods and other methods that set them apart.

- Callbacks are virtual methods with no code initially, so they do not provide any functionality unless they are extended. The exception to this rule is that some of the callback methods for functional coverage already contain a default implementation of a coverage model.

- The callback class is accessible to you so the class can be extended and your code inserted, potentially including testbench specific extensions of the default callback methods, and testbench specific variables and/or methods used to control whatever behavior the testbench is using the callbacks to support.
- Callbacks are called within the sequential flow at places where external access would be useful. In addition, the arguments to the methods include references to relevant data objects. For example, just before a monitor puts a transaction object into an analysis port is a good place to sample for functional coverage since the object reflects the activity that just happened on the pins. A callback at this point with an argument referencing the transaction object allows this exact scenario.
- There is no need to invoke callback methods for callbacks that are not extended. To avoid a loss of performance, callbacks are not executed by default. To execute callback methods, callback class must be registered with the component using append_callback() method.

AXI VIP uses callbacks in the following main applications:

- Access for functional coverage
- Access for scoreboarding
- Insertion of user-defined code
- Slave response generation

3.3.4.1 Master Component Callbacks

In the Master component, the callback methods are called by master transactor, port monitor transactor, and snoop response generator (when axi_interface_type = AXI_ACE).

The following callback classes which contain the callback methods are invoked by the Master component:

- svt_axi_master_callback
- svt_axi_port_monitor_callback
- svt_axi_master_snoop_response_gen_callback

For details of these classes, refer to the class reference HTML documentation.

3.3.4.2 Slave Component Callbacks

In the Slave component, the callback methods are called by slave transactor, port monitor transactor and slave response generator.

The following callback classes which contain the callback methods are invoked by the Slave component:

- svt_axi_slave_callback
- svt_axi_port_monitor_callback
- svt_axi_slave_response_gen_callback

For details of these classes, refer to the Class Reference HTML documentation.

3.3.5 Interfaces and Modports

SystemVerilog models signal connections using interfaces and modports. Interfaces define the set of signals which make up a port connection. Modports define collection of signals for a given port, the direction of the signals, and the clock with respect to which these signals are driven and sampled.

AXI VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top-level interface svt_axi_if is defined. The top-level interface contains an array of Master port sub-interfaces of type svt_axi_master_if, and Slave port sub-interfaces of type svt_axi_slave_if.

The top-level interface is contained in the system configuration class. The top-level interface is specified to the system configuration class using method svt_axi_system_configuration::set_if.

The port interface is contained in the port configuration class. The port interface is specified to the port configuration class using methods, svt_axi_port_configuration:: set_master_if and svt_axi_port_configuration:: set_slave_if. The port interfaces are provided to the port configuration objects in the constructor of the system configuration.

Refer to $DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_svt_vmm_class_reference/html/interfaces.html for details on AXI Interface.

3.3.5.1 Modports

The port interface svt_axi_master_if contains following modports which you should use to connect VIP to the DUT:

- svt_axi_slave_modport: This modport is used to connect master VIP component to slave DUT port
- svt_axi_debug_modport: This modport can be used by you to access the debug port signals. For more details on debug port, refer to "Using Debug Port".

The port interface svt_axi_slave_if contains the following modports which you should use to connect VIP to the DUT:

- svt_axi_master_modport: This modport is used to connect slave VIP component to master DUT port
- svt_axi_debug_modport: This modport can be used by you to access the debug port signals. For more details on debug port, refer to "Using Debug Port".

3.3.5.2 Clocking Modes

The interface works in the following two clocking modes:

- Common clock mode
- Multiple clock mode

The clock mode can be selected using configuration parameter, svt_axi_system_configuration::common_clock_mode. When set to one, the signal common_aclk in the top interface will be used to drive clock of all port sub-interfaces. In this case, the system clock in the environment will need to be connected to common_aclk signal in the top interface. When this configuration parameter is set to 0, the aclk signal of each port sub-interface would need to be connected to appropriate clock in the environment.

3.3.5.2.1 Common Clock Mode

In this mode:

- All port sub-interfaces will operate on a single common clock
- You need to connect system clock to the common_aclk signal in the top interface
- Top-level interface will pass the common clock signal down to all port sub-interfaces

3.3.5.2.2 Multiple Clock Mode

In this mode, each port interface would operate on a separate port interface clock. In this case, aclk signal in the port interface needs to be connected to the appropriate clock in the environment.

3.3.5.3 Bind Interfaces

AXI VIP also supports bind interfaces for master & slave. Bind interface is an interface which contains directional signals for AXI. Users can connect DUT signals to these directional signals. Bind interfaces provided with VIP are svt_axi_master_bind_if and svt_axi_slave_bind_if. To use bind interface, user still needs to instantiate the non-bind interface, and then connect the bind interface to the non-bind interface. VIP provides master and slave connector modules to connect the VIP bind interface to the VIP non-bind interface. User needs to instantiate a connector module corresponding to each instance of VIP master and slave, and pass the bind interface and non-bind interface instance to this connector module.

Refer to AXI intermediate public example, which demonstrates the usage of bind interface.

3.3.5.4 Parameterized Interfaces

AXI VIP supports parameterized interfaces svt_axi_master_param_if and svt_axi_slave_param_if. These interfaces are parameterized for signal widths. The default value of all the parameters are same as the system constants defined in svt_axi_port_defines.svi (refer to Step 3.3.7). These interface parameters can be changed to match the DUT signal widths. The parameter value should be less than or equal to the system constant defined in svt_axi_port_defines.svi or svt_axi_user_defines.svi.

To use parameterized interface, the user still needs to instantiate the top-level interface svt_axi_if. The svt_axi_master_param_if interface should be used for connecting AXI Master VIP component to the DUT and svt_axi_slave_param_if interface should be used to connect AXI Slave VIP component to the DUT.

Currently, there is no separate example for parameterized interface for VMM, hence, refer to the UVM example tb_axi_svt_uvm_basic_param_if_sys for usage of parameterized interface. The README file in the example describes the usage.

3.3.6 Notifications

Master components issue svt_axi_master_transaction::STARTED and svt_axi_master_transaction::ENDED notifications. Slave components issue svt_axi_slave_transaction::STARTED and svt_axi_slave_transaction::ENDED notifications. These notifications denote start of transaction and end of transaction events. These notifications are issued by the Master and Slave component as described below, in both active and passive mode.

For WRITE transactions, STARTED notification is issued on the rising clock edge when awvalid (for address before data) or wvalid (for data before address) is high.

For READ transactions, STARTED notification is issued on the rising clock edge when arvalid is high.

For WRITE transactions, the ENDED notification is issued on the rising clock edge when bvalid and bready both are high.

For READ transactions, the ENDED notification is issued on the rising clock edge when rvalid, rlast and rready are high.

3.3.7 Overriding System Constants

VIP uses include files to define system constants that, in some cases, you may override so the VIP matches your expectations. For example, you can override the maximum delay values. You can also adjust the default simulation footprint, like maximum address width.

The system constants for the VIP are specified (or referenced) in the following files (the first three files reside at $DESIGNWARE_HOME/vip/svt/amba_svt/latest/sverilog/include:

svt_axi_defines.svi: Top-level include file; allows for the inclusion of the common define symbols and the port define symbols in a single file. Also, it contains a `include to read user overrides if the `SVT_AXI_INCLUDE_USER_DEFINES symbol is defined.

svt_axi_common_defines.svi: Defines common constants used by the AXI VIP components. You can override only the "User Definable" constants, which are declared in "ifndef" statements, such as,

```
`ifndef SVT_AXI_MAX_ADDR_VALID_DELAY
 `define SVT_AXI_MAX_ADDR_VALID_DELAY 16
`endif
```

svt_axi_port_defines.svi: Contains the constants that set the default maximum footprint of the environment. These values determine the wire bit widths in the 'wire frame'-- they do not (necessarily) define the actual bit widths used by the components, which is determined by the configuration classes.

svt_axi_user_defines.svi: Contains override values that you define. This file can reside anywhere-- specify its location on the simulator command line.

To override the SVT_AXI_MAX_ID_WIDTH constant from the svt_axi_port_defines.svi file:

- Redefine the corresponding symbol in the svt_axi_user_defines.svi file. For example:

  `define SVT_AXI_MAX_ID_WIDTH 12
- In the simulator compile command:
  - Ensure that the directory containing svt_axi_user_defines.svi is provided to the simulator
  - Provide SVT_AXI_INCLUDE_USER_DEFINES on the simulator command line as follows:

    +define+SVT_AXI_INCLUDE_USER_DEFINES

Note the following restrictions when overriding the default maximum footprint:

- Never use a value of 0 for a MAX_*_WDTH value. The value must be >= 1
- The maximum footprint set at compile time must work for the full design. If you are using multiple instances of AXI VIP, only one maximum footprint can be set and must therefore satisfy the largest requirement.
- The value of less than 32 is not supported for SVT_AXI_MAX_ADDR_WIDTH. SVT_AXI_MAX_ADDR_WIDTH only defines the footprint of address port. The actual used address with is defined by svt_axi_port_configuration::addr_width, which can still be configured to less than 32.

3.4 Functional Coverage

The AXI VMM VIP provides various levels of coverage support. This section describes those levels of support.

3.4.1 Default Coverage

Below is the description of default coverage provided with AXI VIP. For more details on actual cover groups, refer to the AXI VIP Class Reference HTML document.

3.4.1.1 Toggle Coverage

Toggle coverage is a signal-level coverage. Toggle coverage provides baseline information that a system is connected properly, and that higher level coverage or compliance failures are not simply the result of connectivity issues.

Toggle coverage answers the question: Did a bit change from a value of 0 to 1 and back from 1 to 0? This type of coverage does not indicate that every value of a multi-bit vector was seen but measures that all the individual bits of a multi-bit vector did toggle.

The toggle coverage is currently not supported.

3.4.1.2 State Coverage

State coverage is a signal-level coverage. State coverage applies to signals that are a minimum of two bits wide. In most cases, the states (also commonly referred to as coverage bins) can be easily identified as all possible combinations of the signal. For example, for the RRESP[1:0] signal, the states would be 00 (OKAY), 01 (EXOKAY), 10 (SLVERR) and 11 (DECERR). If the state space is too large, an intelligent classification of the states must be made. In the case of the AWADDR signal for example, coverage bins would be one bin to cover the lower address range, one bin to cover the upper address range and one bin to cover all other intermediary addresses.

3.4.1.3 Delay Coverage

Delay coverage is a coverage on various delays between valid and ready signals. The following valid to ready delays are covered:

- Write Address handshake delay
- Read Address handshake delay
- Write Data handshake delay
- Read Data handshake delay
- Write Response handshake delay

3.4.1.4 Transaction Cross Coverage

Cross coverage specifies interesting cross coverage across AXI signals. Table 3-2 shows the cross coverage points.

Table 3-2 AXI3/4 Transaction-Level Cross Coverage

awburst/arburst	awlen/arlen	awaddr/araddr	awid/arid	awsize/arsize	wstrb	awlock/arlock	awcache/awcache	awprot/arprot	bresp/rresp	awqos/arqos	Cross Description
X	X										Cross all transaction types with certain ranges of lengths like SINGLE & BURSTS Covergroup names: trans_cross_axi_arburst_arlen trans_cross_axi_awburst_awlen
X	X	X									Cross all transaction types with all targets Covergroup names: trans_cross_axi_arburst_arlen_araddr trans_cross_axi_awburst_awlen_awaddr
X	X								X		Cross all transaction types with all transaction responses Covergroup names: trans_cross_axi_arburst_arlen_rresp trans_cross_axi_awburst_awlen_bresp
X	X			X							Cross all transaction types with all transaction sizes Covergroup names: trans_cross_axi_arburst_arlen_arsize trans_cross_axi_awburst_awlen_awsize
X	X					X					Cross all transaction types with all transaction access types Covergroup names: trans_cross_axi_arburst_arlen_arlock trans_cross_axi_awburst_awlen_awlock
X	X	X		X							Cross all transaction types with all targets with all sizes to cover aligned and unaligned transactions Covergroup names: trans_cross_axi_arburst_arlen_araddr_arsize trans_cross_axi_awburst_awlen_awaddr_awsize
X	X						X				Cross all transaction types with all cache types Covergroup names: trans_cross_axi_arburst_arlen_arcache trans_cross_axi_awburst_awlen_awcache

Table 3-2 AXI3/4 Transaction-Level Cross Coverage (Continued)

awburst/arburst	awlen/arlen	awaddr/araddr	awid/arid	awsize/arsize	wstrb	awlock/arlock	awcache/awcache	awprot/arprot	bresp/rresp	awqos/arqos	Cross Description
X	X							X			Cross all transaction types with all protection types Covergroup names: trans_cross_axi_arburst_arlen_arprot trans_cross_axi_awburst_awlen_awprot
X										X	Cross all transaction types with all QOS values Covergroup names: trans_cross_axi_arburst_arqos trans_cross_axi_awburst_awqos

Tables 3-3 shows the cross coverage points for coherent transactions in ACE protocol.

Table 3-3 ACE Coherent Transaction-Level Cross Coverage

awsnoop/arsnoop	awdomain/ardomain	awbar/arbar	awburst/awburst	awlen/arlen	awaddr/araddr	awsize/arsize	awcache/awcache	bresp/rresp	Cross Description
X	X								Cross all coherent transaction types with domain types Covergroup names: trans_cross_ace_awsnoop_awdomain trans_cross_ace_arsnoop_ardomain
X		X							Cross all coherent transaction types with barrier types Covergroup names: trans_cross_ace_awsnoop_awbar trans_cross_ace_arsnoop_arbar
X			X						Cross all coherent transaction types with burst type Covergroup names: trans_cross_ace_arsnoop_arburst trans_cross_ace_awsnoop_awburst
X				X					Cross all coherent transaction types with burst length Covergroup names: trans_cross_ace_arsnoop_arlen trans_cross_ace_awsnoop_awlen

Table 3-3 ACE Coherent Transaction-Level Cross Coverage (Continued)

awsnoop/arsnoop	awdomain/ardomain	awbar/arbar	awburst/awburst	awlen/arlen	awaddr/araddr	awsize/arsize	awcache/awcache	bresp/rresp	Cross Description
X					X				Cross all coherent transaction types with address ranges Covergroup names: trans_cross_ace_arsnoop_araddr trans_cross_ace_awsnoop_awaddr
X						X			Cross all coherent transaction types with burst size Covergroup names: trans_cross_ace_arsnoop_arsize trans_cross_ace_awsnoop_awsize
X							X		Cross all coherent transaction types with cache types Covergroup names: trans_cross_ace_arsnoop_arcache trans_cross_ace_awsnoop_awcache
X								X	Cross all coherent transaction types with response Covergroup names: trans_cross_ace_arsnoop_coh_rresp trans_cross_ace_awsnoop_bresp
					X			X	Cross DVM Message type with DVM response Covergroup names: trans_cross_ace_acdvmmessage_acdvmresp

Table 3-4 shows the cross coverage points for snoop transactions in ACE protocol.

Table 3-4 ACE Snoop Transaction-Level Cross Coverage

acsnoop	acaddr	acprot	crresp[4:0]	Cross Description
X	**X**			Cross all Snoop transaction types with address ranges Covergroup names: trans_cross_ace_acsnoop_acaddr
X		**X**		Cross all Snoop transaction types with Snoop Protection types Covergroup names: trans_cross_ace_acsnoop_acprot

Table 3-4 ACE Snoop Transaction-Level Cross Coverage (Continued)

acsnoop	acaddr	acprot	crresp[4:0]	Cross Description
X			X	Cross all Snoop transaction types with Snoop responses Covergroup names: trans_cross_ace_acsnoop_crresp
	X		X	Cross DVM Message type with DVM response Covergroup names: trans_cross_ace_acdvmmessage_acdvmresp

Table 3-5 shows the coverage for cache state transitions. In addition, the table shows the coverage which is not yet supported (marked with "NS").

Table 3-5 ACE Cache State Transition Coverage

Coherent & Snoop Transactions				Cache State Transition
Read Coherent	Write Coherent	Snoop	Speculative Read	
Cover Group: trans_cross_ace_arsnoop_cacheinitialstate_cachefinalstate	Cover Group: trans_cross_ace_awsnoop_cacheinitialstate_cachefinalstate	Cover Group: trans_cross_ace_acsnoop_cacheinitialstate_cachefinalstate	NS	Start State -> Expected End State
NS	NS	Not Applicable	NS	Start State -> Legal End State (with Snoop Filter)
NS	NS	Not Applicable	NS	Start State -> Legal End State (no Snoop Filter)

3.4.2 Coverage Callback Classes

3.4.2.1 Coverage Data Callbacks

This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses "def_cov_data" in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage requests by setting the coverage data and triggering the coverage events. This implementation does not include any coverage groups. The def_cov_data callbacks classes are extended from component callback class.

The coverage data callback class is extended from callback class svt_axi_port_monitor_callback. The extended class is called svt_axi_port_monitor_def_cov_data_callback.

The following callback methods are implemented for triggering coverage events:

- pre_output_port_put

- write_address_phase_ended
- read_address_phase_ended
- write_data_phase_ended
- read_data_phase_ended
- write_resp_phase_ended

3.4.2.2 Coverage Callbacks

This class is extended from the coverage data callback class. The naming convention uses "def_cov" in the class names for easy identification of these classes. It includes default cover groups based on the data and events defined in the data class.

The coverage callback class implementing default cover groups is called svt_axi_port_monitor_def_cov_callback.

3.4.3 Enabling Default Coverage

The default functional coverage can be enabled by setting the following attributes in the port configuration class svt_axi_port_configuration to '1'. To disable coverage, set the attributes to '0'. The attributes are:

- toggle_coverage_enable
- state_coverage_enable
- transaction_coverage_enable

By default, the coverage is disabled.

3.4.4 Coverage Shaping and Control

The handle to the port configuration class svt_axi_port_configuration is provided to the class svt_axi_port_monitor_def_cov_callback, which implements the default cover groups. Based on the port configuration, the coverage bins are shaped. The unwanted bins are ignored. For example, all the burst size bins greater than svt_axi_port_configuration::data_width are ignored.

In addition to above, you also have the ability to disable coverage at cover group level. Class svt_axi_port_configuration provides members svt_axi_port_configuration::<cover_group_name>_enable, to enable/disable cover groups. By default, the value to these members is 1. For example, to disable cover group trans_cross_axi_awburst_awlen, member svt_axi_port_configuration:: trans_cross_axi_awburst_awlen_enable should be set to 0.

3.5 Protocol Checks

The protocol checks can be enabled by setting the configuration attribute protocol_checks_enable in class svt_axi_port_configuration to '1'. To disable checks, set the attribute to '0'. By default, the protocol checks are enabled.

3.5.1 Comprehensive List of Protocol Checks

Important AXI3/4 protocol checks are described in the below sections. For a comprehensive list of all protocol checks for AXI3/4/ACE protocol, refer to the svt_axi_checker class in AXI VIP Class reference HTML documentation.

.

Table 3-6 Write Address Channel Checks

Protocol check	Check condition
signal_valid_awid_when_awvalid_high_check	AWID is not X or Z when AWVALID is high
signal_valid_awaddr_when_awvalid_high_check	AWADDR is not X or Z when AWVALID is high
signal_valid_awlen_when_awvalid_high_check	AWLEN is not X or Z when AWVALID is high
signal_valid_awsize_when_awvalid_high_check	AWSIZE is not X or Z when AWVALID is high
signal_valid_awburst_when_awvalid_high_check	AWBURST is not X or Z when AWVALID is high
signal_valid_awlock_when_awvalid_high_check	AWLOCK is not X or Z when AWVALID is high
signal_valid_awcache_when_awvalid_high_check	AWCACHE is not X or Z when AWVALID is high
signal_valid_awprot_when_awvalid_high_check	AWPROT is not X or Z when AWVALID is high
signal_valid_awready_when_awvalid_high_check	AWREADY is not X or Z when AWVALID is high
signal_stable_awid_when_awvalid_high_check	AWID is stable when AWVALID is high
signal_stable_awaddr_when_awvalid_high_check	AWADDR is stable when AWVALID is high
signal_stable_awlen_when_awvalid_high_check	AWLEN is stable when AWVALID is high
signal_stable_awsize_when_awvalid_high_check	AWSIZE is stable when AWVALID is high
signal_stable_awburst_when_awvalid_high_check	AWBURST is stable when AWVALID is high
signal_stable_awlock_when_awvalid_high_check	AWLOCK is stable when AWVALID is high
signal_stable_awcache_when_awvalid_high_check	AWCACHE is stable when AWVALID is high
signal_stable_awprot_when_awvalid_high_check	AWPROT is stable when AWVALID is high
signal_valid_awvalid_check	AWVALID is not X or Z
awvalid_interrupted_check	AWVALID was interrupted before awready got asserted
awburst_reserved_val_check	The value of awburst=2'b11 when awvalid is high
awvalid_awcache_active_check	The value of awcache[3:2]=2'b00 when awvalid is high and awcache[1] is also low
awsize_data_width_active_check	size of write transfer exceeds the width of the data bus
awaddr_wrap_aligned_active_check	write burst of WRAP type has an aligned address
awlen_wrap_active_check	write burst of WRAP type has a valid length
awaddr_4k_boundary_cross_active_check	write burst cross a 4KB boundary

Table 3-7 Write Data Channel Checks

Protocol check	Check condition
signal_valid_wid_when_wvalid_high_check	WID is not X or Z when WVALID is high
signal_valid_wdata_when_wvalid_high_check	WDATA is not X or Z when WVALID is high
signal_valid_wstrb_when_wvalid_high_check	WSTRB is not X or Z when WVALID is high
signal_valid_wlast_when_wvalid_high_check	WLAST is not X or Z when WVALID is high
signal_valid_wready_when_wvalid_high_check	WREADY is not X or Z when WVALID is high
signal_stable_wid_when_wvalid_high_check	WID is stable when WVALID is high
signal_stable_wdata_when_wvalid_high_check	WDATA is stable when WVALID is high
signal_stable_wstrb_when_wvalid_high_check	WSTRB is stable when WVALID is high
signal_stable_wlast_when_wvalid_high_check	WLAST is stable when WVALID is high
signal_valid_wvalid_check	WVALID is not X or Z
wvalid_interrupted_check	WVALID was interrupted before WREADY got asserted
wdata_awlen_match_for_corresponding_awaddr_check	The number of write data items matches AWLEN for the corresponding address

Table 3-8 Write Response Channel Checks

Protocol check	Check condition
signal_valid_bid_when_bvalid_high_check	BID is not X or Z when BVALID is high
signal_valid_bresp_when_bvalid_high_check	BRESP is not X or Z when BVALID is high
signal_valid_bready_when_bvalid_high_check	BREADY is not X or Z when BVALID is high
signal_stable_bid_when_bvalid_high_check	BID is stable when BVALID is high
signal_stable_bresp_when_bvalid_high_check	BRESP is stable when BVALID is high
signal_valid_bvalid_check	BVALID is not X or Z
write_resp_follows_last_write_xfer_check	A transaction with corresponding ID whose data phase is complete, when a write response is sampled
wlast_asserted_for_last_write_data_beat	WLAST is asserted for the last beat of write data
bvalid_interrupted_check	bvalid was interrupted before bready got asserted

Table 3-8 Write Response Channel Checks (Continued)

Protocol check	Check condition
write_resp_after_last_wdata_check	The slave must only give a write response after the last write data item is transferred
write_resp_after_write_addr_check	A slave must not give a write response before the write address

Table 3-9 Read Address Channel Checks

Protocol check	Check condition
signal_valid_arid_when_arvalid_high_check	ARID is not X or Z when ARVALID is high
signal_valid_araddr_when_arvalid_high_check	ARADDR is not X or Z when ARVALID is high
signal_valid_arlen_when_arvalid_high_check	ARLEN is not X or Z when ARVALID is high
signal_valid_arsize_when_arvalid_high_check	ARSIZE is not X or Z when ARVALID is high
signal_valid_arburst_when_arvalid_high_check	ARBURST is not X or Z when ARVALID is high
signal_valid_arlock_when_arvalid_high_check	ARLOCK is not X or Z when ARVALID is high
signal_valid_arcache_when_arvalid_high_check	ARCACHE is not X or Z when ARVALID is high
signal_valid_arprot_when_arvalid_high_check	ARPROT is not X or Z when ARVALID is high
signal_valid_arready_when_arvalid_high_check	ARREADY is not X or Z when ARVALID is high
signal_stable_arid_when_arvalid_high_check	ARID is stable when ARVALID is high
signal_stable_araddr_when_arvalid_high_check	ARADDR is stable when ARVALID is high
signal_stable_arlen_when_arvalid_high_check	ARLEN is stable when ARVALID is high
signal_stable_arsize_when_arvalid_high_check	ARSIZE is stable when ARVALID is high
signal_stable_arburst_when_arvalid_high_check	ARBURST is stable when ARVALID is high
signal_stable_arlock_when_arvalid_high_check	ARLOCK is stable when ARVALID is high
signal_stable_arcache_when_arvalid_high_check	ARCACHE is stable when ARVALID is high
signal_stable_arprot_when_arvalid_high_check	ARPROT is stable when ARVALID is high
signal_valid_arvalid_check	ARVALID is not X or Z
arvalid_interrupted_check	ARVALID was interrupted before arready got asserted
arburst_reserved_val_check	The value of ARBURST=2'b11 when arvalid is high
arvalid_arcache_active_check	The value of ARCACHE[3:2]=2'b00 when arvalid is high and ARCACHE[1] is also low

Table 3-9 Read Address Channel Checks

Protocol check	Check condition
arsize_data_width_active_check	Size of read transfer exceeds the width of the data bus
araddr_wrap_aligned_active_check	Read burst of WRAP type has an aligned address
arlen_wrap_active_check	Read burst of WRAP type has a valid length
araddr_4k_boundary_cross_active_check	Read burst cross a 4KB boundary

Table 3-10 Read Data Channel Checks

Protocol check	Check condition
signal_valid_rid_when_rvalid_high_check	RID is not X or Z when RVALID is high
signal_valid_rdata_when_rvalid_high_check	RDATA is not X or Z when RVALID is high
signal_valid_rresp_when_rvalid_high_check	RRESP is not X or Z when RVALID is high
signal_valid_rlast_when_rvalid_high_check	RLAST is not X or Z when RVALID is high
signal_valid_rready_when_rvalid_high_check	RREADY is not X or Z when RVALID is high
signal_stable_rid_when_rvalid_high_check	RID is stable when RVALID is high
signal_stable_rdata_when_rvalid_high_check	RDATA is stable when RVALID is high
signal_stable_rresp_when_rvalid_high_check	RRESP is stable when RVALID is high
signal_stable_rlast_when_rvalid_high_check	RLAST is stable when RVALID is high
read_data_follows_addr_check	Sample read data has associated address
signal_valid_rvalid_check	RVALID is not X or Z
rvalid_interrupted_check	RVALID was interrupted before rready got asserted
rdata_arlen_match_for_corresponding_araddr_check	The number of read data items matches ARLEN for the corresponding address

3.5.2 AXI4 Protocol Checks

Table 3-11 Write Address Channel Checks

Protocol check	Check condition
signal_valid_awqos_when_awvalid_high_check	AWQOS is not X or Z when AWVALID is high
signal_valid_awregion_when_awvalid_high_check	AWREGION is not X or Z when AWVALID is high
signal_valid_awuser_when_awvalid_high_check	AWUSER is not X or Z when AWVALID is high

Table 3-11 Write Address Channel Checks

Protocol check	Check condition
signal_stable_awqos_when_awvalid_high_check	AWQOS is stable when AWVALID is high
signal_stable_awregion_when_awvalid_high_check	AWREGION is stable when AWVALID is high
signal_stable_awuser_when_awvalid_high_check	AWUSER is stable when AWVALID is high

Reset Functionality

Table 3-12 Write Data Channel Checks

Protocol check	Check condition
signal_valid_wuser_when_wvalid_high_check	WUSER is not X or Z when WVALID is high
signal_stable_wuser_when_wvalid_high_check	WUSER is stable when WVALID is high

Table 3-13 Write Response Channel Checks

Protocol check	Check condition
signal_valid_buser_when_bvalid_high_check	BUSER is not X or Z when BVALID is high
signal_stable_buser_when_bvalid_high_check	BUSER is stable when BVALID is high

Table 3-14 Read Address Channel Checks

Protocol check	Check condition
signal_valid_arqos_when_arvalid_high_check	ARQOS is not X or Z when ARVALID is high
signal_valid_arregion_when_arvalid_high_check	ARREGION is not X or Z when ARVALID is high
signal_valid_aruser_when_arvalid_high_check	ARUSER is not X or Z when ARVALID is high
signal_stable_arqos_when_arvalid_high_check	ARQOS is stable when ARVALID is high
signal_stable_arregion_when_arvalid_high_check	ARREGION is stable when ARVALID is high
signal_stable_aruser_when_arvalid_high_check	ARUSER is stable when ARVALID is high

Table 3-15 Read Data Channel Checks

Protocol check	Check condition
signal_valid_ruser_when_rvalid_high_check	RUSER is not X or Z when RVALID is high
signal_stable_ruser_when_rvalid_high_check	RUSER is stable when RVALID is high

3.6 Reset Functionality

The AXI VIP samples the assertion of reset signal asynchronously, and de-assertion of reset signal asynchronously. When reset is de-asserted, VIP detects it with or without clock being present. However, it starts driving or sampling signals from the first clock edge received after the reset is de-asserted.

The AXI port configuration attribute `svt_axi_port_configuration::reset_type` controls the behavior of AXI VIP during the reset.

3.6.1 Behavior when svt_axi_port_configuration::reset_type = EXCLUDE_UNSTARTED_XACT (default value)

When reset signal is asserted, all the transactions in the master and slave groups whose address, data, response phases that are in progress are ABORTED. All the aborted transactions are provided from the analysis port `item_observed_port` of the master and slave groups. The `addr_status`, `data_status`, and `write_resp_status` fields of these transactions reflect the value as `ABORTED`. The transactions which are not yet started by the master group are resumed after the reset signal of master group is de-asserted. For `READ` or `WRITE` transactions whose address phase is complete, and the data or response phase is in progress, the transaction ENDED notification is issued on the rising edge of the clock during reset signal assertion.

3.6.2 Behavior when svt_axi_port_configuration::reset_type reset_type = RESET_ALL_XACT

When reset signal is asserted, all the transactions in master and slave groups are ABORTED. For the master group, this includes the transactions whose address, data, response phases are in progress, and also the transactions that are not yet started by the master group (present in the active queue). All the aborted transactions are provided from the analysis port `item_observed_port` of the master and slave groups. The `addr_status`, `data_status`, and `write_resp_status` fields of these transactions reflect the value as `ABORTED`. For `READ` or `WRITE` transactions whose address phase is complete, and the data or response phase is in progress, transaction ENDED notification is issued on the rising edge of the clock during reset signal assertion.

3.7 Support for ACE Protocol in AXI Master

The AXI VIP supports the ARM ACE protocol specification. This support is provided in the axi_master_group component. The axi_master_group component contains a snoop response generator, which responds back to the snoop transactions. It also contains a cache model.

The axi_master_group component supports,

- Generating coherent transactions
- Responding to snoop transactions
- Allocating data and updating state of the cache model, based on generated coherent transactions and received snoop transactions
- Back door access to the cache in the master

3.7.1 Support for Coherent Transactions

When the svt_axi_port_configuration::axi_interface_type is AXI_ACE, all transactions are of type COHERENT i.e., svt_axi_transaction::xact_type is COHERENT. You will set the different types of coherent transactions in svt_axi_transaction::coherent_xact_type. The following section describes the behavior of the AXI master VIP for each of the coherent transaction types. See ACE protocol specification for a list of start

states and expected end states for each of the transaction types. State transitions of the cache model conform to those specified in the ACE protocol specification.

3.7.1.1 ReadNoSnoop

Before transmission:

A ReadNoSnoop transaction is always sent out on the bus.

After read response completion and before RACK transmission:

No allocation is made.

3.7.1.2 ReadOnce

Before Transmission:

A ReadOnce transaction is always sent out on the bus.

After read response completion and before RACK transmission:

No Allocation is made.

3.7.1.3 ReadClean/ReadShared/ReadNotSharedDirty

Before Transmission:

If cache line state is anything other than Invalid, the transaction is not sent out on the bus. Instead, the data in cache is returned in member svt_axi_transaction::data[]. Otherwise, the transaction is transmitted on the bus.

After read response completion and before RACK transmission:

Cache allocation is made based on the data available in svt_axi_transaction::data[].

3.7.1.4 ReadUnique

Such a transaction is normally used when a master wants to do a partial write and it does not have a copy of the line. So it gets a unique copy and then stores into the line.

Before transmission:

If cache line state is anything other than Invalid, the transaction is not sent out on the bus. Instead, the data in cache is returned in member svt_axi_transaction::data[]. Otherwise, the transaction is sent out on the bus.

After read response completion and before RACK transmission:

If svt_axi_transaction::allocate_in_cache is set, cache is allocated. The data to be allocated must be set in member svt_axi_transaction::cache_write_data. The value of member svt_axi_transaction::cache_write_data[] can be programmed upfront, or can also be updated in the pre_cache_update callback. If the svt_axi_transaction::allocate_in_cache bit is not set, the data available in the svt_axi_transaction::data[] field is stored in the cache.

3.7.1.5 CleanUnique

This transaction is normally used when a master wants to do a partial write and has a copy of the line. So it invalidates the line in all other masters and causes dirty data to be written to memory.

Before transmission:

Transaction is transmitted only if the state of cache line is NOT Unique.

After read response completion and before RACK transmission:

If svt_axi_transaction::allocate_in_cache is set, cache is allocated. The data to be allocated must be set in member svt_axi_transaction::cache_write_data.

The value of member svt_axi_transaction::cache_write_data[] can be programmed upfront, or can also be updated in the pre_cache_update callback.

3.7.1.6 CleanShared

Before transmission:

If the line was in a UniqueClean state, the transaction is not transmitted. If the line is in a dirty state, the VIP automatically generates a `WRITEBACK` or `WRITECLEAN` transaction to first write data to memory. The property `svt_axi_transaction::is_auto_generated` is set for the `WRITEBACK/WRITECLEAN` transaction which is auto generated by the VIP. After the `WRITEBACK/WRITECLEAN` transaction is sent, the `CLEANSHARED` transaction is sent.

After read response completion and before RACK transmission:

There is no change in the data in cache. Only the state of the cache line changes.

3.7.1.7 CleanInvalid

Before transmission:

If the line is in a dirty state, the VIP automatically generates a `WRITEBACK` or `WRITECLEAN` transaction to first write data to memory. The property `svt_axi_transaction::is_auto_generated` is set for the `WRITEBACK/WRITECLEAN` transaction which is auto generated by the VIP. After the `WRITEBACK/WRITECLEAN` transaction is sent, the `CLEANINVALID` transaction is sent.

After read response completion and before RACK transmission:

No allocation in cache is made.

3.7.1.8 MakeUnique

Before Transmission:

If line is already in unique state, transaction is not transmitted on the bus. Else, transaction is transmitted.

After read response completion and before RACK transmission:

Model automatically allocates the cache line, and stores the data in member svt_axi_transaction::cache_write_data[] into the cache line. The value of member svt_axi_transaction::cache_write_data[] can be programmed upfront, or can also be updated in the pre_cache_update callback.

3.7.1.9 MakeInvalid

Before Transmission:

If the line is in Valid state, the line is first invalidated.

After read response completion and before RACK transmission:

Line should be in Invalid state. No data is stored in cache.

3.7.1.10 WriteNoSnoop

Before Transmission:

WriteNoSnoop transaction is transmitted if cache line is in UniqueClean or UniqueDirty state.

After write response completion and before WACK transmission:

The cache line state becomes UniqueClean.

3.7.1.11 WriteUnique/WriteLineUnique

Before transmission:

Transaction is dropped if cache line is not in required state.

After write response completion and before WACK transmission:

There is no change in the state or the contents of the cache.

3.7.1.12 WriteBack

Before transmission:

If line is not in dirty state, the transaction is dropped.

After write response completion and before WACK transmission:

Line is invalidated in the cache.

3.7.1.13 WriteClean

Before transmission:

If line is not in dirty state, the transaction is dropped.

After write response completion and before WACK transmission:

Line remains allocated in the cache.

3.7.1.14 Evict

Before transmission:

If the line is not in Clean state, transaction is dropped.

After write response completion and before WACK transmission:

Line is invalidated in the cache.

3.7.2 Support for Snoop Transactions

For received snoop transaction types, the response conforms to ACE protocol specification. The snoop response is provided by the snoop response generator in the master group. The snoop response generator can be programmed in one of the following modes using member svt_axi_port_configuration::snoop_response_generator_type.

3.7.2.1 Cache Based Snoop Response

This mode is set by specifying svt_axi_port_configuration::snoop_response_generator_type to CACHE_SNOOP_RESPONSE_GEN. In this mode, the snoop response generator provides automatic response to snoop transactions. In this mode, snoop response is based on the contents and state of the cache line of the cache instantiated in the master. The member svt_axi_snoop_transaction::snoop_resp_wasunique and svt_axi_snoop_transaction::snoop_data[] reflect the state and contents of the cache line. All the other fields are randomized based on the initial state of the cache line.

3.7.2.2 User Snoop Response

This mode is set by specifying svt_axi_port_configuration::snoop_response_generator_type to USER_SNOOP_RESPONSE_GEN. In this mode, you are responsible to provide the appropriate snoop response.

You need to perform the following steps:

- Extend callback class svt_axi_master_snoop_response_gen_callback and implement method generate_snoop_response. Program the response in this method. You can provide any valid response. Alternatively, you can access the cache model, and provide response based on the cache line state. You can access the cache model using svt_axi_master_group::get_cache() method. For the methods available for accessing the cache, refer to the Class Reference documentation of class svt_axi_cache.
- Append this extended callback class to the snoop response generator (svt_axi_master_group::snoop_response_gen).

For a demonstration of this mechanism, see ACE example, tb_axi_svt_vmm_ace_sys.

3.7.3 Back Door Access to the Cache

The user can directly access the cache instantiated in the master to read and write information, and to set and get cache line state. The method svt_axi_master_group::get_cache() returns a handle to the cache instantiated in the master. The returned handle is of type svt_axi_cache.

Data can be written into the cache using method svt_axi_cache::write(). The cache state can also be optionally updated using this method. Data and cache line state can be read from the cache using methods svt_axi_cache::read_by_addr() or svt_axi_cache::read_by_index(). Method svt_axi_cache::get_any_index() can be used to obtain index of a cache line within the cache. Methods svt_axi_cache::invalidate_addr(), svt_axi_cache::invalidate_index() and svt_axi_cache::invalidate_all() can be used to invalidate cache line entries within the cache.

Please refer to the class reference documentation of class svt_axi_cache for details of above methods used for accessing the cache.

3.7.4 Support for Barrier Transactions

The AXI VIP supports Barrier protocol as defined in the ACE protocol specification. This support is provided in the axi_master_group component.

3.7.4.1 Barrier Transaction Generation

The AXI VIP master can be enabled to generate barrier transactions by programming the port configuration member svt_axi_port_configuration::barrier_enable to '1'. You can program barrier transactions using member svt_axi_transaction::barrier_type. You should generate the barrier transaction pair on read address and write address channels.

If the delay between barrier transactions belonging to a barrier pair exceeds the value specified by svt_axi_port_configuration::barrier_watchdog_timeout, the VIP master would issue a fatal message.

3.7.4.2 Associating a Normal Transaction with Barrier Transaction

A normal transaction be specified as a post-barrier transaction, and can be associated with a barrier transaction pair. When a normal transaction is associated with a barrier transaction pair, this transaction will be blocked till both read and write transactions belonging to barrier pair complete.

You can specify whether a normal transaction needs to be associated to a barrier transaction pair by setting member svt_axi_transaction::associate_barrier to '1'. If this member is set to '1', the VIP can either automatically associate this transaction with one of the outstanding barrier transaction pairs, or the association can be defined by the user. This can be controlled using member svt_axi_port_configuration::barrier_association_type.

When barrier_association_type is defined as RANDOM_ASSOCIATION, the VIP master automatically associates the normal transaction with one of the outstanding barrier transaction pairs. When barrier_association_type is defined as USER_DEFINED_ASSOCIATION, you need to implement the callback method svt_axi_master_callback::associate_xact_to_barrier_pair (svt_axi_master_transaction xact, svt_axi_barrier_pair_transaction barrier_xact[$]).

The arguments of this callback method are,

- Handle to the normal transaction with which Barrier pair needs to be associated
- List of outstanding barrier transaction pairs, to which the normal transaction can be associated

You can associate the normal transaction to any of the outstanding barrier transaction pairs. This association can be done using member svt_axi_transaction::associated_barrier_xact, which is of class type svt_axi_barrier_pair_transaction. After implementing the callback, you should append the callback class to the master transactor. The above callback method is called under following conditions:

1. svt_axi_port_configuration::barrier_enable is set to '1'
2. svt_axi_transaction::associate_barrier is set to '1'
3. svt_axi_transaction::associated_barrier_xact is null. If svt_axi_transaction::associated_barrier_xact is not null, it implies that user has already associated this transaction with a barrier pair

The barrier pair is represented by an object of type svt_axi_barrier_pair_transaction. This object contains handles to read barrier and write barrier transactions, which are of type svt_axi_master_transaction.

When a normal transaction is associated to the barrier pair as a post barrier transaction, all the other transactions specified after the post barrier transaction would also get blocked, till the post barrier transaction is transmitted.

3.7.4.3 IDs for Barrier Transactions

The specification requires that Barrier transactions use different ID values than are in use for non-barrier transactions. To support this, the VIP allows you to specify a range of ID values which can be used for Barrier transactions. The VIP will then validate that the ID specified for Barrier transactions is within this range, and ID specified for normal transactions is outside this range. An error message is issued if this condition is not met. This ensures that Barrier transactions and normal transactions use a mutually exclusive set of IDs.

The range of ID values which can be used for Barrier transactions are specified using members svt_axi_port_configuration::barrier_id_min and svt_axi_port_configuration::barrier_id_max. The non-barrier transactions should use the ID values outside this range.

3.7.4.4 Outstanding Barrier Transactions

Member svt_axi_port_configuration::num_outstanding_xact specifies the total number of outstanding transactions. Barrier transactions are considered as part of this total number of outstanding transactions. That is, total of normal outstanding transactions and barrier outstanding transactions will not exceed svt_axi_port_configuration::num_outstanding_xact.

3.7.5 Support for DVM Transactions

The AXI VIP supports DVM protocol as defined in the ACE protocol specification. This support is provided in the axi_master_group component. The AXI VIP supports DVM Operations, DVM Sync and DVM Complete transactions.

3.7.5.1 DVM Transaction Generation

The AXI VIP master can be enabled to generate DVM transactions by programming the port configuration member svt_axi_port_configuration::dvm_enable to '1'. You can program DVM transactions programming member svt_axi_transaction::coherent_xact_type to DVMMESSAGE or DVMCOMPLETE. You might need to program the values of field svt_axi_transaction::addr to specify the message type. The values specified in this field would get driven on the ARADDR signals. The svt_axi_master_transaction has pre-defined constraints to constrain the values of other fields, when transaction is of type DVM, as required by the ACE protocol specification.

3.7.5.2 Generation of DVM Sync

Behavior of DVM Sync generation is as described below based on ACE protocol specification:

1. When you generate a DVM Sync using master VIP, the master VIP would transmit it only if there are preceding DVM Operations. If a DVM Sync is generated right after another DVM Sync, then the second DVM Sync would be dropped.
2. If you attempt to transmit a new DVM Sync, without having received DVM Complete for previous DVM Sync, the new DVM Sync will be stalled till DVM Complete for previous DVM Sync is received.

3.7.5.3 Generation of DVM Complete

When the Master VIP receives a DVM Sync transaction, it generates a DVM Complete transaction. The generation of DVM Complete transaction is controlled using svt_axi_port_configuration::dvm_complete_generator_type member.

If the value of svt_axi_port_configuration::dvm_complete_generator_type is set to AUTO_DVM_COMPLETE_GEN, the Master VIP automatically generates a DVM Complete transaction, after receiving a DVM Sync transaction. The DVM Complete transaction contains a member of type svt_axi_transaction::dvm_complete_delay. The delay between reception of DVM Sync and transmission of DVM Complete is determined by randomizing this member.

When the value of svt_axi_port_configuration::dvm_complete_generator_type is set to USER_DVM_COMPLETE_GEN, you need to implement callback method svt_axi_master_callback::dvm_complete_gen (svt_axi_master_transaction dvm_complete_xact). You can program the delay member svt_axi_transaction::dvm_complete_delay in the DVM Complete transaction, in the callback method. After implementing the callback, you should append the callback class to the master transactor.

You can also generate the DVM Complete transaction using scenario generator. But the master VIP would ensure that number of transmitted DVM Complete transactions would never exceed number of received DVM Sync transactions. If number of transmitted DVM Complete transactions is equal to number of

received DVM Sync transactions, any further attempts to transmit DVM Complete transactions would result in the DVM Complete transaction getting dropped.

3.7.5.4 IDs for DVM Transactions

The specification requires that DVM transactions use different ID values than are in use for non-DVM transactions. To support this, the VIP allows you to specify a range of ID values which can be used for DVM transactions. The VIP will then validate that the ID specified for DVM transactions is within this range, and ID specified for normal transactions is outside this range. An error message is issued if this condition is not met. This ensures that DVM transactions and normal transactions use a mutually exclusive set of IDs.

The range of ID values which can be used for DVM transactions are specified using members svt_axi_port_configuration::dvm_id_min and svt_axi_port_configuration::dvm_id_max. The non-DVM transactions should use the ID values outside this range.

3.7.5.5 Multi-Part DVM

VIP supports multi-part DVM operation both in active and passive mode. In passive mode, it detects multi-part DVM operation and performs checking associated protocol rules. In active mode, once VIP receives transactions generated from sequence, it first tries to identify multi-part DVM operation and checks all associated rules. If it detects another transaction between the two parts of the multi-part DVM message, then the master VIP drops the transaction and issues an ERROR message. Dropped transaction is not driven on the bus. The transaction which is dropped in such a manner is still written to the analysis port, with the bit `svt_axi_transaction::is_coherent_xact_dropped` set to 1. This signifies that the transaction was dropped by the master VIP and was not actually driven on the bus.

Built-in sequence `svt_axi_ace_master_multipart_dvm_virtual_sequence` is provided as part of the VIP to generate Multi-Part DVM scenario from master VIP.

3.7.6 Support for ACE-Lite

ACE-Lite mode is enabled when svt_axi_port_configuration::axi_interface_type is set to ACE_LITE. In ACE-Lite mode, the Master VIP transmits only following type of coherent transactions, as per the ACE protocol specification:

- ReadNoSnoop
- ReadOnce
- CleanShared
- CleanInvalid
- MakeInvalid
- WriteNoSnoop
- WriteUnique
- WriteLineUnique
- Barrier

If you try to send any other coherent transaction type which is not valid in ACE-Lite mode, the Master VIP will issue a fatal message. The svt_axi_master_transaction class contains constraints such that when the master transaction is randomized, only valid coherent transaction types will be generated in ACE-Lite mode.

In ACE-Lite mode, the un-used signals are driven to Z.

3.7.7 Support for ACE Domain

You can program the domain using field svt_axi_transaction::domain_type. If you program a domain type which violates the combination with AxCACHE, AxSNOOP and AxBAR signals, the master VIP would issue a fatal message. The svt_axi_master_transaction class contains constraints such that when the master transaction is randomized, only valid combinations of AxDOMAIN, AxCACHE, AxSNOOP and AxBAR signals are generated.

3.7.8 Support for Speculative Read

Support for speculative read feature can be enabled by programming the port configuration member svt_axi_port_configuration::speculative_read_enable to 1.

A speculative read is defined as a read of a cache line that a master may already be holding in its cache. Consider a case when user generates a coherent transaction for read of a cache line that a master already holds in its cache.

When speculative read is enabled, the master will transmit this coherent transaction. Also, the master will support cache state transitions as defined in second set of tables in chapter C4 of ACE protocol specification.

When speculative read is disabled, the master will not transmit this coherent transaction. In this case, the master will only support cache state transitions as defined in first set of tables in chapter C4 of ACE protocol specification.

Whether a transaction is a speculative read is indicated by read-only transaction class member svt_axi_transaction::is_speculative_read.

3.7.9 Support for Snoop Filtering

Support for snoop filtering can be enabled by programming the port configuration member svt_axi_port_configuration::snoop_filter_enable to 1. When snoop filtering is enabled in the master VIP, cache state transitions to “Legal End State (with snoop filter)” as specified in tables in chapter C4 of ACE protocol specification, are allowed. These cache state transitions can be done using transaction class member svt_axi_transaction::force_to_shared_state, and by setting svt_axi_port_configuration::cache_line_state_change_type to LEGAL_WITH_SNOOP_FILTER_CACHE_LINE_STATE_CHANGE.

When snoop filtering is enabled, Master VIP would automatically generate evict transactions, when Master VIP removes a cache line to make space to allocate a new cache line.

3.7.10 Cache State Transitions to Legal End States

Tables in chapter C4 of ACE protocol specification specify a set of legal end states (with snoop filter and without snoop filter), in addition to the expected end states. The Master VIP can support cache state transitions to these legal end states using below members:

- svt_axi_port_configuration::cache_line_state_change_type
- svt_axi_transaction::force_to_shared_state
- svt_axi_transaction::force_to_invalid_state

For details of these members, refer to AXI VIP Class reference HTML documentation.

3.7.11 Support for ACE Exclusive Access

The ACE Master model supports the ACE Exclusive access. The master VIP implements a master exclusive monitor. This exclusive monitor is used to monitor the address location used by an Exclusive sequence. This master exclusive monitor is used to determine if another master could have performed a store to the address location during the Exclusive sequence, by monitoring the snoop transaction it receives.

When the master performs an Exclusive Load, the master exclusive monitor is set. The master exclusive monitor is reset when a snoop transaction is observed that indicates another master could perform a store to the location.

Only one outstanding exclusive transaction is allowed, that is, while an exclusive transaction is in progress, any new exclusive transaction will be blocked for execution by master until the completion of the outstanding exclusive transaction.

3.7.11.1 Exclusive Load

This section explains the behavior of the ACE Master VIP with respect to the exclusive load.

If a cache line is either in unique or shared state, then master will not initiate the exclusive load transaction on to the interface. This is based on the below description provided in Chapter "Exclusive Accesses" of the ACE protocol specification.

- If the master holds a copy of the line in a Unique state, then issuing a transaction for the Exclusive Load is permitted, but not recommended.
- If the master holds a copy of the line in a Shared state then issuing a transaction for the Exclusive Load is permitted, but not required

3.7.11.2 Exclusive Store

This section explains the behavior of the ACE Master VIP with respect to the exclusive store.

1. Exclusive store transaction will not be allowed without a prior exclusive load transaction, hence it will be dropped. All dropped exclusive store transactions will be tagged with svt_axi_transaction::is_coherent_xact_dropped set to '1'.
2. If the master receives OKAY response to an exclusive store transaction, based on the status of the master exclusive monitor, the following actions will be taken:
   - If the master exclusive monitor is reset, then the store will fail, that is, cache will not be updated. You need to restart the whole exclusive sequence again.
   - If the master exclusive monitor is set, then the exclusive store will fail, that is, cache will not be updated. You can re-initiate the exclusive store transaction alone, or restart the complete exclusive sequence.

The status of the master exclusive monitor is represented by member svt_axi_transaction::excl_mon_status. The status of exclusive access is represented by member svt_axi_transaction::excl_access_status.

The following table shows how to interpret the various combinations of svt_axi_transaction::excl_mon_status and svt_axi_transaction::excl_access_status, to derive the meaning of failure of an exclusive store associated with the corresponding exclusive store transaction.

Table 3-16

excl_access_status	excl_mon_status	Reason for exclusive store failure	Action needed
EXCL_ACCESS_FAIL	EXCL_MON_INVALID	Exclusive store transaction generated prior to an exclusive load transaction and hence the exclusive store transaction will be dropped.	You need to start the exclusive sequence with Exclusive load.
EXCL_ACCESS_FAIL	EXCL_MON_RESET	While initiating the exclusive store transaction, if the exclusive monitor is reset, the master will drop the exclusive store transaction. This will be indicated by setting svt_axi_transaction::is_coherent_xact_dropped to '1'. In this case, exclusive store fails. While initiating the exclusive store transaction, if the exclusive monitor is set, the master will initiate the exclusive store transaction on the bus. This will be indicated by setting svt_axi_transaction::is_coherent_xact_dropped to '0'. Between the point that the Exclusive Store transaction was issued and the point that it completed, a non-Exclusive Store can reset the exclusive monitor. In such a case, irrespective of the response (EXOKAY/OKAY), exclusive store fails.	You need to restart the complete exclusive sequence. You need to restart the complete exclusive sequence.
EXCL_ACCESS_FAIL	EXCL_MON_SET	The exclusive store transaction is initiated on to the bus, the response received is OKAY, and master exclusive monitor is set. In this case, exclusive store fails, as response indicates exclusive access fail.	Exclusive store transaction alone can be re-initiated, or complete exclusive sequence can be initiated by you.

3.7.12 Known Limitations

- Requirement for WriteUnique and WriteLineUnique transactions specified in ACE protocol specification are not supported. Note that these transactions are typically used by non-cached components, and the restrictions given apply to cacheable components using it (which is a typical case).
- The specification allows a cache line state to transition to the UniqueClean state after it completes. In other words an allocation is made for the transaction. This is currently not supported for READNOSNOOP because a READNOSNOOP need not necessarily be of cache line size, and the allocation may span multiple cache lines.

3.8 Support for ACE-Lite Protocol in AXI Slave

The Slave can be configured to work in ACE-Lite mode. This enables the Slave to receive and respond to barrier and cache maintenance transactions which may get forwarded to it by the interconnect.

ACE-Lite mode is enabled when svt_axi_port_configuration::axi_interface_type is set to ACE_LITE.

3.9 AXI4 Region and Address Range Support in Slave

3.9.1 Slave Address Range Support

System level address map can be specified using system configuration method svt_axi_system_configuration::set_addr_range. The start address and end address can be specified for each slave in the system. Multiple address ranges can also be specified for a single slave. These address ranges can be specified as a continuous or a discontinuous.

3.9.2 Slave Region Support

The specified address ranges can be divided into maximum of 16 regions as guided by the AxREGION[3:0] signal of the AXI4 interface. These regions can be specified as continuous or discontinuous. The regions can be specified using method svt_axi_slave_addr_range::set_region_range. Every region is marked by a region-id ranging from 0-15.

The region can be associated to a region type, which defines the characteristics of the specified region. The region type can be specified as one of the arguments of the method svt_axi_slave_addr_range::set_region_range.

The following region types are currently supported:

- R- Read Only
- W - Write Only
- RW - Read/Write
- RSVD - Reserved

Table 3-17 depicts the slave response generated for a read transaction, based on the region attribute:

Table 3-17 Slave Response for Read Transaction

Type Attribute	Resultant Response
R- Read Only	OKAY
W - Write Only	SLV_ERR
RW - Read/Write	OKAY
RSVD - Reserved	SLV_ERR

Table 3-18 depicts the slave response generated for a write transaction, based on the region attribute:

Table 3-18 Slave Response Generated for Write Transaction

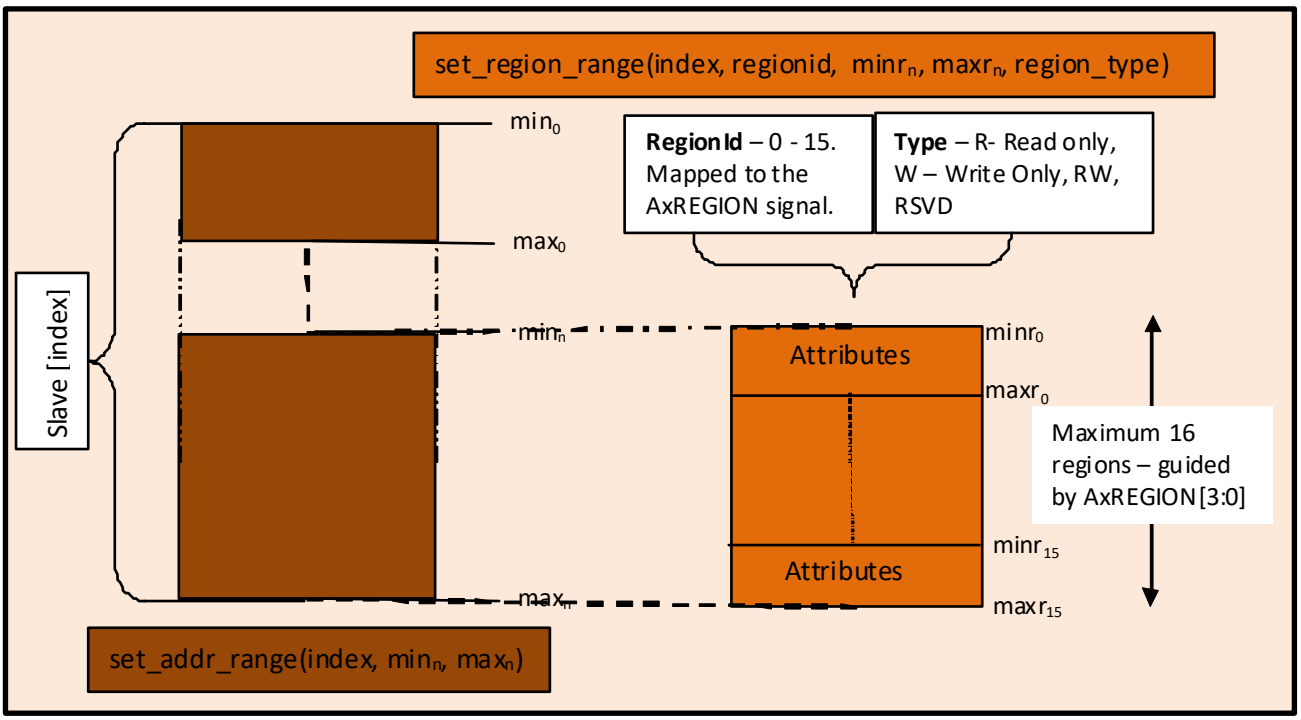
Type Attribute	Resultant Response
R- Read Only	SLV_ERR

Table 3-18 Slave Response Generated for Write Transaction

Type Attribute	Resultant Response
W - Write Only	OKAY
RW - Read/Write	OKAY
RSVD - Reserved	SLV_ERR

Figure 3-4 pictorially shows the range and regions described above.

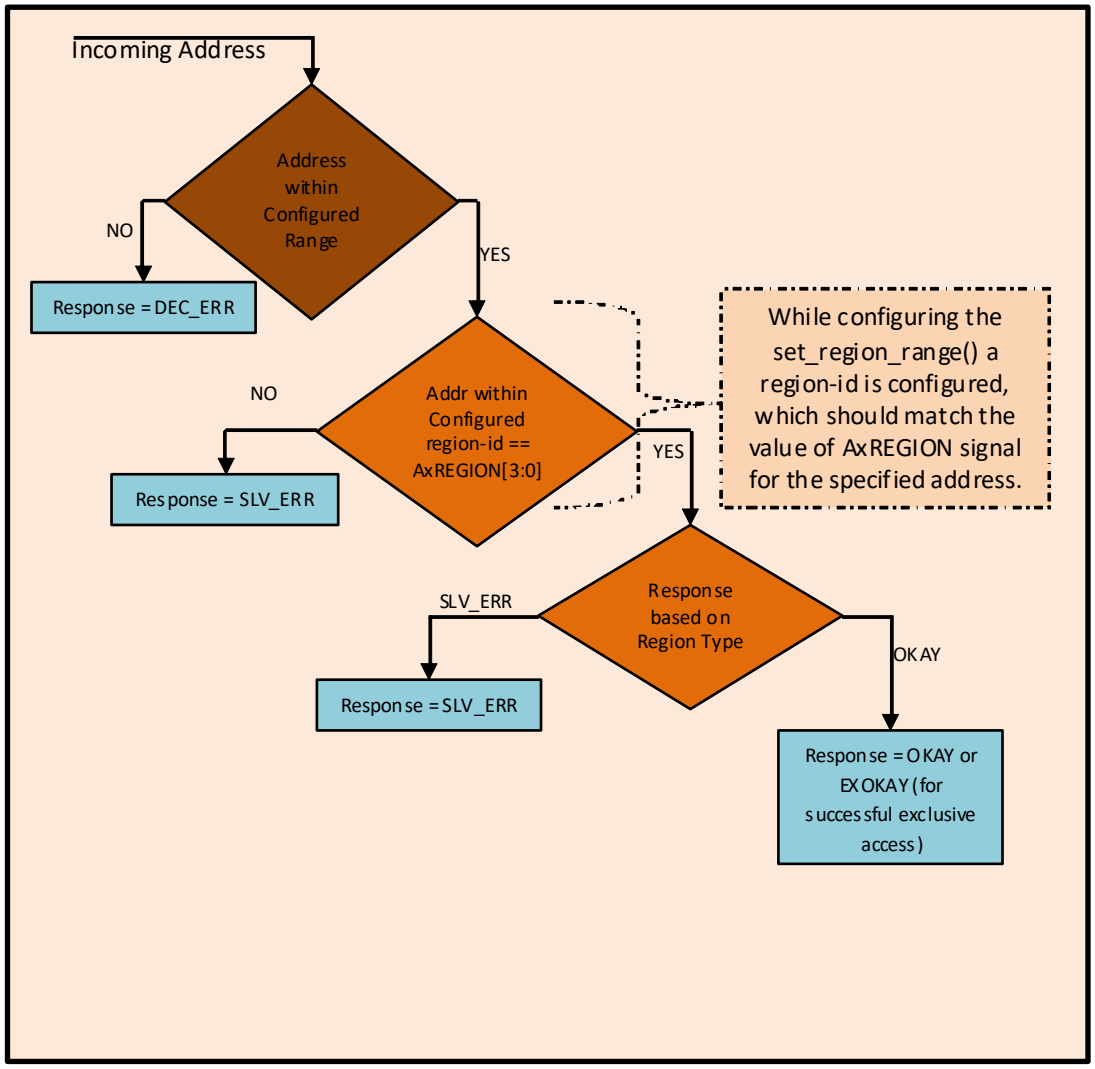
Figure 3-4



3.9.3 Slave Response Generation

For each received transaction, the port monitor within the slave group generates the appropriate response based on address range and region type. This response is populated in the svt_axi_transaction::rresp[] (for read transactions) or svt_axi_transaction::bresp (for write transactions) fields of the slave transaction object. This slave transaction is then provided to the response generator within the slave group. If the slave response generator modifies the pre-populated response in slave transaction, the response may no longer be correct.

Figure 3-5 Flow Diagram for Slave Response Generation:



4 Support for ACE5

This chapter describes the support for ACE5 protocol. This chapter discusses the following topics:

- Overview of ACE5
- Features Supported for ACE5
- Unsupported Features

4.1 Overview of ACE5

The support for ACE5 protocol is based on the ARM IHI 0022F.b (ID122117) specification.

You can enable the support for ACE5 VIP by setting the port configuration parameter `svt_axi_port_configuration::ace_version` to ACE_VERSION_2_0. You must also define the compile time macro `SVT_ACE5_ENABLE` before enabling the support for ACE5.

The ACE5 features are in the early adopter stage and hence the user interface can undergo minor changes.

For details on the transaction members, configuration members and protocol checks, refer to the AXI Class Reference at the following location:
`$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_svt_uvm_class_reference/html/index.html`

4.2 Features Supported for ACE5

The following sections describe the features supported by ACE5 protocol.

4.2.1 WAKEUP SIGNALLING Feature

4.2.1.1 ACWAKEUP

The ACWAKEUP feature is enabled by configuring the `svt_axi_port_configuration` parameter to `acwakeup_enable`.

This enables the ACWAKEUP sideband signal in the VIP when this bit is set to '1'. By default ACWAKEUP signal is '0' when this bit is enabled. You can toggle the ACWAKEUP signal by controlling the following configuration class members:

- svt_axi_port_configuration::acwakeup_toggle_min_delay_during_idle
- svt_axi_port_configuration::acwakeup_toggle_max_delay_during_idle

The following transaction class members are added for this feature:

- svt_axi_snoop_transaction::acwakeup_assert_delay
- svt_axi_snoop_transaction:: acwakeup_deassert_delay
- svt_axi_snoop_transaction:: assert_acwakeup_after_acvalid

4.2.1.2 AWAKEUP

The AWAKEUP feature is enabled by configuring the `svt_axi_port_configuration` parameter to `awakeup_enable`.

This enables AWAKEUP sideband signal in the VIP when this bit is set to '1'. By default AWAKEUP signal is '0' when this bit is enabled. You can toggle the AWAKEUP signal by controlling the following configuration class members:

- svt_axi_port_configuration::awakeup_toggle_min_delay_during_idle
- svt_axi_port_configuration::awakeup_toggle_max_delay_during_idle

The following transaction class members are added for this feature:

- svt_axi_transaction:: awakeup_deassert_delay
- svt_axi_ transaction::assert_awakeup_after_arvalid
- svt_axi_ transaction::assert_awakeup_after_awvalid

4.2.2 CACHE STASHING Feature

The following are the new transaction types or OPCODEs that are added for this feature:

- WRITEUNIQUEPTLSTASH
- WRITEUNIQUEFULLSTASH
- STASHONCESHARED
- STASHONCEUNIQUE

These are added under the `svt_axi_transaction::coherent_xact_type_enum`.

The following are the configuration class members added for this feature:

- svt_axi_port_configuration::cache_stashing_enable

The following transaction class members are added for this feature:

- svt_axi_transaction:: stash_nid
- svt_axi_transaction:: stash_nid_valid
- svt_axi_transaction:: stash_lpid
- svt_axi_transaction:: stash_lpid_valid

Note
The STASHTRANSLATION transaction type is not yet supported for this feature.

4.2.3 UNTRANSLATED Feature

The following configuration class members are added for this feature:

- svt_axi_port_configuration::addr_translation_enable = 0;

The following transaction class members are added for this feature:

- svt_axi_transaction:: stream_id = 0;
- svt_axi_transaction::secure_or_non_secure_stream = 0;
- svt_axi_transaction::sub_stream_id = 0;
- svt_axi_transaction::sub_stream_id_valid = 0;
- svt_axi_transaction::addr_translated_from_pcie = 0;

4.2.4 DATACHECK Feature

The following configuration class members are added for this feature:

- svt_axi_port_configuration::check_type_enum check_type

The following transaction class members are added for this feature:

- svt_axi_transaction::datachk_parity_value[]
- svt_axi_transaction::is_datachk_passed[]
- svt_axi_transaction::is_datachk_parity_error

The checks are enabled only when the corresponding port configuration is enabled.

```
svt_axi_port_configuration.check_type ==
svt_axi_port_configuration::ODD_PARITY_BYTE_DATA;
```

- If parity error is detected in write data by Active Slave VIP, then it asserts the parity error check `wdatachk_parity_calculated_wdata_parity_check`.
- If parity error is detected in read data by Active Master VIP, then it asserts the parity error check `rdatachk_parity_calculated_rdata_parity_check`.
- If parity error is detected in snoop data by Interconnect VIP, then it asserts the parity error check `cddatachk_parity_calculated_cddata_parity_check`.

In the passive mode, if any parity error is detected, then passive monitor asserts the parity error check.

4.2.5 POISON Feature

The following configuration class members are added for this feature:

- svt_axi_port_configuration::poison_enable

The following transaction class members are added for this feature:

- svt_axi_transaction::poison
- svt_axi_snoop_transaction::snoop_poison

4.2.6 Trace Tag Feature

The following transaction class members are added for this feature:

- svt_axi_transaction::trace_tag =0;

- svt_axi_transaction::data_trace_tag =0;
- svt_axi_transaction::resp_trace_tag =0;
- svt_axi_snoop_transaction::trace_tag = 0;
- svt_axi_snoop_transaction::snoop_data_trace_tag = 0;
- svt_axi_snoop_transaction::snoop_resp_trace_tag =0;

4.3 Unsupported Features

The following ACE5 features are not supported:

- Deallocating Transactions
- Atomic Transactions
- User Loop back signaling
- QoS Accept signaling
- Coherency Connection signaling
- Non-secure access identifiers
- CLEANSHAREDPERSIST feature is not supported.
- AXI System monitor is not supported for ACE5 features.
- Functional coverage is not supported for ACE5 features.

5 Verification Features

This chapter describes the various verification features available along with VC Verification IP. This chapter discusses the following topics:

- "ACE Scenarios"
- Verification Planner
- Protocol Analyzer Support
- Error Injection

5.1 ACE Scenarios

The AXI VIP provides a collection of ACE master scenarios. These sequences can be executed on multi-stream scenario generator within the AXI System group, to generate different types of ACE scenarios.

Refer to the test "tests/ts.ace_single_port_readclean_test.sv" present in tb_axi_svt_vmm_ace_sys example, for details on usage of these scenarios. Table 5-1 lists the ACE scenarios.

Table 5-1 ACE Scenarios

S.No.	Scenario Name	Description
1.	svt_axi_ace_master_makeunique_scenario	This scenario initiates MakeUnique transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each Makeunique transaction, cachelines of peer masters are initialized to random, valid states.
2.	svt_axi_ace_master_readshared_scenario	This scenario initiates ReadShared transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each ReadShared transaction, cachelines of peer masters are initialized to random, valid states.
3.	svt_axi_ace_master_readclean_scenario	This scenario initiates ReadClean transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each ReadClean transaction, cachelines of peer masters are initialized to random, valid states.
4.	svt_axi_ace_master_readnosnoop_scenario	This scenario initiates ReadNoSnoop transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user.

Table 5-1 ACE Scenarios (Continued)

S.No.	Scenario Name	Description
5.	svt_axi_ace_master_readonce_scenario	This scenario initiates ReadOnce transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each ReadOnce transaction, cachelines of peer masters are initialized to random, valid states.
6.	svt_axi_ace_master_readnotshareddirty_scenario	This scenario initiates ReadNotSharedDirty transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each ReadNotSharedDirty transaction, cachelines of peer masters are initialized to random, valid states.
7.	svt_axi_ace_master_readunique_scenario	This sequence initiates ReadUnique transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each ReadUnique transaction, cachelines of peer masters are initialized to random, valid states.
8.	svt_axi_ace_master_cleanunique_scenario	This scenario initiates CleanUnique transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each CleanUnique transaction, cachelines of peer masters are initialized to random, valid states.
9.	svt_axi_ace_master_cleanshared_scenario	This scenario initiates CleanShared transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each CleanShared transaction, cachelines of peer masters are initialized to random, valid states.
10.	svt_axi_ace_master_cleaninvalid_scenario	This scenario initiates CleanInvalid transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each CleanInvalid transaction, cachelines of peer masters are initialized to random, valid states.
11.	svt_axi_ace_master_makeinvalid_scenario	This scenario initiates MakeInvalid transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each MakeInvalid transaction, cachelines of peer masters are initialized to random, valid states.
12.	svt_axi_ace_master_writenosnoop_scenario	This scenario initiates WriteNoSnoop transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user.
13.	svt_axi_ace_master_writeunique_scenario	This scenario initiates WriteUnique transaction from the ACE/ACE_LITE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each WriteUnique transaction, cachelines of peer masters are initialized to random, valid states.

Table 5-1 ACE Scenarios (Continued)

S.No.	Scenario Name	Description
14.	svt_axi_ace_master_writeback_scenario	This scenario initiates WriteBack transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each WriteBack transaction, cachelines of peer masters are initialized to random, valid states.
15.	svt_axi_ace_master_writeclean_scenario	This scenario initiates WriteClean transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each WriteClean transaction, cachelines of peer masters are initialized to random, valid states.
16.	svt_axi_basic_writeback_full_cacheline	This scenario generates a writeback transaction for a full cacheline.
17.	svt_axi_basic_writeclean_full_cacheline	This scenario generates a writeclean transaction for a full cacheline.
18.	svt_axi_ace_master_evict_scenario	This scenario initiates Evict transaction from the ACE master specified with port_id, which can be a random port or a specific port configured by the user. Before sending each Evict transaction, cachelines of peer masters are initialized to random, valid states.
19.	svt_axi_ace_master_exclusive_access_virtual_scenario	Basic Exclusive access scenario. This scenario provides ACE Exclusive access at system level and can be used in any AXI_ACE master port to initiate Exclusive access transaction sequence. Transaction sequence Used: Exclusive Load followed by Exclusive store • Initialize cache lines if svt_axi_ace_master_base_scenario::initialize_cachelines bit is set. • Issue READCLEAN or READSHARED to load location and wait for the transaction to end. • Check the cache line state - If in Shared state, issue CLEANUNIQUE - If in Invalid state, then restart Exclusive Access - Else do nothing as Master can store directly to the cacheline and there is no need to inform Interconnect • Stored data is updated to memory through WRITEBACK transaction Note that for generation of exclusive access transactions, svt_axi_port_configuration :: exclusive_access_enable should be explicitly set by user for the targeted master.

Table 5-1 ACE Scenarios (Continued)

S.No.	Scenario Name	Description
20.	svt_axi_ace_master_snoop_during_memory_update_scenario	This scenario attempts to create a scenario where an initiating master receives a snoop while transmitting a WRITEBACK. The scenario first sends MAKEUNIQUE from the ACE master specified through svt_axi_ace_master_two_port_base_virtual_scenario ::first_port_id to get cachelines to UniqueDirty state. It then initiates WRITEBACK from the same master specified through svt_axi_ace_master_two_port_base_virtual_scenario ::first_port_id. At the same time, it initiates random coherent transaction from an ACE/ACE_LITE master specified through the svt_axi_ace_master_two_port_base_virtual_scenario ::second_port_id. This will result in snoop transaction at ACE master specified by svt_axi_ace_master_two_port_base_virtual_scenario ::first_port_id, while it is trying to update memory.
21.	svt_axi_ace_master_overlapping_addr_scenario	This sequence attempts to create a scenario where random coherent transactions targeting the same address are initiated from two different masters in which one is an ACE master specified with first_port_id and another one is an ACE/ACE_LITE master specified through second_port_id.

Table 5-1 ACE Scenarios (Continued)

S.No.	Scenario Name	Description
22.	svt_axi_ace_master_shareable_store_barrier_load_scenario	This scenario does the following: Sends a number of pre barrier store transactions based on num_pre_barrier_stores. Sends a barrier pair, that is, write barrier and read barrier. Sends a post barrier flag transaction. Any master that can observe this flag should be able to observe the transactions before the barrier. From another port in the same domain, the location written through the flag transaction is continuously read (load). When the value set through the flag transaction is read back, the loop terminates. The flag transaction is a post-barrier transaction, so if its value is observable, it then reads back all the locations written through the pre barrier store transactions and checks that all the data that was written is read back correctly. Thus, this scenario is self-checking. Note that this step is not done if pre_barrier_xact_type is PRE_BARRIER_CACHE_MAINTENANCE since the data is not available in cache maintenance transactions. Instead, the scenario checks that when a post-barrier transaction completes, all pre-barrier cache maintenance transactions should have completed. The ports on which the pre barrier stores and the loads are sent are randomly chosen based on configuration. The type of store transaction is based on the setting in pre_barrier_xact_type. Loads can be READSHARED, READONCE, READCLEAN or READNOTSHAREDDIRTY. Some interesting scenarios that can be exercised using this are: (Each of these scenarios is repeated for sequence_length) 1. num_pre_barrier_stores=1,num_observers=1: A single pre-barrier store followed by a post barrier flag is sent. One observer reads the post barrier flag and when the flag is observed as set, it reads the location addressed by pre_barrier store. 2. num_pre_barrier_stores=1,num_observers>1: A single pre-barrier store followed by a post barrier flag is set. Many observers reading the post barrier flag and when the flag is observed as set, they read the location addressed by pre_barrier store. 3. num_pre_barrier_stores>1,num_observers=1: Many pre-barrier stores followed by a post barrier flag is set. One observer reading the post barrier flag and when the flags is set, it reads the locations addressed by pre_barrier store transactions. 4. num_pre_barrier_stores>1, num_observers>1: Many pre-barrier stores followed by a post barrier flag is set. Many observers reading the post barrier flag and when the flag is set, they read the locations addressed by pre_barrier store transactions.

Table 5-1 ACE Scenarios (Continued)

S.No.	Scenario Name	Description
	svt_axi_ace_master_nonshareable_store_barrier_load_scenario	This scenario does the following: • Sends a number of pre barrier write transactions based on num_pre_barrier_stores. • Sends a barrier pair. • Sends post barrier read transaction to the same address. • Since the reads are post barrier transactions, all the previous writes should be observable to the reads. • All write transactions send are WRITENOSNOOP transaction and read transactions are READNOSNOOP transactions.
	svt_axi_ace_master_load_barrier_scenario	This scenario does the following: • Send a number of transactions to load. The number of transactions sent is based on num_pre_barrier_loads. • Send a barrier pair • Send a post barrier transaction that is associated to the barrier pair. This transaction will be send out only after the response to the barrier pair is received. • This is a self-checking scenario. When the post barrier transaction ends, it checks that all pre barrier transactions have also ended.

5.2 Verification Planner

AXI VIP provides verification plans which can be used for tracking verification progress of AXI3/4 and ACE protocols. A set of top-level plans and sub-plans are provided. The verification plans are available at:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/VerificationPlans

For more information, refer to the README file, which is available at:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/VerificationPlans/README

5.3 Protocol Analyzer Support

AXI VIP supports Synopsys® Protocol Analyzer. Protocol Analyzer is an interactive graphical application which provides protocol-oriented analysis and debugging capabilities.

For the AXI SVT VIP, protocol file generation is enabled or disabled through the variable "enable_xml_gen" that is defined in the class "svt_axi_port_configuration". The default value of this variable is "0", which means that protocol file generation is disabled by default.

To enable protocol file generation, set the value of the variable "enable_xml_gen" to '1' in the port configuration of each master or slave for which protocol file generation is desired. The next time that the environment is simulated, protocol files will be generated according to the port configurations. The protocol files will be in .xml format. Import these files into the Protocol Analyzer to view the protocol transactions.

For Verdi documentation, see `$VERDI_HOME/doc/Verdi_Transaction_and_Protocol_Debug.pdf`.

Protocol Analyzer has been enhanced to read FSDB transactions and Verdi can load the FSDB transactions into Browser.

5.3.1 Support for VC Auto Testbench

AXI VIP supports VC AutoTestbench which generates SV UVM testbench for Block level or Sub-System or System Level Design.

For VC ATB documentation, see Verdi_Transaction_and_Protocol_Debug.pdf.

5.3.2 Performance Analyzer

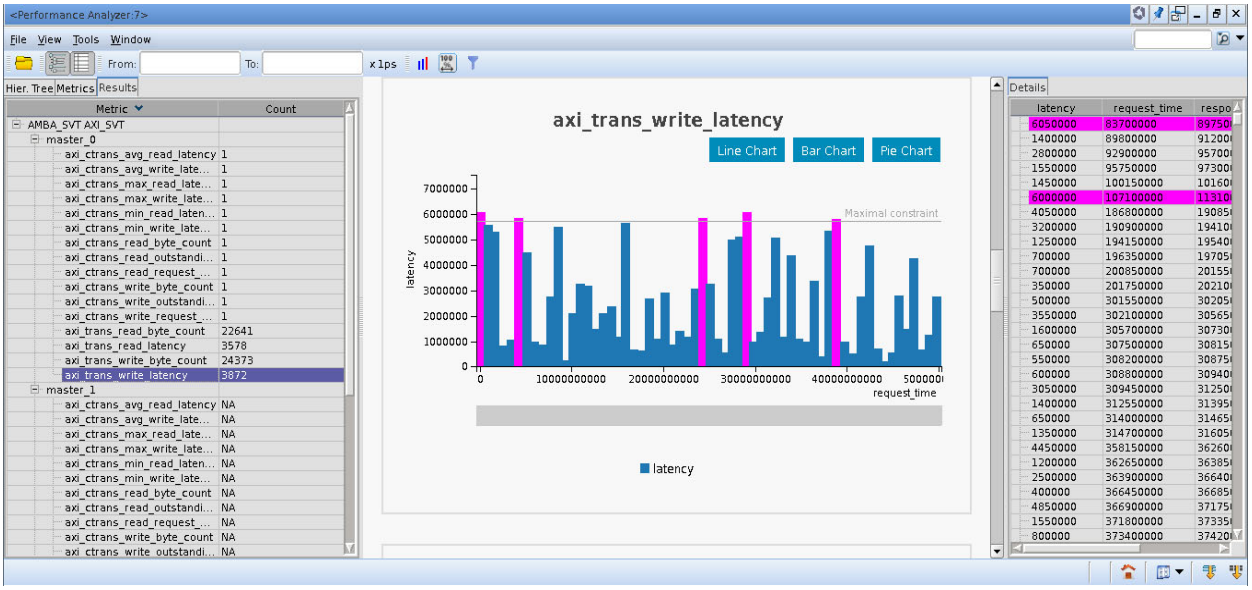
The performance analyzer tool is used to calculate the performance of sub-systems. For more information on FSDB Dumping, see Support for Native Dumping of FSDB.

5.3.2.1 Invoking Verdi GUI After Running the Simulation

1. Invoking Verdi GUI after running the simulation:

```
verdi -lca -ssf test_top.fsdb
```

Figure 5-1 Final View Of Performance Metric With Graph and Data Details



Note For more information, see Verdi_Performance_Analyzer.pdf.

5.3.3 Metrics Description

Performance metrics are used to measure the performance of sub-systems. Each AMBA VIP has three types of performance metrics as follows:

- Transaction metrics
- Cross transaction metrics
- Cross instance metrics

5.3.3.1 Transaction Type Metrics

These metrics are used to measure the performance of any given transaction. The following metrics comes under this type of metrics:

- *_trans_read_latency
- *_trans_write_latency
- *_trans_read_byte_count
- *_trans_write_byte_count

5.3.3.2 Cross Transaction Type

These metrics are used to measure the performance across transactions at a given port. The following metrics comes under this type of metrics:

- *_ctrans_min_read_latency
- *_ctrans_min_write_latency
- *_ctrans_max_read_latency
- *_ctrans_max_write_latency
- *_ctrans_avg_read_latency
- *_ctrans_avg_write_latency
- *_ctrans_read_request_count
- *_ctrans_write_request_count
- *_ctrans_read_outstanding_count
- *_ctrans_write_outstanding_count
- *_ctrans_read_byte_count
- *_ctrans_write_byte_count

5.3.3.3 Cross Instance Type

These metrics are used to measure the performance of the transactions across all ports. The following metrics comes under this type of metrics:

- *_cinst_read_request_count
- *_cinst_write_request_count
- *_cinst_read_request_percentage
- *_cinst_write_request_percentage
- *_cinst_read_bus_bandwidth
- *_cinst_read_bus_bandwidth
- *_cinst_read_byte_count
- *_cinst_write_byte_count

Note * indicates the protocol name.(for example, for AXI *_trans_read_latency will be axi_trans_read_latency)

For more information, see *$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_performance_metrics.pdf*.

5.3.4 Support for Native Dumping of FSDB

Native FSDB supported in AXI VIP.

- **FSBD Generation**: Protocol Analyzer uses transaction-level dump database. You can use the following settings to dump the transaction database:
  - Compile Time Options:
    - `+define+SVT_AXI_ACE_SNPS_INTERNAL_SYSTEM_MONITOR_USE_MASTER_SLAVE_AGENT_CONNECTION`. Required for master-slave latency metrics.
    - -lca -kdb // dumps the work.lib++ data for source coding view
    - +define+SVT_FSDB_ENABLE // enables FSDB dumping
    - -debug_access

  For more information on how to set the FSDB dumping libraries, see Appendix B section in *Linking Novas Files with Simulators and Enabling FSDB Dumping* guide available at: $VERDI_HOME/doc/linking_dumping.pdf.

  - New configuration parameter added for XML/FSDB generation in svt_axi_port_configuartion.sv. It is a port configuration variable. Add the following setting in system configuration to enable the generation of XML/FSDB:

```
 /** Enable protocol file generation for Protocol Analyzer */
    this.master_cfg[0].enable_xml_gen = 1;
    this.slave_cfg[0].enable_xml_gen = 1;
    this.master_cfg[0].pa_format_type = svt_xml_writer:: ::<XML/FSDB/BOTH>;
    this.slave_cfg[0].pa_format_type= svt_xml_writer:: ::<XML/FSDB/BOTH>;
// 0 is XML, 1 FSDB and 2 both XML and FSDB, default it will be zero
```

  - New configuration parameter added for XML/FSDB generation in `svt_axi_system_configuartion` class. These are system configuration variables. Add the following setting in system configuration to enable the generation of XML/FSDB from system monitor. These are required for master-slave latency metrics:

```
/** Enable protocol file generation for Protocol Analyzer */
this.enable_xml_gen = 1;
this.pa_format_type = svt_xml_writer:: ::<XML/FSDB/BOTH>; // 0 is XML, 1 FSDB
and 2 both XML and FSDB, default it will be zero
```

- **Invoking Protocol Analyzer:** Perform the following steps to invoke Protocol Analyzer in interactive or post-processing mode:
  - Post-processing Mode
    - Load the transaction dump data and issue the following command to invoke the GUI:
      `verdi -ssf <dump.fsdb> -lib work.lib++`
    - In Verdi, navigate to Tools > Transaction Debug >Transaction and Protocol Analyzer.
  - Interactive Mode
    - Issue the following command to invoke Protocol Analyzer in an interactive mode:
      `<simv> -gui=verdi`

  Runtime Switch:

  `+svt_enable_pa=fsdb`

  Enables FSDB output of transaction and memory information for display in Verdi.

You can invoke the Protocol Analyzer as described above using Verdi. The Protocol Analyzer transaction view gets updated during the simulation.

5.4 Error Injection

AXI SVT VIP supports implementation of predefined errors that can be injected during the execution of a transaction. This is achieved through AXI SVT exception class `svt_axi_transaction_exception` and exception list class `svt_axi_transaction_exception_list`. The following sections describe these classes in detail.

5.4.1 Exception Class

The exception class for the AXI transaction class is `svt_axi_transaction_exception`. It extends from the svt_exception class. A randomizable error_kind property of type `error_kind_enum` exists in this class. This property defines the type of error to be injected.

```
/** Selects the type of error that will be injected. */
rand error_kind_enum error_kind = COHERENT_XACT_BYTES_LESS_THAN_CACHE_LINE_SIZE_ERROR;
```

For a list of supported error kinds, refer to `svt_axi_transaction_exception::error_kind` member in the AXI Class Reference HTML.

5.4.2 Exception Lists

The Exception list for AXI transaction is `svt_axi_transaction_exception_list`. It basically contains a queue of `svt_axi_transaction_exception` objects. An instance of the exception list class describes all of the exceptions applicable to an individual transaction. Therefore, a transaction descriptor should have a reference to an exception list.

For more details on the `svt_axi_transaction_exception_list` class, see the AXI Class Reference HTML.

5.4.3 Use Model

VIP provides the `svt_axi_master_callback` class. It contains all the callback methods called by the master component. You can implement an error injection callback class by extending from `svt_axi_master_callback` class. It is recommended that exceptions are added in the `post_input_port_get` callback to ensure that all exceptions are added before the transaction is processed and driven on the interface.

The following is an example of error injection callback class that extends from `svt_axi_master_callback` class.

The following is the code snippet for the VMM flow:

```
/**
   * This class extends from svt_axi_master_callback class.
   * It shows how user can inject pre-defined errors during the execution of a
transaction
  * through various callback methods. In this class post_input_port_get() callback is
  *  overridden and specific errors are injected.
   */
class cust_svt_axi_master_error_injection_callback extends svt_axi_master_callback;

   //  Instance of svt_axi_transaction_exception class
  svt_axi_transaction_exception my_exception;
  // Instance of svt_axi_transaction_exception_list class
```

```
  svt_axi_transaction_exception_list my_exception_list;
// vmm_log handle
 vmm_log log;
// Flag for checking randomization success or failure
 int result = 0;

  /**
   * Constructor of the class.
  */
  function new ( string name =
"cust_svt_axi_system_interconnect_error_injection_callback");
    super.new(name);
   log = new("cust_svt_axi_master_error_injection Callbacks", "Active Master");
  endfunction

/**
 * Overrides post_input_port_callback() method to inject errors through the
* use of SVT AXI exception classes.
 */
 virtual function void post_input_port_get(svt_axi_master axi_master,
svt_axi_transaction xact, ref bit drop);
   super.post_input_port_get(axi_master,xact,drop);

 // Construct svt_axi_transaction_exception and svt_axi_transaction_exception_list
class
  my_exception =new(log);
  my_exception_list =new(log);

  // Assign xact and cfg handle of svt_axi_transaction_exception class before
randomization
  my_exception.xact = xact;
  my_exception.cfg = xact.port_cfg;

  // Randomize exception class based on error_kind to be injected
  result = my_exception.randomize() with {error_kind ==
svt_axi_transaction_exception::INVALID_BURST_TYPE_FOR_COHERENT_XACT_ERROR;};

 if (!result) begin
 `vmm_error(log, $psprintf("Randomization FAILED for svt_axi_transaction_exception
class for error_kind = %0s",my_exception.error_kind.name()));
end

/**
    * Add a new exception with the specified content to the exception list.
    */
my_exception_list.add_exception(my_exception);


 // Initialize svt_axi_transaction_exception_list handle
(svt_axi_transaction::exception_list) with the user-defined
 // exception list. This way the error_kind of type
//   svt_axi_transaction_exception::INVALID_BURST_TYPE_FOR_COHERENT_XACT_ERROR
 // gets injected in this particular transaction.
 xact.exception_list = my_exception_list;
```

```
 endfunction //end of function post_input_port_get()
endclass //end of class cust_svt_axi_master_error_injection_callback
```

6 Verification Topologies

This chapter shows you from a high-level, how the AXI VIP can be used to test Master, Slave, or Interconnect DUT. This chapter discusses the following topics:

- Testing a Master DUT Using a VMM Slave VIP
- Testing a Slave DUT Using a VMM Master VIP
- Interconnect DUT and Master/Slave VIP

6.1 Testing a Master DUT Using a VMM Slave VIP

In this scenario, you are testing an AXI Master DUT using a VMM AXI Slave.

- Testbench setup: Configure the AXI System configuration to have one Slave component in active mode. The active Slave component will respond to the transactions generated by master DUT. The Slave component will also perform passive functions such as protocol checking, coverage generation and transaction logging.
- Implementation of this topology requires the setting of the following properties: (Assuming instance name of system configuration is "sys_cfg").
  - System configuration settings:
    - sys_cfg.num_masters = 0;
    - sys_cfg.num_slaves = 1;
  - Port configuration settings:
    - sys_cfg.slave_cfg[0].is_active = 1;

When DUT has a single AXI master port to be verified, testbench can either use a Slave component in standalone mode, or use a System component configured for a single slave component. The following advantages and disadvantages of the two approaches are listed:

Advantages of using standalone component versus system component:

- Testbench becomes light weight as system component and related infrastructure is not required

Disadvantages:

- The testbench does not remain scalable. If the number of AXI master ports to be verified increases, the standalone Slave component should to be replaced with System component, or, multiple Slave components would need to be instantiated by you.
- The AXI system monitor cannot be used, which is part of the System Env.

Figure 6-1 Master DUT and Slave VIP - Usage with Standalone Slave Component

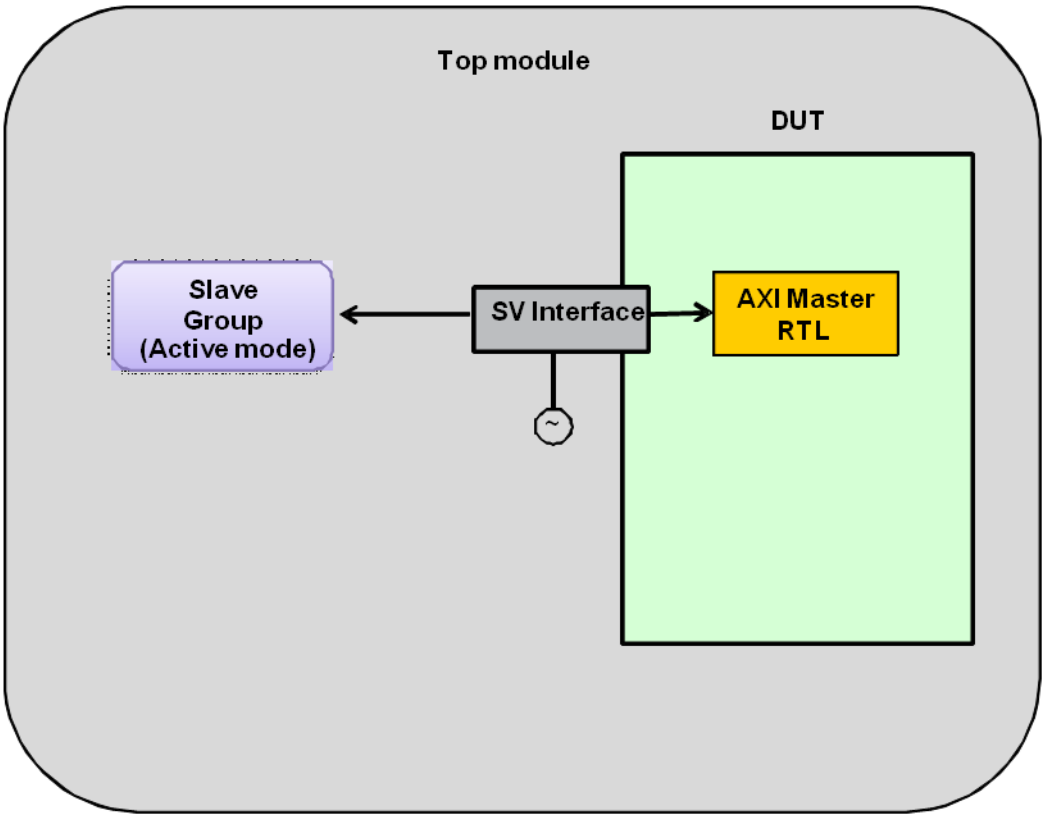
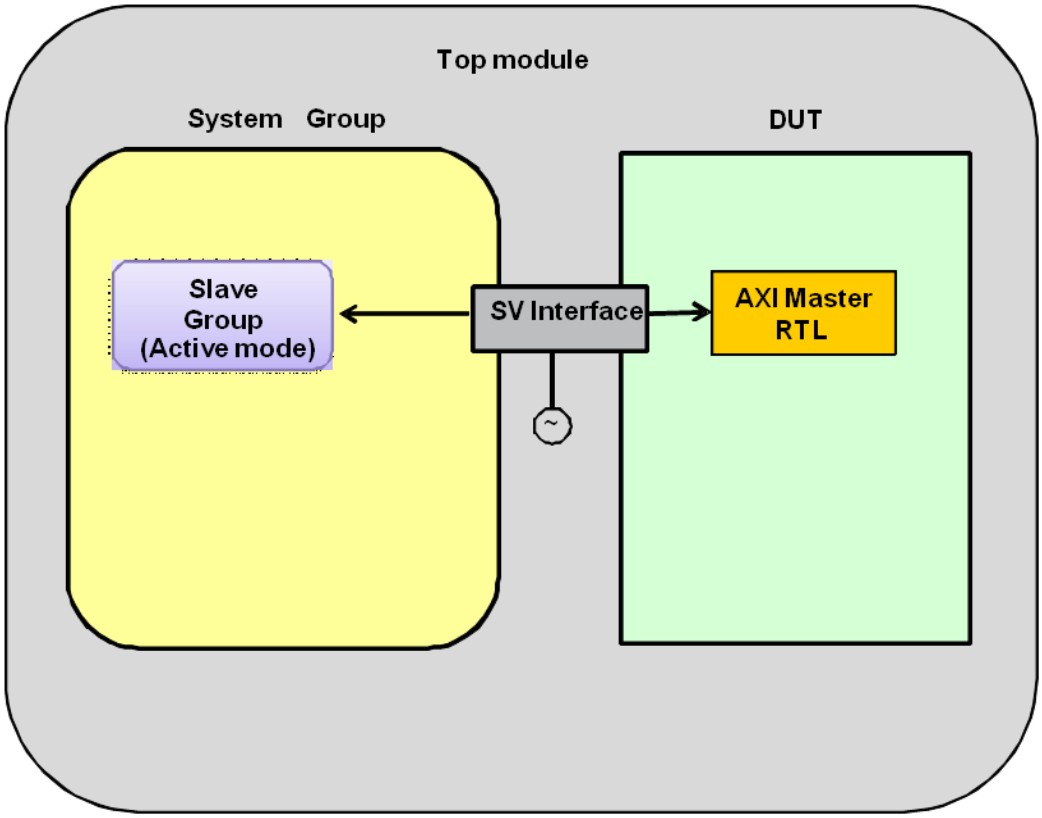


Figure 6-2 Master DUT and Slave VIP - Usage with System Component



6.2 Testing a Slave DUT Using a VMM Master VIP

In this scenario, you are testing an AXI Slave DUT using a VMM AXI Master. Configure the AXI System component to have one Master component, in active mode. The active master component will generate AXI transactions for the Slave DUT. The Master component will also perform passive functions such as protocol checking, coverage generation and transaction logging.

When DUT has a single AXI slave port to be verified, testbench can either use a Master component in standalone mode, or use a System component configured for a single Master component.

The following advantages and disadvantages of the two approaches are listed:

Advantages of using standalone component versus system component:

- Testbench becomes light weight as System component and related infrastructure is not required

Disadvantages:

- The testbench does not remain scalable. If the number of AXI master ports to be verified increases, standalone Slave component should be replaced with System component, or, multiple Slave components would need to be instantiated by you.
- The AXI system monitor cannot be used, which is part of the System Env.

Implementation of this topology requires the setting of the following properties (assuming instance name of system configuration is "sys_cfg"):

- System configuration settings:

- sys_cfg.num_masters = 1;
- sys_cfg.num_slaves = 0;

- Port configuration settings:
  - sys_cfg.master_cfg[0].is_active = 1;

Figure 6-3 Slave DUT and Master VIP - Usage with Standalone Master component

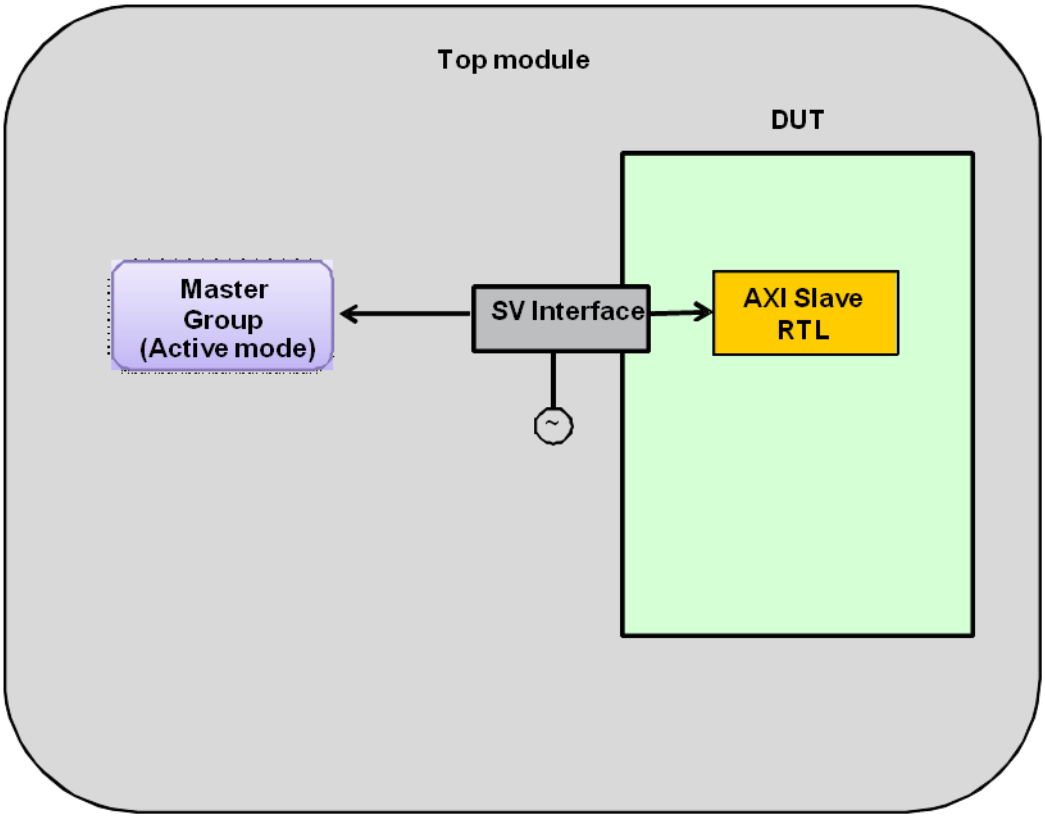
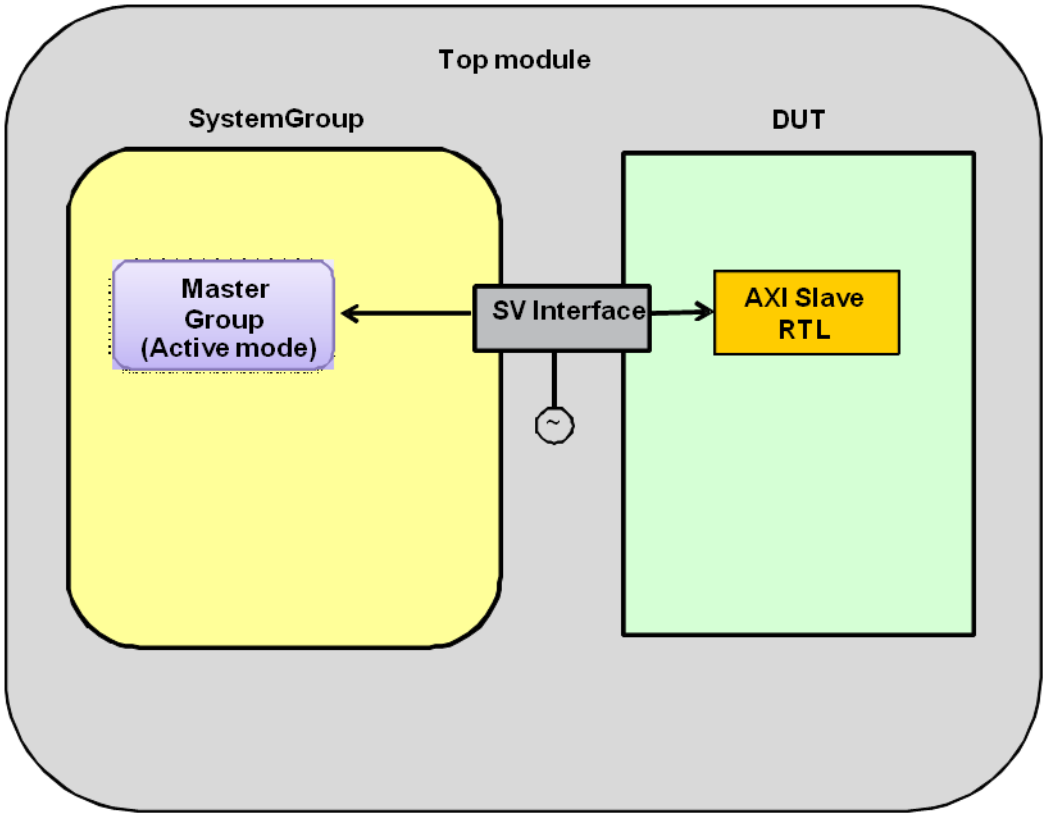


Figure 6-4 Slave DUT and Master VIP - Usage with System Component



6.3 Interconnect DUT and Master/Slave VIP

In this scenario, DUT is an AXI Interconnect tested by a Master and Slave VIP: Assuming that the AXI Interconnect has M master ports and S slave ports, configure the AXI System component to have S master components and M slave components, in active mode. The active master components will generate AXI transactions towards the interconnect slave ports, and active slave components connected would respond to the transactions generated by interconnect master ports. The master and slave components would also perform passive functions such as protocol checking, coverage generation and transaction logging.

Implementation of this topology requires the setting of the following properties:

- Assuming instance name of system configuration is "sys_cfg"
- Assuming number of master ports on interconnect = 2
- Assuming number of slave ports on interconnect = 2

System configuration settings:

- sys_cfg.num_masters = 2;
- sys_cfg.num_slaves = 2;

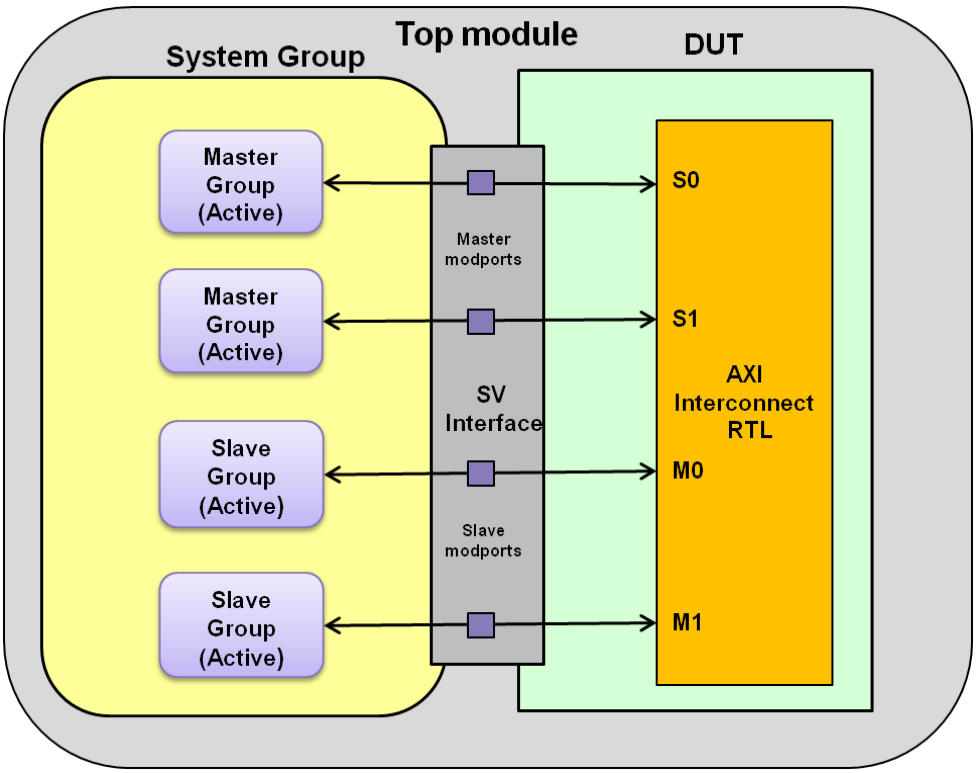
Port configuration settings:

- sys_cfg.master_cfg[0].is_active = 1;
- sys_cfg.master_cfg[1].is_active = 1;
- sys_cfg.slave_cfg[0].is_active = 1;

- sys_cfg.slave_cfg[1].is_active = 1;

Figure 6-5 shows the testbench setup.

Figure 6-5 Interconnect DUT with Master and Slave VIP (Active Mode)



6.3.1 System DUT with Passive VIP

In this setup, DUT is a AXI system with multiple AXI masters, slaves and interconnect. VIP is required to monitor DUT.

Assuming that the AXI System has M masters and S slaves, configure the AXI System component to have M master components and S slave components, in passive mode. The passive master and slave components would perform passive functions such as protocol checking, coverage generation and transaction logging.

- Implementation of this topology requires the setting of the following properties:
- Assuming instance name of system configuration is "sys_cfg"
- Assuming number of master ports on interconnect = 2
- Assuming number of slave ports on interconnect = 2

System configuration settings:

- sys_cfg.num_masters = 2;
- sys_cfg.num_slaves = 2;

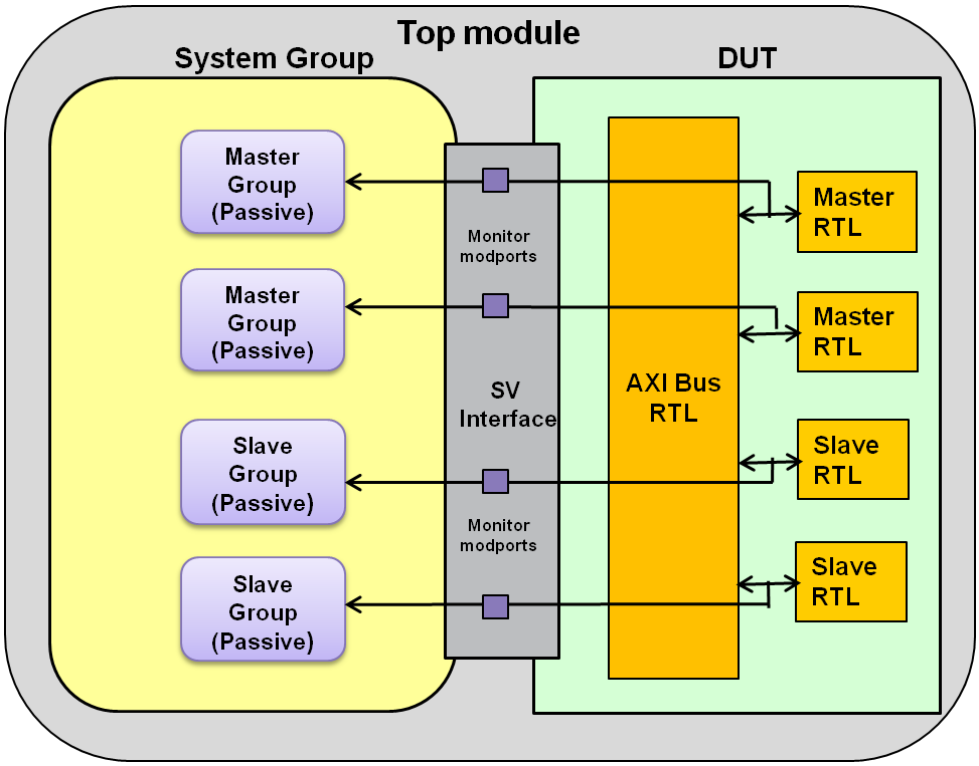
Port configuration settings:

- sys_cfg.master_cfg[0].is_active = 0;
- sys_cfg.master_cfg[1].is_active = 0;

- sys_cfg.slave_cfg[0].is_active = 0;
- sys_cfg.slave_cfg[1].is_active = 0;

Figure 6-6 shows this setup.

Figure 6-6 System DUT with Passive VIP



6.3.2 System DUT with Mix of Active and Passive VIP

In this scenario, DUT is a system with multiple AXI masters, slaves and interconnect. The VIP is required to provide background traffic on some ports, and to monitor on ports.

Assuming that the AXI System DUT has two master ports and two slave ports. VIP is required to provide background traffic to ports S0 and M0. All the ports need to be monitored. Configure the AXI System component to have two master components and two slave components. Configure the master component connected to port S0, and slave component connected to port M0 as active. Configure the master component connected to port M1 and slave component connected to port M1 as passive. All the components would continue to perform passive functions such as protocol checking and coverage.

Assuming instance name of system configuration is "sys_cfg".

System configuration settings:

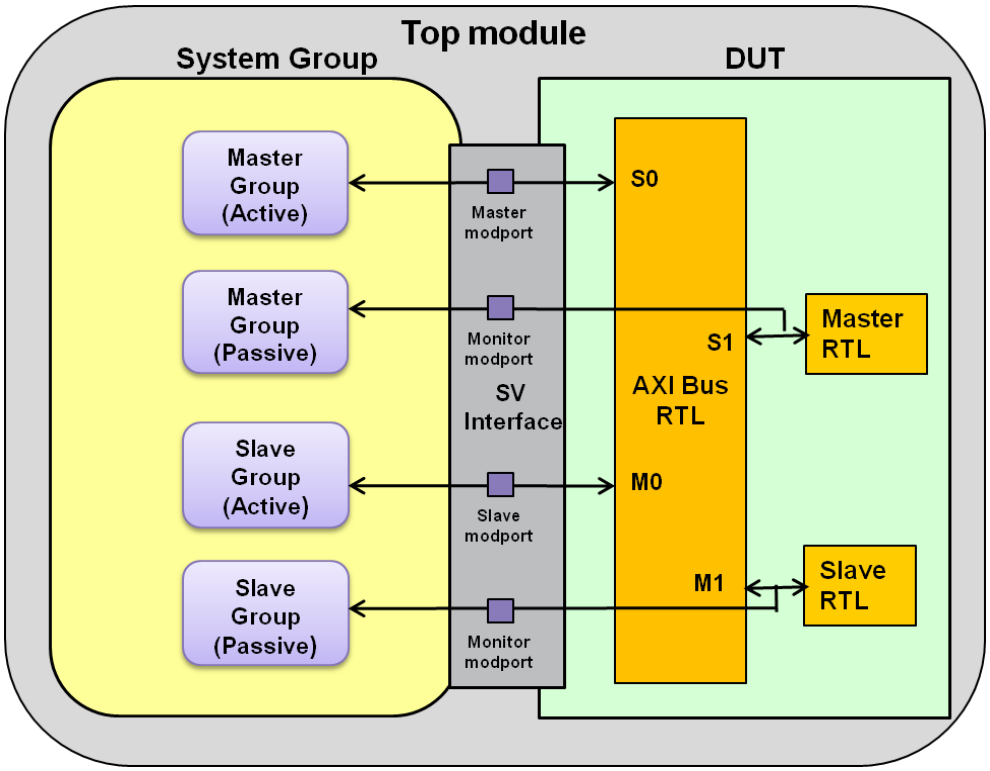
- sys_cfg.num_masters = 2;
- sys_cfg.num_slaves = 2;

Port configuration settings:

- sys_cfg.master_cfg[0].is_active = 1;
- sys_cfg.master_cfg[1].is_active = 0;

- sys_cfg.slave_cfg[0].is_active = 1;
- sys_cfg.slave_cfg[1].is_active = 0;

Figure 6-7 System DUT with Mix of Active and Passive VIP



7 Using AXI Verification IP

This chapter describes how to install and run a getting started example and provides usage notes for AXI Verification IP.

This chapter discusses the following topics:

- SystemVerilog VMM Example Testbenches
- Installing and Running the Examples
- How to Generate Slave Response
- How to Configure AXI Slaves with Overlapping Address
- How to Generate ACE WriteEvict Transactions
- How Does the Interconnect VIP Handle Barrier Transactions?
- How to Access a Byte Level Data of AXI Slave VIP Memory Using backdoor read/write?

7.1 SystemVerilog VMM Example Testbenches

This section describes SystemVerilog VMM example testbenches that show general usage for various applications. A summary of the examples is listed in Tables 7-1

Table 7-1 SystemVerilog Example Summary

Example Name	Level	Description
tb_axi_svt_vmm_basic_sys	Basic	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper and generates system clock • A program block, which includes tests and runs the tests • The program block creates a testbench environment, which in turn creates AXI System group • AXI System Group is configured with one master and one slave group

Table 7-1 SystemVerilog Example Summary (Continued)

Example Name	Level	Description
tb_axi_svt_vmm_intermediate_sys	Intermediate	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper and generates system clock • A program block, which includes tests and runs the tests • The program block creates a testbench environment, which in turn creates AXI System group • AXI System Group is configured with one master and one slave group • AXI VMM Scoreboard • Demonstrates how to override system constants • Coverage generation
tb_axi_svt_vmm_advanced_sys	Advanced	Not yet supported
tb_axi_svt_vmm_ace_sys		Derivative example to demonstrate ACE support. This example also demonstrates the use of system monitor.

The examples are located at:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/examples/sverilog/

7.2 Installing and Running the Examples

Below are the steps for installing and running example tb_axi_svt_vmm_basic_sys. The similar steps are also applicable for other examples:

1. Install the example using the following command line:

   $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -e amba_svt/tb_axi_svt_vmm_basic_sys -svtb

   The example would get installed under:

   <design_dir>/examples/sverilog/amba_svt/tb_axi_svt_vmm_basic_sys

2. Use either one of the following to run the testbench:

   a. Use the Makefile:

      The following tests are provided in the "tests" directory:

      i. ts.basic_random_test.sv
      ii. ts.basic_random_mem_resp_test.sv
      iii. ts.basic_random_user_resp_test.sv

      For example, to run test ts.basic_random_test.sv, perform the following steps:

      gmake USE_SIMULATOR=vcsvlog basic_random_test WAVES=1

      Invoke "gmake help" to show more options.

   b. Use the sim script:

      For example, to run test ts.basic_random_test.sv, perform the following steps:

      ./run_axi_svt_vmm_basic_sys -w basic_random_test vcsvlog

Invoke "./run_axi_svt_vmm_basic_sys -help" to show more options.

For more details of installing and running the example, refer to the README file in the example, located at:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/examples/sverilog/tb_axi_svt_vmm_basic_sys/README

OR

<design_dir>/examples/sverilog/amba_svt/tb_axi_svt_vmm_basic_sys/README

7.3 How to Generate Slave Response

The slave response generation is controlled using configuration parameter svt_axi_port_configuration::generator_type. The below response generator types are applicable for slave:

SIMPLE_RESPONSE_GEN

MEMORY_RESPONSE_GEN

USER_RESPONSE_GEN

7.3.1 Simple Random Response

If the response generator type is configured as SIMPLE_RESPONSE_GEN, then a built-in random response is provided by the slave response generator. In this case, the data is not read from or written into slave memory. Random data is provided for read transactions.

7.3.2 Memory Response

If the response generator type is configured as MEMORY_RESPONSE_GEN, the write data is written into slave memory, and read data is read from the slave memory. The built-in response is random. The memory is written or read only if response type is OKAY. In case of non-OKAY response type, the memory is not written or read, and 0 is returned on RDATA signals.

7.3.3 User Defined Response

If you want to provide a user defined response, below is the procedure:

1. Extend the svt_axi_slave_response_gen_callback class
2. Implement the callback method generate_response() to update the response
3. Register the extended callback class with the slave response generator

Refer to the basic example for mechanisms to provide slave responses.

7.4 How to Configure AXI Slaves with Overlapping Address

If the address map of slaves overlap with each other such as in the case of a dual port memory, the parameter svt_axi_system_configuration::allow_slaves_with_overlapping_addr must be set. The address map of each slave is then set using the svt_axi_system_configuration::set_addr_range method as is usually done.

For details on usage of this parameter, refer to the documentation of svt_axi_system_configuration::allow_slaves_with_overlapping_addr in AXI Class Reference Manual.

The following are some additional notes for configuring AXI Slaves with overlapping address:

- Any number of AXI slaves can have overlapping addresses.

- These slaves must lie within the same svt_axi_system_env instance if the AXI system monitor is used (by setting svt_axi_system_configuration::system_monitor_enable).
- The start address and end address of an address range that overlaps between multiple AXI slaves must match.

  In other words, the entire address range for a given range must match across multiple slaves. Partial overlap of an address range is not allowed. For details, refer to the documentation of svt_axi_system_configuration::allow_slaves_with_overlapping_addr parameter.
- Pass a shared memory across slaves that share a common address space. This can be done by creating an instance of svt_mem in the testbench and passing the same svt_mem instance through uvm_config_db to the slave groups that have a shared address space, as shown in the following code snippet:

```
svt_mem s0_s1_shared_mem = new("s0_s1_shared_mem", // Memory name
   "amba",                       // Suite name
   data_width, // data_width
   0,                            // Address region
   0,                            // Lower address bound to memory
   ((1<<this.cfg.slave_cfg[0].addr_width)-1));     // Upper address bound to memory

s0_s1_shared_mem.set_meminit(svt_mem::ADDRESS,0,0); // Memory initialization
   vmm_opts::set_object("axi_system:slave[0]:axi_slave_mem",s0_s1_shared_mem, this);
   vmm_opts::set_object("ahb_system:slave[0]:ahb_slave_mem",s0_s1_shared_mem, this);
```

- The attributes of the slaves which have overlapping addresses, such as data width, can be different.
- If the slaves that share memory have different data widths, the data width of the shared memory must be equal to or greater than the largest data width of the slaves that share an address space.

  For example, if there are three slaves of data width 32, 64 and 128 bits which share an address range, the data width of the memory created in the testbench can be 128, 256, 512 or 1024.
- If the slaves that share memory have different address widths, the minimum and maximum address of the memory created in the testbench should accommodate the largest addressable region.
- There is no arbitration of accesses between the two shared interfaces to memory. If there are accesses to the same location at the same time, the actual value written to memory is not deterministic.

7.5 How to Generate ACE WriteEvict Transactions

The AXI Verification IP supports the ACE WriteEvict transactions.

The following port configuration class members have been added to support this feature:

- svt_axi_port_configuration::writeevict_enable
- svt_axi_port_configuration::awunique_enable

The following port transaction class members have been added to support this feature:

- svt_axi_transaction::coherent_xact_type
- svt_axi_transaction::is_unique

For details, refer to the AXI Class Reference Manual.

7.6 How Does the Interconnect VIP Handle Barrier Transactions?

When a barrier transaction is received, the VIP blocks further progress of all transactions received after it until the transactions received prior to the barrier are complete. The response to a READBARRIER is sent only when prior READ type transactions are complete. The same applies to WRITEBARRIER. A barrier transaction does not result in a snoop to other masters.

Steps to debug the barrier transactions are as follows:

1. Check the number of transactions began, but did not end. Lookup for the `Transaction started` and `Transaction ended` message to figure this out.
2. From the transactions being performed, check how many are before the barrier and how many are after the barrier.
3. If there are any transactions before the barrier which is not complete, then the response to the barrier will not be sent. Subsequent transactions will also be blocked. So check why transactions before the barrier did not complete.
4. If there are transactions after the barrier which are blocked, check if the barrier itself completed.
5. Note that barriers in ACE are sent as pairs. So the core should be sending a READBARRIER and a WRITEBARRIER to the interconnect corresponding to a synchronization barrier.

The Interconnect VIP is not a behavioral model of the CCI-400. It is only one of the many implementations of the interconnect which is compliant to the ACE specification. In particular, we may not be bothered about ensuring optimal performance (not from a simulation perspective, but from bus utilization perspective etc.). The timings of the transactions (snoop for example) initiated by the interconnect VIP will be very different from CCI-400.

The easiest way to debug the Interconnect VIP is to enable the system monitor and see the transaction summary (the requirement is that there is an AXI VIP connected to every port). The system monitor can be enabled and simulation run in `UVM_HIGH` and you can grep for `TRANSACTION SUMMARY` at the end of the log. If you do not want to run simulation in `UVM_HIGH`, but would like to see the summary report the `svt_axi_system_configuration::display_summary_report` should be set.

7.7 How to Access a Byte Level Data of AXI Slave VIP Memory Using backdoor read/write?

The AXI slave VIP memory is modeled using the class `svt_mem`. The svt_mem's backdoor methods update the memory based on the `data_width`. There is no easy way to read/write a single byte of data at a given address location.

The AXI slave VIP has the following APIs that makes it easy to access byte level data:

```
task write_byte(input bit[`SVT_AXI_MAX_ADDR_WIDTH-1:0] addr, bit[7:0] data);
task read_byte(input bit[`SVT_AXI_MAX_ADDR_WIDTH-1:0] addr, output bit[7:0] data);
```

For example,

1. To write a single byte of data ('h0f) at address 'h100 from the test, you can use:

   ```
   env.axi_system_env.slave[0].write_byte(32'h100, 8'h0f);
   ```

2. To read a single byte of data at the address 'h200 from the test, you can use:

```
bit[7:0] read_data;
env.axi_system_env.slave[0].read_byte(32'h200, read_byte);
$display("data at address 'h200: %h", read_byte);
```

7.8 Data Integrity Checks

The following data integrity checks are performed by the system monitor. Please refer to the class reference documentation of the checks for specific information on the checks.

`svt_axi_system_checker::data_integrity_check`

`svt_axi_system_checker::data_integrity_with_outstanding_coherent_write_check`

`svt_axi_system_checker::master_slave_xact_data_integrity_check`

Note that data_integrity_check performs a check on data integrity by comparing data in a write or read transaction the contents of the memory for the same location. This check is performed only when svt_axi_system_configuration::posted_write_xacts_enable is not set. Please read the documentation of svt_axi_port_configuration::memory_update_for_read_xacts_enable also as its configuration is critical for data integrity checks. This variable must be 1 if a slave DUT is used and if it can return non-default data for locations not written into via frontdoor access (ie, through previous WRITE transactions). The check is bypassed in the following situations:

- A WRITE transaction to an overlapping address is detected during the liftetime of the transaction.
- svt_axi_port_configuration::memory_update_for_read_xacts_enable is set and a READ transaction to an overlapping address is detected during the lifetime of the transaction
- There is a WRITENOSNOOP or READNOSNOOP transaction to an address which is in a shareable domain or the shareability domains of the address received in the WRITENOSNOOP/READNOSNOOP transactions is not configured. Shareability domains are configured using svt_axi_system_configuration::create_new_domain() and svt_axi_system_configuration::set_addr_for_domain()

The master_slave_xact_data_integrity_check is performed to ensure data integrity between master transactions and the corresponding slave transactions across an interconnect. These checks are performed only if either of the following parameters are set: svt_axi_system_configuration::posted_write_xacts_enable or svt_axi_system_configuration:: master_slave_xact_data_integrity_check_enable. This check is performed by correlating slave transactions to master transactions and comparing data. The system monitor requires additional information so that it can properly correlate master transactions and slave transactions. The following fields must be set in the configuration:

`svt_axi_system_configuration::id_based_correlation_enable`

`svt_axi_system_configuration::source_master_info_id_width`

`svt_axi_system_configuration::source_master_info_position`

`svt_axi_system_configuration::source_interconnect_id_xmit_to_slaves`

`svt_axi_system_configuration::source_master_id_wu_wlu_xmit_to_slaves`

`svt_axi_port_configuration::source_master_id_xmit_to_slaves`

You can retrieve a system transaction that has the correlated information through the following callback in svt_axi_system_monitor_callback:

```
virtual function void
master_xact_fully_associated_to_slave_xacts(svt_axi_system_monitor
system_monitor,svt_axi_system_transaction sys_xact);
endfunction
```

The following properties in the system transaction can be used to retrieve the master and slave transaction information from the callback. A user could perform custom checks in the callback based on this.

```
svt_axi_system_transaction::master_xact
svt_axi_system_transaction::assoc_slave_xacts[$]
```

The `master_slave_xact_data_integrity_check` is not intended to replace the data_integrity_check, it is meant to supplement it by addressing the situations where memory based checking has limitations and need to be bypassed. It is also provides a powerful mechanism for users to do custom functional as well as performance checks based on the callback provided.

7.9 Setting up Secure and Non-Secure Access Mechanism for AXI-ACE Master

By default when cacheline is written or read by VIP master ~~agent~~group or backdoor APIs , they are agnostic to any secure/non-secure protection specialization. ACE protocol indicate the Secure/Non-Secure access using AxPROT[1].

Here is how VIP can be configured to manage this protection mechanism

1. The following port configuration attributes enable this mechanism for corresponding VIP group

   ```
   svt_axi_port_configuration::tagged_address_space_attributes_enable = 1;
   ```

   // This Enables support of independent secure and non-secure address space, including cacheline for snoop response

2. The following this configuration setting the AxProt[1] bit will be used as ADDRESS - MSB by VIP to write/read the cacheline , so address width must be set accordingly.

   For example, if you had set SVT_AXI_MAX_ADDR_WIDTH macro to maximum address width possible across any group in given system (for example, 44) , then you need to set it '1+' to accommodate the tagged address bit (MSB appended to address coming from AxPROT[1]).

   Example:

   ```
   `define SVT_AXI_MAX_ADDR_WIDTH 45
   ```

3. You also need to set the address tag attribute width, which is used to indicate on how many bits of AxPROT will be used (currently there is support for only AxPROT[1] , hence setting it to '1' is good enough)

   Example

   ```
   `define SVT_AXI_ADDR_TAG_ATTRIBUTES_WIDTH 1
   ```

4. Set the VIP master group `addr_width <= SVT_AXI_MAX_ADDR_WIDTH - `SVT_AXI_ADDR_TAG_ATTRIBUTES_WIDTH`

Here is how VIP manages secure and non-secure address ranges :-

Whenever VIP master will see the cacheline store (For example, addr 20000000000), then the address information in cache will be stored based on actual address , appended with MSB value from AxPROT[1] (that address MSB value stored in cache will be!(AxPROT[1]))

For example, address stored in cache for the case , where AxPROT[1] was ''1' (i.e non secure) will be 020000000000.

User backdoor WRITE and READ to Master Cache

These operations should access respective cacheline using thess tagged address (MSB appended to actual cacheline address as 1/0 for secure and non-secure respectively)

Each write backdoor should be followed by set_prot_type call for that cacheline. VIP also does it under the hood when it writes to cache.

```
svt_axi_cache::set_prot_type(addr_t addr, int is_privelged = -1, int is_secure = -1 ,
int is_instruction = -1)
```

VIP support only AxPROT[1] value currently that is, is_secure argument , hence passing any/no value to other arguments (is_privileged and is_instruction) will not make any difference.

7.10 Snoop Filter Support

Some interconnects have a snoop filter which keeps track of allocations and de-allocations of cachelines in masters connected to it. The interconnect uses this information to decide which masters to snoop when a coherent transaction is received. Snoops are not broadcasted, but sent only to masters that have an entry in the cache. From a system monitor's point of view, the main difference when it is connected to an interconnect with/without a snoop filter is in the correlations it makes between coherent and snoop transactions and the corresponding checks on the ports on which it expects snoop transactions. If snoop filter is not enabled, all ACE masters will be expected to receive a snoop corresponding to a coherent transaction. If snoop filter is enabled, the system monitor keeps track of allocations and deallocations of cachelines in ACE masters and it expects only those ports which have an allocation to be snooped. Snoop filter configuration is a port level configuration. If the interconnect supports snoop filter, all masters connected to the interconnect must have the following parameter set:

svt_axi_port_configuration::snoop_filter_enable

7.11 Configuration Requirements to Enable System Level Cover Groups Which Use Master to Slave Transaction Association

To enable system level cover groups which use master to slave transaction association, user needs to enable following configuration parameter:

svt_axi_system_configuration::id_based_xact_correlation_enable

7.12 Exclusive Access Support

7.12.1 Exclusive Access Related Configurations

AMBA VIP uses individual exclusive access monitor for each port and therein tracking each exclusive sequences independently through combination of transaction ID (representing master id) and transaction address. Exclusive monitor sets and resets itself depending on the sequence of exclusive or normal type transactions. While it tracks each exclusive sequence it prepares expected response for both read and write transactions based on whether the exclusive sequence was successful or not. If any mismatch found between expected and actual response, then it reports error and possible underlying cause i.e. whether there was no exclusive read performed or if the monitor is already reset or exclusive read address was reset before receiving exclusive write.

```
// enables or disables exclusive access completely. VIP neither generates EXOK response
nor expects it.
exclusive_access_enable
```

// exclusive monitors for each port can be disabled even if exclusive access is enabled. If disabled then VIP doesn't track exclusive accesses and allows all type of responses to exclusive transactions and doesn't report any error related to exclusive access checks.

exclusive_monitor_enable

// indicates the maximum number of open exclusive sequence supported. Once an exclusive sequence is started inside exclusive monitor it is checked against existing number of open exclusive sequences. When exclusive write completes the sequence then that sequence is removed from the exclusive sequence tracking. Attempts to exceed this max number results in a failed exclusive access read response of OKAY instead of EXOKAY.

Currently, it can not be disabled by setting 0. This will be added in later version.

```
max_num_exclusive_access
```

* Number of ADDRESS bits that need to be monitored by the exclusive monitors for current port in order to support one or more independent exclusive access thread. This is currently applicable only for ACE Exclusive transactions.

* NOTE: configuring with value 0 means, no address is being monitored by the

* corresponding exclusive monitor and hence exclusive access to different address

* may also affect current thread.

num_addr_bits_used_in_exclusive_monitor

* similar as above

* This is currently applicable only for ACE Exclusive transactions.

num_id_bits_used_in_exclusive_monitor

* If set to '1' then VIP will not assert error if Master sends Exclusive Store without sending Exlusive Load.

* However, if Exclusive Store is sent from the Invalid cacheline state then VIP will still assert error since,

* that is not a valid state to start Exclusive Store.

rand bit allow_exclusive_store_without_exclusive_load = 0;

* If set to '1' then VIP will respond to very first Exclusive Store with EXOKAY response. This means that if

* no master has performed any excluisve transaction after reset is de-asserted then then if one master issues

* exclusive store then VIP will respond with EXOKAY or will expect EXOKAY response from the coherent interconnect.

* Note: reference point of first exclusive store is reset.

rand bit allow_first_exclusive_store_to_succeed = 0;

* If set to '1' then Exclusive Monitor will get reset once Exclusive Store is successful.

* If set to '0' then Exclusive Monitor will remain set even if Exclusive Store is successful.

* Please Note that, VIP will still reset Exclusive Monitor for failed Exclusive Store attempt

* regardless of the value set for this parameter.

rand bit reset_exclusive_monitor_on_successful_exclusive_store = 1;

7.12.2 Exclusive Access Checks

```
/** Checks that ARLEN and ARSIZE are valid for exclusive read transaction */
 signal_valid_exclusive_arlen_arsize_check;

/** Checks that ARCACHE is valid for exclusive read transaction */
 signal_valid_exclusive_arcache_check;

/** Checks that address is aligned for exclusive read transaction */
 signal_valid_exclusive_read_addr_aligned_check;

/** Checks that AWLEN and AWSIZE are valid for exclusive read transaction */
 signal_valid_exclusive_awlen_awsize_check;

/** Checks that AWCACHE is valid for exclusive read transaction */
 signal_valid_exclusive_awcache_check;

/** Checks that address is aligned for exclusive read transaction */
 signal_valid_exclusive_write_addr_aligned_check;

/** Checks that address is generated same for exclusive read and write
 * transactions */
 exclusive_read_write_addr_check;

/** Checks that id is generated same for exclusive read and write
 * transactions */
 exclusive_read_write_id_check;
```

```
  /** Checks that response generated for exclusive load accesss is correct */
    exclusive_load_response_check;

   /** Checks that response generated for exclusive store accesss is correct */
    exclusive_store_response_check;

   /** Checks that master does not permit an Exclusive Store transaction to be
     * in progress at the same time as any transaction that registers that it
     * is performing an Exclusive sequence
     */
    exclusive_store_overlap_with_another_exclusive_sequence_check;

   /** Checks that, once a master receives successful exclusive store response EXOKAY
     * from interconnect, then no other master should be provided with EXOKAY response,
     * until current master acknowledges completing successful exclusive store by
asserting RACK
     */
     exokay_not_sent_until_successful_exclusive_store_rack_observed_check;


      /** Checks that READ_ONLY_INTERFACE does not support exclusive access
      * Applicable only for AXI4 VIP
      * Passive Master,Passive Slave and Active slave will perform this
      * check
      */
       excl_access_on_read_only_interface_check;

      /** Checks that WRITE_ONLY_INTERFACE does not support exclusive access
       * Applicable only for AXI4 VIP
       * Passive Master,Passive Slave and Active slave will perform this check
       */
        excl_access_on_write_only_interface_check;

      /** Checks that burst length is generated same for exclusive read and write
    * transactions */
    exclusive_read_write_burst_length_check;

   /** Checks that burst size is generated same for exclusive read and write
    * transactions */
    exclusive_read_write_burst_size_check;
```

```
  /** Checks that burst type is generated same for exclusive read and write
   * transactions */
  exclusive_read_write_burst_type_check;

  /** Checks that cache type is generated same for exclusive read and write
   * transactions */
   exclusive_read_write_cache_type_check;

  /** Checks that protection type is generated same for exclusive read and write
   * transactions */
   exclusive_read_write_prot_type_check;

  /** Checks that exclusive transaction sent on AXI_ACE interface are
   * only of WRITENOSNOOP, READNOSNOOP, READCLEAN, READSHARED and CLEANUNIQUE type */
   exclusive_ace_transaction_type_check;

  /**Checks the valid response of EXOKAY response is only for readnosnoop Transactions
*/
   exokay_resp_observed_only_for_exclusive_transactions_check;

  /**Checks that if cacheline is in invalid state then exclusive load transaction is
issued only as READCLEAN or READSHARED */
  exclusive_load_from_valid_state_check;

  /**Checks that if cacheline is in invalid state then exclusive store transaction is
not issued */
   exclusive_store_from_valid_state_check;

  /**Checks that if cacheline is in shared state then exclusive transaction is issued
only as CLEANUNIQUE, READCLEAN or READSHARED*/
   exclusive_transaction_from_shared_state_check;

  /** Checks that an exclusive sequence is reset after a cacheline is
    * invalidated by a snoop. This checks that after a snoop invalidates
    * a cacheline, an exclusive load is always sent prior to sending the
    * exclusive store.     */
  restart_exclusive_seq_post_cache_line_invalidation_check;

  /** Checks that if cacheline is in invalid state then exclusive load transaction is
issued
    * only as READCLEAN or READSHARED     */
```

```
  exclusive_load_from_valid_state_sys_check;

  /** Checks that if cacheline is in invalid state then exclusive store transaction is
not issued    */
  exclusive_store_from_valid_state_sys_check;

signal_valid_exclusive_arlen_arsize_check
signal_valid_exclusive_arcache_check
signal_valid_exclusive_read_addr_aligned_check
signal_valid_exclusive_awlen_awsize_check
signal_valid_exclusive_awcache_check
signal_valid_exclusive_write_addr_aligned_check
exclusive_read_write_addr_check
exclusive_read_write_id_check
exclusive_load_response_check
exclusive_store_response_check
exclusive_store_overlap_with_another_exclusive_sequence_check
exokay_not_sent_until_successful_exclusive_store_rack_observed_check
exclusive_read_write_burst_length_check
exclusive_read_write_burst_size_check
exclusive_read_write_burst_type_check
exclusive_read_write_cache_type_check
exclusive_read_write_prot_type_check
exclusive_ace_transaction_type_check
exokay_resp_observed_only_for_exclusive_transactions_check
exclusive_load_from_valid_state_check
exclusive_store_from_valid_state_check
exclusive_transaction_from_shared_state_check
restart_exclusive_seq_post_cache_line_invalidation_check
exclusive_load_from_valid_state_sys_check
exclusive_store_from_valid_state_sys_check
```

7.12.3 How Exclusive Accesses From Multiple Clusters are Modeled in System Monitor (exclusive monitor per ID)

System Monitor uses Exclusive Monitor for each AXI_ACE port irrespective of exclusive monitor inside port monitor. System Monitor currently doesn't track exclusive accesses for AXI3/AXI4/ACE_LITE ports as those are tracked at port monitor level. Additionally, each of these Exclusive monitors independently tracks supported number of exclusive sequence based on transaction ID and address. Number of bits considered for transaction ID can be used to model multiple processors within a single cluster. For ACE Exclusive accesses, each of these PoS Exclusive Monitors observes all master transactions in the system i.e. both coherent and snoop transactions. This means, for each coherent transaction all PoS Exclusive Monitors take

appropriate actions based on its internal state and the observed transaction. If multiple exclusive sequence is open in more than one PoS Excl Monitor then one may make it successful and others will get reset. It is also possible that based on the observed coherent or snoop transaction monitor will get reset. Invalidating snoop transactions will always reset corresponding PoS Exclusive Monitor.

7.13 Backdoor Cache Access Methods

Cache is modelled using the class 'svt_axi_cache'. This has the following methods for backdoor access. See the html class reference doc for descriptions:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_svt_uvm_class_reference/html/class_svt_axi_cache.html

```
function bit         backdoor_write ( int index , addr_t addr = 0, bit [7:0] data [], bit byteen [], int is_unique = -1, int is_clean = -1, longint age = -1 )
function bit         get_cache_type ( addr_t addr , output bit [3:0] cache_type )
function bit         get_prot_type ( addr_t addr , output bit is_privileged , output bit is_secure , output bit is_instruction )
function bit         get_status ( addr_t addr , output bit is_unique , output bit is_clean )
function bit         invalidate_addr ( addr_t addr )
function void      invalidate_all ( )
function bit         read_by_addr ( input addr_t addr , output int index , output bit [7:0] data [], output bit is_unique , output bit is_clean , output longint age )
function bit         set_cache_type ( addr_t addr , bit [3:0] cache_type )
function bit         set_prot_type ( addr_t addr , int is_privileged = -1, int is_secure = -1, int is_instruction = -1 )
function bit         update_status ( addr_t addr , int is_unique , int is_clean )
```

7.14 AXI4 Stream Protocol

7.14.1 Concepts

The AXI4-stream protocol is used as a standard interface to connect components that share data. The interface can be used to connect a single master, that generates data, to a single slave, that receives data. The protocol can also be used when connecting larger numbers of master and slave components. The data is shared in the form of data streams. A data stream can be a series of individual byte transfers or a series of byte transfers grouped together in packets.

The following section describes the below components:

7.14.1.1 Master Group

The Master Group encapsulates Master Sequencer, Master Driver, and Port Monitor. The Master Group can be configured to operate in active mode and passive mode. You can provide AXI4_STREAM sequences to the Master Sequencer.

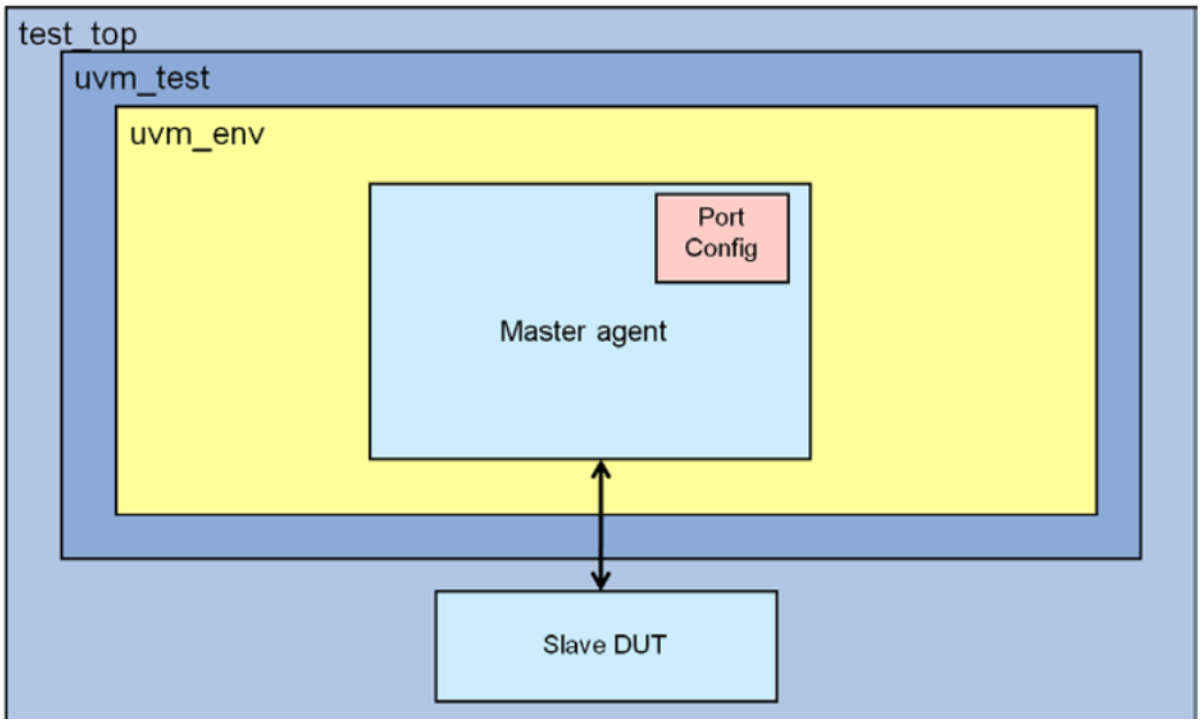
The Master Group is configured using a port configuration, which is available in the system configuration. The port configuration should be provided to the Master Group in the build phase of the test.

Within the Master Group, the Master Driver gets sequences from the Master Sequencer. The Master Driver then drives the AXI4_STREAM transactions on the AXI4_STREAM port. The Master Driver and port

Monitor components within Master ~~Agent~~Group call callback methods at various phases of execution of the AXI4_STREAM transaction.

After the AXI4_STREAM transaction on the bus is complete, the completed sequence item is provided to the analysis port of Port Monitor for use by the testbench.

Figure 7-1 Usage With Standalone Master ~~Agent~~Group



7.14.1.2 Slave Group

The Slave Group encapsulates Slave Sequencer, Slave Driver, and Port Monitor. The Slave Group can be configured to operate in active mode and passive mode. You can provide ATB response sequences to the Slave Sequencer.

The Slave Group is configured using port configuration, which is available in the system configuration. The port configuration should be provided to the Slave Group in the build phase of the test or the testbench environment.

In the Slave Group, the Port Monitor samples the AXI4_STREAM port signals. When a new transaction is detected, the Port Monitor provides a response request sequence item to the Slave Sequencer through port `response_request_port`. The slave response sequence within the sequencer programs the appropriate slave response. The updated response sequence item is then provided by the Slave Sequencer to the Slave Driver. The Slave Driver in turn drives the response on the AXI4_STREAM bus.

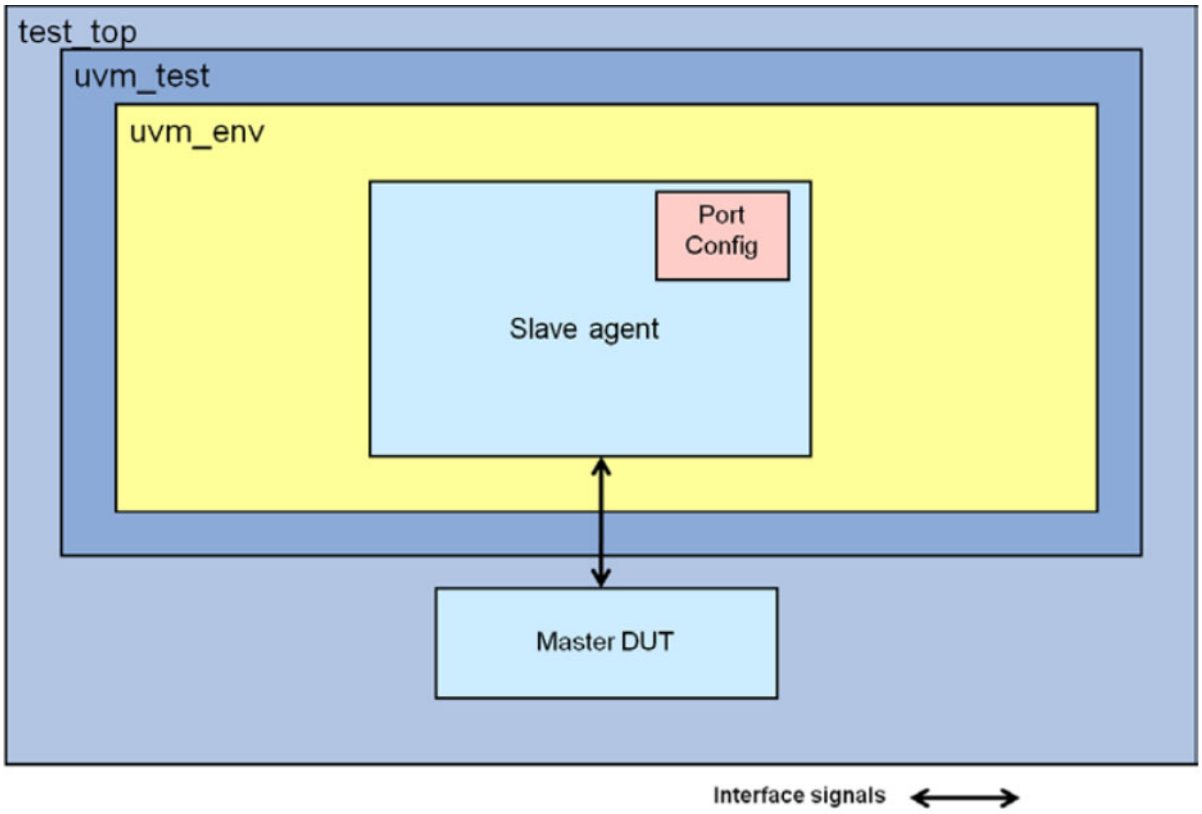
The slave driver expects the slave response sequence to,

- Return same handle of the slave response object as provided to the sequencer by the port monitor
- Return the slave response object in zero time, that is, without any delay after sequencer receives object from the port monitor

If any of the above conditions are violated, the slave group issues a FATAL message.

The Slave Driver and Monitor call callback methods at various phases of execution of the AXI4_STREAM transaction. After the AXI4_STREAM transaction on the bus is complete, the completed sequence item is provided to the analysis port for use by the testbench.

Figure 7-2 Usage with Standalone Slave Group



7.14.1.3 Groups in Active and Passive Mode

Component behavior in active mode

In active mode, Master and Slave components generate transactions on the signal interface.

Master and Slave components continue to perform passive functionality of coverage and protocol checking. You can enable/disable this functionality through configuration.

The Port Monitor within the component performs protocol checks only on sampled signals. That is, it does not perform checks on the signals that are driven by the group. This is because when the group is driving an exception (exceptions are not supported in this release) the Monitor should not flag an error, since it knows that it is driving an exception. Exception means error injection.

Component behavior in passive mode

In passive mode, master and slave components do not generate transactions on the signal interface. These components only sample the signal interface.

Master and Slave components monitor the input and output signals, and perform passive functionality such as coverage and protocol checking. You can enable/disable this functionality through configuration options.

The port monitor within the component performs protocol checks on all signals. In passive mode, signals are considered as input signals.

7.14.1.4 AXI4_STREAM UVM User Interface

The following sections give an overview of the user interface into the AXI4_STREAM VIP.

AXI4_STREAM VIP uses the `svt_axi_port_configuration` class for assigning port configuration values.

The following parameters can be set to use AXI4_STREAM VIP:

```
svt_axi_port_configuration: axi_interface_type
```

`axi_interface_type` must be set to AXI4_STREAM to configure the interface of the port to AXI4_STREAM.

The following is the description of some of the port configuration attributes for AXI4_STREAM.The width of the tdata signal can be set using `svt_axi_port_configuration::tdata_width`

The width of tid_signal can be set using `svt_axi_port_configuration::tid_width`

The width of tdest_signal can be set using `svt_axi_port_configuration::tdest_width`

An elaborate description of port configuration attributes can be found in svdoc.

AXI4_STREAM VIP uses `svt_axi_transaction` class as its base transaction class.

`Xact_type = svt_axi_transaction::DATA_STREAM` must be set for AXI4_STREAM transactions.

The detailed description of transaction class attributes can be found in svdoc.

8 Troubleshooting

This chapter provides some useful information that can help you troubleshoot common issues that you may encounter while using the AXI VIP. This chapter discusses the following topics:

- Using Debug Port

8.1 Using Debug Port

Port interfaces svt_axi_master_if and svt_axi_slave_if of AXI VIP provide a modport svt_axi_debug_modport for debugging purpose. The signals in the debug modport represent the transaction number and beat number which are currently executing on all channels of the AXI port.

The debug port signals starting with "mon" are driven by the port monitor within the Master and Slave component. The signals, read_addr_xact_num, write_addr_xact_num, write_data_xact_num and write_data_beat_num are driven by master driver in Master component. The signals, read_data_xact_num, read_data_beat_num and write_resp_xact_num are driven by slave driver in Slave component.

A Reporting Problems

A.1 Introduction

This chapter outlines the process for working through and reporting VIP transactor issues encountered in the field. It describes the data you must submit when a problem is initially reported to Synopsys. After a review of the initial information, Synopsys may decide to request adjustments to the information being requested, which is the focus of the next section. This section outlines the process for working through and reporting problems. It shows how to use Debug Automation to enable all the debug capabilities of any VIP. In addition, the VIP provides a case submittal tool to help you pack and send all pertinent debug information to Synopsys Support.

A.2 Debug Automation

Every Synopsys model contains a feature called "debug automation". It is enabled through *svt_debug_opts* plusarg. The Debug Automation feature allows you to enable all relevant debug information. The following are critical features of debug automation:

- Enabled by the use of a command line run-time plusarg.
- Can be enabled on individual VIP instances or multiple instances using regular expressions.
- Enables debug or verbose message verbosity:
  - The timing window for message verbosity modification can be controlled by supplying `start_time` and `end_time`.
- Enables at one time any, or all, standard debug features of the VIP:
  - Transaction Trace File generation
  - Transaction Reporting enabled in the transcript
  - PA database generation enabled
  - Debug Port enabled
  - Optionally, generates a file name *svt_model_out.fsdb* when Verdi libraries are available

When the Debug feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named *svt_debug.transcript*.

A.3 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named +*svt_debug_opts. T*his plusarg accepts an optional string-based specification to control various aspects Debug Automation. If this

command control specification is not supplied, then the feature will default to being enabled on all VIP instances with the default options listed as follows:

Note the following about the plusarg:

- The command control string is a comma separated string that is split into the multiple fields.
- All fields are optional and can be supplied in any order.

The command control string uses the following format (white space is disallowed):

```
inst:<inst>,type:<string>,feature:<string>,start_time:<longint>,end_time:<longint>,verbosity:<string>
```

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

Field	Description
inst	Identifies the VIP instance to apply the debug automation features. Regular expressions can be used to identify multiple VIP instances. If this value is not supplied, and if a type value is not supplied, then the debug automation feature will be enabled on all VIP instances.
type	Identifies a class type to apply the debug automation features. When this value is supplied then debug automation will be enabled for all instances of this class type.
feature	Identifies a sub-feature that can be defined by VIP designers to identify smaller grouping of functionality that is specific to that title. The definition and implementation of this field is left to VIP designers, and by default it has no effect on the debug automation feature. (Specific to VIP titles)
start_time	Identifies when the debug verbosity settings will be applied. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the verbosity settings will be applied at time zero.
end_time	Identifies when the debug verbosity settings will be removed. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the debug verbosity remains in effect until the end of the simulation.
verbosity	Message verbosity setting that is applied at the `start_time`. Two values are accepted in all methodologies: DEBUG and VERBOSE. UVM and OVM users can also supply the verbosity that is native to their respective methodologies (UVM_HIGH/UVM_FULL and OVM_HIGH/OVM_FULL). If this value is not supplied then the verbosity defaults to DEBUG/UVM_HIGH/OVM_HIGH. When this feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

- containing the string "endpoint" with a verbosity of UVM_HIGH
- starting at time zero (default) until the end of the simulation (default):

```
+svt_debug_opts=inst:/.*endpoint.*/,verbosity:UVM_HIGH
```

Enable on all instances:

- starting at time 1000 until time 1500:

```
+svt_debug_opts=start_time:1000,end_time:1500,verbosity:VERBOSE
```

Enable debug feature on all instances using default options:

- By setting the macro SVT_DEBUG_OPTS to 1 in the command line, the debug feature is enabled on all instances using default options. The macro will enable the XMLs and Trace files.

```
gmake <testname> SVT_DEBUG_OPTS=1 PA=FSDB
```

Note

The SVT_DEBUG_OPTS option is available through the installed VIP examples, but if required, in customer environments, then a similar feature should be added to their environment.
The PA=FSDB option is available in public examples and is required to enable Verdi libraries, and that when this option is used, then the Debug Opts file will record VIP activity to a file named `svt_model_log.fsdb`.
In addition, the SVT Automated Debug feature will enable waveform generation to an FSDB file, if the Verdi libraries are available. When enabled this feature, it should cause the simulator to dump waveform information only for the VIP interfaces.

When this feature is enabled then all VIP instances that have been enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

A.4 Debug Automation Outputs

The Automated Debug feature generates a *svt_debug.out* file. It records important information about the debug feature itself, and data about the environment that the VIPs are operating in. This file records the following information:

- The compiled timeunit for the SVT package
- The compiled timeunit for each SVT VIP package
- Version information for the SVT library
- Version information for each SVT VIP
- Every SVT VIP instance, and whether the VIP instance has been enabled for debug
- For every SVT VIP enabled for debug, a list of configuration properties that have been modified to enable debug will be listed
- A list of all methodology phases will be recorded, along with the start time for each phase

The following are the output files generated:

- *svt_debug.out*: It records important information about the debug feature itself, and data about the environment that the VIPs are operating. One file is optionally created when this feature is enabled, depending on if the Verdi libraries are available.
- *svt_debug.transcript*: Log files generated by the simulation run.
- *transaction_trace*: Log files that records all the different transaction activities generated by VIPs.
- *svt_model_log.fsdb*: Contains PA FSDB information (if the VIP supports this), and which contains other recorded activity. The additional information records signal activity associated with the VIP interface, TLM input (through SIPP ports), other TLM output activity, configurations applied to the VIP, and all callback activity (recorded by before and after callback execution).

A.5 FSDB File Generation

To enable FSDB writing capabilities, the simulator compile-time options and environment must be updated to enable this. The steps to enable this are specific to the simulator being used (the {LINUX/LINUX64} label needs to be replaced based on the platform being used). The ability to write to an FSDB file requires that the user supplies the Verdi dumper libraries when they compile their testbench. If these are not supplied then the VIP will not be enabled to generate the *svt_model_log.fsdb* file.

A.5.1 VCS

The following must be added to the compile-time command:

```
-debug_access
```

For more information on how to set the FSDB dumping libraries, see Appendix B section in *Linking Novas Files with Simulators and Enabling FSDB Dumping* guide available at:
$VERDI_HOME/doc/linking_dumping.pdf.

A.5.2 Questa

The following must be added to the compile-time command:

```
+define+SVT_FSDB_ENABLE -pli novas_fli.so
```

A.5.3 Incisive

The following must be added to the compile-time command:

```
+define+SVT_FSDB_ENABLE -access +r
```

A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

1. Before you contact technical support, be prepared to provide the following:
   - A description of the issue under investigation.
   - A description of your verification environment.

Enable the Debug Opts feature. For more information, see the Debug Automation.

A.7 Sending Debug Information to Synopsys

To help you debug testing issues, follow the given instructions to pack all pertinent debug information into one file which you can send to Synopsys (or to other users in your company):

1. Create a description of the issue under investigation. Include the simulation time and bus cycle of the failure, as well as any error or warning messages that are part of the failure.
2. Create a description of your verification environment. Assemble information about your simulation environment, making sure to include:
   - OS type and version
   - Testbench language (SystemVerilog or Verilog)
   - Simulator and version
   - DUT languages (Verilog)

3. Use the VIP case submittal tool to pack a file with the appropriate debug information. It has the following usage syntax:

   ```
   $DESIGNWARE_HOME/bin/snps_vip_debug [-directory <path>]
   ```

   The tool will generate a "<username>.<uniqid>.svd" file in the current directory. The following files are packed into a single file:

   - FSDB
   - HISTL
   - MISC
   - SLID
   - SVTO
   - SVTX
   - TRACE
   - VCD
   - VPD
   - XML

   If any one of the above files are present, then the files will be saved in the "<username>.<uniqid>.svd" in the current directory. The simulation transcript file will not be part of this and it will be saved separately.

   The -directory switch can be specified to select an alternate source directory.

4. You will be prompted by the case submittal tool with the option to include additional files within the SVD file. The simulation transcript files cannot be automatically identified and it must be provided during this step.
5. The case submittal tool will display options on how to send the file to Synopsys.

A.8 Limitations

Enabling DEBUG or VERBOSE verbosity is an expensive operation, both in terms of runtime and disk space utilization. The following steps can be used to minimize this cost:

- Only enable the VIP instance necessary for debug. By default, the `+svt_debug_opts` command enables Debug Opts on all instances, but the '`inst`' argument can be used to select a specific instance.
- Use the `start_time` and `end_time` arguments to limit the verbosity changes to the specific time window that needs to be debugged.