Verification Continuum™

VC Verification IP
AMBA AXI
UVM User Guide

Version S-2021.06, June 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 UVM users of the VC Verification for AMBA AXI, and is for design or verification engineers who want to verify AXI operation using an UVM testbench written in SystemVerilog. Readers are assumed to be familiar with AXI, Object Oriented Programming (OOP), SystemVerilog, and Universal Verification Methodology (UVM) techniques.

Guide Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Introduction", introduces the 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 AXI VIP.
- Chapter 3, "General Concepts", introduces the AXI VIP within the UVM environment and describes the data objects and components that comprise the VIP.
- Chapter 6, "Verification Features", describes the verification features supported by AXI VIP such as, Verification Planner and Protocol Analyzer.
- Chapter 7, "Verification Topologies", describes the topologies to verify master, slave and interconnect DUT.
- Chapter 8, "Using AXI Verification IP", shows how to install and run a getting started example.
- Chapter 10, "Troubleshooting", describes the debug port.
- Appendix A, "Reporting Problems", outlines the process for working through and reporting AXI VIP 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 UVM Verification IP.

This chapter discusses the following topics:

1.1 Introduction

The AXI VIP supports verification of SoC designs that include interfaces implementing the AXI Specification. This document describes the use of this VIP in testbenches that comply with the SystemVerilog Universal Verification Methodology (UVM). 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 UVM.

1.3 References

For more information on AXI Verification IP, see 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_uvm_class_reference/html/index.html

1.4 Product Overview

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

The VIP provides an AXI System Env that contains the Master agents, Slave agents, Interconnect Env and System Monitor. The Master and Slave agents support all the functionality normally associated with active and passive UVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging and functional coverage. The System Monitor performs system-level checks across the master and slave ports of the interconnect within the system.The Interconnect Env supports transaction routing between masters and slaves. After instantiating the System Env, you can select and combine active and passive agents to create an environment that verifies AXI features in the DUT.

The Master agents, Slave agents and Interconnect Env can also be used in standalone mode, that is, they can be instantiated in the testbench directly, without the system environment.

1.5 Language and Methodology Support

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

- Languages
  - SystemVerilog
- Methodology
  - Qualified with UVM 1.1d and UVM 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 exclusive access
- AXI3 Locked access
- AXI3 Data Buses (Write strobes, narrow transfers)
- 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 Support
- ACE5 Support (Early Adopter)
- AXI4 Stream

1.6.2 Verification Features

AXI VIP currently supports the following verification functions:

- Default functional coverage (transaction, state and toggle coverage)
- Protocol checking
- 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
- Verification Planner
- Protocol Analyzer
- VC Auto Testbench
- AutoPerformance

1.6.3 Methodology Features

AXI VIP currently supports the following methodology functions:

- VIP organized as AXI System Environment, which includes set of Master agents, Slave agents, Interconnect Env and System Monitor. The Master agents, Slave agents and Interconnect Env can also be used in standalone mode
- Analysis ports for connecting Master/Slave Agents to Scoreboard, or any other component
- Callbacks for Master agents, Slave Agents and Interconnect Env
- Notifications to denote start and end of transactions

1.7 Features Not Supported

For more information on features, see *Known Issues and Limitations* section present in *AXI Verification IP Notes* chapter 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 section 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 section.

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 Synopsys 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 VIP, check the support matrix for SVT-based VIP in the following document: `amba_svt_release_notes.pdf`.

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 DESIGNWARE_HOME to the absolute path where Synopsys AXI VIP is to be 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 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.

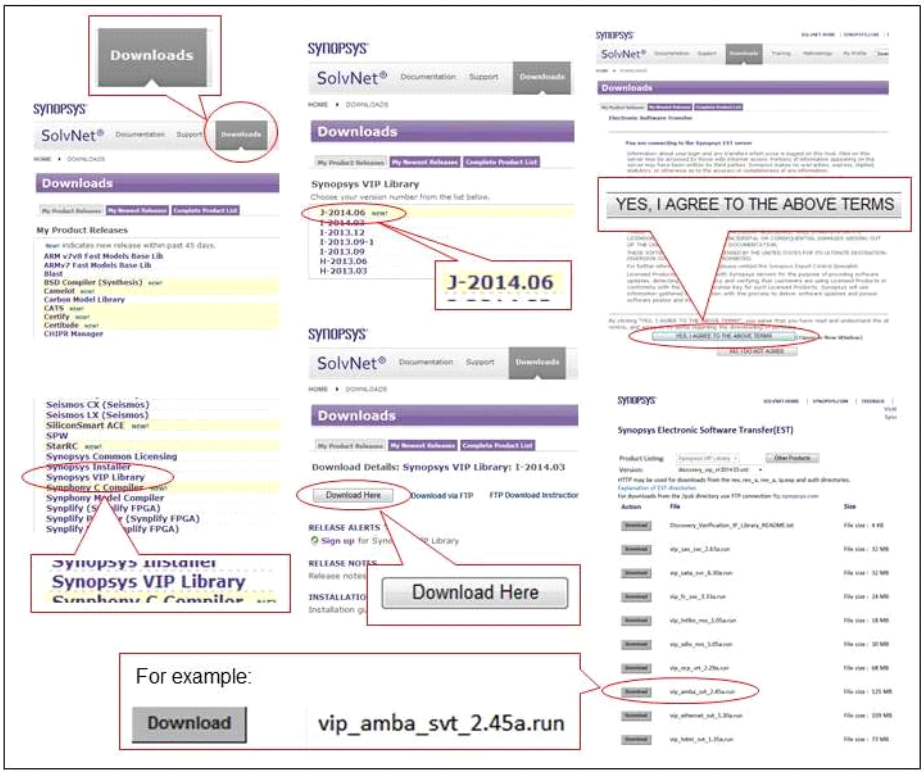
Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, see the following web page:

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)

1. Point your web browser to http://solvnet.synopsys.com.
2. Enter your Synopsys SolvNet Username and Password.
3. Click Sign In button.
4. Make the following selections on SolvNet to download the `.run` file of the VIP (See Figure 2-1).
   a. Downloads tab
   b. VC VIP Library product releases
   c. <release_version>
   d. Download Here button
   e. Yes, I Agree to the Above Terms button
   f. Download `.run` file for the VIP

Figure 2-1 SolvNet Selections for VIP Download



   g. Set the `DESIGNWARE_HOME` environment variable to a path where you want to install the VIP.
   ```
   % setenv DESIGNWARE_HOME VIP_installation_path
   ```
   h. 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

1. Follow the above instructions through the product version selection step.
2. Click the *Download via FTP* link instead of the *Download Here* button.
3. Click the *Click Here To Download* button.
4. Select the file(s) that you want to download.
5. Follow browser prompts to select a destination location.

If you are unable to download the Verification IP using above instructions, see the "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 Synopsys Verification IP 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.

Note 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 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.

Note
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.

Note
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 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 Synopsys VIP Models from a Design Directory"
- "Reporting Information About DESIGNWARE_HOME or a Design Directory"
- "The dw_vip_setup Utility"

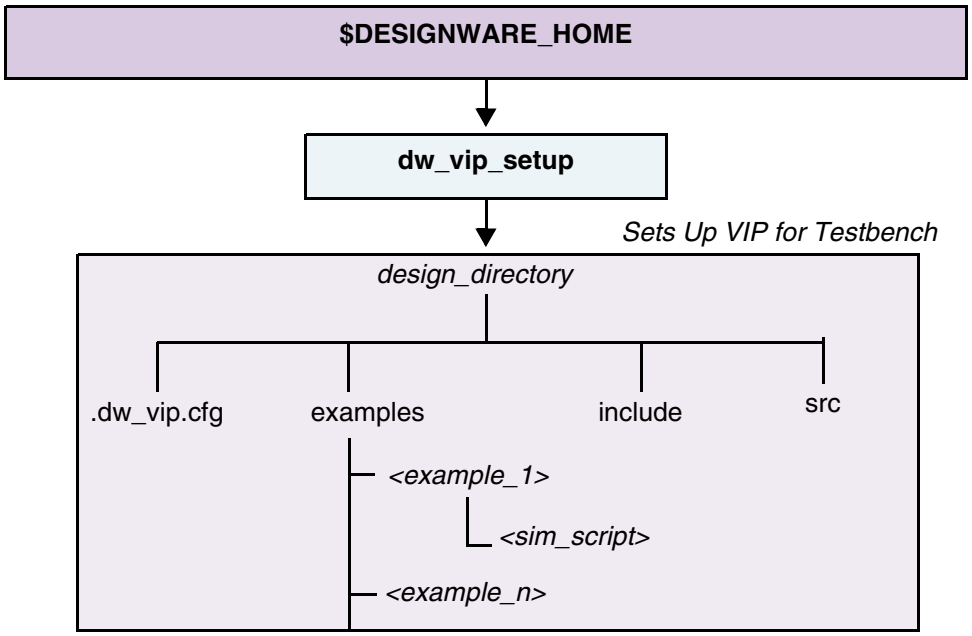
2.5.4.1 Creating a Testbench Design Directory

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

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 Synopsys VIP in your testbench because it is isolated from the DESIGNWARE_HOME installation. When you want, you can use dw_vip_setup to update the 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 VIP models. This directory is specified in simulator command lines.

src VIP-specific include files (not used by all VIPs). 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 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_system_env_svt
- axi_master_agent_svt
- axi_slave_agent_svt
- axi_interconnect_env_svt

Note

1. UVM users are expected to set the value of UVM_PACKER_MAX_BYTES macro to 1500000 on command line. If you are a UVM user, add the following to your command line: +define+UVM_PACKER_MAX_BYTES=1500000. Else, AXI VIP will issue a fatal error.
2. UVM users are required to define the UVM macro UVM_DISABLE_AUTO_ITEM_RECORDING. AXI being a pipelined protocol (that is, previous transaction does not necessarily need to complete before starting new transaction), AXI VIP handles triggering the begin/end events and transaction recording. AXI VIP does not use the UVM automatic transaction begin/end event triggering and recording feature. If UVM_DISABLE_AUTO_ITEM_RECORDING is not defined, VIP issues a FATAL message.

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 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 Synopsys 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_agent_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_agent_svt -v 3.50a -svlog
   ```

2.5.4.5 Removing Synopsys 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 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
```

Note
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_uvm_basic_sys --help

   usage: run_axi_svt_uvm_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

Note
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 AXI VIP models in a design directory.
- Adds example testbenches to a design directory, the AXI 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_agent_svt • axi_slave_agent_svt • axi_interconnect_env_svt • axi_system_env_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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-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_agent_svt • axi_slave_agent_svt • axi_interconnect_env_svt • axi_system_env_svt
-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_agent_svt • axi_slave_agent_svt • axi_interconnect_env_svt • axi_system_env_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 UVM. 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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-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.
-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_agent_svt • axi_slave_agent_svt • axi_interconnect_env_svt • axi_system_env_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`.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-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.
-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 UVM example testbenches, see the“SystemVerilog UVM 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.

Following is a code list of the includes and imports for components within amba_system_env_svt:

```
/* include uvm package before VIP includes,  If not included elsewhere*/
`include "uvm_pkg.sv"

/** Include the AMBA SVT UVM package */
`include "svt_amba.uvm.pkg"
/** Import UVM Package  */
```

```
import uvm_pkg::*;

/** Import the SVT UVM Package */
import svt_uvm_pkg::*;

/** Import the AMBA VIP */
import svt_amba_uvm_pkg::*;
```

You must also include various VIP directories on the simulator command line. Add the following switches and directories to all the 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 common for all simulators)
- `sim_run_options` (contain run time options common for all simulators)
- `vcs_build_options` (contain compile time options for VCS)
- vcs_run_options (contain run time options for VCS)
- `mti_build_options` (contain compile time options for MTI)
- `mti_run_options` (contain run time options for MTI)
- `ncv_build_options` (contain compile time options for IUS)
- `ncv_run_options` (contain run time options for IUS)

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

`vcs_build_options`:

Required: `+define+UVM_PACKER_MAX_BYTES`=1500000
Required: `+define+UVM_DISABLE_AUTO_ITEM_RECORDING`
Optional: `-timescale=1ns/1ps`
Required: `+define+SVT_<model>_INCLUDE_USER_DEFINES`

Note

`UVM_PACKER_MAX_BYTES define` needs to be set to maximum value as required by each VIP title in your testbench. For example, if VIP title 1 needs `UVM_PACKER_MAX_BYTES` to be set to 8192, and VIP title 2 needs `UVM_PACKER_MAX_BYTES` to be set to 500000, you need to set `UVM_PACKER_MAX_BYTES` to 500000.

```
vcs_run_options:
```

Required: `+UVM_TESTNAME=$scenario`

Note

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

3
General Concepts

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

3.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using constrained random verification. The resulting structure also supports directed testing.

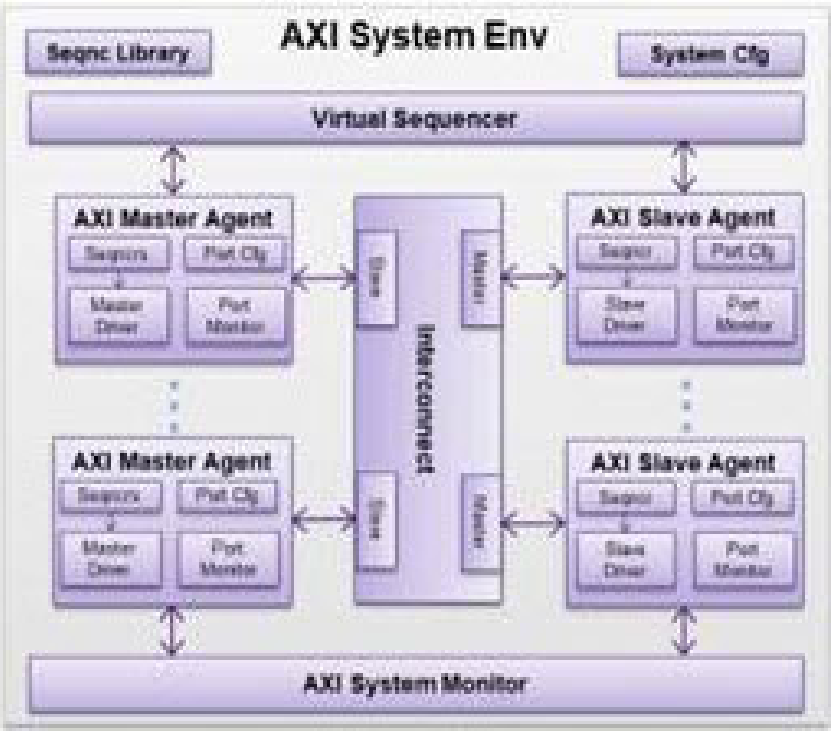
This chapter describes the usage of AXI VIP in UVM environment, and its user interface. See the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter.

This chapter assumes that you are familiar with SystemVerilog and UVM. For more information:

- For the IEEE SystemVerilog standard, see:
  - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language
- For essential guides describing UVM as it is represented in SystemVerilog, along with a class reference, see:
  - *Universal Verification Methodology (UVM) 1.0 User's Manual* at: http://www.accellera.org/.

3.2 AXI VIP in an UVM Environment

The following diagram shows the components of AXI architecture. This includes the AXI3, AXI4, ACE, and ACE5 protocols.



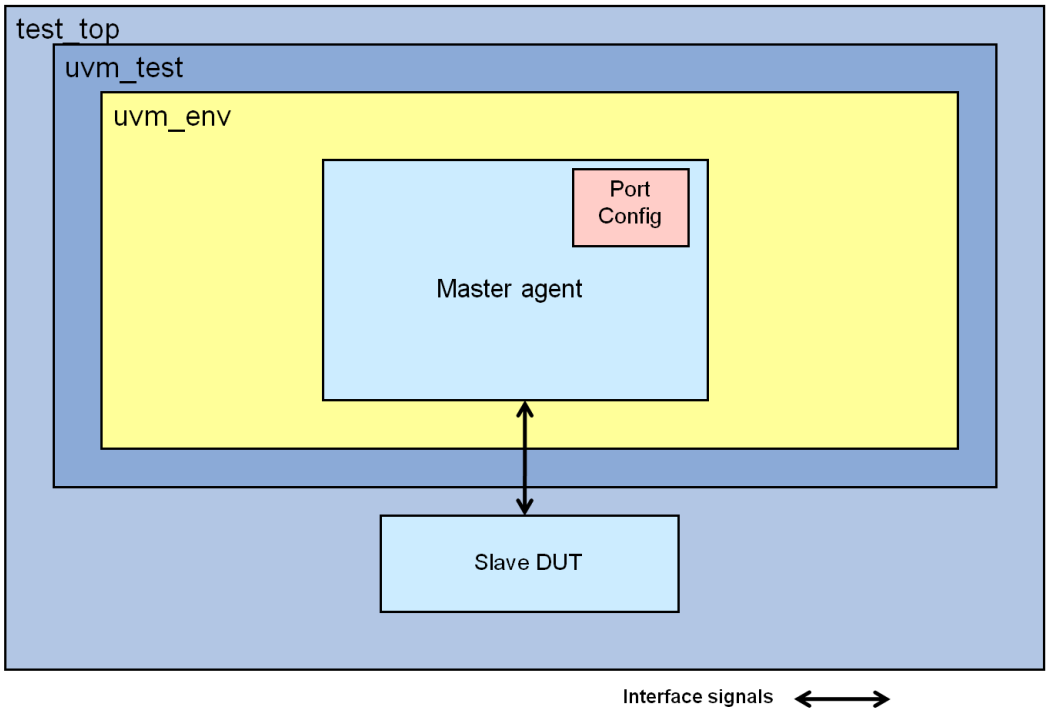
3.2.1 Master Agent

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

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

Within the Master Agent, the Master Driver gets sequences from the Master Sequencer. The Master Driver then drives the AXI transactions on the AXI port. The Master Driver and port Monitor components within Master Agent call callback methods at various phases of execution of the AXI transaction. Details of callbacks are covered in later sections. After the AXI 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 3-1 Usage With Standalone Master Agent



3.2.2 Slave Agent

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

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

In the Slave Agent, the Port Monitor samples the AXI 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 AXI bus.

Note

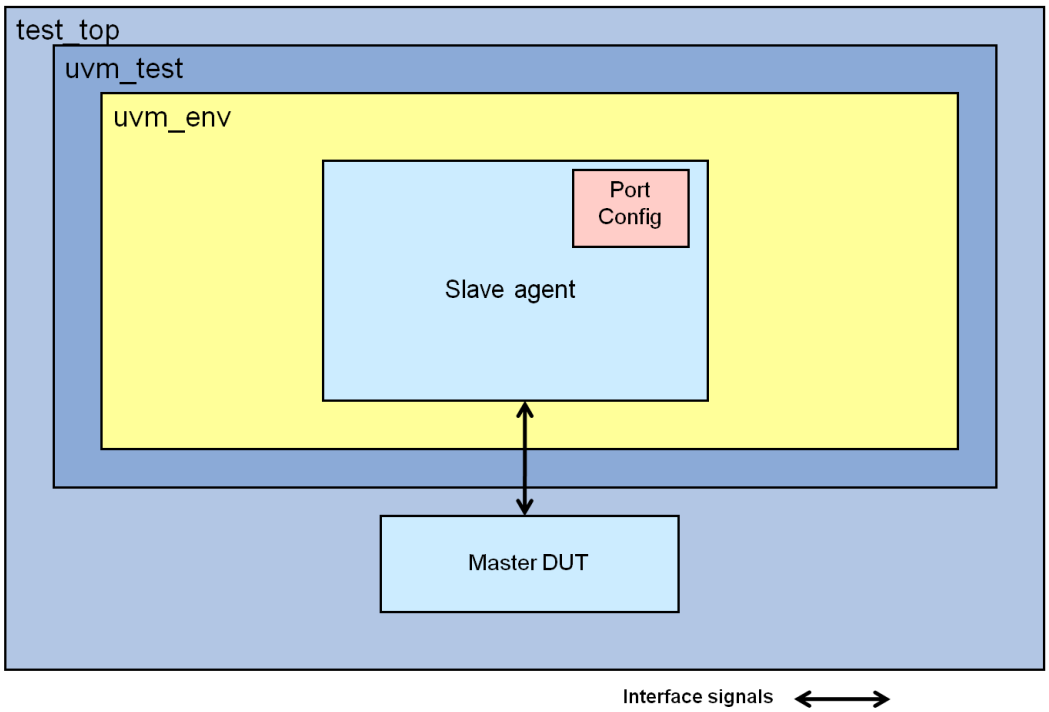
The slave driver expects the slave response sequence to,

- 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 is violated, the slave agent issues a FATAL message.

The Slave Driver and Monitor call callback methods at various phases of execution of the AXI transaction. Details of callbacks are covered in later sections. After the AXI transaction on the bus is complete, the completed sequence item is provided to the analysis port for use by the testbench.

Figure 3-2 Usage With Standalone Slave Agent



3.2.3 Stimulus Modeling

3.2.3.1 Stimulus Modeling at Master

The `svt_axi_master_transaction` is the basic master transaction description class, which depicts a transaction that has to be generated on the master interface. The class provides rich set of attributes such as address, burst type etc. The class contains transaction level attributes which can be directly mapped to interface signals (for example, addr, burst_type burst_lenth), transaction configuration attributes (for example, data_before_addr) and delay configurations on the master side (for example, addr_valid_delay, and bready_delay). The Master transaction constraint take into account the configuration set for the master agent while randomizing the class instances in order to ensure that attributes are within its configured bounds. Some of the commonly used master transaction class attributes are listed below.

- `xact_type`: Specifies the transaction types as READ or WRITE
- `addr` : Specifies the address for the transaction
- `burst_type`: Specifies burst type as INCR/FIXED/WRAP
- `burst_size`: Specifies maximum number of bytes to be transfered in each data transfer.
- `addr_valid_delay`: Specifies the number of cycles AWVALID/ARVALID signal is delayed.
- `burst_length`: Specifies the number of beats in the burst
- `data_before_addr`: Indicates that data will start before address for write transactions

- `reference_event_for_addr_valid_delay` : Specifies the reference event from which `addr_valid_delay` should begin.

For complete list of master transaction class attributes, see the class reference HTML document.

3.2.3.2 Master VIP Sequence Examples

Sample master sequence using some of the common attributes are given as follows. You can use '`svt_axi_master_base_sequence`' base class to create their sequences.

Example 1: Master Directed Sequence

This sequence allows you to write directed test cases for specific scenarios. You have to explicitly specify all relevant master transaction attributes here as the transaction object is not randomized in this sequence.

```
class axi_master_directed_sequence extends svt_axi_master_base_sequence;

  /** Parameter that controls the number of transactions that will be generated */
  rand int unsigned sequence_length = 10;

  /** Constrain the sequence length to a reasonable value */
  constraint reasonable_sequence_length {
    sequence_length <= 100;
  }

  /** UVM Object Utility macro */
  `uvm_object_utils(axi_master_directed_sequence)

  /** Class Constructor */
  function new(string name="axi_master_directed_sequence");
    super.new(name);
  endfunction

  virtual task body();
    svt_axi_master_transaction write_tran, read_tran;
    svt_configuration get_cfg;
    bit status;
    `uvm_info("body", "Entered ...", UVM_LOW)

    super.body();

    status = uvm_config_db #(int unsigned)::get(null, get_full_name(),
"sequence_length", sequence_length);
    `uvm_info("body", $sformatf("sequence_length is %0d as a result of %0s.",
sequence_length, status ? "config DB" : "randomization"), UVM_LOW);
```

```
    /** Obtain a handle to the port configuration */
          /*
```

In case of directed sequence, port configuration handle need to obtain explicitly, whereas in case of random sequence this is done implicitly while randomizing the transaction class with uvm_do macro.

```
         */
    p_sequencer.get_cfg(get_cfg);
    if (!$cast(cfg, get_cfg)) begin
      `uvm_fatal("body", "Unable to $cast the configuration to a
svt_axi_port_configuration class");
    end

    for(int i = 0; i < sequence_length; i++) begin

      /** Set up the write transaction */
      `uvm_create(write_tran)
      write_tran.port_cfg     = cfg;
      write_tran.xact_type    = svt_axi_transaction::WRITE;
      write_tran.addr         = 32'h0000_0100 | ('h100 << i);
      write_tran.burst_type   = svt_axi_transaction::INCR;
      write_tran.burst_size   = svt_axi_transaction::BURST_SIZE_32BIT;
      write_tran.atomic_type  = svt_axi_transaction::NORMAL;
      write_tran.burst_length = 4;
      write_tran.data         = new[write_tran.burst_length];
      write_tran.wstrb        = new[write_tran.burst_length];
      write_tran.data_user    = new[write_tran.burst_length];
      foreach (write_tran.data[i]) begin
        write_tran.data[i] = i;
      end
      foreach(write_tran.wstrb[i]) begin
        write_tran.wstrb[i] = 4'hf;
      end
      write_tran.wvalid_delay = new[write_tran.burst_length];
      foreach (write_tran.wvalid_delay[i]) begin
        write_tran.wvalid_delay[i]=i;
      end

      /** Send the write transaction */
      `uvm_send(write_tran)

      /** Wait for the write transaction to complete */
      get_response(rsp);
```

```
      /** Set up the read transaction */
      `uvm_create(read_tran)
      read_tran.port_cfg     = cfg;
      read_tran.xact_type    = svt_axi_transaction::READ;
      read_tran.addr         = 32'h0000_0100 | ('h100 << i);
      read_tran.burst_type   = svt_axi_transaction::INCR;
      read_tran.burst_size   = svt_axi_transaction::BURST_SIZE_32BIT;
      read_tran.atomic_type  = svt_axi_transaction::NORMAL;
      read_tran.burst_length = 4;
      read_tran.rresp        = new[read_tran.burst_length];
      read_tran.data         = new[read_tran.burst_length];
      read_tran.rready_delay = new[read_tran.burst_length];
      read_tran.data_user    = new[read_tran.burst_length];
      foreach (read_tran.rready_delay[i]) begin
        read_tran.rready_delay[i]=i;
      end

      /** Send the read transaction */
      `uvm_send(read_tran)

      /** Wait for the read transaction to complete */
      get_response(rsp);

    end

    `uvm_info("body", "Exiting...", UVM_LOW)
  endtask: body

endclass: axi_master_directed_sequence
```

Master agent has is_valid check, which checks the transaction programming against protocol specification. So, if transaction attributes are programmed incorrectly, then is_valid check will issue a UVM_WARNING indicating the issue and the test will fail with UVM_FATAL due to 'is_valid' check failure.

Example 1:

The following UVM_WARNING and UVM_FATAL are reported when the interface type is configured as AXI3 and the burst_length is 17 for an INCR burst as this is not allowed as per AXI3 protocol.

UVM_WARNING @ 1025000: reporter [is_valid] Invalid burst_length of 17 provided, must be inside { 1:16 } based on interface type(AXI3), xact_type(WRITE) and `SVT_AXI3_MAX_BURST_LENGTH(16)

UVM_FATAL @ 1025000: uvm_test_top.env.axi_system_env.master[0] [add_to_master_active] {OBJECT_NUM(100000) PORT_ID(0) TYPE(WRITE) ID(0) ADDR(100)} Master Transaction is_valid check failed.

Example 2: Master Random Sequence

This sequence randomizes the transaction class attributes as per the master transaction class constrains. You can specify inline constraints in the sequence if required. Remaining attributes will be assigned with applicable values as per the protocol constraints.

```
class axi_master_wr_rd_sequence extends svt_axi_master_base_sequence;

  /** Parameter that controls the number of transactions that will be generated */
  rand int unsigned sequence_length = 10;

  /** Constrain the sequence length to a reasonable value */
  constraint reasonable_sequence_length {
    sequence_length <= 100;
  }

  /** UVM Object Utility macro */
  `uvm_object_utils(axi_master_wr_rd_sequence)

  /** Class Constructor */
  function new(string name="axi_master_wr_rd_sequence");
    super.new(name);
  endfunction

  virtual task body();
    bit status;
    `uvm_info("body", "Entered ...", UVM_LOW)
    super.body();

    status = uvm_config_db #(int unsigned)::get(null, get_full_name(),
"sequence_length", sequence_length);
    `uvm_info("body", $sformatf("sequence_length is %0d as a result of %0s.",
sequence_length, status ? "config DB" : "randomization"), UVM_LOW);

    repeat (sequence_length) begin
      `uvm_do_with(req,
        {
          xact_type == svt_axi_transaction::WRITE;
          data_before_addr == 0;
        })
```

```
      `uvm_do_with(req,
        {
          xact_type == svt_axi_transaction::READ;
          data_before_addr == 0;
        })
    end

    `uvm_info("body", "Exiting...", UVM_LOW)
  endtask: body

endclass: axi_master_wr_rd_sequence
```

You can use object_id field of transaction for transaction tracking. This field is set by VIP for each transaction. Read transaction object_id start with 0 whereas as write transaction object_id start with 100000. `object_id` will be seen as OBJECT_NUM in VIP log messages as shown in the example. You can track status related information of a specific transaction using the object_id/OBJECT_NUM.

Example:

UVM_INFO @ 1025000: uvm_test_top.env.axi_system_env.master[0] [send_write_addr] {OBJECT_NUM(100000) PORT_ID(0) TYPE(WRITE) ID(0) ADDR(100)} Driving write address channel signals

UVM_INFO @ 15125000: uvm_test_top.env.axi_system_env.master[0] [send_read_addr] {OBJECT_NUM(0) PORT_ID(0) TYPE(READ) ID(0) ADDR(100)} Driving read address channel signals

UVM_INFO @ 15175000: uvm_test_top.env.axi_system_env.slave[0] [sample_read_addr_chan_signals] {OBJECT_NUM(0) PORT_ID(0) TYPE(READ) ID(0) ADDR(100)} Received read address

3.2.3.3 Stimulus Modeling at Slave

The class that describes slave transactions is 'svt_axi_slave_transaction'. Slave generates an object of svt_axi_slave_transaction when a command is received for which response has to be generated. The object defines complete transaction, that is, the command received along with the default response that Slave is configured to generate. Slave transaction class provides attributes to configure the slave response (for example, rresp[], bresp and data[]) and the delay applicable for the slave response (for example, bvalid_delay and addr_ready_delay). Example for slave transaction class attributes are

- `rresp[]`: This is a dynamic array which configures slave read response for each beat
- `bresp`: Configures slave write response
- `bvalid_delay`: Configures slave write response delay
- `addr_ready_delay`: Configures the delay on `arready` assertion.
- `enable_interleave`: Enables/ Disables interleave
- `Interleave_pattern`: Defines the pattern for interleaving

VIP allows you to specify distribution weights for delays with attributes. Attributes `ZERO_DELAY_wt`, `SHORT_DELAY_wt` and `LONG_DELAY_wt` defines the weight used to control distribution of zero delay, short delay and long delay respectively. Example for associated delay is ready signal assertion delay.

For usage details, see `axi_slave_random_response_sequence` sequence.

For complete list attributes and details, see the `svt_axi_slave_transaction` in class reference HTML.

The slave monitor of the AXI Slave agent contains a blocking peek port named response_request_imp which gets updated with any requests that target that slave. The slave agent connects this peek port to the slave sequencer which has a blocking peek port named response_request_port. Slave sequences must obtain a handle to the request through this port in the sequence, fill in the response information, and then submit this response to the driver through the normal means of sequencer/driver communication.

3.2.3.4 Slave VIP Example Sequences

This section demonstrates response generation with slave sequences. These slave sequences are extended from `svt_axi_slave_base_sequence`.

Example1: Random Response Sequence

This sequence generates random response from the slave. For write transactions, data is not written to the slave memory and for read, the response and data are random.

```
class axi_slave_random_response_sequence extends svt_axi_slave_base_sequence;

  svt_axi_slave_transaction resp_req;

  /** UVM Object Utility macro */
  `uvm_object_utils(axi_slave_random_response_sequence)

  /** Class Constructor */
  function new(string name="axi_slave_random_response_sequence");
    super.new(name);
  endfunction

  virtual task body();
    `uvm_info("body", "Entered ...", UVM_LOW)

    forever begin

/**
       * Get the response request from the slave sequencer. The response request is
       * provided to the slave sequencer by the slave port monitor, through
       * TLM port.
       */

      p_sequencer.response_request_port.peek(resp_req);

      $cast(req,resp_req);
```

```
        req.ZERO_DELAY_wt = 10;

        req.SHORT_DELAY_wt = 20;

        req.LONG_DELAY_wt = 100;

        * Demonstration of response randomization with constraints.
        */
       `uvm_rand_send_with(req,
          {
            foreach(rresp[i]) {
             rresp[i] inside { svt_axi_transaction::SLVERR, svt_axi_transaction::OKAY };
            }
            bresp inside { svt_axi_transaction::SLVERR,svt_axi_transaction::OKAY };
          })

    end

    `uvm_info("body", "Exiting...", UVM_LOW)
  endtask: body

endclass: axi_slave_random_response_sequence
```

Example2: Memory Sequence

This sequence makes use of slave VIP memory. Data is written to and read back from the actual slave memory using the methods `put_write_transaction_data_to_mem` and `get_read_data_from_mem_to_zltransaction` respectively. These methods are defined in the parent sequence - `svt_axi_slave_base_sequence`.

```
class axi_slave_mem_response_sequence extends svt_axi_slave_base_sequence;

  svt_axi_slave_transaction req_resp;

  /** UVM Object Utility macro */
  `uvm_object_utils(axi_slave_mem_response_sequence)

  /** Class Constructor */
  function new(string name="axi_slave_mem_response_sequence");
    super.new(name);
  endfunction
```

```
  virtual task body();
    integer status;
    svt_configuration get_cfg;

    `uvm_info("body", "Entered ...", UVM_LOW)

    p_sequencer.get_cfg(get_cfg);
    if (!$cast(cfg, get_cfg)) begin
      `uvm_fatal("body", "Unable to $cast the configuration to a
svt_axi_port_configuration class");
    end

    forever begin
      /**
       * Get the response request from the slave sequencer. The response request is
       * provided to the slave sequencer by the slave port monitor, through
       * TLM port.
       */
      p_sequencer.response_request_port.peek(req_resp);

      /**
       * Randomize the response and delays
       */
      status=req_resp.randomize with {
        bresp == svt_axi_slave_transaction::OKAY;
        foreach (rresp[index])  {
          rresp[index] == svt_axi_slave_transaction::OKAY;
          }
       };
       if(!status)
        `uvm_fatal("body","Unable to randomize a response")

      /**
       * If write transaction, write data into slave built-in memory, else get
       * data from slave built-in memory
       */
      if(req_resp.xact_type == svt_axi_slave_transaction::WRITE) begin
        put_write_transaction_data_to_mem(req_resp);
      end
      else begin
        get_read_data_from_mem_to_transaction(req_resp);
```

```
        end

        $cast(req,req_resp);

        /**
         * send to driver
         */
        `uvm_send(req)

      end

      `uvm_info("body", "Exiting...", UVM_LOW)
    endtask: body

endclass: axi_slave_mem_response_sequence
```

The `object_id` and OBJECT_NUM can be used for slave transaction tracking also.

3.2.3.5 Interconnect VIP Transaction classes

Master and slave ports of interconnect VIP also uses transaction classes. The transaction class used by interconnect VIP slave ports is '`svt_axi_ic_slave_transaction`' whereas master ports uses `svt_axi_master_transaction` class itself.

3.2.3.6 Overriding Master and Slave Transaction Classes

You can override master and slave transaction base classes from test or env class in the build_phase with custom classes with additional user constraints. Overriding can be done globally and locally using uvm methods 'set_type_override_by_type' and by 'set_inst_override_by_type' respectively. The following are the examples:

Example:

```
virtual function void build_phase (uvm_phase phase);
    `uvm_info("build_phase", "Entered...",UVM_LOW)
    super.build_phase(phase);
                ……….
/*Instance Override*/
set_inst_override_by_type( "env.axi_system_env.slave[0].*"
      , svt_axi_slave_transaction::get_type()
      , cust_svt_axi_slave_transaction::get_type() );
/*Type Override*/
set_type_override_by_type (svt_axi_master_transaction::get_type(),
cust_svt_axi_master_transaction::get_type());
set_type_override_by_type (svt_axi_slave_transaction::get_type(),
cust_svt_axi_slave_transaction::get_type());
```

```
set_type_override_by_type(svt_axi_ic_slave_transaction:get_type(),
cust_slave_ic_transaction::get_type());
……..
endfunction
```

Master transaction and Slave response will not start until a reset is observed by Master and Slave respectively.

3.2.3.7 Setting Valid-Ready Delay Values for Master, Slave, and Interconnect

Three types of delays ZERO (0), SHORT (1:max-1) and LONG (max) are used by VIP for valid-ready delay, and they are applied as weighted constraints in VIP transactions using their respective ZERO_DELAY/SHORT_DELAY/LONG_DELAY_wt attributes. All such constraints can be found in HTML document with reasonable.*delay `reg_exp` search.

For example, the following are the two-step process on how you can set all delays to '0' value, leveraging VIP transaction provided attribute `ZERO_DELAY_wt`.

Step 1:

Create custom factory transactions for applicable VIP components in your testbench with ZERO_DELAY_wt=100

```
//Custom factory master transaction , applicable for master agent
cust_svt_axi_master_transaction extends svt_axi_master_transaction;
  `uvm_object_utils(cust_svt_axi_master_transaction)

  function new (string name = "cust_svt_axi_master_transaction");
    super.new(name);
    this.ZERO_DELAY_wt  = 100;
    this.SHORT_DELAY_wt =   0;
    this.LONG_DELAY_wt  =   0;
  endfunction: new
endclass: cust_svt_axi_master_transaction

//Custom factory slave transaction , applicable for slave agent
class cust_svt_axi_slave_transaction extends svt_axi_slave_transaction;
  `uvm_object_utils(cust_svt_axi_slave_transaction)

  function new (string name = "cust_svt_axi_slave_transaction");
    super.new(name);
    this.ZERO_DELAY_wt  = 100;
    this.SHORT_DELAY_wt =   0;
    this.LONG_DELAY_wt  =   0;
  endfunction: new

     endclass: cust_svt_axi_slave_transaction

//Custom factory Interconnect transaction , applicable for interconnect agent
class cust_svt_axi_ic_slave_transaction extends svt_axi_ic_slave_transaction;
  `uvm_object_utils(cust_svt_axi_ic_slave_transaction)

  function new (string name = "cust_svt_axi_ic_slave_transaction");
    super.new(name);
    this.ZERO_DELAY_wt  = 100;
```

```
    this.SHORT_DELAY_wt =   0;
    this.LONG_DELAY_wt  =   0;
  endfunction: new

endclass: cust_svt_axi_ic_slave_transaction
```

Step 2:

Do three factory type-wide replacements in your env/test (or you can apply this to specific instances as applicable)

```
set_type_override_by_type(svt_axi_master_transaction::get_type(),
cust_svt_axi_master_transaction::get_type());
    set_type_override_by_type(svt_axi_slave_transaction::get_type(),
cust_svt_axi_slave_transaction::get_type());
    set_type_override_by_type(svt_axi_ic_slave_transaction::get_type(),
cust_svt_axi_ic_slave_transaction::get_type());
```

3.2.4 Slave Memory

AXI VIP provides slave memory represented by class svt_mem. Slave memory is instantiated in slave agent. In passive mode, the slave agent keeps the memory updated based on the observed data on the bus. This enables the system-level checks in the passive mode. In active mode, the slave memory is updated by the slave sequence.

See the slave sequence `svt_axi_slave_base_sequence` in file `$DESIGNWARE_HOME/vip/svt/amba_svt/latest/axi_slave_agent_svt/sverilog/src/vcs/svt_axi_slave_ sequence_collection.svp`. A reference to the slave memory instantiated in the slave agent is provided in the slave sequence. Using the API of this slave memory, you can read or write into the slave memory through this base sequence, or sequence extended from this base sequence.

For details on `svt_mem` and `svt_axi_slave_base_sequence`, see the AXI SVT class reference HTML documentation.

3.2.4.1 Slave Memory Modeling

3.2.4.2 AXI Slave Memory Modeling

The memory in AXI slave VIP is modeled using the class "`svt_mem`". Slave memory is instantiated in slave agent.

In passive mode, the slave agent keeps the memory updated based on the observed data on the bus. For write transactions, the memory is updated based on the observed write data. For read transactions, the memory is updated based on the read data. The configuration `svt_axi_port_configuration::memory_update_for_read_xact_enable` controls the vip behavior. The default value for this configuration is 1, so the memory will be updated for the read transactions it observes. This doesn't enable system check. At any RTL-to-RTL interface, if passive slave VIP is connected, then memory update from the observed traffic facilitate system level data integrity checks across such port.

3.2.4.3 Front Door Access

In active mode, the slave memory is updated by the slave sequence whenever it observes a transaction on the interface. In passive mode, the slave memory is updated by the monitor based on the observed write/read transactions on the interface. This is referred as front door access of slave memory. See the slave sequence `svt_axi_slave_base_sequence` in the file:

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/axi_slave_agent_svt/sverilog/src/vcs/svt_axi_slave_ sequence_collection.svp. A reference to the slave memory instantiated in the slave agent is provided in the slave sequence. Using the API of this slave memory, you can read or write into the slave memory through this base sequence, or sequence extended from this base sequence.

For more details on svt_mem and svt_axi_slave_base_sequencet, see AXI SVT class reference HTML documentation. The usage of these APIs is shown in the axi_slave_mem_response_sequence that is part of the vip example.

Example:

The body() of a typical slave sequence looks like below:

```
  virtual task body();
    integer status;
    svt_configuration get_cfg;

    `uvm_info("body", "Entered ...", UVM_LOW)

    p_sequencer.get_cfg(get_cfg);
    if (!$cast(cfg, get_cfg)) begin
      `uvm_fatal("body", "Unable to $cast the configuration to a
svt_axi_port_configuration class");
    end

    // consumes responses sent by driver
    sink_responses();

    forever begin
      /**
       * Get the response request from the slave sequencer. The response request is
       * provided to the slave sequencer by the slave port monitor, through
       * TLM port.
       */
      p_sequencer.response_request_port.peek(req_resp);

      /**
       * Randomize the response and delays
       */
      status=req_resp.randomize with {
        bresp == svt_axi_slave_transaction::OKAY;
        foreach (rresp[index])  {
          rresp[index] == svt_axi_slave_transaction::OKAY;
          }
```

```
        };
        if(!status)
         `uvm_fatal("body","Unable to randomize a response")

       /**
        * If write transaction, write data into slave built-in memory, else get
        * data from slave built-in memory
        */
       if(req_resp.xact_type == svt_axi_slave_transaction::WRITE) begin
         put_write_transaction_data_to_mem(req_resp);
       end
       else begin
         get_read_data_from_mem_to_transaction(req_resp);
       end

       $cast(req,req_resp);

       /**
        * send to driver
        */
       `uvm_send(req)

     end

     `uvm_info("body", "Exiting...", UVM_LOW)
  endtask: body
```

The memory is updated from the sequence in the following way:

```
       if(req_resp.xact_type == svt_axi_slave_transaction::WRITE) begin
         put_write_transaction_data_to_mem(req_resp);
       end
       else begin
         get_read_data_from_mem_to_transaction(req_resp);
       end
```

These APIs are part of the slave agent and can be copied over and modified as per the custom requirement. The code related to these APIs is not protected.

axi_slave_random_response_sequence shown in the vip example does not access the slave memory. The response and data is completely random.

3.2.4.4 Backdoor Access

The slave memory can also be accessed using the back door APIs (not through axi interface) from the testbench. The important APIs are:

- `read()`
- `write()`
- `set_meminit()`
- `load_mem()`
- `save_mem()`
- `clear()`

See the HTML doc for details. The following are some articles related to back door memory access:

https://solvnet.synopsys.com/retrieve/2290799.html

https://solvnet.synopsys.com/retrieve/2214179.html

https://solvnet.synopsys.com/retrieve/2042786.html

https://solvnet.synopsys.com/retrieve/034476.html

While doing the backdoor access of memory using read() and write() methods, the number of bytes accesses will be equal to the data_width configured on the slave agent. For example, if the data_width is 32 bit (i.e., 4 bytes), then the address provided to these methods should be aligned to 4 bytes i.e., 0, 4, 8, 12 etc... If an unaligned (intermediate) address is provided, these methods internally align the address and then access the bytes in the memory.

The slave VIP has methods `svt_axi_slave_agent::write_byte() and svt_axi_slave_agent::read_byte()` that can be used to access a single byte of data in slave memory.

You can configure the memory to return some predefined values on the addresses that were not previously written. This can be achieve using the members:

- `meminit`
- `set_meminit()`

These take the enum values of type meminit_enum as inputs. It has the following values. Please see the html class reference doc for details:

- UNKNOWNS
- ZEROES
- ONES
- ADDRESS
- VALUE
- INCR
- DECR
- USER_PATTERN

3.2.4.5 Configuring Slave Memory Address Map

To configure the address ranges of slave memory, you need to use the method svt_axi_system_configuration::set_addr_range().

Example: `set_addr_range(0, 32'h0000_0000, 32'h0000_ffff);  //this will set the address range of slave[0].`

It is possible to have discontinuous address ranges within a single slave vip. This can be done by calling the set_addr_range() multiple times, like:

```
set_addr_range(0, 32'h0000_0000, 32'h0000_ffff);
set_addr_range(0, 32'h0004_0000, 32'h0004_ffff);
set_addr_range(0, 32'h0006_0000, 32'h0006_ffff);
```

It is possible to have a shared memory across multiple slave vips. This can be achieved by creating an object of `svt_mem` in the testbench and setting this as slave memory for all the slave vips. This is demonstrated in the VIP example tb_amba_svt_uvm_basic_sys. Check the description of the member `svt_axi_system_configuration::allow_slaves_with_overlapping_addr` which has the details.

You can also specify rules for masters accessing shared memory through specific slave interfaces. Please check the article:

https://solvnet.synopsys.com/retrieve/2216520.html

3.2.4.6 Complex Memory Map Feature in AXI VIP

Usage Scenario:

AXI interconnects follow certain rules to route the transactions from master to slave. These rules must be provided to VIP configuration, so that the VIP operates based on the interconnect behavior. The main aspects that needs to be considered are:

- How does the interconnect route transactions from master to slave?
- Will there be any transactions that gets terminated within interconnect (for example, interconnect register accesses)?
- Does the interconnect translate the address? This means that the address seen on master will be different from the address seen on slave.
- Does interconnect VIP behavior change dynamically during the simulation with respect to all the above aspects, based on the state of simulation?

For simple interconnect behaviors, where there is a fixed addr ranges for each slave and the routing of transactions is only based on the address and there is no address translation, then `svt_axi_system_configuration::set_addr_range()` can be used to configure the VIP with this information.

For complex interconnects, where the above described aspects need to be modeled dynamically, the complex memory map feature needs to be used.

VIP Components Impacted by this Feature:

Axi system monitor: data_integrity_check, master_slave_xact_data_integrity_check, slave_transaction_routing_check are the main checks impacted by this feature.

Axi interconnect vip: the behavior of interconnect vip wrt routing the transactions, addr translations are impacted by this feature.

Use Model:

1. Set `svt_axi_system_configuration::enable_complex_memory_map`=1;

2. In the configuration class extended from svt_axi_system_configuration, user needs to implement 2 functions:

```
virtual function bit get_dest_global_addr_from_master_addr(
   input  int master_idx,
   input  bit [`SVT_AXI_MAX_ADDR_WIDTH-1:0] master_addr,
   input  bit[`SVT_AMBA_MEM_MODE_WIDTH-1:0] mem_mode = 0,
   input  string requester_name = "",
    input  bit ignore_unmapped_addr = 0,
    output bit is_register_addr_space,
    output bit [`SVT_AXI_MAX_ADDR_WIDTH-1:0] global_addr,
    input svt_axi_transaction xact);
```

This function needs to be implemented only if there is a requirement to perform a transformation of the master address to some global address. In many cases, this function need not be implemented. If not implemented, global address is same as the master address.

```
virtual function bit get_dest_slave_addr_from_global_addr(
    input  bit [`SVT_AXI_MAX_ADDR_WIDTH-1:0] global_addr,
    input  bit[`SVT_AMBA_MEM_MODE_WIDTH-1:0] mem_mode = 0,
    input  string requester_name = "",
    input  bit ignore_unmapped_addr = 0,
    output bit is_register_addr_space,
    output int slave_port_ids[$],
    output bit [`SVT_AXI_MAX_ADDR_WIDTH-1:0] slave_addr,
    input svt_axi_transaction xact);
```

This function gets as input the global_addr computed from previous function (if previous function is not implemented, global_addr will be equal to master_addr), requester_name (uvm tb hierarchy of the master agent from which this xact originated), master transaction handle.

Based on these inputs, this function must compute the slave addr, the slave port id in that system where the xact will be routed to, whether the addr falls in register space. These are provided as output arguments. This function returns 1 only if it finds a slave port in this system where this master xact will be routed to. If not, this function returns 0 (indicates that the xact will not be routed to any of the slave in this system). If this is a register config xact (terminates within the interconnect), then is_register_addr_space output argument must be set to 1. The requester_name can also be set to an unique string using `svt_axi_port_configuration::source_requester_name`.

These two set of functions are executed by the AXI system monitor, interconnect VIP for every master transaction initiated in the system to get the slave transaction routing info and the slave address that will be seen. The description of each argument for these functions can be found in the HTML class reference document.

3. When the complex memory map feature is used, the `svt_axi_system_configuration::set_addr_range()` is not used to specify the addr map information.

Example implementation of the complex memory map API:

The following is an example, and must be implemented based on your DUT interconnect behavior.

```
  function bit get_dest_slave_addr_from_global_addr (input bit
[`SVT_AXI_MAX_ADDR_WIDTH-1:0] global_addr,
                                                     input bit [`SVT_AMBA_MEM_MODE_WIDTH-
1:0] mem_mode = 0,
                                                        input string requester_name ="",
                                                     input bit ignore_unmapped_addr = 0,
                                                     output bit is_register_addr_space ,
                                                         output int slave_port_ids [$],
                                                     output bit [`SVT_AXI_MAX_ADDR_WIDTH-
1:0] slave_addr ,
                                                         input svt_axi_transaction xact);
      begin

        //the transactions are always routed to slave port 0. The below statement
indicates the routing info of interconnect.
        slave_port_ids[0]=0;

        //no addr translation. The below statement indicates any addr translation
performed by interconnect.
        slave_addr = global_addr; //note that global_addr will be tagged with the non-
secure bit if address tagging is enabled.

        //return 1
        get_dest_slave_addr_from_global_addr=1;

        //the below items can be used when applicable. In this example, they are not
applicable
        /*
        * if(xact.port_cfg.port_id == 0 || xact.port_cfg.port_id == 1) begin
//transactions from master[0] and master[1] will be routed to slave 0
           slave_port_ids[0]=0;
        end
        else if(xact.port_cfg.port_id == 2 || xact.port_cfg.port_id == 3) begin
//transactions from master[2] and master[3] will be routed to slave 1
          slave_port_ids[0]=1;
        end

        //register transactions that terminates within the interconnect have to be
specified using:
        if(global_addr>=reg_space_start_addr && global_addr<=reg_space_end_addr) begin
          is_register_addr_space=1;
          slave_port_ids[0]=-1;
```

```
        get_dest_slave_addr_from_global_addr=1;
      end
      */
    end
  endfunction
```

This is also demonstrated in the VIP example `tb_axi_svt_uvm_basic_sys`

3.2.4.7 FIFO Memory

The slave agent has a FIFO memory that can be used for transactions of FIXED burst type. The FIFO memory is represented by class svt_axi_fifo_mem. FIFO memory is instantiated in the slave agent as follows:

```
svt_axi_fifo_mem fifo_mem[];
```

This FIFO is configured based on the port configuration parameters svt_axi_port_configuration::num_fifo_mem and svt_axi_port_configuration::fifo_mem_addresses[]. By default, svt_mem will be used for FIXED burst_type also.

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 slave vip. 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.

The below article shows how to customize the slave vip behavior for accessing the FIFO memory. It also has an example for accessing FIFO memory for INCR transactions of burst_length=1.

https://solvnet.synopsys.com/retrieve/1512706.html

3.2.4.8 AXI System Monitor Considerations

The axi system monitor `data_integrity_checks` are based on the slave memory updates. So you need to use the `slave_mem_response_sequence` on the active slave vips. A passive slave vip needs to be connected to the interfaces where there is a DUT. The memory in the passive slave vip will be updated based on the activity observed on the interface.

3.2.5 FIFO Memory

AXI VIP provides FIFO memory represented by class `svt_axi_fifo_mem`. FIFO memory is instantiated in the slave agent 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.

3.2.6 Interconnect Env

The Interconnect Env routes the AXI transactions between multiple masters and slaves. The Interconnect Env contains configurable number of master and slave ports. The number of master and slave ports of the Interconnect Env can be controlled through interconnect configuration. The Interconnect Env routes the transactions from masters to slaves based on address map. Multiple address ranges can be specified for a single slave. The Interconnect Env can be configured to operate in active mode and passive mode.

In the Interconnect Env component, the ports which are connected to master components are referred to as Interconnect Slave ports and ports which are connected to slave components are referred to as Interconnect Master ports. More details on Interconnect master and slave ports are provided in the following sections.

The Interconnect VIP waits for the entire write data (from the master) to arrive before it initiates a transaction on the slave. The Interconnect VIP is a functionally ideal interconnect which comprises of several AXI masters and slaves. It is not designed to be the most efficient in terms of bandwidth utilization, performance and hence forth (which obviously an RTL is designed for). So for a write transaction, the Interconnect waits for the `wlast` to arrive before it starts sending the corresponding `awvalid` to the slave.

- For more information on Interconnect features related to ACE protocol, see "Support for ACE protocol in AXI Interconnect Env".

3.2.6.1 Interconnect Env Master Ports

The master ports of the Interconnect Env drive the slave components in the system. The master ports within the interconnect Env are represented by Interconnect Master Agent class `svt_axi_ic_master_agent`. The Interconnect Master Agent is configured using a port configuration, which is available in the interconnect configuration.

The Driver within the Interconnect Master Agent drives the AXI transactions towards the slave component connected to the interconnect master port. The Driver and port Monitor components within Interconnect Master Agent call callback methods at various phases of execution of the AXI transaction. Details of callbacks are covered in later sections. At the end of each transaction on the Interconnect Master port, the port monitor within the Interconnect Master Agent provides the completed transaction object from its analysis port, in active and passive mode.

3.2.6.2 Interconnect Env Slave Ports

The slave ports of the interconnect Env are driven by the master components in the system. The slave ports within the interconnect Env are represented by Interconnect Slave Agent class `svt_axi_ic_slave_agent`.

The Interconnect Slave Agent is configured using a port configuration, which is available in the interconnect configuration.

The Driver within the Interconnect Slave Agent responds to the AXI transactions driven by the master component connected to the interconnect slave port. The Driver and port Monitor components within Interconnect Slave Agent call callback methods at various phases of execution of the AXI transaction. Details of callbacks are covered in later sections. At the end of each transaction on the Interconnect Slave port, the port monitor within the Interconnect Slave Agent provides the completed transaction object from its analysis port, in active and passive mode.

3.2.6.3 Connecting Interconnect Env to the DUT

The number of master and slave ports of the Interconnect Env is configured using configuration members `svt_axi_interconnect_configuration::num_ic_master_ports` and `svt_axi_interconnect_configuration::num_ic_slave_ports` respectively. If the System Env also contains master and slave VIP components in addition to the Interconnect Env, these master and slave VIP components get automatically connected to the lowest port indices of the Interconnect Env. The number of master and slave VIP components in system Env is configured using configuration members `svt_axi_system_configuration::num_masters` and `svt_axi_system_configuration::num_slaves` respectively.

For example, if `svt_axi_interconnect_configuration::num_ic_slave_ports = 3`, and svt_axi_system_configuration::num_masters = 2, then master VIP components would automatically connect to slave port 0 and 1 of the Interconnect Env component. Slave port 2 of Interconnect Env can be connected to Master DUT. In such a case, the port monitor in Slave port 2 of the Interconnect Env will continue to carry out the passive functionality like protocol checking.

It is recommended to configure the number of master or slave VIP components in the System Env to match the number of slave or master ports of Interconnect Env. Then, for the Interconnect Env ports which are expected to be connected to the DUT, configure the corresponding Master or Slave VIP components in passive mode. That way, even if you replace Interconnect Env with Interconnect RTL, the passive monitors would continue to function.

For example, if Interconnect Env has 3 slave ports (port 0,1,2), and you need Master VIP components to drive port 0 and 1, and Master DUT to drive port 2.

In this case, configure the number of masters in System Env to 3 (`svt_axi_system_configuration::num_masters = 3`). Configure Master VIP 0 and Master VIP 1 as active, and Master VIP 2 as passive (as Master DUT would drive this port).

3.2.6.4 Configuration Consistency Checks

When the VIP is configured to use the interconnect model by setting the configuration parameter `svt_axi_system_configuration::use_interconnect`, the configuration of master or slave VIP components must match configuration of interconnect ports to which they connect. The VIP does a consistency check between the port configuration of master or slave VIP components and corresponding interconnect port configuration. The consistency checks ensure that:

`sys_cfg.master_cfg[<master_num>].<port_config_param>` matches
`sys_cfg.ic_cfg.slave_cfg[<master_num>].<port_config_param>`

`sys_cfg.slave_cfg[<slave_num>].<port_config_param>` matches
`sys_cfg.ic_cfg.master_cfg[<slave_num>].<port_config_param>`

VIP checks the consistency of below port configuration parameters between master or slave VIP components and corresponding Interconnect port configurations:

```
addr_width
addr_user_width
data_width
data_user_width
resp_user_width
snoop_data_width
wysiwyg_enable
use_separate_rd_wr_chan_id_width
id_width
read_chan_id_width
write_chan_id_width
num_outstanding_xact
num_read_outstanding_xact
num_write_outstanding_xact
num_outstanding_snoop_xact
exclusive_access_enable
barrier_enable
dvm_enable
max_num_exclusive_access
write_data_interleave_depth
serial_read_write_access
```

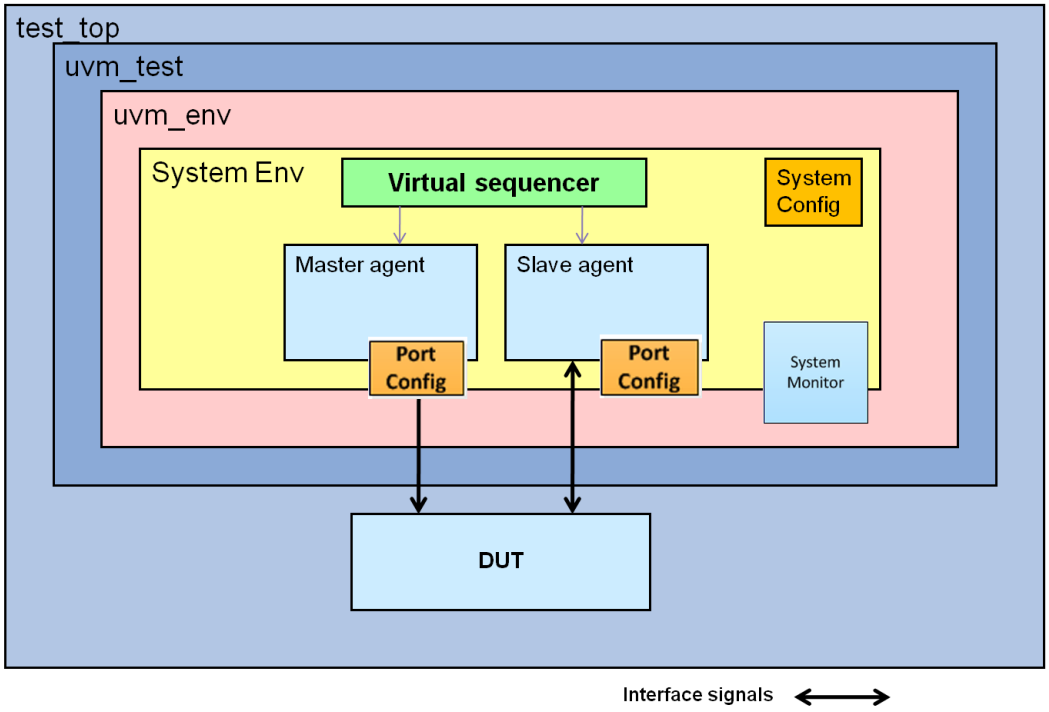
If consistency check fails, VIP issues the following message:

```
UVM_FATAL /…/svt_axi_system_env.svp(189) @ 0ns: uvm_test_top.env.axi_env [build_phase]
Invalid configuration passed. Please pass a valid svt_axi_system_configuration object
or derived object.
```

3.2.7 System Environment

The AXI System Env encapsulates the Master agents, Slave Agents, System Sequencer, Interconnect Env and the system configuration. The number of configured Master and Slave Agents is based on the system configuration provided by you. In the build phase, the System Env builds the Master agents, Slave agents and Interconnect Env. After the Master and Slave Agents are built, they are configured by System Env by using the port configuration information available in the system configuration.

Figure 3-3 Usage With System Environment



3.2.7.1 System Sequencer

AXI System sequencer is a virtual sequencer with references to each Master and Slave Sequencers in the system. The System Sequencer is created in the build phase of the System Env. The system configuration is provided to the System Sequencer. The System Sequencer can be used to synchronize between the sequencers in Master and Slave Agents.

3.2.8 System Monitor

The System Monitor component is instantiated within the AXI System Env 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`.

3.2.8.1 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, see the AXI VIP Class reference HTML documentation.

3.2.9 Active and Passive Mode

Tables 3-1 lists the behavior of Master and Slave Agents in active and passive modes.

Table 3-1 Agents in Active and Passive Mode

Component behavior in active mode	Component behavior in passive mode
In active mode, Master and Slave components generate transactions on the signal interface.	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 continue to perform passive functionality of coverage and protocol checking. You can enable/disable this functionality through configuration.	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 agent. This is because when the agent 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.	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 Agent, are the sampled delay values on the bus.
Interconnect component routes transactions from masters to slaves.	Interconnect component does not route the transactions from masters to slaves.
Interconnect component continues to perform passive functionality of coverage and protocol checking on all the master and slave ports. You can enable/disable this functionality through configuration.	Interconnect component monitors the input and output signals on all the master and slave ports, and performs passive functionality of coverage and protocol checking. You can enable/disable this functionality through configuration.

3.3 AXI UVM User Interface

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

3.3.1 Clock and Reset Connection

There is a small change in the bind use model starting from from the 1.96a release of AXI VIP. Following are the details of the change:

1. For active agent connection:

   Pass '1' as the parameter value and svt_axi_master_async_modport as the first argument to the connector.

```
bind test_top svt_axi_master_connector #(1)
master_bind_connector0(axi_if.master_if[0].svt_axi_master_async_modport,
slave_dut.master_bind_if);
```

2. For passive agent connection:

   Pass '0' as the parameter value and svt_axi_monitor_modport as the first argument to the connector.

```
bind test_top svt_axi_master_connector #(0)
master_bind_connector0(axi_if.master_if[0].svt_axi_monitor_modport,
slave_dut.master_bind_if);
```

3.3.2 VIP Interface Connection

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 provided which contains an array of master & slave interfaces.

For example,

```
/* instantiate top level interface*/
svt_axi_if axi_if();
```

There are two ways on how you can set clock signal "aclk" for each/all of master and slave sub interfaces under this top level svt_axi_if interface instance.

1. If you want to use a common clock for all the master and slave port interfaces, there is 'common_aclk' signal at svt_axi_if level , which can be used. This common clock will then be used by all the port interfaces.

   For example,

```
svt_axi_if axi_if();
assign axi_if.common_aclk = SystemClock;
```

You must leave svt_axi_system_configuration::common_clock_mode set to default value '1' for system configuration object.

1. If any/all master/slave port interface under svt_axi_if instance need to use a separate clock, then the 'aclk' signal in the port interface should be connected to respective individual clocks and system configuration object field " svt_axi_system_configuration::common_clock_mode" should be set to '0' to disable common clock mode.

3.3.3 Configuration Objects

Configuration data objects convey the system level and port level testbench configuration. The configuration of agents is done in the `build()` phase of environment or the testcase. If the configuration needs to be changed later, it can be done through reconfigure() method of the Master, Slave Agent or System Env.

The configuration can be of the following two types:

- Static configuration properties

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

- Dynamic configuration properties

  Dynamic configuration parameters specify configuration value which can be changed at any time, regardless of whether the system is running or not. An example of a dynamic configuration parameter is a timeout value.

The configuration data objects contain built-in constraints, which come into effect when the configuration objects are randomized.

The AXI VIP defines following configuration classes:

- System configuration (`svt_axi_system_configuration`)

  The System configuration class contains configuration information which is applicable across the entire system. You can specify the system level configuration parameters through this class. You need to provide the system configuration to the system env from the environment or the testcase. The system configuration mainly specifies:

  - Number of master and slave agents in the system env
  - Port configurations for master and slave agents
  - Virtual top level AXI interface
  - Address map
  - Timeout values

- Port configuration (`svt_axi_port_configuration`)

  The Port configuration class contains configuration information which is applicable to individual AXI master or slave agents in the system env. Some of the important information provided by port configuration class is:

  - Active or Passive mode of the master or slave port agent
  - Enable or disable protocol checks
  - Enable or disable port-level coverage
  - Interface type (AXI3/AXI4/AXI4-Lite)
  - 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.

- Interconnect configuration (`svt_axi_interconnect_configuration`)

Interconnect configuration class contains configuration information for the interconnect component. It has a handle to the system configuration. In addition, this class contains configuration for number of master and slave ports of the interconnect component, and the respective configuration for these master and slave ports.

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

3.3.4 Transaction Objects

Transaction objects, which are extended from the `uvm_sequence_item` 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 `uvm_sequence_item` base class and implement all methods specified by UVM 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

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 the following:
  - 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.

The 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 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 the transaction, to the master and slave drivers, that is, delays for valid and .ready signals with respect to some reference events.

- 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 agent 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 agent provides object of type `svt_axi_slave_transaction` from its analysis ports, in active and passive mode.

  The master and slave transactions contain 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 port configuration is available in the sequencer of the master or slave agent.

  You should initialize the port configuration handle in the transaction using the port configuration available in the sequencer of the master or slave agent. If the port configuration handle in the transaction is null at the time of randomization, the transaction will issue a fatal message.

- 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 port monitor within the master VIP component provides object of type `svt_axi_master_snoop_transaction` from its analysis ports, in active and passive mode.

- AXI transaction on Interconnect Slave port (`svt_axi_ic_slave_transaction`)

  `svt_axi_ic_slave_transaction` class is used by the slave ports of the Interconnect component, to represent the transaction received on the Interconnect slave port from a master component. At the end of each transaction on the Interconnect Slave port, the port monitor within the Interconnect slave port provides object of type `svt_axi_ic_slave_transaction` from its analysis port, in active and passive mode.

- AXI transaction on Interconnect Master port (`svt_axi_ic_master_transaction`)

  `svt_axi_ic_master_transaction` class is used by the master ports of the Interconnect component, to represent the transaction transmitted on the interconnect master port to a connected slave component. At the end of each transaction on the Interconnect Master port, the port monitor within the Interconnect Master port provides object of type `svt_axi_ic_master_transaction` from its analysis port, in active and passive mode.

  This transaction class is not supported in this release. Currently, the port monitor within the Interconnect Master port provides object of type `svt_axi_master_transaction` from its analysis port.

- AXI ACE Snoop transaction on Interconnect Slave port (`svt_axi_ic_snoop_transaction`)

  `svt_axi_ic_snoop_transaction` class extends from the snoop transaction base class `svt_axi_snoop_transaction`. This class represents the snoop transaction at the interconnect slave ports, which are connected to the external master components. At the end of each snoop transaction on the Interconnect Slave port, the port monitor within the Interconnect Slave port provides object of type `svt_axi_ic_snoop_transaction` from its analysis port, in active and passive mode.

For more information on individual members of transaction classes, see the AXI VIP Class reference HTML documentation.

3.3.5 Analysis Ports

The Port Monitor in the Master and Slave Agent provides `item_started_port` and `item_observed_port` analysis ports.

The Master and Slave Agents respectively write the `svt_axi_master_transaction` and `svt_axi_slave_transaction` object to the `item_started_port` analysis port which provides AXI transactions available just when the transaction starts.

At the end of the transaction, the Master and Slave Agents respectively write the completed

`svt_axi_master_transaction` and `svt_axi_slave_transaction` object to the `item_observed_port` analysis port. This holds true in active as well as passive mode of operation of the master or slave agent. 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 the Interconnect Master Agent and Interconnect Slave agent also provides an analysis

port `item_observed_port`. At the end of the transaction on the interconnect ports, the port monitor within the Interconnect Master Agent and Interconnect Slave agent provides the completed

`svt_axi_ic_master_transaction` and `svt_axi_ic_slave_transaction` object respectively, from its analysis port.

Also you can create user-defined analysis ports for their scoreboarding purpose.

Usage:

Steps to use `item_observed_port` analysis port in an UVM verification environment.

1. Create an axi scoreboard class extending from uvm_scoreboard class and declare the export for the analysis port.

```
//The uvm_analysis_imp_decl allows for a scoreboard (or other analysis component) to
support input from many places
/** Macro that define two analysis ports with unique suffixes */
`uvm_analysis_imp_decl(_initiated)
```

```
`uvm_analysis_imp_decl(_response)

class axi_uvm_scoreboard extends uvm_scoreboard;

  /** Analysis port connected to the AXI Master Agent */
  uvm_analysis_imp_initiated#(svt_axi_transaction, axi_uvm_scoreboard)
item_observed_initiated_export;

  /** Analysis port conneted to the AXI Slave Agent */
  uvm_analysis_imp_response#(svt_axi_transaction, axi_uvm_scoreboard)
item_observed_response_export;


  /** UVM Component Utility macro */
  `uvm_component_utils(axi_uvm_scoreboard)



  function new (string name = "axi_uvm_scoreboard", uvm_component parent=null);
      super.new(name, parent);
  endfunction : new

endclass
```

2. In the Scoreboard::`build()` phase, build export of analysis ports and create `write_***()` method to get the object from the analysis ports.

```
class axi_uvm_scoreboard extends uvm_scoreboard;
  ..
  ..
  function void build_phase(uvm_phase phase);
    super.build();
    /** Construct the analysis ports */
    item_observed_initiated_export = new("item_observed_initiated_export", this);
    item_observed_response_export = new("item_observed_response_export", this);

  endfunction
endclass
```

3. Create `write_***()` method to get the object from the `item_observed_port` analysis port class `axi_uvm_scoreboard` extends `uvm_scoreboard`;

```
  ..
  ..

  /** This method is called by item_observed_initiated_export */
  virtual function void write_initiated(input svt_axi_transaction xact);
    svt_axi_transaction init_xact;

    if (!$cast(init_xact, xact.clone())) begin
      `uvm_fatal("write_initiated", "Unable to $cast the received transaction to
svt_axi_transaction");
    end

    `uvm_info("write_initiated", $sformatf("xact:\n%s", init_xact.sprint()), UVM_FULL)

  endfunction

  /** This method is called by item_observed_response_export */
  virtual function void write_response(input svt_axi_transaction xact);
```

```
    svt_axi_transaction  resp_xact;

    if (!$cast(resp_xact, xact.clone())) begin
      `uvm_fatal("write_response", "Unable to $cast the received transaction to
svt_axi_transaction");
    end

    `uvm_info("write_response", $sformatf("xact:\n%s", resp_xact.sprint()), UVM_FULL)

  endfunction
endclass
```

4. In the ENV create an instance of the `axi_uvm_scoreboard` and build the object class `axi_env` extends `uvm_env`;

```
  /** AXI System ENV */
  svt_axi_system_env axi_system_env;
  /** Master/Slave Scoreboard */
  axi_uvm_scoreboard axi_scoreboard;

   /** UVM Component Utility macro */
  `uvm_component_utils(axi_env)

  /** Class Constructor */
  function new (string name="axi_env", uvm_component parent=null);
    super.new (name, parent);
  endfunction

  /** Build the AXI System ENV */
  virtual function void build_phase(uvm_phase phase);
    `uvm_info("build_phase", "Entered...",UVM_LOW)

    super.build_phase(phase);

    ..
    ..

    /* Create the scoreboard */
    axi_scoreboard = axi_uvm_scoreboard::type_id::create("axi_scoreboard", this);

    ..
    ..

    `uvm_info("build_phase", "Exiting...", UVM_LOW)
  endfunction
endclass
```

5. In the connect phase Connect master & slave agent analysis ports to scoreboard

```
class axi_env extends uvm_env;
  ..
  ..
  function void connect_phase(uvm_phase phase);
    `uvm_info("connect_phase", "Entered...",UVM_LOW)

    /**
     * Connect the master and slave agent's analysis ports with
     * item_observed_before_export and item_observed_after_export ports of the
     * scoreboard.
```

```
        */

    axi_system_env.master[0].monitor.item_observed_port.connect(axi_scoreboard.item_observed
    _initiated_export);

    axi_system_env.slave[0].monitor.item_observed_port.connect(axi_scoreboard.item_observed_
    response_export);
        endfunction
    endclass
```

For complete example, see the `tb_axi_svt_uvm_intermediate_sys` example testbench.

Example: How do I add user-defined TLM analysis ports into Port Monitor callbacks in an UVM environment?

https://solvnet.synopsys.com/retrieve/037331.html

3.3.6 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 Agent 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 `uvm_register_cb` macro.

AXI VIP uses callbacks in three main applications:

- Access for functional coverage
- Access for scoreboarding
- Insertion of user-defined code

3.3.6.1 Master Agent Callbacks

In the Master Agent, the callback methods are called by Master Driver and Port Monitor components.

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

- `svt_axi_master_callback`
- `svt_axi_port_monitor_callback`

For more information on these classes, see the class reference HTML documentation.

The following is the list of callback methods available from `svt_axi_master_callback` class:

- virtual function void `associate_xact_to_barrier_pair` (`svt_axi_master` `axi_master` , `svt_axi_master_transaction` `xact`, `svt_axi_barrier_pair_transaction` `barrier_pair_xact [$]`)

  Callback issued by master transactor when barrier transactions are enabled and when '`associate_barrier_xact`' bit is set to 1 in the `svt_axi_master_transaction` class
- virtual function void `input_port_cov` (svt_axi_master axi_master , svt_axi_transaction xact)

  Callback issued to allow the testbench to collect functional coverage information from a transaction received at the input channel which is connected to the generator.
- virtual function void `post_input_port_get` (svt_axi_master axi_master , svt_axi_transaction xact , ref bit drop)

  Called after the master transactor gets a transaction from the input TLM port.
- virtual function void `post_snoop_input_port_get` (svt_axi_master axi_master , svt_axi_master_snoop_transaction xact, ref bit drop)

  Callback issued by master transactor after pulling the snoop response from the snoop response generator
- virtual function void `pre_address_phase_started` (svt_axi_master axi_master , svt_axi_transaction xact)

  Called just before driving the address phase of a transaction.
- virtual function void `pre_cache_update` (svt_axi_master axi_master , svt_axi_master_transaction xact)

  Callback issued by master transactor just before updating the data into the cache.
- virtual function void `pre_snoop_data_phase_started` (svt_axi_master axi_master , svt_axi_master_snoop_transaction xact)

  Callback issued just before driving the data phase of a snoop transaction.
- virtual function void `pre_snoop_resp_phase_started` (svt_axi_master axi_master , svt_axi_master_snoop_transaction xact)

  Callback issued just before driving response to a snoop transaction.
- virtual function void `pre_write_data_phase_started` (svt_axi_master axi_master , svt_axi_transaction xact)

  Called just before driving a data beat of a write transaction
- virtual function void `snoop_input_port_cov` (svt_axi_master axi_master , svt_axi_master_snoop_transaction xact)

  Callback issued to allow the testbench to collect functional coverage information from a snoop transaction received at the input port of the master transactor, which is connected to the snoop response generator.
- `svt_axi_master_callback` methods arguments description:
  - `axi_master` - A reference to the svt_axi_master component that is issuing this callback. The user's callback implementation can use this to access the public data and/or methods of the component.
  - `xact` - A reference to the transaction descriptor object of interest.

- drop - A ref argument, which if set by the user's callback implementation causes the transactor to discard the transaction descriptor without further action.

3.3.6.2 Slave Agent Callbacks

In the Slave Agent, the callback methods are called by Slave Driver and port monitor components.

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

- svt_axi_slave_callback
- svt_axi_port_monitor_callback

For more information of these classes, see the class reference HTML documentation.

The following is the list of callback methods available from svt_axi_slave_callback class:

- virtual function void input_port_cov (svt_axi_slave axi_slave , svt_axi_transaction xact)

  Callback issued to allow the testbench to collect functional coverage information from a transaction received the input channel which is connected to the generator.

- virtual function void post_input_port_get (svt_axi_slave axi_slave , svt_axi_transaction xact , ref bit drop)

  Called after the slave transactor gets a slave response transaction from the slave response generator.

- virtual function void pre_read_data_phase_started (svt_axi_slave axi_slave , svt_axi_transaction xact)

  Called just before driving the read data phase of a read transaction.

- virtual function void pre_write_resp_phase_started (svt_axi_slave axi_slave , svt_axi_transaction xact)

  Called just before driving a write response phase of a write transaction.

- svt_axi_slave_callback method arguments description:
  - axi_slave - A reference to the svt_axi_slave component that is issuing this callback. The user's callback implementation can use this to access the public data and/or methods of the component.
  - xact - A reference to the transaction descriptor object of interest.
  - drop - A ref argument, which if set by the user's callback implementation causes the transactor to discard the transaction descriptor without further action.

The following is the list of callback methods available from svt_axi_port_monitor_callback class:

- virtual function void new_snoop_transaction_started (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when a new snoop transaction is observed on the port.

- virtual function void new_transaction_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when a new transaction is observed on the port.

- virtual function void pre_output_port_put (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called before putting a transaction to the analysis port. Extension of this method in the default coverage callback class is used for triggering transaction coverage.

- virtual function void pre_response_request_port_put (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called just before the response request transaction is provided by slave port monitor to slave response generator.
- virtual function void pre_snoop_output_port_put (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called before putting a snoop transaction to the analysis port.
- virtual function void pre_tlm_generic_payload_port_put (svt_axi_port_monitor axi_monitor , uvm_tlm_generic_payload xact)

  Called when a transaction completes and when use_tlm_gp_sequencer is set in the port configuration. The completed AXI transaction is converted to a PV-annotated TLM GP and is made available through this callback.
- virtual function void pre_tlm_generic_payload_snoop_port_put (svt_axi_port_monitor axi_monitor , uvm_tlm_generic_payload xact)

  Called when a snoop transaction completes and when use_tlm_gp_sequencer is set in the port configuration. The completed AXI snoop response is converted to a PV-annotated TLM GP and is made available through this callback.
- virtual function void read_address_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when read address handshake is complete, that is, when ARVALID and ARREADY are asserted. Extension of this method in the default coverage callback class is used for signal coverage of read address channel signals.
- virtual function void read_address_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  ```
  Called when ARVALID is asserted.
  ```
- virtual function void read_data_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when read data handshake is complete, that is, when RVALID and RREADY are asserted. Extension of this method in the default coverage callback class is used for signal coverage of read data channel signals.
- virtual function void read_data_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when RVALID is asserted.
- virtual function void snoop_address_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when snoop address handshake is complete, that is, when ACVALID and ACREADY are asserted.
- virtual function void snoop_address_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when ACVALID is asserted.
- virtual function void snoop_data_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

Called when snoop data handshake is complete, that is, when CDVALID and CDREADY are asserted.

- virtual function void snoop_data_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when CDVALID is asserted.

- virtual function void snoop_resp_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when snoop response handshake is complete, that is, when CRVALID and CRREADY are asserted.

- virtual function void snoop_resp_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_snoop_transaction item)

  Called when CRVALID is asserted.

- virtual function void stream_transfer_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when stream handshake is complete, that is, when TVALID and TREADY are asserted.

- virtual function void stream_transfer_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when TVALID is asserted.

- virtual function void transaction_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when a transaction ends.

- virtual function void write_address_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when write address handshake is complete, that is, when AWVALID and AWREADY are asserted. Extension of this method in the default coverage callback class is used for signal coverage of write address channel signals.

- virtual function void write_address_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when AWVALID is asserted.

- virtual function void write_data_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when write data handshake is complete, that is, when WVALID and WREADY are asserted. Extension of this method in the default coverage callback class is used for signal coverage of write data channel signals.

- virtual function void write_data_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when WVALID is asserted.

- virtual function void write_resp_phase_ended (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

Called when write response handshake is complete, that is, when BVALID and BREADY are asserted. Extension of this method in the default coverage callback class is used for signal coverage of write response channel signals.

- virtual function void write_resp_phase_started (svt_axi_port_monitor axi_monitor , svt_axi_transaction item)

  Called when BVALID is asserted.

- svt_axi_port_monitor_callback method arguments description:
  - axi_monitor - A reference to the svt_axi_port_monitor component that is issuing this callback. The user's callback implementation can use this to access the public data and/or methods of the component.
  - item - A reference to the transaction descriptor object of interest.

3.3.6.3 Interconnect Env Callbacks

In the Interconnect Env, callback methods are called by the master and slave ports.

The following callback class contains the Interconnect Env callback method:

- `svt_axi_interconnect_callback`

For more information of these classes, see the class reference HTML documentation.

The following is the list of callback methods available from `svt_axi_interconnect_callback` class:

- virtual function void post_input_port_get(svt_axi_interconnect axi_interconnect, svt_axi_ic_slave_transaction xact)

  Callback issued just after receiving a coherent transaction.

- virtual function void post_slave_xact_gen(svt_axi_interconnect axi_interconnect, svt_axi_master_transaction xact)

  Callback issued after the interconnect randomizes a transaction to be routed to a slave.

- virtual function void pre_output_port_put (svt_axi_interconnect axi_interconnect , svt_axi_ic_slave_transaction xact)

  Callback issued after the interconnect receives all responses from snooped ports and before driving coherent response to corresponding port.

- svt_axi_interconnect_callback method arguments description:
  - axi_interconnect - A reference to the svt_axi_interconnect component that is issuing this callback. The user's callback implementation can use this to access the public data and/or methods of the component.
  - xact - A reference to the transaction descriptor object of interest.

3.3.6.4 System Monitor Callbacks

System Monitor provides hooks in the form of callbacks, which can be used to perform such design specific checks. The following callback class contains the System Monitor callback method:

`svt_axi_system_monitor_callback`

Refer to AXI Class Reference HTML documentation for the specific callback methods of this callback class.

- virtual function void interconnect_generated_dirty_data_write_detected (svt_axi_system_monitor system_monitor , svt_axi_system_transaction sys_xact , svt_axi_transaction slave_xact)

Called after the system monitor detects that a write transaction initiated by the interconnect corresponds to a write of dirty data returned by a snoop transaction.

- virtual function void master_xact_fully_associated_to_slave_xacts (svt_axi_system_monitor system_monitor , svt_axi_system_transaction sys_xact)

  Called after the system monitor correlates all the bytes of a master transaction to corresponding slave transactions.

- virtual function void new_master_transaction_received (svt_axi_system_monitor system_monitor , svt_axi_transaction xact)

  Called when a new transaction initiated by a master is observed on the port.

- virtual function void new_slave_transaction_received (svt_axi_system_monitor system_monitor , svt_axi_transaction xact)

  Called when a new transaction initiated by an interconnect to a slave is observed on the port.

- virtual function void new_snoop_transaction_received (svt_axi_system_monitor system_monitor , svt_axi_snoop_transaction xact)

  Called when a new snoop transaction initiated by an interconnect is observed on the port.

- virtual function void new_system_transaction_started (svt_axi_system_monitor system_monitor , svt_axi_system_transaction sys_xact , svt_axi_transaction xact)

  Called when a new overlapped transaction initiated by a master is observed on the port.

- virtual function void post_coherent_and_snoop_transaction_association (svt_axi_system_monitor system_monitor , svt_axi_transaction coherent_xact , svt_axi_snoop_transaction snoop_xacts [$])

  Called after the system monitor associates snoop transactions to a coherent transaction.

- virtual function void pre_check_execute (svt_axi_system_monitor system_monitor , svt_err_check_stats check , svt_axi_transaction xact , ref bit execute_check)

  Called before a check is executed by the system monitor. Currently supported only for data_integrity_check.

- svt_axi_system_monitor_callback method arguments description:
  - system_monitor - A reference to the svt_axi_system_monitor component that is issuing this callback. The user's callback implementation can use this to access the public data and/or methods of the component.
  - sys_xact - A reference to the system transaction descriptor object of interest.
  - slave_xact - A reference to the slave transaction descriptor object which was detected as a dirty data write.
  - xact - A reference to the data descriptor object of interest.
  - coherent_xact - A reference to the coherent data descriptor object of interest.
  - snoop_xacts - A queue of all associated snoop transactions.
  - check - A reference to the check that will be executed
  - execute_check - A bit that indicates if the check must be performed.

Usage:

Steps to implement callbacks feature in an UVM verification environment.

1. Create a user-defined callback class that extends from the AXI VIP callback class.

class axiPortMonitorCallbacks extends svt_axi_port_monitor_callback;

2. Implement the required callback method in this extended class.

```
virtual function void  new_transaction_started (svt_axi_port_monitor axi_monitor,
svt_axi_transaction item);
    $display("Inside new_transaction_started Port Monitor Callback");
    item.print();
endfunction
```

3. Declare an instance of the user defined callback class (Example: In your env).

```
class axiEnv extends uvm_env;
  axiPortMonitorCallbacks monitor_cb;
  ..
  ..
endclass
```

4. Register the callback with the appropriate component in either connect_phase or start_of_simulation_phase.

```
class axiEnv extends uvm_env;
  ..
  ..
  function void start_of_simulation();
    super.start_of_simulation_phase(phase);
    monitor_cb = new( "monitor_cb" ); //create object of callback class
    uvm_callback#(svt_axi_port_monitor,svt_axi_port_monitor_callback)::add(
   axi_system_env.master[0].monitor, monitor_cb);
         //Registering the callback with AXI Master VIP's monitor.
     //The master monitor type and the master monitor callback type are provided as
parameters to uvm_callbacks class.
     //The add method takes the actual instance of the AXI master monitor and the
callback object.
  endfunction
endclass
```

Example: Usage of AXI port monitor callback to get count of outstanding transactions with AXI SVT VIP slave component

(solvnet: https://solvnet.synopsys.com/retrieve/040575.html)

3.3.7 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`.

Alternatively, the interface can also be specified to the AXI System Env component directly through UVM Configuration database. For more details on usage, see AXI Basic example `tb_axi_svt_uvm_basic_sys`.

If the AXI System Env is used, then it first retrieves the configuration using the config db. It then attempts to retrieve the virtual interface using the config db. If a virtual interface is supplied through the config db, then

the AXI System Env will update the configuration with it (a warning will be generated if the configuration object already has a virtual interface reference). The AXI System Env then passes the configuration object down to the master and slave agents. If the virtual interface is not supplied through the config db, then a fatal error is generated if the virtual interface is not valid in the configuration.

Otherwise the virtual interface in configuration is used without modification. When the System Env has a configuration object with a valid virtual interface, then all the sub-objects receive the interface from the configuration object.

If the Master or Slave Agent is used as standalone, then the process is the same. These classes will continue to receive the configuration object using the config db. In addition, they will retrieve the virtual interface from the config db and perform the same checks done in the AXI System Env to ensure that a valid configuration object is created that contains a virtual interface reference.

For more information on AXI Interface, see the `$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/axi_svt_uvm_class_reference/html/interfaces.html`

3.3.7.1 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 information on debug port, see the "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. See "Using Debug Port" for details on debug port.

3.3.7.2 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.

3.3.7.2.1 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.

3.3.7.2.2 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.

3.3.7.3 Bind Interfaces

AXI VIP also supports bind interfaces for master & slave. Bind interface is an interface which contains directional signals for AXI. You 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, you must 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. You must 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.

For more information on the usage of bind interface, see the AXI intermediate example.

3.3.7.4 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` (see Step 3.3.9). 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.

For usage of parameterized interface, see the `tb_axi_svt_uvm_basic_param_if_sys` example. The README file in the example describes the usage.

3.3.8 Transaction Status Tracking Methods and Events

Transaction status tracking methods provides information on the status of the data transfer at the interface. Different methods and events that you can make use of are as follows:

- Transaction Class Status Attributes
- Transaction Class Methods
- Events

3.3.8.1 Transaction Class Status Attributes

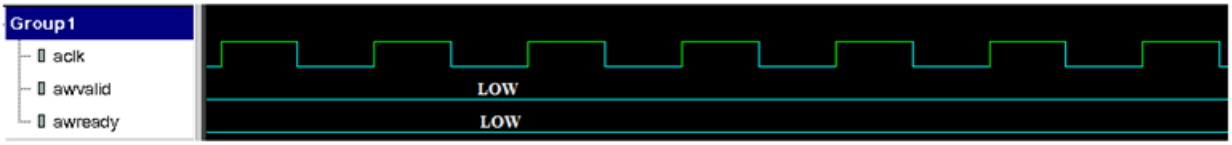
Transaction class status attributes indicate status of transactions based on valid and ready signals of the axi interface. The status attributes are addr_status, data_status, and write_resp_status for address phase, data phase, and write response phase respectively. The status indicator strings are INITIAL, ACTIVE, ACCEPT, PARTIAL _ACCEPT and ABORTED. HTML Class reference document provides detailed description on status strings and transaction status flow.

You can track the transaction flow through transaction object by referring these status indicator strings. This is helpful for transaction tracking in the log file.

INITIAL

Status is considered as INITIAL when valid and ready are both LOW on the channel

Example: `Read addr_status` at INITIAL state is shown as follows:

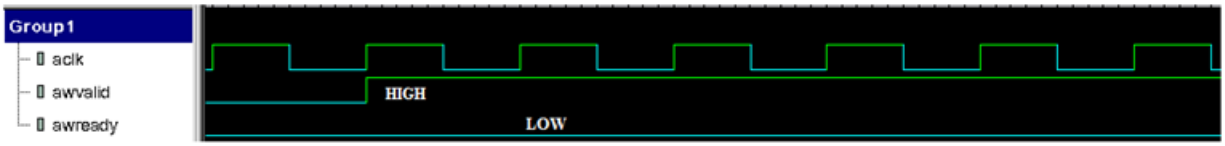


The status indicates that master has not driven an address at the interface

ACTIVE

The Status is considered as ACTIVE when valid signal is HIGH with ready signal at LOW. If the VIP agent is driving a transaction, (that is, Active VIP Agents) the status will be set to ACTIVE when it asserts valid signal whereas if the VIP Agent is monitoring the transfer, (for example, Passive VIP Agents) the status will be set to ACTIVE at the event of sampling valid signal.

Example: Read `addr_status` status changing to ACTIVE is illustrated as follows:

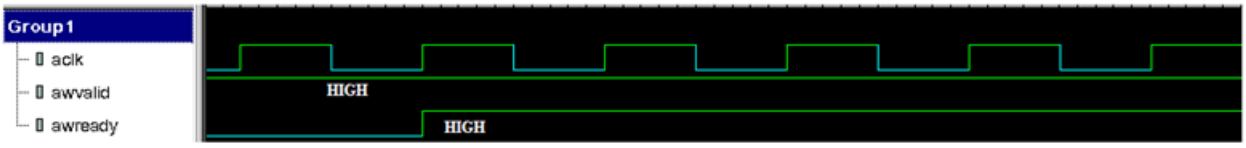


The status indicates that master has driven an address and is not yet accepted by the slave

ACCEPT

Status is considered as ACCEPT when the channel handshake is complete, that is when both ready and valid are both HIGH.

Example: Read `addr_status` changing to ACCEPT is shown as follows:

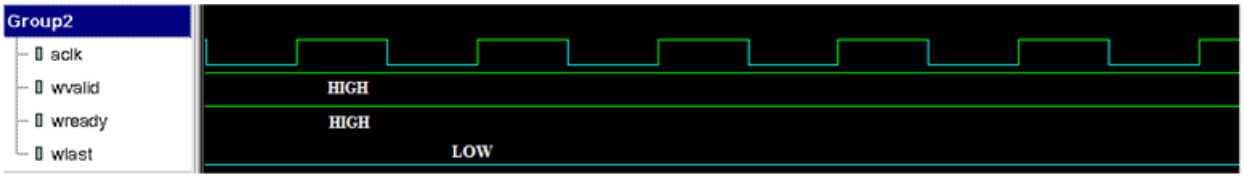


The status indicates that the slave has accepted the address with awvalid-awready handshake

PARTIAL_ACCEPT

PARTIAL_ACCEPT status is applicable on read/write data channels incase of multi-beat or burst transfer. In case of multi-beat transfer, the transfer is complete only when the last beat data is transferred. Status is

considered as PARTIAL_ACCEPT when a beat is completed with hand shake but the last beat data is not transferred. For example, In case of an INCR4 write, for beat 1-3, when wvalid and wready are both HIGH then the status is PARTIAL_ACCEPT. The figure shows write `data_status` as PARTIAL_ACCEPT.



wlast at LOW indicate that the write data beats are not complete.

ABORTED

The status is considered as ABORTED when a transfer is canceled. This happens in case of a mid-simulation reset.

3.3.8.2 Transaction Class Methods

- `get_begin_time()`: This method gives starting time of a transaction.
- `get_end_time()`: This method gives end time of a transaction.
- `wait_for_transaction_end()`: This method waits for the transaction to end. In case of read transfer, the transaction ends when read response is complete with read response handshake.

Similarly, for write, the transaction ends when the write response handshake is complete.

3.3.8.3 Events

VIP components issue transaction begin_event and end_event. These events are provided by the uvm library and they denote the start of transaction and end of transaction. These events are issued by the Master and Slave components as described below, in both active and passive mode.

- `begin_event`: For WRITE transactions, begin_event is issued on the rising clock edge when awvalid (for address before data) or wvalid (for data before address) is high. For READ transactions, begin_event is issued on the rising clock edge when arvalid is high.
- `end_event`: For WRITE transactions, the end_event is issued on the rising clock edge when bvalid and bready both are high. For READ transactions, the end_event is issued on the rising clock edge when rvalid, rlast and rready are high.

3.3.9 Overriding System Constants

The 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. It 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

  This file 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 the following:

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

- `svt_axi_port_defines.svi`

  This file 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`

  This file 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:

- Do not 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.

3.3.10 Support for TLM Generic Payload

The AXI VIP supports TLM Generic Payload feature where the user can develop sequences based on the uvm_tlm_generic_payload transaction type. The AXI VIP then maps these Generic Payload sequences into AXI specific sequences.

This feature is supported for UVM flow only, for interface types AXI3 and AXI4. Also, TLM Generic Payload feature does not yet map TLM Generic Payload transactions to AXI transactions with burst length greater than 16.

3.3.10.1 Generating TLM Generic Payload Stimulus

By default, AXI stimulus is generated using svt_axi_master_transaction sequence items in the AXI master agent sequencer. The bus-agnostic stimulus can be generated using uvm_tlm_generic_payload sequence items.

You can enable this functionality by setting the svt_axi_port_configuration::use_tlm_generic_payload to '1' for the corresponding AXI master before that master's build_phase is executed.

```
function void my_env::build_phase(uvm_phase phase);
   super.build_phase(phase);

   cfg.master_cfg[1].use_tlm_generic_payload = 1;
   uvm_config_db#(svt_axi_system_configuration)::set(this,
                                                     "axi_env",
                                                     "cfg", cfg);
   axi_env = axi_system_env::type_id::create("axi_env", this);
endfunction
```

Enabling this functionality causes the instantiation of svt_axi_tlm_generic_payload_sequencer in the svt_axi_master_agent::tlm_generic_payload_sequencer property and the execution of a layering sequence on the AXI transaction sequencer. The layering sequence pulls generated TLM generic payload sequence items from the generic payload sequencer, maps them to one or more AXI master transactions, and executes them on the driver. The layering sequence executes with a normal priority. It is still possible to execute normal AXI transaction sequences on the AXI transaction sequencer, in parallel with the TLM generic payload layering sequence.

The response from the execution of the generic payload item is annotated in the generic payload sequence item itelf. It is valid only when the completed generic payload sequence item is returned by the uvm_sequence::get_response() method.

```
class my_gp_seq extends uvm_sequence#(uvm_tlm_generic_payload);
   ...
   task body();
      `uvm_create(req);
      req.set_command(UVM_TLM_READ_COMMAND);
      req.set_address('h123456789);
      req.set_length(64);
      `uvm_send(req);
      get_response(rsp);
      if (rsp.is_response_ok()) begin
         // gp.m_data[] is now valid
      end
   endtask
endclass
```

The TLM generic payload sequence items are mapped into one or more AXI transactions that implement the semantics of the Generic Payload transaction, as defined by the TLM 2.0 standard. It is not possible to generate all possible AXI master transactions from generic payload stimulus.

For demonstration of the usage, see the ts.tlm_generic_payload_test.sv test present in the tb_axi_svt_uvm_intermediate_sys example.

By default, generic payload WRITE and READ commands are mapped to WRITENOSNP and READNOSNP AXI transactions respectively, with a maximum 16-beat INCR burst and individual transfer size matching the configured port size. In case different AXI transactions are required, the generic payload sequence item must be annotated with an instance of the svt_amba_pv_extension generic payload AMBA PV (Programmer's View) extension.

```
class my_gp_seq extends uvm_sequence#(uvm_tlm_generic_payload);
   ...
   task body();
      svt_amba_pv_extension   pv;

      `uvm_create(req);
      pv = new("pv");
      req.set_extension(pv);
      ...
      pv.set_size(1);
      pv.set_length(64);
      pv.set_burst(svt_axi_transaction::WRAP);

      `uvm_send(gp);
   endtask
endclass
```

The various attributes of the AMBA PV extension can be set to specify the characteristics of the AXI transaction(s) used to implement the annotated generic payload transaction. Should the annotation be present, it will be further annotated with the relevant response from the execution of the AXI transactions. The relevant response will be annotated within the member svt_amba_pv_extension::m_response of svt_amba_pv_response type.

Note For details of svt_amba_pv_extension, see AXI UVM Class Reference HTML. See ts.amba_pv_test.sv test present in tb_axi_svt_uvm_intermediate_sys example for demonstration of the usage.

3.3.10.2 Connecting a TLM 2.0 Master

By default, TLM generic payload stimulus is generated using SystemVerilog sequences in the AXI master agent generic payload sequencer. If the TLM generic payload transactions are created by an ARM FastModel or a TLM Master model written in SystemC/SystemVerilog, it is possible to connect the AXI master agent to a TLM master. AXI Master agent component in the AMBA VIP provides required sockets for connecting to the TLM master.

Should the TLM Master be implemented in SystemC, you will need to connect the socket on the Master to the socket on the VIP using UVMConnect or VCS/TLI and convert the AMBA PV SystemC transactions to equivalent AMBA PV SystemVerilog transactions.

You can enable this functionality by setting the svt_axi_port_configuration::use_pv_socket to '1' for the corresponding AXI master before that master's build_phase is executed.

```
function void my_env::build_phase(uvm_phase phase);
   super.build_phase(phase);

   cfg.master_cfg[1].use_pv_socket = 1;

   uvm_config_db#(svt_axi_system_configuration)::set(this,
```

```
                                                        "axi_env",
                                                        "cfg", cfg);
   axi_env = axi_system_env::type_id::create("axi_env", this);
endfunction
```

Enabling this functionality implies the enabling of TLM generic payload stimulus (see Section 3.3.10.1).

Enabling this functionality causes the instantiation of an uvm_tlm_b_target_socket interface in the svt_axi_master_agent::b_fwd property and an uvm_tlm_b_initiator_socket interface in the svt_axi_master_agent::b_snoop property. Furthermore, the default run_phase sequence for the ACE snoop response sequencer is replaced with a reactive sequence which forwards all ACE snoop transaction requests (translated to equivalent uvm_tlm_generic_payload transactions annotated with a svt_amba_pv_extension) to the backward svt_axi_master_agent::b_snoop interface to be fulfilled by the coherent TLM master. The coherent TLM master must provide the snoop response by providing the relevant cache line content in the data member of the uvm_tlm_generic_payload and status information in the relevant fields of the attached svt_amba_pv_extension.

In case the TLM master is not coherent, the AXI master agent can be re-configured to handle ACE snoop requests natively using its local cache model. The following is an example code snippet that can be used for this purpose:

```
uvm_config_db#(uvm_object_wrapper)::set("axi_env.master[2]",
   "snoop_sequencer.run_phase", "default_sequence",
   svt_axi_ace_master_snoop_response_sequence::type_id::get());
```

For demonstration of the usage for AXI3/4, see the `ts.amba_pv_test.sv` test within the `tb_axi_svt_uvm_intermediate_sys` example . For demonstration of the usage for ACE, see `tb_axi_svt_uvm_ace_sys` example.

3.3.10.3 Connecting a TLM 2.0 Slave

As Reactive agent, the sequence `svt_axi_slave_tlm_response_sequence` in AXI Slave agent sequencer translates slave transactions into corresponding AMBA-PV extended TLM Generic Payload Transactions. This is applicable for TLM generic payload transactions created by an ARM.

FastModel or a TLM Slave model written in SystemC or SystemVerilog, connects the AXI Slave agent to a TLM Slave. The AMBA VIP provides the sockets required for connecting to the TLM slave in the AXI Slave agent component.

When the TLM Slave is implemented in SystemC, you will need to connect the socket on the Slave to the socket on the VIP using UVM Connect or VCS/TLI and convert AMBA PV SystemVerilog transactions to AMBA PV SystemC transactions.

Support for TLM GP in the AXI slave is through sockets. Therefore, the configuration attribute `svt_axi_port_configuration::use_pv_socket` must be set to '1' to enable TLM GP at the slave for the corresponding AXI Slave before that slave's `build_phase` is executed.

```
function void my_env::build_phase(uvm_phase phase);
   super.build_phase(phase);

   cfg.slave_cfg[1].use_pv_socket = 1;

   uvm_config_db#(svt_axi_system_configuration)::set(this,
                                                     "axi_env",
```

```
                                                        "cfg", cfg);
   axi_env = axi_system_env::type_id::create("axi_env", this);
Endfunction
```

Enabling this functionality causes the instantiation of an `uvm_tlm_b_initiator_socket` interface in the `svt_axi_slave_agent::resp_socket` property.

For demonstration of the usage of AXI3 or AXI4, see `ts.amba_pv_test.sv` test within the `tb_axi_svt_uvm_intermediate_sys` example.

3.4 Functional Coverage

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

3.4.1 Default Coverage

The following sections describe the default coverage provided with AXI VIP. For more details on actual cover groups, see 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.

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 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. Tables 3-2 shows the cross coverage points.

Table 3-2 AXI3/4 Transaction Cross Coverage Matrix

awburst/awburst	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 and 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

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

awburst/awburst	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 cache types Covergroup names: trans_cross_axi_arburst_arlen_arcache trans_cross_axi_awburst_awlen_awcache
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

3.4.2 Coverage Callback Classes

3.4.2.1 Coverage Data Callback

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 agent callback class.

Based on above, the coverage data callback class name is svt_axi_def_cov_data_callbacks.

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 Callback

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, 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 the 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 following sections. For a comprehensive list of all protocol checks for AXI3/4 protocol, see the class `svt_axi_checker` in AXI VIP Class Reference HTML documentation.

Table 3-3 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

Table 3-3 Write Address Channel Checks (Continued)

Protocol check	Check condition
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 3-4 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

Table 3-4 Write Data Channel Checks (Continued)

Protocol check	Check condition
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 3-5 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
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 3-6 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
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 3-7 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 3-8 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
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

Table 3-9 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 3-10 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 3-11 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 3-12 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 agents 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 agents. 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 agent are resumed after the reset signal of master agent 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 agents are ABORTED. For the master agent, this includes the transactions whose address, data, response phases are in progress, and also the transactions that are not yet started by the master agent (present in the active queue). All the aborted transactions are provided from the analysis port item_observed_port of the master and slave agents. 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 Agent

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

The axi_master_agent_svt component supports the following:

- 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, that is, 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 snoop response confirms to ACE protocol specification. The snoop response is provided by the snoop response sequencer within the Master Agent. You need to create a snoop response sequence and register it with the snoop response sequencer within the Master Agent. The snoop response sequence may provide a random snoop response, or a snoop response based on the state of the cache within the Master Agent.

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_agent::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().

The 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.

For more details on accessing the cache, see the Class Reference HTML documentation of `svt_axi_cache` class.

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_agent_svt 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 barrier_association_type RANDOM_ASSOCIATION is currently not supported.

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 driver. 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_agent_svt` component. The AXI VIP supports DVM Operations, DVM Sync and DVM Complete transactions.

As per Specification section C12.2 A, a component must have only one outstanding DVM Sync transaction. A component must receive a DVM Complete transaction before it issues another DVM Sync transaction, but VIP currently blocks all transaction driving (and not just next DVMSYNC), till DVMCOMPLETE comes back for recently completed DVMSYNC.

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 by 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

The Automatic generation of DVM Complete in response to the DVM Sync is not currently supported. However, this feature will be supported in future releases.

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.

Programming Multi-part DVM with VIP

- Set addr[0] to 1, you need to switch off the following constraint before randomizing:

```
tr1.reasonable_no_multi_part_dvm.constraint_mode(0);
```

- You should continue to switch off constraint block and as well call following transaction task before randomization:

```
tr2.set_multipart_dvm_flag();
tr2.reasonable_no_multi_part_dvm.constraint_mode(0) ;
```

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 the following 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 the members, see the 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 3-13

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 Agent

The Slave Agent 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 Support for ACE protocol in AXI Interconnect Env

When the svt_axi_port_configuration::axi_interface_type of the Interconnect slave ports is specified as AXI_ACE, the ACE functionality in the corresponding Interconnect Slave ports is enabled.

3.9.1 Support for Coherent and Snoop Transactions

When the Interconnect Slave port receives a coherent transaction from a Master, the Interconnect initiates appropriate snoop transactions towards other masters. The interconnect uses the recommended snoop transactions, as specified in chapter C6 of the ACE protocol specification.

In case of coherent read transactions (ReadOnce, ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique), the interconnect initiates snoop transactions, and also initiates read transaction to the main memory in parallel to the snoop transactions. If data is available from snoop transaction, that data is returned to the initiating master. If data is not available from snoop transaction, then data from main memory is returned to the initiating master.

If dirty data is returned from the responding master and the initiating master cannot accept dirty data (for example, initiating master generated a ReadClean coherent transaction), then the Interconnect writes back data to main memory before sending clean data to the initiating master.

When the interconnect receives a READONCE transaction that spans across multiple cachelines, a snoop transaction corresponding to each cacheline is sent to each AXI_ACE port.

For each cacheline, if any snoop returns data, the interconnect uses it, else, it uses data from memory. It merges data received for each cacheline and returns it to the initiating master. If any snoop passes dirty data, it is written back to memory.

If the interconnect receives a WRITEUNIQUE transaction, it sends a CLEANINVALID snoop transaction corresponding to each cacheline access to all AXI_ACE ports. The interconnect merges data in the WRITEUNIQUE transaction and any dirty data received from the snoop transaction. If the strobe for a byte is asserted, the data from the WRITEUNIQUE transaction is always used. If the strobe for a byte is not asserted, but the snoop transaction returned dirty data for the corresponding byte, then the data from snoop transaction is used. The merged data based on the above description is written into memory.

3.9.2 Support for ACE Domain

The mapping of masters to inner and outer domains can be specified to the Interconnect component through configuration method svt_axi_system_configuration::create_new_domain. Based on this domain mapping information and the domain specified in the coherent transaction from initiating master, the Interconnect generates snoop transactions to the corresponding masters in the domain.

3.9.3 Support for DVM

Interconnect supports DVM feature as specified in the ACE specification.

3.9.4 Support for Barrier

Interconnect orders the post barrier transactions based on barrier transactions. Interconnect also has the ability to forward barrier transactions to downstream slaves. For more information, see the `svt_axi_interconnect_configuration::forward_barriers` configuration member.

3.9.5 Support for Speculative Read

Interconnect always performs speculative read to memory to optimize system performance.

3.9.6 Unsupported ACE Features in Interconnect Env

The following features are not supported in Interconnect Env:

- Exclusive access

3.9.7 Known Limitation

Interconnect does not support ReadOnce and WriteUnique transactions if the total number of bytes to be transferred is larger than cache line size.

3.10 AXI4 Region and Address Range Support in Slave

3.10.1 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.

3.10.2 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

Tables 3-14 depicts the slave response generated for a read transaction, based on the region attribute:

Table 3-14 Slave Response for a Read Transaction

Type Attribute	Resultant Response
R- Read Only	OKAY

Table 3-14 Slave Response for a Read Transaction (Continued)

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

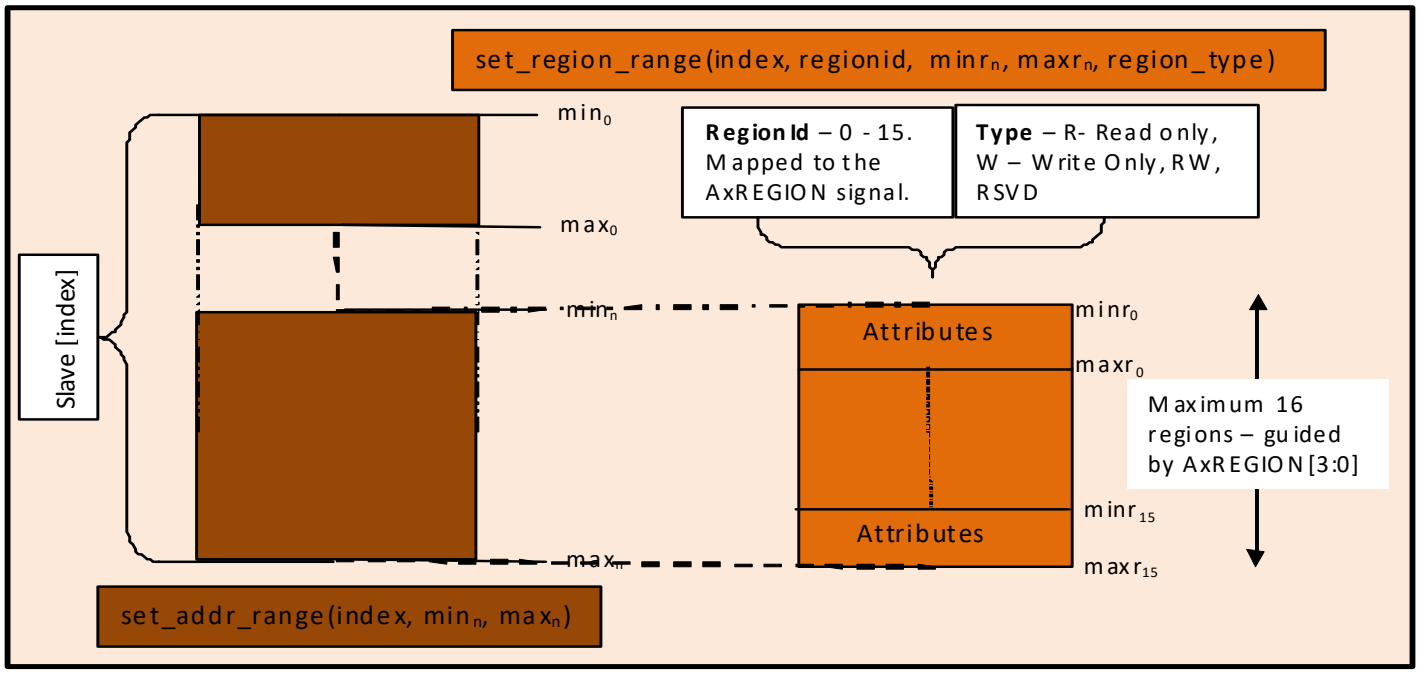
Tables 3-15 depicts the slave response generated for a write transaction, based on the region attribute:

Table 3-15 Slave Response for a Write Transaction

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

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

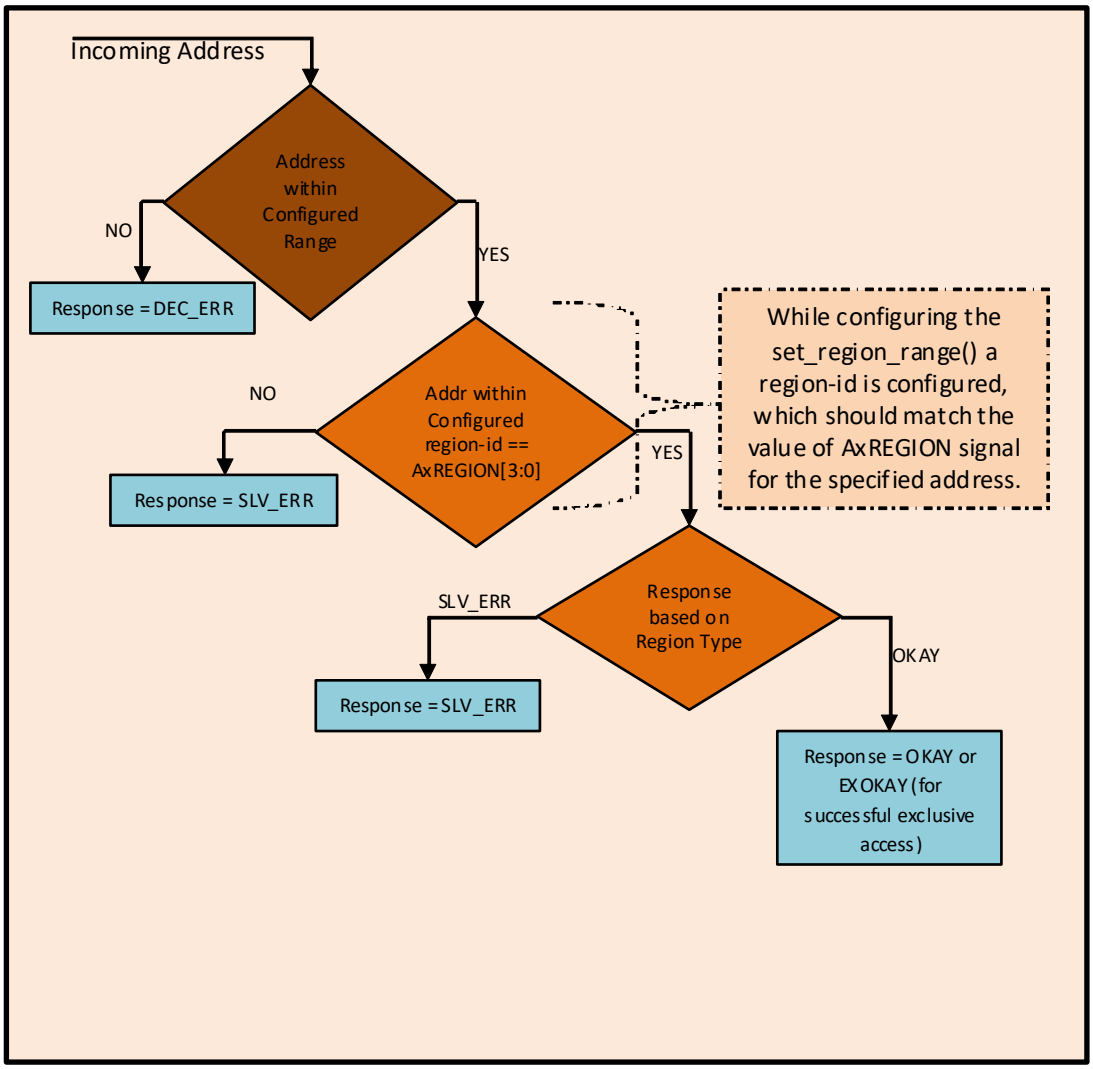
Figure 3-4 Slave Response generated for Write transaction



3.10.3 Slave Response Generation

For each received transaction, the port monitor within the slave agent 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 slave agent sequencer. If the slave sequence running in the slave agent sequencer modifies the pre-populated response in slave transaction, the response might no longer be correct.

Figure 3-5 Flow Diagram for Slave Response Generation



3.11 Support for Monitoring AXI Low Power Interface

The AXI protocol supports a low power interface through the signals ACLK, CACTIVE, CSYSREQ, and CSYSACK signals. These signals allows an AXI component to enter into low power state, and also exit from low power state.

The AXI low power monitor supports monitoring of these signals. It verifies whether the entry to low power state, and exit from the low power state is happening as per the protocol. It also provides the transaction object through the analysis port after a low power handshake is complete.

3.11.1 Module Top

1. `svt_axi_if` should be present in the module top.

   For example,

   ```
   svt_axi_if axi_if();
   ```

   The `svt_axi_if` interface now also contains an array of low power interfaces, `lp_if[]` in addition to `master_if[]` and `slave_if[]`.

2. Connect clock, reset and low power signals to `lp_if`. Signals of low power interface are `aclk`, `aresetn`, `cactive`, `csysreq` and `csysack`.

   For example,

   ```
   assign axi_if.lp_if[0].aclk      = clk;
   assign axi_if.lp_if[0].aresetn = rstn;
   assign axi_if.lp_if[0].cactive =  cactive;
   assign axi_if.lp_if[0].csysreq = csysreq;
   assign axi_if.lp_if[0].csysack = csysack;
   ```

3.11.2 System Configuration

Low power configuration is required in the system configuration file. This involves

1. Configuring the number of low power masters using the attribute 'num_lp_masters'

   For example, `this.num_lp_masters = 1; //This needs to be configured before create_sub_cfgs()`

2. Enable/disable low power protocol checks using `protocol_checks_enable`. This is enabled by default.

3.11.3 Analysis Ports

Low power master monitors have `item_observed_port` which collects and write low power transactions whenever low power activity is observed on the bus.

For example,

```
axi_system_env.lp_master[0].monitor.item_observed_port.connect(lp_listener.analysis_export);
```

The sample print of transaction object obtained from the analysis port:

UVM_INFO ./env/axi_basic_env.sv(45) @ 490000: uvm_test_top.env.lp_listener [lp_listener] inside write method.

```
----------------------------------------------------------------------
Name                              Type                    Size  Value
----------------------------------------------------------------------
lp_entry_obj                      svt_axi_service         -     @1749
  causal_xact                     object                  -     <null>
  implementation                  da(object)              0     -
```

```
  original_xact                 object                  -     <null>
  trace                         da(object)              0     -
  lp_entry_active_req_delay     real                    64    185.000000
  lp_entry_req_ack_delay        real                    64    115.000000
  lp_exit_prp_active_req_delay  real                    64    0.000000
  lp_exit_prp_req_ack_delay     real                    64    0.000000
  lp_exit_ctrl_req_active_delay real                    64    0.000000
  lp_exit_ctrl_req_ack_delay    real                    64    0.000000
  lp_exit_ctrl_active_ack_delay real                    64    0.000000
  lp_handshake_type             lp_handshake_type_enum  32    POWER_DOWN
  lp_initiator                  lp_initiator_type_enum  32    PERIPHERAL
  lp_active_assertion_time      real                    64    190.000000
  lp_req_assertion_time         real                    64    375.000000
  lp_ack_assertion_time         real                    64    490.000000
  begin_time                    time                    64    190000
  end_time                      time                    64    490000
----------------------------------------------------------------------
```

The following is the example snippet. The Master and slave configurations are not included class `cust_svt_axi_system_configuration` extends `svt_axi_system_configuration`;

```
  /** UVM Object Utility macro */
  `uvm_object_utils (cust_svt_axi_system_configuration)

  /** Class Constructor */
  function new (string name = "cust_svt_axi_system_configuration");

    super.new(name);

    /** Assign the necessary configuration parameters. This example uses single
      * master and single slave configuration.
      */
   this.num_masters = 1;
   this.num_slaves  = 1;
   this.num_lp_masters = 1;


    /** Create port configurations */
    this.create_sub_cfgs(1,1);

   this.lp_master_cfg[0].protocol_checks_enable = 1'b1;
```

```
endfunction
endclass
```

4

Support for ACE5, ACE5-Lite, ACE5-Lite+DVM

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

- Overview of ACE5
- Features Supported for ACE5/ACE5-Lite

4.1 Overview of ACE5

The initial features for ACE5 is based on ARM AMBA ACE5 protocol specification ARM IHI 0022H (ID040120) specification. The current release S-2021.06 includes few enhanced features based on the ARM IHI 0022H.c (ID012621) specification.

The VIP features discussed in this chapter are supported only applicable for **ACE5, ACE5-Lite, ACE5-Lite+DVM** interfaces. These features are not applicable for:

- ACE5-LiteACP

4.1.1 Current VIP Use model

Define compile time macro `SVT_ACE5_ENABLE` to enable AMBA5 features(ACE5, ACE5-Lite, ACE5-Lite+DVM).

For a given master/slave agent, set the following port configuration to enable ACE5 features for the port.

Following is the port configuration of VIP master/slave agent with interface as AMBA5-ACE5:

```
svt_axi_port_configuration::ace_version = svt_axi_port_configuration::ACE_VERSION_2_0
svt_axi_port_configuration::axi_interface_type = svt_axi_port_configuration::AXI_ACE
```

4.2 Features Supported for ACE5/ACE5-Lite

These sections describe the features supported by ACE5/ACE5-Lite protocols.

Features of ACE5	Feature Supported in VIP
CMO on Write Channel (CMO on Read, CMO on Write properties)	Not supported
Trace signals	Not supported
User Loopback signaling	Supported in Master
QoS Accept signaling	Not supported

Features of ACE5	Feature Supported in VIP
Wake-up Signaling	Partial: Only AWAKEUP supported; ACWAKEUP not supported
Coherency Connection Signaling	Supported in Master
DVM_v8, DVM_v8.1, DVM_v8.4 (ACE5-LiteDVM interfaces only)	Partial: This is not a port configuration in VIP, it's system configuration. V8.4 is not supported.
Persist CMO (Cache Maintenance for persistence)	Not supported
Untranslated transactions	Supported in Master
Non-secure access identifiers	Not supported
Poison	Not supported
Parity use in AMBA	Not supported
Unique ID indicator	Not supported
Memory Partitioning and Monitoring (MPAM)	Supported in Master
Additional Properties : Exclusive Access	Not supported
Additional Properties : Maximum transaction size and boundary	Not supported
Additional Properties : Consistent DECERR response	Not supported
DVM Message Support	Not supported
Shareable Transactions	Not supported

Features Supported for ACE5-Lite:

Features of ACE5-Lite	Feature Supported in VIP
CMO on Write Channel (CMO on Read, CMO on Write properties)	Not supported
Write with cache maintenance	Partial: Only traffic in external port monitor mode from Master and Slave agents is supported
Atomic transactions	Supported in Master. ACE5-Lite Slave VIP: Only External Port Monitor mode is supported.
Cache stashing	Supported in Master.
Deallocating transactions	Supported in Master. Not Supported in Slave.

Features of ACE5-Lite	Feature Supported in VIP
Trace signals	Supported in Master. Not Supported in Slave.
User Loopback signaling	Supported in Master. Not Supported in Slave.
QoS Accept signaling	Not supported
Wake-up Signaling	Not supported
Coherency Connection Signaling	Not supported
DVM_v8, DVM_v8.1, DVM_v8.4 (ACE5-LiteDVM interfaces only)	Supported in Master. Not supported in Slave.
Persist CMO (Cache Maintenance for persistence)	Supported in Master, Slave
Untranslated transactions	Not supported
Non-secure access identifiers	Not supported
Poison	Supported in Master, Slave
Parity use in AMBA	Supported in Master, Slave. check_type = ODD_PARITY_BYTE_ALL option is not supported. Conversion of data parity error to poison is not fully supported in Slave memory.
Read data chunking	Not supported
Read interleaving property	Not supported
Unique ID indicator	Not supported
Memory Partitioning and Monitoring (MPAM)	Not supported
Memory tagging	Partial: Only traffic in external port monitor mode from Master and Slave agents is supported
Prefetch request and response	Not supported
Write zero with no data	Not supported
Additional Properties : Exclusive Access	Not supported
Additional Properties : Maximum transaction size and boundary	Not supported
Additional Properties : Consistent DECERR response	Not supported
Regular Transactions	Not supported
DVM Message Support	Not supported
Shareable Transactions	Not supported

Note

- Currently, ACE5, ACE5-Lite, ACE5-Lite+DVM features are not supported by AXI System Monitor.
- Currently, ACE5, ACE5-Lite, ACE5-Lite+DVM features are not supported by VIP AXI Interconnect.
- Functional coverage is not supported for ACE5, ACE5-Lite, ACE5-Lite+DVM features.
- Any other feature not listed above is also not supported.

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

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

The following transaction class members are added for this feature:

- svt_axi_transaction:: stream_id
- svt_axi_transaction::secure_or_non_secure_stream
- svt_axi_transaction::sub_stream_id
- svt_axi_transaction::sub_stream_id_valid
- svt_axi_transaction::addr_translated_from_pcie

AxMMUFLOW signals are not yet supported.

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.

Parity checking is currently only supported on Read Data, Write Data and Snoop Data signals.

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
- svt_axi_transaction::data_trace_tag
- svt_axi_transaction::resp_trace_tag
- svt_axi_snoop_transaction::trace_tag
- svt_axi_snoop_transaction::snoop_data_trace_tag
- svt_axi_snoop_transaction::snoop_resp_trace_tag

4.2.7 Atomic Transaction Feature

The following configuration class members are added for this feature:

- svt_axi_port_configuration::atomic_transactions_enable

The following transaction class members are added for this feature:

- svt_axi_transaction::atomic_transaction_type
- svt_axi_transaction::atomic_xact_op_type
- svt_axi_transaction::atomic_read_poison
- svt_axi_transaction::atomic_read_data_status
- svt_axi_transaction::atomic_read_current_data_beat_num
- svt_axi_transaction::atomic_read_data_valid_assertion_cycle
- svt_axi_transaction::atomic_read_data_ready_assertion_cycle

- svt_axi_transaction::atomic_read_data_valid_assertion_time
- svt_axi_transaction::atomic_read_data_ready_assertion_time

4.2.7.1 Updates in Interface Signals

AWATOP signal is added in interface to support ATOMIC signals.

4.2.7.2 Use Model For Atomic Load

a. MASTER SEQUENCE

You can program the following fields in master transactions for generating the atomic transaction traffic

Here xact is `svt_axi_transaction` handle

```
xact.xact_type    = svt_axi_transaction::ATOMIC;
xact.transmitted_channel = svt_axi_transaction::READ_WRITE;
xact.atomic_xact_op_type = svt_axi_transaction::ATOMICLOAD_ADD;
```

Apart from these members, you can also set the newly added members for atomic transactions like `svt_axi_transaction::atomic_read_data` and so on.

b. Slave Sequence

In `svt_axi_slave_mem_response_sequence.sv`, the sequence can be updated on these lines:

Since the transmitted channel for atomic load transaction is `READ_WRITE` as it happens on both READ and WRITE Channel simultaneously.

Here req is `svt_axi_slave_transaction` handle.

```
if(
                (req.transmitted_channel == svt_axi_transaction::READ_WRITE)
              )begin
          // If update_memory_in_request_order is set then call
          // set_update_mem_in_req_order_field(req) task for updating
          // update_mem_in_req_order transaction attribute. Updating memory
          // for this condition will be taken care in
          // put_write_transaction_data_to_mem() task in
          // svt_axi_slave_agent
          if (update_memory_in_request_order)
            set_update_mem_in_req_order_field(req);
          slave_agent.get_read_data_from_mem_to_transaction(req);
          req.perform_atomic_xact_operation(req);
          slave_agent.put_write_transaction_data_to_mem(req);
        end
```

c. APIs for Atomic Load

An API to perform the atomic operation has been added:

```
svt_axi_transaction::perform_atomic_xact_operation(req);
```

You can use this API as shown in point ‘b’, in slave response sequence.

4.2.7.3 Use Model for Atomic Compare

For Atomic compare transaction type, the following new fields have been added in `svt_axi_transaction class`:

1. svt_axi_transaction::atomic_swap_data

2. svt_axi_transaction::atomic_compare_data
3. svt_axi_transaction::atomic_swap_wstrb
4. svt_axi_transaction::atomic_compare_wstrb

You need to program the above fields for atomic compare transaction.

`svt_axi_port_configuration::wysiwyg_enable` would also be used in the above scenario.

There are two use cases:

a. If `svt_axi_port_configuration::wysiwyg_enable` is 0.

   You need to program the following properties in sequence:

   ```
   svt_axi_transaction::atomic_swap_data ,
   svt_axi_transaction::atomic_compare_data,
   svt_axi_transaction::atomic_swap_wstrb ,
   svt_axi_transaction::atomic_compare_wstrb.
   ```

   The VIP converts these values into `svt_axi_transaction::data` and drive it on the interface.

b. If `svt_axi_port_configuration::wysiwyg_enable` is set to 1

   You also need to program `svt_axi_transaction::data` and `svt_axi_transaction::wstrb` along with other properties mentioned above.

   The master drives the data and wstrb as it is on the bus.

5 Support for AXI5

This chapter describes the support for AXI5 protocol.

The ARM AMBA AXI5 Specification reference for the AXI5 features is ARM IHI 0022H.c AXI-H.

This chapter has the following sections:

- Overview of AXI5
- Features Supported for AXI5

5.1 Overview of AXI5

The initial features for AXI5 is based on ARM AMBA AXI5 protocol specification ARM IHI 0022H (ID040120) specification. The current release S-2021.06 includes few enhanced features based on the ARM IHI 0022H.c (ID012621) specification.

The VIP features discussed in this chapter are supported only applicable for **AXI5** interface. These features are not applicable for:

- AXI5-Lite

5.1.1 Current VIP Use model

Define compile time macro `SVT_ACE5_ENABLE` to enable AMBA AXI5 features.

For a given master/slave agent, set the following port configuration to enable AXI5 features for the port.

Following is the port configuration of VIP master/slave agent with interface as AMBA5-AXI5:

```
svt_axi_port_configuration::ace_version = svt_axi_port_configuration::ACE_VERSION_2_0
svt_axi_port_configuration::axi_interface_type = svt_axi_port_configuration::AXI4
```

5.2 Features Supported for AXI5

You can contact Synopsys Support for the latest list of protocol checks of supported features listed here.

Features	Whether Supported
Atomic Transactions	Supported in Master, Slave
Trace signals	Not supported

Features	Whether Supported
User Loopback signaling	Supported in Master, Slave
QoS Accept signaling	Not supported
WakeUp Signaling	Not supported
Untranslated Transactions	Not supported
Non-Secure access identifiers	Not supported
Read Data Chunking	Supported in Master, Slave
Read Interleaving Property	Not supported
Unique ID Identifier	Supported in Master, Slave
MPAM	Supported in Master, Slave
Memory Tagging	Not supported
Exclusive access feature on signal	Not supported
Constant DECERR response	Not supported
Interface additional properties: exclusive access	Not supported
Interface additional properties: Shareable transactions	Not supported
Interface additional properties: Consistent DECERR response	Not supported
Interface and data protection: Poison	Not supported
Interface and data protection: Parity use/parity check	Not supported
Interface and data protection: configuration of i/f protection	Not supported
Interface and data protection: Byte parity checks	Not supported
Interface and data protection: Error detection behavior	Not supported
Data checking feature of ARM IHI 0022F.b (ID122117)	Not supported

Note

- Currently, AXI5 features are not supported by AXI System Monitor and AMBA System Monitor.
- Currently, AXI5 features are not supported by VIP AXI Interconnect.
- Functional coverage is not supported for AXI5 features.
- Any other feature not listed above is also not supported.

5.3 MPAM Feature

MPAM is a technology for partitioning and monitoring memory system resources for physical and virtual machines. When an AXI/ ACE component supports MPAM extension, the following signals are present in the interface.

At interface level

The ARMPAM[11:0] and AWMPAM[11:0] signals are added in master and slave interface

At configuration level

`svt_axi_port_configuration::enable_mpam`

Indicates if Memory Partitioning and Monitoring (MPAM) feature is supported.

It can be set to either of two values below:

```
enable_mpam = svt_axi_port_configuration:FALSE
```

When set to FALSE, MPAM is not supported on the interface.

```
enable_mpam = svt_axi_port_configuration:MPAM_9_1
```

At Transaction level

These fields are added in `svt_axi_transaction` class:

1. rand bit [`SVT_AXI_MAX_MPAM_PARTID_WIDTH-1:0] mpam_partid
2. rand bit [`SVT_AXI_MAX_MPAM_PERFMONGROUP_WIDTH-1:0] mpam_perfmongroup
3. rand bit [`SVT_AXI_MPAM_NS_WIDTH-1:0] mpam_ns

When set to MPAM_9_1, MPAM is supported on the interface with `mpam_partid_width`=9 and `mpam_perfmongroup_width`=1.

Only applicable when SVT_ACE5_ENABLE macro is defined and `svt_axi_port_configuration::ace_version is set to ACE_VERSION_2_0`

When `svt_axi_port_configuration::enable_mpam` is set to `MPAM_FALSE`, the `AxMPAM` fields are constrained to zero and the interface signals corresponding to `mpam_partid`, `mpam_perfmongroup` and `mpam_ns` are driven to zero value.

These features are supported at agent (master/slave) level only and not supported by the AXI System Monitor or AMBA System Monitor of the VIP.

5.4 User Loopback Signaling

User Loopback signaling permits an agent that issues transactions to store information that is related to the transaction in an indexed table. The response to the transaction can use a fast table index to obtain the required information, rather than requiring a more complex lookup that uses the transaction AxID.

At Configuration Level

`svt_axi_port_configuration::enable_loopback_signalling` needs to bet as 1 for given agent to enable loopback signaling feature for given port.

Further loop width settings are required:

```
rand int write_loop_width = `SVT_AXI_MAX_LOOP_W_WIDTH;
rand int read_loop_width = `SVT_AXI_MAX_LOOP_R_WIDTH;
```

In case of atomic transactions, VIP performs check that `write_loop_width` and `read_loop_width` must be equal when atomic transactions are enabled.

Master agent perform `bloop_valid_value_for_write_xacts_check` for loopback enabled transactions, you can refer the HTML user guide for more details/description of this check.

Signal Interface

These signals are added in the master and slave interfaces under the `SVT_ACE5_ENABLE` compile time macro:

```
AXI ACE5 LOOPBACK Write Address Channel Signals
 logic [`SVT_AXI_MAX_LOOP_W_WIDTH-1:0]         awloop;
AXI ACE5 LOOPBACK Read Address Channel Signals
 logic [`SVT_AXI_MAX_LOOP_R_WIDTH-1:0]         arloop;
AXI ACE5 LOOPBACK Feature WRITE Response Channel Signals
logic [`SVT_AXI_MAX_LOOP_W_WIDTH-1:0]         bloop;
AXI ACE5 LOOPBACK Feature Read Response Channel Signals
logic [`SVT_AXI_MAX_LOOP_R_WIDTH-1:0]         rloop;
```

At Transaction Level

- The `svt_axi_transaction::write_request_loopback;` field defines the loopback value in write request that corresponds to AWLOOP.
- The `svt_axi_transaction::read_request_loopback;` field defines the loopback value in read request that corresponds to ARLOOP.

5.5 Unique ID Identifier

At Interface level

```
logic           awidunq;
logic           aridunq;
logic           ridunq;
logic           bidunq;
```

At Configuration level

`svt_axi_port_configuration::unique_id_enable` needs to bet as 1 for given master/slave agent to atomic transactions support for given port.

The value of `svt_axi_port_configuration::single_outstanding_per_id_enable` provided must be set to 0 since `unique_id_enable` is set to 1.

At Transaction level

`svt_axi_transaction unique_id`

This variable is used to indicate that there are no outstanding transactions going on with the same WID/BID/ARID/RID respectively and it will remain unique till the transaction is completed.

5.6 Read Data Chunking

Unsupported features:

- Burst_type WRAP.
- Read data chunking feature under wysiwyg_enable=1

Limitation:

Rchunkv control is supported for the Slave VIP and not Master VIP. This feature is guarded by a macro `SVT_RCHUNKV_ENABLE`

At Interface Level

```
svt_axi_master_if     logic                                          archunken;
svt_axi_master_if     logic                                          rchunkv;
svt_axi_master_if     logic[`SVT_AXI_MAX_CHUNK_NUM_WIDTH-1:0]        rchunknum;
svt_axi_master_if     logic[`SVT_AXI_MAX_CHUNK_STROBE_WIDTH-1:0]     rchunkstrb;
svt_axi_slave_if.svi   logic                                          archunken;
svt_axi_slave_if.svi   logic                                          rchunkv;
svt_axi_slave_if.svi   logic[`SVT_AXI_MAX_CHUNK_NUM_WIDTH-1:0]        rchunknum;
svt_axi_slave_if.svi   logic[`SVT_AXI_MAX_CHUNK_STROBE_WIDTH-1:0]     rchunkstrb;
```

At Configuration Level

`svt_axi_port_configuration::rdata_chunking_enable` needs to set as 1 for given master/slave agent to enable read data chunking feature to support for given port.

`rand int rchunknum_width = `SVT_AXI_MAX_CHUNK_NUM_WIDTH;` is used to set width of the rchunknum signal.

`rand int rchunkstrb_width = `SVT_AXI_MAX_CHUNK_STROBE_WIDTH;` is used to set width of the rchunkstrb signal.

At Transaction Level

`rand bit archunken` = 0; Defines the chunk enable of a AXI5 to enable `read_data_chunking`. When enable, slave will send read data in 128bits of chunk in random order. If disabled, slave will send read data without chunking as per AXI5 protocol.

`rand bit rchunkv` = 0; This signifies the validity of the `rchunkstrb` and `rchunknum` . `rchunkv` when set '1' indicates that read data will be sent in chunks. If this bit is disabled then the read data will not be sent in chunks as `rchunkstrb` and `rchunknum` is not valid. The value of this bit is driven on the interface signal `rchunkv`. This is applicable for the Slave VIP.

`rand bit [`SVT_AXI_MAX_CHUNK_STROBE_WIDTH -1 : 0] rchunkstrb[];` Array of read chunk strobe. Each bit of `rchunkstrb` represents 128bits of read data.

`rand bit [`SVT_AXI_MAX_CHUNK_NUM_WIDTH -1 : 0] rchunknum[];` Indicates that the data chunk number is being transferred.

`int current_data_chunk_trf_num = 0;` This is a counter which is incremented for every chunk of databeat. Useful when user would try to access the transaction class to know its current state during chunking. This represents the chunk databeat transfer number.

`rand rchunkstrb_pattern_enum rchunkstrb_pattern = RCHUNKSTRB_WALKING_ONES;` variable `rchunkstrb_pattern` indicates the pattern generated in the rchunkstrb[].

`extern function void get_chunkstrb_for_wysiwyg_format(ref bit[`SVT_AXI_MAX_CHUNK_STROBE_WIDTH -1:0] rchunkstrb[]);` Ensures that only valid lanes have chunkstrb asserted. In wysisyg format the constraints leave data[] and rchunkstrb[] open. This function is called in `post_randomize` to make sure that `chunkstrb` is asserted only for valid lanes

`extern function int is_unaligned_address();` Indicates the unaligned address

`extern function int get_valid_chunks();` provides the valid number of chunks that are associated with a transaction.

`extern function void get_rchunkstrb_for_address(ref bit[`SVT_AXI_MAX_CHUNK_STROBE_WIDTH -1:0] rchunkstrb[],ref bit [`SVT_AXI_MAX_CHUNK_NUM_WIDTH-1:0] rchunknum []);` ensures that only chunks from valid lanes have chunkstrb asserted.

Since the constrains leave `rchunknum[]` and `rchunkstrb[]` open. This function is called in `post_randomize` to make sure that `chunkstrb` is asserted only for valid lanes in walking ones pattern for both aligned and unaligned address.

`extern virtual function void calcluate_rddata_rchunkstrb_values(ref bit [`SVT_AXI_MAX_CHUNK_STROBE_WIDTH-1:0] rchunkstrb[], ref bit [`SVT_AXI_MAX_CHUNK_NUM_WIDTH-1:0] rchunknum[] ); calulates rchunkstrb[] and rchunknum[]`

6 Verification Features

This chapter describes the various verification features available along with AXI VIP. This chapter discusses the following topics:

- AXI Sequence Collection
- Verification Planner
- Protocol Analyzer Support
- Performance Analysis
- Error Injection
- Phase Jump for AXI VIP

6.1 AXI Sequence Collection

The AXI VIP provides a collection of AXI master and slave sequences. These sequences can be registered with the master and slave sequencers within the master and slave agents respectively, to generate different types of AXI scenarios.

All the AXI Master sequences are extended from base sequence svt_axi_master_base_sequence. All the AXI Slave sequences are extended from base sequence svt_axi_slave_base_sequence.

For a list of all the master and slave sequences, see AXI VIP class reference HTML documentation.

AXI VIP provides a pre-defined AXI Master sequence library svt_axi_master_transaction_sequence_library, which can hold the AXI Master sequences. The library by default has no registered sequences. You are expected to call svt_axi_master_transaction_sequence_library::populate_library() method to populate the sequence library with master sequences provided with the VIP. The port configuration is provided to the populate_library() method. Based on the port configuration, appropriate sequences are added to the sequence library. You can then load the sequence library in the sequencer within the master agent. The AXI master and slave sequences are not encrypted, however, provided as source code.

6.2 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, see the README file, which is available at:

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

6.3 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.

6.3.1 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.

6.3.2 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.

6.4 Performance Analysis

6.4.1 Performance Analyzer

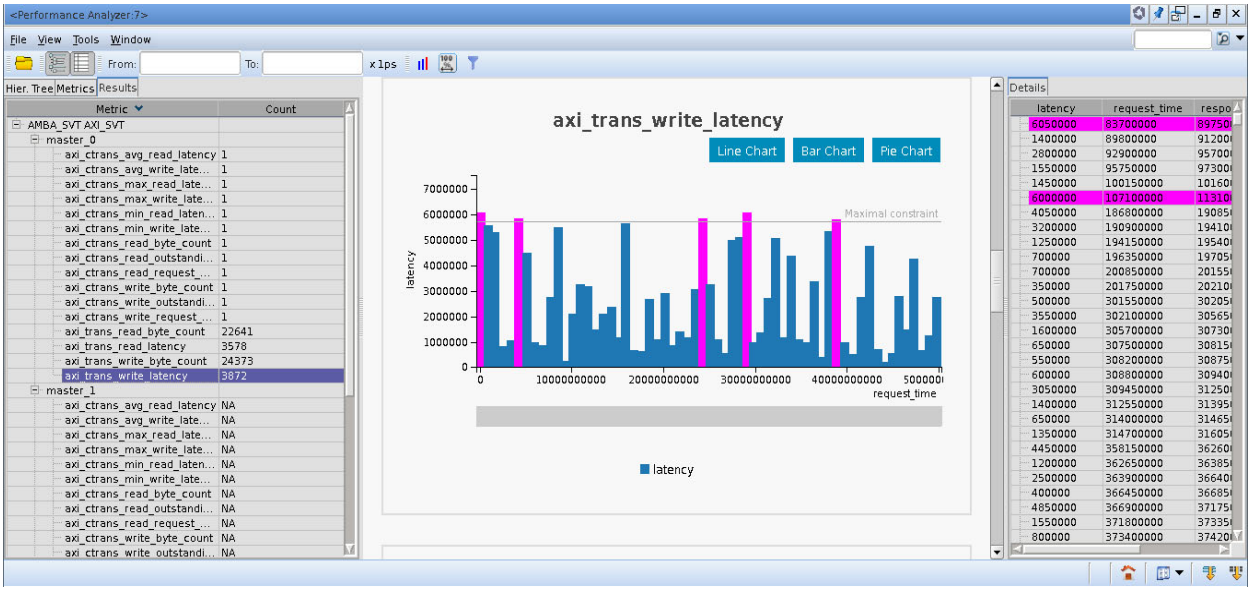
The performance analyzer tool is used to calculate the performance of sub-systems. For more information on FSDB Dumping, see Support for Native Dumping of FSDB.

6.4.1.1 Invoking Verdi GUI After Running the Simulation

1. Invoking Verdi GUI after running the simulation:

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

Figure 6-1 Final View Of Performance Metric With Graph and Data Details



Note For more information, see Verdi_Performance_Analyzer.pdf.

6.4.2 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

6.4.2.1 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`

6.4.2.2 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

6.4.2.3 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

* 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*.

6.5 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.

6.5.1 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 the `svt_axi_transaction_exception::error_kind` member in the AXI Class Reference HTML.

6.5.2 Exception Lists

The Exception list for the 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.

6.5.3 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 the 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 the `svt_axi_master_callback` class.

The following is the code snippet for the UVM 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;
// 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);
  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 = svt_axi_transaction_exception::type_id::create("master_exception");
  my_exception_list =
svt_axi_transaction_exception_list::type_id::create("master_exception_list");

  // 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
   `svt_error("post_input_port_get", $sformatf("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
```

6.6 Phase Jump for AXI VIP

AXI VIP supports UVM Phase jump from `main_phase()` to `reset_phase()`.

7 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:

7.1 Testing a Master DUT Using an UVM Slave VIP

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

- Testbench setup: Configure the AXI System Env to have one Slave Agent, in active mode. The active Slave Agent will respond to the transactions generated by master DUT. The Slave Agent 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 Agent in standalone mode, or use a System Env configured for a single slave agent. The advantages and disadvantages of the two approaches are listed below.

Advantages of using standalone agent versus System Env:

- Testbench becomes light weight as System Env 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 Agent should to be replaced with System Env, or, multiple Slave Agents would need to be instantiated by you.
- The AXI system monitor cannot be used, which is part of the System Env.

Figure 7-1 Master DUT and Slave VIP - Usage With Standalone Slave Agent

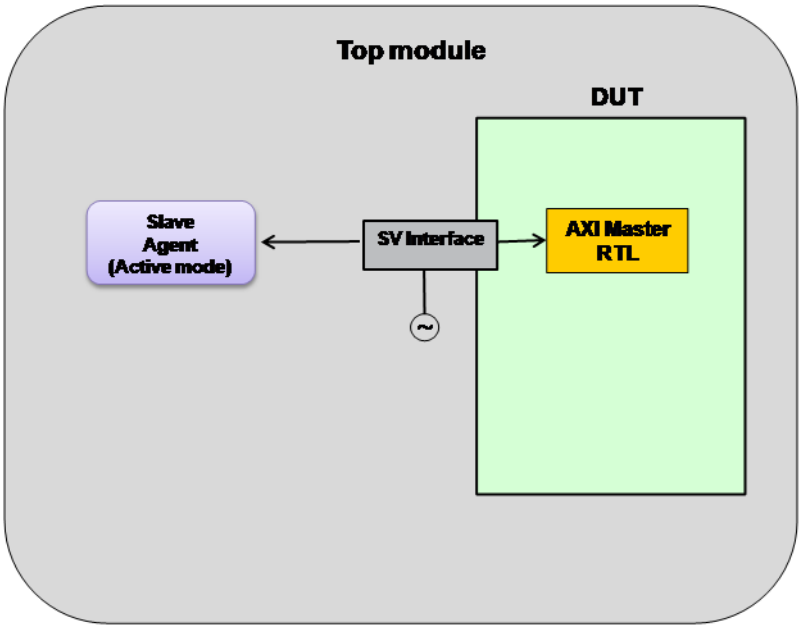
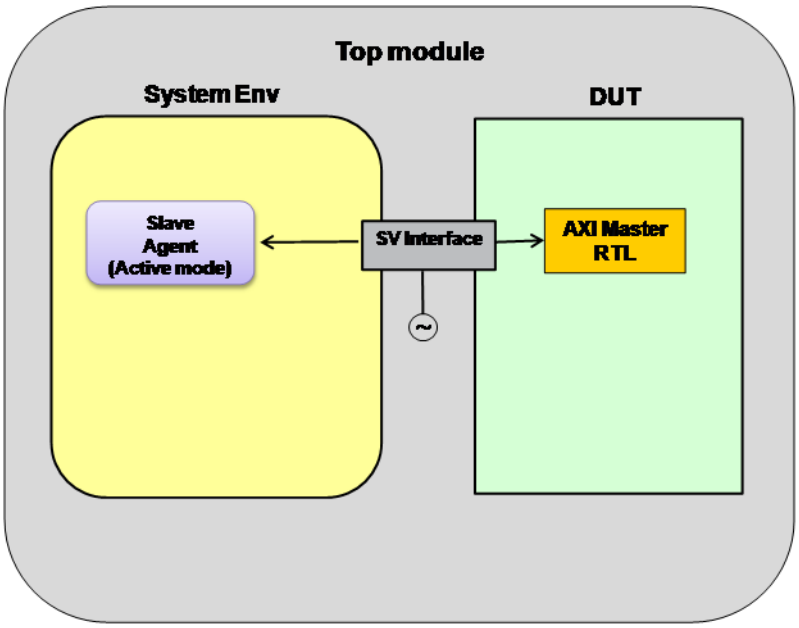


Figure 7-2 Master DUT and Slave VIP - Usage With System Environment



7.2 Testing a Slave DUT Using an UVM Master VIP

In this scenario, you are testing an AXI Slave DUT using an UVM AXI Master.Configure the AXI System Env to have one Master Agent, in active mode. The active master agent will generate AXI transactions for the Slave DUT. The Master Agent 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 Agent in standalone mode, or use a System Env configured for a single Master Agent.

The advantages and disadvantages of the two approaches are listed below.

Advantages of using standalone agent versus System Env:

- Testbench becomes light weight as System Env 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 Agent should be replaced with System Env, or, multiple Slave Agents 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 7-3 Slave DUT and Master VIP - Usage With Standalone Master Agent

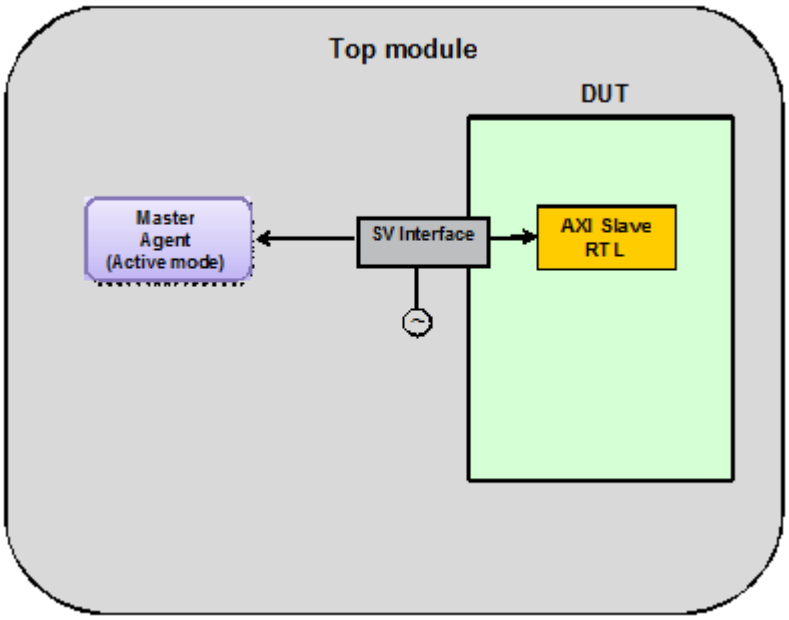
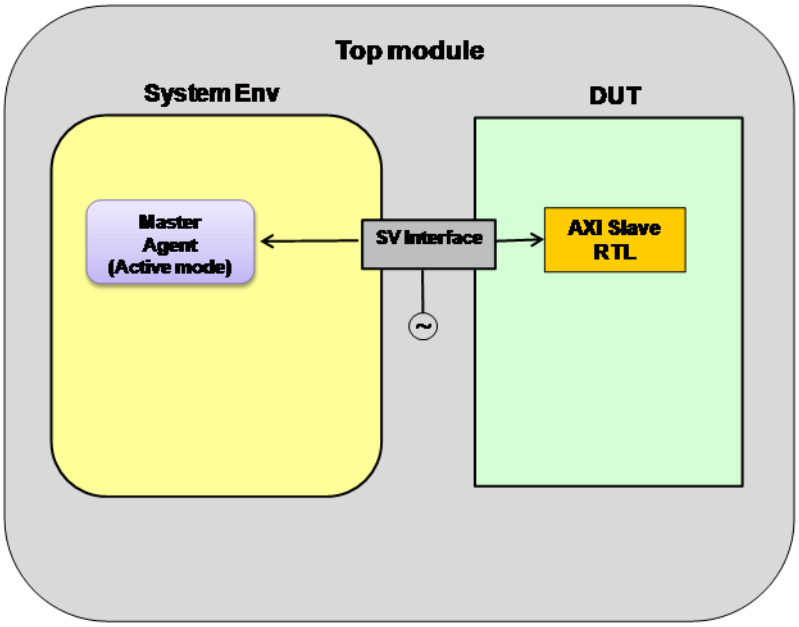


Figure 7-4 Slave DUT and Master VIP - Usage With System Environment



7.3 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 Env to have S master agents and M slave agents, in active mode. The active master agents will generate AXI transactions towards the interconnect slave ports, and active slave agents connected would respond to the transactions generated by interconnect master ports. The master and slave agents 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:

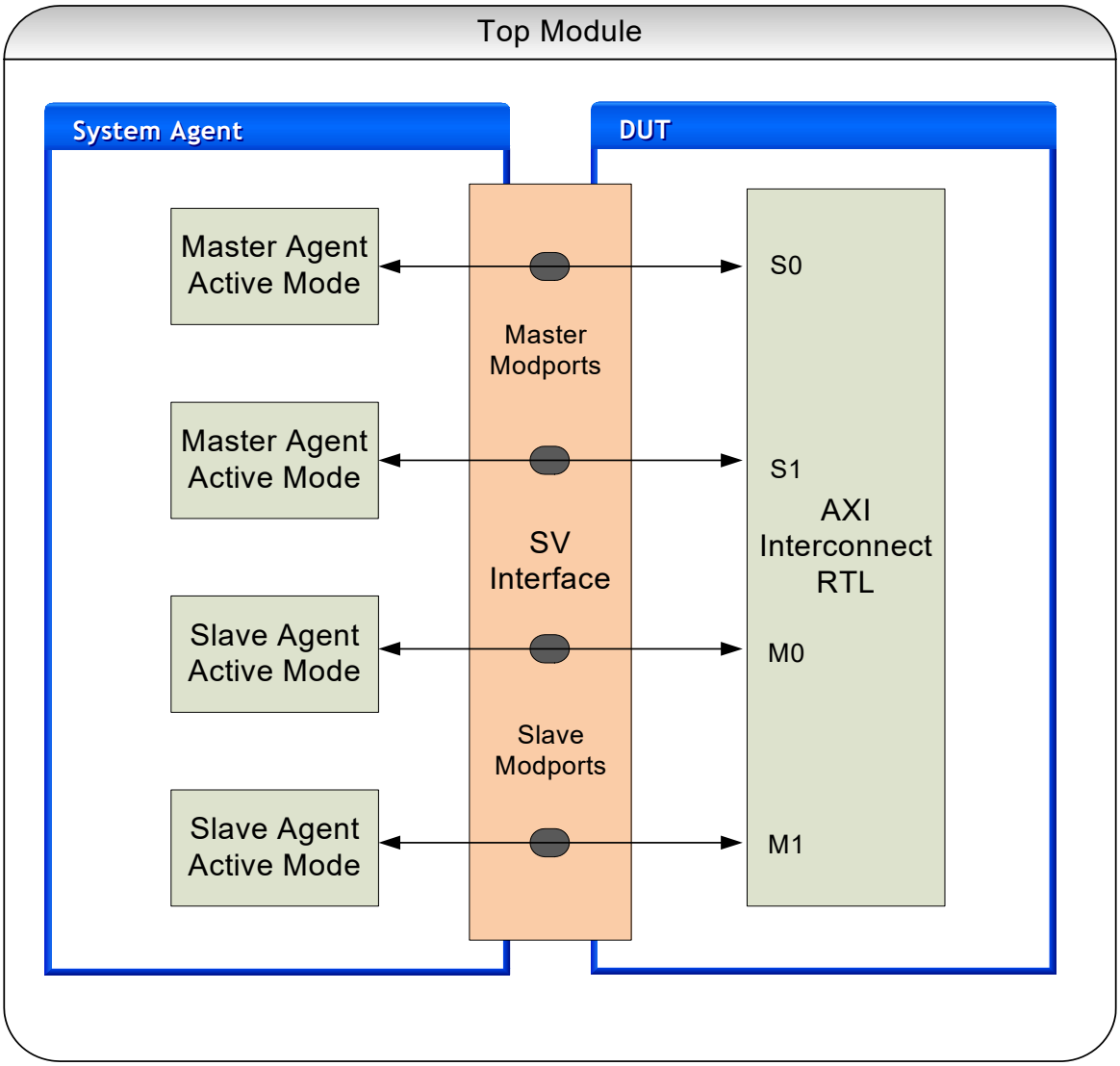
- sys_cfg.num_masters = 2;
- sys_cfg.num_slaves = 2;

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 7-5 shows the testbench setup.

Figure 7-5 Interconnect DUT with Master and Slave VIP (Active Mode)



7.4 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 Env to have M master agents and S slave agents, in passive mode. The passive master and slave agents 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:

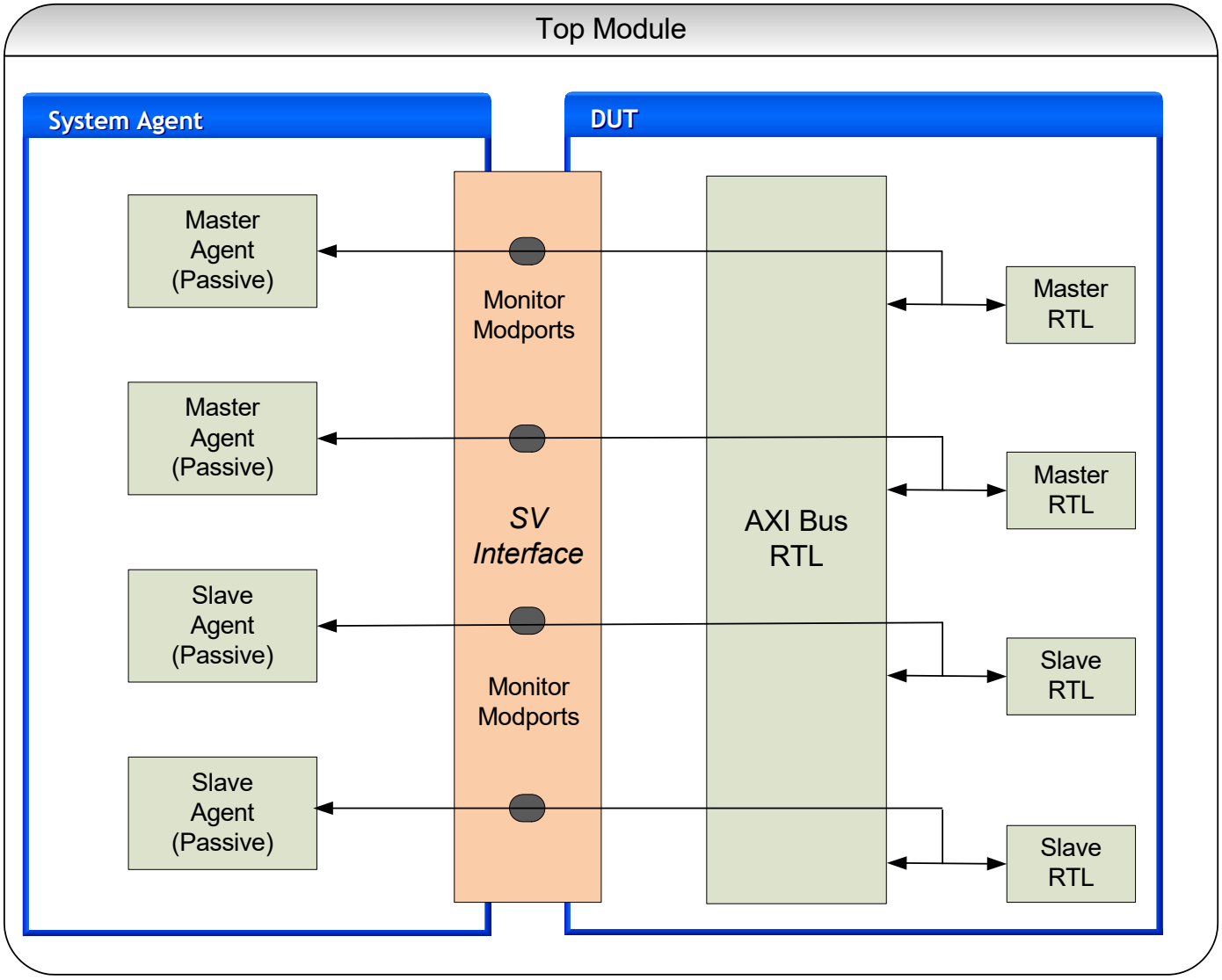
- sys_cfg.num_masters = 2;
- sys_cfg.num_slaves = 2;

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 7-6 shows this setup.

Figure 7-6 System DUT with Passive VIP

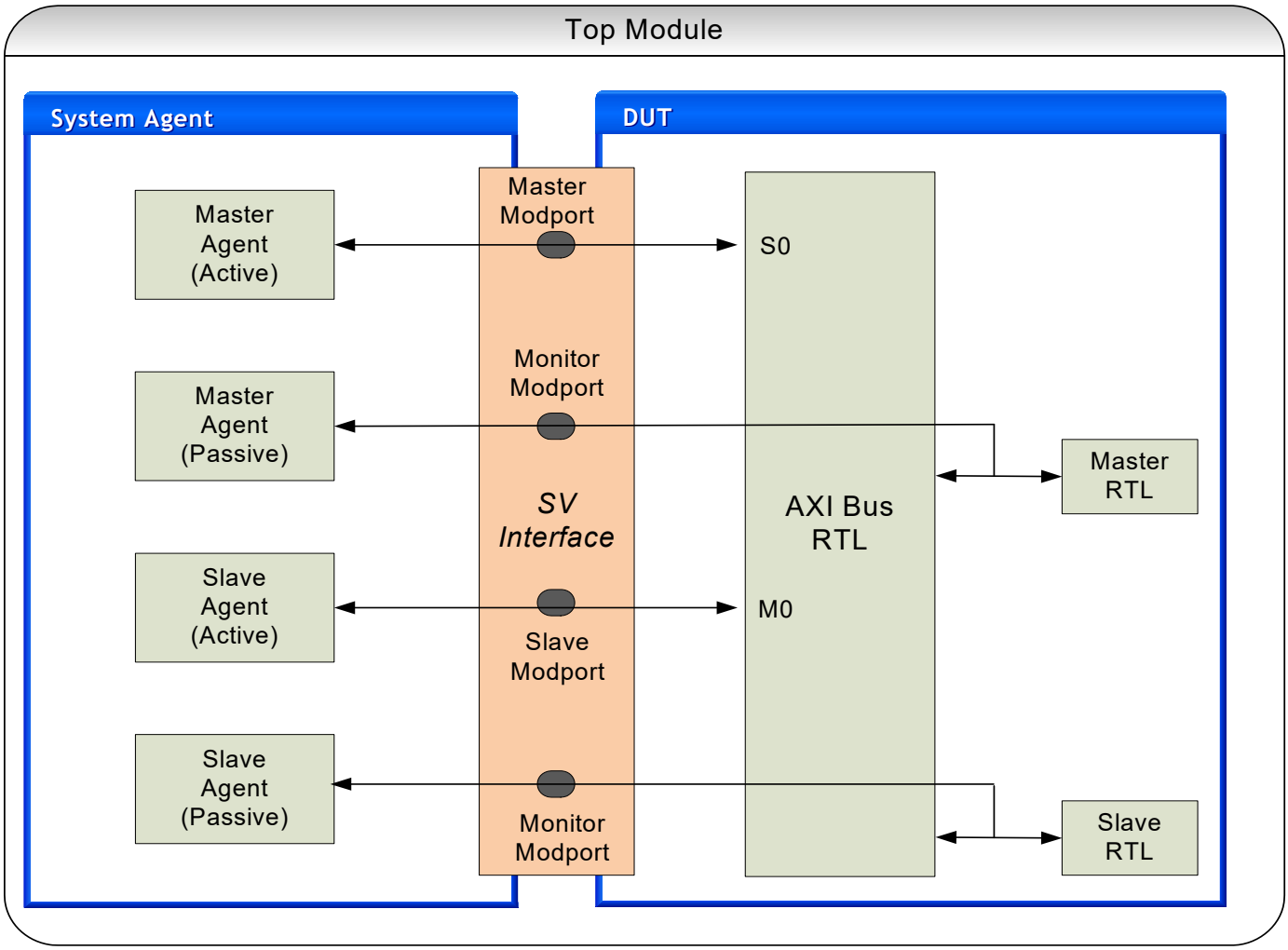


7.5 System DUT with Mix of Active and Passive VIP

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 Env to have two master agents and two slave agents. Configure the master agent connected to port S0, and slave agent connected to port M0 as active. Configure the master agent connected to port M1 and slave agent connected to port M1 as passive. All the agents would continue to perform passive functions such as protocol checking and coverage.

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



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 = 2;
- sys_cfg.num_slaves = 2;

Port configuration settings:

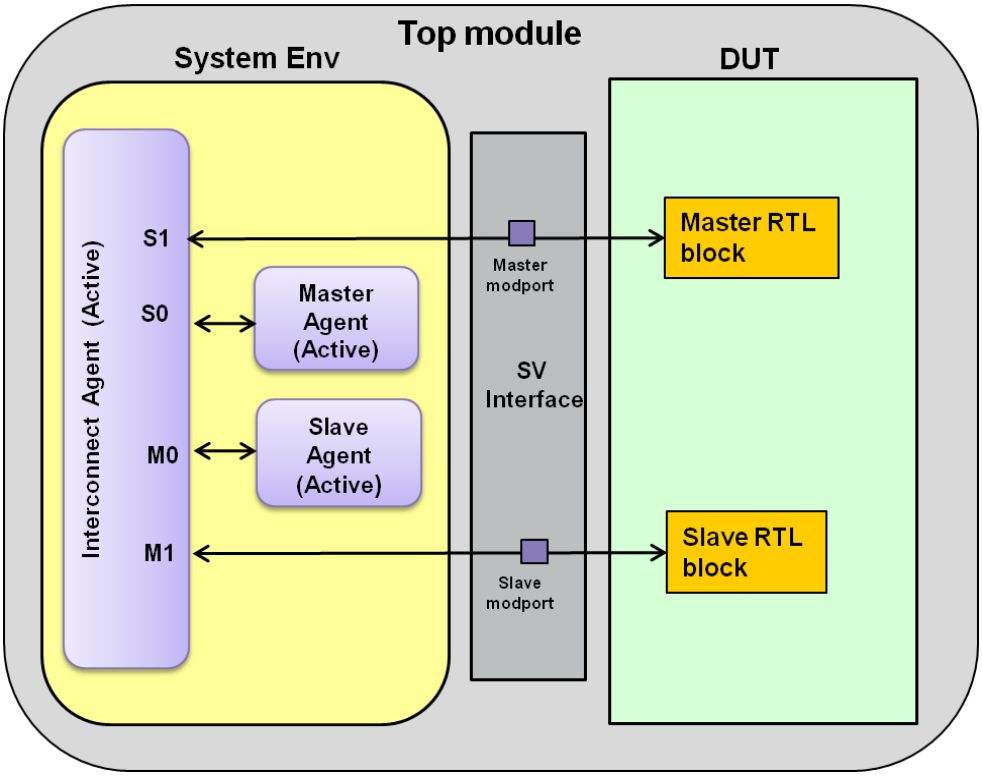
- sys_cfg.master_cfg[0].is_active = 1;
- sys_cfg.master_cfg[1].is_active = 0;
- sys_cfg.slave_cfg[0].is_active = 1;
- sys_cfg.slave_cfg[1].is_active = 0;

7.6 System DUT with Active Interconnect VIP

In this scenario, DUT is a system with multiple AXI masters and slaves. VIP is required to provide the background traffic on some ports, and to route the transactions between master and slave ports.

Assuming that the AXI System DUT has two master ports and two slave ports. VIP is required to provide the background traffic to ports S0 and M0. All the ports need to be monitored. Configure the AXI System Env to have one master agent and, slave agent and Interconnect Env. Configure the master agent, slave agent and Interconnect Env as active. The ports of Interconnect Env would continue to perform passive functions such as protocol checking and coverage. The passive functionality in master and slave agents connected to the Interconnect Env ports may be optionally disabled.

Figure 7-8 System DUT with Active Interconnect VIP



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

Assuming instance name of system configuration is "sys_cfg".

System configuration settings:

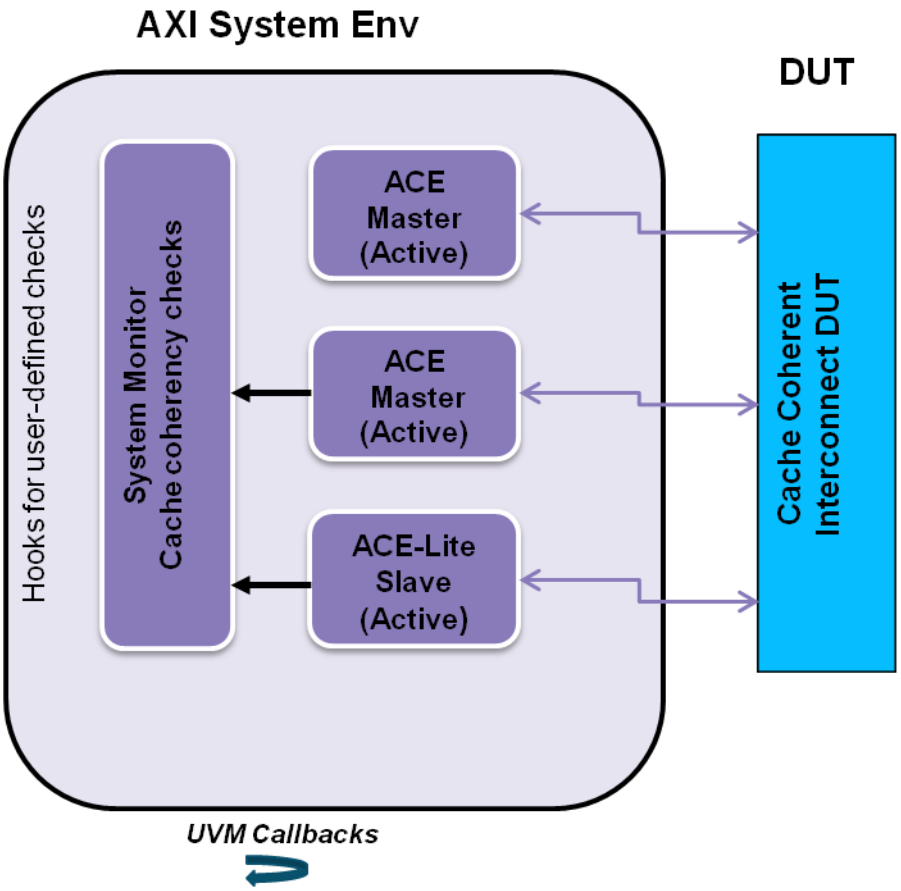
- sys_cfg.num_masters = 1;
- sys_cfg.num_slaves = 1;
- sys_cfg.use_interconnect = 1;

Port configuration settings:

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

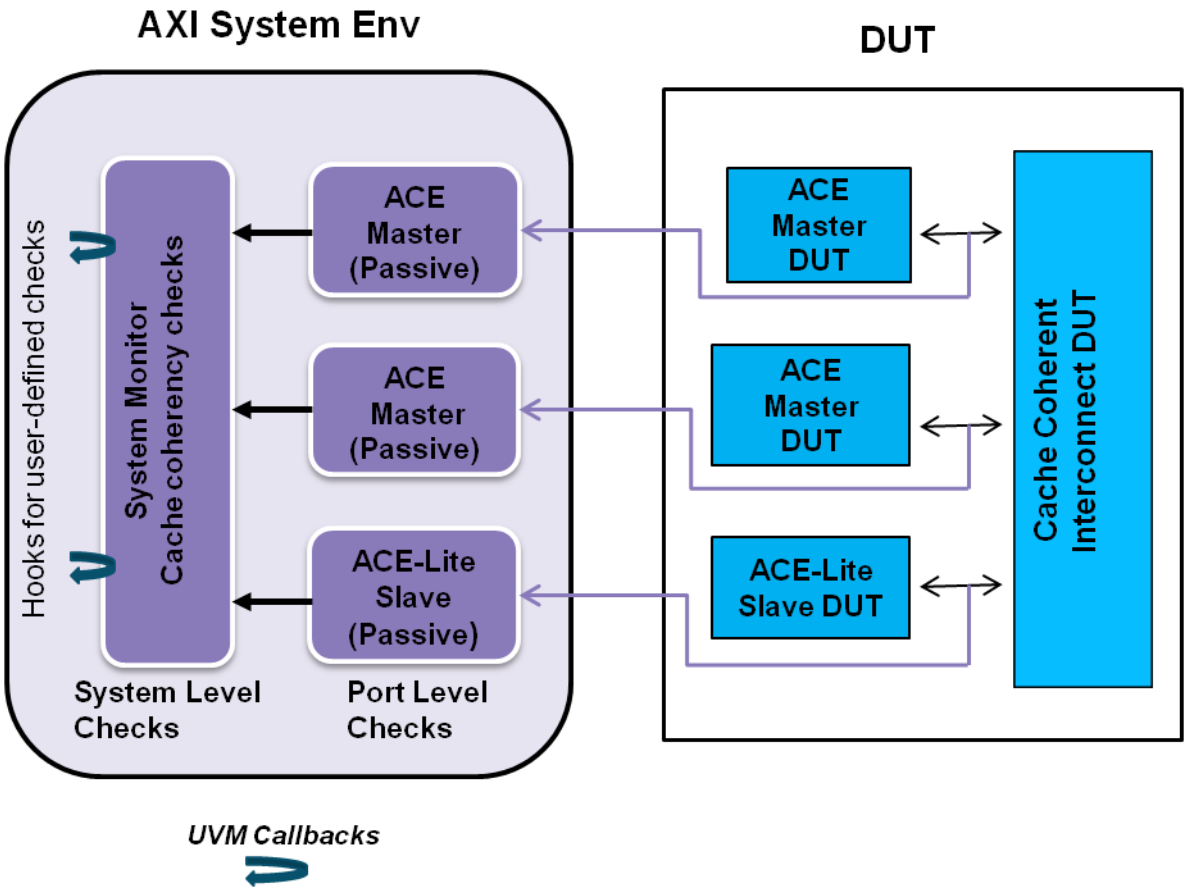
7.7 Interconnect DUT with System Monitor

Figure 7-9 Interconnect DUT with System Monitor VIP



7.8 System DUT with System Monitor

Figure 7-10 System DUT with System Monitor VIP



8
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:

8.1 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in Table 8-1

Table 8-1 SystemVerilog Example Summary

Example Name	Level	Description
tb_axi_svt_uvm_basic_sys	Basic	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests • A base test, which is extended to create a directed and a random test • The tests create a testbench environment, which in turn creates AXI System Env • AXI System Env is configured with one master and one slave agent **Note**: AXI UVM Basic example Quickstart is located at: *$DESIGNWARE_HOME/vip/svt/amba_svt/latest/examples/sverilog/tb_axi_svt_uvm_basic_sys/doc/tb_axi_svt_uvm_basic_sys/index_basic.html*
tb_axi_svt_uvm_basic_program_sys	Basic	The example demonstrates the usage of program block. It consists of the following: • A top-level module, which includes the user program, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests • The `axi_basic_tb.sv` file that contains the user program in the example • A base test, which is extended to create a directed and a random test The tests create a testbench environment, which in turn creates AXI System Env. AXI System Env is configured with one master and one slave agent.
tb_axi_svt_uvm_intermediate_sys	Intermediate	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests • A base test, which is extended to create a directed and a random test • The tests create a testbench environment, which in turn creates AXI System Env • AXI System Env is configured with one master and one slave agent • AXI UVM Scoreboard • Demonstrates how to override system constants • Coverage generation
tb_axi_svt_uvm_advanced_sys	Advanced	Not yet supported

The examples are located at:

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

8.2 Installing and Running the Examples

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

1. Install the example using the following command line:

   % cd <location where example is to be installed>

   % mkdir design_dir <provide any name of your choice>

   % $DESIGNWARE_HOME/bin/dw_vip_setup -path ./design_dir -e amba_svt/tb_axi_svt_uvm_basic_sys -svtb

   The example would get installed under:

   <design_dir>/examples/sverilog/amba_svt/tb_axi_svt_uvm_basic_sys

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

   a. Use the Makefile:

      Three tests are provided in the "tests" directory.

      The tests are:

      i. ts.base_test.sv
      ii. ts.directed_test.sv
      iii. ts.random_wr_rd_test.sv

      For example, to run test ts.directed_test.sv, do following:

      gmake USE_SIMULATOR=vcsvlog directed_test WAVES=1

      Invoke "gmake help" to show more options.

   b. Use the sim script:

      For example, to run test ts.random_wr_rd_test.sv, do following:

      ./run_axi_svt_uvm_basic_sys -w random_wr_rd_test vcsvlog

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

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

`$DESIGNWARE_HOME/vip/svt/amba_svt/latest/examples/sverilog/tb_axi_svt_uvm_basic_sys/README`

OR

`<design_dir>/examples/sverilog/amba_svt/tb_axi_svt_uvm_basic_sys/README`

8.2.1 Defines for Increasing Number of Masters and Slaves

The default max number of masters and slaves that can be used in an `axi_system_env` is 16. This can be increased up to a maximum value of 450. To use more than 16 masters and slaves in an AXI system, you need to define the macros `+define+SVT_AXI_MAX_NUM_MASTERS_<value>`, `+define+SVT_AXI_MAX_NUM_SLAVES_<value>`.

For example:

1. To use 380 AXI masters and 380 AXI slaves in a single AXI system env:

Add compile time options "+define+SVT_AXI_MAX_NUM_MASTERS_380 +define+SVT_AXI_MAX_NUM_SLAVES_380"

2. In the VIP configuration, do:

```
svt_axi_system_configuration::num_masters=380;
svt_axi_system_configuration::num_slaves=380;
```

8.2.2 Support for UVM version 1.2

While using UVM 1.2, note the below requirements:

- When using VCS version H-2013.06-SP1 and lower versions, you must define the USE_UVM_RESOURCE_CONVERTER macro. This macro is not required to be defined with VCS version I-2014.03-SP1 and higher versions.
- It is not required to define the UVM_DISABLE_AUTO_ITEM_RECORDING macro.

8.3 How to Provide Data and Response Information to the Slave After a Time Delay

As a default behavior, the slave driver expects the slave response sequence to return the slave response object in zero time, that is, without any delay after the sequencer receives object from the port monitor. This poses a problem where data and response information may not be available immediately and therefore cannot be given in zero time.

There are two alternatives to address this issue:

- Using svt_axi_transaction::suspend_response:

  The users can use the member svt_axi_transaction::suspend_response to achieve this. This requires the user to fork off the threads which wait for data to become available. This option is recommended when user can provide data & response information for all the beats in one shot to the slave driver, after a certain time delay, after receiving the slave response object from the port monitor.

- Using svt_axi_port_configuration::enable_delayed_response_port:

  This option is recommended when user cannot provide data and response information for all the beats in one shot to the slave driver, after a certain time delay, after receiving slave response object from the port monitor.

The following is the usage model for this feature.

1. This feature will be enabled using a port level configuration parameter (svt_axi_port_configuration::enable_delayed_response_port). By default, this feature will be disabled.
2. The slave monitor provides a handle to a transaction to the sequencer for the following events:
   a. Address phase of read/write transaction
   b. First beat of write data if the data arrives before address

   There is no change to this behavior.
3. The model will still require that the handle to the transaction is returned to the driver in zero time. This is to ensure that the driver has enough information to drive signals such as arready and awready. However, read data and read/write response related information need not be populated at this time. The mechanism detailed in (4) below may be used for the same.

4. A TLM port (uvm_blocking_get_port) is added to the slave driver, which enables the user to supply information related to read data and read/write response at a later point in time. This port is connected to a corresponding 'put' port (delayed_response_request_port) in the sequencer by the VIP.

   When the data and response information is available, the slave sequence can provide the slave response object using this 'put' port in the sequencer. The transaction supplied through this TLM port should NOT be the same transaction handle given by the monitor to the sequencer, but it should be its copy. The information related to data and response, refers to the following properties in svt_axi_transaction:

   a. data[]
   b. rresp[]
   c. coh_rresp[]
   d. rvalid_delay[]
   e. bresp
   f. bvalid_delay

   For READ transaction, a user will have the flexibility to input any number of beats at any point of time. The model detects an availability of data by checking the array size of rresp[] field. For example, if the size of array rresp[] is 1, it indicates that the first beat is available. If the size is two, it indicates that the first two beats are available and so on. The array sizes of all the above properties should be consistent and the responsibility to ensure this lies on the user. The model would allow the transaction to be submitted back to the p_sequencer multiple times, if partial beat information was received. It is user's responsibility to make sure that eventually all the read data required for read transaction to complete, is provided to the model.

   For WRITE transactions, a user has the ability to return the response (bresp) at any point of time.

5. When the above feature is enabled, the constraints will not enforce array sizes of the above mentioned parameters to be consistent with burst length, since the array sizes will be used to recognize availability of data.
6. When this feature is enabled, responses sent through the seq_item_port must have data.size() and rresp.size() as 0. That is, these arrays must be empty. In other words, all the response and data information must be sent through the delayed_response_request_port of the sequencer.
7. If a transaction is randomized before it is sent through the delayed_response_request_port, the property svt_axi_transaction::is_delayed_response_xact must be set for the transaction (this transaction will be the copy of the transaction received from the monitor since only a copy is allowed on this port as mentioned below).
8. The transaction provided through the delayed_response_request_port must be a copy of the original transaction and not a reference. The following fields of the original transaction received from the monitor and the delayed response should match:

   svt_axi_transaction::object_id

   svt_axi_transaction::id

   svt_axi_transaction::addr

Sample code:

```
virtual task body();
      svt_axi_slave_transaction req_resp;
      forever
      begin
         p_sequencer.response_request_port.peek(req_resp);
         // Randomize the initial response. The response will be completed
         // after some time has elapsed, when the block that is external to the
         // slave returns data.
         status = req_resp.randomize (…);
         fork
           svt_axi_slave_transaction delayed_resp;
          delayed_resp = svt_axi_slave_transaction::type_id::create("delayed slave
          response",p_sequencer);
          delayed_resp.copy(req_resp);
           begin
               if((delayed_resp.xact_type == svt_axi_slave_transaction::WRITE) &&
                   (delayed_resp.addr_status != svt_axi_transaction::INITIAL))
              //Provide the data to an external entity which will supply back the write
                response
              put_write_transaction_data_to_external_block(delayed_resp);
              //After write response is available, provide it to
                delayed_response_request_port port
              p_sequencer.delayed_response_request_port.put(delayed_resp);
              // Provide delayed write response
               else
               begin
               //Get the read data from an external entity
               get_read_data_from_external_block_to_transaction(delayed_resp);
               //After read data and response is available, provide it to
                 delayed_response_request_port port
               p_sequencer.delayed_response_request_port.put(delayed_resp);
               // Provide delayed read data.
               end
           end
         end
       join_none
      $cast(req, req_resp);
     //User still needs to provide slave response object back to the slave driver in
       zero time.
     // This is to ensure that the driver has enough information to drive signals such
        as arready and awready.
     //Response and data information can be provided at a later time when
       svt_axi_port_configuration::enable_delayed_response_port is set.

       `uvm_send(req)
```

8.4 How to Disable Objection Management by VIP, and Allow Testbench to Manage Objections

The objection management in AXI VIP components is controlled through a system configuration property svt_axi_system_configuration::manage_objections_enable. This parameter decides whether the objections will be raised and dropped by the drivers of VIP components. If set, the VIP will raise an objection when it receives a transaction in the input port of the driver and will drop the objection when the transaction

completes. If not set, the driver will not raise any objection and complete control of objections is with the user. By default, the configuration parameter will be set, that is, VIP will raise and drop objections.

The following example code shows how a user could potentially control objections from the testbench through callbacks:

```
class cust_svt_axi_system_interconnect_objection_control_callback extends
svt_axi_interconnect_callback;
  // Counter to keep track of number of transactions
  int counter = 0;
  // Handle to the environment class where the callback is instantiated
  axi_env sys_env;
  // Set by env in the run_ph. Need this handle to raise objections on this based on
     transactions
  uvm_phase env_run_ph;

  function new
      ( string name = "cust_svt_axi_system_interconnect_objection_control_callback"
                      axi_env sys_env);
    super.new(name);
    this.sys_env  = sys_env;
  endfunction

  virtual function void post_input_port_get(svt_axi_interconnect axi_interconnect,
                                            svt_axi_ic_slave_transaction xact);
      counter++;
     // Raise objections only for 1000 transactions.
      if (counter < 1000) begin
        env_run_ph.raise_objection(axi_env);
        fork
          wait(`SVT_AXI_XACT_STATUS_ENDED(xact));
          env_run_ph.drop_objection(axi_env);
        join_none
     end
  endfunction
endclass
```

8.5 How to Control the Forwarding of Barrier and Cache Maintenance Transactions to Downstream Slaves by the Interconnect VIP

Forwarding of barrier and cache maintenance transactions to downstream ACE-LITE slaves by the interconnect are determined by the following two configuration parameters:

```
svt_axi_interconnect_configuration::forward_barriers
svt_axi_interconnect_configuration::forward_cache_maintenance_transactions
```

For a detailed description of the two parameters, see the Class Reference HTML documentation.

8.6 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, see the documentation of `svt_axi_system_configuration::allow_slaves_with_overlapping_addr` in AXI Class Reference.

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, see the documentation of `svt_axi_system_configuration::allow_slaves_with_overlapping_addr` parameter.
- The user needs to 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 agents 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
   uvm_config_db#(svt_mem)::set(this, "axi_system_env.slave[0]", "axi_slave_mem",
                                s0_s1_shared_mem);
   uvm_config_db#(svt_mem)::set(this, "axi_system_env.slave[1]", "axi_slave_mem",
                                s0_s1_shared_mem );
```

- 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.

8.7 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 more details, see the AXI Class Reference.

8.8 Why the User Needs to Disable Auto Item Recording

If you are using AHB UVM or AXI UVM Verification IP, you need to define a macro named UVM_DISABLE_AUTO_ITEM_RECORDING. This section describes why this macro needs to be defined, and what are its implications if a user defined driver and sequencer also exist in the same environment.

AXI and AHB protocols are pipelined protocols. In pipelined protocols, driver needs to initiate the next transaction before the previous transaction completes. Thus, the VIP driver indicates seq_item_port.item_done() much before the transaction is completed on the bus, so that the sequencer can provide next sequence item to the driver. Driver does not wait for a transaction to complete before calling seq_item_port.item_done().

For AXI, seq_item_port.item_done() is called as soon as the driver accepts a transaction from the sequencer. For AHB, seq_item_port.item_done() is called when penultimate beat address of the current transaction is accepted by the slave. The VIP explicitly marks end of transaction when the transaction actually completes on the interface, instead of letting UVM do it. Hence, VIP needs to define UVM_DISABLE_AUTO_ITEM_RECORDING macro.

If the environment contains a user defined driver/sequencer, and the macro UVM_DISABLE_AUTO_ITEM_RECORDING is defined, user needs to make sure that the driver explicitly marks end of transaction when transaction actually completes on the interface. For example, for a non-pipelined protocol, user can call req.end_tr() in the driver code after calling seq_item_port.item_done(), assuming that seq_item_port.item_done() is called only after the transaction is complete. Alternatively, user can call req.end_tr() in the corresponding sequence, after the sequence unblocks based on seq_item_port.item_done().

For pipelined protocol, user needs to wait till the transaction is complete on the bus, before calling req.end_tr().

Code snippet of the driver (assuming non-pipelined protocol):

```
seq_item_port.item_done();
req.end_tr();
```

Code snippet of the sequence (assuming non-pipelined protocol):

```
`uvm_do(req);
req.end_tr();
```

Note VIPs for pipelined and non-pipelined protocols are designed to work correctly when UVM_DISABLE_AUTO_ITEM_RECORDING macro is defined.

8.9 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.

Note

The Interconnect VIP is not a behavioural 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.

8.10 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);
   ```

8.11 Data Integrity Checks

The following data integrity checks are performed by the system monitor. For specific information on the checks, see the class reference documentation of 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

8.11.1 Memory Based Data Integrity Check

The data_integrity_check performs a check on data integrity by comparing data in a write or read transaction on the master side, with the data in the same address location in AXI Slave VIP memory. This check is performed only when svt_axi_system_configuration::posted_write_xacts_enable is not set.

Another related configuration member is svt_axi_port_configuration::memory_update_for_read_xacts_enable. This variable must be 1 if a slave is DUT (that is Slave VIP is in passive mode) and if DUT can return data for locations not written through previous WRITE transactions.

The data_integrity_check is bypassed in the following situations:

- A WRITE transaction to an overlapping address is detected during the life time of the transaction being checked.
- 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 being checked.
- 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()

8.11.2 Transaction Correlation Based Data Integrity Check

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 between the correlated master and slave transactions. 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_xact_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

Other configuration parameters which may be optionally set are:

- svt_axi_port_configuration::is_source_master_id_and_dest_slave_id_same

If a DUT does not support ID based transaction routing (which means, there is no defined relationship between IDs generated at the master and ID of corresponding transactions sent to slave), the system monitor can still correlate master and slave transactions based on address and data. Providing ID based correlation gives additional hints to narrow down on the correlated transactions and makes the matches more accurate. If the DUT supports this, it is recommended that the system monitor be configured with the appropriate information about transaction IDs using above mentioned configuration members.

Some DUTs support partial ID based correlation. For example, ID of transactions from some masters have a defined relationship to the ID of corresponding slave transactions. In this scenario, ID based correlation can be enabled on per port level by using the following configuration parameter:

- `svt_axi_port_configuration::id_based_xact_correlation_enable`

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. You can perform custom checks in the callback based on the following:

`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, but it is meant to supplement it by addressing the situations where memory based checking has limitations and need to be bypassed. It also provides a powerful mechanism for users to do custom functional as well as performance checks based on the callback provided.

The system monitor supports correlation of master transactions to slave transactions when Interconnect converts FIXED bursts to INCR bursts as below:

- Master sends a FIXED burst of a given burst size and burst length
- Interconnect converts this into 'burst length' number of INCR transactions, each of which has a burst size of 8-bit and burst length based on the burst size of the master transaction. For example, if you consider that the master sends a FIXED burst with burst_size of 32 bits and length of 8. This is converted into 8 INCR transactions of burst_size 8-bit and burst_length 4 (corresponding to burst_size of 4 bytes of the original transaction).

In above case, the system monitor can associate the INCR slave transactions to FIXED burst master transactions. The protocol check `master_slave_xact_data_integrity_check` performs this check.

8.12 Setting up Secure and Non-Secure access mechanism for AXI-ACE Master

By default when cacheline is written or read by VIP master agent 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 agent

```
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 agent 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 agent `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 supports 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.

8.13 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
```

8.13.1 Snoop Address Translation

Some interconnect DUTs support a feature where there is a translation in snoop address. Typically, higher order bits of the snoop transaction address may be truncated. Any such transformation can be indicated through the following callback:

```
svt_axi_system_monitor_callback::snoop_transaction_user_addr
```

You must update the following parameters in the system transaction to reflect the actual snoop address sent:

```
svt_axi_system_transaction::expected_snoop_addr
svt_axi_system_transaction::expected_snoop_filter_addr
```

The second attribute is expected to have the same value as the first. However, the first attribute (`expected_snoop_addr`) is not present for transactions that do not have a snoop such as WRITEBACK transactions. The second attribute (`expected_snoop_filter_addr`) would be present and must be updated for such transactions because it has an impact on the snoop filter.

8.14 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
```

8.15 Exclusive Access Support

8.15.1 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;

8.15.2 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
```

8.15.3 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.

8.16 Backdoor Cache Access Methods

Cache is modeled 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 )
```

8.17 AXI4 Stream Protocol

8.17.1 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:

8.17.1.1 Master Agent

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

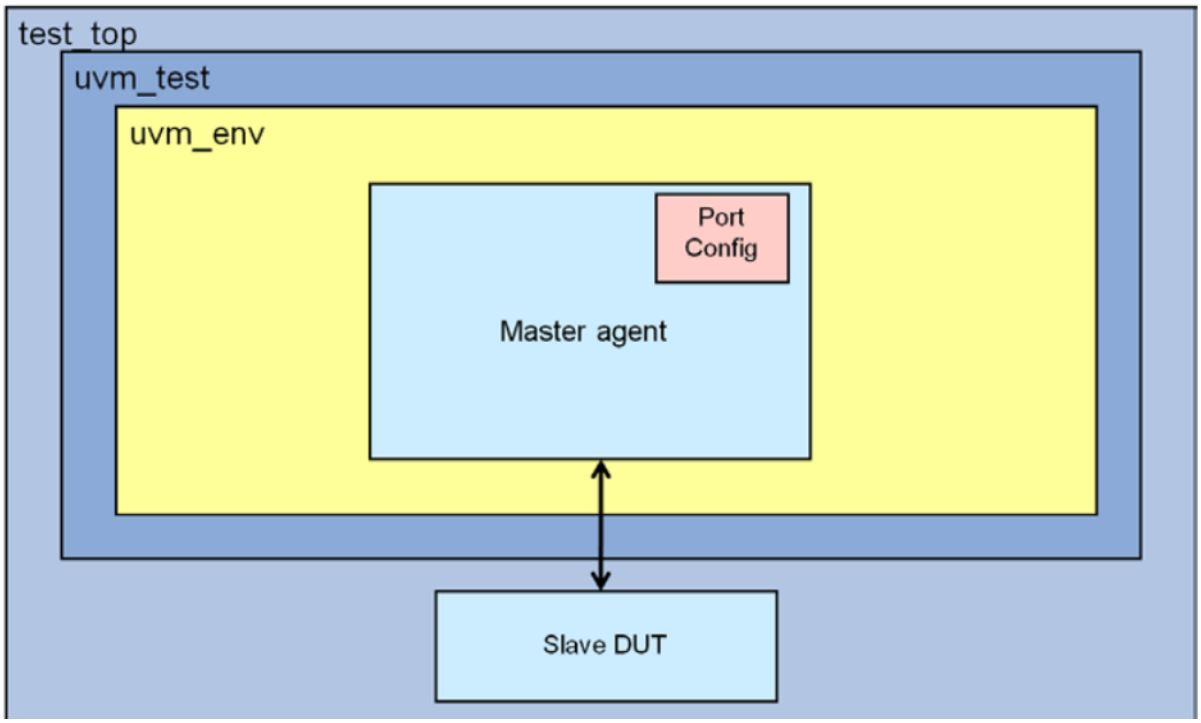
The Master Agent is configured using a port configuration, which is available in the system configuration. The port configuration should be provided to the Master Agent in the build phase of the test.

Within the Master Agent, 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 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 8-1 Usage With Standalone Master Agent



8.17.1.2 Slave Agent

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

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

In the Slave Agent, 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.

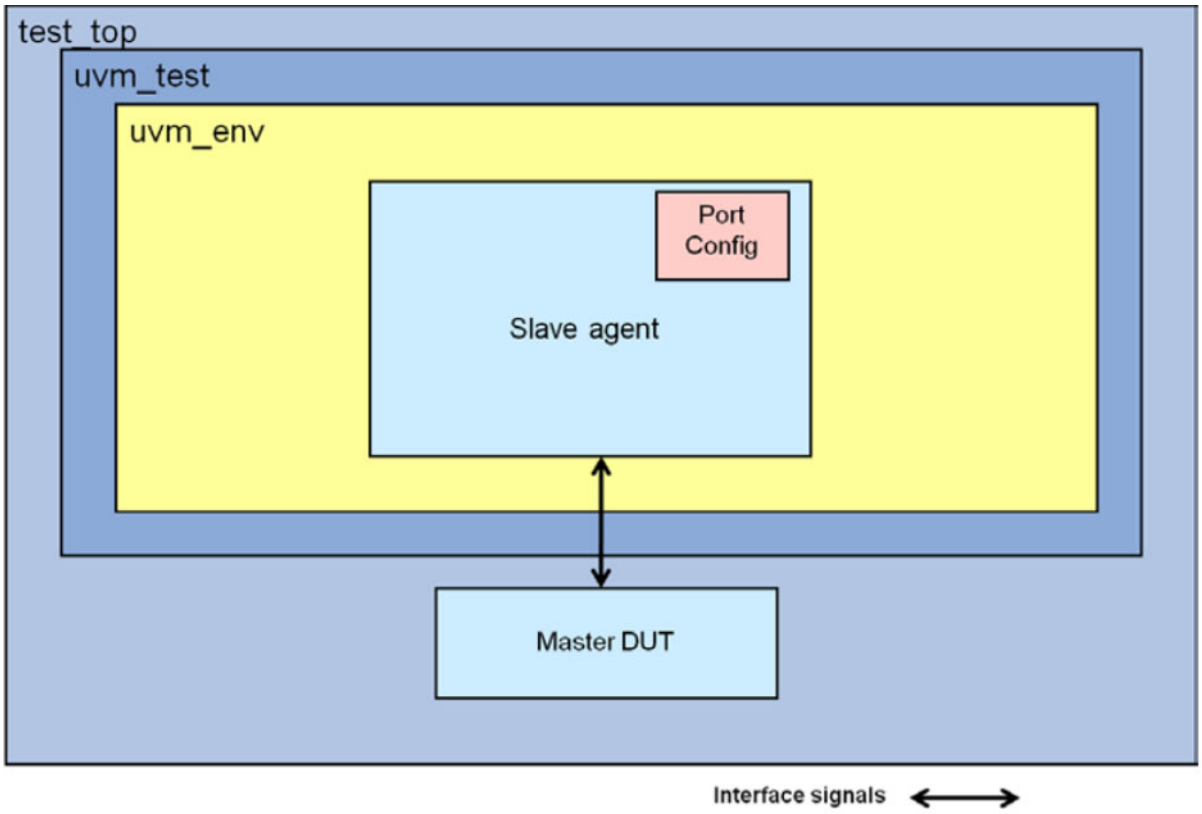
The slave driver expects the slave response sequence to,

- 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 agent 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 8-2 Usage with Standalone Slave Agent



8.17.1.3 Agents 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 agent. This is because when the agent 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.

8.17.1.4 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.

8.18 Steps to Integrate the uvm_reg With AXI VIP

The following are the steps to integrate the uvm_reg flow with AXI Master Agent:

1. Generate the System Verilog file for the register definition, using the ralgen utility.

   `ralgen -uvm -t axi_regmodel  <>.ralf`, this will generate a System Verilog file with register definition

2. Instantiate and create the RAL/uvm_reg model in the `uvm_env` and pass that handle to the AXI Master agent.

```
// Declare RAL model.
  ral_sys_axi_intermediate_slave regmodel;
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
   ..
 /** Check if regmodel is passed to env if not then create and lock it. */
  If (regmodel == null) begin
  regmodel = ral_sys_axi_intermediate_slave::type_id::create("regmodel");
  regmodel.build();
  regmodel.set_hdl_path_root(hdl_path);
  `uvm_info("build_phase", "Reg Model created", UVM_LOW)
  regmodel.lock_model();
  end
  uvm_config_db#(uvm_reg_block)::set(this,"axi_system_env.master[0]", "axi_regmodel",
regmodel);
..
```

```
endfunction : build_phase
```

3. Call the reset() function of the regmodel from the `reset_phase` of `uvm_env`.

```
// Reset the register model
task reset_phase(uvm_phase phase);
   phase.raise_objection(this, "Resetting regmodel");
   regmodel.reset();
   phase.drop_objection(this);
endtask
```

4. To enable the `uvm_reg` adapter of the AXI Master agent, user need to do the following

   Set the `uvm_reg_enable, svt_axi_port_configuration` attribute to 1 for the desired AXI agent.
   `this.master_cfg[i].uvm_reg_enable= 1;`

5. Modify the `uvm_reg` tests, if required and execute them

You can find the complete example in the VIP installation (`tb_axi_svt_uvm_intermediate_ral_sys`)

You can download the example using the `dw_vip_setup_utility` (see Installing and Running the Examples.

8.19 Design Specific Coherent to Snoop Transaction Association

The AXI system monitor associates coherent transactions to snoop transactions. Since there is no implication of the snoops for the corresponding coherent transactions on the interface, hence the information is derived from the activities performed on the interface. There are many DUT specific scheduling behaviours that impact the behaviour in which the system monitor associates coherent transactions to snoop transactions. The snoop transactions can be determined by providing the schedule information of the NOC, if the signals within the NOC are tapped and corresponding transactions provided to the system monitor.

8.19.1 Solution Description

The solution description for design specific coherent to snoop transaction is described in the following example. For example, when the VIP cannot associate snoops to coherent transactions appropriately. Consider the following steps:

A topology where master[0] is a full ACE port and master[1] is an ACE-LITE port.

1. The DUT is configured to generate a CLEANINVALID snoop for READONCE and CLEANINVALID/MAKEINVALID for WRITEUNIQUE transactions.
2. Master[1] issues a READONCE transaction. While this is outstanding a WRITEUNIQUE is also issued from master[1] to the same address.
3. The snoops (CLEANINVALID snoop) corresponding to the READONCE are sent first to master[0], followed by those (MAKEINVALID snoop) corresponding to WRITEUNIQUE.
4. The WRITEUNIQUE completes first, after which, the system monitor tries to associate the snoops.
5. The system monitor associates the CLEANINVALID snoop to WRITEUNIQUE transaction.
6. After the READONCE completes, it tries to associate the MAKEINVALID, but it cannot because MAKEINVALID is not configured as a valid snoop for READONCE transactions.

The system monitor tries to associate snoops in the order in which the responses complete. However, this is not the scheduling sequence maintained by the NOC. If there is visibility into the scheduling of the NOC, then the system monitor can use that information. The signals can be connected from the output of the scheduler to a VIP and provided to the corresponding transactions of the system monitor using a TLM port. The system monitor uses this information to correlate transactions. In the example, the output of the scheduler delivers the READONCE transaction followed by the WRITEUNIQUE transaction. This information is used by the system monitor to determine the READONCE transaction previously associated before the WRITEUNIQUE transaction, thereby verifying the snoops ahead in the queue are associated to the transactions which are ahead as per the scheduler's output to maintain the sequence.

8.19.2 User Interface

The `svt_axi_port_monitor` class has the following API added to push transactions from the scheduler to the port monitors. This is similar to the existing API (`svt_axi_port_monitor::push_coherent_xact_to_port_monitor`)

/** Task that can be used to drive an exernal coherent transaction to the port monitor. This transaction must be sampled from the output received from the scheduler within the interconnect, since the transaction sequence are used as input for this API and is used by the system monitor to associate snoops to coherent transactions */

```
 extern virtual function void
push_coherent_xact_from_ic_scheduler_to_port_monitor(svt_axi_transaction xact);
```

8.20 Single Outstanding Transaction Per AxID

The Master VIP component supports a feature where there can only be a single outstanding transaction for a given AxID value for Non-Device and Non-DVM transactions.

You must configure the following members to enable this feature:

- `svt_axi_port_configuration::single_outstanding_per_id_enable`
- `svt_axi_port_configuration::single_outstanding_per_id_kind`

For more details on each member, see AXI Class Reference HTML documentation.

Added the following protocol check for this feature:

`axi_checker::perform_non_dvm_non_device_with_overlap_id_check`

8.21 Interleaved Port Support

AXI VIP master or slave VIP components support interleaved set of addresses. The master components and slave components can be grouped together in interleaved group. Master or slave components within an interleaved group will support a unique non-overlapping set of address ranges. For example, assume that there are two masters within an interleaved group. Master 0 supports address range 0 to 63, Master 1 supports address range 64 to 127, Master 0 supports address range 128 to 191, Master 1 supports address range 192 to 255, and so on. It means that the Master 0 generates transactions with addresses 0 to 63, 128 to 191 and so on. The interconnect snoops Master 0 for addresses 0 to 63, 128 to 191 and so on.

- AXI System monitor issues error if it observes snoop transaction for an address that is issued to a master which does not support the address range.
- AXI System monitor issues error if it observes snoop transaction issued within the same interleaved group.

- AXI System Monitor checks whether the coherent address observed on master port falls within the interleaving address range. If not, System Monitor generates error.
- AXI System Monitor checks whether the address that is received on slave port falls in the interleaving address range. If the address does not lie in the interleaving address range for this Slave port, then the AXI System Monitor generates error.

In active mode, Master VIP checks if the coherent address generated by it falls in the interleaving address range. If the address does not lie in the interleaving address range for this master port, then the Master VIP generates error through `is_valid`() check. The Master VIP continues to transmit the transaction.

In active mode, Master VIP checks if the snoop address received by it falls in the interleaving address range. If the address does not lie in the interleaving address range for this master port, then the Master VIP generates error through `is_valid`() check.

In passive mode, Master VIP checks whether the coherent or snoop address observed on this port falls within the interleaving address range. If not, then the Master VIP generates error.

In active and passive mode, Slave VIP checks if the address that is received by it falls in the interleaving address range. If the address does not lie in the interleaving address range for this Slave port, then the Slave VIP generates error.

The following configuration members must be programmed to enable this feature. For more details on each member, see AXI Class Reference HTML documentation:

- `svt_axi_port_configuration::port_interleaving_enable`
- `svt_axi_port_configuration::port_interleaving_size`
- `svt_axi_port_configuration::port_interleaving_group_id`
- `svt_axi_port_configuration::dvm_sent_from_interleaved_port`
- `svt_axi_port_configuration::device_xact_sent_from_interleaved_port`
- `svt_axi_port_configuration::port_interleaving_index`
- `svt_axi_port_configuration::port_interleaving_for_device_xact_enable`

Example 8-1 Use Case

- Scenario 1

  Two interleaved ACE master (64 bytes Interleaving Size).

  You must configure the *cust_svt_axi_system_configuration* file as shown below:

```
master_cfg[0].port_interleaving_enable  = 1;
master_cfg[0].port_interleaving_group_id = 2;
master_cfg[0].dvm_sent_from_interleaved_port = 0;
master_cfg[0].port_interleaving_size =64;
master_cfg[0].port_id = 0;
master_cfg[0].port_interleaving_index = 0;
master_cfg[1].port_interleaving_enable  = 1;
master_cfg[1].port_interleaving_group_id = 2;
master_cfg[1].dvm_sent_from_interleaved_port = 0;
master_cfg[1].port_interleaving_size =64;
master_cfg[1].port_id = 1;
master_cfg[1].port_interleaving_index = 1;
```

  The VIP will take care of the interleaving based on the above configuration inputs In the above use case, VIP will generate error if address[6] == 0 address comes on port 1 and also if address[6]==1 comes on port 0.

- Scenario 2

  Two slaves are interleaved with interleaving size 512 bytes.

  You must configure the VIP as shown below:

```
slave_cfg[0].port_interleaving_enable  = 1;
slave_cfg[0].port_interleaving_group_id = 2;
slave_cfg[0].dvm_sent_from_interleaved_port = 0;
slave_cfg[0].port_interleaving_size =512;
slave_cfg[0].port_interleaving_index = 0;
slave_cfg[0].port_id = 0;
slave_cfg[1].port_interleaving_enable  = 1;
slave_cfg[1].port_interleaving_group_id = 2;
slave_cfg[1].dvm_sent_from_interleaved_port = 0;
slave_cfg[1].port_interleaving_size =512;
slave_cfg[1].port_id = 1;
slave_cfg[1].port_interleaving_index = 1;
```

  In the above scenario, slave VIP will generate an error if address[9] == 1 is observed on port 0. This is because slave[0] is configured with port_interleaving_index = 0. Therefore, port 0 is first in the interleaving scheme and expected to receive addresses only with address[9]== 0.

AXI VIP ensures that WriteUnique/WriteLineUnique transactions from one interleaved port do not overlap in time with WriteBack/WriteClean/WriteEvict transactions from the same or another interleaved port. This requirement arises from the fact that the interleaved ports within the same group are considered as a single master component by the interconnect. This behavior is applicable to all the interleaved ports within the interleaved group, irrespective of the address.

8.22 Master to Slave Path Access Coverage

This feature allows you to identify the master to slave paths covered during the simulation. The cover group name defined for this purpose is `trans_cross_master_to_slave_path_access`. Note that this coverage works in conjunction with the AXI Complex Memory Map feature. For more details on covergroup, see AXI Class Reference HTML documentation.

Perform the following steps to enable this feature:

1. Enable the covergroup by setting port configuration `svt_axi_port_configuration::trans_cross_master_to_slave_path_access_cov_enable` to 1.
2. Enable the AXI Complex Memory Map feature by setting the system configuration `svt_axi_system_configuration::enable_complex_memory_map` to 1.
3. Define the macro `SVT_AMBA_PATH_COV_DEST_NAMES` with the names of the slaves in the system. These are user-defined names, which identify the slave ports within the system. These names will be used in the bin names of the covergroup.

   For example,

```
`define SVT_AMBA_PATH_COV_DEST_NAMES slave_0, slave_1, slave_2, slave_3, slave_4, slave_5
```

4. In Master configuration, assign the master name to `svt_axi_port_configuration::source_requester_name`. This is a user-defined name, which identifies the master port. This name will be used in the bin names of the covergroup.

   For example,

```
axi_sys_cfg.master_cfg[0].source_requester_name = $sformatf("master_%0d",0);
```

5. In Master configuration, push back the slave names in to `svt_axi_port_configuration::path_cov_slave_names`. Note that these names should match the names specified in the macro `SVT_AMBA_PATH_COV_DEST_NAMES`. These names signify the slave ports to which the master port can communicate.

   For example,

   ```
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_0);
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_1);
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_2);
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_3);
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_4);
   axi_sys_cfg.master_cfg[0].path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_5);
   ```

6. Slave configuration `svt_axi_port_configuration::svt_amba_addr_mapper dest_addr_mappers[]` is the address mapper, which specifies the slave memory map as part of the AXI Complex Memory Map feature.

7. In the Slave configuration, instantiate the address mapper.

   For example,

   ```
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers = new;
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0] =
   svt_amba_addr_mapper::type_id::create("axi_slave_addr_mapper");
   ```

8. In the Slave configuration, specify the name for the slave port. Note that this name should match the name specified in the macro `SVT_AMBA_PATH_COV_DEST_NAMES`. This name helps to identify the slave port. This name will be used in the bin names of the cover group.

   For example,

   ```
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].path_cov_slave_component_name =
   svt_amba_addr_mapper::slave_0;
   ```

9. This step is optional and must be done only if, for a given address, the destination is different based on originating master. Note that these names should match the names specified in `svt_axi_port_configuration::source_requester_name`.

   For example,

   ```
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_0");
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_1");
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_2");
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_3");
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_4");
   axi_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("master_5");
   ```

   After configuring the above settings, run the simulation and review the covergroup `trans_cross_master_to_slave_path_access` in coverage report.

8.23 AXI_ACE Path Coverage

The path coverage enhancement provides the cross coverage of transaction properties with Master to Slave access paths. This cross coverage can separate cross coverage group within VIP coverage.

The proposed VIP covergroup `trans_cross_master_to_slave_path_access_ace` cross coverage is added in port level.

Use Model:

1. Enable the `svt_axi_system_configuration::enable_complex_memory_map`.
2. `svt_axi_port_configuration::trans_cross_master_to_slave_path_access_cov_enable` enables path coverage for a given master.
3. Define `SVT_AMBA_PATH_COV_DEST_NAMES` based on the names of the slaves in the system

   For example,

   ```
   `ifndef SVT_AMBA_PATH_COV_DEST_NAMES
     `define SVT_AMBA_PATH_COV_DEST_NAMES
   ace_lite_slave_0,ace_lite_slave_1,ace_lite_slave_2
   `endif
   ```

   For slaves: `svt_axi_port_configuration`: `svt_amba_addr_mapper dest_addr_mappers[]; //` This is where slave memory map is specified.
4. The general equation to determine if a given master address will be routed to a particular slave, is as that a `master_addr` and the slave's address mapper must fulfill this condition. Here `global_addr` actually refers to global base address of the slave.
   `master_addr and~slave_mapper.mask = slave mapper.global_addr.`

   Here are some examples on setting address map at the slave:

   - Let us say there is a total slave address space of 2 GB and it is equally divided between two slaves. The definition will be as follows:

     Slave 1 address range is: 0000_0000 to 3FFF_FFFF (0 to 1GB)

     global_addr = 0;local_addr = 0;mask = 0x3fff_ffff; (fullfill the above condition)

     Slave 2: 4000_0000 to 7FFF_FFFF (1GB to 2GB)

     global_addr = 0x4000_0000; local_addr = 0x4000_0000 mask = 0x3fff_ffff (fullfill the above condition)

   - Now let us say that the below address space is interleaved between the two slaves.

     Slave 1 address range is: 0000_0000 to FFFF_FFFF

     Slave 2 address range is: 0000_0000 to FFFF_FFFF

     Slave_0, mapper 0 and Slave 1, mapper 0:

     global_addr = 0;local_addr = 0;mask = ffff_ffff; (fullfill the above condition)

     Now let us say that slaves are interleaved at 400 address boundary. So the mapping of slave 1 and slave 2 has to be divided into multiple mappings. The general rule of thumb is that the mask value cannot exceed the global_addr (except when global_addr is 0 or memory is interleaved).

     Slave_0, mapper 1: 0000_0000 to 0000_03FF -> global_addr = 32'h0000_0000; mask =32'hFFFF_FBFF;

     Slave_1, mapper 1: 0000_0400 to 0000_07FF -> global_addr = 32'h0400_0000; mask =32'hFFFF_FBFF;

     In this case, address 10th bit decides the Slave _0 and Slave_1, If 10th bit of address signal is 0 than Slave1 is being accessed and if its 1 than Slave2 is being accessed. Here, mask is configured

based on the above condition to satisfy and address 10th bit should match to route to the particular slaves.

`svt_amba_addr_mapper ::local addr` should be equal to the `svt_amba_addr_mapper ::global_addr and svt_amba_addr_mapper ::mask` should be configured based on the `svt_amba_addr_mapper ::global_address.`
`svt_amba_addr_mapper::is_register_addr_space` set as 0.

5. Snippet of Master Configuration with respect to Path Coverage:

- `set_amba_sys_config()`

```
foreach (this.axi_sys_cfg[i]) begin
set_axi_system_configuration(this.axi_sys_cfg[i],num_axi_masters,num_axi_slaves);
void'(set_axi_test_suite_system_config(this.axi_sys_cfg[i]));
//Required for test_suite or interconnect VIP, not typically required for users
this.axi_sys_cfg[i].set_addr_range(0,64'h0,(64'hFFF_FFFF_FFFF_FFFF));
this.axi_sys_cfg[i].set_addr_range(1,64'h1000_0000_0000_0000,(64'h1FFF_FFFF_FFFF_FFFF));
this.axi_sys_cfg[i].set_addr_range(2,64'h2000_0000_0000_0000,(64'h2FFF_FFFF_FFFF_FFFF));
end
```

- `set_axi_system_configuration()`

```
foreach(axi_sys_cfg.master_cfg[i]) begin
  axi_sys_cfg.master_cfg[i].source_requester_name = $sformatf("ace_master_%0d",i);
axi_sys_cfg.master_cfg[i].path_cov_slave_names.push_back(svt_amba_addr_mapper::ace_lite_
slave_0);
axi_sys_cfg.master_cfg[i].path_cov_slave_names.push_back(svt_amba_addr_mapper::ace_lite_
slave_1);
axi_sys_cfg.master_cfg[i].path_cov_slave_names.push_back(svt_amba_addr_mapper::ace_lite_
slave_2);
end
```

6. Snippet of Slave Configuration with respect to Path Coverage:

```
foreach(axi_sys_cfg.slave_cfg[i]) begin
      axi_sys_cfg.slave_cfg[i].axi_interface_type =
svt_axi_port_configuration::ACE_LITE;
      // configure the dest_addr_mappers
      if (i == 0) begin
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers = new[1];
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0] =
svt_amba_addr_mapper::type_id::create("axi_slave_addr_mapper");
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].path_cov_slave_component_name =
svt_amba_addr_mapper::ace_lite_slave_0;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].global_addr = 64'h0;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].local_addr = 64'h0;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].mask = 64'hFFF_FFFF_FFFF_FFFF;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].is_register_addr_space = 0;

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_0");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_1");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_2");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_3");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_4");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_5");
```

```
      end
      else if (i == 1) begin
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers = new[1];
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0] =
svt_amba_addr_mapper::type_id::create("axi_slave_addr_mapper");
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].path_cov_slave_component_name =
svt_amba_addr_mapper::ace_lite_slave_1;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].global_addr =
64'h1000_0000_0000_0000;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].local_addr =
64'h1000_0000_0000_0000;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].mask = 64'hFFF_FFFF_FFFF_FFFF;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].is_register_addr_space = 0;

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_0");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_1");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_2");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_3");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_4");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_5");
      end
      else if (i == 2) begin
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers = new[1];
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0] =
svt_amba_addr_mapper::type_id::create("axi_slave_addr_mapper");
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].path_cov_slave_component_name  =
svt_amba_addr_mapper::ace_lite_slave_2;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].global_addr =
64'h2000_0000_0000_0000;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].local_addr =
64'h2000_0000_0000_0000;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].mask = 64'hFFF_FFFF_FFFF_FFFF;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].is_register_addr_space = 0;

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_0");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_1");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_2");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_3");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_4");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_5");
      end
      else if (i == 3) begin
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers = new[1];
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0] =
svt_amba_addr_mapper::type_id::create("axi_slave_addr_mapper");
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].path_cov_slave_component_name =
svt_amba_addr_mapper::ace_lite_slave_3;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].global_addr =
64'h3000_0000_0000_0000;
```

```
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].local_addr =
64'h3000_0000_0000_0000;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].mask = 64'hFFF_FFFF_FFFF_FFFF;
        axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].is_register_addr_space = 0;

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_0");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_1");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_2");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_3");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_4");

axi_sys_cfg.slave_cfg[i].dest_addr_mappers[0].source_masters.push_back("ace_master_5");
      end
    end
```

8.24 Wait State Mechanisms

There are two ways to insert wait states in awready signal.

- Program the value of `svt_axi_transaction::addr_ready_delay` to apply delays in awready signal assertion from the slave. See class reference HTML for further reference.
- Set `svt_axi_transaction::suspend_awready` to 1 . This can be set to 1 from the `svt_axi_slave_memory_reponse_sequence` in the testbench.
  Once `svt_axi_transaction::suspend_awready` is set to 1 , active slave VIP will drive awready signal to 0 and will wait for `svt_axi_transaction::suspend_awready` to become 0 before driving awready signal high.

This is applicable only when `svt_axi_port_configuration::default_awready` is set to 0. See class reference HTML for further reference.

8.25 Interconnect Routing

Interconnect Routing you to redefine the master to slave routing behavior in the interconnect VIP. Interconnect VIP will call this method and use the first slave port returned by this function to route the incoming master transaction.

By default, this method is undefined and returns FALSE.

It is expected to define this method with the routing behavior of your choice and it must return TRUE. Interconnect VIP will then use the first slave port returned by this function for routing master transaction. Else, it will use its default mode of routing master transactions to slave ports based on the address ranges.

This method is applicable for Interconnect VIP and System Monitor component.

The following system configuration function needs to be programmed to enable this feature.

```
extern virtual function bit
get_interconnect_slave_route_port(bit[`SVT_AXI_MAX_ADDR_WIDTH-1:0]
tagged_master_addr, bit is_register_addr_space, int master_port_id, output int
slave_port_ids[$]);
```

For more details, see AXI Class Reference HTML.

Usage Examples:

```
// provide master to slave routing mechanism of user's choice to be used by interconnect
  function bit get_interconnect_slave_route_port(bit[`SVT_AXI_MAX_ADDR_WIDTH-1:0]
tagged_master_addr, bit is_register_addr_space, int master_port_id, output int
slave_port_ids[$]);
    randcase
      30 : begin
             slave_port_ids.push_back(master_port_id);
             return(1);
           end
      70 : return(0);
    endcase
  endfunction: get_interconnect_slave_route_port
```

8.26 Support for Transaction Splitting Across Two Slaves

Some interconnect DUTs have a feature where a single master transaction is routed across two slaves. In order that the transaction routing and data integrity checks are processed correctly in the system monitor, it is required to split the original master transaction as two separate transactions at a user-configured boundary. This user-defined boundary is specified in the following parameter:

- `svt_axi_port_configuration::byte_boundary_for_master_xact_split`

The transactions which are split across two slaves must be indicated through this callback in the system monitor.

```
svt_axi_system_monitor_callback:: pre_add_to_input_xact_queue(svt_axi_system_monitor
system_monitor, svt_axi_transaction xact, ref bit split_original_xact);
```

After splitting, the split transactions are available through this callback:

```
svt_axi_system_monitor_callback::post_xact_split(svt_axi_system_monitor system_monitor,
svt_axi_transaction xact, svt_axi_transaction split_xacts[$]);
```

Once set, the system monitor splits the original transaction at the given boundary and uses the split master transactions for further processing. The original master_xact can be retrieved from the split transactions using a property named 'causal_xact'. The following code indicates that a transaction processed by the system monitor is a split transaction. This code could be used in callbacks if it is required to identify whether a transaction is a split transaction. The original transaction is given in 'parent_xact':

```
if ($cast(parent_xact, xact.get_causal_ref()) && (parent_xact != null))
```

9 Usage Notes

This chapter provides usage notes for AXI Verification IP.

This chapter discusses the following topics:

9.1 Managing Coverage Through Exclude File

As per the requirement, `coverbins` or `covergroups` can be excluded, by using the exclude file. The exclude file can be generated using Verdi and the exclusions can be incorporated in the `urgReport`.

- For more information, see Verdi documentation.

10 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

10.1 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 Agent. 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 Agent. The signals, `read_data_xact_num`, `read_data_beat_num` and `write_resp_xact_num` are driven by slave driver in Slave Agent.

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 the default option 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.