Verification Continuum™

VC Verification IP
AMBA AHB
OVM User Guide

Version R-2021.03, March 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface

About This Guide

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

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

Guide Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Introduction", introduces the Synopsys AHB VIP and its features.
- Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using the Synopsys AHB VIP.
- Chapter 3, "General Concepts", introduces the AHB VIP within a OVM environment and describes the data objects and components that comprise the VIP.
- Chapter 4, "AHB VIP Programming Interface", describes the verification features supported by AHB VIP such as, Verification Planner and Protocol Analyzer.
- Chapter 5, "Using AHB Verification IP", shows how to install and run a getting started example.
- Chapter 7, "Backward Compatibility", provides the details of backward compatibility of the VIP with respect to previous releases.
- Appendix A, "Reporting Problems", outlines the process for working through and reporting Synopsys AHB VIP issues.

Web Resources

- Documentation through SolvNetPlus: 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 AHB VIP.

This chapter discusses the following topics:

1.1 Introduction

The AHB VIP supports verification of designs that include interfaces implementing the AHB Specification. This document describes the use of AHB VIP in testbenches that comply with the SystemVerilog OVM.

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 AHB, object oriented programming, SystemVerilog, and the current version of OVM

1.3 References

For more information on AHB VIP, refer to the following:

- Class Reference for VC Verification IP for AMBA AHB is available at:

  $DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/ahb_svt_ovm_class_reference/html/index.html

1.4 Product Overview

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

1.5 Language and Methodology Support

In this current release, the AHB VIP suite is qualified with the following languages and methodology:

- Languages
  - SystemVerilog
- Methodology
  - Qualified with OVM 2.1.2

1.6 Feature Support

The following sections list supported protocol, verification, and methodology features.

1.6.1 Protocol Features

AHB VIP currently supports the following protocol functions:

- AHB and AHB-Lite support
- All data widths
- All address widths
- All transfer types
- All burst types and burst sizes
- All protection types
- All slave response types
- Support in master for rebuilding transactions
- Lock transactions

1.6.2 Verification Features

AHB VIP currently supports the following verification functions:

- Basic Protocol checking
- Debug Port
- Control on wait states timeouts
- VC Auto Testbench

1.6.3 Methodology Features

AHB VIP currently supports the following methodology functions:

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

1.7 Features Not Supported

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

AMBA SVT VIP Release Notes are present at:

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

2
Installation and Setup

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

The checklist consists of the following major steps:

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

If you encounter any problems with installing the Synopsys AHB VIP, contact Synopsys customer support.

2.1 Verifying the Hardware Requirements

The AHB OVM VIP 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 the Software Requirements

The AHB OVM VIP is qualified for use with certain versions of platforms and simulators.

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 AHB OVM VIP, check the support matrix manual.

2.2.2 Synopsys Common Licensing Software

- The Synopsys Common Licensing (SCL) software provides the licensing function for the AHB OVM 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 AHB VIP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.
- **HTML browser:** Synopsys AHB 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 AHB 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 below for downloading the software from Synopsys. You can download from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If your Synopsys SolvNet password is unknown or forgotten, go to http://solvnet.synopsys.com.

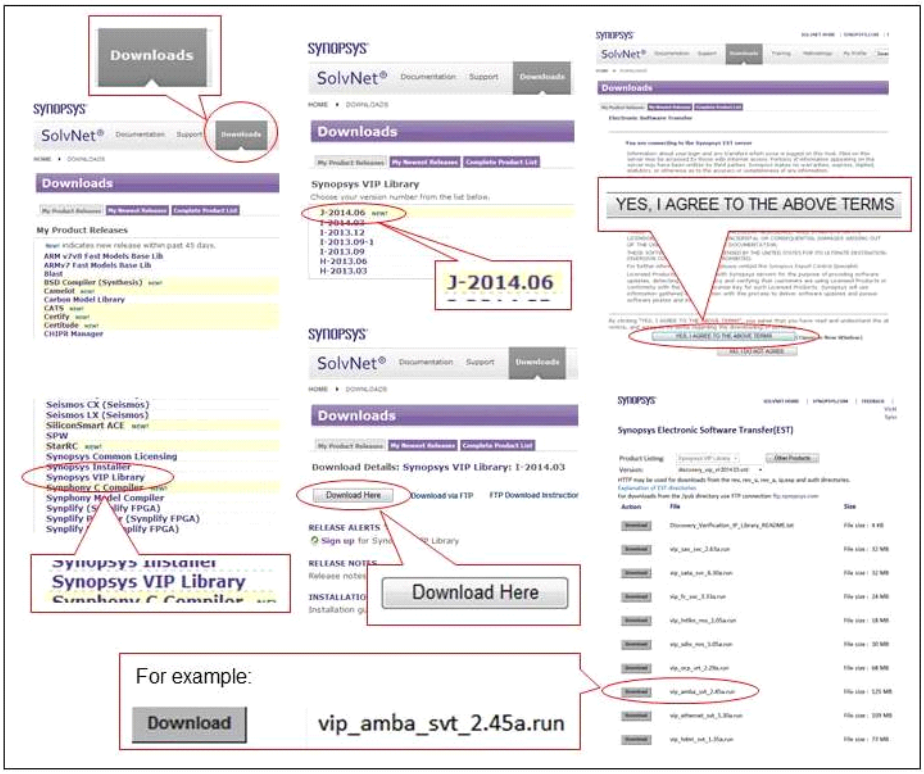
Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, refer to the following web page:

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

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

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, refer to "Customer Support" section to obtain support for download and installation.

2.5 What's Next?

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

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

2.5.1 Licensing Information

The AMBA VIP uses the 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.

2.5.1.1 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 can 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.

2.5.1.2 Simulation License Suspension

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

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

2.5.2 Environment Variable and Path Settings

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

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

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

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

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

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

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

2.5.2.1 Simulator-Specific Settings

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

2.5.3 Determining Your Model Version

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

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

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

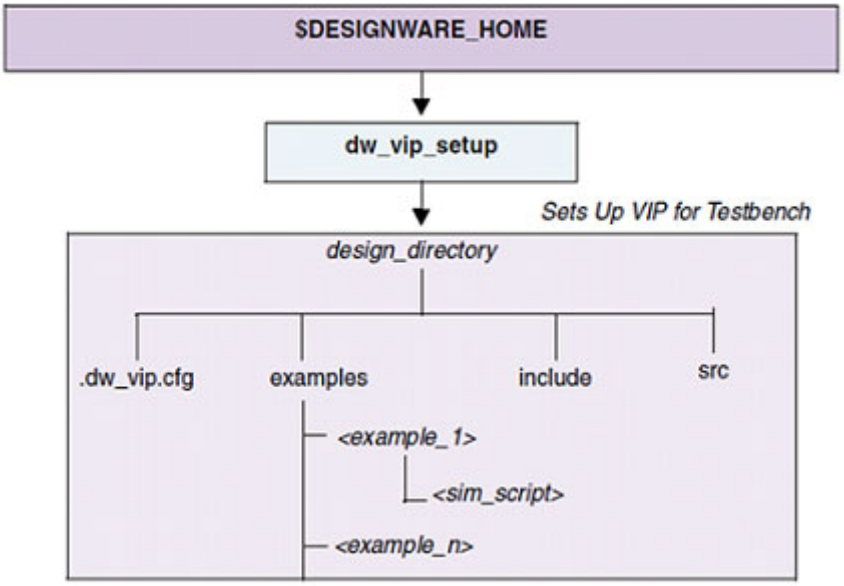
2.5.4.1 Creating a Testbench Design Directory

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

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 the 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 AHB VIP contains the following components.

- ahb_master_agent_svt
- ahb_slave_agent_svt
- ahb_system_env_svt

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

To get help on dw_vip_setup, invoke the following:

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

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

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

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

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

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

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

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

2.5.4.3 Installing and Setting Up More than One VIP Protocol Suite

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

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

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

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

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

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

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

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

To specify other model names, consult the VIP documentation.

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

2.5.4.4 Updating an Existing Model

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

1. Install the model to the same location as your other VIPs 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 ahb_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 ahb_master_agent_svt -v 3.50a -svlog
   ```

2.5.4.5 Removing VIP Models from a Design Directory

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

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

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

2.5.4.6 Reporting Information About DESIGNWARE_HOME or a Design Directory

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

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

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

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

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

2.5.4.7 Running the Example with +incdir+

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

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

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

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

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

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

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

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

2.5.4.8 Getting Help on Example Run/make Scripts

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

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

   run_ahb_svt_ovm_basic_sys --help

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

   where <scenario> is one of: all amba_pv_test base_test directed_test override_test random_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:

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 amba_pv_test base_test directed_test override_test random_wr_rd_test

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

2.5.4.9 The dw_vip_setup Utility

The dw_vip_setup utility:

- Adds, removes, or updates AHB VIP models in a design directory
- Adds example testbenches to a design directory, the AHB 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: • ahb_master_agent_svt • ahb_slave_agent_svt • ahb_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.
-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: • ahb_master_agent_svt • ahb_slave_agent_svt • ahb_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: • ahb_master_agent_svt • ahb_slave_agent_svt • ahb_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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-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 OVM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {*scenario* | *model*/*scenario*}	Cleans the specified scenario/testbench in either the design directory (as specified by the *-path* switch) or the current working directory. This switch deletes *all files in the specified directory*, then restores all Synopsys created files to their original contents.
-i/nfo design | home[:<product>[:<version>[:<methodology>]]]	Generate an informational report on a design directory or VIP installation. `design`: If the '`-info design`' switch is specified, the tool displays product and version content within the specified design directory to standard output. This output can be captured and used as a modellist file for input to this tool to create another design directory with the same content. `home`: If the '`-info home`' switch is specified, the tool displays product, version, and example content within the VIP installation to standard output. Optional filter fields can also be specified such as <product>, <version>, and <methodology> delimited by colons (:). An error will be reported if a nonexistent or invalid filter field is specified. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	Synopsys AHB VIP models are: • ahb_master_agent_svt • ahb_slave_agent_svt • ahb_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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-m/odel_list <filename>	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -m/odel_list switch displays one model name per line and each model includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are ignored.
-b/ridge	Updates the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.
-pa	Enables the run scripts and Makefiles generated by dw_vip_setup to support PA. If this switch is enabled, and the testbench example produces XML files, PA will be launched and the XML files will be read at the end of the example execution. For run scripts, specify `-pa`. For Makefiles, specify `-pa = 1`.
-waves	Enables the run scripts and Makefiles generated by dw_vip_setup to support the `fsdb` waves option. To support this capability, the testbench example must generate an FSDB file when compiled with the WAVES Verilog macro set to `fsdb`, that is, `+define+WAVES=\"fsdb\"`. If a.fsdb file is generated by the example, the Verdi nWave viewer will be launched. For run scripts, specify `-waves fsdb`. For Makefiles, specify `WAVES=fsdb`.
-doc	Creates a doc directory in the specified design directory which is populated with symbolic links to the `DESIGNWARE_HOME` installation for documents related to the given model or example being added or updated.
-methodology <name>	When specified with -doc, only documents associated with the specified methodology name are added to the design directory. Valid methodology names include: OVM, RVM, UVM, VMM, and VLOG.
-copy	When specified with -doc, documents are copied into the design directory, not linked.
-s/uite_list <filename>	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -s/uite_list switch displays one suite name per line and each suite includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are 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.

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

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

/* include AXI , AHB and APB  VIP interface */
  `include "svt_ahb_if.svi"
  `include "svt_axi_if.svi"
  `include "svt_apb_if.svi"

/** Include the AMBA SVT OVM package */
`include "svt_amba.ovm.pkg"
/** Import OVM Package  */
  import ovm_pkg::*;
  /** Import the SVT OVM Package */
  import svt_ovm_pkg::*;
  /** Import the AMBA VIP */
  import svt_amba_ovm_pkg::*;
```

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

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

Supported vendors are vcs, mti and ncv. For example:

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

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

2.5.6 Compile and Run Time Options

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

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

The files containing the options are:

- sim_build_options (contain compile time options 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 AHB VIP, following are the contents of each file, listing optional and required switches:

vcs_build_options

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

vcs_run_options

Required: +OVM_TESTNAME=$scenario

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

3 General Concepts

This chapter describes the usage of AHB VIP in an OVM environment, and its user interface. This chapter discusses the following topics:

- Introduction to OVM
- AHB VIP in an OVM Environment
- Reset Functionality

3.1 Introduction to OVM

OVM 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 AHB VIP in OVM environment, and its user interface. Refer to the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter.

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

- For the IEEE SystemVerilog standard, see:
  - *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*
- For essential guides describing OVM as it is represented in SystemVerilog, along with a class reference, see:
  - *Open Verification Methodology (OVM) 2.1.2 User's Manual*

3.2 AHB VIP in an OVM Environment

The following sections describe how the AHB Verification IP is structured in an OVM testbench.

3.2.1 Master Agent

The master agent encapsulates master sequencer, master driver, and master monitor. The master agent can be configured to operate in active mode and passive mode. The user can provide AHB sequences to the master sequencer.

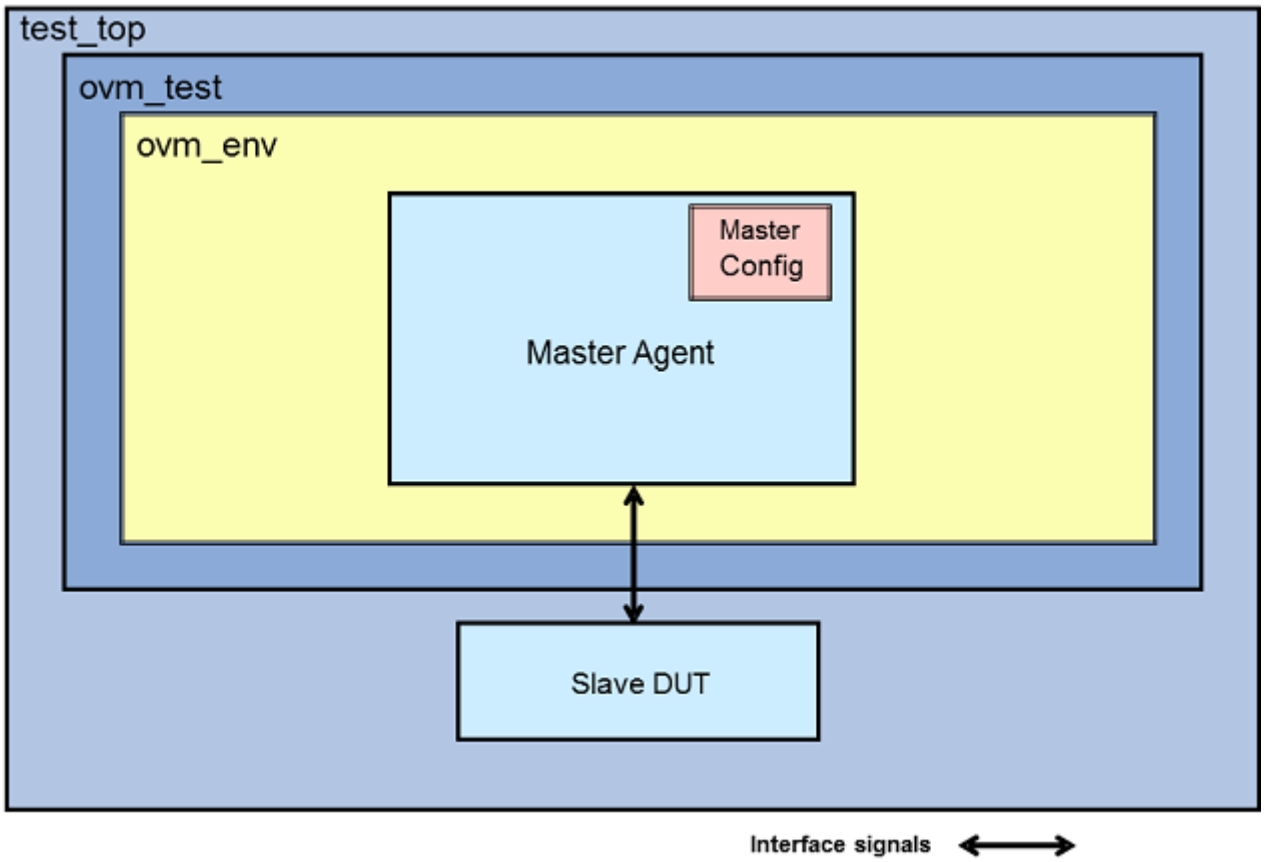
The master agent is configured using the master configuration. The system configuration should be provided to the master agent in the build phase of the test.

Within the master agent, the master driver gets sequences from the master sequencer. The master driver then drives the AHB transactions on the AHB port.

The master driver and system monitor components within master agent call callback methods at various phases of execution of the AHB transaction. Details of callbacks are covered in later sections.

After the AHB transaction on the bus is complete, the completed sequence item is provided to the analysis port of master monitor, which can be used by the testbench. The driver then drives the AHB transactions on the AHB port.

Figure 3-1 Usage With Standalone Master Agent



3.2.1.1 Bus Request

Master driver asserts bus request whenever it receives a sequence from the sequencer. For fixed burst type, master keeps the bus request asserted till the penultimate beat of the burst.

3.2.1.2 Rebuilding Transactions

Master rebuilds transaction under the following conditions:

- Arbiter initiated early burst termination occurs, that is when arbiter de-asserts grant signal before completion of a burst
- SPLIT/RETRY response is received

The master rebuilds transactions using INCR burst type.

3.2.1.3 Error Response

The behavior of master when it receives error response is decided based on the following configuration members:

1. `svt_ahb_system_configuration::master_error_response_policy[]`
2. `svt_ahb_system_configuration::error_response_policy` (when `svt_ahb_system_configuration::master_error_response_policy[]` is not programmed by the user).

The master does not rebuild a transaction that aborted due to ERROR response.

3.2.2 Slave Agent

The slave agent encapsulates slave sequencer, slave driver, and slave monitor. The slave agent can be configured to operate in active mode and passive mode. The user can provide AHB response sequences to the slave sequencer.

The slave agent is configured using slave configuration, which is available in the system configuration. The slave configuration should be provided to the slave agent in the build phase of the test.

In the slave agent, the slave monitor samples the AHB port signals. When a new transaction is detected, slave monitor provides a response request sequence to the slave sequencer. The slave response sequence within the sequencer programs the appropriate slave response. The updated response sequence is then provided by the slave sequencer to the slave driver. The slave driver in turn drives the response on the AHB bus.

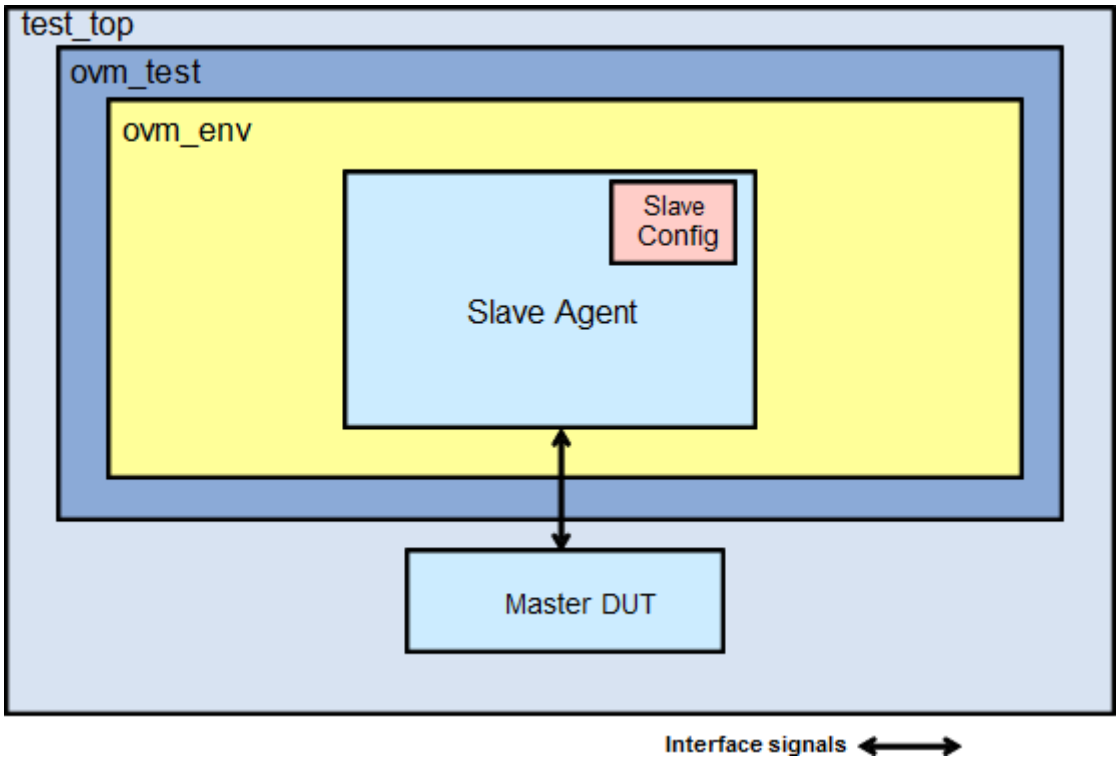
The slave driver expects the slave response sequence to:

- 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 port monitor

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

The slave driver and slave monitor components within the slave agent call the callback methods at various phases of execution of the AHB transaction. Details of callbacks are covered in later sections. After the AHB transaction on the bus is complete, the completed sequence item is provided to the analysis port of port monitor, which can be used by the testbench.

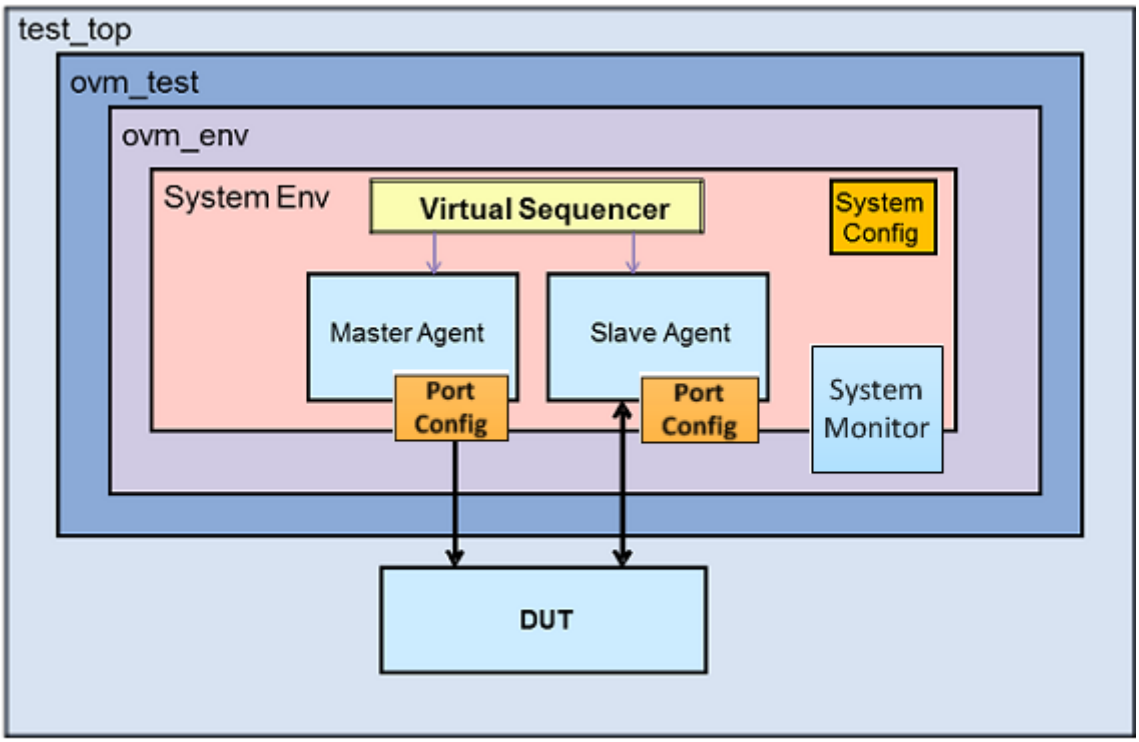
Figure 3-2 Usage With Standalone Slave Agent



3.2.3 System Env

The AHB System Env encapsulates the master agent, slave agents, system sequencer and the system configuration. The number slave agents are configured based on the system configuration provided by the user. In the build phase, the system Env builds the master and slave agents. After the master & slave agents are built, the system Env configures the master & slave agents using the configuration information available in the system configuration.

Figure 3-3 System Env



3.2.3.1 System Sequencer

AHB System sequencer is a virtual sequencer with references to the master sequencers 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 & slave agents.

3.2.4 System Monitor

The System Monitor component is instantiated within the AHB 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_ahb_system_configuration::system_monitor_enable.

System monitor supports the following checks:

- Data Integrity across Interconnect master and slave ports
- Transaction routing

3.2.5 Bus Env

The Bus Env component is instantiated within the AHB System Env component. The Bus Env currently supports maximum of 16 masters and 16 slaves, including Dummy master and Default slave which are built into the Bus Env.

The bus can be configured using bus configuration class svt_ahb_bus_configuration. The system configuration class svt_ahb_system_configuration has a handle to the bus configuration class. Refer to AHB Class reference for members of the bus configuration.

Bus Env component consists of the following components:

- Dummy master
- Default slave
- Arbiter
- Control signals multiplexer
- Write data multiplexer
- Read data/response multiplexer
- Decoder

3.2.5.1 Dummy Master

The Dummy Master is a built-in bus Master model that only performs IDLE transfers. In the AHB Bus model, the Dummy Master is required. The Dummy Master is granted whenever there are no other Masters that can be granted to the System Bus.

There are two cases where the Arbiter must grant the System Bus to the Dummy Master:

1. When a Split response is given to a locked transfer
2. When a Split response is given and all other Masters have already been split

The port index of Dummy Master is configurable using System configuration.

3.2.5.2 Default Slave

The Default Slave is selected when the address is not within any allocated region. For nonsequential and sequential transfers, the default slave provides a two-cycle ERROR response. For busy and idle transfers, the Default Slave provides a zero-wait state OKAY response.

The port index of Default Slave is configurable using System configuration.

3.2.5.3 Arbiter

Every Master has its own bus request pin to indicate the Arbiter when it wants bus ownership. In general, the Arbiter monitors the request pins of all active Masters and decides which Master gains bus ownership based on an arbitration algorithm. The Arbiter asserts that Master's HGRANT pin. In every cycle, only one Master will be granted the bus. The arbiter currently supports round robin arbitration algorithm.

The control of HGRANT by BUS VIP will be as follows:

1. Hgrant is asserted by BUS VIP based on Hbusreq
2. Hgrant is de-asserted for the penultimate beat for fixed length burst that is, Hgrant signal is maintained HIGH all through the transaction for fixed length burst.

However, Early Burst Termination can still occur when Grant is taken away for the penultimate beat, if BUSY cycles are driven between last two beats of fixed length burst.

3.2.5.4 Control Signal Multiplexer

Each Master has its own set of control signals: HADDR, HBURST, HBUSREQ, HLOCK, HSIZE, HTRANS, HWRITE, and HPROT. The Control Signal Multiplexer is controlled by the system bus owner to select the bus owner's control signals.

The data phase is not aligned with the address phase, because of the pipeline nature of the AHB bus. For the data phase of a write transfer, the Data Bus is selected by the Write Data Multiplexer.

For the data phase of a read transfer, the Data Bus is selected by the Read Data/Response Multiplexer. The hmaster port indicates which Master owns the Data Bus.

3.2.5.5 Write Data Multiplexer

The Write Data Multiplexer is responsible for propagating the data of write transfers from the granted master to all slaves in the system.

3.2.5.6 Read Data and Response Multiplexer

The Read Data and Response Multiplexer is responsible for propagating the data and response of read transfers from the selected slave to all masters in the system.

3.2.5.7 Decoder

The Decoder takes the system address that is an output of the Control Signals Multiplexer, then decodes the address and asserts the select pin for the selected Slave. At any time, only one Slave is selected. The decoder uses the address map specified in the system configuration for selecting the slave.

3.2.6 Active and Passive Mode

Table 3-1 lists the behavior of master and slave agents in active and passive modes.

Table 3-1 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. Users can enable/disable this functionality through configuration.	In Passive mode, master and slave components monitor the input and output signals, and perform passive functionality of coverage and protocol checking. Users can enable/disable this functionality through configuration.
In active mode, the port monitor within the component performs protocol checks only on sampled (input) signals, that is, it does not do checks on the signals that are driven (output signals) by the component. This is because when the component is driving an exception (exceptions are not supported in this release) the monitor should not flag an error, since it knows that it is driving an exception. Exception means error injection.	In passive mode, the port monitor within the component performs protocol checks on all signals. In passive mode, signals are considered as input signals.
In active mode, the delay values reported in the AHB transaction provided by the master and slave component, are the values provided by the user, and not the sampled delay values.	In passive mode, the delay values reported in the AHB transaction provided by the master and slave components are the actual sampled delay values on the bus.

3.3 Reset Functionality

The AHB VIP samples the reset assertion asynchronously, while samples reset de-assertion synchronously. When reset is deasserted, it is recommended to keep the clock connected to the VIP running. If the clock input to the VIP is not running, de-assertion of reset is not detected, and the VIP will not sample and drive any signals.

When reset signal is asserted, all the transactions in master and slave agents whose address/data/response phases 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 svt_ahb_transaction::status and svt_ahb_transaction::beat_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/WRITE transactions whose address phase is complete, and data/response phase is in progress, a transaction ENDED notification is issued on the rising edge of clock when reset signal assertion is observed.

4 AHB VIP Programming Interface

This chapter describes the programming interface used by the AMBA AHB OVM VIP and discusses the following topics:

- Configuration Objects
- Transaction Objects
- Callbacks
- Interfaces and Modports
- Events
- Overriding System Constants
- Verification Features

4.1 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 or system Env.

The configuration object properties can be of two types:

- Static configuration properties:

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

- Dynamic configuration properties:

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

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

The AHB VIP defines the following configuration classes:

- System configuration (svt_ahb_system_configuration)

  System configuration class contains configuration information which is applicable across the entire system. User can specify the system level configuration parameters through this class.

User needs 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
- Sub-configurations for master and slave agents
- Virtual top level AHB interface
- Address map
- Endian Mode

- Master and Slave configuration (svt_ahb_master_configuration and svt_ahb_slave_configuration)

  The master and slave configuration class contains configuration information which is applicable to the AHB master and slave agents in the system Env. Some of the important information provided by the slave configuration class is:

  - Active/Passive mode of the agent
  - Address and data bus widths
  - Enable/disable protocol checks
  - Enable/disable port level coverage
  - Values to be driven on bus during idle periods

The master and slave configuration objects within the system configuration object are created in the constructor of the system configuration.

See the AHB VIP Class reference HTML documentation for details on individual members of configuration classes.

4.2 Transaction Objects

Transaction objects, which are extended from the ovm_sequence_item base class, define a unit of AHB 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.

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

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

AHB 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.

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

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.

AHB VIP defines following transaction classes:

- AHB Base Transaction (svt_ahb_transaction)

  This is the base transaction class which contains all the physical attributes of the transaction like address, data, direction, and so on.

- AHB Master Transaction (svt_ahb_master_transaction)

  The master transaction class extends from the AHB transaction base class svt_ahb_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_ahb_master_transaction from its analysis ports, in active and passive mode.

- AHB Slave Transaction (svt_ahb_slave_transaction)

  The slave transaction class extends from the AHB transaction base class svt_ahb_transaction. The slave transaction class contains the constraints for slave specific members in the base transaction class. The slave transaction is used to provide response information to the slave agent. At the end of each transaction, the slave agent provides object of type svt_ahb_slave_transaction from its analysis ports, in active and passive mode.

The transaction contains a handle to the configuration object, which provides the configuration of the port on which this transaction would be applied. The configuration is used during randomizing the transaction. The configuration is available in the sequencer of the master/slave agent. The user sequence should initialize the configuration handle in the transaction using the configuration available in the sequencer of the master/slave agent. If the configuration handle in the transaction is null at the time of randomization, the transaction will issue a fatal message.

See the AHB VIP Class reference HTML documentation for details on individual members of transaction classes.

4.2.1 Analysis Ports

The port monitor in the master & slave agent provides an analysis port called "item_observed_port". At the end of the transaction, the master & slave agents respectively write the completed svt_ahb_master_transaction and svt_ahb_slave_transaction objects to their analysis port. This holds true in active as well as passive mode of operation of the master/slave agent. The user can use the analysis port for connecting to scoreboard, or any other purpose, where a transaction object for the completed transaction is required.

4.3 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 driver and monitor 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 users so the class can be extended and user 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 `ovm_register_cb macro.

  AHB VIP uses callbacks in three main applications:

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

4.3.1 Callbacks in Master Agent

In the master agent, the callback methods are called by master driver and master monitor components.

Below are the callback classes which contain the callback methods invoked by the master agent:

- svt_ahb_master_callback
- svt_ahb_master_monitor_callback

See the class reference HTML documentation for details of these classes.

4.3.2 Callbacks in Slave Agent

In the slave agent, the callback methods are called by slave driver and slave monitor components.

Below are the callback classes which contain the callback methods invoked by the slave agent:

- svt_ahb_slave_callback
- svt_ahb_slave_monitor_callback

See the class reference HTML documentation for details of these classes.

4.4 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.

AHB VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top level interface `svt_ahb_if` is defined. The top level interface contains an array of slave sub-interfaces of type svt_ahb_master_if and svt_ahb_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_ahb_system_configuration::set_if.

The master interface is contained in the master configuration class. The master interface is specified to the master configuration class using svt_ahb_master_configuration::set_master_if method. The master interfaces are provided to the master configuration objects in the constructor of the system configuration.

The slave interface is contained in the slave configuration class. The slave interface is specified to the slave configuration class using svt_ahb_slave_configuration::set_slave_if method. The slave interfaces are provided to the slave configuration objects in the constructor of the system configuration.

See the Class Reference HTML for more information on interfaces and modports.

4.4.1 Bind Interfaces

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

4.4.2 Parameterized Interfaces

AHB VIP supports parameterized interfaces svt_ahb_param_if, svt_ahb_master_param_if and svt_ahb_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_ahb_port_defines.svi file. (refer to Step 4.6). 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_ahb_port_defines.svi or svt_ahb_user_defines.svi file.

To use parameterized interface, the user still needs to instantiate the top-level interface svt_ahb_if. The svt_ahb_param_if interface should be used for connecting top level svt_ahb_if interface signals (Multiplexed Signals from the AHB Bus, such as, hresp_bus, hwrite_bus) to the DUT.

The svt_ahb_master_param_if interface should be used for connecting AHB Master VIP component to the DUT and svt_ahb_slave_param_if interface should be used to connect AHB Slave VIP component to the DUT.

Note that, tb_ahb_svt_ovm_basic_param_if_sys example is not yet supported. Refer to tb_ahb_svt_uvm_basic_param_if_sys example for usage of parameterized interface. The README file in the example describes the usage.

4.5 Events

Master and slave driver and monitor issue EVENT_XACT_STARTED and EVENT_XACT_ENDED events. These events denote the start of transaction and end of transaction events. These notifications are issued by the master and slave components as described below, in both active and passive mode.

- EVENT_XACT_STARTED is issued on the rising clock edge when address phase of transaction is sampled on the bus.
- EVENT_XACT_ENDED is issued on the rising clock edge when the hready is sampled high for last data beat.

4.6 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/amba_svt/latest/include):

- svt_ahb_defines.svi: Top-level include file; allows for the inclusion of the common define symbols and the port define symbols in a single file. Also, it contains a `include to read user overrides if the `SVT_AHB_INCLUDE_USER_DEFINES symbol is defined.
- svt_ahb_common_defines.svi: Defines common constants used by the AHB VIP components. You can override only the "User Definable" constants, which are declared in "ifndef" statements.
- svt_ahb_port_defines.svi: Contains the constants that set the default maximum footprint of the environment. These values determine the wire bit widths in the 'wire frame'-- they do not (necessarily) define the actual bit widths used by the components, which is determined by the configuration classes.
- svt_ahb_user_defines.svi: Contains override values that you define. This file can reside anywhere--specify its location on the simulator command line.

To override the SVT_AHB_MAX_ID_WIDTH constant from the svt_ahb_port_defines.svi file:

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

  `define SVT_AHB_MAX_ID_WIDTH 64
- In the simulator compile command:
  - Ensure that the directory containing svt_ahb_user_defines.svi is provided to the simulator
  - Provide SVT_AHB_INCLUDE_USER_DEFINES on the simulator command line as follows:

    +define+SVT_AHB_INCLUDE_USER_DEFINES

Note the following restrictions when overriding the default maximum footprint:

- Never use a value of 0 for a MAX_*_WDTH value. The value must be >= 1.
- The maximum footprint set at compile time must work for the full design. If you are using multiple instances of AHB VIP, only one maximum footprint can be set and must therefore satisfy the largest requirement.

4.7 Verification Features

This section describes the various verification features available along with AHB VIP. This section discusses the following verification features:

- "Sequence Collection"
- "Protocol Analyzer Support"
- "Verification Planner"

4.7.1 Sequence Collection

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

The master sequences can be used as standalone sequences. These sequences are also added to the sequence library svt_ahb_master_transaction_sequence_library. User can load the sequence library in the sequencer within the master agent. In such case, all sequences in the sequence library would get executed.

4.7.2 Performance Analyzer

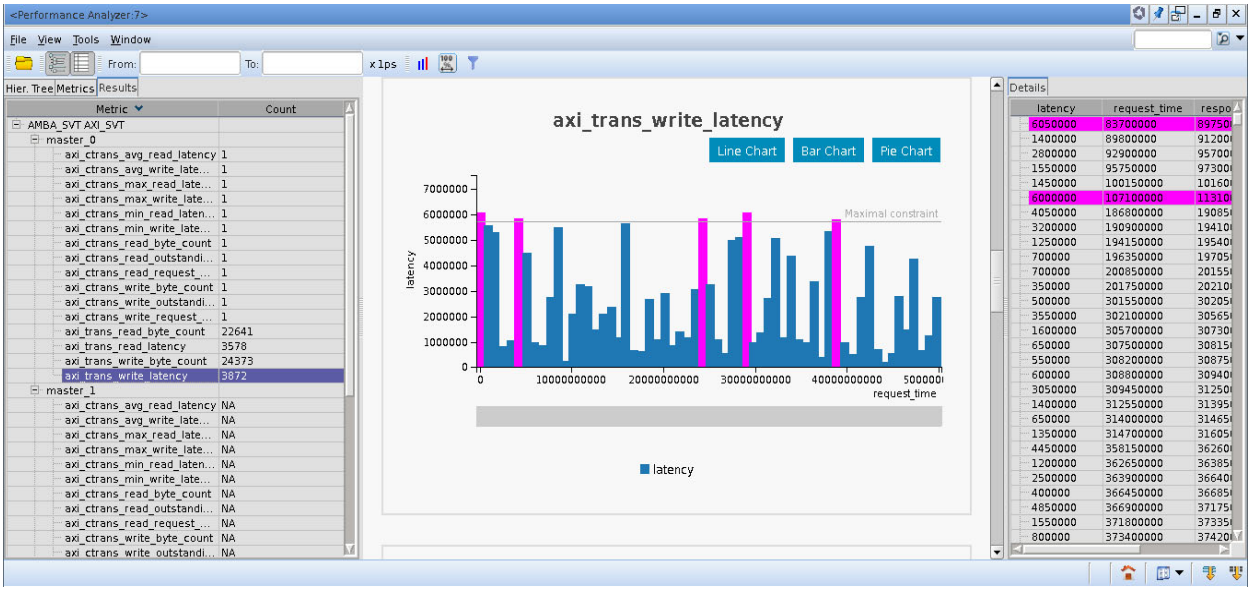
The performance analyzer tool is used to calculate the performance of sub-systems.

4.7.2.1 Invoking Verdi GUI After Running the Simulation

1. Invoking Verdi GUI after running the simulation:

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

Figure 4-1 Final View Of Performance Metric With Graph and Data Details



Note
For more information, see Verdi_Performance_Analyzer.pdf.

4.7.3 Metrics Description

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

- Transaction metrics
- Cross transaction metrics
- Cross instance metrics

4.7.3.1 Transaction Type Metrics

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

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

4.7.3.2 Cross Transaction Type

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

- `*_ctrans_min_read_latency`
- `*_ctrans_min_write_latency`
- `*_ctrans_max_read_latency`
- `*_ctrans_max_write_latency`
- `*_ctrans_avg_read_latency`

- *_ctrans_avg_write_latency
- *_ctrans_read_request_count
- *_ctrans_write_request_count
- *_ctrans_read_outstanding_count
- *_ctrans_write_outstanding_count
- *_ctrans_read_byte_count
- *_ctrans_write_byte_count

4.7.3.3 Cross Instance Type

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

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

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

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

4.7.4 Protocol Analyzer Support

AHB VIP supports Protocol Analyzer. Protocol Analyzer is an interactive graphical application which provides protocol-oriented analysis and debugging capabilities. Protocol file generation is enabled or disabled through the variable "enable_xml_gen" that is defined in the class "svt_ahb_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`.

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

4.7.4.1 Support for VC Auto Testbench

AHB 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.

4.7.4.2 Support for Native Dumping of FSDB

Native FSDB supported in AHB VIP.

- **FSBD Generation**: Protocol Analyzer uses transaction-level dump database. You can use the following settings to dump the transaction database:
  - Compile Time Options
    - -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 pa_format_type is added for XML/FSDB generation in svt_ahb_configuration.sv. Add the following setting in system configuration to enable the generation of XML/FSDB

```
    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
```

- **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.

4.7.5 Verification Planner

AHB VIP provides verification plans which can be used for tracking verification progress of AHB and AHB-Lite protocols. A set of top-level plans and sub-plans are provided.

The verification plans are available at:
$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/VerificationPlans.

For more information, refer to the README file, which is available at:
$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/VerificationPlans/README

5 Using AHB Verification IP

This chapter discusses the OVM concepts and techniques for quickly achieving a basic constrained random testbench that incorporates the AHB VIP.

This chapter discusses the following topic:

- SystemVerilog OVM Example Testbenches
- Installing and Running the Examples
- Common Clock Mode
- VIP configuration while using bus VIP, multiple masters (VIPs, DUTs), multiple slaves (VIPs and DUTs)
- Support for AHB5 Features

5.1 SystemVerilog OVM Example Testbenches

This section describes SystemVerilog OVM example testbenches that show general usage for various applications.

A summary of the examples is listed in Tables 5-1

Table 5-1 SystemVerilog Example Summary

Example Name	Level	Description
`tb_ahb_svt_ovm_basic_sys`	Basic	The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two AHB interfaces • An OVM verification environment • AHB system component in the OVM verification environment • Two tests illustrating directed and random transaction generation

Table 5-1 SystemVerilog Example Summary (Continued)

Example Name	Level	Description
tb_ahb_svt_uvm_basic_param_if_sys **Note :** tb_ahb_svt_ovm_basic_param_if_sys is not yet supported, for more information see tb_ahb_svt_uvm_basic_param_if_sys example.	Basic derivative	This example shows how to implement a basic functioning UVM testbench using AHB Verification IP with parameterized interface. The example consists of the following: • A top-level testbench in SystemVerilog • A dummy DUT in the testbench, which has two AHB interfaces • A system level parameter interface • UVM verification environment • AHB System component in the UVM verification environment • Three test files illustrating - A base test that performs common functions for all tests - Directed transaction generation - Random transaction generation
tb_ahb_svt_ovm_intermediate_sys	Intermediate	Not yet supported
tb_ahb_svt_ovm_advanced_sys	Advanced	Not yet supported

5.2 Installing and Running the Examples

Following are the steps for installing and running the example `tb_ahb_svt_ovm_basic_sys` example. 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_ahb_svt_ovm_basic_sys -svtb
   ```

   The example would get installed under:

   ```
   <design_dir>/examples/sverilog/amba_svt/tb_ahb_svt_ovm_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 the ts.directed_test.sv test, perform the following steps:

      ```
      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`, perform the following steps:

```
./run_ahb_svt_ovm_basic_sys -w random_wr_rd_test vcsvlog
```

Invoke `"./run_ahb_svt_ovm_basic_sys -help"` to show more options.

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

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

OR

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

5.3 Common Clock Mode

This section explains the user interface and use model to use different clock sources for different master and slave components in an AHB system.

The following attribute has been provided for user interface:

- `svt_ahb_system_configuration:: common_clock_mode`

  Data type: bit

  Random Attribute: No

  Default value: 1'b1

  This attribute indicates whether a common clock should be used for all the components in the AHB system or not.

When `svt_ahb_system_configuration:: common_clock_mode` attribute is set to 1:

- This mode is to be used if all AHB VIP components within AHB System VIP component are expected to run on the same clock.
- A common clock supplied to the top level interface is used for all the AHB masters, slaves and bus VIP components within the AHB system VIP component.
  - `svt_ahb_if::hclk` is the top level interface clock signal to be connected to the clock source. The same will be automatically used as clock signal for all the master and slave sub interfaces.

When `svt_ahb_system_configuration:: common_clock_mode` attribute is set to 0:

- This mode is useful when some AHB VIP components need to run at different clocks.
- The user needs to supply separate clock to each of the AHB VIP components through the component interface.
  - `svt_ahb_master_if::hclk` and `svt_ahb_slave_if::hclk` are the port specific clock signals to be connected to respective clock sources.

Clock Source for System Level Components

There are certain components within AHB System VIP component which always need `svt_ahb_if::hclk` signal. These are AHB System Monitor (`svt_ahb_system_monitor`) and AHB Bus VIP component (`svt_ahb_bus_env`).

For AHB system monitor to function correctly, `svt_ahb_if::hclk` should be connected to the fastest clock in the system.

AHB Bus VIP does not support different clock frequencies on different ports.

Use Model for Separate Clock Sources

- Configuration

  Set `svt_ahb_system_configuration::common_clock_mode` to 0 in SVT AHB system configuration object.

  ```
  ahb_sys_cfg.common_clock_mode = 0;
  ```

- Connecting separate clock sources

  Consider an example AHB system with one master M0 and one slave S0. The top.sv file can look like this:

  ```
  svt_ahb_if ahb_if;
  bit SystemClock_m0;
  bit SystemClock_s0;

  // Generate clock sources for M0, S0
  assign SystemClock_m0 = SystemClock;
  assign #1 SystemClock_s0 = SystemClock;

  // If the system monitor is enabled
  assign ahb_if.hclk = SystemClock;

  // Assign separate clocks to M0, S0 interfaces
  assign ahb_if.master_if[0].hclk = SystemClock_m0;
  assign ahb_if.slave_if[0].hclk = SystemClock_s0;
  ```

5.4 VIP configuration while using bus VIP, multiple masters (VIPs, DUTs), multiple slaves (VIPs and DUTs)

While using a bus VIP, two master VIPs, two master DUTs, two slave VIPs, and two slave DUTs, the configuration is as follows:

```
num_masters=5;
```

Note One dummy master built-in to bus VIP, two VIP masters configured as active to drive the stimulus and two master VIPs configured in passive mode connected across the master DUTs

```
num_slaves=5;
```

Note One default slave built-in to bus VIP, two slave VIPs configured as active to respond to transactions, and two slave VIPs configured in passive mode to monitor the slave DUTs

```
use_bus=1;
```

Note This will create an instance of bus VIP inside the `system_env`

```
create_sub_cfgs(5,5,5,5);
dummy_master=0;
default_master=0;
```

Note Can be configured as 1 also

```
default_slave=0;
system_monitor_enable=1;
```

Note If you want to use a system monitor

```
   set_addr_range(1, vip1_start_addr, vip1_end_addr);
   set_addr_range(2, vip2_start_addr, vip2_end_addr);
   set_addr_range(3, dut1_start_addr, dut1_end_addr);
set_addr_range(4, dut2_start_addr, dut2_end_addr);
```

```
create_sub_cfgs(num_masters, num_slaves, num_bus_masters, num_bus_slaves);
```

The requirement for `num_masters` must be equal to `num_bus_masters`. `num_slaves` must be equal to `num_bus_slaves`. Thus, we need to create a passive VIP agent for each of the DUT components.

5.5 Support for AHB5 Features

AHB VIP supports the following AHB5 features:

- Multiple Slave select Signal

- Data Bus Endianness
- Secure transfers
- Memory type

To use all these features, user has to set the system configuration parameter ahb5 to 1 in the extended `svt_ahb_system_configuration` class.

Note If you are using AHB5 feature, you need to define the macro SVT_AHB5_ENABLE in svt_ahb_user_defines.svi files. Refer to section Overriding System Constants for details on how to define a macro through user defines file.

5.5.1 Multiple Slave Select Signal

A single slave interface is permitted to support multiple slave select, HSELx, signals. Each HSELx signal corresponds to a different decode of the higher order address bits.

This permits a single slave interface to provide multiple logical interfaces, each with a different location in the system address map. The minimum address space that can be allocated to a logical interface is 1KB. This approach removes the need for a slave to support the address decode to differentiate between the logical interfaces.

The new hsel signal is added to the AHB Bus interface

1. The Multiple Slave select signal in svt_ahb_if interface block:

```
logic [('SVT_AHB_MAX_HSEL_WIDTH -1) :0] hsel [(`SVT_AHB_MAX_NUM_SLAVES -1):0];
```

2. The Multiple Slave select signal in svt_ahb_slave_if interface block:

```
logic [('SVT_AHB_MAX_HSEL_WIDTH -1) :0] hsel;
```

To make use of Multiple Slave Select Signal, the following System configuration parameter has been added in `svt_ahb_system_configuration` class

- ahb5
- multi_hsel_enable
- multi_hsel_width

For more information on this parameters, please refer the AHB class reference html.

A new method set_hsel_addr_range is added to define the range of address for each multiple select signal for a given selected slave. This method is added in svt_ahb_slave_addr_range class.

Address map for each selected slave components is declared with help of hsel_ranges[], `set_hsel_addr_range()` functions.

Use Model:

```
`define SVT_AHB_MAX_HSEL_WIDTH 10

class cust_svt_ahb_system_configuration extends svt_ahb_system_configuration;
    /** Constructor. Also Assign the Common configuration parameters for all topologies.
*/
  function new(string str="cust_svt_ahb_system_configuration");
     super.new(str);
    this.use_bus = 1;
```

```
    this.system_monitor_enable = 1;
    this.num_masters = 3;
    this.num_slaves  = 3;
    this.ahb_lite = 0;
    this.ahb5 = 1;
    this.default_slave = 0;

    this.multi_hsel_enable = new[this.num_slaves] (this.multi_hsel_enable);
    for(int i=0; i< this.num_slaves; i++) begin
       if(i==0) begin
         this.multi_hsel_enable[i] = 0;
      end
       else begin
         this.multi_hsel_enable[i] = 1;
       end
    end

    this.multi_hsel_width = new[6] (this.multi_hsel_width);
   //Salve 0 does not have multiple select signals
this.multi_hsel_width[0] = 1;
   // Slave 1 having 6 Select signals
    this.multi_hsel_width[1] = 6;
   //Slave 2 having 3 select signals
    this.multi_hsel_width[2] = 3;

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

    /** Set necessary configuration parameter for each master and slave configuration */
   for (int i=0; i<this.num_masters; i++) begin
       this.master_cfg[i].addr_width = 32;
      this.master_cfg[i].data_width = 32;
   end
  for (int i=0; i<this.num_slaves; i++) begin
       this.slave_cfg[i].addr_width = 32;
      this.slave_cfg[i].data_width = 32;
   end

    /** Set mode */
    this.master_cfg[1].is_active = 1;
    this.slave_cfg[1].is_active = 1;

    this.master_cfg[2].is_active = 1;
    this.slave_cfg[2].is_active = 1;

/** define the Slave address range */
    this.set_addr_range(1,32'h0200_0000,32'h0300_FFFF);
    this.set_addr_range(2,32'h0a00_0000,32'h0b00_FFFF);

/** define the Multiple Select Signal Address range with a Slave 1 */
    this.slave_addr_ranges[1].hsel_ranges = new[6]
(this.slave_addr_ranges[1].hsel_ranges);
   foreach(this.slave_addr_ranges[1].hsel_ranges[i]) begin
      case(i)
         0:this.slave_addr_ranges[1].set_hsel_addr_range(0,32'h0200_0001,32'h0200_00FF);
         1:this.slave_addr_ranges[1].set_hsel_addr_range(1,32'h0200_0100,32'h0200_0FFF);
         2:this.slave_addr_ranges[1].set_hsel_addr_range(2,32'h0200_1000,32'h0200_FFFE);
         3:this.slave_addr_ranges[1].set_hsel_addr_range(3,32'h0300_0001,32'h0300_00FF);
```

```
            4:this.slave_addr_ranges[1].set_hsel_addr_range(4,32'h0300_0100,32'h0300_0FFF);
            5:this.slave_addr_ranges[1].set_hsel_addr_range(5,32'h0300_1000,32'h0300_FFFE);
          endcase
        end

    /** define the Multiple Select Signal Address range with a Slave 1 */
        this.slave_addr_ranges[2].hsel_ranges = new[3]
    (this.slave_addr_ranges[2].hsel_ranges);
        foreach(this.slave_addr_ranges[2].hsel_ranges[i]) begin
          case(i)
            0:this.slave_addr_ranges[2].set_hsel_addr_range(0,32'h0a00_0001,32'h0a00_00FF);
            1:this.slave_addr_ranges[2].set_hsel_addr_range(1,32'h0a00_0100,32'h0a00_FFFF);
            2:this.slave_addr_ranges[2].set_hsel_addr_range(2,32'h0b00_1000,32'h0b00_FFFE);
           endcase
        end
       endfunction // new
    endclass
```

5.5.2 Data Bus Endianness

AHB5 introduces the Endian property to define which form of byte-invariant big-endian data access is supported.

The use of byte-invariant big-endian data structures simplifies accessing a mixed-endian data structure in a single memory space.

Using byte-invariant big-endian and little-endian means that, for any multi-byte element in a data structure:

- The element uses the same continuous bytes of memory, regardless of the endianness of the data.
- The endianness determines the order of those bytes in memory, meaning it determines whether the first byte in memory is the MS byte or the LS byte of the element.
- Any byte transfer to a given address passes the eight bits of data on the same data bus wires to the same address location, regardless of the endianness of any data element of which the byte is a part.

Two approaches to big-endian data storage are supported as follows:

- BE8 Byte-invariant big-endian:

  The term, byte-invariant big-endian, is derived from the fact that a byte access (8-bit) uses the same data bus bits as a little-endian access to the same address.

  When larger byte-invariant big-endian transfers occur, data is transferred such that:

  - The MS byte is transferred to the transfer address.
  - Decreasingly significant bytes are transferred to sequentially incrementing addresses
- BE32 Word-invariant big-endian:

  The term, word-invariant big-endian, is derived from the fact that a word access (32-bit) uses the same data bus bits for the Most Significant (MS) and the Least Significant (LS) bytes as a little-endian access to the same address.

For half word and word transfers using word-invariant big-endian, data is transferred such that:

- The most significant byte is transferred to the transfer address.
- Decreasingly significant bytes are transferred to sequentially incrementing addresses.

For transfers larger than a word using word-invariant big-endian, data is split into word size blocks:

- The least significant word is transferred to the transfer address.
- Increasingly significant words are transferred to incrementing addresses.

To make use of Data Bus Endianness property, the following configuration parameter has been added in `svt_ahb_configuration` class.

- ahb5
- invariant_mode

For more information on this parameters, see the AHB class reference html.

USE model:

```
class cust_svt_ahb_system_configuration extends svt_ahb_system_configuration;
    /** Constructor. Also Assign the Common configuration parameters for all
topologies.  */
  function new(string str="cust_svt_ahb_system_configuration");
     super.new(str);
    this.use_bus = 1;
    this.system_monitor_enable = 1;
    this.num_masters = 3;
    this.num_slaves  = 3;
    this.ahb_lite = 0;
    this.ahb5 = 1;
    this.little_endian =0; //Reconfiguring to set to big-endian format
    this.default_slave = 0;

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

    /** Set necessary configuration parameter for each master and slave configuration */
   for (int i=0; i<this.num_masters; i++) begin
       this.master_cfg[i].addr_width = 32;
       this.master_cfg[i].data_width = 32;
       this.master_cfg[i].invariant_mode = svt_ahb_configuration::BYTE_INVARIANT;
   end
  for (int i=0; i<this.num_slaves; i++) begin
       this.slave_cfg[i].addr_width = 32;
       this.slave_cfg[i].data_width = 32;
       this.slave_cfg[i].invariant_mode = svt_ahb_configuration::WORD_INVARIANT;
   end

    /** Set mode */
    this.master_cfg[1].is_active = 1;
    this.slave_cfg[1].is_active = 1;

    this.master_cfg[2].is_active = 1;
    this.slave_cfg[2].is_active = 1;

/** define the Slave address range */
    this.set_addr_range(1,32'h0200_0000,32'h0300_FFFF);
    this.set_addr_range(2,32'h0a00_0000,32'h0b00_FFFF);
endclass
```

5.5.3 Secure Transfer

Secure transfer property is introduced in AHB VIP by adding the signal "hnonsec" to interface and corresponding variable "nonsec_trans" in the svt_ahb_transaction.sv class.

svt_ahb_transaction::nonsec_trans variable is of enum type, with 2 enum values are declared as below

- 'NONSECURE_TRANSFER' with value 1
- 'SECURE_TRANSFER' with value 0.

By default svt_ahb_transaction::nonsec_trans is set to enum value 'NONSECURE_TRANSFER' by default indicating that secure transfer feature not available and if needed set to 'SECURE_TRANSFER'.

To make use of secure transfer property, the following configuration parameter has been added in svt_ahb_configuration class.

- secure_enable

For more information on this parameter, see the AHB class reference html.

Configuration Settings Required for Secure Transfer

User has to set the following configurations in the extended `svt_ahb_system_configuration` class:

```
this.slave_cfg[0].secure_enable = 1;
this.master_cfg[0].secure_enable = 1;
```

5.5.4 Memory Type

AHB5 defines extended memory types. To support the additional memory types defined in AHB5, HPROT signal needs to be 7-bit wide. The width of HPROT signal gets programmed to 7 bit when the compile time macro `SVT_AHB5_ENABLE` is defined.

The following configuration parameters support this feature:

- `svt_ahb_system_configuration::ahb5`
- `svt_ahb_configuration::extended_mem_enable`
- `svt_ahb_bus_configuration::bus_extended_mem_enable`

For more information on these parameters, see the AHB class reference html documentation.

Code snippets to enable this feature are:

```
// Add following line in the class extended from svt_ahb_system_configuration
this.ahb5 = 1;
// Set the configuration parameter for master and slave VIP components which need to
support this feature
for (int i=0; i< this.num_masters; i++) begin
this.master_cfg[i].extended_mem_enable = 1;
end
for (int i=0; i<this.num_slaves; i++) begin
this.slave_cfg[i].extended_mem_enable = 1;
end
// If you use AHB Bus VIP component, add below line in the class extended from
svt_ahb_system_configuration
this.bus_cfg.bus_extended_mem_enable = 1;
```

6 Usage Notes

This chapter provides usage notes for AHB Verification IP.

This chapter discusses the following topics:

- Managing Coverage Through Exclude File

6.1 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.

7 Backward Compatibility

Certain changes were introduced in `svt_ahb_if` interface signals from AMBA SVT EA 1.48a onwards, for ease of use, which are not backwards compatible. This chapter provides the details of all such changes.

Following are the details of these changes:

- The common signals from the bus to all the masters and all slaves are added to `svt_ahb_if` with '`_bus`' suffix.
  - Multiplexed Outputs to All Slaves

    These signals will be connected to corresponding slave interface input signals. This table shows the list of these signals:

Table 7-1 Multiplexed Output Signals to All Slaves

Multiplexed output signals of svt_ahb_if to all slaves	Implicitly connected corresponding input signals of svt_ahb_slave_if
haddr_bus	haddr
hburst_bus	hburst
hmaster_bus	hmaster
hmastlock_bus	hmastlock
hprot_bus	hprot
hsize_bus	hsize
htrans_bus	htrans
hwdata_bus	hwdata
hwrite_bus	hwrite
hready_bus	hready_in
control_huser_bus	control_huser

- Multiplexed Outputs to All Masters

  These signals will be implicitly connected to corresponding master interface input signals.

  Table 7-2 Multiplexed Output Signals to All Masters

Multiplexed output signals of svt_ahb_if to all masters	Implicitly connected corresponding input signal of svt_ahb_master_if
hrdata_bus	hrdata
hready_bus	hready
hresp_bus	hresp

- These respective '_bus' signals are passed by svt_ahb_if onto an array of master and slave interfaces as applicable, so that the implicit connectivity happens between the bus signals and all connected master and slave corresponding input signals.

```
svt_ahb_master_if master_if[`SVT_AHB_MAX_NUM_MASTERS-1:0](hclk,
                                                          hresetn,
                                                          hrdata_bus,
                                                          hready_bus,
                                                          hresp_bus);
svt_ahb_slave_if  slave_if[`SVT_AHB_MAX_NUM_SLAVES-1:0](hclk,
                                                        hresetn,
                                                        haddr_bus,
                                                        hburst_bus,
                                                        hmaster_bus,
                                                        hmastlock_bus,
                                                        hprot_bus,
                                                        hsize_bus,
                                                        htrans_bus,
                                                        hwdata_bus,
                                                        hwrite_bus,
                                                        hready_bus,
                                                        control_huser_bus);
```

This helps when the VIP bus ENV is enabled such that there is an implicit connectivity among the master agents, slave agents and the bus ENV. Even when bus ENV is not used, still the bus signals can be used to minimize the number of connections in the test bench.

The test bench connectivity looks like as follows:

For Example:

You can observe that the "_bus" signals are used in below port map, which replaced the corresponding master and slave interface input signals as mentioned in Table 7-1 and Table 7-2. In the following example, master 0 is active master, and slave 1 is active slave.

```
DW_ahb     u_DW_ahb (
.hclk            (ahb_if.hclk),
.hresetn         (ahb_if.hresetn),
/*master 1 side of ahb protocol interface signals*/
.haddr_m1        (ahb_if.master_if[0].haddr),
.hburst_m1       (ahb_if.master_if[0].hburst),
.hbusreq_m1      (ahb_if.master_if[0].hbusreq),
.hlock_m1        (ahb_if.master_if[0].hlock),
.hsize_m1        (ahb_if.master_if[0].hsize),
.htrans_m1       (ahb_if.master_if[0].htrans),
.hwdata_m1       (ahb_if.master_if[0].hwdata),
```

```
.hwrite_m1      (ahb_if.master_if[0].hwrite),
.hprot_m1       (ahb_if.master_if[0].hprot),
.hgrant_m1      (ahb_if.master_if[0].hgrant),
.hready         (ahb_if.hready_bus), // Earlier: ahb_if.master_if[0].hready
.hresp          (ahb_if.hresp_bus),  // Earlier: ahb_if.master_if[0].hresp
.hrdata         (ahb_if.hresp_bus),  // Earlier: ahb_if.master_if[0].hrdata

/*slave 1 side of ahb protocol interface signals*/
.hsplit_s1      (ahb_if.slave_if[1].hsplit),
.hrdata_s1      (ahb_if.slave_if[1].hrdata),
.hresp_s1       (ahb_if.slave_if[1].hresp),
.hready_resp_s1 (ahb_if.slave_if[1].hready),
.hsel_s1        (ahb_if.slave_if[1].hsel),
.haddr          (ahb_if.haddr_bus),    // Earlier: ahb_if.slave_if[1].haddr
.hburst         (ahb_if.hburst_bus),   // Earlier: ahb_if.slave_if[1].hburst
.hsize          (ahb_if.hsize_bus),    // Earlier: ahb_if.slave_if[1].hsize
.htrans         (ahb_if.htrans_bus),   // Earlier: ahb_if.slave_if[1].htrans
.hprot          (ahb_if.hprot_bus),    // Earlier: ahb_if.slave_if[1].hprot
.hwrite         (ahb_if.hwrite_bus),   // Earlier: ahb_if.slave_if[1].hwrite
.hwdata         (ahb_if.hwdata_bus),   // Earlier: ahb_if.slave_if[1].hwdata
.hmaster        (ahb_if.hmaster_bus),  // Earlier: ahb_if.slave_if[1].hmaster
.hmastlock      (ahb_if.hmastlock_bus) // Earlier: ahb_if.slave_if[1].hmastlock
);

assign ahb_if.slave_if[1].hready_in  = ahb_if.hready_bus; // Earlier:
                                                          ahb_if.master_if[0].hready
```

- If the master VIP agent and slave VIP agent are connected back to back without any module in between, then such usage also needs similar update for all the signals highlighted above. The same applies to all such signals from master-to-slave and slave-to-master.

  For Example,

  Previous code: `assign ahb_if.master_if[0].hresp = ahb_if.slave_if[0].hresp`

  Modified code: `ahb_if.hresp_bus = ahb_if.slave_if[0].hresp`

A Reporting Problems

A.1 Introduction

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

A.2 Debug Automation

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

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

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

A.3 Enabling and Specifying Debug Automation Features

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

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

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

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

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

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

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

A.4 Debug Automation Outputs

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

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

The following are the output files generated:

- *svt_debug.out*: It records important information about the debug feature itself, and data about the environment that the VIPs are operating. One file is optionally created when this feature is enabled, depending on if the Verdi libraries are available.
- *svt_debug.transcript*: Log files generated by the simulation run.
- *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.