SYNOPSYS®

Verification Continuum™

VC Verification IP AMBA

UVM User Guide

,

Version S-2021.06

June 2021

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

1 Introduction

This document describes the use of AMBA System Env component that complies with the SystemVerilog Universal Verification Methodology (UVM).

AMBA System Env encapsulates AXI System Env, AHB System Env, APB System Env and AMBA System monitor. AMBA System monitor performs protocol checks across AXI, AHB and APB protocols.

For the Synopsys AMBA VIP class reference HTML documentation, see:

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

Language and Simulator Support

In this current release, the AMBA VIP suite supports the following languages and simulators:

Languages

- SystemVerilog

Methodology

- Qualified with UVM 1.1d and UVM 1.2
  For more information on UVM 1.2, see http://www.accellera.org/.

Features Not Supported

- None

2 Installation and Setup

This section leads you through installing and setting up the AMBA System Env component. When you complete this checklist, the provided example testbench will be operational and the AMBA System Env component will be ready to use.

The quick start consists of the following major steps:

1. "Verifying the Hardware Requirements"
2. "Verifying Software Requirements"
3. "Preparing for Installation"
4. "Downloading and Installing"
5. "Setting Up a Testbench Design Directory"
6. "What's Next?"

Verifying the Hardware Requirements

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

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

Verifying the Software Requirements

The Synopsys AMBA Verification IP is qualified for use with certain versions of platforms and simulators.

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 Synopsys AMBA Verification IP, check the support matrix manual.

Synopsys Common Licensing (SCL) Software

The SCL software provides the licensing function for Synopsys AMBA Verification IP. Acquiring the SCL software is covered here in the installation instructions in "Licensing Information".

Other Third Party Software

Adobe Acrobat: Synopsys AMBA Verification IP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.

HTML browser: Synopsys AMBA 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)

Preparing for Installation

1. Set DESIGNWARE_HOME to the absolute path where Synopsys AMBA Verification IP 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 Vera and Synopsys Common Licensing software or the port@host reference to this file. % setenv SNPSLMD_LICENSE_FILE $LM_LICENSE_FILE

Downloading and Installing

To receive a new version of the Synopsys AMBA Verification IP:

1. 1.Enter a call through SolvNet.
2. • Go to http://solvnet.synopsys.com/ManageCase?ccf=1 and provide the requested information, including:

   Product: Verification IP
   Sub Product: amba_svt
   Version M-2017.03-1-T0324
   Subject: AMBA
   Enter a request in the Problem Description window

3. Synopsys indicates the FTP location and access instructions for requested run file.
4. Execute the run file: % <vip run file name>.run Answer the prompts that the .run script generates until the install is complete.

Setting Up a Testbench Design Directory

A design directory is placed where the Synopsys AMBA Verification IP is set up for use in a testbench. A design directory is required for using VIP and, for this, the dw_vip_setup utility is provided.

The dw_vip_setup utility allows you to:

- Create the design directory (design_dir), which contains the model, support files (include files), and examples (if any)

- Add a specific version of Synopsys AMBA Verification IP from DESIGNWARE_HOME to a design directory

For a full description of dw_vip_setup, refer to "The dw_vip_setup Utility.

To create a design directory and add a model so it can be used in a testbench, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a amba_system_env_svt -svtb
```

The models provided with Synopsys AMBA Verification IP include:

- amba_system_env_svt

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 VIP into Your Testbench"

Licensing Information

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

You can find general SCL information in the following location:

http://www.synopsys.com/keys

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

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

Controlling License Usage

Using the DW_LICENSE_OVERRIDE environment variable, you can control which license is used as follows.

To use only DesignWare-Regression and VIP-LIBRARY-SVT licenses, set DW_LICENSE_OVERRIDE to:

```
DESIGNWARE-REGRESSION VIP-LIBRARY-SVT
```

To use only a VIP-AMBA-SVT license, set DW_LICENSE_OVERRIDE to:

```
VIP-AMBA-SVT
```

If DW_LICENSE_OVERRIDE is set to any value and the corresponding feature is not available, a license error message is issued.

If Licensing Fails

By default, simulations exit with an error when a VIP license cannot be secured. Alternatively, the DW_NOAUTH_CONTINUE environment variable can be set to allow simulations to continue when one or more VIP models fail to authorize. Unauthorized VIP models essentially become disabled when DW_NOAUTH_CONTINUE is set to any value.

```
% setenv DW_NOAUTH_CONTINUE
```

Also, some simulation environments allow license polling, which pauses the simulation until a license is available. License polling is described next.

If you encounter problems with licensing, see "Customer Support".

License Polling

If you request a license and none are available, license polling allows your request to exist until a license becomes available instead of exiting immediately. To control license polling, you use the DW_WAIT_LICENSE environment variable as follows:

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

Simulation License Suspension

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

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

Environment Variable and Path Setting

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

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

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

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

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

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

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

Simulator Specific Setting

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

Determining Your Model Version

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

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

- To determine the versions of VIP models installed in your $DESIGNWARE_HOME tree, use the setup utility as follows:
  ```
  % $DESIGNWARE_HOME/bin/dw_vip_setup -i home
  ```

- To determine the versions of VIP models in your design directory, use the setup utility as follows:
  ```
  % $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir_path -i design
  ```

Integrating the AMBA VIP in to Your Testbench

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

- "Creating a Testbench Design Directory"
- "The dw_vip_setup Utility"
- "Using Verification IP in Your Testbench"

Creating a Testbench Design Directory

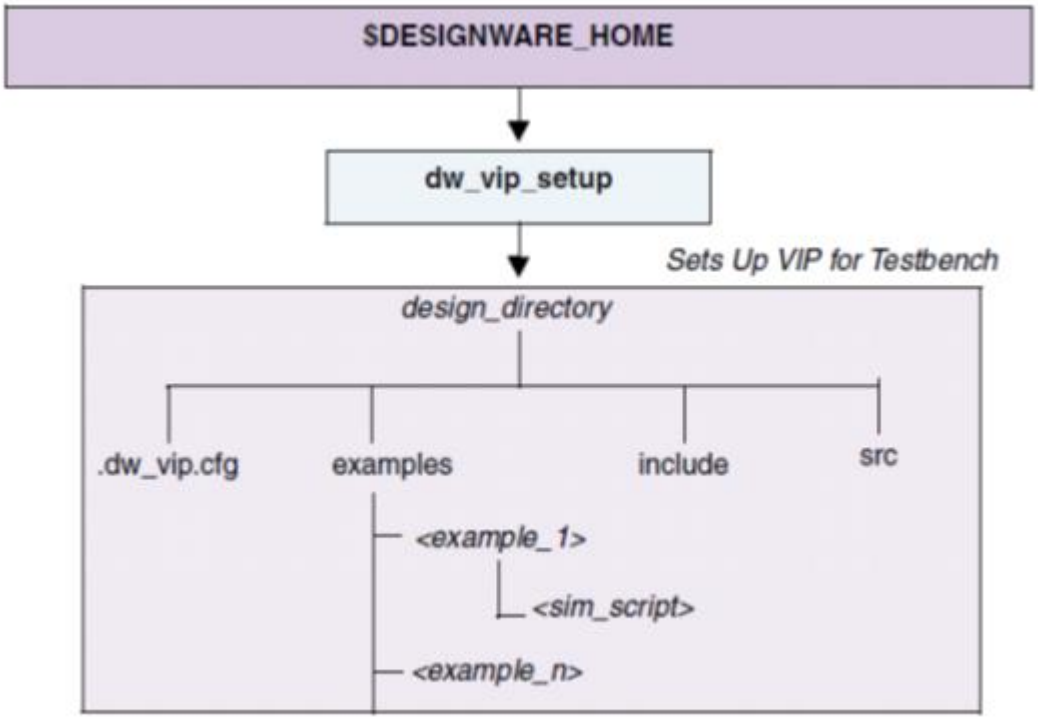
A design directory contains a version of the VIP that is set up and ready for use in a testbench. You use the dw_vip_setup utility to create design directories. For the full description of dw_vip_setup, refer to "The dw_vip_setup Utility"

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

A design directory gives you control over the version of the 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 2-1 shows this process and the contents of a design directory.

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. 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.

This section contains three examples that show common usage scenarios.

- "Adding or Updating VIP Models In a Design Directory"
- "Removing VIP Models from a Design Directory"
- "Reporting Information About DESIGNWARE_HOME or a Design Directory"

Adding or Updating VIP Models in a Design Directory

Synopsys AMBA SVT VIP models include:

- amba_system_env_svt

The following example adds a Synopsys AMBA System Env model to a design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup –a amba_system_env_svt -svtb
```

The following example updates a Synopsys VIP model in a design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -p design_dir -u amba_system_env_svt
-svtb
```

In these examples, the dw_vip_setup utility does the following:

1. Creates an include directory under the current directory and copies:
   - All files in the amba_system_env_svt model include directory
   - All include files in the AMBA VIP suite
   - The latest SVT library include files into the include directory
2. Creates the VIP suite libraries and SVT libraries.

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.

Reporting Information about DESIGNWARE_HOME or a Design Directory

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

The following example lists the 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
```

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_amba_svt_uvm_basic_sys –help
   usage: run_amba_svt_uvm_basic_sys [-32] [-incdir] [-verbose] [-debug_opts] [-waves] [-clean] [-nobuild] [-buildonly] [-norun] [-pa] <scenario> <simulator>

where <scenario> is one of: all base_test callback_test shared_memory_test

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

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

-incdir use DESIGNWARE_HOME include files instead of design directory

-verbose enable verbose mode during compilation

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

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

-seed run simulation with specified seed value

-clean clean simulator generated files

-nobuild skip simulator compilation

-buildonly exit after simulator build

-norun only echo commands (do not execute)

-pa invoke Verdi after execution

2. Invoke the make file with help switch as in:
   gmake help
   Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG_OPTS=1] [SEED=<value>] [FORCE_32BIT=1] [WAVES=fsdb|verdi|dve|dump] [NOBUILD=1] [BUILDONLY=1] [PA=1] [<scenario> ...]
   Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog.
   Valid scenarios are: all base_test callback_test shared_memory_test.

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

The dw_vip_setup Utility

The dw_vip_setup utility:

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

Setting Environment Variables

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

DESIGNWARE_HOME – Points to where the VIP is installed.

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 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: - amba_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: • amba_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: • amba_system_env_svt The -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {scenario | model/scenario} [-v[ersion] version]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulator run program for all supported simulators. If you specify a scenario (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command. **Note:** Use the -info switch to list all available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog VMM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {scenario | model/scenario}	Cleans the specified scenario/testbench in either the design directory (as specified by the -path switch) or the current working directory. This switch deletes all files in the specified directory, then restores all Synopsys created files to their original contents.

-i/nfo design home[:<product>[:<version>[:<methodology>]]]	Generate an informational report on a design directory or VIP installation. design: If the '-info design' switch is specified, the tool displays product and version content within the specified design directory to standard output. This output can be captured and used as a modellist file for input to this tool to create another design directory with the same content. home: If the '-info home' switch is specified, the tool displays product, version, and example content within the VIP installation to standard output. Optional filter fields can also be specified such as <product>, <version>, and <methodology> delimited by colons (:). An error will be reported if a nonexistent or invalid filter field is specified. Valid methodology names include: OVM, RVM, UVM, VMM and VLOG.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
model	The model names are: - amba_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.
-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
-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.
-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.
-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.

3 General Concepts

The following sections describe how the AMBA System Env component is structured in a UVM testbench.

AMBA System Env

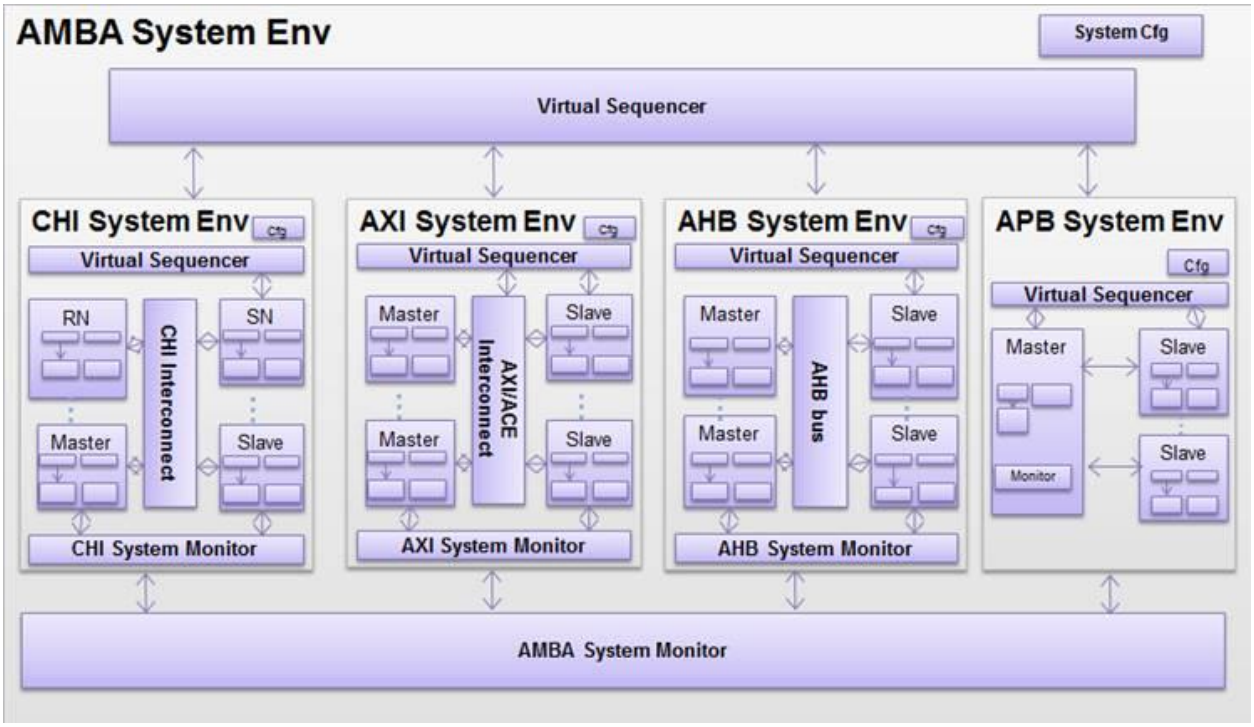


Figure 1 AMBA System Env

The AMBA System Env encapsulates array of CHI System Env, array of array of AXI System Env, array of AHB System Env and array of APB System Env components. It provides ease of use while verifying a system which has AXI, AHB and APB sub-systems.

You can instantiate one or more CHI, AXI, AHB and APB System Envs within AMBA System Env depending on the number of CHI/AXI/AHB/APB sub-systems in the design. The number of AXI, AHB and APB System Envs to be instantiated can be specified using below AMBA System configuration members:

svt_amba_system_configuration::num_chi_systems

svt_amba_system_configuration::num_axi_systems

svt_amba_system_configuration::num_ahb_systems

svt_amba_system_configuration::num_apb_systems

The maximum number of AXI System Envs supported in AMBA System Env is 32.

The maximum number of AHB System Envs supported in AMBA System Env is 32.

The maximum number of APB System Envs supported in AMBA System Env is 128.

The CHI System Env encapsulates CHI RN Agents, SN Agents, System Sequencer, CHI Interconnect , CHI System Monitor and the System Configuration. See CHI VIP User Guide for details.

The AXI System Env encapsulates AXI Master Agents, Slave Agents, System Sequencer, Interconnect Env, AXI System Monitor and the System Configuration. See AXI VIP User Guide for details.

The AHB System Env encapsulates AHB Master Agents, Slave Agents, System Sequencer, AHB System Monitor and the System Configuration. See AHB VIP User Guide for details.

The APB System Env encapsulates one APB Master Agent, Slave Agents, System Sequencer and the System Configuration. See APB VIP User Guide for details.

AMBA System Env contains AMBA System sequencer. AMBA System sequencer is a virtual sequencer with references to the virtual sequencers within CHI System Env, AXI System Env, AHB System Env and APB System Env. The system sequencer can be used to synchronize between the sequencers in CHI, AXI, AHB and APB System Envs.

AMBA System Env contains an array of AMBA System Monitors. You can instantiate one or more system monitors for monitoring AMBA system. The number of system monitors can be specified using AMBA System configuration member:

svt_amba_system_configuration::num_amba_system_monitors

You can specify the upstream ports and the downstream ports in the system which you wish to monitor using the system monitor. The system monitor is configured using AMBA System Monitor configuration. Upstream and downstream ports to be monitored can be specified using below members:

svt_amba_system_monitor_configuration::upstream_port_cfg[]

svt_amba_system_monitor_configuration::downstream_port_cfg[]

Refer to the glossary of the AXI specification for a definition of upstream and downstream components.

For example, if you are verifying AXI-AHB Bridge, you can specify AXI master port as upstream port, and AHB slave port as downstream port in the AMBA System monitor configuration. Refer to AMBA System Env Basic example for usage.

Alternatively, you can instantiate multiple system monitors, if you wish to monitor multiple sub-systems independently. In this case, specify upstream and downstream ports to be monitored for each system monitor.

The System Monitor performs system-level checks across the AXI, AHB and APB protocols. System monitor supports following checks:

- Data Integrity checks across AXI, AHB and APB ports
- Transaction routing checks across AXI, AHB and APB ports

AMBA System Env uses the address map specified in AXI, AHB and APB system configuration. Address map can be specified using the below members in AXI, AHB and APB system configuration:

svt_axi_system_configuration::set_addr_range

svt_ahb_system_configuration::set_addr_range

svt_apb_system_configuration:: slave_addr_ranges

4 AMBA VIP Programming Interface

This chapter presents the programming or user interface of the AMBA System Env.

Configuration objects

The AMBA System Env is controlled using configuration class svt_amba_system_configuration. The AMBA system configuration contains handles to system configurations of AXI, AHB and APB Envs instantiated within the AMBA System Env using below members:

svt_amba_system_configuration::axi_sys_cfg[]
svt_amba_system_configuration::ahb_sys_cfg[]
svt_amba_system_configuration::apb_sys_cfg[]

AMBA System configuration also contains handle to the AMBA System Monitor configuration:
svt_amba_system_configuration::amba_sys_mon_cfg[]

The AMBA system configuration mainly specifies:

- Number of AXI systems within the AMBA System Env (svt_amba_system_configuration::num_axi_systems)
- Number of AHB systems within the AMBA System Env (svt_amba_system_configuration::num_ahb_systems)
- Number of APB systems within the AMBA System Env (svt_amba_system_configuration::num_apb_systems)
- Number of AMBA System Monitors within the AMBA System Env (svt_amba_system_configuration::num_amba_system_monitors)

Refer to the AMBA VIP Class reference HTML documentation for details on individual members of configuration classes.

Transaction Objects

There are no transaction objects specific to AMBA System Env component. The AMBA System Env uses the AXI, AHB and APB VIP transaction objects. Refer to AXI, AHB and APB VIP User Guides for details.

Callbacks

In the AMBA System Env, the callback methods are called by System Monitor component.

Below is the callback class which contains the callback methods invoked by the System Monitor:

- svt_amba_system_monitor_callback

Refer to AMBA class reference HTML documentation for details of these classes.

Interfaces and modports

There are no interfaces specific to AMBA System Env. AMBA System Env uses the AXI, AHB and APB VIP interfaces.

Events

There are no events specific to AMBA System Env. User needs to use the AXI, AHB and APB VIP events.

5 Using the AMBA Verification IP

SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example for AMBA System Env. A summary of the examples is listed in Table 1 SystemVerilog Example Summary.

Table 1 SystemVerilog Example Summary

Example Name	Level	Description
tb_amba_svt_uvm_basic_sys	Basic	The example consists of the following: - A top-level testbench in SystemVerilog - A dummy DUT in the testbench, which represents a AXI-AHB bridge - A UVM verification environment - AMBA System component in the UVM verification environment, which instantiates 1 AXI System Env and 1 AHB System Env - One tests illustrating directed transaction generation.
tb_amba_svt_uvm_intermediate_sys	Intermediate	Not yet supported
tb_amba_svt_uvm_advanced_sys	Advanced	Not yet supported

Installing and Running the Examples

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

1. Install the example using the following command line:

% cd <location where example is to be installed>

% mkdir design_dir <provide any name of your choice>

% $DESIGNWARE_HOME/bin/dw_vip_setup -path ./design_dir –e amba_svt/tb_amba_svt_uvm_basic_sys –svlog

The example would get installed under:
<design_dir>/examples/sverilog/amba_svt/tb_amba_svt_uvm_basic_sys

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

a. Use the Makefile:

One test is provided in the "tests" directory.

1. ts.base_test.sv

To run test ts.base_test.sv, do following:

gmake USE_SIMULATOR=vcsvlog base_test WAVES=1

Invoke "gmake help" to show more options.

b. Use the sim script:

To run test ts.base_test.sv, do following:

./run_amba_svt_uvm_basic_sys -w base_test vcsvlog

Invoke "./run_amba_svt_uvm_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_amba_svt_uvm_basic_sys/README

OR

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

Support for UVM version 1.2

While using UVM 1.2, note the below requirements:

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

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/incl
ude \

-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_bas
ic_sys/. \

+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm
_basic_sys/

env \

+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm
_basic_sys/

dut \

+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm
_basic_sys/

hdl_interconnect \

+incdir+<testbench_dir>/examples/sverilog/ethernet_svt/tb_amba_svt_uvm
_basic_sys/

tests \

-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

How to integrate AMBA System Monitor in your environment?

AMBA System Env contains an array of AMBA System Monitors. You need to import the svt_amba.uvm.pkg and instantiate the svt_amba_system_env to use the AMBA System Monitor.

You can instantiate one or more system monitors for monitoring AMBA system. The number of system monitors can be specified using AMBA System configuration member:

svt_amba_system_configuration::num_amba_system_monitors

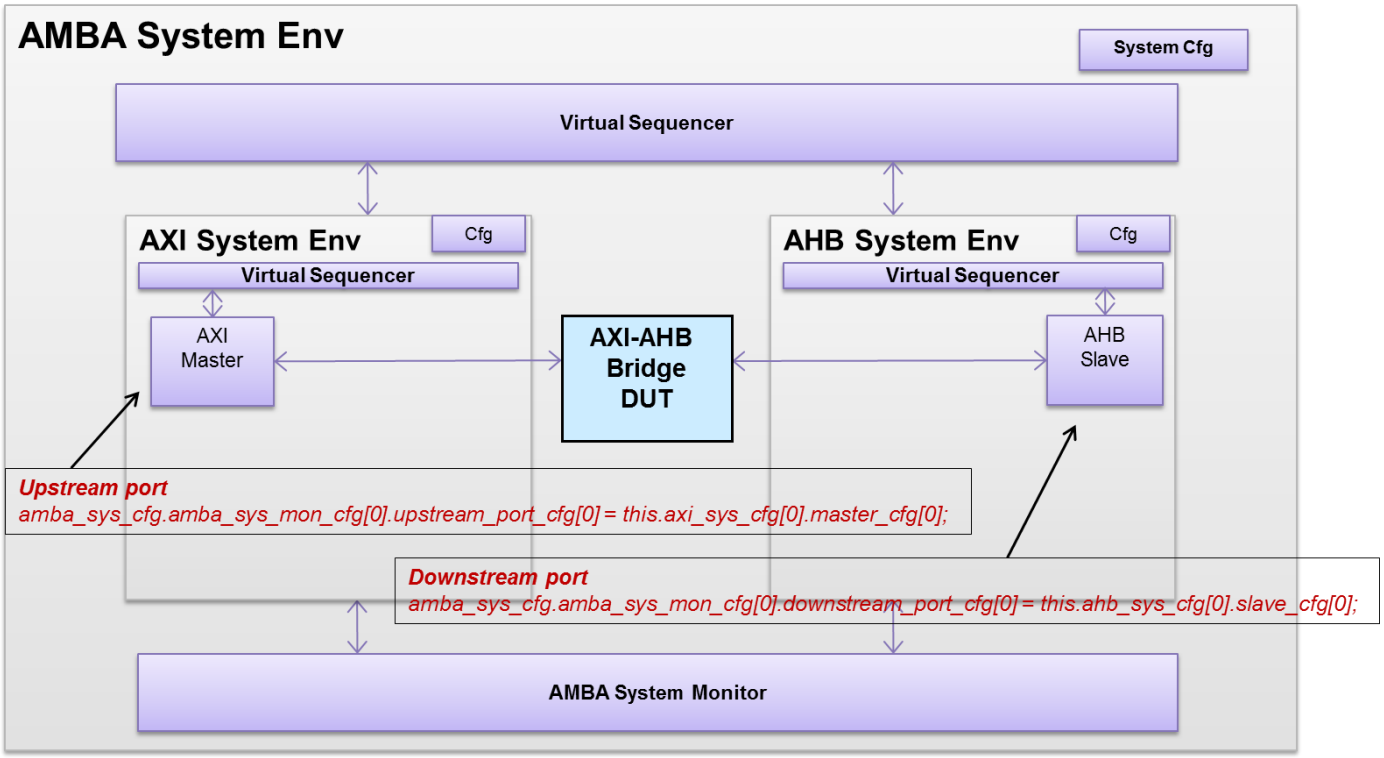
You can specify the upstream ports and the downstream ports in the system which you wish to monitor using the system monitor. The system monitor is configured using AMBA System Monitor configuration. Upstream and downstream ports to be monitored can be specified using below members:

svt_amba_system_monitor_configuration::upstream_port_cfg[]

svt_amba_system_monitor_configuration::downstream_port_cfg[]

These members need to be initialized to the port configuration of the upstream and downstream VIP components.

For example, if you are verifying AXI-AHB Bridge, you can specify AXI master port as upstream port, and AHB slave port as downstream port in the AMBA System monitor configuration.



Alternatively, you can instantiate multiple AMBA system monitors, if you wish to monitor multiple sub-systems independently. In this case, specify upstream and downstream ports to be monitored for each system monitor.

Refer to AMBA System Env Basic example for usage:

tb_amba_svt_uvm_basic_sys

Please refer to the following file

tb_amba_svt_uvm_basic_sys/env/cust_svt_amba_system_configuration.sv

```
/**
 In this example three AMBA system monitors are instantiated. Each monitor is connected to
a sub-system with the corresponding master ports programmed as upstream ports and the
slave ports programmed as downstream ports.
*/

this.amba_sys_mon_cfg = new[3];
```

```
this.amba_sys_mon_cfg[0] =
svt_amba_system_monitor_configuration::type_id::create("amba_sys_mon_cfg 1 ");
this.amba_sys_mon_cfg[1] =
svt_amba_system_monitor_configuration::type_id::create("amba_sys_mon_cfg 2");
this.amba_sys_mon_cfg[2] =
svt_amba_system_monitor_configuration::type_id::create("amba_sys_mon_cfg 3");

  this.amba_sys_mon_cfg[0].upstream_port_cfg = new[2];
  this.amba_sys_mon_cfg[0].upstream_port_cfg[0] = this.axi_sys_cfg[0].master_cfg[0];
  this.amba_sys_mon_cfg[0].upstream_port_cfg[1] = this.ahb_sys_cfg[0].master_cfg[0];
  this.amba_sys_mon_cfg[0].downstream_port_cfg = new[2];
  this.amba_sys_mon_cfg[0].downstream_port_cfg[0] = this.ahb_sys_cfg[0].slave_cfg[0];
  this.amba_sys_mon_cfg[0].downstream_port_cfg[1] = this.axi_sys_cfg[0].slave_cfg[0];

  this.amba_sys_mon_cfg[1].upstream_port_cfg = new[2];
  this.amba_sys_mon_cfg[1].upstream_port_cfg[0] = this.axi_sys_cfg[1].master_cfg[0];
  this.amba_sys_mon_cfg[1].upstream_port_cfg[1] = this.ahb_sys_cfg[1].master_cfg[0];
  this.amba_sys_mon_cfg[1].downstream_port_cfg = new[2];
  this.amba_sys_mon_cfg[1].downstream_port_cfg[0] = this.ahb_sys_cfg[1].slave_cfg[0];
  this.amba_sys_mon_cfg[1].downstream_port_cfg[1] = this.axi_sys_cfg[1].slave_cfg[0];

  this.amba_sys_mon_cfg[2].upstream_port_cfg = new[1];
  this.amba_sys_mon_cfg[2].upstream_port_cfg[0] = this.apb_sys_cfg[0];
  this.amba_sys_mon_cfg[2].downstream_port_cfg = new[1];
  this.amba_sys_mon_cfg[2].downstream_port_cfg[0] = this.apb_sys_cfg[0].slave_cfg[0];
```

Transaction Routing and Data Integrity Checks in AMBA System Monitor

The AMBA System Monitor performs two types of checks which are explained below:

Transaction routing checks:

This checks that a transaction is routed correctly. When a transaction is received at AXI, AHB or APB slave port, the system monitor checks if the address of the transaction as sampled on the slave port interface falls within the address map configured for the slave. The address map is specified in the configuration as follows (Please check tb_amba_svt_uvm_basic_sys for an example):

AXI -> svt_axi_system_configuration::set_addr_range()

AHB -> svt_ahb_system_configuration::set_addr_range()

APB -> set the properties within svt_apb_system_configuration::slave_addr_ranges.

Note that the `slave_addr_allocation_enable` property must be set to 0 if address needs to be explicitly configured for APB slaves.

For example:

```
foreach(apb_sys_cfg.slave_addr_ranges[i]) begin
      apb_sys_cfg.slave_addr_ranges[i] = new();
      apb_sys_cfg.slave_addr_ranges[i].start_addr = 0;
      apb_sys_cfg.slave_addr_ranges[i].end_addr = 0xFF;
      apb_sys_cfg.slave_addr_ranges[i].slave_id = i;
 end
```

An interesting scenario is when interconnect truncates the higher order bits while routing transactions to slaves. For example, let us say that an APB slave is configured in the system address map from 0x8000 to 0x80FF. The higher order bits may be truncated by interconnect, and only address from 0x00 to 0xFF may be transmitted to the APB slave. In such cases, the APB slave must still be configured according to the system address map (that is, from 0x8000 to 0x80FF). Then the higher order address bits of the slave VIP interface must be assigned with the fixed constant value and the lower order address bits must be connected to the DUT signals.

In the above example, following needs to be done:

assign <APB VIP IF>.slave_if[<index>].paddr[15:8] = 8'h80;

Data integrity Checks:

This check checks that the data/payload of a source transaction is transmitted correctly to the destination. The check works as follows:

1. Waits for an upstream transaction to complete at upstream port. Refer documentation of AMBA System monitor for a description of upstream and downstream ports.
2. Based on the address of the transaction, looks for the slave to which this transaction should be routed to, based on system address map. The slave to which the transaction is routed can be AXI, AHB or APB slave.
3. Each slave VIP component has an instance of the memory which is updated with transaction data. The Data Integrity check reads the memory of the target slave component.
4. Compares the contents of the memory with the contents of the upstream transaction and checks if they match.

Some points to take note of with respect to data integrity checks:

1. For this check to work correctly, an active slave must use a sequence that reads and writes the memory instantiated within the slave VIP. The VIPs come with memory access sequence which is capable of doing this.
2. A passive slave VIP automatically writes into memory in the slave based on activity observed on the interface.

Configuring AMBA slaves with overlapping address

Different AMBA slaves can have addresses that overlap. For example an AXI slave can have an address map that overlaps with an AHB slave. The user can set the address map of AXI using svt_axi_system_configuration::set_addr_range method and that of AHB using svt_ahb_system_configuration::set_addr_range method.

If there is overlap between slaves of the same AXI system env, the parameter svt_axi_system_configuration::allow_slaves_with_overlapping_addr must be set. Similarly, if there is overlap between slaves of the same AHB system env, the parameter svt_ahb_system_configuration::allow_slaves_with_overlapping_addr must be set. Any number of slaves can have overlapping addresses.

Note the following points:

- The user needs to pass a shared memory across slaves that share a common address space. This can be done by creating an instance of svt_mem in the testbench and passing the same svt_mem instance through uvm_config_db to the slave agents that have a shared address space. The following example shows how memory is shared between an AXI slave and AHB slave. Refer to AXI or AHB Class reference manual for arguments of constructor (new() method) of svt_mem class.

```
svt_mem  shared_mem = new (

        "axi_ahb_shared_mem", // Memory name
```

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

```
s0_s1_shared_mem.set_meminit (svt_mem::ADDRESS, 0, 0); // Memory initialization

uvm_config_db#(svt_mem)::set(this, "amba_system_env.axi_system[0].slave[0]",
”axi_slave_mem”, shared_mem);

uvm_config_db#(svt_mem)::set(this, "amba_system_env.ahb_system[0].slave[0]",
”ahb_slave_mem”,  shared_mem);
```

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

For an example on configuring slaves with overlapping address please refer shared_memory_test in tb_amba_svt_uvm_basic_sys example.

Why the User Needs to Disable Auto Item Recording

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

AXI and AHB protocols are pipelined protocols. In pipelined protocols, driver needs to initiate the next transaction before the previous transaction completes. Thus, the VIP driver indicates seq_item_port.item_done() much before the transaction is completed on the bus, so that the sequencer can provide next sequence item to the driver. Driver does not wait for a transaction to complete before calling seq_item_port.item_done(). For AXI, seq_item_port.item_done() is called as soon as the driver accepts a transaction from sequencer. For AHB, seq_item_port.item_done() is called when penultimate beat address of the current transaction is accepted by the slave. The VIP explicitly marks end of transaction when transaction actually completes on the interface, instead of letting UVM do it. Hence, VIP needs to define UVM_DISABLE_AUTO_ITEM_RECORDING macro.

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

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

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

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

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

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

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

6 Reporting Problems

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.

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*.

.

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:

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
```

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.

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).

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.

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.

Questa

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

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

Incisive

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

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

Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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.

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):

3. 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.
4. 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)

   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.

   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.

   The case submittal tool will display options on how to send the file to Synopsys.

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.