

VC Verification IP AMBA ATB UVM User Guide

Version O-2018.09, September 2018



Copyright Notice and Proprietary Information

© 2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

Preface	7
Guide Organization	7
Web Resources	7
Customer Support	8
Chapter 1	
Introduction	9
1.1 Introduction	9
1.2 Prerequisites	9
1.3 References	10
1.4 Product Overview	10
1.5 Language and Methodology Support	10
1.6 Feature Support	10
1.6.1 Protocol Features	10
1.6.2 Verification Features	11
1.6.3 Methodology Features	11
1.7 Features Not Supported	11
Chapter 2	
Installation and Setup	13
2.1 Verifying the Hardware Requirements	13
2.2 Verifying the Software Requirements	13
2.2.1 Platform/OS and Simulator Software	13
2.2.2 Synopsys Common Licensing (SCL) Software	14
2.2.3 Other Third Party Software	14
2.3 Preparing for Installation	14
2.4 Downloading and Installing	14
2.4.1 Downloading From the Electronic Software Transfer (EST) System (Download Center)	15
2.4.2 Downloading Using FTP with a Web Browser	16
2.5 Setting Up a Testbench Design Directory	16
2.6 What's Next?	16
2.6.1 Licensing Information	17
2.6.2 Environment Variable and Path Settings	20
2.6.3 Determining Your Model Version	20
2.6.4 Integrating a VIP into Your Testbench	20
Chapter 3	
General Concepts	27
3.1 Introduction to UVM	27
3.2 ATB VIP in an UVM Environment	27

3.2.1 Master Agent	27
3.2.2 Slave Agent	28
3.2.3 Slave Memory	29
3.2.4 System Environment	30
3.2.5 Active and Passive Mode	30
3.3 ATB UVM User Interface	31
3.3.1 3.3.1 Configuration Objects	31
3.3.2 Transaction Objects	32
3.3.3 Analysis Ports	33
3.3.4 Callbacks	34
3.3.5 Interfaces and Modports	35
3.3.6 Events	36
3.3.7 Overriding System Constants	36
3.4 Functional Coverage	37
3.4.1 Enabling Default Coverage	37
3.5 Protocol Checks	38
3.5.1 Comprehensive List of Protocol Checks	39
3.6 Reset Functionality	39
 Chapter 4	 41
Verification Features	41
4.1 ATB Sequence Collection	41
 Chapter 5	 43
Verification Topologies	43
5.1 Testing a Master DUT Using a UVM Slave VIP	43
5.2 Testing a Slave DUT Using a UVM Master VIP	45
5.3 System DUT with Passive VIP	46
5.4 System DUT with Mix of Active and Passive VIP	47
 Chapter 6	 49
Using ATB Verification IP	49
6.1 SystemVerilog UVM Example Testbenches	49
6.2 Installing and Running the Examples	49
6.2.1 Support for UVM version 1.2	50
6.3 How to Generate Slave Response	50
6.4 How to Disable Objection Management by VIP and Allow Testbench to Manage Objections	51
 Chapter 7	 53
Troubleshooting	53
7.1 Using Debug Port	53
 Appendix A	 55
Reporting Problems	55
A.1 Introduction	55
A.2 Debug Automation	55
A.3 Enabling and Specifying Debug Automation Features	55
A.4 Debug Automation Outputs	57
A.5 FSDB File Generation	57
A.5.1 VCS	58
A.5.2 Questa	58

A.5.3 Incisive58

A.6 Initial Customer Information58

A.7 Sending Debug Information to Synopsys58

A.8 Limitations59

Preface

About This Guide

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

Guide Organization

The chapters of this guide are organized as follows:

- ❖ Chapter 1, “[Introduction](#)”, introduces the Synopsys ATB 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 ATB VIP.
- ❖ Chapter 3, “[General Concepts](#)”, introduces the ATB VIP within a UVM environment and describes the data objects and components that comprise the VIP.
- ❖ Chapter 4, “[Verification Features](#)”, describes the topologies to verify master, slave and interconnect DUT.
- ❖ Chapter 5, “[Verification Topologies](#)”, describes the topologies to verify master, slave and interconnect DUT.
- ❖ Chapter 6, “[Using ATB Verification IP](#)”, provides the details of backward compatibility of the VIP with respect to previous releases.
- ❖ Chapter 7, “[Troubleshooting](#)”, describes the debug port.
- ❖ Appendix A, “[Reporting Problems](#)”, outlines the process for working through and reporting Synopsys ATB VIP issues.

Web Resources

- ❖ Documentation through SolvNet: <https://solvnet.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:

- ❖ Open a Case through SolvNet.
 - ◆ Go to <https://onlinecase.synopsys.com/Support/OpenCase.aspx> and provide the requested information, including:
 - ❖ Product: **Verification IP**
 - ❖ Sub Product: **AMBA SVT**
 - ❖ Tool Version: **O-2018.09**
 - ❖ Fill in the remaining fields according to your environment and your issue.
 - ◆ If applicable, provide the information noted in Appendix A, “Reporting Problems”.
- ❖ Send an e-mail message to support_center@synopsys.com.
 - ◆ Include the Product name, Sub Product name, and Tool Version (as noted above) in your e-mail so it can be routed correctly.
 - ◆ If applicable, provide the information noted in Appendix A, “Reporting Problems”.
- ❖ 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:
<http://www.synopsys.com/Support/GlobalSupportCenters>

1

Introduction

This chapter gives a basic introduction, overview and features of the AMBA[®] ATB UVM Verification IP.

This chapter discusses the following topics:

- ❖ [“Introduction”](#) on page 9
- ❖ [“Prerequisites”](#) on page 9
- ❖ [“References”](#) on page 10
- ❖ [“Product Overview”](#) on page 10
- ❖ [“Language and Methodology Support”](#) on page 10
- ❖ [“Feature Support”](#) on page 10
- ❖ [“Features Not Supported”](#) on page 11

1.1 Introduction

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

This approach leverages advanced verification technologies and tools that provide:

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

1.2 Prerequisites

- ❖ Familiarize with ATB, object oriented programming, SystemVerilog, and the current version of UVM.

1.3 References

For more information on ATB Verification IP, see the following:

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

SDESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/atb_svt_uvm_class_reference/html/index.html

1.4 Product Overview

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

The VIP provides an ATB System Env that contains the Master agents and Slave agents. The Master and Slave agents support all the functionality normally associated with active and passive UVM components, including the creation of transactions, checking and reporting the protocol correctness, transaction logging and functional coverage. After instantiating the System Env, you can select and combine active and passive agents to create an environment that verifies ATB features in the DUT.

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

1.5 Language and Methodology Support

In the current release, the ATB VIP suite supports the following languages and methodology:

- ❖ Languages
 - ◆ SystemVerilog
- ❖ Methodology
 - ◆ Qualified with UVM 1.1d and UVM 1.2.

For more information on UVM 1.2, see <http://www.accellera.org/>.

1.6 Feature Support

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

1.6.1 Protocol Features

ATB VIP currently supports the following protocol functions:

- ❖ ATB Trace Data Transfer (Valid, ready signaling)
- ❖ ATB Narrow Trace Data Transfer (Data Valid Bytes signaling)
- ❖ ATB Flow Control (Valid, ready signaling)
- ❖ ATB Flush Request Response (Flush Valid, Ready signaling with Data transfer)
- ❖ ATB Synchronization Request capture and pass to Trace Source (i.e. Sequence layer)
- ❖ ATB Slave Random Response for Trace Data Transfer
- ❖ ATB Slave Random Flush Request Generation
- ❖ ATB Slave Random Synchronization Request Generation
- ❖ ATB Slave Memory (for Trace Data capture)

1.6.2 Verification Features

ATB VIP currently supports the following verification functions:

- ❖ ATB Traffic Generation including Flush and Synchronization Request Service
- ❖ Protocol checking
- ❖ Debug port
- ❖ Programmable value (X, Z, hold previous), when valid signal is low
- ❖ Control on delays and timeouts
- ❖ Default functional coverage (transaction, state and toggle coverage)

1.6.3 Methodology Features

ATB VIP currently supports the following methodology functions:

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

1.7 Features Not Supported

The following features are currently not supported, however, will be supported in the future release.

- ❖ Protocol Checks
- ❖ Verification Features
 - ◆ Exception (error injection)
 - ◆ Sequence coverage
 - ◆ Analysis Port is currently not supported for Slave
- ❖ System Monitor

2

Installation and Setup

This chapter leads you through installing and setting up the Synopsys ATB UVM VIP. When you complete the checklist mentioned below, the provided example testbench will be operational and the Synopsys ATB UVM 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. [“Setting Up a Testbench Design Directory”](#)
6. [“What’s Next?”](#)

**Note**

If you encounter any problems with installing the Synopsys ATB VIP, see [“Customer Support”](#).

2.1 Verifying the Hardware Requirements

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

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

2.2 Verifying the Software Requirements

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

2.2.1 Platform/OS and Simulator Software

- ❖ **Platform/OS and VCS:** You need versions of your platform/OS and simulator that have been qualified for use. To see which platform/OS and simulator versions are qualified for use with the ATB VIP, check the support matrix manual.

2.2.2 Synopsys Common Licensing (SCL) Software

- ❖ The SCL software provides the licensing function for the Synopsys ATB 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 ATB VIP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from <http://www.adobe.com>.
- ❖ **HTML browser:** Synopsys ATB 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 ATB 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>.

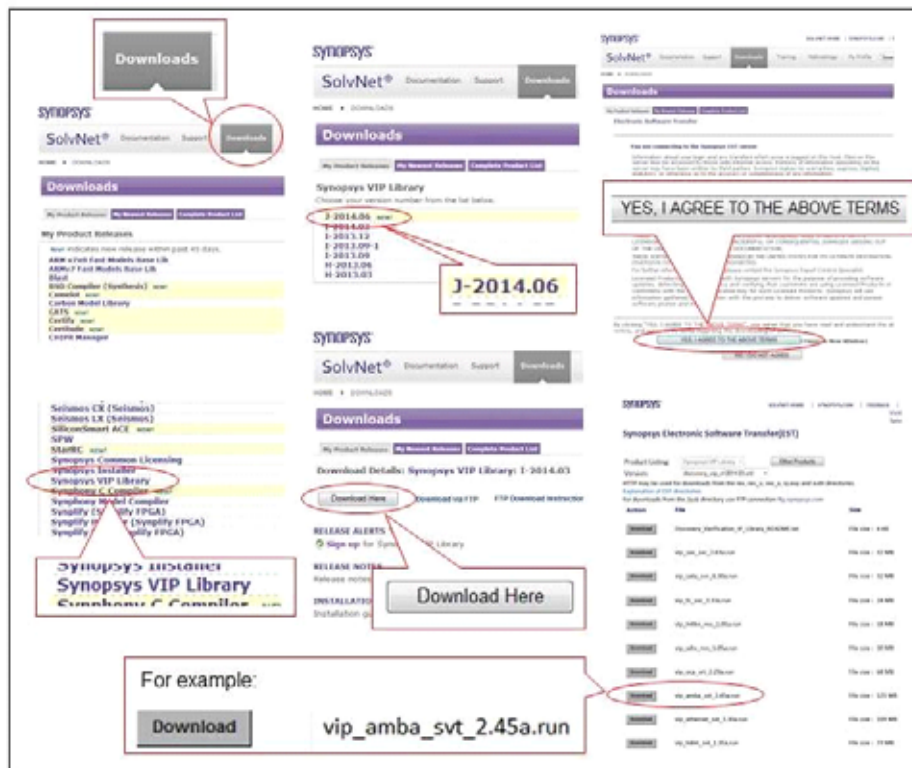
Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, refer to the following web page:

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)

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

Figure 2-1 SolvNet Selections for VIP Download



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


```
% setenv DESIGNWARE_HOME VIP_installation_path
```
- 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.

**Note**

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

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

**Note**

If you are unable to download the Verification IP using above instructions, see the [“Customer Support”](#) section to obtain support for download and installation.

2.5 Setting Up a Testbench Design Directory

A design directory is where the ATB VIP 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 transactors, support files (include files), and examples (if any)
- ❖ Add a specific version of ATB VIP from `DESIGNWARE_HOME` to a design directory.
For a full description of `dw_vip_setup`, see the [“dw_vip_setup Utility”](#) section.
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 atb_master_agent_svt -svtb
```

The models provided with ATB VIP include:

- ❖ `atb_master_agent_svt`
- ❖ `atb_system_env_svt`

**Note**

Set the value of macro `UVM_PACKER_MAX_BYTES` to 1500000 on command line. If you are using UVM, add the following to your command line:

```
+define+UVM_PACKER_MAX_BYTES=1500000
```

Else, ATB VIP will issue a fatal error.

Define the UVM macro `UVM_DISABLE_AUTO_ITEM_RECORDING`. ATB being a pipelined protocol (that is, previous transaction does not necessarily need to complete before starting new transaction), ATB VIP handles triggering the begin/end events and transaction recording. ATB VIP does not use the UVM automatic transaction begin/end event triggering and recording feature. If

`UVM_DISABLE_AUTO_ITEM_RECORDING` is not defined, VIP issues a FATAL message.

2.6 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 a VIP into Your Testbench”

2.6.1 Licensing Information

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

You can find general SCL information in the following location:

<http://www.synopsys.com/keys>

The Synopsys VIP for AMBA uses a licensing mechanism that is enabled by the following license features:

- ❖ VIP-AMBA-ATB-SVT
- ❖ VIP-AMBA-APB-SVT
- ❖ VIP-AMBA-AHB-SVT
- ❖ VIP-AMBA-STREAM-SVT
- ❖ VIP-AMBA-AXI-SVT
- ❖ VIP-AMBA-ACE-SVT
- ❖ VIP-AMBA-SVT
- ❖ VIP-PROTOCOL-SVT
- ❖ VIP-SOC-LIBRARY-SVT
- ❖ VIP-LIBRARY-SVT + DesignWare-Regression

These licenses enable the AMBA VIP components as follows:

- ❖ VIP-AMBA-ATB-SVT enables ATB components.
- ❖ VIP-AMBA-APB-SVT enables APB2, APB3, APB4 components.
- ❖ VIP-AMBA-AHB-SVT enables AHB2, AHB3, AHB5, AHB-Lite, AHB Multi-Layer components.
- ❖ VIP-AMBA-STREAM-SVT enables AXI4 Stream components.
- ❖ VIP-AMBA-AXI-SVT enables AXI3, AXI4, AXI4-Lite components.
- ❖ VIP-AMBA-ACE-SVT enables AXI3, AXI4, AXI4-Lite, ACE, ACE-Lite components.
- ❖ VIP-AMBA-SVT enables ATB/APB2/APB3/APB4/AHB2/AHB3/AHB5/AHB-Lite/AHB-Multilayer/AXI4-Stream/AXI3/AXI4/AXI4-Lite/ACE/ACE-Lite components.

The order in which licenses are checked out is as describes below. The following table summarizes the license requirements for different AMBA interfaces.

The sequence in which the licenses are checked out are described in [Table 2-1](#).

Table 2-1 License Requirements for AMBA Interfaces

VIP Interface	License Checkout Order
ATB	VIP-AMBA-ATB-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression
APB2/APB3/APB4	VIP-AMBA-APB-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression
AHB2/ AHB3/ AHB5, AHB-Lite/ AHB Multi-Layer	VIP-AMBA-AHB-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression
AXI4 STREAM	VIP-AMBA-STREAM-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression
AXI3/AXI4/AXI4-Lite	VIP-AMBA-AXI-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression

Table 2-1 License Requirements for AMBA Interfaces

VIP Interface	License Checkout Order
AXI3/AXI4/AXI4-Lite/ACE/ACE-Lite	VIP-AMBA-ACE-SVT -OR- VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression
ATB/APB2/APB3/APB4/AHB2/AHB3/AHB5/AHB-Lite/AHB-Multilayer/AXI4-Stream/AXI3/AXI4/AXI4-Lite/ACE/ACE-Lite	VIP-AMBA-SVT -OR- VIP-PROTOCOL-SVT -OR- VIP-LIBRARY-SVT + DesignWare-Regression

The following is the description of license consumption for each license type:

- ❖ VIP-AMBA-ATB-SVT, VIP-AMBA-APB-SVT, VIP-AMBA-AHB-SVT, VIP-AMBA-STREAM-SVT, VIP-AMBA-AXI-SVT, VIP-AMBA-ACE-SVT: Single license enables multiple instances of any of AHB/APB/AXI/ACE/ATB/AXI4-STREAM VIP in a single simulation session.
- ❖ VIP-AMBA-SVT: Single license enables multiple instances of AMBA VIP in a single simulation session.
- ❖ VIP-PROTOCOL-SVT: Single license enables multiple instances of a single protocol suite of VIP in a single simulation session. Multiple licenses required to enable multiple VIP protocol suites in a single simulation session or multiple simultaneous simulation sessions.
- ❖ VIP-SOC-LIBRARY-SVT
- ❖ VIP-LIBRARY-SVT + DesignWare-Regression: Single license enables multiple instances of any number of protocol suites in a single simulation session.

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”](#).

2.6.1.1 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.

2.6.1.2 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 one.
- ❖ To disable license polling, unset the `DW_WAIT_LICENSE` environment variable. By default, license polling is disabled.

2.6.1.3 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.

**Note**

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

2.6.2 Environment Variable and Path Settings

The following are environment variables and path settings required by the ATB 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.

2.6.2.1 Simulator-Specific Settings

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

2.6.3 Determining Your Model Version

The version of the ATB VIP verification models at the time of publication is less than 1.0a. The following steps help you to check the version of the models.

**Note**

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

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

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

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

2.6.4 Integrating a 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”](#)

❖ “dw_vip_setup Utility”

2.6.4.1 Creating a Testbench Design Directory

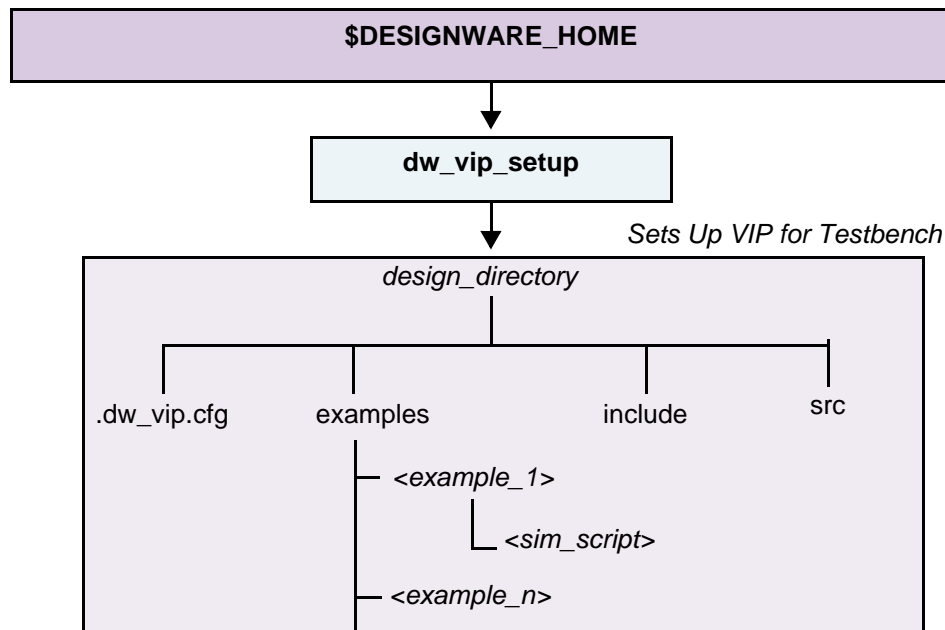
A *design directory* contains a version of VIP that is set up and ready for use in a testbench. You use the `dw_vip_setup` utility to create design directories. For the full description of `dw_vip_setup`, see the “dw_vip_setup Utility”.

**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 Synopsys VIP in your testbench because it is isolated from the `DESIGNWARE_HOME` installation. 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.

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 Synopsys VIP Models from a Design Directory”
- ❖ “Reporting Information About `DESIGNWARE_HOME` or a Design Directory”

2.6.4.1.1 Adding or Updating VIP Models In a Design Directory

ATB VIP models include:

- ❖ `atb_master_agent_svt`
- ❖ `atb_slave_agent_svt`
- ❖ `atb_system_env_svt`

The following example adds a ATB VIP model to a design directory:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -a atb_master_agent_svt -svtb
```

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

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

In these examples, the `dw_vip_setup` utility does the following:

- ❖ Creates an include directory under the current directory and copies:
 - ◆ All files in the `atb_master_agent_svt` model include directory.
 - ◆ All include files in the ATB VIP suite.
 - ◆ The latest library include files into the include directory.
- ❖ Creates the ATB VIP suite libraries and UVM libraries.

2.6.4.1.2 Removing Synopsys VIP Models from a Design Directory

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

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

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

2.6.4.1.3 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
```

2.6.4.2 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_atb_svt_uvm_basic_sys --help
```

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

where <scenario> is one of: all base_test directed_test random_wr_rd_test

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

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

-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] [BUILDOONLY=1] [PA=1]  
[<scenario> ...]
```

Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcssclog nclog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog

Valid scenarios are: all base_test directed_test random_wr_rd_test



Note

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

2.6.4.3 dw_vip_setup Utility

The `dw_vip_setup` utility performs the following:

- ❖ Adds, removes, or updates ATB VIP models in a design directory.
- ❖ Adds example testbenches to a design directory, the ATB 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.

2.6.4.3.1 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.

2.6.4.3.2 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 2-2](#) lists the switches and their applicable sub-switches.

Table 2-2 Setup Program Switch Descriptions

Switch	Description
-a [dd] (<i>model</i> [-v[ersion] <i>version</i>]) ...	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: <ul style="list-style-type: none"> • <code>atb_master_agent_svt</code> • <code>atb_slave_agent_svt</code> • <code>atb_system_env_svt</code> The <code>-add</code> switch causes <code>dw_vip_setup</code> to build suite libraries from the same suite as the specified models, and to copy the other necessary files from <code>\$DESIGNWARE_HOME</code> .
-r [emove] <i>model</i>	Removes all versions of the specified model or models from the design. The <code>dw_vip_setup</code> program does not attempt to remove any include files used solely by the specified model or models. The model names are: <ul style="list-style-type: none"> • <code>atb_master_agent_svt</code> • <code>atb_slave_agent_svt</code> • <code>atb_system_env_svt</code>
-u [pdate] (<i>model</i> [-v[ersion] <i>version</i>]) ...	Updates to the specified model version for the specified model or models. The <code>dw_vip_setup</code> script updates to the latest models when you do not specify a version. The model names are: <ul style="list-style-type: none"> • <code>atb_master_agent_svt</code> • <code>atb_slave_agent_svt</code> • <code>atb_system_env_svt</code> The <code>-update</code> switch causes <code>dw_vip_setup</code> to build suite libraries from the same suite as the specified models, and to copy the other necessary files from <code>\$DESIGNWARE_HOME</code> .

Table 2-2 Setup Program Switch Descriptions (Continued)



Switch	Description
-e [xample] { <i>scenario</i> <i>model/scenario</i> } [-v[ersion] <i>version</i>]	<p>The <code>dw_vip_setup</code> 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.</p> <p>If you specify a <i>scenario</i> (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command.</p> <p> Note Use the <code>-info</code> switch to list all available system examples.</p>
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog UVM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c [lean] { <i>scenario</i> <i>model/scenario</i> }	Cleans the specified scenario/testbench in either the design directory (as specified by the <code>-path</code> switch) or the current working directory. This switch deletes <i>all files in the specified directory</i> , then restores all Synopsys created files to their original contents.
-i nfo design home[:<product>[:<version>[:<methodology>]]]	<p>Generate an informational report on a design directory or VIP installation.</p> <p>design: If the <code>'-info design'</code> 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.</p> <p>home: If the <code>'-info home'</code> 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.</p>
-h [elp]	Returns a list of valid <code>dw_vip_setup</code> switches and the correct syntax for each.
<i>model</i>	<p>Synopsys ATB VIP models are:</p> <ul style="list-style-type: none"> • <code>atb_master_agent_svt</code> • <code>atb_slave_agent_svt</code> • <code>atb_system_env_svt</code> <p>The <i>model</i> argument defines the model or models that <code>dw_vip_setup</code> acts upon. This argument is not needed with the <code>-info</code> or <code>-help</code> switches. All switches that require the <i>model</i> argument may also use a model list.</p> <p>You may specify a version for each listed <i>model</i>, using the <code>-version</code> option. If omitted, <code>dw_vip_setup</code> uses the latest version. The <code>-update</code> switch ignores <i>model</i> version information.</p>

Table 2-2 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 <code>DESIGNWARE_HOME</code> installation. All product versions contained in the design directory must also exist in the current <code>DESIGNWARE_HOME</code> installation.
-doc	Creates a doc directory in the specified design directory which is populated with symbolic links to the <code>DESIGNWARE_HOME</code> 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.

For more information on installing and running SystemVerilog UVM example testbenches, see [“SystemVerilog UVM Example Testbenches”](#) and [“Installing and Running the Examples”](#) sections.

3

General Concepts

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

- ❖ “Introduction to UVM”
- ❖ “ATB VIP in an UVM Environment” on page 27
- ❖ “Reset Functionality” on page 40

3.1 Introduction to UVM

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

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

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

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

3.2 ATB VIP in an UVM Environment

The following sections describe how the ATB Verification IP is structured in a UVM testbench.

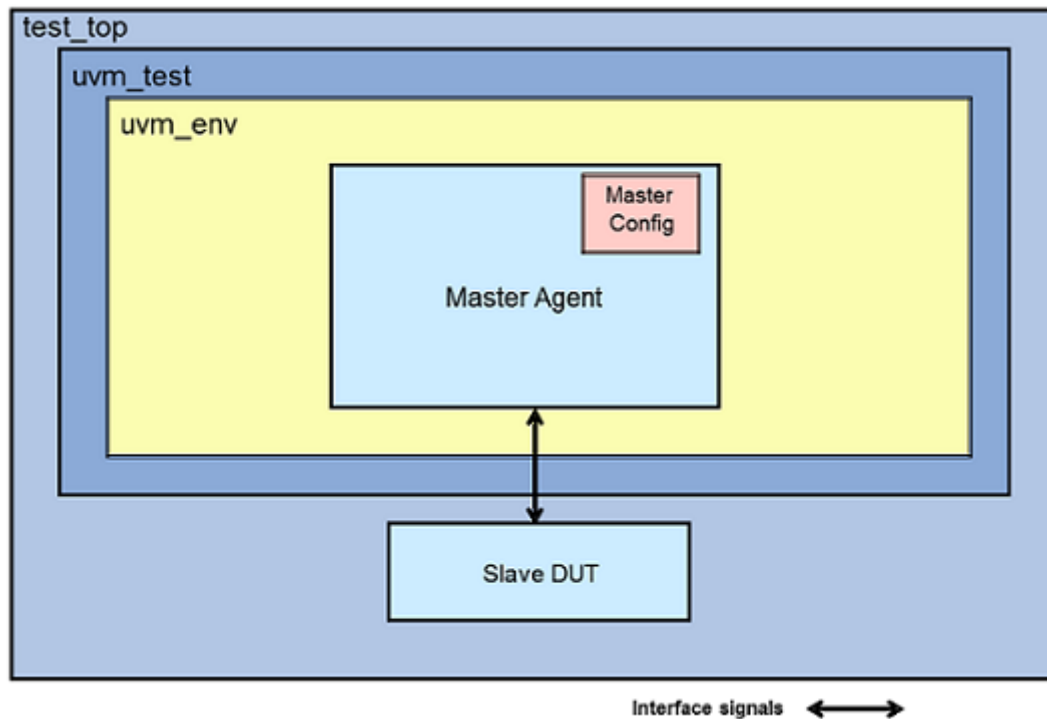
3.2.1 Master Agent

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

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

Within the Master Agent, the Master Driver gets sequences from the Master Sequencer. The Master Driver then drives the ATB transactions on the ATB port. The Master Driver and port Monitor components within Master Agent call callback methods at various phases of execution of the ATB transaction. Details of callbacks are covered in later sections. After the ATB transaction on the bus is complete, the completed sequence item is provided to the analysis port of Port Monitor for use by the testbench.

Figure 3-1 Usage With Standalone Master Agent



3.2.2 Slave Agent

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

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

In the Slave Agent, the Port Monitor samples the ATB port signals. When a new transaction is detected, the Port Monitor provides a response request sequence item to the Slave Sequencer through port `response_request_port`. The slave response sequence within the sequencer programs the appropriate slave

response. The updated response sequence item is then provided by the Slave Sequencer to the Slave Driver. The Slave Driver in turn drives the response on the ATB bus.

**Note**

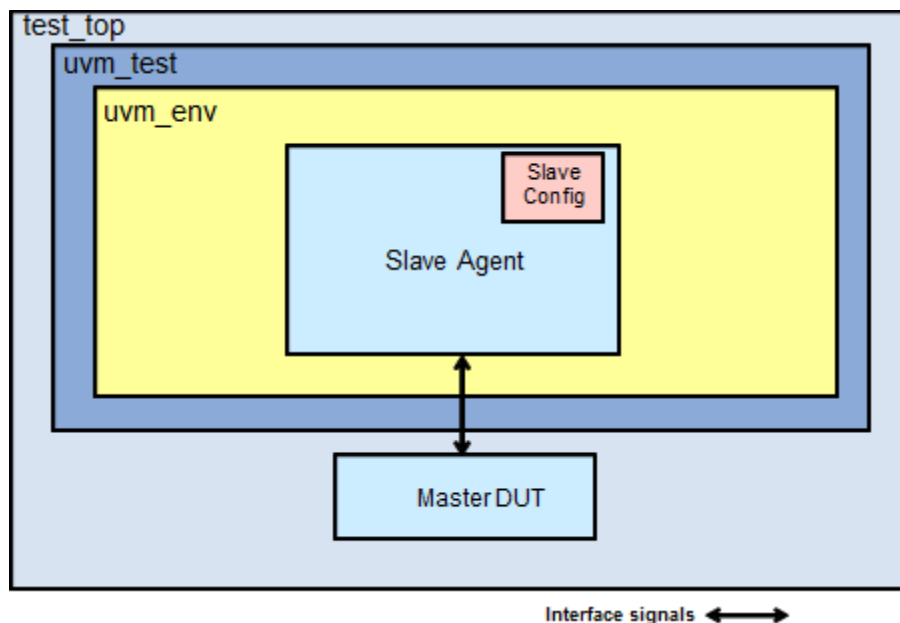
The slave driver expects the slave response sequence to:

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

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

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

Figure 3-2 Usage With Standalone Slave Agent



3.2.3 Slave Memory

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

See slave sequence `svt_atb_slave_base_sequence` in following available at:

```
$DESIGNWARE_HOME/vip/svt/amba_svt/latest/atb_slave_agent_svt/sverilog/src/vcs/svt_atb_slave_sequence_collection.svp.
```

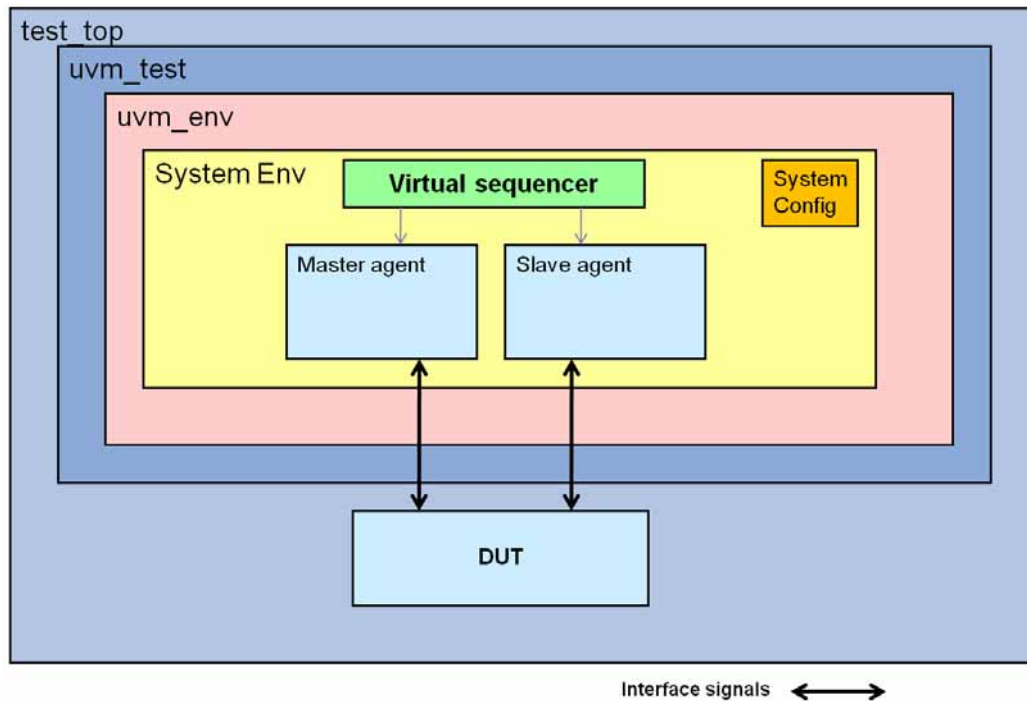
A reference to the slave memory instantiated in the slave agent is provided in the slave sequence. Using the API of this slave memory, you can read or write into the slave memory through this base sequence, or sequence extended from this base sequence.

See ATB SVT class reference HTML documentation for details on `svt_mem` and `svt_atb_slave_base_sequence`.

3.2.4 System Environment

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

Figure 3-3 Usage With System Environment



3.2.4.1 System Sequencer

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

3.2.5 Active and Passive Mode

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

Table 3-1 Agents in Active and Passive Mode

Component Behavior in Active Mode	Component Behavior in Passive Mode
In active mode, Master and Slave components generate transactions on the signal interface.	In passive mode, master and slave components do not generate transactions on the signal interface. These components only sample the signal interface.

Table 3-1 Agents in Active and Passive Mode

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

3.3 ATB UVM User Interface

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

3.3.1 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 Agent or System Environment.

The configuration can be of the following two types:

- ❖ **Static Configuration Properties**

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

- ❖ **Dynamic Configuration Properties**

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

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

The ATB VIP defines following configuration classes:

- ❖ **System configuration (`svt_atb_system_configuration`)**

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

need to provide the system configuration to the system env from the environment or the testcase. The system configuration mainly specifies:

- ◆ Number of master and slave agents in the system env
- ◆ Port configurations for master and slave agents
- ◆ Virtual top level ATB interface
- ◆ Timeout values
- ❖ Port configuration (`svt_atb_port_configuration`)
The Port configuration class contains configuration information which is applicable to individual ATB master or slave agents in the system environment. Some of the important information provided by port configuration class is as follows:
 - ◆ Active/Passive mode of the master/slave port agent
 - ◆ Enable/disable protocol checks
 - ◆ Enable/disable port-level coverage
 - ◆ Interface type (ATB1_0/ATB1_1)
 - ◆ Port configuration contains the virtual interface for the port

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

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

3.3.2 Transaction Objects

Transaction objects, which are extended from the `uvm_sequence_item` base class, define a unit of ATB 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.

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

These data objects extend from the `uvm_sequence_item` base class and implement all methods specified by UVM for that class.

ATB transaction data objects are used to:

- ❖ Generate random stimulus.
- ❖ Report observed transactions.
- ❖ Generate random responses to transaction requests.
- ❖ Collect functional coverage statistics.

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

- ❖ `valid_ranges` constraints limit generated values to those acceptable to the drivers. These constraints ensure basic VIP operation and should never be disabled.
- ❖ `reasonable_*` constraints, which can be disabled individually or as a block, limit the simulation by:
 - ◆ 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.

ATB VIP defines the following transaction classes:

- ❖ **ATB Base transaction** (`svt_atb_transaction`)

This is the base transaction type which contains all the physical attributes of the transaction like id, data, burst length, etc. It also provides the timing information the transaction, to the master and slave drivers, that is, delays for valid and ready signals with respect to some reference events.
- ❖ **ATB Master transaction** (`svt_atb_master_transaction`)

The master transaction class extends from the ATB transaction base class `svt_atb_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_atb_master_transaction` from its analysis ports, in active and passive mode.
- ❖ **ATB Slave transaction** (`svt_atb_slave_transaction`)

The slave transaction class extends from the ATB transaction base class `svt_atb_transaction`. The slave transaction class contains the constraints for slave specific members in the base transaction class. At the end of each transaction, the slave agent provides object of type `svt_atb_slave_transaction` from its analysis ports, in active and passive mode.

The master and slave transactions contain a handle to configuration object of type `svt_atb_port_configuration`, which provides the configuration of the port on which this transaction is applied. The port configuration is used during randomizing the transaction. The port configuration is available in the sequencer of the master/slave agent.

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

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

3.3.3 Analysis Ports

The Port Monitor in the Master Agent provides an analysis port `item_observed_port`. At the end of the transaction, the Master and Slave Agents respectively write the completed `svt_atb_master_transaction` and `svt_atb_slave_transaction` object to the analysis port. This holds true in active as well as passive

mode of operation of the master/slave agent. You can use this analysis port for connecting to scoreboard, or any other purpose, where a transaction object for the completed transaction is required.

3.3.4 Callbacks

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

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

ATB VIP uses callbacks in three main applications:

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

3.3.4.1 Master Agent Callbacks

In the Master Agent, the callback methods are called by Master Driver and Port Monitor components. The following callback classes which contain the callback methods are invoked by the Master Agent:

- ❖ `svt_atb_port_monitor_callback`
 - ◆ `pre_output_port_put`
 - ◆ `new_transaction_started`
 - ◆ `transaction_ended`
- ❖ `svt_atb_master_callback`
 - ◆ `post_input_port_get`

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

3.3.4.2 Slave Agent Callbacks

In the Slave Agent, the callback methods are called by Slave Driver and port monitor components. The following callback classes which contain the callback methods are invoked by the Slave Agent:

- ❖ `svt_atb_port_monitor_callback`

- ◆ `pre_output_port_put`
- ◆ `new_transaction_started`
- ◆ `transaction_ended`
- ❖ `svt_atb_slave_callback`
 - ◆ `post_input_port_get`
 - ◆ `post_resp_phase_ended`

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

3.3.5 Interfaces and Modports

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

ATB VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top-level interface `svt_atb_if` is defined. The top-level interface contains an array of Master port sub-interfaces of type `svt_atb_master_if`, and Slave port sub-interfaces of type `svt_atb_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_atb_system_configuration::set_if`.

Alternatively, the interface can also be specified to the ATB System Env component directly through UVM Configuration database. Refer to ATB Basic example `tb_atb_svt_uvm_basic_sys` for usage.

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

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

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

For details on ATB Interface, see

`$DESIGNWARE_HOME/vip/svt/amba_svt/latest/doc/atb_svt_uvm_class_reference/html/interfaces.html`.

3.3.5.1 Modports

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

- ❖ `svt_atb_slave_modport`

This modport is used to connect master VIP component to slave DUT port.

- ❖ `svt_atb_debug_modport`

This modport can be used by you to access the debug port signals. See "Using Debug Port" for details on debug port.

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

- ❖ `svt_atb_master_modport`

This modport is used to connect slave VIP component to master DUT port.

- ❖ `svt_atb_debug_modport`

This modport can be used by you to access the debug port signals. See *Using Debug Port* for details on debug port.

3.3.5.2 Clocking Modes

The interface works in the following two clocking modes:

- ❖ Common clock mode
- ❖ Multiple clock mode

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

When this configuration parameter is set to zero, the `aclk` signal of each port sub-interface would need to be connected to appropriate clock in the environment.

3.3.5.2.1 Common Clock Mode

In this mode,

- ❖ All port sub-interfaces will operate on a single common clock.
- ❖ Connect system clock to the `common_aclk` signal in the top interface.
- ❖ Top-level interface will pass the common clock signal down to all port sub-interfaces.

3.3.5.2.2 Multiple Clock Mode

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

3.3.6 Events

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

`begin_event` is issued on the rising clock edge when first `atvalid` is asserted for that transaction `end_event` is issued on the rising clock edge when already of the last data transfer of the current transaction is received. However, if current transaction is sent in response to Flush Request then `end_event` is issued after asserting already.

3.3.7 Overriding System Constants

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

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

❖ `svt_atb_defines.svi`

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

❖ `svt_atb_common_defines.svi`

This file defines common constants used by the ATB VIP components. You can override only the *User Definable* constants, which are declared in *ifndef* statements, such as the following:

```
`ifndef SVT_ATB_MAX_DATA_WIDTH
`define SVT_ATB_MAX_DATA_WIDTH - 32
`endif
```

❖ `svt_atb_port_defines.svi`

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

❖ `svt_atb_user_defines.svi`

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

To override the `SVT_ATB_MAX_ID_WIDTH` constant from the `svt_atb_port_defines.svi` file,

❖ Redefine the corresponding symbol in the `svt_atb_user_defines.svi` file.

For example:

```
`define SVT_ATB_MAX_ID_WIDTH 12
```

❖ In the simulator compile command,

◆ Ensure that the directory containing `svt_atb_user_defines.svi` is provided to the simulator

◆ Provide `SVT_ATB_INCLUDE_USER_DEFINES` on the simulator command line as follows:

```
+define+SVT_ATB_INCLUDE_USER_DEFINES
```



Note

The following restrictions must be noted when overriding the default maximum footprint:
Do not use a value of 0 for a `MAX_*_WIDTH` value. The value must be `>= 1`

3.4 Functional Coverage

The ATB VIP provides various levels of coverage support. This section describes the levels of support.

3.4.1 Enabling Default Coverage

The default functional coverage can be enabled by setting the following attributes in the port configuration class `svt_atb_port_configuration` to one. To disable coverage, set the attributes to zero.

The attributes are:

- ❖ `toggle_coverage_enable`
- ❖ `state_coverage_enable`
- ❖ `transaction_coverage_enable`

By default, the coverage is disabled.

3.4.1.1 Toggle Coverage

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

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

3.4.1.2 State Coverage

State coverage is a signal level coverage. State coverage applies to signals that are a minimum of two bits wide. In most cases, the states (also commonly referred to as coverage bins) can be easily identified as all possible combinations of the signal. For example, for the `ATID[6:0]` signal, the states would be `0x00 .. 0x7F`

If the state space is too large, an intelligent classification of the states must be made.

3.4.1.3 Delay Coverage

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

- ❖ `ATVALID_to_ATREADY_Delay` : delays between signals `atvalid` and `atready`.
- ❖ `ATVALID_to_prev_ATVALID_Delay` : delays between current and previous `atvalid` signals.
- ❖ `ATVALID_before_ATREADY` : captures if `ATVALID` asserted before `ATREADY` signal.
- ❖ `ATREADY_before_ATVALID` : captures if `ATREADY` asserted before `ATVALID` signal.

3.4.1.4 Transaction Cross Coverage

Provides Cross Coverage between relevant Transaction attribute coverages. Currently it supports only cross between different ATB transaction event types like, Trace Data, Flush Request, Trace Trigger etc. with Data Valid Bytes that is `ATBYTES`.

For more details on actual cover groups, refer to the ATB VIP Class Reference HTML document.

3.5 Protocol Checks

The protocol checks can be enabled by setting the configuration attribute `protocol_checks_enable` in class `svt_atb_port_configuration` to one. To disable the checks, set the attribute to zero. By default, the protocol checks are enabled.

3.5.1 Comprehensive List of Protocol Checks

Important ATB protocol checks are described in the following sections. For a comprehensive list of all protocol checks for ATB protocol, see `svt_atb_checker` class in ATB VIP Class Reference HTML documentation.

Table 3-2 ATB Protocol Check

Protocol Check	Check Condition
signal_valid_atid_when_atvalid_high_check	ATID is not X or Z when ATVALID is high
signal_valid_atdata_when_atvalid_high_check	ATDATA is not X or Z when ATVALID is high
signal_valid_atbytes_when_atvalid_high_check	ATBYTES is not X or Z when ATVALID is high
signal_valid_atready_when_atvalid_high_check	ATREADY is not X or Z when ATVALID is high
signal_valid_afready_when_afvalid_high_check	AFREADY is not X or Z when AFVALID is high
signal_valid_atvalid_check	Monitor Check for X or Z on ATVALID
signal_valid_afvalid_check	Monitor Check for X or Z on AFVALID
signal_valid_syncreq_check	Monitor Check for X or Z on SYNCREQ
signal_stable_atid_when_atvalid_high_check	ATID is stable when ATVALID is high
signal_stable_atdata_when_atvalid_high_check	ATDATA is stable when ATVALID is high
signal_stable_atbytes_when_atvalid_high_check	ATBYTES is stable when ATVALID is high
atvalid_low_when_reset_is_active_check	Monitor Check for ATVALID low when reset is active
afvalid_low_when_reset_is_active_check	Monitor Check for AFVALID low when reset is active
syncreq_low_when_reset_is_active_check	Monitor Check for SYNCREQ low when reset is active
atvalid_interrupted_check	Monitor Check for ATVALID being deasserted before ATREADY assertion
afvalid_interrupted_check	Monitor Check for AFVALID being deasserted before ATREADY assertion
atid_reserved_val_check	Monitor Checks that ATID doesn't have RESERVED Values
atdata_valid_val_check	Monitor Checks that ATDATA has legal values for corresponding ATID provided

3.6 Reset Functionality

The ATB VIP samples the reset signal synchronously. This means that, when reset is asserted or de-asserted, it is required that the clock connected to VIP is running. If the clock input to VIP is not running, assertion and de-assertion of reset is not detected, and VIP would not sample and drive any signals.

When reset signal is asserted, all the transactions in master and slave agents whose transaction processing threads are in progress are aborted. The `xact_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 transactions whose data transfer is in progress, transaction ENDED notification is issued on rising edge of clock when reset signal assertion is observed.

4

Verification Features

This chapter describes the verification features available along with ATB Verification IP:

4.1 ATB Sequence Collection

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

All the ATB Master sequences are extended from base sequence `svt_atb_master_base_sequence`.

For a list of all the master and slave sequences, refer to ATB VIP class reference HTML documentation.

ATB VIP provides a pre-defined ATB Master sequence library

`svt_atb_master_transaction_sequence_library`, which can hold the ATB Master sequences. The library by default has no registered sequences. You are expected to call

`svt_atb_master_transaction_sequence_library::populate_library()` method to populate the sequence library with master sequences provided with the VIP. The port configuration is provided to the `populate_library()` method. Based on the port configuration, appropriate sequences are added to the sequence library. You can then load the sequence library in the sequencer within the master agent. The ATB master sequences are not encrypted, however, provided as source code.

5

Verification Topologies

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

- ❖ “Testing a Master DUT Using a UVM Slave VIP”
- ❖ “Testing a Slave DUT Using a UVM Master VIP”
- ❖ “System DUT with Passive VIP”
- ❖ “System DUT with Mix of Active and Passive VIP”

5.1 Testing a Master DUT Using a UVM Slave VIP

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

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

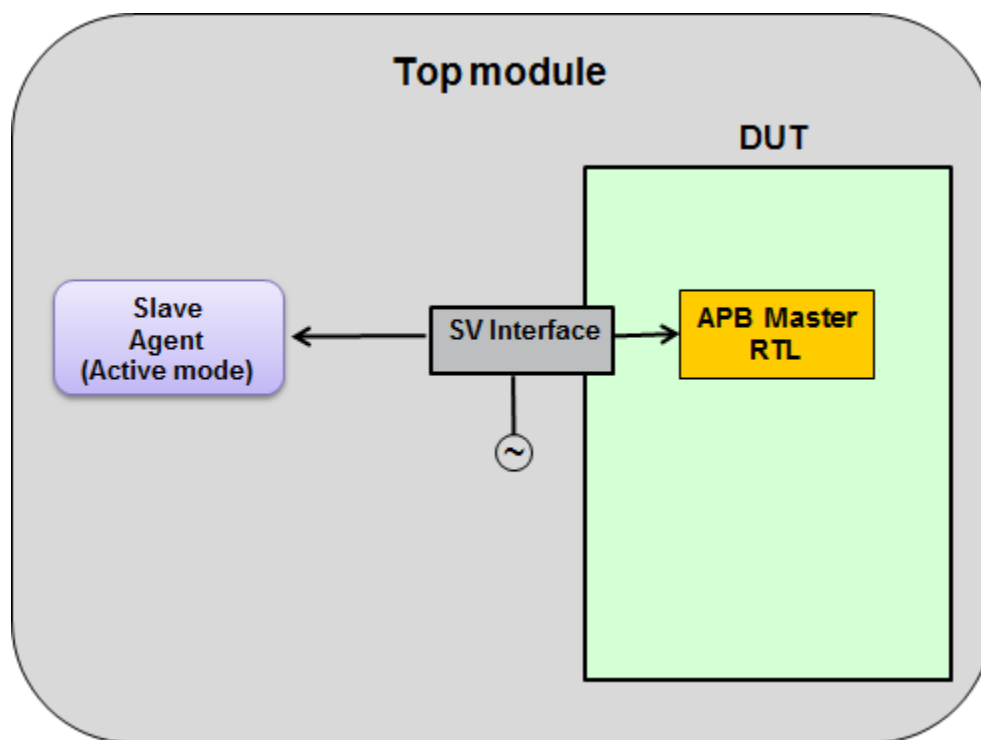
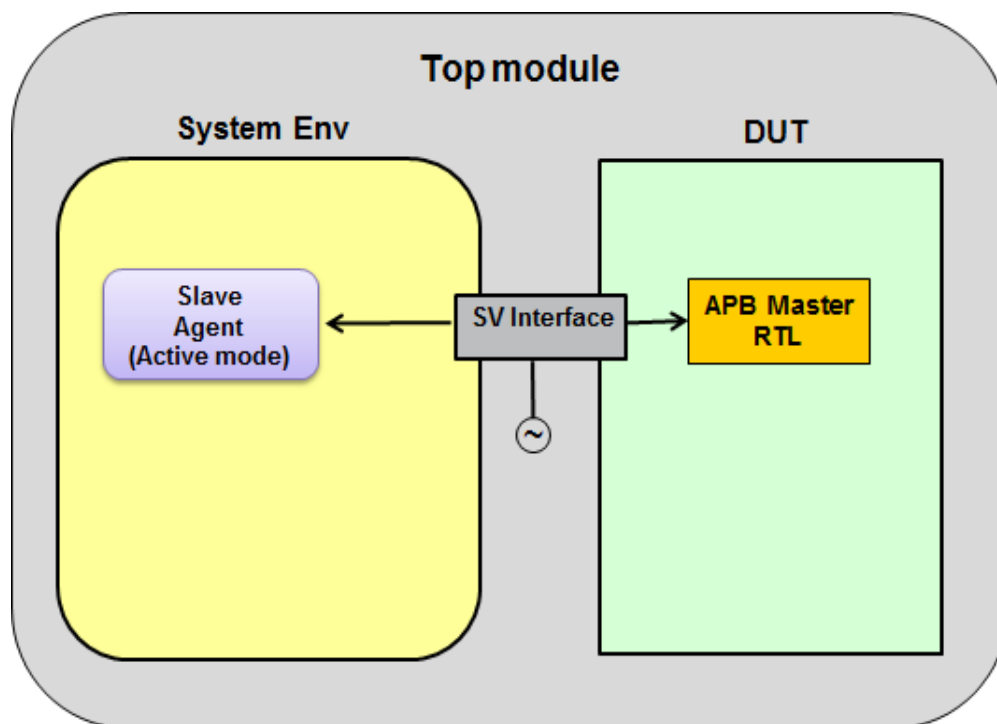
When DUT has a single ATB master port to be verified, testbench can either use a Slave Agent in standalone mode, or use a System Env configured for a single slave agent. The advantages and disadvantages of the two approaches are listed below.

Advantages of using standalone agent versus System Env:

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

Disadvantages:

- ❖ The testbench does not remain scalable. If the number of ATB master ports to be verified increases, the standalone Slave Agent should be replaced with System Env, or, multiple Slave Agents must be instantiated.

Figure 5-1 Master DUT and Slave VIP - Usage With Standalone Slave Agent**Figure 5-2 Master DUT and Slave VIP - Usage With System Environment**

5.2 Testing a Slave DUT Using a UVM Master VIP

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

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

The advantages and disadvantages of the two approaches are listed below. Advantages of using standalone agent versus System Env:

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

Disadvantages:

- ❖ The testbench does not remain scalable. If the number of ATB master ports to be verified increases, standalone Slave Agent should be replaced with System Env, or, multiple Slave Agents would need to be instantiated by you.

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

- ❖ System configuration settings:
 - ◆ `sys_cfg.num_masters = 1;`
 - ◆ `sys_cfg.num_slaves = 0;`
- ❖ Port configuration settings:
 - ◆ `sys_cfg.master_cfg[0].is_active = 1;`

Figure 5-3 Slave DUT and Master VIP - Usage With Standalone Slave Agent

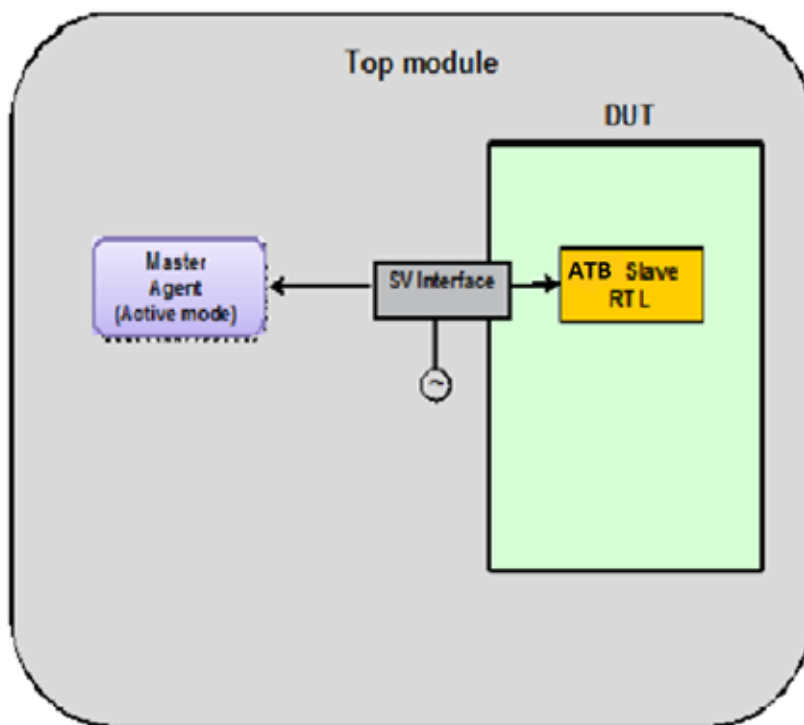
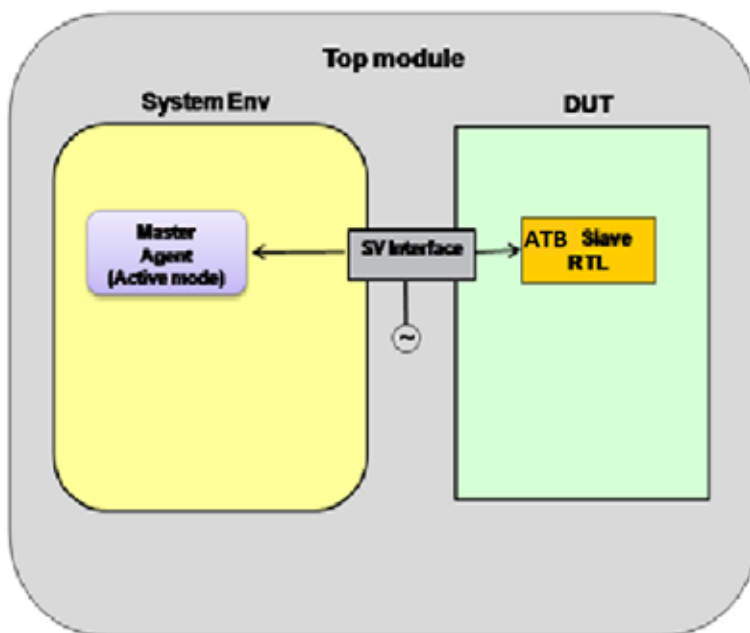


Figure 5-4 Slave DUT and Master VIP - Usage With System Environment



5.3 System DUT with Passive VIP

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

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

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

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

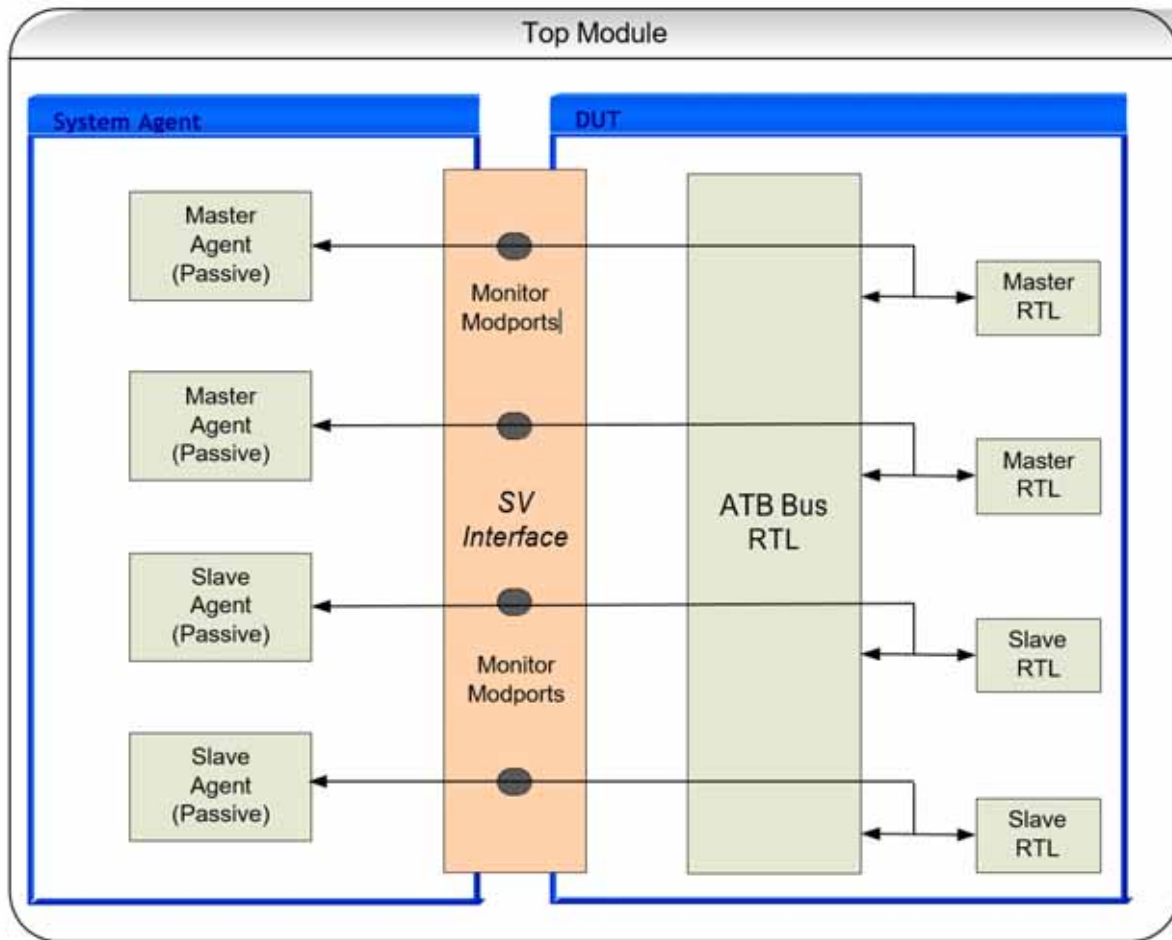
System configuration settings:

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

Port configuration settings:

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

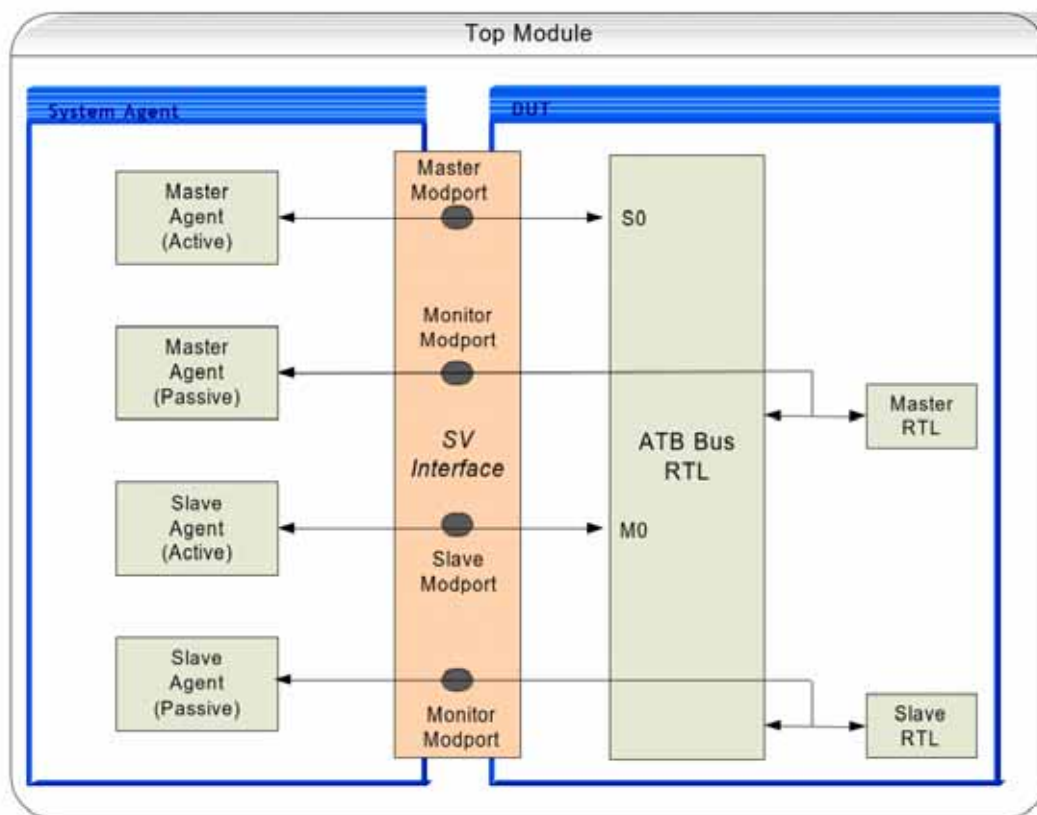
The System DUT with Passive VIP setup is shown in the following figure.

Figure 5-5 System DUT with Passive VIP

5.4 System DUT with Mix of Active and Passive VIP

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

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

Figure 5-6 System DUT with Mix of Active and Passive VIP

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

Assuming instance name of system configuration is "sys_cfg".

System configuration settings:

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

Port configuration settings:

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

6

Using ATB Verification IP

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

This chapter discusses the following topic:

- ❖ [“SystemVerilog UVM Example Testbenches”](#)
- ❖ [“Installing and Running the Examples”](#)
- ❖ [“How to Generate Slave Response”](#)
- ❖ [“How to Disable Objection Management by VIP and Allow Testbench to Manage Objections”](#)

6.1 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in [Table 6-1](#)

Table 6-1 SystemVerilog Example Summary

Example Name	Level	Description
tb_atb_svt_uvm_basic_sys	Basic	The example consists of the following: <ul style="list-style-type: none">• A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests• A base test• The base test creates a testbench environment, which in turn creates ATB System Environment• ATB System Environment is configured with one master and one slave agent
tb_atb_svt_uvm_intermediate_sys	Intermediate	Not yet supported
tb_atb_svt_uvm_advanced_sys	Advanced	Not yet supported

6.2 Installing and Running the Examples

The following steps are used for installing and running the example `tb_atb_svt_uvm_basic_sys`. 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_atb_svt_uvm_basic_sys -svtb
```

The example is installed in the following location:

```
<design_dir>/examples/sverilog/amba_svt/tb_atb_svt_uvm_basic_sys
```

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

a. Use the Makefile:

Three tests are provided in the *tests* directory.

The tests are:

```
✧ ts.base_test.sv
✧ ts.directed_test.sv
✧ ts.random_wr_rd_test.sv
```

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

```
gmake USE_SIMULATOR=vcsvlog directed_test WAVES=1
```

Invoke *gmake help* to show more options.

b. Use the sim script:

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

```
./run_atb_svt_uvm_basic_sys -w base_test vcsvlog
```

Invoke *./run_atb_svt_uvm_basic_sys -help* to show more options.

For more information on installing and running the example, see the README file in the example, located at:

```
$DESIGNWARE_HOME/vip/svt/amba_svt/latest/examples/sverilog/tb_atb_svt_uvm_basic_sys/
READ ME
```

Or

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

6.2.1 Support for UVM version 1.2

While using UVM 1.2, note the following 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.

6.3 How to Generate Slave Response

Slave responses are generated by the ATB Slave Agent component. The slave monitor contains a blocking peek port named `response_request_imp` which gets updated with any requests that target that slave. The slave agent connects this peek port to the slave sequencer which has a blocking peek port named `response_request_port`. Slave sequences must obtain a handle to the request through this port in the sequence, fill in the response information, and then submit this response to the driver through the normal means of sequencer/driver communication. The following code segment demonstrates a simple random response generator:

```

virtual task body();
  forever begin
    // Wait until a request is available p_sequencer.response_request_port.peek(req);

    // Return with an unconstrained random response
    `uvm_rand_send(req)
  end
endtask: body

```

The slave sequence can customize the response as needed before sending it to the driver. The only restriction is that the request handle received from the blocking peek port must be updated and sent to the driver in zero time.

The ATB VIP also provides a sequence which generates a memory response. The memory response sequence obtains a handle to the memory from the slave agent, and updates the memory with the transaction data for write transactions and updates the transaction with memory data on read transactions. For more information on slave sequence, see the `svt_atb_slave_base_sequence` in the file located at:

```

$DESIGNWARE_HOME/vip/svt/amba_svt/latest/atb_slave_agent_svt/sverilog/src/vcs/svt_atb_slave_sequence_collection.svp.

```

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

The objection management in ATB VIP components is controlled through a system configuration property `svt_atb_system_configuration::manage_objections_enable`. This parameter decides whether the objections will be raised and dropped by the drivers of VIP components. If set, the VIP will raise an objection when it receives a transaction in the input port of the driver and will drop the objection when the transaction completes. If not set, the driver will not raise any objection and complete control of objections is with the user. By default, the configuration parameter will be set, that is, VIP will raise and drop objections.

The following example code shows how you can potentially control objections from the testbench through callbacks:

```

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

  function new
    ( string name = "cust_svt_atb_system_interconnect_objection_control_callback" atb_env
      sys_env);
    super.new(name);
    this.sys_env= sys_env;
  endfunction

  virtual function void post_input_port_get(svt_atb_interconnect atb_interconnect,
    svt_atb_ic_slave_transaction xact);
    counter++;
    // Raise objections only for 1000 transactions. if (counter < 1000) begin
    env_run_ph.raise_objection(atb_env);
    fork begin wait(`SVT_ATB_XACT_STATUS_ENDED(xact)); env_run_ph.drop_objection(atb_env);
    end join_none
  endfunction
endclass

```


7

Troubleshooting

This chapter provides some useful information that can help you troubleshoot common issues that you encounter while using the ATB VIP.

7.1 Using Debug Port

Port interfaces `svt_atb_master_if` and `svt_atb_slave_if` of ATB VIP provide a modport `svt_atb_debug_modport` for debugging purpose. The signals in the debug modport represent the transaction number and ID value which are currently executing on all channels of the ATB port.

The signals, `atb_xact_num`, `atb_xfer_id` are driven by master driver in Master Agent and slave driver in Slave Agent.

A

Reporting Problems

A.1 Introduction

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

A.2 Debug Automation

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

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

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

A.3 Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named `+svt_debug_opts`. This 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 <code>start_time</code> . Two values are accepted in all methodologies: <code>DEBUG</code> and <code>VERBOSE</code> . UVM and OVM users can also supply the verbosity that is native to their respective methodologies (<code>UVM_HIGH/UVM_FULL</code> and <code>OVM_HIGH/OVM_FULL</code>). If this value is not supplied then the verbosity defaults to <code>DEBUG/UVM_HIGH/OVM_HIGH</code> . When this feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named <code>svt_debug.transcript</code> .

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  
$(VERDI_HOME)/share/PLI/MODELSIM/{LINUX|LINUX64}
```

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”](#) on page 55.

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 must be debugged.

