VC VIP AMBA APB UVM User Guide

Version W-2024.09, September 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

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 https://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 Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Feedback

Contents

About This Guide 8
Web Resources 8
Customer Support 8
Synopsys Statement on Inclusivity and Diversity 8

1. Introduction 10
Introduction 10
Prerequisites 11
References 11
Product Overview 11
Language and Methodology Support 11
Features Supported 12
Protocol Features 12
Verification Features 12
Methodology Features 13
Features Not Supported 13

2. Installation and Setup 14
Verifying the Hardware Requirements 14
Verifying the Software Requirements 14
Platform/OS and Simulator Software 15
Synopsys Common Licensing (SCL) Software 15
Other Third Party Software 15
Preparing for Installation 15
Downloading and Installing 16
Downloading From the Electronic Software Transfer (EST) System (Download Center) 16
Downloading Using FTP with a Web Browser 17
What's Next? 18
Licensing Information 18
License Polling 19

Simulation License Suspension 19
Environment Variable and Path Settings 19
Simulator-Specific Settings 20
Determining Your Model Version 20
Integrating the VIP into Your Testbench 21
Creating a Testbench Design Directory 21
Setting Up a New VIP 23
Installing and Setting Up More than One VIP Protocol Suite 24
Updating an Existing Model 25
Removing Synopsys VIP Models from a Design Directory 25
Reporting Information About DESIGNWARE_HOME or a Design Directory . . . 25
Running the Example with +incdir+ 25
Getting Help on Example Run/make Scripts 27
The dw_vip_setup Utility 28
Setting Environment Variables 28
The dw_vip_setup Command 28
Include and Import Model Files into Your Testbench 31
Compile and Run Time Options 32

3. General Concepts 34
Introduction to UVM 34
APB VIP in an UVM Environment 34
Master Agent 35
Slave Agent 35
System Env 36
System Sequencer 37
Active and Passive Mode 37
Functional Coverage 38
Default Coverage 38
Toggle Coverage 38
State Coverage 38
Delay Coverage 39
Transaction Coverage 39
Coverage Callback Classes 39
Coverage Data Callbacks 39
Coverage Callbacks 39
Enabling Default Coverage 40

Coverage Shaping and control 40
Reset Functionality 40

4. APB VIP Programming Interface 41
Configuration Objects 41
Transaction Objects 42
Analysis Ports 44
Callbacks 44
Callbacks in the Master Agent 45
Callbacks in Slave Agent 45
Interfaces and modports 45
Modports 46
Events 46
Overriding System Constants 47
Protocol Analyzer Support 48
Support for VC Auto Testbench 48
Support for Native Dumping of FSDB 49
Verification Features 50

5. Verification Topologies 51
Master DUT and Slave VIP 51
Slave DUT and Master VIP 53

6. Using APB Verification IP 57
SystemVerilog UVM Example Testbenches 57
Installing and Running the Examples 58
Defines for Increasing Number of Masters and Slaves 59
Support for UVM version 1.2 59
Steps to Integrate the UVM_REG With APB VIP 59
Master to Slave Path Access Coverage 60
Support for Unaligned Addresses 62
Use Model 62
Configuration Attributes 63
VIP Components 63

Active Manager 63
Passive Manager 64
Active and Passive Subordinates 64
Protocol Checks 65
Functional Coverage 65
Customized Naming of Agent Instances 66
Configuration Attributes 66
Use Model 66
VIP Components 67
APB SolvNetPlus Articles 67

7. Using APB-D and APB-E VIP Features 69
Overview 69
Supported Features 69
Unsupported Features and Limitations 70
Licensing and Keys 70
Use Model to Enable APB-D and APB-E Features 70
Macro Definition 70
Configuration Attribute 70
User Signaling 71
Macro Definition 71
Configuration Attribute 71
Transaction Attributes 72
VIP Behavior 73
Protocol Checks 73
Parity Signaling 73
Macro Definition 73
Configuration Attribute 74
Transaction Attributes 74
VIP Behavior 76
Protocol Checks 78
RME Support 79
Macro Definition 79
Configuration Attribute 80
Transaction Attribute 80
VIP Behavior 81

Protocol Checks . . . 81
Subsystem ID Support . . . 81
Macro Definition . . . 81
Configuration Attribute . . . 82
Transaction Attribute . . . 82
VIP Behavior . . . 82
Protocol Checks . . . 82
Wakeup Signaling . . . 83
Macro Definition . . . 83
Configuration Attribute . . . 83
Transaction Attribute . . . 84
VIP Behavior . . . 85
Protocol Checks . . . 85
VIP Examples for APB-D, APB-E Features . . . 85

8. Reporting Problems . . . 87
Introduction . . . 87
Debug Automation . . . 87
Enabling and Specifying Debug Automation Features . . . 88
Debug Automation Outputs . . . 90
FSDB File Generation . . . 90
VCS . . . 91
Questa . . . 91
Incisive . . . 91
Initial Customer Information . . . 91
Sending Debug Information to Synopsys . . . 92
Limitations . . . 93

Preface

About This Guide

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

Web Resources

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

Customer Support

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

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

   Enter the information according to your environment and your issue.

2. Send an e-mail message tosupport_center@synopsys.com.

   Include the Product name, Sub Product name, and Tool Version in your e-mail so it can be routed correctly.

3. Telephone your local support center.
   - North America:

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

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

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary

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

1

Introduction

This chapter gives a basic introduction, overview and features of the APB UVM Verification IP.

This chapter discusses the following topics:

- Introduction
- Prerequisites
- References
- Product Overview
- Language and Methodology Support
- Features Supported
- Features Not Supported

Note:

Based on the AMBA Progressive Terminology updates, you must interpret the term Master as Manager and Slave as Subordinate in the VIP documentation and messages.

Introduction

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

This approach leverages advanced verification technologies and tools that provide:

- Protocol functionality and abstraction
- Constrained random verification
- Functional coverage
- Rapid creation of complex tests

- Modular testbench architecture that provides maximum reuse, scalability and modularity
- Proven verification approach and methodology
- Transaction-level models
- Self-checking tests
- Object oriented interface that allows OOP techniques

Prerequisites

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

References

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

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

Product Overview

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

Language and Methodology Support

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

- Languages
  - SystemVerilog
- Methodology
  - Qualified with UVM 1.1d and UVM 1.2

Features Supported

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

Protocol Features

APB VIP currently supports the following protocol functions:

- APB2 Features
  - APB Master initiates transfers on the Peripheral Bus
  - APB Master supports Write, Read, and Idle transactions
  - APB Master supports maximum of 16 slave devices
  - APB Slave memory response modeled by sequences
- APB3 features
  - APB Slave supports wait states using PREADY signal
  - APB Slave supports error response using PSLVERR signal
- APB4 features
  - APB Master supports write strobe using PSTRB signal
  - APB Master supports PPROT signal

Verification Features

APB VIP currently supports the following verification functions:

- Default functional coverage (transaction, state and toggle coverage)
- Basic Protocol checking
- Control on delays and timeouts
- Support for Protocol Analyzer
- VC Auto Testbench

Feedback

Methodology Features

APB VIP currently supports the following methodology functions:

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

Features Not Supported

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

AMBA SVT VIP Release Notes are present at:

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

2

Installation and Setup

This chapter leads you through installing and setting up the AMBA APB UVM VIP. When you complete this checklist, the provided example testbench will be operational and the APB 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. What's Next?

Note:

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

Verifying the Hardware Requirements

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

- 1440 MB available disk space for installation
- 16 GB Virtual Memory recommended

Verifying the Software Requirements

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

Feedback

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 APB VIP, check the support matrix manual.

Synopsys Common Licensing (SCL) Software

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

Other Third Party Software

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

Preparing for Installation

1. Set DESIGNWARE_HOME to the absolute path where APB VIP is to be installed:

```
setenv DESIGNWARE_HOME absolute_path_to_designware_home
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>
     ```

Feedback

- `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>
  ```

Downloading and Installing

Important:

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 SolvNetPlus password is unknown or forgotten, go tohttp://solvnetplus.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

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

1. Point your web browser to *http://solvnetplus.synopsys.com*.
2. Enter your Synopsys SolvNetPlus Username and Password.
3. Click Sign In button.
4. Make the following selections on SolvNetPlus to download the `.run` file of the VIP (See Figure 1).

Downloads tab

VC VIP Library product releases

<release_version>

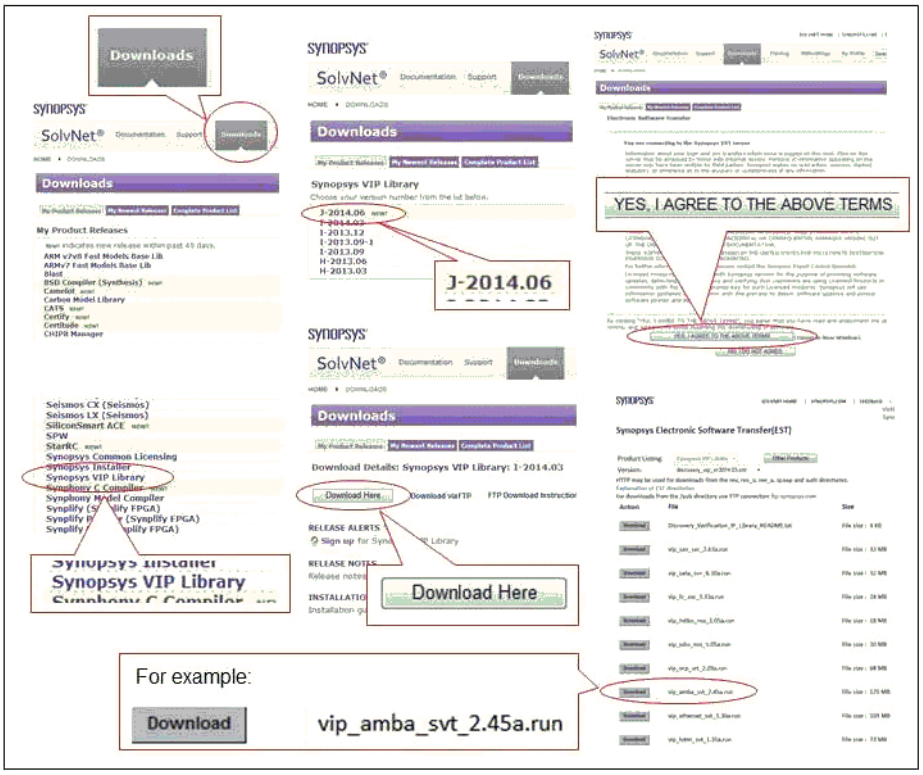
Download Here button

Yes, I Agree to the Above Terms button

Download `.run` file for the VIP

Feedback

Figure 1 SolvNetPlus Selections for VIP Download



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

   ```
   % setenv DESIGNWARE_HOME VIP_installation_path
   ```

2. 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.

Downloading Using FTP with a Web Browser

1. Follow the above instructions through the product version selection step.
2. Click the "Download via FTP" link instead of the "Download Here" button.

3. Click the "Click Here To Download" button.
4. Select the file(s) that you want to download.
5. Follow browser prompts to select a destination location.

Note:

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

What's Next?

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

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

Licensing Information

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

You can find general SCL information in the following location:

http://www.synopsys.com/keys

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

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

Feedback

License Polling

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

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

Simulation License Suspension

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

Note:

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

Environment Variable and Path Settings

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

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

  Note:

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

Feedback

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

  Note:

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

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

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

Simulator-Specific Settings

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

Determining Your Model Version

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

Note:

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

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

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

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

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

Feedback

Integrating the VIP into Your Testbench

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

- Creating a Testbench Design Directory
- Setting Up a New VIP
- Installing and Setting Up More than One VIP Protocol Suite
- Updating an Existing Model
- Removing Synopsys VIP Models from a Design Directory
- The dw_vip_setup Utility

Creating a Testbench Design Directory

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

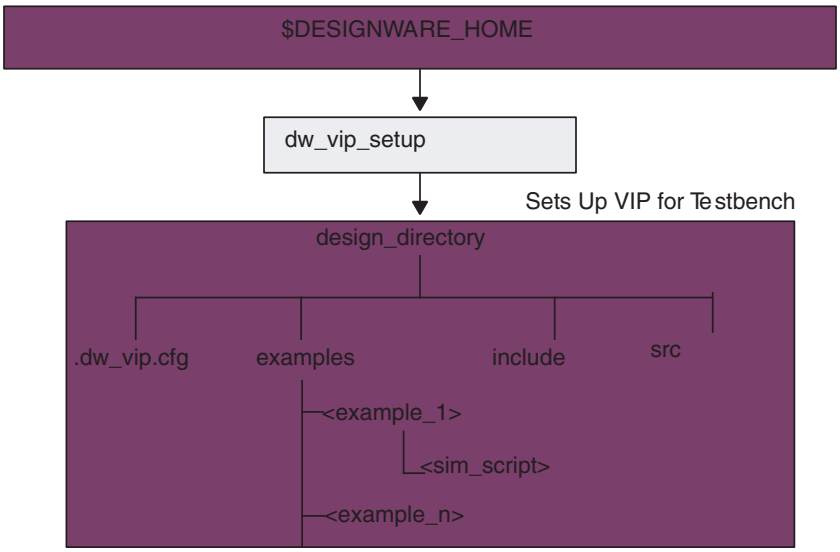
Note:

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. When you want, you can use dw_vip_setup to update the VIP in your design directory. Figure 2 shows this process and the contents of a design directory.

Feedback

Figure 2 Design Directory Created by dw_vip_setup



A design directory contains:

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

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

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

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

Note:

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

Note:

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

Feedback

Setting Up a New VIP

After you have installed the VIP, you must set up the VIP for project and testbench use. All VIP suites contain various components such as transceivers, masters, slaves, and monitors depending on the protocol. The setup process gathers together all the required component files you need to incorporate into your testbench required for simulation runs.

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

- apb_master_agent_svt
- apb_slave_agent_svt
- apb_system_env_svt

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

To get help on dw_vip_setup, invoke the following:

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

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

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

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

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

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

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

Feedback

Important:

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

Installing and Setting Up More than One VIP Protocol Suite

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

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

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

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

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

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

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

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

To specify other model names, consult the VIP documentation.

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

Feedback

Updating an Existing Model

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

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

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

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

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

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.

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
```

Running the Example with +incdir+

In the current setup, you install the VIP under DESIGNWARE_HOME followed by creation of a design

Feedback

directory which contains the versioned VIP files. With every newer version of the already installed VIP

requires the design directory to be updated. This results in:

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

The new alternative approach of directly pulling in all the files from `DESIGNWARE_HOME` eliminates the

need for design directory creation. VIP version control is now in the command line invocation.

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

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

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

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

Note:

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

Feedback

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_apb_svt_uvm_basic_sys --help

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

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

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

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

   -incdir use DESIGNWARE_HOME include files instead of design directory

   -verbose enable verbose mode during compilation

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

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

   -seed run simulation with specified seed value

   -clean clean simulator generated files

   -nobuild skip simulator compilation

   -buildonly exit after simulator build

   -norun only echo commands (do not execute)

   -pa invoke Verdi after execution

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

   Usage: gmake

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

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

Feedback

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.

The dw_vip_setup Utility

The dw_vip_setup utility:

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

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

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.

Feedback

switch The *switch* argument defines dw_vip_setup operation. This table lists the switches and their applicable sub-switches.

Table 1 Setup Program Switch Descriptions

Switch	Description
-a[dd] (*model* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or current working directory. If you do not specify a version, the latest version is assumed. The model names are:apb_master_agent_svtapb_slave_agent_svtapb_system_env_svtThe -add switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-r[emove] *model*	Removes **all versions** of the specified model or models from the design. The dw_vip_setup program does not attempt to remove any include files used solely by the specified model or models. The model names are:apb_master_agent_svtapb_slave_agent_svtapb_system_env_svt
-u[pdate] (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The dw_vip_setup script updates to the latest models when you do not specify a version. The model names are:apb_master_agent_svtapb_slave_agent_svtapb_system_env_svtThe -update switch causes dw_vip_setup to build suite libraries from the same suite as the specified models, and to copy the other necessary files from $DESIGNWARE_HOME.
-e[xample] {*scenario* | *model*/*scenario*} [**-v**[ersion] *version*]	The dw_vip_setup script configures a testbench example for a single model or a system testbench for a group of models. The program creates a simulato run program for all supported simulators.If you specify a *scenario* (or system) example testbench, the models needed for the testbench are included automatically and do not need to be specified in the command. **Note:**Use the -info switch to list all available system examples.
-ntb	Not supported.
-svtb	Use this switch to set up models and example testbenches for SystemVerilog UVM. The resulting design directory is streamlined and can only be used in SystemVerilog simulations.
-c[lean] {*scenario* | *model*/*scenario*}	Cleans the specified scenario/testbench in either the design directory (as specified by the *-path* switch) or the current working directory. This switch deletes *all files in the specified directory*, then restores all Synopsys created files to their original contents.

Feedback

Table 1 Setup Program Switch Descriptions (Continued)

Switch	Description
-i/nfo design | home[:<product>[:<version>[:<methodology>]]]	Generate an informational report on a design directory or VIP installation.*design*: If the '*-info design'* switch is specified, the tool displays productand 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 tocreate another design directory with the same content.*home*: If the '*-info home*' switch is specified, the tool displays product,version, and exampl content within the VIP installation to standard output.Optional filter fields can also be specified such as <product>, <version>, and<methodology> delimite by colons (:). An error will be reported if anonexistent or invalid filter field is specified. Valid methodology namesinclude: OVM, RVM, UVM, VMM and VLOG.
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each
model	Synopsys APB VIP models are: apb_master_agent_svtapb_slave_agent_svtapb_system_env_svtThe *model* argument defines the model or models that dw_vip_setup acts upon. This argument is not needed with the -info or -help switches. All switches that require the *model* argument may also use a model list.You may specify a version for each listed *model*, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores *model* version information.
-b/ridge	Updates the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.
-pa	Enables the run scripts and Makefiles generated by dw_vip_setup to support PA. If this switch is enabled, and the testbench example produces FSDB files PA will be launched and the FSDB files will be read at the end of the example execution.For run scripts, specify `-pa`.For Makefiles, specify `-pa = 1`.
-waves	Enables the run scripts and Makefiles generated by dw_vip_setup to support the `fsdb` waves option . To support this capability, the testbench example must generate an FSDB file when compiled with the WAVES Verilog macro set to `fsdb`, that is, `+define+WAVES=\"fsdb\"`. If a .fsdb file is generated by the example, the Verdi nWave viewer will be launched.For run scripts, specif `-waves fsdb`.For Makefiles, specify `WAVES=fsdb`.
-doc	Creates a doc directory in the specified design directory which is populatedwith symbolic links to the *DESIGNWARE_HOME* installation for documentsrelated to the given model or example being added or updated.
-methodology <name>	When specified with -doc, only documents associated with the specifiedmethodology name are added to the design directory. Valid methodologynames include: OVM, RVM, UVM, VMM, and VLOG.
-copy	When specified with -doc, documents are copied into the design directory, notlinked.

Feedback

Table 1 Setup Program Switch Descriptions (Continued)

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

Note:

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

Include and Import Model Files into Your Testbench

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

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

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

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

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

/** Import UVM Package  */
import uvm_pkg::*;
```

Feedback

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

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

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

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

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

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

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

Compile and Run Time Options

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

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

The files containing the options are:

- sim_build_options (contain compile time options common for all simulators)
- sim_run_options (contain run time options common for all simulators)
- vcs_build_options (contain compile time options for VCS)
- vcs_run_options (contain run time options for VCS)
- mti_build_options (contain compile time options for MTI)
- mti_run_options (contain run time options for MTI)
- ncv_build_options (contain compile time options for IUS)
- ncv_run_options (contain run time options for IUS)

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

vcs_build_options

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

Note:

AMBA SVT VIP implementation does not depend on the macro `UVM_PACKER_MAX_BYTES`. However, if UVM pack or unpack operation needs to be performed on the transaction handle in your testbench, then `UVM_PACKER_MAX_BYTES` macro needs to be defined and set to an optimal value in your testbench. For example, if VIP title 1 needs `UVM_PACKER_MAX_BYTES` to be set to 8192, and VIP title 2 needs `UVM_PACKER_MAX_BYTES` to be set to 500000, you need to set `UVM_PACKER_MAX_BYTES` to 500000.

vcs_run_options

Required: +UVM_TESTNAME=$scenario

Note:

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

3

General Concepts

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

- Introduction to UVM
- APB VIP in an UVM Environment
- Functional Coverage
- Reset Functionality

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

This chapter assumes that you are familiar with SystemVerilog and 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/.

APB VIP in an UVM Environment

The following sections describe how the APB Verification IP is structured in an UVM testbench.

Feedback

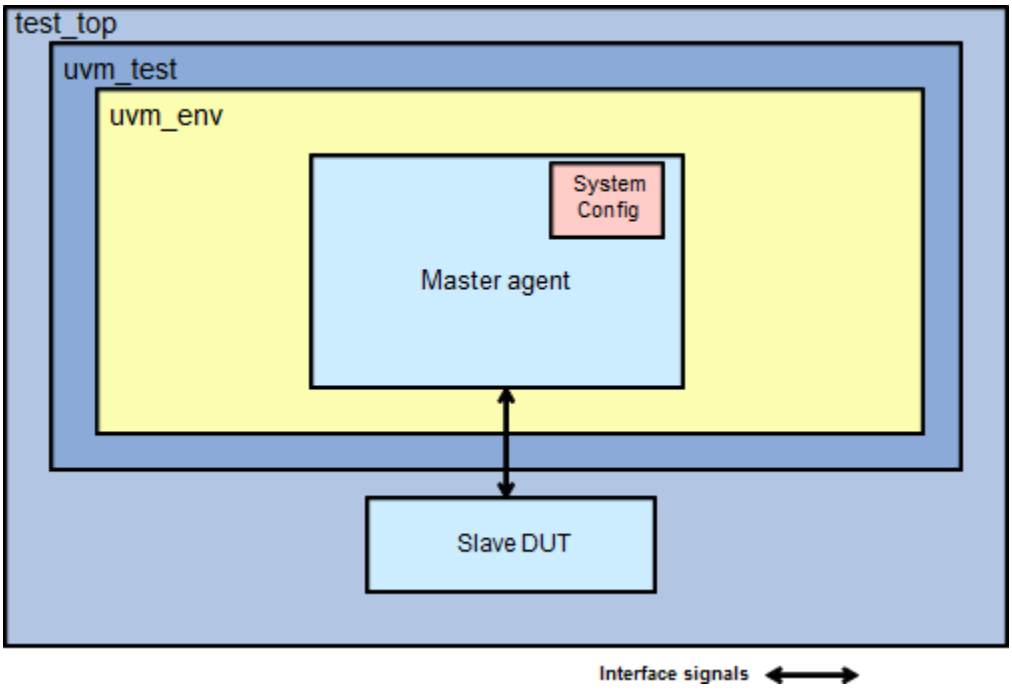
Master Agent

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

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

Within the master agent, the master driver gets sequences from the master sequencer. The master driver then drives the APB transactions on the APB port. The master driver and system monitor components within master agent call callback methods at various phases of execution of the APB transaction. Details of callbacks are covered in later sections. After the APB transaction on the bus is complete, the completed sequence item is provided to the analysis port of system monitor, which can be used by the testbench.

Figure 3 Usage With Standalone Master Agent



Slave Agent

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

Feedback

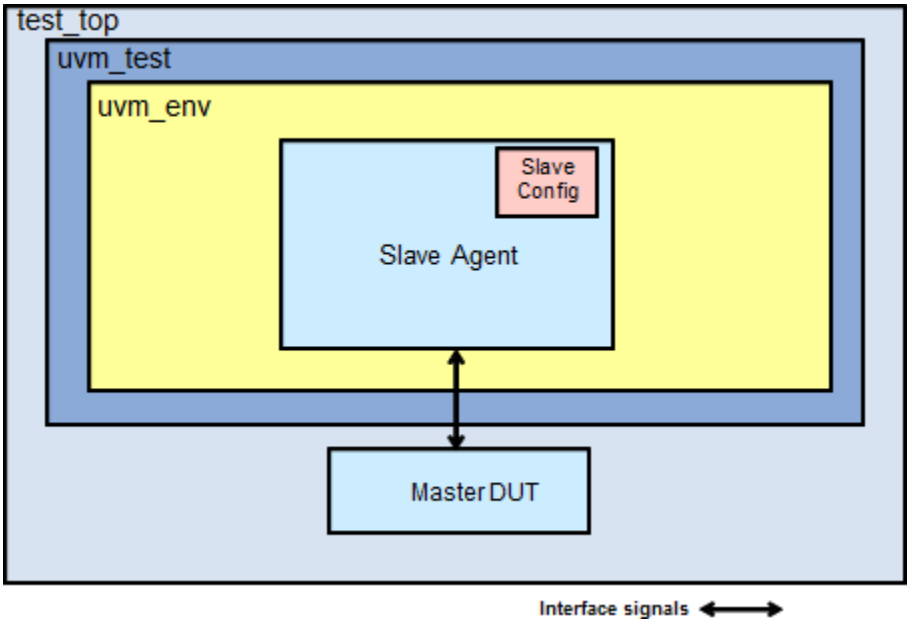
The slave agent is configured using slave configuration, which is available in the system configuration. The slave configuration should be provided to the slave agent in the build phase of the test.

In the slave agent, the slave monitor samples the APB port signals. When a new transaction is detected, slave monitor provides a response request sequence to the slave sequencer.

The slave response sequence within the sequencer programs the appropriate slave response. The updated response sequence is then provided by the slave sequencer to the slave driver. The slave driver in turn drives the response on the APB bus.

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

Figure 4 Usage With Standalone Slave Agent



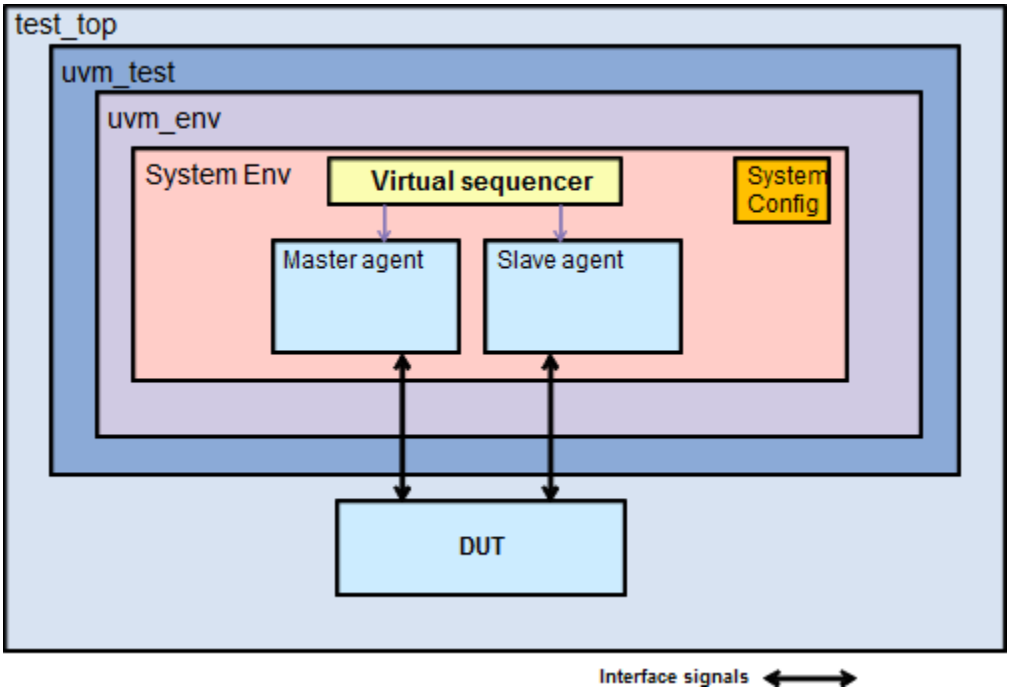
System Env

The APB System Env encapsulates the master agent, slave agents, system sequencer and the system configuration. The number of slave agents are configured based on the system configuration provided by the user. In the build phase, the system Env builds the master and slave agents. After the master & slave agents are built, the system Env

Feedback

configures the master & slave agents using the configuration information available in the system configuration.

Figure 5 System ENV



System Sequencer

APB System sequencer is a virtual sequencer with references to the master sequencer and each of the slave sequencers in the system. The system sequencer is created in the build phase of the system Env. The system configuration is provided to the system sequencer. The system sequencer can be used to synchronize between the sequencers in master & slave agents.

Active and Passive Mode

This table lists the behavior of Master and Slave Agents in active and passive modes.

Table 2 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.

Feedback

Table 2 Agents in Active and Passive Mode (Continued)

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

Functional Coverage

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

Default Coverage

The following sections describes the default coverage provided with APB VIP. For more details on actual cover groups, refer to the APB VIP class reference HTML document.

Toggle Coverage

Toggle coverage is a signal level coverage. Toggle coverage provides baseline information that a system is connected properly, and that higher level coverage or compliance failures are not simply the result of connectivity issues. Toggle coverage answers the question: Did a bit change from a value of 0 to 1 and back from 1 to 0? This type of coverage does not indicate that every value of a multi-bit vector was seen but measures that all the individual bits of a multi-bit vector did toggle.

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, different values of PSTRB, PPROT, PADDR etc., are covered under the state coverage. If the state space is too large, an intelligent classification of the states must be made. In the case of the PADDR signal for example, coverage bins would be one bin to cover the lower address range, one bin to cover the upper address range and one bin to cover all other intermediary addresses.

Delay Coverage

Delay coverage is coverage on delays related with PREADY and PENABLE signals. The following delays are covered:

- PENABLE delay (Master idle cycles)
- PREADY delay (Slave wait cycles)

Transaction Coverage

Transaction coverage covers APB transactions types and Cross coverage across APB signals. The cross coverage involves cross between type of transaction, slave id and operating states.

Coverage Callback Classes

Coverage Data Callbacks

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

The coverage data callback class is extended from callback class svt_apb_port_monitor_callback. The extended class is called svt_apb_port_monitor_def_cov_data_callback.

Below callback methods are implemented for triggering coverage events:

- pre_output_port_put

Coverage Callbacks

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

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

Enabling Default Coverage

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

- toggle_coverage_enable
- state_coverage_enable
- transaction_coverage_enable

By default, the coverage is disabled.

Coverage Shaping and control

The handle to the port configuration class svt_apb_port_configuration is provided to the class svt_apb_port_monitor_def_cov_callback, which implements the default cover groups. Based on the port configuration, the coverage bins are shaped. The unwanted bins are ignored.

In addition to above, user also has the ability to disable coverage at cover group level. Class svt_apb_port_configuration provides members svt_apb_port_configuration::<cover_group_name>_enable, to enable/disable cover groups. By default, the value to these members is 1.

Reset Functionality

The APB VIP samples the reset assertion asynchronously whereas reset de-assertion will be sampled synchronously. This means that, when reset is asserted, it is not required that the clock connected to VIP is running but for de-assertion of reset, the clock should be running. If the clock input to VIP is not running, de-assertion of reset is not detected, and VIP would not sample and drive any signals.

When reset is asserted the current transaction which is in progress is ABORTED. The `curr_state`field of this transaction reflect the value as `ABORT_STATE`. The transaction ENDED notification is issued on rising edge of clock when reset signal assertion is observed.

4

APB VIP Programming Interface

This chapter presents the programming or user interface for the functionality of the APB Verification IP. This chapter discusses the following topics:

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

Configuration Objects

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

The configuration object properties can be of two types:

- Static configuration properties:

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

- Dynamic configuration properties:

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

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

Feedback

The APB VIP defines the following configuration classes:

- System configuration (svt_apb_system_configuration)

  System configuration class contains configuration information which is applicable across the entire system.

  User can specify the system level configuration parameters through this class. User needs to provide the system configuration to the system Env from the environment or the testcase. The system configuration mainly specifies:

  - Number of slave agents in the system Env
  - Sub-configurations for master and slave agents
  - Virtual top level APB interface
  - Address map
  - Timeout values

- Slave configuration (svt_apb_slave_configuration)

  The slave configuration class contains configuration information which is applicable to the APB slave agent in the system Env. Some of the important information provided by the slave configuration class is:

  - Active/Passive mode of the slave port agent
  - Enable/disable protocol checks
  - Enable/disable port level coverage

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

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

Transaction Objects

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

APB transaction data objects store data content and protocol execution information for APB 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.

APB transaction data objects are used to:

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

Class properties are public and accessed directly to set and read values. Transaction data objects support randomization and provide built-in constraints.

Two set of constraints are provided - valid_ranges and reasonable constraints.

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

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

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

APB VIP defines following transaction classes:

- APB Transaction (svt_apb_transaction)

  This is the transaction class which contains all the physical attributes of the transaction like address, data, direction, etc. It also provides control over idle length and transaction delays.

The transaction contains a handle to the configuration object, which provides the configuration of the port on which this transaction would be applied. The configuration is used during randomizing the transaction. The configuration is available in the sequencer

of the master/slave agent. The user sequence should initialize the configuration handle in the transaction using the configuration available in the sequencer of the master/slave agent. If the configuration handle in the transaction is null at the time of randomization, the transaction will issue a fatal message.

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

Analysis Ports

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

Callbacks

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

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

- There is no need to invoke callback methods for callbacks that are not extended. To avoid a loss of performance, callbacks are not executed by default. To execute callback methods, callback class must be registered with the component using `uvm_register_cb macro.

APB VIP uses callbacks in three main applications:

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

Callbacks in the Master Agent

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

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

- svt_apb_master_callback
- svt_apb_master_monitor_callback

Please refer to class reference HTML documentation for details of these classes.

Callbacks in Slave Agent

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

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

- svt_apb_slave_callback
- svt_apb_slave_monitor_callback

Please refer to class reference HTML documentation for details of these classes.

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.

Feedback

APB VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top level interface svt_apb_if is defined. The top level interface contains an array of slave sub-interfaces of type svt_apb_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_apb_system_configuration::set_if. This is also the interface that is used by the master agent.

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

Modports

The port interface svt_apb_if contains following modports which users should use to connect VIP to the DUT:

- svt_apb_master_modport
- svt_apb_slave_modport
- svt_apb_debug_modport

Events

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

- For WRITE transactions, STARTED notification is issued on the rising clock edge when psel and pwrite are both high.
- For READ transactions, STARTED notification is issued on the rising clock edge when psel is high and pwrite is low.
- For WRITE transactions, the ENDED notification is issued on the rising clock edge after a falling edge of penable.
- For READ transactions, the ENDED notification is issued on the rising clock edge after a falling edge of penable.

Feedback

Overriding System Constants

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

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

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

To override the SVT_APB_PADDR_WIDTH constant from the svt_apb_port_defines.svi file:

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

  ```
  `define SVT_APB_PADDR_WIDTH 12
  ```

- In the simulator compile command:
  - Ensure that the directory containing svt_apb_user_defines.svi is provided to the simulator
  - Provide SVT_APB_INCLUDE_USER_DEFINES on the simulator command line as follows:

    ```
    +define+SVT_APB_INCLUDE_USER_DEFINES
    ```

Feedback

Note the following restrictions when overriding the default maximum footprint:

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

  Note:

  A value of less than 32 is not supported for SVT_APB_MAX_ADDR_WIDTH. SVT_APB_MAX_ADDR_WIDTH only defines the footprint of address port. The actual used address with is defined by svt_apb_port_configuration::addr_width, which can still be configured to less than 32.

Protocol Analyzer Support

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

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

To enable protocol file generation, set the value of the variable "enable_xml_gen" to '1' in the port configuration of each master or slave for which protocol file generation is desired.

The next time that the environment is simulated, protocol files will be generated according to the port configurations. The protocol files will be in .xml format. Import these files into the Protocol Analyzer to view the protocol transactions.

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

Note:

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

Support for VC Auto Testbench

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

For VC ATB documentation, see Verdi_Transaction_and_Protocol_Debug.pdf.

Feedback

Support for Native Dumping of FSDB

Native FSDB supported in APB VIP.

- *FSBD Generation*: Protocol Analyzer uses transaction-level dump database. You can use the following settings to dump the transaction database:
  - Compile Time Options

-lca -kdb // dumps the work.lib++ data for source coding view

+define+SVT_FSDB_ENABLE // enables FSDB dumping

-debug_access

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

- New configuration parameter pa_format_type is added for FSDB generation in svt_apb_configuration.sv. Add the following setting in system configuration to enable the generation of FSDB:

```
master_cfg.xml_gen_enable = 1;
master_cfg.pa_format_type  = svt_xml_writer:: FSDB;
master_cfg.slave_cfg[0].xml_gen_enable = 1;
master_cfg.slave_cfg[0].pa_format_type = svt_xml_writer::FSDB;
```

- *Invoking Protocol Analyzer*: Perform the following steps to invoke Protocol Analyzer in interactive or post-processing mode:
  - Post-processing Mode

Load the transaction dump data and issue the following command to invoke the GUI:
`verdi -ssf <dump.fsdb> -lib work.lib++`

In Verdi, navigate to Tools > Transaction Debug >Transaction and Protocol Analyzer.

- Interactive Mode

Issue the following command to invoke Protocol Analyzer in an interactive mode: `<simv> -gui=verdi`

Runtime Switch:

`+svt_enable_pa=fsdb`

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

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

Verification Features

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

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

5

Verification Topologies

This chapter shows you how to connect various types of DUTs to the APB Verification IP. This chapter discusses the following topics:

- Master DUT and Slave VIP
- Slave DUT and Master VIP

Master DUT and Slave VIP

Scenario: DUT is APB Master. VIP is required to verify the APB Master DUT.

Testbench setup: Configure the APB System configuration to have 1 slave agent, in active mode. The active slave agent will respond to the transactions generated by master DUT. The slave agent will also do passive functions like protocol checking, coverage generation, 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_slaves = 1;
```

- Master configuration settings:

```
sys_cfg.is_active = 0;
```

- Slave configuration settings:

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

When the DUT is an APB master port to be verified, the testbench can either use a slave agent in standalone mode, or use a system Env configured for a single slave agent. Below are the pros and cons of the two approaches.

Advantages of using standalone agent versus system Env:

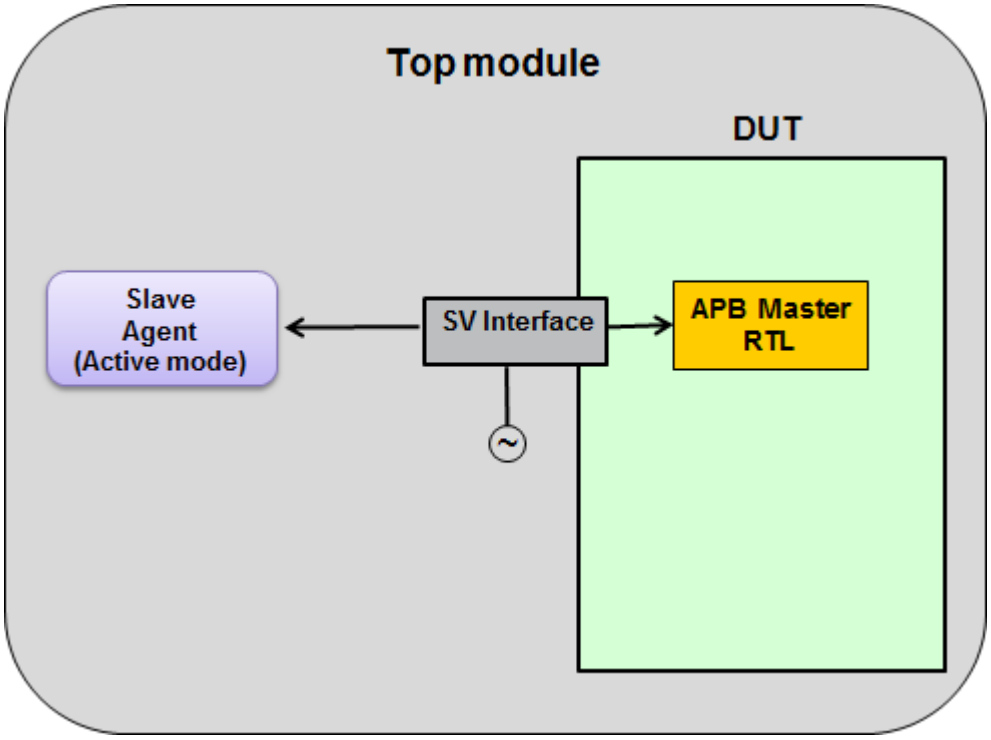
- Testbench becomes light weight as system Env and related infrastructure is not required

Feedback

Disadvantages:

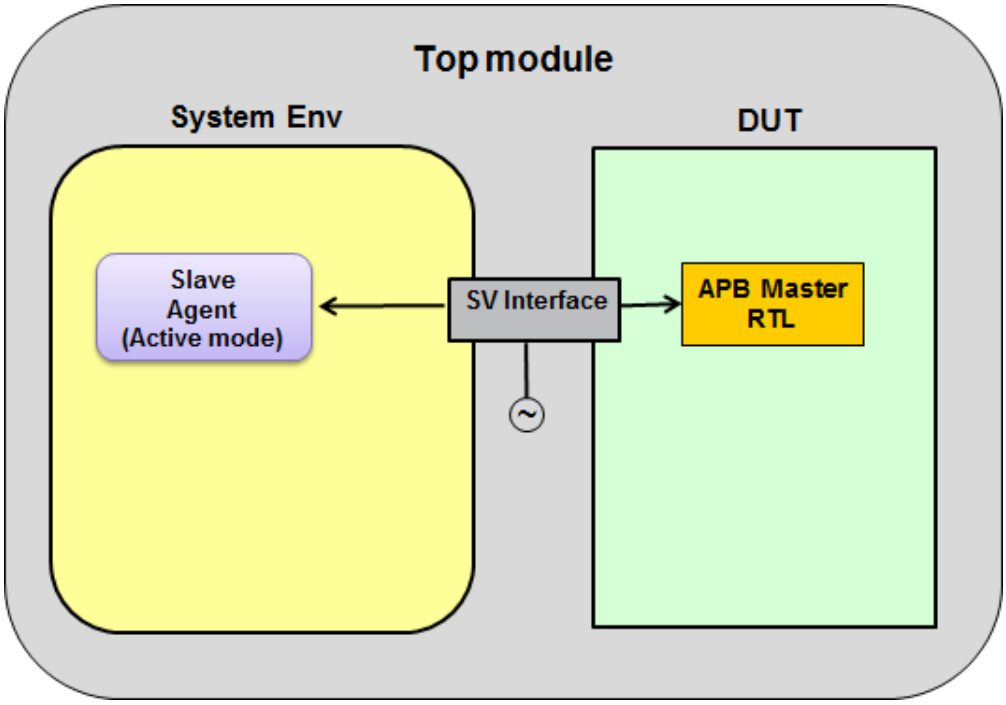
- The testbench does not remain scalable. If number of APB slave ports to be verified increased, standalone slave agent would need to be replaced with system Env, or, multiple slave agents would need to be instantiated by the user.

Figure 6 Master DUT and Slave VIP - Usage With Standalone Slave Agent



Feedback

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



Slave DUT and Master VIP

Scenario: DUT is APB Slave. VIP is required to verify the APB Slave DUT.

Testbench setup: Configure the APB System configuration to put the master agent in active mode. The active master agent will generate APB transactions for the Slave DUT. The master agent will also do passive functions like protocol checking, coverage generation, 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_slaves = 0;
```

- Master configuration settings:

```
sys_cfg.is_active = 1;
```

Feedback

When the DUT has a single APB slave port to be verified, testbench can either use a master agent in standalone mode, or use a system Env. Below are the pros and cons of the two approaches.

Advantages of using standalone agent versus system Env:

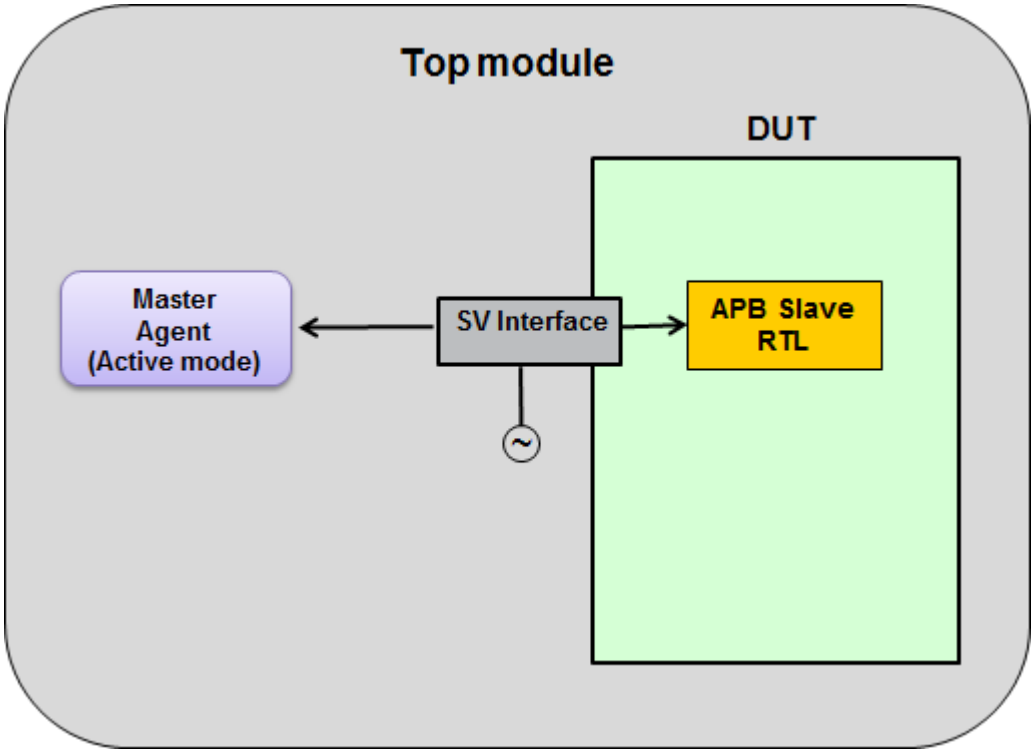
- Testbench becomes light weight as system Env and related infrastructure is not required

Feedback

Disadvantages:

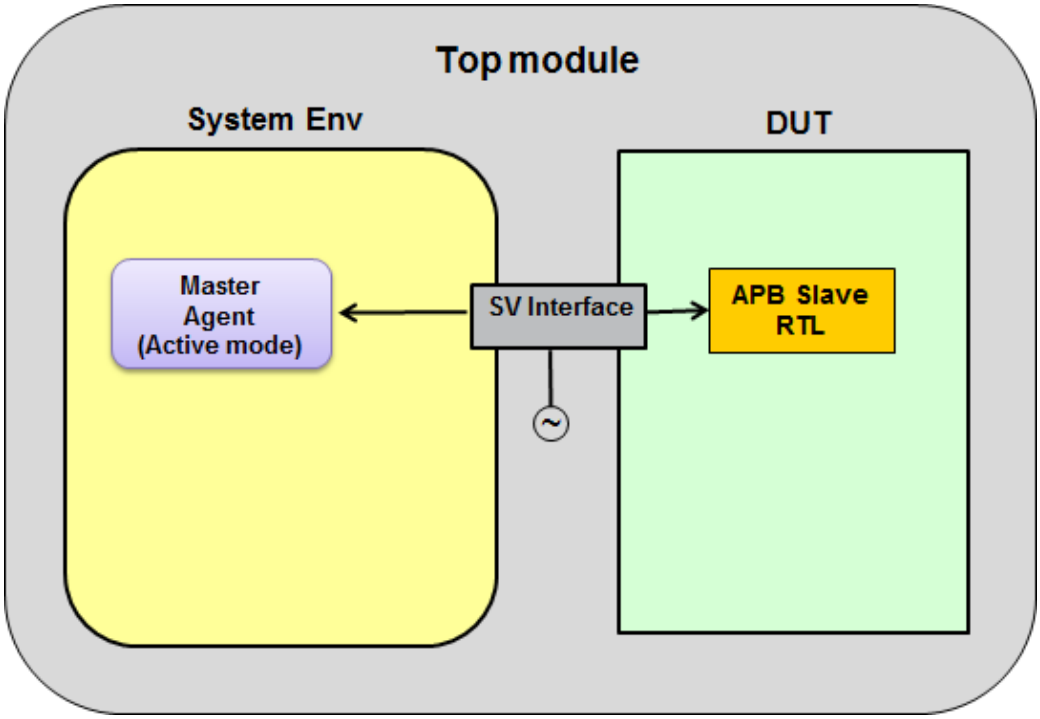
- The testbench does not remain scalable. If number of APB slave ports to be verified increased, standalone slave agent would need to be replaced with system Env, or, multiple slave agents would need to be instantiated by the user.

Figure 8 Slave DUT and Master VIP - Usage With Standalone Master Agent



Feedback

Figure 9 Slave DUT and Master VIP - Usage With System Environment



6

Using APB Verification IP

This chapter shows how to install and run a getting started example. This chapter discusses the following topics:

- SystemVerilog UVM Example Testbenches
- Installing and Running the Examples
- Steps to Integrate the UVM_REG With APB VIP
- Master to Slave Path Access Coverage
- Support for Unaligned Addresses
- Customized Naming of Agent Instances

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 this table.

Table 3 SystemVerilog Example Summary

Example Name	Level	Description
tb_apb_svt_uvm_basic_sys	Basic	The example consists of the following:A top-level testbench in SystemVerilogA dummy DUT in the testbench, which has two APB interfacesAn UVM verification environmentTwo APB System ENVs in the UVM verification environmentTwo tests illustrating directed and random transaction generation
tb_apb_svt_uvm_basic_program_sys	Basic	The example demonstrates the usage of program block.It consists of the following:A top-level testbench in SystemVerilogA dummy DUT in the testbench, which has two APB interfacesThe `apb_basic_tb.sv` file containing the user program in the exampleAn UVM verification environmentTwo APB System ENVs in the UVM verification environmentTwo tests illustrating directed and random transaction generation
tb_apb_svt_uvm_intermediate_sys	Intermediate	Not yet supported
tb_apb_svt_uvm_advanced_sys	Advanced	Not yet supported

Feedback

Installing and Running the Examples

These are the steps for installing and running example tb_apb_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_apb_svt_uvm_basic_sys -svtb

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

gmake USE_SIMULATOR=vcsvlog directed_test WAVES=1

Invoke "gmake help" to show more options.

1. Use the sim script:

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

   ./run_apb_svt_uvm_basic_sys -w random_wr_rd_test vcsvlog

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

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

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

or

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

Defines for Increasing Number of Masters and Slaves

The default max number of slaves that can be used in an `apb_system_env` is 16. This can be increased up to a maximum value of 128. To use more than 16 slaves in an APB system, you need to define the macro `+define+SVT_APB_MAX_NUM_SLAVES_<value>`.

For example:

To use 100 APB slaves in a single APB system env:

1. Add compile time option "`+define+SVT_APB_MAX_NUM_SLAVES_100`"
2. In the VIP configuration, do:

   ```
   svt_apb_system_configuration::num_slaves=100;
   ```

Support for UVM version 1.2

While using UVM 1.2, note the below requirements:

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

Steps to Integrate the UVM_REG With APB VIP

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

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

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

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

   ```
   // Declare RAL model.
   ral_sys_apb_svt_uvm_basic_slave regmodel;
   virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   ..
   /** Check if regmodel is passed to env if not then create and lock it.
    */
   if (regmodel == null) begin
   ```

Feedback

```
regmodel =
 ral_sys_apb_svt_uvm_basic_slave::type_id::create("regmodel");
regmodel.build();
regmodel.set_hdl_path_root(hdl_path);
`uvm_info("build_phase", "Reg Model created", UVM_LOW)
regmodel.lock_model();
end
uvm_config_db#(uvm_reg_block)::set(this,"apb_system_env.master",
 "apb_regmodel", regmodel);
..
endfunction : build_phase
```

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

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

4. To enable the `uvm_reg` adapter of the APB Master agent, do the following:

   Set the `uvm_reg_enable`, `svt_apb_system_configuration` attribute to one for the desired APB Master agent.

```
apb_sys_cfg.uvm_reg_enable= 1;
```

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

   The complete example in available in the VIP installation (`tb_apb_svt_uvm_basic_ral_sys`).

   Note:

   Download the example using the `dw_vip_setup_utility` (see “6.2 Installing and Running the Examples” on page 50).

Master to Slave Path Access Coverage

This feature allows user to identify the APB bridge to slave paths covered during the simulation. The cover group name defined for this purpose is trans_cross_master_to_slave_path_access. Note that this coverage works in conjunction with the APB Complex Memory Map feature. Refer to APB Class Reference HTML for details of the cover group.

Feedback

Below are the steps needed to enable this feature:

1. Enable the cover group by setting apb configuration `svt_apb_configuration::trans_cross_master_to_slave_path_access_cov_enable` to 1.
2. Enable the APB Complex Memory Map feature by setting system configuration `svt_apb_system_configuration::enable_complex_memory_map` to 1.
3. Define macro `SVT_AMBA_PATH_COV_DEST_NAMES` with the names of the slaves in the system. These are user defined names, which identify the slave ports within the system. These names will be used in the bin names of the cover group. For example,
```
`define SVT_AMBA_PATH_COV_DEST_NAMES slave_0, slave_1, slave_2, slave_3, slave_4, slave_5
```
4. In the system configuration, assign the bridge name to `svt_apb_system_configuration::source_requester_name`. This is a user defined name, which identifies the master port. This name will be used in the bin names of the cover group. For example, `apb_sys_cfg.source_requester_name = $sformatf("master_%0d",0);`
5. In bridge configuration, pushback the slave names in to `svt_apb_system_configuration::path_cov_slave_names`. Note that these names should match the names specified in the macro `SVT_AMBA_PATH_COV_DEST_NAMES`. These names signify the slave ports to which the bridge can communicate. For example,
```
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_0);
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_1);
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_2);
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_3);
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_4);
apb_sys_cfg.path_cov_slave_names.push_back(svt_amba_addr_mapper::slave_5);
```
6. Slave configuration `svt_apb_configuration::svt_amba_addr_mapper dest_addr_mappers[]` is the address mapper, which specifies the slave memory map as part of the APB Complex Memory Map feature.
7. In the Slave configuration, instantiate the address mapper. For example, `apb_sys_cfg.slave_cfg[0].dest_addr_mappers = new; apb_sys_cfg.slave_cfg[0].dest_addr_mappers[0] = svt_amba_addr_mapper::type_id::create("apb_slave_addr_mapper");`
8. In the Slave configuration, specify the name for the slave port. Note that this name should match the name specified in the macro `SVT_AMBA_PATH_COV_DEST_NAMES`. This name helps to identify the slave port. This name will be used in the bin names of the cover group. For example,
```
apb_sys_cfg.slave_cfg[0].dest_addr_mappers[0].path_cov_slave_component_name = svt_amba_addr_mapper::slave_0;
```

9. Below is an optional step. This step needs to be done only if, for a given address, the destination is different based on originating master. Note that these names should match the names specified in `svt_apb_system_configuration::source_requester_name`. For example, `apb_sys_cfg.slave_cfg[0].dest_addr_mappers[0].source_masters.push_back("bridgemaste`

Once the above configurations are done, run the simulation, and review the cover group `trans_cross_master_to_slave_path_access` in coverage report.

Support for Unaligned Addresses

The AMBA APB protocol does not mention about address alignment because there is no ‘SIZE’ signal to align the address. However, according to ARM specification, it is not illegal to perform unaligned address transfers in APB.

By default, the addresses are generated and driven such that they are aligned with the data bus width. This feature provides the capability for APB VIP to perform unaligned address transfers.

This is applicable for all APB specifications.

Use Model

To enable unaligned transfer support in each manager and subordinate, the component configuration `svt_apb_configuration::unaligned_address_support` must be set to 1.

```
cfg.unaligned_address_support = 1;
cfg.slave_cfg[0].unaligned_address_support = 1;
```

Sometimes you may require the VIP to generate PSTRB values based on the unaligned address provided in the sequence. To do this, the `generate_pstrb_for_unaligned_addr` configuration must be set to 1.

```
cfg.generate_pstrb_for_unaligned_paddr = 1;
```

Feedback

Configuration Attributes

These configuration attributes are added in `svt_apb_configuration` class that you must program to enable this feature:

Configuration Attribute	Description
bit attribute svt_apb_configuration::unaligned_address_support = 0	This attribute controls the unaligned address support in the APB component.By default, APB component expects the PADDR to be aligned to the data bus width. Enabling this configuration provides the VIP support for unaligned transfers. This generates unaligned addresses and performs any checks for the unaligned addresses that the manager generates.The subordinate would not flag any errors or warnings for the unaligned addresses and manage the unaligned address by aligning it to the appropriate aligned address and write/read for the correct byte lane using PSTRB or address alignment.

These configuration attributes are added in the `svt_apb_system_configuration` class, which users must program to enable PSTRB generation for unaligned addresses from APB Manager VIP:

Configuration Attribute	Description
bit attribute svt_apb_system_configuration::generate_pstrb_for_unaligned_paddr = 0	This attribute controls the generation of PSTRB for unaligned addresses in the APB manager component.This is applicable for APB4 Active Manager only when unaligned transfer support is enabled by using `unaligned_address_support` =1.When set to 1, the active manager would autonomously drive PSTRB to valid values.When set to 0, active manager would drive PSTRB to a value specified by the user.

VIP Components

Active Manager

When `svt_apb_configuration::unaligned_address_support` is enabled, the active manager allows the master transaction to generate unaligned addresses and drive the unaligned address on the interface.

The constraint `reasonable_address_alignment` in `svt_apb_master_transaction` would be disabled to generate the unaligned addresses.

For APB4 systems, the unaligned address needs to generate the pstrb values accordingly to indicate the correct byte access.

For example, if ‘h101 is addressed on 32-bit data bus, then PSTRB must be driven to ‘b1110. This indicates that address ‘h100 location must be disabled when writing.

You can either set PSTRB values accordingly or the VIP sets the PSTRB values, if the manager configuration `svt_apb_system_configuration::generate_pstrb_for_unaligned_paddr` is set to 1.

Note:

For write transaction data field, the provided data must be with respect to the aligned address.

For read transaction data field, the sampled data would be shifted to reflect the data with respect to the unaligned address based on the unaligned address. The VIP shifts the sampled data until the original address.

Passive Manager

When `svt_apb_configuration::unaligned_address_support` is enabled, passive manager ensures that no errors or warnings are reported when the unaligned address is driven on the interface.

Note:

When `svt_apb_configuration::unaligned_address_support` is disabled, passive manager flags a warning which indicates that an unaligned address is observed.

Active and Passive Subordinates

When `svt_apb_configuration::unaligned_address_support` is enabled, the active/passive subordinate accepts the unaligned addresses without reporting any errors or warnings.

When performing the write to memory, correct byte lanes are selected depending on the unaligned address and the PSTRB values provided. For read transaction, the data is provided for the aligned address on the full data width. The manager needs to sample the data accordingly using the unaligned address. For write transfers, APB4 subordinate checks whether the correct PSTRB values are asserted depending on the unaligned address and data bus width.

Feedback

Note:

Subordinate would make an observed address as ‘aligned’ internally and perform the write operation to memory or read from memory with aligned address. The address variable in the transaction object still holds the value seen on interface.

For the read operations, the subordinate would align the address internally and respond on ‘prdata’ bus as if the read was for aligned address. In this way, the data is driven on the correct byte lanes and manager must be able to sample them accordingly based on the address it has sent.

Note:

For write operations:

If PSTRB exists (APB4 and above), then the memory is updated based on PSTRB value received. Also, if PSTRB is asserted for the unaligned part of bus, then this must be indicated as an error.

If PSTRB does not exist (APB4), the memory is updated based on the ‘paddr’ value. For paddr=’h101, with pdata_width=32-bit, only 3 bytes are valid. Therefore, the memory is updated only for the valid bytes based on the ‘paddr’. (by using ‘byteen’ argument from svt_mem::write method)

Protocol Checks

These new protocol checks are added:

1. `address_not_aligned_when_unaligned_address_support_not_enabled`
   - Checks whether the address is aligned, when `unaligned_address_support` configuration is not enabled.
   - It is applicable for Passive manager and Subordinate.
2. `pstrb_asserted_for_invalid_byte_in_unaligned_transfer` (Applicable for APB4)
   - Checks whether the PSTRB is driven low for the invalid byte lanes because of the address unalignment, when the APB4 system is enabled and when ‘`unaligned_address_support`’ is enabled.
   - It is applicable for Passive manager and Subordinate.

Functional Coverage

The functional coverage for this feature is not yet supported.

Feedback

Customized Naming of Agent Instances

In general, when a system environment component creates instances of multiple sub-components and these sub-components do not have a specific name, it uses default names such as master, slave[0], slave[1] and so on.

If you intend to name these as CPU, CTRLR or other meaningful names for these sub-components, then APB VIP offers you a mechanism.

Configuration Attributes

The following configuration is already present in the `svt_configuration::inst` which is inherited to the `svt_apb_configuration` by default.

- `string attribute svt_configuration::inst = SVT_UNSET_INST_NAME`

  You can use this attribute to define the instance name of a component before it is constructed. It is primarily used in situations where the creating component produces multiple sub-components and those sub-components do not have obvious names.

  Note:

  If this attribute is not set when creating the port configurations, then the VIP creates instance names using the default naming standards such as ‘master’, ‘slave[0]’ or similar names

Use Model

In the case of AMBA APB, there are separate port configurations for each subordinate agent. Therefore, the agent instance names can be set using `svt_apb_configuration::inst` from each subordinate configuration handle.

For the manager, the agent instance name can be set using the attribute `svt_apb_system_configuration::master_inst`.

If the APB system configuration instance name is `sys_cfg`, then you can set the custom agent instance as:

```
sys_cfg.master_inst = "CPU";
sys_cfg.slave_cfg[0].inst = "MEM";
```

Feedback

This would change the agent paths in the component hierarchy of UVM testbench. Therefore, the testbench settings that are dependent of VIP component hierarchy must be updated accordingly as follows(typically all settings done using uvm_config_db::set):

- In the case of manager (CPU), the agent path is `env.apb_system_env.CPU`, if the 'env' and 'apb_system_env' are instances of testbench system environment and related APB system environment.
- In the case of subordinate (MEM), the agent path is `env.apb_system_env.MEM`, if the 'env' and 'apb_system_env' are instances of testbench system environment and related APB system environment.

For example:

```
uvm_config_db#(uvm_object_wrapper)::set(this,
 "env.apb_system_env.MEM.sequencer.run_phase", "default_sequence",
 apb_slave_transaction_memory_suspend_response_sequence::type_id::get());
```

VIP Components

This section provides the behavior of each of the VIP components for this feature.

- Manager: APB system configuration has the attributes which are related to the manager and hence the `master_inst` attribute need to be set.
- Subordinate: The relevant subordinate port configuration instance attribute must be set.
- APB System: The relevant APB system environment instance is constructed based on the related system configuration. To provide a custom instance name to the system environment, svt_apb_system_configuration::inst attribute can be used. This is relevant when `svt_amba_system_env` is used. You must ensure the config_db settings appropriately for virtual interface when changing the `system_environment` instance name.

  For example:

  ```
  uvm_config_db#(virtual svt_apb_if)::set(uvm_root::get(),
   "uvm_test_top.env.amba_system_env.custom_apb_env","vif", apb_if_1);
  ```

APB SolvNetPlus Articles

This table lists the SolvNetPlus articles related to AMBA APB.

Feedback

Table 4 APB SolvNetPlus Articles

Article Number	Title
000039967	VC VIP: Example for Demonstrating the Usage of Callbacks and Analysis Port in SVT APB VIP
000022058	VC VIP: Scenario to hit 'trans_apb_state_after_reset_deasserted' in APB
000020316	VC VIP: Resolving the Fatal Error From svt_apb_if interface
000006041	VC VIP: SVT APB: PREADY Timeout Mechanism
000005718	VC VIP: Debugging Constraint Failure for reasonable_address_allignment in APB
000009001	VC VIP: Using a Manual Reg Prediction with SVT APB Register Adapter
000008786	VC VIP: Example Demonstrating ACE-APB Conversion Sequence for DUT Register Programming
000008588	VC VIP: Example to Show the Usage of APB Monitor Callback pre_output_port_put
000008582	VC VIP: Getting the Start Time of all APB Transactions
000008589	VC VIP: Preloading the Slave Memory to Dump an Entire Array of Data like readmemh in AHB, AXI, and APB

7

Using APB-D and APB-E VIP Features

Overview

APB VIP supports the following APB-D, APB-E protocol features:

- User signaling
- Parity signaling
- Protection Unit (RME)
- Subsystem ID support
- Wake-up signaling

The APB VIP supports the APB-D, APB-E features in the following VIP components:

- Active APB master
- Passive APB master (functional aspects only)
- Active APB slave
- Passive APB slave (functional aspects only)

Supported Features

The following features are supported in APB VIP:

- User signaling
- Parity signaling
- Protection Unit (RME)
- Sub-system ID
- Wake-up signaling
- APB-D, APB-E features are also supported with bind interface

Unsupported Features and Limitations

- APB-D, APB-E features are supported only for UVM
- User signaling: No check for user signals value
- Parity signaling: VIP only report the parity error, no reaction for it (ex. Transfer termination)
- APB-D, APB-E features are not supported by AMBA System Monitor (including complex memory map)
- Functional coverage of APB-D/APB-E features are not supported.

Licensing and Keys

APB-D and APB-E new features are supported under the license keys of

- VIP-AMBA-APB5-SVT or
- VIP-LIBRARY2019-SVT and
- VIP-AMBA-APB5-EA-SVT.

Use Model to Enable APB-D and APB-E Features

Macro Definition

The following macro must be defined to use APB-D, APB-E features:

- SVT_AMBA_APB5_ENABLE

Configuration Attribute

The following configuration must be set to 1:

- bit svt_apb_system_configuration::apb5_enable
  - Determines if APB5 capabilities are enabled

Feedback

User Signaling

Macro Definition

The following macros must be defined to use APB-D User signals:

- SVT_APB5_PAUSER_ENABLE
  - Enable PAUSER signal
- SVT_APB5_MAX_PAUSER_WIDTH=N
  - User-defined PAUSER signal width, default value is 1
- SVT_APB5_PWUSER_ENABLE
  - Enable PWUSER signal
- SVT_APB5_PRUSER_ENABLE
  - Enable PRUSER signal
- SVT_APB5_MAX_PWRUSER_WIDTH=N
  - User-defined PWUSER/PRUSER signal width, default value is 1
- SVT_APB5_PBUSER_ENABLE
  - Enable PBUSER signal
- SVT_APB5_MAX_PBUSER_WIDTH=N
  - User-defined PBUSER signal width, default value is 1

Configuration Attribute

The following configurations are added to allow user to control APB-D user signals:

- `int unsigned svt_apb_configuration::pauser_width`
  - This attribute indicates the width that the APB master VIP will drive on PAUSER signal, and the APB slave VIP will extract from PAUSER signal.
  - A value of 0 indicates that the APB VIP will consider PAUSER signal as disabled.
  - Default value = `` `SVT_APB5_MAX_PAUSER_WIDTH ``, can be smaller.
- `int unsigned svt_apb_configuration::pwuser_width`

Feedback

  - This attribute indicates the width that the APB master VIP will drive on PWUSER signal, and the APB slave VIP will extract from PWUSER signal.
  - A value of 0 indicates that the APB VIP will consider PWUSER signal as disabled.
  - Default value = `SVT_APB5_MAX_PWRUSER_WIDTH, can be smaller.

- `int unsigned svt_apb_configuration::pruser_width`
  - This attribute indicates the width that the APB slave VIP will drive on PRUSER signal, and the APB master VIP will extract from PRUSER signal.
  - A value of 0 indicates that the APB VIP will consider PRUSER signal as disabled.
  - Default value = `SVT_APB5_MAX_PWRUSER_WIDTH, can be smaller.
- `int unsigned svt_apb_configuration::pbuser_width`
  - This attribute indicates the width that the APB slave VIP will drive on PBUSER signal, and the APB master VIP will extract from PBUSER signal.
  - A value of 0 indicates that the APB VIP will consider PBUSER signal as disabled.
  - Default value = `SVT_APB5_MAX_PBUSER_WIDTH, can be smaller.

Transaction Attributes

- `rand bit [`SVT_APB5_MAX_PAUSER_WIDTH -1:0] svt_apb_transaction::pauser`
  - This variable represents `PAUSER` value.
  - Only applicable when `svt_apb_system_configuration::apb5_enable`, and `svt_apb_configuration::pauser_width` are set.
- `rand bit [`SVT_APB5_MAX_PWRUSER_WIDTH -1:0] svt_apb_transaction::pwruser`
  - This variable represents `PWUSER`/`PRUSER` value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and `svt_apb_configuration::pwuser_width` or `svt_apb_configuration::pruser_width` are set.
- `rand bit [`SVT_APB5_MAX_PBUSER_WIDTH -1:0] svt_apb_transaction::pbuser`
  - This variable represents `PBUSER` value.
  - Only applicable when `svt_apb_system_configuration::apb5_enable`, and `svt_apb_configuration::pbuser_width` are set.

Feedback

VIP Behavior

- PAUSER:
  - Driven by master in setup phase for write/read transfer
- PWUSER:
  - Driven by master in setup phase for write transfer
- PRUSER:
  - Driven by slave in access phase for read transfer
- PBUSER:
  - Driven by slave in access phase for write/read transfer

Protocol Checks

Signal stable checks:

- svt_apb_checker::pauser_changed_during_transfer
- svt_apb_checker::pwuser_changed_during_transfer

X/Z value checks:

- svt_apb_checker::signal_valid_pauser_check
- svt_apb_checker::signal_valid_pwuser_check
- svt_apb_checker::signal_valid_pruser_check
- svt_apb_checker::signal_valid_pbuser_check

Refer the HTML class reference document for detailed description of these checks.

Parity Signaling

Macro Definition

This macro must be defined to use APB-D Parity signals:

- SVT_APB5_PARITY_ENABLE
  - Enable all P***CHK signals

Feedback

Configuration Attribute

The following configurations are added to allow user-controls on APB-D Parity signals:

- `check_type_enum svt_apb_configuration::check_type`
  - FALSE: no checking signals on the interface.
  - `ODD_PARITY_BYTE_ALL`: odd parity checking is included for all signal.

Transaction Attributes

- bit [`SVT_APB_MAX_ADDRCHK_WIDTH -1:0] svt_apb_transaction::paddrchk
  - This variable represents PADDRCHK value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as `ODD_PARITY_BYTE_ALL`.
- bit svt_apb_transaction::pctrlchk
  - This variable represents PCTRLCHK value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit [`SVT_APB_MAX_DATACHK_WIDTH -1:0] svt_apb_transaction::pwrdatachk
  - This variable represents PWDATACHK/PRDATACHK value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit svt_apb_transaction::pstrbchk
  - This variable represents PSTRBCHK value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit [`SVT_APB5_MAX_PAUSERCHK_WIDTH -1:0] svt_apb_transaction::pauserchk
  - This variable represents PAUSERCHK value.
  - Only present if PAUSER exists.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit [`SVT_APB5_MAX_PWRUSERCHK_WIDTH -1:0] svt_apb_transaction::pwruserchk

    - This variable represents PWUSERCHK/PRUSERCHK value.
    - Only present if PWUSER/PRUSER exists.
    - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit [`SVT_APB5_MAX_PBUSERCHK_WIDTH -1:0] svt_apb_transaction::pbuserchk
    - This variable represents PBUSERCHK value.
    - Only present if PBUSER exists.
    - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- bit svt_apb_transaction::psubsysidchk
    - This variable represents PSUBSYSIDCHK value.
    - Only present if PSUBSYSID exists.
    - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- int svt_apb_transaction::parity_failure_count
    - This variable represents the failure count of parity checks, observe the failure details in the error prints.
    - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::check_type is set as ODD_PARITY_BYTE_ALL.
- function void svt_apb_transaction::gen_parity_attribute_check(string entry = "", bit perform_check, svt_apb_configuration cfg);
    - Generates parity attributes and it is also used for parity check.
    - With entry=“MASTER” and perform_check=0, generate P***CHK attributes sent by the master based on corresponding signals.
    - With entry=“MASTER” and perform_check=1, perform parity checks for signals received by the master.
    - With entry=“SLAVE” and perform_check=0, generate P***CHK attributes sent by the slave based on corresponding signals.
    - With entry=“SLAVE” and perform_check=1, perform parity checks for signals received by the slave.

- function bit svt_apb_transaction::cal_parity_bit_from_data(bit [3:0] data_granularity = 8, bit [7:0] data, bit odd_parity = 1);
  - Calculates parity bit for corresponding data and also used for checking odd_parity as received parity bit. A return value = 1 indicates that the check is passed.

VIP Behavior

- PADDRCHK:
  - Driven by master in setup phase for write/read transfer
  - Represents the parity bit(s) with respect to PADDR
  - Valid and check when PSEL asserted
- PCTRLCHK:
  - Driven by master in setup phase for write/read transfer
  - Represents the parity bit(s) with respect to PPROT/PWRITE/PNSE, if enabled
  - Valid and check when PSEL asserted
- PSELCHK:
  - Driven by master for write/read transfer
  - Represents the parity bit(s) with respect to PSEL
  - Valid and check when PRESETn de-asserted
- PENABLECHK:
  - Driven by master in setup phase for write/read transfer
  - Represents the parity bit(s) with respect to PENABLE
  - Valid and check when PSEL asserted
- PWDATACHK:
  - Driven by master in setup phase for write transfer
  - Represents the parity bit(s) with respect to PWRITE
  - Valid and check when PSEL and PWRITE asserted

- PSTRBCHK:
  - Driven by master in setup phase for write transfer
  - Represents the parity bit(s) with respect to PSTRB
  - Valid and check when PSEL and PWRITE asserted
- PREADYCHK:
  - Driven by slave in access phase for write/read transfer
  - Represents the parity bit(s) with respect to PREADY
  - Valid and check when PSEL and PENABLE asserted
- PRDATACHK:
  - Driven by slave in access phase for read transfer
  - Represents the parity bit(s) with respect to PRDATA
  - Valid and check when PSEL and PENABLE & PREADY and !PWRITE asserted
- PSLVERRCHK:
  - Driven by slave in access phase for write/read transfer
  - Represents the parity bit(s) with respect to PSLVERR
  - Valid and check when PSEL and PENABLE and PREADY asserted
- PWAKEUPCHK :
  - Driven by master for write/read transfer
  - Represents the parity bit(s) with respect to PWAKEUP
  - Valid and check when PRESETn de-asserted
  - Exists if PWAKEUP existed
- PAUSERCHK:
  - Driven by master in setup phase for write/read transfer
  - Represents the parity bit(s) with respect to PAUSER
  - Valid and check when PSEL asserted
  - Exists if PAUSER existed

- PWUSERCHK:
  - Driven by master in setup phase for write transfer
  - Represents the parity bit(s) with respect to PWUSER
  - Valid and check when PSEL and PWRITE asserted
  - Exists if PWUSER existed
- PRUSERCHK:
  - Driven by slave in access phase for read transfer
  - Represents the parity bit(s) with respect to PRUSER
  - Valid and check when PSEL and PENABLE and PREADY and !PWRITE asserted
  - Exists if PRUSER existed
- PBUSERCHK:
  - Driven by slave in access phase for write/read transfer
  - Represents the parity bit(s) with respect to PBUSER
  - Valid and check when PSEL and PENABLE and PREADY asserted
  - Exists if PBUSER existed

Protocol Checks

Signal stable checks:

- svt_apb_checker::paddrchk_changed_during_transfer
- svt_apb_checker::pctrlchk_changed_during_transfer
- svt_apb_checker::pselchk_changed_during_transfer
- svt_apb_checker::penablechk_changed_during_transfer
- svt_apb_checker::pwdatachk_changed_during_transfer
- svt_apb_checker::pstrbchk_changed_during_transfer
- svt_apb_checker::pauserchk_changed_during_transfer
- svt_apb_checker::pwuserchk_changed_during_transfer
- svt_apb_checker::psubsysidchk_changed_during_transfer
- svt_apb_checker::pwakeupchk_changed_during_transfer

X/Z value checks:

- svt_apb_checker::signal_valid_paddrchk_check
- svt_apb_checker::signal_valid_pctrlchk_check
- svt_apb_checker::signal_valid_pselchk_check
- svt_apb_checker::signal_valid_penablechk_check
- svt_apb_checker::signal_valid_pwdatachk_check
- svt_apb_checker::signal_valid_pstrbchk_check
- svt_apb_checker::signal_valid_preadychk_check
- svt_apb_checker::signal_valid_pslverr_check
- svt_apb_checker::signal_valid_prdatachk_check
- svt_apb_checker::signal_valid_pauserchk_check
- svt_apb_checker::signal_valid_pwuserchk_check
- svt_apb_checker::signal_valid_pruserchk_check
- svt_apb_checker::signal_valid_pbuserchk_check
- svt_apb_checker::signal_valid_psubsysidchk_check
- svt_apb_checker::signal_valid_pwakeupchk_check

Refer the HTML class reference document for detailed description of these checks.

RME Support

Macro Definition

The following macro must be defined to use APB-E RME feature:

- SVT_APB5_RME_ENABLE

  - Enables PNSE signal

Feedback

Configuration Attribute

The following configurations are added to allow user-controls on APB-E RME support:

- rme_support_enum svt_apb_configuration::rme_support
  - APB5_RME_TRUE indicates PNSE signal is enabled.
  - APB5_RME_FALSE indicates that the APB VIP will consider PNSE signal as disabled.
- bit svt_apb_system_configuration::enable_extra_physical_mem_region
  - This variable represents if extra memory regions are enabled in this APB system.
  - value 0 disables any memory region.
  - value 1 enables Secure/Non-secure regions in APB4, or Secure/Non-secure/Root/Realm regions in APB5.

Transaction Attribute

- `rand bit svt_apb_transaction::pnse`
  - This variable represents PNSE value, and is determined by svt_apb_transaction::physical_mem_region.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::rme_support are set.
- `rand physical_mem_region_enum svt_apb_transaction::physical_mem_region`
  - This variable represents targeted physical memory region and determines pnse/pprot1 values accordingly.
  - APB_SECURE: pnse=0, pprot1=0, applicable in APB4
  - APB_NON_SECURE: pnse=0, pprot1=1, applicable in APB4
  - APB_ROOT: pnse=1, pprot1=0, only applicable in APB5
  - APB_REALM: pnse=1, pprot1=1, only applicable in APB5
  - Only applicable when svt_apb_system_configuration::apb4_enable, or svt_apb_system_configuration::apb4_enable and svt_apb_system_configuration::apb5_enable and svt_apb_configuration::rme_support are set.

Feedback

VIP Behavior

- The combination of PNSE and PPROT[1] determines the physical address space of the transaction:

Table 5

PNSE	PPROT[1]	Physical Address Space
0	0	Secure
0	1	Non-secure
1	0	Root
1	1	Realm

- The Secure/non-secure regions are supported by default, but Root/Realm regions are only applicable in APB5.
- The PNSE signal is parity protected using the PCTRLCHK signal.

Protocol Checks

Signal stable checks:

- svt_apb_checker::pnse_changed_during_transfer

X/Z value checks:

- svt_apb_checker::signal_valid_pnse_check

Refer the HTML class reference document for detailed description of these checks.

Subsystem ID Support

Macro Definition

The following macro must be defined to use APB-E Subsystem ID:

- SVT_APB5_SUBSYS_ID_ENABLE
  - Enables PSUBSYSID signal

- SVT_APB5_MAX_PSUBSYSID_WIDTH=N
  - User-defined PSUBSYSID signal width, and the default value is 1.

Configuration Attribute

The following configurations are added to allow you to control APB-E PSUBSYSID signal:

- `int unsigned svt_apb_configuration::psubsysid_width`
  - This attribute indicates the width that the APB master VIP will drive on PSUBSYSID signal, and the APB slave VIP will extract from PSUBSYSID signal.
  - A value of 0 indicates that the APB VIP will consider PSUBSYSID signal as disabled.
  - Default value = `` `SVT_APB5_MAX_PSUBSYSID_WIDTH `` and can be smaller.

Transaction Attribute

- rand bit [`SVT_APB5_MAX_PSUBSYSID_WIDTH -1:0] svt_apb_transaction::psubsysid
  - This variable represents PSUBSYSID value.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::psubsysid_width are set.

VIP Behavior

- PSUBSYSID:
  - Driven by master in setup phase for write/read transfer
  - It is parity protected using the PSUBSYSIDCHK signal.

Protocol Checks

Signal stable checks:

- svt_apb_checker::psubsysid_changed_during_transfer

X/Z value checks:

- svt_apb_checker::signal_valid_psubsysid_check

Refer the HTML class reference document for detailed description of these checks.

Feedback

Wakeup Signaling

Macro Definition

The following macro must be defined to use APB-D Wake-up signaling:

- SVT_APB5_WAKEUP_ENABLE
  - Enables PWAKEUP signal

Configuration Attribute

The following configurations are added to allow you to control APB-D PWAKEUP signal:

- `pwakeup_signal_enum svt_apb_configuration::pwakeup_signal`
  - APB5_WAKEUP_TRUE indicates PWAKEUP signal is enabled.
  - APB5_WAKEUP_FALSE indicates that the APB VIP will consider PWAKEUP signal as disabled.
  - Only applicable when svt_apb_system_configuration::apb5_enable is set.
- `int unsigned svt_apb_slave_configuration::slaves_wait_pwakeup_pready`
  - This variable determines if the slave components will wait for asserted PWAKEUP before asserting PREADY, and how many clock cycles to be waited.
  - Value of 0 means the slave components won't wait for asserted PWAKEUP.
  - Value of positive means the max clock cycles that slave components will wait for asserted PWAKEUP, as it could interact with svt_apb_transaction::num_wait_cycles.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
- `int unsigned svt_apb_configuration::pwakeup_assert_min_delay`
  - This variable determines the minimum value of svt_apb_transaction::pwakeup_assert_delay.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
  - Default value is 1.
- `int unsigned svt_apb_configuration::pwakeup_assert_max_delay`

  - This variable determines the maximum value of svt_apb_transaction::pwakeup_assert_delay.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
  - Default value is 4.

- `int unsigned svt_apb_configuration::pwakeup_least_deassert_min_delay`
  - This variable determines the min value of svt_apb_transaction::pwakeup_least_deassert_delay.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
  - Default value is 0.
- `int unsigned svt_apb_configuration::pwakeup_least_deassert_max_delay`
  - This variable determines the max value of svt_apb_transaction::pwakeup_least_deassert_max_delay.
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
  - Default value is 5.

Transaction Attribute

- `rand pwakeup_assert_mode_enum svt_apb_transaction::pwakeup_assert_mode`
- APB5_PWAKEUP_NONE indicates PWAKEUP is disabled.
- PB5_PWAKEUP_IDLE indicates PWAKEUP asserted for IDLE transfer
- APB5_PWAKEUP_BEFORE_PSEL indicates PWAKEUP asserted before PSEL.
- APB5_PWAKEUP_DURING_PSEL indicates PWAKEUP asserted during PSEL.
- APB5_PWAKEUP_AFTER_PSEL indicates PWAKEUP asserted after PSEL, this option could risk the DUT completer to miss setup phase info.
- Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal::pwakeup_assert_mode.
- It is constrained in range of svt_apb_configuration::pwakeup_assert_min_delay (default 1) and svt_apb_configuration::pwakeup_assert_max_delay (default 4).

- Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set, and svt_apb_transaction::pwakeup_assert_mode is set to APB5_PWAKEUP_BEFORE_PSEL/APB5_PWAKEUP_AFTER_PSEL.
- `rand int unsigned svt_apb_configuration::pwakeup_least_deassert_delay`
  - Value of 0 means PWAKEUP will not de-assert until IDLE state, PSELx are all 0.
  - Value of positive means the least delay clock cycles before de-asserting PWAKEUP, as PWAKEUP can only de-assert while transfer completion. Note the exception that PWAKEUP could de-assert in advance for IDLE state.
  - It is constrained in the range of svt_apb_configuration::pwakeup_least_deassert_min_delay (default 0) and svt_apb_configuration::pwakeup_least_deassert_max_delay (default 5).
  - Only applicable when svt_apb_system_configuration::apb5_enable, and svt_apb_configuration::pwakeup_signal are set.
  - Only applicable for IDLE transfers or the first transfer, reset when PSELx are all 0.

VIP Behavior

- PWAKEUP:
  - The slave is permitted to wait for PWAKEUP to be asserted, before asserting PREADY.
  - It is parity protected using the PWAKEUPCHK signal.

Protocol Checks

Signal stable checks:

- svt_apb_checker::pwakeup_changed_during_transfer

X/Z value checks:

- svt_apb_checker::signal_valid_pwakeup_check

Refer the HTML class reference document for detailed description of these checks.

VIP Examples for APB-D, APB-E Features

The VIP installation example `tb_apb_svt_uvm_basic_active_passive_sys` demonstrates the APB-D, APB-E features in VIP.

- tests/ts.apb5_user_test.sv
- tests/ts.apb5_parity _test.sv
- tests/ts.apb5_rme_test.sv
- ts.apb5_subsys_id_test.sv
- tests/ts.apb5_wakeup_test.sv

8

Reporting Problems

Introduction

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

Debug Automation

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

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

Feedback

- Debug Port enabled
- Optionally, generates a file name *svt_model_out.fsdb* when Verdi libraries are available

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

Enabling and Specifying Debug Automation Features

Debug Automation is enabled through the use of a run-time plusarg named *+svt_debug_opts. T*his plusarg accepts an optional string-based specification to control various aspects Debug Automation. If this command control specification is not supplied, then the feature will default to being enabled on all VIP instances with the default options listed as follows:

Note the following about the plusarg:

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

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

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

The following table explains each control string:

Table 6 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.

Feedback

Table 6 Control Strings for Debug Automation plusarg (Continued)

Field	Description
end_time	Identifies when the debug verbosity settings will be removed. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the debug verbosity remains in effect until the end of the simulation.
verbosity	Message verbosity setting that is applied at the `start_time`. Two values are accepted in all methodologies: DEBUG and VERBOSE. UVM and OVM users can also supply the verbosity that is native to their respective methodologies (UVM_HIGH/UVM_FULL and OVM_HIGH/OVM_FULL). If this value is not supplied then the verbosity defaults to DEBUG/UVM_HIGH/OVM_HIGH. When this feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable on all instances:

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

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

Enable on all instances:

- starting at time 1000 until time 1500:

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

Enable debug feature on all instances using default options:

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

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

Note:

The SVT_DEBUG_OPTS option is available through the installed VIP examples, but if required, in customer environments, then a similar feature should be added to their environment. The PA=FSDB option is available in public examples and is required to enable Verdi libraries, and that when this option is used, then the Debug Opts file will record VIP activity to a file named `svt_model_log.fsdb`. In addition, the SVT Automated Debug

feature will enable waveform generation to an FSDB file, if the Verdi libraries are available. When enabled this feature, it should cause the simulator to dump waveform information only for the VIP interfaces.

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

Debug Automation Outputs

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

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

The following are the output files generated:

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

FSDB File Generation

To enable FSDB writing capabilities, the simulator compile-time options and environment must be updated to enable this. The steps to enable this are specific to the simulator

being used (the {LINUX/LINUX64} label needs to be replaced based on the platform being used). The ability to write to an FSDB file requires that the user supplies the Verdi dumper libraries when they compile their testbench. If these are not supplied then the VIP will not be enabled to generate the *svt_model_log.fsdb*file.

VCS

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

```
-debug_access
```

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

Questa

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

+define+SVT_FSDB_ENABLE -pli novas_fli.so

Incisive

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

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

Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

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.

Feedback

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 generates a “<username>.<uniqid>.svd” file in the current directory. These files are packed into a single file:

FSDB

HISTL

MISC

SLID

SVTO

SVTX

TRACE

VCD

VPD

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.

1. 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.
2. The case submittal tool will display options on how to send the file to Synopsys.

Limitations

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

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