Verification Continuum™

VC Verification IP AMBA CXS UVM User Guide

Version U-2022.12, December 2022

Contents

Preface

About This Guide

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

Guide Organization

The chapters of this guide are organized as follows:

- Chapter 1, "Introduction", introduces the CXS VIP and its features.
- Chapter 2, "Installation and Setup", describes system requirements and provides instructions on how to install, configure, and begin using the CXS VIP.
- Chapter 3, "VIP Commissioning", leads you through setting up the VIP.
- Chapter 4, "General Concepts", introduces the CXS VIP within the UVM environment and describes the data objects and components that comprise the VIP.
- Chapter 5, "Verification Features", describes the verification features supported by CXS VIP such as, Error injection.
- Chapter 6, "Verification Topologies", describes the topologies to verify DUT with CXS VIP.
- Chapter 7, "Usage Notes", describes the application notes on how to use the CXS VIP.

Customer Support

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

- Accessing SolvNetPlus

  Access documentation through SolvNetPlus from the following location:

  https://solvnetplus.synopsys.com (Synopsys password required)

- Contacting the Synopsys Technical Support Center
  - Go to https://solvnetplus.synopsys.com and open a case.

    Enter the information according to your environment and your issue.
    - Product: **Verification IP**
    - Sub Product: **CXS SVT**
    - Tool Version: T-2022.06 (example)
  - Send an e-mail message to support_center@synopsys.com
    - Include the Product name, Sub Product name, and Product version for which you want to register the problem.
  - Call 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 document describes the use of CXS 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 (CRV)
- ❖ 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

This chapter discusses the following topics which helps you get an overview of the CXS VIP:

- ✦ Product Overview
- ✦ Prerequisites
- ✦ References
- ✦ Product Features

1.1 Product Overview

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

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

1.2 Prerequisites

Familiarize with the CXS Protocol, object oriented programming, SystemVerilog, and the current supported version of UVM. This section enables you to understand all the specifications related to CXS. Ensure that the system environment on which the CXS is going to be installed is compatible with the specifications as suggested in Table 1-1.

Table 1-1 Product compatibility

Requirement Specification	Product Compatibility
Language	SystemVerilog For the IEEE SystemVerilog standard, refer the following: • IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language
Methodology	• UVM1.1d • UVM 1.2 • IEEE UVM 1800.2-2020 1.0/1.1 For an essential reference guide describing UVM as it is represented in SystemVerilog, along with a class reference, see http://www.accellera.org.
Hardware Environment	Solaris or Linux workstation with: • 400 MB available disk space for installation • 16 GB Virtual memory (recommended)
OS/ Simulator Platform	VCS simulator platform See the simulator matrix table for SVT-based VIP, available at the following location: VC VIP Library page For more information on the simulator matrix and library level updates, see *VC VIP Library Release Notes*.
Software Tools	NA
Software License	Synopsys Common Licensing (SCL) software For details on acquiring the SCL software, see Licensing Information.
Tokens	• `VIP-CXS-SVT` • `VIP-CXS-EA-SVT + VIP-LIBRARY2019-SVT`
Third-Party Software	• **Adobe Acrobat**: CXS VIP documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com. • **HTML browser:** CXS VIP includes the class-reference documentation in HTML. The VIP supports the following browser or platform combinations: - Microsoft Internet Explorer 6.0 or later (Windows) - Firefox 1.0 or later (Windows and Linux) - Netscape 7.x (Windows and Linux)

1.3 References

Table 1-2 lists the document deliverables in this release.

Table 1-2 Documentation for CXS

Reference	Archived Path
Class Reference	*$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/doc/class_ref/cxs_svt_uvm_class_reference/html/index.html*
Release Notes	*$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/doc/PDFs/cxs_svt_release_notes.pdf*
Examples	*$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/examples/sverilog/tb_cxs_svt_uvm_basic_sys*
SolvNetPlus Article	Downloading VC VIP Software from SolvNetPlus

1.4 Product Features

Table 1-3 lists the supported features in this release.

Table 1-3 Supported Features

Feature Classification	Description
Protocol Features	CXS VIP supports the following protocol functions: • CXS Issue A and B support • CXS Flow Control • Support for CXSLINKCONTROL • Support for CXSDATAFLITWIDTH up to 1024 • Support for CXSMAXPKTPERFLIT of 2/3/4 • CXS interface checking signals support • Interleave feature support • Support for CXSERRORFULLPKT
Verification Features	CXS VIP currently supports the following verification functions: • Default functional coverage (transaction, state, and toggle coverage) • Protocol checking • Control on delays and timeouts • Support for selected error scenarios
Methodology Features	CXS VIP supports the following methodology functions: • VIP organized as CXS System Environment, which includes set of CXS TXRX Agents • Analysis ports for connecting CXS TXRX Agents to Scoreboard, or any other component • CXS TXRX Agents support callbacks

2 Installation and Setup

This section leads you through installing and setting up the Synopsys CXS VIP. When you complete this checklist, the provided example testbench will be operational and the Synopsys CXS VIP will be ready to use. This chapter discusses the following topics which helps you to install and setup the CXS VIP:

- Prerequisites
- Installing VIP
- Licensing Information
- Determining Your Model Version
- The dw_vip_setup Administrative Tool

Note If you observe any issues with installing the Synopsys CXS VIP, see *"Customer Support"*.

2.1 Prerequisites

Before installing the CXS VIP, ensure the following:

- The system environment complies with the specifications as mentioned in Product compatibility.
- The software is downloaded from SolvNetPlus.

  See the SolvNetPlus article 000033353, "Downloading VC VIP Software from SolvNetPlus".

2.1.1 Preparing for Installation

- Set DESIGNWARE_HOME to the absolute path where Synopsys CXS VIP is to be installed:

  ```
  setenv DESIGNWARE_HOME absolute_path_to_designware_home
  ```

- Ensure that your environment and PATH variables are set correctly, including:
  - `DESIGNWARE_HOME/bin`: The absolute path as described in the previous step.
  - `LM_LICENSE_FILE`: The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.

    ```
    % setenv LM_LICENSE_FILE <my_license_file|port@host>
    ```

  - `SNPSLMD_LICENSE_FILE`: The absolute path to a file that contains the license keys for Synopsys software or the port@host reference to this file.

    ```
    % setenv SNPSLMD_LICENSE_FILE <my_Synopsys_license_file|port@host>
    ```

✦ DW_LICENSE_FILE: The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.

```
% setenv DW_LICENSE_FILE <my_VIP_license_file|port@host>
```

2.2 Installing VIP

For installing the TileLink VIP, follow the mentioned procedure:

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. However, the mandatory step is to set the DESIGNWARE_HOME environment variable before executing the .run file.

2.3 Licensing Information

The CXS VIP uses the Synopsys Common Licensing (SCL) software to control its usage. You can find the general SCL information from the following link:

http://www.synopsys.com/keys

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

The licensing key must reside in the files that are indicated by specific environment variables. For information about setting these licensing environment variables, see Setting Environment Variables.

The CXS VIP can be enabled by the license combinations mentioned above. They are listed in the order of being checked. If one set is not available, the next set would be checked. If none are available, CXS VIP reports an error.

Note Licensing is required if the VIP component classes are instantiated in the design. This includes envs, agents, drivers, monitors, sequencers, and components in UVM.

2.3.1 Simulation License Suspension

All VIP products support license suspension. The simulators that support license suspension allow a model to check-in its license token while a simulator is suspended and then checkout the license token when the simulation is resumed.

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

2.3.2 License Polling

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

❖ To enable license polling, set the DW_WAIT_LICENSE environment variable to 1.

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

2.3.3 Setting Environment Variables

The CXS VIP verification models require the following environment variables and path settings:

- `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, ensure to place the desired license files at the front of the list of arguments to `SNPSLMD_LICENSE_FILE`.

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

Note You can set the Synopsys VIP License using any of the three license variables in the following order:
`DW_LICENSE_FILE -> SNPSLMD_LICENSE_FILE -> LM_LICENSE_FILE`
If `DW_LICENSE_FILE` environment variable is enabled, the VIP will ignore `SNPSLMD_LICENSE_FILE` and `LM_LICENSE_FILE` settings. Therefore, to get the most efficient Synopsys VIP license checkout performance, set the `DW_LICENSE_FILE` with only the License servers which contain Synopsys VIP licenses. Also, include the absolute path to the third party executable in your PATH variable.

2.3.4 Simulator-Specific Settings

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

2.4 Determining Your Model Version

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

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

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

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

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

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

2.5 The dw_vip_setup Administrative Tool

A *design directory* is where the CXS VIP is set up for use in a testbench. A design directory is required for using VIP and, for this, the `dw_vip_setup` utility is provided.

The `dw_vip_setup` utility provides the following features:

- ❖ Creates the design directory (design_dir), which contains transactors, support files (include files), and examples (if any)
- ❖ Add a specific version of CXS VIP from DESIGNWARE_HOME to a design directory.

The `dw_vip_setup` utility is as follows:

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

To create a design directory and add a model to use it in a testbench, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a
tilelink_<master/slave>_agent_svt -svtb
```

The CXS VIP provides the following models:

- ❖ `cxs_env_svt`
- ❖ `cxs_monitor_svt`
- ❖ `cxs_protocol_svt`
- ❖ `cxs_txrx_agent_svt`

This section consists of the following subsections:

- ❖ Setting Environment Variables
- ❖ The dw_vip_setup Command

2.5.1 Setting Environment Variables

Before running `dw_vip_setup`, the `DESIGNWARE_HOME` environment must point to the location where the VIP is installed.

2.5.2 The dw_vip_setup Command

Invoke `dw_vip_setup` from the command prompt. The `dw_vip_setup` command checks the syntax of command-line arguments and makes sure that the requested input files exist. The syntax of the command is as follows:

% **dw_vip_setup** [**-p**[ath] *directory*] *switch* (*model* [**-v**[ersion] **latest** | *version_no*]) …

or

% **dw_vip_setup** [**-p**[ath] *directory*] *switch* **-m**[odel_list] *filename*

where,

[**-p**[ath] *directory*]	The optional -path argument specifies the path to your design directory. When omitted, `dw_vip_setup` uses the current working directory.

switch	The *switch* argument defines `dw_vip_setup` operation. Table 2-1 lists switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

Switch	Description
-a`[dd]` (*model* [**-v**[ersion] *version*]) …	Adds the specified model or models to the specified design directory or the current working directory. If you do not specify a version, the latest version is assumed. The model names are as follows: • `cxs_env_svt` • `cxs_monitor_svt` • `cxs_protocol_svt` • `cxs_txrx_agent_svt` The -add switch makes `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` command does not attempt to remove any include files used solely by the specified model or models. The model names are as follows: • `cxs_env_svt` • `cxs_monitor_svt` • `cxs_protocol_svt` • `cxs_txrx_agent_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 as follows: • `cxs_env_svt` • `cxs_monitor_svt` • `cxs_protocol_svt` • `cxs_txrx_agent_svt` The -update switch makes `dw_vip_setup` to build suite libraries from the same suite as the specified models, and to copy 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 script creates a simulator-run program for all supported simulators. If you specify a scenario (or system) for 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. The resulting design directory is streamlined and you can use it in SystemVerilog simulations.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-c[lean] {*scenario* | *model/scenario*}	Cleans the specified scenario or testbench in either the design directory (as specified by the `-path` switch) or the current working directory. This switch deletes all files in the specified directory, then restores all Synopsys created files to their original contents.
-i[nfo] *design* | *home*	When you specify the `-info design` switch, `dw_vip_setup` prints a list of all models and libraries, installed in the specified design directory or the current working directory, and their respective versions. You can use the output from `-info` design to create a `model_list` file. When you specify the `-info home` switch, `dw_vip_setup` prints a list of all models, libraries, and examples, available in the currently-defined `$DESIGNWARE_HOME` installation, and their respective versions. The command prints reports to STDOUT.
-h[elp]	Returns a list of valid `dw_vip_setup` switches and their correct syntax.
model	The CXS VIP models are as follows: • `cxs_env_svt` • `cxs_monitor_svt` • `cxs_protocol_svt` • `cxs_txrx_agent_svt` The *model* argument defines the model or models that `dw_vip_setup` acts upon. This argument is not needed with the `-info` or `-help` switches. All switches that require the model argument may also use a model list. You may specify a version for each listed model using the `-version` option. If omitted, `dw_vip_setup` uses the latest version. The `-update` switch ignores the model-version information.
-m[odel_list] *filename*	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -m/odel_list switch displays one model name per line and each model includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are ignored.
-s[uite_list] *filename*	Specifies a file name, which contains a list of suite names to be added, updated, or removed from the design directory. This switch is valid during the following switch operations; for example, -add, -update, or -remove. The -s/uite_list switch displays one suite name per line and each suite includes a version selector. The default version is the latest. This switch is optional, but the filename argument is required whenever mentioned. Lines in the file starting with the pound symbol (#) are ignored
`-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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-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, specify `-waves fsdb`. For Makefiles, specify `WAVES=fsdb`.
`-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` command treats all lines beginning with “#” as comments.

3 VIP Commissioning

This chapter discusses the following topics which helps you commission the CXS VIP - create the design directory, maintain it, and verify the CXS installation by installing and running the examples:

- ❖ Creating a Testbench Design Directory
- ❖ Adding a Single VIP
- ❖ Adding Multiple VIPs
- ❖ Updating an Existing Model
- ❖ Include and Import VIP Files into Your Test Environment
- ❖ VIP Compile-Time and Runtime Options
- ❖ Removing VIP Models from a Design Directory
- ❖ Reporting Information About DESIGNWARE_HOME or a Design Directory
- ❖ Verifying Installation

You can use all the features supported by CXS as mentioned in Table 1-3 only after the successful commissioning.

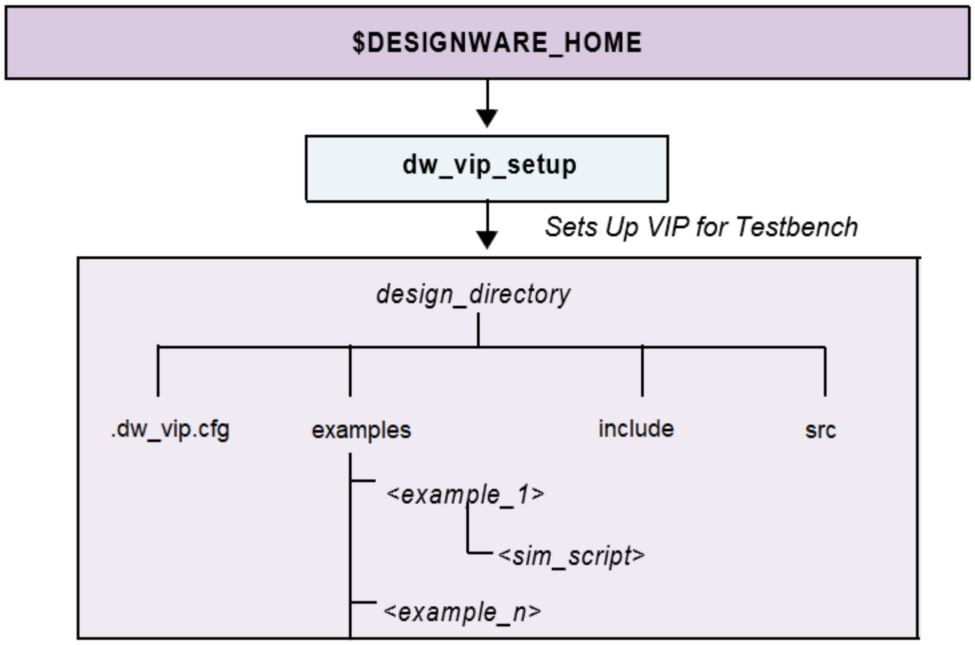
3.1 Creating a Testbench Design Directory

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

Note f 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 the control over the version of VIP in your testbench as it is isolated from the `DESIGNWARE_HOME` installation. You can use `dw_vip_setup` to update the VIP in your design directory. Figure 3-1 shows this process and the contents of a design directory.

Figure 3-1 Design Directory Created by dw_vip_setup



A design directory contains the following sub-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 the files required for model, suite, and system testbenches.
include	The language-specific include files that contain the critical information for VIP models. This directory is specified in simulator command lines.
src	The VIP-specific include files (not used by all VIP). This directory may be specified in simulator command lines.
.dw_vip.cfg	A database of all the VIP models used in the testbench. The `dw_vip_setup` utility reads this file to rebuild or recreate a design setup.

Note Do not modify this file because `dw_vip_setup` depends on the original content.

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.

3.2 Adding a Single VIP

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

You have the choice to set up all of them or only specific components.

For example, the CXS VIP contains the following components:

- `cxs_env_svt`
- `cxs_monitor_svt`
- `cxs_protocol_svt`
- `cxs_txrx_agent_svt`

Note

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

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

To get help on `dw_vip_setup`, use the following command:

```
% $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 cxs_env_svt -svlog
```

This command sets up all required files in `/tmp/design_dir`. The `dw_vip_setup` utility creates three directories in `design_dir`, which contain all necessary model files. The following three directories include files for every VIP:

- **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. The `include/sverilog` directory and the `include/verilog` directory are specified in simulator commands to locate model files.
- **src**: Synopsys-specific include files. The `src/sverilog/vcs` directory and the `src/verilog/vcs` directory must be included in the simulator command to locate model files.

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

Attention There must be only one `design_dir` installation per simulation, regardless of the number of Verification and Implementation IPs you have in your project. It is recommended not to create this directory in `$DESIGNWARE_HOME`.

3.3 Adding Multiple VIPs

All VIPs for a 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 CXS suite setup in the `design_dir` directory. In addition to the CXS 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, you must set up and locate the Ethernet and USB suites in the same `design_dir` location as CXS. Use the following commands:

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


//Add Ethernet to the same design_dir as CXS.
%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 CXS 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.

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

3.4 Updating an Existing Model

To add and update an existing model, perform the following steps:

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

   ```
   %unix> $DESIGNWARE_HOME/bin/dw_vip_setup -path /tmp/design_dir -add cxs_env_svt -svlog
   ```
3. You can also update your `design_dir` by specifying the version number of the model. Use the following command for the same:

   ```
   %unix> dw_vip_setup -path design_dir -add cxs_env_svt -v 2022.03-1 -svlog
   ```

3.5 Include and Import VIP Files into Your Test Environment

Once you set up models, you must include and import various files into your top testbench files to use the VIP. The code snippet of the includes and imports for the TileLink VIP is as follows:

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

    /* include CXS VIP interface */
    `include "svt_cxs_txrx_if.svi"

    /** Include the CXS SVT UVM package */
    `include "svt_cxs.uvm.pkg"
    /** Import UVM Package*/
    import uvm_pkg::*;

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

    /** Import the CXS VIP */
    import svt_cxs_uvm_pkg::*;
```

You must also include various VIP directories on the simulator's 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 `<design_dir>` directory would be `/tmp/design_dir`.

3.6 VIP Compile-Time and Runtime Options

Every Synopsys provided example has ASCII files containing compile-time and runtime options. The examples for models are located at the following location:

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

For example:

$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/examples/sverilog/tb_cxs_svt_uvm_basic_sys

The following files contain the options:

- For compile-time options:
  - `sim_build_options` (contain compile time options common for all simulators)
  - `vcs_build_options` (contain compile time options for VCS)
  - `mti_build_options` (contain compile time options for MTI)
  - `ncv_build_options` (contain compile time options for IUS)

- For runtime options:
  - `sim_run_options` (contain run time options common for all simulators)
  - `vcs_run_options` (contain run time options for VCS)
  - `mti_run_options` (contain run time options for MTI)
  - `ncv_run_options` (contain run time options for IUS)

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

```
vcs_build_options
Required: +define+UVM_PACKER_MAX_BYTES=1500000
Required: +define+UVM_DISABLE_AUTO_ITEM_RECORDING
Optional:   -timescale=1ns/1ps
Optional:   +define+SVT_UVM_INCLUDE_USER_DEFINES
```

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

```
   sim_run_options
   Required: +UVM_TESTNAME=$scenario
```

where, `scenario` is the UVM test name you pass to a simulator.

3.7 Removing VIP Models from a Design Directory

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

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

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

3.8 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
```

3.9 Verifying Installation

3.9.1 Installing and Running Examples

This section consists of the following subsections:

- Installing the Basic Example
- Running the Basic Example
- Getting Help on Example Run/make Scripts

3.9.1.1 Installing the Basic Example

This step occurs after you invoke the `*.run` file to install the entire CXS VIP test suite.

To install the example, you need to use the `dw_vip_setup` script. For more information, see "The dw_vip_setup Administrative Tool" section. Use the following command to invoke `dw_vip_setup`:

```
% 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
cxs_svt/tb_cxs_svt_uvm_basic_sys -svtb
```

where,

- Name of the example: `tb_cxs_svt_uvm_basic_sys`
- Install location of the example: `-path <design_dir>`

The similar steps are applicable for other examples also. The example would get installed under:
<design_dir>/examples/sverilog/cxs_svt/tb_cxs_svt_uvm_basic_sys

3.9.1.2 Running the Basic Example

Use either of the following steps to run the testbench:

1. Use the Makefile:

   Multiple tests are provided in the "tests" directory. Example tests are:

   - *ts.cxs_random_test.sv*
   - *ts. cxs_link_active_deactive_state_test.sv*
   - *ts. cxs_link_activate_packet_send_deactivate_test.sv*

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

   ```
   gmake USE_SIMULATOR=vcsvlog cxs_random_test WAVES=1
   ```

   Invoke "`gmake help`" to show more options.

2. Use the sim script:

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

   ```
   ./run_cxs_svt_uvm_basic_sys -w random_wr_rd_test vcsvlog
   ```

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

The similar steps are applicable for other examples also. For more information on installing and running the example, see the README file in the example, located at the following location:

$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/examples/sverilog/tb_cxs_svt_uvm_basic_sys/REA DME

Or,

<design_dir>/examples/sverilog/cxs_svt/tb_cxs_svt_uvm_basic_sys/README

3.9.1.3 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_cxs_svt_uvm_basic_sys --help
  usage:  run_cxs_svt_uvm_basic_sys [-32] [-verbose] [-debug_opts] [-incdir] [-
sled] [-cencrypt] [-waves] [-clean] [-nobuild] [-buildonly] [-norun] [-pa]
<scenario> <simulator>
  where   <scenario> is one of:   all cxs_continuous_credit_delay_test
cxs_credit_issue_delay_test cxs_last_prcltype_in_continuous_data_mode_test
cxs_link_activate_packet_send_deactivate_test
cxs_link_active_deactive_state_test cxs_random_test cxs_two_packets_in_flit_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
          -cencrypt     enables the access to IEEE encrypted code files
         -sled         enables SLED recording (Supported for limited titles only)
          -verbose      enable verbose mode during compilation
          -debug_opts   enable debug mode for VIP technologies that support this
option
          -waves        [fsdb|verdi|dve|vcd] enables waves dump and optionally
opens viewer (VCS only except fsdb)
          -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)
```

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

```
gmake help
Usage: gmake USE_SIMULATOR=<simulator> [VERBOSE=1] [DEBUG_OPTS=1] [SEED=<value>]
[FORCE_32BIT=1] [WAVES=fsdb|verdi|dve|dump] [NOBUILD=1] [BUILDONLY=1]
[<scenario> ...]
Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog ncvlog
vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog

Valid scenarios are:  all cxs_continuous_credit_delay_test
cxs_credit_issue_delay_test cxs_last_prcltype_in_continuous_data_mode_test
cxs_link_activate_packet_send_deactivate_test
cxs_link_active_deactive_state_test cxs_random_test cxs_two_packets_in_flit_test
```

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

4 General Concepts

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

- Introduction to UVM
- CXS VIP in an UVM Environment
- CXS UVM User Interface
- Functional Coverage
- Protocol Checks

For information on description of attributes and properties of the objects mentioned in this chapter, see the *HTML Class Reference Documentation* archived in the path as specified in Table 1-2.

4.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building testbenches using the constrained random verification. In addition, the resulting structure supports Directed testing. This chapter describes the data objects that support higher structures which comprise CXS VIP.

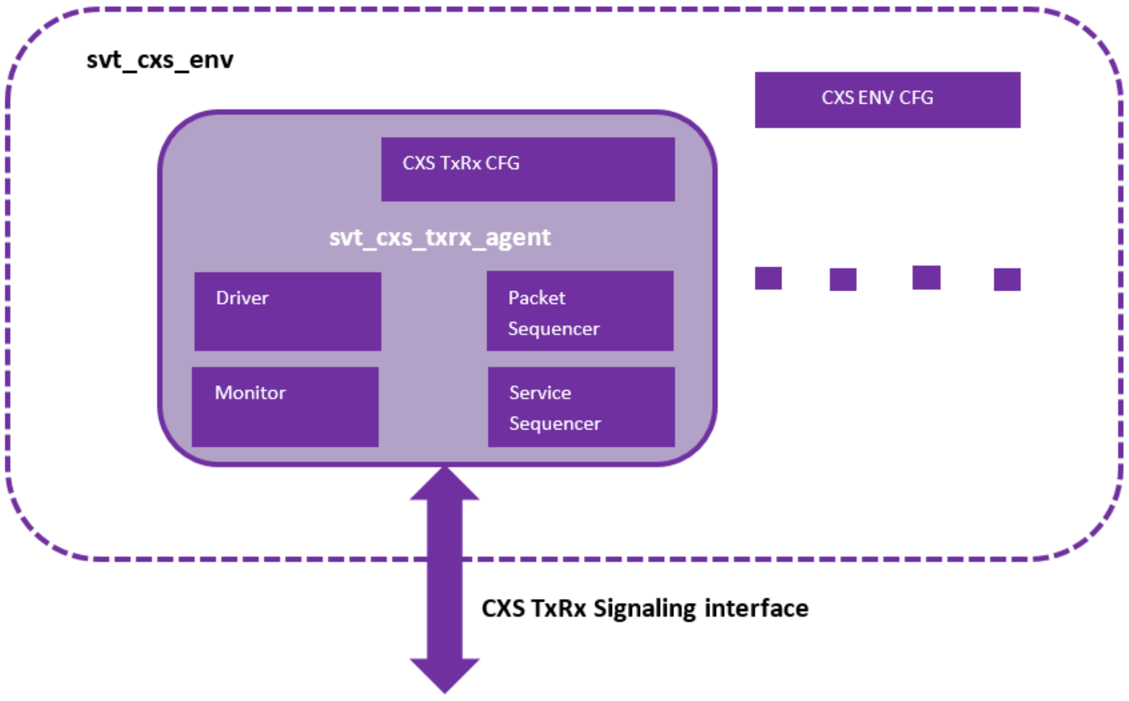
This chapter assumes that you are familiar with SystemVerilog and UVM. For more information, see the following:

- For the IEEE SystemVerilog standard, refer the following:
  - IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language
- For an essential reference guide describing UVM as it is represented in SystemVerilog, along with a class reference, see http://www.accellera.org.

4.2 CXS VIP in an UVM Environment

The following diagram shows the components of CXS VIP architecture. This includes the CXS-A and CXS-B protocols.

Figure 4-1 VIP Architecture



The Synopsys CXS VIP provides the following UVM-based usage:

❖ CXS TXRX Agent
❖ Stimulus Modeling
❖ System Environment
❖ Active and Passive Mode

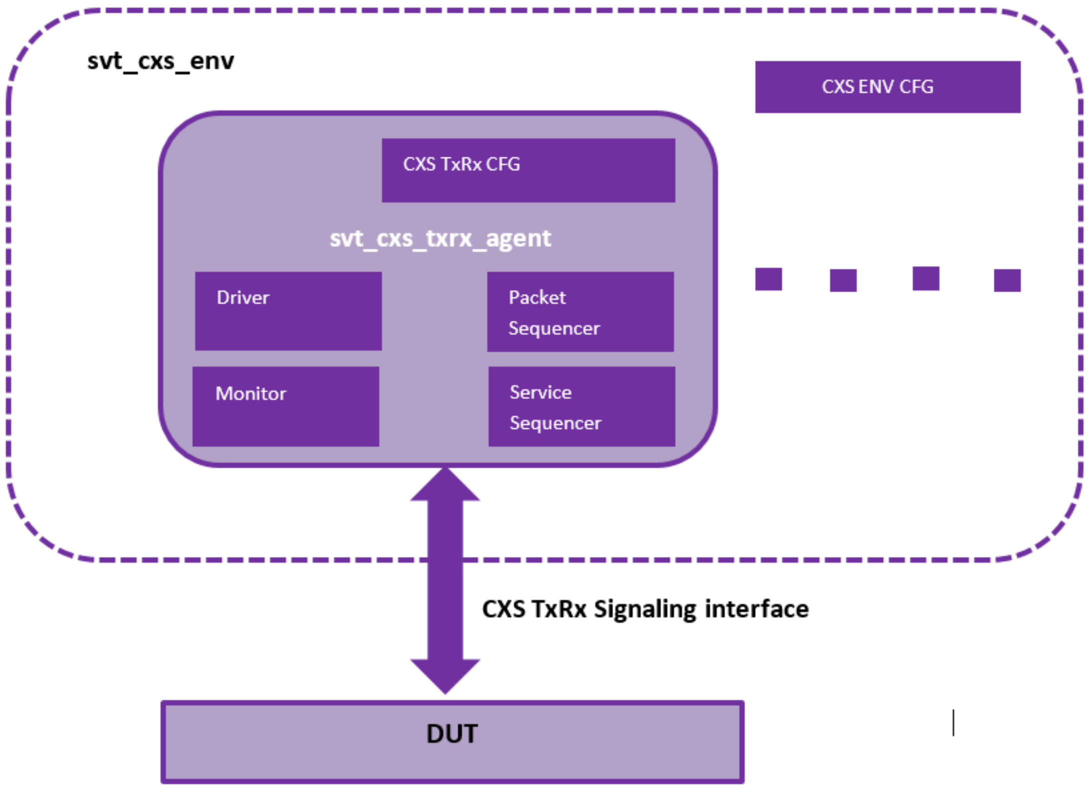
4.2.1 CXS TXRX Agent

The TXRX Agent encapsulates Driver, Monitor, CXS packet sequencer and service sequencer. The Agent can be configured to operate in active mode and passive mode. You can provide CXS packet sequences to the packet Sequencer. Link layer sequencer can be optionally configured on service sequencer for link level control such link activation/deactivation and credit delay controls.

The VIP Agent is configured using a TXRX configuration, which is available in the env configuration. The TXRX configuration should be provided to the TXRX Agent in the build phase of the test.

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

Figure 4-2 Usage with TXRX Agent



4.2.2 Stimulus Modeling

4.2.2.1 Stimulus Modeling at TXRX Agent

The `svt_cxs_packet` is the basic cxs packet description class, which depicts a transaction that has to be generated on the VIP Agent interface. The class provides rich set of attributes such as packet_size, data, delays, and so on. The class contains transaction level attributes which can be directly mapped to interface signals (for example, txrx_enderror, txrx_cxslast, txrx_cxsprcltype) and delay configurations on the TX side. The CXS packet class constraints take into account the configuration set for the VIP agent while randomizing the class instances in order to ensure that attributes are within its configured bounds.

Some of the commonly used CXS transaction class attributes are listed below.

- `inter_packet_delay`: Specifies the delays between CXS packets
- `packet_size`: Specifies the size of the packet in bytes
- `txrx_enderror`: Specifies EndError field in a packet
- `txrx_cxslast`: Specifies CXSLAST field defined by CXS-B.
- `txrx_cxsprcltype`: Specifies the protocol type of a flit.

For complete list of master transaction class attributes, see the *HTML Class Reference Documentation*.

4.2.2.1.1 CXS Packet Sequence Example

Sample VIP sequence using some of the common attributes are given as follows. You can use `'svt_cxs_base_sequence'` base class to create your own sequences.

Example 1: CXS Random Sequence

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

```
class cxs_random_sequence extends svt_cxs_base_sequence ;

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

  /** Variable to control driving of cxs_prcltype */
  bit[(`SVT_CXS_MAX_PRCL_WIDTH-1):0] prcltype_value = 0;

  /** Variable to control the frame_size*/
  int unsigned frame_size = $urandom_range(1,5);

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

  /** UVM Object Utility macro */
  `svt_xvm_declare_p_sequencer(svt_cxs_sequencer)
  `svt_xvm_object_utils_begin(cxs_random_sequence)
  `svt_xvm_field_int(sequence_length, `SVT_XVM_ALL_ON)
  `svt_xvm_object_utils_end

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

  virtual task body();
    bit status;
    `uvm_info("body","Entered ...", UVM_LOW)
    /** Gets the user provided sequence_length. */
    `ifdef SVT_UVM_TECHNOLOGY
      status = uvm_config_db#(int unsigned)::get(m_sequencer, get_type_name(),
"sequence_length", sequence_length);
    `else
      status = m_sequencer.get_config_int({get_type_name(), ".sequence_length"},
sequence_length);
    `endif
    `svt_xvm_debug("body", $sformatf("sequence_length is 'd%0d as a result of %0s.",
sequence_length, status ? "the config DB" : "randomization"));
    for (int i = 0; i <sequence_length; i++) begin
    `ifndef SVT_UVM_1800_2_2017_OR_HIGHER
      `uvm_do_with(req,
        {
        foreach(data[i]) {
          data[i] > 0;
        }
        inter_packet_delay inside{0, 5};
   packet_size inside {48, 60, 152};
      `ifdef SVT_CXS_PLUS
        if ((i == sequence_length-1) || (i% frame_size == 0))
          txrx_cxslast == 1;

        txrx_cxsprcltype == prcltype_value;
      `endif
        })
   `else

     `uvm_do(req,,,
     {
        foreach(data[i]) {
          data[i] > 0;
        }
        inter_packet_delay inside{0, 5};
   packet_size inside {48, 60, 152};
      `ifdef SVT_CXS_PLUS
        if ((i == sequence_length-1) || (i% frame_size == 0))
          txrx_cxslast == 1;

        txrx_cxsprcltype == prcltype_value;
     `endif
     })
  `endif
    end
    /** Wait for transactions to complete */
    for (int i = 0; i <sequence_length; i++)
      get_response(rsp);
    #100;
    `uvm_info("body", "Exiting...", UVM_LOW)
  endtask: body

endclass: cxs_random_sequence
```

4.2.2.1.2 CXS Link Service Sequence

The class that describes link layer transactions is 'svt_cxs_link_service'. VIP generates an object of `svt_cxs_link_service` when a command is received for which link service transaction has to be generated. The object defines the link service activity and associated properties. The `svt_cxs_link_service` class provides attributes to configure the service type (`svt_cxs_link_service::service_type`) and associate properties. Example for link service transaction class attributes are

- `service_type`: This field specifies the type of the link service that needs to be initiated. Permitted service types are ACTIVATE, DEACTIVATE and RX_CRD_GNT. They control link activation, link deactivation and credit grant respectively
- `min_cycles_in_deactive`: This field specifies the minimum number of clock cycles to stay in DEACTIVATE state.

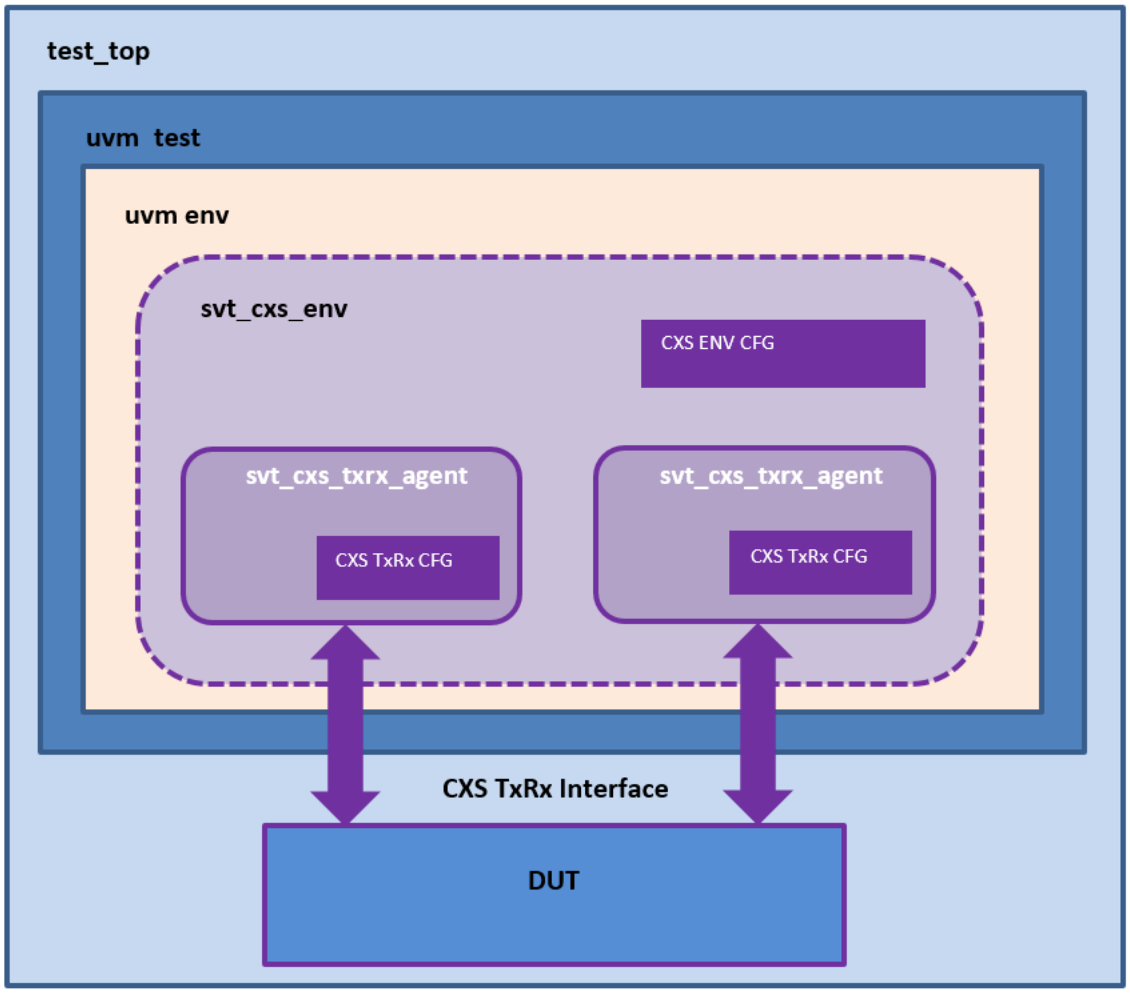
- For usage details, see `cxs_link_service_activate_sequence` and `cxs_link_service_deactivate_sequence` sequence under *tb_cxs_svt_uvm_basic_sys/env* directory.

For complete list of attributes and details, see the `svt_cxs_link_service` in *HTML Class Reference Documentation*.

4.2.3 System Environment

The CXS Env class encapsulates the VIP TXRX Agents and the CXS env configuration. The number of configured TXRX Agents is based on the CXS Env configuration provided by the user. In the build phase, the CXS Env builds the TXRX agents based on Env Configuration Class properties. After the TXRX Agents are built, they are configured by CXS Env by using the txrx configuration information available in the CXS Env configuration.

Figure 4-3 Usage with SVT CXS Environment



4.2.4 Active and Passive Mode

Table 4-1 lists the behavior of VIP Agents in active and passive modes.

Table 4-1 Agents in Active and Passive Mode

Component behavior in active mode	Component behavior in passive mode
In active mode, TXRX VIP components generate transactions on the signal interface.	In passive mode, the VIP components do not generate transactions on the signal interface. These components only sample the signal interface.
The VIP components continue to perform passive functionality of coverage and protocol checking. You can enable/disable this functionality through configuration.	VIP TXRX components monitor the input and output signals, and perform passive functionality such as coverage and protocol checking. You can enable/disable this functionality through configuration options.
The Monitor within the component performs protocol checks on all signals.	The monitor within the component performs protocol checks on all signals. In passive mode, signals are considered as input signals.
The delay values reported in the CXS packet provided by the VIP components are the values provided by the user, and not the sampled delay values.	The delay values are not reported by VIP.

4.3 CXS UVM User Interface

This section gives an overview of the user interface into the CXS VIP:

- VIP Interface Connection
- Configuration Objects
- Transaction Objects
- Analysis Ports
- Callbacks
- Interfaces and Modports

4.3.1 VIP Interface Connection

CXS VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top level interface svt_cxs_if is provided which contains an array of svt_cxs_txrx interfaces.

For example,

```
/* instantiate top level interface*/
svt_cxs_if cxs_if();
```

There are two ways on how you can set clock signal "clk" for each/all of txrx sub interfaces under this top level svt_cxs_if interface instance.

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

   For example,

```
svt_cxs_if cxs_if();
assign cxs_if.common_aclk = SystemClock;
```

Note You must leave svt_cxs_env_configuration::common_clock_mode set to default value '1' for env configuration object.

2. If any/all port interface under svt_cxs_if instance need to use a separate clock, then the 'clk' signal in the port interface should be connected to respective individual clocks and env configuration object field " svt_cxs_env_configuration::common_clock_mode" should be set to '0' to disable common clock mode.

4.3.2 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.

The CXS VIP defines following configuration classes:

❖ System configuration (`svt_cxs_env_configuration`)

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

  ✦ Number of txrx agents in the system env
  ✦ txrx configurations for txrx vip agents
  ✦ Virtual top level CXS interface
  ✦ Clock mode configuration

❖ TXRX configuration (`svt_cxs_txrx_configuration`)

  The TXRX configuration class contains configuration information which is applicable to individual TXRX agents in the CXS env. Some of the important information provided by txrx configuration class is:

  ✦ Active or Passive mode of the txrx agent
  ✦ Enable or disable protocol checks
  ✦ Enable or disable agent-level coverage
  ✦ Txrx configuration contains the virtual interface for the port

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

For details on individual members of configuration classes, see the *CXS VIP HTML Class Reference Documentation*.

4.3.3 Transaction Objects

CXS transaction objects, which are extended from the `uvm_sequence_item` base class, define a unit of CXS protocol information that is passed across the bus. The attributes of CXS transaction objects are public and are accessed directly for setting and getting values. Most of the 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.

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

CXS packet data objects are used to:

- Generate random stimulus
- Report observed transactions
- Collect functional coverage statistics

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

CXS VIP defines following transaction classes:

- CXS Packet class (`svt_cxs_packet`)

  This is the vip transaction type which contains all the physical attributes of the cxs packet like address, data, burst type, burst length, etc. It also provides the timing information the transaction, to the master and slave drivers, that is, delays for valid and ready signals with respect to some reference events.

- CXS link service (`svt_cxs_link_service`)

  The link service class contains the transaction attributes for link layer controls. This class allows the user to manually initiate link activation and deactivation by configuring appropriate link service type. This class also allows the user to configure delay in credits granted on the RX channel of the VIP.

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

4.3.4 Analysis Ports

The Monitor in the VIP TXRX Agent provides tx`_xact_observed_port` and rx_xact_observed_port analysis ports.

The VIP Agent writes the `svt_cxs_packet` object to the `tx_xact_observed_port` analysis port when a CXS transaction packet is issued on transmitter channel of the VIP . Similarly the VIP agent also writes `svt_cxs_packet` object to the `rx_xact_observed_port` analysis port when a CXS transaction packet is received on receiver channel of the VIP. This holds true in active as well as passive mode of operation of the VIP agent. You can use these analysis ports for connecting

VIP to scoreboard, or any other purpose, where a transaction object for the completed CXS packet is required. You can also create user-defined analysis ports using VIP callback class depending on your scoreboard requirements.

Usage:

Follow the mentioned steps to use VIP analysis ports in an UVM verification environment:

1. Create a cxs scoreboard class extending from uvm_scoreboard class and declare the export for the analysis port.

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

```
`uvm_analysis_imp_decl(_received)

class cxs_uvm_scoreboard extends uvm_scoreboard;

/** Analysis port connected to the CXS VIP Agent */
uvm_analysis_imp_transmitted #(svt_cxs_packet,
cxs_uvm_scoreboard)item_transmitted_export;

/** Analysis port conneted to the CXS TXRX Agent */
uvm_analysis_imp_received #( svt_cxs_packet, cxs_uvm_scoreboard) item_received_export;


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



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

endclass
```

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

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

endfunction endclass
```

3. Create `write_***()` method to get the object from the `item_observed_port` analysis port.

```
class cxs_uvm_scoreboard extends uvm_scoreboard;

..

/** This method is called by item_transmitted_export */
virtual function void write_transmitted(input svt_cxs_packet xact);
svt_cxs_packet tx_xact;

if (!$cast(tx_xact, xact.clone())) begin
`uvm_fatal("write_transmitted", "Unable to $cast the received transaction to
svt_cxs_packet ");
end
`uvm_info("write_transmitted", $sformatf("TX xact:\n%s", tx_xact.sprint()), UVM_FULL)

endfunction

/** This method is called by item_received_export */
virtual function void write_received(input svt_cxs_packet xact);
                svt_cxs_packet rx_xact;

                if (!$cast(rx_xact, xact.clone())) begin
```

```
  `uvm_fatal("write_received", "Unable to $cast the received packet to   svt_cxs_packet
");
               end

               `uvm_info("write_received", $sformatf("RX xact:\n%s", rx_xact.sprint()),
UVM_FULL)

endfunction endclass
```

4. In the ENV create an instance of the cxs`_uvm_scoreboard` and build the object

```
class cxs_env extends uvm_env;


/** CXS System ENV */
svt_cxs_env cxs_env;

/** Scoreboard */
cxs_uvm_scoreboard cxs_scoreboard;

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

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

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

super.build_phase(phase);

..
..

/* Create the scoreboard */
cxs_scoreboard = cxs_uvm_scoreboard::type_id::create("cxs_scoreboard",  this);

..
..

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

5. In the connect phase, connect VIP agent analysis ports to scoreboard

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

/* Connect the VIP agent's analysis ports with item_transmitted_export and
item_received_export ports of the  scoreboard*/.

cxs_env.cxs_txrx[0].monitor.tx_xact_observed_port.connect(cxs_scoreboard.item_transmitte
d_export);
```

```
cxs_env.cxs_txrx[0].monitor.rx_xact_observed_port.connect(cxs_scoreboard.item_received_e
xport);

endfunction

endclass
```

4.3.5 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 VIP Agent is associated with a callback class that contains a set of callback methods. These methods are called as part of the normal flow of procedural code. There are a few differences between callback methods and other methods that set them apart.

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

CXS VIP uses callbacks in three main applications:

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

4.3.5.1 CXS TXRX Agent Callbacks

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

- `svt_cxs_monitor_callback`
- `svt_cxs_callback`

Following are the callback method of `svt_cxs_monitor_callback`:

Table 4-2 Callback Methods of svt_cxs_monitor_callback

Callback Method	Description
`new_cxs_tx_packet_started`	Called when a new cxs packet is observed on the TX port
`new_cxs_rx_packet_started`	Called when a new cxs packet is observed on the RX port

Table 4-2 Callback Methods of svt_cxs_monitor_callback

Callback Method	Description
`cxs_tx_packet_ended`	Called when a cxs packet ends on TX port
`cxs_rx_packet_ended`	Called when a cxs packet ends on RX port
`cxs_tx_credit_captured`	Called when a there is a credit change on TX port
`cxs_rx_credit_captured`	Called when there is a credit change on RX port
`cxs_tx_no_traffic`	Called when a credit change without traffic on TX port
`cxs_rx_no_traffic`	Called when there is a credit change without traffic on RX port
`cxs_tx_flit_received`	Called when a cxs flit is received on TX port
`cxs_tx_flit_captured`	Called when a cxs flit is captured on TX port
`cxs_cntl_tx_flit_captured`	Called when a cxs flit is captured on TX port
`cxs_cntl_rx_flit_captured`	Called when a cxs flit is captured on RX port
`cxs_rx_flit_received`	Called when a cxs flit is received on RX port
`cxs_rx_flit_captured`	Called when a cxs flit is captured on RX port
`cxs_tx_request_captured`	Called when there is a state change on request signal on TX port
`cxs_rx_request_captured`	Called when there is a state change on request signal on RX port
`cxs_tx_ack_captured`	Called when there is a state change on ack signal on TX port
`cxs_rx_ack_captured`	Called when there is a state change on ack signal on RX port
`cxs_tx_crdrtn_flit_captured`	Called when there is credit return signal on TX port
`cxs_rx_crdrtn_flit_captured`	Called when there is credit return signal on RX port
`cxs_tx_deacthint`	Called when there is a toggling on CXS_DEACT_HINT_TX signal
`cxs_rx_deacthint`	Called when there is a toggling on CXS_DEACT_HINT_RX signal

For more information on these classes, see the *HTML Class Reference Documentation*.

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

Table 4-3 Callback Method of svt_cxs_callback

Callback Method	Description
`cxs_flit_driven`	Called when a cxs flit driven

For more information on these classes, see the *HTML Class Reference Documentation*.

4.3.6 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.

CXS VIP provides the SystemVerilog interface which can be used to connect the VIP to the DUT. A top-level interface `svt_cxs_if` is defined. The top-level interface contains an array of TXRX port sub-interfaces of type `svt_cxs_txrx_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_cxs_env_configuration::set_if`.

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

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

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

4.3.6.1 Modports

The port interface `svt_cxs_txrx_if` contains following modports:

- `cxs_modport`
- `monitor_modport`

4.3.6.2 Clocking Modes

The interface works in the following two clocking modes:

- Common clock mode
- Multiple clock mode

The clock mode can be selected using configuration parameter, `svt_cxs_env_configuration::common_clock_mode`. When set to one, the signal `common_aclk` in the top interface will be used to drive clock of all port sub-interfaces. In this case, the system clock in the environment will need to be connected to `common_aclk` signal in the top interface. When this configuration parameter is set to 0, the clk signal of each port sub-interface would need to be connected to appropriate clock in the environment.

4.3.6.2.1 Common Clock Mode

In this mode,

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

4.3.6.2.2 Multiple Clock Mode

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

4.3.6.3 Bind Interfaces

CXS VIP also supports bind interfaces for txrx agents. Bind interface is an interface which contains directional signals for CXS. You can connect DUT signals to these directional signals. Bind interfaces provided with VIP is `svt_cxs_txrx_bind_if`. To use bind interface, you must instantiate the non-bind interface, and then connect the bind interface to the non-bind interface. VIP provides `svt_cxs_txrx_connector` module to connect the VIP bind interface to the VIP non-bind interface. You must instantiate a connector module corresponding to each instance of VIP txrx agents and pass the bind interface and non-bind interface instance to this connector module.

4.4 Functional Coverage

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

❖ Default Coverage
❖ Toggle Coverage
❖ State Coverage
❖ Transaction Coverage

4.4.1 Default Coverage

The following sections describe the default coverage provided with CXS VIP. For more details on actual cover groups, see the *CXS VIP HTML Class Reference Documentation*.

4.4.2 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.

Toggle coverage can be enabled using
`svt_cxs_txrx_agent_configuration::transaction_coverage_enable`.

4.4.3 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.

State coverage can be enabled using
`svt_cxs_txrx_agent_configuration::state_coverage_enable`.

4.4.4 Transaction Coverage

Transaction coverage is coverage on various CXS transaction properties and it can be enabled using
`svt_cxs_txrx_agent_configuration::transaction_coverage_enable`.

All coverage enabling attributes are disabled by default and user needs to set the variable to 1 to enable the corresponding coverage the coverage is disabled.

For example,

```
this.cxs_txrx_agent_cfg[0].toggle_coverage_enable=1;
this.cxs_txrx_agent_cfg[0].state_coverage_enable=1;
this.cxs_txrx_agent_cfg[0].transaction_coverage_enable=1;
```

4.5 Protocol Checks

The protocol checks can be enabled by setting the configuration attribute `enable_cxs_protocol_chk` in class `svt_cxs_txrx_agent_configuration` to '1'. To disable the checks, set the attribute to '0'. Protocol checks are enabled by default.

4.5.1 Comprehensive List of Protocol Checks

Important CXS protocol checks are described in the following sections. For a comprehensive list of all protocol checks for CXS protocol, see the *CXS VIP HTML Class Reference Documentation*.

Table 4-4 CXS interface activation and deactivation checks

Protocol check	Check Description
`entry_to_stop_state`	Receiver must wait for all credit returned before de-asserting CXSACTIVEACK and entering STOP state.
`flit_in_deactivate_state`	Transmitter should not send flits in DEACTIVATE state.
`cxsactivereq_in_deactivate_state`	Transmitter must never assert CXSACTIVEREQ in DEACTIVATE state.
`cxsactivereq_in_activate_state`	Transmitter must never de-assert CXSACTIVEREQ when in ACTIVATE state.
`cxscrdrtn_assertion`	Transmitter must never assert CXSCRDRTN when in STOP/ACTIVATE state.

Table 4-5 Packet Control Field Checks

Protocol check	Check Description
`cxsvalid_and_start0_asser`	Check START[0] must be '1' if CXSVALID==1 and new packet is starting
`enderror_with_end`	ENDERROR[n] must be zero if END[n]==0
`endptr_greater_than_prev_endptr`	Check END(n)PTR must be greater than END(n-1)PTR if END > 1
`end_without_start`	Check END is never asserted without preceding START
`end_bit_asserted`	Check if END[n]==1 that END[n-1:0]==’h1

Table 4-6 CXS Protocol Operation checks

Protocol check	Check Description
`credit_consumed_check`	Transmitter must have a clock cycle delay between receiving a credit and using a credit
`cxsvalid_and_cxscrdrtn`	CXS transmitter must never assert CXSVALID and CXSCRDRTN in same clock cycle
`cxsvalid_without_credits`	CXS transmitter must never assert CXSVALID or CXSCRDRTN when it has no credits

Table 4-7 CXS interface checking signal checks

Protocol check	Check Description
`cxsprcltype_chk_type`	Check that the single bit CXSPRCLTYPECHK is maintaing odd parity with value of CXSPRCLTYPE.
`cxslast_chk_type`	Check that the single bit CXSLASTCHK replicates the value of CXSLAST to maintain the odd parity.
`cxsactive_ack_chk_type`	Check that the single bit CXSACTIVEACKCHK replicates the value of CXSACTIVEACK to maintain the odd parity.
`cxsactive_req_chk_type`	Check that the single bit CXSACTIVEREQCHK replicates the value of CXSACTIVEREQ to maintain the odd parity.
`cxscrdrtn_chk_type`	Check that the single bit CXSCRDRTNCHK replicates the value of CXSCRDRTN to maintain the odd parity.

5 Verification Features

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

- ❖ Protocol Analyzer
- ❖ Error Injection

5.1 Protocol Analyzer

Protocol analyzer is currently not supported by CXS VIP.

5.2 Error Injection

SVT CXS VIP supports implementation of predefined errors that can be injected during the execution of a transaction. This is achieved through agent level configurations in VIP. The following table describes these configurations in detail.

Table 5-1 Error Injection Scenarios

Configuration Attribute	Error Injection scenario	CXS VIP Channel	Description
`active_ack_assert_error_enable`	CXSTXACTIVEREQ is asserted by the TX but RX does not assert CXSTXACTIVEACK.	Receiver	Since this will block the link to go to RUN state, it will also not allow RX VIP to send credit grants. To recover from the hang, apply reset and restart the link. On reset, the error enable variable will also be reset to 0.
`req_to_ack_deassert_delay`	CXSTXACTIVEREQ is de-asserted by the TX but RX does not de-assert.	Receiver	To recover from the hang, apply reset and restart the link. On reset, the error enable variable will also be reset to 0.
`credit_return_error_enable`	During link deactivation, TX should not return any credits.	Transmitter	Once enabled, the VIP TX will block all credit returns. To recover, apply reset. There could be timeout error from VIP if the reset is not applied before the timeout timer value.

Table 5-1 Error Injection Scenarios

Configuration Attribute	Error Injection scenario	CXS VIP Channel	Description
`svt_cxs_credit_return_error_num`	During link deactivation, TX should return more credits than available.	Transmitter	This is an integer value that gets added to the credits available counter and hence, those many extra credits are returned by the Transmitter VIP. This is a static configuration, hence will be applicable for the entire simulation or the test. Note that there will be protocol checker error due to this error scenario. On applying reset, the error settings go back to normal.
`max_available_credits`	RX should issue more than 15 credits.	Receiver	Once enabled, this configuration bypasses the constraint and configuration check for the existing agent configuration, `max_available_credits`. To enable Rx to issue more than 15 credits, set `svt_cxs_credit_grant_error_enable=1` and `max_available_credits>15`. This is a static configuration, hence will be applicable for the entire simulation or the test. Note that there will be protocol checker error due to this error scenario. On applying reset, the error settings go back to normal.

6 Verification Topologies

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

- ❖ Using Active VIP for Testing a DUT
- ❖ Using Passive VIP for Monitoring a DUT

6.1 Using Active VIP for Testing a DUT

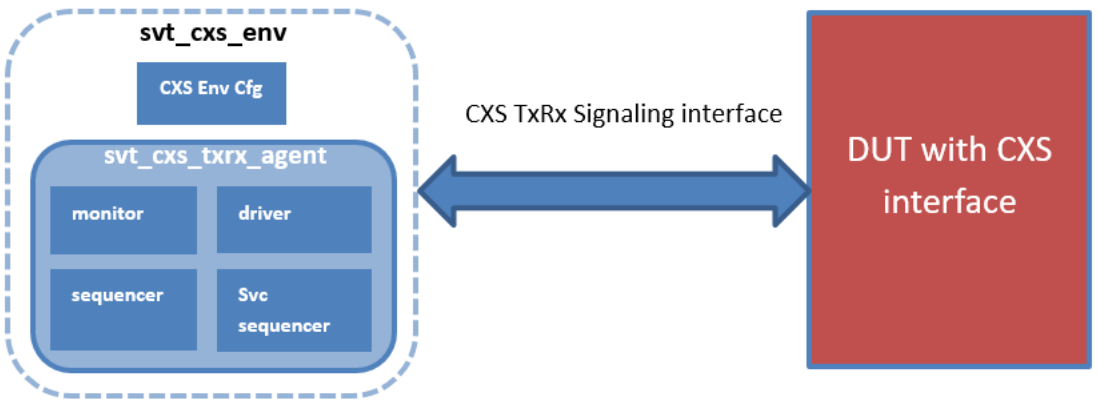
In this scenario, you are testing a CXS DUT using an UVM CXS TXRX Agent.

- ❖ Testbench setup: Configure the SVT CXS Env to have one TXRX Agent, in active mode. The active VIP Agent initiates transactions on its transmitter (TX) channel. Active mode VIP will also respond to the transactions generated by DUT on the receiver (RX) channel of the VIP. The VIP Agent will also perform passive functions such as protocol checking, coverage generation and transaction logging.
- ❖ Implementation of this topology requires the setting of the following properties: (Assuming instance name of cxs environment configuration is "`env_cfg`").
  - ✦ Env configuration settings:

    ```
    env_cfg.num_cxs = 1;
    ```
  - ✦ Port configuration settings:

    ```
    env_cfg.cxs_txrx_agent_cfg[0].is_active = 1;
    ```

Figure 6-1 Active VIP for Testing a DUT



6.2 Using Passive VIP for Monitoring a DUT

In this scenario, you are using CXS VIP to monitor transactions on a CXS interface. You can configure the CXS System Env to have one CXS Agent, in passive mode. VIP agent will perform passive functions such as protocol checking, coverage generation and transaction logging.

Implementation of this topology requires the setting of the following properties: (Assuming instance name of cxs env configuration is "`env_cfg`")

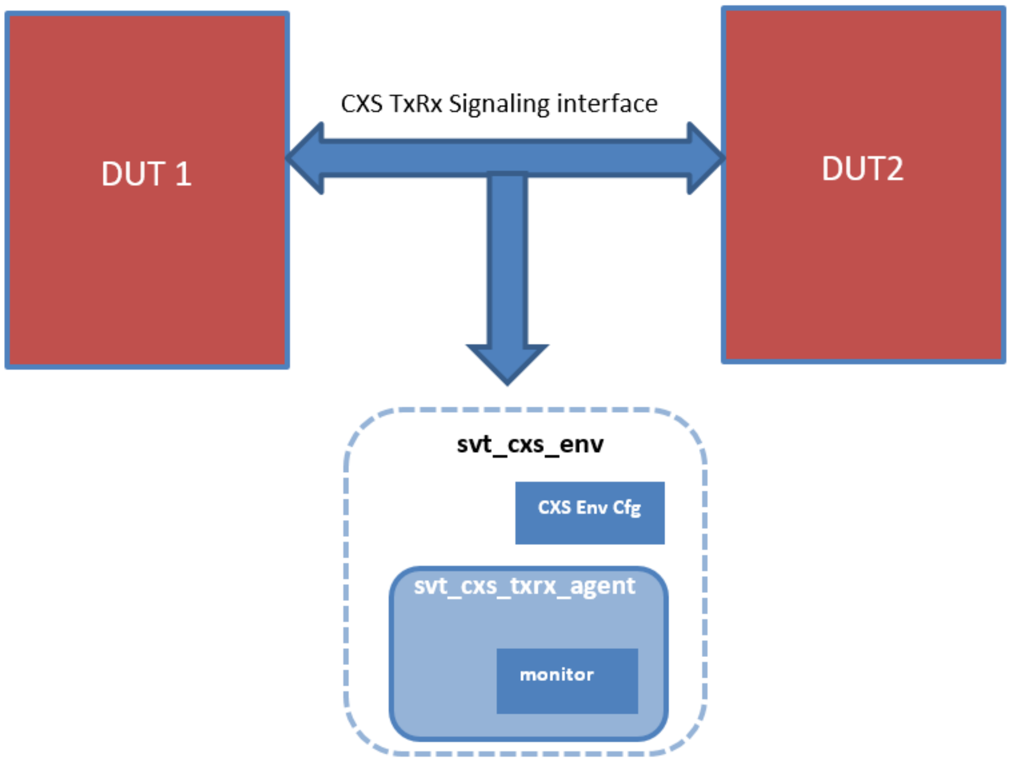
- Env configuration settings:

```
env_cfg.num_cxs = 1;
```

- Agent configuration settings:

```
env_cfg.cxs_txrx_agent_cfg[0].is_active = 0;
```

Figure 6-2 Passive VIP for Monitoring a CXS DUT Interface



7 Usage Notes

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

This chapter discusses the following topics:

- SystemVerilog UVM Example Testbenches
- Installing and Running the Examples
- CXS Protocol Support

7.1 SystemVerilog UVM Example Testbenches

This section describes SystemVerilog UVM example testbenches that show general usage for various applications. A summary of the examples is listed in Table 7-1.

Table 7-1 SystemVerilog Example Summary

Example Name	Level	Description
`tb_cxs_svt_uvm_basic_sys`	Basic	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, generates system clock, and runs the tests. CXS TXRX Agent level interface, `svt_cxs_txrx_if` instances are used in the module • A base test, which is extended to create a directed and a random test • The tests create a testbench environment, which in turn creates 3 SVT CXS Env instances: `cxs_env_1`, `cxs_env_2` and `cxs_env_passive`. • `cxs_env_1` and `cxs_env_2` are both configured in active mode with a single agent. VIP agents in `cxs_env_1` and `cxs_env_2` are connected back to back. VIP agent in `cxs_env_passive` is configured in passive mode to monitor the transactions between the agents in `cxs_env_1` and `cxs_env_2`.

Table 7-1 **SystemVerilog Example Summary**

Example Name	Level	Description
`tb_cxs_svt_uvm_multiple_agent_sys`	Basic	The example consists of the following: • A top-level module, which includes tests, instantiates interfaces, generates system clock, and runs the tests. • CXS Top level interface, `svt_cxs_if` instances are used in the module • A base test, which is extended to create a directed and a random test • The tests create a testbench environment, which in turn creates 3 SVT CXS Env instances: `cxs_env_1`, `cxs_env_2` and `cxs_env_passive`. • `cxs_env_1` and `cxs_env_2` are both configured in active mode with a single agent. VIP agents in `cxs_env_1` and `cxs_env_2` are connected back to back. VIP agent in `cxs_env_passive` is configured in passive mode to monitor the transactions between the agents in `cxs_env_1` and `cxs_env_2`.

The examples are located at the following location:

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

7.2 Installing and Running the Examples

For steps related to installation and running of the examples, see "Installing and Running Examples".

7.2.1 Configuring the Number of VIP Agents

The default max number of TXRX Agents that can be used in an `cxs_env` is 32. To use VIP with more than 32 VIP agents in a CXS Env, you need to define the macros
`+define+SVT_MAX_NUM_CXS_TXRX_AGENTS_<value>`

For example,

1. To use 45 TXRX agents in a single AXI system environment:

   Add compile time options `"+define+SVT_MAX_NUM_CXS_TXRX_AGENTS_45"`

2. In the VIP configuration, configure as follows:

   ```
   svt_cxs_env_configuration::num_cxs=45;
   ```

7.3 CXS Protocol Support

7.3.1 CXS Issue A Support

CXS VIP is configured in CXS Issue A protocol by default. So, no additional configurations are required to enable CXS A Protocol features.

7.3.2 CXS Issue B Support

CXS VIP can be configured in CXS Issue B protocol by passing compile time macro `SVT_CXS_PLUS`.

Example: `+define+SVT_CXS_PLUS`

7.3.2.1 CXS B Interface Signals

Following signals are added to VIP interface, `svt_cxs_txrx_if` to support CXS Issue B:

- `CXS_LAST_TX`: CXSLAST Signal on VIP Transmitter channel
- `CXS_LAST_RX`: CXSLAST Signal on VIP Receiver channel
- `CXS_PRCLTYPE_TX`: CXSPRCLTYPE Signal on VIP Transmitter channel
- `CXS_PRCLTYPE_RX`: CXSPRCLTYPE Signal on VIP Receiver channel

See VIP TXRX interface for details on Interface definition:

$DESIGNWARE_HOME/vip/svt/cxs_svt/latest/sverilog/include/svt_cxs_txrx_if.svi

7.3.2.2 CXS B Transaction Class Variables

Following are the transactions class variables associated with CXS Issue B signals:

- `svt_cxs_packet::txrx_cxslast`: This variable configures the user configurable value of the CXSLAST.
- `svt_cxs_packet::txrx_cxsprcltype`: This variable configures value of the CXS Protocol type signal.

7.3.2.3 Support for Interleave Feature

Interleaving feature can be enabled by setting `svt_cxs_txrx_agent_configuration::enable_interleaving` to 1. As this is CXS-B feature, `SVT_CXS_PLUS` macro must be set. When the variable is set, VIP looks for the same PRCLTYPE in continuous data mode to drive the packets till the assertion of CXSLAST. If the ongoing PRCLTYPE is not found, it will wait for a programmed counter variable, `svt_cxs_txrx_agent_configuration::svt_cxs_same_prcltype_packet_wait_timer` value and issue an error in case of a timeout.