Verification Continuum™

VC Verification IP CXL Subsystem UVM User Guide

Version S-2021.06, June 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface

About This Manual

This manual contains installation, setup, and usage material for SystemVerilog UVM users of the VC VIP CXL Subsystem, and is for design or verification engineers who want to verify CXL Subsystem operation using a UVM testbench written in SystemVerilog.

Web Resources

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

Customer Support

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

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

   Enter the information according to your environment and your issue.
2. Send an e-mail message to support_center@synopsys.com.

   Include the Product name, Sub Product name, and Tool Version in your e-mail so it can be routed correctly.
3. Telephone your local support center.
   - North America:

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

     http://www.synopsys.com/Support/GlobalSupportCenters

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

1.1 Introduction

This guide describes the use model and high-level architecture of the CXL Subsystem solution for a Compute Express Link (CXL) based system.

These are the specification references:

- Compute Express Link Specification: 20190701_ComputeExpressLink_Specification_r1p1.pdf Revision 1.1 (June 2019)
- Compute Express Link Specification: CXL Specification_rev2p0_ver1p0_2020Oct26_clean_Approved.pdf
- Logical PHY Interface (LPIF) Specification - LPIF 1.0 (March 23, 2019)
  - Contact Synopsys for LPIF 1.1 specification information.
- PCIe Base specification Gen5 v1.0: NCB-PCI_Express_Base_5.0r1.0-2019-05-22.pdf
  - Key interest: Gen5 and Alternate Protocol Negotiation (APN)
- PIPE Specification: phy-interface-pci-express-sata-usb30-architectures-3.1_v5_1_1.pdf
  - Key interest: SerDes Arch features

1.2 CXL Subsystem Features

1.2.1 Verification Features

- Facilitates CXL DUT Verification at Block level, Sub-system level, Chip level (Full Stack) topologies
- Facilitates CXL.io, CXL.cache, CXL.mem (Type 1, Type 2, Type 3) devices.
- Callbacks, error injection, protocol checks, configuration, status objects, built in sequences.
- Multiple topology support is present. For example, full stack, CXL.io only, CXL.cache/mem only and so on. Refer the chapter 4 for more information.
- Supports active mode.
- Supports normal PCIE Mode.
- CXL.io, CXL.cache and CXL.mem traffic and CXL flex bus framing after successful APN
- Supports LPIF/PIPE SerDes Arch / Serial interface
- CXL VIP as a host or device can be configured as Type 1, Type 2 or Type 3 device.

- Debug features - Logging support
  - Transaction logging feature for CXL.cache/mem in the transaction layer.
  - Transaction logging feature for CXL.io at the Transaction layer and Data Link layer level
  - Transaction logging feature for CXL.io Flits
  - Transaction logging feature for ARB-MUX component.
  - Symbol logging feature at Physical layer level
- CXL.io/CXL.cache/CXL.mem callbacks
- CXL.io/CXL.cache/CXL.mem analysis ports
- Protocol checks are supported for these:
  - CXL.io/cache/mem
  - LPIF signaling
  - ARB-MUX
  - Flex bus

1.2.2 Protocol Features

Spec version: CXL Specification_rev2p0_ver1p0_2020Oct26_clean_Approved.pdf

Chapter	Feature Description	Status
Chapter 1 Section 1.5	• Native PCIe mode, full feature support as defined in the PCIe specification • CXL mode, as defined in this specification • Configuration of PCIe vs CXL protocol mode • Signaling rate of 32 GT/s, degraded rate of 16GT/s or 8 GT/s in CXL mode • Link width support for x16, x8, x4, x2 (degraded mode), and x1 (degraded mode) in CXL mode	In the CXL Virtual Hierarchy, the ability to configure VIP as the following: • CXL Host • CXL Device • PCIE Device
Chapter 2	• CXL System Architecture - Type 1 Device - Type 2 Device - Type 3 Device • LD-ID for CXL.mem • LD-ID for CXL.io (TLP prefix)	EA feature: • Multi-Logical Device feature Phase-1 • Host Bias and Device Bias

Chapter	Feature Description	Status
Chapter 3	CXL Transaction Layer: • CXL.io - Endpoint - PCIe RC integrated EP / PCIe Endpoint - CXL Power management VDM (Credit & PM Initialization, VDM Error format) - Optional PCIe features required for CXL - Error Propagation - Memory Type Indication on ATS - Deferrable Writes • CXL.cache - - D2H Request - D2H Response - D2H Data - H2D Request - H2D Response - H2D Data - Cacheability details and request restrictions - Poison - Bogus data for write transactions • CXL.mem - - QoS Telemetry - M2S Req - M2S RwD - S2M NDR - S2M DRS - Forward Progress and Ordering Rules - Poison • Transaction Flows: - Transaction flows for Type1, Type 2 and Type 3 devices - Forward flows for Type 2 devices	• **CXL.io** - CXL Subsystem Application Layer VDM transaction and service requests can be used for Power Management VDM exchange. - PCIe TL/CXL.IO layer can be used for Advanced Error Reporting (AER) exchange from the test. - Existing ATS API can be used to override 4th DW reserved field bit 3 of byte 4 to indicate CXL Src. **Roadmap item***: - Autonomous AER reporting, ATS bit handling CXL ATS • **CXL.cache**: - CXL.cache Channel Crediting: Controlled and handled at VIP Cache/mem DL - VIP models Infinite number of cache lines which indicates that no Cache replacement policy is used and back invalidation is not supported. - 32B/64B Mixed mode transfers - CacheFlushed transaction **Roadmap items***: • Error Response: - Go_Writepull_Drop - GO-I response for RdShared, RdAny, RdOwn transactions - Snoop Filter in Host • **CXL Mem**: - 32B/64B Mixed mode transfers **Roadmap items***: - QoS for Multi-Logical Device - Out of ordering
Chapter 4	**CXL Link Layer:** **CXL.io** - • PCIe Data Link Layer as the link layer of CXL.io **CXL.mem and CXL.cache Common Link Layer** - • High-Level CXL.cache/CXL.mem Flit Overview - Flit Packing Rules - Link Layer Control Flit - Link Layer Initialization • CXL.cache/ CXL.mem Link Layer Retry • CXL.cache-Side Poison and Viral	• Updated for IDE. LDID, H6 slot format, Viral LD-ID fields. • 32B/64B Mixed mode transfers **Roadmap items***: - Viral LD-ID field handling for Multi-Logical Devices

Chapter	Feature Description	Status
Chapter 5	**ARB/MUX:** • Virtual LSM States • ARB/MUX Link Management Packets • Arbitration and Data Multiplexing/ De-multiplexing	• vLSM State - Reset, Active, Retrain, L1.1, L1.2, L1.3, L1.4, Link Reset, L2 • Status Synchronization • Protocol/ARB/MUX Link Management Packets (ALMP) • State Request ALMP, State Status ALMP • ARB/MUX Bypass Feature **EA feature**: Error scenarios with ArbMux **Roadmap items***: - Update for ArbMux State resolution for Error conditions - LinkError/LinkDisable - L1 /L2 Abort - DAPM
Chapter 6	**Flex Bus Physical Layer:** • Flex Bus.CXL Framing and Packet Layout • Link Training • Recovery.Idle and Config.Idle Transactions to L0 • L1 Abort Scenario • Retimers and Low Latency Mode	• Update to support APN for CXL 2.0, CXL 1.1 and combinations • 32GT/s, x16 or degraded modes with 16.0 GT/s and 8.0 GT/s and x8, x4,x2,x1 widths • Flits with implied EDS can be seen only with Null flit (with implied EDS) • Sync header bypass support • Drift Buffer • Bifurcation @ full stack - based on TB multi-instance feature • Link sub-division @ LPIF topology **Roadmap items***: - Implied EDS with CXL.IO / Cache.Mem / ARB-MUX - Retimer Presence Detection - L1 / L2 Abort - Bifurcation and Aggregation (aka Multi-link) natively in VIP
Chapter 7	Switching	• Feature not supported **Roadmap items***: - VIP as switch Model - VIP as switch Downstream Port - VIP as switch Upstream Port

Chapter	Feature Description	Status
Chapter 8	Control and Status Registers • Configuration Space Registers • Memory Mapped Registers	• CXL Host Enumeration API updated for CXL 2.0 device discovery and PCIE device discovery as applicable • VIP configuration and status class provide attribute to configure for desired operation and report status • Backdoor programming API for CXL Device VIP to setup Configuration and status registers **Roadmap items***: - Custom Hooks for Control and Status Registers management, control and verification
Chapter 9	Reset, Initialization, Configuration, and Manageability • Boot and Reset • Link Device Boot Flow • Warm Reset Entry flow • Cold Reset Entry flow • Sleep State Entry Flow • Functions Level Reset(LSR) • Cache Management • Enhanced FLRCXL Reset • Global Persistent Flush • Hot Plug • Software Enumeration • Software view of HDM • Manageability Model of CXL Devices	• Boot and Reset: By CXL linkup with APN @ flex bus layer, ArbMux vLSM exchange, Link layer Data packet exchange • API for PMREQ / RESETPrep using VDM transactions • LTSSM Detect entry for Surprise reset • API for GPF exchange using VDM transactions and GPF phase handling • FLR CXL.io • CXL Reset of Cache/Mem TL • Hot Plug (Hot remove and Hot Add) • PCIE and CXL Enumeration API (for CXL 1.1 and CXL 2.0 discovery) with CXL Host & Device • Memory Interleaving with single VIP in single link **Roadmap items***: • Cold Reset - PERST# • GPF - Type 1/2 • Cache Management • CXL Reset effect on Volatile HDM / Software actions • Enumeration with Virtual Hierarchy (with Switches and CXL Devices) • CXL Memory Device Label Storage Area • CXL OSC

Chapter	Feature Description	Status
Chapter 10	Power Management • Policy based Runtime Control - Idle Power - Protocol Flow • Compute Express Link Physical Layer Power Management States - CXL Power Management (Link PM Entry and Exit) - CXL.io Link Power Management - CXL.cache + CXL.mem Link Power Management	• API for PM Req Exchange (L1/L2) - using VDM transactions • API for Power management Entry & Exit (L1/L2) for both CXL.io & CXL Cache/mem **Roadmap items***: - CXL Cache/Mem link only in low power - CXL.io only link in low power - Simultaneous Host/Device PM exit scenarios
Chapter 11	Security • CXL.io IDE • CXL.Cache/Mem IDE	• Cache/Mem IDE - Containment mode, skid mode, PCRC enable/disable **Roadmap items***: - CXL.io IDE
Chapter 12	Reliability, Availability, and Serviceability • Supported RAS Features • Link CRC and Retry, Link Retraining and • Recovery, eDPC, ECRC, Hot-Plug • Corrected Error Count Information, Data Poisoning, Viral • CXL Error Handling • CXL Link Down Handling • CXL Viral Handling • CXL Error Injection	• Link CRC Error injection, detection, and Retry • Force Link Retraining and recovery • Data poison detection and reporting for Cache/mem and CXL.io • Viral detection, reporting and Viral exit through hot reset • Detected errors reported as protocol check assertions • PCIE error reporting messages to be handled by tests **Roadmap items***: • eDPC in RAS features • Autonomous CXL Error handling (logging and reporting as CIE /UIE) using PCIE AER reporting mechanism • Error logging in CXL RAS capability structure - VIP Device
Chapter 13	Performance Considerations	**Roadmap items***: VIP hooks for traffic statistics and Cache/Mem Latency measurements
Chapter 14	CXL Compliance Testing	Refer to *CXL Test Suite Product Collaterals* for details.
Appendix A	Taxonomy	• VIP configuration and topology to work as CXL.io only, CXL.io + CXL.mem, CXL.io+CXL.mem+CXL.cache • **Roadmap items***: - Bias mode and flows
Appendix B	Protocol Tables for Memory	Updates in protocol checks based on the table for Type 2 and Type 3 Cache/Mem transaction semantics

Roadmap item/s* - Contact Synopsys for these features.

1.2.3 LPIF Features

Chapter	Feature Description	Status
Section 1.2	LPIF Signal List: • LPIF Interface Signals for Logical PHY • List of Signals for PCIe support • Clock Gating Interface • Configuration Interface	• Table 1 • Table 3 **Note** - Table 2 and 4 not supported
Section 1.3-1.8	• Interface reset requirements • Exit clock gating mechanism • Data Transfer and Stall mechanism • Stream ID rules • LPIF Request and Status • Optional Handshake `pl_clk_req/lp_clk_ack` • Hot plug and Mid-Sim Reset Support	**Note** - Support for CXL.$ (CXL.io, CXL.mem and CXL.cache)
Section 1.9-1.18	LPIF State Machine • Reset, Link Reset, and Retrain • L2, L1, and L1 sub-states • Active State • Link Error and Disabled • L1 Abort and L2 Abort • L1 Reject	All states are supported. Exception - CXl.io - Reset, Active, L1 and L1 Sub-states, L2 Disabled state supported.
Section 1.19	PTM Support	Roadmap item*
Section 1.20	**Scaling Scale Up-down** • Changing number of data lanes • Scaling by clock frequency	Note - Supported scaling by clock frequency
Section 1.21	**Configuration Interface** • Sideband transfer of information • Optional Interface	Roadmap item*
Section 1.22	**Support for Pipelining** • Insertion of staging buffers • logPHY implementation for pipelining	**Note** - VIP works with DUT having pipelining support.

Chapter	Feature Description	Status
Section 1.23	**Support for PCIe** • Encoding for 2.5GT/s & 5.0 GT/s data rates • L0s support • LTSSM to LPIF state mappings	Roadmap item*
Section 1.24	**Support for CXL** • CXL over Flexbus logPHY • L0s support • LTSSM to LPIF state mappings	All features are supported for CXL
Section 1.25	Bifurcation Support • Each bifurcated port is expected to have its own LPIF interface	Note - Support present through configuration variables

1.3 Limitations

These features are not supported in this release of CXL Subsystem:

- Passive mode

2 Installation and Licensing

2.1 Installing the Product

You need to follow these steps for installation of the CXL Subsystem:

- Download the .run file (`vip_cxl_subsystem_svt_<version>.run`) for the CXL Subsystem.

Downloading From the Electronic File Transfer (EFT) (Download Center):

1. Point your web browser to https://solvnetplus.synopsys.com/DownloadCenter/dc/product.jsp.
2. Enter your Synopsys SolvNetPlus Username and Password.
3. Click 'Sign In' button.
4. Choose 'VC VIP Library' from the list of available products under 'My Product Releases'.
5. Select the Product Version from the list of available versions.
6. Click the 'Download Here' button for HTTPS download.
7. After reading the legal page, click on 'Agree and Sign In'.
8. Click on the file name to download.
9. Follow browser prompts to select a destination location.
10. You may download multiple files simultaneously.

Downloading Using FTPS/SFTP

1. Follow the above instructions through the product version selection step.
2. Click the 'FTPS Download Instructions'/ 'SFTP Download Instructions' instead of the 'Download Here' button.
3. Use the common-line lftp/sftp utility to connect to EFT and download using FTPS/SFTP protocol

If you are unable to download the Verification Subsystem IP using above instructions, obtain support for download and installation over SolvNetPlus.

- Set DESIGNWARE_HOME to the absolute path where Synopsys CXL subsystem VIP needs to be installed:

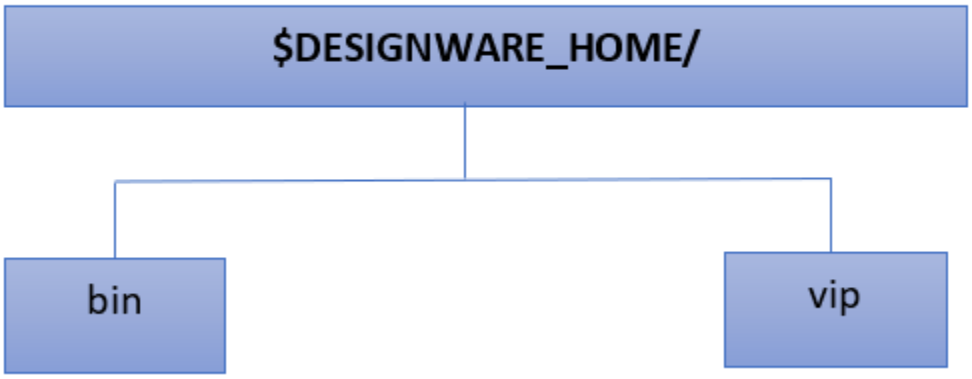
  ```
  setenv DESIGNWARE_HOME <absolute_path_to_designware_home>
  ```

- Execute the .run file by invoking its filename. For example,
  `vip_cxl_subsystem_svt_<version>.run --dir $DESIGNWARE_HOME`.

The VIP is unpacked, and all the 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.

Once the .run file is extracted, you can see the following directories inside the `DESIGNWARE_HOME`.

Figure 2-1 Directories Inside DESIGNWARE_HOME



The CXL (which is a combination of 3 products - PCIe, CXL and CXL subsystem) gets installed inside the VIP directory.

After extracting the .run file, you can see the following directories inside VIP directory.

- **common**: This contains the VIP common files and Synopsys Common Licensing files.
- **svt**: svt is set of base classes and library which are created on top of UVM/OVM/VMM methodologies. All Synopsys VIP's share a common structure and agents, environments, sequencers are designed in such a way that the VIP conveniently supports all three methodologies.

Inside svt directory, you can find the following directories.

- **common**: This contains the common structure of agents, environments and sequencers etc…
- cxl_subsystem_svt: is a superset containing CXL.IO & CXL MEM.Cache (cxl_svt) i.e. contains all the three components CXL.io, CXL.cache, and CXL.mem
- **cxl_svt**: contains the code/contents related to CXL.cache/mem component only. If you have cxl_subsystem_env then cxl_svt is automatically included.

Note cxl_svt is subset cxl_subsystem_svt.

- **pcie_svt**: contains the code/contents related to CXL.io component.

2.2 Installing and Running Examples

2.2.1 Installing the Example

These are the steps for installing and running the example `tb_cxl_subsystem_uvm_basic_sys`:

- Install the example using the following steps:
  - Create a directory where the example needs to be installed.

    ```
    % mkdir design_dir <provide any name of your choice>
    ```
  - Go inside the directory

    ```
    % cd <location_where_example_is_to_be_installed>
    ```
  - Create a new directory called 'design_dir' to install the example inside this folder.

    ```
    % mkdir design_dir
    ```

    Note: `<design_dir>` is the project directory where the example is going to be installed. You can provide any name of your choice.
  - Check the available examples using the following command:

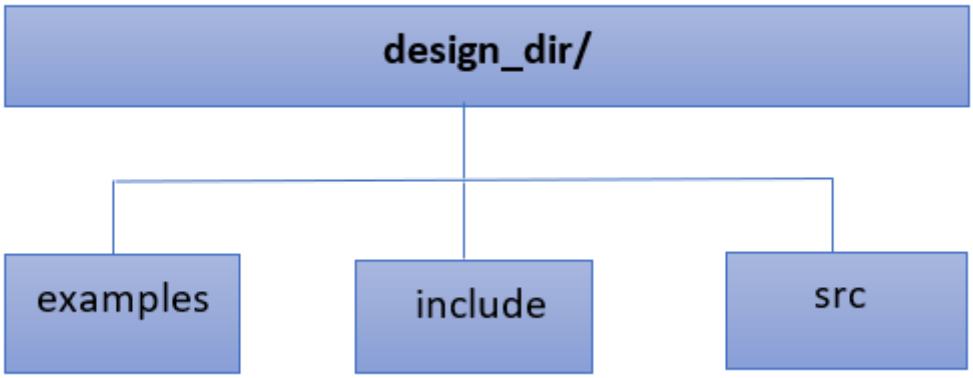
    ```
    % $DESIGNWARE_HOME/bin/dw_vip_setup -i home
    ```
  - Install the CXL Subsystem VIP example in the directory named 'design_dir'

    ```
    % $DESIGNWARE_HOME/bin/dw_vip_setup -path ./design_dir  -e
    cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys -svtb
    ```

  The above command sets up all the required files inside 'design_dir'. The dw_vip_setup utility creates three directories under design_dir which contain all the necessary model files. Files for every VIP are included in these three directories.

  You can find the following directories under 'design_dir'.

Figure 2-2 Directories inside 'design_dir'



- **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**: This contains language-specific include files. 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.

The example would get installed under:

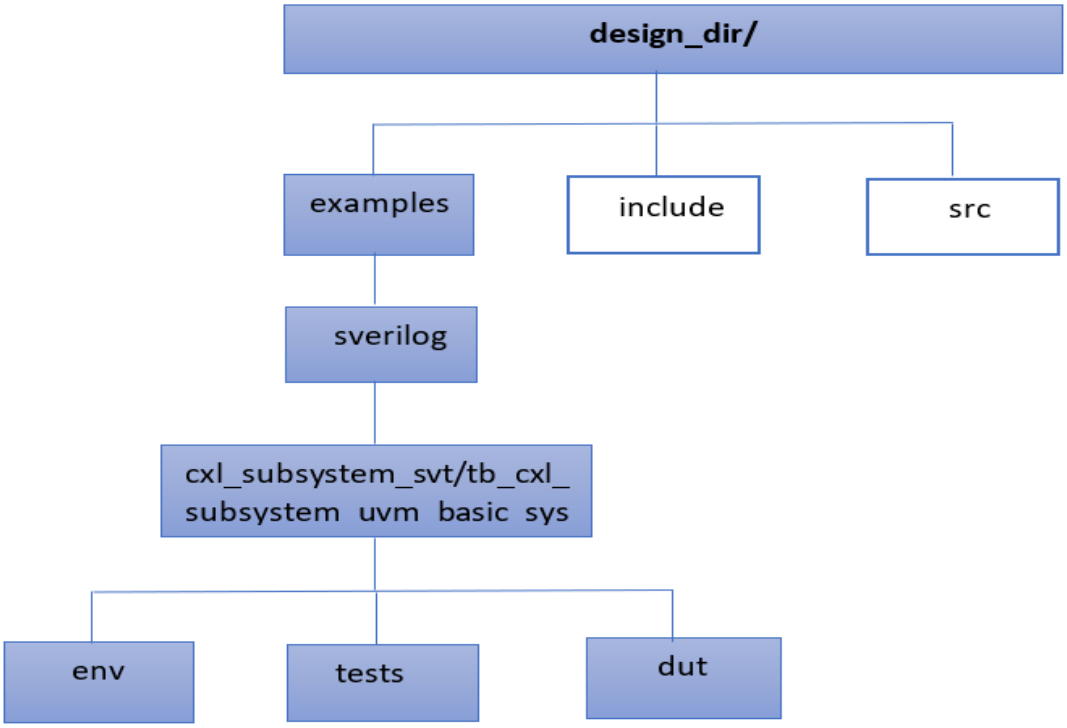
```
<design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys
```

- Go inside the example and run the example (using the command mentioned in next step)

  ```
  %cd <design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys
  ```

  Here, you can find three directories named 'env', 'tests', 'dut'

Figure 2-3 View of tb_cxl_subsystem_uvm_basic_sys Example



Note You must use the `svt_cxl_subsystem.uvm.pkg` package, if the top-level environment that is instantiated in the testbench is `svt_cxl_subsystem_env`.

2.2.1.1 Overview of tb_cxl_subsytem_uvm_basic_sys Example

After extracting the CXL Subsystem VIP example, you can find the following directories and files present inside the `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys directory`.

The class reference for VC Verification IP for CXL Subsystem is available at:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/tb_cxl_subsystem_uvm_basic_sys.html`

Files present inside `tb_cxl_subsytem_uvm_basic_sys` example:

The following files are present inside the example - `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys`

top.sv : This file includes all the required packages that must be part of the testbench in order to use the CXL Subsystem VIP. It also includes the topology specific continent. Further it contains active high 'Reset' block for the VIP to configure its' internal registers and buffers (at least 100ns)

README: This offers an overview on how to run various test cases that are part of the example with all possible topologies. It also provides description about the Testbench.

prescript: This is the prescript. This removes the topology file and compile file used in previous simulation. Also, when using CXL IDE feature, it builds the shared object if `ELLIPSYS_HOME` is set.

Makefile: This is the script to run the existing test cases in VIP back to back setup

Note: These three option files would be used when using the Makefile options to run the simulation:

vcs_build_options: This file contains the compilation options when using the VCS simulator

vcs_build_options_with_aes: This file contains the compilation options for the VCS simulator to include ELLIPSYS library files when using CXL IDE feature

vcs_elab_options*: This file contains the elaboration options when using the VCS simulator

sim_build_options : This file contains the compile time options to be picked up irrespective of the simulator used

sim_run_options : This file contains the Run time options to be picked up irrespective of the simulator used

ncv_build_options: This file contains the compilation options when using the Xcelium simulator

ncv_run_options* : This file contains the runtime options when using the Xcelium simulator

mti_build_options: This file contains the compilation options when using the MTI simulator

The following are the topology files available as part of the example. These files contain the topology specific information (Creating the VIP instances and connecting them back to back)

- topology_snps_vip_cxl_b2b.svi
- topology_snps_vip_cxl_b2b_lpif.svi
- topology_snps_vip_cxl_io_dl_b2b.svi

These are the compile files available as part of the example. These compile files are used to manage the components to be enabled and the interface to be used for the connection

Compile file related to Full Stack topology:

- compile_snps_vip_pcie_serial.f
- compile_snps_vip_cxl_io_only_pcie_pipe.f
- compile_snps_vip_cxl_io_only_pcie_serial.f

Compile files related to LPIF topology:

- compile_snps_vip_io_lpif.f
- compile_snps_vip_cache_mem_only_dl_tl_lpif.f
- compile_snps_vip_cache_mem_lpif.f

Compile files related to TL+DL topology:

- compile_snps_vip_cxl_cache_mem.f
- compile_snps_vip_cxl_io_tl_dl_only.f

Compile file related to TL only topology:

- compile_snps_vip_cxl_cache_mem_tl.f

For more information of the above topology files and/or compiles files (and for the description about the HDL Interconnect Macros used in topology files and compile options used in compile files) refer to the following section in *VC Verification IP CXL Subsystem VIP DUT Integration guide*

Directories present inside tb_cxl_subsytem_uvm_basic_sys Example:

The following directories are present inside the example - `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys`

dut/ : This directory is to keep the DUT specific files. This directory contains the following file in it. Customer specific files can be included inside this directory.

cust_pre_tb_top.svi - This files is provided for the user in order to include user packages and/or classes

tests/ : All the test cases that are part of the example are present inside this tests directory. The following test cases are part of the example `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys`

Example Name	Description

tb_cxl_subsystem_uvm_basic_sys	This example consists of following:
	❖ CXL Subsystem Host and Device connected back to back.
	❖ A top-level module, which includes tests, instantiates interfaces, HDL interconnect wrapper, generates system clock, and runs the tests
	❖ CXL.io traffic test (ts.cxl_io_random_mem_wr_rd.sv)
	❖ Type3 device CXL.mem traffic test (ts.cxl_tl_type3_mem_wr_rd.sv).
	❖ Type1 device CXL.cache traffic test (ts.cxl_tl_type1_cache_wr_rd.sv)
	❖ Type 2 Device test case (ts.cxl_tl_random_mem_wr_rd.sv)
	❖ Backdoor Memory Write test (ts.cxl_tl_backdoor_mem_wr_rd.sv)
	❖ Warm Reset test (ts.cxl_io_warm_reset.sv)
	❖ Type3 device GPF test (ts.cxl_io_tl_type3_gpf.sv)
	❖ IDE Sanity test (ts.cxl_ide_sanity_test.sv)
	❖ CXL DOE test (ts.cxl_crt_doe_capabilities.sv)
	For more details of installing and running the example, refer to the README file in the example, located at:
	`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/examples/sverilog/tb_cxl_subsystem_uvm_basic_sys/README`
	OR
	`<design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys/README`

env/ : All the environment related files are present inside the env directory. The following files and directories are present inside this directory.

svt_cxl_subsystem_base_test.sv - This is the base test. It will instantiate all the required objects like agents, env, configuration and status required for using the CXL Subsystem VIP. This will also contain the basic configurations required to configure the Subsystem VIP

svt_cxl_subsystem_basic_env.sv -

hdl_interconnect_macros.sv - This will contain implementation of all the HDL Interconnect macros. These macros will be used in topology file for creating VIP instances and for connecting signals.

svt_cxl_subsystem_ts_env.pkg - This env package file includes all other files present in env directory.

svt_cxl_subsystem_test_suite_utils.svi* - This file is a container for all complex macros which are used at multiple places to reduce lines of code.

svt_cxl_subsystem_user_defines.svi - This file contains the definition of user defines present in this testbench

svt_cxl_dispatch_connector.sv - This files contains a connector class which is of type generic and used to connect dispatch sequencer of a 'local' VIP to blocking put port of a 'remote' VIP in a typical back to back testbench setup

The following files include the test cases specific to that component

- cxl_cache_mem_testlist.svi - This file includes CXL.cache/mem test cases
- cxl_io_testlist.svi - This file includes CXL.io test cases
- cxl_lpif_testlist.svi* - This file includes CXL LPIF test cases

svt_cxl_subsystem_ts_sequence_collection.svi - This file includes all the sequence collection lists present inside env/seq_and_cb directory.

seq_and_cb/ - This directory contains all the sequences and sequence collection lists present as part of the example - cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys

2.2.2 Running the Example

Use one of the following to run the testbench:

2.2.2.1 Use the Makefile

These tests are provided in the `tests` directory:

i. ts.cxl_io_random_mem_wr_rd.sv
ii. ts.cxl_tl_type3_mem_wr_rd.sv
iii. ts.cxl_tl_type1_cache_wr_rd.sv
iv. ts.cxl_tl_random_mem_wr_rd.sv

Generic gmake command to run any test case:

```
gmake USE_SIMULATOR=vcsvlog <TEST NAME>
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_pcie_serial.f
SVT_CXL_SUBSYSTEM_TOPOLOGY_FILE=topology_snps_vip_cxl_b2b.svi
```

The above command is to run the test - ts.cxl_io_random_mem_wr_rd.sv for CXL.io traffic generation with FULL_STACK topology over serial interface using VCS Simulator.

```
CXL Subsystem VIP Demo >> gmake USE_SIMULATOR=vcsvlog cxl_io_random_mem_wr_rd
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_pcie_serial.f SVT_CXL_SUBSYSTE
M_TOPOLOGY_FILE=topology_snps_vip_cxl_b2b.svi
```

Possible usage scenarios with valid combinations of topology and compile files are present in Chapter 4 CXL Subsystem Verification Topologies.

You can also invoke the command `gmake help` to show more options.

Note `UVM_HIGH` verbosity can be enabled for individual component in CXL VIP to reduce space and simulation time consumption.

```
VCSPLUSARGS+=+uvm_set_verbosity=uvm_test_top.env.host_env.cache_mem_env*,_ALL_,UVM
_HIGH,time,0
VCSPLUSARGS+=+uvm_set_verbosity=uvm_test_top.env.device_env.cache_mem_env*,_ALL_,U
VM_HIGH,time,0
```

2.2.2.2 Use the sim script to run the test

For example, to run ts.cxl_io_random_mem_wr_rd.sv, do following:

```
./run_cxl_subsystem_uvm_basic_sys -w cxl_io_random_mem_wr_rd vcsvlog
```

The above command is to run the test - ts.cxl_io_random_mem_wr_rd.sv for CXL.io traffic generation with FULL_STACK topology over serial interface using VCS Simulator.

Invoke `./run_cxl_subsystem_uvm_basic_sys -help` to show more options.

2.2.2.3 Running the Example with +incdir+

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

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

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

The following code snippet shows how to run the basic example from gmake command:

```
cd <testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys
```

// To run the example using the gmake command with +incdir+

```
gmake <TEST_NAME> USE_SIMULATOR=vcsvlog
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_cxl_io_only_pcie_serial.f VERBOSE=1
WAVES=fsdb SVT_CXL_SUBSYSTEM_ENABLE_LOGGING=1 INCDIR=1
```

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 -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/cxl_subsystem_svt/R-2020.09-2/sverilog/include \
+incdir+<DESIGNWARE_HOME>/vip/svt/cxl_svt/R-2020.09-2/sverilog/include \
+incdir+<DESIGNWARE_HOME>/vip/svt/pcie_svt/R-2020.09-2/sverilog/include \
-ntb_opts uvm -full64 -sverilog +define+SVT_MEM_SA_STATUS_RD_B4_WR=0
+define+SVT_MEM_SA_STATUS_RD_RD_NO_WR=0 \
+define+UVM_PACKER_MAX_BYTES=1500000 +define+UVM_DISABLE_AUTO_ITEM_RECORDING -f
svt_cxl_subsystem_compile_file.f \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/env/seq_and_cb \
-timescale=1ns/1fs +libext+.v+.sv -y <testbench_dir>/src/verilog/vcs \
-y <testbench_dir>/src/sverilog/vcs \
```

```
+define+UVM_VERDI_NO_COMPWAVE -P pli.tab msglog.o -debug_acc+pp+dmptf+thread -
debug_region=cell+encrypt \
+define+SVT_FSDB_ENABLE +define+WAVES_FSDB +define+WAVES=\"fsdb\" +plusarg_save -
debug_access+pp+dmptf+thread \
-debug_region=cell+encrypt -notice -P /global/apps/verdi_2020.03-SP2-
1/share/PLI/VCS/LINUX64/novas.tab \
/global/apps/verdi_2020.03-SP2-1/share/PLI/VCS/LINUX64/pli.a +define+SVT_UVM_TECHNOLOGY
\
+define+SYNOPSYS_SV
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/. \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/../../env \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/../env \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/env \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/dut \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/lib \
+incdir+<testbench_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_
sys/tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

2.3 Licensing

CXL features are enabled by the following license features:

- SUB-CXL20-SVT (or)
- SUB-CXL-SVT (or)
- VIP-LIBRARY2019-SVT

Note SUB-CXL-SVT / SUB-CXL20-SVT is a superset of VIP-PCIE-G5-SVT license. This means that you can exercise all PCIe related functionality using this license.

To debug license issues, you can capture the SLI information into a log file by following these steps:

Step1: Set the following environment variables to capture the SLI information.

```
setenv FLEXLM_DIAGNOSTICS 5
setenv SLI_DEBUG_SERVER 1
setenv SLI_DEBUG_CLIENT 1
```

Step2: Generate the lic.log using the following command along with the gmake.

```
gmake .. |&tee lic.log
```

If you face any license issue, open a case over SolvNetPlus to obtain Synopsys VIP Support.

2.4 Updating an Existing Model

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

1. Install the model to the same location as your other VIPs by setting the $DESIGNWARE_HOMEenvironment variable.
2. CXL subsystem contains dependent components. The `.run` file contains dependent models such as cxl_svt, pcie_svt, user has to make sure all the VIP points to the latest version.

   You needs to update version of cxl_subsystem_svt along with all other subsystem component while migrating to a new release.

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

   Example:

   ```
   $DESIGNWARE_HOME/bin/dw_vip_setup -path basic -example
   cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys -v R-2020.12-1-T-20201221
   $DESIGNWARE_HOME/bin/dw_vip_setup -path basic -update -suite_list ./suitelistfile
   ```

   Content of suite_list file:

   ```
   cxl_svt -v R-2020.12-1-T-20201221
   pcie_svt -v R-2020.12-1-T-20201221
   svt -v R-2020.12
   common -v R-2020.12
   ```

   Invoke `$DESIGNWARE_HOME/bin/dw_vip_setup -help` to see more options.

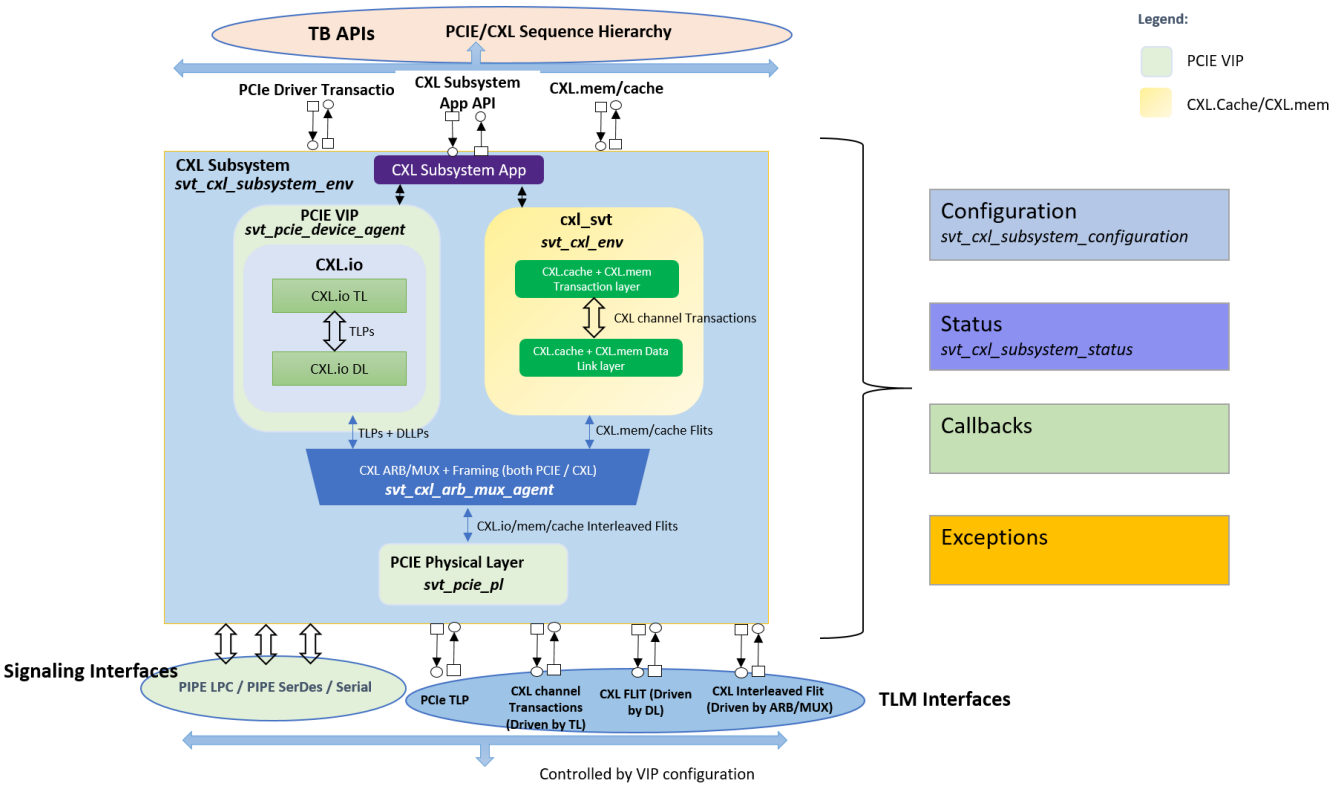
3 General Concepts

3.1 High Level Architecture

CXL Subsystem encapsulates the following components which are connected with each other to implement the CXL based subsystem environment.

1. CXL.io Link and Transaction Layer with Gen5 APN (type= svt_pcie_device_agent instance= io[0])
2. CXL.cache/CXL.mem Link and Transaction Layer (type= svt_cxl_env= cache_mem_env)
3. CXL ARB/MUX component. (type= svt_cxl_arb_mux_agent instance= arb_mux[0]()
4. Flexbus component.

This figure shows the high-level architecture of the CXL subsystem.



3.1.1 PCIE VIP with Gen5 APN and CXL.io Link and Transaction Layer

The PCIe VIP link and transaction layer is enhanced to handle the CXL.io protocol messages.

The CXL VIP autonomously negotiates CXL by sending Modified TS Ordered Sets advertising Alternate Protocol during configuration states.

3.1.2 CXL.cache/CXL.mem Link and Transaction Layer

Link Layer implements Flit Packing/Unpacking, integrity checking, ACK protocol and retry. Transaction Layer implements Tx/Rx Channel interfaces for each message channel defined by the CXL protocol, and Flow control mechanism.CXL.cache/CXL.mem Link Layer transaction layers implement the flit packing/unpacking, ack and retry protocol.

3.1.3 CXL ARB/MUX component

The ARB/MUX dynamically multiplexes the CXL.io and CXL.cache/CXL.mem inbound and outbound traffic. It interfaces with the PCIe physical layer. For outbound traffic, the ARB/MUX arbitrates between requests from the CXL.io and CXL.cache/CXL.mem link layers and multiplexes the data to forward to the physical layer. For inbound traffic, the ARB/MUX determines the protocol ID and forwards the packets to the appropriate link layer. It implements a vLSM for CXL.io and CXL.cache/CXL.mem, which negotiates the virtual link state with a link partner vLSM.

3.1.3.1 CXL ARB/MUX

ARB/MUX multiplexes and de-multiplexes CXL.io, CXL.cache/mem, and CXL ARB/MUX link management packets (ALMPs).Virtual Link State Machine (vLSM): The ARB/MUX maintains vLSM state for each CXL link layer it interfaces with - CXL.io and CXL.cache/mem.

3.1.4 Flex Bus and Physical Layer

Physical layer is common to CXL.io and CXL.cache/CXL.mem. It negotiates with the physical link. You can refer the CXL Subsystem Class reference guide for CXL Subsystem Component, sequencer, transaction classes and other related details.

The class reference for VC Verification IP for CXL Subsystem is available at:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/index.html`

3.2 UVM Components of the CXL Subsystem Environment

The Verification VIP for CXL is a suite of advanced verification components and data objects based on SystemVerilog UVM compliant technology. The Verification Compiler CXL VIP is based on the following UVM agent architecture and data objects. CXL Subsystem env is the top-level env for CXL based subsystem which encapsulates the PCIE VIP with Gen5 APN and CXL.io enhancements, CXL Cache/mem layer (CXL.cache/mem) and CXL ARB/MUX component which are connected with each other to implement the CXL based system.

svt_cxl_subsystem_env :Defines a CXL Subsystem env for the CXL Subsystem SVT suite. The CXL Subsystem env encapsulates the PCIE VIP with Gen5 APN and CXL.io enhancements, CXL Cache/mem layer (CXL.cache/mem) and CXL ARB/MUX component which are connected with each other to implement the CXL based system.

svt_cxl_subsystem_env

```
                                |
                                |--------> io[$] (type= svt_pcie_device_agent)
                                |
                                |--------> arb_mux[$](type= svt_cxl_arb_mux_agent)
                                |
                                |--------> io_lpif[$] (type= svt_cxl_io_lpif_agent)
                                |
                                |--------> subsystem_app[$]  (type= svt_cxl_subsystem_app_agent)
                                |
                                |--------> cache_mem_lpif [$](type= svt_cxl_cache_mem_lpif_agent)
                                |
                                |--------> cache_mem_env(type= svt_cxl_env)
                                                |
                                                |--------> cache_mem[$] (type= svt_cxl_agent)
```

Refer HTML class Reference for more details:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/Env_class_list.html
```

3.2.1 UVM Component for CXL Agents

- **svt_pcie_device_agent** :PCIE SVT agent acting as RC/EP. This generates CXL.io transactions. Refer the PCIE SVT class reference for more details.
- **svt_cxl_arb_mux_agent** :The `svt_cxl_arb_mux_agent` encapsulates ARB/MUX driver, sequencer and monitor.
- **svt_cxl_io_lpif_agent** : Agent class that models IO LPIF component
- **svt_cxl_cache_mem_lpif_agent** : Agent class that models Cache/Mem LPIF component.
- **svt_cxl_env** :The CXL ENV encapsulates the CXL.cache/mem Host or Device agent, CXL system sequencer and the CXL system configuration. This generates CXL.cache/mem transactions.

  Note: `svt_cxl_subsystem_env` is top level env, and it contains the svt_cxl_env, which is nothing but the `cache_mem_env`

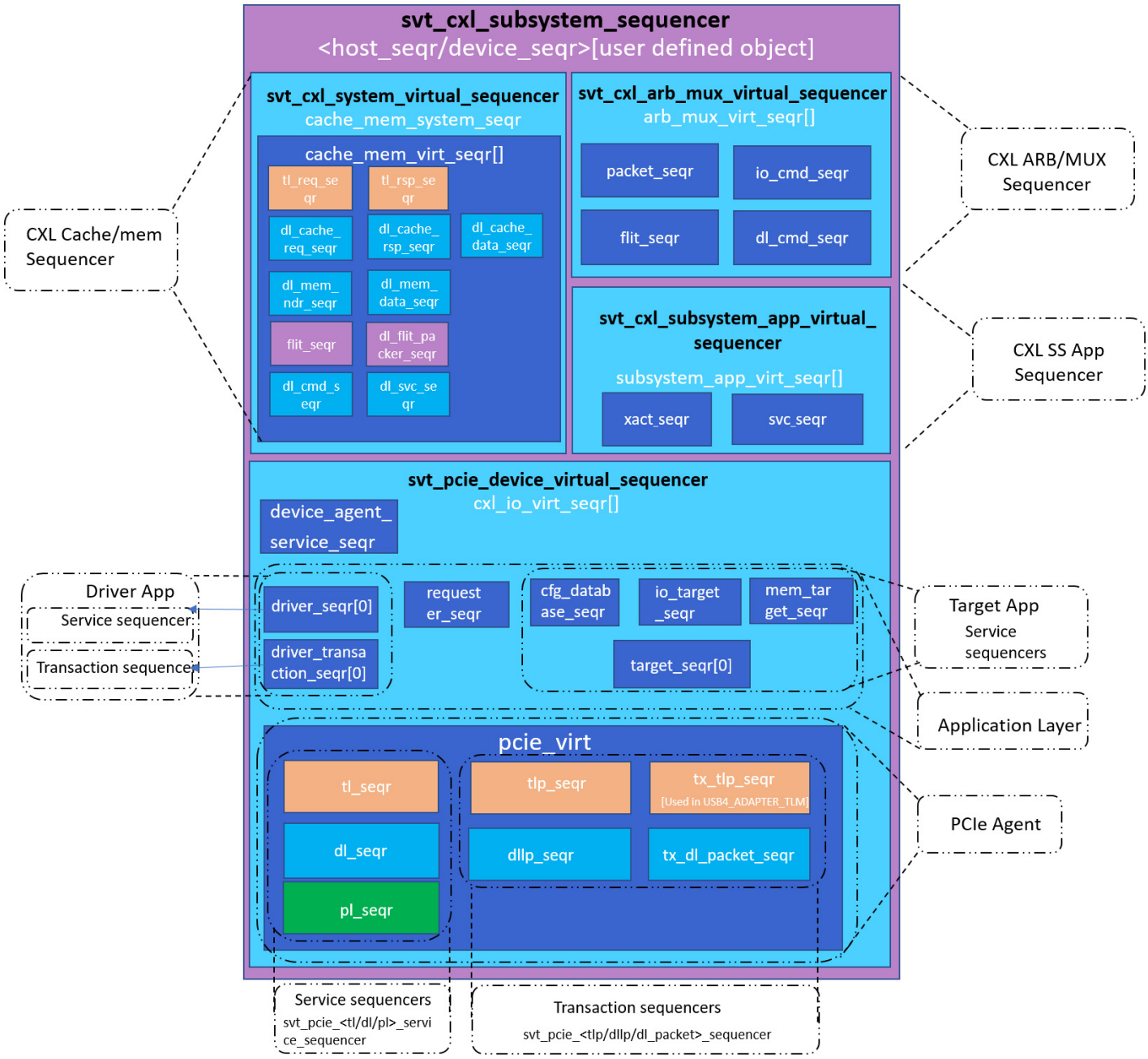
  - svt_cxl_agent : This class is the CXL Agent class. This signifies the cache_mem agent. This contains the Transaction Layer and Link Layer of CXL Cache/mem component

  Refer HTML class reference for more details:

  ```
  $DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_publi
  c/html/Agent_class_list.html
  ```

3.3 Sequencers and Sequence APIs

The CXLVIP supports extending UVM sequence item 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.



3.3.1 Sequencers

Each component in the agent has its own service sequencer. All SVT sequences are derived from `uvm_sequence`. Sequences or individual sequence items are executed on the appropriate sequencer.

svt_cxl_subsystem_sequencer: This class is the UVM System sequencer class. Top level virtual sequencer which can be used to control all of the CXL.io/cache/mem and ARB-MUX/LPIF sequencers present in the CXL Subsystem.

svt_cxl_subsystem_sequencer

```
                    |
                    |--------> cxl_io_virt_seqr[] (type= svt_pcie_device_virtual_sequencer)
                    |
                    |--------> arb_mux_virt_seqr[] (type= svt_cxl_arb_mux_virtual_sequencer)
                    |
                    |--------> lpif_virt_seqr[]) (type= svt_cxl_lpif_virtual_sequencer)
                    |
                    |-------> subsystem_app_virt_seqr[]  (type= svt_cxl_subsystem_app_virtual_sequencer)
                    |
                    |--------> cache_mem_system_seqr (type= svt_cxl_system_virtual_sequencer)
                                |
                                |----------->  cache_mem_virt_seqr (type= svt_cxl_virtual_sequencer)
```

3.3.1.1 UVM Component for Top Level CXL Sequencers

The list of sequencer is mentioned in this table.

Sequencer Type	Description
svt_cxl_system_virtual_sequencer (cache_mem_system_seqr)	This class is the CXL System sequencer class. This is a virtual sequencer which can be used to control all of the cache mem sequencers that are in the cache mem system
svt_cxl_virtual_sequencer(cache_mem_virt_seqr)	This virtual sequencer can be used to control all of the sequencers that are operating at CXL Transaction and Data Link Layer.
svt_pcie_device_virtual_sequencer(cxl_io_virt_seqr[])	The device agent class has a virtual sequencer object of type `svt_pcie_device_virtual_sequencer` that connects to all sequencers that are defined under its hierarchy.
svt_cxl_arb_mux_virtual_sequencer(arb_mux_virt_seqr[])	This class defines a virtual sequencer that can be connected to the `svt_cxl_arb_mux_agent`.
svt_cxl_lpif_virtual_sequencer(lpif_virt_seqr[])	This class defines a virtual sequencer that can be connected to the svt_cxl_lpif_agent.
svt_cxl_arb_mux_service_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_arb_mux_service_transaction` class object. This service transaction sequencer is used to provide Link Up information to Arb/Mux when PL is not present as a start point for Arb/Mux functionality.

Sequencer Type	Description
svt_cxl_lpif_service_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_lpif_service_transaction` class object. This sequencer helps modelling user specific LPIF operations.
svt_cxl_dl_service_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_dl_service_transaction` class object.
svt_cxl_flit_packer_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_flit_packer_transaction` class object. This Flit packer sequencer is used to provide flit packer transaction with CXL.cache for Request, response and data or CXL.mem for Request/NDR and Request/Response with data as a start point for CXL flit packer when to start its packing during MANUAL Mode. Based on above inputs, flit packer will wait for receiving respective number of request from channel sequencers and starts flit packer operation.
svt_cxl_tl_rsp_sequencer	This class is UVM Sequencer that provides responses for the `svt_cxl_tl_agent` class when configured as a HOST or a DEVICE. The `svt_cxl_tl_agent` class is responsible for connecting this sequencer to the driver if the agent is configured as UVM_ACTIVE.
svt_cxl_tl_req_sequencer	This class is UVM Sequencer that provides stimulus for the `svt_cxl_tl_driver` class. The `svt_cxl_tl_agent` class is responsible for connecting this sequencer to the driver if the agent is configured as UVM_ACTIVE.
svt_cxl_channel_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_channel_transaction` class object. This Channel sequencer is used to mimic three independent channels for CXL.cache for Request, response and data and two independent channels for CXL.mem for Request/NDR and Request/Response with data. These channel sequencers provides sequence items of `svt_cxl_channel_transaction` type to Data link Layer Driver for further processing.

Refer HTML class reference for details:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/sequencer/class_svt_cxl_subsystem_sequencer.html
```

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/Sequencer_class_list.html
```

For the usage of the sequencers related to CXL.io, you can refer the section 5.4 Sequencers, for CXL.cache.mem sequencers, refer to the section 6.4 Sequencers and for ARB/MUX Sequencers refer to the section 7.4 Sequencers.

CXL Subsystem has the following transaction and service class:

- CXL.io re-use PCIe VIP transaction objects for communication with testbench
  - Transactions -- Correspond to a transaction item that is to be sent across the bus.

- svt_pcie_driver_app_transaction -- A transaction class that models PCIExpress application level packets which will be translated into TLPs. Supports all the transaction types supported by the Driver.

- Services -- Commands to give the model that are related to behavior or configuration
  - `svt_pcie_device_agent_service` -- A transaction class which supports all the service types supported by the device agent.
  - `svt_pcie_driver_app_service` -- A transaction class which supports all the service requests that can be processed by the Driver application
  - `svt_pcie_requester_app_service` -- A transaction class which supports all the service requests that can be processed by the Requester application
  - `svt_pcie_target_app_service` -- A transaction class which supports all the service requests that can be processed by the Target application
  - `svt_pcie_io_target_service` -- A transaction class which supports all the service requests that can be processed by the IO Target application
  - `svt_pcie_mem_target_service` -- A transaction class which supports all the service requests that can be processed by the Memory Target application
  - `svt_pcie_cfg_database_service` -- A transaction class which supports all the service requests that can be processed by the Cfg Database application
  - `svt_pcie_global_shadow_service` -- A transaction class which supports all the service requests that can be processed by the Global Shadow Memory application
  - `svt_pcie_tl_service` -- A transaction class which supports all the service requests that can be processed by the Transaction layer
  - `svt_pcie_dl_service` -- A transaction class which supports all the service requests that can be processed by the Data Link layer
  - `svt_pcie_pl_service` -- A transaction class which supports all the service requests that can be processed by the PHY layer

You can refer the PCIe VIP Class reference guide for details.

- CXL mem/cache supports the following communication interface with user testbench
  - svt_cxl_transaction - A transaction class for CXL Transaction layer.
  - svt_cxl_channel_transaction - A transaction class for CXL Link layer.
- svt_cxl_flit - A transaction class for communication with Link Layer to ARB-MUX Layer.

3.3.2 Sequence APIs

Many APIs are available to ease and accelerate the development. You can refer the sequence class `svt_cxl_subsystem_api_collection_sequence` for complete details on the available sequence APIs.

Some of the available APIs are:

a. `activate_cxl_io_link()` - Method to activate PL and DL link up of CXL.io. This is a blocking method which waits till link is up.

b. `generate_cxl_io_mem_wr()` - Method to initiate CXL.io Mem.Wr driver app transactions.

c. `generate_cxl_io_mem_rd()` - Method to initiate CXL.io Mem.Rd driver app transactions. It is a blocking method and returns payload after successful operation.

d. `generate_cxl_tl_rand_mem_traffic()` - Method to initiate Random CXL.mem RwD and Req transactions from the Transaction Layer.

e. `generate_cxl_tl_rand_cache_mem_traffic()` - Method to initiate Random CXL.cache/mem transactions from the Transaction Layer based on the device type.

f. `generate_cxl_pm_vdm()` - Method to initiate CXL PM VDM transactions.

g. `pm_credit_initialization()` - Method to perform Credit and PM initialization.

h. `perform_warm_reset()` - Method to perform Warm Reset followed by Hot.Reset entry

i. `enumerate_cxl_device()` - API to perform the enumeration for CXL device DUT.

j. `suspend_cxl_tl_tx_traffic()` - API to suspend new traffic processing from the sequencer of component in TX direction.

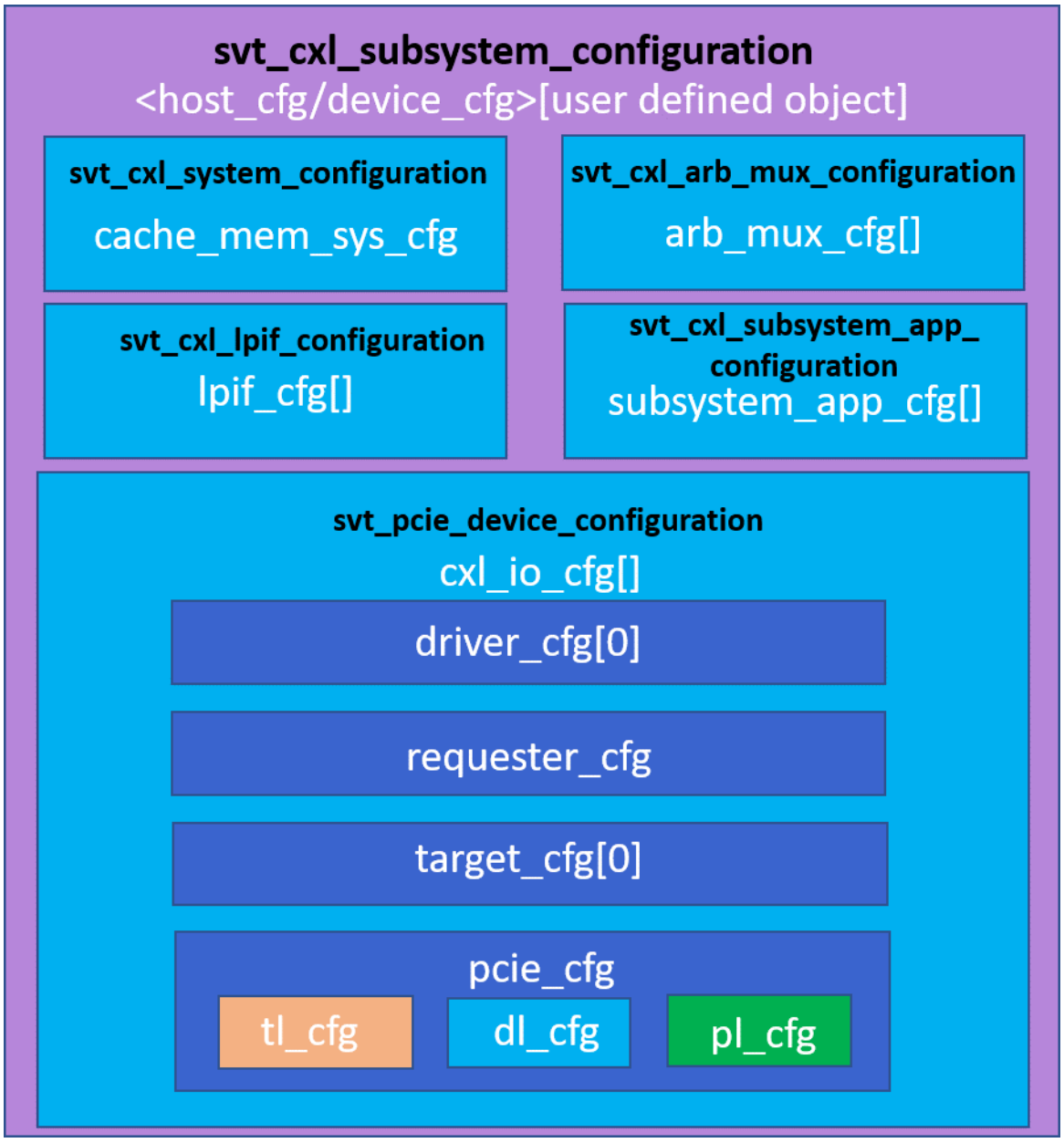
k. `resume_cxl_tl_tx_traffic()` - API to resume the new traffic processing from the sequencer of component in TX direction.

You can refer the VC Verification IP for CXL Subsystem class reference available at: *$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html*

3.4 Configuration Data Objects

3.4.1 CXL Subsystem Configuration Data Objects

This illustration shows the inheritance diagram for all the configuration objects.



Configuration data objects are abstracted data objects that represent the content of CXL VIP configuration data and protocol transactions. The top-level configuration data objects are:

`svt_cxl_subsystem_link_configuration` is used to encapsulate host and device configuration information.

`svt_cxl_subsystem_configuration`: CXL Subsystem Configuration data and methods used to configure CXL Subsystem. It encapsulates CXL cache and mem configurations, ARB-MUX configurations and CXL io agent configurations.

svt_cxl_subsystem_link_configuration(cust_cfg)
|
|------>svt_cxl_subsystem_configuration(host_cfg/device_cfg)
|
|--------> cxl_io_cfg[] (type= svt_pcie_device_configuration)
|
|--------> arb_mux_cfg[] (type= svt_cxl_arb_mux_configuration)
|
|--------> lpif_cfg[] (type= svt_cxl_lpif_configuration)
|
|--------> subsystem_app_cfg[] (type= svt_cxl_subsystem_app_configuration)
|
|--------> cache_mem_sys_cfg (type= svt_cxl_system_configuration)
|
|----------> host_cfg/device_cfg (type= svt_cxl_cache_mem_configuration)

UVM top level configuration objects:

- `svt_cxl_system_configuration(cache_mem_sys_cfg)`: CXL System Configuration
- `svt_cxl_arb_mux_configuration (arb_mux_cfg[])`: This class contains CXL ARB/MUX agent Configuration.
- `svt_cxl_lpif_configuration(lpif_cfg[])`: This class contains CXL LPIF Configuration
- `svt_cxl_cache_mem_configuration[host_cfg/device_cfg)`: This class contains CXL cache mem agent Configuration. This class acts as a base class for Host and Device agent configuration, and contains fields required by both the Host and Device agent configurations.
- `svt_pcie_device_configuration(cxl_io_cfg[])`: The PCIe Agent is configured using an object of class type `svt_pcie_configuration`. An object of this type with an instance name of `pcie_cfg` is defined in `svt_pcie_device_configuration` class. This class is comprised of data members and class object members. The object members represent the configuration of sub-components of the PCIe Agent class.

Refer HTML Class Reference for more details:

`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/Configuration_class_list.html`

Usage note for creating instance of `svt_cxl_subsystem_link_configuration`.

```
svt_cxl_subsystem_link_configuration   cust_cfg(user_cfg)
```

Note : `<user_cfg>` is the configuration attribute handle name which is given by user.

```
Class user_base_test extends svt_cxl_subsystem_base_test;
/**
  * Instance of CXL Subsytem native system environment
   */
  svt_cxl_subsystem_env env;
/**
   * Instance of CXL Subsytem native system configuration
   */
  svt_cxl_subsystem_link_configuration   cust_cfg;
virtual function void build_phase( uvm_phase phase );
/** Create the configuration object required for this test */ This create object of CXL
VIP configuration
    cust_cfg = svt_cxl_subsystem_link_configuration ::type_id::create("cust_cfg");
endfunction
endclass
```

Accessing CXL.io or cache/mem Configuration

These can be accessed through below hierarchy.

```
cust_cfg.host_cfg.cxl_io_cfg[i].apn_mode != svt_pcie_types::APN_PCIE_TLP_DLLP_ONLY
cust_cfg.device_cfg.cache_mem_sys_cfg.device_cfg[i].device_type
=svt_cxl_cache_mem_configuration::TYPE2_DEVICE;
```

cache_mem_sys_cfg -> is a common handle to cache and mem blocks of VIP. This can accessed as explained above in a hierarchical manner. For more details, refer to respective chapter 6.2 Configurations cachemem section.

Example of configuration attributes usage in CXL subsystem

The following illustration shows the inheritance diagram for the `svt_pcie_device_configuration` configuration objects. CXL subsystem can be programmed as a Host or a Device.
`svt_cxl_subsystem_configuration::set_subsystem_type(subsystem_type_enum subsystem_type)`

- Setup CXL subsystem as CXL.io, Cache.mem with ARB MUX, Cache.mem & PCIE Only.

  svt_cxl_subsystem_configuration::configure_subsystem(subsystem_type_enum);
- Set existing or user-created PCIe VIP configuration handle

  svt_cxl_subsystem_configuration::set_cxl_io_cfgs(svt_pcie_device_configuration)
- Enable/Setup APN negotiation of PCIe VIP. It also sets APN capabilities for Flex bus support.

  svt_cxl_subsystem_configuration::configure_flex_bus(common_clk, sync_header_bypass, retimer_1_aware, retimer_2_aware, pcie_cxl_capables , vendor_id);

Usage Notes for Flex bus configuration:

- Flexbus Layer must be configured to support Gen5 rate and Specification version should be 5.0 or above. Refer below:

  ```
  cfg.cxl_io_cfg[i].pcie_spec_ver = svt_pcie_device_configuration::PCIE_SPEC_VER_5_0;
  cfg.cxl_io_cfg[i].pcie_cfg.pl_cfg.set_link_speed_values(`SVT_PCIE_SPEED_32_0G |
  `SVT_PCIE_SPEED_16_0G | `SVT_PCIE_SPEED_8_0G | `SVT_PCIE_SPEED_5_0G |
  `SVT_PCIE_SPEED_2_5G);
  ```

- To peform APN negotiation in CXL mode, "configure_flex_bus" API can be used directly after the specification version is set using "set_spec_ver" API. This allows to configure different settings of flexbus

```
/** Configure PCIe APN capabilities for Flex bus operation for both host and device flex
bus */
    cust_cfg.host_cfg.configure_flex_bus(0,0,0,0);
    cust_cfg.device_cfg.configure_flex_bus(0,0,0,0);
OR
    cust_cfg.host_cfg.configure_flex_bus(1,1,0,0);// common_clk enabled with
sync_header_bypass feature
    cust_cfg.device_cfg.configure_flex_bus(1,1,0,0);// common_clk enabled with
sync_header_bypass feature
```

To enable and wait for link up of flexbus, `wait_for_flexbus_linkup` API can be used available inside `svt_cxl_subsystem_virtual_api_collection_sequence`.

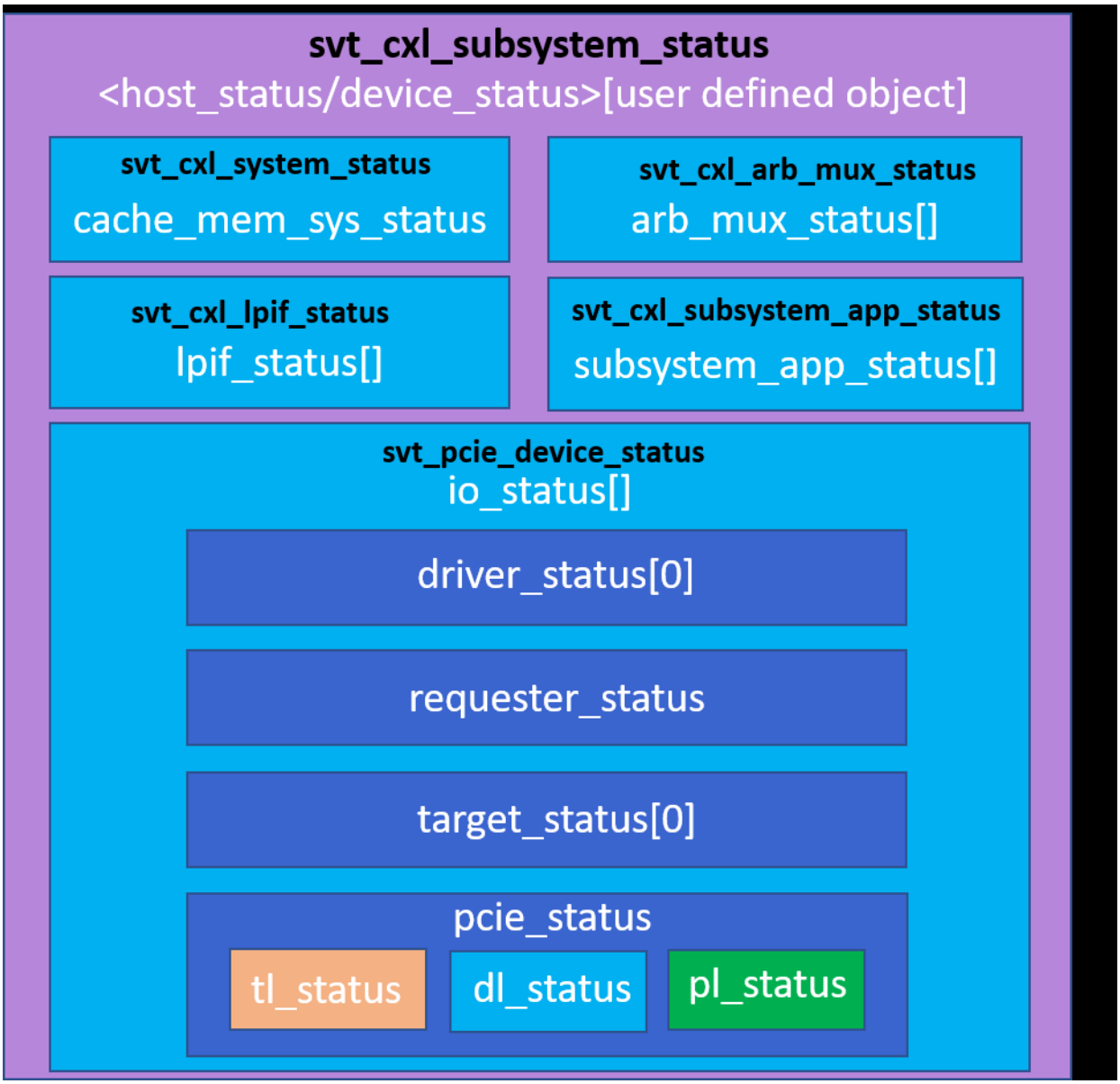
For the usage of the configurations related to CXL.io, you can refer the section 5.2 Configuration, for CXL.cache.mem configurations, refer to the section 6.2 Configurations and for ARB/MUX configurations refer to the section 7.2 Configurations.

3.5 Status Data Objects

3.5.1 CXL Subsystem Status Data Objects

Status data objects are abstracted data objects that represent the content of CXL Subsystem VIP statistics. Registered status objects are updated in real time. Separate statistics are kept per layer/application. Status objects are useful for:

- functional coverage
- reporting testcase progress
- debug



The top-level status data objects are:

svt_cxl_subsystem_status(host_status/device_status) :This is the CXL Subsystem 'top level' status class. It encapsulates the IO status, Cache/mem system status and ARB/MUX status class objects corresponding to the CXL.io, Cache/mem and ARB-MUX agents present in the CXL Subsystem.

```
|------>svt_cxl_subsystem_status(host_status/device_status)
                |
                |--------> io_status[](type= svt_pcie_device_status)
                |
                |--------> arb_mux_status[] (type= svt_cxl_arb_mux_status)
                |
```

|--------> lpif_status[](type= svt_cxl_lpif_status)

|

| --------> subsystem_app_status[](type= svt_cxl_subsystem_app_status)

|

|----------> cache_mem_sys_status (type= svt_cxl_system_status)

UVM component for top level status attribute:

- svt_cxl_system_status(cache_mem_sys_status) :This is the CXL System status class that contains an array of CXL.cache/mem status handles corresponding to each CXL agent
- svt_cxl_arb_mux_status(arb_mux_status[]) :This is the CXL ARB_MUX status class used by arb_mux members.
- svt_cxl_lpif_status(lpif_status[]): This is the CXL LPIF status class
- svt_cxl_tl_status(tl_status):This class contains status information regarding Transaction layer of Host, Device agents.
- svt_cxl_status:This is the CXL VIP 'top level' status class used by Host, Device agents.
- svt_cxl_dl_status(dl_status): This class contains status information regarding a CXL Link Layer Host, Device agents.
- svt_pcie_device_status(io_status[]): The PCIe Agent has a set of state values representing the status of its constituent blocks at any given time in the test simulation. And these state values are encapsulated within the `svt_pcie_status` class. The `svt_pcie_device_status` class has an object of type `svt_pcie_status` instanced as `pcie_status`.

Refer HTML Class Reference for more details:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/Status_class_list.html`

An example of status attribute:

```
/** Instances of Subsystem status */
  svt_cxl_subsystem_status host_status, device_status;
/** Create status objects for Host and Device subsystems */
    host_status = svt_cxl_subsystem_status::type_id::create("host_status");
    device_status = svt_cxl_subsystem_status::type_id::create("device_status");
```

a.

```
if(device_status.io_status[i].pcie_status.pl_status.alternate_protocol_negotiation_status != svt_pcie_pl_status::APN_SUCCESSFUL) begin
            //statement
            end
```

b.

```
if(device_status.io_status[i].pcie_status.pl_status.flexbus_enabled) begin
             //statement
             end
```

c.

```
wait((vip_status.cache_mem_sys_status.host_cache_mem_status[link_num].dl_status.rx_cache_rsp_credits== 0)
```

Refer to section 6.3 status for details of cache_mem status attribute.

3.6 CXL Subsystem Callbacks

Callbacks provide a means to examine or modify transactions at various points in the protocol stack. This chapter describes their basic usage, provides some examples, and gives tips for debugging them. Within a transaction, you can use the transaction handle to:

- Understand where in the protocol layers a particular transaction is being processed.
- Examine a particular field that was added to a transaction (for example, the Link Layer Sequence Number).

The top level callbacks are

- svt_cxl_arb_mux_callback: CXL ARB/MUX callback class defines the component callback methods
- svt_cxl_io_lpif_callback: CXL IO LPIF callback class defines the component callback methods
- svt_cxl_cache_mem_lpif_callback: CXL Cache/Mem LPIF callback class defines the component callback methods
- svt_cxl_dl_monitor_callback: CXL DL Monitor callback class defines the component callback methods
- svt_cxl_dl_callback: CXL Data Link Layer callback class defines the component callback methods
- svt_cxl_tl_callback: CXL Transaction layer callback class defines the component callback methods

Refer HTML Class Reference for more details:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/Callback_class_list.html
```

3.7 Testbench Interface

DUT Interface

The CXL Subsystem can be connected to the DUT at any of the following signaling interfaces:

- LPIF/PCIe PIPE LPC/SerDes Arch
- PCIe Serial signaling interface

Refer HTML Class reference for more details:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/interfaces.html
```

Note:

Refer the CXL Subsystem Host DUT/ Device DUT Integration Guide for more details.

DUT Integration CXL Subsystem

You can use Link Configuration (`svt_cxl_subsystem_link_configuration`) shipped with CXL Subsystem which instantiate both Host and Device Subsystem Configuration and provide APIs for easy use model for DUT integration.

The APIs available in Link Configuration for DUT integration are:

- `set_dut_type`: API to setup DUT type and DUT Side. For example, the following reference configuration considers VIP as DUT and Host side as DUT.

```
set_dut_type(`SVT_CXL_SUBSYSTEM_TEST_CONFIGURATION_TYPE::VIP_DUT,svt_cxl_subsystem_configuration::HOST_SUBSYSTEM)
```

- `set_cxl_subsytem_env_host_cfg`: API to setup Host side of Subsystem. For example, the following configuration does the setup on Host side in full stack topology.

```
cust_cfg.set_cxl_subsystem_env_host_cfg(0,svt_cxl_subsystem_configuration::IO_TL_DL_ONLY_STACK);
```

- `set_cxl_subsytem_env_device_cfg`: API to setup the Device side of Subsystem. For example, the following configuration does the setup on device side for full stack topology.

```
cust_cfg.set_cxl_subsystem_env_device_cfg(1,svt_cxl_subsystem_configuration::IO_TL_DL_ONLY_STACK);
```

Internally these APIs reuse the Subsystem APIs to setup the Host side.

For example: `set_subsystem_type` and `configure_subsystem` APIs.

Refer the CXL Class reference guide for details of these APIs.

Note In the case of CXL.io enabled topologies, you need to take care of the VIP instance.

3.8 Available Protocol Checks

Different protocol checks are present for CXL.io/cache/mem, LPIF signaling and ARB/MUX. The complete list is present in the following location:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/protocolChecks.html
```

Usage Notes on LPIF Protocol Checks:

You can view the LPIF specific protocol checks that are supported in the CXL subsystem VIP. You can find the complete list in HTML class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/protocolChecks.html?col_0=LPIF_CHECKS#item_cxl_subsystem_svt
```

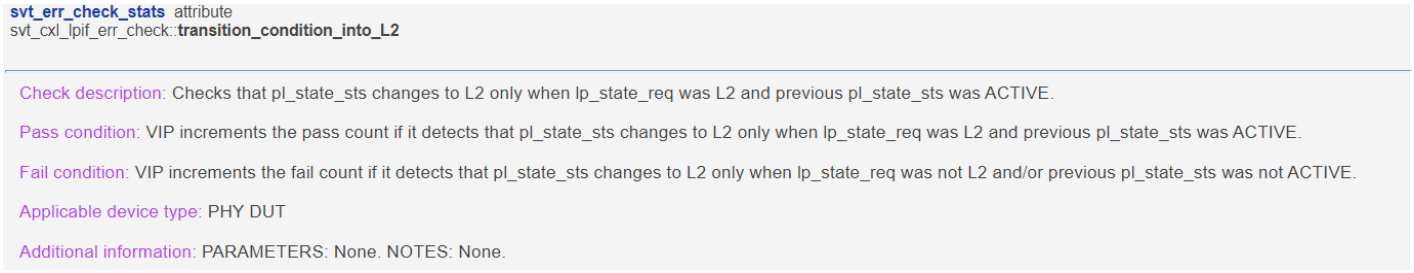
Here is a glimpse of protocol check table. You can select the Group as LPIF to see all the available checks. For specific features, you can selec the Sub Group. For example, here selected sub group is "Active State Rules". You can see all the available checks under this category. The details of every check are present in Reference and Description column.

LPIF_CHECKS	Active State Rules			
Group	Sub Group	Protocol Check Instance name	Reference	Description
LPIF_CHECKS	Active State Rules	transition_condition_into_L2	LPIF v1.0: 5.4, LPIF v1.1: 1.11 Active State Rules	The physical layer will transition to L2 based on observing lp_state_req == L2 while in the ACTIVE state.
LPIF_CHECKS	Active State Rules	transition_condition_into_L1x	LPIF v1.0: 5.4, LPIF v1.1: 1.11 Active State Rules	The physical layer will transition to L1.x based on observing lp_state_req == L1.x while in the ACTIVE state.
LPIF_CHECKS	Active State Rules	transition_condition_into_retrain_state	LPIF v1.0: 5.4, LPIF v1.1: 1.11 Active State Rules	The physical layer will transition to the RETRAIN state upon observing lp_state_req == RETRAIN or due to an internal request to retrain the link while pl_state_sts == ACTIVE.

Also you can select a particular instance name and the hyperlink will take you to the instance description. You will find below fields there:

- Check Description - Describes which signal/cfg VIP checks to trigger the protocol checker
- Pass condition - How should the signal/cfg behave to pass the checker
- Fail condition - How should the signal/cfg behave to fail the checker
- Applicable device type - DUT type for which the checker is valid
- Additional information - Any VIP related parameters which can trigger the checker.

For example, refer below snippet:



3.9 Dynamic Configuration

The configurations set during the build phase can be dynamically modified during the run phase if the test requires a change in the configuration of the Subsystem VIP. The VIP provides `reconfiguration()` APIs which can be used.

1. reconfiguration() method inside `svt_cxl_subsystem_sequencer`:
   a. Test Sequences can invoke this in body() after modification of any layer configuration attributes.

      For example:

      ```
      svt_configuration cfg;
      p_sequencer.host_seqr.get_cfg(cfg);
      void'($cast(link_cfg.host_cfg, cfg.clone()));
      <do necessary updates to the link_cfg.host_cfg object>
      p_sequencer.host_seqr.reconfigure(link_cfg.host_cfg);
      ```

2. reconfiguration() method inside `svt_cxl_subsystem_env`:
   a. Tests can use this in their run phase after modification of any layer configuration attributes.

      Example:

      ```
      svt_configuration cfg;
      env.host_env.get_cfg(cfg);
      void'($cast(cust_cfg.host_cfg, cfg.clone()));
      <do necessary updates to the cust_cfg.host_cfg object>
      env.host_env.reconfigure(cust_cfg.host_cfg);
      ```

The targeted modification can be on a specific layer of the Subsystem VIP or across multiple layers depending on what the test is trying to achieve.

It is must to get the cfg (using get_cfg) and clone it first before applying the changes using reconfiguration.

3.10 CXL Subsystem Configuration (svt_cxl_subsystem_configuration) APIs

CXL subsystem can be programmed as a Host or a Device.

`svt_cxl_subsystem_configuration::set_subsystem_type(subsystem_type_enum subsystem_type)`

For example, `set_subsystem_type`(svt_cxl_subsystem_configuration::HOST_SUBSYSTEM);

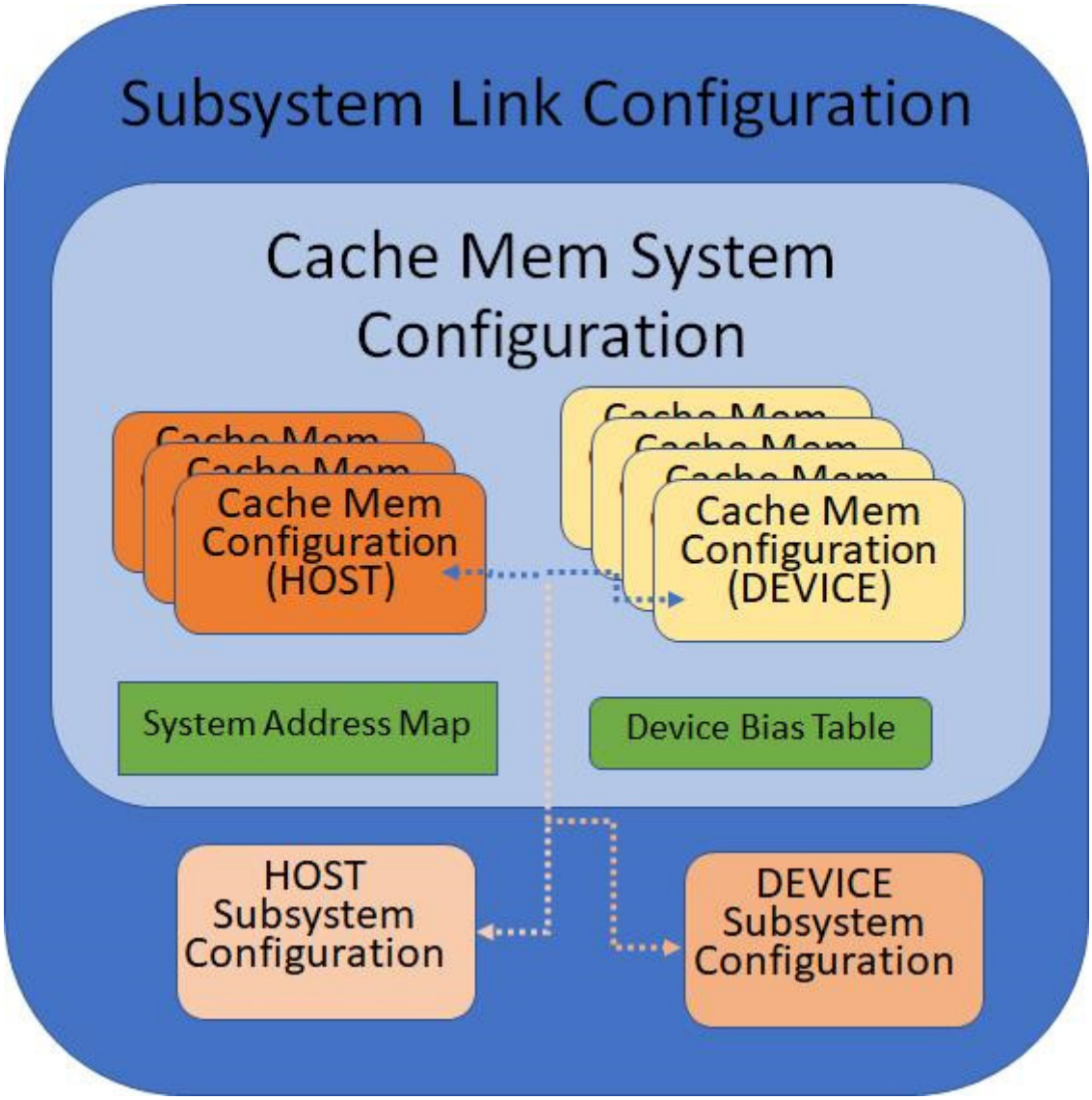
- Setup CXL subsystem as CXL.io, Cache.mem with ARB MUX, Cache.mem & PCIE Only.
  `svt_cxl_subsystem_configuration::configure_subsystem(subsystem_type_enum);`
- Enables/Setup APN negotiation of PCIe VIP. It also sets APN capabilities for Flex bus support.
  `svt_cxl_subsystem_configuration::configure_flexbus(<common_clk` bit, `sync_header_bypass, retimer_1_aware, retimer_2_aware, pcie_cxl_capables)`
  - common_clk : Specifies if common clock mode is supported or otherwise.
  - sync_header_bypass : Specifies standard or low latency mode of operation.
  - retimer_1_aware ; Specifies Re-timer 1 aware capability.
  - retimer_2_aware ; Specifies Re-timer 2 aware capability.
  - pcie_cxl_capables : Specifies the override values for `pcie_capable`, `cxl_io_capable`, `cxl_mem_capable`, and `cxl_cache_capable`.
- Set existing or user-created PCIe VIP configuration handle
  `svt_cxl_subsystem_configuration::set_cxl_io_cfgs(svt_pcie_device_configuration)`

3.11 Configuring CXL System Address Map

System Address Map specifies the address ranges dedicated for each CXL component, such as Host or Device, accessible to the entire CXL system. This means that the address ranges are global parameters and not local properties of any component, sub-component, or a cluster of components. All components recognize and honor the address ranges specified in the `svt_cxl_system_confiuration`. However, at present only `cxl cache_mem` transaction layer host and device components are using system address map to check correct transaction routing.

You must follow these steps to apply effective System Address Map:

1. `svt_cxl_system_configuration` is instantiated within the `svt_cxl_subsystem_link_configuration`. This means that `cache_mem_sys_cfg` is constructed within the link configuration. The same `svt_cxl_system_configuration` object handle `cache_mem_sys_cfg` is also provided to the host or device specific subsystem configurations. Refer the diagram and code example for same.

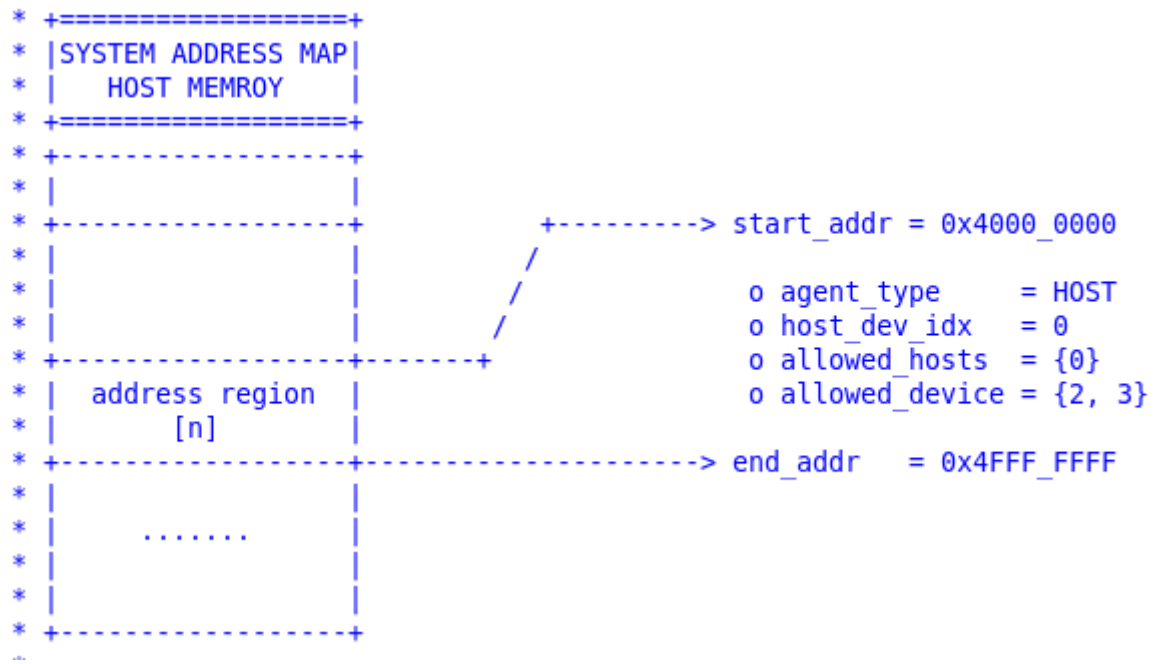



```
function svt_cxl_subsystem_link_configuration::new(vmm_log log = null);
    static vmm_log shared_log = new("svt_cxl_subsystem_link_configuration", "class");
    super.new((log == null) ? shared_log : log,
svt_cxl_subsystem_suite_spec_override::get_suite_spec());
`else
  function new(string name = "svt_cxl_subsystem_link_configuration");
    super.new(name, svt_cxl_subsystem_suite_spec_override::get_suite_spec());
`endif
`ifdef SVT_VMM_TECHNOLOGY
  cache_mem_sys_cfg  = svt_cxl_system_configuration::create_instance(this,
"cache_mem_sys_cfg", `__FILE__, `__LINE__);
  cache_mem_sys_cfg.log.set_instance($sformatf("%s.cache_mem_sys_cfg",
log.get_instance()));
  this.log.is_above(cache_mem_sys_cfg.log);
`else
```

```
  cache_mem_sys_cfg  =
svt_cxl_system_configuration::type_id::create("cache_mem_sys_cfg");
`endif
`ifdef SVT_VMM_TECHNOLOGY
  host_cfg  = svt_cxl_subsystem_configuration::create_instance(this, "host_cfg",
`__FILE__, `__LINE__);
  host_cfg.log.set_instance($sformatf("%s.host_cfg", log.get_instance()));
  this.log.is_above(host_cfg.log);
`else
  host_cfg  = svt_cxl_subsystem_configuration::type_id::create("host_cfg");
`endif
  host_cfg.cache_mem_sys_cfg = this.cache_mem_sys_cfg;
`ifdef SVT_VMM_TECHNOLOGY
  device_cfg  = svt_cxl_subsystem_configuration::create_instance(this, "device_cfg",
`__FILE__, `__LINE__);
  device_cfg.log.set_instance($sformatf("%s.device_cfg", log.get_instance()));
  this.log.is_above(device_cfg.log);
`else
  device_cfg  = svt_cxl_subsystem_configuration::type_id::create("device_cfg");
`endif
  device_cfg.cache_mem_sys_cfg = this.cache_mem_sys_cfg;
  endfunction: new
```

1. Specify the address ranges for each host and device `cxl cache_mem` agents by calling `set_addr_range(<host/device>,<host/device agent index>,<start_addr>, <end_addr>,<host_indices_can_access_addr>,<device_indices_can_access_addr>`.

   The API sets the address range for a specified Host or Device agent and this address range specifically refers to the memory attached to the corresponding Host or Device agent. By specifying distinct address ranges for each Host and Device, it is possible to query the Host or Device agent memory to which a specific address belongs.



Set the address range as below,

```
function void svt_cxl_subsystem_base_test::create_cfg();
  `svt_note("create_cfg", "Entering...");
.
.
.

    if(configure_system_address_map) begin
      // configure system address ranges for each HOST and DEVICE component for Register
and Memory access

cust_cfg.cache_mem_sys_cfg.set_addr_range(svt_cxl_cache_mem_configuration::DEVICE,-1,
'h0, 'h3fff);     //register space

cust_cfg.cache_mem_sys_cfg.set_addr_range(svt_cxl_cache_mem_configuration::DEVICE, 0,
'h8000, 'hffff);  //memory
       cust_cfg.cache_mem_sys_cfg.set_addr_range(svt_cxl_cache_mem_configuration::HOST,
-1, 'h4000, 'h7fff);  //register space
      cust_cfg.cache_mem_sys_cfg.set_addr_range(svt_cxl_cache_mem_configuration::HOST,
0, 'h10000, 'h1ffff);//memory
      end
    end
    create_cfg_called = 1;
  end

`svt_note("create_cfg", "Exiting...");
Endfunction
```

The register spaces are indicated by setting host/device agent index as -1. This ensures that no host or device requests are expected to these address ranges.

2. CXL Cache_Mem request generating sequences must also be updated, so that it generates traffic targeting to the valid address ranges. For example, the device that generates transactions to Host or Device address ranges. Otherwise, CXL VIP reports as routing error. An example sequence is given below with system address range specific assignments and constraints.

```
task svt_cxl_tl_virtual_cache_mem_sequence::body();
< variable declarations... >
raise_phase_objection();
.................
fork
// Cache Request
begin //{
  int addr_range_index;
  foreach(cache_req_xact[i]) begin

    if(cache_req_xact[i] == null) begin

      `svt_xvm_create_on(cache_req_xact[i], p_sequencer.tl_req_seqr);
      cache_req_xact[i].cfg = this.cfg;
      addr_range_index = cfg.sys_cfg.get_rand_addr_range_index(~is_host);
      rand_status = cache_req_xact[i].randomize() with {xact_protocol ==
      ((cfg.agent_type == svt_cxl_cache_mem_configuration::HOST) ?
      svt_cxl_common_transaction::CACHE_H2D_XACT:
      svt_cxl_common_transaction::CACHE_D2H_XACT);

      if(addr_range_index >= 0) {
```

```
        addr >= cfg.sys_cfg.addr_ranges[addr_range_index].start_addr;
        addr <= cfg.sys_cfg.addr_ranges[addr_range_index].end_addr;
      } else {
        addr inside {0,64,128,256};
      }

      if(cfg.agent_type == svt_cxl_cache_mem_configuration::HOST){
        xact_type dist { svt_cxl_transaction::H2D_SNPDATA := snpdata_wt,
        ..............
        svt_cxl_transaction::D2H_CACHEFLUSHED := cacheflushed_wt
      };}
      };

    End

    `svt_xvm_send(cache_req_xact[i]);
    track_responses(cache_req_xact[i]);
    track_data(cache_req_xact[i]);
  end
end

//Mem Req
begin
  int addr_range_index;
  foreach(mem_req_xact[i]) begin
    if (cfg.agent_type == svt_cxl_cache_mem_configuration::HOST) begin
      if(mem_req_xact[i] == null) begin
        `svt_xvm_create_on(mem_req_xact[i], p_sequencer.tl_req_seqr);
        mem_req_xact[i].cfg = this.cfg;
        addr_range_index = cfg.sys_cfg.get_rand_addr_range_index(0);
        //TODO open the xact_type
        rand_status = mem_req_xact[i].randomize() with {xact_protocol ==
        svt_cxl_common_transaction::MEM_REQ_XACT;
        if(addr_range_index >= 0) {
          addr >= cfg.sys_cfg.addr_ranges[addr_range_index].start_addr;
          addr <= cfg.sys_cfg.addr_ranges[addr_range_index].end_addr;
        } else {
          addr inside {0,64,128,256};
        }
      end
      `svt_xvm_send(mem_req_xact[i]);
      track_responses(mem_req_xact[i]);
      track_data(mem_req_xact[i]);
      check_completion_timeout(mem_req_xact[i], 10000);
    end //if(HOST)
  end //foreach(memreq)
end //MemReq
join
fork
  forever get_response(rsp);
join_none
`svt_xvm_debug("body", "Waiting for all responses to be received");
wait (trans_count == received_responses);
`svt_xvm_debug("body", "Received all responses. Dropping objections");
drop_phase_objection();
`svt_note(method_name, "Ending.................");
endtask : body
```

3.12 DUT Interface

The CXL Subsystem can be connected to the DUT at any of the following signaling interfaces:

- LPIF/PCIe LPC/ PIPE SerDes Arch
- PCIe Serial signaling interface

Note Refer the Chapter 'PCIe Verification Topologies' from PCIe VIP User guide for details on DUT integrations.

This chapter provides details of PCIe Unified VIP component. Refer the section 'DUT integration' for DUT integration details.

3.12.1 DUT Integration with CXL Subsystem

You can use Link Configuration (`svt_cxl_subsystem_link_configuration`) shipped with CXL Subsystem which instantiate both Host and Device Subsystem Configuration and provide APIs for easy use model for DUT integration.

The APIs available in Link Configuration for DUT integration are:

- set_dut_type: API to setup DUT type and DUT Side. For example, the following reference configuration considers VIP as DUT and Host side as DUT.

  ```
  set_dut_type(`SVT_CXL_SUBSYSTEM_TEST_CONFIGURATION_TYPE::VIP_DUT,svt_cxl_subsyste
  m_configuration::HOST_SUBSYSTEM);)
  ```

  - `` `SVT_CXL_SUBSYSTEM_SNPS_DUT_TYPE `` can be used to replace the DUT TYPE setting. The default value of define is `VIP_DUT`.

- set_cxl_subsytem_env_host_cfg: API to setup Host side of Subsystem. For example, the following configuration does the setup on Host side in full stack topology.

  ```
  cust_cfg.set_cxl_subsystem_env_host_cfg(0,svt_cxl_subsystem_configuration::IO_TL_
  DL_ONLY_STACK);
  ```

- set_cxl_subsytem_env_device_cfg: API to setup the Device side of Subsystem. For example, the following configuration does the setup on device side for full stack topology.

  ```
  cust_cfg.set_cxl_subsystem_env_device_cfg(1,svt_cxl_subsystem_configuration::IO_T
  L_DL_ONLY_STACK);
  ```

Internally these APIs reuse the Subsystem APIs to setup the Host side.

For example: `set_subsystem_type` and `configure_subsystem` APIs.

Refer the CXL Class reference guide for details of these APIs.

Note
In the case of CXL.io enabled topologies, you need to take care of Unified VIP instance.

3.13 Key Use Cases of Cache Mem Transaction Layer

1. Coherent Traffic Generation and Coherency Checks

   Even though all aspects of coherency are not yet supported, CXL VIP can be used for coherent and memory traffic generation and perform related protocol checks.

- Device and Host both can send coherent requests to allocate or de-allocate a cacheline targeted to the Host attached Memory address range. CXL VIP establishes the required transaction flow and track allocation and de-allocation of the targeted cache lines. It also performs the read and write from the attached memory based on the requirement of the corresponding transaction.

  Example Scenario:

  - Device sends allocating coherent request for Exclusive Ownership
  - Allocates the cacheline once response is received from the Host
  - Performs a store operation and modifies the cacheline data
  - Host at a later point sends Snoop to claim the ownership
  - Device invalidates the line in it's cache and forwards the data to Host
  - Host receives snoop response and allocates the cacheline with snoop data in it's cache
  - Later Host can update Host attached memory with the cacheline data

2. System Address Map based Routing Check

   Host and Devices can be configured with the applicable address ranges as a System Address Map and CXL VIP performs request routing check. If any coherent or memory request is found to be sent to any unmapped address space, then VIP reports an error.

   This feature can be used to verify any unintended request sent by any connected component, which are either Host or Device.

   - Host and Device both can be configured with the same or overlapped address ranges by setting `allow_overlapping_addr` to '1' in `svt_cxl_system_configuration`

3. CacheFlushed Transaction Support

   The CacheFlushed opcode is not specific to a cacheline, it is an indication to the Host that all of the Device's caches are flushed. Thus, the Device must not issue CacheFlushed if there is any cacheline buried in Modified, Exclusive, or Shared state. As the CacheFlushed operation affects the entire cache structure, it blocks the current driver to not accept any further request from sequencer as soon as it is received.

   CacheFlushed transaction Processing:

   - Blocks all new requests from sequencer.
   - Wait for all pending transactions to get completed.
   - Scan through the cache & detect the modified cachelines.
   - Create an auto generated D2H_DIRTYEVICT transactions for modified cachelines, so that modified data will be written back to memory.
   - Wait for all these auto-generated transactions to get completed.
   - Invalidate the entire cache .
   - Proceed with CacheFlushed transaction.
   - Unblock driver to accepts new requests from sequencer.

   However, any snoop request received during this process will appropriately be responded.

3.14 Transaction Logging Support

Transaction Trace logging feature is supported for CXL IO/Cache/mem Protocols of CXL Subsystem SVT for the following layers.

3.14.1 Transaction Layer

Transaction Layer trace creates log files of CXL Cache/mem transactions. Logging enabling attribute would be `svt_cxl_cache_mem_configuration::enable_transaction_logging`.

Example code present in the test to enable the transaction logging for the HOST is shown here:

```
for (int i = 0 ; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_cache_mem_agent; i++)
begin
cust_cfg.host_cfg.cache_mem_sys_cfg.cache_mem_cfg[i].enable_transaction_logging =
1;
end
```

By default, the log gets generated / saved with Hierarchy of the component appended with. xact_log.

Example: uvm_test_top.env.device_env.cache_mem_env.cache_mem[0].tl_driver.xact_log

If you want to print the transaction log with user-defined name, add it as:

```
for (int i = 0 ; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_cache_mem_agent; i++)
begin
cust_cfg.host_cfg.cache_mem_sys_cfg.cache_mem_cfg[i].transaction_log_filename=
"debug_log";
end
```

3.14.2 ARB/Mux Layer

ARB/Mux Layer trace creates log files of CXL IO/Cache/mem input transactions coming from DL/PHY layer. Logging enabling attribute would be
`svt_cxl_arb_mux_configuration::enable_transaction_logging`.

Example code present in the test to enable the transaction logging for the HOST is shown here:

```
if(cust_cfg.host_cfg.num_arb_mux != 0) begin
      for (int i = 0 ; i < cust_cfg.host_cfg.num_arb_mux; i++) begin
        cust_cfg.host_cfg.arb_mux_cfg[i].enable_transaction_logging = 1;
      end
    end
```

By default, the log gets generated/ saved with Hierarchy of the component appended with. `xact_log`.

Example:

`uvm_test_top.env.device_env.arb_mux[0].driver.xact_log`

`uvm_test_top.env.host_env.arb_mux[0].driver.xact_log`

If you want to print the transaction log with user-defined name, add it as:

```
if(cust_cfg.host_cfg.num_arb_mux != 0) begin
      for (int i = 0 ; i < cust_cfg.host_cfg.num_arb_mux; i++) begin
        cust_cfg.host_cfg.arb_mux_cfg[i].transaction_log_filename = "debug_log";
      end
    end
```

3.15 Steps to Move from PCIe VIP based Testbench to CXL Subsystem Based Testbench

These are the list of steps/changes based on PCIE VIP Unified Testbench (`tb_pcie_svt_uvm_unified_vip_sys`) to have support for CXL Subsystem, and also these are steps based on single instance usage.

3.15.1 Include the CXL Related Files (Make sure DESIGNWARE_HOME created & set from CXL Subsystem)

3.15.2 Include and Import CXL Subsystem Packages in top

```
// CXL UVM package
`include "svt_cxl.uvm.pkg"
`include "svt_cxl_subsystem.uvm.pkg"
--------

  // Import the CXL VIP Package
  import svt_cxl_uvm_pkg::*;
  // Import the CXL SubSystem VIP
  import svt_cxl_subsystem_uvm_pkg::*;
```

3.15.3 Base Test

- Create CXL Subsystem configuration

```
// CXL Configuration for CXL Host or Device.
  svt_cxl_subsystem_configuration cxl_cfg;
```

- Pass configuration to environment or place from where PCIe VIP agent instance is created.

```
 cxl_cfg = svt_cxl_subsystem_configuration::type_id::create($sformatf("cxl_cfg"),this);
```

You need to set Subsystem Id & Host/Device subsystem for configuration variable `subsystem_type` as HOST_SUBSYSTEM/DEVICE_SUBSYSTEM.

Note
An API to configure the host/device is planned to be offered through which you can do complete configuration without the need for individual configuration.

```
cxl_cfg.subsystem_id =  <Subsystem ID>;
    /** Passing existing PCIe VIP configuration from TB. */
    cxl_cfg.set_cxl_io_cfgs(<PCIe VIP Configuration handle>);
   /** Configuring CXL Subsystem as CXL IO. */
cxl_cfg.configure_subsystem(svt_cxl_subsystem_configuration::PCIE_IO_FULL_STACK);
   /** Setting up CXL subsystem as Host. */
    cxl_cfg.set_subsystem_type(svt_cxl_subsystem_configuration::HOST_SUBSYSTEM);
    /** Configure PCIe APN capabilities for Flex bus operation for flex bus */
   cxl_cfg.configure_flex_bus(0,0,0,0);
   uvm_config_db#(svt_cxl_subsystem_configuration)::set(this, {env_name,"*"}, "cxl_cfg",
cxl_cfg);
```

Note
For complete details on Configuration APIs, you can refer CXL Subsystem Class reference guide.

Class Reference for VC Verification IP for CXL Subsystem is available at:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/index.html`

3.15.4 Environment:

- Capture the configuration from test and pass the configuration to CXL Subsystem

```
    if(uvm_config_db#(svt_cxl_subsystem_configuration)::get(get_parent(), get_name(),
"cxl_cfg", cxl_cfg)) begin
      `uvm_info("build_phase", "Using test case provided cxl_cfg", UVM_LOW)
    end
    else `uvm_fatal("build_phase", "Expecting test case provided cxl_cfg")
```

- Create CXL Subsystem Host/Device ENV and pass the existing PCIe VIP component through uvm_config_db

```
    //Pass configurations to individual component E.g. Host component.
    uvm_config_db#(svt_cxl_subsystem_configuration)::set(this, "cxl_host_env", "cfg",
cxl_cfg);
   // Passing existing PCie VIP Root component to CXL Host.
    uvm_config_db#(svt_pcie_device_agent)::set(this, "cxl_host_env",
"io_agent[0]",root);

    /** Create status objects for Host subsystems */
    host_status = svt_cxl_subsystem_status::type_id::create("host_status");

    /** Pass status objects for Host */
    uvm_config_db#(svt_cxl_subsystem_status)::set(this,  "cxl_host_env",
"shared_subsystem_status",  this.host_status);

   /** Create the Host Env */
    cxl_host_env = svt_cxl_subsystem_env::type_id::create("cxl_host_env", this);
```

- Create the CXL subsystem Sequencer and connect PCIe sequence with CXL sequencers, such as connecting CXL Host sequence with PCIe Sequencer.

```
/** Create Subsystem sequencer for host */
cxl_env_seqr = svt_cxl_subsystem_sequencer::type_id::create("cxl_env_seqr",this);

cxl_env_seqr   = cxl_host_env.subsystem_virt_seqr;
root.virt_seqr = cxl_env_seqr.cxl_io_virt_seqr[0];
```

Note

Avoid using combination of PCIE VIP API '<PCIe VIP Configuration>.pcie_cfg.pl_cfg.set_modified_ts_mode_values' and '<CXL Subsystem configuration>.configure_flex_bus(0,0,0,0);'. This might result in unexpected behavior.

- Subsystem VIP provides `svt_cxl_subsystem_link_configuration`(encapsulates host_cfg and device_cfg of svt_cxl_subsystem_configuration) and `svt_cxl_subsystem_link_status` (encapsulates host_status and device_status of svt_cxl_subsystem_status) to provide easier control over link. You can use these classes instead of creating them individually. See the VIP example areas for more details on usage. Based on link behavior, additional APIs are also part of these classes.

3.16 Enumeration

This section provides the CXL 2.0 Enumeration applicable for devices such as, D1 - CXL 1.1 Device, D2 - CXL 2.0 Device.

3.16.1 Requirements

- CXL Defined Configuration space DVSECs (8.1)
- CXL Register Interface (8.2.8)
- CXL Memory Register (8.2.5)
- CXL Command Interface (8.2.9)
- Component register enumeration & Base address of capabilities.
- CXL 1.1 device Enumeration for CXL 2.0 defined applicable DVSECs & memory mapped capabilities.

3.16.2 Key Updates

CXL 2.0 Device is an Endpoint device, while CXL1.1 was RCiEP. In CXL, the registers are distributed in configuration space and memory space. CXL 2.0 introduced additional DVSECs for CXL1.1 as well as 2.0 device.

CXL 2.0 device memory mapped registers are available in one of the BAR of the device which can be found from Register Locator DVSEC.

Overall, the CXL 2.0 memory space is divided in to three register regions:

- Component Registers
- BAR Virtualization ACL Register Block
- CXL Memory Device Registers

Naming convention in CSR Status class are referenced from CXL Compliance Tests chapter and register names defined in Control and Status Registers.

3.16.3 Use Model

CXL VIP provides an API to perform enumeration on a CXL Device - enumerate_cxl_device(). It enumerates CXL.io resources, RCRB and Membar0 registers. The basic actions performed by the API are:

- CXL.io resources are enumerated through `svt_pcie_device_virtual_ep_enumeration_sequence`.
- Checking for device type correctness.
- Reading PCIe Express Configuration space Register for DVSEC Capability presence

The `enumerate_cxl_device` API is enhanced to enumerate Configuration space & Memory mapped registers for CXL 1.1 as well as 2.0 device. Internally many other APIs (Refer the HTML Class reference `svt_cxl_subsystem_sequence_collection` file) are created which are triggered from the `enumerate_cxl_device` API by default but you can override those if required to do additional operations.

Enumeration status for configuration space and memory space mapped capabilities are available in CSR (Control and Status Register) Status class. CSR Status class `svt_cxl_subsystem_csr_status` is available in CXL Subsystem Status class.

3.16.3.1 CXL 1.1/2.0 Configuration space Registers (DVSECs) (Section 8.1)

CXL2.0 specifies CXL DVSEC mandatory/ optional for CXL Devices. Following table provides an overview (Snippet from CXL spec).

Table 3-1 CXL DVSEC ID Assignment

CXL Capability	DVSEC ID	Highest DVSEC Revision ID	Mandatory[1]	Not Permitted	Optional
PCIe DVSEC for CXL Device (Section 8.1.3)	0	1	D1,D2,LD, FMLD	P,UP1, DP1,R, USP,DSP	
Non-CXL Function Map DVSEC (Section 8.1.4)	2	0		P,UP1, DP1,R, DSP	D1,D2,LD, FMLD,USP
CXL 2.0 Extensions DVSEC for Ports (Section 8.1.5)	3	0	R, USP, DSP	P,D1, D2, LD,FMLD, UP1, DP1	
GPF DVSEC for CXL Ports	4	0	R, DSP	P,D1, D2, LD,FMLD, UP1, DP1, USP	
GPF DVSEC for CXL Devices	5	0	D2, LD	P,UP1, DP1, R, USP, DSP, FMLD	D1
PCIe DVSEC for Flex Bus Port (Section8.1.8)	7	1	D1, D2, LD, FMLD, UP1, DP1, R, USP, DSP	P	
Register Locator for DVSEC (Section8.1.9)	8	0	D2, LD, FMLD, R, USP, DSP	P	D1, UP1, DP1
MLD DVSEC (Section8.1.10)	9	0	FMLD	P, D1, D2, LD, UP1, DP1, R, USP, DSP	
PCIe DVSEC for Test Capability (Section 14.16.1)	0Ah	0	D1	P, LD, FMLD,DP1, UP1, R, USP, DSP	D2

1. P-PCI Express Device, D1 - CXL 1.1 Device, D2-CXL 2.0 Device, LD-Logical Device, FMLD- Fabric Manager Owned LD 0xFFFF, UP1 - CXL 1.1 Upstream Port RCRB, DP1- CXL 1.1 Downstream Port RCRB, R- CXL 2.0 Root Port, USP- CXL Switch Upstream Port, DSP - CXL Switch Downstream Port

This is the table mapping the DVSEC's/Capabilities/Memory Mapped Regions with the CSR Status class attribute:

CXL DVSECs

DVSECs/Capabilty	CSR Status class attributes
PCIe DVSEC for CXL Device	cxl_device_dvsec
Non-CXL Function Map DVSEC	non_cxl_function_dvsec_base
GPF DVSEC for CXL Devices	cxl_gpf_device_dvsec_base
PCIe DVSEC for Flex Bus Port	cxl_flexbus_dvsec_base
Register Locator DVSEC	cxl_register_dvsec_base
CXL DVSEC for Test Capability	cxl_test_capability_dvsec_base
Additional Capabilities	
Memory Device Configuration Space Layout	device_serial_num_ext_capability_base
DOE compliance/ CDAT	• DOE_object_type_addr • cdat_DOE_mailbox_addr • compliance_DOE_mailbox_addr
Register Locator DVSEC Information	
Register BIR	register_bir[]
Register Block Identifier	register_block_identifier[]
Register Block Offset	register_block_offset[]
Memory Mapped Register pointers	
CXL Component Registers	cxl_device_component_register_base_addr
BAR Virtualization ACL Register Block	cxl_device_bar_virtualization_acl_register_block_base_addr
CXL Memory Device Registers	cxl_memory_device_register_base_addr

CSR Status class is available in CXL Subsystem Status class and this can be used at sequence. The following is reference usage:

```
<host status>.csr_status[link_num].cxl_device_dvsec
```

3.16.3.2 CXL Component Registers (Section: 8.2.3)

CXL 2.0 Enumeration, Component registers are reached through the Register locator DVSEC. Register Block Identifier should be "01" and Register BIR points to BAR which capture Component register.

Reference snippet from CXL Specification for Component registers:

Table 141. CXL Subsystem Component Register Ranges

Range	Size	Destination
0000_0000h - 0000_0FFFh	4K	CXL.io registers
0000_1000h - 0000_1FFFh	4K	CXL.cache and CXL.mem registers
0000_2000h - 0000_DFFFh	48K	Implementation specific
0000_E000h - 0000_E3FFh	1K	CXL ARB/MUX registers
0000_E400h - 0000_FFFFh	7K	Reserved

The layout and discovery mechanism of the Component Register is identical for CXL 1.1 Upstream Ports, CXL 1.1 Downstream Ports, CXL 2.0 Ports and CXL 2.0 Host Bridges (CHBCR). Table 141 lists the relevant offset ranges from the Base of the Component Register Block for CXL.io, CXL.cache, CXL.mem, and CXL ARB/MUX registers.

VIP Enumeration API captures the base address of Component registers, and you can add the offset and traverse the specific registers.

Component Register Base address in CSR Status class : `cxl_device_component_register_base_addr`

For example, cxl_device_component_register_base_addr + 1000h will lead to CXL.Cache and CXL.mem Registers.

3.16.3.3 Capability Base address (CXL_Capability_ID Assignment Table 142)

Table 142. CXL_Capability_ID Assignment

Capability	ID	Highest version	Mandatory[1]	Not Permitted	Optional
CXL Capability (Section 8.2.5.1)	1	1	D1, D2, LD, FMLD, UP1, DP1, R, USP, DSP	P	
CXL RAS Capability (Section 8.2.5.9)	2	2	D1, D2, LD, FMLD, UP1, DP1, R, USP, DSP	P	
CXL Security Capability (Section 8.2.5.10)	3	1	DP1	All others	
CXL Link Capability (Section 8.2.5.11)	4	1	D1, D2, LD, FMLD, UP1, DP1, R, USP, DSP	P	
CXL HDM Decoder_Capability (Section 8.2.5.12)	5	1	Type 3 D2, LD, R, USP	All others	Type 2 D2
CXL Extended Security Capability (Section 8.2.5.13)	6	1	R	All others	
CXL IDE Capability(Section 8.2.5.14)	7	1		P, D1, LD, UP1, DP1,	D2, FMLD, R, USP, DSP
CXL Snoop Filter Capability (Section 8.2.5.14.5)	8	1	R	P, D1, D2, LD, FMLD, UP1, USP, DSP	DP1

- cxl_device_comp_register_cap_base_addr[int] => Indexing based on Capability ID

Reference snippet from the VIP execution transcript:

```
-------------------------------------------------------------------------------
                                Capability |    ADDRESS / Content
                    PCIe DVSEC for CXL Device | 3c8
                 PCIe DVSEC for Test Capability | 400
          Component Register Capability ID [2] | 21020
          Component Register Capability ID [4] | 21110
```

3.16.3.4 BAR Virtualization ACL Register Block (Section: 8.2.7)

CXL 2.0 Enumeration, BAR Virtualization ACL registers are reached through Register locator DVSEC. Register Block Identifier must be '02' and Register BIR points to BAR which capture BAR Virtualization ACL register block.

CSR Status class attributes for Register Block:
`cxl_device_bar_virtualization_acl_register_block_base_addr`

3.16.3.5 CXL Memory Device Registers (Section: 8.2.8)

CXL 2.0 Enumeration, CXL Memory Device registers are reached through Register locator DVSEC. Register Block Identifier should be '03' and Register BIR points to BAR which capture CXL Memory Device register.

CSR Status class attributes for Register Block: `cxl_memory_device_register_base_addr`

CXL Registers Interface (Section: 8.2.8)

At the beginning of the CXL device register block is a CXL Device Capabilities Array Register which defines the size of the CXL Device Capabilities Array followed by a list of CXL Device Capability headers. Each header contains an offset to the capability specific register structure from the start of the CXL device register block.

Following is the table describe overall Capability ID mapping.

Capability ID Range	Capability Information
0000h - 3FFFh	Generic CXL Device Capabilities
4000h -7FFFh	Specific Capabilities associated with class code Register in PCIe Header (offset 09h)
8000h - FFFFh	Vendor Specific Capabilities

CSR Status class keep each capability information captured through associative array:
`cxl_device_capability_base_addr[int]`

For example:

cxl_device_capability_base_addr[1] - Device Status Register

cxl_device_capability_base_addr[2] - Primary Mailbox

cxl_device_capability_base_addr[3] - Secondry Mailbox

cxl_device_capability_base_addr[4000] - Memory Device Status Register

3.16.3.6 CXL Command Interface (Section: 8.2)

Primary and Secondary Mailbox Capabilities (Capability ID as 2 and 3) identified through CXL Register Interface Capabilities provides the ability to issue a command to device. You need to follow the CXL Specification defined flow to initiate a command over CXL Command interface.

This table captures the Primary/Secondary mailbox Registers extraction from VIP enumeration:

Primary Mailbox Capability Register	cxl_device_capability_base_addr[2] + 00h
Primary Mailbox Control Register	cxl_device_capability_base_addr[2] + 04h
Primary Mailbox Command Register	cxl_device_capability_base_addr[2] + 08h
Primary Mailbox Status Register	cxl_device_capability_base_addr[2] + 10h
Primary Mailbox Background Command Status Register	cxl_device_capability_base_addr[2] + 18h

Primary Mailbox Payload Register	cxl_device_capability_base_addr[2] + 20h

Similarly, Secondary Mailbox Registers can also be extracted.

3.16.3.7 Use Model for Command Interface

You can initiate MRd/MWr TLP to Primary Mailbox Command Registers as described in specification.

CXL 2.0v0.9 Specification section 8.2.8.4 Reference for Command Registers Interface usage:

The flow for executing a command is described below. The term `caller` represents the entity submitting the command:

1. Caller reads MB Control Register to verify doorbell is clear => MemRd to "Primary Mailbox Control Register".
2. Caller writes Command Register => MemWR to "Primary Mailbox Control Register"
3. Caller writes Command Payload Registers if input payload is non-empty => MemWr to "Primary Mailbox Payload Register"
4. Caller writes MB Control Register to set doorbell => MemWR to "Primary Mailbox Control Register"
5. Caller either polls for doorbell to be clear or waits for interrupt if configured => MemRd to "Primary Mailbox Control Register" or wait for Interrupt.
6. Caller reads MB Status Register to fetch Return code => MemRd to "Primary Mailbox Status Register".
7. If command successful, Caller reads Command Register to get Payload Length => MemRd to "Primary Mailbox Payload Register"
8. If output payload is non-empty, then the host reads Command Payload Registers.

MSI TLP can be captured through the analysis port.

3.16.3.8 Use Model to Control Parameter

The enumeration API use model is used to control parameter of enumeration. For example, bus number, all capabilities enumeration and so on.

You can pass the `svt_pcie_ep_enumeration_pf_control` handle with `enumerate_cxl_device` API to control much more parameters.

You can refer the following code:

```
// Checking required configuration space for CXL device.

if((link_cfg.dut_type != `SVT_CXL_SUBSYSTEM_TEST_CONFIGURATION_TYPE::VIP_DUT) &&
(link_cfg.dut_side == svt_cxl_subsystem_configuration::DEVICE_SUBSYSTEM)) begin


svt_pcie_ep_enumeration_pf_control device_parms;
device_parms=new();
device_parms.max_num_functions_supported=max_num_function;
device_parms.enable_sriov =0;
device_parms.max_num_vfs_per_pf = 32;
```

```
device_parms.bus_number = 8'hD0;
device_parms.device_number=5'b11000;
host_seq.disable_cxl_upstream_membar0_enumeration =
disable_cxl_upstream_membar0_enumeration;
host_seq.disable_cxl_upstream_rcrb_enumeration = disable_cxl_upstream_rcrb_enumeration;
 // Enumeration API for CXL Device.
      host_seq.enumerate_cxl_device(0/*link num*/,max_num_function/*Max Number of
      Function*/,0/*SRIOV*/,32/*Max VFs per PF*/,,device_parms);
//
```

3.16.4 Usage with DWC CXL Controller

- Program Controller for Cache/Mem enable before APN negotiation.

  CXL Controller Databook reference section "CXL Register Module".

- Internally, there is an issue with completion handling with "TD=1", which means that the TLP Digest bit asserted. It is recommended to de-assert for completions.

  If CXL Applications are responsible for generating completion, then you can disable the generation of CPL with TD bit asserted.

3.16.5 Enumeration FAQ

1. How to setup CXL 1.1 RCRB Address?

   Use these configuration attributes to control the RCRB base address and Membar0 programming in RCRB.

   These are available in Subsystem Configuration.

   ```
   /**
   * sets the Upstream port RCRB base address, first Memory access after link
   * initiation will be initiated on this address by host.
   * Note : This address should not lie inside any of the BAR ranges.
   */
   rand bit[63:0] upstream_port_rcrb_base_addr=64'h0000_0000_0001_1000;

   /**
   * sets the Upstream port MEMBAR base address.
   * 64K Memory range is required for MEMBAR.
   * Note : This address should not lie inside any of the BAR ranges.
   */
   rand bit[63:0] upstream_port_membar0_base_addr=64'h0000_0000_0002_0000;
   ```

   Reference Access path:

   ```
   cust_cfg.host_cfg.upstream_port_rcrb_base_addr
   cust_cfg.host_cfg.upstream_port_membar0_base_addr
   ```

2. How to manage multiple instances of EP Enumeration status?

The `cxl_io_enumeration_status_id` attribute is added in `svt_cxl_subsystem_virtual_api_collection_sequence` Sequence. You need to program this attribute and it is be mapped internally with `pcie_root_hierarchy_%0d_ep_enumeration_status`.

3. How to program different BARs for different instances or control of BAR ranges to be programmed for DUT?

   If you need non-overlapping BAR addresses for multiple EP instances, then you must set the following attributes of `device_parms`:

   `enable_incremental_bar_allocation =1` (So that random mode of BAR selection is not enabled)

   Then you need to specify non-overlapping values for following limits for each EP instance:

   `min_non_pref_mem_base_addr, max_non_pref_mem_base_addr, min_io_base_addr, max_io_base_addr, min_pref_mem_base_addr` and `max_pref_mem_base_addr`.

4. How to enable DOE Object type enumeration?

   DOE Object type enumeration can be enabled by controlling the `"disable_cxl_doe_object_type_enumeration"` parameter. By default, this parameter is disabled.

   CSR Status class keeps the supported DOE Object Types capability information captured through associative array:

   ```
   DOE_object_type_addr[<Object Type><Vendor ID>]
   ```

   For example:

   ```
   DOE_object_type_addr[021e98] - CXL_COMPLIANCE_DOE_MAILBOX
   DOE_object_type_addr[001e98] - CXL_CDAT_DOE_MAILBOX
   ```

5. How to access DOE Capabilities?

   - DOE capabilities base addr is stored in `ep_enumeration_status.ext_cap_base_addr[index]` where index is => `{24'h0,8'h DOE Cap Instance number,16'h Cap ID, 16'h Target BDF}`
   - DOE Types enumeration is supported via all DOE capabilities.

     The `"enumerate_cxl_doe_object_types"` API has been enhanced to visit each DOE Capability to discover object protocol types.

6. How to access DVSEC/ VSEC after enumeration?

   To access DVSEC/ VSEC after enumeration, check the EP Enumeration status that gets updated for DVSEC/VSEC storage.

   - Existing PCIe EP enumeration status class attribute is updated for direct differentiation of DVSECs/ VSEC.

     ```
     bit [11:0] ext_cap_base_addr[bit[39:0]]; => bit[11:0]
     ext_cap_base_addr[bit[63:0]];
     ```

   - The accessing/ indexing of all extended capabilities is illustrated as below:

     ```
     DVSEC : <16'h DVSEC VENDOR ID> <16'h DVSEC ID> <16'h23 (DVSEC Capability ID)>
     <16'h BDF>
     VSEC : <16'h0> <16'h VSEC ID> <16'hB (VSEC Capability ID)> <16'h BDF>
     Others : <16'h 0> <16'h 0> <16'h Capability ID> <16'h BDF>
     ```

7. How to wait for enumeration completion?

   The enumeration completion is based on the CSR status attribute, `"enumeration_completed"`.

3.16.6 Debug

3.16.6.1 Transcript

The Enumeration of CSR Status class can be used to get some information about the capabilities enumerated. The following snippet is for CXL1.1 mode enumeration for CXL 2.0 device.

```
UVM_INFO svt_cxl_subsystem_sequence_collection.sv(2710) @ 42180000.00 ps: reporter [enumerate_cxl_device]  Enumerated Capabilities captured in CSR Status are :
-------------------------------------------------------------------------------
                                Capability |    ADDRESS / Content
                     PCIe DVSEC for CXL Device | 3c8
                   PCIe DVSEC for Test Capability | 400
              Component Register Capability ID [2] | 21020
              Component Register Capability ID [4] | 21110
            Component Register Capability ID [127] | 21000
           Component Register Capability ID [8192] | 21000
          Component Register Capability ID [53247] | 21000
               CXL Upstream port Capability ID [0] | 11100
               CXL Upstream port Capability ID [3] | 11148
              CXL Upstream port Capability ID [25] | 11158
              CXL Upstream port Capability ID [37] | 11370
              CXL Upstream port Capability ID [38] | 11188
              CXL Upstream port Capability ID [39] | 111b8
              CXL Upstream port Capability ID [42] | 11200
         CXL Upstream port DVSEC Capability ID [7] | 1138c
-------------------------------------------------------------------------------
```

3.16.6.2 XACT File

To debug the configuration space information through XACT file, you must dump the one Data word of Payload also. This helps to reach directly to places applicable for CXL.

Normally Payload of CFG TLP is byte swapped, and hence it is necessary to be vigilant on Endianness.

Finding CXL DVSECs in XACT File

```
372 u_vip_bfm[1] 35330.000 35331.000 T CfgRd0     0x0000/000   44  0 0    0 0 0  BDF:0x0100 R:0x0f2   0 f      1 (H) 04000001 0000000f 010003c8
373 u_vip_bfm[1] 35340.000 35340.000 T ACK                      43
374 u_vip_bfm[1] 35362.000 35362.000 T UPDATEFC_CPL_VC0                          145 OFF 1055 OFF
375 u_vip_bfm[1] 35381.000 35381.000 R ACK                      44
376 u_vip_bfm[1] 35411.000 35411.000 R UPDATEFC_NP_VC0                           61 OFF   30 OFF
377 u_vip_bfm[1] 35415.000 35415.000 R CplD       0x0000/000   44  0 0    0 0 0  ID:0x0100 Stat:SC  BC:0004    1 (H) 4a000001 01000004 00000000
378                                                                                                              0 (D) 23000140
379 u_vip_bfm[1] 35419.000 35419.000 T CfgRd0     0x0000/000   45  0 0    0 0 0  BDF:0x0100 R:0x0f3   0 f      1 (H) 04000001 0000000f 010003cc
380 u_vip_bfm[1] 35430.000 35430.000 T ACK                      44
381 u_vip_bfm[1] 35446.000 35446.000 T UPDATEFC_CPL_VC0                          146 OFF 1056 OFF
382 u_vip_bfm[1] 35469.000 35469.000 R ACK                      45
383 u_vip_bfm[1] 35499.000 35499.000 R UPDATEFC_NP_VC0                           62 OFF   30 OFF
384 u_vip_bfm[1] 35504.000 35504.000 R CplD       0x0000/000   45  0 0    0 0 0  ID:0x0100 Stat:SC  BC:0004    1 (H) 4a000001 01000004 00000000
385                                                                                                              0 (D) 981e8103
386 u_vip_bfm[1] 35507.000 35508.000 T CfgRd0     0x0000/000   46  0 0    0 0 0  BDF:0x0100 R:0x0f4   0 f      1 (H) 04000001 0000000f 010003d0
387 u_vip_bfm[1] 35518.000 35518.000 T UPDATEFC_CPL_VC0                          147 OFF 1057 OFF
388 u_vip_bfm[1] 35532.000 35532.000 T ACK                      45
389 u_vip_bfm[1] 35558.000 35558.000 R ACK                      46
390 u_vip_bfm[1] 35589.000 35589.000 R UPDATEFC_NP_VC0                           63 OFF   30 OFF
391 u_vip_bfm[1] 35593.000 35593.000 R CplD       0x0000/000   46  0 0    0 0 0  ID:0x0100 Stat:SC  BC:0004    1 (H) 4a000001 01000004 00000000
392                                                                                                              0 (D) 0000aec0
```

- Search for "981e" (Reverse of CXL Device ID IE98h).
  - Line 385 in snippet.
- DVSECs has Capability ID of 23, so just before this 981e TLP it should be DVSEC Capability.

- Line 378 in snippet.

- Next offset read should give us DVSEC ID.
- Line 392 in snippet.

3.17 CXL 2.0 Features

As mentioned in the Licensing section, CXL 2.0 features can be enabled and used when using the license SUB-CXL20-SVT (or)VIP-LIBRARY2019-SVT+SUB-CXL20-EA-SVT.
The spec version must be set to `spec_ver = svt_cxl_types::CXL_VER_2_0`.

3.17.1 GPF Feature

The APIs under `svt_cxl_subsystem_sequence_collection`. Refer the Class reference document "PM VDM API" group. Other CXL PM VDM related APIs are also part of this API.

3.17.1.1 CXL GPF APIs

`perform_cxl_gpf()`: to perform GPF VDM exchange and moves from Phase 1 followed by Phase 2.

`perform_cxl_gpf_phase1()` : to perform GPF VDM exchange for Phase 1.

`perform_cxl_gpf_phase1()` : to perform GPF VDM exchange for Phase 2.

`start_traffic_post_gpf ()`: to start generating traffic post GPF phase 2. By default, VIP will stop generating new traffic post GPF Phase 2 completion.

3.17.1.2 Example Usage

These variables are to be removed and same to be replaced by variables in CXL Subsystem App configuration:

1. perform_cxl_gpf(int link_num = 0,

```
bit[3:0] p1_timeout_base  = 1,
bit[3:0] p1_timeout_scale = 0,
bit[3:0] p2_timeout_base  = 1,
bit[3:0] p2_timeout_scale = 0,
bit powerfail_expected    = 0,
bit disable_caching       = 0);
```

2. perform_cxl_gpf_phase1(int link_num = 0,

```
bit[3:0] p1_timeout_base  = 1,
bit[3:0] p1_timeout_scale = 0,
bit powerfail_expected    = 0,
bit disable_caching       = 0);
```

3. perform_cxl_gpf_phase2(int link_num = 0,

```
bit[3:0] p2_timeout_base   = 1,
bit[3:0] p2_timeout_scale  = 0,
bit p1_ended_with_err      = 0);
```

3.17.1.2.1 Configuration Variables for GPF

The variables are added as part of `svt_cxl_subsystem_app_configuration`, `gpf_p1_timeout_scale_down` and `gpf_p2_timeout_scale_down` to help you to further scale down the GPF phase timeout values. These variables are added as control to reduce the huge timeout values when the `time_base` and `time_scale` variables are resultant of GPF control register access.

3.17.1.3 Known Limitations

- No support claimed for Type 1 / 2 Device involved.

3.17.2 Qos Telemetry

1. These parameters of the `svt_cxl_cache_mem_configuration` class need to be set to the following values to enable this feature:
   - enable_qos_telemetry = 1;
   - spec_ver = svt_cxl_types::CXL_VER_2_0;
2. `svt_cxl_system_configuration` API `set_qos_telemetry_control_parameters`() to control throttling:
   - @param dev_idx Device agent index for which QoS Telemetry address range is to be specified.

     Index for Nth device is specified by (N-1), starting at 0.
   - @param start_addr Start address of the QoS Telemetry address range.

     Currently, CXL VIP considers complete device address range for QoS. Therefore, if it is set to any value, it considers the device start address only.
   - @param end_addr End address of the QoS Telemetry address range

     Currently, CXL VIP considers complete device address range for QoS.

     Therefore, if it is set to any value, it considers the device end address only.
   - @param thrtl_window Throttling th parameter which indicates when to sample the loadmax value for throttling control.

     This argument is used by host VIP only.
   - @param normal_delta_time The wait time which will be increased for moderate load (or) decreased for light load

     This argument is used by host VIP only.
   - @param severe_delta_time The wait time which will be increased for moderate load (or) decreased for light load

     This argument is used by host VIP only.
   - @param max_thrtl_wait_time The maximum wait time which is allowed before sending the transaction from the TL layer.

     This argument is used by host VIP only.

Example

- Host VIP

  Suppose previous transaction was sent without waiting for any delay.

So, wait time for previous transaction (previous_wait_time) was 0ns.

Now,the sampled Loadmax value at host end is `SEVERE_OVERLOAD`.

Hence, the new transaction needs to wait for:

wait_time = severe_delta_time i.e.(previous_wait_time + severe_delta_time) before sending from the TL layer.

Now, suppose the sampled Loadmax value at host end is MODERATE_OVERLOAD.

Therefore, the new transaction needs to wait for

wait_time = (severe_delta_time+normal_delta_time)

i.e.(previous_wait_time + normal_delta_time) before sending from the TL layer.

Now, suppose, the sampled Loadmax value at host end is LIGHT_LOAD.
Therefore, the new transaction needs to wait for

wait_time = (severe_delta_time+normal_delta_time-normal_delta_time) i.e.(previous_wait_time - normal_delta_time) before sending from the TL layer.

- extern function void set_qos_telemetry_control_parameters(int dev_idx, bit [`SVT_CXL_MAX_ADDR_WIDTH-1:0] start_addr, bit [`SVT_CXL_MAX_ADDR_WIDTH-1:0] end_addr, int thrtl_window, int normal_delta_time, int severe_delta_time, int max_thrtl_wait_time);
- svt_cxl_tl_cache_mem_resp_sequence API get_dev_load() to populate the device load value to svt_cxl_transaction object.

  This API calculate the device load based on the pending transactions at device end. This API internally gets pending transactions and based on numbers it decide the device load. You can modify this API to calculate the dev load.

3.17.2.1 Usage Example

Host VIP parameters:

```
cust_cfg.cache_mem_sys_cfg.set_qos_telemetry_control_parameters();
  cust_cfg.cache_mem_sys_cfg.host_cfg[0].enable_qos_telemetry=1;
 cust_cfg.host_cfg.set_spec_ver(svt_cxl_types::CXL_VER_2_0, 0);
Device VIP parameters:
  cust_cfg.cache_mem_sys_cfg.set_qos_telemetry_control_parameters();
  cust_cfg.cache_mem_sys_cfg.device_cfg[0].enable_qos_telemetry=1;
  cust_cfg.device_cfg.set_spec_ver(svt_cxl_types::CXL_VER_2_0, 0);

"cust_cfg" is the handle of svt_cxl_subsystem_link_configuration.
```

The value of svt_cxl_transaction::dev_load can be changed by extending the svt_cxl_tl_cache_mem_resp_sequence and overriding the get_dev_load() task

```
 task  svt_cxl_tl_cache_mem_resp_sequence::get_dev_load (  svt_cxl_transaction xact  )
```

Find the details below:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_class_refere
nce/html/sequences/class_svt_cxl_tl_cache_mem_resp_sequence.html#item_get_dev_load
```

For example:

```
cust_cfg.cache_mem_sys_cfg.set_qos_telemetry_control_parameters(0, 'h8000,
'hffff,50,5,10,300);
```

```
task svt_cxl_tl_cache_mem_resp_sequence::get_dev_load(svt_cxl_transaction xact);
xact.dev_load = svt_cxl_types::SEVERE_OVERLOAD;
endtask
```

Note
This is just an illustration, and you must keep in mind other aspects while changing the dev_load.

Keywords to debug QoS scenarios:

- "get_qos_wait_time"
- "set_qos_telemetry_control_parameters"
- "Throttling "

3.17.2.2 Known Limitations

- Device load is calculated based on internal loading. Currently, it does not consider below methods:
  - Egress Port Backpressure
  - Temporary Throughput Reduction

3.17.3 CXL IDE

3.17.3.1 IDE

- **Prerequisites**: Synopsys DesignWare Cores Cryptography Library

Note Cryptographic algorithms are not shipped as part of VIP deliverable. As a prerequisite, you must have access to **Access to Synopsys Designware Core Cryptographic Software Library** (CSL).

- Cryptography Library version : 4.30a
- CXL IDE feature is supported for VCS simulator only

3.17.3.2 Usage Notes

Compile time requirements to execute with IDE:

Cryptography Library : libellipsys.a

VIP DPI-C Object file : svt_aes.o or svt_aes_dpi.cpp

Reference for building object file : $(GCC) -B/usr/bin -DLinux -Damd64 -Dx86_linux -I./ -I$(VCS_HOME)/include -I$(ELLIPSYS_HOME)/src/headers -DELP_SOURCE -DELPSOFT -DELPAES -DELPDEBUG_TRACE -DELPMEM_QUIET -DELPSTACK_QUIET -DELPTESTING -DELP_SOURCE -DELPSOFT -lellipsys -Wno-deprecated -m64 -fPIC -shared -O3 -o svt_aes_dpi.o -c svt_aes_dpi.cpp

Compile time Macro : SVT_AES_DPI_ENABLE

Cryptography Library Linking requirements:

- Link Ellipsys Library :

```
ELLIPSYS_HOME = <Cryptography Library Install Path>/ellipsys
-CFLAGS -I<ELLIPSYS_HOME>
```

- Link Header file : -CFLAGS -I<ELLIPSYS_HOME>/src/header

Configuration attributes or APIs:

- Use `configure_cxl_ide` API of `svt_cxl_cache_mem_configuration` to configure VIP for IDE.

  ```
  <LinkConfiguration>.cache_mem_sys_cfg.<host/device>_cfg[i].configure_cxl_ide(svt_
  cxl_types::CONTAINMENT, host_tx_key, device_tx_key);
  ```

All the CXL IDE configuration attributes are listed under `cxl_dl_ide_members` group.

Note

- Default Endianness is updated to `"BIG Endian"`.
- AES-GCM Debug control is passed via Cache-Mem configuration class.
  - `enable_aes_gcm_dpic_messages`
  - `cxl_aes_gcm_endianness`
- Key: Default input is treated as Little Endian.

Steps to run IDE Sanity test in Example:

1. Add the following define in `sim_build_options` file:

   ```
   +define+SVT_AES_DPI_ENABLE
   ```

2. Set the ELLIPSYS_HOME environment variable:

   ```
   setenv ELLIPSYS_HOME <Cryptography Library Install Path>/ellipsys
   ```

3. Copy the `vcs_build_options_with_aes` to `vcs_build_options`.
4. Run the test case:

   ```
   gmake USE_SIMULATOR=vcsvlog cxl_ide_sanity_test
   SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_pcie_serial.f
   SVT_CXL_SUBSYSTEM_TOPOLOGY_FILE=topology_snps_vip_cxl_b2b.svi
   EN_SVT_CXL_IDE=1 ELLIPSYS_HOME=" <Cryptography Library Install Path>/ellipsys"
   ```

   The above command is to run the test – `ts.cxl_ide_sanity_test.sv` with FULL_STACK topology over serial interface using VCS Simulator.

3.17.3.3 Debug

AES-GCM Trace file to debug:

You can enable the trace file from AES-GCM Layer to debug the message encrypted or decrypted via `"enable_transaction_logging"` attribute available in Cache-Mem Configuration.

Following is the snippet for the trace file:

Note

Prints in the Trace file are in the following order:
`{Byte[0], Byte[1] ….. Byte[N]}`

```
27193000.00 ps |   TX | IV   | 000000000c | 80000000000000000000002
27194000.00 ps |   TX | AAD  | 0000000004 | 888094d0
27194000.00 ps |   TX | PT   | 0000000030 | 636f0f8553000000000000000000000334e0000000541200000000000000d3510000000005c0c0000000000000
27194000.00 ps |   TX | CT   | 0000000030 | 761a84f8d3ea25de51d41cbddf234369ccfd147e4990e0124a23e8a8276cc4c5cd0e6ca031626c6c9c20f457f1361f13
27196000.00 ps |   TX | AAD  | 0000000004 | 88816210
27196000.00 ps |   TX | PT   | 000000003c | d7520000000c00e000000045310000000441c0000000000000000610093c762000000001800000000000030c5f2938000000001000000000000
27196000.00 ps |   TX | CT   | 000000003c | 2c2bd44994c35f8de4945a121aa252ca4cc2933f99d2deb2dd3e8ad021710d15340cbe7a44ea9734416142de280a98b3aa990c16892f42cd68e8a23
27197000.00 ps |   TX | AAD  | 0000000000 | 0
27197000.00 ps |   TX | PT   | 0000000040 | a7c1109ef859dcb7e4bf73d36782a2360b102543635a7f394c7d7b5e657d698a5a425d028dc1b345f6f8ae11999a3f32c451a0b6efd986e5f2d5a569ed3e4112
27197000.00 ps |   TX | CT   | 0000000040 | ccb86ac985ea15462de22067ec87fd08572a2c698a76e6408481162cff10aa98f96bb54277b2682e910740713d5852f4d3bc4d100b8ad61180ec90343fe0fbed
27198000.00 ps |   TX | AAD  | 0000000004 | 88816210
27198000.00 ps |   TX | PT   | 000000003c | 9321000000000301d000000008729000000000e43c0000000000000000610a1cd4c000000001800000000000063a007b33e000000001800000000000
27198000.00 ps |   TX | CT   | 000000003c | f3ca8959187bd8efd3fdf620fc3ee91a895aabe9acfedc250a86304d2335230a6da10520c390a143d86f76451ce84e9bfca692f2e1288934c119dcf8
27199000.00 ps |   TX | AAD  | 0000000000 | 0
27199000.00 ps |   TX | PT   | 0000000040 | 963f3a844b8776b0aa054bd1d289f00b8f185240f2f9f71ac1648e8909f9117c1e34ea00f7d62615b7b055eaaa8c3ac0a83d40daf056ea36a6741cb857145a45
27199000.00 ps |   TX | CT   | 0000000040 | 8f71417858aa82f52878da803bfb1bb6285e4549a5c35737bb09b7a7359ab7fe7a65743a1d77d3f54f774c3d1c3a8576a41bbdd404f0b9b970aa3575949bfae7
27199000.00 ps |   TX | TAG  | 000000000c | 84b60034194d92eb2c3d055c

27199000.00 ps |   TX | IV   | 000000000c | 80000000000000000000003
27200000.00 ps |   RX | AAD  | 0000000004 | 88804930
27200000.00 ps |   RX | CT   | 000000003c | a6b226a1f4a86f13d1de97327de3799da1d60fdb41df9aaec66b816ce959d985511bb1bb156bcc952d814418855a4f8ea94b89f30aa4d1d9093d9172
27200000.00 ps |   RX | PT   | 000000003c | 837704780e0000000000000dd0b021c11000000000000000000000030498088000000000000000000000030498088000000000000000000000000
27200000.00 ps |   TX | AAD  | 0000000004 | 888122d4
27200000.00 ps |   TX | PT   | 0000000030 | 853000000000d8030000000000000000730b774f4d0000000010000000000259ba3037500000001000000000000
27200000.00 ps |   TX | CT   | 0000000030 | 326330f713f938f74c836e99ad492a312183896606 9c906eddb9b272c7489254cd52e7c7266af5b6878c7e8065a2519d
27201000.00 ps |   RX | AAD  | 0000000004 | 88804930
27201000.00 ps |   RX | CT   | 000000003c | 5751c6fbd8ef9e10f6c6201082a85810d0661d00bb3637119263cd155cd21f0fc2f096298b698f3500f28bc78ac45982a6009f1d657950f3687f5904
27201000.00 ps |   RX | PT   | 000000003c | 5dd8800640d0000000000000049ab0650100000000000000000000000030498088000000000000000000000030498088000000000000000000000000
27201000.00 ps |   TX | AAD  | 0000000004 | 888005b0
27201000.00 ps |   TX | PT   | 000000003c | 73f88aa867000000001000000030c73893c0000000008000000000009b2ed913a7c158f6a93a69b5fa07c2ba44a65fe7cd3a3fa71e30511fbed101bd
27201000.00 ps |   TX | CT   | 000000003c | a02791819 4c02f4168e91abf61c8daf2271f2eb892bba89a8b01e83790f15c42633773611fea38dc82ce9b849756cc6ad09230149 19614 37df8631c4
27202000.00 ps |   RX | AAD  | 0000000004 | 88804934
27202000.00 ps |   RX | CT   | 000000003c | ae800515acfe9546dc49ee824de76ac44ece8c22fb589ab11e10869614 537f445366ac29fee704ccc848ebc6f3a7fe163fcf10954d9f19f3b5fe9ee
27202000.00 ps |   RX | PT   | 000000003c | 51202b819000000000000000872b02d81b0000000000000000000000034498088000000000000000000000034498088000000000000000000000000
27202000.00 ps |   TX | AAD  | 0000000004 | 88808010
27202000.00 ps |   TX | PT   | 000000003c | 13370000000090 0f00000000f490d53a7f1fb7ee1651c0622a8dce5f4c18c80dcfd30ef2fb199b251fb26ff00525000000002c2b0000000000000000
27202000.00 ps |   TX | CT   | 000000003c | 185085b5428ae87dd4022dfbb778ab974df773622f67c560ae69b7ca4a6eeeaaaf81f51bc7dd0398b802daa05e11c10d3f993568d5376b951b27f934
27203000.00 ps |   RX | AAD  | 0000000004 | 88804930
27203000.00 ps |   RX | CT   | 000000003c | 24627a6d5aad47ab2a0c54d51a454dcc3ce3ef2f192c53ec4288a0759f32fc7aee3a032bab0a4a70f73a0f7bd770c50972f29ea261ee4c858d7e345
27203000.00 ps |   RX | PT   | 000000003c | 9bfa06f00c0000000000000030498088000000000000000000000000030498088000000000000000000000030498088000000000000000000000000
27203000.00 ps |   TX | AAD  | 0000000004 | 888005b8
27203000.00 ps |   TX | PT   | 000000003c | 73f3210371000000001800000058c78e1220000000008000000000004 1007f693b43bed7b017f255e185925fffe788967edb8a3920c8145cd428afc4
27203000.00 ps |   TX | CT   | 000000003c | 1ba8953edb8b8b52af718394eb97c7541ff93a8d2b71fd2a5967fa5085523c3cca4e11c31a0a6102e084cdbcbfaec4ad7513b5d33a66257bf47ddb9
27204000.00 ps |   TX | AAD  | 0000000004 | 88800010
27204000.00 ps |   TX | PT   | 000000003c | 454100000000e8370000000003128bf2fe13b3198563d32808cfc28564 5dff5a62407bdfec855f19469aa134b8c3de0f6176d1fa4000000000000000
27204000.00 ps |   TX | CT   | 000000003c | 7ad603ba25b54501ae12cdb45f6f88033a6e94a3a7bcbe02529c7e1eea920c30a9ab9f8adb4b530e192e1b1204d2f65399b9fde6f20ae7f978bee53d
27204000.00 ps |   TX | TAG  | 000000000c | ae184933f401d7ce945cbf34
```

3.17.4 Memory Interleaving

CXL VIP Host and Device agent components support interleaved set of addresses. The agent components can be grouped together in one interleaved set. The agents within an interleaved set supports a unique non-overlapping set of address ranges. For example, assume that there are 2 Devices within an interleaved set. Device 0 supports address range 0 to 255, Device 1 supports address range 256 to 511, Device 0 supports address range 512 to 767, Device 1 supports address range 768 to 1023, and so on. What it means is that Device 0 would process transactions with addresses 0 to 255, 512 to 767 and so on.

3.17.4.1 CXL Device Agent

CXL TL layer monitor issues error if it observes memory transactions with an address which must not be accessed by CXL device agent.

3.17.4.2 CXL Host Agent

Host VIP checks if the memory address generated by it falls in the interleaving address range. If the address does not lie in the interleaving address range for this Host, then the Host VIP generates error through is_valid() check. Host VIP would still transmit the transaction.

3.17.4.3 CXL System Configuration Controls

This configuration API need to be programmed to enable this feature. Refer to CXL class reference for more details:

- svt_cxl_system_configuration::configure_memory_interleaving_parameters(bit is_host_interleaved_group, bit [`SVT_CXL_MAX_ADDR_WIDTH-1:0] interleaving_start_addr, bit [`SVT_CXL_MAX_ADDR_WIDTH-1:0] interleaving_end_addr, int interleave_granularity, int interleave_ways, int tgt_agent_id[]);

3.17.4.4 Usage Examples

1. Two Interleaved CXL Devices (i.e. 2 interleave ways) with 256 Bytes Interleave Granularity:

   You need to configure the VIP as below:

   cust_cfg is a handle of svt_cxl_subsystem_link_configuration:

   cust_cfg.cache_mem_sys_cfg.device_cfg[0].spec_ver = svt_cxl_types::cxl_ver_2_0;

   cust_cfg.cache_mem_sys_cfg.configure_memory_interleaving_parameters(0, 'h8000, 'hffff, 256, 2, {0,1});

   You need to call reconfigure method in connect_phase() after above configuration settings.

   device_env is handle of svt_cxl_subsystem_env.

   device_env.cache_mem_env.reconfigure(cust_cfg.cache_mem_sys_cfg);

The VIP takes care of the interleaving based on the above configuration settings. In the above use case 1, VIP generates an error under the following cases:

- svt_cxl_transaction::addr[8] == 0 address comes on agent_id 1
- svt_cxl_transaction::addr[8]==1 comes on agent_id 0

Note CXL VIP does not support multi host or multi device topology. So, the configuration API configure_memory_interleaving_parameters should be called for only one host or device(i.e. interleave ways=1).

Example:

```
cust_cfg.cache_mem_sys_cfg.configure_memory_interleaving_parameters(0, 'h8000, 'hffff,
256, 1, {0});
```

3.18 PM VDM Features

CXL subsystem application layer now handles the PM VDM feature handshakes. These are the features and their APIs available inside sequence collection. With introduction of App layer, these APIs are modified/enhanced to utilize Subsystem App configurations and auto response flow.

These APIs are under `svt_cxl_subsystem_sequence_collection`. Refer Class reference document for “application layer “ and “PM VDM API” group.

3.18.1 CXL PM VDM and PM Credit initialization APIs

1. Before any PM VDM feature initiation, the `pm_credit_initialization()` API need to be invoked from VIP to perform PM credit initialization.
2. `generate_cxl_pm_vdm()` is used to populate and generate any type of CXL PM VDM messages.

3. `restore_credits()` is used to restore credits to the remote partner. (Invoked internally by the other APIs for the features supported)

3.18.2 CXL Reset APIs

- `reset_prep_vdm_exchange()` is used to perform any Reset type VDM exchange. (Warm, Sleep State etc...)
- `perform_warm_reset()` API is used to perform Warm Reset VDM exchange and Hot.Reset entry.

  This internally invokes `reset_prep_vdm_exchange` API for the VDM exchange.

3.18.3 CXL PMREQ APIs

- `initiate_pm_req()`: API to initiate PMReq.Req from Device
- `initiate_pm_rsp()`: API to initiate PMReq.Rsp from Host

  You do not need to invoke this API unless PM.Rsp needs to be initiated unilaterally.
- `Initaite_pm_go()`: API to initiate PMReq.Go from Host

  You do not need to invoke this API unless PM.Go needs to be initiated unilaterally.
- `perform_pm_req_entry()`: API to perform PMReq Entry with handshake between Req, Rsp and Go

  This API invokes/waits till PMReq entry handshake is successful.

3.18.4 Warm Reset

To perform warm reset, you must complete the power management and credit initialization and then send reset request. The CXL power management messages are sent as PCIe Vendor Defined Type0 messages with a 4DW data payload.

PM Credits and initialization process is link local. `PM2IP.CREDIT_RTN` and `PM2IP.AGENT_INFO` messages are required to initialize Power Management messaging protocol intended to facilitate communication between the Downstream and the Upstream Port.

This can be achieved by using the API: `pm_credit_initialization`

Description:

```
task  svt_cxl_subsystem_virtual_api_collection_sequence::pm_credit_initialization  (
int link_num = 0  )
```

API to perform Credit and PM initialization.

For warm reset, the host shall issue a CXL PM VDM defined as ResetPrep (ResetType = Warm Reset;

PrepType = General Prep) to the CXL device. CXL device shall flush any relevant context to the host and

take any additional steps that are necessary for the CXL host to enter LTSSM Hot-Reset. After all the Reset preparation is completed, the CXL device issues a CXL PM VDM defined as ResetPrepAck (ResetType = Warm Reset; PrepType =

General Prep)

The above can be done through the API: `perform_warm_reset`

Description -

```
task svt_cxl_subsystem_virtual_api_collection_sequence::perform_warm_reset (
int link_num = 0  )
```

API to perform Warm Reset VDM exchange and moving to Detect via Hot.Reset entry. (Implicitly calls reset_prep_vdm_exchange API with value of 8'h10)

Usage example

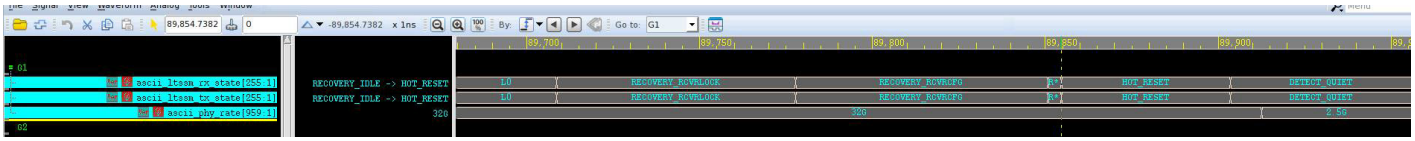
```
vip_seq.pm_credit_initialization();
vip_seq.perform_warm_reset();
```

Logs Snippet

Transaction log shows the Message with Data VDM packets.

```
1519 spd_0        89539.000 89539.000 R CplD       0x0001/015   90  0 0    0 0 0  ID:0x0100 Stat:SC  BC:0096   24 (H) 4a000018 01000060 00011518   ----   0          0x1d8af813
1520                                                                                                              0 (D) 04570622 cfaa07b5 e94f34bc 27e0ed46
1521                                                                                                              4 (D) 4b1edd0e b5d16443 6fcf81f2 85552e88
1522                                                                                                              8 (D) 21b38786 9cfdbf0e e1028668 253e305d
1523                                                                                                             12 (D) ac794b20 31023289 a548b1e9 1949b3b6
1524                                                                                                             16 (D) e6275b72 b6256afe e91f5d7f c1df8a08
1525                                                                                                             20 (D) 63568ea5 76b0f1bb 36688060 68eec962
1526 spd_0        89543.000 89544.000 T MsgD       0x0001/000  182  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0xb95050b9
1527                                                                                                              0 (D) fe000000 027f0000 00000000 00000000
1528 spd_0        89548.000 89548.000 T ACK                    90                                                                                                    0xfcfa
1529 spd_0        89550.000 89550.000 T UPDATEFC_CPL_VC0                          191 OFF 1460 OFF                                                                    0xa60
1530 spd_0        89554.000 89554.000 R UPDATEFC_P_VC0                            193 OFF 1465 OFF                                                                    0xd81a
1531 spd_0        89556.000 89556.000 R UPDATEFC_NP_VC0                           192 OFF 1025 OFF                                                                    0x6ccf
1532 spd_0        89558.000 89558.000 R MsgD       0x0001/000   91  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0xe1bc47c6
1533                                                                                                              0 (D) fe7f0000 02000000 00000000 00000000
1534 spd_0        89561.000 89562.000 T MsgD       0x0001/000  183  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0xbffdf0ab
1535                                                                                                              0 (D) 00000100 01000000 00000000 00000000
1536 spd_0        89563.000 89563.000 T UPDATEFC_CPL_VC0                          192 OFF 1466 OFF                                                                    0x197
1537 spd_0        89574.000 89574.000 T ACK                    91                                                                                                    0x5de1
1538 spd_0        89575.000 89575.000 R MsgD       0x0001/000   92  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0xc89c0431
1539                                                                                                              0 (D) fe7f0000 01000000 00000000 00000000
1540 spd_0        89576.000 89576.000 R MsgD       0x0001/000   93  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0x7da4efa4
1541                                                                                                              0 (D) 007f0000 01000000 00000000 00000000
1542 spd_0        89579.000 89580.000 T MsgD       0x0001/000  184  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0x8fc7de79
1543                                                                                                              0 (D) fe000000 01000000 00000000 00000000
1544 spd_0        89580.000 89580.000 T UPDATEFC_P_VC0                            194 OFF 1026 OFF                                                                    0x9e11
1545 spd_0        89580.000 89581.000 T MsgD       0x0001/000  185  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0xbc9a2761
1546                                                                                                              0 (D) 02000100 10000000 00000000 00000000
1547 spd_0        89582.000 89582.000 R UPDATEFC_P_VC0                            195 OFF 1467 OFF                                                                    0x42f0
1548 spd_0        89585.000 89585.000 R ACK                   183                                                                                                    0xdfde
1549 spd_0        89595.000 89595.000 T ACK                    93                                                                                                    0x9bb8
1550 spd_0        89595.000 89595.000 R MsgD       0x0001/000   94  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0x2ebde15c
1551                                                                                                              0 (D) fe7f0000 01000000 00000000 00000000
1552 spd_0        89596.000 89596.000 R MsgD       0x0001/000   95  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0x6b0117d9
1553                                                                                                              0 (D) 027f0000 10000000 00000000 00000000
1554 spd_0        89600.000 89600.000 T MsgD       0x0001/000  186  0 0    0 0 0      VendorDefined0     100   4 (H) 74000004 0001007e 00001e98 00000068 0          0x69e63b14
1555                                                                                                              0 (D) fe000000 01000000 00000000 00000000
1556 spd_0        89603.000 89603.000 T UPDATEFC_P_VC0                            197 OFF 1029 OFF                                                                    0xcde0
1557 spd_0        89606.000 89606.000 T ACK                    95                                                                                                    0xd98f
1558 spd_0        89607.000 89607.000 R ACK                   185                                                                                                    0x115a
1559 spd_0        89617.000 89617.000 R UPDATEFC_P_VC0                            197 OFF 1469 OFF                                                                    0xa88a
```

Waveform will show the LTSSM going into Hot Reset state:



Signals:

```
<test_top>.spd_0.m_ser.port0.pl0.ascii_ltssm_rx_state[255:1]
<test_top>.spd_0.m_ser.port0.pl0.ascii_ltssm_tx_state[255:1]
<test_top>.spd_0.m_ser.port0.pl0.ascii_phy_rate[959:1]
```

Keywords to debug:

- pm_credit_initialization

- generate_cxl_pm_vdm
- perform_warm_reset
- reset_prep_vdm_exchange

3.18.5 Known Limitations

- No response timeouts or protocol checks are implemented currently for VDM messages.

3.19 CXL DOE

The Data Object Exchange capability provides a mechanism for software to directly exchange data with a function through the use of configuration reads and writes. This section outlines how DOE discovery and data transfers take place.

3.19.1 Overview

The class `svt_pcie_device_virtual_doe_sequence` is introduced to encapsulate a DOE transfer. You must provide all the fields defined in this specification for a doe_transaction. It performs the following steps:

1. Read the DOE status register. Check that the DOE busy bit is clear. If not, start the polling routine until the busy bit is clear.
2. Write the provided DOE structure to the DOE write data mailbox register. The user is responsible for constructing a proper DOE structure.
3. Set the DOE go bit in the DOE control register.
4. Optionally set the DOE interrupt register.
5. If the interrupt register is set, wait for MSI interrupt. If not, you must poll the data object ready bit in configurable intervals.
6. If polling and polling timeout is hit, write the abort bit in the DOE control register and end the transaction and update status with failure.
7. After the MSI is received or while polling, read the DOE status register. Ensure that the DOE error bit is clear. If not, flag an error. If interrupts are enabled, check that the interrupt status bit is set.
8. Check that the data object ready bit is set. If an interrupt is received and the data object ready bit is not set, then flag an error. If an error occurs, abort the transaction and update the status as unsuccessful.
9. Once a response is ready, read data, one dword at a time from the DOE read data mailbox.
10. If interrupts are used, write 1 to the interrupt status bit to clear it.
11. You must update the status as successful or unsuccessful after all data has been read and return the data.

There is no attempt to interpret the data that was exchanged. It is at your discretion to process this data.

Example using DOE discovery:

The DOE discovery object contains 3 DW. The first 2 DW are described as above. For PCIE the vendor ID is ‘h0001. For CXL the vendor ID is ‘h1E98. Since a DOE discovery object is 3DW the length is set to 3. The discovery 1DW payload is defined as:

Table 3-2 DOE Discovery Request Data Object Contents(1DW)

Bit Location	Description
7:0	Index - Indicates DOE Discovery entry index queried. Indices must start at 00h and increase monotonically by 1.
31:8	Reserved - Requesters must place 00 0000h in this field. Responders must ignore the value in this field.

You need to construct a 3DW array using these definitions and assign it to `write_mailbox` in the doe transaction.

This transaction uses interrupts. If during discovery the attached device did not support interrupts, then the `interrupt_enable` bit must not be set.

Example pseudo-code:

```
svt_pcie_device_virtual_doe_sequence doe_discovery;
doe_discovery =
svt_pcie_device_virtual_doe_sequence::type_id:create("doe_discovery");
doe_discovery.write_mailbox.push_back({8'b0, 8'b0, 16'h1e98}); // 8 reserved bits, data
object type=discover, CXL ID
doe_discovery.write_mailbox.push_back(32'h0000_0003);
doe_discovery.write_mailbox.push_back(32'h0000_0000); // Index 0 points to discovery.
So we are asking if the device is capable of CXL discovery.
doe_discovery.doe_capabilities_base_addr = 12'h0abcd;
doe_discovery.interrupt_enable = 1'b1; // use interrupts.
doe_discovery.doe_polling_interval = 1000; // poll every 1us
doe_discovery.doe_timeout = 1000000; // command timeout after 1ms.
doe_discovery.interrupt_number = 7; // Look for MSI with bit 7 set to indicate the
device's read mailbox is ready.
start_item(doe_discovery);
finish_item(doe_discovery);
get_response(doe_discovery);
if (doe_discovery.status != doe_discovery::SUCCESSFUL) $dislpay("ERROR!");
else begin// check the contents of read_mailbox_data
   for (int i=0; i<doe_read_mailbox.size();i++) begin
    $display(doe_discovery.read_mailbox[i]); // print the response
…
   end
end
```

3.19.2 Data Classes

These fields will be used for DOE transfer:

1. bit [11:0] doe_capabilities base_addr- This address points to the beginning of the DUT's doe capabilities extended capabilities register. The sequence will add offsets to this base address to access the DOE control, read and write mailbox registers.
2. bit [31:0] doe_read_mailbox_data[$]- Data from the DOE read mailbox.
3. bit [31:0] doe_write_mailbox_data[$]- Data to be sent to the DOE write mailbox.

4. interrupt_enable- Indicates whether the interrupt bit should be set when writing to the DOE control register.
5. interrupt_number- Indicates the MSI interrupt number that should be used to determine if the DOE read mailbox is ready.
6. int doe_polling_interval_ns- Indicates the polling interval to be used to check the data_object_ready bit.
7. int doe_timeout_ns- Indicates the amount of time the DOE sequence should wait before declarding a timeout on the DOE transfer.
8. typedef enum doe_status- Indicates wether or not the DOE transaction was successful.

3.19.3 Protocol Checks and Exceptions

1. A DOE timeout error is issued if the transaction does not complete in the time specified in the sequence.
2. An error is flagged if the doe error bit is set in the doe status register.
3. If a DOE interrupt was received but the data object ready bit is not set then flag an error.
4. If you are using interrupts, check that the interrupt status bit is set in the DOE status register.
5. Check whether the PCIe specification version is 4.0 or greater before executing a DOE sequence.

3.20 Hot Plug Feature

Hot plug flow comprises of Hot removal, and Hot add. Hot removal results into Flexbus link down, PCIe unplug, and other CXL components unplugged as well. Hot add results into Flexbus link up, PCIe plugging-in and other CXL components plugging-in. Mid-sim reset feature is supported as one of the Hot Plug flow i.e. “Hot removal followed by immediate Hot add”.

3.20.1 Usage API

VIP provides a sequence to execute Hot Plug flow namely `svt_cxl_subsystem_app_hot_plug_sequence`. The sequence offers controllability on various arguments as listed below. The details are available in Class reference.

- Hot Plug type
- Wait event for Hot removal.
- Timed delay for Hot add.
- Blocking or non-blocking nature of sequence execution
- Option to preserve applicable settings.

3.20.2 Topologies validated

Validated with VIP-VIP topologies as listed below.

- Full stack with CXL.IO and CXL.Cache/Mem
- All supported LPIF topologies

3.20.3 Validation

Preliminary validation covers following scenario.

- Hot plug type – HOT_REMOVAL_AND_AND, Wait event – NONE, Port ID – 0
- Blocking execution of sequence invocation.
- Invoked when system is idle.
- When it is invoked, Flexbus goes down and entire Subsystem is unplugged until Hot added back.
- Once Hot added, Flexbus comes back up, followed by DL initialization and PM credit initialization and entire Subsystem is back up again.
- Test execution of basic traffic flow post Hot Plug execution.

3.20.4 Known limitations

- Additional fine grain controls are TBD.
- CXL.Cache/Mem reset spec requirements are being analyzed. Some of the functional aspects listed below are yet to be implemented:
  - TL handling of modified cache lines during hot removal
  - TL handling of cache and mem configuration compatibility with host during hot add
- These Hot Plug sequence arguments are not completely validated yet:
  - Non-blocking sequence invocation.
  - Hot Plug types - HOT_ADD_AND_REMOVAL, HOT_ADD, HOT_REMOVAL
  - Hot removal wait event – LINK_IP, PM_INIT, SYS_IDLE, TIME_DELAY
- These topologies are not yet validated for Hot Plug flow:
  - Full stack CXL.IO ONLY topology.
  - Full stack CXL.IO and CXL Cache/Mem DL only.
  - TLM modes

3.21 Viral Handling

3.21.1 Configuration variable / Enabling Viral handling support

`svt_cxl_cache_mem_configuration::enable_viral_handling` is set by default enabling the feature at Cache-mem Transaction and Link Layer.

Process to enter viral state

Make the Link layer to enter into Viral by inserting un-correctable error i.e sending unexpected Init.Param. Use the sequence API `send_init_param()` as below to insert it.

```
vip_seq.send_init_param();
```

Under TL Only Topology

Use this API to bring the Cache-mem TL into Viral state. This can be invoked based on Viral detection at DUT DL.

```
vip_seq.update_cxl_viral_status(.cache_mem_link_num(link_num),.viral_state(svt_cxl_types::VIRAL_ACTIVE));
```

Also indicate the Retry.Ack reception/ sending as shown. This is required to start executing the protocol checks for the Mem read transaction's responses at Host end.

```
vip_seq.update_cxl_viral_status(.cache_mem_link_num(link_num),.viral_state(svt_cxl_types::VIRAL_COMMUNICATED));
```

Process to Exit out of viral state

Any Conventional reset say, hot, warm or cold reset must cause the Viral exit, as shown:

```
// Initiating reset at Host end to push Host and devices out of Viral state
vip_seq.enter_hot_reset();
```

3.22 TLM Port

At the transaction layer, data link layer, and ARB/MUX, TLM ports are available.

The same connector class - `svt_cxl_dispatch_connector` is used at all the levels (ARB/MUX, Link Layer and Transaction Layer)

Description about `svt_cxl_dispatch_connector`:

This is type generic.

Usage is as follows:

Usage in typical back to back setup:

Dispatch connector to connect dispatch sequencer of a 'local' VIP to blocking put port of a 'remote' VIP in a typical back to back testbench setup

3.22.1 Example usage

- TLM Port declaration in TL back to back:

```
svt_cxl_dispatch_connector #(svt_cxl_channel_transaction,
`SVT_XVM(blocking_put_imp_rx_cache_req_chan)#(svt_cxl_channel_transaction,
svt_cxl_tl)) h2d_tl_cache_req_chan_connector;
```

- TLM Port declaration in DL back to back:

```
svt_cxl_dispatch_connector #(svt_cxl_flit,
`SVT_XVM(blocking_put_imp_rx_cxl_flit)#(svt_cxl_flit, svt_cxl_dl))
h2d_dl_flit_connector;
```

Files for reference:

```
<design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys/env/svt_cxl_dispatch_connector.sv
```

```
<design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys/env/svt_cxl_subsystem_basic_env.sv
```

For more details, refer the class reference:

```
https://spdocs.synopsys.com/dow_retrieve/latest/vg/snps_vip_lib/class_ref/cxl_subsystem_svt_uvm_class_reference/html/tlm_ports.html
```

3.23 MLD Feature

CXL 2.0 supports only Type 3 MLD components. An MLD component can partition its resources into up to 16 isolated Logical Devices. Each Logical Device is identified by a Logical Device Identifier (LD-ID) in CXL.io and CXL.mem protocols. Each Logical Device visible to a Virtual Hierarchy (VH) operates as a Type 3 device. The LD-ID is transparent to software accessing a VH.

Existing APIs:

- `set_cxl_subsystem_env_device_cfg`

  The API `set_cxl_subsystem_env_device_cfg` is enhanced to support MLD. Following new attributes are added to the API:

  - `num_agent`: This attribute is used to specify number of Physical links/devices connected in subsystem. The CXL Agents are created in the subsystem based on this attribute value.
  - `Number_of_ldid_supported[$]`: This attribute is used to determine how many LD-ID is supported for MLD.
    - If the value of this attribute is 0, then the device is considered as SLD.
    - The attribute value is an array and the depth should be equal to `num_agent`. Index will define the device number and the value will define how many LD-ID needs to be configured for that device.

- `set_cxl_subsystem_env_host_cfg`

  The API `set_cxl_subsystem_env_host_cfg` is also enhanced with same new attributes. However, as the host does not have multiple TL layer, therefore, it is not required to pass these attributes. These are added in this API to maintain the similarity between `set_cxl_subsystem_env_host_cfg` and `set_cxl_subsystem_env_device_cfg`.

Example:

DUT is Host: Configure Device VIP

```
set_cxl_subsystem_env_device_cfg (.num_agent(3), num_ldid({0,4,0}))
```

The above example is described below:

1. Total 3 devices (0, 1, 2).
2. Device 0 and 2 are SLD.
3. Device 1 is mld.
4. Device 1 has 4 LD-ID supported.

The indexing of `cache_mem_configuration` in this case is illustrated as shown below:

```
Device 0
Device 1 -  (0)            MLD Device having 4 LD-ID, so 4 cxl agent will be created.
Device 2-   (1)
Device 3 -  (2)
Device 4 -  (3)
Device 5
```

With the help of above API creation of multiple CXL agents, MLD can be taken care in VIP. Also, the `cache_mem_configuration` variables can be set internally.

Host can check `enable_mld` and `number_of_LD_supported` for device to check if device is MLD or not through `CXL_cache_mem_configuration` class.

Other APIs:

- `get_mld_cfg_index`

  The `get_mld_cfg_index` API returns the index of LD-ID agent/configuration for MLD devices. This is used to get the proper index to get access to a particular LD-ID. This is needed only for Device and not for the Host as there are no multiple agents for MLD host. The API contains the following attributes:

  - `agent_id`: This attribute is used to specify the device number or index (based on the current use model) in subsystem. If there is only one device, then the `agent_id` is set to 0.
  - `Ld_id`: This attribute is used to specify the LD-ID number for which index is needed.
  - `cfg_index`: This attribute is returned in the response.

    0th index for every device will be the base configuration which will be used to configure DL layer also and other configurations should be used only for TL layer.

  Example 1: **DUT is Host**

  ```
  get_mld_cfg_index(.agent_id (0),  .ld_id(2))
  ```

  Take the above API example. This API describes as below:

  a. Device 0 and `ld_id` is 2.

  b. API will return the value “2”.

- `set_addr_range()`

  This is the legacy API present in `svt_cxl_system_configuration` that is required to configure the start and end address for the agent. All the LDs of the MLD should be configured as non-overlapping address, distinct with respect to any of the existing agent’s address map. Overlapping addresses are not supported for MLD. The API should be accessed from the base test and for each device ‘i’, following fields should be set:

  - `agent_type` as DEVICE
  - `host_dev_id` as index of the agent
  - `start_addr` as start address in the memory space
  - `end_addr` as end address in the memory space

  For example, for each (`cust_cfg.cache_mem_sys_cfg.device_cfg[i]`)

  ```
  cust_cfg.cache_mem_sys_cfg.set_addr_range(svt_cxl_cache_mem_configuration::DEVICE, i,
  s_addri,e_addri);
  ```

  where,

  - `cust_cfg` is handle to the user config extending from `svt_cxl_subsystem_link_configuration`.
  - `s_addri` is start address for ‘i’th device.
  - `e_addri` is end address for ‘i’th device.

Note

- The address map is used by TL Host to update the `svt_cxl_transaction` class field `ld_id` with correct value. If address map is not set for a LD, then the read write traffic is not initiated to that device.
- Address poisoning feature will not work for that LD.
- Only memory address space is supported. Register address space is not supported.
- Only non-overlapping addresses are supported.

- `get_agent_idx_as_per_addr()`

  This is the new API added in `svt_cxl_system_configuration` for ease of getting an agent's index based on the input address. This API should be used to get the agent index on which address poisoning needs to be done.The existing API `svt_cxl_subsystem_virtual_api_collection_sequence::update_cxl_tl_mem_poison()` should be updated to drive the agent index to the argument `cache_mem_link_num`. This helps to drive the poison sequence on the said device sequencer. The user trying to use poison for MLD should use the API as:

```
if (link_cfg.cache_mem_sys_cfg.device_cfg[0].enable_mld)
device_seq.update_cxl_tl_mem_poison(
.cache_mem_link_num(link_cfg.cache_mem_sys_cfg.get_agent_idx_as_per_addr(dev_pois_addr))
,
.addr(dev_pois_addr),
.poison(1));

else

device_seq.update_cxl_tl_mem_poison(

.cache_mem_link_num(0),
.addr(dev_pois_addr),
.poison(1));
```

  where,

  - `device_seq` is the handle to `svt_cxl_subsystem_virtual_api_collection_sequence`.
  - `dev_pois_addr` is the address to be poisoned.

  The API `get_agent_idx_as_per_addr` returns the `agent_index` for the agent that `dev_pois_addr` maps. This update is done in the sequence, `svt_cxl_subsystem_ts_tl_mem_poison_sequence`.

3.23.1 Configurations

New configurations are added in `svt_cxl_cache_mem_configuration` class for MLD.

- The below variables gets configured internally by VIP depending on the API, `set_cxl_subsystem_env_device_cfg`.
  - Flag to enable MLD feature in VIP: `enable_mld`
  - Variable to define number of LDs supported in VIP: `number_of_LD_supported`
  - Variable to define LDID number for the configuration: `ldid_num`

3.23.2 Usage

- The following existing APIs are modified to enable MLD in Device:

```
set_cxl_subsystem_env_device_cfg(1,svt_cxl_subsystem_configuration:: CACHE_MEM_STACK,
svt_cxl_subsystem_types::DEFAULT_IF, 0, .num_agent(1), .num_ldid({8}));
```

  - `num_agent`: This attribute is used to specify number of Device agents to be configured.
  - `num_ldid[$]`: This attribute is used to specify the number of LD-ID supported for each MLD agent. For SLD agent, this value should be 0.
- Device will automatically set the below configuration class variables based on the above API:
  - `svt_cxl_cache_mem_configuration::enable_mld`
  - `svt_cxl_cache_mem_configuration:: num_of_ld_id_supported`
- If VIP is Host, then the Host must read device configuration. If the device is MLD, then set the VIP Host accordingly. Set the following variables for HOST:
  - `svt_cxl_cache_mem_configuration::enable_mld`
  - `svt_cxl_cache_mem_configuration:: num_of_ld_id_supported`
- Set the following configurations as suggested:
  - `vip_cfg.set_spec_ver(version,link_num)` - Version should be 2.0
  - Set `device_type` equal to `TYPE3_DEVICE`.
- If you are running with `svt_cxl_cache_mem_configuration:: enable_tl_user_rsp_seq =0`, then the response sequence of the device is handled by the VIP.
- If you are driving their response sequence, then it is required to set the `svt_cxl_cache_mem_configuration:: enable_tl_user_rsp_seq =1`. See the example in the mentioned sequence:
  /tb_cxl_subsystem_uvm_basic_sys/env/seq_and_cb/svt_cxl_subsystem_ts_cache_mem_sequence_collection/svt_cxl_subsystem_ts_cache_mem_tl_rand_resp_sequence.sv
- For Error injection on the response path in each of the LD, the `svt_cxl_ts_tl_cm_err_injection_cb` needs to be created for each of them and added to callback pool. For example,

```
/** Instance of CXL TL Mem Error Injection Call back */

  svt_cxl_ts_tl_cm_err_injection_cb drv_mem_err_cb;


task cxl_test_main_phase();

foreach (cust_cfg.device_cfg.cache_mem_sys_cfg.device_cfg[i]) begin

        drv_mem_err_cb = new($sformatf("drv_mem_err_cb_%0d",i));

        drv_mem_err_cb.rsp_err_inject = 1;


svt_cxl_tl_callback_pool::add(env.device_env.cache_mem_env.cache_mem[i].tl_driver,
drv_mem_err_cb);

      end
```

```
endtask
```

- Hot Plug can be performed per LD using the API, `svt_cxl_subsystem_virtual_api_collection_sequence::perform_hot_plug`.

  As the LDs are communicating to a single host, so it does not require handshakes before getting `hot_removed`. Thus, device VIP will hot remove all the LDs attached to it.

  For example, to `HOT_REMOVE` LDs of the MLD in your sequence, call the API as:

  ```
  device_seq.perform_hot_plug(svt_cxl_subsystem_app_service::HOT_REMOVAL);
  ```

  where, `device_seq` is handle to `svt_cxl_subsystem_virtual_api_collection_sequence`.

4 CXL Subsystem Verification Topologies

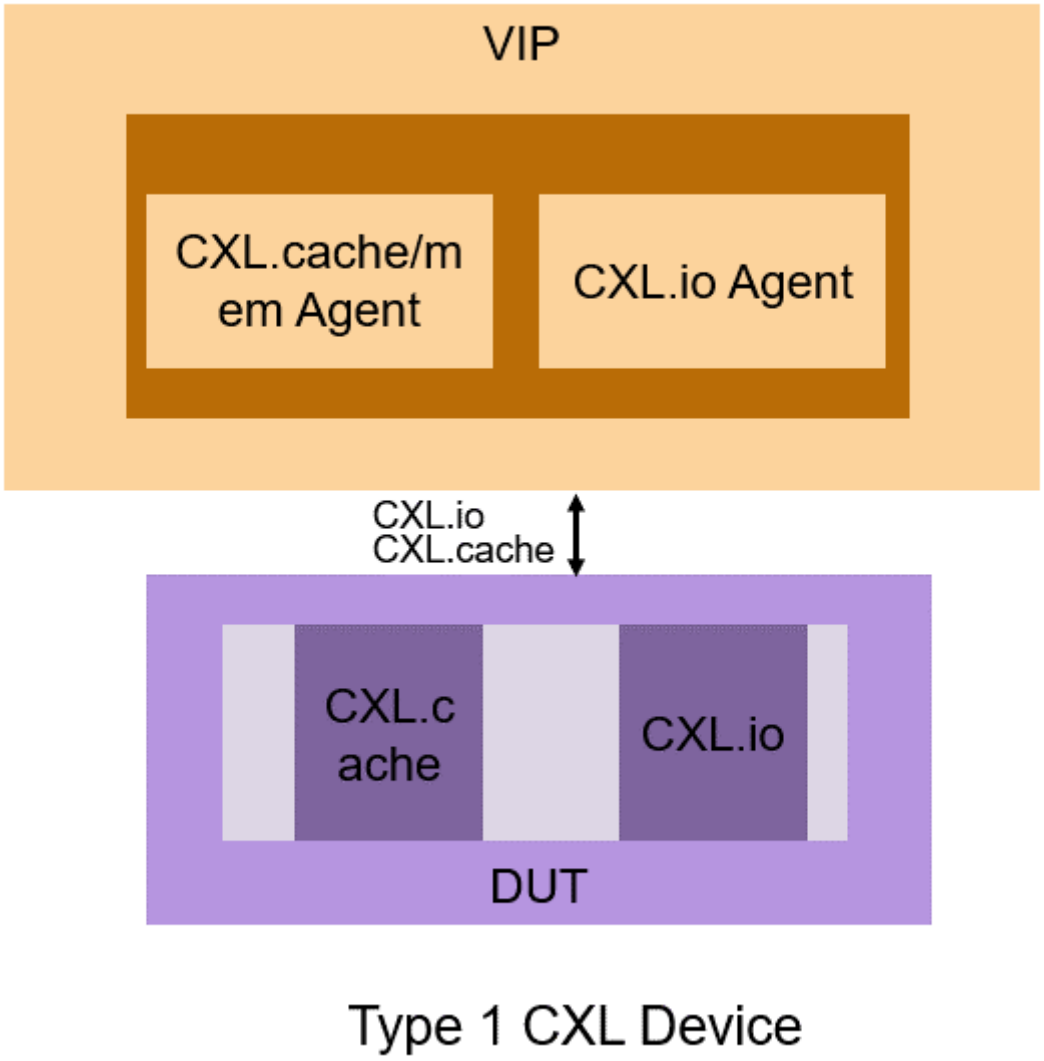
4.1 Verification Requirements - - Supported CXL Device Types

The CXL Subsystem VIP supports all three types of Devices.

- Type 1 Device:
- Type 2 Device:
- Type 3 Device:

4.1.1 CXL Subsystem VIP as Type 1 Device:

The CXL Subsystem VIP can be used as either 'Host' or 'Device' when using it in Type 1 Device mode. You need to do the following required configuration for using the CXL Subsystem VIP based on the requirement.



Type 1 CXL Device

CXL Subsystem VIP supports the Type1 CXL.cache transactions and all CXL.io transactions when using in this mode.

4.1.1.1 Required Configurations

For using CXL Subsystem VIP as Type1 Device do the following configuration.

When using CXL Subsystem VIP as 'Host' -

```
<cust_cfg>.host_cfg.cache_mem_sys_cfg.host_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE1_DEVICE;
```

When using CXL Subsystem VIP as 'Device' -

```
<cust_cfg>.device_cfg.cache_mem_sys_cfg.device_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE1_DEVICE;
```

4.1.1.2 Reference Test Case Available in CXL VIP Example Area

Following test case is available in the example `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys`

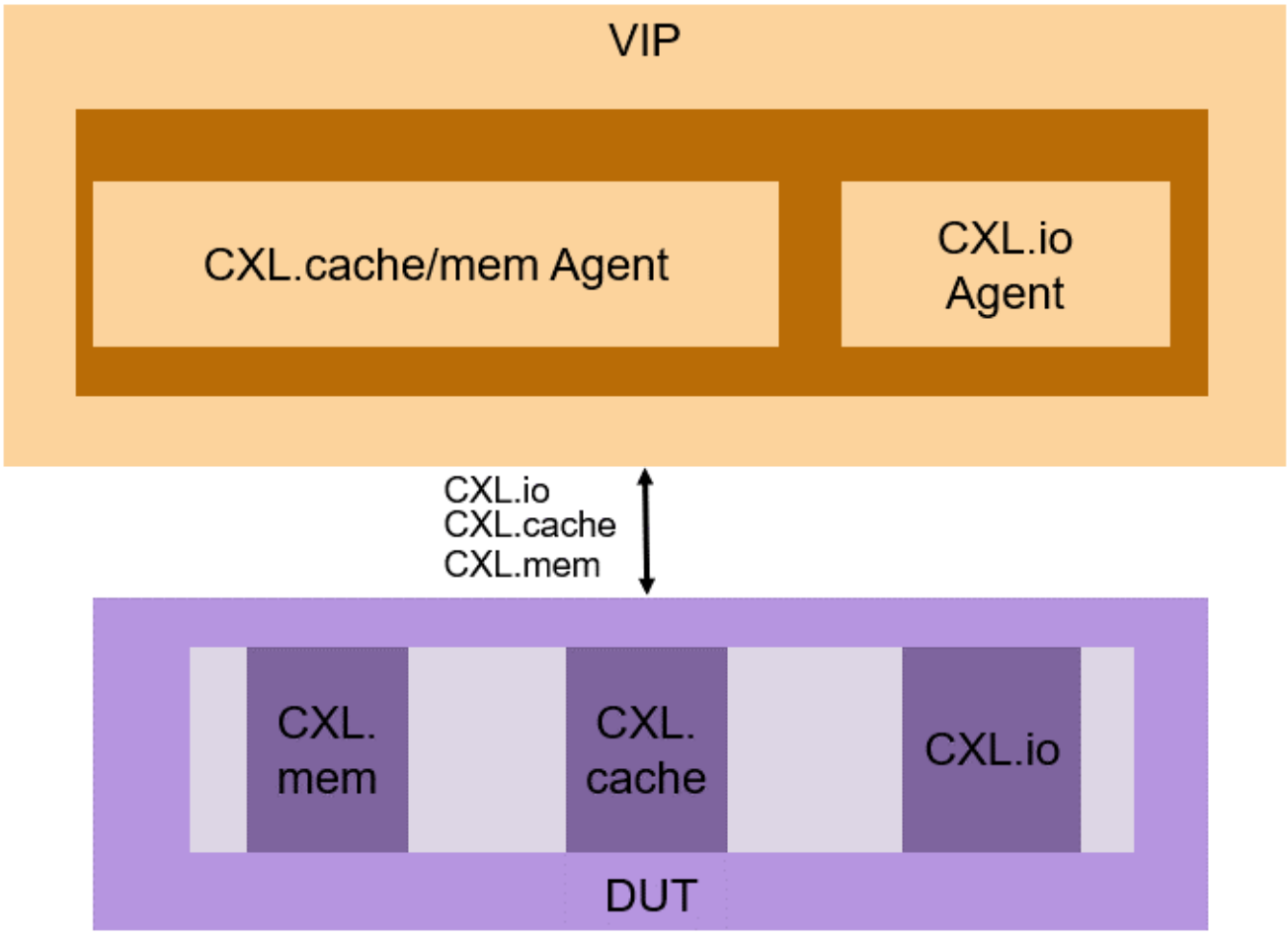
Test name: `ts.cxl_tl_type1_cache_wr_rd.sv`

Description: This test is for Type 1 Device CXL.cache traffic.

Note: The above test is created for VIP back to back setup. You can use it as a reference for creating the test cases related to Type 1 Device.

4.1.2 CXL Subsystem VIP as Type 2 Device

CXL Subsystem VIP can be configured and used as a Type 2 Device. Based on the CXL Subsystem VIP Usage, which is as either Host or Device, the required configuration related to Type 2 Device usage needs to be done.



Type 2 CXL Device

CXL Subsystem supports all three kinds of transactions (CXL.io, CXL.cache, and CXL.mem traffic) when using it in Type 2 Device.

4.1.2.1 Required Configurations

For using CXL Subsystem VIP as Type2 Device do the following configuration.

When using CXL Subsystem VIP as 'Host' -

```
<cust_cfg>.host_cfg.cache_mem_sys_cfg.host_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE2_DEVICE;
```

When using CXL Subsystem VIP as 'Device' -

```
<cust_cfg>.device_cfg.cache_mem_sys_cfg.device_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE2_DEVICE;
```

4.1.2.2 Reference Test Case Available in CXL VIP Example Area

This test case is available in the example cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys

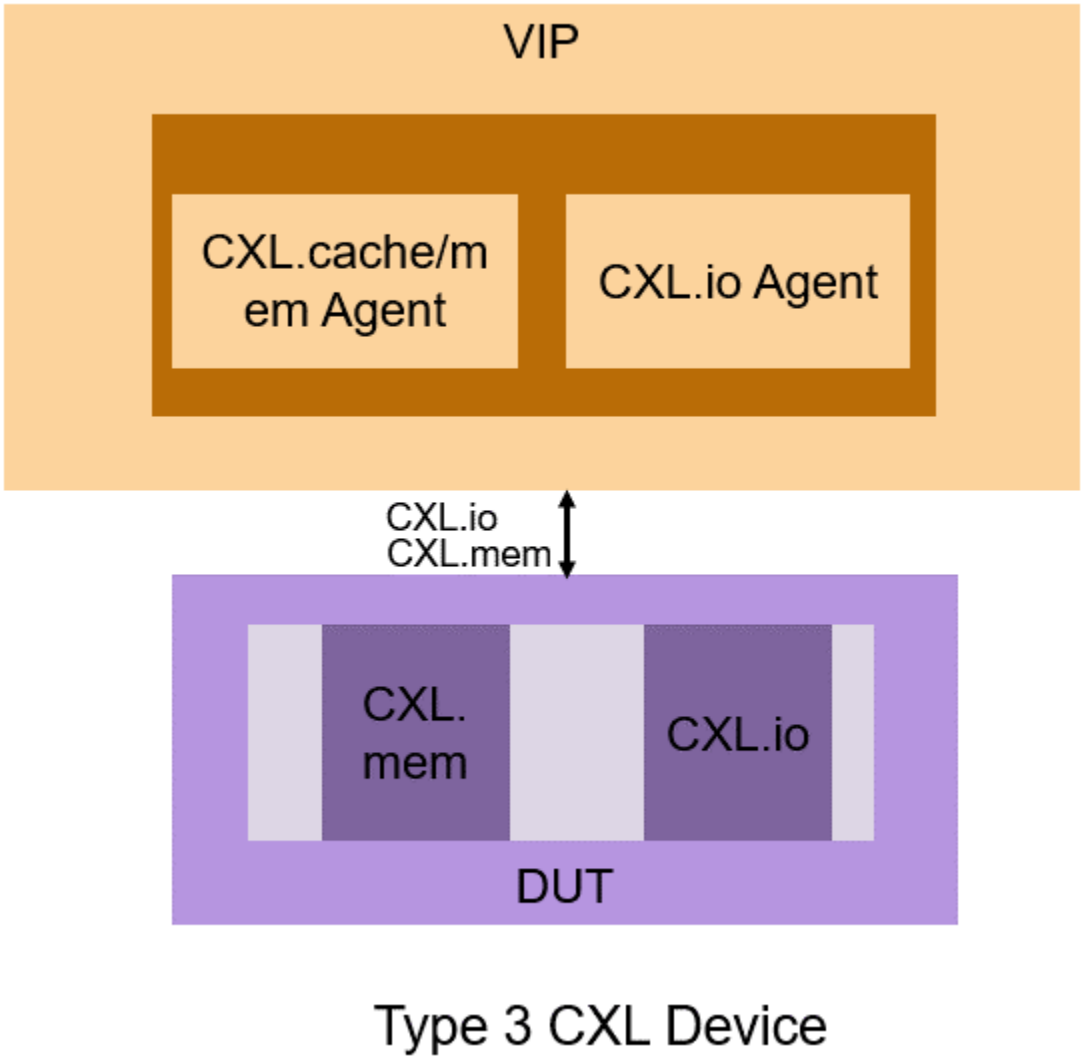
Test name: ts.cxl_tl_random_mem_wr_rd.sv

Description: This test is for CXL.cache and CXL.mem traffic.

The above test is created for CXL.cache and CXL.mem traffic in VIP back to back setup. You can use it as a reference for creating the test cases related to Type 2 Device.

4.1.3 CXL Subsystem VIP as Type 3 Device

CXL Subsystem VIP can be used as either 'Host' or 'Device' when using it in Type 3 Device. Type 3 Device usage is enabled by doing the below required configuration.



Type 3 CXL Device

CXL Subsystem VIP supports the Type3 CXL.mem transactions when using in this mode.

4.1.3.1 Required Configurations

For using CXL Subsystem VIP as Type3 Device do the following configuration.

When using CXL Subsystem VIP as 'Host' -

```
<cust_cfg>.host_cfg.cache_mem_sys_cfg.host_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE3_DEVICE;
```

When using CXL Subsystem VIP as 'Device' -

```
<cust_cfg>.device_cfg.cache_mem_sys_cfg.device_cfg[0].device_type =
svt_cxl_cache_mem_configuration::TYPE3_DEVICE;
```

4.1.3.2 Reference Test Case Available in CXL VIP Example Area

Following test case is available in the example cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys

Test name: ts.cxl_tl_type3_mem_wr_rd.sv

Description: This test is for Type 3 Device CXL.mem traffic.

The above test is created for VIP back to back setup. You can use it as a reference for creating the test cases related to Type 3 Device.

4.2 Verification Requirement - Supported Topologies

The CXL Subsystem VIP can be used as any of the Device mentioned above in various topologies.

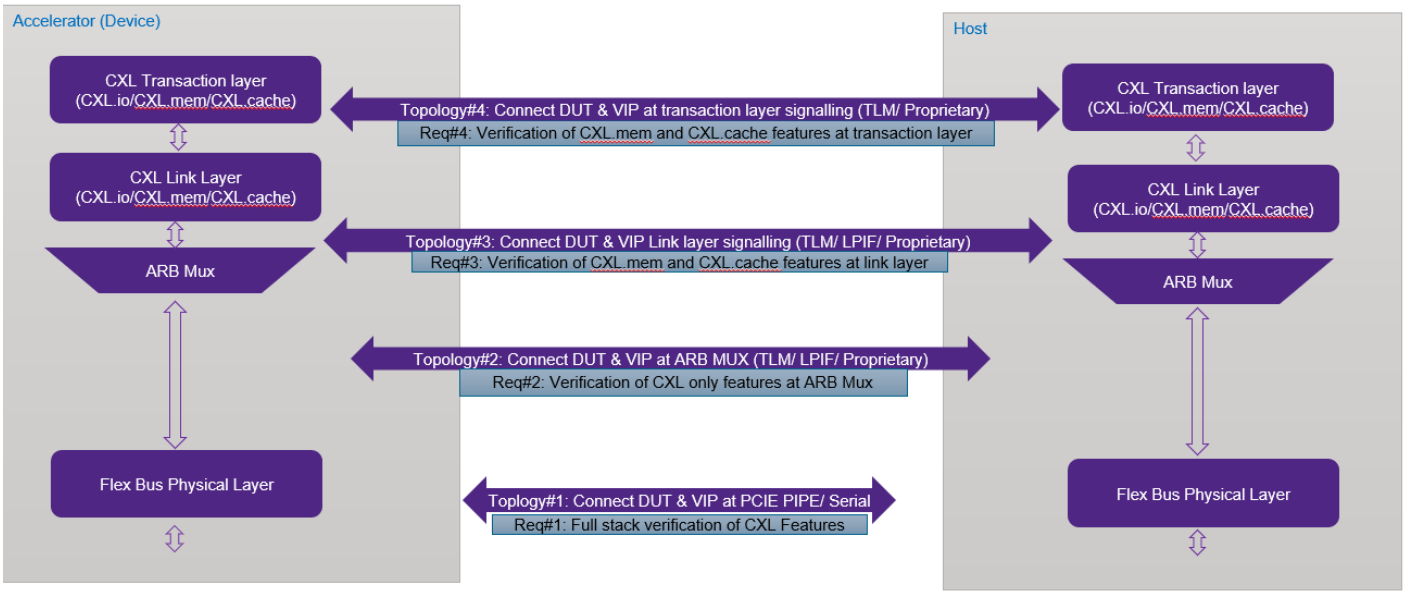
The CXL Subsystem VIP solution supports the following verification requirements by connecting the VIP and the DUT based on the corresponding topology.

Table 4-1 Verification Topologies

Verification Requirement	Topology
Full stack verification of CXL features	Connect DUT and VIP at PCIE PIPE/Serial Interface
Verification of CXL features at ARB/MUX	Connect DUT and VIP at ARB/MUX (TLM/LPIF/Proprietary)
Verification of CXL.mem and CXL.cache features at link layer	Connect DUT and VIP link layer signaling (TLM/LPIF/Proprietary)
Verification of CXL.mem and CXL.cache features at Transaction layer	Connect DUT and VIP transaction layer signaling (TLM/Proprietary)

This diagram shows the topologies supported for the Verification requirement.

Figure 4-1 Verification Requirements and Topologies



All possible topologies with different modes

The above four topologies are further divided into sub-topologies.

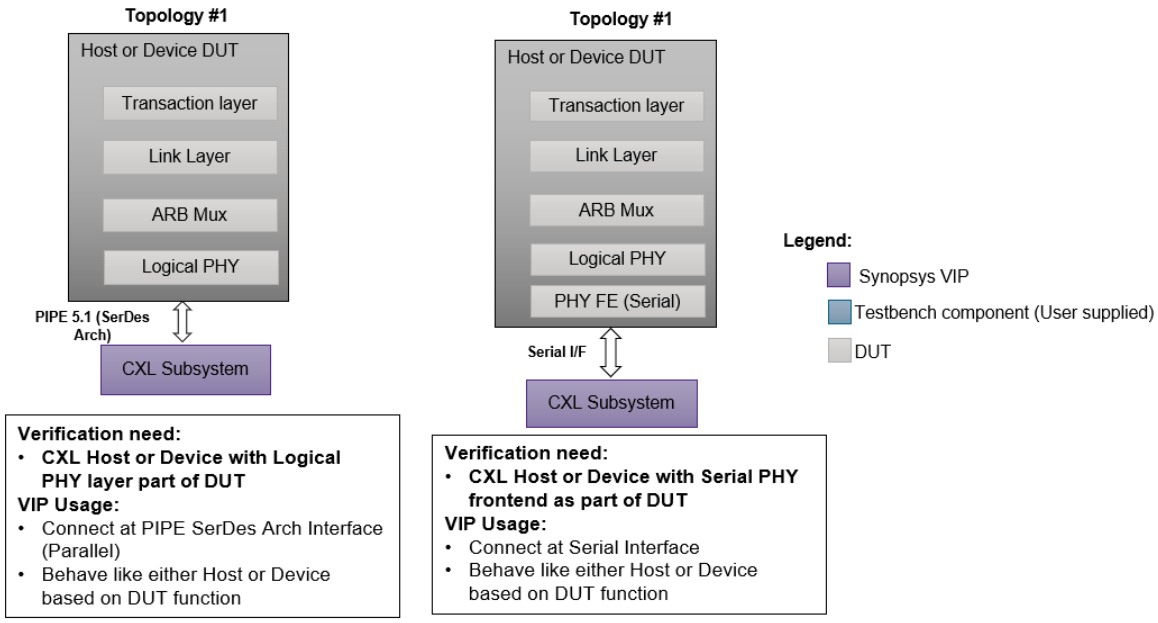
This table shows all supported topologies in different modes.

Topology	Connection at	Mode	Description and Device Types Supported
#1	Full stack	PIPE with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected at the PIPE interface with the DUT. CXL.io, CXL.cache & CXL.mem components are enabled. All Device Types (1, 2 and 3) are supported.
#1	Full stack	PIPE with only CXL.io	CXL Subsystem VIP connected at the PIPE interface with the DUT. Only CXL.io component is enabled.
#1	Full stack	Serial with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected at the Serial interface with the DUT. CXL.io, CXL.cache & CXL.mem components are enabled. All Type 1, 2 and 3 Device types are supported.
#1	Full stack	Serial with only CXL.io	CXL Subsystem VIP connected at the Serial interface with the DUT. Only CXL.io component enabled.
#2	ARB Mux	LPIF with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected at the LPIF interface with the DUT. CXL.io, CXL.cache & CXL.mem components are enabled. All Type 1, 2 and 3 Device types are supported.
#2	ARB Mux	LPIF with only CXL.io	CXL Subsystem VIP connected at the LPIF interface with the DUT. Only CXL.io component is enabled. All Type 1, 2 and 3 Device types are supported.

Topology	Connection at	Mode	Description and Device Types Supported
#2	ARB Mux	LPIF with only CXL.cache/CXL.mem	CXL Subsystem VIP connected at the LPIF interface with the DUT. Only CXL.cache & CXL.mem component is enabled. All Type 1, 2 & 3 Device types are supported.
#2	ARB Mux	TLM with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected with DUT using TLM ports. Only CXL.io component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.
#2	ARB Mux	TLM with only CXL.io	CXL Subsystem VIP connected with DUT using TLM ports. Only CXL.io component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.
#2	ARB Mux	TLM with only CXL.cache/CXL.mem	CXL Subsystem VIP connected with DUT using TLM ports. Only CXL.cache & CXL.mem component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.
#3	Link Layer	LPIF with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected with the DUT using LPIF interface . CXL.io, CXL.cache & CXL.mem components are enabled. All Type 1, 2 & 3 Device types are supported
#3	Link Layer	LPIF with only CXL.io [only TL+DL]	CXL Subsystem VIP connected with the DUT using LPIF interface with only Transaction and Link layer enabled. Only CXL.io component is enabled. All Type 1, 2 & 3 Device types are supported
#3	Link Layer	LPIF with only CXL.cache/CXL.mem	CXL Subsystem VIP connected at the LPIF interface with the DUT. Only CXL.cache & CXL.mem component is enabled. All Type 1, 2 & 3 Device types are supported
#3	Link Layer	TLM with CXL.io and CXL.cache/CXL.mem	CXL Subsystem VIP connected with DUT using TLM ports at link layer. CXL.io, CXL.cache & CXL.mem components are enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.
#3	Link Layer	TLM with only CXL.io	CXL Subsystem VIP connected with DUT using TLM ports at link layer. Only CXL.io component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.

Topology	Connection at	Mode	Description and Device Types Supported
#3	Link Layer	TLM with only CXL.cache/CXL.mem	CXL Subsystem VIP connected with DUT using TLM ports at link layer. Only CXL.cache & CXL.mem component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.
#4	Transaction Layer	TLM with only CXL.cache/CXL.mem	CXL Subsystem VIP connected with DUT using TLM ports at the Transaction layer. Only CXL.cache & CXL.mem component is enabled. All Type 1, 2 & 3 Device types are supported. Note: TLM to DUT signaling interface BFM is provided by user (or) is part of the DUT module.

4.2.1 Topology 1: Connect DUT and VIP at PCIE PIPE/ Serial



CXL Subsystem VIP is providing the following modes of usage under Topology #1

- PIPE with CXL.io and CXL.cache/CXL.mem
- PIPE with only CXL.io
- Serial with CXL.io and CXL.cache/CXL.mem
- Serial with only CXL.io

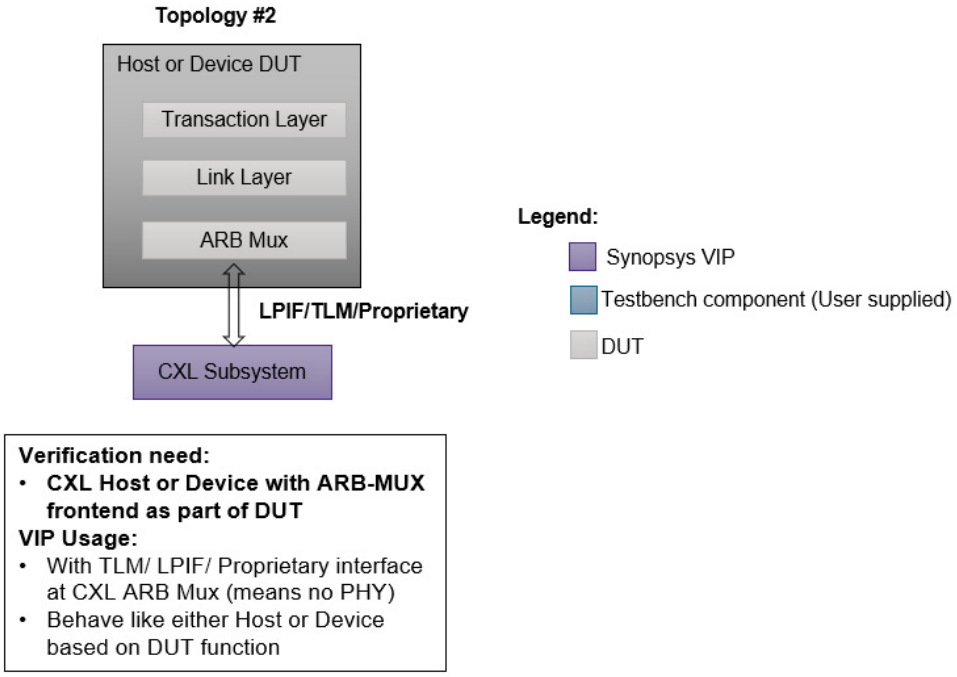
4.2.1.1 Running CXL Subsystem VIP Example Test Cases with Topology #1

Here is the list of valid combinations of topology and compile files to be used for Topology #1

Valid Combinations of Compile and Topology Files for Full Stack Topology

	Full Stack (Serial)
CXL.io traffic test cxl_io_random_mem_wr_rd	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
Type 3 Device CXL.mem test cxl_tl_type3_mem_wr_rd	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
Type 1 Device CXL.cache test cxl_tl_type1_cache_wr_rd	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
Type 2 Device random traffic test cxl_tl_random_mem_wr_rd	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
CXL Warm Reset test: (With 2.0 APN negotiation) cxl_io_warm_reset	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
Type3 device GPF test ts.cxl_io_tl_type3_gpf.sv	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
IDE Sanity test ts.cxl_ide_sanity_test.sv	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi
CXL DOE test ts.cxl_crt_doe_capabilities.sv	compile_snps_vip_pcie_serial.f topology_snps_vip_cxl_b2b.svi

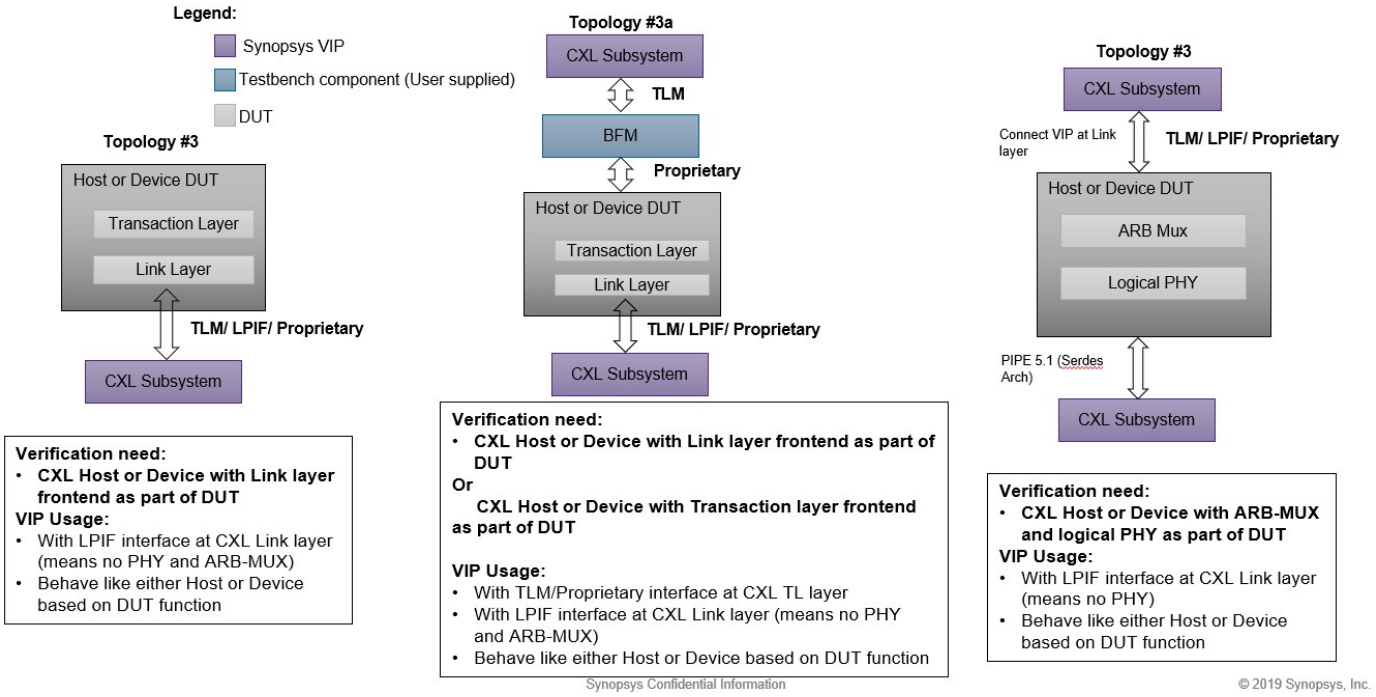
4.2.2 Topology 2: Connect DUT and VIP at ARB MUX (TLM/ LPIF/ Proprietary)



CXL Subsystem VIP is providing the following modes of usage under Topology #2

- LPIF with CXL.io and CXL.cache/CXL.mem
- LPIF with only CXL.io
- LPIF with only CXL.cache/CXL.mem
- TLM with CXL.io and CXL.cache/CXL.mem
- TLM with only CXL.io
- TLM with only CXL.cache/CXL.mem

4.2.3 Topology 3: Connect DUT and VIP Link Layer Signalling (TLM/ LPIF/ Proprietary)



CXL Subsystem VIP is providing these modes of usage under Topology #3

- LPIF with CXL.io and CXL.cache/CXL.mem
- LPIF with only CXL.io [only TL+DL]
- LPIF with only CXL.cache/CXL.mem
- TLM with CXL.io and CXL.cache/CXL.mem
- TLM with only CXL.io
- TLM with only CXL.cache/CXL.mem

4.2.4 Running CXL Subsystem VIP example Test Cases with Topology #3

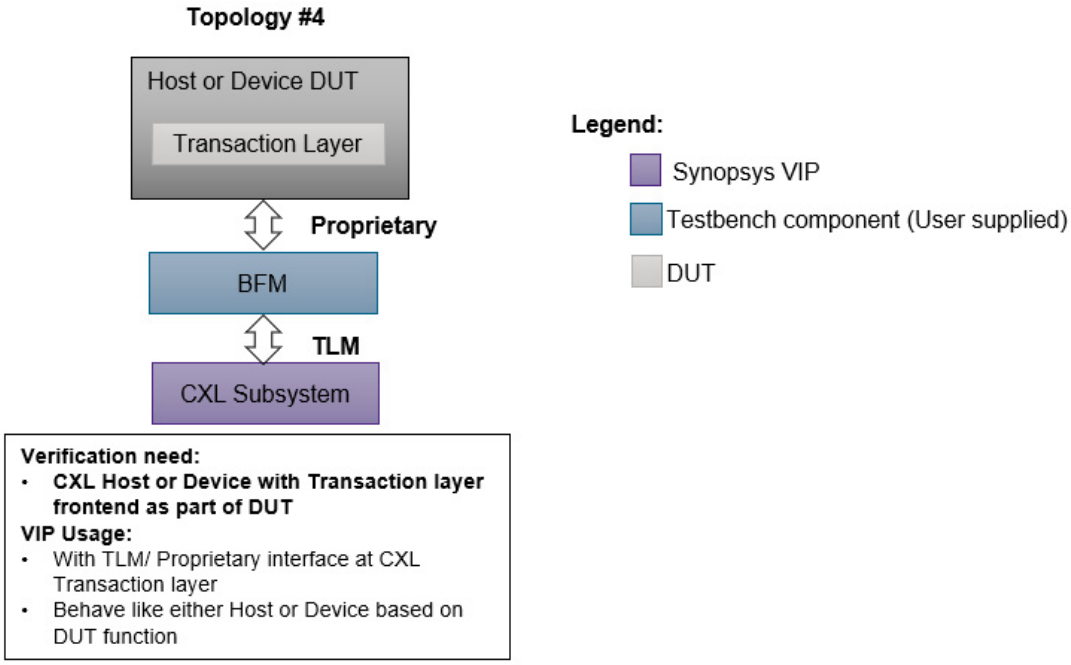
Here is the list of valid combinations of topology and compile files to be used for Topology #3

Valid Combinations of Compile and Topology Files for TL+DL (with and without LPIF) Topology

	TL+DL topology (without LPIF)	LPIF topology
CXL.io traffic test cxl_io_random_mem_wr_rd	compile_snps_vip_cxl_io_tl_dl_only.f topology_snps_vip_cxl_io_dl_b2b.svi	compile_snps_vip_io_lpif.f topology_snps_vip_cxl_b2b_lpif.svi

	TL+DL topology (without LPIF)	LPIF topology
Type 3 Device CXL.mem test cxl_tl_type3_mem_wr_rd	compile_snps_vip_cxl_cache_mem.f topology_snps_vip_cxl_b2b.svi	compile_snps_vip_cache_mem_only_dl_tl_lpif.f topology_snps_vip_cxl_b2b_lpif.svi
Type 1 Device CXL.cache test cxl_tl_type1_cache_wr_rd	compile_snps_vip_cxl_cache_mem.f topology_snps_vip_cxl_b2b.svi	compile_snps_vip_cache_mem_only_dl_tl_lpif.f& topology_snps_vip_cxl_b2b_lpif.svi
Type 2 Device random traffic test cxl_tl_random_mem_wr_rd	compile_snps_vip_cxl_cache_mem.f topology_snps_vip_cxl_b2b.svi	-NA-

4.2.5 Topology 4: Connect DUT and VIP at Transaction Layer Signalling (TLM/Proprietary)



CXL Subsystem VIP is providing the following mode of usage under Topology #4

- TLM with only CXL.cache/CXL.mem

4.2.5.1 Running CXL Subsystem VIP Example Test Cases with Topology #4

Here is the list of valid combinations of topology and compile files to be used for Topology #4

Valid Combinations of Compile and Topology Files for TL Only Topology

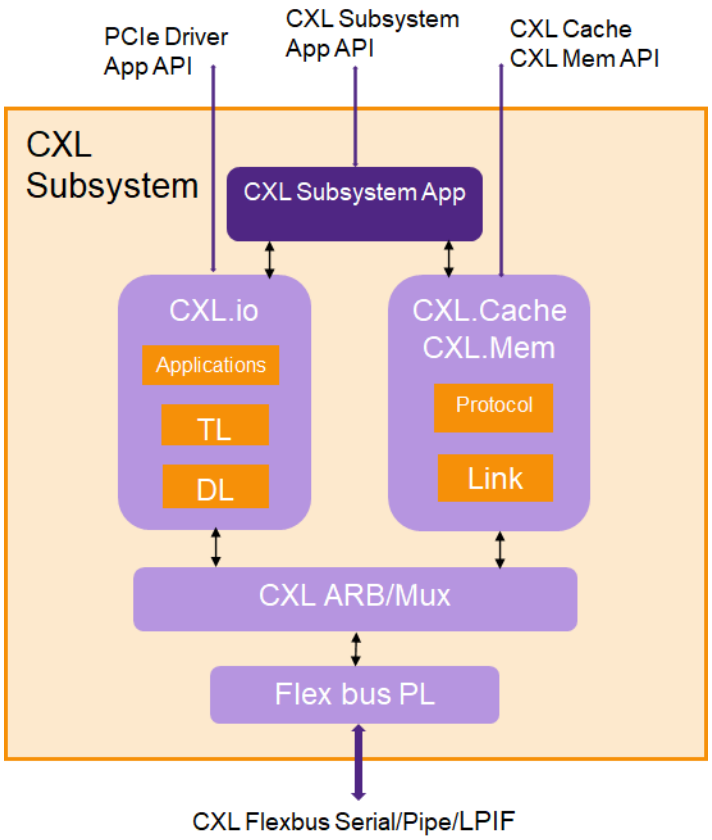
	TL Only topology
CXL.io traffic test cxl_io_random_mem_wr_rd	-NA-
Type 3 Device CXL.mem test	compile_snps_vip_cxl_cache_mem_tl.f
cxl_tl_type3_mem_wr_rd	topology_snps_vip_cxl_b2b.svi
Type 1 Device CXL.cache test cxl_tl_type1_cache_wr_rd	compile_snps_vip_cxl_cache_mem_tl.f topology_snps_vip_cxl_b2b.svi
Type 2 Device random traffic test cxl_tl_random_mem_wr_rd	compile_snps_vip_cxl_cache_mem_tl.f topology_snps_vip_cxl_b2b.svi

5 CXL Application Layer Agent

5.1 Agent Overview

CXL Subsystem has added an application layer which sits on top of CXL.io stack and CXL.Cache/Mem stack. It communicates with internal components of CXL Subsystem and is responsible for following features of an CXL Subsystem.

- Hot Plug
- Power Management(PM VDM) Controller



CXL Subsystem VIP uses the `rx_tlp_out_port` TLM port of PCIe TL and Target App. You are recommended to use the `rx_tlp_peek_port` in their testbench instead of this `put_port`.

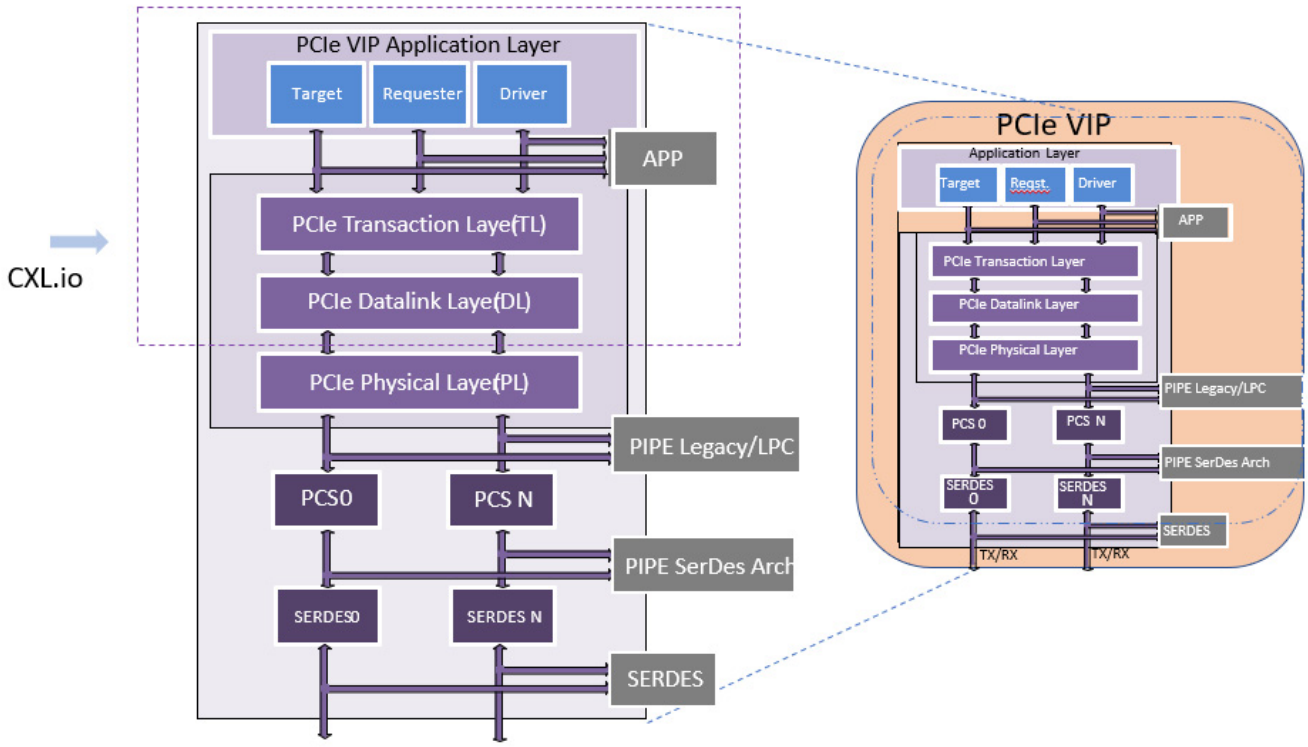
5.2 User API of CXL Subsystem Application

- Provides `svt_cxl_subsystem_app_configuration` class to control various attributes of an application.
- Provides `svt_cxl_subsystem_app_status` class that provides internal status of an application.
- Provides `svt_cxl_subsystem_app_transaction` class for issuing transactions. Also provides associated sequences.
- Provides dedicated `svt_cxl_subsystem_app_vdm_transaction` for PM/Error VDM transactions which is encapsulated inside `svt_cxl_subsystem_app_transaction`.
- Provides `svt_cxl_subsystem_app_service` class for issuing commands. Also, provides associated sequences.
- Provides `svt_cxl_subsystem_app_virtual_sequencer` for issuing transaction and command sequences.

6 CXL.io Agent

6.1 Overview

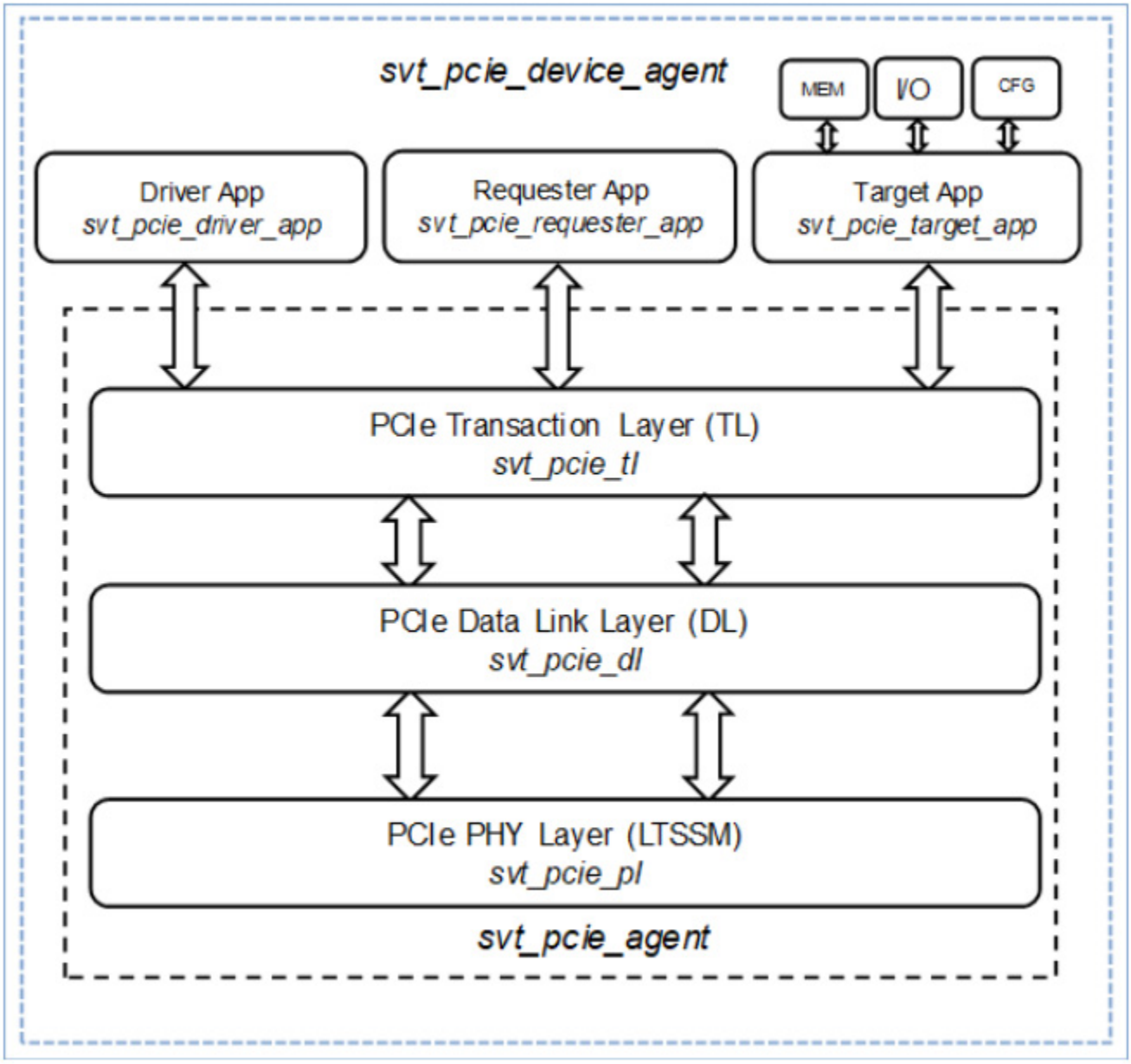
Figure 6-1 CXL.io Agent View



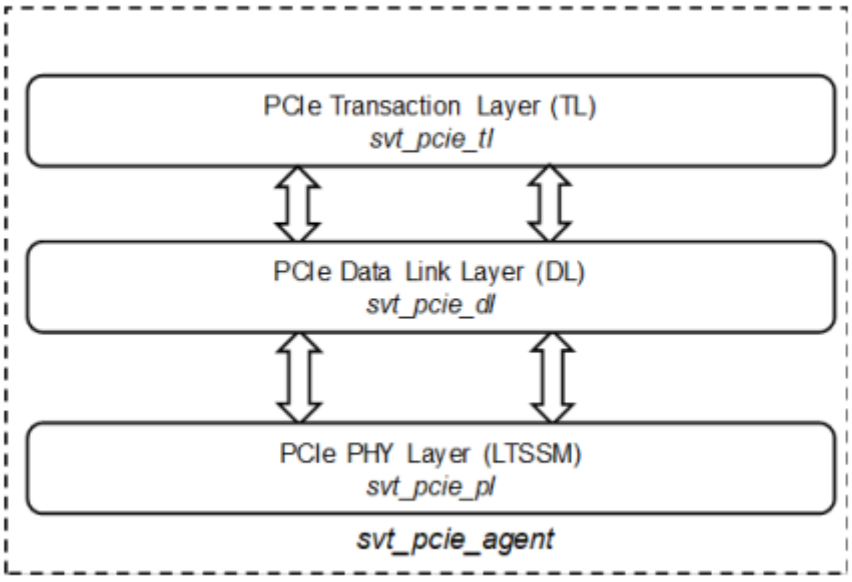
The CXL VIP, at its highest level, encapsulates the PCIe UVM VIP, which is composed of the `svt_pcie_device_agent` class acting as the CXL.io agent. This will generate CXL.io transactions.

CXL VIP leverages the PCIe VIP agent for all CXL.io operations in a similar way where CXL protocol leverages PCIe features for non-coherent load/store interface for IO devices.

`svt_pcie_device_agent` class, encapsulates the PCIe Agent (Class type=svt_pcie_agent)-an application layer that comprises of Driver Application, Requester Application, and Target Application (Memory, I/O, Configuration database). These applications are used to generate the CXL.io traffic through different APIs.



- The Application Layer: The CXL VIP has a layer on top of the CXL.io stack representing the software layer in a real application of the PCIe bus. The application layer is responsible for generating and handling transactions. The application layer of the CXL.io agent is the layer that is typically programmed by the test to generate stimulus or respond to the incoming requests. The application layer has the following blocks that perform specific functions:
  - Driver Application (Type=svt_pcie_driver_app, Instance=driver[0]): The Driver application provides a simple interface that can be used to quickly create CXl.io transaction requests (memory read/write request, I/O read/write requests etc.,). The application deals with driver application transaction objects (svt_pcie_driver_app_transaction) which is an abstract description of the transaction layer packet.
  - Requester Application (Type=svt_pcie_requester_app, Instance=requester): The requester application can be used to generate PCIe memory/read transaction to a remote target. The application can be configured to choose addresses at random (constrained by minimum/maximum configuration parameters) and have varying lengths (again, constrained by minimum/maximum configuration parameters) and generate traffic at a requested bandwidth (again, constrained by minimum/maximum configuration parameters). This application will be useful where a background exerciser of write/read request is required.
  - Target Application (Type=svt_pcie_target_app, Instance=target[0]): The target application is the block that automatically responds to various inbound CXL.io requests. Read transaction requests can be optionally broken up into multiple completions, potentially interleaved with other read completions by configuration. The target application has auxiliary blocks that represent the completion memory used while generating completions. These blocks include Memory Target, IO Target and Configuration Database. The blocks comprise of a sparse memory allowing a wide variety memory/IO addresses/registers to be accessed by a DUT.
  - Memory Target (Type=svt_pcie_mem_target, Instance=mem_target): The memory target is the PCIe VIP's sparse memory model used to store write data and return read data to the incoming memory requests. This sparse memory model responds to a wide variety of addresses (32-bit and 64-bit) when accessed by a requester. The memory target has APIs to write into its sparse memory through backdoor or read from its sparse memory through backdoor to meet different kinds of testing requirements.
  - I/O Target (Type=svt_pcie_io_target, Instance=io_target): The I/O target is the PCIe VIP's sparse memory model used to store write data and return read data to the incoming I/O requests. This sparse memory model responds to a wide variety of addresses (32-bit and 64-bit) when accessed by a requester. The I/O target has APIs to write into its sparse memory via backdoor or read from its sparse memory via backdoor to meet different kinds of testing requirements.
  - Configuration Database (Type=svt_pcie_cfg_database, Instance= svt_pcie_target_app::cfg_database): The configuration database is the sparse memory model of PCIe VIP that is used to store write data and return read data to the incoming configuration requests. This sparse memory model responds to a wide variety of addresses (type 0, type 1, extended capability registers, and so on) when accessed by a requester. The configuration database has APIs to write into its sparse memory through backdoor or read from its sparse memory through backdoor to meet different kinds of testing requirements.



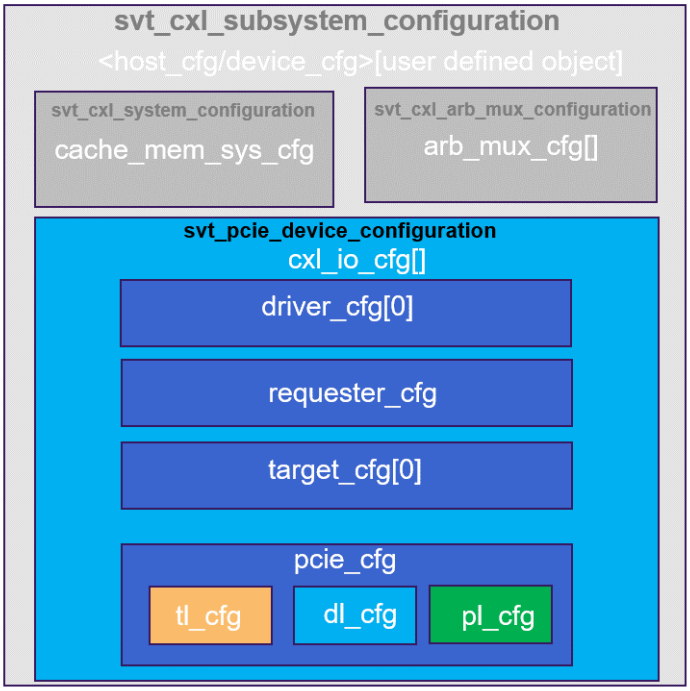
- PCIe Agent: The PCIe Agent encapsulates the UVM drivers that represents the Transaction Layer, the Data-link Layer, and the Physical Layer of the PCIe protocol stack. The `svt_pcie_agent` class represents this encapsulation in the PCIe VIP. The PCIe agent class is instanced as `pcie_agent` within `svt_pcie_device_agent`. The agent class consists of the following layers:
  - The Transaction Layer (Type=svt_pcie_tl, Instance=pcie_tl): The `svt_pcie_tl` class defines functions of the transaction layer (TL) in the PCIe VIP. The applications transfer transaction requests to the TL for transmission. The TL composes the transaction layer packet (TLP) and hands it down to the data-link layer located below it. The transaction layer also receives TLPs from the data-link layer which gets routed up to the correct application. It is also possible for the test to interface directly with the TL to generate TLPs. But it is recommended to use the application layers.
  - The Data-link Layer (Type=svt_pcie_dl, Instance=pcie_dl): The `svt_pcie_dl` class defines the functions of the data-link (DL) layer in the PCIe VIP. The TL transfers TLPs to the DL to be framed with a sequence number and a CRC and ensure the remote receiver receives the packet without any errors. The DL also performs the other standard functions of link management using data-link layer packets (DLLPs).
  - Physical Layer (Type=svt_pcie_pl, Instance=pcie_pl): The `svt_pcie_pl` class defines the functions of the physical layer (PL) in the PCIe VIP. The PL breaks down packets it receives (from the DL) into symbols and encodes them as per the specification before transmission. It also composes packets from data received on the receive data lanes and sends them back to the DL. It also performs other functions to maintain the link such as the LTSSM and functions for low power and so on. This includes Flex Bus also. Refer Chapter 3 for more details.

6.2 Configuration

The CXL.io Agent is configured using an object of class `svt_pcie_device_configuration` (cxl_io_cfg[] as the handle) defined within the top level configuration class `svt_cxl_subsystem_configuration` of CXL VIP. This class has other class objects defined within it to form a hierarchy that corresponds to the hierarchy inside the CXL.io Agent.

Note Refer the Section 3.4 for top level configuration class details. This chapter assumes that you have already created environment.

Figure 6-2 CXL.io Configuration Hierarchical View



svt_pcie_device_configuration (cxl_io_cfg[])

|

|--------> driver_cfg[] (type=svt_pcie_driver_app_configuration)

|

|--------> requester_cfg (type=svt_pcie_requester_app_configuration)

|

|--------> target_cfg[] (type=svt_pcie_target_app_configuration)

|

|--------> pcie_cfg (type=svt_pcie_configuration)

|
|--------> tl_cfg (type=svt_pcie_tl_configuration)
|
|--------> dl_cfg (type=svt_pcie_dl_configuration)
|
|--------> pl_cfg (type=svt_pcie_pl_configuration)

This class is comprised of direct variables and class objects that are used to configure other agents/drivers that are part of the CXL.io agent.

Example with the usage:

- How to configure Polling Active timeout for Host?

  You can configure the Polling Active LTSSM state timeout for the CXL.io agent.

```
/** Create the top level configuration handle **/
svt_cxl_subsystem_link_configuration cust_cfg;
/** Configuring the Polling Active LTSSM state timeout for ith CXL.io agent **/
// Sample Format ->
//<cust_cfg>.<host_cfg/device_cfg>.cxl_io_cfg[0].pcie_cfg.pl_cfg.<pl_cfg_attrbute>
cust_cfg.host_cfg.cxl_io_cfg[0].pcie_cfg.pl_cfg.polling_active_timeout_ns = 240000;
```

  You can refer the `svt_cxl_subsystem_base_test.sv` and `svt_cxl_subsystem_base_env.sv` in the installed unified example for more details on the usage.

Only the top-level configuration handle is user-defined. After that, the instance handle names remains fixed and must not be changed by the user.

6.3 Status

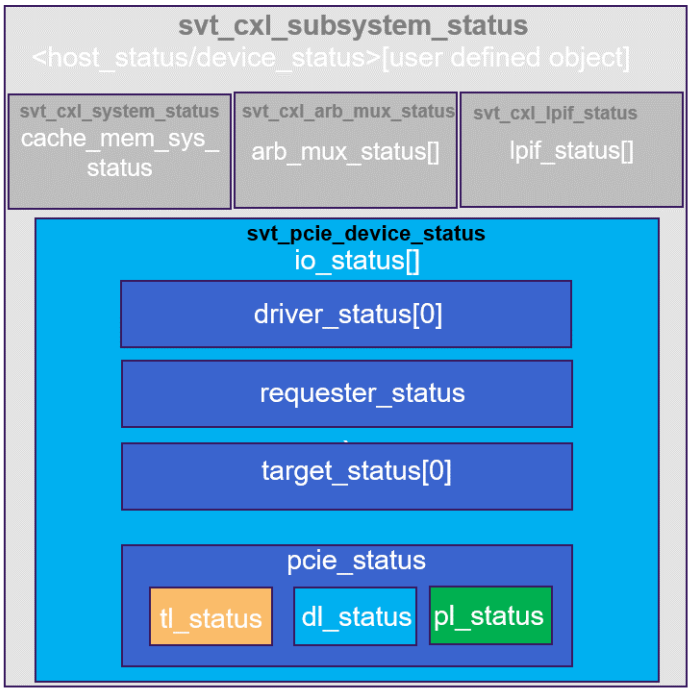
Status is used to get run time information of PL/DL/TL layer aspects of a CXL.io agent. The CXL.io Agent provides a set of state values representing the status of its sub-components at anytime in the test simulation.

`svt_cxl_subsystem_status` is the CXL Subsystem 'top level' status class. It encapsulates the IO status class object corresponding to the CXL.io agents present in the CXL Subsystem.

Refer Section 3.5 for top level status class details. This chapter assumes the you have already created the environment.

The `svt_pcie_device_status` class (io_status[] as handle) will provide us the CXL.io agent status.

Figure 6-3 CXL.io Status Hierarchial View



svt_pcie_device_status (io_status[])

|

|--------> driver_status[] (type = svt_pcie_driver_app_status)

|

|--------> requester_status (type= svt_pcie_requester_app_status)

|

|--------> target_status [] (type = svt_pcie_target_app_status)

|

|--------> pcie_status (type = svt_pcie_status)

|

|--------> tl_status (type = svt_pcie_tl_status)

|

|--------> dl_status (type = svt_pcie_dl_status)

|

|--------> pl_status (type = svt_pcie_pl_status)

Example usage:

How to check Host APN status?

```
/** Instances of Subsystem status */
  svt_cxl_subsystem_status host_status;
/** Check the if the APN handshake is successful for CXL.io agent in
cxl_test_report_phase**/
// Sample Format ->
//<status_class_obj_handle>.io_status[0].pcie_status.pl_status.<status_attribute>
if
(host_status.io_status[0].pcie_status.pl_status.alternate_protocol_negotiation_status
== svt_pcie_pl_status::APN_SUCCESSFUL)
`svt_note(method_name, "APN Successful for Host");
```

You can refer the `svt_cxl_subsystem_base_test.sv` and `svt_cxl_subsystem_base_env.sv` in the installed unified example for more details on the usage.

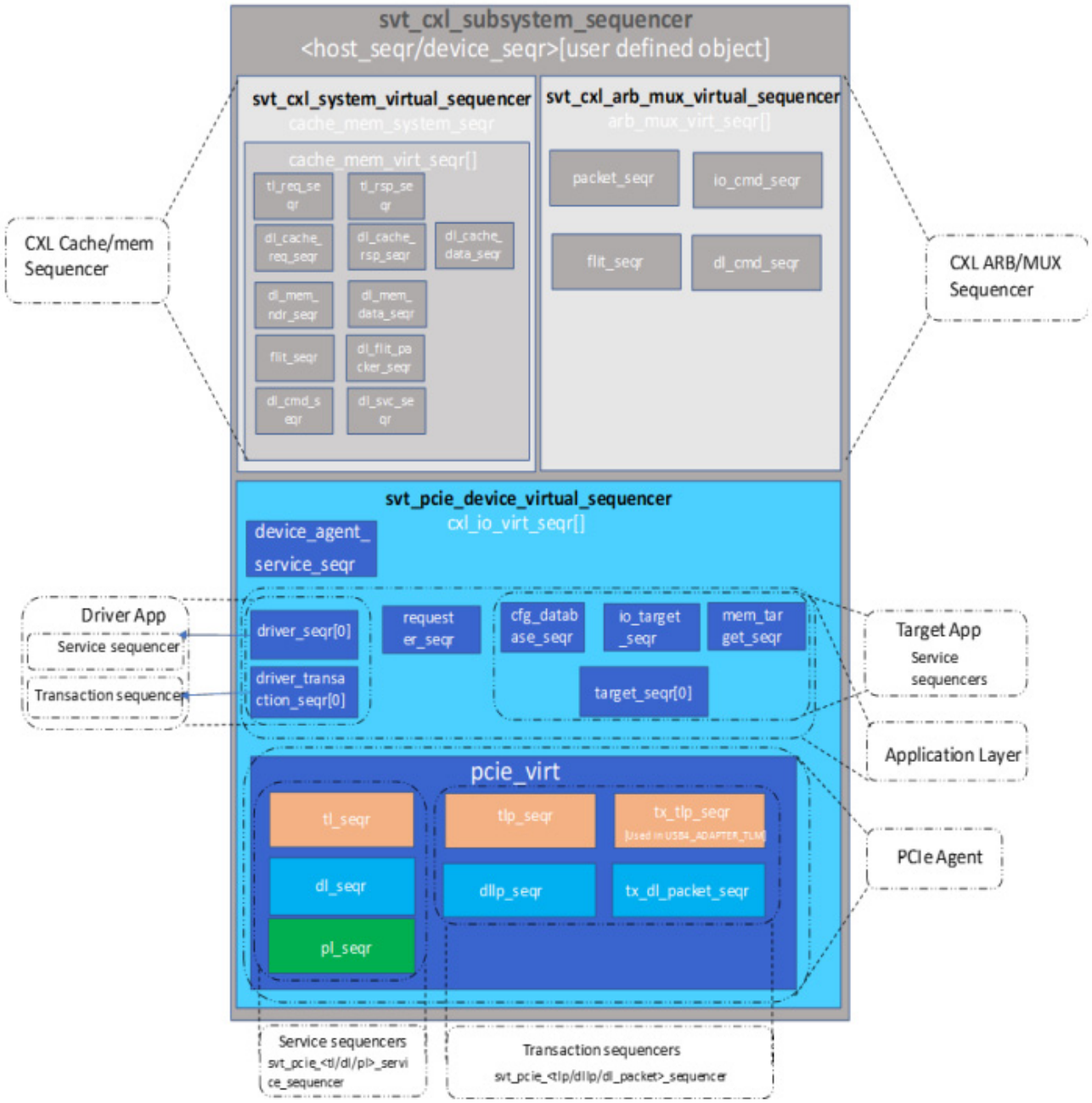
Note Only the top-level status handle is user-defined. After that, the instance handle names remain fixed and must not be changed by the user.

6.4 Sequencers

The CXL.io Agent class (`svt_pcie_device_agent`) has different UVM sequencer objects to schedule transaction requests or service requests on any of its subcomponent Drivers. A UVM sequencer is an arbiter that controls the transaction flow from multiple stimulus generators. The sequencers communicate with drivers using the TLM interfaces.

Note Refer Section 3.3 for top level sequencer details. This chapter assumes that you have already created environment.

For CXL.io, `svt_pcie_device_virtual_sequencer`(`cxl_io_virt_seqr[]` as handle) class handle is present inside `svt_cxl_subsystem_sequencer` (Top level virtual sequencer).



Using the `cxl_io_virt_seqr[]`, you can use all the sequencers present for the CXl.io agent.

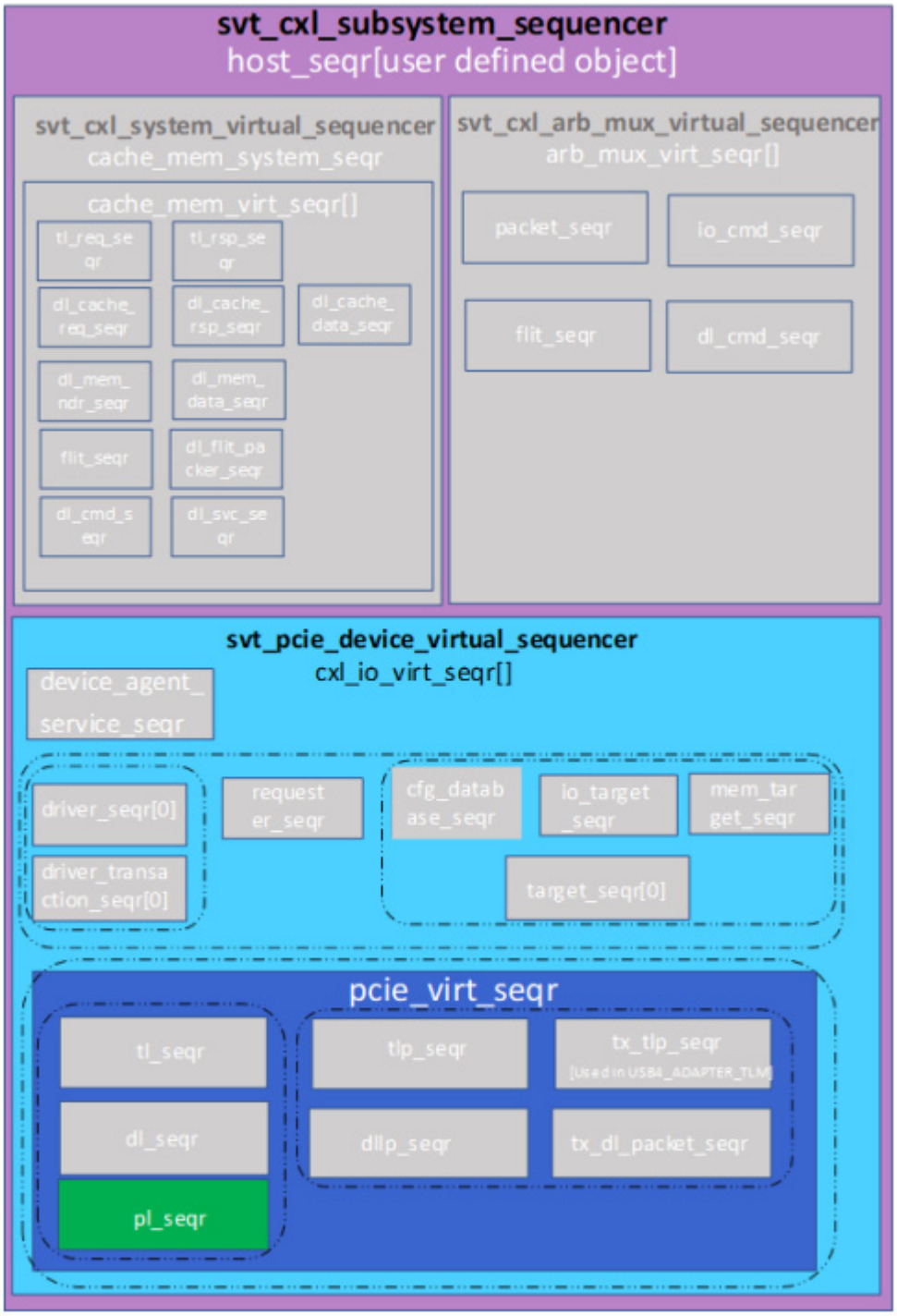
Example Usage:

How to access the sequencer hierarchy for pl_seqr?

Assuming you already have created the handle for top-level virtual sequencer as host_seqr, for using any sequencer of CXL.io Agent, you can use `cxl_io_virt_seqr[]`. It contains of `pcie_virt_seqr` for pl/dl/tl protocol layer sequencers. Inside `pcie_virt_seqr`, you will have `pl_seqr`.

```
<p_sequencer/top_level_seqr_handle>.cxl_io_virt_seqr[0].pcie_virt_seqr.pl_seqr
```

Refer this figure and follow the colored part.



Note Only the top-level sequencer handle is user-defined. After that, the instance handle names remain fixed and must not be changed by the user.

6.5 Sequences

Many sequences are present for CXL.io agent. They can be divided in two categories:

- Transaction sequences
- Service sequences.

Refer the HTML class reference for more information.

Example Usage:

How do I enable link up or send a Memory TLP?

```
/** Creating object handle of sequence class */
   /** PCIe PL Link sequences */
  svt_pcie_pl_service_set_phy_en_sequence pl_link_en_seq;
  /** PCIE DL Link Enable sequence */
  svt_pcie_dl_service_set_link_en_sequence dl_link_en_seq;
  /** PCIE Driver APP Mem traffic sequence for Mem Write/Read transactions */
  svt_pcie_driver_app_mem_request_sequence mem_wr_seq, mem_rd_seq;


……
  bit[32:0] local_addr;
  bit[9:0] local_len;
……
/** Enabling PHY for CXL.io agent */
`svt_uvm_do_on_with(pl_link_en_seq,
p_sequencer.cxl_io_virt_seqr[0].pcie_virt_seqr.pl_seqr, {phy_enable == 1'b1;})

/** Enabling DL link for CXL.io 0th agent */
`svt_uvm_do_on_with(dl_link_en_seq,
p_sequencer.cxl_io_virt_seqr[0].pcie_virt_seqr.dl_seqr, {enable == 1'b1;})

/** Initiating MEM_WR transaction for CXl.io agent*/
`svt_uvm_create_on(mem_wr_seq,p_sequencer.cxl_io_virt_seqr[0].driver_transaction_seqr[0
])
`svt_uvm_rand_send_with(mem_wr_seq, {transaction_type ==
                                    svt_pcie_driver_app_transaction::MEM_WR;
                                    address inside {[32'h4000_0000 : 32'h8000_0000]};
                                    length == 10;
                                                                        first_dw_be
== 4'b1111;
                                    last_dw_be   == 4'b1111;
                                    })
 local_addr = mem_wr_seq.address;
 local_len = mem_wr_seq.length;
```

```
/** Initiating MEM_RD transaction for CXl.io agent */
`svt_uvm_create_on(mem_rd_seq,p_sequencer.cxl_io_virt_seqr[0].driver_transaction_seqr[0
])
`svt_uvm_rand_send_with(mem_rd_seq, {transaction_type ==
                                svt_pcie_driver_app_transaction::MEM_RD;
                                  address == local_addr;
                                  length == local_len;
          first_dw_be  == 4'b1111;
                                  last_dw_be   == 4'b1111;
})
```

Here p_sequencer represents the top-level virtual sequencer (svt_cxl_subsystem_sequencer). Refer Section 3.3 for more details.

6.6 APIs

There are various APIs that are available to ease the development process. All the APIs are present in the `svt_cxl_subsystem_virtual_api_collection_sequence` class. This sequence is for API based traffic generation for complete Subsystem.

As an example, there are APIs to generate memory request for CXL.io such as `generate_cxl_io_mem_rd`, `generate_cxl_io_mem_wr` and so on.

For complete list of available APIs and their input parameters, refer the HTML class reference:

`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html`

Example with the usage:

How do I activate CXL.io link using APIs?

/** In base sequence, this sequence handle is set with vip sequencer */

```
  svt_cxl_subsystem_virtual_api_collection_sequence vip_seq;
```

/** Extend your custom sequence from base sequence and use the API */

```
/** Blocking API till CXL.io link up */
// Sample format -> <sequence_handle>.<api_name>(input/output parameters);
  vip_seq.activate_cxl_io_link(0); // input int io_link_num
```

Refer HTML Class reference for more details on this API:

`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html#item_activate_cxl_io_link`

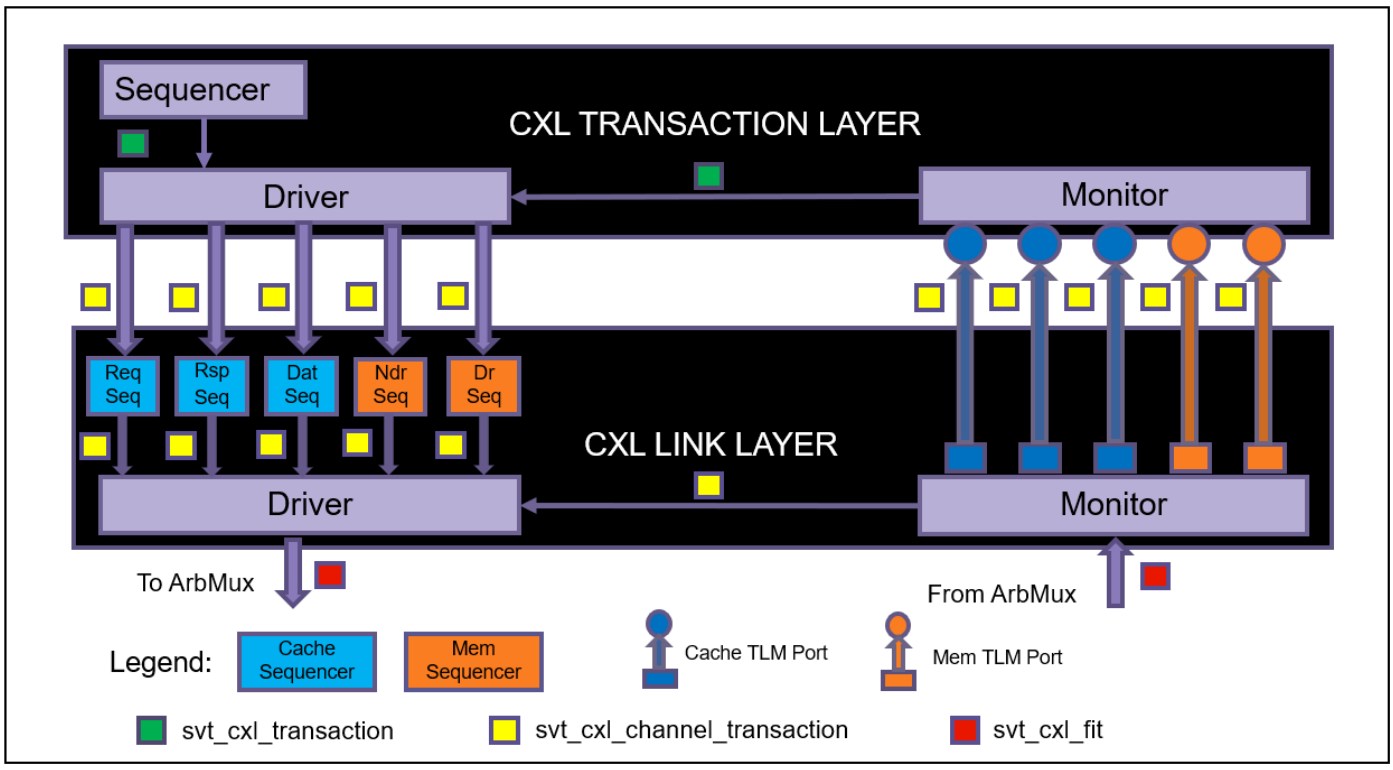
7 CXL.mem and CXL.cache Agent

7.1 Overview

7.1.1 CXL.mem and CXL.cache Agent

The CXL Subsystem environment, `svt_cxl_subsystem_env` contains the `svt_cxl_env` class which comprises the CXL Cache/mem Agent - is the CXL Cache/mem component with the Transaction Layer and Link Layer.

Figure 7-1 CXL.cache/mem Agent View



The following component is present inside the svt_cxl_env.

- svt_cxl_agent

`svt_cxl_agent` encapsulates the Transaction Layer and Link Layer of CXL Cache/mem component. This comprises of Driver, Monitor, and Sequencer at Transaction Layer level and Data Link Layer level.

Components present inside the svt_cxl_agent

- CXL Transaction Layer Driver (Type= svt_cxl_tl, Instance= tl_driver):

  The `svt_cxl_tl_monitor` class defines the functions of the CXL Cache/mem Transction Layer (TL) Monitor in the CXL Subsystem VIP. Once the CXL transaction on the bus is complete, the completed sequence item is provided to the analysis node of monitor, which can be used by the testbench.

- CXL Transaction Layer Monitor (Type= svt_cxl_tl_monitor, Instance= tl_mon):

  The `svt_cxl_tl_monitor` class defines the functions of the CXL Cache/mem Transction Layer (TL) Monitor in the CXL Subsystem VIP. Once the CXL transaction on the bus is complete, the completed sequence item is provided to the analysis node of monitor, which can be used by the testbench.

  For the available Analysis TLM Ports at this level, refer the `svt_cxl_tl_monitor` class in the HTML Class reference.

  Link to HTML Class reference:

  `$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/monitor/class_svt_cxl_tl_monitor.html`

- CXL Transaction Layer Sequencers

  You can provide CXL sequences to the CXL Transaction Layer sequencer. All the available CXL Transaction Layer sequencers are discussed in the Section 6.4 Sequencers.

- CXL Data Link Layer Driver (Type= svt_cxl_dl, Instance= dl_driver):

  The `svt_cxl_dl` class defines the functions of the CXL Cache/mem Data Link Layer (DL) Driver in the CXL Subsystem VIP. Within the Data Link Layer, the CXL driver gets sequences from the CXL sequencer. The CXL driver then drives the CXL transactions on the CXL node.

- CXL Data Link Layer Monitor(Type= svt_cxl_dl_monitor, Instance= dl_mon):

  The `svt_cxl_dl_monitor` class defines the functions of the CXL Cache/mem Data Link Layer (DL) Monitor in the CXL Subsystem VIP. Once the CXL transaction on the bus is complete, the completed sequence item is provided to the analysis node of monitor, which can be used by the testbench.

  For the available Analysis TLM Ports at this level, refer the `svt_cxl_dl_monitor` class in the HTML Class reference.

  Link to HTML Class reference:

  `$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/monitor/class_svt_cxl_dl_monitor.html`

- CXL Data Link Layer Sequencers

  You can provide CXL sequences to the CXL Data Link Layer sequencer. All the available CXL Data Link Layer sequencers are discussed in the section 6.4 Sequencers.

7.1.2 Classes and Applications for Using CXL.mem/cache Component

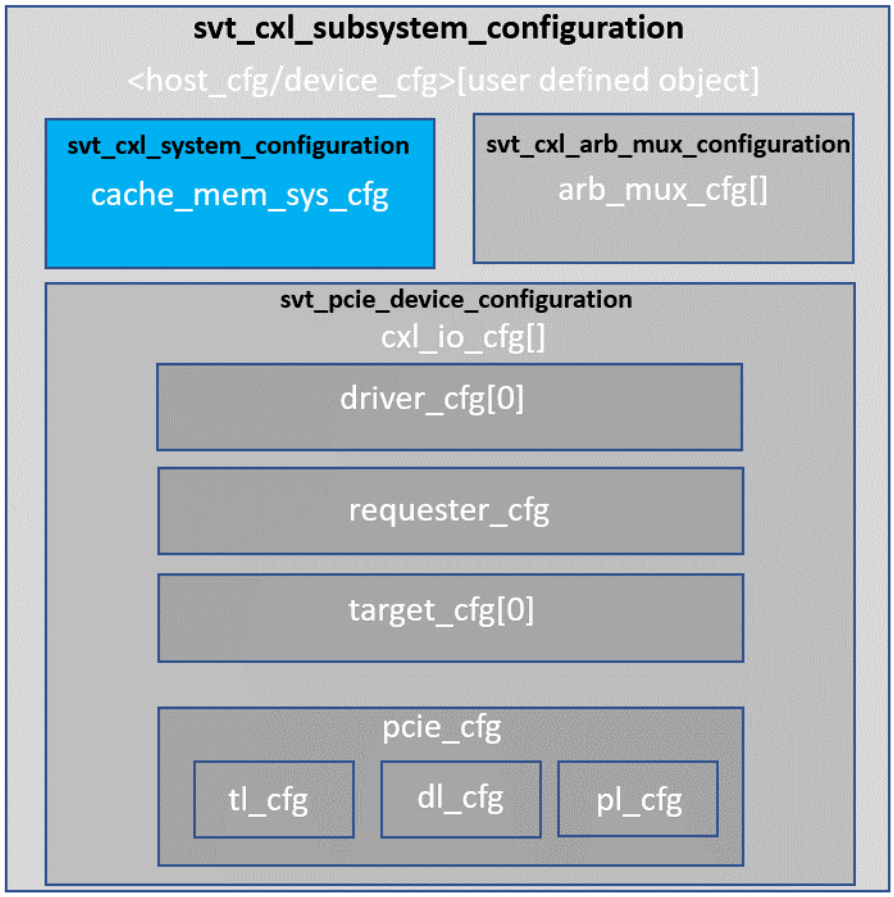
The following classes have members and tasks related to CXL.cache/mem component features and operation.

- Component Class svt_cxl_env: Encapsulates the CXL.cache/mem
- Configuration class svt_cxl_system_configuration: This class contains members to configure the behavior of the CXL.Cache/mem Agent.
- Status class svt_cxl_system_status: Used for returning the status related to CXL.Cache/mem Agent to the testbench.

7.2 Configuration

The CXL.Cache/mem Agent is configured using an object of class `svt_cxl_system_configuration` (cache_mem_sys_cfg is the handle) defined within the top level configuration class `svt_cxl_subsystem_configuration` of CXL Subsystem VIP.

Note Refer the Section 3.4 for top level configuration class details. This chapter assumes that you have already created the environment.




```
svt_cxl_system_configuration contains instances of the class
svt_cxl_cache_mem_configuration
```

```
svt_cxl_cache_mem_configuration is used to configure the CXL.Cache/mem agent.

svt_cxl_system_configuration (cache_mem_sys_cfg)
                             |
                             |--------> host_cfg/device_cfg (<host_cfg/device_cfg>)
```

For all the available configuration attributes and functions related to CXL.Cache/mem agent, refer to the HTML Class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/configuration/class_svt_cxl_cache_mem_configuration.html
```

Example usage:

How to configure the DL Credit return priority between CXL.cache Vs CXL.mem for Host?

```
svt_cxl_subsystem_link_configuration cust_cfg;
/** Configuring the CXL.Cache/mem agent **/
// Sample Format ->
//<cust_cfg>.<host_cfg/device_cfg>.cache_mem_sys_cfg.<host_cfg[0]/device_cfg[0]>.<Cache
/mem configuration attribute>
```

/** Setting DL Credit return priority mode as PROT_PRIORITY_MODE - This indicates that the protocol selected for credit return on the next outgoing flit will be based on the priority of protocol. If the desired protocol (CXL.Cache or CXL.mem) has at least 1 credit, it gets priority over the other.**/

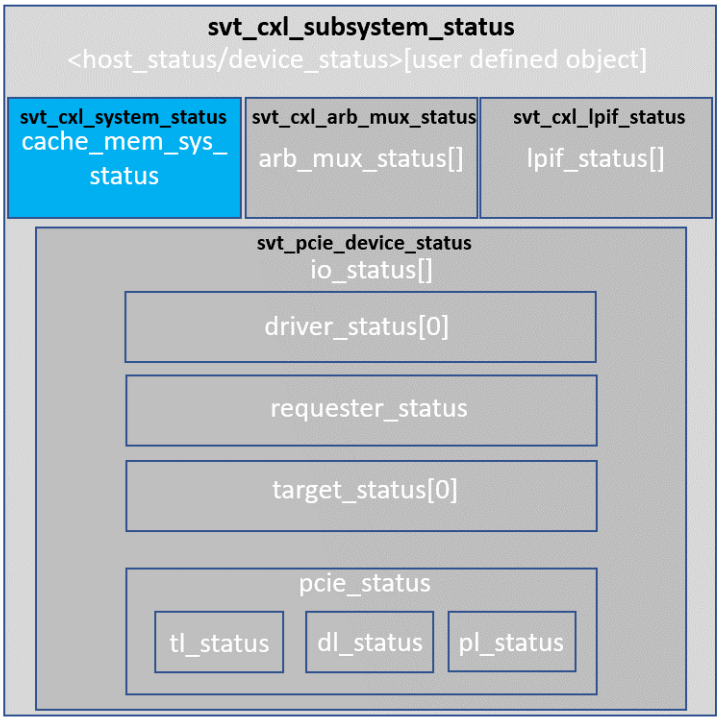
```
cust_cfg.host_cfg.cache_mem_sys_cfg.host_cfg[0].dl_credit_return_priority_mode =
svt_cxl_cache_mem_configuration::PROT_PRIORITY_MODE ;
```

Only the top-level configuration would be defined by you. After that, the instance handle names remains fixed and must not be changed by you.

7.3 Status

This figure highlights the `svt_cxl_system_status`, which contains the set of status attributes related to CXL Cache/mem Component that is present inside the top level status class: `svt_cxl_subsystem_status`.

Refer Section 3.5 for top level status class details. This chapter assumes that you have already created environment `svt_cxl_system_status` is the Status class used to get the information about CXL.Cache/mem Agent's status. `svt_cxl_subsystem_status` is the CXL Subsystem 'top level' status class. It encapsulates the CXL.Cache/mem status class as an object.



`svt_cxl_system_status` contains instances of the class `svt_cxl_status` for both Host and Device statuses

`svt_cxl_status contains` all the status attributes related to `the CXL.Cache/mem agent.`

```
svt_cxl_system_status (cache_mem_sys_status)
                |
                |--------> svt_cxl_status(host_cache_mem_status[0]/
                          device_cache_mem_status[0]>)
                          |
                          |--------> svt_cxl_tl_status (tl_status)
                          |
                          |--------> svt_cxl_dl_status (dl_status)
```

- CXL Transaction Layer Status (Type= svt_cxl_tl_status, Instance= tl_status):

  `svt_cxl_tl_status` is a Transaction Layer Status class. This contains information related to the CXL.Cache/mem Transaction Layer. Refer to the documentation of `#svt_cxl_tl_status` class for more details.

- CXL Link Layer Status (Type= svt_cxl_dl_status, Instance= dl_status):

  `svt_cxl_dl_status` is a Link Layer Status class. This contains information related to the CXL.Cache/mem the Link layer. Refer to the documentation of `#svt_cxl_dl_status` class for more details.

For all the available status attributes related to CXL.Cache/mem agent, refer to the HTML Class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/status/class_svt_cxl_system_status.html
```

Example usage:

How to wait for CXL.Cache/mem DL link to be up?

```
/** Instances of Subsystem status */
  svt_cxl_subsystem_status host_status;
// Sample Format:
//<subsystem_status_class_obj_handle>.cache_mem_sys_status.
<host_cache_mem_status[0]/device_cache_mem_status[0]>.dl_status.<status_attribute>
/** wait for the CXL.Cache/mem DL link to be up**/
wait(host_status.cache_mem_sys_status.host_cache_mem_status[0].dl_status.curr_fsm_state
== svt_cxl_dl_status::LINK_UP);
```

Only the top-level status handle is user-defined. After that, the instance handle names remains fixed and must not be changed by the user.

7.4 Sequencers

There are two kinds of sequencers available in CXL Subsystem VIP.

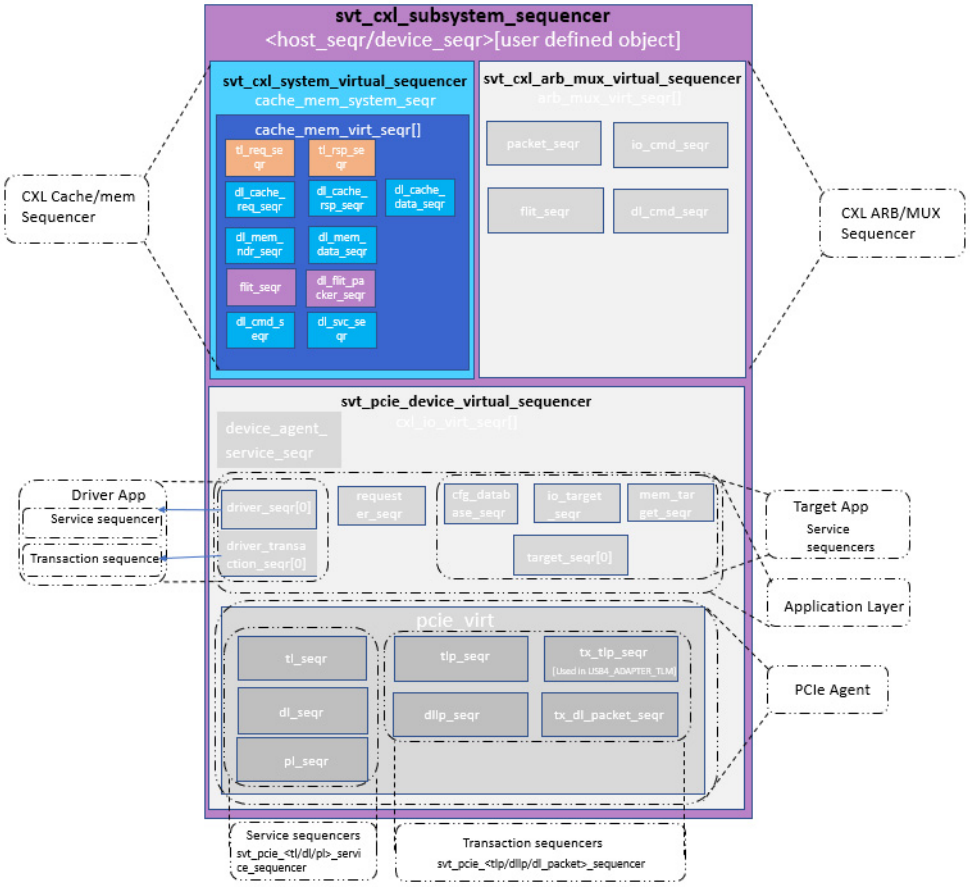
1. Service Sequencers

   These are used to schedule service sequences. These won't create any transactions and are used to request a change in the behavior of the VIP.

2. Transaction Sequencers

   These are used to schedule transactions sequences. These sequencers will generate the transactions.

The following figure highlights the `svt_cxl_system_virtual_sequencer`, which contains all the sequencers related to CXL Cache/mem Component, present inside the top level sequencer class - `svt_cxl_subsystem_sequencer`.



Note Only the top-level sequencer handle is user-defined. After that, the instance handle names remains fixed and must not be changed by the user.

`svt_cxl_system_vitual_sequencer` contains the instance of the class `svt_cxl_virtual_sequencer`.

`svt_cxl_virtual_sequencer` is the CXL.Cache/mem virtual sequencer used to control all of the sequencers that are operating at CXL Transaction and Data Link Layer.

```
svt_cxl_system_virtual_sequencer (cache_mem_system_seqr)
                  |
                  |--------> svt_cxl_virtual_sequencer(cache_mem_virt_seqr[0]>)
                            |
                            |--------> svt_cxl_tl_req_sequencer (tl_req_seqr)
                            |
                            |--------> svt_cxl_tl_rsp_sequencer (tl_rsp_seqr)
                            |
                            |--------> svt_cxl_channel_sequencer (dl_cache_req_seqr)
```

```
|
|--------> svt_cxl_channel_sequencer (dl_cache_rsp_seqr)
|
|--------> svt_cxl_channel_sequencer (dl_cache_data_seqr)
|
|--------> svt_cxl_channel_sequencer (dl_mem_ndr_seqr)
|
|--------> svt_cxl_channel_sequencer (dl_mem_data_seqr)
|
|--------> svt_cxl_flit_packer_sequencer
          (dl_flit_packer_seqr)
|
|--------> svt_cxl_flit_sequencer (flit_seqr)
|
|--------> svt_cxl_dl_service_sequencer (dl_svc_seqr)
|
|--------> svt_cxl_dl_command_sequencer (dl_cmd_seqr)
```

7.4.1 Description of CXL Cache/mem Sequencers

This table lists all the CXL Cache/mem Sequencers available at Transaction Layer and Data Link Layer.

List of CXL Cache/mem Sequencers

Sequencer Class	Instance	Available at	Description
svt_cxl_tl_req_sequencer	tl_req_seqr	CXL Transaction Layer	CXL.cache/mem Request sequencer
svt_cxl_tl_rsp_sequencer	tl_rsp_seqr	CXL Transaction Layer	CXL.cache/mem Response sequencer
svt_cxl_channel_sequencer	dl_cache_req_seqr	CXL Data Link Layer	CXL.cache Request Channel Sequencer for the Tx Request
svt_cxl_channel_sequencer	dl_cache_rsp_seqr	CXL Data Link Layer	CXL.cache Response Channel Sequencer for the Tx Response
svt_cxl_channel_sequencer	dl_cache_data_seqr	CXL Data Link Layer	CXL.cache Data Channel Sequencer for the Tx Data
svt_cxl_channel_sequencer	dl_mem_ndr_seqr	CXL Data Link Layer	CXL.mem Non-Data Request/Response Channel Sequencer
svt_cxl_channel_sequencer	dl_mem_data_seqr	CXL Data Link Layer	CXL.mem Data Request/Response Channel Sequencer
svt_cxl_flit_packer_sequencer	dl_flit_packer_seqr	CXL Data Link Layer	CXL Flit packer Sequencer

Sequencer Class	Instance	Available at	Description
svt_cxl_flit_sequencer	flit_seqr	CXL Data Link Layer	CXL flit sequencer to dispatch flits to Arb/Mux from DL
svt_cxl_dl_service_sequencer	dl_svc_seqr	CXL Data Link Layer	Sequencer which can supply Link Service requests to the DL driver.
svt_cxl_dl_command_sequencer	dl_cmd_seqr	CXL Data Link Layer	CXL link command sequencer to dispatch control information from DL to Arb/Mux
svt_cxl_tl_service_sequencer	tl_svc_seqr	CXL Transaction Layer	Sequencer which can supply TL Service requests to the TL driver. For triggering CXL Cache/mem TL service sequence, use the service sequencer - svt_cxl_tl_service_sequencer vip_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[0].tl_svc_seqr

You need to choose the sequencer based on the sequence type.

For example, if you want to generate the response using this sequence - `svt_cxl_tl_cache_mem_resp_sequence` then the following response sequencer must be chosen.

```
vip_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[0].tl_rsp_seqr
```

If you want to use the TL Transaction sequence -to generate the traffic at Transaction layer level then the following sequencer should be choosen.

```
vip_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[0]
```

For triggering CXL Cache/mem DL service sequence, use the service sequencer - `svt_cxl_dl_service_sequencer`

```
vip_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[0].dl_svc_seqr
```

7.5 Sequences

Similar to Sequencers there are two types of sequences related to CXL Cache/mem available in CXL Subsystem VIP.

Those are:

1. Transaciton Sequences
2. Service Sequences

Transaction Sequences are used to generate traffic at different layers. On the other hand, Service sequences are used for changing the behavior of VIP.

For the description about all the CXL Cache/mem sequences available, refer to the HTML Class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequencepages.html
```

These transaction sequences are present at CXL Cache/meme Transaction Layer and Data Link Layer in the CXL Subsystem VIP.

List of CXL.cache/mem Transaction sequences present at each layer

Sequencer Name	Level	Description
svt_cxl_tl_virtual_cache_mem_sequence	At Transaction Layer	For generating TL Cache/Mem Request/Response/Data transfers
svt_cxl_tl_cache_mem_resp_sequence	At Transaction Layer	For generating random response for cache_mem TL Host and Devices
svt_cxl_dl_virtual_cache_mem_sequence	At Data Link Layer	For generating all the DL Cache/Mem Request/Response/Data transfers

Example Usage:

How do I generate Single CXL.mem request using the Transaction sequence available ?

/* CXL.Cache/mem transaction sequence to generate the CXL.cache/mem requests */

```
svt_cxl_tl_virtual_cache_mem_sequence tl_cache_mem_seq;
```

/* Put the constraints to take no. of Mem Requests with Data (i.e. Mem Write) to 1 and remaining to 0 */

```
`uvm_do_on_with(
tl_cache_mem_seq,
vip_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[0],
{ tl_cache_mem_seq.num_mem_rwd == 1;
tl_cache_mem_seq.num_mem_req == 0;
tl_cache_mem_seq.num_cache_req == 0;});
```

Here vip_seqr represents the top-level virtual sequencer (`svt_cxl_subsystem_sequencer`) for VIP. Refer to Section 3.3 for more details.

7.6 APIs

The APIs related to CXL.Cache/mem component are available in the CXL Subsystem VIP. These can be used to create sequences (The existing sequences which are part of the example - `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys` can be referred for the usage of the APIs).

These APIs are present inside the sequence class - `svt_cxl_subsystem_api_collection_sequence`

For complete details on the available sequence APIs, refer to HTML Class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html
```

To use these APIs, you can create their own sequence class by extending from the API base sequence class and use the APIs to activate the link and to initiate the traffic.

Example for using generate_cxl_tl_rand_mem_traffic() methods:

```
class svt_cxl_subsystem_ts_tl_mem_req_sequence extends
svt_cxl_subsystem_ts_api_base_sequence;
……..
……

virtual task body();
int unsigned host_seq_cnt; int unsigned device_seq_cnt;

`svt_note("body", "Entered ..."); super.body();
void'(svt_config_int_db#(int unsigned)::get(null, get_full_name(),
"cache_mem_link_num", cache_mem_link_num));
`svt_debug("body", $sformatf("cache_mem_link_num %0d ",cache_mem_link_num ));

void'(svt_config_int_db#(int unsigned)::get(null, get_full_name(), "num_mem_req",
num_mem_req));
`svt_debug("body", $sformatf("num_mem_req %0d ",num_mem_req ));

void'(svt_config_int_db#(int unsigned)::get(null, get_full_name(), "num_mem_rwd",
num_mem_rwd));
`svt_debug("body", $sformatf("num_mem_rwd %0d ",num_mem_rwd ));

repeat (sequence_length) begin
`svt_debug({get_name(),".body"}, $sformatf("Executing Host Sequence Count [%0d]
.............", host_seq_cnt));

host_seq.generate_cxl_tl_rand_mem_traffic(.cache_mem_link_num(cache_mem_link_num),
.num_mem_req(num_mem_req), .num_mem_rwd(num_mem_rwd)); host_seq_cnt++;
end

`svt_note("body", "Exiting..."); endtask: body
```

Example for using generate_cxl_tl_rand_cache_mem_traffic() methods

```
class svt_cxl_subsystem_ts_tl_cache_req_sequence extends
svt_cxl_subsystem_ts_api_base_sequence;
……..
……

virtual task body();
int unsigned host_seq_cnt; int unsigned device_seq_cnt;
`svt_note("body", "Entered ..."); super.body();
fork
begin
repeat (sequence_length) begin
`svt_debug({get_name(),".body"}, $sformatf("Executing Host Sequence Count [%0d]
.............", host_seq_cnt));

host_seq.generate_cxl_tl_rand_cache_mem_traffic(.cache_mem_link_num(0),.num_cache_req(1
), .num_mem_req(0), .num_mem_rwd(0)); host_seq_cnt++;
end end begin
repeat (sequence_length) begin
```

```
`svt_debug({get_name(),".body"}, $sformatf("Executing Device Sequence Count [%0d]
.............", device_seq_cnt));

device_seq.generate_cxl_tl_rand_cache_mem_traffic(.cache_mem_link_num(0),.num_cache_req
(1), .num_mem_req(0), .num_mem_rwd(0));
device_seq_cnt++; end
end join
`svt_note("body", "Exiting..."); endtask: body
```

Example and Usage for using update_cxl_tl_mem_poison() method

This API is applicable for CXL.Mem and can be used for user specific poison injection/clearing at a given Device address

```
  virtual task body();
    int unsigned host_seq_cnt;
    int unsigned device_seq_cnt;
    bit [(`SVT_CXL_MAX_ADDR_WIDTH-1):0] addr = 'h8000;
    bit [(`SVT_CXL_MAX_DATA_WIDTH-1):0] clean_data;
    `svt_note("body", "Entered ...");
    clean_data ='h1234;
    super.body();

    fork
      begin
        // Guarded against the TL B2B topology
        if (link_cfg.get_subsystem_comp_avlb(1,
svt_cxl_subsystem_configuration::CACHE_MEM_LL))
host_seq.activate_cache_mem_link(.link_num(cache_mem_link_num));

      end
      begin
        // Guarded against the TL B2B topology
        if (link_cfg.get_subsystem_comp_avlb(0,
svt_cxl_subsystem_configuration::CACHE_MEM_LL))
device_seq.activate_cache_mem_link(.link_num(cache_mem_link_num));
      end
    join
      // Inject Poison at given address using TL Service Request
      device_seq.update_cxl_tl_mem_poison(.cache_mem_link_num(0),
.addr(addr),.poison(1));
      // Clear Poison for given address using TL Service Request
      device_seq.update_cxl_tl_mem_poison(.cache_mem_link_num(0),
.addr(addr),.poison(0),.data(clean_data));
    `svt_note("body", "Exiting...");
  endtask: body
```

7.7 Auto/Default Response Sequence Support

Auto response generation from the Host and Device VIP's Cache Memory Transaction Layer is supported. Therefore, a response sequence is set as default sequence for the Host and Device response sequencer. If you wish to drive your own sequence as a response sequence, you can just set the respective configuration variable, `svt_cxl_cache_mem_configuration::enable_tl_user_rsp_seq` variable to 1 in the test. The

following code allows you to disable the default response sequence for all the Cache Memory Host and Device response sequencers of the Transaction Layer.

```
virtual function void build_phase(uvm_phase phase ); string method_name =
"build_phase"; super.build_phase( phase );
   for (int i = 0; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_host_agent ; i++) begin
     cust_cfg.host_cfg.cache_mem_sys_cfg.host_cfg[i].enable_tl_user_rsp_seq = 1;
   end
   for (int i = 0; i < cust_cfg.device_cfg.cache_mem_sys_cfg.num_device_agent ; i++)
     begin
     cust_cfg.device_cfg.cache_mem_sys_cfg.device_cfg[i].enable_tl_user_rsp_seq = 1;
   end
endfunction
```

After enabling this user-defined response sequence configuration, you can create your own response sequence and use it in the test case.

7.7.1 GO-Err/GO-Err-WritePull Response

GO-Err and GO-Err-WritePull response generation from the Host VIP are supported. Host VIP sends GO-Err response for D2H reads that targets an address, which is not accessible by the device.

For correct address range, the Host VIP sends only positive responses for D2H transactions. You can extend the response sequence to enable GO-Err/GO-Err-WritePull response.

7.8 Backdoor Access to Memcore (Memory Core)

7.8.1 Initialize Memory Content Through Backdoor Access

The memory content can be accessed by creating the mem backdoor handle and getting the mem backdoor instance from the sequencer in svt_cxl_agent. You can access the memory array through backdoor by getting the svt_mem_backdoor handle for memcore.

To pre-load or dump the contents of memory core the APIs like initialize, load and dump can be used. Follow these steps for such operations:

Step 1: Create a handle to `svt_mem_core`

```
svt_mem_core host_mem_backdoor;
```

Step 2: Point the back door handle to the memory core which is instantiated inside the sequencer

// For CXL Host / Device agents, `cache_mem_virt_seqr[]` provides pointer to the sequencer present in the `svt_cxl_agent`, same used to get the backdoor handle as shown:

```
host_mem_backdoor =
p_sequencer.host_seqr.cache_mem_system_seqr.cache_mem_virt_seqr[cache_mem_link_num].get_
backdoor();
```

For more details you can refer the `svt_mem_backdoor_base` and `svt_mem_backdoor` APIs in the HTML documentation at:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/svt_uvm_class_reference/html/clas
s_svt_mem_backdoor_base.html
```

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/svt_uvm_class_reference/html/clas
s_svt_mem_backdoor.html
```

Look for testcase `ts.cxl_tl_backdoor_mem_wr_rd.sv` in the basic examples which are present at:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/examples/sverilog`

The test uses the sequence which is present in the HTML documentation at:
`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/examples/sverilog/tb_cxl_subsystem_uvm_basic_sys/env/seq_and_cb/svt_cxl_subsystem_ts_cache_mem_sequence_collection/svt_cxl_subsystem_ts_backdoor_mem_tl_wr_rd_sequence.sv`

Step 3: Perform memory content initialization/ pre-loading using the API as below:

// Different initialization patterns are allowed, refer the HTML documentation

```
host_mem_backdoor.initialize(svt_mem_backdoor::INIT_CONST,0,0,((1 <<
link_cfg.cache_mem_sys_cfg.host_cfg[cache_mem_link_num].addr_width)-1));
```

7.8.2 Accessing the Memory Content Through Backdoor Mechanisms like Peek/Poke

To access memory content using backdoor approach, follow above Step 1 and Step 2 to create the backdoor handle to memcore. Refer the above-mentioned test and sequence for reference. Then follow this step:

Step 3: Perform memory content write followed by read operation using the peek and poke APIs as below:

// First argument is aligned logical address, second argument is data to be poked into memory

```
void'(host_mem_backdoor.poke(address, data));
```

// First argument is aligned logical address, second argument is data peeked from the memory.

```
void'(host_mem_backdoor.peek(address, data));
```

The address must be aligned wrt the data width (defined by `SVT_CXL_MAX_DATA_WIDTH`), and the address width will be defined by `SVT_CXL_MAX_ADDR_WIDTH`.

To run test `ts.cxl_tl_backdoor_mem_wr_rd.sv`, do following:

For running with Full Stack topology, use compile_snps_vip_pcie_serial.f

```
gmake USE_SIMULATOR=vcsvlog cxl_tl_backdoor_mem_wr_rd
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_pcie_serial.f
```

For running with TL+DL topology, use compile_snps_vip_cxl_cache_mem.f

```
gmake USE_SIMULATOR=vcsvlog cxl_tl_backdoor_mem_wr_rd
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_cxl_cache_mem.f
```

For running with TL Only topology, use compile_snps_vip_cxl_cache_mem_tl.f

```
gmake USE_SIMULATOR=vcsvlog cxl_tl_backdoor_mem_wr_rd
SVT_CXL_SUBSYSTEM_COMPILE_FILE=compile_snps_vip_cxl_cache_mem_tl.f
```

7.9 Generic FAQs

1. Why is the CXL Cache-mem TL driver driving the transactions sequentially, even when they are getting driven in parallel from the sequence?

   You can check the address or the Tag / IDs of the transaction getting driven. If there are multiple transactions driven with the same address or Tag/ IDs, then they are processed and driven by the driver in the sequential order, following the *"Forward Progress & Ordering Rules"* of the CXL Spec.

8 CXL ARB/MUX Agent

8.1 Overview

8.1.1 Agent Overview

ARB/MUX is needed for sharing the same Physical Layer for multiple Link Layers. On Data Path, multiplexing is done at Flit level. Link Layers for each stack have separate credit checks before reaching ARB/MUX. On Control Path, ARB/MUX provides abstraction and coordination with the remote side for ACTIVE/PowerManagement transitions.

The `svt_cxl_arb_mux_agent` encapsulates ARB/MUX driver, sequencer and monitor. The `svt_cxl_arb_mux_agent` can be configured to operate in active mode. The `svt_cxl_arb_mux_agent` is configured using configuration `svt_cxl_arb_mux_configuration`. For transmit path, `svt_cxl_arb_mux_sequencer` will receive the dispatched CXL.cache/mem flits from the DL driver. Within the `svt_cxl_arb_mux_agent`, the `svt_cxl_arb_mux` driver gets `svt_cxl_flit` type sequence items from the svt_cxl_arb_mux_agent sequencer. The driver also gets CXL.io Flits (`svt_pcie_dl_packet`) from PCIE DL layer. The `svt_cxl_arb_mux` driver and monitor components within `svt_cxl_arb_mux_agent` call callback methods at various phases of execution of CXL.cache/mem/io Flit. After the 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.

Virtual Link State Machine (vLSM) and ALMPs

The ARB/MUX maintains vLSM state for each CXL link layer it interfaces with - CXL.io and CXL.mem. It receives requests from power management controller (PMC), local link layer and the remote ARB/MUX on behalf of a remote link layer to resolve a single state request to forward to the physical layer.

The ARB/MUX uses ALMPs to communicate virtual link state transition requests and responses associated with each link layer to the remote ARB/MUX. CXL ALMP transaction generated whenever there is ALMP State Request or Status from Host or Device.

- ALMP transaction generated whenever there is L1.x, L2 request from Host or Device.
- ALMP transactions are generated after link-up and on successful ALMP handshake indicate ARB_MUX is in active state and can accept packets from Data Link Layers.
- When link is up in cxl.io mode ALMP generation is bypassed as there is only one active link.
- Different vLSM states are RESET_NOP, ACTIVE, DAPM, L1_1, L1_2, L1_3, L1_4, L2, LINKRESET, LINKERROR, RETRAIN LINKDISABLE

Description of the vLSM States

vLSM State	Description
Reset	Power-on default state
Active	Normal operation state (PCIe – L0)
Retrain	Link training (PCIe – Recovery)
L1.1, L1.2, L1.3, L1.4	Power savings states (PCIe – L1 substates)
SLEEP_L2	Power savings state (PCIe – L2)
LinkReset	Reset propagation state (PCIe - HotReset)
LinkDisable	Disabled state
LinkError	Error state
DAPM	Resolves to one of L1.1, L1.2, L1.3,L1.4

Figure 8-1 vLSM states for CXL.io and CXL.cache/mem

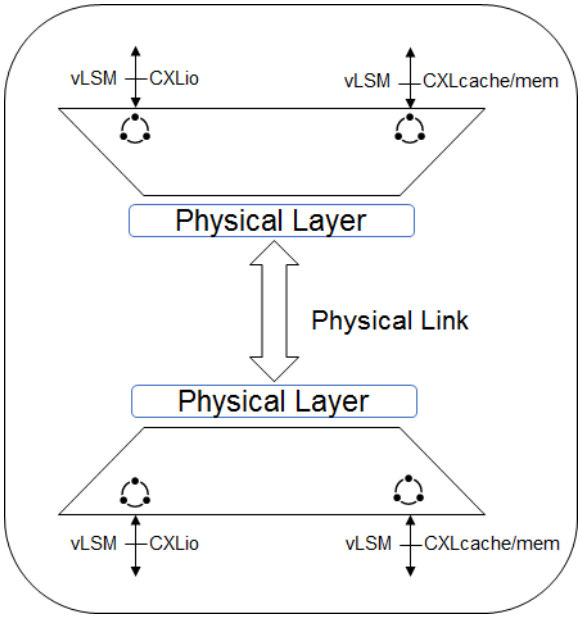
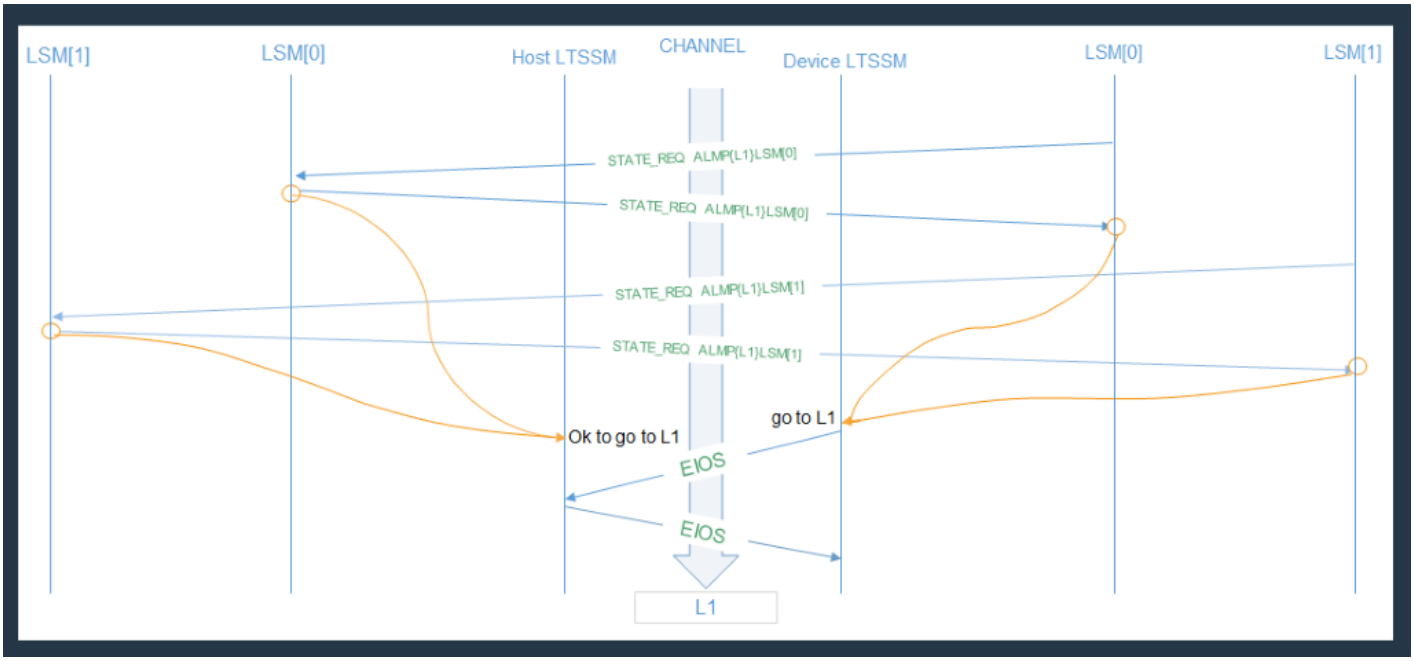


Figure 8-2 vLSM flow for L1 state transition



Link Management Packets(ALMPs)

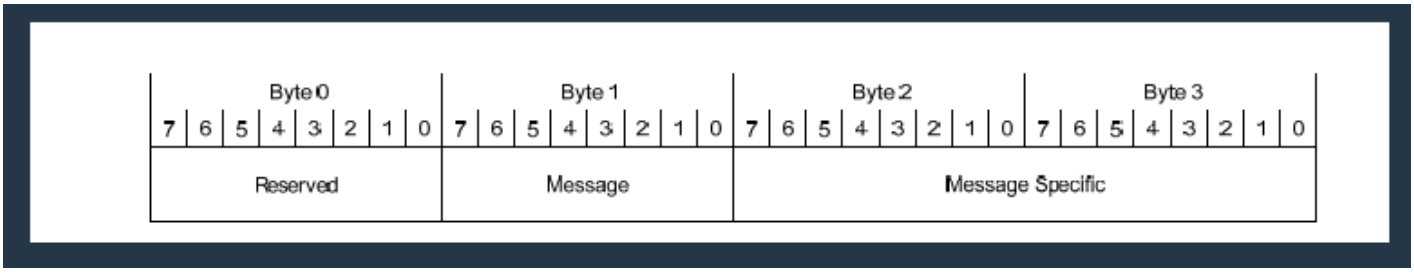
ALMPs are 1DW packets with the format shown below. If the ARB/MUX detects an error in the ALMP, then it initiates a retrain of the link (Recoveryflow). For data integrity protection, the 1DW packet is replicated four times on the lower 16-bytes of a 528-bit flit

Two types of ALMPs:

Request: Initiates transitions of the remote vLSM

Status: Indicates reception of Request or used for vLSM synchronization after Retrain.

Figure 8-3 ALMP Packet Format



8.1.2 Classes and Applications for Using ARB/MUX Component

In CXL VIP, you can enable number of ARB/MUX through this define. It is done through MACRO `SVT_CXL_MAX_NUM_ARB_MUX`.

Number of ARB MUX in the subsystem (`svt_cxl_subsystem_configuration`), rand int `num_arb_mux` = 0

Min value: 0

Max value: `\`SVT_CXL_MAX_NUM_ARB_MUX (`SVT_CXL_MAX_NUM_ARB_MUX)`

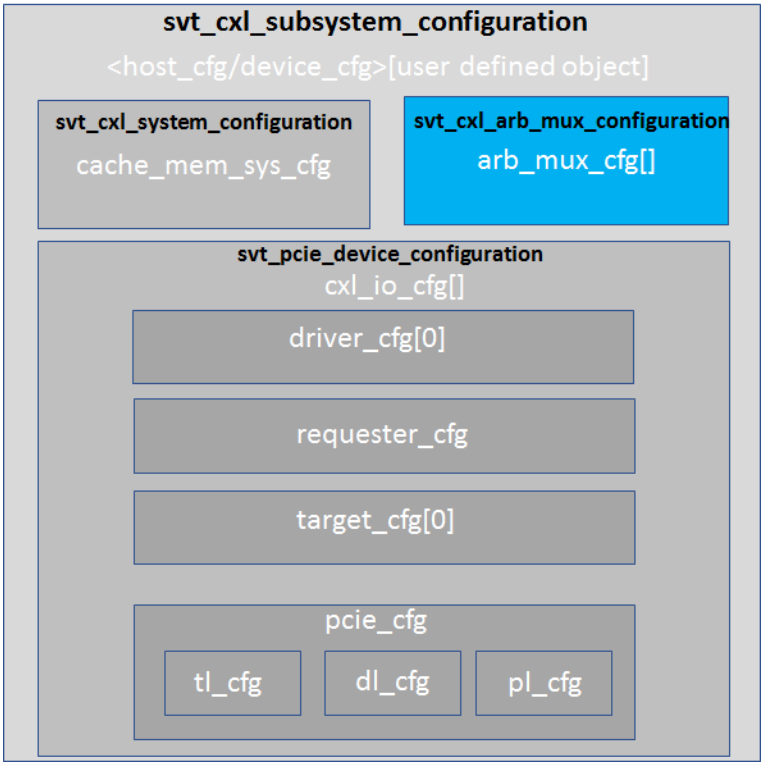
Note This parameter (`num_arb_mux`) is to be treated as read-only for any accesses from outside the CXL subsystem configuration Writing/modifying this attributes may lead to unexpected results from the VIP.

8.2 Configuration

Use the `svt_cxl_arb_mux_configuration` to define the overall behavior of the ARB/MUX component. You must note that most configurable attributes are defined as SystemVerilog types. Consult the CXL HTML Class Reference on declared data types.

Note Refer Section 3.4 for top level configuration class details. This chapter assumes the you have already created the environment.

Figure 8-4 Configuration for CXL ARB/MUX Agent



For all the available configuration attributes and functions related to CXL ARB/MUX agent, refer to the HTML Class reference:

`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/configuration/class_svt_cxl_arb_mux_configuration.html`

Example usage:

How to introduce the delay while sending the ALMP requests from VIP to remote IO vLSM?

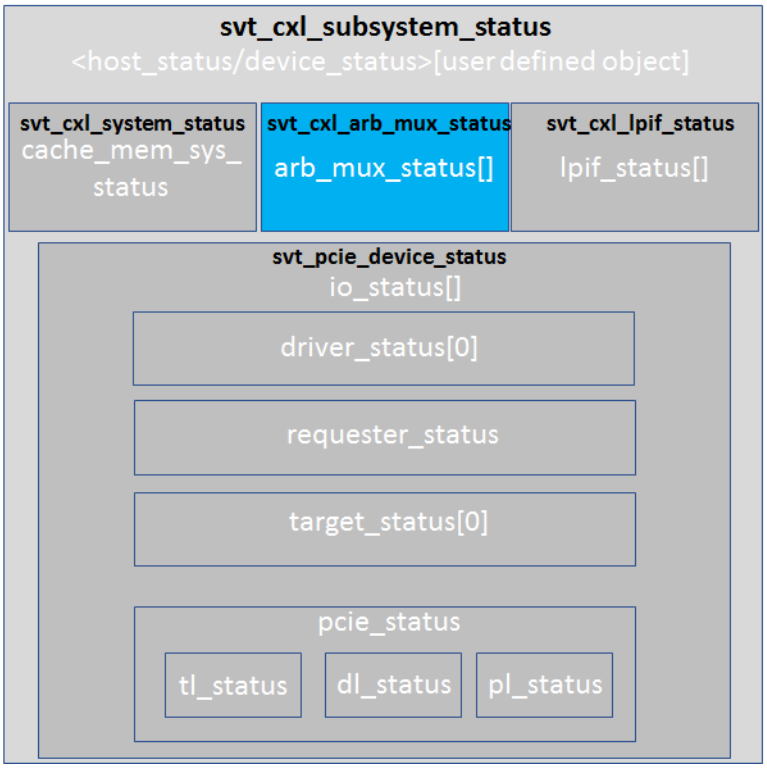
```
svt_cxl_subsystem_link_configuration cust_cfg;
/** Configuring the CXL.Cache/mem agent **/
// Sample Format ->
//<cust_cfg>.<host_cfg/device_cfg>.arb_mux_cfg[0].<host_cfg[0]/device_cfg[0].< ARB/MUX
configuration attribute>

/*Set the confiugrations for the dealy, in ns, after which VIP will send Active State
Request ALMP to remote IO vLSM during ARB/MUX Initialization */

/*Configurations from Host VIP side*/
for (int i = 0 ; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_host_agent ; i++) begin
    cust_cfg.host_cfg.arb_mux_cfg[i].io_state_req_l1_almp_delay_ns_min_val = 150;
    cust_cfg.host_cfg.arb_mux_cfg[i].io_state_req_l1_almp_delay_ns_max_val = 250;
end

/*Configurations from Device VIP side*/

  for (int i = 0 ; i < cust_cfg.device_cfg.cache_mem_sys_cfg.num_device_agent; i++)
begin
    cust_cfg.device_cfg.arb_mux_cfg[i].io_state_req_l1_almp_delay_ns_min_val = 100;
    cust_cfg.device_cfg.arb_mux_cfg[i].io_state_req_l1_almp_delay_ns_max_val = 200;
  end
```

8.3 Status

The `svt_cxl_arb_mux_status` class gives the specific status information related to ARB/MUX component.

Refer Section 3.5 for top level status class details. This chapter assumes the you have already created the environment.

Figure 8-5 Status for CXL ARB/MUX Agent



For all the available status attributes related to CXL ARB/MUX agent, refer to the HTML Class reference:

```
$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html
/status/class_svt_cxl_arb_mux_status.html
```

Example usage:

How to wait for CXL.io vLSM to be in L1 states when creating low power scenario?

```
// Sample Format:
//<subsystem_status_class_obj_handle>.arb_mux_status[0].<ARB//MUX status_attribute>

/** wait for the CXL.io vLSM to be inside L1 States*/
wait (host_seq.cxl_subsystem_status.arb_mux_status[0].cxl_io_current_vlsm_fsm_state
inside {`SVT_CXL_TEST_L1_STATES});
```

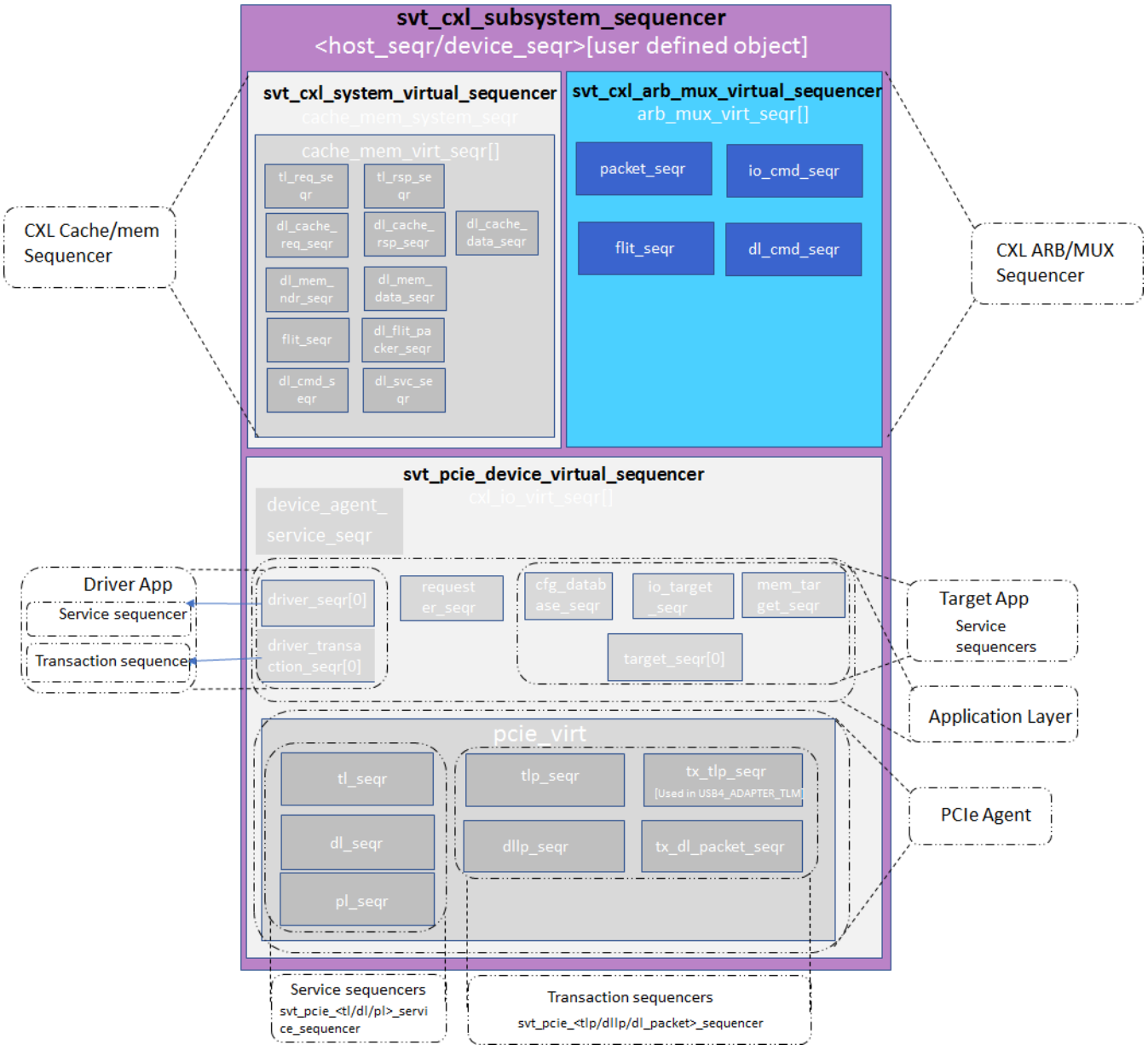
Note In this example, we used the `cxl_subsystem_status` handle of type - `svt_cxl_subsystem_status` present inside `svt_cxl_subsystem_virtual_api_collection_sequence` (host_seq is a handle of this type)

Only the top-level status handle is user-defined. After that, the instance handle names remain fixed and must not be changed by the user.

8.4 Sequencers

CXL ARB/MUX agent has these service sequencers present for providing information to Arb/Mux for Arb/Mux functionality. For transmit path, `svt_cxl_arb_mux_sequencer` receives the dispatched CXL.cache/mem flits from the DL driver. Within the `svt_cxl_arb_mux_agent`. The `svt_cxl_arb_mux` driver gets `svt_cxl_flit` type sequence items from the `svt_cxl_arb_mux_agent` sequencer.

Figure 8-6 Sequencer for CXL ARB/MUX Agent



The top level sequencer of ARB/MUX are the following:

```
svt_cxl_arb_mux_virtual_sequencer (arb_mux_virt_seqr)
                  |
```

```
|--------> svt_cxl_arb_mux_service_sequencer (arb_mux_svc_seqr)
|--------> svt_cxl_dl_command_sequencer (dl_cmd_seqr)

|--------> svt_cxl_flit_sequencer (flit_seqr)
```

List of sequencers present in ARB/MUX layer are:

svt_cxl_arb_mux_virtual_sequencer	This class defines a virtual sequencer that can be connected to the `svt_cxl_arb_mux_agent`.
svt_cxl_arb_mux_service_sequencer	This class is UVM Sequencer parameterized by `svt_cxl_arb_mux_service_transaction` class object. This service transaction sequencer is used to provide Link Up information to Arb/Mux when PL is not present as a start point for Arb/Mux functionality
svt_cxl_dl_command_sequencer	This class is Sequencer that provides stimulus for the `svt_cxl_dl_command_driver` class. The `svt_cxl_dl_command_agent` class is responsible for connecting this `uvm_sequencer` to the driver if the agent is configured as UVM_ACTIVE
svt_cxl_flit_sequencer	This class is Sequencer that provides stimulus for the `svt_cxl_flit_driver` class. The `svt_cxl_flit_agent` class is responsible for connecting this `uvm_sequencer` to the driver if the agent is configured as UVM_ACTIVE.

8.5 Sequences

List of sequences present at ARB/MUX layer:

Sequence Name	Description
svt_cxl_arb_mux_service_base_sequence	This sequence is the base class for CXL ARB MUX Service sequences. All the other sequences are extended from this sequence.
svt_cxl_arb_mux_init_sequence	This sequence initiate Arb/Mux initialization through service sequencer. This sequence shall be used only in topologies where PCIe PL is not available.

8.6 APIs

The APIs related to CXL.ARB/MUX component are available in the CXL Subsystem VIP. These can be used to create sequences (The existing sequences which are part of the example - `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys` can be referred for the usage of the APIs).

These APIs are present inside the sequence class:
`svt_cxl_subsystem_virtual_api_collection_sequence`

For complete details on the available sequence APIs, refer to HTML Class reference:

`$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html`

Example of using `generate_arb_mux_dl_cmd()` methods

```
task generate_arb_mux_dl_cmd(int
link_num=0,svt_cxl_arb_mux_service_transaction::service_type_enumservice_type);
```

Method to activate DL link up of CXL.Cahe/Mem.This is a blocking method which waits till link is up and generates `dl_commands` intermediately. '`link_num`' indicates Cache Mem Link/instance to initiate traffic.

We can use this API as below.

```
host_seq.generate_arb_mux_dl_cmd(0,svt_cxl_arb_mux_service_transaction::LINK_RECOVERY);
  #1;
host_seq.generate_arb_mux_dl_cmd(0,svt_cxl_arb_mux_service_transaction::LINK_ACTIVE);
```

8.7 ARB/MUX Callback svt_cxl_arb_mux_callback

ARB/MUX callback used by tests to inject ALMP error into the flits transmitted by the `svt_cxl_arb_mux` driver to PL layer and also to inject PID error into the flits.

It has the following functions that can be used in error injection. We can refer to the HTML class reference for description of these callback functions.

virtual function void	post_tx_cache_mem_dl_command_seq_item_get (svt_cxl_arb_mux driver, svt_cxl_dl_command cxl_dl_cmd_xact, ref bit drop)
virtual function void	post_tx_cache_mem_dl_flit_seq_item_get (svt_cxl_arb_mux driver, svt_cxl_flit cxl_flit_xact, ref bit drop)
virtual function void	post_tx_io_dl_packet_seq_item_get (svt_cxl_arb_mux driver, svt_pcie_dl_packet pcie_dl_packet, ref bit drop)
virtual function void	post_tx_io_pl_service_seq_item_get (svt_cxl_arb_mux driver, svt_pcie_pl_service pcie_pl_service, ref bit drop)
virtual function void	pre_tx_drive_almp_data (svt_cxl_arb_mux driver, svt_cxl_almp_transaction almp_data)

Example:

```
class svt_cxl_arb_mux_almp_err_injection_cb extends svt_cxl_arb_mux_callback;

 /**Specifies the almp count on which ALMP error to be injected */
  int unsigned almp_err_count = $urandom_range(1,3);

  /**Specifies the value of ALMP error to be injected */
  int unsigned almp_err_value = $urandom_range(1,84);
```

```
  /**Specifies the byte number on which ALMP error to be injected */
  int unsigned byte_error = $urandom_range(0,9);

  /**Specifies the almp byte_2 error values by corrupting rsvd fields to be injected */
  int unsigned almp_byte2_error = $urandom_range(13,127);


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

  /** Overriding pre_tx_drive_flit, a callback method of svt_cxl_arb_mux driver class*/
  virtual function void pre_tx_drive_almp_data(svt_cxl_arb_mux driver,
svt_cxl_almp_transaction almp_data);
    string method_name = "pre_tx_drive_almp_data";
    `svt_xvm_note(method_name,$sformatf("Entered Callback Override to inject ALMP
Error"));
    `svt_note("pre_tx_drive_almp_data"," extended injection class");
    if (driver.common.error_injection ==1) begin
    `svt_note("pre_tx_drive_almp_data", "Error injection enabled");
    end

    `svt_note (method_name,$sformatf("almp_err_cnt is %h",almp_err_count));
    if((driver.common.error_injection==1) && (driver.common.almp_count ==
almp_err_count)) begin
      case (byte_error)
              'h0:
        begin //ALMP byte_0 error
          `svt_note(method_name,$sformatf("Value before Injecting the ALMP Error on the
BYTE0 ALMP DATA : \n %h",almp_data.almp_flit[(`SVT_CXL_ALMP_BYTE_WIDTH-1):0]));
          almp_data.almp_flit[(`SVT_CXL_ALMP_BYTE_WIDTH-1):0] = $urandom_range(1,255);
          `svt_note(method_name,$sformatf("ALMP after injecting the Error on BYTE0 is :
\n %h ",almp_data.almp_flit[(`SVT_CXL_ALMP_BYTE_WIDTH-1):0]));
        end
endcase
   end
  endfunction : pre_tx_drive_almp_data
endclass : svt_cxl_arb_mux_almp_err_injection_cb

//For registering the callback in the testcase
/** Instance of CXL ARB MUX ALMP Error Injection Call back */
  svt_cxl_arb_mux_almp_err_injection_cb drv_almp_err_cb;

virtual function void connect_phase(uvm_phase phase);
      `uvm_info ("connect_phase", "Entered...", UVM_LOW)
      super.connect_phase(phase);

      for (int i = 0 ; i <  `SVT_CXL_SUBSYSTEM_MAX_NUM_LINKS ; i++) begin
        drv_almp_err_cb = new("drv_almp_err_cb");
        svt_cxl_arb_mux_callback_pool::add(env.host_env.arb_mux[i].driver,
drv_almp_err_cb);
```

```
        svt_cxl_arb_mux_callback_pool::add(env.device_env.arb_mux[i].driver,
drv_almp_err_cb);
      end
    endfunction :connect_phase
```

9 Logical PHY Interface

9.1 Logical PHY Interface (LPIF) Use Model

9.1.1 LPIF Usage Modes

LPIF VIP can be used in these two modes:

1. **Link-LPIF**: When LPIF VIP is connected to PHY DUT, it acts as Link-LPIF and drives all lp_* signals. This mode is enabled by setting `svt_cxl_subsystem_configuration:: is_phy_lpif` to 0.
2. **PHY-LPIF**: When LPIF VIP is connected to Link DUT, it acts as PHY-LPIF and drives all pl_* signals. This mode is enabled by setting `svt_cxl_subsystem_configuration:: is_phy_lpif` to 1.

9.2 Integration Steps

9.2.1 Interface

1. Instantiate `svt_cxl_subsystem_if` in TOP module. Refer HTML Class reference for more details:

   ```
   $DESIGNWARE_HOME
   /vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/interface
   s.html#intf_svt_cxl_if
   ```

2. Assign input signals to the interface(lclk ,reset ,tb_clk).
3. Pass interface for environment access through `uvm_config_db`.

   ```
   svt_config_vif_db#(virtual svt_cxl_subsystem_if)::set(null, "*_test_top.env.host_env",
   "vif", cxl_if);
   svt_config_vif_db#(virtual svt_cxl_subsystem_if)::set(null, "*_test_top.env.device_env",
   "vif", cxl_if);
   ```

   See this topology file for more details:

   ```
   <design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys/topology
   _snps_vip_cxl_b2b_lpif.svi
   ```

9.2.2 Generation of LPIF Agent

You can use the following API of `svt_cxl_subsystem_configuration` to configure the `svt_cxl_subsystem_env` for LPIF:

```
configure_subsystem(subsystem_topology, if_type, is_phy_lpif)
```

Attribute	Supported Values
`subsystem_topology`	Currently supported values are: • svt_cxl_subsystem_configuration::IO_TL_DL_ONLY_STACK • svt_cxl_subsystem_configuration ::CACHE_MEM_STACK (supported only when subsystem bottom_layer is LINK and top_layer is TRANSACTION.)
`If_type`	svt_cxl_subsystem_types:LPIF_IF
`is_phy_lpif`	1,0 indicates whether the agent will drive the pl_* signals (1'b1) or lp_* signals (1'b0)

9.3 LPIF Example Testbench

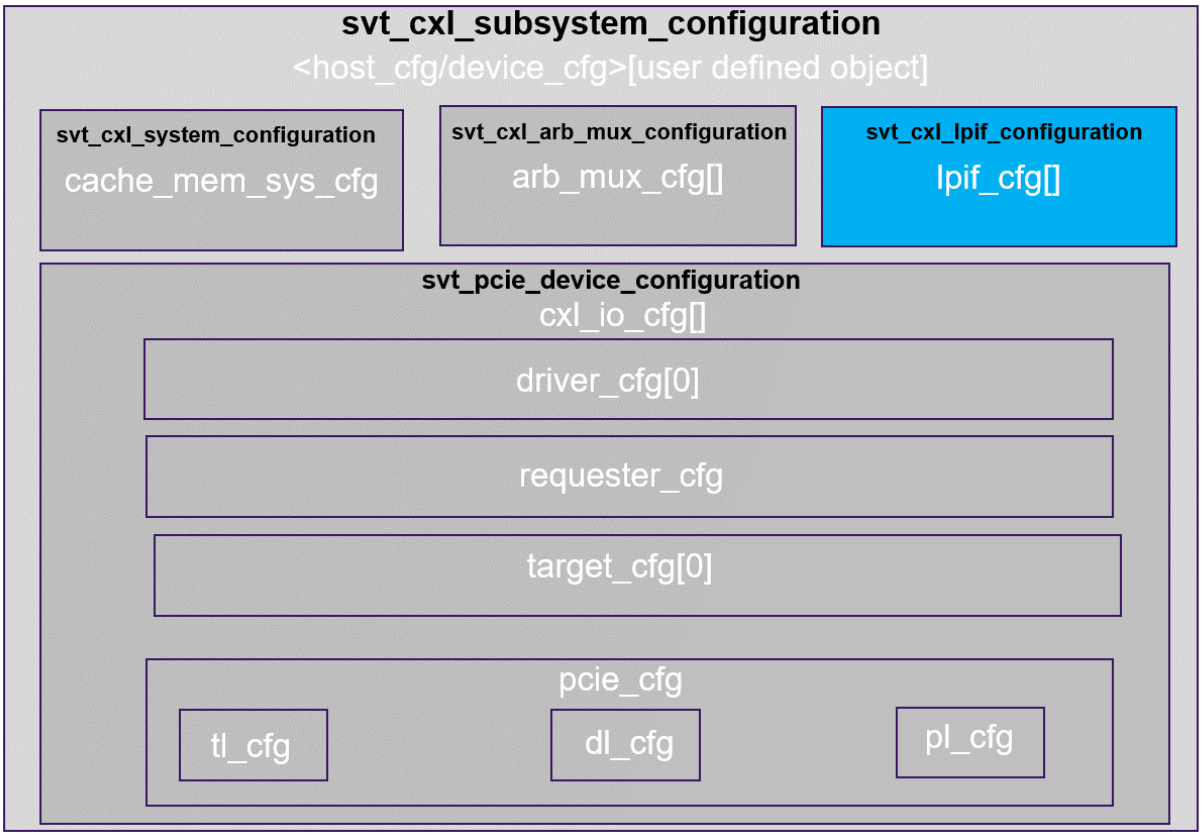
To run tests with LPIF, refer to the Section 4.2.3.2 Running CXL Subsystem VIP example test cases with Topology #3 In this, you would find the valid combinations of topology and compile files to be used.

9.4 Configurations

The CXL LPIF Agent is configured using an object of class `svt_cxl_lpif_configuration` (`lpif_cfg[]` is the handle) defined within the top level configuration class `svt_cxl_subsystem_configuration` of CXL Subsystem VIP.

Refer *section 3.4* for top level configuration class details. It is assumed that you have already created the environment.

Figure 9-1 CXL LPIF Agent Configuration



For all the available configuration attributes and functions related to CXL LPIF agent, refer to the *HTML Class Reference* available at the following location:

$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_class_reference/html/configuration/class_svt_cxl_lpif_configuration.html

Example Usage: Configuring `n_bytes` value

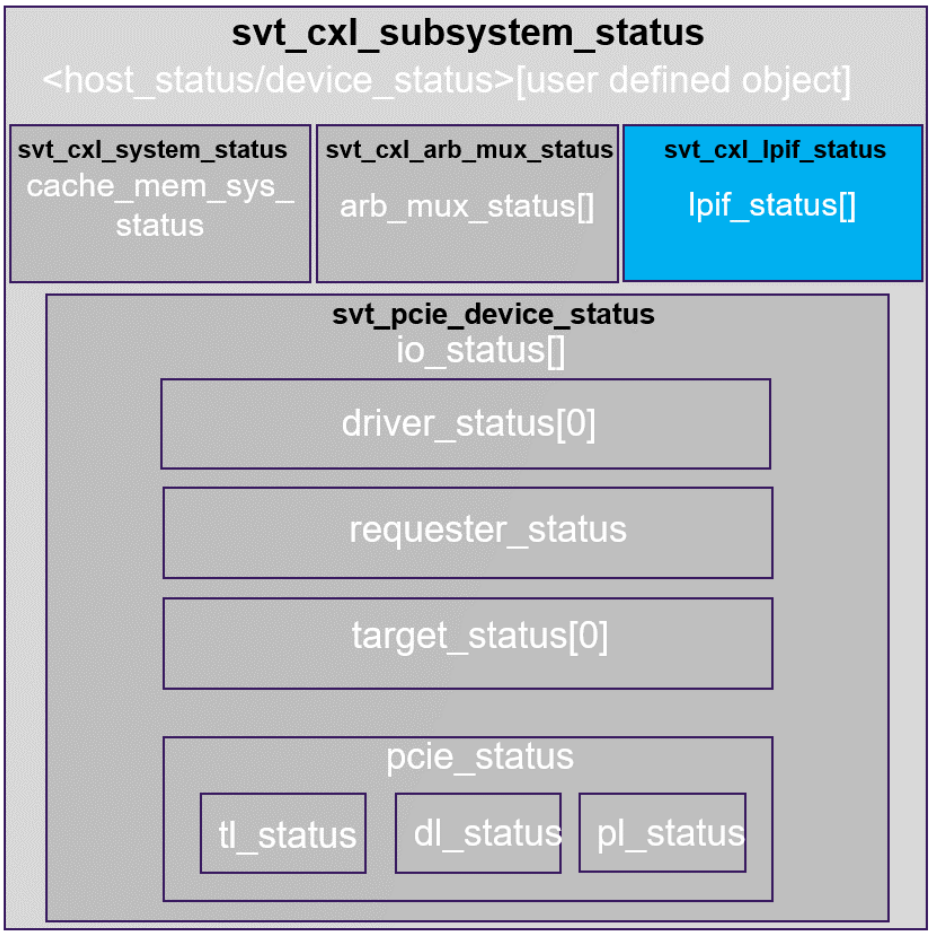
```
svt_cxl_subsystem_link_configuration cust_cfg;
/** Configuring the CXL LPIF agent **/
// Sample Format ->
//<cust_cfg>.<host_cfg/device_cfg>.lpif_cfg[0].<LPIF configuration attribute>
cust_cfg.host_cfg.lpif_cfg[0].nbytes = 64;
```

9.5 Status

The following figure highlights the `svt_cxl_lpif_status`, which contains the set of status attributes related to CXL LPIF Component that is present inside the top level status class, `svt_cxl_subsystem_status`.

Note Refer *section 3.5* for top level status class details. It is assumed that you have already created the environment.

Figure 9-2 CXL LPIF Agent Status



For all the available status attributes related to CXL LPIF agent, refer to the *HTML Class Reference* available at the following location:

$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_class_reference/html/status/class_svt_cxl_lpif_status.html

Example Usage: Waiting for `pl_trdy` status to be asserted

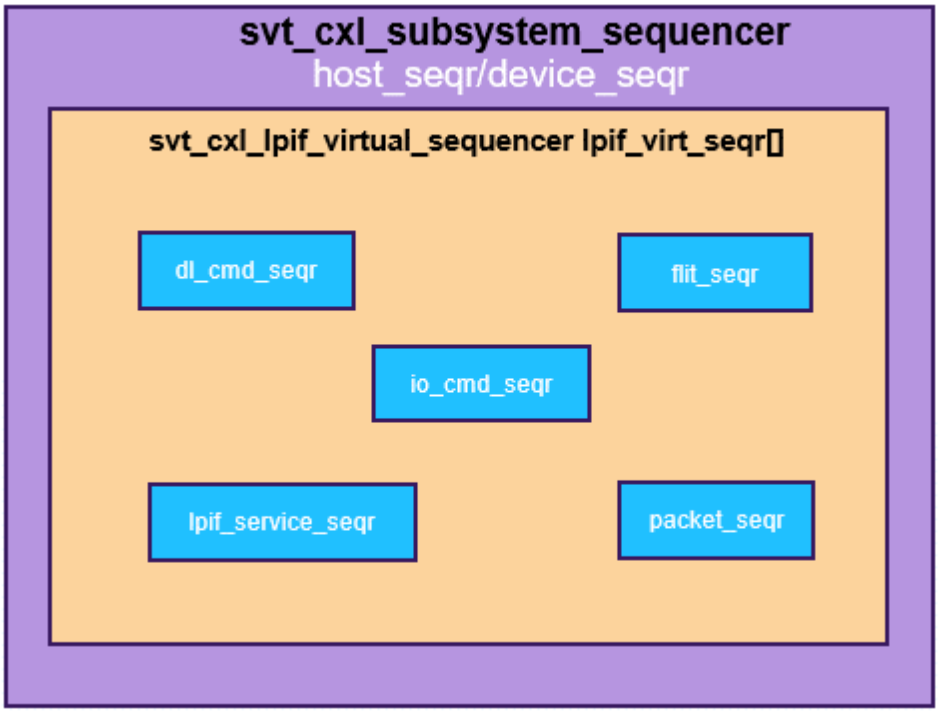
```
/** Instances of Subsystem status */
svt_cxl_subsystem_link_status link_status;
// Sample Format:
//<subsystem_link_status_class_obj_handle>.<host_status/device_status>.lpif_status[i].<status_attribute>
/** wait for the pl_trdy to be asserted**/
wait(link_status.device_status.lpif_status[0].pl_trdy == 1)
```

Note You only need to define the top-level configurations. After that, the instance handle (`lpif_status[]`) names remain fixed and you must not change the same.

9.6 Sequencers

There are sequencers present in the CXL LPIF agent for providing information to LPIF.

Figure 9-3 CXL LPIF Agent Sequencers



The top level sequencer of LPIF are as follows:

```
svt_cxl_lpif_virtual_sequencer (lpif_virt_seqr[])
|
|--------> svt_pcie_pl_service_sequencer (io_cmd_seqr)
|--------> svt_cxl_dl_command_sequencer (dl_cmd_seqr)
|--------> svt_cxl_flit_sequencer (flit_seqr)
|--------> svt_cxl_lpif_service_sequencer (lpif_service_seqr)
|--------> svt_pcie_dl_packet_sequencer (packet_seqr)
```

For all the available sequencers related to CXL LPIF, refer to the *HTML Class Reference* available at the following location:

$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_class_reference/html/sequencer/class_svt_cxl_lpif_virtual_sequencer.html

9.7 Sequences

For the list of sequences present for LPIF, refer to the *HTML Class Reference* present at the following location:

$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_class_reference/html/Sequence_class_list.html

Some of the sequences are listed in the table. It is recommended that you follow the API approach explained in the subsequent section for using these sequences.

Table 9-1 LPIF Sequences

Sequence Name	Description
`svt_lpif_hot_unplug_req_sequence`	Sequence for issuing `HOT_UNPLUG_REQ` LPIF service request.
`svt_lpif_update_pl_protocol_vld_sequence`	Sequence for issuing `UPDATE_PL_PROTOCOL_VLD` LPIF service request.
`svt_lpif_abort_l1_x_sequence`	Sequence for issuing `ABORT_L1_X` LPIF service request.

9.8 APIs

The APIs related to CXL LPIF component are available in the CXL Subsystem VIP. These APIs can be used to create sequences.

You can refer to the existing sequences which are part of the example, `cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys` for the usage of the APIs.

These APIs are present inside the sequence class, `svt_cxl_subsystem_virtual_api_collection_sequence`.

For complete details on the available sequence APIs, refer to the *HTML Class Reference* present at the following location:

$DESIGNWARE_HOME/vip/svt/cxl_subsystem_svt/latest/doc/cxl_subsystem_svt_uvm_public/html/sequences/class_svt_cxl_subsystem_virtual_api_collection_sequence.html

To use these APIs, you can create your own sequence class by extending from the API base sequence class and use the APIs to activate the link and perform different LPIF interface activities.

Example Usage: Ignoring PM request for LPIF Device VIP

You can use the API as illustrated:

```
abort_lpif_pm_req (  int delay_val = -1, int link_num = 0  )
//API to execute ABORT_L1_X LPIF service request from PHY LPIF VIP. Upon receiving this
request, PHY LPIF VIP will enter into RETRAIN state.
device_seq.abort_lpif_pm_req(,0);

//where host_seq is the handle of svt_cxl_subsystem_virtual_api_collection_sequence
```

9.9 CXL LPIF VIP for Link-Subdivision

9.9.1 API Used for Creating Link Subdivision

The `set_cxl_subsystem_env_device_cfg` API is used to specify the number of agents for subdivision.

Usage example:

```
cust_cfg.set_cxl_subsystem_env_device_cfg(1,svt_cxl_subsystem_configuration::CACHE_MEM_S
TACK,svt_cxl_subsystem_types::LPIF_IF, device_drive_pl_signals, 4);
```

Here, 4 is the `num_agent` attribute used to specify number of Device agents to be configured.

9.9.2 Connections

Refer the back to back CXL LPIF VIP connections via macro for interface signal connection, which connects signals across all interfaces based on the `num_agent` configured. Refer the sample code as illustrated:

```
generate
          for(genvar link_num=0;link_num<max_link; link_num++) begin

            `SVT_LPIF_IF_SIGNAL_CONNECT_1_0(lpif_host_if,lpif_device_if,link_num)
            `SVT_LPIF_IF_SIGNAL_CONNECT_1_1(lpif_host_if,lpif_device_if,link_num)
          end
endgenerate
```

Macro definition is present in *<design_dir>/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys/env/hdl_interconnect_macros.sv*

You need to connect VIP LPIF interface in similar way with the DUT interface. Bifurcation/Subdivision sample signal connection with DUT:

```
   assign   DUT.cxlcm_lpif_pl_data[7:0] = lpif_device_if.lpif_if[0].pl_data[0][7:0];
   assign   DUT.cxlcm_lpif_pl_data[15:8] = lpif_device_if.lpif_if[0].pl_data[1][7:0];
….

   assign   DUT.cxlcm_lpif_pl_data[135:128] =
lpif_device_if.lpif_if[1].pl_data[16][7:0];
    assign   DUT.cxlcm_lpif_pl_data[143:136] =
lpif_device_if.lpif_if[1].pl_data[17][7:0];
…
    assign   DUT.cxlcm_lpif_pl_data[263:256] =
lpif_device_if.lpif_if[2].pl_data[32][7:0];
    assign   DUT.cxlcm_lpif_pl_data[271:264] =
lpif_device_if.lpif_if[2].pl_data[33][7:0];


------
   assign   lpif_device_if.lpif_if[0].lp_data[0] = DUT.cxlcm_lpif_lp_data[7:0];
   assign   lpif_device_if.lpif_if[0].lp_data[1] = DUT.cxlcm_lpif_lp_data[15:8];
   ….
   assign   lpif_device_if.lpif_if[1].lp_data[16] = DUT.cxlcm_lpif_lp_data[135:128];
   assign   lpif_device_if.lpif_if[1].lp_data[17] = DUT.cxlcm_lpif_lp_data[143:136];

….
  assign   lpif_device_if.lpif_if[2].lp_data[32] = DUT.cxlcm_lpif_lp_data[263:256];
  assign   lpif_device_if.lpif_if[2].lp_data[33] = DUT.cxlcm_lpif_lp_data[271:264];
```

9.9.3 LPIF Configuration for Link Subdivision

For link subdivision, it is required to set the following configurations based on the requirement:

- `data_byte_start`: Indicates the start of byte of data bus used by each LPIF agent. Agent can use bytes (`data_byte_start` to `actual_max_data_bytes`) from pool of 64 as data bus is shared. This will be configured at 0 time and will not change during simulation. It is assumed that the VIP will drive/sample data/valid only on data bytes (`data_byte_start` to `actual_max_data_bytes`) of `lp_data/pl_data/lp_valid/pl_valid` signal of interface.

- `actual_max_data_bytes`: Indicates the maximum number of data bytes each LPIF agent can use from pool of 64 as data bus is shared. This will be configured at 0 time and will not change during simulation. It is assumed that the VIP will drive/sample data/valid only on data bytes (`data_byte_start` to `actual_max_data_bytes`) of `lp_data/pl_data/lp_valid/pl_valid` signal of interface.
- `nbytes`: Indicates the number of data bytes valid for a bit of `lp_valid` and `pl_valid`. This will be configured at 0 time and will not change during simulation. Configuration type: Static. Valid values are 1, 2, 4, 8, 16, 32, and 64.

Example for device configuration:

```
device_cfg.lpif_cfg[0].data_byte_start=0
device_cfg.lpif_cfg[0].actual_max_data_bytes=16
device_cfg.lpif_cfg[0].nbytes=16
device_cfg.lpif_cfg[1].data_byte_start=16
device_cfg.lpif_cfg[1].actual_max_data_bytes=16
device_cfg.lpif_cfg[1].nbytes=16
device_cfg.lpif_cfg[2].data_byte_start=32
device_cfg.lpif_cfg[2].actual_max_data_bytes=16
device_cfg.lpif_cfg[2].nbytes=16
device_cfg.lpif_cfg[3].data_byte_start=48
device_cfg.lpif_cfg[3].actual_max_data_bytes=16
device_cfg.lpif_cfg[3].nbytes=16
```

9.9.4 Disabling LPIF Interfaces

To disable the unused LPIF interface, set the LPIF configuration variable `"active_if"` to 0.

- `active_if`:
  - The configuration value when set to 1 specifies that the LPIF port is being used, when used in Multi-link setup.
  - The configuration value when set to 0 specifies that the LPIF port is disabled, when used in Multi-link setup.

Example Usage:

```
device_cfg.lpif_cfg[2].active_if = 0// Disabling third interface which is not active
```

9.9.5 Compile Time Defines

```
+define+SVT_CXL_MAX_NUM_CXL_CACHE_MEM_OVER_LPIF=<max agent>
+define+SVT_CXL_MAX_NUM_HOST_CACHE_MEMS=<max_agent>
+define+SVT_CXL_MAX_NUM_DEVICE_CACHE_MEMS=<max_agent>
```

Note Do not define `"SVT_ENABLE_XCHECK"`.

9.10 LPIF Protocol Checks

These are the LPIF specific protocol checks that are supported in the CXL subsystem VIP:

S.No	Protocol Check Name	Check Description
1	lp_state_req_encoding	Verify that lp_state_req does not indicate any reserved encodings

S.No	Protocol Check Name	Check Description
2	pl_state_sts_encoding	Verify that pl_state_sts does not indicate any reserved encodings
3	pl_link_cfg_value	Verify that pl_lnk_cfg does not indicate any reserved encoding when pl_state_sts=RETRAIN or ACTIVE.
4	lp_port_activity	Verify that none of the lp_* signal change their values till pl_portmode_val is asserted
5	pl_speedmode_value	Verify that pl_speedmode does not indicate any reserved encoding when pl_state_sts=RETRAIN or ACTIVE.
6	lp_exit_cg_ack_assertion_aftr_exit_cg_req	Verify that LL asserts lp_exit_cg_ack, within user specified time, in response to pl_exit_cg_req assertion by PHY
7	pl_exit_cg_req_deassertion_bfr_exit_cg_ack	Verify that when lp_exit_cg_ack is deasserted, pl_exit_cg_req was 0. Also verify that lp_exit_cg_ack gets deasserted, within user specififed time, after pl_exit_cg_req gets deasserted
8	pl_exit_cg_req_val_wrt_rst_and_low_pwr_state	Verify that once asserted while exiting LP states or reset, pl_exit_cg_req stays asserted untill LTSSM advances
9	pl_exit_cg_req_is_glitch_free	Verify that once asserted, pl_exit_cg_req does not toggle till lp_exit_cg_ack gets asserted
10	lp_stallack_deasserted_at_reset	Verify that when reset is asserted, lp_stallack is deasserted
11	pl_stallreq_hgh_wn_stallack_deassrt	Verify that when PHY asserts pl_stallreq, lp_stallack is clear.
12	pl_state_sts_transit_frm_active_state	Verify that when pl_state_sts changes from ACTIVE to any other state, both pl_stallreq and lp_stallack are high.
13	pm_or_cg_rmvd_for_active_state_entry	Verify that PHY asserts pl_exit_cg_req before updating pl_state_sts to ACTIVE
14	lpif_exit_condition_from_reset_state	Verify that PL transition to the ACTIVE state upon observing lp_state_req == RESET (NOP) for at least one clock while pl_state_sts is indicating RESET; and then followed by observing lp_state_req == ACTIVE
15	stallack_low_when_stallreq_or_reset_deassrtd	Verify that when LL deasserts lp_stallack, pl_stallreq is deasserted or reset is asserted.
16	stallack_val_wrt_lp_valid_and_lp_irdyl	Verify that when LL asserts lp_stallack, lp_valid and lp_irdy are both deasserted as well on the same lclk edge.
17	pl_portmode_val_asserted_bfr_pl_protocol_vld	Verify that when PHY asserts pl_protocol_vld, pl_portmode_val was 1.
18	stallreq_low_wn_stallack_or_reset_assrtd	Verify that when PHY deasserts pl_stallreq, lp_stallack is high or reset is asserted. Also verify that pl_stallreq deasserts, within user specified time, after lp_stallack is asserted.
19	lpif_stallack_hgh_whn_stallreq_assrtd	Verify that when LL asserts lp_stallack, pl_stallreq is high or reset is asserted. Also verify that LL asserts lp_stallack, within user specified time, after PHY asserts pl_stallreq.

S.No	Protocol Check Name	Check Description
20	phy_exiting_pm_state_upon_request	Verify that PHY asserts pl_exit_cg_req, within user specified time, after LL updates lp_state_req from PM to ACTIVE
21	initiate_exit_cg_req_to_exit_pm	Verify that PHY asserts pl_exit_cg_req the pl_state_sts is updated to a non-PM value
22	remove_cg_when_ll_init_pm_exit	Verify that when pl_state_sts is L1.x and LL update on lp_state_req is seen, then lp_wake_req assertion must have taken place
23	lp_wake_ack_transition_wrt_lp_wake_req	Verify that pl_wake_ack is 1 only when lp_wake_req is asserted and gets cleared when lp_wake_req has de-asserted
24	pl_wake_ack_toggled_wrtr_lclk	Verify that pl_wake_ack only toggles when lclk is operational
25	transit_through_retrain_exit_frm_l1x_to_active	Verify that pl_state_sts gets updated to RETRAIN if previous pl_state_sts was a PM state. Also verify that after RETRAIN, next state is ACTIVE.
26	stall_req_ack_handshake	Verify that PHY asserts pl_stallreq, within user specified time, after LL updates lp_state_req to a PM state.
27	trdy_val_aftr_stallreqack_completion	Verify that after successful completion of Stall protocol, PHY keeps pl_trdy deasserted till pl_state_sts updates to ACTIVE
28	pl_state_sts_updated_to_pm	Verify that PHY updates pl_state_sts to the requested PM (decided after DAPM & L1.x resolution) state, within user specified time, on successful completion of Stall protocol.
29	pl_phyinl1_pl_phyinl2_val_in_l1_or_l2	Verify that PHY asserts pl_phyinl1 or pl_phyinl2, within user specified time, when lp_state_req was DAPM/L1.x or L2 respectively after successful completion of Stall protocol.
30	l1x_pl_state_sts_val_for_dapm	Verify that when LL requested DAPM, PHY updated pl_state_sts with any of the L1.x values.
31	l1x_substate_pl_state_sts_returned	Verify that when LL requested L1.x, PHY updated pl_state_sts with same or a lower L1.x value.

9.10.1 LPIF Protocol Check Coverage

LPIF Protocol checks generate coverage for Pass and Fail scenarios through the following attributes present in `svt_cxl_lpif_configuration` class:

enable_lpif_chk_cov	Attribute used to control LPIF protocol check coverage. When set to 1'b1, it enables LPIF check coverage.
lpif_check_cov_fail_ctrl	Attribute used to control LPIF FAIL protocol check coverage. When set to 1'b1, it enables LPIF FAIL check coverage.
lpif_check_cov_pass_ctrl	Attribute used to control LPIF PASS protocol check coverage. When set to 1'b1, it enables LPIF PASS check coverage.

9.11 LPIF Service Requests

These are the LPIF service requests that have been added:

1. DEASSERT_PL_TRDY: Directs PHY LPIF VIP to deassert pl_trdy signal of interface for number of lclk edges specified by 'num_clk_edges' attribute of service class. This service call would be ignored, if `pl_trdy` is low or `pl_state_sts` is not active.

   The following VIP sequence has been added to exercise this service request:
   `svt_lpif_deassert_pl_trdy_sequence`
2. ASSERT_PL_EXIT_CG_REQ: Directs PHY LPIF VIP to assert `pl_exit_cg_req` signal of interface. This service call would be ignored, if `pl_state_sts` is not active or if either of `pl_exit_cg_req` or `lp_exit_cg_ack` is 1.

   The following VIP sequence has been added to exercise this service request:
   `svt_lpif_assert_pl_exit_cg_req_sequence.`

9.12 LPIF XCHECK

To check the presence of 'X/Z' on LPIF interface signals, you must pass the macro `SVT_ENABLE_XCHECK` in the setup.

9.13 LPIF Specification Version 1.1 Support

1. New signals in compliance with LPIF specification v1.1 are added to the interface in Q-2020.03-T-20200428 release.
2. Configuration variable to enable LPIF spec version is added.

svt_cxl_lpif_configuration::lpif_spec_ver	Specifies LPIF specification version number. The default value remains LPIF spec 1.0 when not configured. Configuration type: Static When set to SVT_LPIF_SPEC_VER_1_0, LPIF spec 1.0 is enabled When set to SVT_LPIF_SPEC_VER_1_1, LPIF spec 1.1 is enabled

9.14 HVP Plans details

HVP Plan can be found at location

`<DESIGNWARE_HOME_PATH>/vip/svt/cxl_subsystem_svt/latest/doc / VerificationPlans`

Details of all Plans

Plan	Toplevel/subplan	Location
cxl_subsystem_svt_lpif_top_level_pan.hvp	Toplevel	VerificationPlans
cxl_subsystem_svt_link_lpif_dut_pc_subplan.hvp	Subplan	VerificationPlans/ protocol_check_plans/link_lpif_dut
cxl_subsystem_svt_phy_lpif_dut_pc_subplan.hvp	Subplan	VerificationPlans/ protocol_check_plans/phy_lpif_dut

Plan	Toplevel/subplan	Location
cxl_subsystem_svt_lpif_fc_subplan.hvp	Subplan	VerificationPlans/ functional_coverage_plans

Riders for using Specification Linking:

The specification used for spec linking is LPIF 1_1.The same specification must be present in parallel to subplans to leverage spec linking. You need to add the specification document here.

Usage Details:

DUT Operation	VIP Operation	Plan
Either	Either	cxl_subsystem_svt_lpif_fc_subplan.hvp
PHY LPIF	LINK LPIF	cxl_subsystem_svt_link_lpif_dut_pc_subplan.hvp
LINK LPIF	PHY LPIF	cxl_subsystem_svt_phy_lpif_dut_pc_subplan.hvp

Attributes Used:

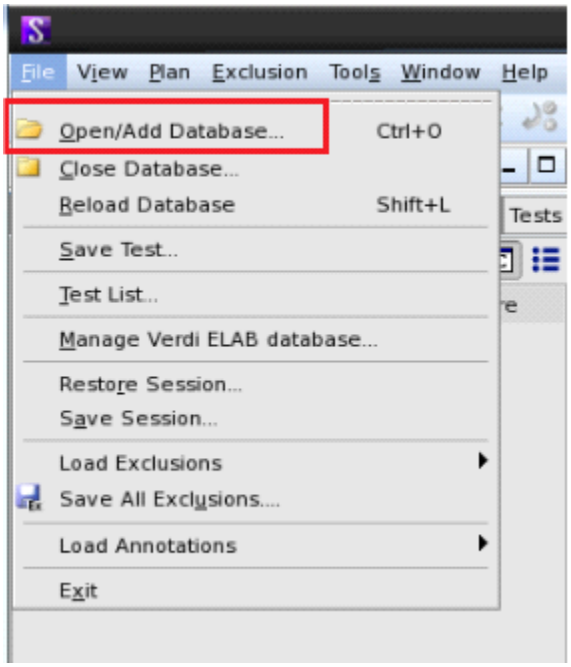
Attributes	Default value	Description
ENV_PATH	env.device_env.cache_mem_lpif	Path to AGENT ENV
AGENT_ID	0	Agent ID number

Back annotating with Verdi

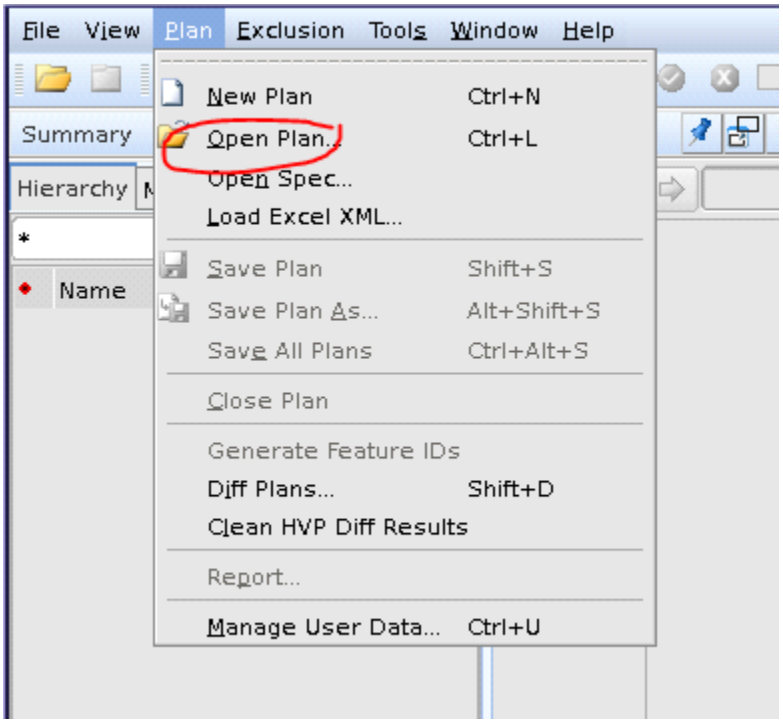
a. To invoke Verdi coverage, you need to add the -cob option to the Verdi command line. You can also specify the directory name in the command line while invoking Verdi coverage as shown below:

```
> verdi -cov &
> verdi -cov -covdir <filename>.vdb &
```

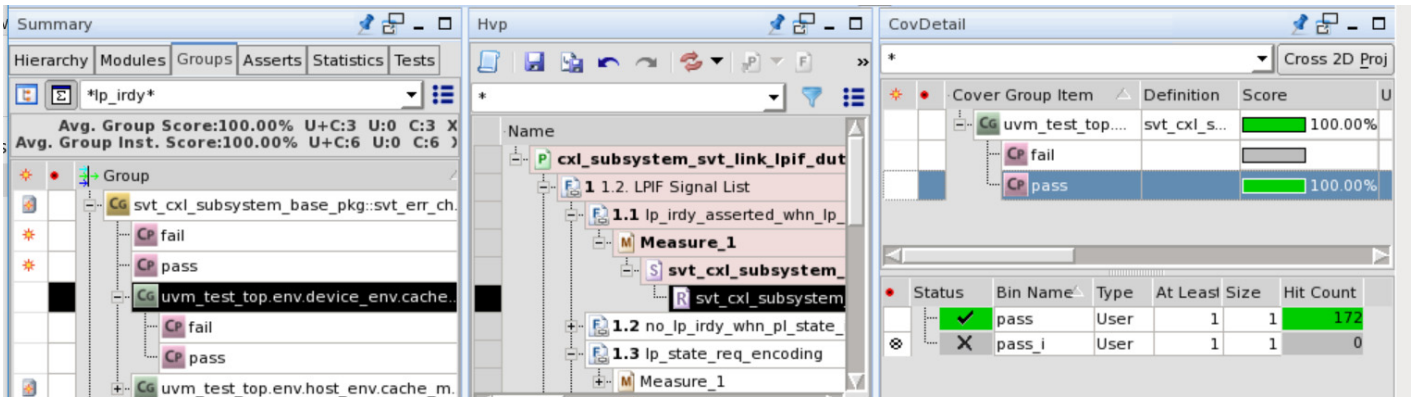
b. In the Verdi GUI, select File > Open/Add Database to load the database.



c. Select Plan > Open Plan to load the HVP plan.

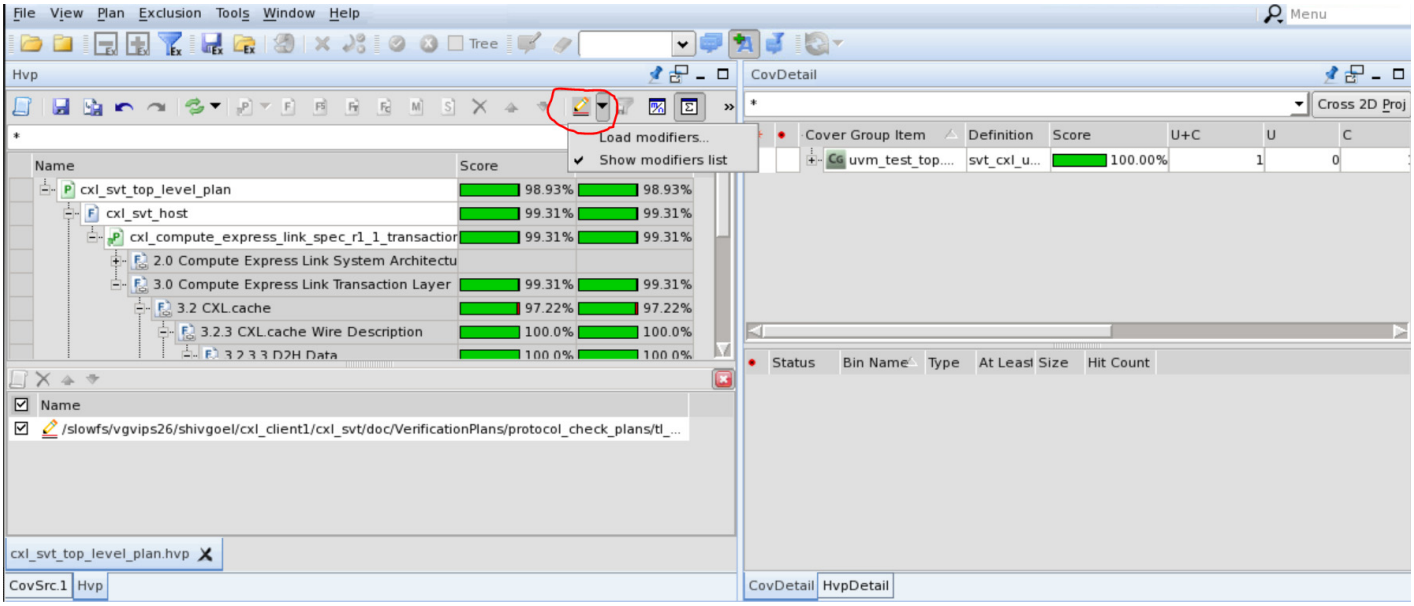


d. Opening the plan will automatically back annotate the coverage based on the HVP.



e. For Cache/Mem TL Protocol checks, load modifiers to filter the checks based upon Agent Type. Click this Load modifiers icon in the Hvp window.

Select the file `tl_protocol_check_coverage_filter.hvpmod` from the Hvp window.



9.14.1 CXL Cache/Mem HVP Plans

`<DESIGNWARE_HOME_PATH>/vip/svt/cxl_svt/latest/doc/VerificationPlans`

Details of all plan:

Plan	Top Level/ Sub plan	Location
cxl_svt_checker_coverage_top_level_plan.hvp	Toplevel	`VerificationPlans/protocol_check_plans`

Plan	Top Level/ Sub plan	Location
cxl_compute_express_link_spec_r1_1_transaction_layer_pc_subplan.hvp	Subplan	`VerificationPlans/ protocol_check_plans`
cxl_compute_express_link_spec_r2_0_transaction_layer_pc_subplan.hvp	Subplan for CXL 2.0	`VerificationPlans/protocol_check_plans`

Riders for using Specification Linking:

The specification used for spec linking is CXL 1.1. The same specification must be present in parallel to subplans to leverage spec linking. You need to add the specification document here.

Attributes Used:

Attribute	Description
ENV_PATH	Path to AGENT ENV
AGENT_ID	AGENT ID number
APPLICABLE_AGENT_TYPE	The agent type for which the protocol check is applicable

Modifier for CXL.Cache/Mem TL Protocol checks:

Modifier name: `tl_protocol_check_coverage_filter.hvpmod`

Location: `VerificationPlans/ protocol_check_plans`

Refer */VerificationPlans/protocol_check_plans/Readme_Protocol_Checks_Plan* for further details.

9.14.2 CXL IO HVP Plans

`<DESIGNWARE_HOME_PATH>/vip/svt/cxl_subsystem_svt/latest/doc/VerificationPlans`

9.14.2.1 CXL IO Protocol Check HVP Plans

Details of all plan:

Plan	Top Level/ Sub plan	Location
cxl_io_pcie_express_checker_coverage_top_level_plan.hvp	Toplevel	`VerificationPlans/protocol_check_plans/cxl_io`
cxl_io_pcie_express_base_spec_r5_v1_data_link_layer_pc_subplan.hvp	Subplan	`VerificationPlans/protocol_check_plans/cxl_io`
cxl_io_pcie_express_base_spec_r5_v1_transaction_layer_pc_subplan.hvp	Subplan	`VerificationPlans/protocol_check_plans/cxl_io`

Attributes Used:

Attribute	Description
ENV_PATH	Path to AGENT Environment.
APPLICABLE_AGENT_TYPE	Indicates the VIP Agent type for which the protocol check is applicable.
APPLICABLE_GROUP	Indicates the group feature relates to.
APPLICABLE_SUB_GROUP	Indicates the sub group feature relates to.

Modifiers for Protocol Checks:

Modifiers	Description
protocol_check_coverage_for_host_dut_filter.hvpmod	Filter to load protocol check coverage for HOST DUT.
protocol_check_coverage_for_device_dut_filter.hvpmod	Filter to load protocol check coverage for DEVICE DUT.
dl_only_protocol_check_coverage_filter.hvpmod	Filter to load protocol check coverage for DL Layer.
tl_only_protocol_check_coverage_filter.hvpmod	Filter to load protocol check coverage for TL Layer.

Location: `VerificationPlans/protocol_check_plans/cxl_io`

Refer */VerificationPlans/protocol_check_plans/cxl_io/Readme* for further details.

9.14.2.2 CXL IO Functional Coverage HVP Plans

Details of all plan:

Plan	Top Level/ Sub plan	Location
cxl_io_pcie_express_functional_coverage_top_level_plan.hvp	Toplevel	`VerificationPlans/_functional_coverage_plans/cxl_io`
cxl_io_pcie_express_base_spec_r5_v1_transaction_layer_fc_subplan.hvp	Subplan	`VerificationPlans/_functional_coverage_plans/cxl_io`
cxl_io_pcie_express_base_spec_r5_v1_data_link_layer_fc_subplan.hvp	Subplan	`VerificationPlans/_functional_coverage_plans/cxl_io`

Attributes Used:

Attribute	Description
ENV_PATH_FC_CXL_IO_DL	Path to DL AGENT.

Attribute	Description
ENV_PATH_FC_CXL_IO_TL	Path to TL AGENT.
APPLICABLE_AGENT_TYPE	Indicates the VIP agent type for which the check is applicable.
DIRECTION_TX	Indicates the direction TX for Coverage items.
DIRECTION_RX	Indicates the direction RX for Coverage items.

Modifiers for Protocol Checks:

Modifiers	Description
protocol_check_coverage_for_host_dut_filter.hvpmod	Filter to load protocol check coverage for HOST DUT.
protocol_check_coverage_for_device_dut_filter.hvpmod	Filter to load protocol check coverage for DEVICE DUT.
dl_only_protocol_check_coverage_filter.hvpmod	Filter to load protocol check coverage for DL Layer.
tl_only_protocol_check_coverage_filter.hvpmod	Filter to load protocol check coverage for TL Layer.

Location: `VerificationPlans/protocol_check_plans/cxl_io`

Refer */VerificationPlans/protocol_check_plans/cxl_io/Readme* for further details.

10 Functional Coverage

The CXL subsystem provides notification routines which users can utilize for functional coverage. The notifications are called inside of a class which can easily be extended by users to meet their specific needs. A set of default covergroups is provided, which you can use some or all of.

10.1 Enabling Functional Coverage

10.1.1 TL Coverage

For enabling the TL functional coverage, you can use the configuration control as suggested:

- For coverage of TL:

  ```
  svt_cxl_cache_mem_configuration::enable_tl_cov
  ```

  This needs to be set for both positive and error coverage.

- For coverage of TL transactions:

  ```
  svt_cxl_cache_mem_configuration::enable_tl_transaction_cov
  ```

  This needs to be set for both positive and error coverage

- For Error scenario coverage:

  ```
  svt_cxl_cache_mem_configuration::enable_tl_err_condition_cov
  ```

  This needs to be set to cover error scenarios.

By default, all these are set to 0. Each of these controls has to be set to 1 to enable corresponding coverage.

10.1.2 DL Coverage

For enabling the DL functional coverage, you can use the configuration control as suggested:

```
svt_cxl_cache_mem_configuration:: enable_dl_cov
svt_cxl_cache_mem_configuration::enable_dl_feature_fxn_cov[`SVT_CXL_MAX_FXN_COVERAGE_FEA
TURE]='{`SVT_CXL_MAX_FXN_COVERAGE_FEATURE{1}}
```

10.1.3 ARM/MUX Coverage

For enabling the ARM/MUX functional coverage, you can use the configuration control as suggested:

```
svt_cxl_arb_mux_configuration:: enable_arb_mux_cov
svt_cxl_arb_mux_configuration::enable_arb_mux_feature_fxn_cov[`SVT_CXL_ARB_MUX_MAX_FXN_C
OVERAGE_FEATURE]='{`SVT_CXL_ARB_MUX_MAX_FXN_COVERAGE_FEATURE{1}}
```

Covergroups are constructed only when each of these configuration parameter is set to 1.

For detailed information on Functional coverage, refer to the "Coverage" tab available in Class Reference.

Verification plans are available in doc directory in the installation. Usage details are provided in *Readme* files in each plan category.

10.1.4 CXL IO Coverage

The subsystem VIP supports the CXL IO Coverage for the following components:

- CXL IO Transaction Layer
- CXL IO Data Link Layer

For enabling the IO functional coverage, you can use the configuration control, `svt_cxl_subsystem_configuration:: enable_io_cov`

The CXL IO Coverage is verified only for VCS simulator.

11
Debug Features

To ease the debugging process, Synopsys CXL Subsystem Verification IP provides you the following logging support:

- CXL.cache/mem Transaction Logger
- CXL.io Transaction Logger
- CXL.io Flit Logger
- ARB/MUX Transaction Logger
- Symbol Logger

Transaction logging feature is supported for CXL IO/ Cache/ mem Protocols of CXL Subsystem SVT for capturing the information at the following layers

- Transaction Layer and Data Link Layer:

  At this level, VIP can capture both CXL.io transactions (TLPs, DLLPs) and CXL.Cache/mem transactions into separate log files. Those are CXL.io Transaction Logger and CXL.cache/mem Transaction Logger respectively.

  Along with CXL.io Transaction Logger (which captures the information about TLPs and DLLPs transmitted from and received by VIP), CXL.io Flit Logger is also supported. This log captures the CXL.io flits handshake between CXL.io Data Link layer and ARB/MUX

- ARB/MUX Layer:

  At this level, the transactions arrived and created at ARB/MUX will be captured into ARB/MUX transaction log file.

- Flex bus Physical Layer:

  At this level, VIP captures the symbols transmitted and received (with respect to VIP) on the lanes into a separate log file.

By default, no transaction log file is generated (all the transaction log files generation is disabled by default). To enable all the transaction log files generation, use the following command line option.

CXL Subsystem VIP specific command line option:

SVT_CXL_SUBSYSTEM_ENABLE_LOGGING=1

Or

Configurations available for enabling individual log file generation can also be used.

11.1 CXL.cache/mem Transaction Logger

11.1.1 Printing CXL.cache/mem Transaction Data into Log file

Transaction Layer trace creates log files of CXL Cache/mem transactions. Logging enabling attribute would be `svt_cxl_cache_mem_configuration::enable_transaction_logging`.

Example code present in the test to enable the transaction logging for the HOST through configuration is shown here:

```
for (int i = 0 ; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_cache_mem_agent; i++)
begin
cust_cfg.host_cfg.cache_mem_sys_cfg.cache_mem_cfg[i].enable_transaction_logging =
1;
end
```

By default, the log gets generated / saved with file name as mentioned below

`<test_name>_<host/device_components_instance_number>_cm_<host/device> appended with .xact_log.`

For example, if the test - cxl_tl_random_mem_wr_rd is ran then the CXL.cache/mem transaction log file will gets generated with the following file name:

cxl_tl_random_mem_wr_rd_0_cm_device.xact_log (if VIP is Device)

cxl_tl_random_mem_wr_rd_0_cm_host.xact_log (if VIP is Host)

If you want to print the transaction log with user-defined name, add it as:

```
for (int i = 0 ; i < cust_cfg.host_cfg.cache_mem_sys_cfg.num_cache_mem_agent; i++)
begin
cust_cfg.host_cfg.cache_mem_sys_cfg.cache_mem_cfg[i].transaction_log_filename=
"debug_log";
end
```

If different components are programmed to generate the transaction log with same name then a combined transaction log will be generated.

11.1.2 Fields of the CXL.cache/mem Transaction Log Header

The fields of CXL.Cache/mem Transaction log header are described in this section. These fields are listed from left to right as they appear on the header.

Sample header:

Reporter	Start Time (ns)	End Time (ns)	DIR	Type	Obj Num	TC	Tag/ ID	Address [Hex]	Response / Opcode	Meta Field	Meta Value	Snp Type	RSP pre	Rsp data	NT	Data [Hex]

Field: Reporter

Description:

This field represents an instance of the VIP in the test environment. The transaction log information is reported for this VIP instance.

Field: Start Time (ns)

Description:

This field represents the simulation time in "ns" when the transaction is started.

Field: Start Time (ns)

Description:

This field represents the simulation time in "ns" when the transaction is ended.

Field: DIR

Description:

This field represents the direction of the transaction, based on the source and destination components

Field: Type

Description:

This field represents the Request Transaction type

Field: Obj Num

Description:

This field represents the Object Number which is a unique number being assigned by VIP to each transaction, N/A for channel transaction

Field: TC

Description:

This field represents the Traffic Class of the Request

Field: Tag / ID

Description:

This field represents the Tag for the mem transactions and ID for the cache transaction

Field: Address

Description:

Address of transaction

Field: Response/ Opcode

Description:

Response type for cxl_transaction, Request / response Opcode for the cxl_channel_transaction

Field: Meta Field

Description:

Meta Field, applicable only for Mem transaction

Field: Meta Value

Description:

Meta Value, applicable only for Mem transaction

Field: Snp Typ

Description:

Snoop type associated with the transaction

Field: Rsp pre

Description:

Rsp pre field of response

Field: Rsp Data

Description:

Rsp Data field of transaction

Field: NT

Description:

Non-Temporal type of transaction

Field: DATA

Description:

Data involved with the data transfer

DATA print:

Print of DATA in the log will be in 4 double word format(2 MSB, 2 LSB are picked).

11.2 CXL.io Transaction Logger

All the CXL.io traffic (inbound or outbound transactions) to and from CXL Subsystem VIP are sent to the CXL.io Transaction Logger. These transactions are distilled and written (one transaction per line) to the transaction log file.

By default, the transaction logger is disabled. To enable it, and cause it to start writing transactions to a file, use the `enable_transaction_logging` member of the `svt_pcie_configuration` class. The transaction log filename can also be mentioned.

11.2.1 Printing CXL.io Transaction Data into Log File

In the following code, it is shown how to print TLP payload data to the Transaction log.

You must first enable transaction logging. By default, it is off. Also, set the filename of the transaction log.

```
for (int i = 0 ; i < cust_cfg.host_cfg.num_cxl_io ; i++) begin
        cust_cfg.host_cfg.cxl_io_cfg[i].pcie_cfg.enable_transaction_logging = 1'b1;
        cust_cfg.host_cfg.cxl_io_cfg[i].pcie_cfg.transaction_log_filename =
"cxl_io_transaction.log");
      end
```

Next, set how many dwords of the payload you want the model to write into the transaction log file.

/** Set the payload display limit */

```
svt_pcie_dl_disp_pattern::default_max_payload_print_dwords = 1024;
```

Note the default is zero. In the example, it has been set to 1024.

11.2.2 Fields of the Transaction Log Header

The fields of the transaction log header are described in this section. These fields are listed from left to right as they appear on the header.

Field: Reporter

Description:

This field represents an instance of the VIP in the test environment. The transaction log information is reported for this VIP instance.

Field: Start Time (ns)

Description:

This field represents the simulation time in ns when the transaction starts.

Field: End Time (ns)

Description:

This field represents the simulation time in ns when the transaction ends.

Field: Dir

Description:

This field represents the direction of the transaction from the VIP instance. "T" represents a transmit

transaction. "R" represents a receive transaction.

Field: TLP Type/DLLP Type

Description:

This field represents the type of TLP (Transaction Layer Packet) or DLLP (Data Link Layer Packet) as defined in Table 2-3 and Table 3-1 of the PCIe Specification respectively. For TLP memory read (MRd) and memory write (MWr) packets, a "32" or "64" is appended to the type. This number represents a 32-bit or 64-bit memory addressing.

For example:

MRd32

MWr64

Field: R_ID /Tag | ST

Description:

This field has 2 different representations for a TLP transaction. The Requester ID and Tag are displayed, or the Steering Tag is displayed. This field is blank for DLLP transactions.

Field: Seq Num

Description:

This field represents the sequence number of the transaction.

Field: TC VC

Description:

This field represents the value of the Traffic Class field of the TLP.

Field: TH

Description:

This field represents the 1-bit TH field of the common TLP packet header. The TH field is an indication of the TLP Processing Hints (TPH) and the Optional TPH TLP Prefix when applicable presented in the TLP header.

Field: PH

Description:

This field represents processing hint.

Field: IDO RO

Description:

These 2 fields represent the Ordering Attribute Bits as defined in Table 2-10 of the PCIe Specification.

The table is shown below:

Attribute Bit [2] (IDO)	Attribute Bit[1] (RO)	Ordering Type	Ordering Model
0	0	Default Ordering	PCI Strongly Ordered Model
0	1	Relaxed Ordering	PCI-X Relaxed Ordering Model
1	0	ID-based Ordering	Independent ordering based on Requester/Completer ID
1	1	Relaxed Ordering plus ID-Based Ordering	Logical "OR" of Relaxed Ordering and IDO

Field: NS

Description:

This field represents the No Snoop bit value of the TLP.

Field: Address

Reg#/MsgRt/Cpl

HdrFC DataFC

Description:

This field has multiple representations. For a TLP transaction, the value(s) displayed depends on the TLP type as shown in the following table. For a DLLP transaction, the Flow Control header and data are displayed.

TLP Field	Description
<address>	Memory Request: This field presents the memory
<address>	IO request: This field represents the IO address

TLP Field	Description
BDF: <> R: <> or BDF:<> O: <>	Configuration request: "BDF" represents the bus device function "R" represents the register number "O" represents the register byte offset. This offset is not displayed by default. To enable the display, you must set the dl_trace_options[1] attribute of the PCIe configuration class (svt_pcie_configuration). For example: <agent_cfg>.pcie_cfg.dl_trace_options[1]=1
<message>	Message request: This field represents the message routing as defined in Table 2-18 of the PCIe Specification.
ID: <> Stat: <>	Completion request: "ID" represents the Completion ID. "Stat" represents the Completion status as defined in Table 2-29 of the PCIe Specification. The status are SC, UR, CRS and CA.

DLLP Field	Description
<header> <data>	Flow Control header and data

Field: BE | ST

BC

MCode

Description:

This field has multiple representations of a TLP transaction.

TLP Field	Description
<byte enable>	Memory request/IO request/ Configuration request: This field represents the byte enable.
<steering tag>	Memory request: When the TH field has a value of "1", this field represents the steering tag value
BC: < >	Completion request: BC represents the byte count
<message code>	Message request: This field represents the message code as define in Table F-1 of the PCIe Specification

Field: Len/Idx DW

Description:

This field has 2 representations of a TLP transaction.

TLP Field	Description
<payload length>	For the TLP Header displayed on the first row of data, this field represents the length of the payload in double word (DW)
<data index>	For the TLP payload displayed from the second row of data and on, this field represents the accumulative count of DW data

Field: Prefix / Header / Data

(All values in Hex)

Description:

This field represents the raw dword values of a TLP transaction.

- Values with an "(H)" prefix represent header DWORDs.
- Values with a "(P)" prefix represent the PCIe Prefix DWORDS.
- When the model is configured to show transaction data, values with a "(D)" prefix represent payload DWORDS.

By default, only the TLP header data is displayed along with the header fields. You can enable the display of payload data by using one of the following methods:

a. In the build phase of the simulation, set the static attribute `default_max_payload_print_dwords` of class `svt_pcie_dl_disp_pattern` to the default maximum number of payload DWORDs to be displayed.

b. Set the `SVT_PCIE_XACT_LOG_MAX_PAYLOAD_DWORDS_DEFAULT` macro to the default maximum number of payload DWORDs to be displayed.

Payload data for CfgWr and CfgRd (corresponding CplD) are displayed with a single DWORD. By default, this DWORD value is displayed in the Big Endian format. Alternatively, you can enable the DWORD to be displayed in the Little Endian format by setting the "dl_trace_options[0]" attribute of the PCIe configuration class (svt_pcie_configuration).

For example:

```
Default Big Endian payload data
0 06001000
Enable Little Endian format
<agent cfg>.pcie_cfg.dl_trace_options[0] = 1;
Little Endian payload data
0 00100006 LittleEndian
```

Field: EP

Description:

This field represents the poison bit as defined in the PCIe Specification.

Field: ECRC
Description:
This field represents the ECRC value of a TLP as defined in the PCIe Specification.

Field: LCRC
CRC
Description:
This field has 2 representations. For a TLP transaction, the LCRC value as defined in the PCIe Specification is displayed. For a DLLP transaction, the CRC value as defined in the PCIe Specification is displayed.
Field: TX/RX Error
Description:
This field represents the type of error injection when error injection is enabled in a transmit (tx) transaction. For a receive (rx) transaction, this field represents the detected error.

Error Injection Type	Description
BadSeq	Illegal Sequence Number
CodeViol	TX code Violation
CrcEr	CRC Error
Disparity	Disparity Error
DupSeq	Duplicate Sequence Number
EIErr	Scenario injects error, then VIP reported this
HdrCRC	PCIE 8G Header CRC Error
HdrPAR	PCIE 8G Header Parity Error
LCRC	LCRC Error
NAK	NAK Received for TLP
NoACK	Missing ACK for Transaction
NoEND	Missing END
NoSTART	Missing START
NoSTP	NoSTART Missing START
NullLCRC	Nullified TLP with corrupt CRC
NullTLP	Nullified TLP
ReplCnt	Replay count of 4 exceeded

11.3 CXL.io Flit Logger

11.3.1 Printing CXL.io Flit handshaking into Log File

Logging of CXL.io Data Flit transmission and reception is possible by configuring the CXL Subsystem VIP like below.

- Enable the CXL.io Flit mode
- Enable the CXL.io Flit logging

Example usage:

```
<cust_cfg>.host_cfg.cxl_io_cfg[0].pcie_cfg.dl_cfg.enable_cxl_io_flit_mode = 1'b1;
<cust_cfg>.host_cfg.cxl_io_cfg[0].pcie_cfg.enable_cxl_io_flit_logging = 1'b1;
```

11.3.2 Fields of the CXL.io Flit Log Header

Sample Header

Time	TX/RX	Flit Data

The fields of the CXL.io flit log header are described in this section. These fields are listed from left to right as they appear on the header.

Field: Time

Description: This field represents the simulation time in ns when the transaction starts

Field: TX/RX

Description: This field represents the direction of the Flit. It will specify whether the flit is transmitted or received with respect to VIP.

Field: Flit Data

Description: This field represents the 64 byte Flit Data.

11.4 ARB/MUX Transaction Logger

VIP captures all the transactions from DL/PHY to ARB/MUX and from ARB/MUX to DL/PHY into the ARB/MUX Transaction Log file.

The transactions transmitted or received from Data Link layer and Physical layer are differentiated by specifying the direction field as 'DL_IO/DL_CM' and 'PHY' respectively.

11.4.1 Printing Mem Transaction Data into Transaction Log File

ARB/Mux Layer trace creates log files of CXL IO/Cache/mem input transactions coming from DL/PHY layer. Logging enabling attribute would be:

`svt_cxl_arb_mux_configuration::enable_transaction_logging`.

Example code present in the test to enable the transaction logging for the HOST is shown here:

```
if(cust_cfg.host_cfg.num_arb_mux != 0) begin
for (int i = 0 ; i < cust_cfg.host_cfg.num_arb_mux; i++) begin
cust_cfg.host_cfg.arb_mux_cfg[i].enable_transaction_logging = 1;
end
end
```

By default, the log gets generated/ saved with Hierarchy of the component appended with. xact_log.

Example:

```
uvm_test_top.env.device_env.arb_mux[0].driver.xact_log
uvm_test_top.env.host_env.arb_mux[0].driver.xact_log
```

If you want to print the transaction log with user-defined name, add it as:

```
if(cust_cfg.host_cfg.num_arb_mux != 0) begin
for (int i = 0 ; i < cust_cfg.host_cfg.num_arb_mux; i++) begin
cust_cfg.host_cfg.arb_mux_cfg[i].transaction_log_filename = "debug_log";
end
end
```

11.4.2 Fields of the ARB/MUX Transaction Log Header

Sample Header:

Reporter	Event Time (ns)	Flit Dir (Tx/ Rx)	VSLM Type (IO/ CM)	Request (IO/ CM/ ALMP) [Prv VLSM]	Flit Type IO DLLP/ TLP/ VLSM Req/Sts ====>	Flit/ DLLP SubType TLP Type R_ID _Tag/ ST VLSM Req/ Sts State [Curr VLSM]	Credit Return Count [Req | Data | Rsp]	Slot Format S0 S1 S2 S3	ACK	BE	SZ	CRC	| | |Phy | | | | |Ilr | |E- |Re-|Re- |Em-|Vi-|Wr-|Free|Wrap| |Seq|try|Init|pty|ra|| Ptr|Buf |Val |

Field: Reporter

Description:

This field represents the VIP component (HOST/DEVICE) with respect to which the Transactions received or transmitted are mentioned

Field: Event time (ns)

Description:

This field represents the time when ARB Mux is observing the event

Field: Flit Dir (Tx/Rx)

Description:

This filed represents the Flit/ALMP Transaction Direction (Tx/Rx) from VIP perspective.

Tx(DL_IO): Outbound Flit from IO DL to ArbMux ,

Tx(DL_CM): Outbound Flit from CM DL to ArbMux,

Tx(PHY): Outbound Flit/ALMP from ArbMux to PHY,

Rx(PHY): Inbound Flit/ALMP from PHY to ArbMux

Rx(DL_CM): Inbound Flit from ArbMux to CM DL

Field: VLSM Type (IO/CM)

Description:

This filed represents which VLSM detected Transaction. i.e IO or CM

Field: Request (IO/CM/ALMP) / [Prv VLSM]

Description:

This field has 2 representations.

Request(IO/CM/ALMP) - Indicates type of request IO/CM/ALMP_IO/ALMP_CM .
[Prv VLSM] - Represents Previous IO/CM VLSM State on VLSM State Transition.

Field: Flit Type / IO DLLP/TLP / VLSM Req/Sts

Description:

This field has 3 representations.

Flit Type - Indicates CM Flit type ADF (All Data Flit) or NON_ADF

IO DLLP/TLP - Indicates type of IO packet (DLLP/TLP).

VLSM Req/Sts - Indicates ALMP Type VLSM request or status

Field: Flit/ DLLP SubType / TLP Type R_ID_Tag/ST / VLSM Req/Sts State / [Curr VLSM]

Description:

This field has 4 representations.

Flit/DLLP SubType - Indicates CM flit type(RETRY/INIT/LLCRD..etc) and DLLP packet type (ACK/INITFC2_P_VC/INITFC1_CPL/UPDATEFC_NP_VC..etc)

TLP Type R_ID_Tag/ST - Indicates TLP Type (MWr32/MRd32/MWr64/MRd64/CplD ..etc)

VLSM Req/Sts State - Indicates VLSM State for requested ALMP request or status.

[Curr VLSM] - Represents Currents IO/CM VLSM State on VLSM State Transition.

Field: Credit Return Count [Req | Data | Rsp]
Description:
In case of Control Flit, this field represents the Credit return count for Req, Data and Rsp.
C - indicates Cache Credit information and M - indicates Mem Credit information

Field: Slot Format S0 S1 S2 S3
Description: This field represents the flit slot format.

Field: ACK
Description: This field indicates an acknowledgment of 8 successful flit transfers

Field: BE
Description:This field represents the Byte Enable field of the flit header

Field: SZ
Description:This Size field reflects the transmission of data at the half cache line granularity

Field: CRC
Description:This field represents CRC of the flit

Field: E-Seq | Retry | Phy-Reinit | Empty | Viral | WrPtr | FreeBuf | llrWrapVal
Description:
This field has multiple representations of the transaction.
E-Seq - Indicates the requester's retry sequence number
Retry - Indicates the local NUM_RETRY LLR variable.
Phy-Reinit - Indicates the NUM_PHY_REINIT LLR variable.
Empty - Indicates that the LLR contains no valid data.
Viral - Indicates that the
transmitting agent is in a Viral state
WrPtr - Indicates WrPtr value of the retry queue.
FreeBuf - Indicates free LLRB entries.
llrWrapVal - Indicates the expected sequence number for Control Flit.

The Flit details are printed when flit is received from IO/CM DL. When Stream is received from PHY layer, only flit type IO/CM is printed.

11.5 Symbol Logger

11.5.1 Printing Transmitted and Received symbols into Symbol Log File

One log file is created for each simulation. All agents share the log file. Each agent must be enabled independently as shown in this example. If more than one `symbol_log_filename` is set, then the last one set within the simulation serves as the filename. It is recommended that you have only one agent set the filename.

Example:

```
 for (int i = 0 ; i < cust_cfg.host_cfg.num_cxl_io ; i++) begin
      /** Enable Symbol logging */
      cust_cfg.host_cfg.cxl_io_cfg[i].pcie_cfg.enable_symbol_logging = 1'b1;
      cust_cfg.host_cfg.cxl_io_cfg[i].pcie_cfg.symbol_log_filename ="symbol.log"; //
Recommended to only set the filename once
  end
```

The symbol log filename is appended to the full hierarchical name of the port0 instance generating it.

11.5.2 Fields of the Symbol Log Header

The fields of the symbol log header are described in this section. These fields are listed from left to right as they appear on the header. Note that the Symbol logging is performed at the PIPE interface. If the PIPE interface is configured as multiple bytes, all bytes transferred at a time step are logged at the same time step.

Field: TIME

Description: This field represents the simulation time in ns.

Field: INSTANCE

Description: This field represents an instance of the VIP in the test environment. The symbol log information is reported for this VIP instance.

Field: < lane symbols >

Description: This field represents symbols on the active lane(s). The format of the field header is:

[R00] [R01] [R02] … [R<n>] | [T00] [T01] [T02] … [T<n>] Where, "R" represents the receive symbol on the lane. "T" represents the transmit symbol on the lane.

<n> is the number of the highest configured lane.

The encodings of the lane symbols are listed in the following tables.

Field: LTSSM State

Description: This field represents the state of the LTSSM state machine as defined in section 4.2.5 of the PCIe specification. For states such as L0 and L0s where the receive (rx) and transmit (tx) LTSSM states may diverge, the rx and tx states are displayed separately. Otherwise, only a single state is displayed for both rx and tx states.

11.5.3 Special Encodings

Special Symbol Encodings for All Link Data Rates

Symbol	Description
z	Electrical Idle
?	Invalid or unknown value
.	No information available to log. This may occur at startup, at changes to link speed or link width, or if the Rx and Tx sides are operating at offset time steps at either 2.5GT/s or 5 GT/s
q	Error injection pending: appended on each symbol that will have disparity inverted. Only applies on Tx lanes
j	Error injection pending: appended on each symbol that will have a random bit flipped. Only applies on TX lanes.
v	Error injection pending: appended on each symbol that will have an invalid encoding. Only applies on TX lanes.

For link operation at 8GT/s, symbols after the sync headers are prepended with encodings of the sync headers listed in this table.

Sync Header Encodings for Link Operation at 8GT/s

Symbol	Description
@	2'b'00 (Reserved)
*	2'b'01 (OS block)
=	2'b'10 (Data block)
$	2'b'11 (Reserved)

Additional encodings for link operation at 8GT/s are listed in this table.

Special Character Encodings for Link Operation at 8GT/s

Symbol	Description
::	Data skip cycle (no valid data)
+	Start of TLP Token. Prepended on 1st symbol of token. Replaces = symbol at start of block. Only noted on TX lanes.
^	Start of DLLP Token. Prepended on 1st symbol of token. Replaces = symbol at start of block. Only noted on TX lanes.

Symbol	Description
}	End of packet. Appended on last symbol of a tlp or dllp. Only noted on TX lanes
Q	Error injection pending on last symbol of a tlp or dllp: appended if last symbol has disparity inverted. Only applies on TX lanes.
J	Error injection pending on last symbol of a tlp or dllp: appended if last symbol has a random bit flipped. Only applies on TX lanes.
V	Error injection pending on last symbol of a tlp or dllp: appended if last symbol has an invalid encoding. Only applies on TX lanes.

11.5.4 Special Messages

Configuration Messages in the Symbol Log

In addition to lane symbols, messages that indicate link changes are displayed in the symbol log. These messages are prepended by "--".

For example:

spd_0 -- Detected change in link width from 1 to 4.

spd_0 -- Detected change in data rate to 32 Gt/s. TX Precoding: Off, RX Precoding: Off. See file header for special encodings.

Symbol Log with OS Information

The current symbol log format has been enhanced to include OS information. This enhanced format is enabled by setting the configuration attribute `svt_pcie_configuration::enable_enhanced_symbol_log_format` to 1. When the enhanced format is enabled, the symbol log will also contain information about when and what OS are terminated on individual lanes. The default value of `enable_enhanced_symbol_log_format` is 0. For most operating systems this would be on the last symbol of the OS, but for some operating systems that have indeterminate length like SKP OS the completion information would be associated with the COM/sync header of the next OS/data block.

11.5.5 Synchronization of Simulation Time Between Transaction Log and Symbol Log

Transaction logging times represent times at the periphery of the VIP. Symbol logging times are captured at the PIPE interface, which may be internal or external to the VIP depending on the interface type. For Serial and PMA interface, there is no correlation between the time displayed in the transaction log and the symbol log. Due to delay through the PHY layer, symbols are logged at a different time than the transaction log for the same packet.

For PIPE interface, the simulation time displayed in the transaction log and symbol log are synchronized for the same packet. The ‘Start Time’ of the transaction log corresponds to the time of a transaction with the "STP" or "SDP" symbol in the symbol log. The ‘End Time’ of the transaction log corresponds to the time of a transaction with the ‘END’ symbol in the symbol log.

Transaction Log	Symbol Log
Start Time (ns)	TIME with ‘STP’ or ‘SDP’ symbol
End Time (ns)	TIME with ‘END’ symbol

11.6 Enabling UVM Recording

UVM port recording helps in dumping the TLM ports information into a waveform which in turn helps in graphically visualizing the traffic in Verdi Protocol Analyzer. You can enable this feature by setting the mentioned flags in the Makefile:

```
Compile options     : +define+UVM_VERDI_NO_COMPWAVE +define+UVM_VERDI_PORT_RECORDING
Elaboration options : -lca -kdb -debug_acc+all
Run time options    : +UVM_TR_RECORD +UVM_LOG_RECORD
+UVM_VERDI_TRACE=HIER+TLM+MSG+PRINT+UVM_AWARE
```

An example is shown in `vcs_build_options`, `vcs_elab_options`, and `sim_run_options` files present in the examples as

$DESIGNWARE_HOME/examples/sverilog/cxl_test_suite_svt/tb_dut_cxl/

$DESIGNWARE_HOME/examples/sverilog/cxl_subsystem_svt/tb_cxl_subsystem_uvm_basic_sys

Note The `UVM_HOME` should be native to VCS and Verdi, that is, it should be set to *$(VCS_HOME)/etc/uvm* where VCS_HOME points to the VCS installation in your CAD area.

Running the simulation with above flags generate a `novas.fsdb` which can be viewed for transaction debug as

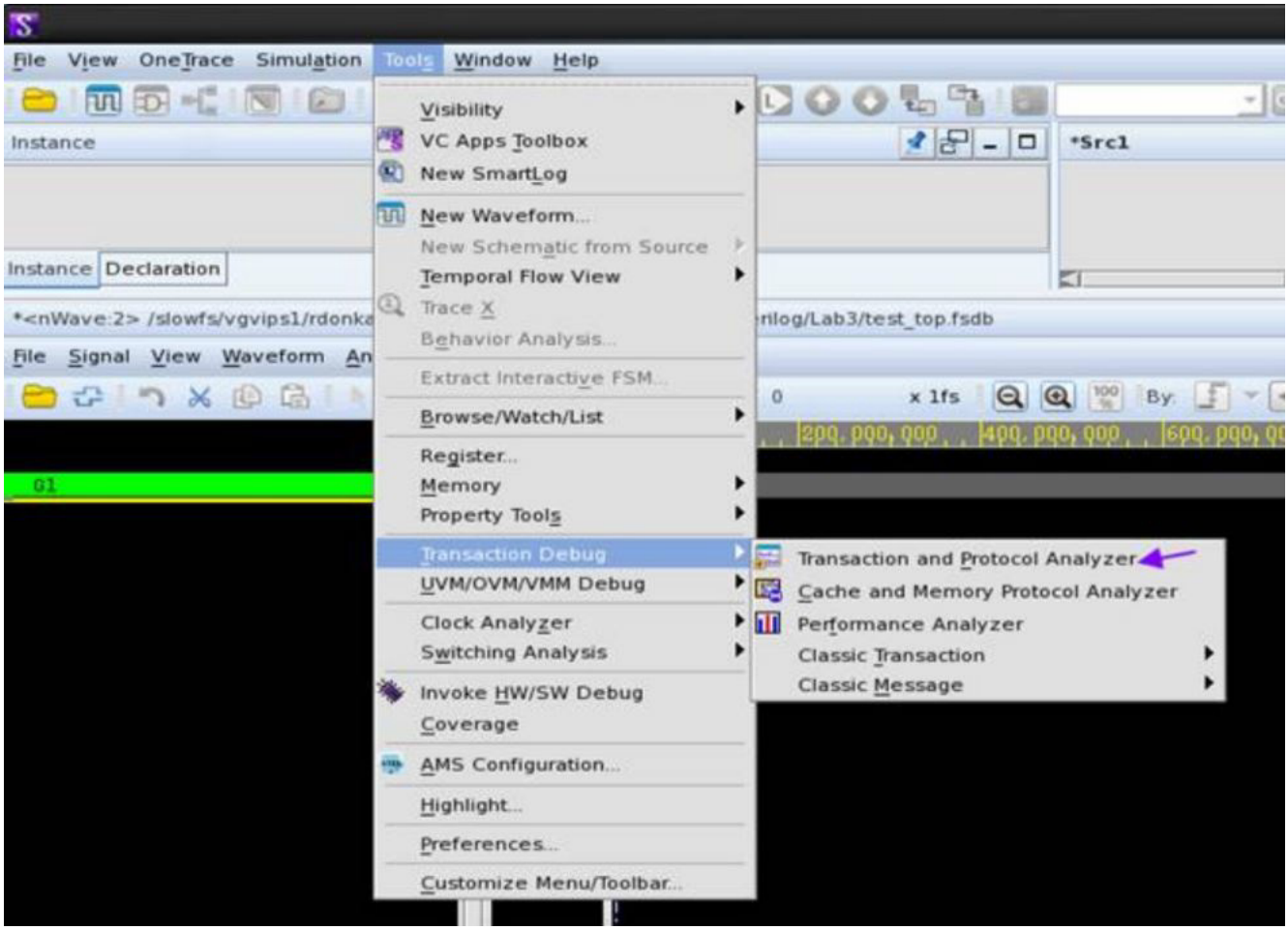
```
% verdi -ssf novas.fsdb &
```

11.6.1 Debugging CXL.mem/Cache transactions

To debug or track the CXL.mem/cache transactions (with various attributes available for debugging) using the Verdi Transaction and Protocol Analyzer, follow the mentioned steps:

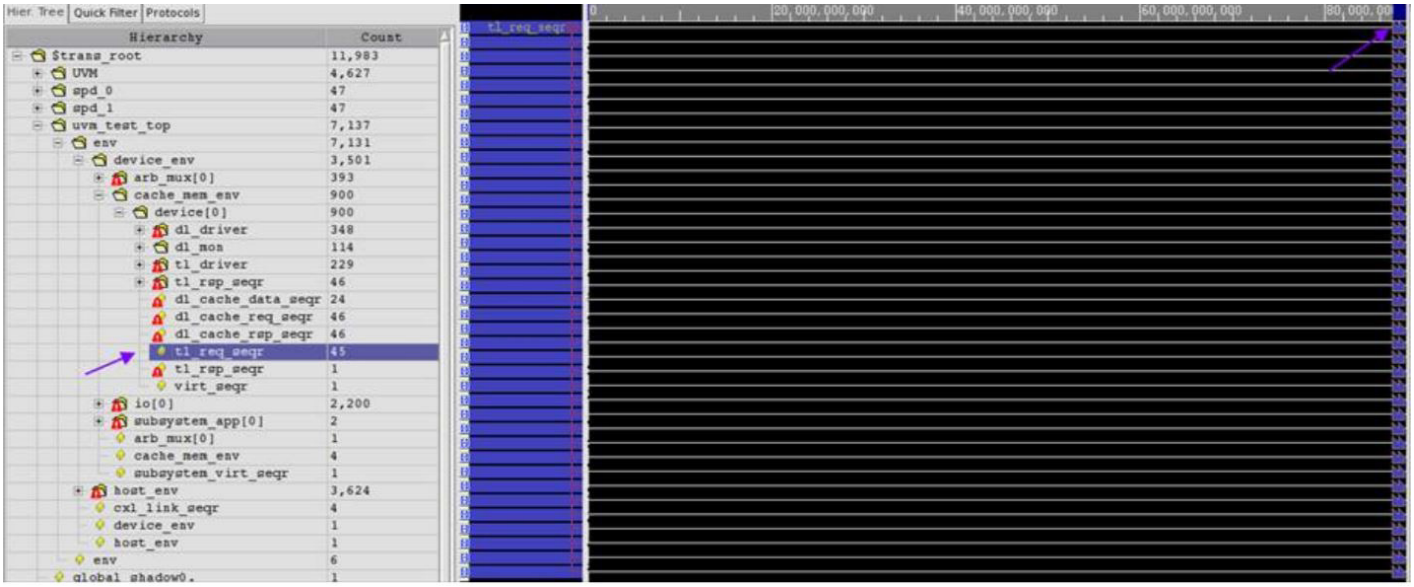
1. Open *Verdi* > *Tools* > *Transaction Debug* > *Transaction and Protocol Analyzer*.

Figure 11-1 Transaction and Protocol Analyzer



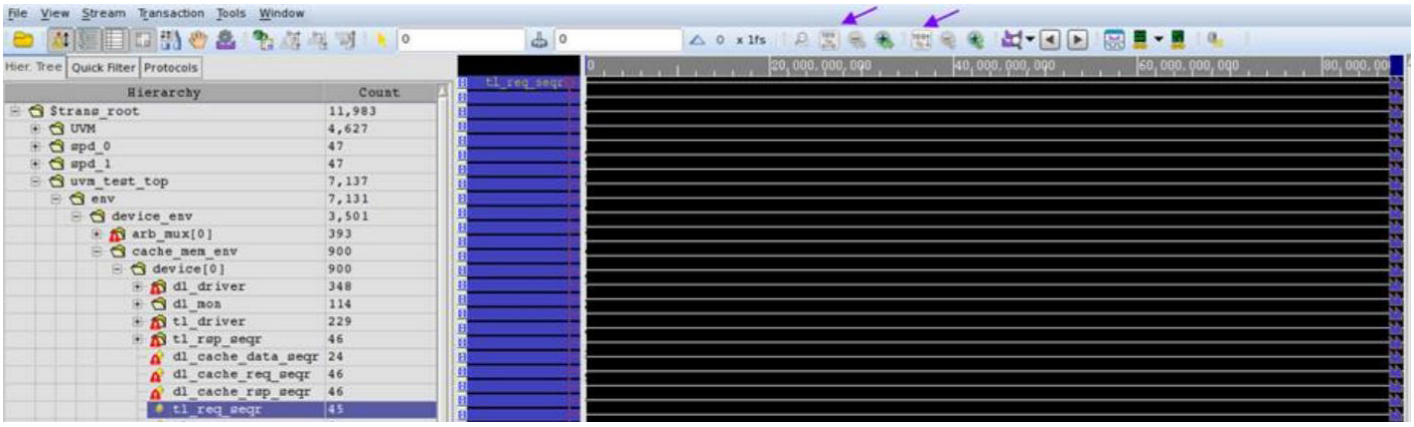
2. Click the sequencer hierarchy and check the required transaction to debug.
   - ✦ Transaction structure can be seen as a tree on the left pane (as illustrated in the figure below).
   - ✦ Add all those to the transaction activity area by clicking on sequencer given on the tree.

Figure 11-2 Checking Required Transaction to Debug



Click the two 100% options to view all the transactions.

Figure 11-3 Viewing All Transactions

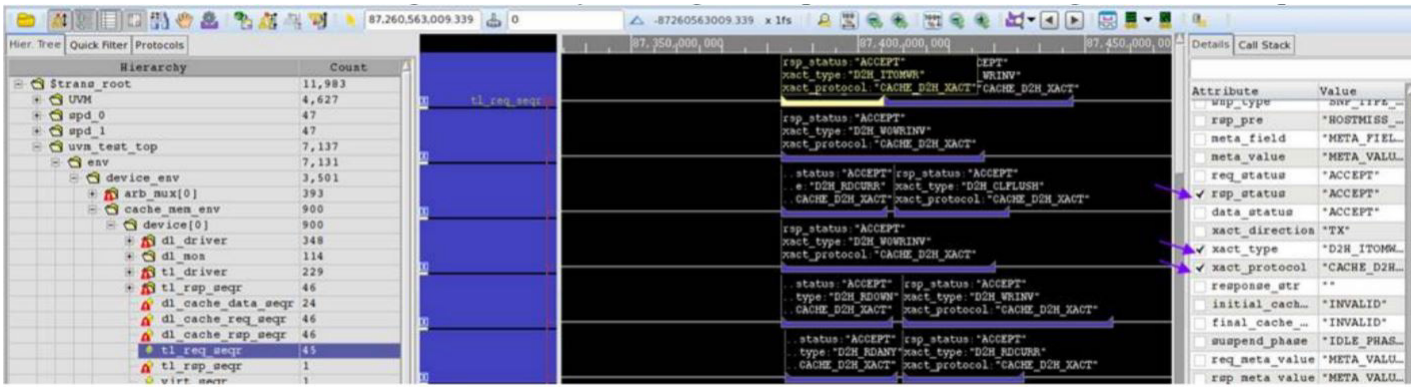


3. Load the interesting attributes related to the transaction based on your requirement.

   Select the particular object as shown in the figure.

Note You can see the interesting attributes by clicking the particular checkbox corresponding to each option.

Figure 11-4 Attribute Selection

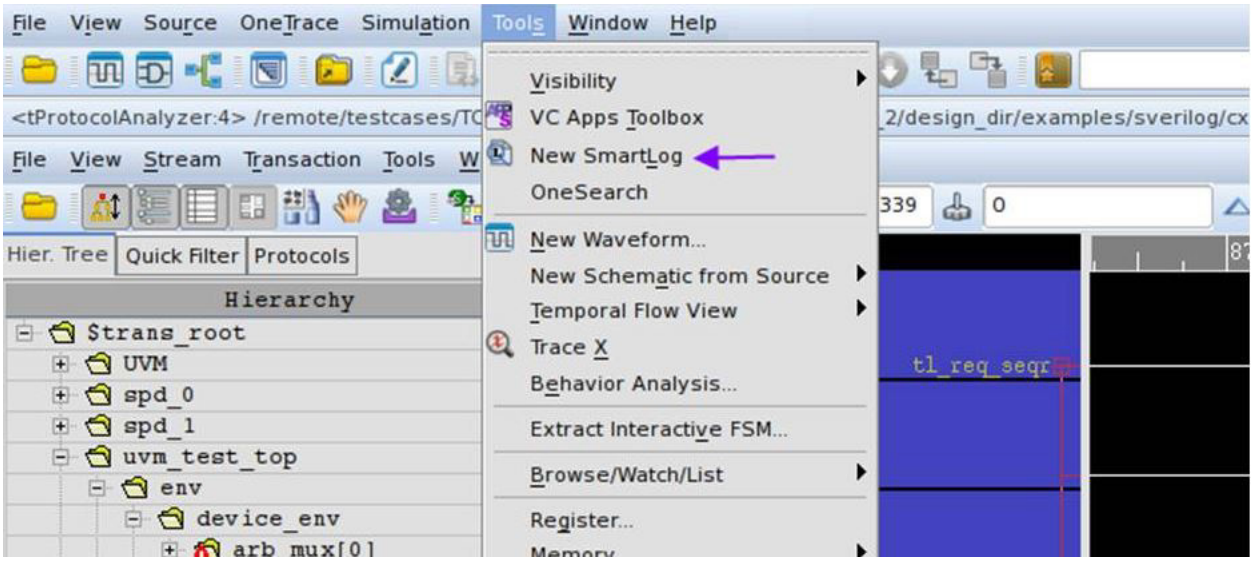


4. [**Optional**] Verify the error transaction by annotating the `UVM_WARNING` to the transaction display using Log file.

   - Load the log file into the `"SmartLog"` application in Verdi:

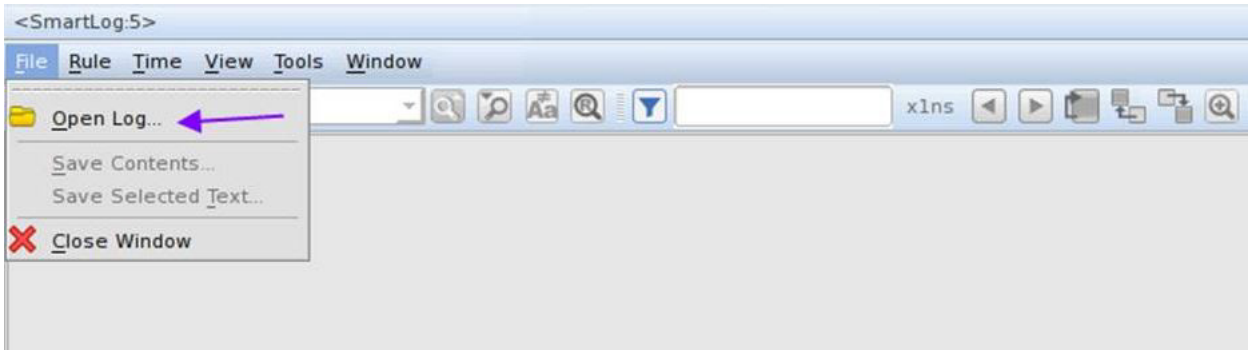
     Go to Menu (in Verdi) *Tools* > *New SmartLog*. This opens a new SmartLog window.

Figure 11-5 Opening SmartLog Window



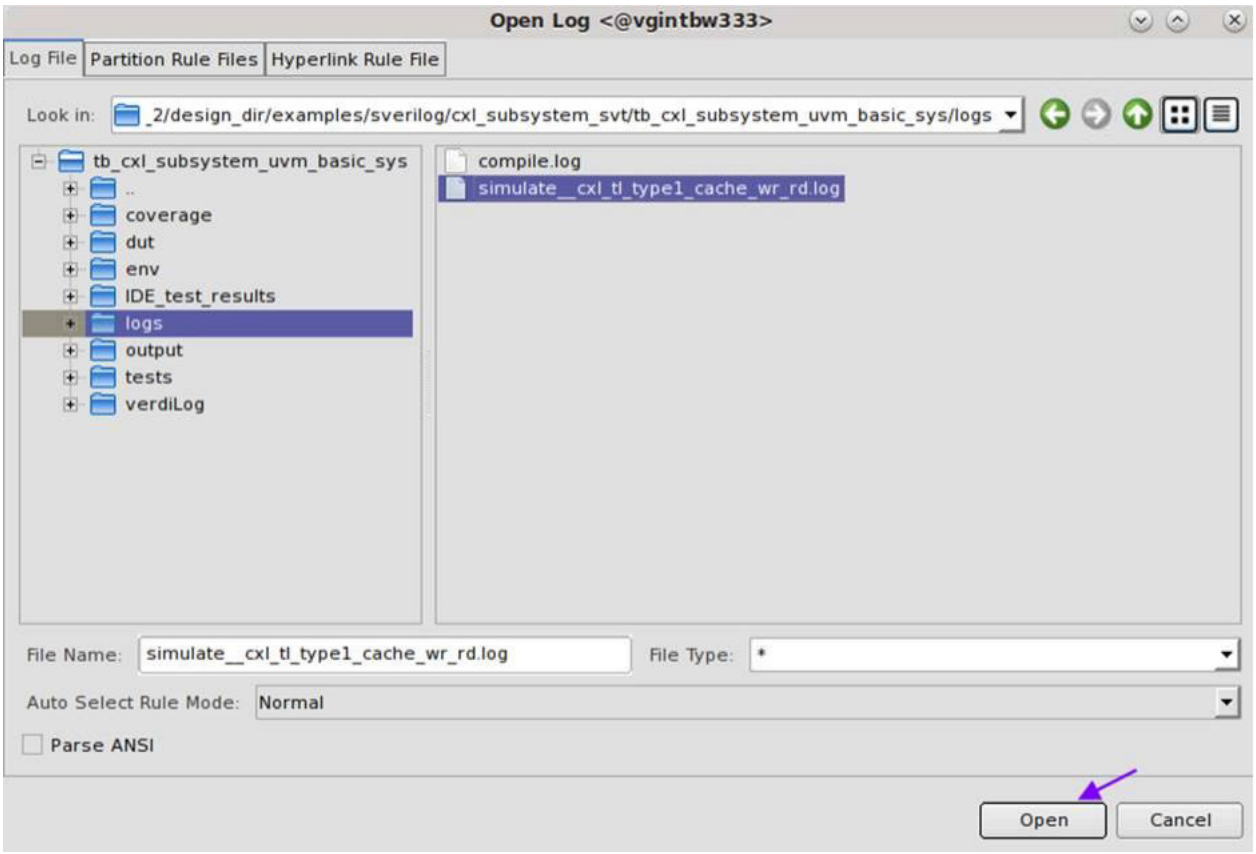
   - In the new SmartLog window, go to *File* > *Open Log*.

Figure 11-6 SmartLog Window



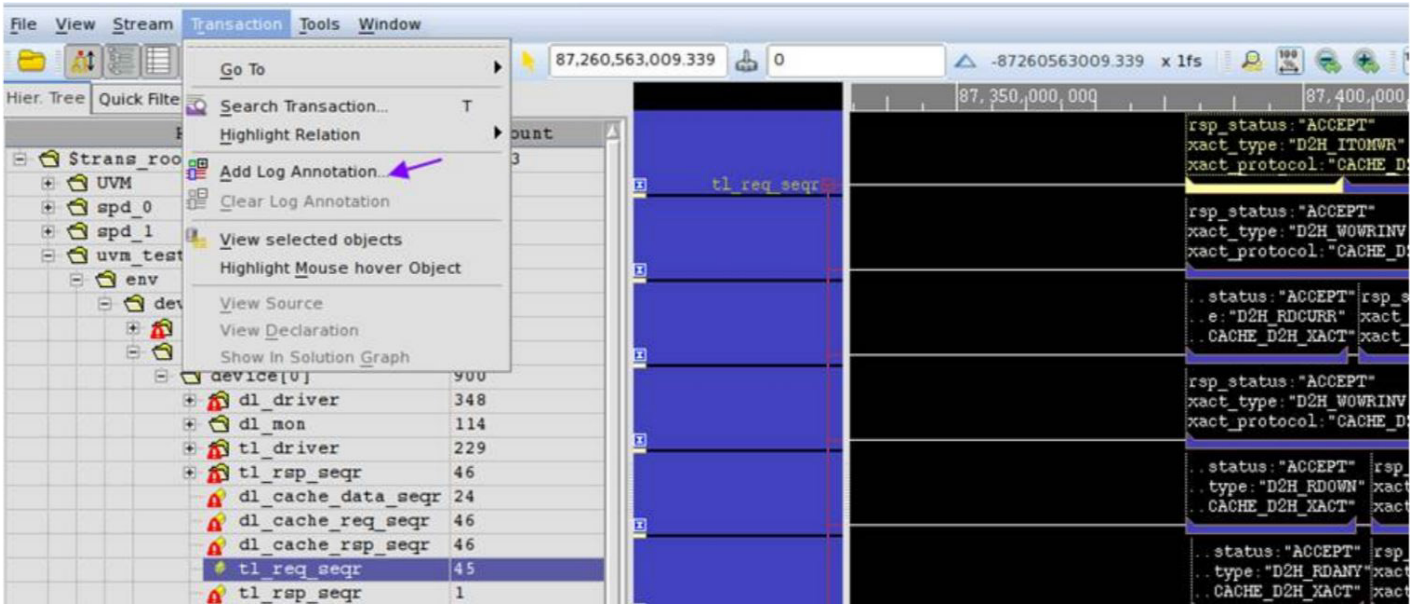
- Click *Open Log*. A pop up window appears. Open the simulation log file (for example, *logs/simulate_cxl_tl_type1_cache_wr_rd.log*).

Figure 11-7 Opening Simulation Log File



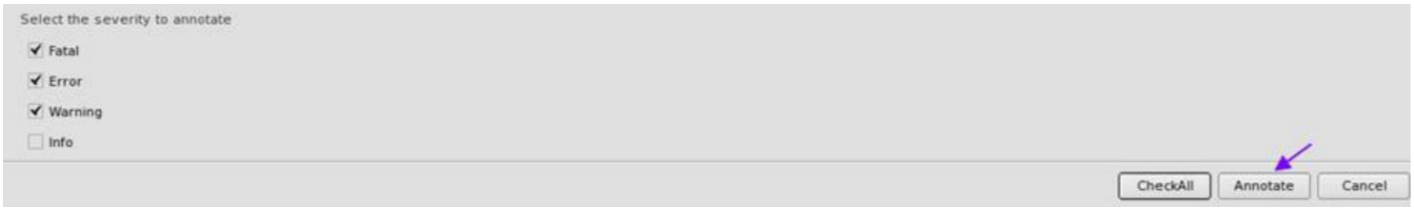
- Open the *.fsdb* file into PA (**Note**: If you have already opened Transaction and PA, waveform dump file in Verdi, then there is no need to open the file again. You can directly go to the tab) and click *"Add Log Annotation…"* action in *"Transaction"* menu in PA window.

Figure 11-8 Add Log Annotation



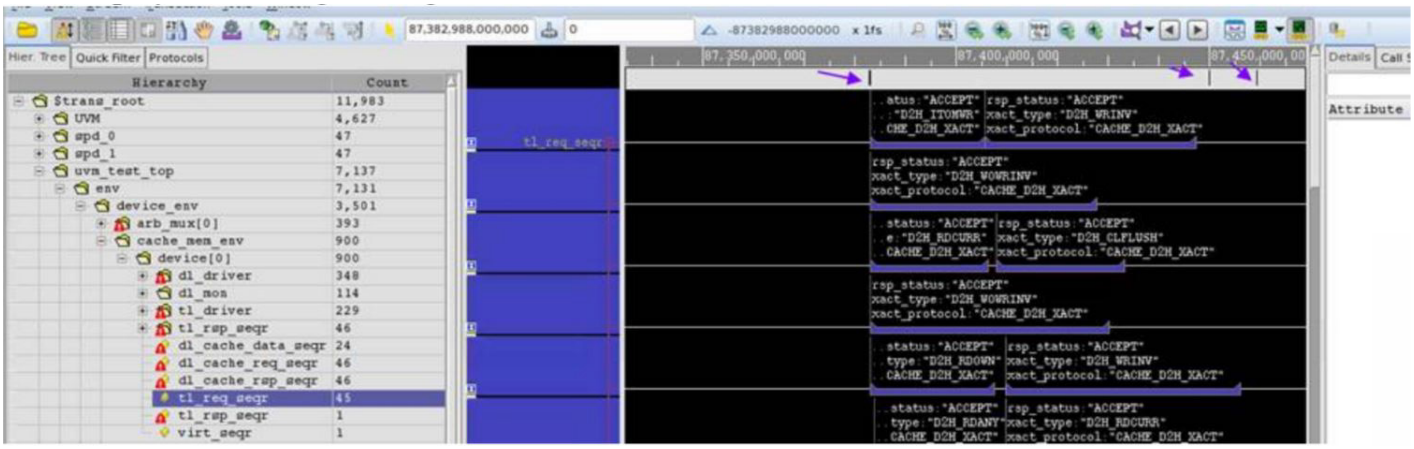
- Select the severity to annotate and click the *Annotate* option.

Figure 11-9 Log Severity Selection



This helps you to bring the log details into the transaction window as illustrated.

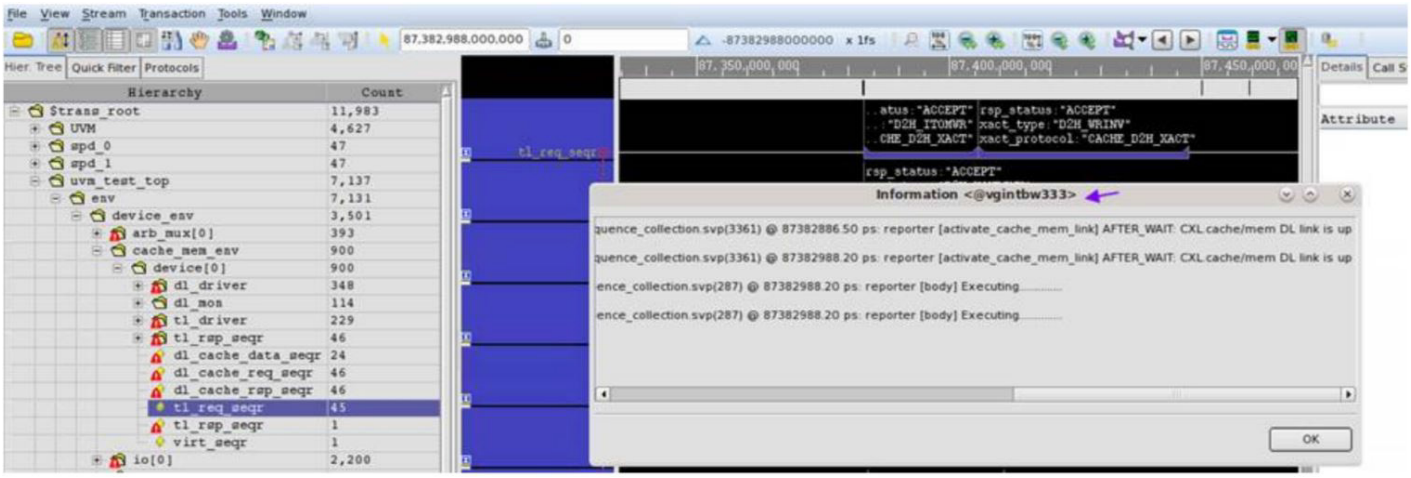
Figure 11-10 Log Details in Transaction Window



Example Illustrating the UVM INFO Messages in Transaction Window:

Log annotation on required time (Pressing a mouse button and drag on the annotation shows the actual log message as illustrated)

Figure 11-11 Example Illustrating UVM INFO Messages in Transaction Window



11.7 Enabling Auto Debug Support Using SVT_DEBUG_OPTS

The `SVT_DEBUG_OPTS` control can be used to automatically enable debug support for all the VIP instances included in the test bench. On enabling this feature, the `svt_debug.out` and `svt_debug.transcript` files are created. You need to send the `svt_debug.out` and `svt_debug.transcript` files to Synopsys support for further debug.

The `svt_debug.out` file contains:

- Information about the enabled debug features.
- Data about the environment that the VIPs are operating in .

The `svt_debug.transcript` file contains the `UVM_HIGH` or `UVM_FULL` verbosity messages from VIP instance.

Note This feature only captures VIP specific information, and does not capture any design information.

To enable the feature, pass the below plusargs to simulator command line:

```
+svt_debug_opts
```

Or

```
+svt_debug_opts=<options>
```

Table 11-1 Field Description

Field	Description
inst	Full instance path identifying the VIP instance to apply the debug automation features to. Regular expressions can be used to identify multiple VIP instances. If this value is not provided and if the **type** value is not defined, then the debug automation feature gets enabled on all SVT VIP instances.
type	Identifies an SVT class type (for example, `svt_cxl_agent`) to apply the debug automation features to. When this value is defined, the debug automation gets enabled for all instances of this class type.
start_time	Identifies when the debug verbosity settings need to be applied. The time must be defined in terms of the timescale that the VIP is compiled in. If this value is not provided, then the verbosity settings get applied at time zero.
end_time	Identifies when the debug verbosity settings need to be removed. The time must be defined in terms of the timescale that the VIP is compiled in. If this value is not provided, 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 defined, then the verbosity is set to **DEBUG/UVM_HIGH/OVM_HIGH** by default. When this feature is enabled, all the VIP instances enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

For example,

```
+svt_debug_opts="inst:uvm_test_top.env.host_env.cache_mem_env.host[0],verbosity:VERBOSE
"
```

This enables `UVM_FULL` verbose messaging from `svt_cxl_agent` instanced as host[0]. This also enables the transaction log creation for host[0] as `"uvm_test_top.env.host_env.cache_mem_env.host[0].tl_driver.xact_log"`.