cadence

Cadence CDN_APB VIP User Guide

Product Version 11.3

May 2013

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Table of Contents

List of Figures

List of Tables

Preface

The Cadence CDN_APB VIP User Guide describes the CDN_APB VIP based on the PureSpec™ API.

The CDN_APB VIP provides the solution for verifying compliance and compatibility of the protocol. It includes highly configurable and flexible simulation models of all the protocol layers, devices, and transaction types. It also supports integration and traffic generation in all popular verification environments. The CDN_APB VIP is built on top of the MMAV™ architecture to ensure high quality, high performance, and seamless EDA integration.

Using CDN_APB VIP, you can generate tests quickly and efficiently to ensure that your design under test (DUT) is compatible with the other components that may be used in the end system. The extensive protocol checks built into the CDN_APB VIP help you verify your design by generating warnings and errors when the protocol violations occur.

Note

The CDN_APB VIP is supported only when installed from a VIPCAT release.

1. About This Manual

This manual describes the CDN_APB VIP interfaces and configuration.

2. Terminology

Included below is a set of definitions of terms used throughout this document.

$CDN_VIP_ROOT	An environment variable that contains the path of the VIP installation.
Design Under Test (DUT)	This is the device being verified. Sometimes the term design under verification (DUV) is used.
Active Mode	In this mode, the CDN_APB VIP interface model acts as an actual device in your verification environment, both receiving and generating transactions.
Passive Mode	Connected directly to the DUT, the CDN_APB VIP model in this mode does not generate transactions, but verifies both incoming and outgoing transactions from the DUT.
SOMA	Specification of Model Architecture. SOMA is a Cadence-standard format to parameterize the model for user-specific verification needs.

Bus Functional Model (BFM)	Verification software that emulates a given device or protocol.
Data-Driven Verification API (DDV API)	The Data-Driven Verification API (DDV API) is an extension to the simulation environment offered using the Memory Model Portfolio and PureSpec Verification IP software. The DDV API can be used to integrate applications within the simulation process.
MMAV	Memory Modeler Advanced Verification is the industry-standard solution for memory simulation and system verification. MMAV dramatically enhances verification by enabling observations and operations on system-level data transactions during simulation. This "data-driven verification" approach is key to optimizing regressions and accelerating your overall verification process.

3. Customer Support

If you have a problem using this product or the documentation, you can submit a customer Support Request (SR) to Cadence Support. When you file a customer support request, you should provide as much detailed information as possible with regards to the problem you have encountered along with a tracefile of your simulation.

To obtain a tracefile of your simulation, you must run your simulation with the following configuration options set in your `.denailrc` file:

- `Historyfile denali.his`
- `Historydebug on`
- `Tracefile denali.trc`

Note

You can use different naming convention for the files, but keep the same file extension types.

3.1. Service Requests

SRs are your way of giving feedback, asking questions, getting solutions, and reporting problems. Unless told otherwise, Cadence support staff will respond to your service request. If Cadence Support cannot answer your question, Cadence research and development personnel will get involved. It is important to specify the severity level of the service request as accurately as possible. There are three levels of severity:

- Critical — You cannot proceed without a solution to the issue.
- Important — You can proceed, but you need a solution to the issue.

- Minor — You prefer to have a solution, but you can wait for it.

  Note

  You can request support to increase the severity level of an issue. Therefore, do not use Critical unless immediate resolution of an issue is absolutely necessary and urgently required.

3.2. Using Cadence Online Support

Cadence encourages you to submit requests using Cadence Online Support. With Cadence Online Support you can also track your open service requests.

To use Cadence Online Support to submit a service request:

1. If you do not yet have a Cadence Online Support account, go to http://support.cadence.com and click *Register Now* under the *New User* heading.

   You must provide a valid HostID for any VIP product. The `HostID` is contained in the `SERVER` line of your VPA product license file.

   Note

   If you already have a Cadence Online Support account, then you only need to update your Cadence Online Support preferences to include a valid `HostID` for a VIP product.

2. Log in to Cadence Online Support, and on the upper left side of the page click *Create Service Request* under the *My Service Requests* heading.

   A form is presented for submission of your service request. Select *Verification IP* in the *Product list* box and click *Continue*. Follow the online instructions to complete the Service Request.

3.2.1. Creating Group Privileges in Cadence Online Support

Sometimes it is beneficial to view the service requests of others on your project.

To create group privileges in Cadence Online Support:

1. Open a Cadence Online Support service request by clicking *Create Service Request* under the *My Service Requests* heading.

2. Select *Verification IP* in the *Product list* box and click *Continue*.

3. Fill in the required fields in the form presented.

   Explain in the *Stated Problem* text box that you want to create a group of users.

4. In the *People to notify* upon SR creation field, include the email addresses of the users you want to have group privileges.

Note

Each person receiving group privileges must have a Cadence Online Support account.

5. Click Submit SR to complete the Service Request.

Visit http://www.cadence.com/support/Pages/default.aspx to learn more about Cadence Global Customer Support and the support offerings we provide. For more details about our support process, visit http://www.cadence.com/support/Pages/support_process.aspx

4. Related Documentation

Besides the information in this document about the CDN_APB VIP, the following related information is available:

Table 1. Release-Related Documentation

To Find Out About ...	Look In ...
Release compatibility, installation, and configuration	The “Release Information” chapter of the *VIP Catalog Release Information* document, which is available from downloads.cadence.com, in your `$CDN_VIP_ROOT/doc` directory, or in Cadence Help.
What's new	The “What’s New in Verification IP” chapter of the *VIP Catalog Release Information* document, which is available from downloads.cadence.com, in your `$CDN_VIP_ROOT/doc` directory, or in Cadence Help.
Known limitations, problems, and solutions or fixes for them	The “Limitations and Workarounds” and “Known Problems and Solutions” chapters of the *VIP Catalog Release Information* document, which is available from downloads.cadence.com, in your `$CDN_VIP_ROOT/doc` directory, or in Cadence Help.
Fixed CCRs	The “Fixed CCRs” chapter of the *VIP Catalog Release Information* document, which is available from downloads.cadence.com, in your `$CDN_VIP_ROOT/doc` directory, or in Cadence Help.

To start Cadence Help, use the following command:

```
$CDN_VIP_ROOT/tools/bin/cdnshelp &
```

API Reference - Each release also includes API reference documents that are automatically generated with information about structs, fields, methods, and events. These documents can be very helpful

for debugging. One API reference is created each for UVM and OVM methodologies. To read these references, open the `index.html` files in the following locations with your Web browser:

- `$CDN_VIP_ROOT/doc/cdn_apb/apb_uvm_sv_ref/index.html`
- `$CDN_VIP_ROOT/doc/cdn_apb/apb_ovm_sv_ref/index.html`

Chapter 1. General VIP Concepts

The VIP is a full-timing, bus-functional model that supports many protocols. Using Cadence VIP you can quickly create a verification hierarchy that fits your design requirements.

You can perform comprehensive protocol checking by enabling or disabling the individual rules. The VIP issues callbacks to your testbench with error information when a protocol violation is detected. Model callbacks are also available at multiple points along the transaction flow through the model to aid scoreboarding or reactive testing.

Each protocol's functionality is configurable. The specification of model architecture (SOMA) configuration file describes the VIP modeling rules and properties of each device. You can use PureView, a graphical tool, to generate both the SOMA file and the instantiation interface for each device in your test configuration. The VIP architecture supports a wide range of commercial simulators and testbench platforms. You can use almost any commercial simulator, write your test cases in almost any language, and then use the VIP APIs to hook up your test cases. Callbacks allow you to take backdoor peeks into your test data as it flows through your test configuration, and to modify transaction at selected points during this flow. For example, you can introduce errors or discard the transaction. Finally, the model’s output includes various transaction history logs that show the flow of data transactions through your configuration.

1.1. VIP Verification Flow

The following diagram illustrates how to verify your design with the VIP:

Figure 1.1. The VIP Verification Flow

The VIP is typically utilized and linked in an event-driven or cycle-accurate logical simulation environment.

To perform VIP verification:

1. **Create a SOMA file using PureView**. A protocol model is characterized with a SOMA file. This SOMA file captures a complete device behavior, including device type, device capabilities, pin interface, and so on. Every component in the system/testbench hierarchy can be described by SOMA. You can create a SOMA file using PureView GUI tool. For details on how to use this tool, refer to Chapter 3, *PureView Graphical Tool*.

2. **Generate the instantiation interface using PureView**. Select the proper simulation HDL language format.

   Note

   Currently, only SystemVerilog is supported.

   Save the generated instantiation interface, which corresponds to the configuration and has the matching pin interface.

3. **Instantiate the generated interface in your testbench**. This entity becomes an instance in the simulation. Depending on the configuration, this instance can receive and generate traffic, or act like a monitor (only receiving traffic) on the DUT.

   You can find more details on how to instantiate the CDN_APB VIP model in the chapter Chapter 5, *CDN_APB VIP Model Operation*.

4. **Link the VIP (library) with the simulator**. The VIP is supplied in the form of a binary object file. By linking this library, the VIP instance communicates with simulator program through the API function calls. Note that these are in the same UNIX process.

5. **Simulate**. At the beginning of simulation time 0, the CDN_APB VIP model instance locates and loads the SOMA file for this configuration. As simulation time proceeds and signals change values, the model updates its state machine and takes an appropriate action based on the current state. The appropriate actions include transmitting transactions/symbols, reporting error conditions, and providing the testbench with event callbacks.

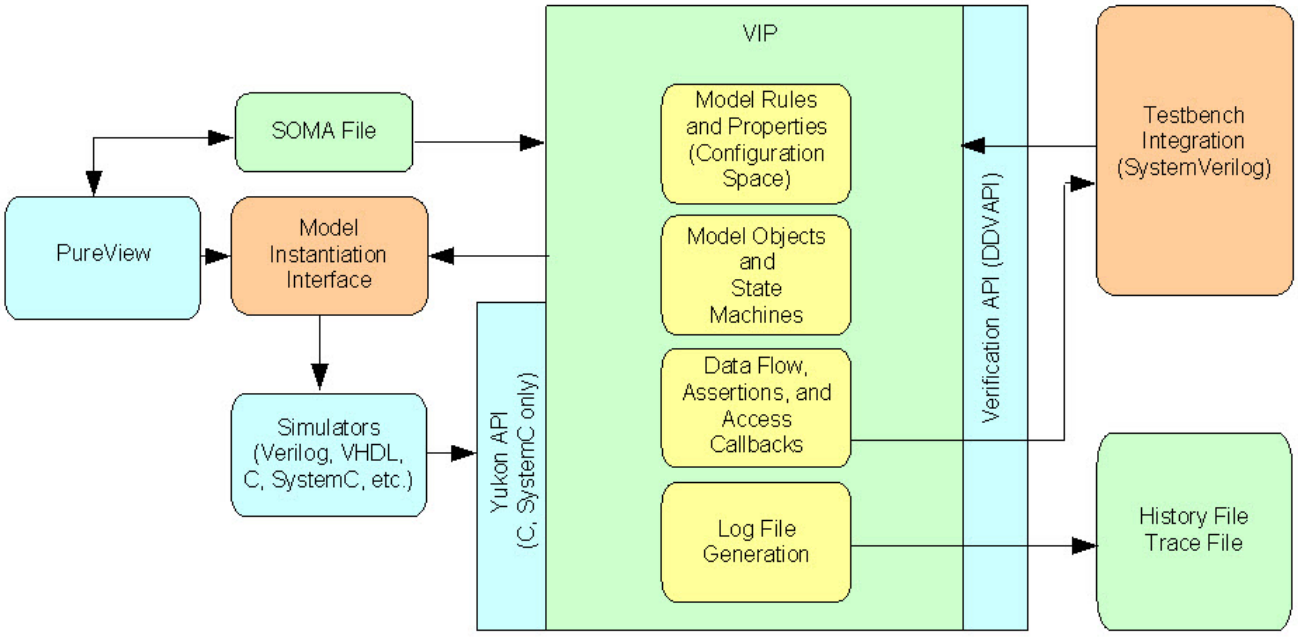
   Refer to Chapter 7, *CDN_APB VIP Simulation* for more details.

1.2. Architecture

The VIP is based on a hybrid C and **e** implementation, an architectural approach that combines the most comprehensive protocol checking with the highest performance. Additionally, the CDN_APB VIP model is highly configurable based on the SOMA file and run-time configuration controls.

The following figure illustrates the general architecture of the VIP.

Figure 1.2. The VIP Architecture



Note

Currently, the Yukon Interface (for C and SystemC) is not supported for the CDN_APB VIP.

The key blocks in the VIP architecture are described in the following sections.

1.2.1. PLI, VHPI, and so on

For Verilog simulators, a thin PLI version 1.0 is applied to establish the integration with the VIP. For VHDL simulators, either FLI, FMI, or VHPI is used for the integration.

1.2.2. SOMA File

Each device in your model has its own specification of modeling architecture (SOMA) parameter file describing its attributes. The SOMA file parameters are read at 0 simulation time.

1.2.3. Configuration Options via SOMA

The VIP model is highly configurable via the SOMA parameter file. In addition to supporting all the configurable options in the CDN_APB specifications, implementation-specific options and testability options are also covered.

1.2.4. Protocol Rules and Checks

One of the most important capabilities of the VIP is the extensive and complete checking of protocol rules. All aspects of the given protocol's specification are covered by the VIP checks. Hundreds of

run-time checks are built into the VIP and are tested during the simulation to catch any potential protocol violations.

1.2.5. Configuration Space

The VIP models the complete configuration space as:

- **Protocol Configuration Space** - The VIP defines all the protocol-specific configuration register spaces, which can be accessed via the testbench in addition to the protocol-specific access methods.
- **VIP Configuration Space** - Apart from the protocol-specific configuration register space, the VIP also defines a configuration space to allow for more granular testbench control of the VIP model.

For details, refer to Chapter 5, *CDN_APB VIP Model Operation*.

1.2.6. Model Objects and State Machines

The VIP models closely reflect the state machines and structures described in the protocol's specifications. Within each model, various queues and state machines are implemented, and can be manipulated or accessed through the testbench routines.

1.2.7. Callbacks

Within the VIP model, there are a number of key data structures, memories, and transaction activities. The accesses or events trigger a callback to the testbench code with the relevant information. You can choose your callback point from the list of all possible callback points.

1.2.8. Verification Simulation Interface

The VIP verification simulation interface is a key testbench interface for monitoring the CDN_APB VIP model state, controlling model behavior, and generating and modifying traffic. This is the DDVAPI that provides the backdoor interface to all testbench environments. This enables the CDN_APB VIP to be used in any sophisticated testbench environment or methodology and includes the ability to develop truly directed, constrained-random, or fully random self-checking test environments. This universal capability is available to any C, Verilog, VHDL, TCL, VERA, **e**, SystemC, or SystemVerilog environment.

1.3. Supported Operating Modes

The VIP model has the following main mode(s) of operation:

- Active Mode
- Passive Mode

In active mode, the model behaves like a component in the system. It can be used to either initiate or respond to the protocol traffic and to verify that the traffic from the design under test (DUT) conforms to the specification.

In passive mode, the model acts like a shadow reference model of the DUT and checks the DUT. However, it does not generate any traffic.

You must specify the operating mode separately for each model, using the PureView interface to generate the SOMA and HDL files. Every model in your configuration except for the monitor should be set to Active mode. The monitor, which is an exact copy of your DUT, is the only model that should be set to function in passive mode.

1.3.1. Active Mode

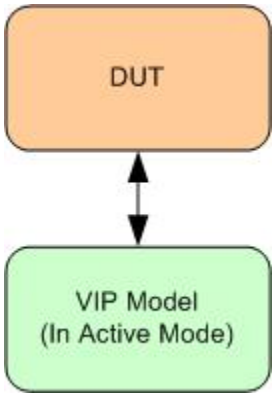
In active mode, the VIP model acts as a bus functional model (BFM) with protocol checking. The model characteristics are described in the SOMA specification.

The VIP verification functions can be used to generate transactions from the BFM to the DUT. The verification functions also provide backdoor access from the testbench for initializing and checking the active mode model register and memory spaces.

To use the VIP model in active mode, you need to have a SOMA file that describes the model and have the active mode parameter set. Connect the DUT pins to the appropriate VIP model connections.

For details on how to specify the model to work in the active mode using PureView, refer to Chapter 3, *PureView Graphical Tool*.

Figure 1.3. The VIP Model in Active Mode

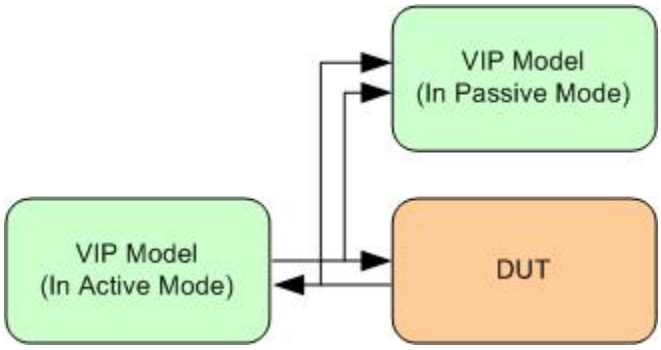


1.3.2. Passive Mode

The model in passive mode is a protocol checker enforced on the design under test (DUT). The monitor functions in a passive mode because it does not drive any data on the bus.

To specify that the model functions in passive mode, you need a SOMA file configured to run in passive mode. Connect the DUT pins to the appropriate VIP model connections. The SOMA configuration should exactly match the DUT so that it can monitor the DUT precisely. The VIP model in passive mode (monitor) records the inputs to the DUT, and verifies correctness by tracking the DUT's state, based upon what has been received, and checks that the DUT transmits appropriately for a given state.

Figure 1.4. The VIP Model in Passive Mode



Chapter 2. CDN_APB VIP Product Overview

2.1. Features

CDN_APB VIP enables you to ensure compliance to the CDN_APB specifications. You can test all possible configurations of CDN_APB devices, as well as monitor them using CDN_APB VIP. Furthermore, easy integration into a variety of design and verification tool flows gives you a test methodology that fits into with your development and testing cycle.

The key features of the CDN_APB VIP include:

- CDN_APB specification modeling that transfers two main device types:
  - Master -- issues read and write transfers
  - Slave -- responds to read/write transfers
- Model usage in either active mode or passive mode

  The passive mode provides the DUT shadow reference model that compares DUT responses to the expected responses.
- Supports for the CDN_APB AMBA3 configuration
- Supports for the CDN_APB AMBA4 configuration
- Customizable address and data width up to 32 bits
- Customizable error reporting

When the CDN_APB VIP model functions in an active mode (Bus Functional Mode), it interacts with complementary devices, and issues and responds to commands. When the CDN_APB VIP model functions in a passive mode (monitor), it attachés itself to the DUT's interfaces, listens to the incoming and outgoing traffic, and flags any deviation from the specification of the DUT. All checks are configurable and can be disabled. CDN_APB VIP provides powerful callback capabilities for the user-defined testbench to enable full control over all the transactions that take place within the CDN_APB model.

2.2. Model Types

There are two CDN_APB VIP models. You can use each of these device types either in an active mode (BFM) or passive mode (monitor).

2.2.1. Active Model of a Device with a Single Master Port

You can use the active model of a device with a single master port to drive high-level read and write transactions to a slave device under test (Decice Under Test)

- One slave Device under test (DUT)
- Mux that connected to several slaves devices

2.2.2. Active Model of a Device with a Single Slave Port

You can use the active model of a device with a single slave port to:

- Convert signals from a DUT master to high-level transactions
- Generate automatic responses
- Transmit these responses back to the master DUT

2.2.3. Passive Model of a Device with a Single Slave Port

You can instantiate the CDN_APB VIP passive model as a shadow model of the DUT. To accomplish this, prepare the SOMA file and fill the model's memory so that the model behaves like the DUT. All the pins of the CDN_APB VIP passive model are input pins. The slave monitor receives signals from the master driving DUT, generates the expected response, and compares the response with the actual response from the DUT that is received by the monitor through the wires. The monitor generates all the error messages of the CDN_APB VIP active model (transaction checking and timeouts).

In addition, the monitor generates errors about transaction mismatch with the expected transactions from the master-driven DUT, generates the expected response, and compares the response with the actual response from DUT received by the monitor through the wires.

Examples of such error messages include:

- Expected transactions that are missing
- Unexpected DUT slave responses

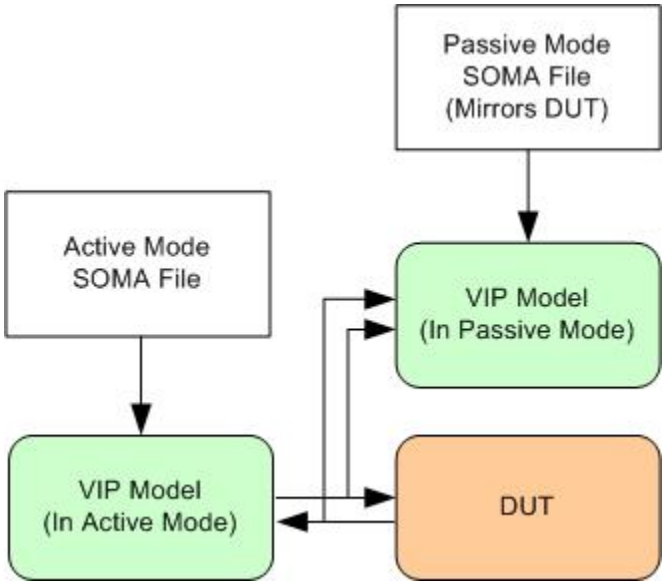
Chapter 3. PureView Graphical Tool

The protocol model is characterized with a Specification of Modeling Architecture (SOMA) file. This file captures a complete model behavior, including model type, model capabilities, pin interface, and so on.

The PureView graphical user interface enables you to configure a SOMA file and generate an instantiation interface file. You can use PureView to set most of the model's configurable features, such as the CDN_APB VIP model mode, interface type, and so on.

To use CDN_APB VIP, you need to generate two SOMA files. One for the passive mode model (this SOMA file mirrors your DUT's configuration) and one for the active mode model. The active mode model SOMA file should be configured to mirror an appropriate target to communicate with using the desired protocol. You can use PureView to create these two SOMA files.

Figure 3.1. SOMA Files for the VIP Model



To instantiate any CDN_APB VIP model, you must first configure a SOMA file and generate an instantiation interface using PureView.

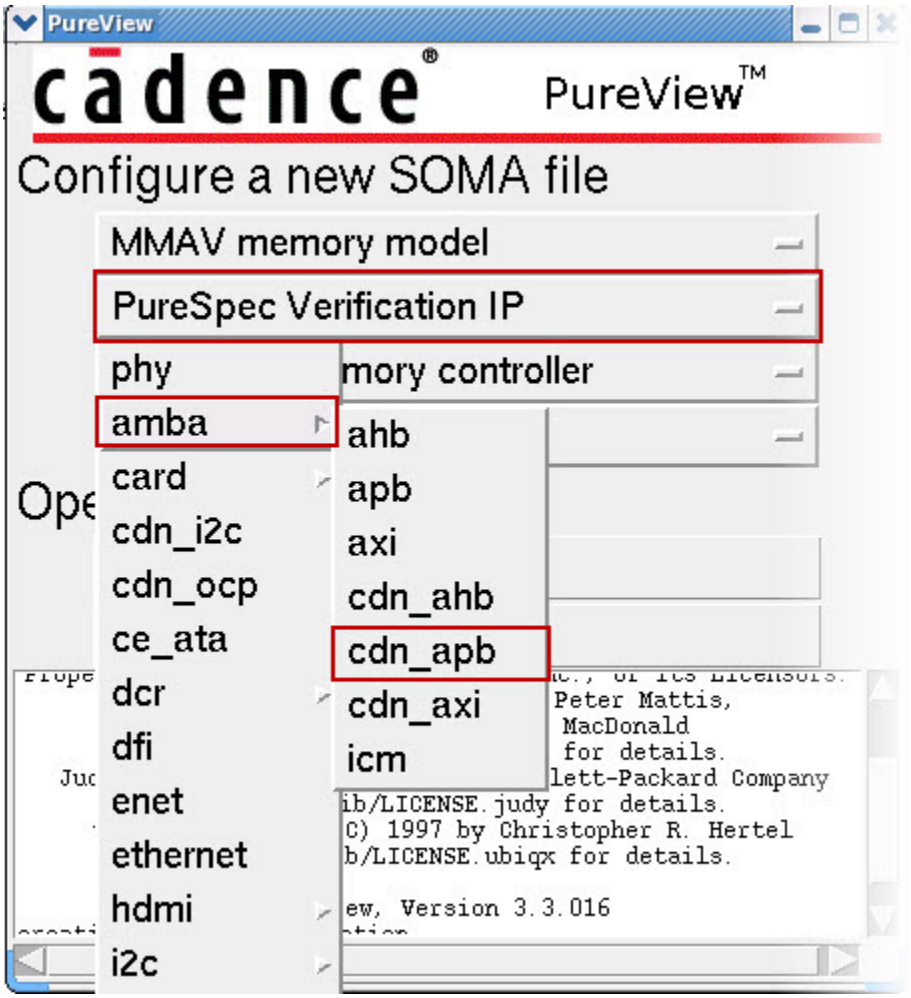
3.1. Creating a New SOMA File or Reconfiguring An Existing SOMA Setting

To create a new SOMA file or to reconfigure an existing SOMA setting:

1. Invoke PureView from your working directory:

   % pureview

Figure 3.2. PureView Opening Window



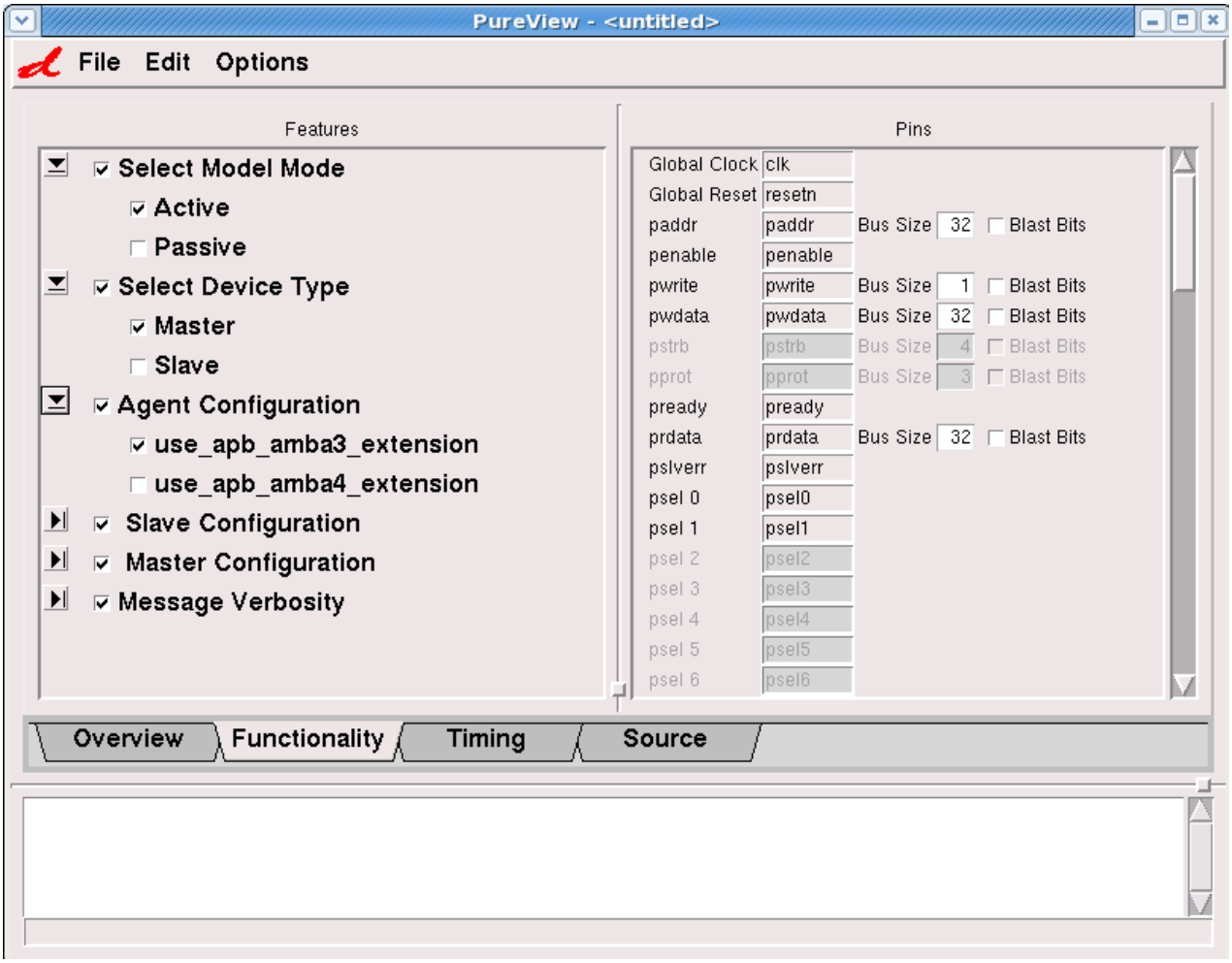
2. From the drop down list, either **Configure a new SOMA file** or **Open an existing file**.

3. Click on the **Functionality** tab. The following window appears.

Note

Here **Device Mode** refers to the *Active Mode* and **Monitor Mode** refers to the *Passive Mode*.

Figure 3.3. PureView - Functionality Tab



4. Select or de-select SOMA features as per your requirements.

5. Verify the SOMA file settings using **File > Check SOMA**.

6. Select **File > Save As** to save the SOMA file. You can save this file with an extension as either `.spc` or `.soma`.

When you have successfully saved the SOMA file, you need to create an instantiation interface for the device suitable for your particular simulation environment.

3.2. Creating an HDL Instantiation Interface

To create an instantiation interface:

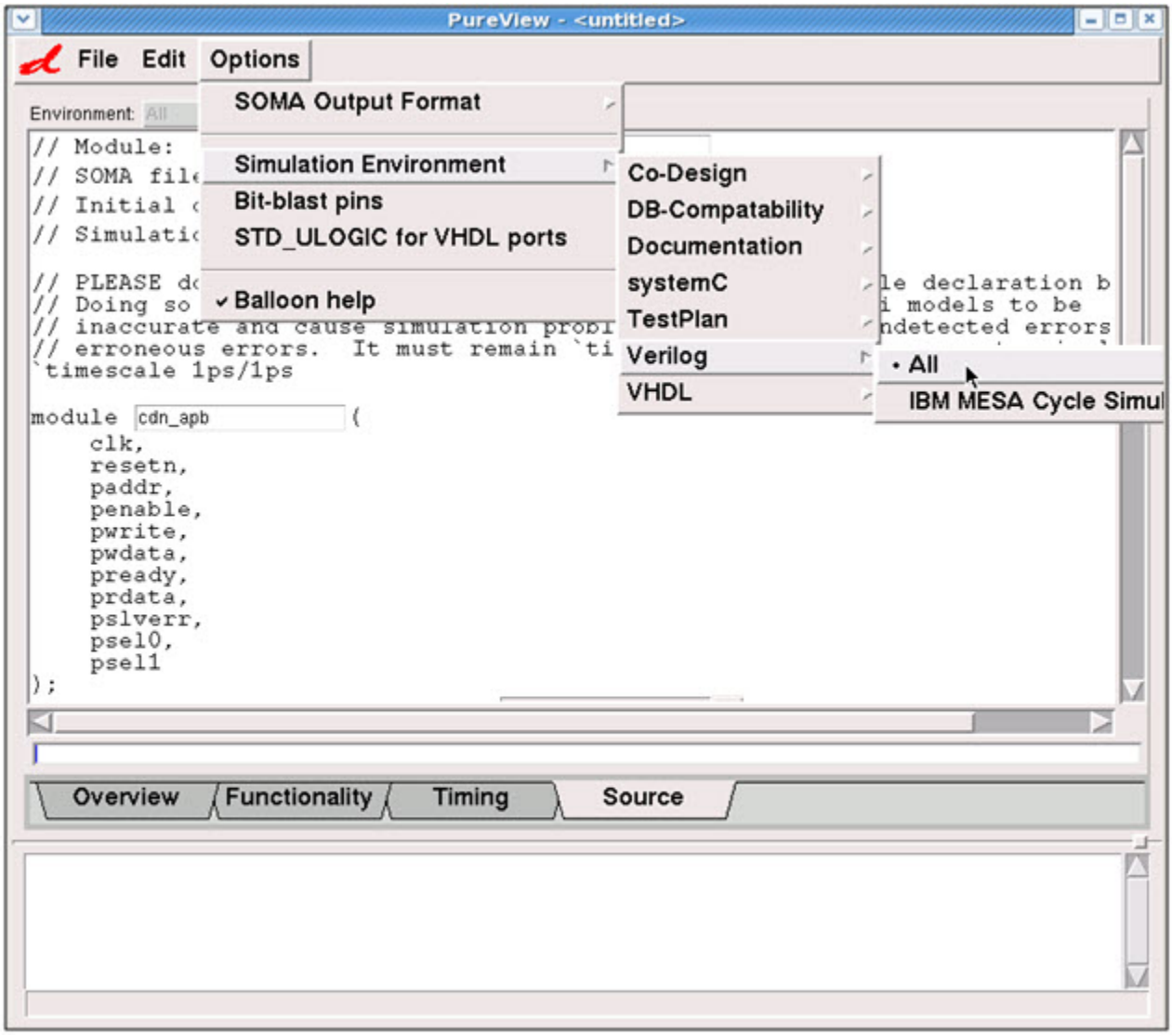
1. Select the **Source** tab in the PureView user interface.

   The **Source** window is initially blank.

2. Select **Options > Simulation Environment**.

   Though the user interface has three choices for HDL (**Verilog**, **VHDL**, or **SystemC**) **please note** that only **SystemVerilog** is currently supported. Then, proceed to select the actual simulator you want to use. The actual code for the HDL simulator appears in the **Source** window. Note that some of the fields can be edited. These fields have been preloaded with an appropriate data from the SOMA file you just saved, and do not need to be modified. If you choose to modify the location of the `.soma` file or the initial contents of SOMA file, select the ... Browse button next to these fields, at which point, the file selection window appears. Use this window to select an alternate file.

Figure 3.4. PureView - Source Tab



3. Select **File > Save Source As** to write the instantiation interface to a file so it can be instantiated into your design.

A file selection window appears in which you can save your file.

Chapter 4. CDN_APB VIP Model Configuration

The CDN_APB VIP model is configurable via the specification of modeling architecture (SOMA) parameter file and the `.denalirc` file. The SOMA parameters are used to configure individual models and the `.denalirc` parameters are used for the run-time options. In addition to the configurations implied by CDN_APB specifications, the CDN_APB VIP also provides implementation-specific and testability options.

4.1. .denalirc

The `.denalirc` file is a simple text file containing keyword-value pairs used to store your CDN_APB VIP model-specific settings.

A default `.denalirc` file is located in `$DENALI/.denalirc`. You can create up to four `.denalirc` files to control simulation or to set basic logging and simulation parameters. See Chapter 7, *CDN_APB VIP Simulation* and Section 7.2, *"Controlling Model Behavior"* for further information on simulation-specific settings and the locations of `.denalirc` files.

4.1.1. General .denalirc Options

The following section details the `.denalirc` parameters that are not protocol-specific.

Table 4.1. General .denalirc Parameters

Name	Values	Default
	Description	
Historyfile	Filename	None
	The filename where the simulation history information will be stored. You can specify the `-gzip` option if you want to gzip the files during the simulation run. By default, the model does not zip the files. `Historyfile -gzip denali.his.gz` When you create the history file using the `-gzip` option, Cadence recommends that you specify the filename with extension `.gz`. If you do not specify the `.gz` extension, the model still creates a legal zip file. However, you need to add the `.gz` extension later (using UNIX command `mv`) to unzip successfully using the *gunzip* utility. The operating system IO file size (2 GB) limitation also applies on the gzipped version.	
HistoryDebug	On/Off	Off
	Enables more detailed history messages.	
Tracefile	Filename	None

Name	Values	Default
	Description	
	Enables the Tracefile. You can specify the `-gzip` option if you want to gzip the files during the simulation run. By default, the model does not zip the files. `Tracefile -gzip denali.trc.gz` When you create the trace file using the `-gzip` option, Cadence recommends that you specify the filename with extension `.gz`. If you do not specify the filename extension, the model still creates a legal zip file. However, you need to add the `.gz` extension later (using UNIX command `mv`) to unzip successfully using the *gunzip*utility. The operating system IO file size (2 GB) limitation also applies on the gzipped version.	
HistoryInSimLog	0/1	0
	Redirects the history file to your simulation log versus the filename mentioned in the `Historyfile`	
HistoryPattern	Pattern string	All instances
	Adds only information about the specified `pattern` to the history file. This parameter is used to limit the amount of information dumped into the history file.	
TracePattern	Pattern string	All instances
	Adds only information about the specified `pattern` to the trace file. This parameter is used to limit the amount of information dumped into the trace file.	
ReportTimeUnits	Integer, us, ns, ps, or fs	Integer
	Displays values in the specified format in the history and trace files. By default, the time units are reported in the largest unit for which the numeric part will be an integer. For example, to specify the time units in picoseconds, use the following: `ReportTimeUnits ps`	
InitMessages	On/Off	On
	Turns on/off messages pertaining to the instances in the design.	
LicenseQueueTimeout	Time value	0
	Specifies the number of minutes to wait before exiting, if the license is not available.	
LicenseQueueRetryDelay	Time value	0
	Specifies the number of seconds to wait before pinging for available licenses.	
MaxHeapSize	Integer	0

Name	Values	Default
	Description	
	Some models control the maximum heap size that a simulation process can use. You can override the default heap size by setting the MaxHeapSize variable with a non-zero integer for the size (in MegaBytes). The value zero indicates that the default setting should be used. For 32_bits, the max size should not exceed 3072 (MB), which is the default.	
EnableCoverage	On/Off	An instance that is not covered by an enable will have the default behavior for that kind of model.
	Some models allow you to turn coverage on with a variable. You can use the EnableCoverage variable to override the default behavior of one or more instances. An instance that is not covered by an enable will have the default behavior for that kind of model. Patterns identify the set of instances that you are enabling. They use shell glob syntax (*,",[]). For example: *[Pp]assive* or top.Active*. Here * specifies all instances.	
DisableCoverage	On/Off	An instance that is not covered by a disable will have the default behavior for that kind of model.
	Some models allow you to turn coverage off with a variable. You can use the DisableCoverage variable to override the default behavior of one or more instances. An instance that is not covered by a disable will have the default behavior for that kind of model. Patterns identify the set of instances that you are disabling. They use shell glob syntax (*,",[]). For example: *[Pp]assive* or top.Active*. Here * specifies all instances.	
EnableProtocolChecks	On/Off	An instance that is not covered by an enable will have the default behavior for that kind of model.
	Some models allow you to turn protocol checks on with a variable. You can use EnableProtocolChecks to override the default behavior of one or more instances. An instance that is not covered by an enable will have the default behavior for that kind of model. Patterns identify the set of instances that you are enabling. They use shell glob syntax (*,",[]). For example: *[Pp]assive* or top.Active*. Here * specifies all instances.	
DisableProtocolChecks	On/Off	An instance that is not covered by a disable will have the default behavior for that kind of model.

Name	Values	Default
	Description	
	Some models allow you to turn protocol checks off with a variable. You can use DisableProtocolChecks to override the default behavior of one or more instances. An instance that is not covered by a disable will have the default behavior for that kind of model. Patterns identify the set of instances that you are disabling. They use shell glob syntax (*,'',[]). For example: *[Pp]assive* or top.Active*. Here * specifies all instances.	

Note

The CDN_APB VIP model does not support save or restore features during simulation.

4.2. SOMA

The CDN_APB VIP model has its own SOMA parameter file that describes its attributes.

4.2.1. Details / Table

The following table lists the SOMA parameters available to configure the CDN_APB VIP model. You can change most of the parameters during the runtime by using the CDN_APB VIP registers.

Table 4.2. SOMA Features

Name	Description
`DeviceMode`	Active The model models an actual device. The default value is `1`
`MonitorMode`	Passive The model models a passive device, monitoring The default value is `0`
`AcceleratedMode`	Accelerated VIP (AVIP) The model can run in simulation or emulation The default value is `0`
`Master`	Master Port can drive PADDR, PWRITE, PWDATA and receive PWDATA and PSLVERR from the slave. The default value is `1`
`Slave`	Slave

Name	Description
	Port can receive and process PADDR, PWRITE, PWDATA and issue WDATA and PSLVERR. The default value is `0`
`cdn_apb_agent_config`	Agent Configuration The default value is `1`
`use_apb_amba3_extension`	use_apb_amba3_extension The default value is `1`
`use_apb_amba4_extension`	use_apb_amba4_extension The default value is `0`
`endianity`	endianity The default value is `0`
`endianity_LITTLE`	LITTLE The default value is `1`
`endianity_BIG`	BIG The default value is `0`
`cdn_apb_slave_config`	Slave Configuration = slave configuration unit. = The default value is `1`
`use_memory`	use_memory Indicates if the user uses APB EVC memory implementation. The default value is `1`
`cdn_apb_master_config`	Master Configuration = master configuration unit. = The default value is `1`
`no_master_slaves_mux`	no_master_slaves_mux This flag indicates if the environment have master to slaves MUX. when FALSE - The MUX is an outside component. For this mode, the master has one set of PSLVERR, PREADY, and PRDATA signals which are connected to the outside MUX. when TRUE - The MUX is placed inside a master module. For this mode, the master has PSLVERR,

Name	Description
	PREADY, and PRDATA signals for each slave separately. Default : FALSE. The default value is `0`
`number_of_slaves`	number_of_slaves The default value is `2`

4.2.2. SOMA Configurability vs. Model Register Space

The following table lists the SOMA parameters that can be modified at the runtime using the CDN_APB VIP registers. The table shows the SOMA parameter name and its corresponding runtime counterpart register.

Table 4.3. SOMA Parameters vs. Register Names

SOMA Parameter Name	Register Name
use_memory	DENALI_CDN_APB_REG_UseMemory
number_of_slaves	DENALI_CDN_APB_REG_NumberOfSlaves

4.3. Registers

You can use the model registers to dynamically control the simulation by writing appropriate values to them. The following table shows a complete list of the registers that are available in the CDN_APB VIP model.

Table 4.4. Summary of Registers

Register	Description
`DENALI_CDN_APB_REG_ErrCtrl`	Control over behavior when error
`DENALI_CDN_APB_REG_AvipBatchMode`	Only for accelerated VIP (AVIP). Configures the AVIP to Batching (TRUE) or Reactive (FALSE) mode. Default value - FALSE
`DENALI_CDN_APB_REG_AvipFlushBuffers`	Only for accelerated VIP (AVIP) in Batching mode. The register is set by user to enable flushing AVIP buffers in Batching mode. The register is reset by AVIP when flush buffers is ended.
`DENALI_CDN_APB_REG_AvipWaitClks`	Only for accelerated VIP (AVIP). This registers is set by the user to start a timer in HW to count configured numbers of clocks. The register is reset by AVIP when the timer expires
`DENALI_CDN_APB_REG_DataWidth`	
`DENALI_CDN_APB_REG_AddrWidth`	

Register	Description
DENALI_CDN_APB_REG_HasTrRecording	When this field is true, the agent record transactions
DENALI_CDN_APB_REG_TransferTimeout	This field set the number of clock cycles the VIP waits before triggering ERR_CDN_APB018_REQ_TIMEOUT or ERR_CDN_APB103_REQ_TIMEOUT (Default: 100 cycles)
DENALI_CDN_APB_REG_SegmentStartAddress	Start address of slave memory segment.
DENALI_CDN_APB_REG_SegmentEndAddress	End address of slave memory segment.
DENALI_CDN_APB_REG_SegmentPselIndex	The slave index which the address segment is mapped to. (the PSELx signal index that expected to be raised when accessing this segment)
DENALI_CDN_APB_REG_ErrorID	Error ID enum value to which the severity will be set. This register was deprecated. Please use the new Err Ctrl register.
DENALI_CDN_APB_REG_ErrorSeverity	Set Severity to the Error ID enum value in the ErrorID register. This register was deprecated. Please use the new ErrCtrl register.
DENALI_CDN_APB_REG_ResetSignalsSimStart	Module to reset signals at simulation start.
DENALI_CDN_APB_REG_RetrieveItem	Available when using ICM - Is activated when user want to retrieve an item associated with ID
DENALI_CDN_APB_REG_ModuleId	Available when using ICM - Returns a unique identifier of the instance in the tb to be used when connecting to the ICM.
DENALI_CDN_APB_REG_UseMemory	Indicates if the user uses APB EVC memory implementation.
DENALI_CDN_APB_REG_NumberOfSlaves	
DENALI_CDN_APB_REG_Verbosity	

Chapter 5. CDN_APB VIP Model Operation

5.1. Model Instances

CDN_APB VIP saves configuration information in various memory spaces associated with each instance. When the model loads in the simulator, the CDN_APB VIP model creates several internal memories for each CDN_APB VIP instance. These internal memories hold the model, configuration, and registers.

5.1.1. Instance ID

CDN_APB VIP has a simple interface to access model, model memory space, model control registers, and transactions. Each of these objects has an instance ID associated with it. Before issuing the transactions or setting up callbacks, write code to instantiate these models in your testbench.

The following memory models are associated with each instance:

- Registers -- Used to handle the instance register space.

To initiate all the memory models in SystemVerilog, refer to the following example:

```
// A denaliCdn_apbInstance should be instantiated per agent
// You should extend it and create your own, to implement callbacks
class cdn_apbInstance extends denaliCdn_apbInstance;

    denaliMemInstance regInst ; // Handle to the register-space
       function new(string instName);
        super.new(instName);
        regInst = new( { instName, "(registers)" } );
            endfunction

    ...
endclass
```

5.1.2. Model's Control Registers and Storage Memory

CDN_APB VIP includes several memories including port-specific model control register arrays and the storage memory. You can access these by reading, writing, and setting up callbacks, as described below.

5.1.2.1. Accessing Control Registers

You can access control registers using the `regWrite()` function. This function has two parameters:

- register address
- data

To change the value of the `DENALI_CDN_APB_REG_UseMemory` register, refer to the following example:

```
    initial
    begin
```

```
      activeMaster.regWrite(DENALI_CDN_APB_REG_UseMemory, 0)
```

After this call, the slave will not store the data from the Write transaction and will always return random data on Read transactions. By Default, the value of `use_memory` is read from the SOMA file.

To use register addresses in SystemVerilog, your testbench module must include `the denaliCdn_apbType.sv` file supplied with the CDN_APB VIP model. You can find this file under `$DENALI/ddvapi/sv`.

5.1.2.2. Backdoor Reads and Writes to the Slave's Memory

You can perform backdoor reads and writes to the slave's memory space by using `memRead()` and `memWrite()` functions.

To define the memory space and perform a backdoor write, refer to the following example:

```
class apbInstance extends denaliCdn_apbInstance;

    denaliMemInstance regInst ; // Handle to the register-space
    denaliMemInstance memInst ; // Handle to the main memory-space

    function new(string instName);
        super.new(instName);
        regInst = new( { instName, "(registers)" } );
        memInst = new( { instName, "(memory)" } );
    endfunction
    ...
    ...
    //
    // Raad one byte from main memory
    //
    virtual function reg [7:0] memRead( reg [31:0] addr);
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        void'( memInst.read( trans ) );
        memRead[7:0] = trans.Data[0];
    endfunction

    //
    // Write one byte to main memory
    //
    virtual function void memWrite( reg [31:0] addr, reg [7:0] data );
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        trans.Data = new[1];
        trans.Data[0] = data[7:0];
        void'( memInst.write( trans ) );
    endfunction
endclass
```

To perform backdoor writing to the slave's main memory at address 0x1000, do the following:

```
 Initial
        begin
            activeSlave.memWrite('h1000,8b'10101010);
```

5.1.2.3. Defining Memory Segments

The register model lets you define different address segments related to the different slaves ports.

```
    // Map address segments to slave ports.
```

```
   // According to this mapping the master will know which PSELx signal to assert for each address
   //
   // from_address - the start address of the segment
   // to_address - the start address of the segment
   // slave_port_index - slave port index
   function void mapAddressSegment(reg [31:0] from_address, reg [31:0] to_address, reg[3:0]
slave_port_index);
        regWrite(DENALI_CDN_APB_REG_SegmentStartAddress, from_address);
        regWrite(DENALI_CDN_APB_REG_SegmentEndAddress, to_address);
        regWrite(DENALI_CDN_APB_REG_SegmentPselIndex, slave_port_index);
   endfunction
```

The following example shows how to define two address segments:

- The range [0..0xFFFFFFF] to be connected to slave port 0
- The range [0x10000000..0x1FFFFFFF] to be connected to slave port 1

```
activeMaster.mapAddressSegment('h0,'h0fffffff, 0);
activeMaster.mapAddressSegment('h10000000,'h1fffffff, 1);
```

5.2. Instantiation / Elaboration

To instantiate any CDN_APB VIP model, you must first configure a SOMA file and generate a Verilog instantiation interface through the PureView user interface. For detailed information on the configuration parameters of the CDN_APB VIP SOMA files, refer to Section 4.2, “*SOMA*”.

The following example uses the SystemVerilog code to show how an APB3 model is instantiated:

```
cdn_apb_master M0(PCLK,
                  PRESETN,
                  PADDR,
                  PENABLE,
                  PWRITE,
                  PWDATA,
                  PREADY,
                  PRDATA,
                  PSLVERR,
                  PSEL0,
                  PSEL1,
```

The module `cdn_apb_master` is defined in the Verilog instantiation interface code, which is automatically generated from a SOMA file. For details, refer to Section 4.2, “*SOMA*”. The parameter `interface_soma` must point to the SOMA file path.

5.3. Transactions

After completing the setup, you can start generating transactions and test various verification scenarios. This section gives a brief overview of how the CDN_APB VIP represents transactions, followed by a detailed description of the interface you can use to generate transactions.

5.3.1. Transaction Generation

To send a transaction, instantiate the module `denaliCdn_apbTransaction`. You can assign module fields and call task `transAdd()` located in the `denaliCdn_apbInstance` module under the `$DENALI/ddvapi/sv/denaliCdn_apb.sv` file.

Note

Cadence recommends that not to use the `denaliCdn_apbInstance` module as it is and extending it and adding DUT specific constraints.

Following is an example:

```
class myTransaction extends denaliCdn_apbTransaction;
                constraint user_dut_information {
                         NumberOfSlaves == 3;
                         Addr < 32'h1fffffff;
                         ...
                }
endclass
```

Refer to the following example to send a read transaction, where some of the fields are constrained:

```
import DenaliSvCdn_apb::*;
import DenaliSvMem::*;
module testbench;
...
cdn_apbInstance activeMaster;
myTransaction trans;
initial
begin
activeMaster = new ("testbench.master");
trans = new();
trans.Direction = DENALI_CDN_APB_DIRECTION_READ;
trans.Addr = 'h100;
status = activeMaster.transAdd(trans,0);
```

You can also randomize all fields in SystemVerilog using the `randomize()` task, as shown below:

```
assert( trans.randomize() with {
Direction = DENALI_CDN_APB_DIRECTION_WRITE;
Data == 'h7;
});
status = activeMaster.transAdd(trans,0);
```

5.3.1.1. CDN_APB4 Transaction Generation

CDN_APB4 VIP provides two additional signals: write strobe and protection.

When you are running the CDN_APB4 simulation, you can constrain the strobe value, when sending a WRITE transfer, as shown in the following example:

```
assert( trans.randomize() with {
      Direction == DENALI_CDN_APB_DIRECTION_WRITE;
      Addr == 'h200;
      Data == 'h1234;
      Strobe == 4'b1001;
});

status = activeMaster.transAdd(trans,0);
```

When you are running the CDN_APB4 simulation, you can constrain the value of PPROT signal as shown in the following example:

```
assert( trans.randomize() with {
    Privileged == DENALI_CDN_APB_PRIVILEGEDMODE_NORMAL;  // PPROT[0] == 0
    Secure == DENALI_CDN_APB_SECUREMODE_NONSECURE;  // PPROT[1] == 1
    DataInstr == DENALI_CDN_APB_FETCHKIND_DATA;   // PPROT[2] == 0
});

status = activeMaster.transAdd(trans,0);
```

5.3.2. Transaction Flow

The master and slave maintain a scoreboard of outstanding transactions with various associated data about their timing, ordering, and so forth.

You can modify various transaction's data during the `BeforeSend` callback.

5.3.2.1. Master – Sending Read or Write Transfer

The basic transaction flow steps include:

1. The master gets a high-level transaction from the user queue.
2. Callback `BeforeSend`

   During this callback, you can modify the transaction's fields including `Direction, Addr, ReqDelay` and `Data` (in case of Write transfer).
3. Callback `TransferStarted.`
4. Callback `MonTransferEnded` (passive master) or `DriverTransferEnded` (active master).

5.3.2.2. Slave – Receiving Transfer and Sending Transfer response

The basic transaction flow steps include:

1. The slave gets the `Read or Write` transfer.
2. Callback `BeforeSend` where you can control `Slverr, ReqResponseDelay` and `Data` (in case of Read Transfer).
3. Callback `TransferStarted`.
4. Callback `MonTransferEnded` (passive slave) or `DriverTransferEnded` (active slave).

5.3.3. Transaction Types and Fields

Table 5.1. Transaction Types (DENALI_CDN_APB_TR_)

Transaction Type	Description
UNSET	Undefined type - this value should not be used
Transfer	Basic transfer struct sent by the master

Transaction Type	Description
Response	Slave response for a transfer

Table 5.2. Fields Common to All Transaction Types

Name	R/W	Type	Size (Bits)	Description
Type	RW	denaliCdn_apbTrTypeT	32	Transaction type.
PortNum	RO	unsigned	32	Port/Queue number
Callback	RO	denaliCdn_apbCbPointT	32	ID of the callback currently associated with the transaction. Useful in the callback processing user code.
ErrorString	RO	string	*	Character string describing the transaction errors for the debug
ErrorId	RO	denaliCdn_apbErrorTypeT	32	This is the Error ID associated with error callbacks
ErrorInfo	RO	denaliCdn_apbErrorInfoT	32	Transaction error severity
UserData	RW	unsigned	32	Placeholder for the user-specific information
Delay	RW	unsigned	32	Transaction delay (in clock cycles). Please do not use this field
ReqDelay	RW	unsigned	32	The delay until driving the req channel
Direction	RW	denaliCdn_apbDirectionT	32	The Direction of the transfer. either READ or WRITE
Addr	RW	unsigned	32	The address of the transfer
Data	RW	unsigned	32	The data of the transfer
DataXMask	RW	unsigned	32	Bit mask for the transfer's data monitored on the bus, where '1' indicates 'X' value
DataZMask	RW	unsigned	32	Bit mask for the transfer's data monitored on the bus, where '1' indicates 'Z' value
Slverr	RW	boolean	32	The Slave response for the transfer (AMBA3 onward)
Strobe	RW	unsigned	4	The strobe for the write data (AMBA4)
Privileged	RW	denaliCdn_apbPrivilegedModeT	32	AMBA4 PPROT[0] - NORMAL or PRIVILEGED

Name	R/W	Type	Size (Bits)	Description
Secure	RW	denaliCdn_apb SecureModeT	32	AMBA4 PPROT[1] - SECURE or NONSECURE
DataInstr	RW	denaliCdn_apb FetchKindT	32	AMBA4 PPROT[2] - DATA or INSTRUCTION
IsStarted	RW	boolean	32	Was the transfer started.
IsEnded	RW	boolean	32	Was the transfer ended.
Duration	RW	unsigned	64	Transfer duration
ModelGeneration	RW	boolean	32	if TRUE - use the core model's constraints for generation. if FALSE - use only SV constraints for generation.
ReqResponseDelay	RW	unsigned	32	The delay until driving the req channel
StartTime	RW	unsigned	64	Time stamp for beginning of send/receive
EndTime	RW	unsigned	64	Time stamp for end of send/receive
TOTAL	RW	unsigned	32	

5.3.4. Enumeration Types

Chapter 6. CDN_APB VIP Callbacks

The callback mechanism enables you to monitor the incoming transactions. Every time some significant event happens, the CDN_APB VIP model issues a callback so that you can see the incoming transactions and modify the transactions that are automatically generated by the model.

6.1. Callback Types

The following table shows a list of all callbacks provided by the CDN_APB VIP pertaining to operations on transactions.

Table 6.1. Callback Types (DENALI_CDN_APB_CB_)

Callback Type	Description
UNSET	Undefined type - this value should not be used
Error	A generic error has been detected. During this callback, Transaction field ErrorString stores character string that is a concatenation of all error messages associated with the transaction at the moment, field ErrorId holds the current error id, and field ErrorInfo holds the error severity.
ResetStarted	Callback ResetStarted is activated when the protocol ARESETn signal is de-asserted.
ResetEnded	Callback ResetEnded is activated when the protocol ARESETn signal is asserted.
RetrieveItem	Available when using ICM - Callback RetreiveItem is activated when user write to the RetriveItem register with the id of the wanted item
TransferStarted	Callback TransferStarted is called from the MONITOR when a new Transfer starts
BeforeSend	Callback BeforeSend is called from the driver right before a new transfer starts. During BeforeSend the user can modify the transaction.
MonTransferEnded	Callback MonTransferEnded is called from the MONITOR when a new Transfer ends
DriverTransferEnded	Callback DriverTransferEnded is called from the Driver when a new Transfer ends
CoverageSample	An internal callback for coverage collection this callback is only for internal use.
TOTAL	This is not a real queue, but a symbolic constant useful to enumerate queue in a loop

6.2. Transaction Callbacks

You can set these callbacks at specific points in the flow of a transaction through the system. You can use the callbacks to intercept a transaction, change its characteristics, and re-insert it at the same point so that the transaction resumes its previous flow. You can also use callbacks as a means to know that certain events have occurred in the model, without the use of polling.

You can use data transaction flow callbacks to:

- Change a field in a transaction.
- Synchronize user transactions from the CDN_APB VIP model with transaction received from the DUT.
- Implement scoreboards to enable the testbench to compare responses from the DUT with expected data.
- Inject errors into transactions after these have been generated by the CDN_APB VIP, but before the transaction is transmitted on the physical link.

Within a test configuration, there may be many possible callback points. You must specifically enable a callback point in order to make use of it in your test cases.

6.2.1. Transaction Callback Interface

The CDN_APB VIP transaction callback interface provides callback initialization and handling. The `denaliCdn_apbCallback` base module contains an integer variable that is changed by the model when a sensitized event occurs in the model. A change on this variable triggers a procedural block in the testbench to process the callback events.

Each pre-defined callback `DENALI_CDN_APB_CB_<callback_name>` triggers a pre-defined function. For example:

```
virtual function int  <callback_name>CbF(ref denaliCdn_apbTransaction trans) ;
```

You can add procedural code to this function in the testbench to process the user-defined behavior. Each pre-defined callback triggers the default method as shown below:

```
virtual function int DefaultCbF(ref denaliCdn_apbTransaction trans);
```

You can write general code (for example, message print) that can be performed on all callback types.

6.2.2. Instantiating Transaction Callbacks

The pre-defined callbacks are specified in the `$DENALI/ddvapi/sv/denaliCdn_apbTypes.svh` file.

6.2.3. Enabling Transaction Callbacks

You can use the predefined callback type IDs in SystemVerilog by including the `denaliCdn_apbTypes.svh` file in your testbench module. The `denaliCdn_apbTypes.svh` file is supplied with the CDN_APB VIP model. It is located at `$DENALI/ddvapi/sv`.

6.2.3.1. setCallback()

You can set up a model callback using the `setCallback()` function as shown below:

```
cdn_apbMaster.setCallback(DENALI_CDN_APB_CB_Ended);
```

Here, `cdn_apbMaster` is one of the initialized models of type `denaliCdn_apbInstance` and `DENALI_CDN_APB_CB_Ended` is one of the predefined callbacks of type `denaliCdn_apbCbPointT`.

6.2.3.2. clearCallback()

Callbacks that have been enabled with `setCallback()` can be disabled with the `clearCallback()` function as shown below:

```
cdn_apbMaster.clearCallback(DENALI_CDN_APB_CB_Ended);
```

6.2.4. Processing Transaction Callbacks

The variable `Event` toggles when any enabled callback triggers. There is only a single event for each instance in the design hierarchy even if multiple callbacks are enabled. These functions are defined (for SystemVerilog) in `$DENALI/ddvapi/sv/denaliCdn_apb.sv`.

To process transaction callbacks, you can use the following functions:

6.2.4.1. reasonStr()

This function displays a text string for the reason for a callback.

Chapter 7. CDN_APB VIP Simulation

7.1. Running the Model

This section describes the main steps to run the VIP model.

7.1.1. Setting Basic Logging and Simulation Parameters

Before running your simulation, ensure that you have appropriately directed the output to the desired log file locations. With the exception of the simulation time parameter, the following parameters are all in the `.denalirc` file. For details on the `.denalirc` parameters, refer to Section 4.1.1, *"General .denalirc Options"*.

- Set the history file location using the `HistoryFile` parameter.
- Set the history debugging level using the `HistoryDebug` parameter. Setting this to "On" copies history file information into your trace file for easy debugging.
- Set the trace file location using the `TraceFile` parameter.
- Set the simulation time parameter as described in Section 7.1.4, *"Ending Simulation"*.

7.1.2. Linking the CDN_APB VIP Library and Running the Simulation

The CDN_APB VIP product is supplied in the form of a binary object file. By linking in this library, the CDN_APB VIP instance communicates with the simulator program through the API function calls. Notice these are in the same UNIX process.

Refer to Chapter 8, *CDN_APB VIP Testbench Integration* for more details.

After the simulation completes, the history and trace files are created.

7.1.3. Viewing Results

After completing the simulation run, the CDN_APB VIP creates the following files (subject to the corresponding settings in the `.denalirc` file):

History File — The history file is a very useful resource that is generated during simulation. With this file, you can view the details of the CDN_APB VIP model at a given simulation time, which aids in debugging the design under test.

Trace File — The trace file contains all events the model receives, such as reads, writes, and so on. This is primarily used by *Cadence Customer Support* as a valuable diagnostic tool for understanding and recreating your simulation environment.

7.1.4. Ending Simulation

There are two primary ways to determine that the simulation has completed. The simulation can end after either one of these conditions:

- A specified duration of simulation time has elapsed.
- A specified number of data transaction have been processed.

  If you use this method, you can enable callbacks to look for the ID of the last transaction sent, and end the simulation when that transaction ID is detected.

You can also add a timeout to prevent your simulation from hanging indefinitely.

You can also set a time limit for your simulation in your testbench instantiation file, as shown in the example below:

```
initial begin
for(i=0;i< `SIMTIME; i=i+1)
    #1000 ;
    $finish;
end
```

The above code allows you to run the simulation for a set amount of time as defined by the variable `SIMTIME`. You can set this variable via the following command line when you run your simulation:

```
`define SIMTIME <x>
```

Here, <x> is a valid number such as 5000.

7.2. Controlling Model Behavior

You can control the model behavior using the run-time initialization file `.denalirc`. This file stores settings that control the CDN_APB VIP model behavior during simulation.

The `.denalirc` file is a text file containing keyword-value pairs used to store your initialization settings. All lines beginning with # are comments and are ignored. The default `.denalirc` file in your installation also includes commented lines that provide a complete description of all the switches and their usage. The descriptions use mixed-case for clarity but the flags are NOT case-sensitive, even though the values might be.

Note

The name of the `.denalirc` file cannot be changed.

7.2.1. The .denalirc File Hierarchy

Before running the simulation for the first time, the CDN_APB VIP needs the `.denalirc` to control the model behavior during simulation. You can use up to four `.denalirc` files to store these settings. These files are listed below in the order of precedence:

$DENALIRC	*Environment variable*
`.denalirc`	*Simulation specific defaults*
`.denalirc`	*User Defaults*

`$DENALI/.denalirc` *System Defaults*

If the `$DENALIRC` environment variable is set, and a file exists in the specified path, that file is used, and the others are ignored. If `$DENALIRC` is not set (or if the file is not found at the specified location), then the simulator looks in the current directory for `.denalirc`.

For example, if you want to change only one setting for a particular simulation, create a `.denalirc` file in the working directory to store the specific simulation settings.

7.3. Output Files

You can enable the CDN_APB VIP to generate different types of output log files by setting different options in the `.denalirc` file. These log files are helpful during the debug process. The following sections describe these output files and some of the aspects of each log file that can be used during the debug process.

7.3.1. History File

The history file allows you to view the details of the CDN_APB VIP model at a given simulation time, which aids you in debugging your design. The history file includes the following information:

- Memory/register read and write along with address, values, and simulation time
- Model state transition along with the simulation time
- Model activities during different states of state machines
- Data flow at different callback points
- Data transmitted and received on different pins
- Order-of-model activities with respect to the simulation time

The history file contains all messages, so the size of this file can be significant depending on the test case intent.

Every line of the file represents an event. The syntax is as follows:

```
<instance_name> <time> <debug_flag> <action> <information>
```

where:

- `<instance_name>` is an instance, such as an endpoint device
- `<time>` is the time when the event occurred
- `<debug_flag>` is for lines that only appear if the `HistoryDebug` parameter is set to *On*
- `<action>` represents the type of action, such as **WrReg** or **SIM WRITE**. These are described below in more detail.

- `<information>` contains details related to the event

The history file contains detailed model simulation messages in order of simulation time. It maintains a consistent structure, and includes following information:

- Register Writes (**WrReg**)

  WrReg stands for Write to Register. Any line denoted by **WrReg** shows the details of the write. The message shows the Register Name and the value to be written.

- Memory Same (**WRREG_SAME**)

  Certain registers may not be written with the exact value specified by the write command due to access types of one or more bits. However, if the exact value is written, the **WREG_SAME** is issued.

- Memory Difference (**WRREG_DIFF**)

  Certain registers may not be written with the exact value specified by the write command due to access types of one or more bits. However, if the exact value is NOT written, the **WREG_DIFF** is issued.

- Memory Writes (**SIM WRITE**)

  SIM WRITE is the actual write to model memory resources. It shows both the address and the data value written to memory location. The address mapping can be found in the `denaliCdn_apbTypes.svh` file for SystemVerilog or `api_cdn_apb.h` for use with the DRAPE.

- Memory Reads (**SIM READ**)

  SIM READ is the actual read from model memory resources. It shows both the address and the data that is read.

- Event Information (**Info**)

  The line containing **Info** keyword shows general information about different events happening. This information is very useful for debugging purposes. Whenever any major event happens inside the model and could be useful for debugging, it is reported as Info.

- Protocol Layer Information

  The Layer messages are general informational messages that describe events that occur with the model at different layers.

- State Machine Transition (**State**)

  The **State** history message shows a transition of state machine. This message gives information about state machine name and context along with previous state and new state.

- Cycle

  The **Cycle** message shows the value received or transmitted at a given simulation time on RX or TX bus respectively.

- Configuration Instance

  The configuration instance message shows whether a given port in a device is a downstream port or upstream port function.

- Memory Instance

  The memory instance message shows the memory resource defined by BAR and Expansion Rom BAR in the design.

- Bus Range

  The message shows the bus range defined by a bridge.

- Tag Management

  The tag messages are used for tag Management activities

- Reset

  The reset messages indicate activities related to the reset process.

- Data Flow Control (Callback)

  The history file also contains information about transaction flow when it passes different callback locations. This information can be helpful in debugging the contents of a transaction at different points of data flow.

7.3.2. Trace File

The trace file is a very useful to debug and reproduce issues offsite. A test case can be extracted from this trace file because it stores sufficient information to create a test case for a particular issue. The trace file includes the following information:

- Trace File Header

  The top part of tracefile contains HDL simulator, operating system and the VIP version information used for simulation.

- `.denalirc` Settings

  The trace file stores the `.denalirc` parameters settings used during simulations.

- Device Instance

The trace file contains information about instance name and corresponding instance ID. This instance ID is used in rest of the trace file to identify model. For example:

2 i testbench.i0

where,

i0 – An instance in the testbench

2 – The instance ID corresponding to *testbench.i0*. The trace file will use '2' (first character on the line) to represent any information related to *testbench.i0*.

I/i – An instance in the trace file

- SOMA File

  The trace file contains information about the SOMA file used for each CDN_APB VIP model. It contains the path of the SOMA file used for a model as well as all settings in the SOMA file.

  The CDN_APB VIP model reads all these SOMA values for different parameters and writes to corresponding registers present inside model at 0 simulation time. All these writes can be seen in the trace file at the start of simulation.

- Simulation Time

  The trace file contains all the activities on a specific simulation time followed by simulation time stamp. The format for the time stamp is: `-<simulation time> <time unit>-`

- Event on Pin

  The Symbol 'E' represents an event on a particular pin in the trace file. The first character always represents the instance ID. For example:

  2 E RX 00000000

  2 – The instance ID

  E – The symbol for event followed by value (0000000 – Represents the data value)

  RX – The pin name

- Scheduled Event on Pin

  The Symbol 'S' represents an event scheduled on a particular pin in the trace file. For example.:

  2 S TX 00001100 10 ns T

  2 – The Instance ID

S – The symbol to represent event scheduling followed by value 10 ns – after 10 ns of current simulation time.

TX – The TX pin name

- History File Info

  In the `.denalirc` file, if the setting `HistoryDebug` is *On* then all the history file information can be found in the trace file too. The symbol 'H' is used to represent the history information.

7.3.2.1. Creating a Trace File

To create a trace file:

- In your `.denalirc` file, turn on trace file generation by adding the following line:

  ```
  Tracefile denali.trc
  ```

  where `denali.trc` is the name of your trace file.

  Note

  A line containing `Tracefile denali.trc` might already be present but commented out in your `.denalirc` file. If so, uncomment it (by removing the "#" at the beginning of the line).

  The `.trc` suffix for the name of the trace file is not mandatory. It is recommended for ease of identification.

7.4. Verification Messages

7.4.1. Changing Message Severity

You can control the severity of a verification message by writing to the DENALI_CDN_APB_REG_ErrCtrl register. Bits [3:0] of this register represent the severity field, which determines how the CDN_APB VIP displays the error. Valid values are specified by the denaliCdn_apbErrSeverityCtrlT enum type. For example:

- DENALI_CDN_APB_ERR_CONFIG_SEVERITY_Error = 2 /* Change the severity level to Error
- DENALI_CDN_APB_ERR_CONFIG_SEVERITY_Warn = 3 /* Change the severity level to Warn
- DENALI_CDN_APB_ERR_CONFIG_SEVERITY_Info = 4 /* Change the severity level to Info

The following code illustrates how to change the severity of an item in the testbench from Error to Info:

```
errId = <Some Protocol Error value>
severity = DENALI_CDN_APB_ERR_CONFIG_SEVERITY_Info;
value = (errId << DENALI_CDN_APB_Rpos__MODEL_ERROR_CTRL_errId) |
```

```
(severity << DENALI_CDN_APB_Rpos__MODEL_ERROR_CTRL_severity);
regInst.writeReg(DENALI_CDN_APB_REG_ErrCtrl,value);
```

To change the severity of several messages, you can write to this register multiple times. Every time you write to this register, the CDN_APB VIP processes it immediately and stores the information.

Note

The method for changing message severity differs when using UVM. For information on changing message severity using UVM, see Section 8.3.5, *"Error Reporting and Control"*.

7.4.2. Message List

Following is a list of the messages that can be reported by the CDN_APB VIP. Each item consists of the enumeration, followed by the string (with all formatting parameters) that is printed during simulation. Below the string are listed various reporting criteria for that message.

1. **DENALI_CDN_APB_UNSET**

   This field is not set

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

2. **DENALI_CDN_APB_NONFATAL_WARNING_MASTER_DATA_ITEM_CREATION_INVALID_INPUT**

   Master could not create a legal VIP transaction from input and the item is discarded.

   Report on: Tx Rx *Severity: WARNING* *Callback: Yes* *Coverage: Yes*

3. **DENALI_CDN_APB_NONFATAL_WARNING_SLAVE_DATA_ITEM_CREATION_INVALID_INPUT**

   Slave could not create a legal VIP transaction from input and the item is discarded.

   Report on: Tx Rx *Severity: WARNING* *Callback: Yes* *Coverage: Yes*

4. **DENALI_CDN_APB_FATAL_ERR_CDN_APB901_ENV_CONFIGURATION_NO_SLAVES_DEFINED**

   ENV configuration no slaves defined.

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

5. **DENALI_CDN_APB_FATAL_ERR_CDN_APB902_ENV_CONFIGURATION_TWO_OR_MORE_SLAVES_HAS_THE_SAME_NAME**

   ENV configuration two or more slaves has the same name.

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

6. **DENALI_CDN_APB_FATAL_ERR_CDN_APB903_ENV_CONFIGURATION_AMBA4_SUPPORT_WITH_NO_AMBA3**

   ENV configuration set AMBA4 support when not supporting AMBA3.

DENALI_CDN_APB_FATAL_ERR_CDN_APB903_ENV_CONFIGURATION_AMBA4_SUPPORT_WITH_NO_AMBA3

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

7. **DENALI_CDN_APB_FATAL_ERR_CDN_APB012_BAD_SIGNAL_VALUE**

   MASTER has driven Bad Signal values.

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

8. **DENALI_CDN_APB_FATAL_ERR_CDN_APB013_PADDR_CHANGED_ON_WAIT_STATE**

   MASTER has changed the paddr signal during a wait state.

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

9. **DENALI_CDN_APB_FATAL_ERR_CDN_APB014_PSEL_CHANGED_ON_WAIT_STATE**

   MASTER has changed the psel signal during a wait state.

   Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

10. **DENALI_CDN_APB_FATAL_ERR_CDN_APB015_PENABLE_CHANGED_ON_WAIT_STATE**

    MASTER has changed the penable signal during a wait state.

    Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

11. **DENALI_CDN_APB_FATAL_ERR_CDN_APB016_PWRITE_CHANGED_ON_WAIT_STATE**

    MASTER has changed the pwrite signal during a wait state.

    Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

12. **DENALI_CDN_APB_FATAL_ERR_CDN_APB017_PWDATA_CHANGED_ON_WAIT_STATE**

    MASTER has changed the pwdata signal during a wait state.

    Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

13. **DENALI_CDN_APB_FATAL_ERR_CDN_APB024_PENABLE_WAS_NOT_DEASSERTED_AFTER_PREADY_DEASSERTED**

    MASTER did not deassert penable signal after selected slave pready signal was deasserted.

    Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

14. **DENALI_CDN_APB_FATAL_ERR_CDN_APB018_REQ_TIMEOUT**

    MASTER did not finish its transfer transmission within 100 cycles.

DENALI_CDN_APB_FATAL_ERR_CDN_APB018_REQ_TIMEOUT

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

15. **DENALI_CDN_APB_FATAL_ERR_CDN_APB019_WRONG_PSEL_ASSERTION**

MASTER did not assert the correct psel signal.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

16. **DENALI_CDN_APB_FATAL_ERR_CDN_APB020_PENABLE_ASSERTED_WHEN_PSEL_DEASSERTED**

MASTER asserted the penable signal when psel was deasserted.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

17. **DENALI_CDN_APB_FATAL_ERR_CDN_APB021_PENABLE_WAS_NOT_ASSERTED_WITHIN_1_CYCLE_AFETR_PSEL_ASSERTION**

MASTER did not assert the penable signal within 1 cycle after the assertion of the psel.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

18. **DENALI_CDN_APB_FATAL_ERR_CDN_APB022_PENABLE_WAS_ASSERTED_TOO_EARLY**

MASTER asserted the penable signal too early after the assertion of the psel.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

19. **DENALI_CDN_APB_NONFATAL_ERR_CDN_APB023_ADDR_NOT_ALIGNED**

MASTER drove a not aligned address.

Report on: Tx Rx *Severity: WARNING* *Callback: Yes* *Coverage: Yes*

20. **DENALI_CDN_APB_FATAL_ERR_CDN_APB026_PPROT_CHANGED_ON_WAIT_STATE**

MASTER has changed the pprot signal during a wait state.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

21. **DENALI_CDN_APB_FATAL_ERR_CDN_APB027_PSTRB_CHANGED_ON_WAIT_STATE**

MASTER has changed the pstrb signal during a wait state.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

22. **DENALI_CDN_APB_FATAL_ERR_CDN_APB028_NON_ZERO_STROBE_FOR_READ_TRANSFER**

MASTER drove a none zero value for the pstrb signal when driving a READ transfer.

DENALI_CDN_APB_FATAL_ERR_CDN_APB028_NON_ZERO_STROBE_FOR_READ_TRANSFER

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

23. **DENALI_CDN_APB_FATAL_ERR_CDN_APB029_WRONG_STROBE**

MASTER drove a strobe that enables bytes which are outside the valid range.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

24. **DENALI_CDN_APB_FATAL_ERR_CDN_APB102_BAD_SIGNAL_VALUE**

SLAVE has driven Bad Signal values.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

25. **DENALI_CDN_APB_FATAL_ERR_CDN_APB103_REQ_TIMEOUT**

SLAVE did not finish its transfer transmission within 100 cycles.

Report on: Tx Rx *Severity: ERROR* *Callback: Yes* *Coverage: Yes*

26. **DENALI_CDN_APB_NONFATAL_ERR_CDN_APB025_UNMAPPED_ADDRESS**

MASTER started a transfer to address which is unmapped to any slave port.

Report on: Tx Rx *Severity: WARNING* *Callback: Yes* *Coverage: Yes*

7.5. Debugging the Simulation

Running a test case with the CDN_APB VIP often results in the display of a few errors or informational messages. The CDN_APB VIP can recover from some errors and continue the execution of the test case as intended. However, serious errors on the part of the DUT might place the model in an unrecoverable state and halt the simulation. This section describes how you can effectively debug your model.

The first step is to carefully read through each error message completely. These messages contain a lot of information, such as transaction fields, time of error, state in which the error occurred, which may give a clue as to what is wrong with the DUT.

Chapter 8. CDN_APB VIP Testbench Integration

This chapter describes the CDN_APB VIP integration with supported testbench interfaces and simulators. Currently, only SystemVerilog is supported.

The CDN_APB VIP provides a native class-based object-oriented interface to support SystemVerilog testbench methodologies, such as UVM and OVM.

- UVM - Cadence provides the UVM layer for CDN_APB. The UVM layer is located at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb`. For details on getting started with the UVM Layer, refer to Section 8.3, "*SystemVerilog Interface for UVM*". This section also provides an example test case.
- OVM - Cadence provides the OVM layer for CDN_APB. The OVM layer is located at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/ovm/cdn_apb`. For a description of each component in the OVM Layer, refer to the UVM Layer description at Section 8.3, "*SystemVerilog Interface for UVM*". You may use the UVM example as a reference for OVM provided you change the UVM specific naming conventions and syntax to OVM specific naming conventions and syntax.

8.1. Simulator Integration

The VIPCAT installation provides a set of utility scripts in `$CDN_VIP_ROOT/bin`. These scripts set up your environment for VIP simulation for various simulators and provide key simulator-specific options to integrate the VIP into your overall simulation setup, using your scripts or Makefiles.

If you are using Incisive Enterprise Simulator (IES) for your simulation environment, then you can also use *irun* to easily run a test case with the VIP models. Starting from IES product release 12.1, *irun* recognizes the VIP models from their instantiation interface and handles most of the invocation details for you.

8.1.1. VIP Scripts

The VIP scripts need to know the location of the VIPCAT installation. Throughout these instructions, the environment variable `CDN_VIP_ROOT` is used to refer to the path of the VIP installation. Although you are not required to set this environment variable, you are encouraged to do so. Otherwise, you need to specify the `-cdn_vip_root` option for all scripts.

Before running these scripts, you must ensure that you have set up all required third-party tools available in the environment (for example, the simulator, GCC, and so on).

The VIP scripts provide the following functions:

- Environment Setup (cdn_vip_setup_env)

Generates shell commands that set up the simulation environment. This script needs to know the VIPCAT installation path, the specific simulator, and the testbench methodology - either SystemVerilog with UVM layer, or basic SystemVerilog.

- Example Setup (cdn_vip_setup_example)

  Generates simulator-specific commands to run a VIP example from the release. This script needs to know the specific example being set up, along with the other key environment components, such as VIPCAT installation, specific simulator, and so forth.

  Note

  All scripts have an `-h` option that provides information on arguments to the script. For example, `cdn_vip_check_env -h` lists detailed information on each argument to the script.

8.1.1.1. cdn_vip_setup_env

The `cdn_vip_setup_env` script generates a shell file with necessary commands to set up your environment. This script requires the information described below:

- The VIPCAT installation directory is expected to be specified with the CDN_VIP_ROOT environment variable. Or, it must be explicitly specified with the `-cdn_vip_root` optional argument to the script.
- The simulator must be explicitly specified with the `-sim` argument to the script. The script is capable of setting up the environment for the following simulators: NCSIM (multi-step mode), NCSIM (single-step mode), VCS, and MTI. The corresponding simulator executables (ncsim, vcs, vlog) must be available in the path environment variable. Or, the simulator installation directory can be explicitly specified with the `-sim_root` optional argument to the script.
- The testbench methodology must be explicitly specified with the `-method` argument to the script. The script is capable of setting up the environment for SystemVerilog with UVM Layer or basic SystemVerilog.
- NCSIM users only: The user directory containing VIP-compiled libraries must be explicitly specified with the `-cdn_vip_lib` option to the script.
- NCSIM users only: The `-install` option must be specified in the following three scenarios:
  - First VIP download
  - Download of updated VIP (with additional functionality, etc.)
  - Upgrade of Cadence simulator installation

  This option installs the VIP Compiled libraries necessary for NCSIM in the specified user directory (specified with `-cdn_vip_lib <VIP_Compiled_Library_Directory>`).

8.1.1.1.1. Completing Environment Setup

`cdn_vip_setup_env` generates Bourne shell scripts `cdn_vip_env_<...>.sh` by default. If you use this shell (/bin/sh) or its derivatives, such as Bash (/bin/bash) or Z (/bin/zsh), you need to execute the following command to complete your environment setup (on your command-line prompt, or, in your script):

```
. cdn_vip_env_<...>.sh
```

If you use C shell (/bin/csh) or its derivatives such as Tenex C shell (/bin/tcsh), you need to specify an additional `-csh` option to generate shell scripts `cdn_vip_env_<...>.csh`. Then, you need to execute the following command to complete your environment set up, either on the command line or in your script:

```
source cdn_vip_env_<...>.csh
```

8.1.1.2. cdn_vip_setup_example

The cdn_vip_setup_example script generates a shell file with simulator-specific commands to compile and simulate a VIP example in the release. This script requires the key information detailed below. Information on other/optional arguments to the script is available by executing the following command:

```
cdn_vip_setup_example -h
```

- The VIPCAT installation directory is expected to be specified with CDN_VIP_ROOT environment variable. Or, it must be explicitly specified with the `-cdn_vip_root` optional argument to the script.
- The specific release example to be setup must be explicitly specified with the `-exampledir` argument to the script.
- The simulator must be explicitly specified with the `-sim` argument to the script. The script is capable of setting up examples for the following simulators: NCSIM (multi-step mode), NCSIM (single-step mode), VCS and MTI.
- The testbench methodology must be explicitly specified with the "-method"argument to the script. The script is capable of setting up examples for SystemVerilog with UVM Layer, basic SystemVerilog.
- NCSIM users only: The user directory containing VIP Compiled libraries must be explicitly specified with the `-cdn_vip_lib` option to the script.

`cdn_vip_setup_example` generates a Bourne shell script `cdn_vip_run_<...>.sh`. The optional `-run` argument to `cdn_vip_setup_example` also executes the generated script.

Important: for the generated script to compile/simulate successfully, the environment must be set up correctly with `cdn_vip_setup_env`.

8.1.1.3. cdn_vip_check_env

The cdn_vip_check_env script checks that the environment (path, LD_LIBRARY_PATH, etc.) are setup correctly for VIP simulation. This script requires the key information detailed below. Information on other/optional arguments to the script is available by executing the following command:

```
cdn_vip_check_env -h
```

- The VIPCAT installation directory is expected to be specified with CDN_VIP_ROOT environment variable. Or, it must be explicitly specified with the `-cdn_vip_root` optional argument to the script.
- The simulator must be explicitly specified with the `-sim` argument to the script. The script is capable of checking the environment for the following simulators: NCSIM (multi-step mode), NCSIM (single-step mode), VCS and MTI.
- The testbench methodology must be explicitly specified with the `-method` argument to the script. The script is capable of checking the environment for SystemVerilog with UVM Layer or basic SystemVerilog.
- NCSIM users only: The user directory containing VIP-compiled libraries must be explicitly specified with the `-cdn_vip_lib` option to the script. The script confirms that the VIP-compiled libraries have been correctly set up in the user directory.
- The optional argument `-licensing` lists all Cadence VIP licenses available in the environment in the file `cdn_vip_licenses.log`.

8.1.2. NCSIM

The `cdn_vip_setup_env` and `cdn_vip_setup_example` scripts generate shell commands to set up your simulation environment and run an example in the VIPCAT release with NCSIM (in either three-step mode or single-step mode).

8.1.2.1. Three-Step Mode

The key arguments (libraries and command-line options) for VIP simulations with NCSIM in three-step mode (compile, elaborate, simulate) are clearly specified in the generated script `cdn_vip_run_ncsim_sv.sh` (for NCSIM three-step simulations in a 32-bit environment). After confirming that the provided example compiles and simulates successfully, you are strongly encouraged to modify this generated script, and compile and simulate your own SystemVerilog testbench before proceeding to integrate the VIP with other components in your overall NCSIM three-step simulation framework (your Makefiles, scripts).

Examples of script invocations for NCSIM in three-step mode:

- Setting up C shell environment for the first VIP download (VIP compiled libraries need to be installed with -i option)

```
cdn_vip_setup_env -s ncsim_3s -cdn_vip_root <VIPCAT_Installation_Directory> -m sv -i apb -
cdn_vip_lib <VIP_Compiled_Library_Directory> -csh
```

```
source cdn_vip_env_ncsim_sv.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s ncsim_3s -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv -cdn_vip_lib <VIP_Compiled_Library_Directory>
cdn_vip_run_ncsim_sv.sh
```

Exact script invocations to set up one specific example in the VIPCAT release for NC-SIM in three-step mode are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/example/cdn_apb/svExamples/simpleExample/example_setup_ncsim.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_ncsim_sv.sh with environment setup commands
    #Note: NCSIM Compiled libraries path to be provided + libraries need to be installed
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s ncsim_3s -cdn_vip_root ${CDN_VIP_ROOT} -m sv -csh -
cdn_vip_lib vip_lib -i apb
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_ncsim_sv.csh

    #Step 3. Create cdn_vip_run_ncsim_sv.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s ncsim_3s -e ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/
svExamples/simpleExample -m sv -cdn_vip_lib vip_lib
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_ncsim_sv.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- The CDN_VIP_ROOT environment variable is correctly specified.
- The 'ncsim' executable is available in your path (and you have an ncsim license).

8.1.2.2. Single-Step Mode

The key arguments (libraries/command-line options) for VIP simulations with NCSIM in single-step mode (irun) are clearly specified in the generated script cdn_vip_run_irun.sh (for NCSIM single-step simulations in a 32-bit environment). After confirming that the provided example compiles/simulates successfully, you are strongly encouraged to modify this generated script and compile/simulate your own SystemVerilog testbench before proceeding to integrate the VIP with other components in your overall NCSIM single-step simulation framework (your Makefiles, scripts).

Examples of script invocations for NCSIM in single-step mode:

- Setting up C shell environment for the first VIP time (VIP compiled libraries need to be installed with -i option)

```
cdn_vip_setup_env -s ncsim_irun -cdn_vip_root <VIPCAT_Installation_Directory> -m sv -i apb -
cdn_vip_lib <VIP_Compiled_Library_Directory> -csh
source cdn_vip_env_irun_sv.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s ncsim_irun -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv -cdn_vip_lib <VIP_Compiled_Library_Directory>
cdn_vip_run_irun_sv.sh
```

Exact script invocations to set up one specific example in the VIPCAT release for NC-SIM in single-step mode are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/example/cdn_apb/svExamples/simpleExample/example_setup_irun.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_irun_sv.sh with environment setup commands
    #Note: NCSIM Compiled libraries path to be provided + libraries need to be installed
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s ncsim_irun -cdn_vip_root ${CDN_VIP_ROOT} -m sv -csh -
cdn_vip_lib vip_lib -i apb
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_irun_sv.csh

    #Step 3. Create cdn_vip_run_irun_sv.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s ncsim_irun -e ${CDN_VIP_ROOT}/tools/denali/example/
cdn_apb/svExamples/simpleExample -m sv -cdn_vip_lib vip_lib
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_irun_sv.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- The CDN_VIP_ROOT environment variable is correctly specified.
- The 'ncsim' executable is available in your path (and you have an ncsim license).

8.1.2.3. Using *irun*

If your simulator is NCSIM, you can use the *irun* invocation tool to handle the details of incorporating VIP models. When you add an option to specify the location of the VIPCAT installation, *irun* uses that information to treat the VIP models as native simulation components. *irun* automatically compiles the model and loads the necessary VIP shared objects, so all you need to do is specify the model instantiation interface and the VIP HDL packages that you are using.

To get native treatment for the VIP models, you need the following:

- Incisive Enterprise Simulator release 12.1 or later
- The VIP model instantiation interface that originates from the VIP release 11.30.012-s or later. If you are using older versions, you should regenerate them with Pureview or get new copies from Cadence customer support.

The `-CDN_VIP_ROOT` option tells irun where the VIPCAT installation is located. Throughout these instructions, the environment variable `CDN_VIP_ROOT` is used to refer to the path to the VIP installation. You are not required to set this environment variable; it is simply a shorthand for this location.

Here is an example that uses *irun* to simulate with the CDN_APB VIP:

```
irun \
   -cdn_vip_root ${CDN_VIP_ROOT} \
   ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/denaliMem.sv \
   ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/denaliCdn_apb.sv \
   ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/svExamples/simpleExample/activeMaster.v \
   ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/svExamples/simpleExample/activeSlave.v \
   ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/svExamples/simpleExample/passiveMaster.v \
   ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/svExamples/simpleExample/tb.sv \
   -incdir ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/svExamples/simpleExample \
   -top pstest -timescale 1ps/1ps
```

Note

The files used in this example are located in the VIPCAT installation. If you want to run your own copy of the example, copy the example directory into a local directory and update the paths to the files in the `cdn_apb_irun.csh` script.

Note

If you use an old HDL instantiation interface by mistake, you will get an elaborator error that begins like this:

```
ncelab: *W,MISSYST
```

You can verify that you have new code by looking for the following line at the top of the file:

```
// pragma cdn_vip_model -class cdn_apb
```

8.1.3. VCS

The cdn_vip_setup_env and cdn_vip_setup_example scripts generate shell commands to set up your simulation environment and run an example in the VIPCAT release with VCS.

The key arguments (libraries and command-line options) for VIP simulations with VCS are clearly specified in the generated script `cdn_vip_run_vcs_sv.sh` (for VCS simulations in a 32-bit environment). After confirming that the provided example compiles and simulates successfully, you are strongly encouraged to modify this generated script, and compile and simulate your own SystemVerilog testbench before proceeding to integrate the VIP with other components in your overall VCS simulation framework (your Makefiles, scripts).

Examples of script invocations for VCS:

- Setting up your Bourne shell environment:

```
cdn_vip_setup_env -s vcs -cdn_vip_root <VIPCAT_Installation_Directory> -m sv
. cdn_vip_env_vcs_sv.sh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s vcs -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv
```

```
cdn_vip_run_vcs_sv.sh
```

Exact script invocations to set up one specific example in the VIPCAT release for VCS are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/example/cdn_apb/svExamples/simpleExample/example_setup_vcs.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_vcs_sv.sh with environment setup commands
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s vcs -cdn_vip_root ${CDN_VIP_ROOT} -m sv -csh
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_vcs_sv.csh

    #Step 3. Create cdn_vip_run_vcs_sv.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s vcs -e ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/
svExamples/simpleExample -m sv
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_vcs_sv.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- The CDN_VIP_ROOT environment variable is correctly specified.
- The 'vcs' executable is available in your path (and you have a vcs license).

8.1.4. MTI

The `cdn_vip_setup_env` and `cdn_vip_setup_example` scripts generate shell commands to set up your simulation environment and run an example in the VIPCAT release with MTI.

The key arguments (libraries and command-line options) for VIP simulations with MTI are clearly specified in the generated script `cdn_vip_run_mti_sv.sh` (for MTI simulations in a 32-bit environment). After confirming that the provided example compiles and simulates successfully, you are strongly encouraged to modify this generated script, and compile and simulate your own SystemVerilog testbench before proceeding to integrate the VIP with other components in your overall MTI simulation framework (your Makefiles, scripts).

Examples of script invocations for MTI:

- Setting up your C shell environment:

```
cdn_vip_setup_env -s mti -cdn_vip_root <VIPCAT_Installation_Directory> -m sv
source cdn_vip_env_mti_sv.sh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s mti -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv
cdn_vip_run_mti_sv.sh
```

Exact script invocations to set up one specific example in the VIPCAT release for MTI are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/example/cdn_apb/svExamples/simpleExample/example_setup_mti.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_mti_sv.sh with environment setup commands
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s mti -cdn_vip_root ${CDN_VIP_ROOT} -m sv -csh
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_mti_sv.csh

    #Step 3. Create cdn_vip_run_mti_sv.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s mti -e ${CDN_VIP_ROOT}/tools/denali/example/cdn_apb/
svExamples/simpleExample -m sv
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_mti_sv.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- The CDN_VIP_ROOT environment variable is correctly specified.
- The 'vsim' executable is available in your path (and you have a vsim license).

8.2. SystemVerilog Interface

This interface is used when you instantiate transaction objects in your SystemVerilog testbench and would like to employ tasks that can be used to perform the actions, such as transaction creation, call-back processing, and so on.

8.2.1. Transaction Interface

8.2.1.1. Instance Class

8.2.1.1.1. Class denaliCdn_apbInstance

The `denaliCdn_apbInstance` corresponds to the CDN_APB VIP instance instantiated in the testbench. Each such instance must have a corresponding `denaliCdn_apbInstance` object associated with it.

8.2.1.1.1.1. Constructor

```
function new(string instName, string cbFuncName = "");
```

This creates a new transaction to be initiated from the specified instance and returns a transaction handle pointing to the newly created empty transaction.

The `instName` must be a full path and the `cbFuncName` can be null for the constructor. However, if you want to define an explicit DPI callback function, it must be set before any callback point is added for monitoring.

The format of `cbFuncName` is `"<hdlScope>::<funcName>"`.

```
denaliCdn_apbInstance inst;
inst = new(top.i0);
```

8.2.1.1.1.2. Methods

8.2.1.1.1.2.1. getInstName()

Name

getInstName — Retrieves the testbench instance name that corresponds to this `denaliCdn_apbInstance`.

Synopsis

```
function string getInstName() ;
```

Description

Retrieves the testbench instance name that corresponds to this `denaliCdn_apbInstance`.

Example

```
denaliCdn_apbInstance inst;
inst = new("top.i0");
$display("InstName = %s", inst.getInstName());
```

8.2.1.1.1.2.2. setCbFuncName()

Name

setCbFuncName — Sets the DPI callback function name.

Synopsis

```
function void setCbFuncName(string cbFuncName) ;
```

Description

Sets the DPI callback function name.

The format of `cbFuncName` is `"<hdlScope>::<funcName>"`.

Example

```
denaliCdn_apbInstance inst;
inst = new("top.i0");
inst.setCbFuncName("top.my_if::myCbFunc");
```

8.2.1.1.1.2.3. getCbFuncName()

Name

getCbFuncName() — Gets the DPI callback function name.

Synopsis

```
function string getCbFuncName() ;
```

Description

Gets the DPI callback function name.

Example

```
denaliCdn_apbInstance inst;
inst = new("top.i0");
inst.setCbFuncName("top.my_if::myCbFunc");
$display("FuncName = %s", inst.getCbFuncName());
```

8.2.1.1.1.2.4. transAdd()

Name

transAdd — Adds a newly created transaction to the model’s user queue.

Synopsis

```
function integer transAdd(denaliCdn_apb obj, integer portNumOrQueueNum, denaliArgTypeT insertType =
DENALI_ARG_trans_append) ;
```

Description

Adds a newly created transaction to the model’s user queue. Any transaction fields that have been assigned are propagated to the internal model transaction.

8.2.1.1.1.2.5. setCallback()

Name

setCallback — Adds a given callback reason to the list of callback points being monitored.

Synopsis

```
virtual function integer setCallback(denaliCdn_apbCbPointT cbRsn) ;
```

Description

Adds a given callback reason to the list of callback points being monitored.

8.2.1.1.1.2.6. clearCallback()

Name

clearCallback — Removes a given callback reason from the list of callback points being monitored.

Synopsis

```
virtual function integer clearCallback(denaliCdn_apbCbPointT cbRsn) ;
```

Description

Removes a given callback reason to the list of callback points being monitored.

8.2.1.1.1.2.7. getQueuePending()

Name

getQueuePending — Returns the number of pending transactions in the instance's transaction queue.

Synopsis

```
function int getQueuePending(integer portNum) ;
```

Description

Retrieves the number of pending transactions currently in the instance's transaction queue.

Example

```
denaliCdn_apbInstance inst;
inst = new("top.i0");
$display("Queue pending items=", inst.getQueuePending(0));
```

8.2.1.2. Packet Classes

8.2.1.2.1. Class denaliCdn_apbTransaction

8.2.1.2.1.1. Fields

Table 8.1. Class denaliCdn_apbTransaction Fields

Field Name	Field Type	Description
Type	rand denaliCdn_apbTrTypeT	Transaction type.
PortNum	rand reg [31:0]	Port/Queue number
Callback	rand denaliCdn_apbCbPointT	ID of the callback currently associated with the transaction. Useful in the callback processing user code.
ErrorString	rand string	Character string describing the transaction errors for the debug
ErrorId	rand denaliCdn_apbErrorTypeT	This is the Error ID associated with error callbacks
ErrorInfo	rand reg [31:0]	Transaction error severity
UserData	rand reg [31:0]	Placeholder for the user-specific information
Delay	rand reg [31:0]	Transaction delay (in clock cycles). Please do not use this field
ReqDelay	rand reg [31:0]	The delay until driving the req channel
Direction	rand denaliCdn_apbDirectionT	The Direction of the transfer. either READ or WRITE
Addr	rand reg [31:0]	The address of the transfer
Data	rand reg [31:0]	The data of the transfer
DataXMask	rand reg [31:0]	Bit mask for the transfer's data monitored on the bus, where '1' indicates 'X' value

Field Name	Field Type	Description
DataZMask	rand reg [31:0]	Bit mask for the transfer's data monitored on the bus, where '1' indicates 'Z' value
Slverr	rand reg	The Slave response for the transfer (AMBA3 onward)
Strobe	rand reg [3:0]	The strobe for the write data (AMBA4)
Privileged	rand denaliCdn_apbPrivilegedModeT	AMBA4 PPROT[0] - NORMAL or PRIVILEGED
Secure	rand denaliCdn_apbSecureModeT	AMBA4 PPROT[1] - SECURE or NONSECURE
DataInstr	rand denaliCdn_apbFetchKindT	AMBA4 PPROT[2] - DATA or INSTRUCTION
IsStarted	rand reg	Was the transfer started.
IsEnded	rand reg	Was the transfer ended.
Duration	rand reg [63:0]	Transfer duration
ModelGeneration	rand reg	if TRUE - use the core model's constraints for generation. if FALSE - use only SV constraints for generation.
ReqResponseDelay	rand reg [31:0]	The delay until driving the req channel
StartTime	rand reg [63:0]	Time stamp for beginning of send/receive
EndTime	rand reg [63:0]	Time stamp for end of send/receive

8.2.1.2.1.2. Constructor

8.2.1.2.1.2.1. new()

Name

new — Constructor for the class denaliCdn_apbTransaction.

Synopsis

```
function new();
```

Description

This constructs an object of the class denaliCdn_apbTransaction.

8.2.1.2.1.3. Methods

8.2.1.2.1.3.1. transSet()

Name

transSet — Propagates all modified transaction fields to the internal model transaction.

Synopsis

```
virtual function transSet();
```

Description

Propagates all modified transaction fields to the internal model transaction.

8.2.1.2.1.3.2. printInfo()

Name

printInfo — Prints the fields/contents of the transaction.

Synopsis

```
virtual function integer printInfo(integer arraySize = 32);
```

Description

Prints the fields/contents of the transaction. The 'arraySize' parameter specifies the maximum number of elements to print for arrays.

8.2.2. Coverage Interface / CDN_APB VIP Coverage

Coverage is based on monitor events/callbacks and data structures. It lets you see both how well you are proceeding with the verification task and provides statistics about your DUT in the APB environment.

The VIP has predefined protocol coverage defined for all major events, data items and their fields. You can add your own coverage groups and items, and you can disable coverage collection from parts of your environment or specific coverage groups. Cadence recommends adjusting the coverage specifically for your DUT.

The Coverage can be used in specifics instance in your SystemVerilog or SystemVerilog UVM testbench.

All the coverage files are under `$DENALI/ddvapi/sv/coverage/cdn_apb`.

8.2.2.1. Coverage Class

The denaliCdn_apbCoverageInstance class is instantiated under the CDN_APB VIP instance once your enable coverage for that instance.

8.2.2.1.1. cover_sample()

Name

cover_sample — Triggers covergroup sampling.

Description

This method is called by an internal pre-defined CDN_APB VIP callback every time a covergroup needs to be sampled. The method gets a covergroup value and a relevant transaction, and based on the covergroup value it got, it triggers the relevant covergroup sampling.

8.2.2.1.2. Coverage Definitions

The coverage class includes the definition of CDN_APB predefined protocol coverage. The coverage is divided to covergroups sampled on a relevant callback, when each group contains several cover-points.

8.2.2.2. Coverage File Structure

The following table describes coverage files, which can be found in `$DENALI/ddvapi/sv/coverage/cdn_apb`.

Table 8.2. Coverage File Structure

Filename	Description
denaliCdn_apbCoverage.svh	This file contains the VIP coverage definitions and the trigger for the covergroups sampling.
denaliCdn_apbCoverageInstance.sv	This file contains definition of the coverage class.
denaliCdn_apbCoverGroupNew.svh	This file contains new actions for the cover-groups.
denaliCdn_apbCoverGroupEnable.svh	This file contains *_enable fields for each cover group that can be used to disable specific cover groups from being collected. By default all are set to 1 (enabled).

8.2.2.3. Enabling Coverage Collection

To enable coverage collection in your simulation, please use the following steps:

1. Enable the coverage option in your simulator.
2. Set up coverage collection for the relevant instances, as described in Section 8.2.2.3.2, “*SystemVerilog Coverage Setup*”.
3. Enable the coverage:

For the SystemVerilog interface, see Section 8.2.2.3.3, *"Enabling Coverage using SystemVerilog"*.
For the SystemVerilog for UVM, see Section 8.2.2.3.4, *"Enabling Coverage using UVM SystemVerilog"*.

8.2.2.3.1. Enable the Coverage Option in your Simulator

In addition to setting up the VIP for coverage, you need to ensure your run command includes any flags required by the simulator to collect coverage.

For example, for ncsim, you need to provide the following flags:

```
-coverage functional -covoverwrite -write_metrics
```

8.2.2.3.2. SystemVerilog Coverage Setup

For coverage collection, you must first enable the collection of coverage for the relevant instance from the `.denalirc` file, using the following field:

```
EnableCoverage <instance pattern>
```

The following are examples of <instance pattern>.

- `*` is used to specify all instances.
- `*[Pp]assive*` is used to specify all the passive instances.
- `top.<instance_name>` is used to specify an instance called instance_name under top, where top is your top module. (The patterns come from the HDL instance names.)

8.2.2.3.3. Enabling Coverage using SystemVerilog

- Start with doing the SystemVerilog coverage setup using the `.denalirc` file. For more details, refer to Section 8.2.2.3.2, *"SystemVerilog Coverage Setup"*.
- After you instantiate the relevant denaliCdn_apbInstance using its `new` function, call the instance `create_cover()` function. See the code example in Section 8.2.4, *"Example Testcase"*.

8.2.2.3.4. Enabling Coverage using UVM SystemVerilog

- Start with doing the SystemVerilog coverage setup using the `.denalirc` file. For more details, refer to Section 8.2.2.3.2, *"SystemVerilog Coverage Setup"*.
- To enable your UVM monitor to collect coverage, set the **coverageEnable** bit to 0. For details, refer to the UVM user monitor code example in Section 8.3.8, *"Example Testcase"*.

8.2.2.4. Creating DUT-Specific Coverage Definitions

This section discusses how to filter predefined coverage definitions and add customized coverage definitions.

8.2.2.4.1. Filtering Predefined or Irrelevant Coverage Definitions

Some coverage definitions might be irrelevant in your verification environment. In that case, you can filter the predefined coverage definitions to disable some of the coverage.

To filter covergroups, we use configuration fields which are defined in the coverage class.

For each covergroup we have an enable field called <*covergroup name*>_enable field. These fields are used to disable creation of the covergroup in the new function (covergroups can be created only in the class constructor, per the SystemVerilog LRM), and to check whether the covergroup needs to be sampled.

By default, all the relevant fields for the used VIP configuration are enabled (set to 1).

8.2.2.4.1.1. Filtering Irrelevant Covergroups using the UVM SV intreface

Use the set_config_field method in your testbench to disable the *_enable field you want, by setting its value to 0.

8.2.2.4.1.2. Filtering Irrelevant Covergroups using the SV interface without UVM

The coverage class contains an empty hook method called userDisableCoverGroups ().

To disable the covergroup you don't need:

1. Create your own coverage class which inherits from the given coverage class (denaliCdn_apbCoverageInstance).
2. In this class, implement the userDisableCoverGroups () method to set the relevant cover group *_enable field to 0, and by that disable the equvivalent covergroups.
3. Create your own VIP instance which inherits from the given VIP instance (denaliCdn_apbInstance).
4. In this VIP instance, re-implement the new_cover_class() method to new the coverInst field with the coverage class you created instead if the given coverage class.

```
   // Implement the new_cover_class to create the coverInst of the new coverage class you have
created

   virtual function void new_cover_class();

    // Create a new instance of your coverage class type
    // which extends the provided coverage class and assign it for coverInst
    userCoverageClass myCoverInst;
    myCoverInst = new(instName, this);
    coverInst = myCoverInst;
   endfunction
```

8.2.2.4.2. Adding Customized Coverage Definitions

Cadence recommends adding coverage groups specific to your DUT implementation, as required.

8.2.2.4.2.1. Adding Covergroup Definitions using UVM SV

To add coverage definitions:

1. Create your own UVM coverage class which inherits from the base VIP UVM coverage class (cdnApbUvmCoverage)
2. In this class:
   a. Define the required cover groups using the denaliCdn_apbTransaction field (defined in the base coverage class), and the cdnApbInstance register space.
   b. Be sure to new the new coverage groups in your class new function.
3. Ensure you call the new covergroups sample method on the right events in your environment code.
4. In you monitor, which inherits from cdnApbUvmMonitor, use the factory to ensure that the coverage class instance is created using your new coverage class.

8.2.2.4.2.2. Adding Covergroup Definitions using the SV interface without UVM

To add coverage definitions:

1. Create your own coverage class which inherits from the given coverage class (denaliCdn_apbCoverageInstance)
2. In this class:
   a. Define the required cover groups using the denaliCdn_apbCoverageInstance class denaliCdn_apbTransaction field, and the denaliCdn_apbInstance register space.
   b. Be sure to new the new coverage groups in your class new function.
3. Create your own VIP instance which inherits from the given VIP instance (denaliCdn_apbInstance).
4. In this VIP instance class:
   a. Re-implement the new_cover_class() method to new the coverInst field with the coverage class you created instead if the given coverage class.

```
     // Implement the new_cover_class to create the coverInst of the new coverage class you
have created

     virtual function void new_cover_class();

      // Create a new instance of your coverage class type
      // which extends the provided coverage class and assign it for coverInst
      userCoverageClass myCoverInst;
      myCoverInst = new(instName, this);
      coverInst = myCoverInst;
```

```
    endfunction
```

b. Implement the relevant denaliCdn_apbInstance callbacks to activate your covergroups sample().

8.2.3. SystemVerilog File Structure

The file structure that supports the CDN_APB VIP SystemVerilog interface includes the following:

- `$DENALI/ddvapi/sv/denaliCdn_apb.sv`

  This file defines all the CDN_APB VIP transaction classes for SystemVerilog.

  You must import everything that is defined within `DenaliSvCdn_apb` package within your testbench to make use of these transaction classes.

- `$DENALI/ddvapi/sv/denaliCdn_apbTypes.svh`

  This file defines all the enumeration types for each of the registers supported within the CDN_APB configuration space, the common header and all support capability structures.

- `$DENALI/ddvapi/sv/denaliCdn_apbImports.sv`

  This file defines all the SystemVerilog-DPI function calls mapped to equivalent C functions as included and used within the `denaliCdn_apb.sv` and `denaliCdn_apbSvIf.c` files.

- `$DENALI/ddvapi/sv/denaliCdn_apbSvIf.c`

  This file defines all the integration code between SystemVerilog and C-API needed to pass to the model interface.

- `$DENALI/ddvapi/sv/denaliMem.sv`

  This file defines an optional memory support package that can be used for memory models as well as internal memories and registers. Package `DenaliSvMem` contains global routines to support a traditional procedural programming style, and class `denaliMemInstance` to support an object oriented programming style. The package includes utilities for reads/writes, callbacks, memory transactions, SOMA parameter value access, TCL command evaluation, and so on.

- `$DENALI/ddvapi/sv/denaliMemSvIf.c`

  This file defines the integration interface between `denaliMem.sv` and the model C code, and is required to be compiled as part of creating your testbench if you are using the `DenaliSvMem` package.

8.2.4. Example Testcase

To create a testcase:

1. Use PureView to create SOMA files for requisite model instances. For details, refer to Chapter 3, *PureView Graphical Tool*.
2. Use PureView and the same configuration to create an HDL instantiation interface for those instances. For details, refer to Chapter 3, *PureView Graphical Tool*.
3. Create the top level and instantiate the CDN_APB VIP and DUT models in it.
4. Set the required `.denalirc` variables.
5. Create the testbench and add transactions to the user queue of the host model.

The following example shows the `tb.sv` file.

```
import DenaliSvCdn_apb::*;
import DenaliSvMem::*;

class myTransaction extends denaliCdn_apbTransaction;

 function new();
     super.new();
   endfunction

    constraint user_dut_information {
        NumberOfSlaves == 1;
        Addr < 'h3fffffff; // max mapped address
    }

endclass

// A denaliCdn_apbInstance should be instantiated per agent
// You should extend it and create your own, to implement callbacks
class apbInstance extends denaliCdn_apbInstance;

    denaliMemInstance regInst ; // Handle to the register-space

    function new(string instName);
        super.new(instName);
        regInst = new( { instName, "(registers)" } );
    endfunction

    virtual function int DefaultCbF(ref denaliCdn_apbTransaction trans);

        if (trans != null)
        $display("*******************************");
        $display("DefaultCbF :  Type:  ",trans.Type.name());
        $display("DefaultCbF :  Callback:   ",trans.Callback.name());
        $display("DefaultCbF :  Agent name:   ", instName);
        $display("*******************************");

        begin
            if (trans.Callback == DENALI_CDN_APB_CB_Error) begin
                $display("DefaultCbF :  trans cb is DENALI_CDN_APB_CB_Error");
                $display("DefaultCbF :  Description: ",trans.ErrorString);
                $display("DefaultCbF :  ErrorId:  ",trans.ErrorId.name());

            end

            if (trans.Callback == DENALI_CDN_APB_CB_DriverTransferEnded) begin
             void'(trans.printInfo());
            end
```

```
            if (trans.Callback == DENALI_CDN_APB_CB_MonTransferEnded) begin
             void'(trans.printInfo());
            end

     end
        return super.DefaultCbF(trans);

    endfunction

    // Map address segments to slave ports.
    // According to this mapping the master will know which PSELx signal to assert for each address
    // from_address - the start address of the segment
    // to_address - the start address of the segment
    // slave_port_index - slave port index
    function void mapAddressSegment(reg [31:0] from_address, reg [31:0] to_address, reg[3:0]
 slave_port_index);
     regWrite(DENALI_CDN_APB_REG_SegmentStartAddress, from_address);
  regWrite(DENALI_CDN_APB_REG_SegmentEndAddress, to_address);
  regWrite(DENALI_CDN_APB_REG_SegmentPselIndex, slave_port_index);
    endfunction

    //
    // Read from a 32-bit DEANLI register
    //
    virtual function reg [31:0] regRead( denaliCdn_apbRegNumT addr );
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        void'( regInst.read( trans ) );
        regRead[31:24] = trans.Data[0];
        regRead[23:16] = trans.Data[1];
        regRead[15:08] = trans.Data[2];
        regRead[07:00] = trans.Data[3];
    endfunction

    //
    // Write to a 32-bit DEANLI register
    //
    virtual function void regWrite( denaliCdn_apbRegNumT addr, reg [31:0] data );
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        trans.Data = new[4];
        trans.Data[0] = data[31:24];
        trans.Data[1] = data[23:16];
        trans.Data[2] = data[15:08];
        trans.Data[3] = data[07:00];
        void'( regInst.write( trans ) );
    endfunction

endclass

// Test module
module pstest;

 integer status;

    reg aclk;
    reg aresetn;
    wire [31:0] paddr;
    wire penable;
    wire pwrite;
    wire [31:0] pwdata;
    wire pready;
    wire [31:0] prdata;
 wire pslverr;
 wire psel0;

    always #50 aclk = ~aclk;
```

```
   initial
   begin
       aclk = 1'b0;
       aresetn = 1'b1;
       #100
       aresetn = 1'b0;
       #500
       aresetn = 1'b1;
   end

activeMaster MasterDevice(aclk,aresetn,paddr,penable,pwrite,pwdata,pready,prdata,pslverr,psel0);
passiveMaster MasterMonitor(aclk,aresetn,paddr,penable,pwrite,pwdata,pready,prdata,pslverr,psel0);

activeSlave SlaveDevice(aclk,aresetn,paddr,penable,pwrite,pwdata,pready,prdata,pslverr,psel0);

   apbInstance master;
   apbInstance monitor;
   apbInstance slave;

   //myTransaction extends denaliCdn_apbTransaction with DUT specific constraints
   myTransaction masterTransfer;
   myTransaction SlaveResponse;

initial
begin
 master = new ("pstest.MasterDevice");
 monitor = new ("pstest.MasterMonitor");
 slave = new ("pstest.SlaveDevice");
 $display("Created all CDN_APB instances");

 void'(master.setCallback(DENALI_CDN_APB_CB_Error));

    void'(slave.setCallback(DENALI_CDN_APB_CB_Error));

    void'(monitor.setCallback(DENALI_CDN_APB_CB_Error));
    void'(monitor.setCallback(DENALI_CDN_APB_CB_ResetEnded));
    void'(monitor.setCallback(DENALI_CDN_APB_CB_MonTransferEnded));

 $display("Set callbacks for the all instances");

 void'(monitor.create_cover());
 $display("Enable coverage collection");

 //set test verbosity to MEDIUM
 //we can use the enum value as set in the denaliCdn_apbTypes.sv file
       master.regWrite( DENALI_CDN_APB_REG_Verbosity, DENALI_CDN_APB_MESSAGEVERBOSITY_MEDIUM );

       // Reset signals values at simulation start
       master.regWrite( DENALI_CDN_APB_REG_ResetSignalsSimStart, 1 );
       slave.regWrite( DENALI_CDN_APB_REG_ResetSignalsSimStart, 1 );

       // set this check severity to IGNORE
       monitor.regWrite( DENALI_CDN_APB_REG_ErrorID,
CDN_APB_NONFATAL_ERR_CDN_APB023_ADDR_NOT_ALIGNED);
       monitor.regWrite( DENALI_CDN_APB_REG_ErrorSeverity, DENALI_CDN_APB_CHECKEFFECT_IGNORE);

       // Map the Master's address segments to slaves ports
       // If part of the address space is unmapped, the user should add constraint on 'Addr' field.
       master.mapAddressSegment('h0,'h3fffffff, 0);
       monitor.mapAddressSegment('h0,'h3fffffff, 0);

       #1000

 //We can set a transaction with only a few parameters and the module will randomize the rest
       masterTransfer = new();
       masterTransfer.Addr = 'h0;
```

```
        masterTransfer.ReqDelay = 'h5; // delay until assert PSELx
        status = master.transAdd(masterTransfer, 0);

        SlaveResponse = new();
        SlaveResponse.ReqResponseDelay = 'h3; // delay until assert PREADY
        status = slave.transAdd(SlaveResponse, 0);


        //We can send several transactions in a loop
        for (int i=0; i<100; i++) begin
         assert(masterTransfer.randomize());
         status = master.transAdd(masterTransfer, 0);

         assert(SlaveResponse.randomize());
         status = slave.transAdd(SlaveResponse, 0);

         #100;
        end

        #200000;

        $finish;

    end

endmodule
```

8.3. SystemVerilog Interface for UVM

This section provides details on how to integrate CDN_APB VIP into a UVM compliant test environments.

The CDN_APB VIP UVM layer is located under `$CDN_VIP_ROOT/tools/denali/ddva-pi/sv/uvm/cdn_apb`. All UVM layer classes are defined under the `cdnApbUvm` package.

8.3.1. Prerequisites

The prerequisites to integrate CDN_APB VIP to the UVM environment are as follows:

- Understanding of the SystemVerilog UVM. For details on UVM, refer to the website http://www.uvmworld.org [http://www.uvmworld.org].
- Understanding of the CDN_APB VIP, including the SystemVerilog Interface (Section 8.2, *"SystemVerilog Interface"*).

You can also refer to the *Cadence UVM SystemVerilog User Guide* installed in the Incisive release.

Each release also includes a UVM API reference document that is automatically generated with information about structs, fields, methods, and events. This document can be very helpful for debugging. To read this reference, open the following file with your Web browser:

- `$CDN_VIP_ROOT/doc/cdn_apb/apb_uvm_sv_ref/index.html`

The Cadence UVM Layer supports UVM version 1.1 and supports simulators NCSIM 10.2, VCS 2011.03-SP1-2, and MTI 10.0c.

8.3.2. Using UVM with Different HDL Simulators

Refer to Section 8.1, "*Simulator Integration*".

Here are some examples that show how you can simulate the CDN_APB VIP and SystemVerilog UVM with different HDL simulators.

8.3.2.1. NCSIM

This section provides examples of VIP script invocations for setting up the environment and running UVM examples in the release with NCSIM.

Three-Step Mode

Examples of script invocations for NCSIM in three-step mode:

- Setting up C shell environment for the first time (VIP compiled libraries need to be installed with the -i option):

```
cdn_vip_setup_env -s ncsim_3s -cdn_vip_root <VIPCAT_Installation_Directory> -m sv_uvm -i apb -
cdn_vip_lib  <VIP_Compiled_Library_Directory> -csh
source cdn_vip_env_ncsim_sv_uvm.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s ncsim_3s -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv_uvm -cdn_vip_lib  <VIP_Compiled_Library_Directory>
cdn_vip_run_ncsim_sv_uvm.sh
```

Exact script invocations to set up one specific UVM example in the VIPCAT release for NCSIM in three-step mode are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/simpleExample/example_setup_ncsim.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_ncsim_sv_uvm.sh with environment setup commands
    #Note: NCSIM Compiled libraries path to be provided + libraries need to be installed
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s ncsim_3s -cdn_vip_root ${CDN_VIP_ROOT} -m sv_uvm -csh -
cdn_vip_lib vip_lib -i apb
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_ncsim_sv_uvm.csh

    #Step 3. Create cdn_vip_run_ncsim_sv_uvm.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s ncsim_3s -e ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/uvm/
cdn_apb/examples/simpleExample -m sv_uvm -cdn_vip_lib vip_lib
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_ncsim_sv_uvm.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- CDN_VIP_ROOT environment variable is correctly specified.

- The 'ncsim' executable is available in your path (and you have a ncsim license).

Single-Step Mode

Examples of script invocations for NCSIM in single-step mode:

- Setting up C shell environment for the first time (VIP compiled libraries need to be installed with the -i option):

```
cdn_vip_setup_env -s ncsim_irun -cdn_vip_root <VIPCAT_Installation_Directory> -m sv_uvm -i apb -
cdn_vip_lib  <VIP_Compiled_Library_Directory> -csh
source cdn_vip_env_irun_sv_uvm.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s ncsim_irun -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv_uvm -cdn_vip_lib <VIP_Compiled_Library_Directory>
cdn_vip_run_irun_sv_uvm.sh
```

Exact script invocations to set up one specific UVM example in the VIPCAT release for NCSIM in single-step mode are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/simpleExample/example_setup_irun.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_irun_sv_uvm.sh with environment setup commands
    #Note: NCSIM Compiled libraries path to be provided + libraries need to be installed
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s ncsim_irun -cdn_vip_root ${CDN_VIP_ROOT} -m sv_uvm -csh -
cdn_vip_lib vip_lib -i apb
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_irun_sv_uvm.csh

    #Step 3. Create cdn_vip_run_irun_sv_uvm.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s ncsim_irun -e ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/uvm/
cdn_apb/examples/simpleExample -m sv_uvm -cdn_vip_lib vip_lib
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_irun_sv_uvm.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- CDN_VIP_ROOT environment variable is correctly specified.
- The 'ncsim' executable is available in your path (and you have a ncsim license).

8.3.2.2. VCS

This section provides examples of VIP script invocations for setting up the environment and running UVM examples in the release with VCS.

Examples of script invocations for VCS:

- Setting up C shell environment:

- The environment variable `CDN_VIP_ROOT` is used to refer to the path to the VIP installation.

```
cdn_vip_setup_env -s vcs -cdn_vip_root <VIPCAT_Installation_Directory> -m sv_uvm
source cdn_vip_env_vcs_sv_uvm.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s vcs -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv_uvm
source cdn_vip_env_vcs_sv_uvm.csh
```

Exact script invocations to set up one specific UVM example in the VIPCAT release with VCS are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/simpleExample/example_setup_vcs.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_vcs_sv_uvm.sh with environment setup commands
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s vcs -cdn_vip_root ${CDN_VIP_ROOT} -m sv_uvm -csh
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_vcs_sv_uvm.csh

    #Step 3. Create cdn_vip_run_vcs_sv_uvm.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s vcs -e ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/uvm/
cdn_apb/examples/simpleExample -m sv_uvm
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_vcs_sv_uvm.sh
```

Please confirm the following two requirements are met before executing the above scripts:

- CDN_VIP_ROOT environment variable is correctly specified.
- The 'vcs' executable is available in your path (and you have a vcs license).

8.3.2.3. MTI

This section provides examples of VIP script invocations for setting up the environment and running UVM examples in the release with MTI.

Examples of script invocations for MTI:

- Setting up C shell environment:

```
cdn_vip_setup_env -s mti -cdn_vip_root <VIPCAT_Installation_Directory> -m sv_uvm
source cdn_vip_env_mti_sv_uvm.csh
```

- Setting up (and running) an example provided in the VIPCAT release:

```
cdn_vip_setup_example -s mti -cdn_vip_root <VIPCAT_Installation_Directory> -e
 <VIPCAT_Example_Directory> -m sv_uvm
cdn_vip_run_mti_sv_uvm.sh
```

Exact script invocations to set up one specific UVM example in the VIPCAT release with MTI are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/`

`tools/denali/ddvapi/sv/uvm/cdn_apb/examples/simpleExample/example_setup_mti.csh`.

```
 #!/bin/csh -f
    #Step 1. Create cdn_vip_env_mti_sv_uvm.sh with environment setup commands
${CDN_VIP_ROOT}/bin/cdn_vip_setup_env -s mti -cdn_vip_root ${CDN_VIP_ROOT} -m sv_uvm -csh
    #Step 2. Complete environment setup by executing generated shell commands
source cdn_vip_env_mti_sv_uvm.csh

    #Step 3. Create cdn_vip_run_mti_sv_uvm.sh with simulator comands
    #Note: CDN_VIP_ROOT environment variable guaranteed to be available after Step 2
${CDN_VIP_ROOT}/bin/cdn_vip_setup_example -s mti -e ${CDN_VIP_ROOT}/tools/denali/ddvapi/sv/uvm/
cdn_apb/examples/simpleExample -m sv_uvm
    #Step 4. Execute generated script with compile/simulate commands (also doable by -r option to
 cdn_vip_setup_example)
./cdn_vip_run_mti_sv_uvm.sh
```

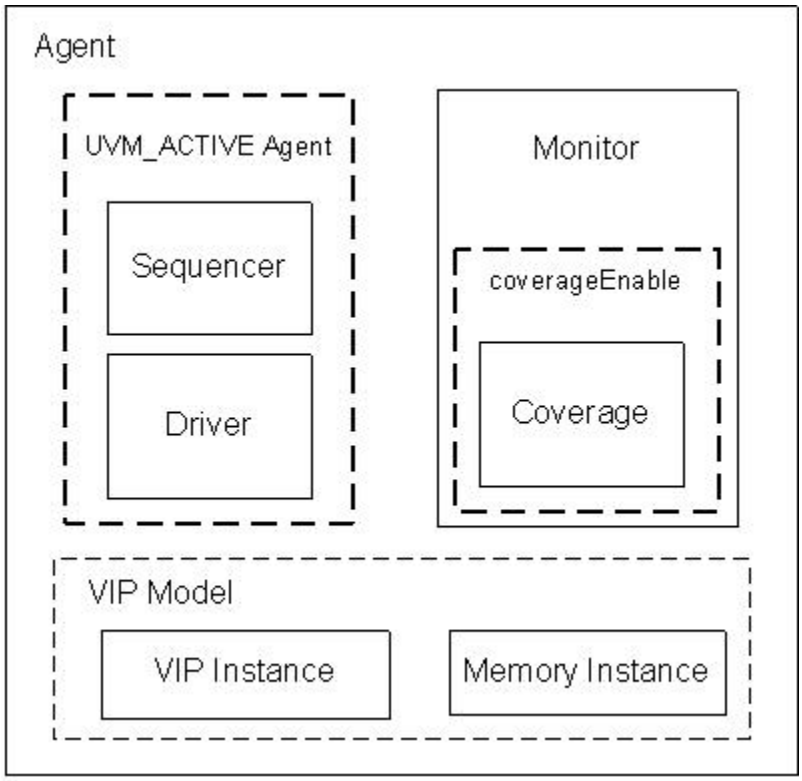
Please confirm the following two requirements are met before executing the above scripts:

- CDN_VIP_ROOT environment variable is correctly specified.
- The 'vsim' executable is available in your path (and you have a vsim license).

8.3.3. Architecture

The following diagram shows the CDN_APB VIP UVM agent architecture.

Figure 8.1. The CDN_APB VIP UVM Agent Architecture



The CDN_APB VIP SystemVerilog interface contains the following UVM components.

8.3.3.1. Agent

The Agent is an independent UVM device that contains the standard VIP UVM components. The CDN_APB VIP UVM agent is of type cdnApbUvmAgent and inherits from uvm_agent. The agent instantiates monitor (of type cdnApbUvmMonitor with a reference name **monitor**), driver (of type cdnApbUvmDriver with a reference name **driver**), Seqeuncer (of type cdnApbUvmSequencer with a reference name **sequencer**), instance (of type cdnApbUvmInstance with a reference name **inst**) and memory instance (of type cdnApbUvmMemInstance with a reference name **regInst**).The driver and sequencer get instantiated only when `is_active` is set to UVM_ACTIVE.

You must configure the agent hdlPath to have the full path of the testbench module that correspond to the specific agent. Refer to example shown in the `cdnApbUvmUserSve.sv` file.

Refer to the CDN_APB VIP UVM agent example at: `$CDN_VIP_ROOT/tools/denali/ddva-pi/sv/uvm/cdn_apb/examples/cdnApbUvmUserAgent.sv.`

8.3.3.2. Monitor

The monitor is a passive entity that samples DUT signals but does not drive them. The monitor collects transactions, extracts events, performs checking and coverage and in general provides information about the device activity.

The monitor instantiates coverage (of type cdnApbUvmCoverage with reference name coverModel), the coverage is instantiated only when coverageEnable is set to 1.

The CDN_APB VIP monitor is of type cdnApbUvmMonitor and inherits from uvm_monitor. It contains the following analysis ports and events.

Note that the relevant analysis port/event is triggered when the equivalent callback is called by the model and only if the callback is enabled. To enable a callback use the Instance's `setCallback()` method. For details on the CDN_APB VIP Callbacks, refer to Chapter 6, *CDN_APB VIP Callbacks*.

8.3.3.2.1. CDN_APB Events and Ports

The following table lists the CDN_APB events and ports used in the UVM layer.

Table 8.3. CDN_APB Events

Event Name	Description
DefaultCbEvent	This event is triggered when the DefaultCbF function in the instance class is called.
ErrorCbEvent	This event is triggered when the ErrorCbF function in the instance class is called if the Error callback is enabled. For more information, refer to the Error callback.

Event Name	Description
ResetStartedCbEvent	This event is triggered when the ResetStartedCbF function in the instance class is called if the ResetStarted callback is enabled. For more information, refer to the ResetStarted callback.
ResetEndedCbEvent	This event is triggered when the ResetEndedCbF function in the instance class is called if the ResetEnded callback is enabled. For more information, refer to the ResetEnded callback.
RetrieveItemCbEvent	This event is triggered when the RetrieveItemCbF function in the instance class is called if the RetrieveItem callback is enabled. For more information, refer to the RetrieveItem callback.
TransferStartedCbEvent	This event is triggered when the TransferStartedCbF function in the instance class is called if the TransferStarted callback is enabled. For more information, refer to the TransferStarted callback.
BeforeSendCbEvent	This event is triggered when the BeforeSendCbF function in the instance class is called if the BeforeSend callback is enabled. For more information, refer to the BeforeSend callback.
MonTransferEndedCbEvent	This event is triggered when the MonTransferEndedCbF function in the instance class is called if the MonTransferEnded callback is enabled. For more information, refer to the MonTransferEnded callback.
DriverTransferEndedCbEvent	This event is triggered when the DriverTransferEndedCbF function in the instance class is called if the DriverTransferEnded callback is enabled. For more information, refer to the DriverTransferEnded callback.
CoverageSampleCbEvent	This event is triggered when the CoverageSampleCbF function in the instance class is called if the CoverageSample callback is enabled. For more information, refer to the CoverageSample callback.

Table 8.4. CDN_APB Ports

Port Name	Description
DefaultCbPort	This port is written when the DefaultCbF function in the instance class is called.
ErrorCbPort	This port is written when the ErrorCbF function in the instance class is called if the Error callback is enabled. For more information, refer to the Error callback.
ResetStartedCbPort	This port is written when the ResetStartedCbF function in the instance class is called if the ResetStarted callback is enabled. For more information, refer to the ResetStarted callback.
ResetEndedCbPort	This port is written when the ResetEndedCbF function in the instance class is called if the ResetEnded callback is enabled. For more information, refer to the ResetEnded callback.

Port Name	Description
RetrieveItemCbPort	This port is written when the RetrieveItemCbF function in the instance class is called if the RetrieveItem callback is enabled. For more information, refer to the RetrieveItem callback.
TransferStartedCbPort	This port is written when the TransferStartedCbF function in the instance class is called if the TransferStarted callback is enabled. For more information, refer to the TransferStarted callback.
BeforeSendCbPort	This port is written when the BeforeSendCbF function in the instance class is called if the BeforeSend callback is enabled. For more information, refer to the BeforeSend callback.
MonTransferEndedCbPort	This port is written when the MonTransferEndedCbF function in the instance class is called if the MonTransferEnded callback is enabled. For more information, refer to the MonTransferEnded callback.
DriverTransferEndedCbPort	This port is written when the DriverTransferEndedCbF function in the instance class is called if the DriverTransferEnded callback is enabled. For more information, refer to the DriverTransferEnded callback.
CoverageSampleCbPort	This port is written when the CoverageSampleCbF function in the instance class is called if the CoverageSample callback is enabled. For more information, refer to the CoverageSample callback.

For more information on how to use the CDN_APB monitor, refer to the CDN_APB VIP UVM example at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples`.

8.3.3.3. Driver

The Driver is a verification component that takes items from the Sequencer and drive them to the DUT.

The CDN_APB VIP UVM driver is of type cdnApbUvmDriver and inherits from uvm_driver. It includes the predefined implementation of the `run_phase()` method. For details about the Cadence UVM driver implementation and predefined methods refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/cdnApbUvmDriver.sv`

For details on how to use the CDN_APB VIP UVM driver, refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserDriver.sv.`

The CDN_APB VIP UVM driver is instantiated only under UVM_ACTIVE agent.

8.3.3.4. Sequencer

The Sequencer is an advanced stimulus generator that controls the items provided to the driver for execution.

The CDN_APB VIP UVM sequencer is of type cdnApbUvmSequencer and inherits from uvm_sequencer. The CDN_APB VIP UVM sequencer's sequence item is of type: denaliCdn_apbTransaction.

Refer to section on Transaction Interface in Section 8.2.1, "*Transaction Interface*" for more information on the CDN_APB VIP Transaction Interface.

8.3.3.5. Sequence

A sequence is a stream of data items generated and sent to the DUT. The sequence represents a scenario.

The CDN_APB VIP UVM Sequence is of type cdnApbUvmSequence and inherits from uvm_sequence. The CDN_APB VIP UVM sequence's data item is of type `denaliApbTransaction`.

Refer to Section 8.2.1, "*Transaction Interface*" for more information.

For details on how to use CDN_APB VIP UVM Sequence refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserSeqLib.sv` and `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserVirtualSeqLib.sv.`

8.3.3.6. Transaction

The CDN_APB VIP contains a UVM compliant data item `denaliCdn_apbTransaction` class, which models the data items required to generate the protocol-specific traffic and test scenarios. This class is extended from the `uvm_sequence_item` base class to facilitate the use of UVM sequencers to generate protocol traffic.

For details on how to use CDN_APB VIP UVM transactions, refer to `$CDN_VIP_ROOT/tools/ denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserSeqLib.sv` and `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserVirtualSeqLib.sv`.

8.3.3.7. CDN_APB Instance

The CDN_APB VIP UVM agent contains a reference to cdnApbUvmInstance which inherits from denaliCdn_apbInstance class that provides an interface to access the CDN_APB VIP model. For example you can activate the model callbacks using the instance's `setCallback()` method.

For more information about denaliCdn_apbInstance, refer to Section 8.2.1.1, "*Instance Class*".

For details on how to trigger callbacks using the denaliCdn_apbInstance instance, refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_apb/examples/cdnApbUvmUserEnv.sv`.

Note that setting the model callbacks, using `setCallback()` method (Section 8.2.1.1.1.2.5, *"set-Callback()"*), enables the monitor equivalent analysis ports and events. For details on the monitor analysis ports and events, refer to Section 8.3.3.2, *"Monitor"*.

For details on CDN_APB VIP Callbacks, refer to Chapter 6, *CDN_APB VIP Callbacks*.

8.3.3.8. Memory Instance

The CDN_APB VIP UVM agent contains a reference to cdnApbUvmMemInstance which inherits from denaliMemInstance class that provides an interface to access the CDN_APB Verification IP model configuration and status registers. For more information about the control and status registers of the CDN_APB VIP, refer to Chapter 4, *CDN_APB VIP Model Configuration*.

8.3.3.9. Coverage

The CDN_APB VIP UVM monitor contains a reference to cdnApbUvmCoverage which inherits form denaliCdn_apbCoverageInstance class that contains CDN_APB VIP coverage definitions.

For more information about denaliCdn_apbCoverageInstance, refer to Section 8.2.2, *"Coverage Interface / CDN_APB VIP Coverage"*.

8.3.4. Sequence-Related Features

8.3.4.1. Sending Processed Data Back to Sequence

In some sequences, a generated value depends on the response to previously generated data. By default, the driver is non-blocking, which means that the `item_done` is called at the same cycle the transaction is send to driver using one of the `uvm_do` macros. This causes the `uvm_do` operation to finish.

Sequences can wait for the transmission to be completed using the UVM sequence task `get_response` method. Refer to the example below:

```
    `uvm_do_with(tr,
{…
})
//Wait till transaction is transmitted to DUT
get_response(response,tr.get_transaction_id());
```

8.3.4.2. Modify Transaction Sequence

This sequence enables you to modify transaction that is already in process. It is very useful specially for error injection and responses modification.

The `cdnApbUvmModifyTransactionSequence` encapsulates the logic of the scenario and the logic of the modification in one class. The scenario logic is available in the sequence `body()` task.

The modification logic is available in the sequence `modifyTransaction()` function.

To use the `cdnApbUvmModifyTransactionSequence` perform the following steps:

1. Create your own sequence class which extends from the `cdnApbUvmModifyTransaction-Sequence`.
2. Describe the scenario you want to perform in the sequence `body()` task.
3. Describe the errors/transaction modification you want to perform in the sequence predefined `modifyTransaction` function.

From the time when the `cdnApbModifyTransactionSequence` sequence is started till the sequence ends, the `modifyTransaction` function will be called to any transaction being transmitted by the model.

By default, only transactions with the same sequence id will trigger the `modifyTransaction` functions.

8.3.4.2.1. CDN_APB Modify Sequence Triggering Callbacks

The `modifyTransaction` function is called at the DENALI_CDN_APB_CB_BeforeSend callback.

8.3.5. Error Reporting and Control

8.3.5.1. Changing Message Severity using UVM

All errors from the VIP are reported by the monitor. You can control the severity of an error message using UVM functions such as `set_report_severity_id_override`. For example, if an item should not be reported as an *Error*, you can convert its severity from *Error* to *Info* in the testbench using a function such as the following:

```
agent.monitor.set_report_severity_id_override(UVM_ERROR," <Some Protocol Error value>",UVM_WARNING);
```

8.3.6. The UVM Layer File Structure

You can find the CDN_APB VIP UVM related files under `$DENALI/ddvapi/sv/uvm/cdn_apb`. The following table lists and describes UVM files.

Table 8.5. UVM Files

Filename	Description
cdnApbUvmTop.sv	This file defines the cdnApbUvm package and includes all the CDN_APB UVM Layer files. To have CDN_APB UVM Layer you must compile this file and import the cdnApbUvm package
cdnApbUvmAgent.sv	This file defines the cdnApbUvmAgent class.
cdnApbUvmMonitor.sv	This file defines the cdnApbUvmMonitor class, that includes all available events and analysis ports.

Filename	Description
cdnApbUvmDriver.sv	This file defines the cdnApbUvmDriver class, that includes the default implementation of the driver run_phase method.
cdnApbUvmSequencer.sv	This file defines the cdnApbUvmSequencer.
cdnApbUvmSequence.sv	This file defines the cdnApbUvmSequence.
cdnApbUvmInstance.sv	This file defines the cdnApbUvmInstance, that includes triggering and writing the monitor events and analysis ports. For details on Instance Class, refer to Section 8.2.1.1, *"Instance Class"*.
cdnApbUvmMemInstance.sv	This file defines the cdnApbUvmMemInstance, that includes predefined methods for reading from registers, writing to registers and implementing memory callbacks. For details on registers, refer to the chapter on Registers.
cdnApbUvmCoverage.sv	This file defines the cdnApbUvmCoverage, that includes coverage definitions and collection. For details on Coverage class refer to Section 8.2.2, *"Coverage Interface / CDN_APB VIP Coverage"*.

8.3.7. Generating a Test Case

To create a test case:

1. Refer to Chapter 3, *PureView Graphical Tool*.
2. Create the top level and instantiate the CDN_APB VIP and DUT models in it.
3. Set the required `.denalirc` variables.
4. Create the testbench and add transactions to the user queue of the host model.

The CDN_APB VIP UVM example illustrates how to build UVM compliant test environment CDN_APB models and classes that are part of the VIPCAT installation.

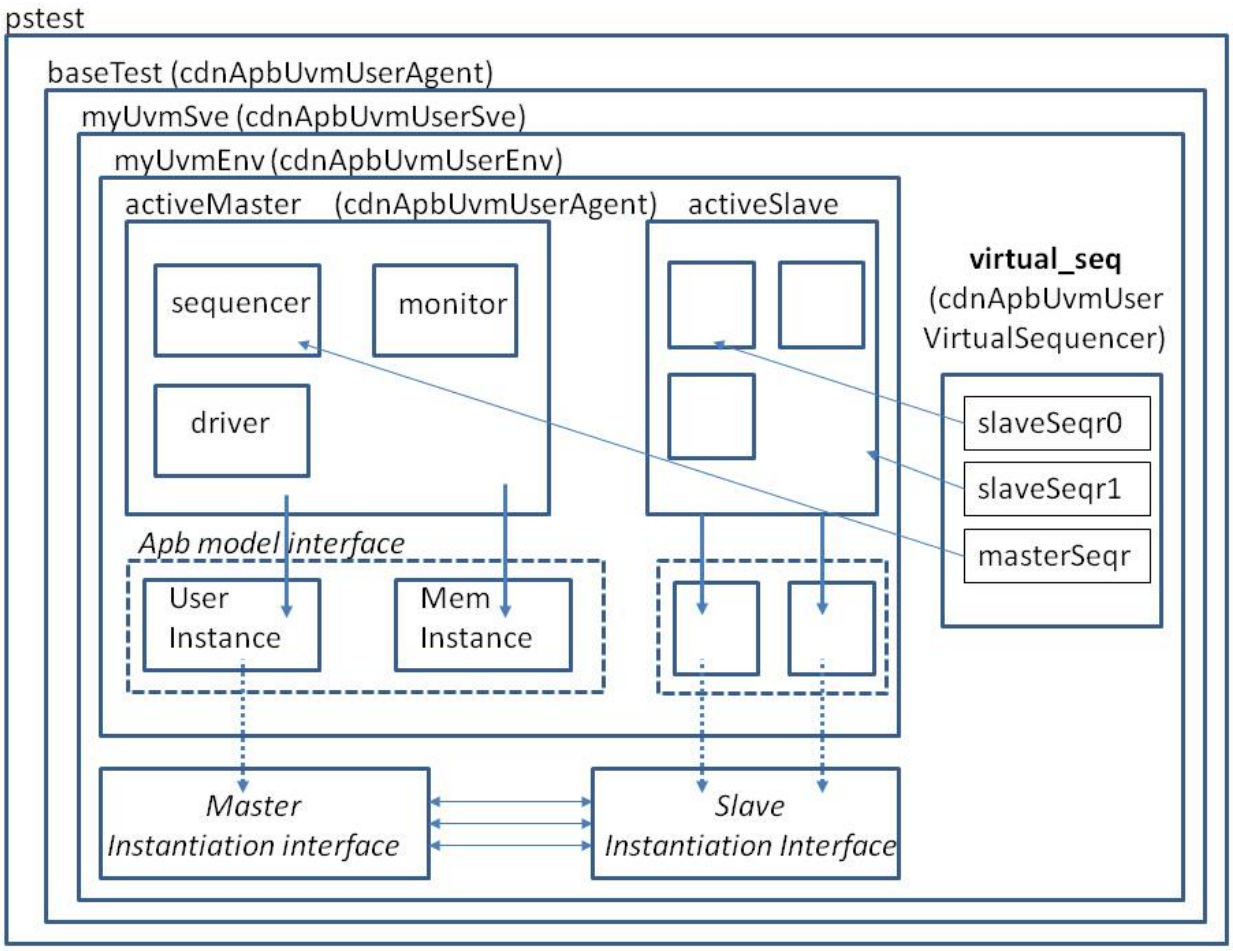
8.3.8. Example Testcase

You can access the testbench example located at `<VIPCAT>/tools/denali/ddvapi/sv/uvm/cdn_apb/examples`.

This example contains the verification environment with instantiations of three agents, SVE (System Verification Environment) that instantiates the environment and a virtual sequencer, and a test that instantiates the SVE.

The following diagram illustrates the CDN_APB VIP UVM example architecture.

Figure 8.2. Example Testcase Architecture



The following table describes the testcase example files.

Table 8.6. Testcase Example Files

File Name	Description
cdnApbUvmUserTb.sv	This file creates testbench, including model and instantiation code.
cdnCApbUvmUserTest.sv	This file includes the definition of the base test class. This test: - Creates the CDN_APB VIP SVE. - Sets the default sequence for the virtual sequencer.

File Name	Description
	- Sets the logs verbosity levels.
cdnApbUvmUserSve.sv	This file includes the definition of the CDN_APB VIP UVM SVE class. This code does the following: - Instantiates the CDN_APB VIP UVM SVE. - Instantiates the virtual sequencer. - Overrides the types of all the UVM layer base components to the user-specific classes - Sets the agents hdlPath to the testbench module instantiations. - Connects the virtual sequencer pointers to the agent's sequencers. - Instantiates a coverage model by setting the relevant agent monitor coverageEnable bit to TRUE using `set_config_int()`.
cdnApbUvmUserEnv.sv	This file includes the definition of the user-specific CDN_APB environment. This code does the following: - Instantiates agents. In this example, activeMaster is considered as DUT. - Activates each agent's callbacks.
cdnApbUvmUserAgent.sv	This file includes the definition of the user-specific CDN_APB agent. In this example, there is no specific implementation added to the CDN_APB agent base class.
cdnApbUvmUserMonitor.sv	This file includes the definition of the user-specific CDN_APB monitor. This example: - Connects the coverage model analysis important to the monitor relevant analysis ports.
cdnApbUvmUserDriver.sv	This file includes the definition of the user-specific CDN_APB driver.

File Name	Description
cdnApbUvmUserSequencer.sv	This file includes the definition of the user-specific CDN_APB sequencer. The user sequencer in the example does not have any specific functionality.
cdnApbUvmUserSeqLib.sv	This file includes an example of the user-specific sequence library.
cdnApbUvmUserVirtualSeuquecer.sv	This file includes an example of user implementation of the virtual sequencer. The virtual sequencer in this example has pointers to the agent's sequencers.
cdnApbUvmUserVirtualSeqLib.sv	This file includes an example of a user virtual sequence library.

Chapter 9. Frequently Asked Questions

9.1. Generic FAQs

9.1.1. How programmable is CDN_APB VIP? Is there a list of the programmable fields available?

The models are extremely programmable. The CDN_APB VIP provides a configuration file called SOMA that can be used to specify the device in terms of configuration and feature set. Additionally, the CDN_APB VIP models allow dynamic changes or updates to the complete protocol-specific configuration space (if any) and CDN_APB VIP specific configuration registers.

9.1.2. How can I change the pin names in CDN_APB VIP model?

You can change pin names by using the PureView user interface. Regenerate the model instantiation interface. Do not manually edit the instantition code or the SOMA file. Both should be edited and saved using PureView.

9.1.3. What is the difference between .spc and .soma files?

The `.spc` and `.soma` files contain the same information, in a different format. The `.soma` file is in compressed XML format and the `.spc` file is in ASCII text format. Cadence recommends that you save the files from PureView in the XML syntax.

9.1.4. How can I disable the `timescale directive in SystemVerilog?

You can use the following macros to disable `timescale directive in the VIP interface:

Table 9.1. Macros to Disable `timescale Directive

Filename	Macro Name
denaliCdn_apb.sv	DENALI_SV_CDN_APB_NO_TIMESCALE
denaliMem.sv file	DENALI_SV_MEM_NO_TIMESCALE
Interface files	DENALI_SV_NO_TIMESCALE

You can use the `DENALI_SV_NO_PKG` macro to disable interface files declaration as package.

9.1.5. How can I print a message in hexadecimal format?

You can print a message in a hexadecimal format by adding `set_config(print, radix, hex)` command to `SPECMAN_PRE_COMMANDS` as shown below:

```
setenv SPECMAN_PRE_COMMANDS "set_config(print, radix, hex);"
```

9.1.6. Is it ok to use double slashes (//) in paths for executing Specman commands?

When using the scripts: `cdn_vip_env.[c]sh`, `run_vcs_sv.sh`, `run_mti_sv.sh`, and `run_nc_sv.sh`, as well as when executing Specman commands directly, do not pass a path that contains double slashes because Specman will treat everything followed by the double slashes as a comment.

For example, the following command will fail:

```
sn_compile.sh -32 -shlib -exe -o uvc /path/to/some/top/e//file/ -enable_DAC
```

Error signatures may appear as:

```
1. *** Error: No match for file '/path/to/some/top/e.e' in command
(SPECMAN_PRE_COMMANDS)

2. *** Error: In preprocessing: Input is empty. Maybe no opening <'
and closing '> were found at line 1 in @...
```

9.1.7. What should the LD_LIBRARY_PATH be set to?

You must set the `LD_LIBRARY_PATH` to the following:

```
setenv LD_LIBRARY_PATH ${DENALI}/verilog:${DENALI}/lib:${DE-
NALI}/../lib:${DENALI }/../psui/lib:$LD_LIBRARY_PATH
```

Note

When you use Specman along with the CDN_APB, Symbol resolutions happen through the `$INSTALL_ROOT/specman/libsn.so`. This path should be part of the `LD_LIBRARY_PATH` or should be passed through the `-pli` option.

9.1.8. Is there anything I need to do before compiling the C-libraries and system Verilog files in the run script?

Compile the `denaliCdn_apbSvIf.c` file into the DPI libraries before the compilation of corresponding SV files, which call functions like `denaliCdn_apbInit`.

9.1.9. Where does the DENALI variable point to?

`$DENALI` points to the `<package_location>/tools/denali` directory.

9.1.10. How do I change SPECMAN_HOME?

When you want to run a simulation with a different Specman version, SPECMAN_HOME can be changed in the following manner:

```
setenv SPECMAN_HOME <Package Installation Directory>/tools/specman
setenv PATH <Package Installation Directory>/tools/specman:$PATH
```

9.1.11. What should I do when the message "Couldn't create a legal Pkt transaction from input. Item is dropped." comes during simulation ?

This message comes when a randomization failure happens while generating a packet. To aid debugging the randomization failure in such cases, the following commands should be set in succession on the Specman command prompt before running the simulation:

- break on error
- break on gen error

When you run the simulation with these commands set, the debugger will show the exact contradicting constraints, and the constraints can be modified accordingly.

Chapter 10. Troubleshooting

10.1. Generic Troubleshooting

10.1.1. Specman license attempted a checkout

When using Specman installed under VIPCAT, you must set the environment variable `UVC_MSI_MODE` in your environment/run scripts/setup scripts. Not setting this variable will cause Specman to invoke a standard Specman license.

For example:

```
setenv UVC_MSI_MODE 1
```

You can find more details in the *VIP Catalog User Guide*, section *Using the UVC Virtual Machine*.

10.1.2. CDN_PSIF_ASSERT_0031

Error signature:

```
Generating the test with IntelliGen using seed 1...
                *** Error: CDN_PSIF_ASSERT_0031
Internal error or configuration problem at line 503 in encrypted module cdn_psif_e_utils_ext

No PS memories alocated. Check your setup and configuration
Failed condition: (FALSE)
```

Problem:

The stub file `<CDN_VIP_ROOT>/tools/psui/lib/[64bit/][vcs|mti]_psui.sv` should be the last SystemVerilog file passed to the vcs/vlog command. If the order has been modified, this error is raised.

Solution:

If the order of SV files cannot be changed, do the following to resolve the issue:

1. Remove "test;" from the SPECMAN_PRE_COMMANDS environment variable.
2. Do **one** of the following
   - Invoke "test" from the simulator by typing "sn test".
   - Add the following code into an initial block:

     ```
     initial
        begin
           $sn("test");
     end
     ```

     This code has to happen during time 0, before any register write (before the instantiation).

Note

The "test;" command must either be specified in SPECMAN_PRE_COMMANDS or invoked by the simulator. When modifying the default behavior, please make sure the test command is executed only once.

A missing test command will result the following error:

```
*** Error: Specman cannot start running: 'test' command not issued yet
```

10.1.3. Library Error During Simulation

When using VCS or MTI with VIPCAT, you might run into the following error

```
*** Error: Cannot open library
Failed to open `$CDN_VIP_ROOT/tools.lnx86/ucd/lib/libucdb.so' shared library
Operating System error:
<path to gcc>/gcc/*/lib/libstdc++.so.6: version `GLIBCXX_3.4.*' not found (required by $CDN_VIP_ROOT/
tools.lnx86/ucd/lib/libucdb.so)
```

Problem

Both VIPCAT and the simulator use libstdc++ which is loaded just once. If the simulator loads the libstdc++ before VIPCAT does, there will be no other load of the same library with the same major version. If the libucdb.so used for the simulation was compiled with an earlier gcc version than the one used to compile the VIPCAT libraries, this error is issued.

Workaround

As a first step, make sure your LD_LIBRARY_PATH includes the following path in the beginning: `$CDN_VIP_ROOT/tools/lib[64bit]`.

When executing `cdn_vip_env.[c]sh`, `run_vcs_sv.sh`, `run_mti_sv.sh` or `run_nc_sv.sh` with the `-setup` flag enabled, the `cdn_vip_setup_runme.sh/runme.sh` file created will include this step.

The above workaround is expected to resolve the issue, unless advanced environment settings were define to change/override LD_LIBRARY_PATH settings. In that case:

1. Check whether you have a wrapper script for VCS or MTI that changes the LD_LIBRARY_PATH.

2. Check `/etc/ld.so.conf` file/imports.

3. Check whether the `rpath` flag is set in your gcc configuration. `rpath` is a linker flag for searching dynamic libraries that have precedence over the `LD_LIBRARY_FLAG` variable.

4. Make sure the option `-noautoldlibpath` is not passed to MTI (since the default of the vsim command WILL change `LD_LIBRARY_FLAG`).

Which gcc version should you use if no standard location contains an advanced enough version? The safest thing to do is to:

1. Pick up gcc from VIPCAT (located under `$CDN_VIP_ROOT/tools.lnx86/systemc/gcc/bin[/64bit]`).

2. Set the environment variable NCROOT_OVERRIDE in your env to point to the vipcat root.