cadence

Cadence AXI VIP User Guide

Product Version 11.3
June 2016

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

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

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

The AXI 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 AXI VIP is built on top of the MMAV™ architecture to ensure high quality, high performance, and seamless EDA integration.

Using AXI 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 AXI VIP help you verify your design by generating warnings and errors when the protocol violations occur.

Note

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

1. About This Manual

This manual describes the AXI 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 AXI 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 AXI 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 Case to Cadence Support. When you file a customer support case, 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. Customer Cases

Cases are your way of giving feedback, asking questions, getting solutions, and reporting problems. Unless told otherwise, Cadence support staff will respond to your case. 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 cases using Cadence Online Support. With Cadence Online Support you can also track your open cases.

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 VIP 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 Case* under the *Cases* 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 request.

3.2.1. Creating Group Privileges in Cadence Online Support

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

To create group privileges in Cadence Online Support:

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

2. Select *Verification IP* in the *Product list* box.

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, and click *Continue*.

4. In the *People to notify* upon Case 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 Case to complete the Case.

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 AXI 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:

- `$CDN_VIP_ROOT/tools/denali/doc/cdn_axi/axi_uvm_sv_ref/index.html`
- `$CDN_VIP_ROOT/tools/denali/doc/cdn_axi/axi_ovm_sv_ref/index.html`

Chapter 1. CDN_AXI VIP Product Overview

Note

By default, the ACE Coherency Extensions are not in effect. Refer to chapter on *CDN_AXI VIP Testbench Integration* for instructions on how to compile the AXI user interface for ACE.

1.1. Features

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

Key features of the CDN_AXI VIP include:

- CDN_AXI specification modeling on two levels:
  - Transfer
  - Transaction/Burst
- Two main devices types:
  - Master -- issues read and write transactions, accepts snoop transactions, and stores cache data.
  - Slave -- responds to read/write transactions and acts as a dummy interconnect and can send snoop transactions to the cache master.
- Model usage in either active or passive mode.

  The passive mode provides the DUT shadow reference model that compares DUT responses to expected responses.
- Independent state machines for pin-level valid/ready signal protocol for each channel - *read address*, *write address*, *read data*, *write data*, and *write response*.
- Transfer-level state machines monitoring the latency and consistency of read and write transactions sequences of transfers, as well as snoop transactions. For example, `WriteAddress`, multiple `WriteTransfers`, and `WriteResponse`, as referred to in the CDN_AXI VIP model.
- Automatic segmentation of a high-level `Read`, `Write`, and `Snoop` transaction into a sequence of transfers. For example, `WriteAddress`, multiple `WriteTransfers`, and `WriteResponse`.
- Automatic retrieval of transfers from pin changes and high-level transactions from transfers.

- Multiple interleaving transactions.

  Note

  There is no write interleaving (as per the *Specification)*.

- Pipelined operation.
- Out-of-order transaction support.
- Automatic reordering transactions for memory coherency that maximizies performance.
- Customizable address and data width.
- Automatic write strobe generation for non-aligned transactions.
- Exclusive access support.
- Customizable error reporting.

When the CDN_AXI VIP model functions in active mode (Bus Functional Model), it interacts with complementary devices, and issues and responds to the commands. When the CDN_AXI VIP model functions in passive mode (Monitor), it attaches 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_AXI VIP provides powerful callback capabilities for the user defined testbenches to enable full control over all the transactions that take place within the CDN_AXI model.

1.2. Model Types

There are several CDN_AXI VIP model types. You can use each of these device types either in an active mode (BFM) or passive mode (Monitor).

1.2.1. 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 (Device Under Test)
- Parse DUT responses and snoop responses
- Convert snoop transactions signals from a DUT interconnect into high-level snoop transactions
- Generate automatic snoop responses according to the state of the cache

1.2.2. 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:

- Create snoop transactions
- Convert signals from a DUT master into high-level transactions
- Generate automatic responses
- Transmit these responses back to the master DUT
- Drive high-level snoop transactions to the DUT cached master

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

You can instantiate the CDN_AXI VIP passive model as a shadow model of the DUT. To do this, prepare the SOMA file and fill the model's memory so that the model behaves like DUT.

All pins of the CDN_AXI VIP passive model are input pins. The slave monitor receives transactions from the master driving DUT, generates the expecting response, and compares this response with the actual response from DUT received by the monitor through the wires.

Monitor generates all error messages of the CDN_AXI VIP active model (transaction checking and timeouts). In addition, the monitor generates errors about transaction mismatch with the expected transaction, indicating that some expected transaction is missing, and that the DUT slave response is unexpected.

Chapter 2. Getting Started

2.1. Integration Check List

This section provides step by step instructions on how to integrate the VIP and run it for the *first* time. Skip the steps that are not relevant for you.

1. Download the latest VIPCAT version.

   Refer to the *README_VIPCAT_Release_Info.pdf* document in the VIPCAT113 release area on downloads.cadence.com for details on how to install the release, the VIP availability matrixes, platform support, release dependencies, and licensing information.

2. Set up your environment to simulate the VIP by executing the scripts provided with the VIPCAT release.

   a. Set `CDN_VIP_ROOT` to point to the root directory of the VIPCAT installation.

   b. Execute `$CDN_VIP_ROOT/bin/cdn_vip_setup_env` with the appropriate arguments. This generates a shell script with the necessary commands to set up your environment. The generated shell script is named `cdn_vip_env_<...>.[c]sh`.

      Note

      - `cdn_vip_setup_env -h` provides information on the arguments to this script.
      - The simulator executable (ncsim, vcs or vlog) must be available in your path (or the simulator installation directory must be explicitly specified with the `-sim_root` option to `cdn_vip_setup_env`).
      - For IUS users only: You must specify the `-install` and `-cdn_vip_lib` options for `cdn_vip_setup_env`. The argument for the `-cdn_vip_lib` option must be a user-writeable directory. `cdn_vip_setup_env` compiles the libraries necessary to run the VIP with IUS into this directory.
      - If your environment is based on C shell (csh, tcsh, etc.), the `-csh` option must be specified for `cdn_vip_setup_env`. By default, commands to set up the environment are generated for Bourne shell (sh).

   c. Execute the generated script.

      C-shell users: `source cdn_vip_env_<...>.csh`
      Bourne-shell users: `. cdn_vip_env_<...>.sh`

3. Set up and run the example provided in the release.

For details on the command line for each simulator, refer to the following:

- UVM

  - NCSIM (Section 8.3.2.1, “*NCSim*”)

  - VCS (Section 8.3.2.2, “*VCS*”)

  - MTI (Section 8.3.2.3, “*MTI*”)

- SV non UVM

  - NCSIM (Section 8.1.2, “*NCSim*”)

  - VCS (Section 8.1.3, “*VCS*”)

  - MTI (Section 8.1.4, “*MTI*”)

4. Create a SOMA and an HDL instantiation interface for each component type. A SOMA file represents the component configuration.

   Refer to Section 2.2, “*Creating SOMA and HDL Instantiation Interface for CDN_AXI VIP*”.

5. Create a testbench for your environment.

   a. Create the testbench file. Refer to Section 2.3, “*Creating a Testbench*”.

   b. Connect the components together. Refer to Section 2.4, “*Connecting the Components*”.

   c. Configure the components. Refer to Section 2.5, “*Configuring the Components*”.

6. Create setup, compilation, and run scripts for your environment. Refer to Section 8.1.1, “*VIP Scripts*”. If your simulator is NCSIM, you can use the **irun** invocation tool to handle the details of incorporating VIP models. For details, refer to Section 8.1.2.3, “*Using irun* ”

7. Adjust the memory segments according to the design (check that the segments definition of agents in the same interface matches). Refer to Section 4.1.2.4, “*Defining Memory Segments*”.

8. Write your first test. Refer to Section 2.6, “*Writing a Simple Test Scenario*”.

   Once you have written your first test, you can you can start generating transactions and test verification flow. Refer to Section 4.3, “*Transactions*”, Section 7.1, “*Test Flow*”, respectively. For details on how to debug, refer to Section 6.5, “*Debugging the Simulation*”.

9. Write more advanced tests. Refer to Chapter 7, *CDN_AXI Verification Test Scenarios*.

a. Add specific constraints to the transaction. Make sure you limit the `IdTag` and `address` according to the corresponding signals size. See the `class myTransaction` definition in the example of section Section 2.6, "*Writing a Simple Test Scenario*".

   You can find more explanation about the VIP SV constraint in Section 7.4, "*Built-in SV Constraints*".

b. For better control over the model, use the VIP callbacks. For details on how to use these callbacks, refer to Chapter 5, *AXI VIP Callbacks*.

10. If you encounter any issue, first check Chapter 9, *Frequently Asked Questions* and Chapter 10, *Troubleshooting*.

2.2. Creating SOMA and HDL Instantiation Interface for CDN_AXI VIP

A SOMA file and an HDL instantiation interface must be created for each component type in the simulation.

Note

A SOMA file and an HDL instantiation interface are needed for each type and not for each instance.

The SOMA file is a XML file that is edited using the Pureview GUI tool. The SOMA file defines a specific component type configuration; you should configure it to match a DUT port. The HDL instantiation interface is a **.v* file that is created using the Pureview GUI tool.

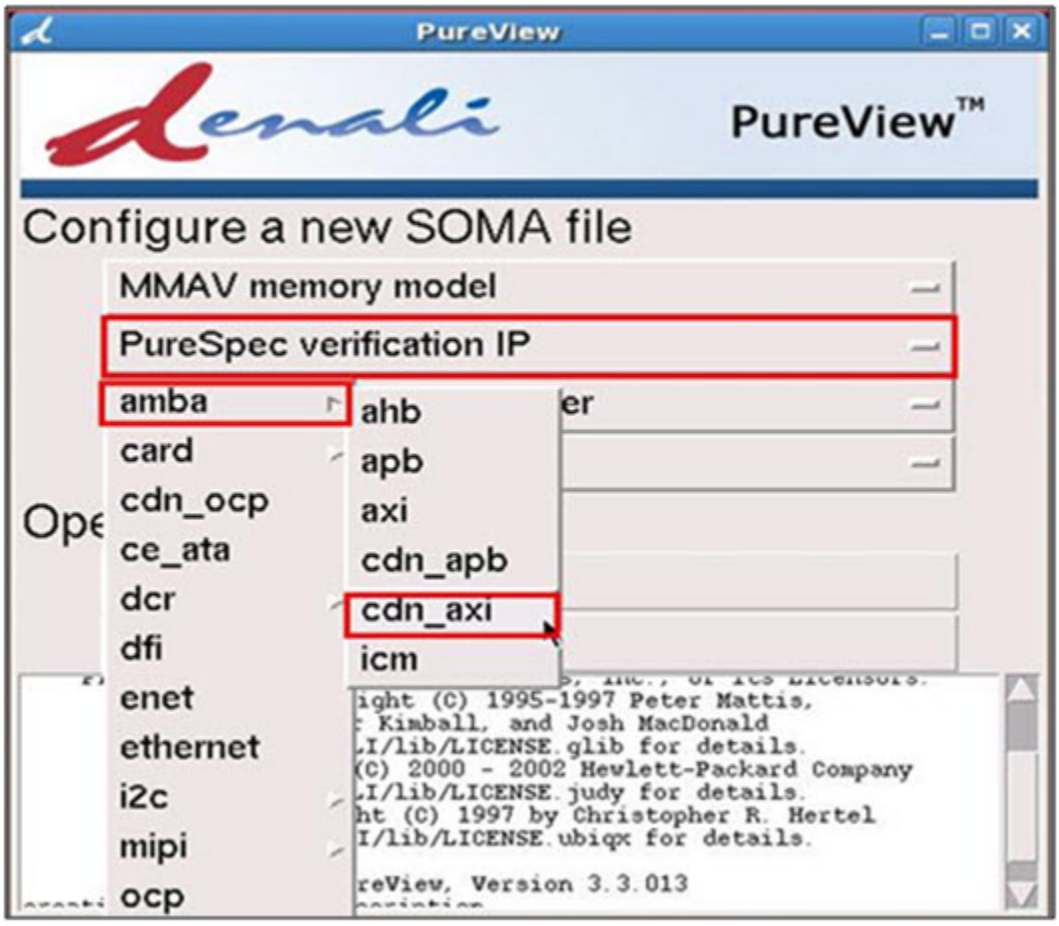
Every HDL instantiation interface has a corresponding SOMA file associated with it. The HDL instantiation interface contains a module with the interface between the testbench and the VIP.

In this example, we will show how to create SOMA and HDL instantiation interface for two component types:

- Active master
- Active slave

1. Open Pureview
2. Choose **PureSpec Verification IP** >> **amba** >> **cdn_axi,** as shown in the figure below:

Figure 2.1. PureView Opening Window

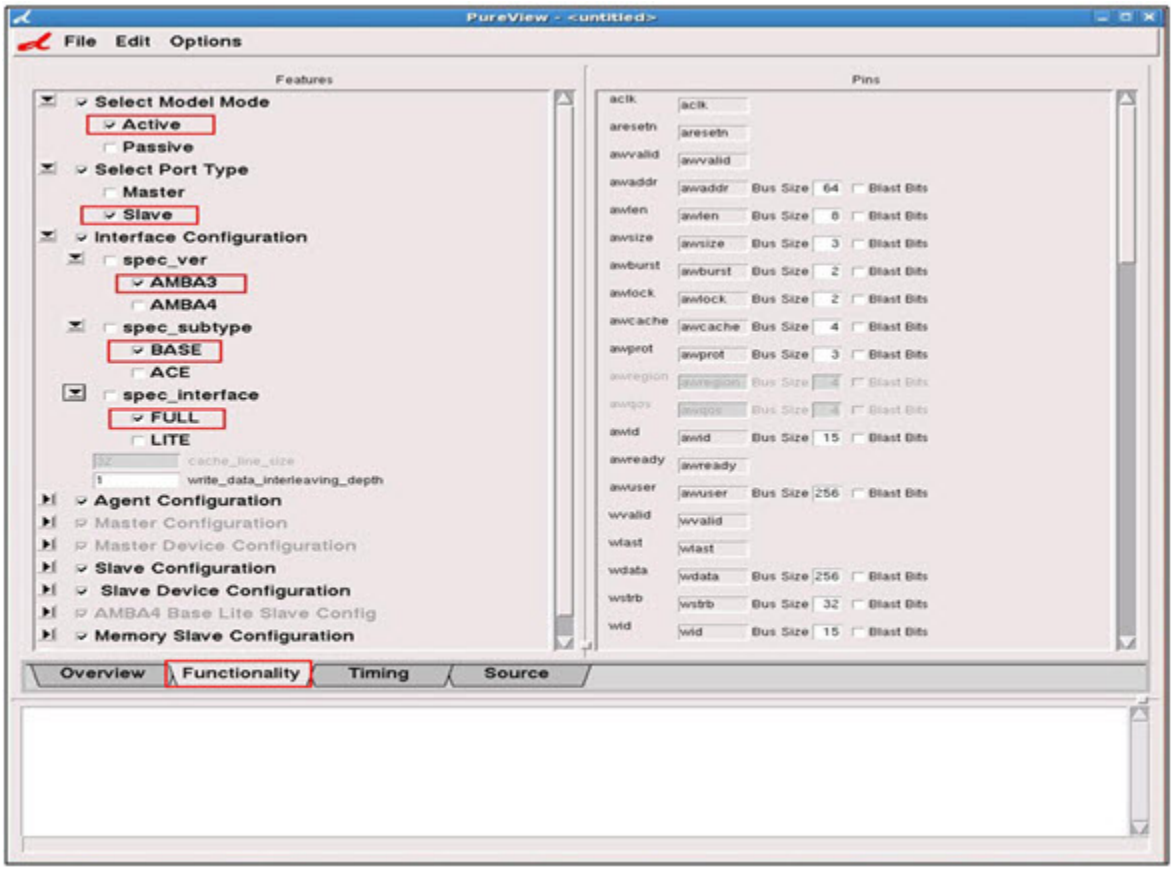


3. The PureView window opens.

4. Choose the **Functionality** tab at the bottom of the window to view the default configuration set for AXI Slave.

5. Modify the default configuration parameters to AXI3 as shown below in the **Features** pane.

Figure 2.2. PureView - Functionality Tab



6. **Select Model Mode** >> **Active**

7. **Select Port Type** >> **Slave**

8. **Interface Configuration** >> **spec_ver** >> **AMBA3**

9. **Interface Configuration** >> **spec_subtype** >> **BASE**

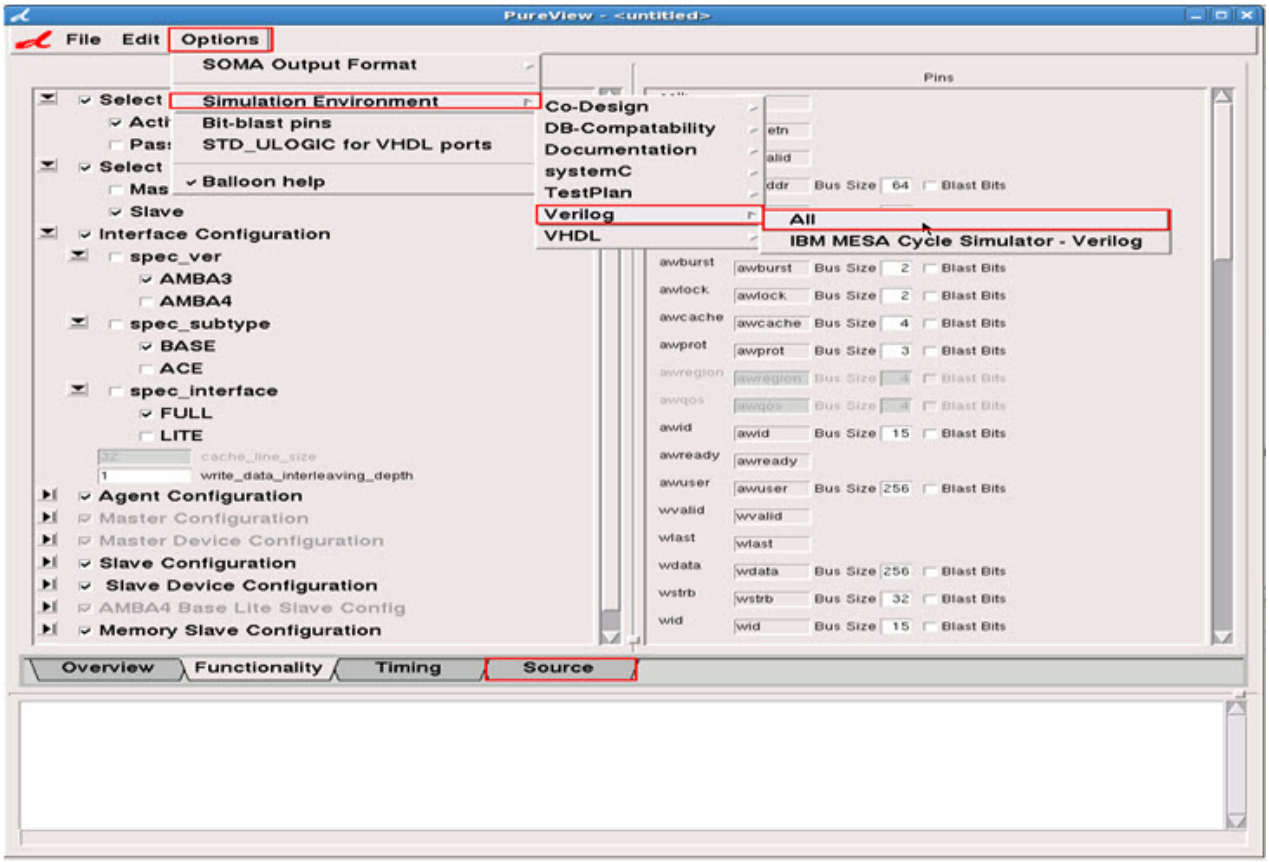
10. **Interface Configuration** >> **spec_interface** >> **FULL**

Once all the above parameters are set, you can control other configuration including `signals names` and `width` in the **Pins** pane.

1. Navigate using **File** >> **Save as** to save the agent configuration file as `activeSlave.soma` file.

2. Navigate using **Options** >> **Simulation Environment** >> **Verilog** >> **All** to save the interface, as shown in the figure below.

Figure 2.3. PureView - Source Tab



Note

Choose the **Source** tab to review the file that is created when you select **Save Source as**. You can edit the default module name, which is the same as the SOMA file. At the bottom of the file you can also view the module that is created with a path to the associated SOMA file. By default this is an absolute path and you can edit it to be a relative path using environment variables.

3. Repeat the previous steps for each component types that are needed in your environment. In this case create *ActiveMaster.soma* (by Choosing **Select Port Type** >> **Master**) and *ActiveMaster.v* files.

Note

Do not re-open *Pureview* since the default name is that of the last saved file. Do not overwrite the previous files and remember to type the new file name each time.

At this stage, you should have **.soma file* and **.v* files for each component type in your system, as shown in the example below:

- `ActiveSlave.soma` and `ActiveSlave.v`
- `ActiveMaster.soma` and `ActiveMaster.v`

2.3. Creating a Testbench

1. Create a testbench file `tb.sv` and declare top module in it.

```
module axi_top;
endmodule
```

2.4. Connecting the Components

1. Add all the interface signals, which are to be connected in the testbench, to the top module you declared.
   - Clock and reset should be **reg**
   - All the other signals should be **wire**
   - **Tip:** Open one of the HDL instantiation interface files (For example activeSlave.v). Copy the declaration of the interface signals and change them to wire using find and replace.

     Expected *tb.sv*

```
reg aclk;
    reg aresetn;
    wire awvalid;
    wire [31:0] awaddr;
    wire [7:0] awlen;
    wire [2:0] awsize;
    wire [1:0] awburst;
    wire [1:0] awlock;
    wire [3:0] awcache;
    wire [2:0] awprot;
    wire [3:0] awid;
    wire awready;


    wire [31:0] awuser;
    wire wvalid;
    wire wlast;
    wire [31:0] wdata;
    wire [3:0] wstrb;
    wire [3:0] wid;
    wire wready;


    wire [31:0] wuser;
    wire bvalid;


    wire [1:0] bresp;


    wire [3:0] bid;
```

```
    wire bready;
    wire [31:0] buser;


    wire arvalid;
    wire [31:0] araddr;
    wire [7:0] arlen;
    wire [2:0] arsize;
    wire [1:0] arburst;
    wire [1:0] arlock;
    wire [3:0] arcache;
    wire [2:0] arprot;
    wire [3:0] arid;
    wire arready;


    wire [31:0] aruser;
    wire rvalid;


    wire rlast;


    wire [31:0] rdata;


    wire [1:0] rresp;


    wire [3:0] rid;


    wire rready;
                   wire [31:0] ruser;
```

Existing *activeSlave.v*

```
input aclk;
    input aresetn;
    input awvalid;
    input [31:0] awaddr;
    input [7:0] awlen;
    input [2:0] awsize;
    input [1:0] awburst;
    input [1:0] awlock;
    input [3:0] awcache;
    input [2:0] awprot;
    input [3:0] awid;
    output awready;
      reg den_awready;
      assign awready = den_awready;
    input [31:0] awuser;
    input wvalid;
    input wlast;
    input [31:0] wdata;
    input [3:0] wstrb;
    input [3:0] wid;
    output wready;
      reg den_wready;
      assign wready = den_wready;
    input [31:0] wuser;
    output bvalid;
      reg den_bvalid;
      assign bvalid = den_bvalid;
    output [1:0] bresp;
      reg [1:0] den_bresp;
```

```
      assign bresp = den_bresp;
    output [3:0] bid;
      reg [3:0] den_bid;
      assign bid = den_bid;
    input bready;
    output [31:0] buser;
      reg [31:0] den_buser;
      assign buser = den_buser;
    input arvalid;
    input [31:0] araddr;
    input [7:0] arlen;
    input [2:0] arsize;
    input [1:0] arburst;
    input [1:0] arlock;
    input [3:0] arcache;
    input [2:0] arprot;
    input [3:0] arid;
    output aready;
      reg den_aready;
      assign aready = den_aready;
    input [31:0] aruser;
    output rvalid;
      reg den_rvalid;
      assign rvalid = den_rvalid;
    output rlast;
      reg den_rlast;
      assign rlast = den_rlast;
    output [31:0] rdata;
      reg [31:0] den_rdata;
      assign rdata = den_rdata;
    output [1:0] rresp;
      reg [1:0] den_rresp;
      assign rresp = den_rresp;
    output [3:0] rid;
      reg [3:0] den_rid;
      assign rid = den_rid;
    input rready;
    output [31:0] ruser;
      reg [31:0] den_ruser;
      assign ruser = den_ruser
```

2. Add clock block and reset control. For example:

```
Initial
begin
     aclk = 1'b1;
     aresetn = 1'b1;
     #100
     aresetn = 1'b0;
     #500
     aresetn = 1'b1;
end
always #50 aclk = ~aclk;
```

3. Instantiate the code in the top module scope. The module name was determined when you saved the file, but you can set a different name in the **Source** tab in *Pureview*.

   Tip: Add the declaration for the instantiation interface, for example `activeMaster activeMasterDevice()`. Also copy the signals for connection.

   Expected *tb.sv*

```
activeMaster activeMasteDevice(
```

```
    aclk,
    aresetn,
    awvalid,
    awaddr,
    awlen,
    awsize,
    awburst,
    awlock,
    awcache,
    awprot,
    awid,
    awready,
    awuser,
    wvalid,
    wlast,
    wdata,
    wstrb,
    wid,
    wready,
    wuser,
    bvalid,
    bresp,
    bid,
    bready,
    buser,
    arvalid,
    araddr,
    arlen,
    arsize,
    arburst,
    arlock,
    arcache,
    arprot,
    arid,
    arready,
    aruser,
    rvalid,
    rlast,
    rdata,
    rresp,
    rid,
    rready,
    ruser);
```

Existing *activeMaster.v*

```
module activeMaster(
    aclk,
    aresetn,
    awvalid,
    awaddr,
    awlen,
    awsize,
    awburst,
    awlock,
    awcache,
    awprot,
    awid,
    awready,
    awuser,
    wvalid,
    wlast,
    wdata,
    wstrb,
    wid,
    wready,
```

```
   wuser,
   bvalid,
   bresp,
   bid,
   bready,
   buser,
   arvalid,
   araddr,
   arlen,
   arsize,
   arburst,
   arlock,
   arcache,
   arprot,
   arid,
   arready,
   aruser,
   rvalid,
   rlast,
   rdata,
   rresp,
   rid,
   rready,
   ruser);
```

4. Repeat the process for all components.
5. *tb.sv* should now contain:
   - Testbench signals
   - Clock block and reset control
   - Instantiation code for each component bound to testbench signals:

```
module axi_top;

  reg aclk;
  reg aresetn;
  wire awvalid;
  wire [31:0] awaddr;
  wire [7:0] awlen;
  wire [2:0] awsize;
  wire [1:0] awburst;
  wire [1:0] awlock;
  wire [3:0] awcache;
  wire [2:0] awprot;
   wire [3:0] awid;
   wire awready;

   wire [31:0] awuser;
   wire wvalid;
   wire wlast;
   wire [31:0] wdata;
   wire [3:0] wstrb;
   wire [3:0] wid;
   wire wready;

   wire [31:0] wuser;
   wire bvalid;

   wire [1:0] bresp;

   wire [3:0] bid;
```

```
    wire bready;
    wire [31:0] buser;

    wire arvalid;
    wire [31:0] araddr;
    wire [7:0] arlen;
    wire [2:0] arsize;
    wire [1:0] arburst;
    wire [1:0] arlock;
    wire [3:0] arcache;
    wire [2:0] arprot;
    wire [3:0] arid;
    wire arready;

    wire [31:0] aruser;
    wire rvalid;

    wire rlast;

    wire [31:0] rdata;

    wire [1:0] rresp;

    wire [3:0] rid;

    wire rready;
    wire [31:0] ruser;


activeMaster activeMasterDevice(aclk, aresetn, awvalid, awaddr, awlen, awsize, awburst, awlock,
 awcache, awprot, awid, awready, awuser,  wvalid, wlast, wdata, wstrb, wid, wready, wuser,
 bvalid, bresp, bid, bready, buser, arvalid, araddr, arlen, arsize, arburst, arlock, arcache,
 arprot, arid, arready, aruser, rvalid, rlast, rdata,  rresp, rid, rready, ruser);

activeSlave activeSlaveDevice(aclk, aresetn, awvalid, awaddr, awlen, awsize, awburst, awlock,
 awcache, awprot, awid, awready, awuser,  wvalid, wlast, wdata, wstrb, wid, wready, wuser,
 bvalid, bresp, bid, bready, buser, arvalid, araddr, arlen, arsize, arburst, arlock, arcache,
 arprot, arid, arready, aruser, rvalid, rlast, rdata,  rresp, rid, rready, ruser);

initial
 begin
     aclk = 1'b1;
     aresetn = 1'b1;
     #100
     aresetn = 1'b0;
     #500
     aresetn = 1'b1;
end
 always #50 aclk = ~aclk;
endmodule
```

2.5. Configuring the Components

1. For each component, do the following:

   - Import the CDN_AXI VIP package `denaliSvCdn_axi,` which includes the CDN_AXI VIP instance `denaliCdn_axiInstance`.
   - Import the `denaliSvMem` package, which includes utilities for reads/writes, callbacks, memory transactions, SOMA parameter value access, TCL command evaluation, and so on.

Recommendation: Testbench configuration of an instance is largely based on reading and writing to models such as memory model, control registers model, and so on. Each of these models has an instance ID associated with it; therefore you should write code to instantiate these models in your testbench and implement methods to read and write to them. See Section 4.1.1, *"Instantiating the Models in the Testbench"*.

The syntactic patterns for the most-used methods are readily available at `$DENALI/example/cdn_axi/denaliAxiUserInstance.sv` for AXI and in `$DENALI/example/cdn_axi/denaliAceUserInstance.sv` for ACE. If you choose to include this file in your testbench and use the available methods, make sure that the instance name is `axiInstanceTemplate` (extending `denaliCdn_axiInstance`).

Use `axiInstanceTemplate` in this example. The code will look as shown below:

```
package myPackage;
 import DenaliSvCdn_axi::*;
 import DenaliSvMem::*;
`include "denaliAxiUserInstance.sv"
endpackage

module axi_top;

    import myPackage::*;
    import DenaliSvCdn_axi::*;
    import DenaliSvMem::*;


    axiInstanceTemplate activeMasterInstance;
    axiInstanceTemplate activeSlaveInstance;

    initial
    begin
        activeMasterInstance = new ("axi_top.activeMasterDevice");
        activeSlaveInstance = new ("axi_top.activeSlaveDevice");
    end
endmodule
```

2. Add the actual configuration - the minimum is to configure the memory space.

```
package myPackage;
 import DenaliSvCdn_axi::*;
 import DenaliSvMem::*;
`include "denaliAxiUserInstance.sv"
endpackage

module axi_top;

 import myPackage::*;
import DenaliSvCdn_axi::*;
import DenaliSvMem::*;


    axiInstanceTemplate activeMasterInstance;
    axiInstanceTemplate activeSlaveInstance;

    …

    initial
    begin
```

```
        activeMasterInstance = new ("axi_top.activeMasterWrapper");
        activeSlaveInstance = new ("axi_top.activeSlaveWrapper");

// mapMemorySegment is declared in the file denaliAxiUserInstance.sv

        activeMasterInstance.mapMemorySegment(64'h0,64'h3FFF);
        activeSlaveInstance.mapMemorySegment(64'h0,64'h3FFF);
    end
endmodule
```

2.6. Writing a Simple Test Scenario

This section shows how to write and send a transaction. For more test scenarios, refer to Chapter 7, *CDN_AXI Verification Test Scenarios*.

Note

Cadence recommends not using `denaliCdn_axiTransaction` as it is; you must extend it and add your DUT-specific constraints.

```
package myPackage;
import DenaliSvCdn_axi::*;
import DenaliSvMem::*;
`include "denaliAxiUserInstance.sv"
class myTransaction extends denaliCdn_axiTransaction;

 function new();
  super.new();
  // these fields must be set in addition to the SOMA setting
  this.SpecVer = DENALI_CDN_AXI_SPECVERSION_AMBA3;
  this.SpecSubtype = DENALI_CDN_AXI_SPECSUBTYPE_BASE;
  this.SpecInterface = DENALI_CDN_AXI_SPECINTERFACE_FULL;
 endfunction

 constraint user_dut_information {
   (StartAddress >= 'h0) && (StartAddress <= 'h1FF);
  BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
  IdTag < (1 << 3);
  Cacheable == DENALI_CDN_AXI_CACHEMODE_NON_CACHEABLE;
  ModelGeneration == 0;
 }
endclass
endpackage
module axi_top;
import myPackage::*;
import DenaliSvCdn_axi::*;
import DenaliSvMem::*;
axiInstanceTemplate activeMasterInstance;
axiInstanceTemplate activeSlaveInstance;

 reg aclk;
reg aresetn;
wire awvalid;
wire [31:0] awaddr;
wire [7:0] awlen;
wire [2:0] awsize;
wire [1:0] awburst;
wire [1:0] awlock;
wire [3:0] awcache;
wire [2:0] awprot;
```

```
wire [3:0] awid;
wire awready;

wire [31:0] awuser;
wire wvalid;
wire wlast;
wire [31:0] wdata;
wire [3:0] wstrb;
wire [3:0] wid;
wire wready;

wire [31:0] wuser;
wire bvalid;

wire [1:0] bresp;
wire [3:0] bid;

wire bready;
wire [31:0] buser;

wire arvalid;
wire [31:0] araddr;
wire [7:0] arlen;
wire [2:0] arsize;
wire [1:0] arburst;
wire [1:0] arlock;
wire [3:0] arcache;
wire [2:0] arprot;
wire [3:0] arid;
wire arready;

wire [31:0] aruser;
wire rvalid;

wire rlast;
wire [31:0] rdata;

wire [1:0] rresp;
wire [3:0] rid;

wire rready;
wire [31:0] ruser;

activeMaster activeMasterDevice(aclk, aresetn, awvalid, awaddr, awlen, awsize, awburst, awlock,
 awcache, awprot, awid, awready, awuser,  wvalid, wlast, wdata, wstrb, wid, wready, wuser, bvalid,
 bresp, bid, bready, buser, arvalid, araddr, arlen, arsize, arburst, arlock, arcache, arprot, arid,
 arready, aruser, rvalid, rlast, rdata,  rresp, rid, rready, ruser);

activeSlave activeSlaveDevice(aclk, aresetn, awvalid, awaddr, awlen, awsize, awburst, awlock, awcache,
 awprot, awid, awready, awuser,  wvalid, wlast, wdata, wstrb, wid, wready, wuser, bvalid, bresp,
 bid, bready, buser, arvalid, araddr, arlen, arsize, arburst, arlock, arcache, arprot, arid, arready,
 aruser, rvalid, rlast, rdata,  rresp, rid, rready, ruser);

initial
 begin
aclk = 1'b1;
 aresetn = 1'b1;
 #100
aresetn = 1'b0;
#500
aresetn = 1'b1;
end
 always #50 aclk = ~aclk;

myTransaction masterBurst; // transaction with dut specific constraints

integer status;
```

```
initial
begin
activeMasterInstance = new ("axi_top.activeMasterDevice");
         activeSlaveInstance = new ("axi_top.activeSlaveDevice");

// mapMemorySegment is declared in the file denaliAxiUserInstance.sv
         activeMasterInstance.mapMemorySegment(64'h0,64'h3FFF);
         activeSlaveInstance.mapMemorySegment(64'h0,64'h3FFF);

masterBurst = new();

 activeMasterInstance.regWrite(DENALI_CDN_AXI_REG_Verbosity,DENALI_CDN_AXI_MESSAGEVERBOSITY_MEDIUM);

 for (int ii=0; ii<1000; ii++) begin
assert(masterBurst.randomize() with {
Type == (ii % 2 == 0 ? DENALI_CDN_AXI_TR_Read : DENALI_CDN_AXI_TR_Write);
                            Direction == (ii % 2 == 0 ? DENALI_CDN_AXI_DIRECTION_READ :
 DENALI_CDN_AXI_DIRECTION_WRITE); Cacheable == DENALI_CDN_AXI_CACHEMODE_NON_CACHEABLE;
Access == DENALI_CDN_AXI_ACCESS_NORMAL;
Length == 4;
StartAddress < 'h400;
Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
Data.size() == 16;
foreach (Data[i])
Data[i] == i;
             Size == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
             IdTag == 0;
}) else $fatal;
status = activeMasterInstance.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);

                  #1000;

                 end

        $finish;
end
endmodule
```

Chapter 3. AXI VIP Model Configuration

The AXI 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. The AXI VIP provides the configurations suggested by the AXI specification, and other testable configuration options.

3.1. SOMA

A detailed listing of the AXI SOMA parameters appears in the *AXI VIP User Interface Reference for UVM SystemVerilog.*

3.2. Registers

A detailed listing of the AXI registers appears in the *AXI VIP User Interface Reference for UVM SystemVerilog.*

Chapter 4. CDN_AXI VIP Model Operation

4.1. Model Instances

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

4.1.1. Instantiating the Models in the Testbench

The CDN_AXI VIP has a simple interface to access model, model memory space, model control registers, transactions, etc. 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.

Note

The syntactic patterns for the most used methods relating to accessing these models are readily available at `$DENALI/example/cdn_axi/denaliAxiUserInstance.sv` for AXI.

The following three kinds of memory models are associated with each instance:

- Registers - Used to handle the instance register space.
- Cache - Used to read and write into the cache associated with the *CDN_AXI ACE FULL* master device.
- Memory (main memory) - Used to read and write into the memory space associated with the slave device.

Refer the following example to initiate all three memory models in SystemVerilog:

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

      denaliMemInstance regInst ; // Handle to the register-space
      denaliMemInstance cache ;  // Handle to ACE-Full master cache
      denaliMemInstance memory ; // Handle to main memory

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

      ...
endclass
```

4.1.2. Model's Control Registers and Storage Memory

The CDN_AXI 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.

4.1.2.1. Accessing Control Registers

You can access control registers using the `regWrite()` and `regRead()` functions.

4.1.2.1.1. regWrite()

Refer the following example to write the control registers:

```
virtual function void regWrite( denaliCdn_axiRegNumT 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
```

Refer the following example to change the value of the `DENALI_CDN_AXI_REG_WriteIssuingCapability` register:

```
initial
begin
  activeMaster.regWrite(DENALI_CDN_AXI_REG_WriteIssuingCapability, 10 );
```

After this call, the write issuing capability is changed to 10. By default, the value of write issuing capability is read from the SOMA file.

4.1.2.1.2. regRead()

This function has one parameter - register address.

Following example shows how to read from a control register:

```
virtual function integer regRead( denaliCdn_axiRegNumT 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
```

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

4.1.2.2. Backdoor Reads and Writes to Slave Memory

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

To perform backdoor reads and writes refer the following examples:

```
class cdn_axiInstance extends denaliCdn_axiInstance;

    ...

    virtual function reg [7:0] memRead( reg [63:0] addr);
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        void'( memory.read( trans ) );
        memRead[7:0] = trans.Data[0];
    endfunction

    virtual function void memWrite( denaliCdn_axiRegNumT addr, reg [7:0] data );
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        trans.Data = new[1];
        trans.Data[0] = data[7:0];
        void'( memory.write( trans ) );
    endfunction

endclass
```

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

```
    Initial
    begin
      activeslave.memWrite('h1000,8b’10101010);
      #1000;
```

4.1.2.2.1. Pre-loading Memory from an External Data File

An agent memory (such as an Active Slave memory) can be pre-loaded from an external data file, using the `mmload` command. You cannot load undefined values into an agent memory.

To pre-load memory from an external data file:

1. Add the following code to your testbench:

```
integer success;

initial begin

     // Pre-load Active Slave memory from an external file using $mmload
     $display("Pre-loading Active Slave's memory from data file");
     success = $mmload("pstest.axiActiveSlaveDevice(memory)","../mr.data");

end
```

Note

The data file location is relative to the directory where your test is run. You can use environment variables in the path, if needed.

2. Create the data file and populate it with data/address mapping. Both address and data are expressed in Hex radix. For example:

```
0/  A;
1/  B;
2/  C;
3/  D;
4/  E;
5/  F;
6/  6;
7/  7;
```

   The memory data is loaded on a byte per address location basis. You can load data for the entire contiguous address space using *<start_addr>* : *<end_addr>* / *<data>* ; notation. (See the *Cadence Memory Model Portfolio User Guide* for more information.)

3. You can confirm that the memory content has been updated by searching your log file for a message similar to the following:

```
*Denali* Memory contents loaded from file "../mr.data" into instance
 "pstest.axiActiveSlaveDevice(memory)".
```

4.1.2.3. Backdoor Reads and Writes to the Master Cache

You can perform backdoor reads and writes to the CDN_AXI Full master cache space as shown below:

```
class cdn_axiInstance extends denaliCdn_axiInstance;

    ...

    function void user_write_entry(reg [63:0] StartAddress, reg [7:0] Data [],
        integer CacheLineSize, reg [7:0] state);
        reg [7:0] data_tmp [];
        data_tmp = new[Data.size()+1]; // Cache line size in bytes + 1
        data_tmp[0] = state;
        for (int i=1; i<data_tmp.size(); i++) begin
            data_tmp[i]=Data[i-1];
        end
        void'(cacheWrite(StartAddress / CacheLineSize ,data_tmp));
    endfunction

    virtual function void cacheWrite( reg [63:0] addr, reg [7:0] data []);
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        trans.Data = data;
        void'( cache.write( trans ) );
    endfunction

    virtual function void user_read_entry( reg [63:0] addr, integer CacheLineSize,
        ref reg [7:0] Data [], ref reg [7:0] state);
        reg [7:0] temp [];
        temp = new[Data.size()+1]; // cache line size in bytes + 1

        void'(cacheRead(addr / CacheLineSize, temp));
        for (int i=1; i<temp.size(); i++) begin
            Data[i-1] = temp[i];
        end
        state = temp[0];
    endfunction

    virtual function void cacheRead( reg [63:0] addr ,ref reg [7:0] cache_data []);
```

```
        denaliMemTransaction trans = new();
        trans.Address = addr ;
        void'( cache.read( trans ) );
        cache_data = trans.Data;
    endfunction
endclass
```

To perform backdoor writing to the master cache at address 0x0 with a dirty cache line, refer to the following example:

```
reg[7:0] Data[];
...
Initial begin
    Data = new [16]; //assuming cache line size is 16
    for(int i = 0; i < 16; i++) begin
        Data [i] = i;
    end
    // writing to cache address 0, the given data, cache line size is 16, and moving the cache line
    // state to Unique Dirty
    activeMaster.user_write_entry('h0,Data,16,DENALI_CDN_AXI_CACHELINESTATE_UNIQUE_DIRTY);
    #1000;
end
```

4.1.2.4. Defining Memory Segments

The register model lets you define memory regions with different domains. For example, attribute `DENALI_CDN_AXI_REGION_INNER` means that the memory segment is of INNER domain.

To define memory segments refer the following example:

```
 function void mapMemorySegment(reg [63:0] fromAddress, reg [63:0] toAddress, denaliCdn_axiDomainT
 domain);
        regWrite( DENALI_CDN_AXI_REG_RegionAddressFromHigh,fromAddress[63:32]);
        regWrite( DENALI_CDN_AXI_REG_RegionAddressFromLow, fromAddress[31:0]);
        regWrite( DENALI_CDN_AXI_REG_RegionAddressToHigh ,toAddress[63:32]);
        regWrite( DENALI_CDN_AXI_REG_RegionAddressToLow ,toAddress[31:0]);
        regWrite( DENALI_CDN_AXI_REG_RegionType ,domain);
    endfunction
```

Note

For AXI domain parameter is not needed, because the domain will always be `DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE`

The following example shows how to define a memory segment in the range [0..0xFFF]:

```
   axiActiveMaster.mapMemorySegment(64'h0,64'hFFF,DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE);
```

Note

The `regWrite` function is defined in the example Section 4.1.2, *"Model's Control Registers and Storage Memory"*

4.2. Instantiation / Elaboration

To instantiate any CDN_AXI VIP model, you must first configure a SOMA file and generate a Verilog instantiation interface through the PureView user interface. To create an HDL instantiation interface, see Section 2.2, *"Creating SOMA and HDL Instantiation Interface for CDN_AXI VIP"*

The following example uses the SystemVerilog code to show how an AXI3 model gets instantiated:

```
cdn_axi_master master
(
ACLK, ARESETN, AWVALID, AWADDR, AWLEN, AWSIZE, AWBURST,
AWLOCK, AWCACHE, AWPROT, AWREGION, AWQOS, AWID, AWREADY,
AWUSER, WVALID, WLAST, WDATA, WSTRB, WREADY, WUSER,
BVALID, BRESP, BID, BREADY, BUSER, ARVALID, ARADDR, ARLEN,
ARSIZE, ARBURST, ARLOCK, ARCACHE, ARPROT, ARREGION,
ARQOS, ARID, ARREADY, ARUSER, RVALID, RLAST, RDATA, RRESP,
RID, RREADY, RUSER, AWDOMAIN, AWSNOOP, AWBAR, WACK, ARDOMAIN,
ARSNOOP, ARBAR, RACK, ACVALID,ACREADY, ACADDR, ACSNOOP,
ACPROT, CRVALID, CRREADY, CRRESP, CDVALID, CDREADY, CDDATA, CDLAST
);
defparam master.interface_soma = "/home/cdn_axi/example/master.soma";
```

The module `cdn_axi_master` is defined in the Verilog instantiation interface code, which is automatically generated from a SOMA file. The parameter `interface_soma` must point to the SOMA file path.

4.3. 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_AXI VIP represents the transactions, followed by a detailed description of the interface that you can use to generate transactions.

4.3.1. Basic AXI Transactions Information

4.3.1.1. Frequently Used AXI Transaction Fields

Use the following files for test writing and debugging:

- `<VIPCAT_Inst_Dir>/tools/denali/ddvapi/sv/denaliCdn_axi.sv` file contains definitions of `denaliCdn_axiTransaction` (data item, including all transaction fields) and `denaliCdn_axiInstance` (VIP proxy) classes as well as large number of constraint blocks used in SV generation. These Blocks can be disabled if default behavior is not desired. See Section 7.4, *"Built-in SV Constraints"* section.
- `<VIPCAT_Inst_Dir>/tools/denali/ddvapi/sv/denaliCdn_axiTypes.svh` file contains all enumeration types defined in the VIP
- `<VIPCAT_Inst_Dir>/tools/denali/ddvapi/sv/denaliCdn_axiErrTable.svh` file contains all errors related enumeration types defined in the VIP

The following are the frequently used fields in test writing, callback writing and debugging.

Direction - Read or Write.

The following enums are defined in `denaliCdn_axiTypes.svh` file:

```
typedef enum {

DENALI_CDN_AXI_DIRECTION_UNSET = 0,
DENALI_CDN_AXI_DIRECTION_READ = 1,
```

```
DENALI_CDN_AXI_DIRECTION_WRITE = 2

} denaliCdn_axiDirectionT;
```

Length - The number of transfers in the burst (equals to `ALEN` + 1`)`Unsigned integer.

Size - Transfer size. Corresponds to `ARSIZE/AWSIZE`.

The following enums are defined in `denaliCdn_axiTypes.svh` file:

```
typedef enum {
  DENALI_CDN_AXI_TRANSFERSIZE_UNSET = 0,
  DENALI_CDN_AXI_TRANSFERSIZE_BYTE = 1,
  DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD = 2,
  DENALI_CDN_AXI_TRANSFERSIZE_WORD = 3,
  DENALI_CDN_AXI_TRANSFERSIZE_TWO_WORDS = 4,
  DENALI_CDN_AXI_TRANSFERSIZE_FOUR_WORDS = 5,
  DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS = 6,
  DENALI_CDN_AXI_TRANSFERSIZE_SIXTEEN_WORDS = 7,
  DENALI_CDN_AXI_TRANSFERSIZE_K_BITS = 8
} denaliCdn_axiTransferSizeT;
```

BurstMaxSize. Similar to *Size* but defines the *upper Size* value which could be used during SV transaction randomization

IdTag - Burst ID tag. Corresponds to `AWID/ARID/WID/RID/BID`. Unsigned integer

Data - Each entry in the array is one byte of the data used in transfers. For write transfers, the array includes write data. For read transfers, when the transaction is finished, the array includes all the read data received from transfers

Kind - FIXED, `INCR` or `WRAP`. Corresponds to `ARBURST/AWBURST`.

The following enums are defined in `denaliCdn_axiTypes.svh` file:

```
typedef enum {
  DENALI_CDN_AXI_BURSTKIND_UNSET = 0,
  DENALI_CDN_AXI_BURSTKIND_FIXED = 1,
  DENALI_CDN_AXI_BURSTKIND_INCR = 2,
  DENALI_CDN_AXI_BURSTKIND_WRAP = 3,
  DENALI_CDN_AXI_BURSTKIND_RESERVED = 4
} denaliCdn_axiBurstKindT;
```

StartAddress - First address in the burst. Unsigned integer.

Access - `NORMAL, EXCLUSIVE` or `LOCKED` access. Corresponds to `ARLOCK/AWLOCK`.

The following enums are defined `in denaliCdn_axiTypes.svh file`):

```
typedef enum {
  DENALI_CDN_AXI_ACCESS_UNSET = 0,
  DENALI_CDN_AXI_ACCESS_NORMAL = 1,
  DENALI_CDN_AXI_ACCESS_EXCLUSIVE = 2,
```

```
  DENALI_CDN_AXI_ACCESS_LOCKED = 3,
  DENALI_CDN_AXI_ACCESS_RESERVED = 4
} denaliCdn_axiAccessT;
```

Privileged - NORMAL or PRIVILEGED access. Corresponds to ARPROT [0] / AWPROT [0].

The following enums are defined in denaliCdn_axiTypes.svh file:

```
typedef enum {
  DENALI_CDN_AXI_PRIVILEGEDMODE_UNSET = 0,
  DENALI_CDN_AXI_PRIVILEGEDMODE_NORMAL = 1,
  DENALI_CDN_AXI_PRIVILEGEDMODE_PRIVILEGED = 2
} denaliCdn_axiPrivilegedModeT;
```

Secure - SECURE or NON-SECURE access. Corresponds to ARPROT [1] / AWPROT [1] .

The following enums are defined in denaliCdn_axiTypes.svh file:

```
typedef enum {
  DENALI_CDN_AXI_SECUREMODE_UNSET = 0,
  DENALI_CDN_AXI_SECUREMODE_SECURE = 1,
  DENALI_CDN_AXI_SECUREMODE_NONSECURE = 2
} denaliCdn_axiSecureModeT;
```

DataInstr - Data or Instruction access. Corresponds to ARPROT [2] / AWPROT [2].

The following enums are defined in denaliCdn_axiTypes.svh file:

```
typedef enum {
  DENALI_CDN_AXI_FETCHKIND_UNSET = 0,
  DENALI_CDN_AXI_FETCHKIND_DATA = 1,
  DENALI_CDN_AXI_FETCHKIND_INSTRUCTION = 2
} denaliCdn_axiFetchKindT;
```

Bufferable - NON_BUFFERABLE or BUFFERABLE. Corresponds to ARCACHE[0] / AW-CACHE[0].

The following enums are defined in denaliCdn_axiTypes.svh file:

```
typedef enum {
  DENALI_CDN_AXI_BUFFERMODE_UNSET = 0,
  DENALI_CDN_AXI_BUFFERMODE_NON_BUFFERABLE = 1,
  DENALI_CDN_AXI_BUFFERMODE_BUFFERABLE = 2
} denaliCdn_axiBufferModeT;
```

Cacheable - NON_CACHEABLE or CACHEABLE. Corresponds to ARCACHE[1] / AW-CACHE[1].

The following enums are defined in denaliCdn_axiTypes.svh file:

```
typedef enum {
```

```
  DENALI_CDN_AXI_CACHEMODE_UNSET = 0,
  DENALI_CDN_AXI_CACHEMODE_NON_CACHEABLE = 1,
  DENALI_CDN_AXI_CACHEMODE_CACHEABLE = 2
} denaliCdn_axiCacheModeT;
```

ReadAllocate - `NO_READ_ALLOCATE or READ_ALLOCATE`. Corresponds to `ARCACHE[2] / AWCACHE[2]`.

The following enums are defined in `denaliCdn_axiTypes.svh file`:

```
typedef enum {
  DENALI_CDN_AXI_READALLOCATE_UNSET = 0,
  DENALI_CDN_AXI_READALLOCATE_NO_READ_ALLOCATE = 1,
  DENALI_CDN_AXI_READALLOCATE_READ_ALLOCATE = 2
} denaliCdn_axiReadAllocateT;
```

WriteAllocate - `NO_WRITE_ALLOCATE` or `WRITE_ALLOCATE`. Corresponds to `ARCACHE[3] / AWCACHE[3]`.

The following enums are defined in `denaliCdn_axiTypes.svh file`:

```
typedef enum {
  DENALI_CDN_AXI_WRITEALLOCATE_UNSET = 0,
  DENALI_CDN_AXI_WRITEALLOCATE_NO_WRITE_ALLOCATE = 1,
  DENALI_CDN_AXI_WRITEALLOCATE_WRITE_ALLOCATE = 2
} denaliCdn_axiWriteAllocateT;
```

4.3.1.2. Using the ModelGeneration Field

The `ModelGeneration` field is a flag that indicates to the internal core to complete certain transaction fields, which are not been generated in SV environment, That is, the fields which remained `UNSET` in SV environment.

Following are the examples for clarification:

- If you assign only `Direction` and `StartAddress` fields in SV code and do not randomize your transaction in SV, then only `Direction` and `StartAddress` will be set to certain values and rest of the fields, even the `ModelGeneration`, will remain `UNSET`. When this transaction arrives to *e* core, all the remaining transaction fields besides `Direction` and `StartAddress` will get generated properly in *e* according to built-in *e* constraints.
- If you randomize your transaction in SV (Example; by using `randomize() or`uvm_do()), then all the transaction fields will be assigned in SV according to built-in SV constraints that are included in `denaliCdn_axi.sv` file, and then the `ModelGeneration` field will automatically be set to *0* (according to `fast_di_const` constraint block in the same file), indicating to the internal core that there is no need for internal generation, because everything was already generated in SV code.
- If you randomize your transaction in SV but intercept it later and assign any of its fields to `UNSET`, you must set the `ModelGeneration` field to *1* in order to notify the internal core that it must internally generate `UNSET` fields, so that all transaction fields get generated.

Example 4.1. ModelGeneration Usage

Following example shows how to randomize the `StartAddress` and `Domain` fields, and assign them to the burst.

Create a class that holds the variables to randomize.

```
class my_vars;
    rand reg [63:0] StartAddress;
    rand denaliCdn_axiDomainT Domain;
   …
endclass
```

In your test, instantiate the class, randomize the class and assign it*:*

```
my_vars random_vars;
   …
   initial
   begin
       random_vars = new();
       …
       void'(random_vars.randomize() with {
           StartAddress inside {['h1000:'h1FFF]};
           Domain == DENALI_CDN_AXI_DOMAIN_INNER;s]
       });
        …
       masterBurst.StartAddress = random_vars.StartAddress;
       masterBurst.Domain = random_vars.Domain;
       masterBurst.ModelGeneration = 1;
       …
       status = aceMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
    end
```

4.3.1.3. Using Slave Memory and Master Cache

- *`denaliMemInstance`* class is used for accessing register, memory, and cache spaces. It is instantiated inside active or passive agents
- **Master Cache:** by default initially all cache lines are in Invalid state. Byte number *0* in cache line contains the line status.
- **Slave Memory:** by default the entire memory content is randomized
- Both Cache and Memory could be overridden using `backdoor` access

1. Cache handling functions which includes

```
virtual function void user_write_entry(reg [63:0] StartAddress, reg [7:0] Data [], integer
 CacheLineSize, reg [7:0] state);
virtual function void user_read_entry( reg [63:0] addr, integer CacheLineSize, ref reg [7:0]
 Data [], ref reg [7:0] state);
virtual function reg [7:0] get_cache_line_state( reg [63:0] addr, integer cache_size);
```

2. Memory handling functions which includes

```
virtual function reg [7:0] memRead( reg [63:0] addr); <new line>virtual function void
 memWrite( denaliCdn_axiRegNumT addr, reg [7:0] data );
```

4.3.1.4. Defining Transactions for Various Flavors of AXI and ACE Specs

The same `denaliCdn_axiTransaction` data item class is used for `AXI3`, `AXI4`, and `ACE`.

Follow the below instruction to define a transaction:

- For either `AXI3` or `AXI4` mode, `CDN_ACE` should NOT be defined.
- For `ACE` mode it MUST be defined.

Besides `CDN_ACE`, you should define `Spec Version`, `Spec Subtype`, and `Spec Interface` fields for your transactions, in addition to the **SOMA** settings.

- In AXI mode, by default these fields are set to standard AXI3 spec
- In ACE mode, by default these fields are set to full ACE

It is recommended to define `myBaseTransaction` which is derived from `denaliCdn_axiTransaction` and uses `myBaseTransaction` in the sequences.

Following are the examples for `AXI4` and `ACE Lite` transactions:

Example 4.2. Defining transactions for AXI4

```
class myAxiBaseTransaction extends denaliCdn_axiTransaction;
    `uvm_object_utils(myAxiBaseTransaction)

     function new();
            super.new();
              // Set transaction to AXI4
              this.SpecVer = DENALI_CDN_AXI_SPECVERSION_AMBA4;
              this.SpecSubtype = DENALI_CDN_AXI_SPECSUBTYPE_BASE;
              this.SpecInterface = DENALI_CDN_AXI_SPECINTERFACE_FULL;
    endfunction


    constraint user_dut_information {
        BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
    }
endclass : myAxiBaseTransaction
```

Example 4.3. Defining transactions for ACE Lite

```
class myAceBaseTransaction extends denaliCdn_axiTransaction;
    `uvm_object_utils(myAceBaseTransaction)

     function new();
        super.new();
          // Specify ACE Lite transactions
          this.SpecVer = DENALI_CDN_AXI_SPECVERSION_AMBA4;
          this.SpecSubtype = DENALI_CDN_AXI_SPECSUBTYPE_ACE;
          this.SpecInterface = DENALI_CDN_AXI_SPECINTERFACE_LITE;
    endfunction

    constraint basic_const {
        CacheLineSize == 32;
        BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
        IdTag < (1 << 4);

    }
endclass : myAceBaseTransact
```

4.3.2. Transaction Generation

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

Note

Cadence recommends not to use the `denaliCdn_axiTransaction` module as it is. You must extend it and add DUT specific constraints.

Following is an example:

```
class myTransaction extends denaliCdn_axiTransaction;
    constraint user_dut_information {
        CacheLineSize == 16; // Cache line size in bytes
        BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_WORD; // For data bus size of 32 bits
        IdTag < (1 << 8); // For id signals of 8 bits
    ...
    }
endclass
```

The following example shows how to send a read transaction, where few of the fields are constrained:

```
import DenaliSvCdn_axi::*;
import DenaliSvMem::*;

module testbench;

...
cdn_axiInstance activeMaster;
myTransaction burst;

initial
   begin

     activeMaster = new ("testbench.master");
     burst = new();
     #1000;
     burst.Direction  = DENALI_CDN_AXI_DIRECTION_READ;
     burst.StartAddress = 'h100;
```

```
    burst.Size  = DENALI_CDN_AXI_TRANSFERSIZE_WORD;
    Burst.Length = 4;

    status = activeMaster.transAdd(burst, DENALI_CDN_AXI_QUEUE_Burst);
```

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

```
initial
   begin

     activeMaster = new ("testbench.master");
     burst = new();
     #1000;
     assert( burst.randomize() with {
           Direction       = DENALI_CDN_AXI_DIRECTION_READ;
           StartAddress = 'h100;
           Size = DENALI_CDN_AXI_TRANSFERSIZE_WORD;
           Length       = 4;
     });
     status = activeMaster.transAdd(burst, DENALI_CDN_AXI_QUEUE_Burst);
```

4.3.3. Transaction Flow

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

You can modify various transaction's data during the `BeforeSend` callback but not before `Started` and `Ended` callbacks.

4.3.3.1. Master – Sending a Write

The basic transaction flow steps include:

1. The master gets a high-level `Write` transaction from the user queue.
2. Callback `BeforeSend`.
   - During this callback, you can modify the transaction's data.
3. The `Write` transaction is split into two transactions:
   - `WriteAddress`
   - `WriteData`

   The `WriteData` transaction is not very useful, it exists solely for consistency with `ReadAddress` / `ReadData` pair.
4. `Callback BeforeSendAddress.`
5. The `WriteAddress` transaction is transmitted through the wires to the slave.
6. Callback `BeforeSendTransfer` for each `Write Transfer` transaction.
7. The `WriteData` transactions are transmitted through the wires to the slave.

8. Callback `EndedTransfer` for each `Write Transfer` transaction.
9. Callback ended for the `Write` transaction after the `WACK` signal is sent.

4.3.3.2. Slave – Receiving Write Transactions and Sending WriteResponse

The basic transaction flow step includes:

1. The slave gets the `Write Address` and all of the `Write Data` transactions from its input wires.
2. Callback started for `Write Address` and each of the `Write Data` transactions.
3. Callback ended `Write Address` and each of the `Write Data` transactions.
4. Slave creates and sends the `WriteResponse` transaction.
5. Callback `BeforeSendResponse`.
6. The `Write Response` transaction is transmitted through the wires to the master.
7. Callback `EndedResponse` for the `WriteResponse` transaction.

4.3.3.3. Master – Receiving WriteResponse

The basic transaction flow step includes:

1. The master gets `Write Response` from its input wires.
2. Callback `StartedResponse` for the `Write Response` transaction.
3. Callback `EndedResponse` for the `Write Response` transaction.
4. The Write transaction process is completed

4.3.3.4. Master – Sending ReadAddress

The basic transaction flow step includes:

1. The master gets a high-level `Read` transaction from the user queue.
2. Callback `BeforeSend`. During this callback, you can modify the transaction's control data.
3. Callback `BeforeSendAddress`.
4. The `Read Address` transaction is transmitted through the wires to the slave.

4.3.3.5. Slave – Receiving ReadAddress and Sending ReadData

The basic transaction flow step includes:

1. The slave gets `Read Address` from its input wires.

2. Callback `StartedAddress`.
3. The slave creates a high-level `Read Data` transaction.
4. Callback `BeforeSendTransfer` for the `Read Data` transactions.
5. Callback `BeforeSendTransfer` for each of the `Read Transfer` transactions.
6. All `Read Transfer` transactions are transmitted through the wires to the master.
7. Callback `EndedTransfer` for each of the `Read Transfer` transactions.

4.3.3.6. Master – Receiving ReadData

The basic transaction flow step includes:

1. The master gets all `Read Transfer` transactions from its input wires.
2. Callback `StartedTransfer` for each of the `Read Transfer` transactions.
3. Callback `EndedTransfer` for each of the `Read Transfer` transactions.
4. Callback ended for the `Read` transaction after the `RACK` signal is sent.
5. The `Read` transaction process is done.

4.3.3.7. Slave – Sending Snoop

The basic transaction flow step includes:

1. The slave gets a high-level `Snoop` transaction from the user queue.
2. Callback `BeforeSend`. During this callback, you can modify the transaction's control data.
3. Callback `BeforeSendAddress`.
4. The `Snoop Address` transaction is transmitted through the wires to the master.

4.3.3.8. Master – Receiving SnoopAddress and Sending SnoopResponse

The basic transaction flow step includes:

1. The master gets the `Snoop Address` from its input wires.
2. Callback `StartedAddress`.
3. Master creates the `Snoop Response` transaction.
4. Callback `BeforeSendResponse` for the `Snoop Response` transactions.
5. The `Snoop Response` transaction is transmitted through the wires to the slave.

6. If the Snoop Response contains data, the master creates a high-level Snoop Data transaction.
7. Callback BeforeSendTransfer for each of the SnoopTransfer transactions.
8. All Snoop Transfer transactions are transmitted through the wires to the master.
9. Callback EndedTransfer for each of the SnoopTransfer transactions.

4.3.3.9. Slave – Receiving SnoopResponse and SnoopData

The basic transaction flow step includes:

1. The slave gets the Snoop Response transaction from its input wires.
2. Callback StartedResponse for the SnoopResponse transaction.
3. The slave gets all the Snoop Transfer transactions (if exist) from its input wires.
4. Callback StartedTransfer for each of the Snoop Transfer transactions.
5. Callback EndedTransfer for each of the Read Transfer transactions.
6. Callback ended for the Snoop Transaction after the last data transfer in received.
7. The Snoop Transaction process is completed.

4.3.4. Controlling Delays on AXI Channels

This section explains how to constrain each of the delays for the different transaction types using specific fields of denaliCdn_axiTransaction class.

The delays can be set at two points:

- Before inserting a transaction to the relevant queue using transAdd() (either by doing randomize or set the fields procedurally).
- Setting the relevant callback using transSet( )for the relevant transaction types as listed below.

4.3.4.1. Controlling VALID Signals

4.3.4.1.1. Controlling AWVALID and ARVALID Master Delay

The master delay on the address channels is calculated as per the number of cycles, from the cycle the BFM first attempts to send the burst until asserting AWVALID or ARVALID and put the burst's control information on the address channel.

To control the master delay on the address channels:

- Constrain the TransmitDelay field of denaliCdn_axiTransaction

- Disable a built-in `TransmitDelay_const` constraint block which constraints `TransmitDelay` to zero

Example 4.4. Controlling AxVALID Signal Delay

```
masterBurst = new();
masterBurst.TransmitDelay_const.constraint_mode(0);
masterBurst.TransmitDelay = 10;
status = activeMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: You can set the delay by using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction types `DENALI_CDN_AXI_TR_Write / DENALI_CDN_AXI_TR_Read`.

4.3.4.1.2. Controlling WVALID Master Delay

The delay for write transfers on the data channel is the number of cycles from the first cycle that the write transfer is ready and chosen until the assertion of `WVALID`.

To control the master delay on the write data channel constrain the `TransfersChannelDelay` field of `denaliCdn_axiTransaction`.

Example 4.5. Controlling WVALID Signal Delay

```
masterBurst = new();
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.Length = 4;
masterBurst.TransfersChannelDelay = new [4];
for (int i=0; i< masterBurst.TransfersChannelDelay.size(); i++) begin
masterBurst.TransfersChannelDelay[i] = i;
end
status = activeMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction type `DENALI_CDN_AXI_TR_Write`.

4.3.4.1.3. Controlling RVALID Slave Delay

The delay for read transfers on the data channel is the number of cycles from the first cycle that the read transfer response is ready and chosen until the assertion of `RVALID`.

To control the master delay on the write data channel constrain the `TransfersChannelDelay` field of `denaliCdn_axiTransaction`.

Example 4.6. Controlling RVALID Signal Delay

```
slaveResp = new();
slaveResp.Length = 4;
slaveResp.Direction = DENALI_CDN_AXI_DIRECTION_READ;
slaveResp.TransfersChannelDelay = new [4];
for (int i=0; i< slaveResp.TransfersChannelDelay.size(); i++) begin
slaveResp.TransfersChannelDelay[i] = i;
end
status = activeSlave.transAdd(slaveResp, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSendResponse` with transaction type `DENALI_CDN_AXI_TR_ReadData.`

4.3.4.1.4. Controlling BVALID Slave Delay

The slave delay on the write response channel is calculated as the number of cycles from the first cycle that the burst can be sent on the write response channel until the assertion of `BVALID`.

To control the `BVALID` delay on the write response channel constrain the `ChannelDelay` fields of `denaliCdn_axiTransaction`.

Example 4.7. Controlling BVALID Signal Delay

```
slaveResp = new();
slaveResp. Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
slaveResp.ChannelDelay = 10;
status = activeSlave.transAdd(slaveResp, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSendResponse` with transaction type `DENALI_CDN_AXI_TR_WriteResponse.`

4.3.4.1.5. Controlling RACK and WACK Master Delay

The delay in cycles between the last response from the slave/interconnect, to the cycle in which the `RACK` (after last read transfer) or `WACK` (after write response phase) signal will be raised.

To control the `RACK` and `WACK` master delay, constrain the `AckDelay` field of `denaliCdn_axiTransaction.`

Example 4.8. Controlling RACK and WACK Signal Delay

```
masterBurst = new();
masterBurst.AckDelay = 5;
status = activeMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction type `DENALI_CDN_AXI_TR_Read / DENALI_CDN_AXI_TR_Write`. Or callback `DENALI_CDN_AXI_CB_BeforeSendAddress` with transaction type `DENALI_CDN_AXI_TR_ReadAddress / DENALI_CDN_AXI_TR_WriteAddress`.

4.3.4.1.6. Controlling ACVALID Slave Delay

The delay in cycles from the time BFM first attempts to send the snoop burst until asserting `ACVALID` and putting the snoop burst's control information on the snoop address channel.

To control the slave delay on the snoop address channel, constrain the `TransmitDelay` field of `denaliCdn_axiTransaction`

Disable a built-in `TransmitDelay_const` constraint block that constraints `TransmitDelay` to zero.

Example 4.9. Controlling ACVALID Signal Delay

```
slaveSnoop = new();
slaveSnoop.TransmitDelay_const.constraint_mode(0);
slaveSnoop.TransmitDelay = 5;
status = activeMaster.transAdd(slaveSnoop, DENALI_CDN_AXI_QUEUE_Snoop);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction type `DENALI_CDN_AXI_TR_Snoop.`

4.3.4.1.7. Controlling CRVALID Master Delay

The delay in cycles until the master drives `CRVALID HIGH` as a response for a Snoop burst. To control the master delay on the snoop response channel; Constrain the `ResponseDelay` field of `denaliCdn_axiTransaction`

Example 4.10. Controlling CRVALID Signal Delay

```
masterSnoopResp = new();
masterSnoopResp.ResponseDelay = 5;
status = activeMaster.transAdd(masterSnoopResp, DENALI_CDN_AXI_QUEUE_Snoop);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction type `DENALI_CDN_AXI_TR_SnoopResponse.`

4.3.4.1.8. Controlling CDVALID Master Delay

The delay for snoop transfers on the snoop data channel, is the number of cycles from the first cycle that the snoop transfer is ready and chosen until the assertion of `CDVALID`.

To control the master delay on the snoop data channel constrain the `TransfersChannelDelay` field of `denaliCdn_axiTransaction`.

Example 4.11. Controlling CDVALID Signal Delay

```
masterSnoopResp = new();
masterSnoopResp.TransfersChannelDelay = new [4];
for (int i=0; i< masterSnoopResp.TransfersChannelDelay.size(); i++) begin
masterSnoopResp.TransfersChannelDelay[i] = i;
end
status = activeMaster.transAdd(masterSnoopResp, DENALI_CDN_AXI_QUEUE_Snoop);
```

Control through callback: This delay can also be set using `transSet()` inside the callback `DENALI_CDN_AXI_CB_BeforeSend` with transaction type `DENALI_CDN_AXI_TR_SnoopResponse.`

4.3.4.2. Controlling READY Signals

For each ready signal in each channel, there is a control field and a delay field. These fields control the behavior of the ready signals. The following table describes the control and delay fields for each channel:

Table 4.1. READY Signal Control Fields

Channel	Control Field	Delay Field	Agent	Signal
Write address	AreadyControl	AddressDelay	Slave	AWREADY
Read address	AreadyControl	AddressDelay	Slave	ARREADY
Write data	WreadyControl	TransfersChannelDelay	Slave	WREADY
Read data	RreadyControl	TransfersChannelDelay	Master	RREADY
Write response	BreadyControl	BreadyDelay	Master	BREADY
Snoop address	AcreadyControl	AddressDelay	Master	ACREADY
Snoop response	CrreadyControl	CrreadyDelay	Master	CRREADY
Snoop data	CdreadyControl	SnoopTransfersCdreadyDelay	Slave	CDREADY

Note

The control and delay fields for READY signals in each data item always affect the next item on the channel. Therefore, constraining a burst or transfer influences only the following burst or transfer. This also means that in a test, only the second burst and the second transfer can effectively be constrained.

The control field is of type **denaliCdn_axiReadyControlT** and can be set to one of the following values:

`DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION`

The READY signal remains low for the number of cycles specified by the corresponding delay field, and then it is asserted.

`DENALI_CDN_AXI_READYCONTROL_OSCILLATING`

The READY signal is switched between low and high. The number of cycles at each value is the number of cycles specified by the corresponding delay field. The cycle restarts when the VALID signal is asserted. This setting allows the READY signal to be low when the corresponding VALID signal is low.

`DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID`

The READY signal remains low until the corresponding VALID signal is asserted.

`DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID_AND_DELAY`

When the VALID signal is asserted, the corresponding READY signal remains low for the number of cycles specified by the corresponding delay field, and then it is asserted.

`DENALI_CDN_AXI_READYCONTROL_`
`STALL_UNTIL_VALID_AND_OTHER_PHASE_STARTED`

Relevant only for the write address and write data channels. On the write data channel, the READY signal is kept low until the write address VALID signal is asserted and the burst is started on the write address channel. On the write address channel, the READY signal is kept low until the write data VALID signal is asserted and the burst is started on the write data channel.

Note

Use this controller with caution because it can cause a deadlock in some scenarios.

When the ready controller waits for another channel to raise its VALID signal, a deadlock will occur if the other channel fails to do so.

A deadlock can occur in the following scenario: If the WREADY controller is set to **DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID_AND_OTHER_PHASE_STARTED** but the burst sends data before the address, then the WREADY signal stalls until the address phase starts, but the address phase does not start until data transfers are sent. This effectively causes a deadlock.

To prevent a deadlock when you set the control field to **DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID_AND_OTHER_PHASE_STARTED**, use the EnforceReadyController and MaxDelayReadyController transaction fields.

4.3.4.2.1. Controlling AWREADY and ARREADY Slave Delay

The slave delay on the address channel is calculated as the number of cycles to keep AWREADY or ARREADY low after the completion of the burst address phase.

To control the slave delay on the address channel, constrain the `AddressDelay` and `AreadyControl` fields of `denaliCdn_axiTransaction`.

Example 4.12. Controlling AxREADY Signal Delay

```
slaveResp = new();
slaveResp.AddressDelay = 5;
slaveResp.AreadyControl = DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION;
status = activeSlave.transAdd(slaveResp, DENALI_CDN_AXI_QUEUE_Burst);
```

Control through callback: In DENALI_CDN_AXI_TR_ReadData and DENALI_CDN_AXI_TR_WriteResponse transactions, you can set this delay using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSendResponse callback.

Example 4.13. Controlling ARREADY Signal Delay

```
`uvm_info(get_type_name(), "Virtual sequence pre-loads Slave Response with delayed ARREADY signal",
 UVM_LOW);
`uvm_create_on(slaveResp, p_sequencer.slave_seqr);

// Turn off the built-in delay_timing_const_AddressDelay constraint
slaveResp.delay_timing_const_AddressDelay.constraint_mode(0);

`uvm_rand_send_with(slaveResp, {
          slaveResp.AreadyControl == DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION;
          slaveResp.AddressDelay == 5;
          })
```

Note

In this example, the built-in constraint block **delay_timing_const_AddressDelay** must be disabled before issuing customized a slave response with **AddressDelay** = 1.

Control through callback: In DENALI_CDN_AXI_TR_ReadData and DENALI_CDN_AXI_TR_WriteResponse transactions, you can set this delay using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSendResponse callback.

For more code examples and for images of the resulting waveforms, see Section 4.3.4.2.4, *"Controlling BREADY Master Delay"*

4.3.4.2.2. Controlling RREADY Master Delay

The delay for read transfers on the read data channel is the number of cycles from the end of the transfer until the assertion of RREADY.

To control the delay on the read data channels, constrain the TransfersChannelDelay and RreadyControl fields of `denaliCdn_axiTransaction.`

Example 4.14. Controlling RREADY Signal Delay:

```
`uvm_create_on(masterBurst, p_sequencer.master_seqr);

// Turn off the built-in Arraysizes_const_TransfersChannelDelay constraint
masterBurst.Arraysizes_const_TransfersChannelDelay.constraint_mode(0);

`uvm_rand_send_with(masterBurst, {
       masterBurst.Direction ==  DENALI_CDN_AXI_DIRECTION_READ;
       masterBurst.Length == 4;
       masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
       masterBurst.BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
       masterBurst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
       masterBurst.IdTag == 3;
       masterBurst.StartAddress > 'h0000;
       masterBurst.StartAddress < 'h1000;
       masterBurst.RreadyControl == DENALI_CDN_AXI_READYCONTROL_OSCILLATING;
       masterBurst.TransfersChannelDelay.size() == 4;
       masterBurst.TransfersChannelDelay[0] == 2;
       masterBurst.TransfersChannelDelay[1] == 4;
       masterBurst.TransfersChannelDelay[2] == 6;
       masterBurst.TransfersChannelDelay[3] == 8;
       })
```

Control through callback: In DENALI_CDN_AXI_TR_Read transactions, you can set this delay using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSend callback.

```
denaliCdn_axiInstance inst;
inst = new("top.i0");
```

For more code examples and for images of the resulting waveforms, see Section 4.3.4.2.4, "*Controlling BREADY Master Delay*"

4.3.4.2.3. Controlling WREADY Slave Delay

The delay for write transfers on the data channel is the number of cycles from the end of the transfer until the assertion of WREADY.

To control the slave delay on the write data channel, constrain the WreadyControl and TransfersChannelDelay fields of `denaliCdn_axiTransaction`.

Example 4.15. Controlling WREADY Signal Delay

```
`uvm_create_on(slaveResp, p_sequencer.slave_seqr);

`uvm_rand_send_with(slaveResp, {
      slaveResp.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
      slaveResp.Length == 4;
      slaveResp.WreadyControl == DENALI_CDN_AXI_READYCONTROL_OSCILLATING;
      slaveResp. TransfersChannelDelay.size() == 4;
      for (int i=0; i< slaveResp.TransfersChannelDelay.size(); i++) begin
            slaveResp.TransfersChannelDelay[i] == 2;
            end
      })
```

Control through callback: In DENALI_CDN_AXI_TR_WriteResponse transactions, you can set this delay using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSendResponse callback.

For more code examples and for images of the resulting waveforms, see Section 4.3.4.2.4, "*Controlling BREADY Master Delay*"

4.3.4.2.4. Controlling BREADY Master Delay

The master delay on the write response channel is calculated as the number of cycles from the end of the write burst (assertion of both BREADY and BVALID) until another assertion of BREADY.

To control the master delay on the write response channel, constrain the BreadyDelay and BreadyControl fields of `denaliCdn_axiTransaction`.

You can also set the BreadyDelay field using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSend callback with transaction type DENALI_CDN_AXI_TR_Write.

Example 4.16. Controlling BREADY Signal Delay

```
masterBurst = new();
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.BreadyDelay = 5;
masterBurst.BreadyControl = DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION;
status = activeMaster.transAdd(slaveResp, DENALI_CDN_AXI_QUEUE_Burst);
```

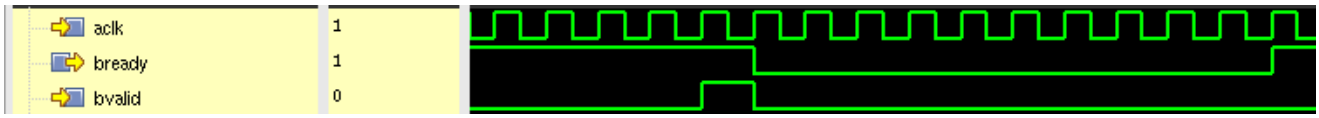
Control through callback: In DENALI_CDN_AXI_TR_Write transactions, you can set this delay using `transSet()` in the DENALI_CDN_AXI_CB_BeforeSend callback.

Example 4.17. BREADY Delayed Assertion

```
`uvm_info(get_type_name(), "Virtual sequence issuing a write master burst", UVM_LOW);
`uvm_create_on(masterBurst, p_sequencer.master_seqr);

`uvm_rand_send_with(masterBurst, {
        masterBurst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
        masterBurst.Length == 16;
        masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
        masterBurst.BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
        masterBurst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
        masterBurst.StartAddress > 'h0000;
        masterBurst.StartAddress < 'h1000;
        masterBurst.BreadyControl == DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION;
        masterBurst.BreadyDelay == 10;
        })
```

This example code produces the following signal behavior:

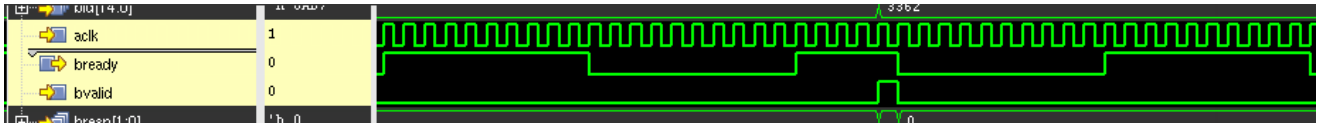


BREADY is de-asserted for 10 clock cycles after BVALID/BREADY handshaking.

Example 4.18. BREADY Oscillating

```
`uvm_create_on(masterBurst, p_sequencer.master_seqr);
for (int i=0; i<2; i++) begin
`uvm_rand_send_with(masterBurst, {
          masterBurst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
          masterBurst.Length == 16;
          masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
          masterBurst.BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
          masterBurst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
          masterBurst.StartAddress > 'h0000;
          masterBurst.StartAddress < 'h1000;
          masterBurst.BreadyControl == DENALI_CDN_AXI_READYCONTROL_OSCILLATING;
          masterBurst.BreadyDelay == 10;
          })
```

This example code produces the following signal behavior:



After BVALID/BREADY handshaking, BREADY continues to oscillate. The next BVALID/BREADY handshaking causes the cycle to restart.

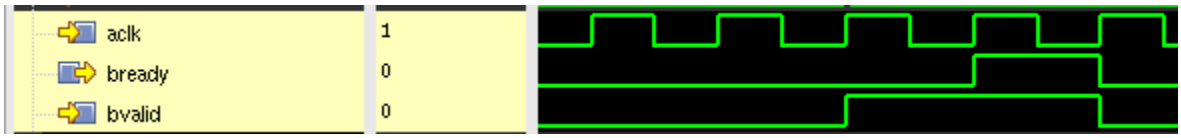
The half-period of the cycle of the BREADY signals is 10 clock cycles.

Example 4.19. BREADY Stall Until Valid

```
`uvm_create_on(masterBurst, p_sequencer.master_seqr);

for (int i=0; i<2; i++) begin
`uvm_rand_send_with(masterBurst, {
          masterBurst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
          masterBurst.Length == 16;
          masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
          masterBurst.BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
          masterBurst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
          masterBurst.StartAddress > 'h0000;
          masterBurst.StartAddress < 'h1000;
          masterBurst.BreadyControl == DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID;
          })
```

This example code produces the following signal behavior:

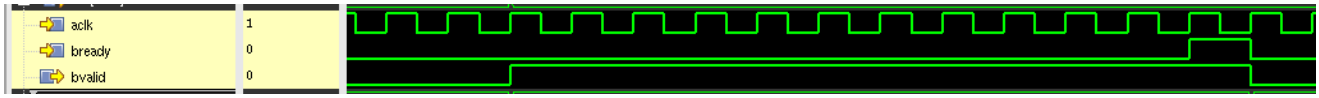


BREADY is de-asserted after the BVALID/BREADY handshaking and remains low until the next BVALID assertion.

Example 4.20. BREADY Stall Until Valid and Delay

```
for (int i=0; i<2; i++) begin
`uvm_rand_send_with(masterBurst, {
          masterBurst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
          masterBurst.Length == 16;
          masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
          masterBurst.BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
          masterBurst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
          masterBurst.StartAddress > 'h0000;
          masterBurst.StartAddress < 'h1000;
          masterBurst.BreadyControl == DENALI_CDN_AXI_READYCONTROL_STALL_UNTIL_VALID_AND_DELAY;
          masterBurst.BreadyDelay == 10;
          })
```

This example code produces the following signal behavior:



After BVALID is asserted, BREADY remains low for the number of clock cycles specified in the **BreadyDelay** field. After the BVALID/BREADY handshaking is over, BREADY is de-asserted.

4.3.4.3. Setting all Master Delays to Zero

You can remove all delays for the CDN_AXI active master by setting the following fields:

- TransfersChannelDelay: Set all items to zero.
- TransmitDelay: Set to zero.
- ChannelDelay: Set to zero.

- BreadyDelay: Set to zero.
- BreadyControl: Set to `DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION`.
- RreadyControl: Set to `DENALI_CDN_AXI_READYCONTROL_DELAYED_ASSERTION`.

If you are using SystemVerilog randomization, you can add constraints on the above mentioned fields. If not, you can set them procedurally for each transaction or set them in the DENALI_CDN_AXI_CB_BeforeSend callback. For more information, see Chapter 5, *AXI VIP Callbacks*.

The following SOMA configuration parameters can also cause delays on master transmissions, and therefore should be unset:

- stall_addr_after_max_write_depth
- stall_addr_after_max_read_depth
- stall_addr_after_max_write_acceptance_capability
- stall_addr_after_max_read_acceptance_capability

4.3.5. Auxiliary Transaction Field

There are two auxiliary transaction fields that are set for the users transaction class extending `denaliCdn_axiTransaction` and NOT set using `transAdd()` or `transSet()`.

BurstMaxSize: The maximum size of a transfer in the burst; should be set to be the data bus width. The field is of type `denaliCdn_axiTransferSizeT.`

CacheLineSize: The size in bytes of the cache line. The field type is unsigned integer.

Example 4.21. Example

```
class myTransaction extends denaliCdn_axiTransaction;

constraint user_dut_information {
CacheLineSize == 16;
BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
}

endclass
```

The fields *SpecVer*, *SpecSubtype* and *SpecInterface* are also defined in the users transaction class.

To define transactions for various flavors of AXI/ACE specs see:Section 4.3, *"Transactions"*

Chapter 5. AXI VIP Callbacks

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

5.1. Transaction Callbacks

You can set 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 AXI 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 AXI 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.

5.1.1. Transaction Callback Interface

The AXI VIP transaction callback interface provides callback initialization and handling. The `denaliCdn_axiCallback` 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_AXI_CB_<callback_name>` triggers a pre-defined function. For example:

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

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

5.1.2. Instantiating Transaction Callbacks

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

5.1.3. Enabling Transaction Callbacks

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

5.1.3.1. setCallback()

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

```
cdn_axiMaster.setCallback(DENALI_CDN_AXI_CB_Ended);
```

Here, `cdn_axiMaster` is one of the initialized models of type `denaliCdn_axiInstance` and `DENALI_CDN_AXI_CB_Ended` is one of the predefined callbacks of type `denaliCdn_axiCbPointT`.

5.1.3.2. clearCallback()

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

```
cdn_axiMaster.clearCallback(DENALI_CDN_AXI_CB_Ended);
```

5.1.4. 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_axi.sv`.

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

5.1.4.1. reasonStr()

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

5.1.4.2. Example

The following example shows how the above mentioned functions are used:

```
class denaliCdn_axiInstance;
    ...
    virtual function int EndedCbF (ref denaliCdn_axiTransaction trans) ;
       if (trans == null) begin
           $display("==== TRANS is null");
       end
       else begin
```

```
            if (trans.Type == DENALI_CDN_AXI_TR_Write) begin
                $display("Detected DENALI_CDN_AXI_TR_Write");
                trans.printInfo();
                // Process write transaction callback
            end

            if (trans.Type == DENALI_CDN_AXI_TR_Read) begin
                $display("Detected DENALI_CDN_AXI_TR_Read");
                trans.printInfo();
                // Process read transaction callback
            end
        end
        return super.EndedCbF(trans);
    endfunction

    ...
endclass
```

Chapter 6. AXI VIP Simulation

6.1. Running the Model

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

6.1.1. 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.

- 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 6.1.4, “*Ending Simulation*”.

6.1.2. Linking the AXI VIP Library and Running the Simulation

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

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

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

6.1.3. Viewing Results

After completing the simulation run, the AXI 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 AXI 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.

6.1.4. 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.

6.2. Controlling Model Behavior

You can control the model behavior using the run-time initialization file `.denalirc`. This file stores settings that control the AXI 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.

6.2.1. The .denalirc File Hierarchy

Before running the simulation for the first time, the AXI 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.

6.3. Output Files

You can enable the AXI 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.

6.3.1. History File

The history file allows you to view the details of the AXI 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_axiTypes.svh` file for SystemVerilog or `api_cdn_axi.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.

6.3.2. 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 AXI VIP model. It contains the path of the SOMA file used for a model as well as all settings in the SOMA file.

  The AXI 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.

6.3.2.1. 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.

6.4. Verification Messages

6.4.1. Changing Message Severity

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

- DENALI_CDN_AXI_ERR_CONFIG_SEVERITY_Error = 2 /* Change the severity level to Error
- DENALI_CDN_AXI_ERR_CONFIG_SEVERITY_Warn = 3 /* Change the severity level to Warn
- DENALI_CDN_AXI_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_AXI_ERR_CONFIG_SEVERITY_Info;
value = (errId << DENALI_CDN_AXI_Rpos__ErrCtrl_errId) |
(severity << DENALI_CDN_AXI_Rpos__ErrCtrl_severity);
regInst.writeReg(DENALI_CDN_AXI_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_AXI 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*".

6.4.2. Message List

A detailed listing of the AXI error messages appears in the *AXI VIP User Interface Reference for UVM SystemVerilog.*

6.5. Debugging the Simulation

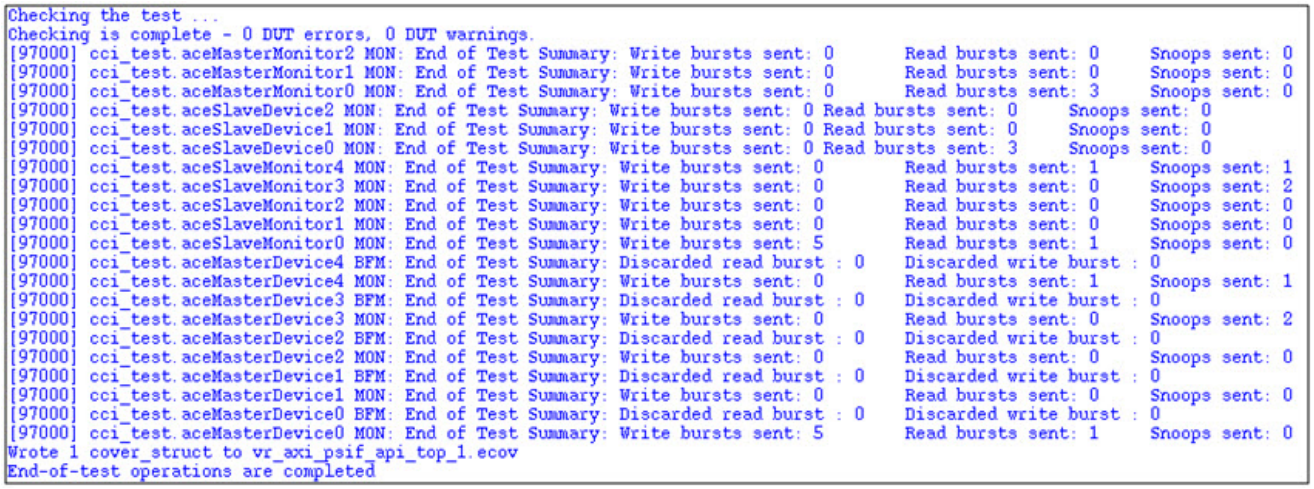
Running a test case with the AXI VIP often results in the display of a few errors or informational messages. The AXI VIP can recover from some errors and continue the execution of the test case as intended. However, serious errors on the part of the DUT might place the model in an unrecoverable state 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.

6.5.1. Check the Run Summary

In the end of the test, a transaction summary is being printed to the screen as shown below:

Figure 6.1. Transaction Summary



You can view a summary for each device in the environment, and monitor it to match your expectations.

Note

If you see any bursts getting discarded, search the log to find the precise reason. For more information see Section 10.3, "*CDN_AXI Specific Troubleshooting*".

6.5.2. Adding Transaction Recording to the Waveform (available only in ncsim)

You can add some information about the transaction to the waveform.

To activate the transaction recording write `1` to the `DENALI_CDN_AXI_REG_HasTrRecording register`

To add the transaction to the wave form, perform the following:

1. In the Design Browser, navigate to the simulator using **sys** >>**psif_cdn_axi_env_list** in the left hand side menu.

   The **psif_cdn_axi_env_list** contains the list of all the VIP agents.

2. Navigate to **<agent_num>** >> **active_masters/active_slaves** to view the related signal on the right side window.

   Figure 6.2. Agent List

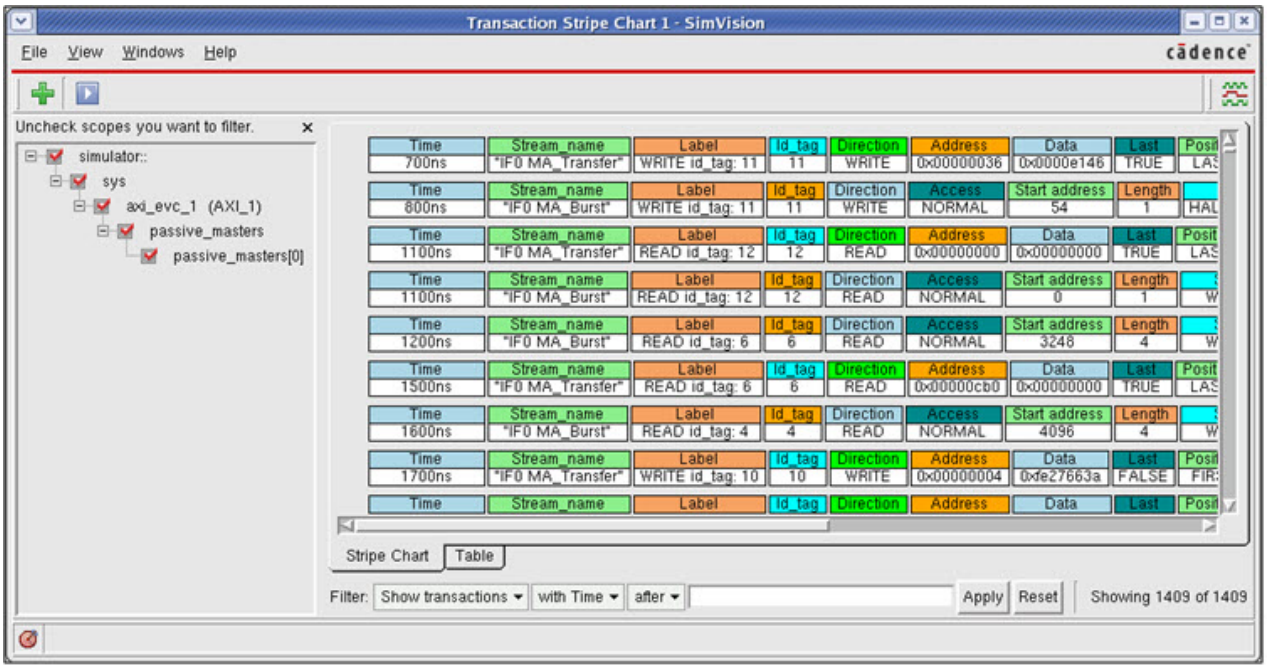
3. Right click and select **Send to Waveform** option.

6.5.3. Send the Transaction Information to the Transaction Stripe Chart (available only in ncsim)

1. Navigate to the simulator window using **Windows** >>**New** >>**Transaction Stripe Chart.**

2. Click on the green + icon at the top left corner, and select **All recorded transaction streams** option to display the transaction table as shown below.

Figure 6.3. Transaction Stripe Chart



3. Choose your preferred time in the table and click **OK** to view the selected transactions time.

Chapter 7. CDN_AXI Verification Test Scenarios

7.1. Test Flow

7.1.1. SystemVerilog UVM Test Flow

A typical SystemVerilog UVM test flow begins with writing a sequence. In the sequence, you have two options:

1. Use the SV constraints to randomize the transaction using `` `uvm_do() ``, `` `uvm_do_with() ``, or `` `uvm_rand_send() ``. When you use these macros, you can pass the transaction to the sequencer.

2. Set the transaction fields manually and pass the transaction to the sequencer using `` `uvm_send(). ``

The sequencer then passes the transaction to the CDN_AXI VIP. Before the CDN_AXI VIP passes the transaction to the wires, it activates the relevant `BeforeSend` callbacks (for more details, refer to Chapter 5, *AXI VIP Callbacks*). If you change the transaction in `BeforeSend`, you must use the `transSet()` function to pass the changes to the CDN_AXI VIP.

If you set `ModelGeneration` field, the CDN_AXI VIP will generate all the fields that you did not set.

Note

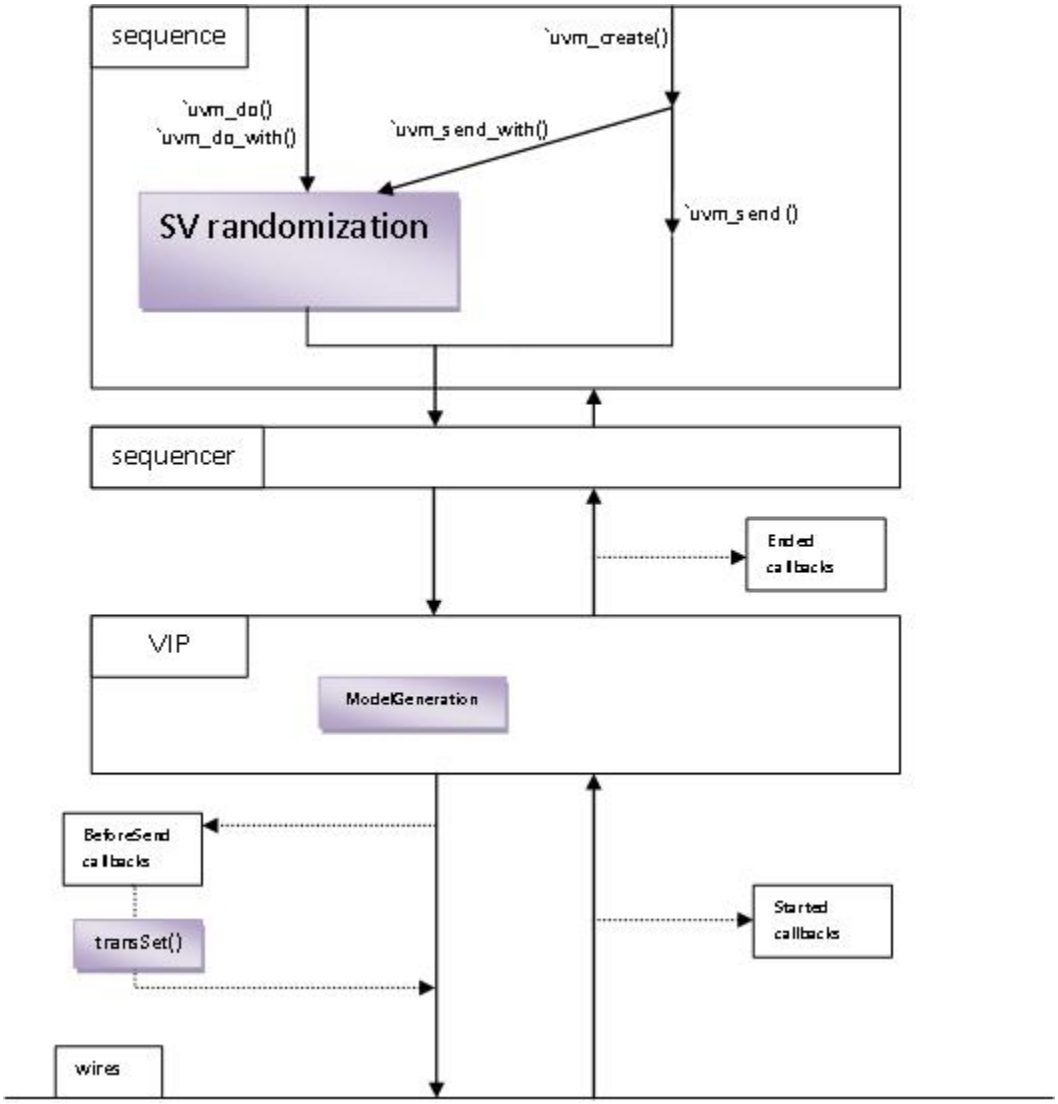
This is only relevant if you used `` `uvm_send() `` macro previously.

When the transaction gets to the wires, the model issues `Started` callbacks at the appropriate time.

When the transaction is ended, the model would issue the Ended callbacks.

Refer to the following figure that illustrates the overall SystemVerilog UVM test flow.

Figure 7.1. SystemVerilog UVM Test Flow



7.1.2. SystemVerilog Test Flow

A typical SystemVerilog test flow begins with writing a sequence. After you create a transaction using the `new()` command, you can manually edit the transaction or turn on or turn off SV constraints. You can then randomize the transaction using the `randomize()` function.

Once you are done with the transaction, pass it to the agent sequencer using the `transSet()` function.

The sequencer then passes the transaction to the CDN_AXI VIP. Before the CDN_AXI VIP passes the transaction to the wires, it activates the relevant `BeforeSend` callbacks (for more details, refer to Chapter 5, *AXI VIP Callbacks*). If you change the transaction in `BeforeSend`, you must use the `transSet()` function to pass the changes to the CDN_AXI VIP.

If you set the `ModelGeneration` field, the CDN_AXI VIP will generate all the fields that you did not set.

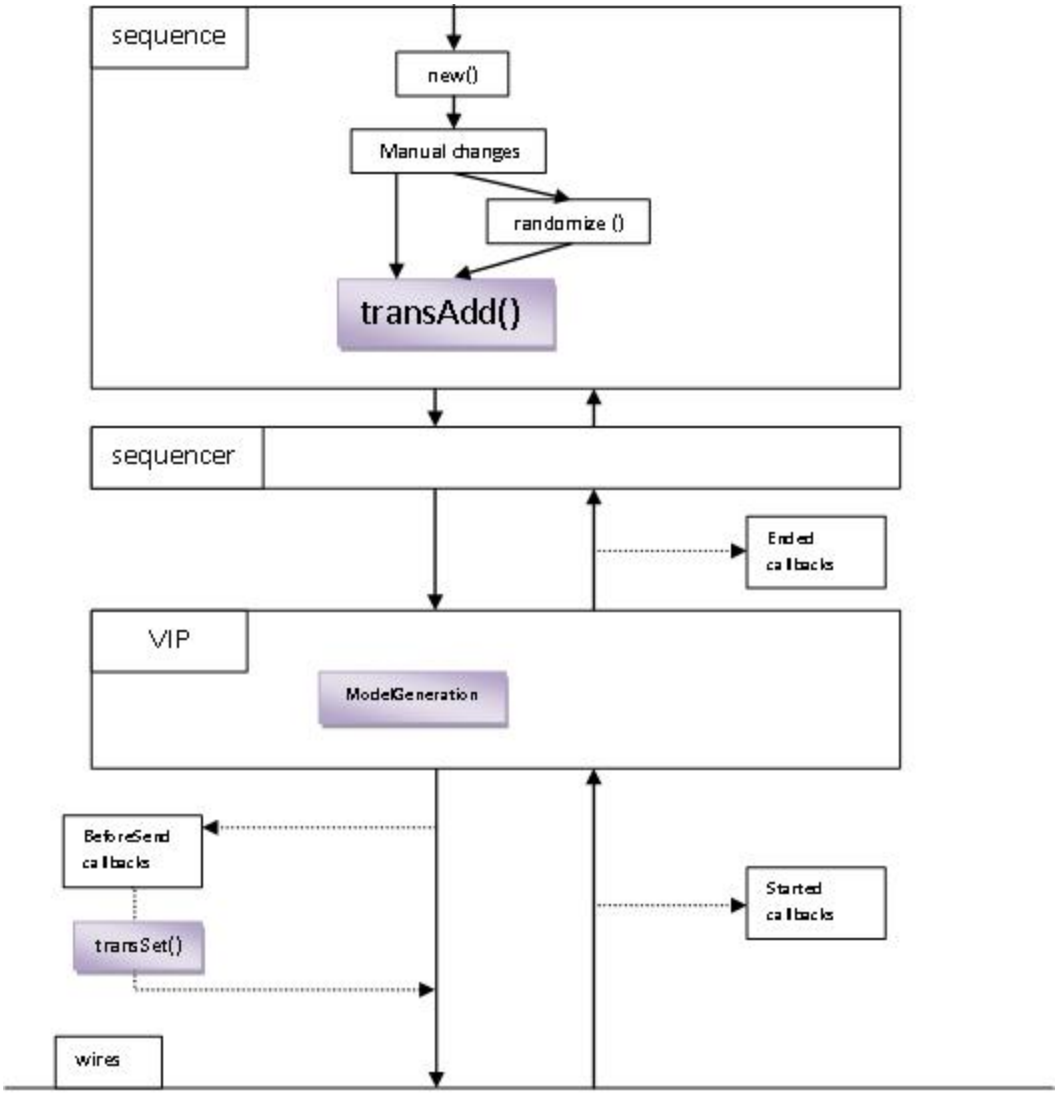
Note

This is only relevant if you used `randomize()` function previously.
When the transaction gets to the wires, the model issues `Started` callbacks at the appropriate time.

When the transaction is ended, the model would issue the Ended callbacks.

Refer to the following figure that illustrates the overall SystemVerilog test flow.

Figure 7.2. SystemVerilog Test Flow



7.2. Active Master Test Scenarios

The examples in the following sections work with Slave DUT or Slave interface of an interconnect DUT.

7.2.1. Simple FIXED Write Burst

This example shows how to use `uvm_do_with to send a simple WRITE burst. The burst is FIXED, with Length 3 and size of half word. The rest of the values are randomized.

```
  `uvm_do_with(burst,
    {burst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
    burst.Length == 3;
    burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD;
    burst.Kind == DENALI_CDN_AXI_BURSTKIND_FIXED;
   });
```

Example of the output in the log:

```
UVM_INFO /myenv/cdnAxiUvmUserMasterDriver.sv(66) @ 1000:
uvm_test_top.axi_sve.myUvmEnv.activeMaster.driver [cdnAxiUvmUserMasterDriver]
Adding transaction to UVC Transmission Queue with Unique ID = 0
…
[1050] testbench.a_master BFM: Sending New WRITE burst: vr_axi_master_driven_burst-@293
kind:FIXED address:0xf2340a96012c2310 id:0x003f3 len:0x3 size:HALFWORD
```

7.2.2. Simple WRAP Read Burst

This example shows how to send a simple WRAP read burst. The burst start address is br ‘h3100 and it uses tag of 15 and size of byte.

```
    `uvm_do_with(burst,
    {burst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
    burst.Kind == DENALI_CDN_AXI_BURSTKIND_WRAP;
    burst.StartAddress == 'h3100;
    burst.IdTag == 15;
    burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_BYTE;
   });
```

Example of the output in the log:

```
UVM_INFO /myenv/axi3/cdnAxiUvmUserMasterDriver.sv(66) @ 4050:
uvm_test_top.axi_sve.myUvmEnv.activeMaster.driver [cdnAxiUvmUserMasterDriver]
Adding transaction to UVC Transmission Queue with Unique ID = 1
…
[4250] testbench.a_master BFM: Sending New READ burst: vr_axi_master_driven_burst-@317
kind:WRAP address:0x0000000000003100 id:0x0000f len:0x8
size:BYTE
```

7.2.3. Write Burst with Specific Data

This example shows how to send a random write burst with specific data.

Note that before assigning Data, the Data array must be properly sized

```
     `uvm_do_with(burst, {
               burst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
```

```
                burst.Length == 4;
                burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
                burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD;
                burst.IdTag == 4;
                burst.StartAddress == 'h2000;
                // Need to define the require amount of Data. The Data is an array of bytes, so
                // the size is number of byte in a transfer (2 for HALFWORD) * Length
                 burst.Data.size() == 8;
                burst.Data[0] == 'h0;
                burst.Data[1] == 'h11;
                burst.Data[2] == 'h22;
                burst.Data[3] == 'h33;
                burst.Data[4] == 'h44;
                burst.Data[5] == 'h55;
                burst.Data[6] == 'h66;
                burst.Data[7] == 'h77;
        })
```

Example of the output in the log:

```
UVM_INFO /myenv/axi3/cdnAxiUvmUserMasterDriver.sv(66) @ 6000:
uvm_test_top.axi_sve.myUvmEnv.activeMaster.driver [cdnAxiUvmUserMasterDriver]
Adding transaction to UVC Transmission Queue with Unique ID = 5
…
 [6050] testbench.a_master BFM: Sending New WRITE burst:
vr_axi_master_driven_burst-@349  kind:INCR address:0x0000000000002000 id:0x00004 len:0x4
size:HALFWORD
```

7.2.4. Write Non Random Burst with Specific Data

This example shows how to send a non random burst with specific fields set, including the data field .

```
  virtual task body();
      // allocate the burst using the common factory and initialize its properties
      `uvm_create(burst);
      // setting the required data items
      burst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
      burst.Length = 3;
      burst.Size = DENALI_CDN_AXI_TRANSFERSIZE_BYTE;
      burst.Kind = DENALI_CDN_AXI_BURSTKIND_FIXED;
      // initializing the data according to the required Length
      burst.Data = new[3];
      burst.Data[0] = 'h12;
      burst.Data[1] = 'h34;
      burst.Data[2] = 'h56;
      burst.StartAddress = 'h1000;
      // sending the burst to the sequencer without allocation and randomization
      `uvm_send(burst);
  endtask : body
```

Example of the output in the log:

```
UVM_INFO /myenv/axi3/cdnAxiUvmUserMasterDriver.sv(66) @ 2000:
uvm_test_top.axi_sve.myUvmEnv.activeMaster.driver [cdnAxiUvmUserMasterDriver]
Adding transaction to UVC Transmission Queue with Unique ID = 1
…
[2050] testbench.a_master BFM: Sending New WRITE burst: vr_axi_master_driven_burst-@302
kind:FIXED address:0x0000000000001000 id:0x00009 len:0x3 size:BYTE
```

7.2.5. Read After Write Burst

This example shows how to send a Write burst with 4 transfers to address ‘h2000. After a short wait, it sends a Read burst to the same address with the same Length.

```
    `uvm_do_with(burst, {
              burst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
              burst.Length == 4;
              burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
              burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD;
              burst.StartAddress == 'h2000;
              // Need to define the require amount of Data. The Data is an array of bytes, so
              // the size is number of byte in a transfer (2 for HALFWORD) * Length
              burst.Data.size() == 8;
              burst.Data[0] == 'h0;
              burst.Data[1] == 'h11;
              burst.Data[2] == 'h22;
              burst.Data[3] == 'h33;
              burst.Data[4] == 'h44;
              burst.Data[5] == 'h55;
              burst.Data[6] == 'h66;
              burst.Data[7] == 'h77;
       })
       #5000;
       `uvm_do_with(burst, {
              burst.Direction ==  DENALI_CDN_AXI_DIRECTION_READ;
              burst.Length == 4;
              burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
              burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD;
              burst.StartAddress == 'h2000;
       })
```

Example of the output in the log:

```
[6050] testbench.a_master BFM: Sending New WRITE burst: vr_axi_master_driven_burst-@341
kind:INCR address:0x0000000000002000 id:0x0099c len:0x4 size:HALFWORD
…
[11050] testbench.a_master BFM: Sending New READ burst: vr_axi_master_driven_burst-@370
kind:INCR address:0x0000000000002000 id:0x02c8d len:0x4 size:HALFWORD
```

7.2.6. Burst with Delays

In the CDN_AXI VIP, you can control the delays between the bursts and transfers. The following example shows how to send a write Burst with 0 delay on the AWVALID signal, and 0 delay on the WVALID signal before the first transfer. If there is no outstanding write bursts, sending such burst means that the AWVALID and the WVALID will be raised at the same cycle. If there are write transfers waiting to be sent, this will not happen.

See more information, refer to Section 4.3.4, *"Controlling Delays on AXI Channels"*".

```
       `uvm_do_with(burst, {
              burst.Direction ==  DENALI_CDN_AXI_DIRECTION_WRITE;
              burst.Length == 4;
              burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
              burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_HALFWORD;
              burst.StartAddress == 'h9000;
              burst.TransmitDelay == 0;
              burst.TransfersChannelDelay[0] == 0;
       })
```

Example of the output in the log (sending the burst and the transfer at the same cycle):

```
[1050] testbench.a_master BFM: Sending New WRITE burst: vr_axi_master_driven_burst-@293
kind:INCR address:0x0000000000009000 id:0x05857 len:0x4size:HALFWORD
[1050] testbench.a_master BFM: Sending New WRITE transfer:
```

```
vr_axi_master_driven_transfer-@294  address:0x0000000000009000 id:0x05857 last:FALSE
```

7.2.7. Exclusive Burst

This example shows how to send an Exclusive read burst and then an Exclusive write burst.

Note

Sending an Exclusive write without a matching Exclusive read will result in an error.

```
        `uvm_create(burst);
         // This pre-defined constraint is constraining the Access field to be Normal
         burst.normal_access_const.constraint_mode(0);

         `uvm_rand_send_with(burst, {
                burst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
                burst.Access == DENALI_CDN_AXI_ACCESS_EXCLUSIVE;
                burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
                burst.StartAddress == 'h7000;
                burst.IdTag == 10;
                burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
                burst.Cacheable == DENALI_CDN_AXI_CACHEMODE_CACHEABLE;
                burst.ReadAllocate == DENALI_CDN_AXI_READALLOCATE_NO_READ_ALLOCATE;
                burst.WriteAllocate == DENALI_CDN_AXI_WRITEALLOCATE_NO_WRITE_ALLOCATE;
                burst.Length == 2;
        });
        #10000;
        `uvm_create(burst);
         // This pre-defined constraint is constraining the Access field to be Normal
         burst.normal_access_const.constraint_mode(0);

        `uvm_rand_send_with(burst, {
                burst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
                burst.Access == DENALI_CDN_AXI_ACCESS_EXCLUSIVE;
                burst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
                burst.StartAddress == 'h7000;
                burst.IdTag == 10;
                burst.Size == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
                burst.Cacheable == DENALI_CDN_AXI_CACHEMODE_CACHEABLE;
                burst.ReadAllocate == DENALI_CDN_AXI_READALLOCATE_NO_READ_ALLOCATE;
                burst.WriteAllocate == DENALI_CDN_AXI_WRITEALLOCATE_NO_WRITE_ALLOCATE;
                burst.Length == 2;
        });
```

Example of the output in the log:

```
20050] testbench.a_master BFM: Sending New READ burst: vr_axi_master_driven_burst-@403
kind:INCR address: 0x0000000000007000 id:10 len:2 size:WORD
…
[30050] testbench.a_master BFM: Sending New WRITE burst:
vr_axi_master_driven_burst-@426  kind:INCR address: 0x0000000000007000  id:10 len:2 size:WORD
```

Example of a related error message that might appear in the log file:

```
ERROR: Trying to send exclusive write when there was no exclusive read with the same id
```

7.2.8. Lock Burst

This example shows how to send a locked burst, and then send a normal burst with the same parameters to unlock the address.

To send locked burst, you need to turn off the `normal_access_const` constraint. This constraint is setting the `Access` field to be not Exclusive and not Locked.

```
  // The following variables are defined outside the body. We will use them
  // to ensure that the second burst is using the same parameters as the first burst.
  rand reg [19:0] idTag;
  denaliCdn_axiPrivilegedModeT privileged;
  denaliCdn_axiSecureModeT secure;
  denaliCdn_axiFetchKindT dataInstr;
  denaliCdn_axiBufferModeT bufferable;
  denaliCdn_axiCacheModeT cacheable;
  denaliCdn_axiReadAllocateT readAllocate;
  denaliCdn_axiWriteAllocateT writeAllocate;

   …

       `uvm_create(burst);
    // This pre-defined constraint is constraining the Access field to be Normal
    burst.normal_access_const.constraint_mode(0);

    `uvm_rand_send_with(burst,
            {burst.Access == DENALI_CDN_AXI_ACCESS_LOCKED;
            burst.StartAddress == 'h3000;
    });

    bufferable = burst.Bufferable;
    cacheable = burst.Cacheable;
    readAllocate = burst.ReadAllocate;
    writeAllocate = burst.WriteAllocate;
    privileged = burst.Privileged;
    secure = burst.Secure;
    dataInstr = burst.DataInstr;
    idTag = burst.IdTag;

    #5000;

    `uvm_rand_send_with(burst,
            {burst.Access == DENALI_CDN_AXI_ACCESS_NORMAL;
            burst.StartAddress == 'h3000;
            burst.Bufferable == bufferable;
            burst.Cacheable == cacheable;
            burst.ReadAllocate == readAllocate;
            burst.WriteAllocate == writeAllocate;
            burst.Privileged == privileged;
            burst.Secure == secure;
            burst.DataInstr == dataInstr;
            burst.IdTag == idTag;
    });

    burst.normal_access_const.constraint_mode(1);
```

Example of the output in the log:

```
[18450] testbench.a_master BFM: Sending New READ burst: vr_axi_master_driven_burst-@404
kind:INCR address:0x0000000000003000 id:0x026e9 len:0x9 size:FOUR_WORDS
…
[19650] testbench.a_slave BFM: Sent READ transfer vr_axi_slave_driven_transfer_response-@407
address:0x0000000000003000 id:0x026e9 last:FALSE resp:OKAY
```

Examples of related error message that might appear in the log file:

```
ERROR: Trying to send exclusive burst while in a locked transaction

ERROR: burst id tag: 0x06a59 is different from current locked transaction id: 0x026e9
```

```
WARNING: burst start address: 0x0000000000007000 is not in the same4K region as the current
locked transaction(start of block): 0x0000000000003000
```

7.2.9. Unaligned Transfers

The CDN_AXI VIP supports unaligned transfers. For any burst that is made of data transfers wider than one byte, the first bytes accessed might be unaligned with the natural address boundary. For example, a 32-bit data packet that starts at a byte address of 0x1002 is not aligned to the natural 32-bit address boundary.

By default the CDN_AXI VIP will not send unaligned transfers based on a built-in SV constraint. To enable the VIP to generate such transfers you must turn-off the `constraintAligned` constraint:

```
masterBurst.constraintAligned.constraint_mode(0);
```

The following two examples show how to send unaligned transfers using SV UVM.

Example 1: Sending non random unaligned write burst.

Note

All the unspecified fields will be set by the VIP core.

```
`uvm_create(masterBurst);
masterBurst.constraintAligned.constraint_mode(0);
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.StartAddress = 'h2;
masterBurst.ModelGeneration = 1;
`uvm_send(masterBurst);
```

Example 2: Sending unaligned write burst using `uvm_do_with`

```
`uvm_create(masterBurst);
masterBurst.constraintAligned.constraint_mode(0);
`uvm_rand_send_with(masterBurst, {
   masterBurst.StartAddress == 'h7;
   masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
});
```

The following two examples show how to send unaligned transfers using SV.

Example 1: Sending non random unaligned write burst.

Note

All the unspecified fields will be set by the VIP core.

```
  masterBurst = new();
  masterBurst.constraintAligned.constraint_mode(0);
  masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
  masterBurst.StartAddress = 'h2;
  masterBurst.ModelGeneration = 1;
  status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Example 2: Sending unaligned write burst using randomize.

```
masterBurst = new();
masterBurst.constraintAligned.constraint_mode(0);
  assert(masterBurst.randomize() with {
    StartAddress == 'h7;
    Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
  });
status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Note

As per the *AMBA AXI Specification*, the unaligned transactions are relevant only for transactions of type INCR and FIXED.

7.2.10. Unaligned Data

The CDN_AXI VIP enables you to send transactions with unaligned data i.e. the data in the last transfer of the transaction can be partial. In that case, the strobe of the last transfer will be partial.

For example:

```
Data width 64bit, strobe 8bit
Address 0x100, Length 2, Size TWO_WORDS (8 bytes), kind INCR
Data: 9d d0 59 0f 91 76 41 ec 17 d5 72 63 (12 bytes)
```

The strobes will be:

1st transfer: 0xff

2nd transfer: 0x0f

By default the CDN_AXI VIP will not generate or allow such transactions based on a built-in SV constraint, in order to enable this VIP behavior, you must turn-off the `constraintAligned` constraint:

```
masterBurst.constraintAligned.constraint_mode(0);
```

When this constraint is turned off, the CDN_AXI VIP will generate and allow to randomize such transactions for INCR type only.

You can enable this behavior for WRAP and FIXED transactions by using `` `uvm_create() `` or `new()`.

Note the following:

- If you choose to create your transactions using `` `uvm_create `` or `new()` and you are setting the data in the `Data` field, you must provide the necessary fields:
  - StartAddress
  - Size
  - Kind

If these fields are not set, the transaction will be dropped.

- You must provide the minimal number of Data bytes based on the fields you set. That is, the last transfer should have at least one valid byte in it.

  For example:

  For INCR burst of length 3, address 0x0, Size WORD, there should be at least 9 bytes of Data. For an INCR burst of length 3, address 0x2, Size WORD, there should be at least 7 bytes of data. If the `Data` field is set but not enough data is provided the transaction will be dropped.

The following three examples show legal transactions: (data width is 256bits) using SV UVM.

Example 1: INCR burst with unaligned address and unaligned data.

```
`uvm_create(masterBurst);
masterBurst.constraintAligned.constraint_mode(0);
uvm_rand_send_with(masterBurst, {
   masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
   masterBurst.StartAddress == 2;
   masterBurst.Length == 3;
   masterBurst.Size == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
   masterBurst.Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
   masterBurst.ModelGeneration == 1;
   masterBurst.Data.size() == 8;
   masterBurst.Data[0] == 'hd;
   masterBurst.Data[1] == 'he;
   masterBurst.Data[2] == 'hf;
   masterBurst.Data[4] == 'h11;
   masterBurst.Data[5] == 'h12;
   masterBurst.Data[6] == 'h13;
   masterBurst.Data[7] == 'h14;
   });
```

Strobes will be: 0000000c 000000f0 00000300

Example 2: WRAP burst with unaligned data.

```
`uvm_create(masterBurst);
masterBurst.constraintAligned.constraint_mode(0);
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.StartAddress = 'h10;
masterBurst.Length = 2;
masterBurst.Size = DENALI_CDN_AXI_TRANSFERSIZE_FOUR_WORDS;
masterBurst.Kind = DENALI_CDN_AXI_BURSTKIND_WRAP;
masterBurst.Data = new[20];
for (int ii=0;ii<masterBurst.Data.size();ii++) begin
   masterBurst.Data[ii] = ('hd)+ii;
end
masterBurst.ModelGeneration = 1;
`uvm_send(masterBurst);
```

Strobes will be: ffff0000 0000000f

Example 3: FIXED burst with unaligned address and unaligned data.

```
`uvm_create(masterBurst);
```

```
masterBurst.constraintAligned.constraint_mode(0);
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.StartAddress = 2;
masterBurst.Length = 3;
masterBurst.Size = DENALI_CDN_AXI_TRANSFERSIZE_WORD;
masterBurst.Kind = DENALI_CDN_AXI_BURSTKIND_FIXED;
masterBurst.Data = new[5];
for (int ii=0;ii<masterBurst.Data.size();ii++) begin
   masterBurst.Data[ii] = ('hd)+ii;
end
//all fields that were not set by the user will be randomize in the
//VIP core. If you use `uvm_rand_send, then all the fields will be
//randomize according to the SV constraints
masterBurst.ModelGeneration = 1;

`uvm_send(masterBurst);
```

Strobes will be: 0000000c 0000000c 00000004

The following three examples show legal transactions: (data width is 256bits) using SV.

Example 1: INCR burst with unaligned address and unaligned data.

```
masterBurst = new();
masterBurst.constraintAligned.constraint_mode(0);
assert(masterBurst.randomize() with {
   Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
   StartAddress == 2;
   Length == 3;
   Size == DENALI_CDN_AXI_TRANSFERSIZE_WORD;
   Kind == DENALI_CDN_AXI_BURSTKIND_INCR;
   ModelGeneration == 1;
   Data.size() == 8;
   Data[0] == 'hd;
   Data[1] == 'he;
   Data[2] == 'hf;
   Data[3] == 'h10;
   Data[4] == 'h11;
   Data[5] == 'h12;
   Data[6] == 'h13;
   Data[7] == 'h14;
 });
status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Strobes will be: 0000000c 000000f0 00000300

Example 2: WRAP burst with unaligned data.

```
masterBurst = new();
masterBurst.constraintAligned.constraint_mode(0);
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.StartAddress = 'h10;
masterBurst.Length = 2;
masterBurst.Size = DENALI_CDN_AXI_TRANSFERSIZE_FOUR_WORDS;
masterBurst.Kind = DENALI_CDN_AXI_BURSTKIND_WRAP;
masterBurst.Data = new[20];
for (int ii=0;ii<masterBurst.Data.size();ii++) begin
   masterBurst.Data[ii] = ('hd)+ii;
end
masterBurst.ModelGeneration = 1;
status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Strobes will be: ffff0000 0000000f

Example 3: FIXED burst with unaligned address and unaligned data.

```
masterBurst = new();
masterBurst.constraintAligned.constraint_mode(0);
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.StartAddress = 2;
masterBurst.Length = 3;
masterBurst.Size = DENALI_CDN_AXI_TRANSFERSIZE_WORD;
masterBurst.Kind = DENALI_CDN_AXI_BURSTKIND_FIXED;
masterBurst.Data = new[5];
for (int ii=0;ii<masterBurst.Data.size();ii++) begin
   masterBurst.Data[ii] = ('hd)+ii;
end
//all the fields that were not set by the user will be randomize in the
//VIP core. If you use `uvm_rand_send, then all the fields will be
//randomize according to the SV constraints
masterBurst.ModelGeneration = 1;

status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Strobes will be: 0000000c 0000000c 00000004

7.2.11. Simple WriteUnique

This example shows how to use `uvm_do_with macro to randomize a WriteUnique write snoop transaction.

```
     `uvm_do_with(masterBurst, {
       masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
       masterBurst.WriteSnoop == DENALI_CDN_AXI_WRITESNOOP_WriteUnique;
       masterBurst.StartAddress == 'h100;
     })
```

Example of the output in the log:

```
[50050] testbench.a_master BFM: Sending New WRITE burst:
vr_axi_master_driven_burst-@306  kind:WRAP address:0x0000000000000100
id:0x03db7 len:0x8 size:BYTE domain:NON_SHAREABLE write_snoop:WriteUnique
```

7.2.12. Simple WriteNoSnoop

This example shows how to use `uvm_do_with macro to randomize a WriteNoSnoop write snoop transaction. In the ACE specification, the AWSNOOP value of WriteNoSnoop equals to the WriteUnique value (both 0b000). The only difference between these burst types is the Domain value. Therefore, there is only one enumerated value for these two values and there is no DENALI_CDN_AXI_WRITESNOOP_WriteNoSnoop value. To send WriteNoSnoop, you need to set WriteSnoop to DENALI_CDN_AXI_WRITESNOOP_WriteUnique and set the Domain field to DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE or DENALI_CDN_AXI_DOMAIN_NON_SYSTEM. If the Domain is set to DENALI_CDN_AXI_DOMAIN_INNER or DENALI_CDN_AXI_DOMAIN_OUTER, a WriteUnique burst will be sent.

```
    `uvm_do_with(masterBurst, {
       masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
       masterBurst.WriteSnoop == DENALI_CDN_AXI_WRITESNOOP_WriteUnique;
       masterBurst.Domain == DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE;
       masterBurst.StartAddress == 'h200;
```

```
    })
```

Example of the output in the log (a `WriteUnique` with domain of `NON_SHAREABLE` is sent):

```
[52050] testbench.a_master BFM: Sending New WRITE burst: vr_axi_master_driven_burst-@28
kind:FIXED address:0x0000000000000200 id:19727 len:9 size:TWO_WORDS domain:NON_SHAREABLE
write_snoop:WriteUnique
```

7.2.13. Simple ReadOnce

This example shows how to use the `` `uvm_do_with `` to send random `ReadOnce` snoop transaction with `StartAddress` smaller than ‘h100.

```
       `uvm_do_with(masterBurst, {
       masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
       masterBurst.ReadSnoop == DENALI_CDN_AXI_READSNOOP_ReadOnce;
       masterBurst.StartAddress < 'h100;
```

Example of the output in the log:

```
[52050] testbench.a_master BFM: Sending New READ burst:
vr_axi_master_driven_burst-@321  kind:INCR address:0x00000000000000ec
id:0x03a79 len:0xe size:WORD domain:NON_SHAREABLE read_snoop:ReadOnce
```

7.2.14. Simple ReadNoSnoop

This example shows how to use `` `uvm_do_with `` macro to randomize a `ReadNoSnoop` write snoop transaction. In the ACE specification, the `ARSNOOP` value of `ReadNoSnoop` equals to the `ReadOnce` value (both 0b000). The only difference between these burst types is the `Domain` value. Therefore, there is only one enumerated value for these two values and there is no `DENALI_CDN_AXI_READSNOOP_ReadNoSnoop` value. To send `ReadNoSnoop`, you need to set `ReadSnoop` to `DENALI_CDN_AXI_READSNOOP_ReadOnce` and set the `Domain` field to `DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE` or `DENALI_CDN_AXI_DOMAIN_NON_SYSTEM`. If the `Domain` is set to `DENALI_CDN_AXI_DOMAIN_INNER` or `DENALI_CDN_AXI_DOMAIN_OUTER`, a `ReadOnce` burst will be sent.

```
    `uvm_do_with(masterBurst, {
       masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
       masterBurst.ReadSnoop == DENALI_CDN_AXI_READSNOOP_ReadOnce;
       masterBurst.Domain == DENALI_CDN_AXI_DOMAIN_NON_SHAREABLE;
       masterBurst.StartAddress < 'h200;
     })
```

Example of the output in the log (a `ReadOnce` with domain of `NON_SHAREABLE` is sent):

```
[56050] testbench.a_master BFM: Sending New READ burst: vr_axi_master_driven_burst-@63
kind:INCR address:0x0000000000000120 id:18931 len:24 size:TWO_WORDS domain:NON_SHAREABLE
read_snoop:ReadOnce
```

7.2.15. Read Barrier

This example shows how to send a read barrier transaction. Note that by default the `ace_no_barrier_cont` constraint is turned on, and it prevents the active master from sending barriers.

To reduce randomization time, by default the `ace_barrier_const` constraint is turned off. This constraint is setting all required barrier fields to the correct value.

Refer to Section 7.2.16, "*Write Barrier*" for details on how to send write barrier.

```
    `uvm_create(masterBurst);
    //ace_no_barrier_const sets the IsBarrier to be DENALI_CDN_AXI_ISBARRIER_NOT_BARRIER
    masterBurst.ace_no_barrier_const.constraint_mode(0);
    //ace_barrier_const sets all the required barrier fields
    masterBurst.ace_barrier_const.constraint_mode(1);

    `uvm_rand_send_with(masterBurst, {
        masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
        masterBurst.ReadSnoop == DENALI_CDN_AXI_SNOOP_ReadOnce;
        masterBurst.IsBarrier == DENALI_CDN_AXI_ISBARRIER_BARRIER;
        masterBurst.Domain == DENALI_CDN_AXI_DOMAIN_INNER;
        masterBurst.Barrier == DENALI_CDN_AXI_BARRIER_MEMORY_BARRIER;
        masterBurst.Privileged == DENALI_CDN_AXI_PRIVILEGEDMODE_PRIVILEGED;
        masterBurst.Secure == DENALI_CDN_AXI_SECUREMODE_NONSECURE;
        masterBurst.DataInstr == DENALI_CDN_AXI_FETCHKIND_DATA;
        masterBurst.IdTag == 3;})

    // Turn off pre-defined constraints
    masterBurst.ace_no_barrier_const.constraint_mode(1);
    masterBurst.ace_barrier_const.constraint_mode(0);
```

Example of the output in the log:

```
54050] testbench.a_master BFM: Sending New READ barrier:
vr_axi_master_driven_burst-@341  kind:INCR address:0x0000000000000000
id:0x00003 len:0x1 size:TWO_WORDS domain:OUTER read_snoop:ReadOnce
[54150] testbench.a_master BFM: Sending New WRITE barrier:
vr_axi_master_driven_burst-@343  kind:INCR address:0x0000000000000000
id:0x00003 len:0x1 size:TWO_WORDS domain:OUTER write_snoop:WriteUnique
```

7.2.16. Write Barrier

This example shows how to send a write barrier transaction. Note that by default the `ace_no_barrier_cont` constraint is turned on, and it prevents the active master from sending barriers.

To reduce randomization time, by default the `ace_barrier_const` constraint is turned off. This constraint is setting all required barrier fields to the correct value.

Refer to Section 7.2.15, "*Read Barrier*" for details on how to send write barrier.

```
    `uvm_create(masterBurst);
    //ace_no_barrier_const sets the IsBarrier to be DENALI_CDN_AXI_ISBARRIER_NOT_BARRIER
    masterBurst.ace_no_barrier_const.constraint_mode(0);
    //ace_barrier_const sets all the required barrier fields
    masterBurst.ace_barrier_const.constraint_mode(1);

    `uvm_rand_send_with(masterBurst, {
        masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
        masterBurst.WriteSnoop == DENALI_CDN_AXI_WRITESNOOP_WriteUnique;
        masterBurst.IsBarrier == DENALI_CDN_AXI_ISBARRIER_BARRIER;
        masterBurst.Domain == DENALI_CDN_AXI_DOMAIN_INNER;
        masterBurst.Barrier == DENALI_CDN_AXI_BARRIER_MEMORY_BARRIER;
```

```
        masterBurst.Privileged == DENALI_CDN_AXI_PRIVILEGEDMODE_PRIVILEGED;
        masterBurst.Secure == DENALI_CDN_AXI_SECUREMODE_NONSECURE;
        masterBurst.DataInstr == DENALI_CDN_AXI_FETCHKIND_DATA;
        masterBurst.IdTag == 3;})

    // Turn off pre-defined constraints
    masterBurst.ace_no_barrier_const.constraint_mode(1);
    masterBurst.ace_barrier_const.constraint_mode(0);
```

Example of the output in the log:

```
[54150] testbench.a_master BFM: Sending New WRITE barrier:
vr_axi_master_driven_burst-@343  kind:INCR address:0x0000000000000000
id:0x00003 len:0x1 size:TWO_WORDS domain:OUTER write_snoop:WriteUnique
```

7.2.17. DVM message

This example shows how to send a DVM message. This message can be sent by a master only. By default, here the active master is not sending DVM transactions. Therefore, to send DVM message you must turn off the `ace_no_dvm_const`. This constraint makes sure that DVM will not be sent randomly.

To save randomization time, by default the `ace_dvm_const` constraint is turned off. Turn it on and the VIP will randomize the required DVM fields.

```
    `uvm_create(masterBurst);
    // the ace_no_dvm_const sets the transaction to be non DVM transaction
    masterBurst.ace_no_dvm_const.constraint_mode(0);
    // the ace_dvm_const sets the transaction fields to match DVM rules
    masterBurst.ace_dvm_const.constraint_mode(1);

    `uvm_rand_send_with(masterBurst, {
        masterBurst.Direction == DENALI_CDN_AXI_DIRECTION_READ;
        masterBurst.ReadSnoop == DENALI_CDN_AXI_READSNOOP_DVM_Message;
      })

    // Turn off pre-defined constraints
    masterBurst.ace_no_dvm_const.constraint_mode(1);
    masterBurst.ace_dvm_const.constraint_mode(0);
```

Example of the output in the log:

```
[64150] testbench.a_master BFM: Sending New READ burst:
vr_axi_master_driven_burst-@372  kind:INCR address:0xdde008d51f111000
id:0x04050 len:0x1 size:TWO_WORDS domain:INNER read_snoop:DVM_Message
```

7.3. Active Slave Test Scenarios

The examples in the following sections work with Master DUT or Master interface of an interconnect DUT

7.3.1. Response with BVALID Slave Delay

This example shows how to send a response to a write transaction with delay on the BVALID signal.

For more details, refer to Section 4.3.4, “*Controlling Delays on AXI Channels*”.

```
  `uvm_do_with(burst, {
      burst.Direction == DENALI_CDN_AXI_DIRECTION_WRITE;
      burst.ChannelDelay == 5;
   });
```

Example of the output in the log (note that the transaction was added to the slave response queue at time 1000):

```
[6150] testbench.a_slave BFM: Sending Response for Address phase to WRITE burst:
vr_axi_slave_driven_burst_response-@298  kind:INCR address:36864
id:22615 len:4 size:HALFWORD
```

7.3.2. How to Control Data in Slave Read Responses

There are three (3) ways to control data in slave read responses:

- Pre-populate the slave's sparse memory with a specific data pattern using backdoor writes as explained in Section 4.1.2.2, *"Backdoor Reads and Writes to Slave Memory"*.

  When the slave receives the read transaction, it returns a response with the data taken from the sparse memory model.

- Intercept the slave response in the `BeforeSendResponse` callback, modify the Data field, and put the modified response back on the stack for transmission using `transSet()`.

- Define the slave response with specific data, and place it in the slave's transmission queue.

  Make sure to set the `IgnoreConstraints` field to zero, as shown in the code snippet below. When the slave receives the read transaction, it will get the response from the transmission queue, then transmit it.

```
  `uvm_info(`gtn, "Pre-loading Slave Response", UVM_LOW);
    `uvm_create_on(slaveResp, p_sequencer.slave_seqr);

    `uvm_rand_send_with(slaveResp, {
              slaveResp.Direction == DENALI_CDN_AXI_DIRECTION_READ;
              slaveResp.Data.size() == 4;
              slaveResp.Data[0] == 'h12;
              slaveResp.Data[1] == 'h34;
              slaveResp.Data[2] == 'h56;
              slaveResp.Data[3] == 'h78;
              slaveResp.IgnoreConstraints == 0;
     })
```

7.3.3. Sending Snoop Bursts from the Slave

The VIP Active Slave can emulate an interconnect module. To do that, you can send snoop bursts. If you are using UVM, just use one of the UVM macros to send it (`` `uvm_do_with `` for example). If you are using `transAdd`, make sure you send the transaction to the snoop queue:

```
status = aceActiveSlave.transAdd(slaveSnoop, DENALI_CDN_AXI_QUEUE_Snoop);
```

To see how to generate a snoop burst, see Section 7.2, *"Active Master Test Scenarios"*.

7.4. Built-in SV Constraints

The CDN_AXI VIP has some built-in SV constraints to control the basic generation in the VIP. The SV constraints are located in `$DENALI/ddvapi/sv/denaliCdn_axi.sv`

You can turn *OFF* any of the built-in constraints by using `constraint_mode(0)` or turn *ON* a constraint by using `constraint_mode(1)`.

Example 7.1. AXI Locked and Exclusive constraint

By default the CDN_AXI VIP will not send exclusive and locked bursts based on the following constraint:

```
constraint normal_access_const {
      Access != DENALI_CDN_AXI_ACCESS_EXCLUSIVE;
      Access != DENALI_CDN_AXI_ACCESS_LOCKED;
}
```

To turn this constraint off and have the VIP generate all types of Access:

```
masterBurst.normal_access_const.constraint_mode(0);
```

Example 7.2. AXI Write Data before address constraint

By default the CDN_AXI VIP will not generate write data before address transactions based on the following constraint:

```
constraint DisableWriteAddressOffset_const {
    WriteAddressOffset == 0;
}
```

To turn this constraint off and have the VIP generate write data before address transactions:

```
masterBurst.DisableWriteAddressOffset_const.constraint_mode(0);
```

Example 7.3. Sending BARRIER transactions in ACE

There are two constraints related to sending *Barriers* in the simulation.

- `ace_barrier_const`: To send barrier transactions
- `ace_no_barrier_const`: To not to send barrier transactions.

  By default `ace_no_barrier_const` is turned *ON*, and `ace_barrier_const` is turned *OFF*, therefore the VIP will not generate barrier transactions.

  To enable sending of Barriers use the following:

  ```
  masterBurst.ace_no_barrier_const.constraint_mode(0);
  masterBurst.ace_barrier_const.constraint_mode(1);
  ```

Example 7.4. Sending DVM transactions in ACE

There are two constraints relating to sending DVM transactions in the simulation:

- `ace_dvm_const`: To send DVM transactions
- `ace_no_dvm_const`: To not to send DVM transactions

  By default `ace_no_dvm_const` is turned *ON*, and `ace_dvm_const` is turned *OFF*, therefore the VIP will not generate DVM transactions.

```
masterBurst. ace_no_dvm_const.constraint_mode(0);
masterBurst. ace_dvm_const.constraint_mode(1);
```

Note

The VIP will generate only DVM transactions if you activate DVM transactions.

7.5. Error Injection

By default, transactions will be generated with no errors. The CDN_AXI VIP lets you inject errors through using `transSet()` inside the different callbacks. When using `transSet()` in callbacks to modify field values in a transaction, the values will not be checked for validity before being sent.

Therefore, you can use this mechanism to purposely inject errors or set the values in `transSet()` correctly if you are not interested in error injection.

7.5.1. Injecting Wrong Strobe on a WRITE Transfer Using BeforeSendTransfer Callback

Here is an example that shows how to inject wrong strobe on a WRITE transfer using the `Before-SendTransfer` callback:

```
// Set the BeforeSendTransfer callback for the ACTIVE Master
void'(axiActiveMaster.setCallback(DENALI_CDN_AXI_CB_BeforeSendTransfer));

virtual function int BeforeSendTransferCbF(ref denaliCdn_axiTransaction trans);
if (trans.Type == DENALI_CDN_AXI_TR_WriteTransfer) begin
  trans.Strobe = 'h0f00ffff;
  void'(trans.transSet());
 end
return super.BeforeSendTransferCbF(trans);
endfunction

//send one transaction
masterBurst.Size = DENALI_CDN_AXI_TRANSFERSIZE_FOUR_WORDS;
masterBurst.Type = DENALI_CDN_AXI_TR_Write;
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
status = axiActiveMaster.transAdd(masterBurst,DENALI_CDN_AXI_QUEUE_Burst);
```

Error result: `ERR_VR_AXI158_WRONG_WSTRB` since the `wstrb` enabled bytes that are outside the valid range (5 bytes instead of 4 bytes).

7.5.2. Injecting Wrong Length on a WRITE Burst Using BeforeSend Callback

Here is an example that shows how to inject wrong length on a WRITE burst using the `BeforeSend` callback:

```
// Set the BeforeSend callback for the ACTIVE Master
void'(axiActiveMaster.setCallback(DENALI_CDN_AXI_CB_BeforeSend));

virtual function int BeforeSendCbF(ref denaliCdn_axiTransaction trans);
if (trans.Type == DENALI_CDN_AXI_TR_Write) begin
  trans.Length = 10;
  void'(trans.transSet());
 end
return super.BeforeSendCbF(trans);
endfunction

//send one transaction
masterBurst.Type = DENALI_CDN_AXI_TR_Write;
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
masterBurst.Length = 4;
status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Error result: `ERR_VR_AXI125_INVALID_WLAST` science the master drove `WLAST` high in the 4th transfer while the length is 10.

7.5.3. Injecting Wrong Transfer Response on a READ Burst Using BeforeSendResponse Callback

Here is an example that shows how to inject a wrong transfer response on a READ burst using the `BeforeSendResponse` callback:

```
// Set the BeforeSendResponse callback for the Active Slave
void'(axiActiveSlave.setCallback(DENALI_CDN_AXI_CB_BeforeSendResponse));

virtual function int BeforeSendResponseCbF(ref denaliCdn_axiTransaction trans);
 if (trans.Type == DENALI_CDN_AXI_TR_ReadData) begin
  trans.TransfersResp = new[3];
  trans.TransfersResp[0] = DENALI_CDN_AXI_RESPONSE_OKAY;
  trans.TransfersResp[1] = DENALI_CDN_AXI_RESPONSE_DECERR;
  trans.TransfersResp[2] = DENALI_CDN_AXI_RESPONSE_OKAY;
  void'(trans.transSet());
 end
 return super.BeforeSendResponseCbF(trans);
endfunction

//send one transaction
masterBurst.StartAddress = 0;
masterBurst.Type = DENALI_CDN_AXI_TR_Read;
masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_READ;
masterBurst.Length = 3;
status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
```

Error result: `ERR_VR_AXI231_READ_TRANSFER_MAPPED_ADDRESS_AND_DECERR` since the slave received a read transfer to a mapped address - 0 - but responded with `DECERR.`

7.5.4. Injecting Wrong Start Address on a SNOOP Burst Using BeforeSend Callback

Here is an example that shows how to inject wrong start address on a SNOOP burst using the `BeforeSend` callback:

```
// The Snoop Data Width is 4 bytes
// Set the BeforeSend callback for the Active Slave
void'(aceActiveSlave.setCallback(DENALI_CDN_AXI_CB_BeforeSend));

virtual function int BeforeSendCbF(ref denaliCdn_axiTransaction trans) ;
 if (trans.Type == DENALI_CDN_AXI_TR_Snoop) begin
  trans.StartAddress = 3;
  void'(trans.transSet());
 end
endfunction

// Send one transaction
slaveSnoop.StartAddress = 0;
slaveSnoop.Type = DENALI_CDN_AXI_TR_Snoop;
status = aceActiveSlave.transAdd(slaveSnoop, DENALI_CDN_AXI_QUEUE_Snoop);
```

Error result: `ERR_VR_AXI3230_ACADDR_NOT_ALIGNED_TO_DATA_WIDTH` since the `ACADDR` must be aligned to the snoop data transfer width.

7.5.5. Injecting Wrong CRRESP[2] PassDirty on a Snoop Response Using BeforeSendResponse Callback

Here is an example that shows how to inject wrong CRRESP[2] PassDirty on a snoop response using the `BeforeSendResponse` callback:

```
// Set the BeforeSendResponse callback for the Active Master
void'(aceActiveMaster.setCallback(DENALI_CDN_AXI_CB_BeforeSendResponse));

virtual function int BeforeSendResponseCbF(ref denaliCdn_axiTransaction trans) ;
 if (trans.Type == DENALI_CDN_AXI_TR_SnoopResponse) begin
  trans.RespPassDirty = DENALI_CDN_AXI_SNOOPRESPPASSDIRTY_PASS_DIRTY;
  void'(trans.transSet());
 end
endfunction

// Send one transaction
masterBurst.StartAddress = 0;
masterBurst.Type = DENALI_CDN_AXI_TR_Snoop;
masterBurst.SnoopType = DENALI_CDN_AXI_READSNOOP_ReadOnce;
```

Error result: `ERR_VR_AXI3122_WRONG_RESP_TO_TRANSACTION_TYPE` since the cache line state is `INVALID` the `CRRESP[2]` bit is illegal based on table 5-7 in the ACE spec.

7.5.6. Controlling Distribution of OKAY vs. SLVERR Responses for Write and Read Transactions in UVM

You can control slave's status response behavior via `BeforeSendResponse` callback. The following example shows how to control distribution of `OKAY` vs. `SLVERR` responses for write and read transactions in the UVM environment.

```
    // OKAY/SLVERR distribution of write responses
    int wr_resp_dist_threshold=80;

    // OKAY/SLVERR distribution of read transfer responses
    int rd_resp_dist_threshold=80;
    ...

    // ************************************************************
    // This function gets triggered by BeforeSendResponseCbPort import port
```

```
   // ************************************************************
   function void write(denaliCdn_axiTransaction trans);

   int wr_resp_dist;
   int rd_resp_dist;

   `uvm_info(`gtn, $sformatf("Slave response in BeforeSendResponse CB before modification: \n %0s",
    trans.sprint()), UVM_LOW)

   // Response distribution control for write transaction responses
   if (trans.Type == DENALI_CDN_AXI_TR_WriteResponse) begin
       wr_resp_dist = $urandom_range(1,100);

       if (wr_resp_dist > wr_resp_dist_threshold)
           trans.Resp = DENALI_CDN_AXI_RESPONSE_SLVERR;
       else
           trans.Resp = DENALI_CDN_AXI_RESPONSE_OKAY;


           // Push modified transaction back in the stack for transmission
           trans.transSet();

       `uvm_info(`gtn, $sformatf("Slave response in BeforeSendResponse CB after modification: \n
%0s", trans.sprint()), UVM_LOW)
       end


   // Response distribution control for read transaction responses
   if (trans.Type == DENALI_CDN_AXI_TR_ReadData) begin
               foreach (trans.TransfersResp[index]) begin
           rd_resp_dist = $urandom_range(1,100);
               if (rd_resp_dist > rd_resp_dist_threshold)
                   trans.TransfersResp[index] = DENALI_CDN_AXI_RESPONSE_SLVERR;
               else
                   trans.TransfersResp[index] = DENALI_CDN_AXI_RESPONSE_OKAY;
               end

           // Push modified transaction back in the stack for transmission
           trans.transSet();

       `uvm_info(`gtn, $sformatf("Slave response in BeforeSendResponse CB after modification: \n
%0s", trans.sprint()), UVM_LOW)
       end

   endfunction : write
```

Chapter 8. AXI VIP Testbench Integration

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

The AXI 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 AXI. The UVM layer is located at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi`. 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 AXI. The OVM layer is located at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/ovm/cdn_axi`. 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).

> **Note**
> When your DUT is coded in VHDL, add the `-vhdlsync` option to your `irun` command.

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 axi -
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 NCSim in three-step mode are shown below. The complete script with all steps is available at `$CDN_VIP_ROOT/tools/denali/example/cdn_axi/svExamples/simpleExample/example_setup_ncsim.csh`.

```
      #!/bin/csh -f
    #Step 1. Create cdn_vip_env_ncsim_sv.csh 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 axi
    #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_axi/
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 axi -
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_axi/svExamples/simpleExample/example_setup_irun.csh`.

```
     #!/bin/csh -f
    #Step 1. Create cdn_vip_env_irun_sv.csh 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 axi
    #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_axi/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` refers 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 AXI 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_axi.sv \
${CDN_VIP_ROOT}/tools/denali/example/cdn_axi/svExamples/simpleExample/activeMaster.v \
${CDN_VIP_ROOT}/tools/denali/example/cdn_axi/svExamples/simpleExample/activeSlave.v \
${CDN_VIP_ROOT}/tools/denali/example/cdn_axi/svExamples/simpleExample/tb.sv \
-incdir ${CDN_VIP_ROOT}/tools/denali/example/cdn_axi/svExamples/simpleExample \
-timescale 1ps/1ps -top pstest
```

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_axi_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_axi
```

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_axi/svExamples/simpleExample/example_setup_vcs.csh`.

```
     #!/bin/csh -f
    #Step 1. Create cdn_vip_env_vcs_sv.csh 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_axi/
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_axi/svExamples/simpleExample/example_setup_mti.csh`.

```
    #!/bin/csh -f
    #Step 1. Create cdn_vip_env_mti_sv.csh 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_axi/
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, callback processing, and so on.

8.2.1. Transaction Interface

8.2.1.1. Instance Class

8.2.1.1.1. Class denaliCdn_axiInstance

The `denaliCdn_axiInstance` corresponds to the AXI VIP instance instantiated in the testbench. Each such instance must have a corresponding `denaliCdn_axiInstance` 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_axiInstance inst;
inst = new(top.i0);
```

8.2.1.1.1.2. Methods

8.2.1.1.1.2.1. getInstName()

Syntax

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

Description

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

Example

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

8.2.1.1.1.2.2. setCbFuncName()

Syntax

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

Description

Sets the DPI callback function name.

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

Example

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

8.2.1.1.1.2.3. getCbFuncName()

Syntax

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

Description

Gets the DPI callback function name.

Example

```
denaliCdn_axiInstance inst;
inst = new("top.i0");
```

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

8.2.1.1.1.2.4. transAdd()

Syntax

```
function integer transAdd(denaliCdn_axi 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.

> **Note**
> In CDN_AXI, only one queue exists called DENALI_CDN_AXI_QUEUE_Burst

8.2.1.1.1.2.5. setCallback()

Syntax

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

Description

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

8.2.1.1.1.2.6. clearCallback()

Syntax

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

Description

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

8.2.1.1.1.2.7. getQueuePending()

Syntax

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

Description

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

Example

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

8.2.2. Coverage Interface / AXI VIP Coverage

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

Coverage can be used in specific instances in your SystemVerilog or SystemVerilog UVM testbench.

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

8.2.2.1. Coverage Class

The denaliCdn_axiCoverageInstance class is instantiated under the AXI VIP instance when you 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 AXI 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 receives, it triggers the relevant covergroup sampling.

8.2.2.1.2. Coverage Definitions

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

8.2.2.2. Coverage File Structure

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

Table 8.1. Coverage File Structure

Filename	Description
denaliCdn_axiCoverage.svh	This file contains the VIP coverage definitions and the trigger for the covergroups sampling.
denaliCdn_axiCoverageInstance.sv	This file contains definition of the coverage class.
denaliCdn_axiCoverGroupNew.svh	This file contains new actions for the covergroups.
denaliCdn_axiCoverGroupEnable.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

To enable coverage collection in your simulation, do the following:

1. Enable the coverage option in the simulator (NCSim only). See Section 8.2.2.3.1, *"Enable the Coverage Option in the NCSim Simulator"*.
2. Enable coverage collection:

   For the SystemVerilog interface, see Section 8.2.2.3.2, *"Enabling Coverage Collection Using SystemVerilog"*.
   For the SystemVerilog for UVM, see Section 8.2.2.3.3, *"Enabling Coverage Collection Using UVM SystemVerilog"*.

   Note
   By default, in UVM, coverage collection is enabled in all passive agents, and disabled in all active agents.

8.2.2.3.1. Enable the Coverage Option in the NCSim Simulator

In your **run** command, include all flags required by the simulator to collect coverage.

For example, you must provide the following flags:

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

8.2.2.3.2. Enabling Coverage Collection Using SystemVerilog

After you instantiate the relevant denaliCdn_axiInstance using its `new` function, call the `create_cover()` function for the instance. See the code example in Section 8.2.4, *"Example Testcase"*.

8.2.2.3.3. Enabling Coverage Collection Using UVM SystemVerilog

To enable your UVM monitor to collect coverage, set the **coverageEnable** bit to 1.

Syntax:

```
set_config_int("<env_name>.<agent_name>.monitor", "coverageEnable",<value>);
```

where:

- **<env_name>** is the name of your environment.
- **<agent_name>** is the name of the agent for which you want to enable or disable coverage collection.
- **<value>** is 0 to disable coverage collection, or 1 to enable coverage collection.

Example 1:

```
set_config_int("my_environment.cdn_axi_ActiveAgent2.monitor", "coverageEnable",1);
```

This example enables coverage collection for the agent named **cdn_axi_ActiveAgent2**.

Example 2:

```
set_config_int("*", "coverageEnable",1);
```

This example enables coverage collection for all agents.

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, there is an enable field called **<covergroup name>_enable**. 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 Interface

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 that you do not need:

1. Create your own coverage class that inherits from the given coverage class (denaliCdn_axiCoverageInstance).
2. In this class, implement the **userDisableCoverGroups()** method to set the relevant cover group ***_enable** field to 0, and by that disable the equivalent covergroups.
3. Create your own VIP instance that inherits from the given VIP instance (denaliCdn_axiInstance).
4. In this VIP instance, re-implement the **new_cover_class()** method to **new** the **coverInst** field with the coverage class you created instead of 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 (cdnAxiUvmCoverage)
2. In this class:
   a. Define the required cover groups using the denaliCdn_axiTransaction field (defined in the base coverage class), and the cdnAxiInstance 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 cdnAxiUvmMonitor, 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_axiCoverageInstance)
2. In this class:
   a. Define the required cover groups using the denaliCdn_axiCoverageInstance class denaliCdn_axiTransaction field, and the denaliCdn_axiInstance 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_axiInstance).
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_axiInstance callbacks to activate your covergroups sample().

8.2.3. SystemVerilog File Structure

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

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

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

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

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

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

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

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

- `$DENALI/ddvapi/sv/denaliCdn_axiSvIf.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 ob-

ject 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 ???.
2. Use PureView and the same configuration to create an HDL instantiation interface for those instances. For details, refer to ???.
3. Create the top level and instantiate the CDN_AXI 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 the DDVAPI ACE SV interface and the generic Mem inetrface
import DenaliSvCdn_axi::*;
import DenaliSvMem::*;

class myTransaction extends denaliCdn_axiTransaction;

 function new();
  super.new();
  this.SpecVer = DENALI_CDN_AXI_SPECVERSION_AMBA3;
  this.SpecSubtype = DENALI_CDN_AXI_SPECSUBTYPE_BASE;
  this.SpecInterface = DENALI_CDN_AXI_SPECINTERFACE_FULL;
   endfunction

    constraint user_dut_information {

        (StartAddress >= 'h0) && (StartAddress <= 'h1FFF);
        BurstMaxSize == DENALI_CDN_AXI_TRANSFERSIZE_EIGHT_WORDS;
        IdTag < (1 << 14);
      Cacheable == DENALI_CDN_AXI_CACHEMODE_NON_CACHEABLE;

      ModelGeneration == 0;
    }

endclass

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

    function new(string instName);
        super.new(instName);
```

```
    endfunction


    virtual function int DefaultCbF(ref denaliCdn_axiTransaction trans);

     if (trans != null && trans.Callback == DENALI_CDN_AXI_CB_CoverageSample) return
 super.DefaultCbF(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_AXI_CB_Error) begin
                $display("DefaultCbF :  ErrorInfo:  ",trans.ErrorInfo);
                $display("DefaultCbF :  ErrorId:  ",trans.ErrorId);
            end
        end

        return super.DefaultCbF(trans);
    endfunction

endclass

module pstest;

        reg aclk;
        reg aresetn;
        integer status;

  wire awvalid;
  wire [63:0] awaddr;
  wire [3:0] awlen;
  wire [2:0] awsize;
  wire [1:0] awburst;
  wire [1:0] awlock;
  wire [3:0] awcache;
  wire [2:0] awprot;
  wire [14:0] awid;
  wire awready;
  wire [255:0] awuser;
  wire wvalid;
  wire wlast;
  wire [255:0] wdata;
  wire [31:0] wstrb;
  wire [14:0] wid;
  wire wready;
  wire [255:0] wuser;
  wire bvalid;
  wire [1:0] bresp;
  wire [14:0] bid;
  wire bready;
  wire [255:0] buser;
  wire arvalid;
  wire [63:0] araddr;
  wire [3:0] arlen;
  wire [2:0] arsize;
  wire [1:0] arburst;
  wire [1:0] arlock;
  wire [3:0] arcache;
  wire [2:0] arprot;
  wire [14:0] arid;
  wire arready;
  wire [255:0] aruser;
  wire rvalid;
```

```
wire rlast;
wire [255:0] rdata;
wire [1:0] rresp;
wire [14:0] rid;
wire rready;
wire [255:0] ruser;

      always #50 aclk = ~aclk;

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

      activeMaster axiMasterDevice(aclk,
   aresetn,
   awvalid,
   awaddr,
   awlen,
   awsize,
   awburst,
   awlock,
   awcache,
   awprot,
   awid,
   awready,
   awuser,
   wvalid,
   wlast,
   wdata,
   wstrb,
   wid,
   wready,
   wuser,
   bvalid,
   bresp,
   bid,
   bready,
   buser,
   arvalid,
   araddr,
   arlen,
   arsize,
   arburst,
   arlock,
   arcache,
   arprot,
   arid,
   arready,
   aruser,
   rvalid,
   rlast,
   rdata,
   rresp,
   rid,
   rready,
   ruser);

      activeSlave axiSlaveDevice(aclk,
         aresetn,
         awvalid,
         awaddr,
```

```
          awlen,
          awsize,
          awburst,
          awlock,
          awcache,
          awprot,
          awid,
          awready,
          awuser,
          wvalid,
          wlast,
          wdata,
          wstrb,
          wid,
          wready,
          wuser,
          bvalid,
          bresp,
          bid,
          bready,
          buser,
          arvalid,
          araddr,
          arlen,
          arsize,
          arburst,
          arlock,
          arcache,
          arprot,
          arid,
          arready,
          aruser,
          rvalid,
          rlast,
          rdata,
          rresp,
          rid,
          rready,
          ruser);

      axiInstance axiActiveMaster;
      axiInstance axiActiveSlave;

      //myTransaction extends denaliCdn_axiTransaction with DUT specfic constraints
      myTransaction masterBurst;
      myTransaction slaveResp;

  initial
  begin
      $display("Going to create new instances ");
      axiActiveMaster = new ("pstest.axiMasterDevice");
      axiActiveSlave = new ("pstest.axiSlaveDevice");
      $display("Created new instances");

      void'(axiActiveMaster.setCallback(DENALI_CDN_AXI_CB_Error));
      void'(axiActiveMaster.setCallback(DENALI_CDN_AXI_CB_Ended));

      void'(axiActiveSlave.setCallback(DENALI_CDN_AXI_CB_Error));

      $display("Did the set call backs for the active instances");

      // Enable coverage collection
      //void'(axiActiveMaster.create_cover());
      //void'(axiActiveSlave.create_cover());

      //we can encaplsuate the logic with USER code to avoid writing too much code
      axiActiveMaster.mapMemorySegment(64'h0,64'h1FFF);
```

```
        axiActiveSlave.mapMemorySegment(64'h0,64'h1FFF);

        //set test verbosity to LOW
        //we can use the enum value as set in the denaliCdn_axiTypes.svh file
        axiActiveMaster.regWrite( DENALI_CDN_AXI_REG_Verbosity, DENALI_CDN_AXI_MESSAGEVERBOSITY_LOW );
        axiActiveSlave.regWrite( DENALI_CDN_AXI_REG_Verbosity, DENALI_CDN_AXI_MESSAGEVERBOSITY_LOW );

      axiActiveMaster.regWrite(DENALI_CDN_AXI_REG_ResetSignalsSimStart, 1);
        axiActiveSlave.regWrite(DENALI_CDN_AXI_REG_ResetSignalsSimStart, 1);

        axiActiveMaster.regWrite(DENALI_CDN_AXI_REG_WriteDataInterleavingDepth, 6);
        axiActiveSlave.regWrite(DENALI_CDN_AXI_REG_WriteDataInterleavingDepth, 6);

        masterBurst = new();
        slaveResp = new();

        #1000;

        //send one transaction
        masterBurst.StartAddress = 0;
        masterBurst.Direction = DENALI_CDN_AXI_DIRECTION_WRITE;
        status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);
        #1000;

        //you can send several transactions in a loop
        for (int i=0; i<50; i++) begin
            assert(masterBurst.randomize() with {
              Type inside {DENALI_CDN_AXI_TR_Read, DENALI_CDN_AXI_TR_Write};
                StartAddress >= 'h0;
                StartAddress <= 'h1FFF;
            });
            status = axiActiveMaster.transAdd(masterBurst, DENALI_CDN_AXI_QUEUE_Burst);

            #100;

        end

        #500000;

        $finish;

    end

endmodule
```

8.3. SystemVerilog Interface for UVM

This section provides details on how to integrate AXI VIP into a UVM compliant test environment.

The AXI VIP UVM layer is located under `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi`. All UVM layer classes are defined under the `cdnAxiUvm` package.

8.3.1. Prerequisites

The prerequisites to integrate AXI 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 AXI 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_axi/axi_uvm_sv_ref/index.html`

The Cadence UVM Layer supports UVM version 1.1 and supports simulators NCSim 11.10.072 or later, 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 AXI 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 axi -
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_axi/examples/axiSimpleExample/example_setup_ncsim.csh`.

```
   #!/bin/csh -f
#Step 1. Create cdn_vip_env_ncsim_sv_uvm.csh 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 axi
    #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_axi/examples/axiSimpleExample -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 axi -
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 NC-Sim 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_axi/examples/axiSimple-Example/example_setup_irun.csh`.

```
     #!/bin/csh -f
    #Step 1. Create cdn_vip_env_irun_sv_uvm.csh 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 axi
    #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_axi/examples/axiSimpleExample -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).

8.3.2.1.1. Single-Step Mode with Incremental Elaboration

Exact script invocations demonstrating incremental elaboration 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_axi/examples/axiIncrIrunExample/example_setup_irun_incr.csh`.

```
      #!/bin/csh -f
    #Step 1. Create cdn_vip_env_irun_sv_uvm.csh 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 axi
    #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_incr -e ${CDN_VIP_ROOT}/tools/denali/ddvapi/
sv/uvm/cdn_axi/examples/axiIncrIrunExample -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)
    #Important: This script contains 2 irun commands; The second irun command
    #can be executed repeatedly (incremental elab) for multiple tests
./cdn_vip_run_irun_incr_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_axi/examples/axiSimpleExample/example_setup_vcs.csh`.

```
      #!/bin/csh -f
    #Step 1. Create cdn_vip_env_vcs_sv_uvm.csh 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_axi/examples/axiSimpleExample -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_axi/examples/axiSimpleExample/example_setup_mti.csh`.

```
      #!/bin/csh -f
    #Step 1. Create cdn_vip_env_mti_sv_uvm.csh 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_axi/examples/axiSimpleExample -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
```

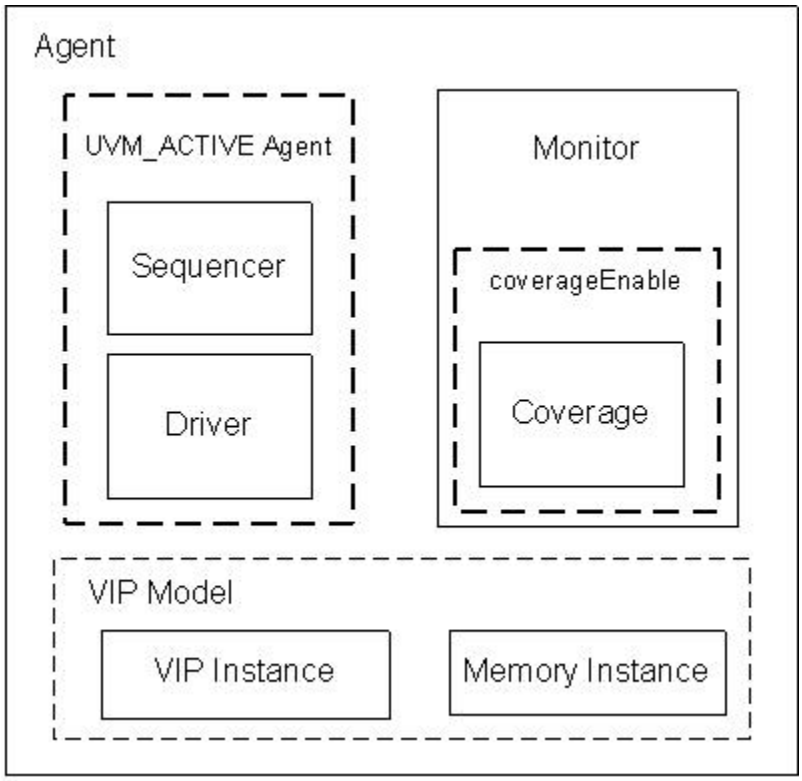
Please confirm the following two requirements are met before executing the above scripts:

- 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 AXI VIP UVM agent architecture.

Figure 8.1. The AXI VIP UVM Agent Architecture



The AXI 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 AXI VIP UVM agent is of type cdnAxiUvmAgent and inherits from uvm_agent. The agent instantiates monitor (of type cdnAxiUvmMonitor with a reference name **monitor**), driver (of type cdnAxiUvmDriver with a reference name **driver**), Seqeuncer (of type cdnAxiUvmSequencer with a reference name **sequencer**), instance (of type cdnAxiUvmInstance with a reference name **inst**) and memory instance (of type cdnAxiUvmMemInstance 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 `cdnAxiUvmUserSve.sv` file.

Refer to the CDN_AXI VIP UVM agent example at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserAgent.sv`, the CDN_AXI VIP UVM Master agent example at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserMasterAgent.sv` and the CDN_AXI VIP UVM Slave agent example at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserSlaveAgent.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 cdnAxiUvmCoverage with reference name coverModel), the coverage is instantiated only when coverageEnable is set to 1.

The AXI VIP monitor is of type cdnAxiUvmMonitor 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 AXI VIP Callbacks, refer to Chapter 5, *AXI VIP Callbacks*.

For more information on how to use the AXI monitor, refer to the AXI VIP UVM example at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/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 AXI VIP UVM driver is of type cdnAxiUvmDriver and inherits from uvm_driver. It includes the predefined implementation of the `run_phase()` method. For details about the Cadence UVM driv-

er implementation and predefined methods refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/cdnAxiUvmDriver.sv`

For details on how to use the different CDN_AXI VIP UVM drivers, refer to the examples at `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserMasterDriver.sv` and`$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserSlaveDriver.sv.`

The AXI 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 AXI VIP UVM sequencer is of type cdnAxiUvmSequencer and inherits from uvm_sequencer. The AXI VIP UVM sequencer's sequence item is of type: denaliCdn_axiTransaction.

Refer to section on Transaction Interface in Section 8.2.1, "*Transaction Interface*" for more information on the AXI 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 AXI VIP UVM Sequence is of type cdnAxiUvmSequence and inherits from uvm_sequence. The AXI VIP UVM sequence's data item is of type `denaliAxiTransaction`.

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

For details on how to use AXI VIP UVM Sequence, refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUvmUserSeqLib.sv` and`$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUvmUserVirtualSeqLib.sv`.

8.3.3.6. Transaction

The AXI VIP contains a UVM compliant data item `denaliCdn_axiTransaction` 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 AXI VIP UVM Sequence, refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxUvmUserSeqLib.sv` and`$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxiUvmUserVirtualSeqLib.sv`.

8.3.3.7. CDN_AXI Instance

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

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

For details on how to trigger callbacks using the denaliCdn_axiInstance instance, refer to `$CDN_VIP_ROOT/tools/denali/ddvapi/sv/uvm/cdn_axi/examples/axi3/cdnAxUvmUserEnv.sv`.

Note that setting the model callbacks, using `setCallback()` method (Section 8.2.1.1.1.2.5, *"setCallback()"*), 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 AXI VIP Callbacks, refer to Chapter 5, *AXI VIP Callbacks*.

8.3.3.8. Memory Instance

The AXI VIP UVM agent contains a reference to cdnAxiUvmMemInstance which inherits from denaliMemInstance class that provides an interface to access the AXI VIP model configuration and status registers.

8.3.3.9. Coverage

The AXI VIP UVM monitor contains a reference to cdnAxiUvmCoverage which inherits form denaliCdn_axiCoverageInstance class that contains AXI VIP coverage definitions.

For more information about denaliCdn_axiCoverageInstance, refer to Section 8.2.2, *"Coverage Interface / AXI 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 `cdnAxiUvmModifyTransactionSequence` 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 `cdnAxiUvmModifyTransactionSequence` perform the following steps:

1. Create your own sequence class which extends from the `cdnAxiUvmModifyTransaction-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 `cdnAxiModifyTransactionSequence` 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.

8.3.5. 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 AXI VIP UVM related files under `$DENALI/ddvapi/sv/uvm/cdn_axi`. The following table lists and describes UVM files.

Table 8.2. UVM Files

Filename	Description
cdnAxiUvmTop.sv	This file defines the cdnAxiUvm package and includes all the AXI UVM Layer files. To have

Filename	Description
	AXI UVM Layer you must compile this file and import the cdnAxiUvm package
cdnAxiUvmAgent.sv	This file defines the cdnAxiUvmAgent class.
cdnAxiUvmMonitor.sv	This file defines the cdnAxiUvmMonitor class, that includes all available events and analysis ports.
cdnAxiUvmDriver.sv	This file defines the cdnAxiUvmDriver class, that includes the default implementation of the driver run_phase method.
cdnAxiUvmSequencer.sv	This file defines the cdnAxiUvmSequencer.
cdnAxiUvmSequence.sv	This file defines the cdnAxiUvmSequence.
cdnAxiUvmInstance.sv	This file defines the cdnAxiUvmInstance, 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"*.
cdnAxiUvmMemInstance.sv	This file defines the cdnAxiUvmMemInstance, 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.
cdnAxiUvmCoverage.sv	This file defines the cdnAxiUvmCoverage, that includes coverage definitions and collection. For details on Coverage class refer to Section 8.2.2, *"Coverage Interface / AXI VIP Coverage"*.

8.3.7. UVM Flow

This section provides step-by-step details on the recommended flow that you can use to work with the UVM layer.

8.3.7.1. Creating the UVM Environment

To create the UVM environment:

1. Update the testbench file.

   a. Include and import the relevant packages in the testbench module.

```
`include "uvm_macros.svh"

// Test module
module testbench;

import uvm_pkg::*;
```

```
   // Import the DDVAPI CDN_AXI SV interface and the generic Mem interface
   import DenaliSvCdn_axi::*;
   import DenaliSvMem::*;

   // Include the VIP UVM base classes
   import cdnAxiUvm::*;

   // Include the User UVM classes and sequences
   `include "cdnAxiUvmUserTop.sv"

   // tests are here
   `include "cdnAxiUvmUserTest.sv"
```

b. Instantiate the VIP HDL instantiation interface in the testbench module.

```
// activeMaster is defined in the HDL instantiation interface generated using PureView

activeMaster
 a_master(aclk,aresetn,awvalid,awaddr,awlen,awsize,awburst,awlock,awcache,awprot,awid,
awready,awuser,wvalid,wlast,wdata,wstrb,wid,wready,wuser,bvalid,bresp,bid,bready,buser,arvalid,
araddr,arlen,arsize,arburst,arlock,arcache,arprot,arid,arready,aruser,rvalid,rlast,rdata,rresp,
rid,rready,ruser);

// activeSlave is defined in the hdl instantiation interface generated using PureView

activeSlave
 a_slave(aclk,aresetn,awvalid,awaddr,awlen,awsize,awburst,awlock,awcache,awprot,awid,
awready,awuser,wvalid,wlast,wdata,wstrb,wid,wready,wuser,bvalid,bresp,bid,bready,buser,arvalid,
araddr,arlen,arsize,arburst,arlock,arcache,arprot,arid,arready,aruser,rvalid,rlast,rdata,rresp,
rid,rready,ruser);

// passiveMaster is defined in the hdl instantiation interface generated using PureView

passiveMaster
 p_master(aclk,aresetn,awvalid,awaddr,awlen,awsize,awburst,awlock,awcache,awprot,awid,
awready,awuser,wvalid,wlast,wdata,wstrb,wid,wready,wuser,bvalid,bresp,bid,bready,buser,arvalid,
araddr,arlen,arsize,arburst,arlock,arcache,arprot,arid,arready,aruser,rvalid,rlast,rdata,rresp,
rid,rready,ruser);
```

c. Call `run_test()`.

```
module testbench;
…
 initial
   begin
    run_test();
   end
…
```

2. Instantiate the UVM agent in your environment.

   This example shows the instantiation of the user-specific agent class that is inherited from the UVM agent.

   a. Inherit from the UVM agent (optional).

```
class cdnAxiUvmUserAgent extends cdnAxiUvmAgent;
// your specific additions to the UVM agent
…

class cdnAxiUvmUserMasterAgent extends cdnAxiUvmUserAgent;
// Specific implementation for your master agent
…
```

b. Instantiate the agents in the environment.

```
class cdnAxiUvmUserEnv extends uvm_env;

  `uvm_component_utils(cdnAxiUvmUserEnv)
  // ************************************************************
  // The environment instantiates Master and Slave components
  // ************************************************************
  cdnAxiUvmUserMasterAgent activeMaster;
  cdnAxiUvmUserMasterAgent passiveMaster;
  cdnAxiUvmUserSlaveAgent activeSlave;
…
```

c. Set the `is_active` field and create the agent in the build phase.

```
virtual function void build_phase(uvm_phase phase);
    …
    set_config_int("activeMaster","is_active",UVM_ACTIVE);
    set_config_int("passiveMaster","is_active",UVM_PASSIVE);
    set_config_int("activeSlave","is_active",UVM_ACTIVE);

    // Active Master
    activeMaster = cdnAxiUvmUserMasterAgent::type_id::create("activeMaster", this);
    // Passive Master
    passiveMaster = cdnAxiUvmUserMasterAgent::type_id::create("passiveMaster", this);
    // Active Slave
    activeSlave = cdnAxiUvmUserSlaveAgent::type_id::create("activeSlave", this);class
…
```

d. Enable the relevant model callbacks, per agent instance, in the end_of_elaboration phase.

When you enable the model callbacks, it updates the UVM monitor through its analysis ports and events about internal activities.

```
function void end_of_elaboration_phase(uvm_phase phase);

    super.end_of_elaboration_phase(phase);
    …
    void'(activeMaster.inst.setCallback( DENALI_CDN_AXI_CB_BeforeSend));
    void'(activeMaster.inst.setCallback( DENALI_CDN_AXI_CB_BeforeSendTransfer));
    void'(activeMaster.inst.setCallback( DENALI_CDN_AXI_CB_Ended));
    …
```

3. Instantiate the environment class in the System Verification Environment (SVE). Unlike the `env` class, the SVE is not considered reusable and may change between different tests.

a. Instantiate the environment, and potentially your virtual sequencer in the SVE.

```
class cdnAxiUvmUserSve extends uvm_env;

  cdnAxiUvmUserEnv myUvmEnv;
  cdnAxiUvmUserVirtualSequencer vs;
…
```

b. Update the factory about your new classes in the constructor.

```
function new(string name = "cdnAxiUvmUserSve", uvm_component parent);
    super.new(name,parent);
    factory.set_type_override_by_type
       (cdnAxiUvmSequencer::get_type(), cdnAxiUvmUserSequencer::get_type());
```

```
    factory.set_type_override_by_type
       (cdnAxiUvmInstance::get_type(),cdnAxiUvmUserInstance::get_type());
    factory.set_type_override_by_type
       (cdnAxiUvmMonitor::get_type(),cdnAxiUvmUserMonitor::get_type());
    factory.set_type_override_by_type
       (cdnAxiUvmMemInstance::get_type(),cdnAxiUvmUserMemInstance::get_type());
    …
endfunction // new
```

c. Set the agent's `hdlPath` field and create the environment class.

```
  virtual function void build_phase(uvm_phase phase);
     …
    //set Passive Master and Passive Slave monitors to collect coverage
    set_config_int("myUvmEnv.passiveMaster.monitor", "coverageEnable",1);

    //set the full HDL path of the agent - this setting is mandatory.
    set_config_string("myUvmEnv.activeMaster","hdlPath", "testbench.a_master");
    set_config_string("myUvmEnv.passiveMaster","hdlPath", "testbench.p_master");
    set_config_string("myUvmEnv.activeSlave","hdlPath", "testbench.a_slave");
    …
    myUvmEnv = cdnAxiUvmUserEnv::type_id::create("myUvmEnv", this);
    vs = cdnAxiUvmUserVirtualSequencer::type_id::create("vs", this);
  endfunction
```

4. Create your tests.

a. Instantiate the SVE in the test and create it in the build phase.

```
  class cdnAxiUvmUserTest extends uvm_test;

  cdnAxiUvmUserSve axi_sve;
    …
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axi_sve = cdnAxiUvmUserSve::type_id::create("axi_sve", this);
  endfunction : build_phase
    …
```

b. Set the default sequence to the virtual sequencer or sequencers.

```
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    //set the starting sequence to system
    uvm_config_db#(uvm_object_wrapper)::set(this, "axi_sve.vs.run_phase",
 "default_sequence",simple_seq::type_id::get());
    …
  endfunction : build_phase
```

8.3.7.2. Creating Virtual Sequence

The virtual sequence is typically used to create system scenarios or scenarios that involve controlling more than one agent.

To create virtual sequencer and sequences:

1. Create a virtual sequencer with the pointers to the relevant sequencers that are used by the virtual sequences.

```
class cdnAxiUvmUserVirtualSequencer extends uvm_sequencer;
```

```
    cdnAxiUvmUserSequencer masterSeqr;
    cdnAxiUvmUserSequencer slaveSeqr;
     …
  endclass
```

2. Connect the sequencers to the pointers in the `connect_phase` function of the SVE class.

```
class cdnAxiUvmUserSve extends uvm_env;
  …
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    $cast(vs.slaveSeqr, myUvmEnv.activeSlave.sequencer);
    $cast(vs.masterSeqr, myUvmEnv.activeMaster.sequencer);
  endfunction
  …
endclass
```

3. Create the virtual sequences.

   a. Create the virtual sequence base class.

      Cadence recommends that you create a base class for all your virtual sequences to enable you to write functionality that applies for all sequences only one time.

      In the following example, raising and dropping objections is in the `pre_body` and `post_body` functions.

```
class cdnAxiUvmVirtualSequence extends uvm_sequence;

  `uvm_declare_p_sequencer(cdnAxiUvmUserVirtualSequencer)
   …
  virtual task pre_body();
    if (starting_phase != null) begin
      starting_phase.raise_objection(this);
    end
  endtask

  virtual task post_body();
    if (starting_phase != null) begin
      starting_phase.drop_objection(this);
    end
  endtask
 …
endclass : cdnAxiUvmVirtualSequence
```

   b. Use your virtual sequence base class and your sequence library to create virtual sequences.

```
// Read after write example sequence
class readAfterWriteSeq extends cdnAxiUvmVirtualSequence;

  cdnAxiUvmWriteSeq writeSeq;
  cdnAxiUvmReadSeq readSeq;

  rand reg [3:0] same_length;

  reg [63:0] same_address = 64'h0;
  denaliCdn_axiTransferSizeT same_size = DENALI_CDN_AXI_TRANSFERSIZE_WORD;
  denaliCdn_axiBurstKindT same_kind = DENALI_CDN_AXI_BURSTKIND_INCR;

  reg [7:0] w_data[];
```

```
 …

 virtual task body();

…

   // Send a Write following by a Read Transfer to the same address:
   `uvm_do_on_with(writeSeq, p_sequencer.masterSeqr, {
       writeSeq.address == same_address;
       writeSeq.length  == same_length;
       writeSeq.size    == same_size;
       writeSeq.kind    == same_kind;
   })

   // wait for write burst to end
   wait_for_EndedCbEvent();
…
   `uvm_do_on_with(readSeq, p_sequencer.masterSeqr, {
       readSeq.address == same_address;
       readSeq.length  == same_length;
       readSeq.size    == same_size;
       readSeq.kind    == same_kind;
   })
…
   // wait for read burst to end
   wait_for_EndedCbEvent();
…
 endtask // body

 //This task waits for Ended Callback to be triggered
 virtual task wait_for_EndedCbEvent();
    …
   p_sequencer.pEnv.activeMaster.monitor.EndedCbEvent.wait_trigger_data(obj);
…
 endtask
endclass
```

8.3.8. Generating a Test Case

To create a test case:

1. Refer to ???.
2. Create the top level and instantiate the AXI 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 AXI VIP UVM example illustrates how to build UVM compliant test environment AXI models and classes that are part of the VIPCAT installation.

8.3.9. Example Testcase

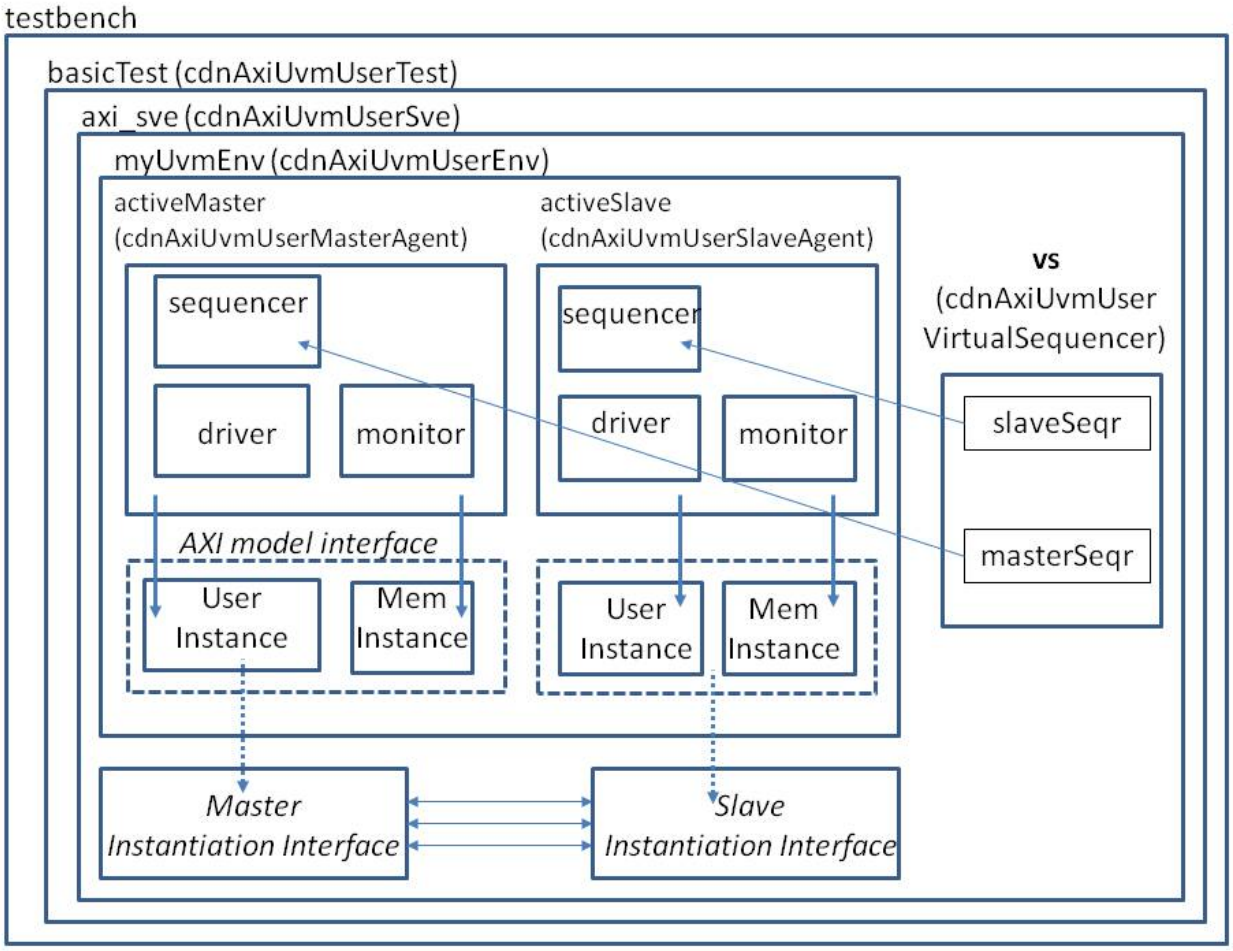
You can access the testbench examples located at `<VIPCAT>/tools/denali/ddvapi/sv/uvm/cdn_axi/examples`.

Each example contains the verification environment with instantiations of three agents - active master (mimicking DUT); passive master (to shadow DUT); and an active slave, 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_AXI VIP UVM example architecture.

Figure 8.2. Example Testcase Architecture



The following table describes the testcase example files.

Table 8.3. Testcase Example Files

File Name	Description
cdnAxiUvmUserTb.sv	This file creates a testbench, including model and instantiation interface.
cdnAxiUvmUserTest.sv	This file includes the definition of a basic test class. This test: - Creates the CDN_AXI VIP SVE. - Sets the default sequence for the virtual sequencer.

File Name	Description
	- Sets the log's verbosity levels. - Sends several random transactions.
cdnAxiUvmUserSve.sv	This file includes the definition of the CDN_AXI VIP UVM SVE class. This code does the following: - Instantiates the CDN_AXI VIP UVM SVE. - Instantiates the virtual sequencer. - Overrides the types of all the UVM layer base components to the user-specific classes - Sets the agent's 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 1 using `set_config_int()`.
cdnAxiUvmUserEnv.sv	This file includes the definition of the user-specific CDN_AXI environment. This code does the following: - Instantiates agents. In this example, activeMaster is considered as DUT. - Activates each agent's callbacks.
cdnAxiUvmUserMasterCfg.sv	This file includes the definition of the master configuration object. It enables you to set some of the master's configuration settings at the beginning of the run. It also enables the mapping of the master agent's address segments.
cdnAxiUvmUserSlaveCfg.sv	This file includes the definition of the Slave configuration object. It enables you to set some of the slave's configuration settings at the beginning of the run. It also enables the mapping of the slave agent's address segments.

File Name	Description
cdnAxiUvmUserAgent.sv	This file includes the definition of the user-specific CDN_AXI agent. In examples, this file implements the additional memory spaces to access the main memory space.
cdnAxiUvmUserMasterAgent.sv	This file includes the definition of the master agent component. This code does the following: - Instantiates the master's configuration object (Defined in the `cdnAxiUvmUserMasterCfg.sv` file) - Randomizes the configuration values according the user-defined constraints. - Sets the master agent's configuration by a set of register writes. - Sets the master address segment mapping.
cdnAxiUvmUserSlaveAgent.sv	This file includes the definition of the slave agent component. This code does the following: - Instantiates the slave's configuration object (defined in the `cdnAxiUvmUserSlaveCfg.sv` file) - Randomizes the configuration values according the user-defined constraints. - Sets the slave agent's configuration by a set of register writes. - Sets the slave address segment mapping.
cdnAxiUvmUserMonitor.sv	This file includes the definition of the user-specific CDN_AXI monitor. This example: - Instantiates a coverage model. - Connects the coverage model analysis important to the monitor relevant analysis ports.
cdnAxiUvmUserMasterDriver.sv	This file includes the definition of the user-specific CDN_AXI master's driver.
cdnAxiUvmUserSlaveDriver.sv	This file includes the definition of the user-specific CDN_AXI slave's driver.
cdnAxiUvmUserSequencer.sv	This file includes the definition of the user-specific CDN_AXI sequencer. The user sequencer in the example does not have any specific functionality.
cdnAxiUvmUserSeqLib.sv	This file includes an example of the user-specific sequence library.
cdnAxiUvmUserVirtualSequencer.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.
cdnAxiUvmUserVirtualSeqLib.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 AXI VIP? Is there a list of the programmable fields available?

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

9.1.2. How can I change the pin names in AXI 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_axi.sv	DENALI_SV_CDN_AXI_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 AXI, 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_axiSvIf.c` file into the DPI libraries before the compilation of corresponding SV files, which call functions like `denaliCdn_axiInit`.

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
```

9.2. CDN_AXI Specific FAQs

1. How can I change address, ID and data signal width?

   You can change them at time *0* of simulation by writing the desired value to the `DENALI_CDN_AXI_REG_ReadDataWidth, DENALI_CDN_AXI_REG_WriteDataWidth, DENALI_CDN_AXI_REG_AddrWidth, DENALI_CDN_AXI_REG_IdWidth, DENALI_CDN_AXI_REG_SnoopDataWidth` registers.

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
```

For this configuration, the `perform_cl73_an` field in SOMA can be enabled in each agent's SOMA file individually. The `cl73_an_physical_link_number` field in SOMA should be set to a different number for each agent.

10.2. Too many garbage collections slow simulation in 64-bit mode

In very big simulation environments, the default size of allocated heap memory might not be enough and cause frequent garbage collections. To avoid this, you can increase the default size of memory allocated for the machine by setting the .denalirc parameter MaxHeapSize.

You can view the heap memory size allocated for the simulation by issuing the following command from the simulator:

```
sn show conf mem -optimal_process_size
```

10.3. CDN_AXI Specific Troubleshooting

Table 10.1. Troubleshooting

Symptom	Reason	Solution

End of test summary says that the items were dropped in the test.
 	
OR
 	
You get the one of following messages in the log:
 	
`DENALI_CDN_AXI_`
 	
`WARNING_VR_AXI3190_`
 	
`WRITE_BURST_DOESNT_`
 	
`MATCH_THE_CACHE_`
	The master tried to send a burst that is illegal or it did not match the current cache state.
	
	Note that for bursts sent to a shareable domain that are longer than one cache line, all relevant cache lines must be legal in order to send this burst.
		1. Search in the log file for the following string "Couldn't create a legal Burst transaction from input Item is dropped".
		
		2. See the reason, and fix constraints according to it. For example: `ERROR: size is set to TWO_WORDS but data width is 3`.

Symptom	Reason	Solution
STATE_AND_DISCARDED. DENALI_CDN_AXI_ WARNING_VR_AXI3191_ READ_BURST_DOESNT_ MATCH_THE_CACHE_ STATE_AND_DISCARDED.		