Verification Continuum™

VC Verification IP LPDDR Memory UVM User Guide

Version Q-2020.06, June 2020

SYNOPSYS®

Contents

Preface

About This Manual

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

Web Resources

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

Customer Support

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

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

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

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

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

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

1 Introduction

1.1 Overview

The Synopsys LPDDR Verification IP supports verification of designs that include LPDDR memory. This document describes the use of LPDDR VIP in testbenches that comply with the SystemVerilog Universal Verification Methodology (UVM). The benefits include:

- ❖ Protocol functionality and abstraction
- ❖ Constrained random verification
- ❖ Functional coverage
- ❖ Rapid creation of complex tests
- ❖ Proven testbench architecture that provides maximum reuse, scalability and modularity
- ❖ Proven verification approach and methodology
- ❖ Transaction-level, self-checking tests
- ❖ Object oriented interface that allows OOP techniques.

This document assumes that you are familiar with the LPDDR protocol, object oriented programming, SystemVerilog, and UVM.

Synopsys provides a LPDDR UVM basic QuickStart for LPDDR2 ,LPDDR3 , LPDDR4 and LPDDR5 showing many features of both the UVM methodology, and tasks showing how to use the model. It is located at:

- ❖ `$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/examples/sverilog/tb_lpddr2_svt_uvm_basic_sys`
- ❖ `$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/examples/sverilog/tb_lpddr3_svt_uvm_basic_sys`
- ❖ `$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/examples/sverilog/tb_lpddr4_dual_ch_svt_uvm_basic_sys`
- ❖ `$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/examples/sverilog/tb_lpddr5_dual_ch_svt_uvm_basic_sys`

The README file describes how to run the examples.

1.2 Product Overview

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

1.3 LPDDR VIP Model Features

LPDDR VIP currently supports the following protocol features:

- LPDDR2 and LPDDR3 common features:
  - Eight internal banks for concurrent operation
  - Differential clocks (ck_t/ck_c) inputs.
  - Bidirectional/differential data strobe per byte of data (DQS/DQS#)
  - Burst WRITE/READ operations
  - Seamless burst WRITE/READ operations
  - Mode register WRITE/READ
  - Programmable READ and WRITE latencies (RL/WL)
  - Partial array self refresh (PASR)
  - Idle and active power down
  - Deep power down mode (DPD)
  - Clock stop capability
  - Frequency change capability
  - ZQ calibration
- LPDDR2 specific features:
  - Supports both LPDDR2 S2 and LPDDR2 S4 models
  - Clock rates up to 533 MHz
  - Density 64Mb to 2Gb
  - Data width 8,16,32 bits
  - BST (Burst Termination)
  - WRITE/READ transaction truncation
- LPDDR3 specific features:
  - Clock rates up to 1066MHz
  - Density 4Gb to 16Gb
  - Data width of 16, 32 bits
  - CA training mode
  - Write leveling
  - On Die Termination (ODT)
  - DQ Calibration

LPDDR4 VIP Model Features:

- Dual Channel Architecture
- Command Truth Table
  - Four Clock Architecture (ALL commands sampled as two commands like READING-1 and CAS-2 or MRR1 and CAS-2)
  - Tw
  - Clock Architecture (Commands like PD, Self Refresh, Refresh)
  - MPC Commands
    - Write Fifo
    - Read Fifo
    - MPC NOP
    - DQ Calib
    - ZQ Calib (Start and Latch)
  - MASKED WRITE Command
  - Seamless Read Write
  - Burst Read Write
  - Precharge
  - Refresh
  - Power Down
- DMI and DBI Feature
- Backdoor Mode register Access
- Trainings
  - CA Training
  - DQ Training (Read/Write Fifo and DQ Calib)
  - ZQ Calib
  - Write Leveling
- Target Row Refresh
- All Basic State Machines as Per LPDDR3
- PPR mode (Post Package Repair)
- Self Refresh Abort

LPDDR5 VIP Model Features:

- Dual Channel Architecture
- Command truth table

Power down Entry and Power down exit command, NOP Command

  - MPC Commands
    - Start WCK2DQI Interval Oscillator
    - Stop WCK2DQI Interval Oscillator

        - Start WCK2DQO Interval Oscillator
        - Stop WCK2DQO Interval Oscillator
        - ZQ Cal Start
        - ZQ Cal latch
- DMI and DBI Feature
- Backdoor Mode register Access
- Trainings
    - CA Training
    - DQ Training (Read/Write Fifo and DQ Calib)
    - ZQ Calib
    - WCK2CK Leveling Mode
- RDQS mode
- EDC Hold Pattern
- Deep sleep mode

2 Installation and Setup

2.1 Introduction

This section leads you through installing and setting up the VC VIP LPDDR. When you complete this checklist, the provided example testbench will be operational and the VC VIP LPDDR will be ready to use.

Note If you encounter any problems with installing the VC VIP LPDDR, see "Customer Support" on page 7.

2.2 Verifying the Hardware Requirements

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

- ❖ 400 MB available disk space for installation
- ❖ 16 GB Virtual Memory recommended
- ❖ FTP anonymous access to ftp.synopsys.com (optional)

2.3 Verifying Software Requirements

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

2.3.1 Platform/OS and Simulator Software

- ❖ Platform/OS and VCS: You need versions of your platform/OS and simulator that have been qualified for use. To see which platform/OS and simulator versions are qualified for use with the LPDDR VIP, check the support matrix for "SVT-based" VIP in the following document:

  Support Matrix for SVT-Based LPDDR VIP is in:

  VC LPDDR Release Notes

2.4 Synopsys Common Licensing (SCL) Software

- ❖ The SCL software provides the licensing function for the VC VIP LPDDR. Acquiring the SCL software is covered here in the installation instructions in "Licensing Information" on page 21.

2.5 Other Third Party Software

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

2.6 Preparing for Installation

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

   ```
   setenv DESIGNWARE_HOME absolute_path_to_designware_home
   ```

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

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

   - `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for Vera and Synopsys Common Licensing software or the *port@host* reference to this file.

     ```
     % setenv SNPSLMD_LICENSE_FILE $LM_LICENSE_FILE
     ```

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

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

2.7 Downloading and Installing

Attention The Electronic Software Transfer (EST) system only displays products your site is entitled to download. If the product you are looking for is not visible please contact est-ext@synopsys.com.

Follow the instructions below for downloading the software from Synopsys. You can download from the Download Center using either HTTPS or FTP, or with a command-line FTP session. If your Synopsys SolvNet password is unknown or forgotten, go to http://solvnet.synopsys.com.

Passive mode FTP is required. The passive command toggles between passive and active mode. If your FTP utility does not support passive mode, use http. For additional information, refer to the following web page:

https://www.synopsys.com/apps/protected/support/EST-FTP_Accelerator_Help_Page.html

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

1. Point your web browser to "https://solvnet.synopsys.com/DownloadCenter".
2. Enter your Synopsys SolvNet username.
3. Enter your Synopsys SolvNet password.
4. Click the "Sign In" button.
5. Choose the product name from the list of available products under "My Product Releases".
6. Select the product version from the list of available versions.
7. Click the "Download Here" button for HTTPS download.
8. After reading the legal page, click on "Yes, I agree to the above terms".
9. Click the download button(s) next to the file name(s) of the file(s) you wish to download.
10. Follow browser prompts to select a destination location.
11. You may download multiple files simultaneously.
12. Execute the run file:

    % <*vip run file name*>.run

    Answer the prompts that the.run script generates until the install is complete.

2.7.2 Downloading Using FTP with a Web Browser

1. Follow the above instructions through the product version selection step.
2. Click the "Download via FTP" link instead of the "Download Here" button.
3. Click the "Click Here To Download" button.
4. Select the file(s) that you want to download.
5. Follow browser prompts to select a destination location.
6. Execute the run file:

    % <*vip run file name*>.run

    Answer the prompts that the.run script generates until the install is complete.

2.8 Adding a Single VIP

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

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

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

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

The following command adds an model, <model_svt>, to the `design_dir directory`.

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

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

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

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

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

2.9 Adding Multiple VIPs

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

The examples in this chapter call that directory as `design_dir`, but you can use any name. In this example, assume you have the LPDDR-suite setup in the `design_dir` directory. In addition to the LPDDR VIP, you require the Ethernet and USB VIP suites.

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

Once installed, you must set up and locate the Ethernet and USB suites in the same `design_dir` location as AMBA. Use the following commands:

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

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

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

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

2.10 Updating and Deleting VIP Components

To update an existing model, perform the following steps:

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

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

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

   ```
   %unix> dw_vip_setup -path design_dir -add svt_lpddr_agent -v 3.50a model_vmt -v 3.50a
   ```

To delete an existing model, use the following command:

```
%unix> dw_vip_setup -path <design_dir> -remove <model>
```

Note
The command removes all the versions of the specified model from `design_dir`.

2.11 Running the Example with +incdir+

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

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

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

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

```
cd <testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/
// To run the example using the generated run script with +incdir+
./run_lpddr4_svt_uvm_basic_sys -verbose -incdir lpddr4_basic_wr_rd_test vcsvlog
```

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

```
vcs -l ./logs/compile.log -q -Mdir=./output/csrc
+define+DESIGNWARE_INCDIR=<DESIGNWARE_HOME> \
+define+SVT_LOADER_UTIL_ENABLE_DWHOME_INCDIRS
+incdir+<DESIGNWARE_HOME>/vip/svt/lpddr_svt/<vip_version>/sverilog/include \
-timescale=1ns/1ps
<testbench_dir>/lpddr/vip/svt/common/latest/C/lib/amd64/libmemserver.so \
<testbench_dir>/hbm/vip/svt/common/latest/C/lib/amd64/libmemserver.so \
+define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV +incdir+ <testbench_dir>
/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/../../e
nv \
```

```
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/../env
\
+incdir+<testbench_dir>examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/env \
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/dut \
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/hdl_int
erconnect \
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/lib \
+incdir+<testbench_dir>/examples/sverilog/lpddr_svt/tb_lpddr4_svt_uvm_basic_sys/tests \
-o ./output/simvcssvlog -f top_files -f hdl_files
```

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

2.12 Getting Help on Example Run/make Scripts

You can get help on the generated make/run scripts in the following ways:

1. Invoke the run script with no switches, as in:

```
run_lpddr4_svt_uvm_basic_sys
usage: run_lpddr4_svt_uvm_basic_sys [-32][-incdir][-verbose] [-debug_opts] [-
waves] [-
clean] [-nobuild] [-buildonly] [-norun] [-pa] <scenario> <simulator>
where <scenario> is one of: all backdoor_memory_memh_test
backdoor_memory_mif_test base_test lpddr4_basic_wr_rd_test
<simulator> is one of: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog
ncvlog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog
-32 forces 32-bit mode on 64-bit machines
-incdir use DESIGNWARE_HOME include files instead of design directory
-verbose enable verbose mode during compilation
-debug_opts enable debug mode for VIP technologies that support this
option
-waves [fsdb|verdi|dve|dump] enables waves dump and optionally
opens viewer (VCS only)
-seed run simulation with specified seed value
-clean clean simulator generated files
-nobuild skip simulator compilation
-buildonly exit after simulator build
-norun only echo commands (do not execute)
-pa invoke Verdi after execution
```

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

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

```
Valid simulators are: vcsmxvlog mtivlog vcsvlog vcszsimvlog vcsscvlog
ncvlog vcszebuvlog vcsmxpcvlog vcsvhdl vcsmxpipvlog ncmvlog vcspcvlog
Valid scenarios are: all backdoor_memory_memh_test
backdoor_memory_mif_test base_test callback_test lpddr4_ac_timings_test
lpddr4_basic_wr_rd_test lpddr4_dqs_noise_injection_test
lpddr4_rd_wr_same_bank_with_dbi_test lpddr4_temperature_derating_test
```

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

2.13 Include and Import VIP Files into Your Test Environment

After you set up models, you must include and import various files into your top testbench files to use the VIP. Following is a code snippet of the includes and imports for LPDDR:

```
/** Include LPDDR VIP package */
`include "svt_lpddr_full.uvm.pkg"

/** Import the required packages */

import uvm_pkg::*;
`include "uvm_macros.svh"

import svt_uvm_pkg::*;

import svt_mem_uvm_pkg::*;

import svt_lpddr_full_uvm_pkg::*;
```

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

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

Supported vendors are vcs, mti, and ncv. For example:

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

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

2.14 VIP Compile-Time and Runtime Options

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

`$DESIGNWARE_HOME/vip/svt/<model>/latest/examples/sverilog/<test_name>`

The following files contain the options:

- For compile-time options:

  `sim_build_options (also, vcs_build_options)`

- ❖ For runtime options:

  `sim_run_options (also, vcs_run_options)`

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

sim_build_options

```
Required:  +define+UVM_PACKER_MAX_BYTES=1500000
Required:  +define+UVM_DISABLE_AUTO_ITEM_RECORDING
Required:  +define+SYNOPSYS_SV
Required:  +define+SVT_MEM_DPI_OK
Required:
${DESIGNWARE_HOME}/vip/svt/common/latest/C/lib/${platform}/libmemserver.so
```

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

sim_run_options

2.15 Reporting Information About DESIGNWARE_HOME or a Design Directory

In these examples, the setup program sends output to STDOUT.

The following example lists the VC VIP LPDDR VIP libraries, models, example testbenches, and license version in a `DESIGNWARE_HOME` installation:

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

The following example lists the VC VIP LPDDR libraries, models, and license version in a testbench design directory:

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

2.16 Setting Up a Testbench Design Directory

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

The dw_vip_setup utility allows you to:

- ❖ Create the design directory (design_dir), which contains the transactors, support files (include files), and examples (if any)

  Add a specific version of VC VIP LPDDR LPDDR Memory from DESIGNWARE_HOME to a design directory

For a full description of dw_vip_setup, refer to "The dw_vip_setup Utility" on page 26.

To create a design directory and add a model so it can be used in a testbench, use the following command:

```
% $DESIGNWARE_HOME/bin/dw_vip_setup -path design_dir -a svt_lpddr_agent -svtb
```

The models provided with VC VIP LPDDR include:

- ❖ `svt_lpddr_agent`
- ❖ `svt_lpddr4_env`

2.17 Licensing Information

The VC VIP LPDDR uses the Synopsys Common Licensing (SCL) software to control its usage. You can find general SCL information at:

http://www.synopsys.com/keys

Attention Licensing is required if the VIP component classes are instantiated in the design. This includes envs, agents, drivers, monitors, sequencers, and components in UVM and OVM. This includes groups, subenvs, and transactors in VMM.

For LPDDR2:

❖ VIP-LPDDR-SVT+VIP-LPDDR2-SVT

❖ VIP-MEMORY-SVT

❖ VIP-SOC-LIBRARY-SVT

❖ VIP-LIBRARY-SVT + DesignWare-Regression

For LPDDR3:

❖ VIP-LPDDR-SVT+VIP-LPDDR3-SVT

❖ VIP-MEMORY-SVT

❖ VIP-SOC-LIBRARY-SVT

❖ VIP-LIBRARY-SVT + DesignWare-Regression

For LPDDR4:

❖ VIP-LPDDR-SVT+VIP-LPDDR4-SVT

❖ VIP-MEMORY-SVT+VIP-LPDDR4-SVT

❖ VIP-SOC-LIBRARY-SVT

❖ VIP-LIBRARY-SVT + DesignWare-Regression

For LPDDR5:

❖ VIP-LPDDR-SVT+VIP-LPDDR5-SVT

❖ VIP-SOC-LIBRARY-SVT

❖ VIP-LIBRARY-SVT + DesignWare-Regression

Only one license is consumed per simulation session, no matter how many VC VIP LPDDR VIP models are instantiated in the design.

The licensing key must reside in files that are indicated by specific environment variables. For information about setting these licensing environment variables, refer to "Environment Variable and Path Settings" on page 23.

2.17.1 If Licensing Fails

By default, simulations exit with an error when a VC VIP LPDDR VIP license cannot be secured. Alternatively, the DW_NOAUTH_CONTINUE environment variable can be set to allow simulations to continue when one or more VIP models fail to authorize. Unauthorized VC VIP LPDDR VIP models essentially become disabled when DW_NOAUTH_CONTINUE is set to any value.

```
% setenv DW_NOAUTH_CONTINUE
```

Also, some simulation environments allow *license polling*, which pauses the simulation until a license is available. License polling is described next.

If you encounter problems with licensing, see "Customer Support" on page 7.

2.17.2 License Polling

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

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

2.17.3 Simulation License Suspension

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

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

2.18 Environment Variable and Path Settings

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

- `DESIGNWARE_HOME` – The absolute path to where the Synopsys VC VIP is installed.
- `SNPSLMD_LICENSE_FILE` – The absolute path to a file that contains the license keys for Synopsys Common Licensing software or the *port@host* reference to this file.
- `LM_LICENSE_FILE` – The absolute path to a file that contains the license keys for your third-party tools. Also, include the absolute path to the third party executable in your PATH variable.
- `DW_LICENSE_FILE` – The absolute path to a file that contains the license keys for VIP product software or the port@host reference to this file.
- Your simulation environment and PATH variables must be set as required to support your simulator.

2.19 Determining Your Model Version

The version of the LPDDR at time of publication is O-2018.09. The following steps tell you how to check the version of the models you are using.

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

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

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

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

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

2.20 Integrating a VC VIP LPDDR VIP into Your Testbench

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

- "Creating a Testbench Design Directory"
- "The dw_vip_setup Utility"

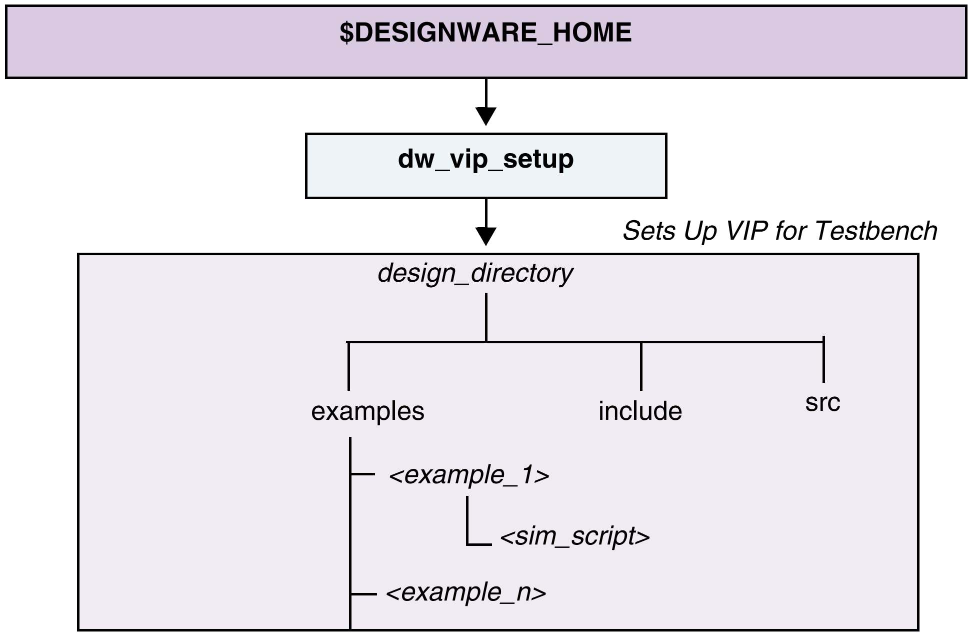
2.20.1 Creating a Testbench Design Directory

A *design directory* contains a version of the VC VIP LPDDR VIP that is set up and ready for use in a testbench. You use the dw_vip_setup utility to create design directories. For the full description of dw_vip_setup, refer to "The dw_vip_setup Utility" on page 26.

Note If you move a design directory, the references in your testbenches to the include files will need to be revised to point to the new location. Also, any simulation scripts in the examples directory will need to be recreated.

A design directory gives you control over the version of VC VIP LPDDR VIP in your testbench because it is isolated from the DESIGNWARE_HOME installation. When you want, you can use dw_vip_setup to update the VIP in your design directory. Figure 2-1 shows this process and the contents of a design directory.

Figure 2-1 Design Directory Created by dw_vip_setup



A design directory contains:

examples	Each VC VIP LPDDR VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.
include	Language-specific include files that contain critical information for VC VIP LPDDR VIP models. This directory is specified in simulator command lines.

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

.

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

2.20.2 Enhanced Access to PDF Documents from a Design Directory

The documentation for VC VIP is located in the document directory of the DESIGNWARE_HOME installation.

$DESIGNWARE_HOME/vip/svt/*vip_title*/J-2014.12/doc

For example:

$DESIGNWARE_HOME/vip/svt/lpddr_svt/J-2014.12/doc

These documents are not linked or copied when a design directory is created or when a module is added to a design directory. Direct access to the VIP documents from a design directory is not available.

Starting with this release, you can use the -doc option of the dw_vip_setup utility to create symbolic links to PDF documents in the DESIGNWARE_HOME installation from a design directory. This feature allows you to access the VIP documents from a design directory. The -doc option must be specified with the following options:

- `-add, -update or -example`
- `-path <design_directory>`

The syntax of the -doc option is:

```
-doc [-methodology <name>][-copy]
```

`-doc`

Creates symbolic links to PDF documents in the DESIGNWARE_HOME installation from the specified design directory. Only documents related to the models in the specified design directory are linked. The linked documents are located in the document directory of the specified design directory.

`-methodology <name>`

Creates symbolic links only to the documents associated with the specified methodology name. The valid methodology names are UVM, VMM, OVM, RVM and VLOG.

`-copy`

Copies the PDF documents instead of creating symbolic links.

For example:

```
DESIGNWARE_HOME = /tools/install
```

1. `dw_vip_setup -path /n/proj -a lpddr_agent_svt -svlog -doc`

   Design directory (/n/proj):

   ./doc/lpddr_svt/lpddr_svt_hdl_user_guide.pdf -> /tools/install/vip/svt/lpddr_svt/
   J-2014.12/doc/lpddr_svt_hdl_user_guide.pdf

```
 .....
./doc/lpddr_svt/lpddr_svt_release_notes.pdf -> /tools/install/vip/svt/lpddr_svt/
                                               J-2014.12/doc/lpddr_svt_release_notes.pdf
./doc/lpddr_svt/lpddr_svt_uvm_getting_started.pdf -> /tools/install/vip/svt/lpddr_svt/
                                               J-
2014.12/doc/lpddr_svt_uvm_getting_started.pdf
./doc/lpddr_svt/lpddr_svt_uvm_user_guide.pdf -> /tools/install/vip/svt/lpddr_svt/
                                               J-2014.12/doc/lpddr_svt_uvm_user_guide.pdf
./doc/lpddr_svt/lpddr_svt_vmm_user_guide.pdf -> /tools/install/vip/svt/lpddr_svt/
                                               J-2014.12/doc/lpddr_svt_vmm_user_guide.pdf
```

2. `dw_vip_setup -path /n/proj -u lpddr_agent_svt -svlog -doc -copy -methodology UVM`

   Design directory (/n/proj):

   ```
   ./doc/lpddr_svt/lpddr_svt_uvm_getting_started.pdf
   ./doc/lpddr_svt/lpddr_svt_uvm_user_guide.pdf
   ```

2.20.3 Selective Display of Content Within the VIP Installation

The -info home option of the dw_vip_setup utility displays product, version and example content within the VIP installation specified by DESIGNWARE_HOME to standard output. Starting with this release, you can specify optional attributes with the -info home option to display specific VIP content.

`-info home[:<product>[:<version>[:<methodology>]]]`

Displays product, version and example content within the VIP installation specified by `DESIGNWARE_HOME` to standard output. Optional arguments <product>, <version> and <methodology> delimited by colons (:) can be used to select specific content to be displayed. An error message is issued if a nonexistent or an invalid value is specified for an optional argument.

`<product>`

Specifies a VIP to be displayed. The valid product names are listed under the “LIBRARIES” section displayed by the “dw_vip_setup -info home” command.

`<version>`

Specifies a version of the VIP to be displayed. The valid versions are the versions installed in the VIP installation specified by DESIGNWARE_HOME.

`<methodology>`

Specifies the VIP content associated with the methodology to be displayed. The valid methodology names are UVM, VMM, OVM, RVM and VLOG.

2.21 The dw_vip_setup Utility

The `dw_vip_setup` utility:

- Adds, removes, or updates VC VIP LPDDR VIP models in a design directory.
- Adds example testbenches to a design directory, the VC VIP LPDDR VIP models they use (if necessary), and creates a script for simulating the testbench using any of the supported simulators.

- Restores (cleans) example testbench files to their original state.
- Reports information about your installation or design directory, including version information.

2.21.1 Setting Environment Variables

Before running dw_vip_setup, the following environment variables must be set:

- `DESIGNWARE_HOME` – Points to where the VC VIP LPDDR VIP is installed

2.21.2 The dw_vip_setup Command

You invoke dw_vip_setup from the command prompt. The dw_vip_setup program checks command line argument syntax and makes sure that the requested input files exist. The general form of the command is:

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

or

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

where

[-p[ath] directory]	The optional -path argument specifies the path to your design directory. When omitted, `dw_vip_setup` uses the current working directory.
switch	The *switch* argument defines `dw_vip_setup` operation. Table 2-1 lists the switches and their applicable sub-switches.

Table 2-1 Setup Program Switch Descriptions

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

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
model	The *model* argument defines the model or models that dw_vip_setup acts upon. This argument is not needed with the -info or -help switches. All switches that require the *model* argument may also use a model list. You may specify a version for each listed *model*, using the -version option. If omitted, dw_vip_setup uses the latest version. The -update switch ignores *model* version information.
-m[odel_list] *filename*	The -model_list argument causes dw_vip_setup to use a user-specified file to define the list of models that the program acts on. The model_list, like the *model* argument, can contain model version information. Each line in the file contains: *model_name* [**-v** *version*] –or– **# Comments**
-b/ridge	Update the specified design directory to reference the current DESIGNWARE_HOME installation. All product versions contained in the design directory must also exist in the current DESIGNWARE_HOME installation.
-doc	Creates symbolic links to PDF documents in the DESIGNWARE_HOME installation from a design directory
-info home `[:<product>[:<version>` `[:<methodology>]]]`	Displays product, version and example content within the VIP installation specified by DESIGNWARE_HOME to standard output. Optional arguments <product>, <version> and <methodology> delimited by colons (:) can be used to select specific content to be displayed. An error message is issued if a nonexistent or an invalid value is specified for an optional argument.
`-simulator <vendor>`	When used with the -`example` switch, only simulator flows associated with the specified vendor are supported with the generated run script and Makefile. **Note** Currently the vendors VCS, MTI, and NCV are supported.

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

3 General Concepts

3.1 Introduction to UVM

UVM is an object-oriented approach. It provides a blueprint for building test-benches using a constrained random verification. The resulting structure also supports directed testing. This chapter describes the data objects that support the higher structures that comprise the LPDDR VIP. Refer to the Class Reference HTML for a description of attributes and properties of the objects mentioned in this chapter. This chapter assumes that you are familiar with SystemVerilog and UVM.

3.2 UVM Support in Verification IP

In Synopsys VIP, UVM-compliant classes and attributes (members) are provided to represent protocol activity and the characteristics of that activity. The Agents also have components named sequencers. These are enabled if configured as an active model, and disabled if passive. In the case of Memory VIP, they are a reactive component, and so they generally just respond to detected requests. Reactive models typically will run a single sequence that runs in a forever block.

3.2.1 Base Classes

In an object-oriented programming environment, a set of base classes form the foundation for the entire system. Base classes provide common functionality and structure. The UVM base classes are specifically designed for the UVM approach to verification. They provide common functionality and structure needed for simulation (such as logging) and they support any sort of verification function.

Important UVM base classes used by the Synopsys VIP include:

- `uvm_analysis_port` - object-based interface; connects elements in a verification environment.
- `uvm_sequence_item` - base class for all data objects (such as transactions and configuration).
- `uvm_driver` - base class for driver model.
- `uvm_monitor` - base class for monitor model.
- `uvm_report_object` - standard logging object.
- `uvm_agent` - base class for the agent.
- `uvm_env` - base class for the verification environment that is built in the test-bench.

3.2.2 Environments

An environment is a verification component that consists mainly of two or more agents. It can be reused in a system-level test environment as a sub-environment.

3.2.3 Agents

An agent is a verification environment subset that is reusable in different verification environments. Agents are composed of two or more connected drivers or monitors that are linked to elements such as scoreboard or a sequencer.

3.2.3.1 Configuration Objects

Predefined configuration objects, extended from the uvm_sequence_item base class, are used for configuring the Synopsys LPDDR VIP to fit specific test-bench applications. The configuration objects specify agent attributes and support test-bench capabilities, such as randomization and constraints. Configuration objects apply to all appropriate transactors in the stack.

Configuration data objects convey the system level and port level test bench configuration. The configuration of agents is done in the build() phase of environment or the testcase. The configuration object properties can be of two types:

❖ Static configuration properties. Static configuration parameters specify configuration which cannot be changed when the system is running. Examples of static configuration parameters are data bus width, address width, density.

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

Configuration objects are used by agents, drivers and monitors. Configuration objects set by the configuration infrastructure must not be null and must be valid. If the object is not null, the build calls the is_valid method of the configuration object. If this method returns true, the transactor continues the construction; otherwise, a message is displayed and the simulation is halted. Configuration objects are controlled by direct access to their data properties or through randomization. Default randomization allows for a complete randomization of the configuration, including static as well as dynamic information.

The LPDDR VIP defines the following configuration classes.

❖ `svt_lpddr_configuration`. This configuration class is used for a LPDDR2 or LPDDR3 device. The base configuration class contains configuration information which is applicable to individual components. This is the top level configuration class; it has handles of mode register and the timing configuration classes. The configuration mainly specifies:

  ✦ VIP type (LPDDR2, LPDDR3, LPDDR4 or LPDDR5)

  ✦ Density, type of interface, type of delays, clock rate

  ✦ Address and data widths, burst size

  ✦ Timeout values

  ✦ Active/Passive mode of the agent

  ✦ Enable/disable coverage

  ✦ Enable/disable protocols checkers

  ✦ Bypass initialization

- `svt_lpddr5_configuration`: This configuration class is used for dual channel LPDDR5 device. This configuration class has two handles (cfg_channel_a, cfg_channel_b) of svt_lpddr_configuration configuration class.
- `svt_lpddr4_configuration`: This configuration class is used for dual channel LPDDR4 device. This configuration class has two handles (cfg_channel_a, cfg_channel_b) of svt_lpddr_configuration configuration class.
- `svt_lpddr_mode_register` . The mode register configuration class contains the default configuration information for all the fields of the various mode registers. Mode Register configuration class is further extended to the following:
  - LPDDR2 Mode Register Configuration class svt_lpddr2_mode_register
  - LPDDR3 Mode Register Configuration class svt_lpddr3_mode_register
  - LPDDR4 Mode Register Configuration class svt_lpddr4_mode_register
  - LPDDR5 Mode Register Configuration class svt_lpddr5_mode_register
- `svt_lpddr_timing_configuration`. The timing configuration class contains LPDDR timing parameters information which is applicable to individual LPDDR Memory components.

3.2.3.2 Transaction Objects

Transaction objects, which are extended from the uvm_sequence_item base class, define a unit of bus protocol information that is passed across the bus. The attributes of transaction objects are public and are accessed directly for setting and getting values. Most transaction attributes can be randomized. The transaction object can represent the desired activity to be simulated on the bus, or the actual bus activity that was monitored. The LPDDR VIP has several types of transaction objects, corresponding to different areas of the protocol.

Transaction data objects store data content and protocol execution information for LPDDR connection transactions in terms of bit-level and timing details of the transactions. These data objects extend from the uvm_sequence_item base class and implement all methods specified by UVM for that class.

LPDDR transaction data objects are used to:

- Generate random scenario stimulus, when used with UVM sequencer macros.
- Report observed transactions from receiver transactors.
- Generate random responses to transaction requests.
- Collect functional coverage statistics.
- Support error injection.

Class properties are public and accessed directly to set and read values. Transaction data objects support randomization for varying stimulus and to provide valid ranges and reasonable constraints:

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

The VIP supports extending transaction data classes for customizing randomization constraints. This allows you to disable some reasonable_* constraints and replace them with constraints appropriate to your system. Individual reasonable_* constraints map to independent fields, each of which can be disabled. The class provides the `reasonable_constraint_mode()` method to enable or disable blocks of reasonable_* constraints.

The LPDDR VIP defines the following transaction class.

- `svt_lpddr_transaction`:This class implements the transactions component for the LPDDR. Each transaction mentioned in the JEDEC specification are valid and implemented. The LPDDR model will have access to only those transaction properties and methodologies which are declared here in this class. All the fields and properties declared in this class will be random but, The randomization is kept off for transaction component.

3.2.4 Transactor Components

Transactors are objects like drivers and monitors in a UVM compliant verification environment. The test-bench and transactors exchange transactions through two different types of transactor interfaces:

- TLM Ports: The transactor objects communicate via the TLM Ports.
- Callbacks:
  - Callbacks are defined in a callback facade class (associated with each transactor), and accessed by registering (with the associated transactor) an instance of a class extended from that facade class.
  - Each transactor supports additional callbacks to access to data at internal dataflow points. Refer to the HTML documentation for a complete callback list.

3.2.4.1 Analysis Ports

The monitor inside the lpddr agent provides an analysis port called `"item_observed_port"`. At the end of the transaction, the monitor write the completed `svt_lpddr_transaction` object to the analysis port. This holds true in active as well as passive mode of operation of the lpddr agent. The user can use the analysis port for connecting to scoreboard, or any other purpose, where a transaction object for the completed transaction is required.

3.2.4.2 Exception and Exception List Objects

Exception objects, which extend from the uvm_sequence_item base class, represent injected errors or protocol variations. Each transaction object has an exception list, which is an object that serves as an array of exception objects that may apply. To enable exception generation, provide an exception list factory when constructing a transactor. If one is not provided, exception generation is disabled.

The LPDDR VIP defines the following exception classes:

- `svt_lpddr_transaction_exception`:Exception class. An alternate use model for injecting exceptions is to populate the exception list before sending it to the driver, or to populate the list after the transaction is received by the driver (through a callback).
- `svt_lpddr_transaction_exception_list`: Extends the SVT exception list base class providing strict typing for the extended exception class.

3.2.4.3 Callbacks

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

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

LPDDR VIP uses callbacks in three main applications:

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

The VIP defines the following types of callbacks:

- `traffic or dataflow event callbacks`: called in response to critical traffic or dataflow events, providing a mechanism for responding to the event or introducing errors into the event processing.

The following are the coverage callback classes defined by the VIP:

- `svt_lpddr_monitor_def_cov_data_callback`. Class containing the default coverage callbacks which respond to the transactor coverage callbacks, constructs data fields based on what is seen in the callbacks and then triggers coverage events indicating the data is available to be sampled
- `svt_lpddr_monitor_def_state_cov_callbacks`. State coverage is a signal level coverage.
- `svt_lpddr_monitor_def_state_cov_data_callbacks`. This callback class defines default data and event information that are used to implement the coverage groups. T
- `svt_lpddr_monitor_def_toggle_cov_callbacks`. Toggle coverage is a signal level coverage.
- `svt_lpddr_monitor_def_toggle_cov_data_callbacks`. This callback class defines default data and event information that are used to implement the coverage groups.
- `svt_lpddr_monitor_transaction_report_callback`.
- `svt_lpddr_monitor_transaction_xml_callback`.

Note the LPDDR UVM HTML documentation describes all the callbacks and when they occur. Following are other LPDDR class callbacks.

- `svt_lpddr_driver_callback`:Defines generic callback methods available in all memory drivers.
- `svt_lpddr_monitor_callback`: Monitor callback class contains the callback methods called by the monitor component.

- `svt_mem_driver_callback`: Driver callback class contains the callback methods called by the driver component.

3.2.5 Interfaces and Modports

SystemVerilog models signal connections using interfaces and modports. Interfaces define the set of signals which make up a port connection. Modports define logical connections supported by the port.

The LPDDR driver communicates with the LPDDR ports through modports. Modports provide a logical connection between transactors and the testbench. This connection is bound in after the interface is instantiated and other transactors are connected to its other modports.

In UVM there are two options for setting the interface:

- User can use the configuration database to set the interface which will be retrieved during the build phase.
- User can pass this in through the configuration object, and the interface is extracted during the build phase.

The following are the interfaces that the LPDDR VIP includes:

- `svt_lpddr_jedec_chip_if`: For LPDDR2 ,LPDDR3 and Single channel LPDDR4 interface.
- `svt_lpddr4_jedec_chip_if`: For dual channel LPDDR4 interface.
- `svt_lpddr5_jedec_chip_if`: For Dual channel LPDDR5 Interface.
- `svt_lpddr5_dual_chan_jedec_chip_if` : For Dual channel LPDDR5 interface.

See the LPDDR SVT - Interfaces Reference page in the HTML Class Reference for more information. “Interface Options” describes LPDDR interface options.

3.2.6 Constraints

Synopsys VIP uses objects with constraints for transactions, configurations, and exceptions. Tests in a UVM flow are primarily defined by constraints. The constraints define the range of randomized values that are used to create each object during the simulation.

Classes that provide random attributes allow you to constrain the contents of the resulting object. When you call the `randomize()` method, which is a built-in method, all random attributes are randomized using all constraints that are enabled. Constraint based randomization used in this way allows for the benefits of randomization along with fine control of attributes as needed.

Constraint randomization is sometimes misunderstood and seen as a process whereby the simulation engine takes the control of class members away from the user. In fact, the opposite is true. Randomization is an additional way for the user to assign class members, and there are several ways to control the process.

Note Disabling these constraints may slow the simulation and introduce system memory issues.

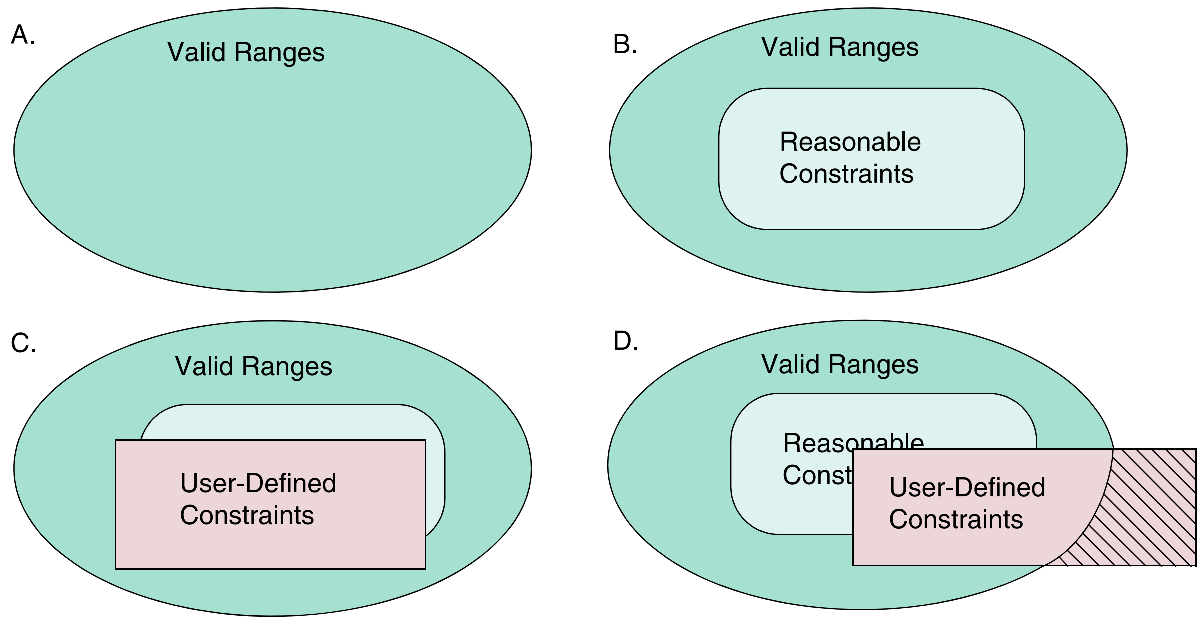
The following techniques apply when working with randomization:

- Randomization only occurs when an object's `randomize()` method is called, and it is completely up to the test code when, or even if, this occurs.

- Constraints form a rule set to follow when randomization is performed. By controlling the constraints, the testbench has influence over the outcome. Direct control can be exerted by constraining a member to a single value. Constraints can also be enabled or disabled.
- Each rand member has a rand mode that can be turned ON or OFF, giving individual control of what is randomized.
- A user can assign a member to a value at any time. Randomization does not affect the other methods of assigning class members.
- Valid range constraints:
  - Provided with Synopsys VIP.
  - Keep values within a range that the transactors can handle.
  - Are not tied to protocol limits.
  - On by default, and should not be turned off or modified .
- Reasonable constraints:
  - Provided with Synopsys VIP.
  - Keep values within protocol limits (typically) to generate worthwhile traffic.
  - In some cases, keep simulations to a reasonable length and size.
  - Allow all legal values defined by the protocol.
  - Bias towards most commonly used values for efficient simulation.
  - May result in conditions that are a subset of the protocol.
  - On by default and can be turned off or modified (user should review these constraints).
- User-defined constraints:
  - Provide a way for you to define specific tests.
  - Constraints that lie outside of the valid ranges are not included during randomization.
- All constraints that are enabled are included in the simulation. The constraint solver resolves any conflicts:

The following diagram shows the scope of the constraints that are part of all VIP.

Figure 3-1 Constraints: Valid Ranges, Reasonable, and User-Defined



3.2.7 Messages

Messages can be controlled individually or in groups. This section describes messages and how to use them. Messages generated by VIP transactors are compatible with the `uvm_report_object` base class. The messages originate in two scopes:

- ❖ Methodology messages, which report base class conditions and errors.
- ❖ Protocol-specific messages that report protocol conditions, events, and errors.

Messages can have a number of attributes, such as type, severity, ID, and text. Here are some qualities of these attributes:

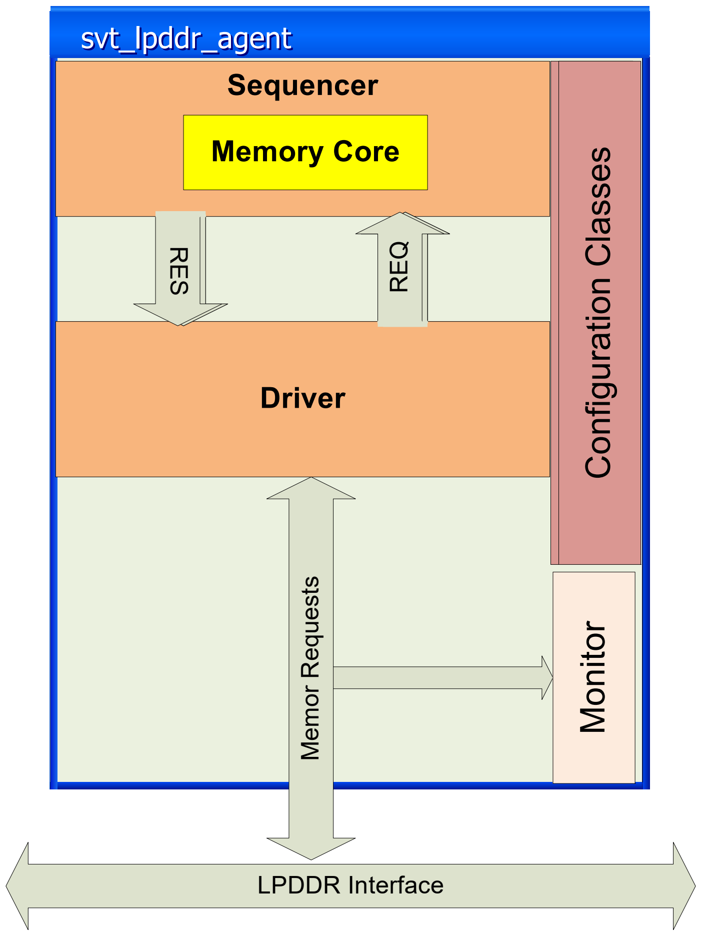
- ❖ Type**:** Messages are categorized into types. The possible types are listed in the UVM Manual.
- ❖ Severity**:** Severity is similar to the urgency of the message or how serious it is. The possible values for severity are listed in the UVM manual.

4 LPDDR Agent and LPDDR4/LPDDR5 Env

4.1 LPDDR Agent

Agents are UVM components that are composed of basic components such as drivers, monitors and sequencers. The LPDDR VIP defines an agent that can be customized to support a variety of verification scenarios. For modeling a LPDDR2, LPDDR3, Single channel LPDDR4,and Single channel LPDDR5 device, the name of the LPDDR memory agent is called `svt_lpddr_agent`. The following figure shows the architecture of the VIP agent.

Figure 4-1 LPDDR Memory Agent Architecture



4.2 UVM Driver

The `svt_lpddr_agent` agent contains a UVM driver called `svt_lpddr_driver`. Because the `svt_lpddr_agent` is a reactive agent, a request for a sequence item must be generated from the driver to the sequencer. This occurs before the sequencer generates a response and sends it back to the driver.

4.3 Sequencer and Memory Core

The sequencer of the memory agent is reactive. It waits for a request from the driver and will generate an appropriate response. Unless you specify a different default responses, the model responds as follows:

- Read - reads data from the memory core.
- MRR Read - content of addressed mode register.
- Write Leveling - regular response based on observed DQS.
- Write with DM enabled/disabled - no response.

The sequencer has a *memory core.* The memory core updates the memory model after a write command. The sequencer will also access memory through the memory core in the case of a read command.

The memory core interacts with three interface classes to update the memory contents. The classes are:

- `svt_lpddr_transaction`:Transaction class shared between LPDDR memory agent sequencer and monitor.
- `svt_mem_backdoor`: This class provides a backdoor and iterator interface to a memory core. Multiple instances of this interface may exist on the same memory core.
- `svt_mem_transaction`: This memory access transaction class is used as the request and response type between a memory driver and a memory sequencer.

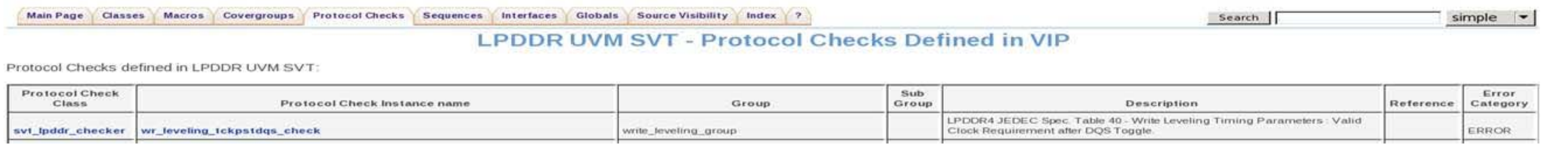
4.4 UVM Monitor

The LPDDR agent contains a UVM monitor `svt_lpddr_monitor`. It does the following:

- Provides protocol checks which can be individually turned on/off. You can find the complete list of protocol checks in the LPDDR HTML Class Reference at:

  `$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/doc/lpddr_svt_uvm_class_reference/html/index.html`

  The HTML help system has a tab at the top for all checks as shown in the following illustration:



LPDDR UVM SVT - Protocol Checks Defined in VIP

Protocol Checks defined in LPDDR UVM SVT:

Protocol Check Class	Protocol Check Instance name	Group	Sub Group	Description	Reference	Error Category
svt_lpddr_checker	wr_leveling_tckpstdqs_check	write_leveling_group		LPDDR4 JEDEC Spec. Table 40 - Write Leveling Timing Parameters : Valid Clock Requirement after DQS Toggle.		ERROR

- Coverage information
- Callbacks
- Transaction logging

4.5 Passive and Active Agent Behavior

The following table lists the behavior of the agent in active and passive mode.

Table 4-1 Agent Behaviors in Passive and Active Mode

Agent behavior in active mode	Agent behavior in passive mode
Model components continue to perform the active functionality of decoding the commands and perform memory operations (write/read).	In Passive mode, model components monitor the input and output signals, and perform passive functionality of coverage and protocol checking. Users can enable/disable this functionality through configuration.

4.6 Agent Configuration

You configure the `svt_lpddr_agent` agent by setting members in the following configuration classes:

- `svt_lpddr_configuration` class. The `svt_lpddr_configuration` class allows you to set the following device properties:
  - VIP type:You can set the memory model to work at LPDDR2, LPDDR3, LPDDR4 or LPDDR5. This setting must match your license.
  - Memory capacity
  - Data width
  - Bypass initialization
  - Enable checks, coverage and trace messaging etc.
- `svt_lpddr_mode_register`:
  - Configures mode registers for LPDDR2, LPDDR3, LPDDR4 and LPDDR5.
  - Most LPDDR configuration is defined through Mode Registers.
- `svt_lpddr_timing_configuration`:
  - Allows you set values for all timing intervals defined by the LPDDR protocol.

Note, the previous list defines the base agent configuration classes. In normal testbench construction you would configure the LPDDR agent by using catalog and part number *.cfg files. The catalog*.cfg files contain a complete listing of all the parameters of a given vendor part. You can set the agent's configuration to the values specified in the *.cfg catalog file by assigning those values to `svt_lpddr_configuration` in one step.

4.7 Virtual Interfaces for LPDDR2, LPDDR3, Single channel LPDDR4 and Single channel LPDDR5 Interfaces

The LPDDR agent is the component which is connected to the LPDDR signals. The Agent has virtual interface of:

- `svt_lpddr_jedec_chip_if`
- `svt_lpddr5_jedec_chip_if` (for LPDDR5)

4.8 LPDDR4/LPDDR5 Environment

The LPDDR VIP provides the `svt_lpddr4_ env /svt_lpddr5_env` class to model the dual channel LPDDR4/LPDDR5 device. It has two handles of type `svt_lppdr_agent` to model channel A and channel B of a LPDDR4/LPDDR5device.

4.9 LPDDR4/LPDDR5 Environment Configuration

You configure the `lpddr4_env / svt_lpddr5_env` agent by setting members in the following configuration class:

- `svt_lpddr4_configuration /svt_lpddr5_configuration class`: The `svt_lpddr4_configuration /svt_lpddr5_configuartion` has two handles (cfg_channel_a, cfg_channel_b) of `svt_lpddr_configuration` class to model the channel A and channel B configuration

4.10 Virtual Interface for Dual Channel LPDDR4/ Dual Channel LPDDR5

The LPDDR Env is the component which is connected to Dual Channel LPDDR4 / Dual Channel LPDDR5 signals. The Env has a virtual interface of:

- `svt_lpddr4_jedec_chip_if / svt_lpddr5_jedec_chip_if .`

5 Memserver

5.1 Overview

A critical component of the LPDDR Memory Model VIP is the Memory Core or Memserver. Memserver is the part of the model that directly manages memory. It has been implemented using SystemVerilog and C/C++ classes and functions with the most efficient storage requirements.

You access the Memserver capabilities through two classes:

- `svt_mem_core`. Class which implements a SystemVerilog wrapper around the C-based memserver.
- `svt_mem_backdoor`. Provides a backdoor and iterator interface to Memserver. Multiple instances of this interface may exist on the same memory core.

The Memserver provides two ways of access to stored memory:

- Standard - a device requests across the LPDDR bus read and write transactions.
- Backdoor - used by a test bench to inject data, examine data, and monitor the status of Memserver. Backdoor accesses are lower priority than Standard transactions. They must not over-ride access-lock protection. Backdoor access attempts to a locked range of memory will abort returning a `ACCESS_LOCK` error status. This eliminates multiple-master problems when multiple models share one Memserver.

5.2 Memserver Features

- Maximum address: 64 bits
- Maximum data width (per memory word): 32767 bits
- Type of storage: 2-state or 4-state, 4 state encoding:

Table 5-1 Storage Combinations

A	B	Value
0	0	0
1	0	1
0	1	U
1	1	X

- Standard load/dump formats: Memory initialization format (MIF) and Verilog readmemh (MEMH).
- Messages are produced using the `` `svt_fatal(), `svt_error(), `svt_warning(), `svt_info(), `svt_debug() ``, and `` `svt_verbose() `` methods.
- Virtual patterns provide background data without allocating simulation host memory.
- Memory instances are write protectable.

5.2.1 2-State, 4-State Data, and Location Defaults

The Memserver provides either 4-state data storage and 2-state storage. The 4-state storage allows storing of 'U' bit values in 4-state modes so that Memserver can identify which memory location bits are un-initialized. Note, that in 2-state mode, if a partial (masked) write is made to a previously un-initialized location, then the entire location's contents will be marked initialized (even though some bits may not have actually been written). Consequently RD_B4_WR and PARTIAL_READ status can be returned for a partially initialized location in 4-state data, but not in 2-state data.

All newly allocated locations are initialized to a default value. The values are:

- 2-state- bits are set to '0'.
- 4-state- bits are set to 'U' - Note, 'U' bits are sparse array-encoded as Verilog 'z' bits.

5.2.2 Attribute Bits

In addition to data bits, Memserver provides attribute bits for each allocated memory location. The two least significant attribute bits track the access status of each location as follows:

- `UNINIT` - un-initialized.
- `LAST_WR` - the last access of this location was a write.
- `LAST_RD` - the last access of this location was a read.
- `LAST_VRD` - returned when a read to an unallocated or un-initialized location is satisfied with a virtual pattern.

Other system uses of attribute bits include:

- Write Protected - allows write protection of individual memory locations.
- Access-Lock Markers - marks locations that have been accessed by a legal read or write operation within an active access-lock address range. The start_access() function clears any existing `MARK` attribute bits in the locked memory range.
- The absolute maximum number of attributes bits is 31.

The read() and write() commands effect the access status (and sometimes the attribute bits) associated with a memory location. If the poke() function is called with the address of un-initialized memory, then that memory location will be created if necessary, written with the supplied data, and the access status set to `INIT`.

5.2.3 Sequential Operation Detection

By making use of the access state bits, Memserver can detect the following access sequences:

- ❖ `RD_RD_NO_WR` - two reads of the same location without an intervening write.
- ❖ `WR_SAME` - a write with the same data existing at a location occurred. This check will trigger regardless of the source of the existing data - writes, loads from a file, and an algorithmic pattern are legitimate sources of original data. This check will not trigger for a write to an un-initialized location (access status is UNINT).
- ❖ `WR_LOSS` - new (different) data was written to the same location without an intervening read. This implies a WR_WR. The reporting of the WR_WR is suppressed when a WR_LOSS is detected.
- ❖ `WR_WR` - two writes occurred to the same location without an intervening read. Two sequential writes are required to trigger this check. A write following a data initialization (such as a load() command) will not trigger this check.
- ❖ `RD_B4_WR` - a read of an un-initialized location (or a partially un-initialized location) occurred.

Attributes can be checked or set directly for any location using `get_attributes(), set_attributes(), clear_attributes(), and clear_attributes()`. Access status values are not modified by these functions.

Write protect checking is enabled or disabled using the set_checks() function. It's default state is disabled.

The write protect attribute can be set for locations initialized by a load() (load memory contents from a file. You control this by the write_protected bit in the `svt_mem_core::load() method: function void load(string filename, bit write_protected = 0).`

5.3 Default Pattern Generation

You use the function initialize() to generate a pattern in any part of memory. It uses virtual fill patterns. That is, when an access is made to an un-initialized memory location, then Memserver looks to the virtual pattern database to find out what the value should be and calculates it on the fly.

If such a pattern is found, a data value is computed for that address and is returned. If no address matching pattern is found, the default data value (all 0's for 2-state simulation and all 'u's for 4-state simulation) is returned.

5.4 Burst Operations and Address Range Locking

While the Memserver does not directly implement block (multi-location) writes or reads, it does provide support for burst operations by providing a marking/locking mechanism to prevent unauthorized access of a memory location by a test bench or remote accesses. A lock may be for 1 to 'n' addresses. Multiple simultaneous access-locks are supported.

When an access-lock is active, note the following:

- ❖ Read and read accesses are permitted outside of the lock's address range. But a peek or poke within the address range will fail and return `ACCESS_LOCKED` status. Memory contents are not modified by such a write.
- ❖ Read and read accesses are permitted only within the address range. If multiple locks are active, then the read() or write() may occur in any of them. Memserver does not determine which range is appropriate for which access-lock.

- If `start_access()` was called with the `MARK` option active and `NO_ACCESS_ATTR` was not used when creating the memserver instance, then each location touched by read and read (depending on the type of lock, read or write) will mark that location with the `ACCESS` system attribute bit.
- `access_end()` terminates an access lock. It identifies which lock to terminate using with the base address of the active lock. It returns the number of times a location with the locked region was touched.

5.5 Burst Operation Collisions

When a collision between a backdoor or remote access and a locked burst address range occurs, the following table list what happens:

Table 5-2 Collision between a Backdoor or Remote Access

	Poke	Peek
Addr marked for Burst Read	ERROR, poke doesn't complete	ERROR, poke completes
Addr marked for Burst Write	ERROR, poke doesn't complete	ERROR, poke doesn't complete

5.6 Status Bit Defines

A number of the member functions pass back special conditions in a status parameter. The bit definitions of this value are defined in svt_mem_sa_defs.svi. The following is a list of them:

- `SVT_MEM_SA_STATUS_OK` - this value is Zero. No other bits are present.
- `SVT_MEM_SA_STATUS_RD_RD_NO_WR` - two reads to the same location with no intervening write occurred.
- `SVT_MEM_SA_STATUS_WR_LOSS` - two writes with no intervening read occurred at a location and the second write altered the data of that location.
- `SVT_MEM_SA_STATUS_WR_SAME` - a location was re-written with the same data it already held
- `SVT_MEM_SA_STATUS_WR_WR` - two writes with no intervening read occurred at a location.
- `SVT_MEM_SA_STATUS_RD_B4_WR` - a location was read before it was initialized or written. 32'h00000010
- `SVT_MEM_SA_STATUS_WR_PROT` - a write was attempted to a write protected instance or to a write protected location.
- `SVT_MEM_SA_STATUS_ADR_ERR` - an address values with active bits beyond the specified address width of an instance was detected or an address range was used where lo > hi.
- `SVT_MEM_SA_STATUS_DATA_ERR` - a data value exceeded the specified data width in bits.
- `SVT_MEM_SA_STATUS_ACCESS_LOCKED` - a backdoor access (peek or poke) was attempted to a location within an active access-locked range.
- `SVT_MEM_SA_STATUS_ACCESS_ERROR` - an error was detected involving access lock ranges.
- `SVT_MEM_SA_STATUS_FORMAT_ERR` - a problem occurred with a load/dump/compare formatter.
- `SVT_MEM_SA_STATUS_PARTIAL_RD` - a read was made from a location where only some bits had been initialized. Only applies to 4-state instances.

- `SVT_MEM_SA_STATUS_OPEN_FAILED` - an attempt to open a disc file failed.
- `SVT_MEM_SA_STATUS_IO_ERROR` - an error occurred while performing disc I/O.
- `SVT_MEM_SA_STATUS_MISCOMPARE` - an incorrect data value was encountered.

5.7 Peak and Poke Directly into Memory

You may directly write or read memory ranges using svt_mem_backdoor::peek and svt_mem_backdoor::poke functions.

5.8 Loading and Dumping Memory Contents Using Disk Files

Using Memserver you can load and dump the contents of memory to and from disk files. The following sections describe those features.

5.8.1 Loading Memory Data

The load() method loads memory locations with the contents of a specified disk file. Disk file format is determined automatically based on the file suffix and/or file contents. If the file format cannot recognized, then Memserver sets the status to `SVT_MEM_SA_STATUS_FORMAT_ERR`.

The load() function sets the access status of each loaded location to `SVT_MEM_SA_ACCESS_STATUS_INIT` (initialized).

Following is the syntax for the method:

```
int load( filename, bit write_protected = 0);
```

Where:

- filename: is a string naming the file to be loaded.
- Locations can be flagged as write protected by setting the write_protected argument. When set to protect, memserver allocates an attribute for system use and marks each loaded memory location with it. When write protect checking is enabled, a write to a protected location will cause a `STATUS_WR_PROT` error.

Note the following:

- Use the `set_checks()` function to enable or disable write protect checking (default is disabled).
- Both MEMH and MIF are supported.
- Formatters may be added on the simulator command line by the user in the form of C++ source extending the `svt_mem_sa_format` class and automatically incorporated in the simulation.
- Returns 0 if successful, a positive warning value, or a negative error value.

5.8.2 Dumping the Contents of Memory

The dump() member writes the contents of memory within an address range to a disk file using a specified file format. Following is the syntax:

```
int dump( string filename , string filetype , bit append , svt_mem_alpddr_t alpddr_lo ,
svt_mem_alpddr_t alpddr_hi);
```

Where:

- filename - Name of the file to write to. The file extension determines which format the file is created in.

- filetype - The string name of the format to be used when writing a memory dump file. The "MEMH" (Verilog readmemh format) is built into Memserver.
- append - Start a new file, or add onto an existing file.
- `addr_lo` - Starting address.
- `addr_hi` - Ending address.

5.8.3 Comparing the Contents of Memory

Memserver provides a number of members to compare the contents of memory.

The compare functions have no effect the access status of each checked location. Compares the content of the memory in the specified address range (entire memory by default) with the data found in the specified file, using the relevant policy based on the filename.

The following comparison mode are available:

- Subset: The content of the file is present in the memory core. The memory core may contain additional values that are ignored.
- Strict: The content of the file is strictly equal to the content of the memory core.
- Superset: The content of the memory core is present in the file. The file may contain additional values that are ignored.
- Intersect: The same addresses present in the memory core and in the file contain the same data. Addresses present only in the file or the memory core are ignored.

```
int compare( string filename , compare_type_enum compare_type , int max_errors ,
svt_mem_alpddr_t alpddr_lo , svt_mem_alpddr_t alpddr_hi );
```

Where:

- filename - Name of the file to compare to. The file extension determines which format the file is created in.
- `compare_type` - Determines which kind of compare is executed.
- `max_errors` - Data comparison terminates after reaching `max_errors`. If `max_errors` is zero, assume a maximum error count of 10.
- `addr_lo` - Starting address.
- `addr_hi` - Ending address.
- Returns the number of miscompares.

Disk file format is determined automatically based on file suffix and/or file contents (in the same manner as the load() command). If the file format is not recognized, then the status is set to `SVT_MEM_SA_STATUS_FORMAT_ERR`.

5.8.4 Resetting Memory

Use the reset() method to do the following:

- ❖ All host data allocated to memory locations are deallocated.
- ❖ All attribute bits are deallocated.
- ❖ Memory pattern generators are removed.

Note

Use the `free()` command to flush only part of memory. The `free()` method takes parameters defining the start and end of the memory segment you want to flush.

6 Monitor and Coverage

6.1 Introduction

This section provides information on the LPDDR monitor which is instantiated in the `svt_lpddr_agent`. The monitor class `svt_lpddr_monitor` in turn instantiates the `svt_lpddr_checker` class that provides the built-in protocol checking capability.

6.2 Analysis Port

The monitor in the `svt_lpddr_agent` provides an analysis port `item_observed_port`. At the end of the transaction, the monitor writes the completed transaction objects to their analysis port. This holds true in active as well as passive mode of operation. You can use the analysis port for connecting to a scoreboard, or any other purpose where a transaction object for the completed transaction is required.

6.3 Protocol Checkers

The monitor instantiates the protocol checker. To enable checking you must explicitly turn on checking using the `svt_lpddr_configuration` member `enable_checks`.

6.4 Coverage

The LPDDR functional coverage approach is bottom-up. That is, the analysis starts at the signal level and goes up to the transaction level. Along this path several coverage types are used. The signal level uses toggle, state, and meta coverage, while the transaction level uses cross and meta coverages. These types of coverage are described in the following sections.

6.4.1 Toggle Coverage

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

6.4.2 State Coverage

State coverage is a signal level coverage. State coverage applies to signals that are a minimum of two bits wide. In most cases, the states (also commonly referred to as coverage bins) can be easily identified as all possible combinations of the signal. For example, different values of ca, dm, dqs, and dq are covered under the state coverage. If the state space is too large, an intelligent classification of the states must be made. In the case of the ca signal for example, coverage bins would be one bin to cover the lower ca range, one bin to cover the upper ca range and one bin to cover all other intermediary ca range.

6.4.3 Transaction Coverage

Transaction coverage covers LPDDR transactions types and Cross coverage across bank address. The cross coverage involves cross between type of transaction and bank address.

6.4.4 Mode Register Coverage

Mode Register Coverage covers each field of LPDDR Mode Register.

6.4.5 Meta Coverage

Meta coverage covers important transaction done by the VIP. They are sampled when specific event is generated corresponding to specific functionality.

LPDDR2 supported properties for meta coverage:

- `seamless_read`
- `seamless_write`
- `BST_effective_BL`
- `write_truncation_flag`
- `read_truncation_flag`
- `write_truncated_valid_bursts`
- `read_truncated_valid_bursts`
- `frequency_changed`
- `clock_stopped`
- `boot_period`

LPDDR3 and LPDDR4 supported properties for meta coverage:

- `seamless_read`
- `seamless_write`
- `frequency_changed`
- `clock_stopped`
- `boot_period`
- `ca_training_mode`

6.5 Enabling Default Coverage

The default functional coverage can be enabled by setting the following attributes in the port configuration class `svt_lpddr_configuration` to one. To disable coverage, set the attributes to zero. The attributes are:

- `enable_cov[0] for transaction coverage (default: 0)`
- Checker coverage:
  - `enable_cov[1] controls Pass coverage(default: 0)`
  - `enable_cov[2] controls Fail coverage(default: 0)`
  - `enable_cov[3] controls State coverage(default: 0)`
  - `enable_cov[4] controls Toggle coverage (default: 0)`

- `enable_cov[5] controls AC Timing coverage (default: 0)`

6.6 Coverage Shaping and Control

The handle to the configuration class `svt_lpddr_configuration` is provided to the class `svt_lpddr_monitor_def_cov_callback`, which implements the default cover groups. Based on the configuration, the coverage bins are shaped.

Note
The unwanted bins are ignored.

6.7 Coverage Callbacks

The coverage callback class implementing default cover groups is called `svt_lpddr_monitor_def_cov_callback`.: T his class is extended from the coverage data callback class. The naming convention uses “`def_cov`” in the class names for easy identification of these classes. It includes default cover groups based on the data and events defined in the data class.

6.7.1 LPDDR Coverage Callback Classes

The following table lists LPDDR coverage callback classes.

Table 6-1 LPDDR Coverage Callback Classes

LPDDR Coverage Callback Class	Description
`svt_lpddr_monitor_def_toggle_cov_callbacks`	Toggle coverage is a signal level coverage. Toggle coverage provides baseline information that a system is connected properly, and that higher level coverage or compliance failures are not simply the result of connectivity issues. Toggle coverage answers the question: Did a bit change from a value of 0 to 1 and back from 1 to 0?
`svt_lpddr_monitor_def_toggle_cov_data_callbacks`	This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses "`def_*_cov_data`" in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage requests by setting the coverage data and triggering the coverage events.
`svt_lpddr_monitor_def_cov_data_callback`	Class containing the default coverage callbacks which respond to the transactor coverage callbacks, constructs data fields based on what is seen in the callbacks and then triggers coverage events indicating the data is available to be sampled.
`svt_lpddr_monitor_def_state_cov_callbacks`	State coverage is a signal level coverage. It applies to signals that are minimum of two bits wide. In most cases, the states (also commonly referred to as coverage bins) can be easily identified as all possible combinations of the signal. This Coverage Callback consists of having covergroup definition and declaration.
`svt_lpddr_monitor_def_state_cov_data_callbacks`	This callback class defines default data and event information that are used to implement the coverage groups. The naming convention uses "`def_*_cov_data`" in the class names for easy identification of these classes. This class also includes implementations of the coverage methods that respond to the coverage requests by setting the coverage data and triggering the coverage events.

7 Catalog and Part Numbers

7.1 Overview

The Synopsys LPDDR VIP provides you with catalogs of vendor memory parts which you can use to configure your VIP and manage your testbench. The following sections describe catalog and part numbers and how to use them.

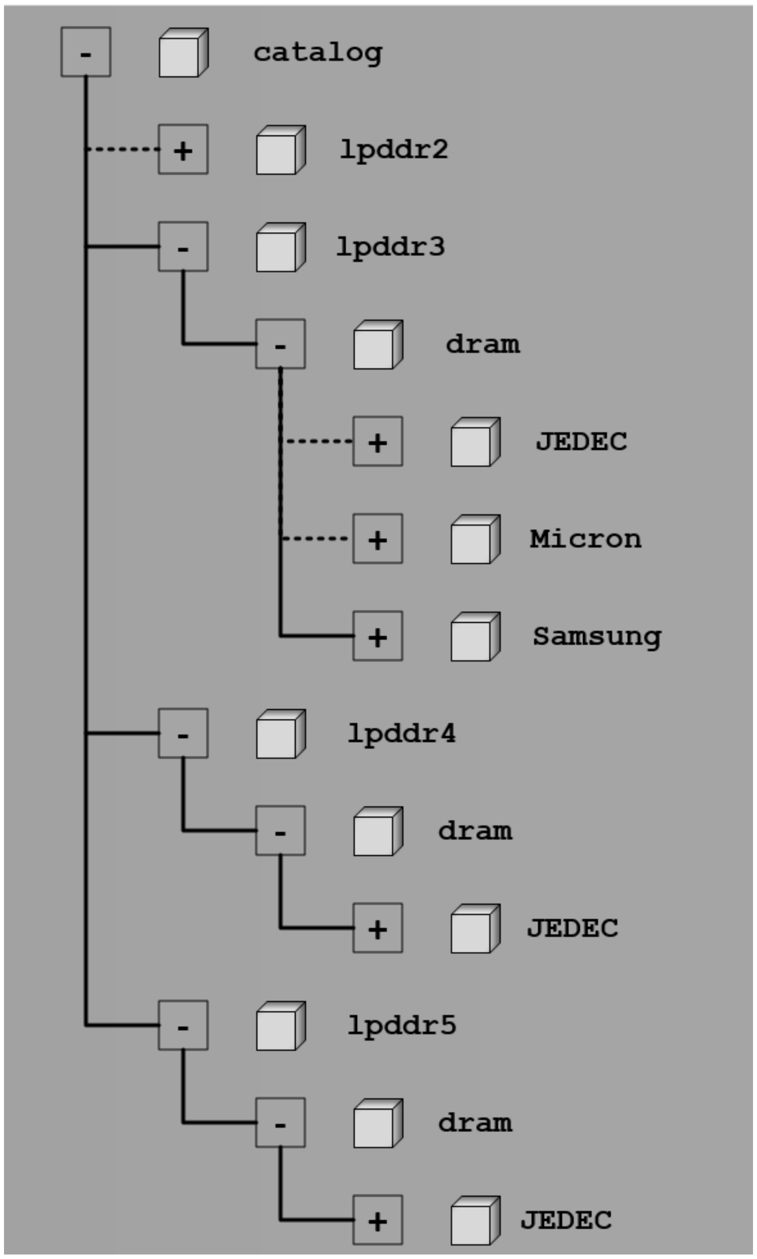
7.2 Location of Vendor Catalogs After Installation

After installation, vendor catalogs are located in the following directory:

```
$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/catalog
```

The catalog directory has a typical structure shown in the following illustration.

Figure 7-1 Catalog Directory



7.3 Contents of a Part Number *.cfg File

The part *.cfg files are ASCII files which sets values for the behavior of any given model. The following shows a LPDDR2 parts catalog contents:

Attention In this release, *.cfg files are not encrypted. In the future, there will be a mix of unencrypted and encrypted files.

```
//
```

```
//Description:64Mb, 1.8V
//Density    :4Mbx16
//Speed      :533MHz
//
catalog_part_number="jedec_lpddr2_S2_64M_x16_1066_1_875"
catalog_vendor="JEDEC"
catalog_package="dram"
catalog_class="lpddr2"
vip_type=LPDDR2
vif_type=JEDEC_CHIP
vip_mode_type=TYPICAL
lpddr2_type=LPDDR2_S2
svt_lpddr_mem_depth=SVT_MEM_64Mb
num_data_bursts=4
data_width=16
bank_addr_width=2
row_addr_width=12
column_addr_width=8
chip_select_addr_width=1
addr_width=23
data_width_type=DATA_WIDTH_16_BITS
minimum_refresh_count=2048
clock_rate@timing_cfg=533
tREFIab_us@timing_cfg=15.6
tREFIpb_us@timing_cfg=0
tRFCab_ns@timing_cfg=90
tRFCpb_ns@timing_cfg=0
tXSR_ns@timing_cfg=100
tREFW_ms@timing_cfg=32
tREFBW_us@timing_cfg=2.88
tCCD_ck@timing_cfg=1
tINIT0_ms@timing_cfg=20
tINIT1_ns@timing_cfg=100
tINIT2_ck@timing_cfg=5
tINIT3_us@timing_cfg=200
tINIT4_us@timing_cfg=1
tINIT5_us@timing_cfg=10
tCK_avg_ns@timing_cfg=1.875
tCH_avg_min_ck@timing_cfg=0.45
tCH_avg_max_ck@timing_cfg=0.55
tCL_avg_min_ck@timing_cfg=0.45
....
tERR_3per_ps@timing_cfg=-157
tERR_4per_ps@timing_cfg=-175
tERR_5per_ps@timing_cfg=-188
tERR_6per_ps@timing_cfg=-200
tERR_7per_ps@timing_cfg=-209
tERR_8per_ps@timing_cfg=-217
tERR_9per_ps@timing_cfg=-224
tERR_10per_ps@timing_cfg=-231
tERR_11per_ps@timing_cfg=-237
tERR_12per_ps@timing_cfg=-242
tZQINIT_us@timing_cfg=1
tZQCL_ns@timing_cfg=360
```

```
tZQCS_ns@timing_cfg=90
tZQRESET_ns@timing_cfg=50
tDQSCK_min_ps@timing_cfg=2500
tDQSCK_max_ps@timing_cfg=5500
tDQSCKDS_ps@timing_cfg=330
tDQSCKDM_ps@timing_cfg=680
...
tQSL_ck@timing_cfg=0.38
tRPRE_ck@timing_cfg=0.9
tRPST_ck@timing_cfg=0.38
tLZ_DQS_ps@timing_cfg=2200
tLZ_DQ_ps@timing_cfg=2178
tHZ_DQS_ps@timing_cfg=5400
tHZ_DQ_ps@timing_cfg=5780
tDH_ps@timing_cfg=210
tDS_ps@timing_cfg=210
tDIPW_ck@timing_cfg=0.35
tDQSS_min_ck@timing_cfg=0.75
tDQSS_max_ck@timing_cfg=1.25
tDQSH_ck@timing_cfg=0.4
tDQSL_ck@timing_cfg=0.4
tDSS_ck@timing_cfg=0.2
tDSH_ck@timing_cfg=0.2
tWPST_ck@timing_cfg=0.4
tWPRE_ck@timing_cfg=0.35
tCKE_ck@timing_cfg=3
tISCKE_ck@timing_cfg=0.25
....
tFAW_ns@timing_cfg=50
tDPD_us@timing_cfg=500
```

7.4 Hierarchical Structure of Vendor Memory Parts

The parts catalog is structured in the following hierarchical way as shown in Table 7-1.

Table 7-1 Structure of Vendor Catalogs

Name	Description
Catalog	One particular collection of part numbers for a specific memory class.
Class	A sub-designation that allows a suite to release multiple catalogs. For example, LPDDR will ship with LDDR2 and LDDR3 catalogs, and these are differentiated by 'Class'.
Part	A specific memory part. This is matched up with a.cfg file that supplies the VIP configuration needed to configure the VIP for that particular part.

7.5 Catalog Classes and Features

The LPDDR Memory classes used to manage catalogs and part numbers provides are as follows:

❖ `svt_mem_vendor_part`:Default part catalog entry. If additional or different part selection criteria are required for a specific suite, they should be added in a derived class. It must be specialized with a policy class that contains a static `#get()` method returning the full path to the installation directory of the suite.

❖ `svt_mem_part_mgr`: This class is used to choose vendor parts within all available catalogs.

7.5.1 Configuring An Agent Using Catalog Configuration Files

To configure your Agent to behavior as a Vendor part, follow these general steps.

1. Select a class (LPDDR2, LPDDR3, LPDDR4, LPDDR5). This selection determines which parts are available for this catalog.
2. Create a Policy Class. A policy class defines the search criteria you want to follow or limit in searching for a suitable part.
3. Use the `svt_mem_part_mgr` class to select a specific part using any or all of the following characteristics:
   - ✦ Vendor name (JEDEC, Micron, etc) user `get_vendor_name()`
   - ✦ Part name `(MT42L128M16D1_18, jedec_lpddr2_s4_256M_x8_1066_1_875,` and so on`)` using `get_part_number()`
   - ✦ Memory depth `(MT42L128M16D1_18, jedec_lpddr2_s4_256M_x8_1066_1_875,` and so on`)` using `get_depth()`
   - ✦ Memory width `(SVT_MEM_64Mb, SVT_MEM_4Gb,` and so on`)` using `get_width()`
   - ✦ Clock rate `(SVT_MEM_600MHz, SVT_MEM_1500MHz,` and so on`)` using `get_clkrate()`
4. Once a part is selected, then use `get_cfgfile()` to get the path to the *.cfg file which can be loaded into the configuration object.

You can find online documentation about the `svt_mem_part_mgr` class using the LPDDR UVM Online Help located at:

```
$DESIGNWARE_HOME/vip/svt/lpddr_svt/latest/doc/lpddr_svt_uvm_class_reference/html/index.
html
```

8
Using Protocol Analyzer with Memory Models

8.1 Overview

Synopsys provides Protocol Analyzer (PA) to help you debug designs with LPDDR Memory VIP. The purpose of this chapter is to show you how to enable the generation of PA data, and an overview of PA features which support debugging testbenches with LPDDR VIP.

8.2 Enabling the LPDDR VIP to Generate Data for Protocol Analyzer

To use Protocol Analyzer to debug memory transactions, you must enable the LPDDR VIP to generate data for the Protocol Analyzer. In your UVM environment class, set the following to generate data for the Protocol Analyzer:

```
// Members to generate data defined in top level svt_lpddr_configuration class.
   svt_lpddr_configuration cfg;

// Enable .mempa file generation from memcore.
   cfg.enable_memcore_xml_gen = 1;

// Enable XML generation for memory transaction activity.
   cfg.enable_xact_xml_gen = 1;

// Enable XML generation for FSM (all state changes) activity.
   cfg.enable_fsm_xml_gen = 1;
```

8.3 Using Native Protocol Analyzer for Debugging

8.3.1 Introduction

This feature enables you to invoke Protocol Analyzer from Verdi GUI. You can synchronize the Verdi wave window, smart log and the source code with the Protocol Analyzer transaction view. Protocol Analyzer can be enabled in an interactive and post-processing mode. The new features available in Native Protocol Analyzer includes layer based grouping of the transactions, Quick filter, Call stack, horizontal zoom and reverse debug with the interactive support.

8.3.2 Prerequisites

Protocol Analyzer uses transaction-level dump database. You can use the following settings to dump the transaction database:

8.3.3 Compile Time Options

❖ `.-lca`

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

❖ `.+define+SVT_FSDB_ENABLE //` enables FSDB dumping

❖ `-debug_access`

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

You can dump the transaction database either by setting the `pa_format_type` configuration variable or by passing the runtime switch as shown below:

Runtime Switch:

```
+svt_enable_pa=fsdb
```

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

8.3.4 Invoking Protocol Analyzer

Perform the following steps to invoke Protocol Analyzer in interactive or post-processing mode.

8.3.4.1 Post-processing Mode

1. Load the transaction dump data and issue the following command to invoke the GUI:

   ```
   verdi -ssf <dump.fsdb> -lib work.lib++
   ```

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

8.3.4.2 Interactive Mode

1. Issue the following command to invoke Protocol Analyzer in an interactive mode:

   ```
   <simv> -gui=verdi
   ```

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

8.3.5 Limitations

Interactive support is available only for VCS.

9 Using the LPDDR Verification IP

9.1 Introduction

This chapter provides code examples of common testbench tasks when using the LPDDR Verification IP.

9.2 Using the UVM Basic QuickStart

Synopsys provides two UVM Basic QuickStart examples: one for LPDDR2 and one for LPDDR3. The example illustrates the following:

- ❖ Connection using modports.
- ❖ Building a UVM environment.
- ❖ Instantiation and configuration of the models.
- ❖ LPDDR UVM scoreboard.
- ❖ Coverage generation.
- ❖ Generating stimulus.
- ❖ Backdoor operations.

9.3 Setting The LPDDR Protocol Level of the VIP

This is an important step. You must set the protocol level of the model to match your license key. Otherwise, the model will not execute. The `vip_type` member in the top-level configuration class `svt_lpddr_configuration` sets the protocol level.

```
class cust_svt_lpddr_configuration extends svt_lpddr_configuration;

    // Selecting the vip as LPDDR2.
    this.vip_type = svt_lpddr_configuration::LPDDR2;

    // Enable PA mem core XML file generation
    this.enable_memcore_xml_gen = 1;
```

9.4 Using Catalogs

The following UVM example explainsthe usage of catalogs.

```
//--------------------------------------------------------------------------------
  /** This method demonstrates various features of the catalog system. */
  virtual function void generate_configuration();
    svt_lpddr2_vendor_part lpddr2_vendor_part;

    // Display the parts available in the catalog
    svt_lpddr2_vendor_catalog::write_shelf();

    // Select a completely random part (no constraints)
    lpddr2_vendor_part = svt_mem_part_mgr#(svt_lpddr2_vendor_part)::pick();
    if (lpddr2_vendor_part != null) begin
      `uvm_info("generate_configuration", $sformatf("Chose %s using an unconstrained
         selection. This part has the following characteristics:%s",
         lpddr2_vendor_part.get_part_number(),
         extract_part_details(lpddr2_vendor_part)), UVM_LOW);
    end

    // Use a policy class to select only parts smaller than 8G
    lpddr2_vendor_part =
      svt_mem_part_mgr#(svt_lpddr2_vendor_part,svt_lpddr2_lt_8G_policy)::pick();
    if (lpddr2_vendor_part != null) begin
      `uvm_info("generate_configuration", $sformatf("Chose %s using a policy class to
      exclude parts smaller than 8G. This part has the following characteristics:%s",
      lpddr2_vendor_part.get_part_number(), extract_part_details(lpddr2_vendor_part)),
      UVM_LOW);
    end

    // The load_prop_vals() method is used to load the configuration values for the
    // selected part into the configuration object.
    if (cfg.load_prop_vals(lpddr2_vendor_part.get_cfgfile())) begin
      `uvm_info("generate_configuration", $sformatf("Successfully loaded the
      configuration values for %s", lpddr2_vendor_part.get_part_number()), UVM_LOW);
    end
    else begin
      `uvm_fatal("generate_configuration", $sformatf("Unable to load the configuration
         values for %s", lpddr2_vendor_part.get_part_number()));
    end

  endfunction
```

Note
In the similar way as above, the example can be modeled in LPDDR2, LPDDR3, LPDDR4 and LPDDR5.

9.5 Initialize Memory Directly from the Testbench

Initialize a segment of memory using the backdoor access method `svt_mem_core::intialize()`.

```
// Initialize the memory with various patterns.
// Initialize the memory with CONSTANT Pattern.
   mem_core.initialize (svt_mem_core::INIT_CONST ,'h1111, 'h1020 , 'h102f) ;

// The seeded data ('h0001) will be left shifted for every consecutive address
mem_core.initialize (svt_mem_core::INIT_WALK_LEFT ,'h0001, 'h1040 , 'h104f) ;

// The seeded data ('h1000) will be right shifted for every consecutive address
   mem_core.initialize (svt_mem_core::INIT_WALK_RIGHT ,'h1000, 'h1050 , 'h105f) ;
```

```
// The seeded data ('haaaa) will be decremented for every consecutive address
   mem_core.initialize (svt_mem_core::INIT_DECR ,'haaaa, 'h1060 , 'h106f) ;

// The seeded data ('hbbbb) will be incremented for every consecutive address
   mem_core.initialize (svt_mem_core::INIT_INCR ,'hbbbb, 'h1070 , 'h107f) ;
```

9.6 Loading and Dumping Data Using a Memory Initialization File (MIF)

The following shows code for loading and dumping to and from a *.MIF file.

```
// Load the memory with the datafile "loadfile.mif"
   mem_core.load("./env/loadfile.mif" ,0);
// Dump the contents of the memory locations between given address.
    mem_core.dump("dumpfile.mif","MIF",0,'h1000,'h100c);
```

9.7 Using Peek and Poke (Backdoor Access)

The following shows some uses of both backdoor functions `peek()` and `poke()`.

```
/* Create a task to display memory contents obtained by peek().
 * The first two arguments are address ranges between which the contents
 * value is displayed and the third argument is the pointer of the memory
 */
  task display_contents (bit [31:0] start_address , bit [31:0] end_address,
      svt_mem_backdoor mem_back_door);
svt_mem_data_t rd_data;
    int loop_max;
    loop_max = end_address - start_address;
    for ( int loop = 0 ; loop <= loop_max; loop = loop + 1) begin
      mem_back_door.peek (start_address + loop, rd_data);
      `uvm_info("display_contents",$sformatf("Address: %h contains the Data: %h",
         start_address + loop, rd_data), UVM_LOW);
    end
  endtask: display_contents
```

The following example illustratesthe loading of memory, poking of values to un-initialized locations, and the usageof previous task to display the values loaded into memory.

```
// Load the memory with the datafile "loadfile.mif"
   mem_core.load("./env/loadfile.mif" ,0);

//Do backdoor poke to additional locations not initialized by the data file
    mem_back_door.poke ('h100b, 'hbbbb);
    mem_back_door.poke ('h100c, 'hcccc);

// Peek into tthe memory locations initialized by the datafile and backdoor
// poke and display them
    display_contents ('h1000 , 'h100c , mem_back_door);
    display_contents ('h2000 , 'h2002 , mem_back_door);
    display_contents ('h0200 , 'h020f , mem_back_door);
    display_contents ('h0300 , 'h030f , mem_back_door);
    display_contents ('h0400 , 'h0403 , mem_back_door);
```

9.8 Comparing the Contents of Memory

```
//Free complete memory
   mem_core.free_all();

// Load the memory with datafile
    mem_core.load("./env/loadfile.mif" ,0);

// Backdoor write to a address which is not initialized by
// the datafile. Datafile is subset of memory since memory
// has additional address initialized
   mem_back_door.poke ('h100b, 'hbbbb);

// The memory has additional location ‘h100b initialized which is
// not in datafile. In the comparison it should be ignored
   mem_core.compare ("./env/loadfile.mif" ,svt_mem_core::SUBSET , 1000,'h1000 ,
      'h100b);
```

10
Partition Compile and Precompiled IP

In design verification, every compilation and recompilation of the design and testbench contributes to the overall project schedule. A typical System-on-Chip (SoC) design may have one or more VIPs where changes are performed in the design or the testbench outside of VIPs. During the development cycle and the debug cycle, the complete design along with the VIP is recompiled, which leads to increased compilation time.

Verification Compiler offers the integration of VIPs with Partition Compile (PC) and Precompiled IP (PIP) flows. This integration offers a scalable compilation strategy that minimizes the VIP recompilations, and thus improves the compilation performance. This further reduces the overall time to market of a product during the development cycle and improves the productivity during the debug cycle.

The PC and PIP features in Verification Compiler provide the following solutions to optimize the compilation performance:

- The partition compile flow creates partitions (of module, testbench or package) for the design and recompiles only the changed or the modified partitions during the incremental compile.
- The PIP flow allows you to compile a self-contained functional unit separately in a design and a testbench. A shared object file and a debug database are generated for a self-contained functional unit. All of the generated shared object files and debug databases are integrated in the integration step to generate a simv executable. Only the required PIPs are recompiled with incremental changes in design or testbench.

For more information on the partition compile and Precompiled IP flows, see the VCS/VCS MX LCA Features Guide.

10.1 Use Model

You can use the three new simulation targets in the Makefiles of the VIP UVM examples to run the examples in the partition compile or the precompiled IP flow. In addition, Makefiles allow you to run the examples in back-to-back VIP configurations. The VIP UVM examples are located in the following directory:

```
$VC_HOME/examples/vl/vip/svt/vip_title/sverilog
```

For example,

```
/project/vc_install/examples/vl/vip/svt/lpddr_svt/sverilog
```

Each VIP UVM example includes a configuration file called as the pc.optcfg file. This configuration file contains predefined partitions or precompiled IPs for the SystemVerilog packages that are used by VIP. The predefined partitions are created using the following heuristics:

- ❖ Separate partitions are created for packages that are common to multiple VIPs.
- ❖ The VIP level partitions are defined in a way that all of the partitions are compiled in the similar duration of time. This enables the optimal use of parallel compile with the -fastpartcomp option.

Note
You can modify the pc.optcfg configuration file to include additional testbench or DUT level partitions.

10.2 The vcspcvlog Simulator Target in Makefiles

The vcspcvlog simulator target in the Makefiles of the VIP UVM examples, enable compilation of the examples in the two-step partition compile flow. The following partition compile options are used:

```
-partcomp +optconfigfile+pc.optcfg -fastpartcomp=j4 -lca
```

One partition is created for each line specified in the `pc.optcfg` configuration file. The `-fastpartcomp=j4` option enables parallel compilation of partitions on different cores of a multi-core machine. You can incorporate the partition compile options listed above into your existing vcs command line.

In the partition compile flow, changes in the testbench, VIP, or DUT source code trigger recompilation in only the required partitions. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

10.3 The vcsmxpcvlog Simulator Target in Makefiles

The vcsmxpcvlog simulator target in the Makefiles of the VIP UVM examples enables compilation of the examples in the three-step partition compile flow. The following partition compile options are used:

```
-partcomp +optconfigfile+pc.optcfg -fastpartcomp=j4 -lca
```

There is no change in the vlogan commands. One partition is created for each line specified in the pc.optcfg configuration file. The -fastpartcomp=j4 option enables parallel compilation of partitions on different cores of a multi-core machine. You can incorporate the partition compile options listed above into your existing vcs command line.

In the partition compile flow, changes in the testbench, VIP, or DUT source code trigger recompilation only in the required partitions. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

10.4 The vcsmxpipvlog Simulator Target in Makefiles

The vcsmxpipvlog simulator target in the Makefiles of the VIP UVM examples enables compilation of the examples in the PIP flow. There is no change in the vlogan commands. One PIP compilation command with the -genip option is created for each line specified in the pc.optcfg configuration file. The -integ option is used in the integration step to generate the simv executable.

In the PIP flow, changes in the testbench, VIP, or DUT source code trigger recompilation in only the required PIPs. You must ensure that the Verification Compiler compilation database is not deleted between successive recompilations.

10.5 Partition Compile and Precompiled IP Implementation in Testbenches with Verification IPs

You can use the Makefiles in the VIP UVM examples as a template to set up the partition compile or PIP flow in your design and verification environment by performing the following steps:

- ❖ Modify the pc.optcfg configuration file to include the user-defined partitions. The recommendations are as follows:
  - ✦ Create four to eight overall partitions (DUT and VIP combined).
  - ✦ Some VIP packages may include separate packages for transmitter and receiver VIPs. If only a transmitter or a receiver VIP is required, then the unused package can be removed from the configuration file.
  - ✦ Continue to use separate partitions for common packages, such as `uvm_pkg` and `svt_uvm_pkg`, as defined in the VIP configuration file.
- ❖ Incorporate the partition compile or precompiled IP command line options documented in previous sections or issued by the Makefile targets into the vcs command lines.

For more information on partition compile and precompiled IP options, such as, `-sharedlib` and `-pcmakeprof`, see the VCS/VCS MX LCA Features Guide.

10.6 Example

The following are the steps to integrate VIPs into the partition compile and PIP flows:

1. Once you set the `VC_HOME` variable, the `VC_VIP_HOME` variable is automatically set to the following location:

   ```
   $VC_HOME/vl
   ```
2. Verify VIP examples using the following command:

   ```
   $VC_VIP_HOME/bin/dw_vip_setup -i home
   ```
3. Install the example.

   For example, to install the LPDDR UVM Basic Example, use the following command:

   ```
   $VC_VIP_HOME/bin/dw_vip_setup -e lpddr_svt/tb_lpddr_svt_uvm_basic_sys -svtb
   cd examples/sverilog/lpddr_svt/tb_lpddr_svt_uvm_basic_sys
   ```
4. Run the tests present in the tests directory in the example.

   For example, to run the ts.base_test.sv test in the VCS two-step flow with partition compile, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcspcvlog
   ```

   To run the *ts.base_test.sv* test in the VCS UUM flow with partition compile, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcsmxpcvlog
   ```

   To run the *ts.base_test.sv* test in the VCS UUM flow with precompiled IP, use the following command:

   ```
   gmake base_test USE_SIMULATOR=vcsmxpipvlog
   ```
5. To modify or change the partitions, you must change the pc.optcfg file for the example.

11 Integrated Planning for VC VIP Coverage Analysis

To leverage the Verdi planning and management solutions, the VIP verification plans in the .xml format were required to be converted to the .hvp format manually. Now, VC VIPs provide an executable Verification Plan in the .hvp format together with the .xml format. You can now load the VIP verification plans easily to the Verdi verification and management solutions in a single step. Further, you can easily integrate these plans to the top level verification plan by a single click drag and drop.

Coverage results are annotated to the plan that helps to map the verification completeness on a feature by feature basis at the aggregate level.

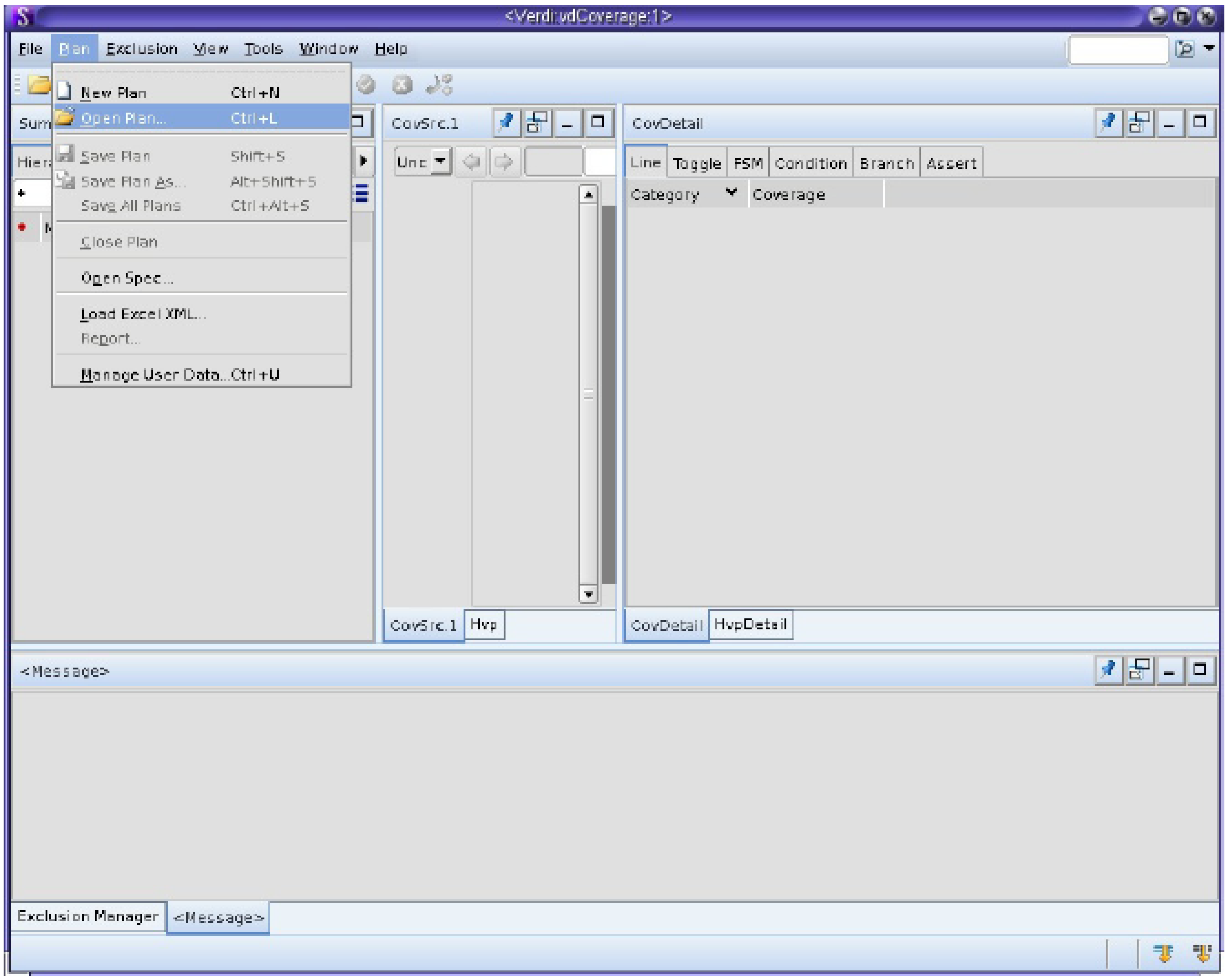
11.1 Use Model

The Verdi Coverage flow requires the .hvp files that capture the Verification Plan to be loaded on the Verdi Coverage Graphical User Interface (GUI), and the coverage annotated within the GUI.

To leverage the verification plan, perform the following steps:

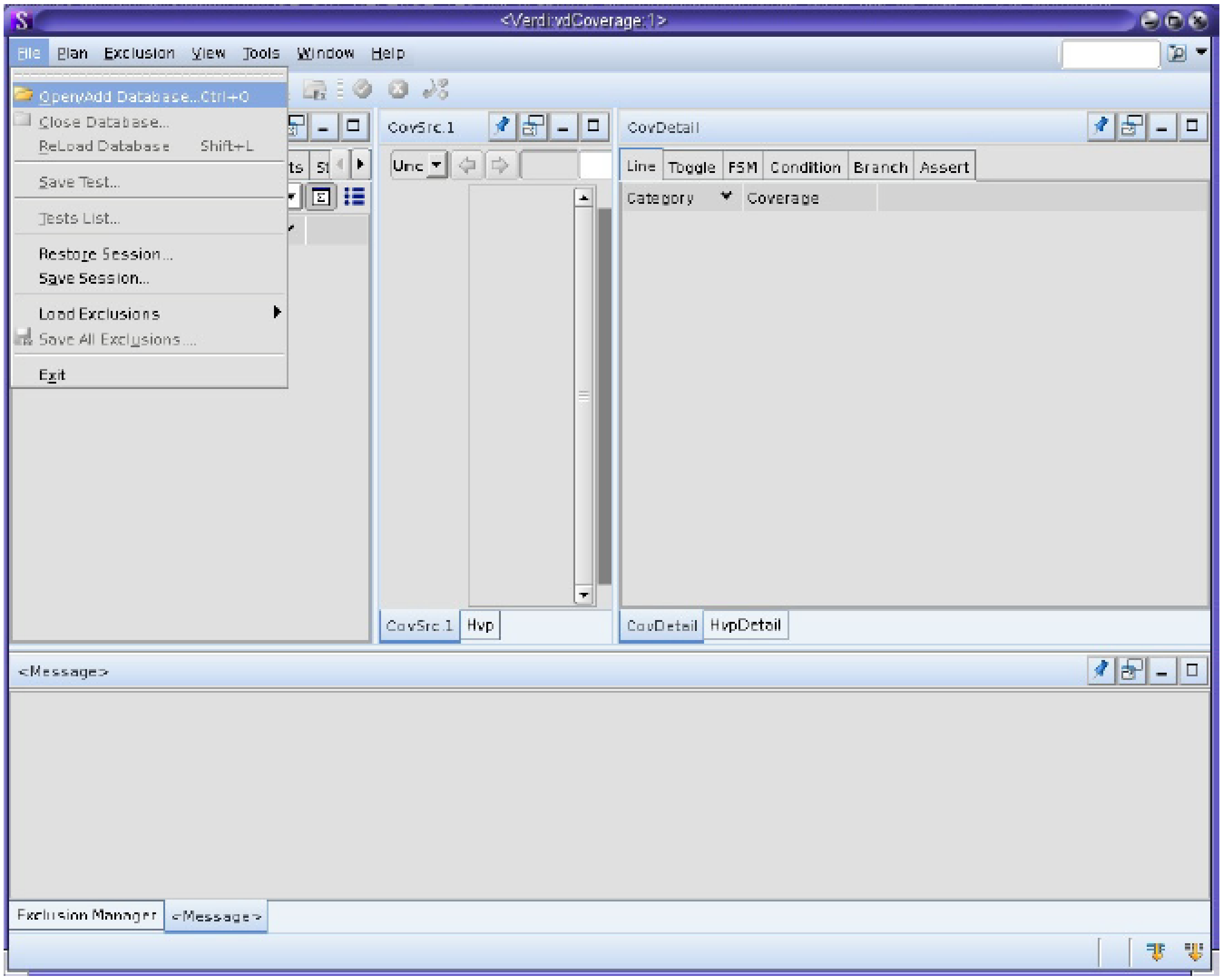
1. Navigate to the example directory using the following command:
   ```
   cd <intermediate_example_dir>
   ```
2. 2.Invoke the make command with the name of the test you want to run, as follows:
   ```
   gmake USE_SIMULATOR=vcsvlog <test_name>
   ```
3. Invoke the Verdi GUI using the following command:
   ```
   verdi -cov &
   ```
4. Copy the Verification Plans folder from the installation path to the current work area, as follows:
   ```
   cp $VC_VIP_HOME/vip/svt/lpddr_svt/latest/doc/VerificationPlans.
   ```
5. Select the load plan from the Plan drop down menu as shown in the following illustration.

Figure 11-1 Open Plan



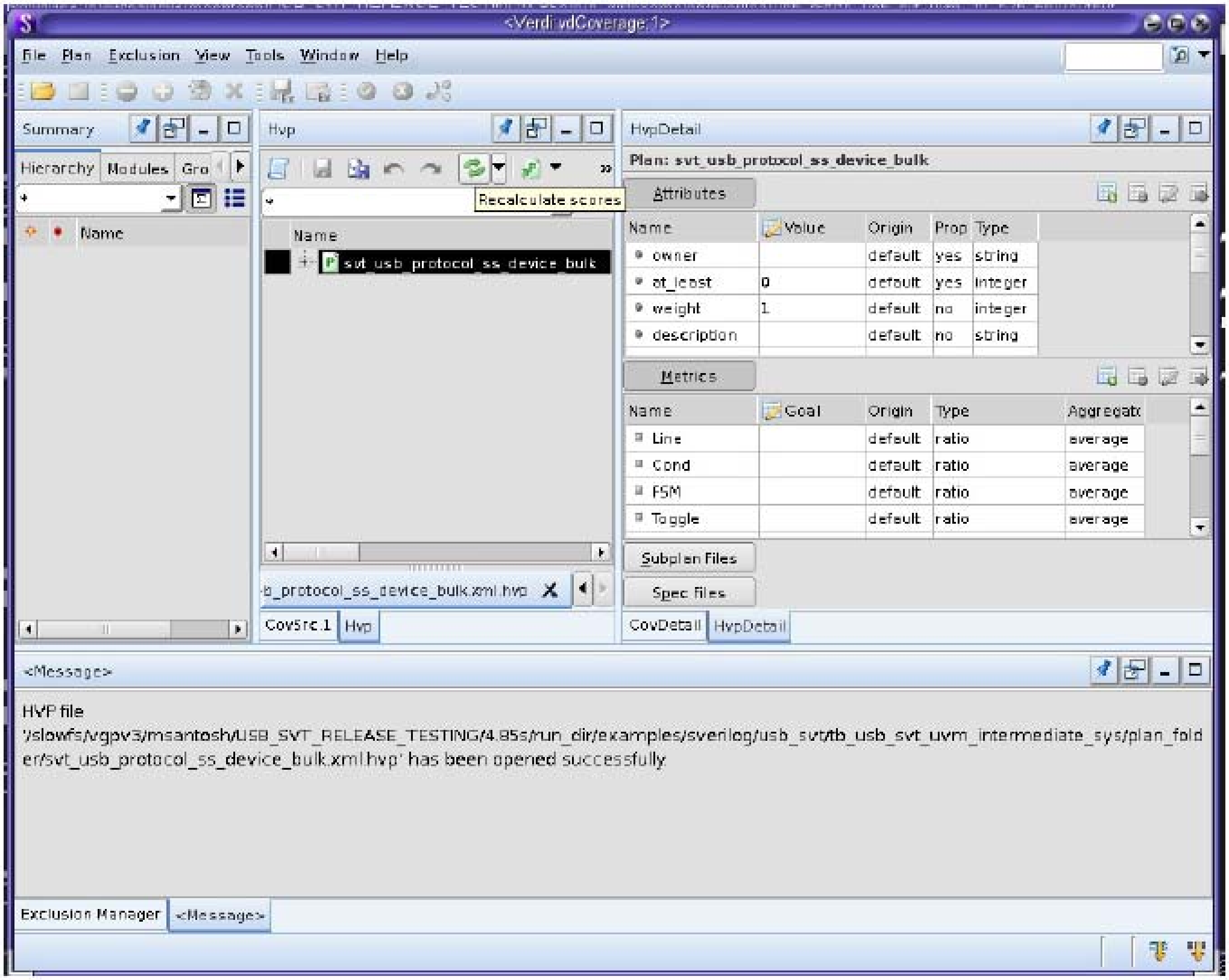
6. Load the coverage database (simvcssvlog.vdb) from the output folder in the current directory using the Verdi drop down menu, as shown in the following illustration.

Figure 11-2 Verdi GUI



7. Click Recalculate button to annotate the coverage results, as shown in the following illustration.

Figure 11-3 Recalculate



Plan: svt_usb_protocol_ss_device_bulk

Name	Value	Origin	Prop	Type
owner		default	yes	string
at_least	0	default	yes	integer
weight	1	default	no	integer
description		default	no	string

Name	Goal	Origin	Type	Aggregat
Line		default	ratio	average
Cond		default	ratio	average
FSM		default	ratio	average
Toggle		default	ratio	average

HVP file
'/slowfs/vgpv3/msantosh/USB_SVT_RELEASE_TESTING/4.95s/run_dir/examples/sverilog/usb_svt/tb_usb_svt_uvm_intermediate_sys/plan_folder/svt_usb_protocol_ss_device_bulk.xml.hvp' has been opened successfully.

A
Reporting Problems

A.1 Introduction

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

A.2 Debug Automation

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

❖ Enabled by the use of a command line run-time plusarg.

❖ Can be enabled on individual VIP instances or multiple instances using regular expressions.

❖ Enables debug or verbose message verbosity:

  ✦ The timing window for message verbosity modification can be controlled by supplying `start_time` and `end_time`.

❖ Enables at one time any, or all, standard debug features of the VIP:

  ✦ Transaction Trace File generation.

  ✦ Transaction Reporting enabled in the transcript.

  ✦ PA database generation enabled.

  ✦ Debug Port enabled.

  ✦ Optionally, generates a file name `svt_model_out.fsdb` when Verdi libraries are available.

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

A.3 Enabling and Specifying Debug Automation Features

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

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

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

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

The following table explains each control string:

Table A-1 Control Strings for Debug Automation plusarg

Field	Description
inst	Identifies the VIP instance to apply the debug automation features. Regular expressions can be used to identify multiple VIP instances. If this value is not supplied, and if a type value is not supplied, then the debug automation feature will be enabled on all VIP instances.
type	Identifies a class type to apply the debug automation features. When this value is supplied then debug automation will be enabled for all instances of this class type.
feature	Identifies a sub-feature that can be defined by VIP designers to identify smaller grouping of functionality that is specific to that title. The definition and implementation of this field is left to VIP designers, and by default it has no effect on the debug automation feature. (Specific to VIP titles)
start_time	Identifies when the debug verbosity settings will be applied. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the verbosity settings will be applied at time zero.
end_time	Identifies when the debug verbosity settings will be removed. The time must be supplied in terms of the timescale that the VIP is compiled. If this value is not supplied, then the debug verbosity remains in effect until the end of the simulation.
verbosity	Message verbosity setting that is applied at the `start_time`. Two values are accepted in all methodologies: DEBUG and VERBOSE. UVM and OVM users can also supply the verbosity that is native to their respective methodologies (`UVM_HIGH/UVM_FULL` and `OVM_HIGH/OVM_FULL`). If this value is not supplied then the verbosity defaults to `DEBUG/UVM_HIGH/OVM_HIGH`. When this feature is enabled, then all VIP instances that are enabled for debug will have their messages routed to a file named `svt_debug.transcript`.

For example,

- ❖ Enabled on all VIP instances with default options:

  +svt_debug_opts

- ❖ Enabled on all instances:
  - ✦ containing the string "endpoint" with a verbosity of UVM_HIGH.
  - ✦ starting at time zero (default) until the end of the simulation (default):

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

- Enabled on all instances:
  - starting at time 1000 until time 1500:

    ```
    +svt_debug_opts=start_time:1000,end_time:1500,verbosity:VERBOSE
    ```
- Enabled debug feature on all instances using default options:
  - By setting the macro `SVT_DEBUG_OPTS` to 1 in the command line, the debug feature is enabled on all instances using default options. The macro will enable the XMLs and trace files.

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

Note
The SVT_DEBUG_OPTS option is available through the installed VIP examples, but if required, in customer environments, then a similar feature should be added to their environment.
The PA=FSDB option is available in public examples and is required to enable Verdi libraries, and that when this option is used, then the Debug Opts file will record VIP activity to a file named `svt_model_log.fsdb.`
In addition, the SVT Automated Debug feature will enable waveform generation to an FSDB file, if the Verdi libraries are available. When enabled this feature, it should cause the simulator to dump waveform information only for the VIP interfaces.

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

A.4 Debug Automation Outputs

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

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

The following are the output files generated:

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

A.5 FSDB File Generation

To enable FSDB writing capabilities, the simulator compile-time options and environment must be updated to enable this. The steps to enable this are specific to the simulator being used (the {LINUX/LINUX64} label needs to be replaced based on the platform being used). The ability to write to an FSDB file requires that the user supplies the Verdi dumper libraries when they compile their testbench. If these are not supplied then the VIP will not be enabled to generate the `svt_model_log.fsdb` file.

A.5.1 VCS

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

```
-debug_access
```

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

A.5.2 Questa

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

```
+define+SVT_FSDB_ENABLE -pli novas_fli.so
```

A.5.3 Incisive

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

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

A.6 Initial Customer Information

Follow these steps when you call the Synopsys Support Center:

1. Before you contact technical support, be prepared to provide the following:
   - ✦ A description of the issue under investigation.
   - ✦ A description of your verification environment.

Enable the Debug Opts feature. For more information, see the "Debug Automation" on page 77.

A.7 Sending Debug Information to Synopsys

To help you debug testing issues, follow the given instructions to pack all pertinent debug information into one file which you can send to Synopsys (or to other users in your company):

1. Create a description of the issue under investigation. Include the simulation time and bus cycle of the failure, as well as any error or warning messages that are part of the failure.
2. Create a description of your verification environment. Assemble information about your simulation environment, making sure to include:
   - ✦ OS type and version
   - ✦ Testbench language (SystemVerilog or Verilog)
   - ✦ Simulator and version
   - ✦ DUT languages (Verilog)

3. Use the VIP case submittal tool to pack a file with the appropriate debug information. It has the following usage syntax:

   ```
   $DESIGNWARE_HOME/bin/snps_vip_debug [-directory <path>]
   ```

   The tool will generate a <*username*>.<*uniqid*>.*svd* file in the current directory. The following files are packed into a single file:

   - FSDB
   - HISTL
   - MISC
   - SLID
   - SVTO
   - SVTX
   - TRACE
   - VCD
   - VPD
   - XML

   If any one of the above files are present, then the files will be saved in <*username*>.<*uniqid*>.*svd* in the current directory. The simulation transcript file will not be part of this and it will be saved separately.

Note
The -directory switch can be specified to select an alternate source directory.

4. You will be prompted by the case submittal tool with the option to include additional files within the SVD file. The simulation transcript files cannot be automatically identified and it must be provided during this step.
5. The case submittal tool will display options on how to send the file to Synopsys.

A.8 Limitations

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

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