Verification Continuum™

VC Verification IP
DDR Memory
UVM User Guide

Version Q-2020.06, June 2020

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

Preface

About This Manual

This manual contains installation, setup, and usage material for SystemVerilog UVM users of the Synopsys DDR, and is for design or verification engineers who want to verify DDR operation using a UVM testbench written in SystemVerilog. Readers are assumed to be familiar with DDR, 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 Product version for which you want to register the problem.
   - If applicable, provide the information noted in Appendix A, "Reporting Problems" on page 59.
3. Telephone your local support center.
   - North America:

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

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

1 Introduction

1.1 Overview

The Synopsys DDR Verification IP supports verification of designs that include DDR memory. This document describes the use of DDR 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 DDR protocol, object oriented programming, SystemVerilog, and UVM.

Synopsys provides a DDR UVM basic QuickStart example showing many features of both the UVM methodology and tasks showing how to use the model. It is located at:

$DESIGNWARE_HOME/.../ddr_svt/examples/sverilog/tb_ddr3_svt_uvm_basic_sys

The README file describes how to run the example.

1.2 Product Overview

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

1.3 DDR VIP Model Features

The following subsections provide an overview of supported DDR features for verification.

- Supports the DDR3, DDR4 and DDR5 specifications.
- Support basic Memory functions such as initialize, save, restore, load, dump, and compare.
- Support Bypass Initialization
- Timing checks
  - Independent of the timescale used to compile the user environment
  - No timescale limit
  - Configurable values, bounds (min, max, typ)
  - Dynamic reconfiguration
- Backdoor access to memory contents
  - Backdoor access methods in virtual sequencer
  - Detect concurrent backdoor write with frontdoor read/write and backdoor read with frontdoor write.
  - Masked writes
  - Backdoor access to mode registers.
- Logical Memories
  - Subset
  - Width concatenation
  - Length concatenation
  - Interleaved concatenation
  - Address scrambling
  - Data scrambling
  - Holes
  - Endianness
  - Address granularity
  - Byte placement
- Force power-down
- Save/restore/compare to/from file
- Pre-defined MEMH policy (see $readmemh in Verilog LRM)
- Pre-defined MIF policy
- Protocol and Usage Checks

- Fast reset/initialization/calibration mode
- Access protocol
  - Protocol state transitions
- Differential polarity
- Memory accesses
  - Read before write
  - Two Writes without read
  - Two reads without write
  - Write different banks without read
  - Write same bank

- Rich debug capabilities
  - Access trace
  - Transactions into Verdi
  - Memory browser
    - All instances
    - Configuration view
    - Memory content view
  - Transactions into Performance Analyzer
  - Interactive, post-simulation
  - Transaction data content searchable
- Special Memory Mode in Protocol Analyzer

Note
For more details on Verdi, see Verdi documentation.

1.3.1 DDR3 and DDR4 JEDEC Specifications

- 8n Prefetch architecture (maintaining BL8 and BC4)
- Upto 4 bank groups in DDR4, each group has 4 banks and 8 Banks in DDR3
  - Up to 16 Gbit devices
- Clock speed bins up to 2400Mbps for DDR4.
- Signal integrity features
  - Data bus inversion (DBI)
  - Parity detection for command/address bus
  - Cyclical Redundancy Check (CRC) for error detection on data
- Write leveling
- ZQ calibration
- Power down
- Burst ordering

- Burst chop
- Command Encoding
- Preamble Training
- Internal DQ Vref
- Low Power Array Self Refresh
- Gear down Mode
- Per DRAM Addressability

1.3.2 DDR5 Features

- 2n mode – not default
- DDR5 RCD Parity
- DDR5 ODT mode reg
- Read Training Pattern
- Refresh sb, precharge sb
- Write partial
- PDA mode
- Read CRC
- CRC auto disable
- MRR commands
- MPC support
- CS training, CA training
- Write leveling
- CA parity
- Fast zero mode
- Memory density upto 32Gb except 24Gb
- Speedbin support from 3200 to 6400
- Write and Read with auto precharge
- 3DS support
- Upto 2 ranks supported

1.3.3 Methodology Features

The DDR VIP currently supports the following methodology functions:

- Analysis ports
- Callbacks
  - Pre/post read & write
  - Specific address, address range, address mask
  - Data compare, data mask

- Count, time range
- Physical-to-virtual address translation (only get physical addresses from the DUT)
- Physical address to chip select, bank, row and column addresses
  - Error injection capability
    - Provides a backdoor access mechanism for data corruption.
- Configuration objects

2 Installation and Setup

2.1 Introduction

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

If you encounter any problems with installing the Synopsys DDR, see "Customer Support" on page 7.

2.2 Verifying the Hardware Requirements

The DDR 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 Synopsys DDR is qualified for use with certain versions of platforms and simulators. This section lists software that the Synopsys DDR 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 DDR VIP, check the support matrix for "SVT-based" VIP in the following document:

  Support Matrix for SVT-Based DDR VIP is in:

  VC VIP DDR Release Notes

2.4 Synopsys Common Licensing (SCL) Software

- The SCL software provides the licensing function for the Synopsys DDR. Acquiring the SCL software is covered here in the installation instructions in "Licensing Information" on page 25.

2.5 Other Third Party Software

- **Adobe Acrobat**: Synopsys DDR documents are available in Acrobat PDF files. You can get Adobe Acrobat Reader for free from http://www.adobe.com.
- **HTML browser:** Synopsys DDR 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 DDR 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 ddr_agent_svt
-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 Synopsys 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 DDR-suite setup in the `design_dir` directory. In addition to the DDR 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  ddr_agent_svt -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 ddr_agent_svt -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 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 <*model*>:

```
/`timescale 1ps/100fs
`include "svt_ddr_defines.svi"
`include "svt_mem.uvm.pkg"

 // Import SVT UVM
 import svt_uvm_pkg::*;

 // Import SVT MEM UVM
 import svt_mem_uvm_pkg::*;

/** Include Memory Agent files */
`include "svt_ddr3_agent_hdl.sv"
`include "svt_ddr3_catalog.svi"
```

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`.

2.12 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/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/
// To run the example using the generated run script with +incdir+
./ run_ddr4_dimm_svt_uvm_basic_sys -incdir catalog_udimm_rw_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/ddr_svt/latest/sverilog/include \
+incdir+/<DESIGNWARE_HOME>/vip/svt/dfi_svt/latest/sverilog/include \
-ntb_opts uvm -full64 -sverilog +define+UVM_DISABLE_AUTO_ITEM_RECORDING
+define+UVM_PACKER_MAX_BYTES=1500000 \
+define+SVT_MEM_DPI_OK
+define+LOADFILE_MEMH_FILE=/<DESIGNWARE_HOME>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/env/loadfile.memh \
+define+LOADFILE_MIF_FILE=/<DESIGNWARE_HOME>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/env/loadfile.mif \
-timescale=1ps/100fs
/<DESIGNWARE_HOME>/vip/svt/common/latest/C/lib/amd64/libmemserver.so \
/<DESIGNWARE_HOME> /vip/svt/common/latest/C/lib/amd64/libmemserver.so \
+define+SVT_UVM_TECHNOLOGY +define+SYNOPSYS_SV
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/. \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/../../env \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/../env \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/env \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/dut \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/hdl_interconnect \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_svt_uvm_basic_sys/lib \
+incdir+<testbench_dir>/examples/sverilog/ddr_svt/tb_ddr4_dimm_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.

2.13 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:

```
Usage:  run_ddr4_dimm_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 catalog_config_timing_update_test catalog_lrdimm_rw_test
catalog_rdimm_rw_test catalog_udimm_nibble_swap_test catalog_udimm_rw_test
dimm_monitor_callback_test dram_monitor_callback_test
driver_callback_rd_data_corruption_test enable_coverage_rw_test
nonjedec_data_width_rw_test randomized_dimm_configuration_rw_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
catalog_config_timing_update_test catalog_lrdimm_rw_test catalog_rdimm_rw_test
catalog_udimm_nibble_swap_test catalog_udimm_rw_test dimm_monitor_callback_test
dram_monitor_callback_test driver_callback_rd_data_corruption_test
enable_coverage_rw_test nonjedec_data_width_rw_test
randomized_dimm_configuration_rw_test
```

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

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`
Optional: `-timescale=1ns/1ps`
Required: `+define+SVT_<model>_INCLUDE_USER_DEFINES`
Required: `+define+SYNOPSYS_SV`
Required for NCV Simulator only
`${DESIGNWARE_HOME}/vip/svt/common/latest/C/lib/${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

Required: `+UVM_TESTNAME=$scenario`

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 Synopsys VIP libraries, models, example testbenches, and license version in a DESIGNWARE_HOME installation:

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

The following example lists the Synopsys VIP 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 DDR VIP is set up for use in a testbench. A design directory is required for using Synopsys 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 Synopsys DDR from DESIGNWARE_HOME to a design directory

For a full description of dw_vip_setup, refer to "The dw_vip_setup Utility" on page 31.

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 ddr_agent_svt_hdl -svtb
```

The models provided with Synopsys DDR include:

- ddr_agent_svt_hdl

2.17 Licensing Information

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

http://www.synopsys.com/keys

You must have access to a license server that contains at least one of the license combinations mentioned in the following table, corresponding to the protocol being used:

Protocol	License Keys
DDR5	VIP-DDR5-SVT VIP-DDR5-NVDIMMP-SVT VIP-DDR5-3DS-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT
DDR4	VIP-DDR4-SVT VIP-MEMORY-SVT VIP-SOC-LIBRARY-SVT VIP-DDR4-NVDIMMP-SVT VIP-DDR4-3DS-SVT VIP-DDR4-MRAM-SVT VIP-DDR5-SVT VIP-DDR5-NVDIMMP-SVT VIP-DDR5-3DS-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT
DDR3	VIP-DDR3-SVT VIP-DDR4-SVT VIP-MEMORY-SVT VIP-SOC-LIBRARY-SVT VIP-DDR4-NVDIMMP-SVT VIP-DDR4-3DS-SVT VIP-DDR4-MRAM-SVT VIP-DDR5-SVT VIP-DDR5-NVDIMMP-SVT VIP-DDR5-3DS-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT

Protocol	License Keys
DDR2	VIP-DDR2-SVT VIP-DDR3-SVT VIP-DDR4-SVT VIP-MEMORY-SVT VIP-SOC-LIBRARY-SVT VIP-DDR4-NVDIMMP-SVT VIP-DDR4-3DS-SVT VIP-DDR4-MRAM-SVT VIP-DDR5-SVT VIP-DDR5-NVDIMMP-SVT VIP-DDR5-3DS-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT
DDR5-NVDIMMP	VIP-DDR5-NVDIMMP-SVT VIP-LIBRARY2019-SVT+VIP-DDR5-NVDIMMP-EA-SVT
DDR4-NVDIMMP	VIP-DDR5-NVDIMMP-SVT VIP-DDR4-NVDIMMP-SVT VIP-LIBRARY2019-SVT
DDR5-3DS	VIP-DDR5-3DS-SVT VIP-LIBRARY2019-SVT
DDR4-3DS	VIP-DDR5-3DS-SVT VIP-DDR4-3DS-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT
DDR4-MRAM	VIP-DDR4-MRAM-SVT VIP-LIBRARY-SVT + DESIGNWARE-REGRESSION VIP-LIBRARY2019-SVT

Only one license is consumed per simulation session, no matter how many Synopsys VIP models are instantiated in the design. The license check is in order and feature names as per the following steps:

1. check VIP-<suite>-SVT
2. check VIP-LIBRARY-SVT + DesignWare-Regression

Note: “+” means “AND” and all those features are required.

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 28.

2.17.1 If Licensing Fails

By default, simulations exit with an error when a Synopsys 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 Synopsys 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 Synopsys DDR 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 productsoftware 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 DDR 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 Synopsys 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 Synopsys 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 Synopsys VIP into Your Testbench

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

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

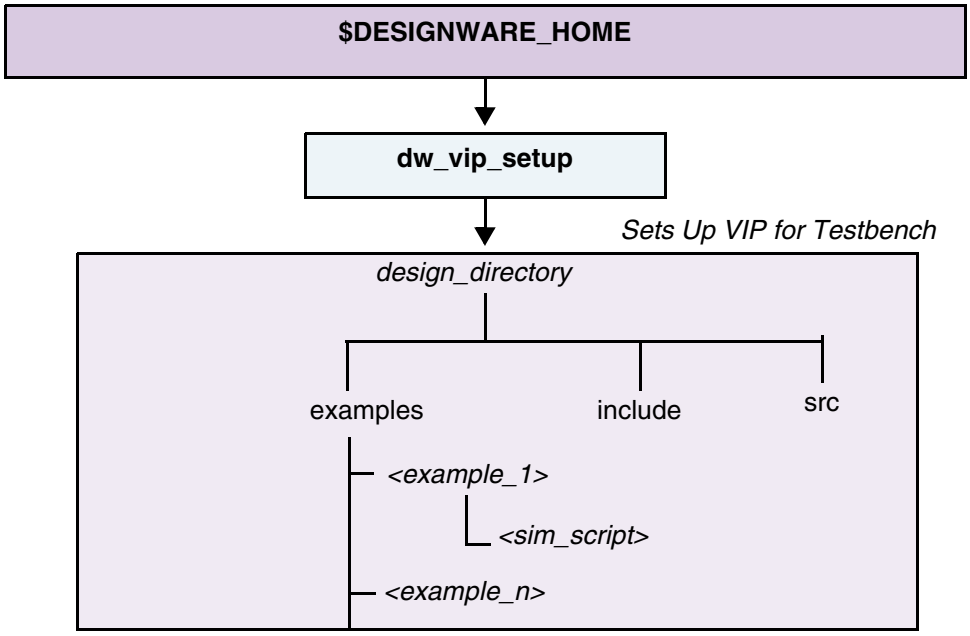
2.20.1 Creating a Testbench Design Directory

A *design directory* contains a version of the Synopsys 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 31.

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 Synopsys 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 Synopsys VIP includes example testbenches. The dw_vip_setup utility adds them in this directory, along with a script for simulation. If an example testbench is specified on the command line, this directory contains all files required for model, suite, and system testbenches.

include Language-specific include files that contain critical information for Synopsys VIP models. This directory is specified in simulator command lines.

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

.

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/ddr_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 ddr_agent_svt -svlog -doc`

   Design directory (/n/proj):

   ./doc/ddr_svt/ddr_svt_hdl_user_guide.pdf -> /tools/install/vip/svt/ddr_svt/J-2014.12/doc/ddr_svt_hdl_user_guide.pdf

```
.....
./doc/ddr_svt/ddr_svt_release_notes.pdf -> /tools/install/vip/svt/ddr_svt/
                                           J-2014.12/doc/ddr_svt_release_notes.pdf
./doc/ddr_svt/ddr_svt_uvm_getting_started.pdf -> /tools/install/vip/svt/ddr_svt/
                                           J-2014.12/doc/ddr_svt_uvm_getting_started.pdf
./doc/ddr_svt/ddr_svt_uvm_user_guide.pdf -> /tools/install/vip/svt/ddr_svt/
                                           J-2014.12/doc/ddr_svt_uvm_user_guide.pdf
./doc/ddr_svt/ddr_svt_vmm_user_guide.pdf -> /tools/install/vip/svt/ddr_svt/
                                           J-2014.12/doc/ddr_svt_vmm_user_guide.pdf
```

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

   Design directory (/n/proj):

   ./doc/ddr_svt/ddr_svt_uvm_getting_started.pdf

   ./doc/ddr_svt/ddr_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 Synopsys VIP models in a design directory
- Adds example testbenches to a design directory, the Synopsys 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 Synopsys 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 dfi_agent_svt. 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 `dfi_agent_svt`.
-u[pdate] (*model* [**-v**[ersion] *version*]) …	Updates to the specified model version for the specified model or models. The dw_vip_setup script updates to the latest models when you do not specify a version. The model names are is `dfi_agent_svt`. 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.

Table 2-1 Setup Program Switch Descriptions (Continued)

Switch	Description
-h[elp]	Returns a list of valid dw_vip_setup switches and the correct syntax for each.
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*	Specifies a file name which contains a list of model names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one model name per line and each model may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
-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.
-**s/**uite_list <filename>	Specifies a file name which contains a list of suite names to be added, updated or removed in the design directory. This switch is only valid when following an operation switch such as -add, -update or -remove. Only one suite name per line and each suite may include a version selector. The default version is 'latest'. This switch is optional, but if given the filename argument is required. Lines in the file starting with the pound symbol (#) will be ignored.
-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 DDR 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 provided for configuring the Synopsys DDR 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 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 DDR VIP defines the following configuration classes.

- svt_ddr_configuration. The base configuration class contains configuration information which is applicable to the svt_ddr_agent class to model DDR3/DDR4 discrete device components. This is the “top level” configuration class which includes the handles of the mode registers and the timing configuration classes.
- svt_ddr_mode_register_configuration. Base class for both DDR3 and DDR4 Mode Registers.
- svt_ddr_timing_configuration. The timing configuration class contains DDR timing parameters information which is applicable to individual DDR Memory or Controller components.
- svt_mem_configuration. This class contains configuration information which is applicable to svt_ddr_dimm_env. This is used for both DDR3 and DDR4 DIMMs which can be configured as DDR3/DDR4 LRDIMM/RDIMM/UDIMM.
- svt_mem_suite_configuration. This memory configuration class encapsulates the configuration information for a single memory core instance. This is a SVT base class common to memory VIPs.

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 DDR 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 DDR 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.

DDR transaction data objects are used to:

- 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.

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 analysis ports provide a standard interface for passing data objects between components. There are methods in analysis ports that are used to get and put data. One component puts the data while the other gets it. The analysis port is item_oberved_port for the monitor inside the svt_ddr_agent and svt_ddr_dimm_env.

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 DDR VIP defines the following exception classes:

- svt_ddr_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_ddr_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 append_callback() method of the transactor.

DDR VIP uses callbacks in the following applications:

- Access for functional coverage
- Access for scoreboarding

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_ddr_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_ddr_monitor_def_state_cov_callbacks State coverage is a signal level coverage.
- svt_ddr_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_ddr_monitor_def_toggle_cov_callbacks. Toggle coverage is a signal level coverage.
- svt_ddr_monitor_def_toggle_cov_data_callbacks. This callback class defines default data and event information that are used to implement the coverage groups. T

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

- svt_ddr_monitor_callback. Monitor callback class contains the callback methods called by the monitor component.
- svt_ddr_dimm_monitor_callback. DIMM Monitor callback class contains the callback methods called by the DIMM monitor 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 DDR driver communicates with the DDR 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 DDR VIP includes:

- svt_ddr3_jedec_if
- svt_ddr4_jedec_if
- svt_ddr3_dimm_if
- svt_ddr4_dimm_if
- svt_ddr5_dimm_if

See the DDR SVT - Interfaces Reference page in the HTML Class Reference for more information. "Interface Options" describes DDR 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.

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

The following techniques apply when working with randomization:

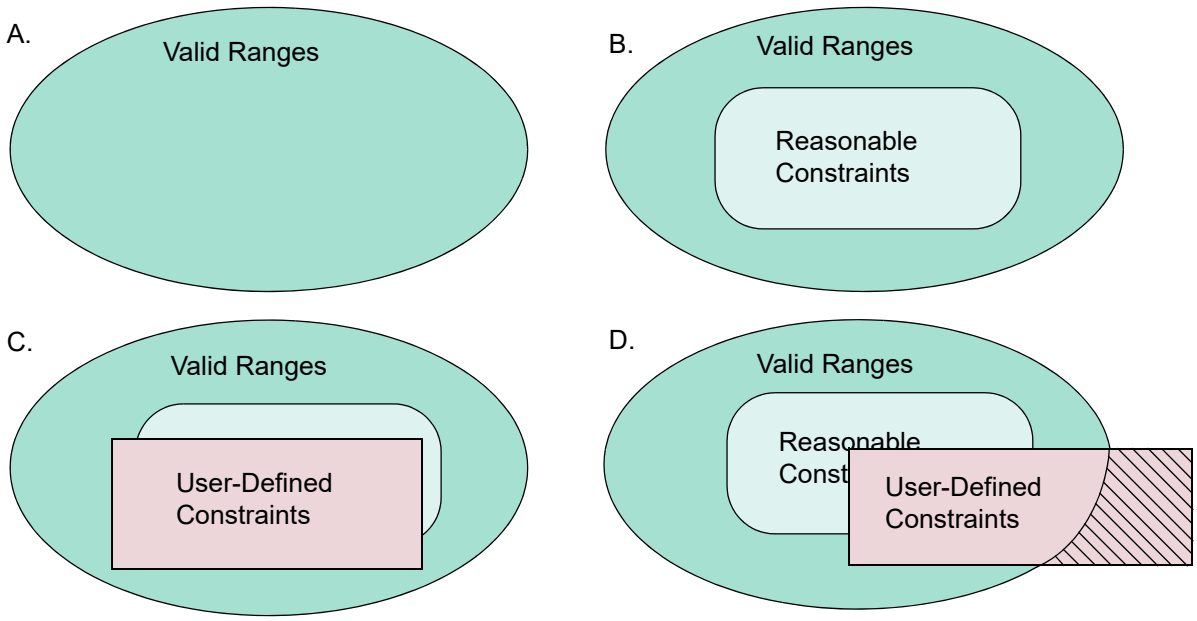
- Randomization only occurs when an object's randomize() method is called, and it is completely up to the test code when, or even if, this occurs.
- Constraints form a rule set to follow when randomization is performed. By controlling the constraints, the testbench has influence over the outcome. Direct 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



3.2.8 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:

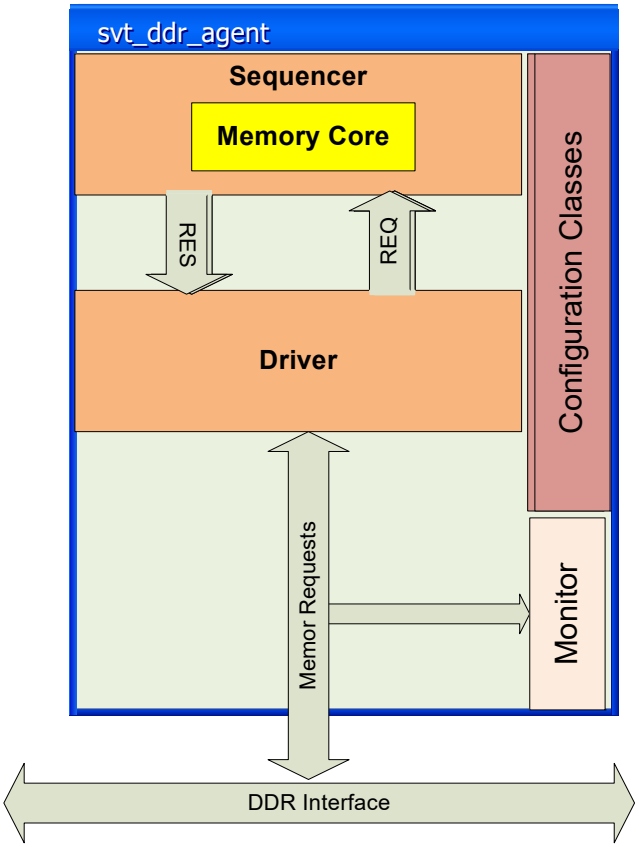
- **Type:** Messages are categorized into types. The possible types are listed in the UVM Manual.
- **Severity:** Severity is similar to the urgency of the message or how serious it is. The possible values for severity are listed in the UVM manual.

4 DDR Agent

4.1 Overview

Agents are UVM components that are composed of basic components such as drivers, monitors and sequencers. The DDR VIP defines an agent that can be customized to support a variety of verification scenarios. The name of the DDR memory agent is called **svt_ddr_agent** which is used for modelling DDR3/DDR4 discrete devices. The following figure shows the architecture of the VIP agent.

Figure 4-1 DDR Memory Agent Architecture



4.2 UVM Driver

The svt_ddr_agent agent contains a driver component called svt_ddr_driver. Because the svt_ddr_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 and calculates DBI on the data
- MPR Read - content of addressed MPR page
- Write Leveling - regular response based on observed DQS
- Write with CRC enabled - no alert response
- Command with C/A Parity - no alert 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.

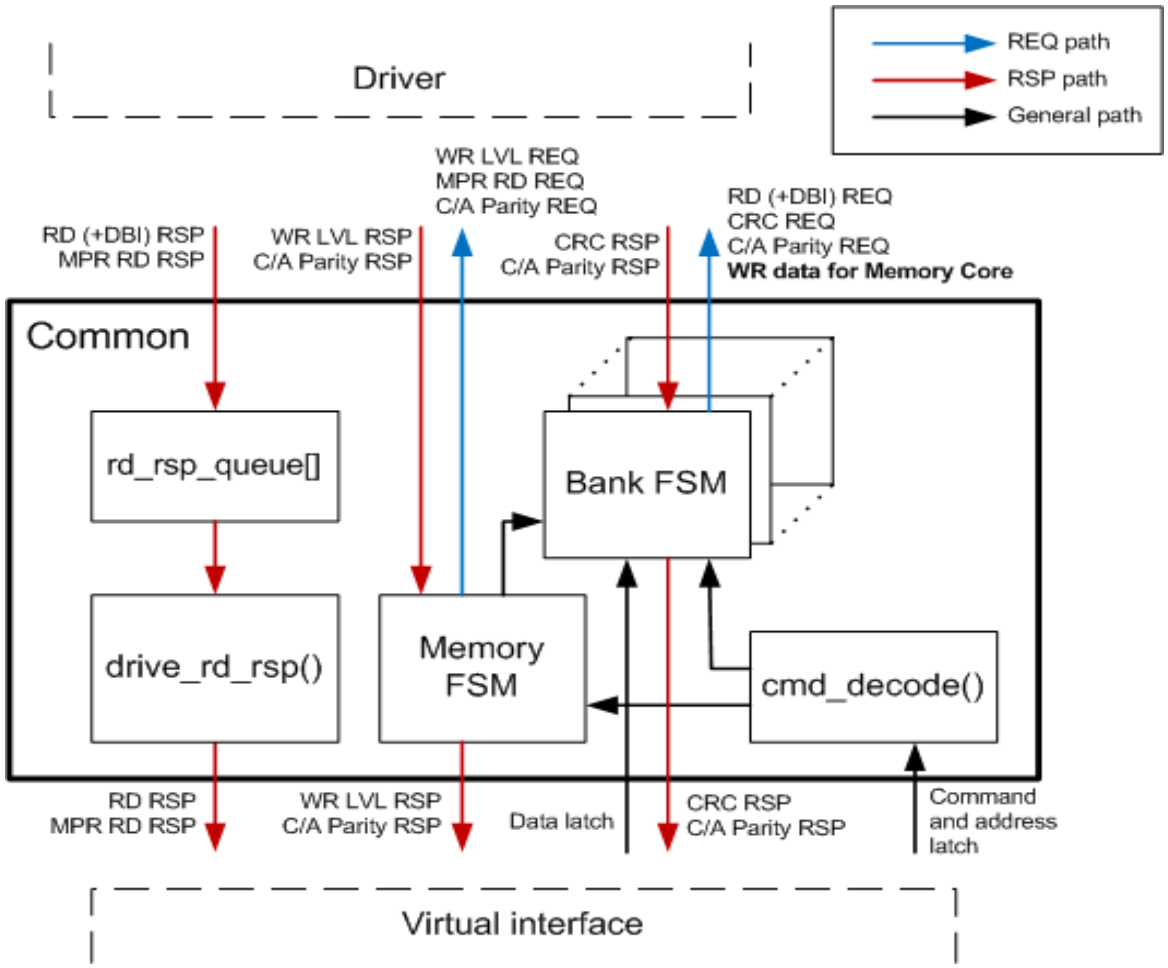
The memory core interacts with three interface classes to update the memory contents. The classes are:

- **svt_ddr_transaction**. Transaction class shared between DDR 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 General Command Flow

The following figure shows a high level view of the Agent's internal block diagram.

Figure 4-2 Internal Block Diagram



Following is the general command flow through the agent:

1. The cmd_decode() task recognizes the received command. If the received command is recognized as a valid (not DES/NOP) command, then cmd_decode() creates new transaction for the received command. It then updates the transaction with the address latched and triggers a new_cmd event.
2. The Memory FSM or an addressed Bank FSM may change the state on new_cmd event if any appropriate command is received.
3. If the received command is:
   a. A Read/Write/ACT/PRE(A): it is the addressed bank FSM's state duty to update other relevant transaction fields and take needed actions.
   b. Any other valid command: it is the Memory FSM's state duty to update other relevant transaction fields and take needed actions.

Note, you cannot access any of the internal interfaces.

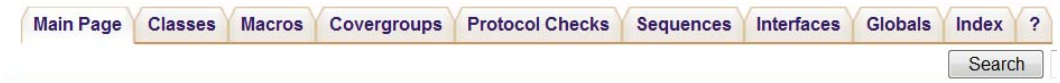
4.5 UVM Monitor

The DDR agent contains a UVM monitor called svt_ddr_monitor which has analysis port with the name of item_observed_port. 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 DDR HTML Class Reference at:

  $DESIGNWARE_HOME/vip/svt/ddr_svt/doc/ddr_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:



  The following shows a few protocol checks from the HTML help:

Protocol Check Class	Protocol Check Instance name	Group	Sub Group	Description	Reference	Error Category
svt_ddr_checker	write_odth8_timings_check	DDR		Time from Write command with BL=8 to ODT signal low ODTH8 was not satisfied	12.3 table 101-102	ERROR
svt_ddr_checker	write_odth4_timings_check	DDR		Time from Write command with BL=4 to ODT signal low ODTH4 was not satisfied	12.3 table 101-102	ERROR
svt_ddr_checker	odth4_timings_check	DDR		ODT signal high time ODTH4 with no Write command was not satisfied	12.3 table 101-102	ERROR

- Coverage information
- Callbacks
- Transaction logging

4.6 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
In active mode, the agent generates transactions on the signal interface.	In Passive mode, the agent monitors the interface signals, and performs passive functionality of coverage and protocol checking. Users can enable/disable this functionality through configuration.
Agent performs passive functionality of coverage and protocol checking. Users can enable/disable this functionality through configuration.	

4.7 Agent Configuration

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

- **svt_ddr_configuration** class. The svt_ddr_configuration class allows you to set the following device properties:
  - Protocol type. You can set the memory model to work at DDR2, DDR3 or DDR4. This setting must match your license.
  - Memory capacity
  - AC input levels
  - Type of address mapping
  - Power mode
- **svt_ddr_mode_register_configuration**
  - Configures mode registers for DDR3, DDR4 and DDR5.
  - Most DDR configuration is defined through Mode Registers.
- **svt_ddr_timing_configuration**.
  - Allows you set values for all timing intervals defined by the DDR protocol.

Note, the previous list defines the base agent configuration classes. In normal testbench construction you would configure the DDR agent by using catalog and part number *.cfg files defining the behavior of a Vendor part. 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_ddr_configuration in one step.

4.8 Virtual Interfaces

The DDR agent is the component which is connected to the DDR signals. The Agent has four types of interfaces:

- svt_ddr3_jedec_if
- svt_ddr4_jedec_if
- svt_ddr5_jedec_if

5 Memserver

5.1 Overview

A critical component of the DDR 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 DDR 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 0. 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_addr_t addr_lo ,
svt_mem_addr_t addr_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_addr_t addr_lo , svt_mem_addr_t addr_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 0 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, 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.

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 DIMMS

6.1 Introduction

The VIP supports DDR3 and DDR4 LRDIMMs, RDIMMs, SODIMMs, DDR5 RDIMM and UDIMM. The VIP allows the user to configure the structure of the DIMM for different bus width and rank structure. The DIMM environment allows modeling DIMM inner delays and making communication between DIMM subcomponents visible to the user.

The VIP could be configured with or without ECC support. This chapter presents the architecture with eight DRAMs (no ECC and X8 Bus width), however, similar architecture should be used for all other cases. When ECC is enabled for (X8 Bus width), then there will be an additional DRAM for ECC making the data width 72. For DDR5, with ECC, data width is 80 and without ECC 72.

A DIMM environment makes use of virtual interfaces of svt_ddr3_dimm_if and svt_ddr4_dimm_if. A DIMM environment has the following components:

- Eight instances of the DRAM agent with single sequencer and memory Core
- DIMM monitor
- SPD agent
- RCD component (LRDIMM/RDIMM)

All DIMM subcomponents are connected through physical interfaces so all the traffic can be easily tracked and delays can be easily modeled. DRAM agent's instantiated in the DIMM environment do not have a sequencer instances. All DRAM agents make use of a single sequencer instantiated in the DIMM environment and single memory core inside of it.

Figure 6-1 and following illustrations show this architecture.

Figure 6-1 DDR4 LRDIMM Environment

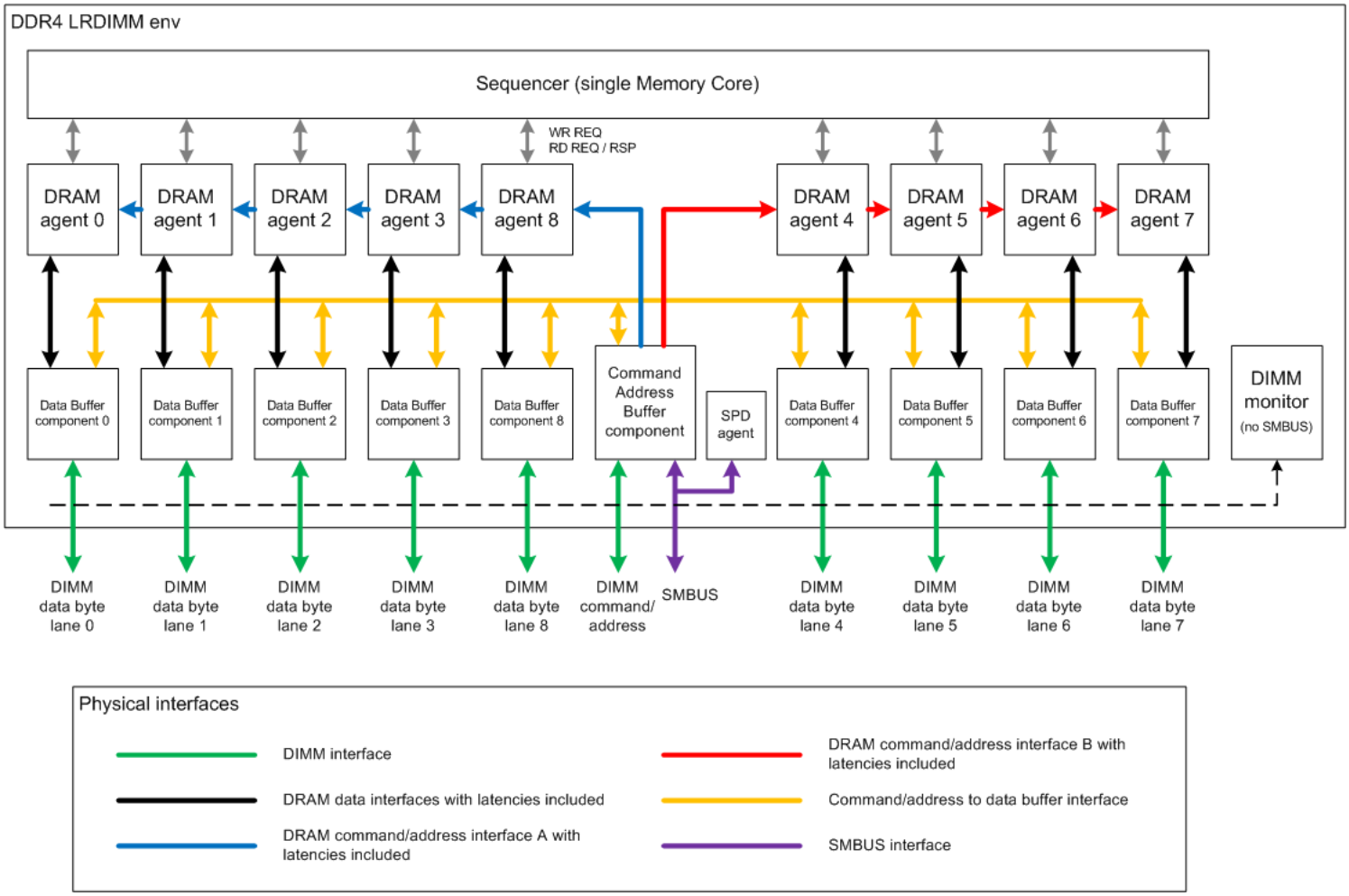


Figure 6-2 DDR4 RDIMM environment

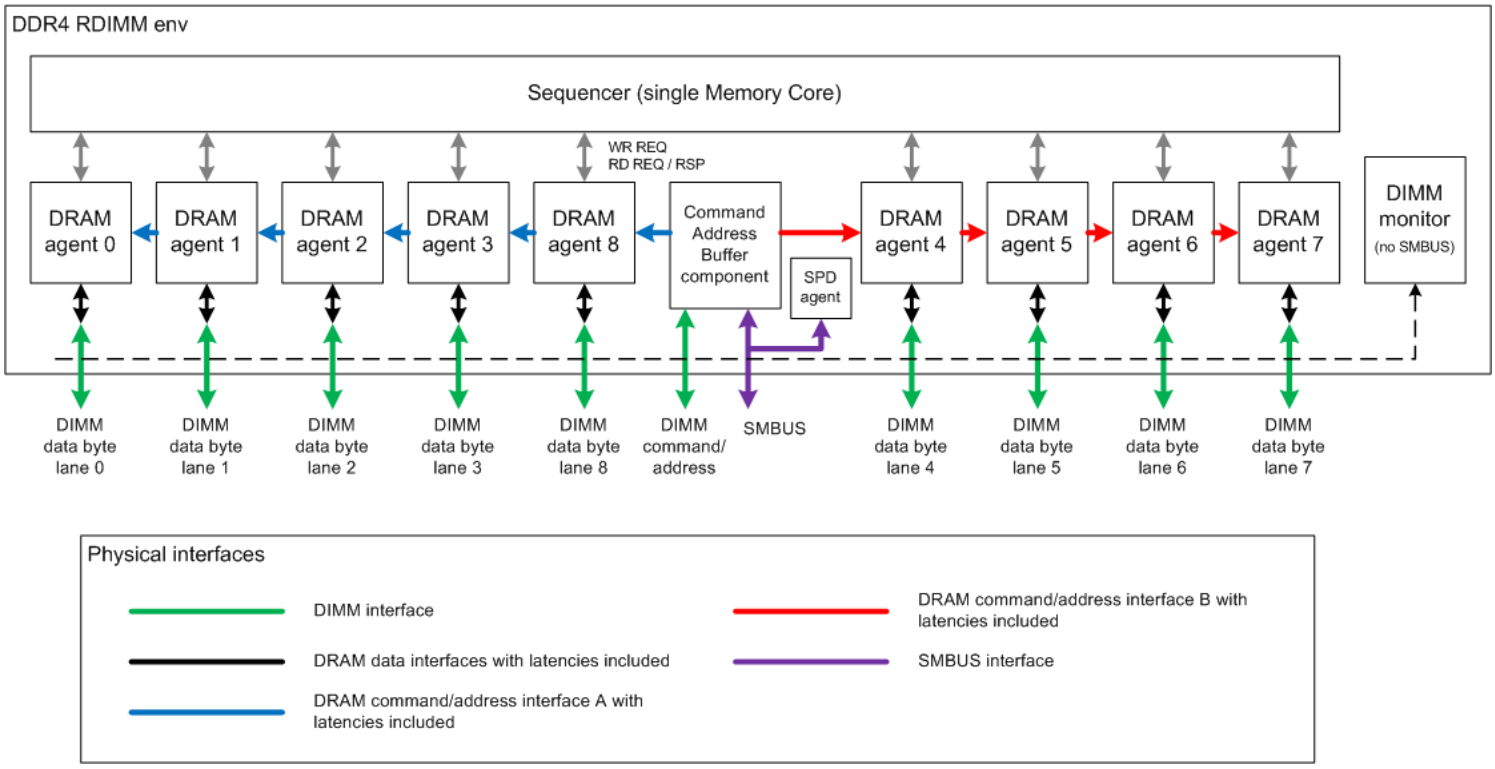


Figure 6-3 DDR4 UDIMM Environment

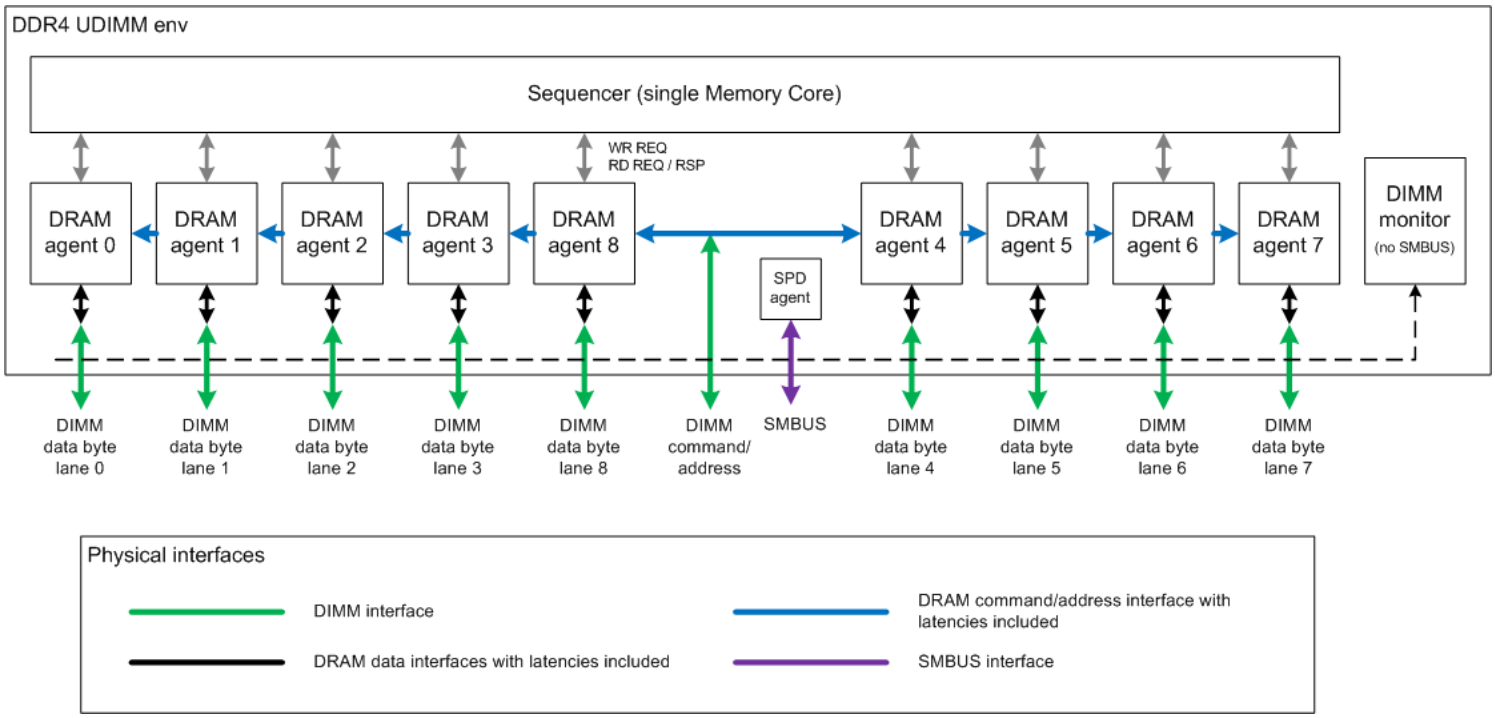


Figure 6-4 DDR3 LRDIMM Environment

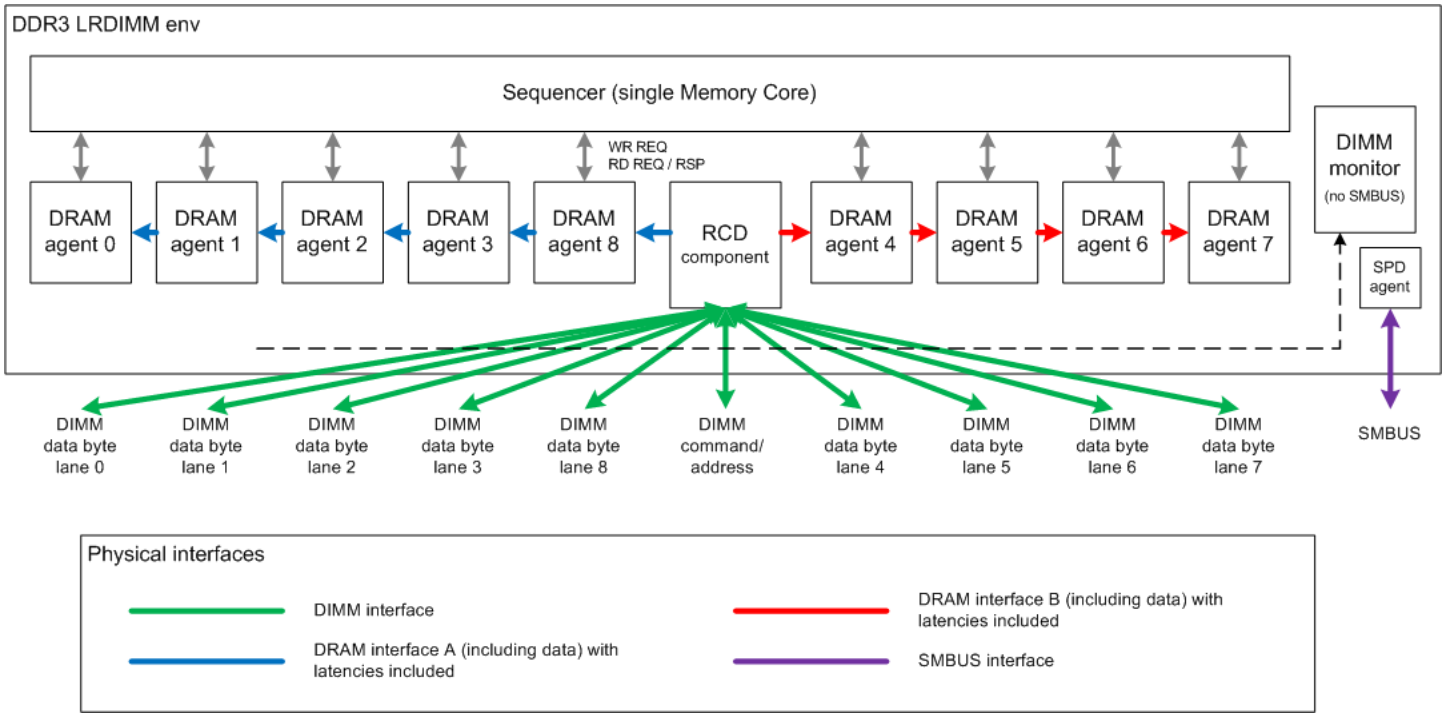


Figure 6-5 DDR3 RDIMM Environment

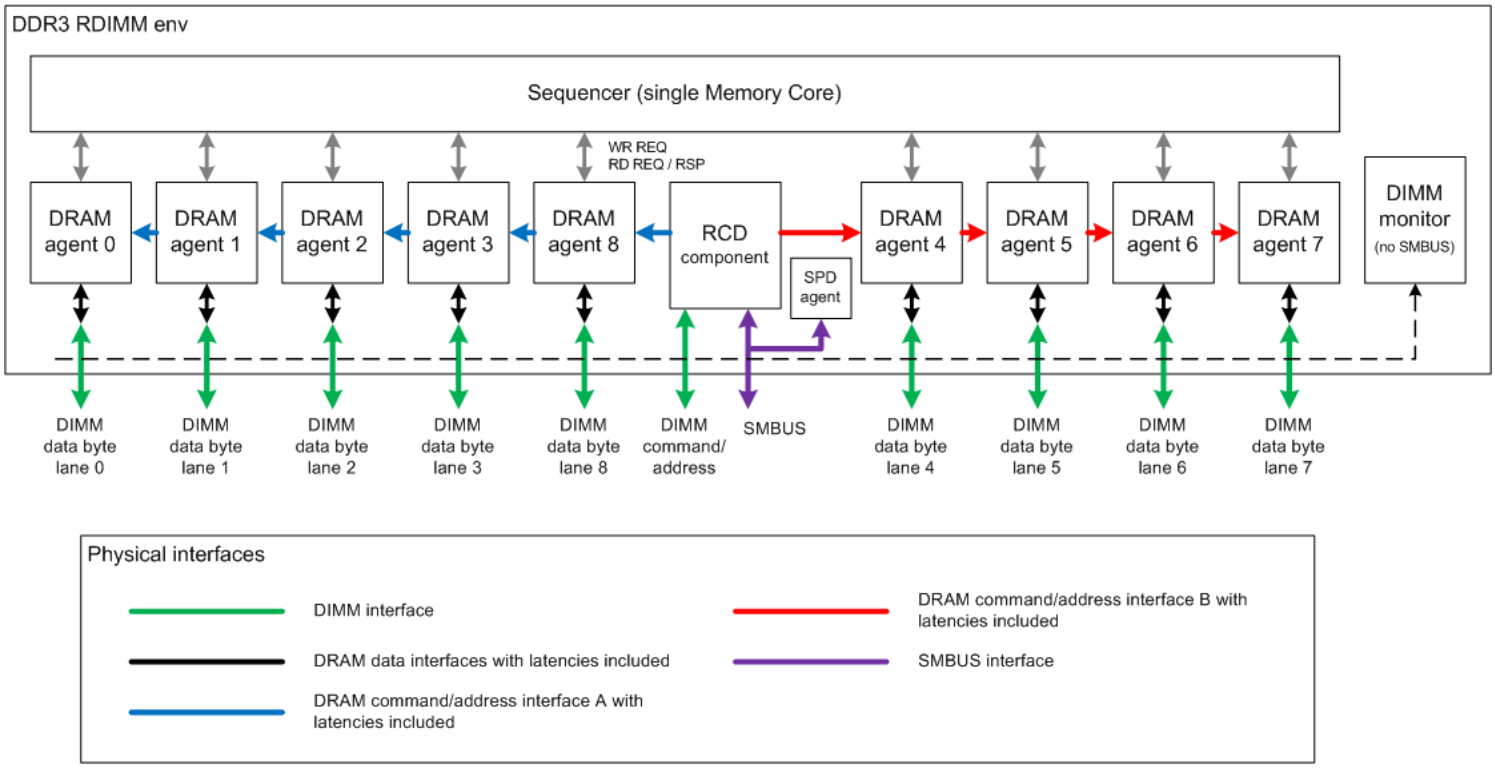


Figure 6-6 DDR3 UDIMM Environment

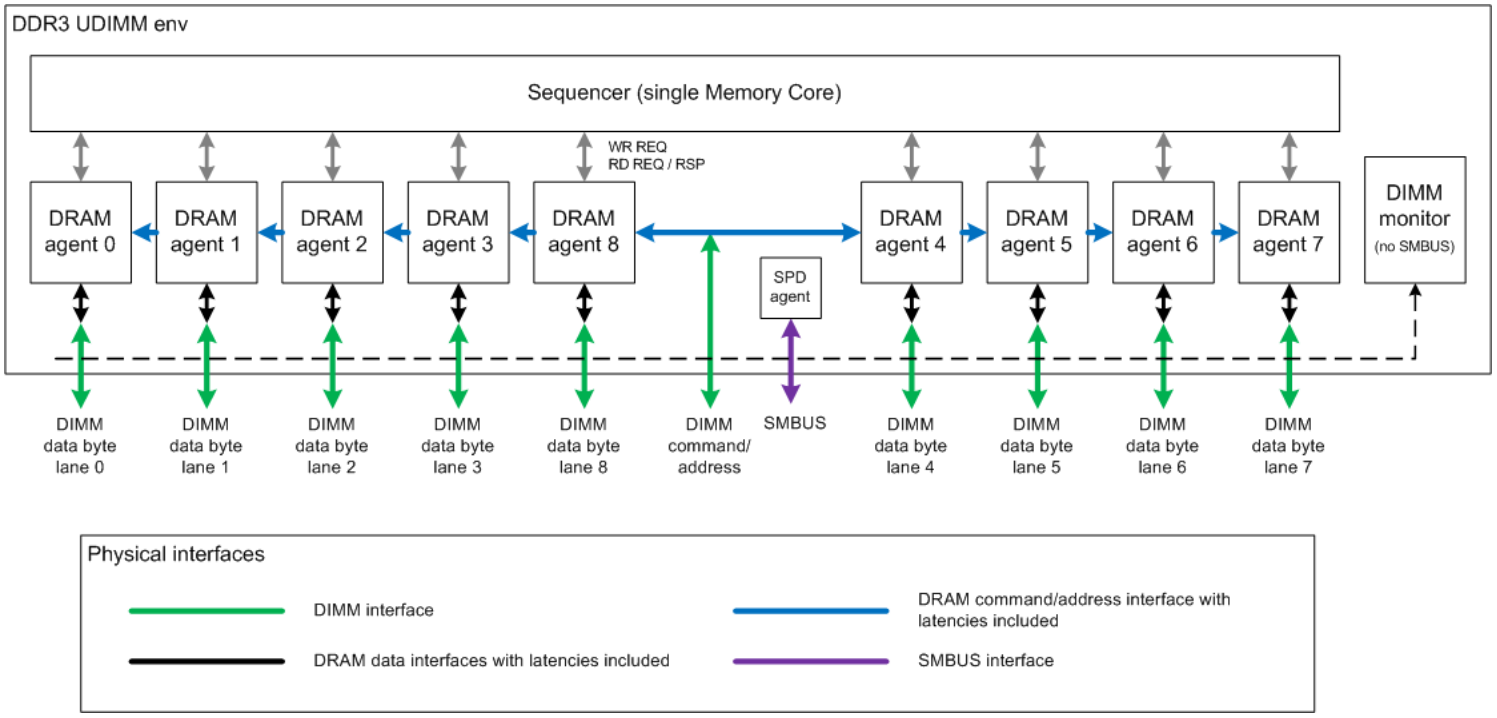


Figure 6-7 DDR5 RDIMM Environment

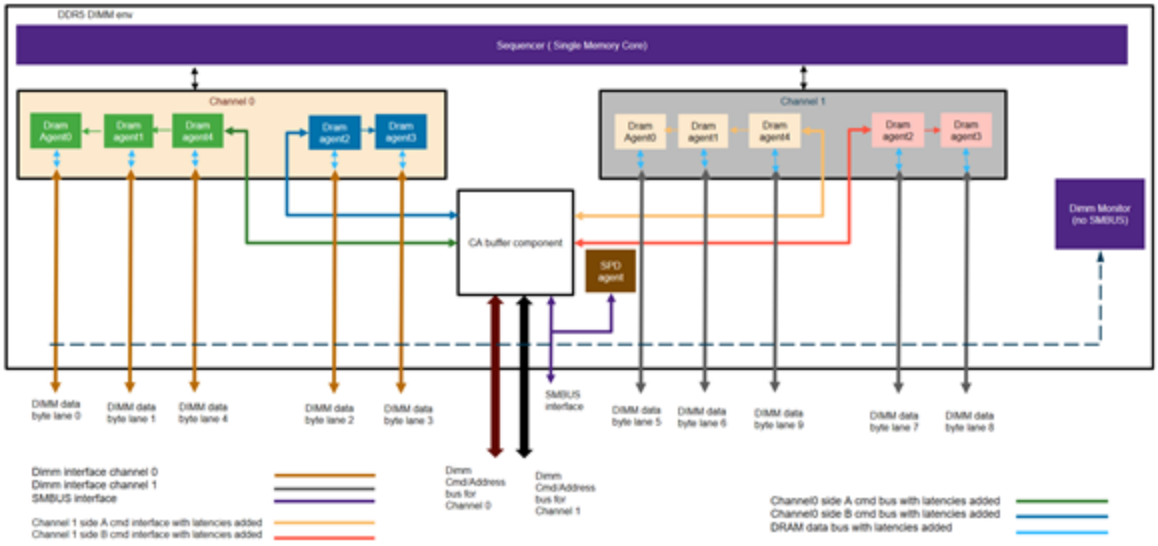
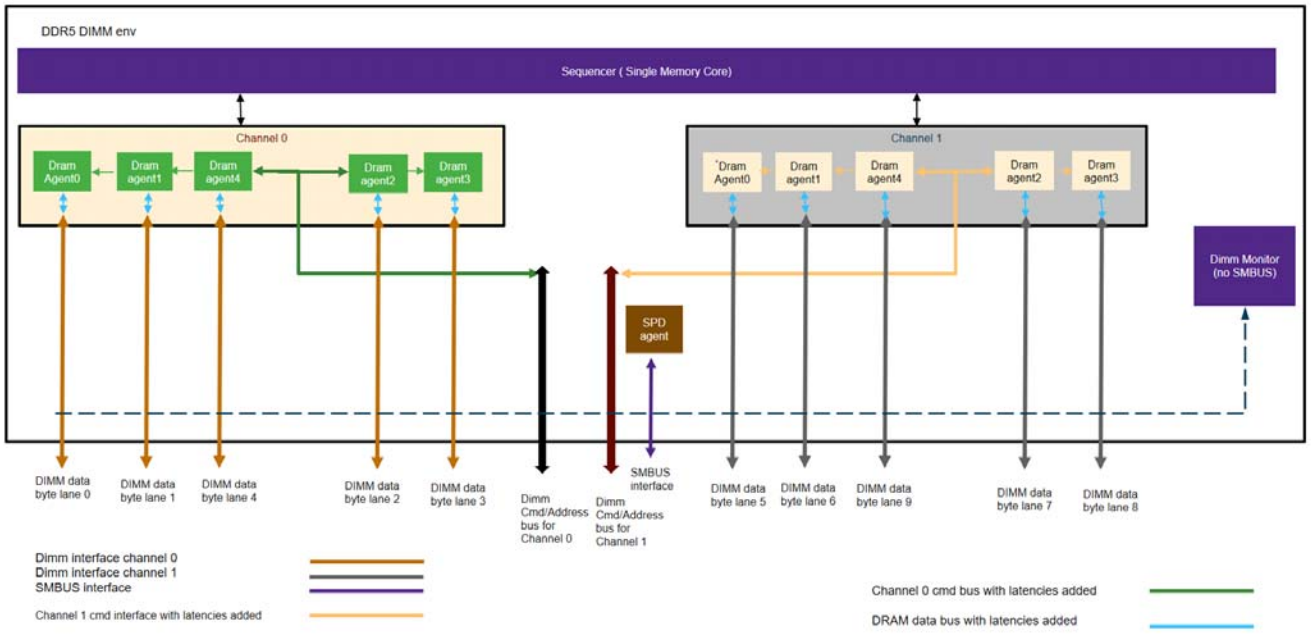


Figure 6-8 DDR5 UDIMM Environment



6.1.1 RCD Component

The RCD Component does the following:

- Receives commands and addresses from the controller
- Buffers commands and addresses
- Takes additional actions if needed (e.g. checks the parity)
- Passes information to all the DRAMs.

In DDR3 RDIMM, the RCD works the same way as CA Buffer; that is, it buffers only command and address lines whereas RCD also buffers data lines in DDR3 LRDIMM. Checks are implemented as per the CA Buffer (RCD) specification.

DRAM agents will continue to do transaction collecting and checking while CA Buffer (RCD) component perform its buffering functionality without collecting DDR transaction. All CA Buffer (RCD) outputs are determined by the received command and address and by the internal registers' settings. The CA Buffer (RCD) component has no sequencer or driver.

CA Buffer (RCD) has a common object, monitor, configuration and status objects as shown in figure below. Similar as with DRAM agent, the Configuration object holds initial CA Buffer (RCD) registers' settings, while the Status object will contain runtime CA Buffer (RCD) registers' values.

To Configure DDR4/DDR5 RDIMM

You can use DDR4/DDR5 RCD register space and it can be configured as defined as follows:

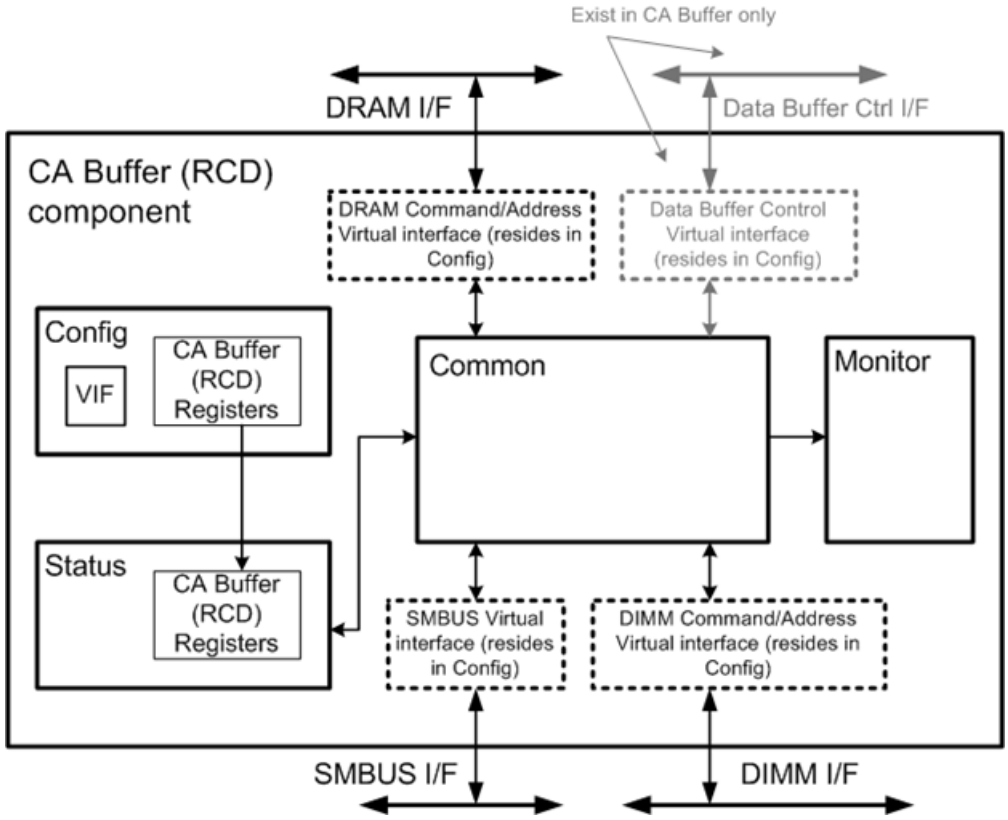
```
cfg.ca_buffer_cfg.timing_cfg.tSTAOFF_ps  = 1625;  (tPDM_ps(1000)+tCK_ps(625))
cfg.ca_buffer_cfg.timing_cfg.tPDM_ps = 1000;
```

Note
In DDR4, timing parameter tSTAOFF=tPDM+1/2*tCK_ps, but in DDR5 tSTAOFF would be tSTAOFF= tPDM+tCK_ps. tSTAOFF_ps is configurable parameter, you can configure it to different time value (as defined above).

RCD Functionality

RCD will sample the input from the host and drive it DRAMs in A and B sides after the configured delay.

Figure 6-9 RCD Component



6.2 DIMM Delay Modeling

All delays in the DIMM will be separately modeled for several signal groups

- CMD/ADDR (RAS, CAS, WE, S, BA, A, PAR),
- CTRL (CKE, ODT),
- CLK and
- DATA (DQ, DQS, CB).

For CMD/ADDR, CTRL and CLK group's delays to be modeled are

- Fly-by delay (for LRDIMM/RDIMM/UDIMM[SODIMM])
- Pre-buffer delay (in case of LRDIMM/RDIMM)
- Post-buffer delay (in case of LRDIMM/RDIMM).

The default usage model for controlling fly-by delays will be to set worst case values, which then will be used for all the connections. You have the option to set each fly-by delay separately.

DATA group delays are modeled in the following way:

- data lane delay (in case of RDIMM/UDIMM[SODIMM]),
- pre-buffer delay (in case of LRDIMM),
- post-buffer delay (in case of LRDIMM).

6.3 DIMM Monitor

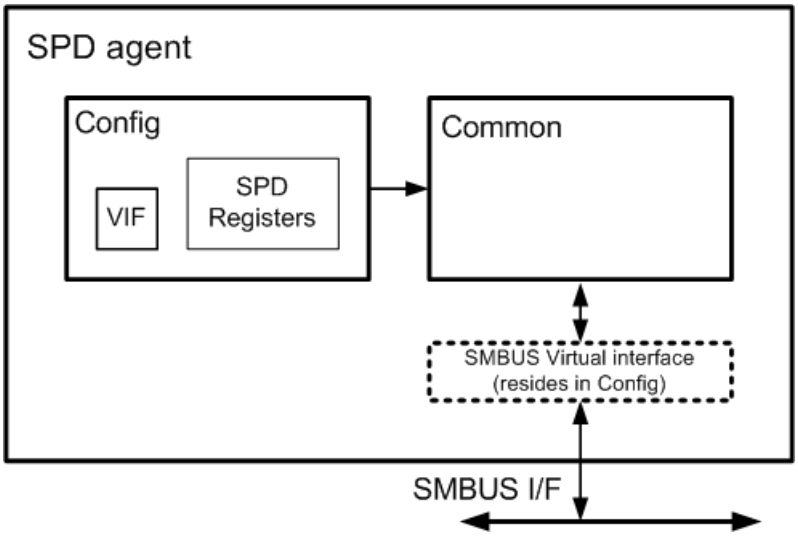
DIMM monitor's purpose is to collect DIMM transactions from the interface, and make them available to the user. It monitors DIMM command/address lines and all DIMM data lanes. It does not contain any checkers implemented in other agents/components.

The Command Address Buffer (RCD) component, Data Buffer component, and SPD agents will be connected through physical interfaces as shown on the figures.

6.4 SPD Buffer Agent

The SPD agent relies on SMBus read requests addressed to it. It has a configuration object which holds the registers file and common object which implements the functionality. It has no checkers.

Figure 6-10 SPD Buffer Agent



7 DDR4 3DS Stack Support

7.1 Introduction to Discrete DDR4 Device with a 3DS Stack

The VC DDR VIP provides a class called "svt_ddr_3ds_env" for modeling a Discrete Device with 3DS stack. For users with existing non 3DS Discrete DDR4 Device setups, you need to replace instances of "svt_ddr4_agent" with "svt_ddr_3ds_env" class.

7.1.1 Configuration Settings

Use the following configuration parameter for 3DS Stack support:

```
cfg.protocol_kind = DDR4;  //cfg is handle of svt_ddr_configuration class
cfg.set_memory_type(svt_ddr_configuration::DDR_3DS);
cfg.num_logical_ranks = 8; // It can have values of 2, 4 and 8
```

Note the Transaction Class Properties for Logical Rank within the 3DS Stack is used specify the logical rank within the 3DS. Use the chip_id property in svt_ddr_transaction class for this.

You can find configuration examples in the following example:

ddr_svt/examples/sverilog/tb_ddr4_3ds_svt_uvm_basic_sys

7.1.2 Catalog Files for Discrete DDR4 with 3DS Stack Conventions

The naming convention of the 3DS catalog files have additional information about 3DS and about the number of logical ranks. Example:

edec_ddr4_3ds_4G_x4_1600J_1_25_8H.

There is no change in the use model for loading the 3ds catalog files.

Following is the path to 3DS catalog files:

$DESIGNWARE_HOME/vip/svt/ddr_svt/latest/catalog/3ds/dram

7.1.3 Usage

7.1.3.1 Randomization Approach for Configuring 3DS DDR4 Discrete Device Catalog Parts

The following shows how to use randomization to configure a device:

```
/** Randomize dram configuration and set data width, number of logical ranks
*/ and other fields
if (! dram_cfg.randomize() with{// dram_cfg is handle of svt_ddr_configuration
                                data_width == 8; // it can have values of 4, 8 and 16
                                num_logical_rank == 8;  //Canan have values of 2, 4
                                                        //and 8 for modeling number of
                                                        //logical ranks
                                    …….
                                   })
   begin
      `uvm_fatal("randomize_cfg","DRAM configuration object randomization failed");
   end
cfg.memory_type=svt_ddr_configuration::DDR_3DS; // Set memory type to be DDR_3DS
```

7.2 Introduction to DDR4 DIMMs (UDIMM/RDIMM/LRDIMM) with 3DS Stack

7.2.1 Configuration Settings

You use the svt_ddr_dimm_env and svt_ddr_dimm_configuration classes for modeling the DDR4 DIMMs (UDIMM/RDIMM/LRDIMM) with a 3DS Stack.

To specify the logical rank within the 3DS stack use chip_id property in svt_ddr_transaction class

7.2.2 Catalog File Conventions for DDR4 DIMMs (UDIMM/RDIMM/LRDIMM) with 3DS Stack

Naming convention of 3DS catalog files have additional information on the 3DS and on the number of logical ranks. Example:

jedec_ddr4_*3ds*_lrdimm_4G_x4_1600J_1_25_2R_8H_ecc.cfg

There is no change in the use model for loading the 3ds catalog files.

The path to the catalog files is:

$DESIGNWARE_HOME/vip/svt/ddr_svt/latest/catalog/3ds/dimm

7.2.3 Usage

The following is an example for DDR4 DIMMs(UDIMM/RDIMM/LRDIMM) with 3DS stack:

ddr_svt/tb_ddr4_3ds_dimm_svt_uvm_basic_sys

7.2.3.1 Randomization Based Approach for Modeling 3DS DDR4 DIMM Catalog Parts

```
/** Randomize dram configuration and set data widt, number of logical ranks and other
fields */
if (! dram_cfg.randomize() with { // dram_cfg is handle of svt_ddr_configuration
         data_width == 8; // it can have values of 4, 8 and 16
   num_logical_rank == 8; // it can have values of 2, 4 and 8
                                    …….
                                    })
begin
```

```
`uvm_fatal("randomize_cfg","DRAM configuration object randomization failed");
end

/** Set the memory type of DIMM cfg as svt_ddr_configuration::DDR_3DS */
dram_cfg.memory_type=svt_ddr_configuration::DDR_3DS;

/** Set the DIMM type */
cfg.set_protocol_kind(svt_ddr_dimm_configuration::DDR4_UDIMM); // it can be DDR4_UDIMM,
                                                               //DDR4_RDIMM and
                                                               //DDR4_LRDIMM

/** Randomize DIMM configuration */
if (! cfg.randomize() with {has_ecc == 0; // it can have values of 0 and 1
                            values of
                            num_ranks == 1; // it can have values of 1, 2 and 4
                            num_logical_rank  == dram_cfg.num_logical_rank;
                                 // 3DS logical rank information at DIMM level
                           memory_type       == dram_cfg.memory_type;

memory_speed_bin  == dram_cfg.memory_speed_bin;
                                 mem_size          == dram_cfg.mem_size;
                                 bank_addr_width   == dram_cfg.bank_addr_width;
                                 column_addr_width == dram_cfg.column_addr_width;
                                 row_addr_width    == dram_cfg.row_addr_width;
                                 })
begin
        `uvm_fatal("randomize_cfg","DIMM configuration object randomization failed");
      end

/** Set dram cfg to dimm configuration */
cfg.set_dram_cfg(dram_cfg);
```

8 DDR4 NVDIMM-P

8.1 Introduction

This chapter provides Non-volatile DIMM support present in Synopsys VC DDR VIP. The Synopsys VC DDR VIP now supports Non-Volatile DIMM-P on top of LRDIMM configuration only.

8.2 Setting The DDR4 NVDIMM-P

VC DDR VIP supports DDR4 NVDIMM-P configuration which is implemented on top of LRDIMM based on the NVDIMM-P specification. This configuration can be enabled in VIP by implementing the following changes in the VIP.

Provide the `protocol_kind` parameter in `svt_ddr_dimm_configuration` class as `DDR4_NVDIMMP`.

For example,

```
dimm_cfg.set_protocol_kind(svt_ddr_dimm_configuration::DDR4_NVDIMMP);
```

8.3 Usage

1. Approach 1:

   Randomization approach for configuring DDR4 NVDIMM-P DIMM model configuration. For example, `dimm_cfg` is the handle of `svt_ddr_dimm_configuration` class.

   ```
   dimm_cfg.set_protocol_kind(svt_ddr_dimm_configuration::DDR4_NVDIMMP);
   if( !dimm_cfg.randomize() with {
       data_width == 72;
       memory_speed_bin == svt_ddr_configuration::DDR4_2400U;
       addr_width == 40;
       }) begin
         `svt_fatal("generate_cfg","DIMM configuration object randomization
   failed");
   end
   ```

2. Approach 2: Catalog File Conventions for DDR4 NVDIMM-P

   These are the catalogs present for DDR4 NVDIMM-P which can be used to load the configuration. For example:

   jedec_ddr4_nvdimmp_4TB_x72_2400R_0_833_1R.cfg

   jedec_ddr4_nvdimmp_512GB_x72_2400R_0_833_1R.cfg

   jedec_ddr4_nvdimmp_8TB_x72_2400R_0_833_1R.cfg

There is no change in the use model for loading the DDR4 NVDIMM-P catalog files.

The path to the catalog files is:

```
$DESIGNWARE_HOME/vip/svt/ddr_svt/latest/catalog/ddr4/dimm/JEDEC/
```

8.4 Features

The DDR4 NVDIMM-P features are as follows:

- The VIP will be compliant to the Samsung ballot (`TG456_2^20181029^2233.98^Samsung^DDR4_NVDIMM-P_spec_r869`).
- Supports out-of-order transactions, with Burst access (BL=8).
- Scalable architecture supports proposed protocol features at JEDEC TG
- Supports all commands - MRS,XWRITE,PWRITE,XREAD,XADDR,SEND,MRACT_WGID,SEND-MRR,SREAD,UNMAP,IOP,NOP,DESELECT,PDE,PDX,ZQCL,ZQCS
- All responses are supported such as RD_RDY, W_PER, URGENT and INTERRUPT response from VIP.
- Shared Available Write Buffer (AWB) and PWRITE Credit Limit (PWC) support is available.
- Customized user-defined Write Credit support is available. CTH mode is also available.
- All modes of FLUSH command are supported.
- All ordering rules are followed in VIP.
- All modes of READ-STATUS command is available.
- IOP mode 0 and mode 1 is available.
- Power-down feature is supported for both modes namely Non-clock stopped and Clock-stopped power-down mode
- Complete transactional information is supported for RIDs/WGIDs, Poison, User-data bits, WC, CTH, D_VALID.
- Message packet format (Supports RDY_PWR, Valid AWC, Valid PWC, IOP and Interrupt)
- Mode Registers are supported.
- MCW register support is also available.
- Other VIP features such as the following:
  - Native SV/UVM based Protocol checks (Checks Pass/Fail Coverage)
  - Flexibility to demote checks
  - Analysis ports for score boarding
  - Callback support
  - Bypass and fast-memory initialization support
  - Flexibility to override timing values
  - Backdoor memory and register access (initialize, peek, poke, load, dump and so on) and
  - Dynamic reconfiguration support.

- Power-up initialization is supported. Initialization may be skipped by setting the flag `skip_init=1`. Controller has the liberty to issue the XREAD/XWRITE command immediately after `RESET_n de-assertion`.
- Following is the NVDIMM-P interface pin-out symbols with their type. It will be like DDR4 JEDEC DIMM Interface, with the changes in CKE and ODT pins. CKE1, ODT1 will be made output from NVDIMM-P module.

Table 8-1 PIN-OUT Symbols

Symbol	Type
CK_t, CK_c	Input
CKE0-CKE1	CKE0: Input CKE1: Output
CS_n[1:0]	Input
C0, C1, C2	Input
ODT0-ODT1	ODT0: Input ODT1: Output
ACT_n	Input
DM0-DM8	Input/Output
BG0-BG1	Input
BA0-BA1	Input
A0-A17	Input
RESET_n	Input
DQ0 – DQ63	Input/Output
CB0-CB7	Input/Output
DQS0_t - DQS17_t, DQS0_c – DQS17_c	Input/Output
PAR	Input
ALERT_n	Input/output
TEN	Input

Note
DM pins are not used for NVDIMM-P.

- An explicit FLUSH command can be issued optionally in case of all write credits have been consumed before issuing any further WRITE command. The other way to empty out the buffer contents is by enabling auto-flush feature of VIP. This can be done by setting `enable_nvdimmp_auto_flush` to 1.

Note
Logical address is being calculated as {addr[39:33], addr[32:23], addr[22:12], addr[11:2], 2'b00}.

Note
The last two bits are always 0.

Note
VIP is performing writing and reading operation in sequential manner of address with A [1:0] as zero always.

Note
There is no wrapping functionality for addressing and VIP expects byte addressability for efficient buffer usage.

Note

- Data returned for the READ command is received from the latest element matching for same address in the buffer. If multiple PWRs exist in the buffer for the same address and a new PWR is issued to same address with persist=1 then, this new element is deleted from the buffer.
- For SREAD command, the data would be searched immediately in the buffer after the command is received. After the expiry of `tSEND_RD/tRL` timer, the data along with valid bit information would be shared if it was a hit case else, 0 would be returned on DQ [63:0] with non-valid bit information.
- If XREAD/SREAD command is delivered before completion of previous an XWRITE/PWRITE command on the same address and if data is available in buffer, data is given from the buffer else previous data/0 is provided.
- If SREAD along with XADR command comes with an address A [39:2] then the complete address should match with exact A [39:2] address of any prior WR command to mark this SREAD as a hit.
- ECC support is present along with 'salt' feature.
- DQ training using both the modes : advance fifo mode and mpr pattern read mode is present in the VIP.

8.5 Examples

You can find NVDIMM-P configuration examples using UVM from the following location:

```
$DESIGNWARE_HOME/vip/svt/ddr_svt/examples/sverilog/tb_ddr4_nvdimmp_dimm_svt_uvm_basic_sys
```

NVDIMM-P configuration examples can be installed using the following command:

```
$VC_VIP_HOME/bin/dw_vip_setup -e ddr_svt/tb_ddr4_nvdimmp_dimm_svt_uvm_basic_sys -svtb
```

8.6 Current Limitations

The curent limitations for DDR4 NVDIMM-P are as follows:

- This VIP version supports UVM only.
- One `do_is_valid` added for `data_width`. The `data_width` should be 72 for NVDIMM-P VIP.
- VIP does not support Protocol Analyzer.
- DIMM monitor not available.
- Mode registers have the following limitations:

- MR0: DLL Reset and TM are not supported.
- MR5: CA Parity at dram is not supported. It is supported at RCD level.
- MR7: Extended MR Address Space is not supported

- I2C interface support is not available
- SPD register support is not available
- MMR register is not supported
- DB Training for MRD and MWD is not available.
- Jitter Support is not available.
- Clock-change support is not available.
- Reset Initialization with Stable Power is not supported.

8.7 VIP Customizations to Model Real-life Scenarios

8.7.1 NVDIMMP_RANDOM_WC_3BIT_OPTION

VIP will support random write credits return facility to the host for option A as per Table 9 in section 2.3.4 of specification: TG456_2^20181029^2233.98^Samsung^DDR4_NVDIMM-P_spec_r869.

8.7.1.1 Feature

VIP will provide random write credits back to the host that is equal to or less than the maximum credit limit set by the user if actual number of credits to be returned are more than the maximum credit limit set by the user. In case, the actual credits to be returned are less than the maximum credit limit set by the user then VIP will return credits that is equal to or less than the actual credits to be returned.

8.7.1.2 Requirement

1. Host should enable this flag: `enable_nvdimmp_rand_3bit_wc_option` in dimm configuration. This is onetime configurable at build phase.
2. Host should set the maximum value of write credit to be return using the function in dimm configuration like: `<dimm_cfg_handle>.set_nvdimm_max_wc_return_limit (16)`. The flag `nvdimm_max_wc_return_limit` can have values among (1,2,4,8,16,32,64 only). If this is not set by the user, VIP will randomize it and will pick one value for it.
3. Host should set write credit update option in MRS2 as 3bit option.

8.7.1.3 Illustration

- In case, if the host has set flag `nvdimm_max_wc_return_limit` to 8, then thehost has issued 50 XWRs to the VIP and FLUSH command is issued. When Host issues READ STATUS command, then VIP can return any value from (0,1,2,4,8) in response to the read status command and so on.
- In case, if the host has set flag `nvdimm_max_wc_return_limit` to 32 then the host has issued 30 PWRs to VIP without persist bit being set with same WGID and the host issues another PWR with the same WGID and persist bit being set.. In total, these 31 PWRs should be processed by VIP and data should be moved to NVM. Now, if host issues READ STATUS command, then VIP would return any credit from (0,1,2,4,8,16) as 32 is more than the actual credits to be returned for the 1st READ STATUS command. Similarly, in lieu of 2nd READ STATUS command issued by host, VIP would return any random value from 0 to remaining credits to be returned.

User can configure `nvdimm_max_wc_return_limit` variable at run-time using reconfigure, so that the value can be changed dynamically.

Note

Though credits would be returned based on the above configuration is selected, but internally VIP would process all writes into NVM once it is to provide W_PER response back to the host which marks the completion of FLUSH/PWR command (if persist bit was set to 1).

9 MRAM

9.1 MRAM Configuration

VIP supports MRAM DDR3/DDR4 Discrete Devices and MRAM DDR3/DDR4 UDIMMs.

Except for configuring the memory_type as DDR_MRAM, the MRAM use model is the same as for DDR3/DDR4 discrete devices and DDR3/DDR4 UDIMM devices. Following shows how to configure the DDR model for MRAM Discrete Device behavior:

```
cfg.memory_type = svt_ddr_configuration::DDR_MRAM;  // cfg is handle of
                                                    // svt_ddr_configuration
```

cfg.protocol_kind = svt_ddr_configuration::DDR3/svt_ddr_configuration::DDR4;

Following shows how to configure for MRAM UDIMM:

```
dimm_cfg. memory_type = svt_ddr_configuration::DDR_MRAM;  // dim_cfg is handle of
                                                        // svt_ddr_dimm_configuration
dimm_cfg.protocol_kind  =
svt_ddr_dimm_configuration::DDR3_UDIMM/svt_ddr_dimm_configuration::DDR4_UDIMM;
```

For the MRAM catalog, please contact the Synopsys support team.

10 Monitor and Coverage

10.1 Introduction

This section provides information on the DDR monitor which is instantiated in svt_ddr_agent and svt_ddr_dimm_env. The monitor class svt_ddr_monitor in turn instantiates the svt_ddr_checker class that provides the built-in protocol checking capability.

10.2 Protocol Checkers

The monitor instantiates the protocol checker. To enable checking you must explicitly turn on checking using the svt_ddr_configuration member “enable_checks”.

10.3 Analysis Port

The monitor in the svt_ddr_agent and svt_ddr_dimm_env 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. The user can use the analysis port for connecting to a scoreboard, or any other purpose where a transaction object for the completed transaction is required.

10.4 Coverage

The DDR 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.

10.4.1 Toggle Coverage

DDR 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. In certain cases, not all bits can toggle. These bits must be filtered from toggle coverage.

10.4.2 State Coverage

DDR 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, for the CA[0:9] signal, there could be 256 states. A possible choice of the coverage bins could be the following:

- One bin to cover the lower address range
- One bin to cover the upper address range
- One bin to cover all other intermediary addresses.

10.4.3 Meta Coverage

Meta coverage is collecting second-order coverage data. Possible meta coverage measurements include valid to ready delays and inter-phase delays. Meta coverage information is particularly useful to flag excessive latencies (possibly indicating deadlocks).

10.4.4 Transaction Cross Coverage

Cross coverage specifies interesting cross coverage across DDR signals and measures the activity across those signals.

10.5 Coverage Callback Classes

A 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. This implementation does not include any coverage groups. The def_cov_data callbacks classes are extended from an agent callback class.

10.5.1 DDR Coverage Callback Classes

The following table lists DDR coverage callback classes.

Table 10-1 DDR Coverage Callback Classes

DDR Coverage Callback Class	Description
svt_ddr_def_cov_callback class	Meta group coverage class that contains covergroups for FSM and transaction data.
svt_ddr_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_ddr_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_ddr_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_ddr_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_ddr_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.

10.5.2 Enabling Default Coverage

The default functional coverage must be enabled by setting the following attributes in the configuration classes svt_ddr_configuration or svt_ddr_dimm_configuration to '1'. To disable coverage, set the attributes to '0', which is the default. The attributes are:

- toggle_coverage_enable
- state_coverage_enable
- transaction_coverage_enable

11 Catalog and Part Numbers

11.1 Overview

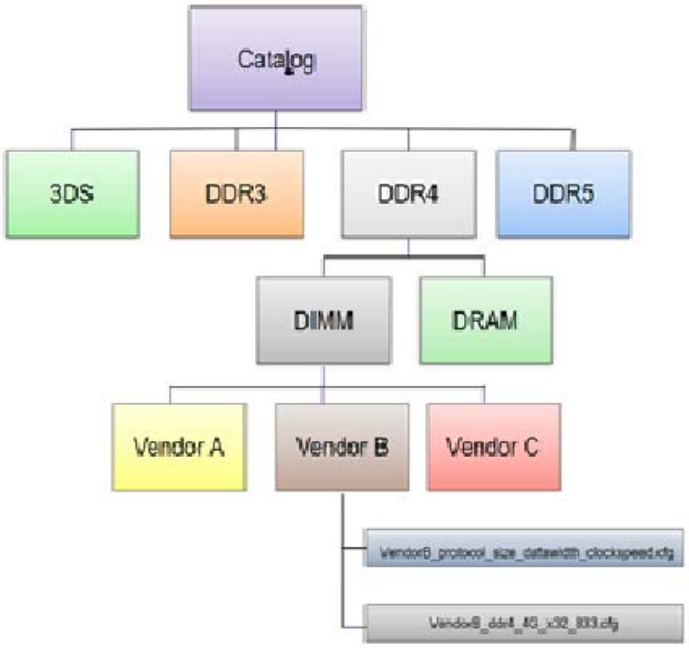
The Synopsys DDR VIP provides you with catalogs of vendor memory parts which you can use to configure your VIP and manage your testbench. The following sections describe catalog and part numbers and how to use them.

11.2 Location of Vendor Catalogs After Installation

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

$DESIGNWARE_HOME/vip/svt/ddr_svt/latest/catalog

The catalog directory has a typical structure shown in the following illustration. It will change after EA to handle various types of vendor and custom parts created by users.



11.3 Contents of a Part Number *.cfg File

The part *.cfg files are ASCII files which sets values for the behavior of any given model.

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

```
//
// Description: 4Gb, 1.2V
// Density    : 256Mbx16
// Speed      : 1066MHz
//
data_width=16
addr_width=25
attr_width=1
bank_addr_width=2
row_addr_width=15
column_addr_width=10
chip_select_addr_width=1
data_mask_width=2
data_strobe_width=2
cmd_addr_width=18
prefetch_length=8
num_data_bursts=8
vif_type=JEDEC_CHIP
tCCD_S=5
tCCD=4
tCAL=12
tRCD_ns=12.500000
tCK_ns=1.250000
tRC_ns=50.000000
...
VREF_TIME_LONG_ns=150.000000
tDLLK=512
protocol_kind=DDR4
mem_size=SIZE_4GB
memory_speed_bin=DDR4_2133P
```

11.4 Hierarchical Structure of Vendor Memory Parts

The parts catalog is structured in the following hierarchical way as shown in Table 11-1.

Table 11-1 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, DDR will ship with DDR2, DDR3, and DDR4 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.

11.5 Catalog Classes and Features

The DDR 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 the contains a static "#get()" method returning the full path to the installation directory of the suite.
- **svt_mem_part_mgr**. Class you use to choose vendor parts within all available catalogs.

11.5.1 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 (DDR2, DDR3, or DDR4). 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 (M100H01, jedec_ddr3_4G_x16_1600J_1_250, etc) using **get_part_number()**
   - Memory depth (SVT_MEM_64Mb, SVT_MEM_4Gb, etc) using **get_depth()**
   - Memory width (SVT_MEM_x16, SVT_MEM_x32, etc) using **get_width()**
   - clock rate (SVT_MEM_600MHz, SVT_MEM_1500MHz, etc) 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 DDR UVM Online Help located at:

$DESIGNWARE_HOME/vip/svt/ddr_svt/doc/ddr_svt_uvm_class_reference/html/index.html

12 Using Protocol Analyzer with Memory Models

12.1 Overview

Synopsys provides Protocol Analyzer (PA) to help you debug designs with DDR 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 DDR VIP.

12.2 Enabling the DDR VIP to Generate Data for Protocol Analyzer

To use Protocol Analyzer to debug memory transactions, you must enable the DDR 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_ddr_configuration class.
   svt_ddr_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;
```

12.3 Using Native Protocol Analyzer for Debugging

12.3.1 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.

12.3.2 Prerequisites

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

12.3.3 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
```

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

12.3.4 Invoking Protocol Analyzer

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

12.3.4.1 Post-processing Mode

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

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

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

12.3.4.2 Interactive Mode

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.

12.3.5 Limitations

Interactive support is available only for VCS.

12.4 Performance Analyzer

12.4.1 Overview

The performance analyzer tool is used to calculate the performance of sub-systems. Performance Analyzer uses the transaction data recorded inside the FSDB which is generated for Protocol Analyzer debug.

12.4.2 Enabling DDR VIP to generate data for Performance analyzer

12.4.2.1 Configuration setting to enable data for performance analyzer:

12.4.2.1.1 Compile options

```
+define+SVT_FSDB_ENABLE  //enable FSDB dumping
-lca
-debug_access
```

12.4.2.1.2 VIP configuration options

In the UVM environment class set the following to generate data for Performance Analyzer

```
//Members to generate data defined in top level svt_ddr_configuration.sv class
   svt_ddr_configuration cfg

   cfg.enable_memcore_xml_gen = 1;
   cfg.enable_xact_xml_gen = 1;
   cfg.enable_fsm_xml_gen = 1;
   cfg.enable_cfg_xml_gen = 1;
svt_ddr_configuration.pa_format_type=svt_xml_writer::FSDB;
```

Or

Run-time Switch

```
svt_enable_pa=FSDB
```

Note
The performance analyzer is part of Verdi and will be available for Verdi users only.

12.4.2.2 Invoking Verdi GUI after running the simulation:

Invoking Verdi GUI after running the simulation:

```
verdi -lca -ssf test_top.fsdb
```

Note
For more information, see Verdi_Performance_Analyzer.pdf.

12.5 Metrics Description

This section contains a list of performance metrics which are present for DDR VIP. Here's the initial list of metrics currently supported by the PA tool for the DDR VIP

- ddr_count_all_cmd.tcl - This metric counts the number of ACT,PRE, SRE, RD, WR , PDE, PDX , DES, NOP commands found in the test run.
- ddr_trans_read_write_latency.tcl - This metric calculates the latency between RD and WR commands
- ddr_trans_read_write_cyclic_latency.tcl - This metric calculates the latency between 2 different RD/WR for the same logical address
- ddr_trans_count_bank_cmd.tcl - This metric counts the number of commands for each bank address
- ddr_trans_act_same_row_addr.tcl - This metric counts the number of commands for the same row

- ddr_trans_act_followed_precharge_cmd_count.tcl - This metric counts the number of times ACT was followed by pre-charge (row getting closed before making any RD/WR accesses

13 Using the DDR Verification IP

13.1 Introduction

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

13.2 Setting The DDR 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 protocol_kind member in the top-level configuration class svt_ddr_configuration sets the protocol level.

```
/ ------------------------------------------------------------------------------
  virtual function void create_cfg();

    // Load configuration from ddr_cfg_file
    svt_ddr3_vendor_part p =
      svt_mem_part_mgr#(svt_ddr3_vendor_part,svt_ddr3_part_number_policy)::pick();

    cfg = svt_ddr_configuration::type_id::create("cfg", this);

   // Set the protocol to DDR3.
    cfg.protocol_kind = svt_ddr_configuration::DDR3;*/
```

13.3 Using Catalogs to Configure Your DDR Agent

The following UVM example shows how to configure your agent using policy classes and catalog classes.

```
/**
 * This class is policy class for selecting user defined part number from the ddr3
 * catalog.
 */
class svt_ddr3_part_number_policy;
  static function int weight(svt_ddr3_vendor_part part);
    string ddr3_part_number;
    if ($value$plusargs("part_number=%s", ddr3_part_number)) begin
      $display( $sformatf("Set policy from +part_number=%s", ddr3_part_number));
      return part.get_part_number() == ddr3_part_number;
    end else
      return 0;
  endfunction : weight
endclass : svt_ddr3_part_number_policy
```

```
/**
 * This class is policy class for selecting DDR3 part number from the ddr3
 * catalog.  By default this example is using the jedec_ddr3_4G_x16_800e part
 * from the JEDEC catalog, but extended tests can override this behavior by
 * implementing the generate_configuration() method on the base test.
 */

class ddr3_default_part_policy;
  static function int weight(svt_ddr3_vendor_part part);
    return part.get_part_number() == "jedec_ddr3_1G_x16_1600G_1_250";
  endfunction: weight
endclass: ddr3_default_part_policy

class ddr_base_test extends uvm_test;

  /** Factory registration */
  `uvm_component_utils(ddr_base_test)

  /** Instance of ddr Customized configuration  */
  cust_svt_ddr_configuration cfg;
   ...
  /**
   * Defines the constructor "new". In the constructor, super methods is called
   * and the parent object is passed.
   */
  function new(string name = "ddr_base_test", uvm_component parent = null);
    super.new(name,parent);
  endfunction: new

  /** build_phase: To build various component of base class */
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Allocate the configuration object
    cfg = cust_svt_ddr_configuration::type_id::create("cfg");
    cfg.protocol_kind = svt_ddr_configuration::DDR3;
    cfg.ddr3_jedec_vif = test_top.memory_vif;

    // Initialize the configuration with values from the catalog
    generate_configuration();

   ...
endfunction: build_phase

  /**
   * This method selects a default part number from the DDR3 catalog and uses that
   * to populate the VIP configuration.  Extended tests can override this behavior by
   * implementing this method.
   */
  virtual function void generate_configuration();
    svt_ddr3_vendor_part ddr3_vendor_part;
```

```
    ddr3_vendor_part =
      svt_mem_part_mgr#(svt_ddr3_vendor_part,svt_ddr3_part_number_policy)::pick();


  /* Default Configuration */
  virtual function void default_cfg();
    svt_ddr3_vendor_part ddr3_vendor_part;

    ddr3_vendor_part =
      svt_mem_part_mgr#(svt_ddr3_vendor_part,ddr3_default_part_policy)::pick();
     if (ddr3_vendor_part != null) begin
      `uvm_info("generate_configuration", $sformatf("Chose %s using default selection",
         ddr3_vendor_part.get_part_number()), UVM_LOW);
    end

    if (cfg.load_prop_vals(ddr3_vendor_part.get_cfgfile())) begin
       `uvm_info("build_phase", "Successfully loaded default svt_ddr_configuration
         configuration ...", UVM_LOW)
        cfg.mode_register_cfg.set_default_mode_register_values();
    end
    else begin
      `uvm_fatal("build_phase", "Failed attempting to load default svt_ddr_configuration
         configuration ...")
    end
  endfunction
```

13.4 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) ;
```

13.5 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);
```

13.6 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_addr_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
```

In the following example shows loading memory, poking values to un-initialized locations, and then using the 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);
```

13.7 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);
```

13.8 DIMM Examples

13.8.1 Setting the DDR DIMM 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 protocol_kind member in the top-level configuration class svt_ddr_dimm_configuration sets the protocol level.

```
/** This functions create and randomize the DIMM, Controller and SMBUS configuration */
    virtual function void create_cfg();

    /** Create and set Controller configuration*/
    controller_cfg =
      svt_ddr_controller_configuration::type_id::create("controller_cfg", this);
    controller_cfg.dram_cfg.protocol_kind  = svt_ddr_dimm_configuration::DDR3;
    controller_cfg.operating_mode = svt_ddr_controller_configuration::DIMM;
```

13.8.2 Using Catalogs to configure your DDR DIMM Agent

The following UVM example shows how to configure your agent using policy classes and catalog classes.

```
/**
 * This class is policy class for selecting DDR3 DIMM part number from the DDR3
 * catalog. If a part number is not supplied in the command line then the
 * default part number jedec_ddr3_dimm_1G_x8_1600G_1R_noecc is selected from the
 * DIMM catalog.  */

class svt_ddr3_part_number_policy;
  static function int weight(svt_ddr3_vendor_part part);
    string ddr3_part_number;
    if ($value$plusargs("part_number=%s", ddr3_part_number)) begin
      return part.get_part_number() == ddr3_part_number;
      `uvm_info ("weight",$sformatf("User provided part_number=%s selected",
ddr3_part_number),UVM_HIGH);
    end
    else  begin
      return part.get_part_number() == "jedec_ddr3_rdimm_1G_x8_1600G_1R_noecc";
      `uvm_info ("weight",$sformatf("Default part_number=%s selected ",
ddr3_part_number),UVM_HIGH);
    end
  endfunction: weight
endclass: svt_ddr3_part_number_policy
class ddr_base_test extends uvm_test;

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

  /** Instance of the environment */
  ddr_basic_env env;

  /** Customized DIMM configuration */
  cust_svt_ddr_dimm_configuration cfg;

  /** Customized DDR controller configuration */
  svt_ddr_controller_configuration controller_cfg;
```

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

/** build_phase: To build various component of base class */
  virtual function void build_phase(uvm_phase phase);
    string phase_name = "build_phase";
    super.build_phase(phase);
    `uvm_info("build_phase", "Entered...",UVM_LOW);

/** Create the configuration object */
    create_cfg();

/** Set configuration in environment */
    uvm_config_db#(svt_ddr_controller_configuration)::set(this, "env",
      "controller_cfg", this.controller_cfg);
    uvm_config_db#(svt_ddr_dimm_configuration)::set(this, "env", "cfg", this.cfg);
    uvm_config_db#(svt_ddr_smbus_master_agent_configuration)::set(this, "env",
      "smbus_master_agent_cfg", this.smbus_master_agent_cfg);

    /** Create the environment */
    env = ddr_basic_env::type_id::create("env", this);

/** Generate RDIMM confifuration */
    generate_configuration();


/**
   * This method selects a default part number from the DDR3 catalog and uses that
   * to populate the VIP configuration. Extended tests can override this behavior by
   * implementing this    method.
   */
  virtual function void generate_configuration();
    svt_ddr3_vendor_part ddr3_vendor_part;

    /** Dram cfg handle. This handle is passed to all the dram agents present
    inside the RDIMM */
    svt_ddr_dimm_configuration dram_cfg;

    /** Create dram_cfg */
    dram_cfg = svt_ddr_dimm_configuration::type_id::create("dram_cfg",this);

    /** Display the parts available in the catalog */
    svt_ddr3_vendor_catalog::write_shelf();

   /** Use a policy class to select DIMM part number and print various catalog features
*/
    ddr3_vendor_part =
svt_mem_part_mgr#(svt_ddr3_vendor_part,svt_ddr3_part_number_policy)::pick();

/**
     * Load the selected part
```

```
      * The load_prop_vals() method is used to load the configuration values
      * or the selected part into the configuration object.
      */
     if (dram_cfg.load_prop_vals(ddr3_vendor_part.get_cfgfile())) begin
       `uvm_info("generate_configuration", $sformatf("Successfully loaded the
          configuration values for %s", ddr3_vendor_part.get_part_number()), UVM_LOW);
       if (dram_cfg.is_valid(0)) begin
         `uvm_info("build_phase", "Loaded svt_ddr_configuration is VALID ...",UVM_LOW);
       end
       else begin
         `uvm_fatal("build_phase", "Loaded svt_ddr_configuration is NOT VALID ...");
       end
     end
     else begin
       `uvm_fatal("generate_configuration", $sformatf("Unable to load the configuration
values for %s", ddr3_vendor_part.get_part_number()));
     end
endfunction
```

13.8.3 Initialize Memory Directly from the Testbench (Backdoor Access)

Initialize a segment of memory using the backdoor access method svt_mem_core::intialize(). Initialize the memory with various patterns and display them.

```
    `uvm_info("run_phase",$sformatf(" Initialize the memory with CONSTANT Pattern"),
         UVM_LOW);
    mem_core.initialize (svt_mem_core::INIT_CONST ,'h1111, 'h1020 , 'h102f) ;

    // The seeded data ('h0001) will be left shifted for every consecutive address
    `uvm_info("run_phase",$sformatf(" Initialize the memory with WALKING LEFT Pattern"),
         UVM_LOW);
    mem_core.initialize (svt_mem_core::INIT_WALK_LEFT ,'h0001, 'h1040 , 'h104f) ;

    // The seeded data ('h1000) will be right shifted for every consecutive address
    `uvm_info("run_phase",$sformatf(" Initialize the memory with WALKING RIGHT
         Pattern "), UVM_LOW);
    mem_core.initialize (svt_mem_core::INIT_WALK_RIGHT ,'h1000, 'h1050 , 'h105f) ;

    // The seeded data ('haaaa) will be decremented for every consecutive address
    `uvm_info("run_phase",$sformatf(" Initialize the memory with DECREMENT Pattern "),
         UVM_LOW);
    mem_core.initialize (svt_mem_core::INIT_DECR ,'haaaa, 'h1060 , 'h106f) ;

    // The seeded data ('hbbbb) will be incremented for every consecutive address
    `uvm_info("run_phase",$sformatf(" Initialize the memory with INCREMENT Pattern "),
         UVM_LOW);
    mem_core.initialize (svt_mem_core::INIT_INCR ,'hbbbb, 'h1070 , 'h107f) ;
```

13.8.4 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);
```

13.8.5 Using Peek and Poke

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_addr_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
```

In the following example shows loading memory, poking values to un-initialized locations, and then using the 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 the 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);
```

13.8.6 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);
```

13.9 Configurable DIMM Data Width Support

DDR4 VIP supports configurable DIMM width. The following configuration parameter is added in `svt_ddr_dimm_coniguration` class to enable this feature.

Name	Type	Default Value	Description
configurable_data_width	bit	0	0: Default operation with DIMM width 64. 1: User can configure dimm widths like 8, 16, 24, 32, 40, 48, 56, 64 from Test Bench.

This feature is supported only with randomization. It is not applicable with catalog loading. If catalog is loaded, then you need to override dimm configuration after catalog is loaded. For RDIMM support, you need to set the compile time define `SVT_DDR4_DIMM_IF_COMPONENT_CONNECT` in environment.

13.9.1 Use Case with ECC disable

In the case of RDIMM, if dram data width 4 is selected, then DIMM widths 8, 16, 24, 32, 40, 48, 56, 64 are supported. For dram width 8, supported dimm widths are 16,32,48, and 64.

```
dram_cfg.randomize() with {data_width == 4;};
dimm_cfg.configurable_data_width = 1'b1;


    if (! dimm_cfg.randomize() with {
            has_ecc                         == 0;
            dram_data_width    == dram_cfg.data_width;
            dimm_data_width    == 16;               // User defined dimm width is used
            num_ranks                  == 1;
            memory_speed_bin == dram_cfg. memory_speed_bin;
```

```
            mem_size                    ==  dram_cfg.mem_size;
            bank_addr_width     ==  dram_cfg.bank_addr_width;
            column_addr_width == dram_cfg.column_addr_width;
            row_addr_width       == dram_cfg.row_addr_width;
            ddr3_power_mode  == dram_cfg.ddr3_power_mode;
          if(dram_cfg.memory_type == svt_ddr_configuration::DDR_3DS) num_logical_rank
== dram_cfg.num_logical_rank;
          }
    ) begin
      `svt_fatal("randomize_cfg","DIMM configuration object randomization failed");
    end
```

Total DRAM created in this example will be NUM_OF_DRAM = dimm_cfg.dimm_data_width/dimm_cfg.dimm_data_width;

In the above example, DIMM width 16 is selected with dram width as 4, there will be 4 DRAM in system.

13.9.2 Use Case with ECC Enable

In RDIMM case, If dram data width 4 is selected DIMM widths 16, 24, 32, 40, 48, 56, 64 are supported. For dram width 8, supported dimm widths are 24,40,56.

```
dram_cfg.randomize() with {dram_width == 8;};
dimm_cfg.configurable_data_width = 1'b1;

    if (! dimm_cfg.randomize() with {
            has_ecc                        == 1;
            dram_data_width    == dram_cfg.data_width;
            dimm_data_width   == 40;
            num_ranks                 == 1;
            memory_speed_bin == dram_cfg. memory_speed_bin;
            mem_size                    ==  dram_cfg.mem_size;
            bank_addr_width     ==  dram_cfg.bank_addr_width;
            column_addr_width == dram_cfg.column_addr_width;
            row_addr_width       == dram_cfg.row_addr_width;
            ddr3_power_mode  == dram_cfg.ddr3_power_mode;
          if(dram_cfg.memory_type == svt_ddr_configuration::DDR_3DS) num_logical_rank
== dram_cfg.num_logical_rank;
          }
    ) begin
      `svt_fatal("randomize_cfg","DIMM configuration object randomization failed");
    end
```

Total DRAM created in this example will be NUM_OF_DRAM = dimm_cfg.dimm_data_width/dram_cfg.dimm_data_width;

In the above example, DIMM width 40 is selected with dram width as 8, and there are 4 DRAM in system and one will be ECC DRAM.

DIMM Width such as 32 is not supported with DRAM width 8 when ECC is enabled because there must be even number of DRAM for A-Side and B-Side in RCD buffer.

13.10 Jitter Support

VIP calculates average clock from the clock which is running on the interface and uses this value for clock jitter related checks.

The following are the variables description related to jitters:

Variable Name	Type	Default Value	Description
continue_avg_clk_calculation	Bit	0	By default VIP calculates tCK_avg over initial 200 valid clocks (so that the performance does not get affected). If required to calculate tCK_avg through-out simulation over last 200 valid clocks, you must set this variable.
enable_clk_jitter_checks	Bit	0	By default the clock jitter checks(for example, `tch_avg_timings_check`, and `tjit_cc_total_timings_check`) are disabled. To enable the clock jitter checks, you must set this variable.
clk_jitter_delta	Int	1	By default the delta value for clock jitter checks is 1. You can increase this value if the clock is going out of range knowingly.

13.10.1 Usage Note

This is an example to configure different value:

```
if (! dram_cfg.randomize() with {
                    continue_avg_clk_calculation == 1;
                    enable_clk_jitter_checks == 1;
                    clk_jitter_delta == 20;
                  }
              )
              begin
                  `svt_fatal("randomize_cfg","DIMM configuration object randomization
failed");
              end
```

14
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.

14.1 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/ddr_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.

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

14.2 The “vcspcvlog” Simulator Target in Makefiles

The vcspcvlog simulator target in the Makefiles of the VIP UVM examples enables 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.

14.3 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.

14.4 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.

14.5 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.

14.6 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. Check the available VIP examples using the following command:

   $VC_VIP_HOME/bin/dw_vip_setup -i home

3. Install the example.

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

   $VC_VIP_HOME/bin/dw_vip_setup -e ddr_svt/tb_ddr_svt_uvm_basic_sys -svtb

   cd examples/sverilog/ddr_svt/tb_ddr_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.

15 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.

15.1 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:

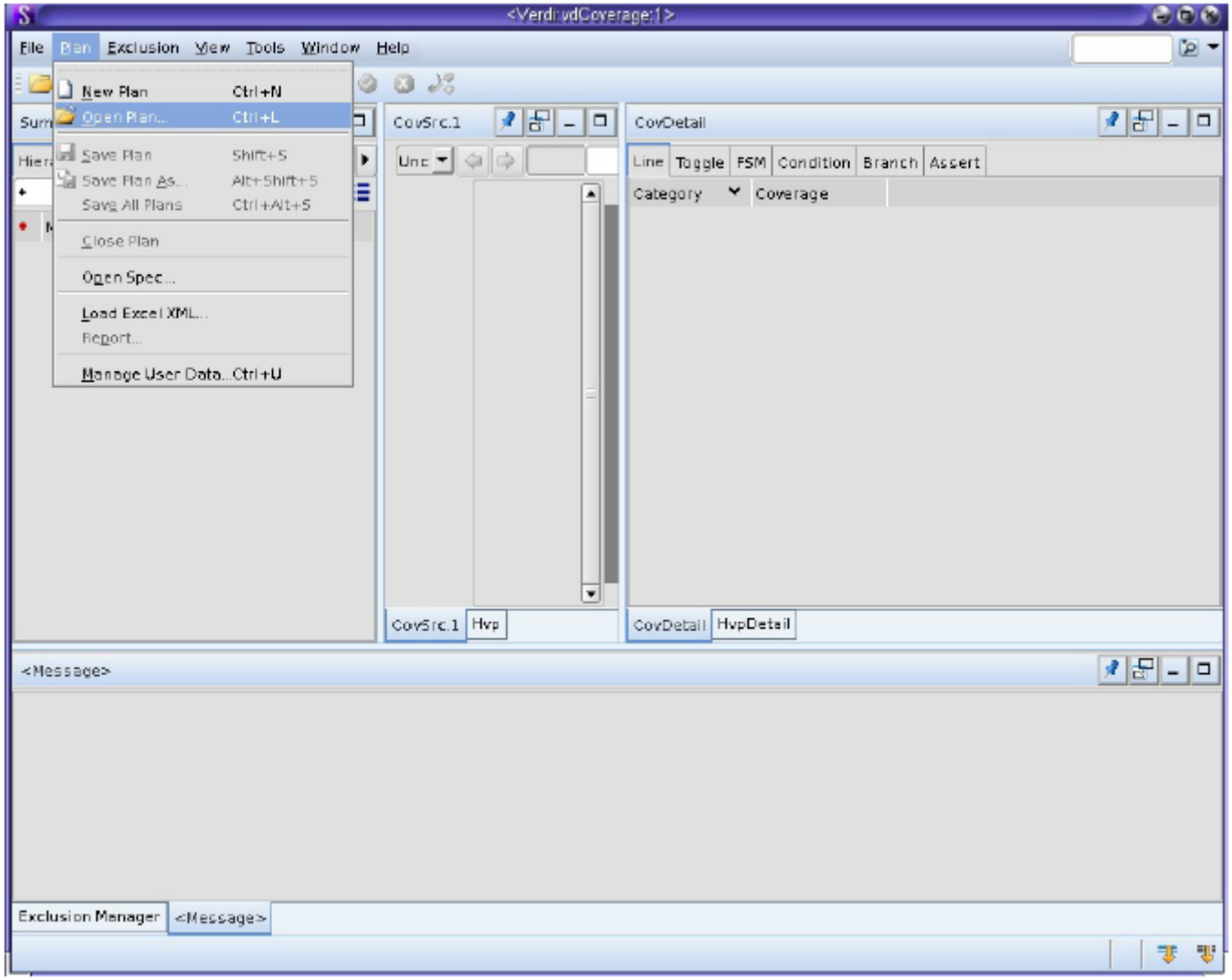
   cd <intermediate_example_dir>
2. 2.Invoke the make command with the name of the test you want to run, as follows:

   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:

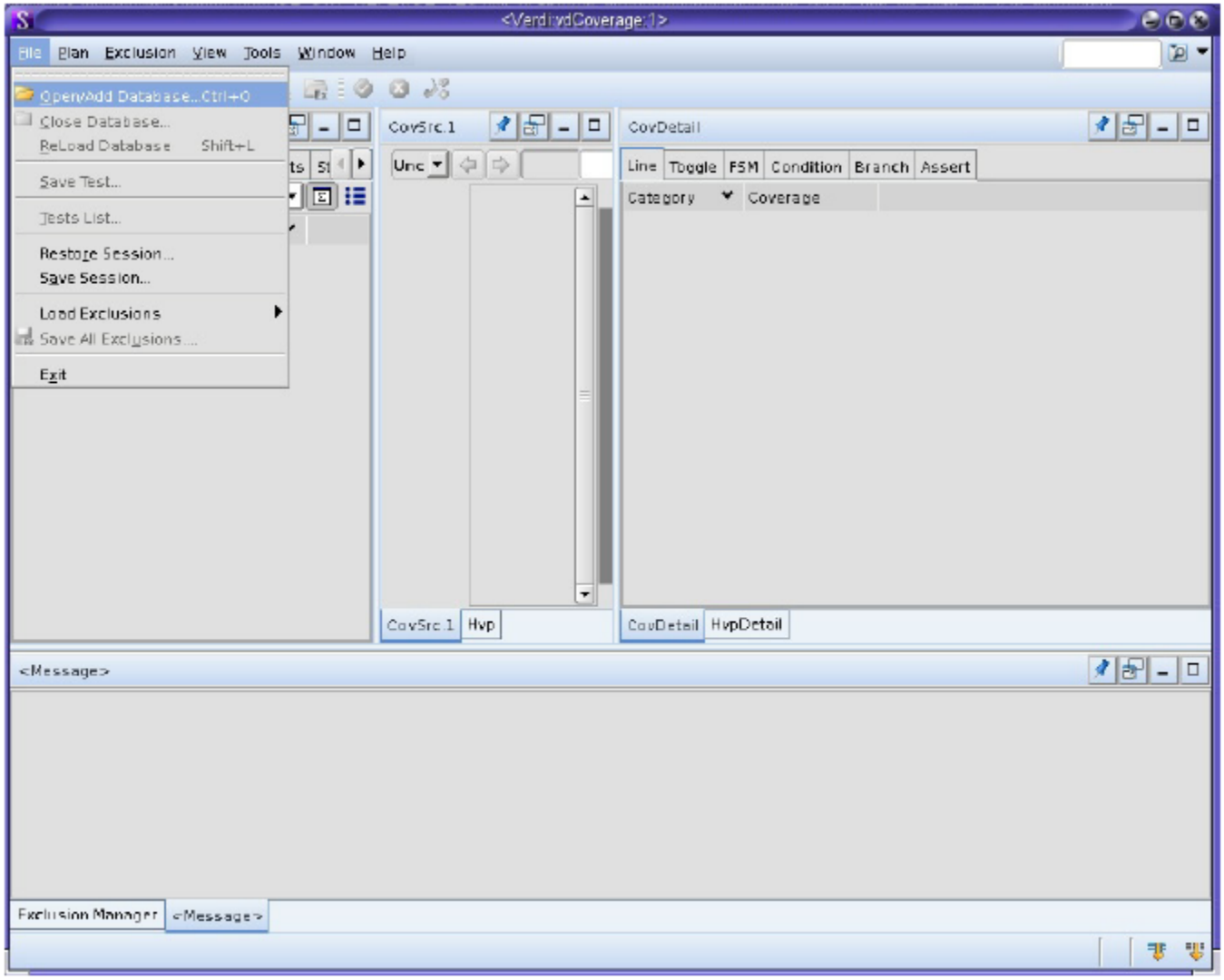
   cp $VC_VIP_HOME/vip/svt/ddr_svt/latest/doc/VerificationPlans .
5. Select the load plan from the Plan drop down menu as shown in the following illustration.

Figure 15-1 Open Plan



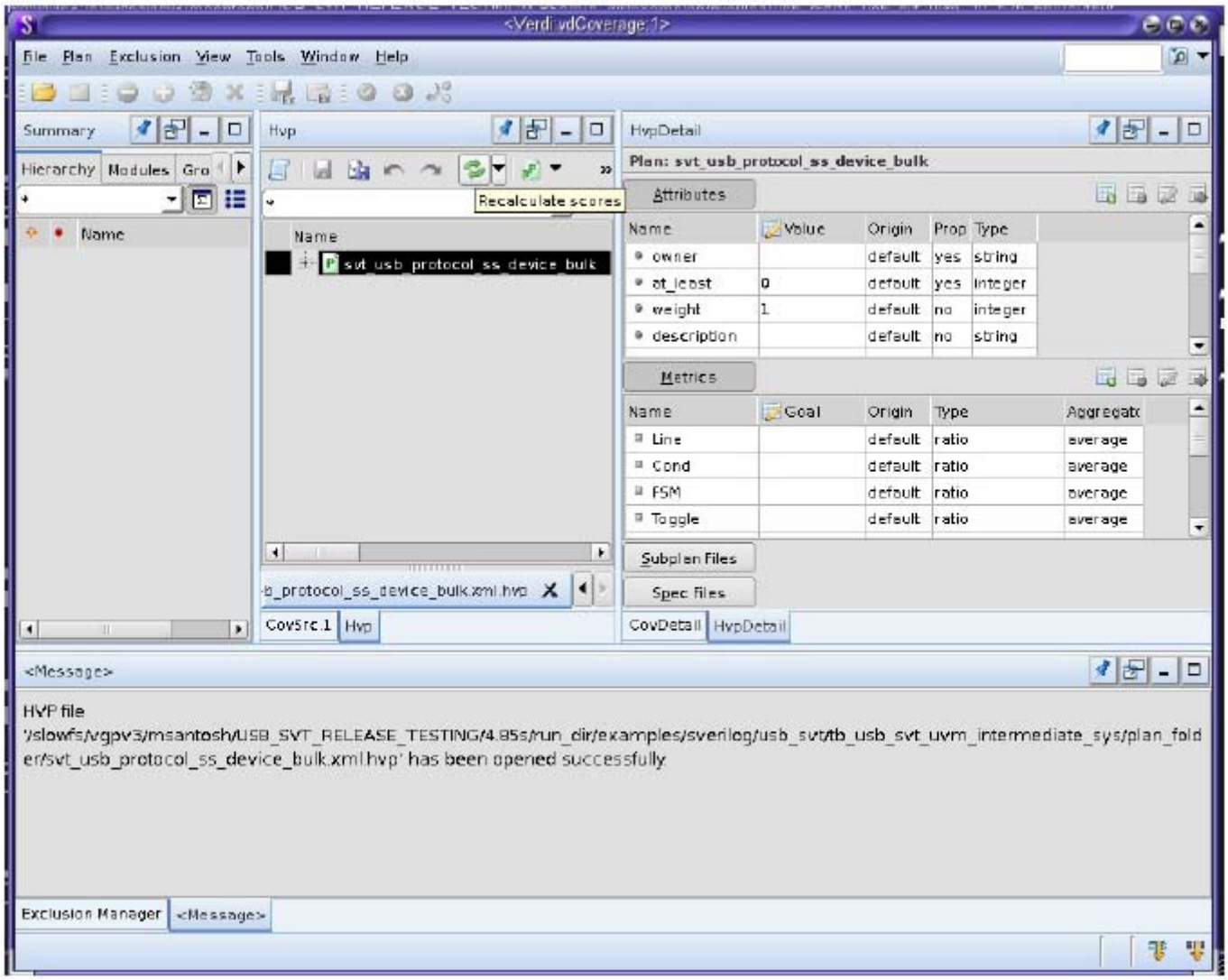
Load the coverage database (simvcssvlog.vdb) from the output folder in the current directory using the Verdi drop down menu, as shown in the following illustration.

Figure 15-2 Verdi GUI



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

Figure 15-3 Recalculate



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 will default to being enabled on all VIP instances with the default options listed as follows:

Note the following about the plusarg:

- 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.

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

```
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`.

Examples:

Enable on all VIP instances with default options:

```
+svt_debug_opts
```

Enable 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
```

Enable on all instances:

- starting at time 1000 until time 1500:

```
+svt_debug_opts=start_time:1000,end_time:1500,verbosity:VERBOSE
```

Enable 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=1
```

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=1 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 105.

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 the "<username>.<uniqid>.svd" in the current directory. The simulation transcript file will not be part of this and it will be saved separately.

   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.