SYNOPSYS®

DesignWare Memory Model User's Manual

To search the entire manual set, press this toolbar button. For help, refer to intro.pdf.

Release 2002.12

November 19, 2002

Contents

Figures

Tables

Preface

About This Manual

This manual contains user information for DesignWare Memory Model software.

Related Documents

This manual is part of the DesignWare Memory Model documentation set. The documents in this set are listed in the *Guide to DesignWare Memory Model Documentation*, and located at $LMC_HOME/doc/dwmm/manuals.

Manual Overview

This manual contains the following chapters and appendices.

Table 1: Manual Contents

Chapter	**Contents**
Chapter 1, “Getting Started”	Introduces the DesignWare Memory Model product and its features, and provides a quick overview of the installation process.
Chapter 2, “Installation”	Describes how to install and license the DesignWare Memory Model software, and contains links to instructions for instantiating your models into your testbench and simulating your design.
Chapter 3, “Using DesignWare Memory Models”	Discusses using DesignWare Memory Models in your simulation.
Chapter 4, “MemScope”	Introduces the MemScope interactive debugging tool, and explains how to use it.

Table 1: Manual Contents (Continued)

Chapter	Contents
Chapter 5, “HDL Testbench Interface”	Describes the HDL testbench commands included with the DesignWare Memory Model technology software package.
Chapter 6, “C Testbench Interface”	Describes the C testbench commands included with the DesignWare Memory Model technology software package.
Chapter 7, “VERA Testbench Interface”	Describes the VERA testbench commands included with the DesignWare Memory Model technology software package.
Chapter 8, “Library Tools”	Discusses several library tools included with the DWMM technology software package.
Appendix A, “Reporting Problems”	Describes how to troubleshoot the DesignWare Memory Model software, and provides procedures for getting help from Synopsys Customer Support.
Appendix B, “Selecting the Model Version”	Explains how to install and maintain multiple versions of a model or tool, and how to select a specific version for use.
Appendix C, “Licensing Information”	Discusses DesignWare Memory Model licensing.
Appendix D, “Memory Image Files”	Explains how to preload DesignWare Memory Models from files, or dump model data to files.

Retiring MemPro Classic

As of this release, the “classic” MemPro technology (including the tools MemGen and MemSpec, and all model libraries and specification files) are no longer shipped or supported as part of the DesignWare Memory Model software. These products are no longer part of the DWMM technology software package.

If you are already using MemPro models in your simulation, you can continue using them with the DWMM software, with no modifications needed. If you need to add a MemPro model to your DWMM system, you can easily do so -- refer to the appropriate “custom DesignWare Memory Model” section in the *Simulator Configuration Guide for Synopsys Models*.

You also won’t need new license keys — your current MemPro keys will license this release.

Typographical and Symbol Conventions

This document uses the following conventions.

Table 2: Documentation Conventions

Convention	Description and Example
%	Represents the UNIX prompt.
Bold	User input (text entered by the user). `% `**`cd $LMC_HOME/hdl`**
`Monospace`	System-generated text (prompts, messages, files, reports). `No Mismatches: 66 Vectors processed: 66 Possible"`
Italic or *`Italic`*	Variables for which you supply a specific value. As a command line example: `% `**`setenv LMC_HOME`** *`prod_dir`* In body text: In the previous example, *prod_dir* is the installation directory.
| (Vertical rule)	Choice among alternatives, as in the following syntax example: `-effort_level low | medium | high`
[ ] (Square brackets)	Enclose optional parameters: *`pin1`* `[`*`pin2`* `...` *`pinN`*`]` In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* … *pinN*]).

Table 2: Documentation Conventions (Continued)

Convention	Description and Example
TopMenu > SubMenu	Pulldown menu paths, such as: **File > Save As …**

Getting Help

If you have a question about Synopsys products, use the following resources:

- Product documentation installed on your network or computer, or on the root level of your Synopsys CD-ROM (if applicable).
- Product documentation for the latest version of all products on the web:

  http://www.synopsys.com/products/designware/docs
- Datasheets for all memory verification models:

  http://www.synopsys.com/memorycentral
- The online Support Center available at one of the following URLs:
  - DesignWare Macrocells, DesignWare Foundation Library, or coreBuilder Tools customers:

    http://solvnet.synopsys.com/
  - SmartModel, FlexModel, DesignWare Memory Model, and VMC customers:

    http://www.synopsys.com/support/lm/support.html

If you still have questions about the following products, you can call a Synopsys support center:

- DesignWare Macrocells, DesignWare Foundation Library, and coreBuilder Tools:
  - United States:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific Time, Mon–Fri.
  - Canada:
    Call 1-650-584-4200 from 7 AM to 5:30 PM Pacific Time, Mon–Fri.
  - All other countries:
    Find other local support center telephone numbers at the following URL:

    http://www.synopsys.com/support/support_ctr/
- SmartModels, FlexModels, DesignWare Memory Models, and VMC:
  - North America:
    Call 1-800-445-1888 from 7:00 AM to 5 PM Pacific Time, Mon–Fri.
  - All other countries:
    Call your local sales office.

Additional Information

You can find general information about Synopsys and its products at this URL:

http://www.synopsys.com

For additional Synopsys documentation, refer to the following web page:

http://www.synopsys.com/products/designware/docs/

For up-to-date information about the latest implementation IP and verification models, visit the IP Directory on the web:

http://www.synopsys.com/memorycentral

Comments?

To report errors in this document or make suggestions for improving it, please send e-mail to this address:

doc@synopsys.com

To report an error that occurs on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your e-mail message. This will provide us with information to identify the source of the problem.

1 Getting Started

The DesignWare Memory Model software provides you with pre-compiled, SWIFT-compatible, platform- and language-independent binary memory models for supported Verilog or VHDL simulators, as well as tools for debugging the models in your simulation.

This chapter introduces the DesignWare Memory Model (DWMM) product and its features, and provides a quick overview of the installation process.

The chapter contains the following topics:

Introduction to DesignWare Memory Models

DesignWare Memory Models provide the following capabilities in your simulation environment:

- Dynamically managed 4-state (0, 1, X, or U) data storage. Models efficiently allocate workstation memory for all designs, whether low-density or memory-intensive.
- Word storage address ranges from 0 to 2^N-1, where N is an integer in the range 2 through 64. Supports integral word widths from 1 through 2048 bits.
- Unloading of a range of memory addresses. Unloaded memory addresses return to the uninitialized state and return the workstation memory associated with those addresses.
- Loading memory contents from or dumping memory contents to external files.
- Continuous control of model messages.
- Creation and deletion of traces on a range of addresses. When the simulation run performs a read or a write operation on a traced address, the model causes an event and provides the memory instance, address, data, and type of access.
- Means of interrogating and controlling the models through functions implemented as Verilog tasks, VHDL procedures, C functions, or VERA methods.

The DWMM software also includes the MemScope interactive debugging tool. You can configure your testbench to generate database and history files during simulation, and then use those files with MemScope to view and debug your post-simulation results. You can also connect MemScope to your running simulation, and interactively view and modify your simulation’s memory contents dynamically. Then, as needed, you can revise your testbench and resimulate until you achieve the results you want.

Installing the DWMM Software

Installing the DesignWare Memory Model software consists of the following primary tasks:

- Downloading the DWMM Technology Software Package and the FLEXlm License Software
- Downloading at least one DWMM Memory Model
- Installing the DWMM Technology Software Package and Memory Model
- Checking the integrity of the installation, and setting up licensing
- Instantiating the DWMM Memory Models into your testbench

For detailed instructions on these and related tasks, see "Installation" on page 25.

Finding New and Updated Models

Your installation contains the latest model and tool versions that were available at the time of release. But models are continually being updated, so be sure to check for the latest versions of all available models at the Synopsys Memory Central web page:

http://www.synopsys.com/memorycentral

You can easily search this directory for the model you need, and follow the displayed instructions to download and install new and updated models. For details, see "Updating DWMM Software" on page 41.

Features of DesignWare Memory Models

DesignWare Memory Models have many common features, and share many of the characteristics of Synopsys SmartModels. Features include 64-bit time, supported logic values, error checking, unknown handling, and selectable propagation delays.

64-Bit Time

All DWMM models use 64 bits to compute elapsed simulation time. If simulation time exceeds this capacity, the models behave unpredictably.

Logic Values

DWMM models use a four-state, multivalue logic system, as shown in Table 3.

Table 3: DWMM Model Logic Values

Symbol	Meaning
0	Strong 0
1	Strong 1
X	Strong X
U	Uninitialized (treated as unknown)

Error Checking

DWMM models provide usage and timing checks that display error, note, trace, or warning messages during simulation. The format and location of these messages depends on the design environment, but the content is essentially the same.

Usage Checks

Usage checks, which vary greatly with device type, help ensure a chip is used correctly. For example, an SRAM check might produce a message like: “Address, A0-A13, changed within Write cycle.” These checks also document times, device names, instances, and error types.

For example:

```
WARNING at time 105 from cy7c1324.
Unknown address on the address bus, WRITE ignored.
```

Timing Checks

Timing checks include the component-specific set-up, hold, frequency, pulse width, and recovery times specified in the vendor’s specifications. Timing checks generate a single value—they do not have a range and thus are not affected by the propagation delay range.

For example:

```
WARNING at time 255 from cy7c1324.
PULSE WIDTH check tCL on CLK was 15; 30 is the specified minimum.
Violation of System Clock Deasserted Pulse Width.
```

Nominal and Worst-Case Specifications for Timing Checks

In cases where a manufacturer specifies both nominal and worst-case values for a timing parameter, the model always uses the worst-case specification.

Handling Unknowns (X-Handling)

DesignWare Memory Models make the most of each simulation by generating or propagating unknowns only when necessary. When appropriate, a model issues a warning message rather than propagating an unknown. This “pessimistic” method of handling unknowns can preserve the usefulness of a simulation.

For details about the X-handling features of a particular model, see the reference document for the model’s memory type.

Selectable Propagation Delays

All DWMM models support a range of propagation delay values to represent minimum, typical, and maximum specifications.

Model Reconfiguration

DWMM models support the ability to force a model to reload its memory image at any time during a simulation.

2 Installation

This chapter describes how to install and license the DesignWare Memory Model software, and contains the following topics:

Note

Installing the DesignWare Memory Model products in this release requires write privileges to the $LMC_HOME directory.

Step 1 — Prepare for Installation

Before you install any DWMM software, do the following to make installation as problem-free as possible.

- Make sure DWMM supports your simulators, platform, and verification tools — see *DesignWare Memory Model Supported Simulators and Platforms.*
- Configure your system as described in the *Simulator Configuration Guide for Synopsys Models*.
- Make sure you have at least 200 MB of free disk space per hardware platform. The actual requirement varies depending on the number of models installed, the number of versions of each model, and the simulator options specified. If you install on more than one platform, the second and all subsequent platforms will require roughly 25% of the disk space used by the first platform installation.

Note

If you already have an LMC_HOME directory on your system, install the DWMM software into that directory.

- When installing a given release of a DWMM product for multiple platforms (for example, Solaris and HP-UX), install the products in the same top-level directory.
- Make sure the C-shell (/bin/csh) is available. You can use any shell to install the DWMM software, but you must have access to a C-shell during the installation.

Search Paths

When your system is configured as described in the *Simulator Configuration Guide for Synopsys Models*, the DWMM software uses the following search paths:

- LMC_HOME <*products-models tree*> (required)
- LMC_CONFIG <*model_config1*:*model_config2*:…:*model_configN*> (optional)

Step 2 — Download the DWMM Technology Software Package

Follow this procedure to download the DWMM technology software package from the Synopsys web site.

Note
If you are installing from a CD-ROM, skip this step and go to “Step 5 — Install the DWMM Technology Software Package” on page 29.

1. In your web browser, open the following URL:

   http://www.synopsys.com/memorycentral

2. Click “Memory Models.”
3. Click “Quick Guide to Download and Installation.”
4. Click “Download Technology Software Package.”
5. Follow the on-screen instructions to download the Technology Software Package (TSP) and sl_unzip package for your platform.

Note
You might need to hold down the Shift key when you click on an sl_unzip file, so that the file is downloaded rather than displayed.

Step 3 — Download and Install the FLEXlm License Software

DWMM applications use the Synopsys Common Licensing (SCL) software to control their usage. You will need to download and install the SCL software before you can use the DWMM software and models.

1. Go to the SCL software from the Synopsys web site:

   http://www.synopsys.com/keys

2. Check the table “Synopsys Common Licensing Supported Platforms by OS & Keyword” to find the keyword for your platform and OS. You will need this keyword to determine which files to download and install.

3. Follow the instructions displayed on the web site to perform these tasks:

   - Download and install the SCL files.
   - Download and customize your site’s license key file.
   - Start the license daemon.
   - Set up the user environment to access the key file.

   You may want to save or print the displayed instructions for reference.

Step 4 — Download a DWMM Memory Model

Follow this procedure to download one or more DWMM memory models. The package you download includes the model binaries for the platforms you select, and the model datasheets (PDF format).

1. Use your web browser to go to the Synopsys Memory Central web site:

   http://www.synopsys.com/memorycentral

2. Click Memory Models, and follow the displayed instructions to search for and download the models you need.

Step 5 — Install the DWMM Technology Software Package

You'll use the Admin tool (sl_admin) to install the DWMM technology software package. Simply follow the prompts that appear on-screen: Admin guides you through the entire process, including setting your $LMC_HOME environment variable and selecting your installation platform.

Note

Do not install the DWMM product on DEC Alpha, IBM RS/6000, Sun SunOS, or PC NT workstations.

Install the DWMM technology software package as follows:

Note

If you're installing from a CD-ROM, begin at step 3, below.

1. Make the file sl_unzip.*platform* executable:

   ```
   chmod a+x sl_unzip.platform
   ```

2. Extract the installation image:

   ```
   sl_unzip.platform dwmm_platform.zip
   ```

   This extracts the image to a new directory named cdrom.

3. Set your LMC_HOME environment variable with the path to the directory you want to install the DWMM software into:

   ```
   % setenv LMC_HOME install_dir
   ```

4. Move to the cdrom directory (containing the installation image), and invoke the Admin tool (sl_admin):

   ```
   % sl_admin.csh
   ```

   After a moment, the Admin window opens, displaying the Install From dialog.

5. Click Open, select your platform (if not already selected) from the dialog that appears, and click Install.

   Installation then begins. The process displays status messages to show its progress, and displays "Install complete" when finished.

6. Exit the Admin tool by selecting **File** > **Exit**.

Note
When installation is complete, you can delete the following to save disk space: the files dwmm_*platform*.zip and sl_unzip.*platform*, and the directory cdrom and its contents.

Step 6 — Install the DesignWare Memory Model

Install your downloaded DesignWare Memory Model as follows.

Note
You must have the DWMM technology software package installed before you can install a DWMM memory model — see page 29.

1. Extract the installation image:

   ```
   sl_unzip.platform mximage_name.zip
   ```

   This extracts the image to a new directory *image_name*/*platform* (for example, mx20020516140235/hp700).

2. Make sure your LMC_HOME environment variable is set to the path that contains the DWMM technology software package (see step 3 on page 29).

3. Move to the *image_name*/*platform* directory (which contains the installation image), and invoke the Admin tool (sl_admin):

   ```
   % sl_admin.csh
   ```

   After a moment, the Admin window opens, displaying the Install From dialog.

4. Click Open, then click Install.

   Installation then begins. The process displays status messages to show its progress, and displays “Install complete” when finished.

5. Exit the Admin tool by selecting **File** > **Exit**.

Note
You’ll need to perform this process once for each platform you’re installing.

Step 7 — Install Adobe Acrobat Reader

The Adobe® Acrobat® Reader is required to view or print the DWMM online documentation, which is provided as PDF (Portable Document Format) files.

To get a free copy of the Reader, go to Adobe’s web site:

http://www.adobe.com

Even if you already have the Reader installed on your system (search for acroread.exe), you should perform this step to make sure you have the latest version available.

Step 8 — Check Installation Integrity

To verify that your software is installed properly, Synopsys provides a tool called swiftcheck that can identify many common installation problems. The swiftcheck tool does the following:

- Verifies the values of the following environment variables:
    - $LMC_HOME (required; produces a fatal error if not set)
    - $LMC_COMMAND
    - $SNPSLMD_LICENSE_FILE
    - $LD_LIBRARY_PATH (Linux)
- Verifies that the necessary runtime utilities are installed, using the $LMC_HOME environment variable for the path.
- Makes sure the DWMM software files are installed in the correct locations.
- Loads a model that you specify and attempts to exercise basic functionality. This provides you with a simple way to test configuration files.

The swiftcheck tool displays error messages if errors occur, and records all messages in a log file called swiftcheck.out.

Syntax

Use this command-line syntax to run the swiftcheck tool:

```
% $LMC_HOME/bin/swiftcheck model [-switches]
```

Argument

model	Specifies the installed model that you want to load and exercise as a test of basic installation integrity.

Switches

-e[rrorlog] *filename*	Specifies an error log output file other than the default (swiftcheck.out).
-h[elp]	Provides a help listing.
-hh[elp]	Prints a detailed help message.
-j[edecfile] *filename*	Loads the specified configuration file for the JEDEC model.
-m[emoryfile] *filename*	Loads the specified configuration file for the memory model.
-n[omodels]	Runs swiftcheck without loading and exercising a model.
-p[clfile] *filename*	Loads the specified configuration file for the HV model.
-t *timing_version*	Loads a particular timing version for the specified model.
-u[sage]	Provides a help listing.

Examples

This causes swiftcheck to load and exercise the CY7C1324_MX memory model:

```
% $LMC_HOME/bin/swiftcheck cy7c1324_mx
```

Step 9 — Set Up Licensing

Once you’ve installed the DWMM software and checked the installation, you need to set up the licenses required to run the product.

Network Licensing for DesignWare Memory Models

Synopsys simulation models use GLOBEtrotter Software FLEXlm floating licenses to control their use. This section describes the elements that are unique to the Synopsys implementation of FLEXlm.

Note

The licensing software requires a live connection with the license server. If this connection is broken, the license is revoked.

When a DWMM application is invoked or model is first loaded by a simulator, it must obtain a valid license token before it can run. The model keeps its license token until the simulator exits. Models do not release license tokens when the simulator is paused.

After installation, the FLEXlm documentation is available at $LMC_HOME/doc/flexlm/TOC.htm. For more information about FLEXlm, see the Globetrotter web site:

http://www.globetrotter.com

License Tokens

The DWMM product can use license tokens from several different categories, called features. The DWMM application or model looks for available tokens by searching license features in a specific order.

For run-time model usage, the order is this:

- DESIGNWARE-REGRESSION
- DESIGNWARE
- DIRECTMEM

Before using the DWMM tools or models, you must set the SNPSLMD_LICENSE_FILE environment variable. This variable can contain the path to the license.dat file, or the port number and hostname of the server process. Port number selection is controlled by the SERVER record in license.dat.

```
setenv SNPSLMD_LICENSE_FILE path_to_license.dat
```

or

```
setenv SNPSLMD_LICENSE_FILE port@hostname
```

You can use the environment variable DW_LICENSE_OVERRIDE to override this behavior. DW_LICENSE_OVERRIDE allows two values: DESIGNWARE and DESIGNWARE-REGRESSION. If set to one of these values, only the specified feature name can be used to authorize a model. Only DESIGNWARE-REGRESSION enabled models can be authorized with the DESIGNWARE-REGRESSION feature name.

For more information, see "Licensing Information" on page 203.

Step 10 — Instantiate the Model

Once the DWMM software is installed, you'll need to instantiate your models into your testbench and simulate the design.

Use the links below to view detailed instantiation instructions for your simulator in the *Simulator Configuration Guide for Synopsys Models*:

- "Using DesignWare Memory Models with SWIFT Simulators"
- "Using DesignWare Memory Models with VCS"
- "Using DesignWare Memory Models with Verilog-XL"
- "Using DesignWare Memory Models with NC-Verilog"
- "Using DesignWare Memory Models with MTI Verilog"
- "Using DesignWare Memory Models with Scirocco"
- "Using DesignWare Memory Models with MTI VHDL"
- "Using DesignWare Memory Models with NC-VHDL"
- "Using DesignWare Memory Models with QuickSim II"
- "Using DesignWare Memory Models with SystemC"
- "Using VERA with DesignWare Memory Models"

Step 11 — View the Documentation

After you've finished installing and instantiating the DWMM software, take a moment to become familiar with the online documentation provided with the software.

Be sure to read the file readme.txt, and the *DesignWare Memory Model Release Notes*.

Note

The documents installed with the DWMM software or located on the CD-ROM were the most current available when the download package or CD-ROM was produced. For the latest versions of the DWMM documents, be sure to check the Synopsys Memory Central web site:

http://www.synopsys.com/products/designware/docs

Viewing DWMM Manuals and Other Documents

You can access the online manuals for the DWMM software at the following locations:

- In your $LMC_HOME/doc/dwmm/manuals directory.
- On the Synopsys Memory Central web site:

  http://www.synopsys.com/products/designware/docs
- If you have a CD-ROM, in the /manuals directory at the root level of the CD-ROM.

You can view the PDF files using the Adobe Acrobat Reader. For details about the documentation files, see the *Guide to DesignWare Memory Model Documentation*.

Viewing Model Datasheets

You can view the model datasheets from the Browser tool (sl_browser) using its Doc pull-down menu. Although the datasheets are in PDF format, you should use Browser instead of Adobe Reader to read the datasheets, because Browser automatically selects the correct datasheet version according to the selected model version.

Note

Before invoking the Browser, make sure your $LMC_HOME environment variable is set to your install directory.

1. Invoke the Browser tool:

```
% $LMC_HOME/bin/sl_browser
```

2. Select the model you want by clicking it in the Browser model selection window.

3. Click the datasheet icon (it looks like an AND gate) to display the datasheet for that version of the model.

Note

For details about the Browser tool, see the *SmartModel Library User’s Manual*.

You can find additional model datasheets by going to this URL and searching for the models you want:

http://www.synopsys.com/memorycentral

DWMM Installation Tree

When you install the DesignWare Memory Model software, the result is an LMC_HOME file structure that includes all DWMM product files. Figure 1 shows the structure of an LMC_HOME tree where DWMM is the only installed product.

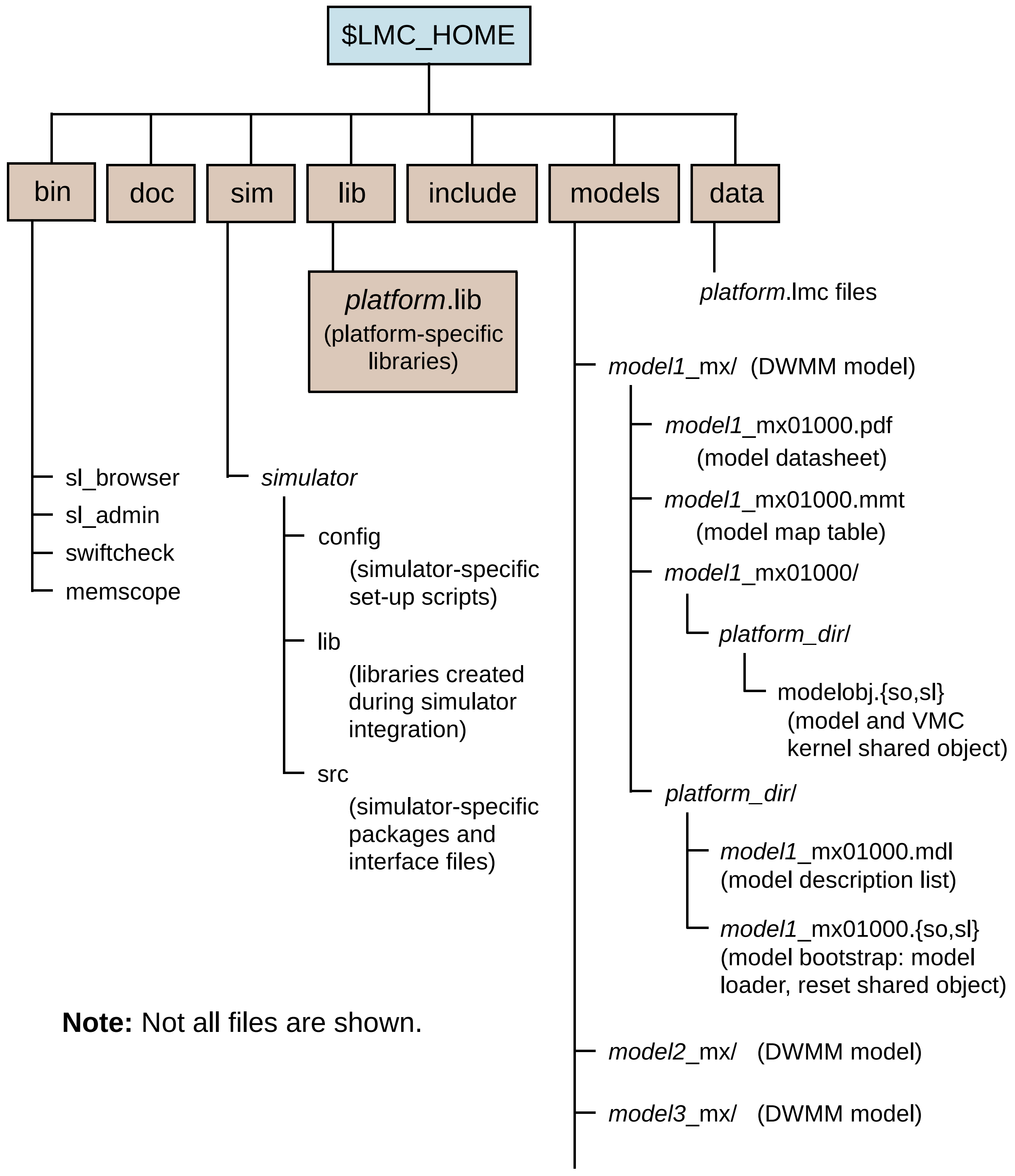

$LMC_HOME
bin
doc
sim
lib
include
models
data
platform.lib
(platform-specific libraries)
platform.lmc files
sl_browser
sl_admin
swiftcheck
memscope
simulator
config
(simulator-specific set-up scripts)
lib
(libraries created during simulator integration)
src
(simulator-specific packages and interface files)
*model1*_mx/ (DWMM model)
*model1*_mx01000.pdf
(model datasheet)
*model1*_mx01000.mmt
(model map table)
*model1*_mx01000/
platform_dir/
modelobj.{so,sl}
(model and VMC kernel shared object)
platform_dir/
*model1*_mx01000.mdl
(model description list)
*model1*_mx01000.{so,sl}
(model bootstrap: model loader, reset shared object)
Note: Not all files are shown.
*model2*_mx/ (DWMM model)
*model3*_mx/ (DWMM model)

Figure 1: $LMC_HOME Directory Structure

DWMM Software Directories

As shown in Figure 1, the DWMM software installation includes the following directories:

- **$LMC_HOME/bin** — contains all user tools, including MemScope, Admin (sl_admin) and Browser (sl_browser), and command-line tools such as compile_timing, smartccn, and swiftcheck.
- **$LMC_HOME/doc** — contains the product documentation, and includes the following subdirectories:
  - **/dwmm/manuals** — contains the product manuals and release notes (all in PDF format).
  - **/dwmm/ref_docs** — contains reference manuals for individual memory models.
  - **/dwmm/readme** — contains the readme and related files (in ASCII format).
  - **/dwmm/help** — contains PDF-based help files for MemScope, and for the Admin and Browser tools.
- **$LMC_HOME/lib** — contains binaries, scripts, and data not directly associated with a specific model. The files for each platform are in a subdirectory *platform*.lib, where *platform* is hp700, sun4Solaris, or x86_linux.

Caution
Do not alter the contents of the $LMC_HOME/lib directory.

- **$LMC_HOME/include** — contains simulator interface source files. These are generally header files for Verilog PLI simulations, but may be C source or other source code that must be compiled and linked with a simulator.

Caution
Do not alter the contents of the $LMC_HOME/include directory.

- **$LMC_HOME/models** — contains all installed models. For descriptions of these files, see “DWMM Model Files” on page 39.
- **$LMC_HOME/data** — contains library configuration files (*platform*.lmc) needed by the DWMM version control software, each of which consists of a list of DWMM models with specific versions defined. For more information on configuration files, see “Configuration (LMC) Files” on page 198.

DWMM Model Files

As shown in Figure 1, each DWMM model's subdirectory includes the following files:

- **Model datasheet (.pdf)** — a model-specific datasheet that provides information you need to use the model properly. The model datasheets supplement, but do not duplicate, the manufacturer's datasheets for the modeled hardware.

  In general, the model datasheets include the following:

  - Information about the model name, title, function, and memory type, and the date of the last change to the model.
  - Supported timing versions.
  - Information sources (specific vendor databooks or datasheets) used to develop the model.
  - Usage notes, including a hyperlink to the reference manual for the model's class, which contains generic information for all models in that class (SDRAM, SRAM, etc.).
  - Port descriptions.
  - Values of all timing parameters for the supported components, and the minimum, typical, and maximum delay ranges for each.
  - A model history that lists changes to the model.
  - A tracking number — called the MDL version number — that indicates the model version and is useful for technical support.

    Getting Datasheets

    You can get DWMM model datasheets several ways:

    - From the Memory Central web site:

      http://www.synopsys.com/memorycentral

      Follow the displayed instructions to search for and download the model datasheets you want.
    - Using the Browser tool (sl_browser): select the model and click the datasheet icon in the upper left portion of the vertical tool bar — for details, see "Viewing Model Datasheets" on page 35.

      Because you can have more than one version of a model in the same $LMC_HOME/models directory, be sure to use the Browser tool to access the correct version of the model datasheet.

- **Model map table (.mmt)** — contains information about the device, model ports, and attributes that can be retrieved by SWIFT function calls.
- **Model description list (.mdl)** — specifies all LMC_HOME shared libraries and core services (see below) used by the model at runtime, along with the tool and data file versions used to create the model.
- **Required LMC libraries and core services** — the LMC_HOME libraries and core services specified in the model description list. (Libraries and tools used for debugging — such as the Synopsys MemScope tool, and an interface for HDL control — are not included, but are available for download as the DWMM Technology Software Package.)

Naming Conventions

DWMM models use the naming convention *devicename*_mx.

Updating DWMM Software

From time to time, you may need to update your DWMM software to obtain enhancements for the core DWMM technology software, or to obtain new or updated DWMM memory models.

Updating the DWMM Technology Software

Use the following procedure to update your installation of the DWMM technology software.

1. Obtain the updated DWMM technology software package using the procedure given in "Step 2 — Download the DWMM Technology Software Package" on page 27.
2. Install the software package as described in "Step 5 — Install the DWMM Technology Software Package" on page 29.
3. Check the installation as described in "Step 8 — Check Installation Integrity" on page 31.
4. Recompile the following libraries, if required by your simulator:
    - **VHDL users:** mempro_pkg.vhd and slm_hdlc.vhd
    - **Verilog users:** mempro_pkg.v
    - **VERA users:** mempromodel.vr, lstmodel.vr, and vera_user.c

    For details, see the *Simulator Configuration Guide for Synopsys Models*.
5. Check the update's readme.txt file and release notes for the latest information on the updated software — see "Step 11 — View the Documentation" on page 35 for details.

Downloading a New or Updated Memory Model

Use the following procedure to download and install a new or updated DesignWare memory model.

1. Download the model as described in "Step 4 — Download a DWMM Memory Model" on page 28.
2. Install the model as described in "Step 6 — Install the DesignWare Memory Model" on page 30.
3. Check the model's datasheet for the latest information — see "Viewing Model Datasheets" on page 35 for details.

Versioning

Models are the basic units of the DWMM package. The DWMM software environment enables you to install more than one version of any model in an $LMC_HOME directory.

Multiple model versions allow design teams to use different versions of a given model without interfering with each other: design team #1 (for example) can get an enhancement they need for a particular model without affecting design team #2 that may not need that enhancement.

You can also install new model shipments that you receive from Synopsys directly into an existing DWMM installation.

Figure 2 illustrates the benefits of this flexible versioning system. Different design teams can select new or revised models using custom configuration files. For more information, see "Versioning" on page 42.

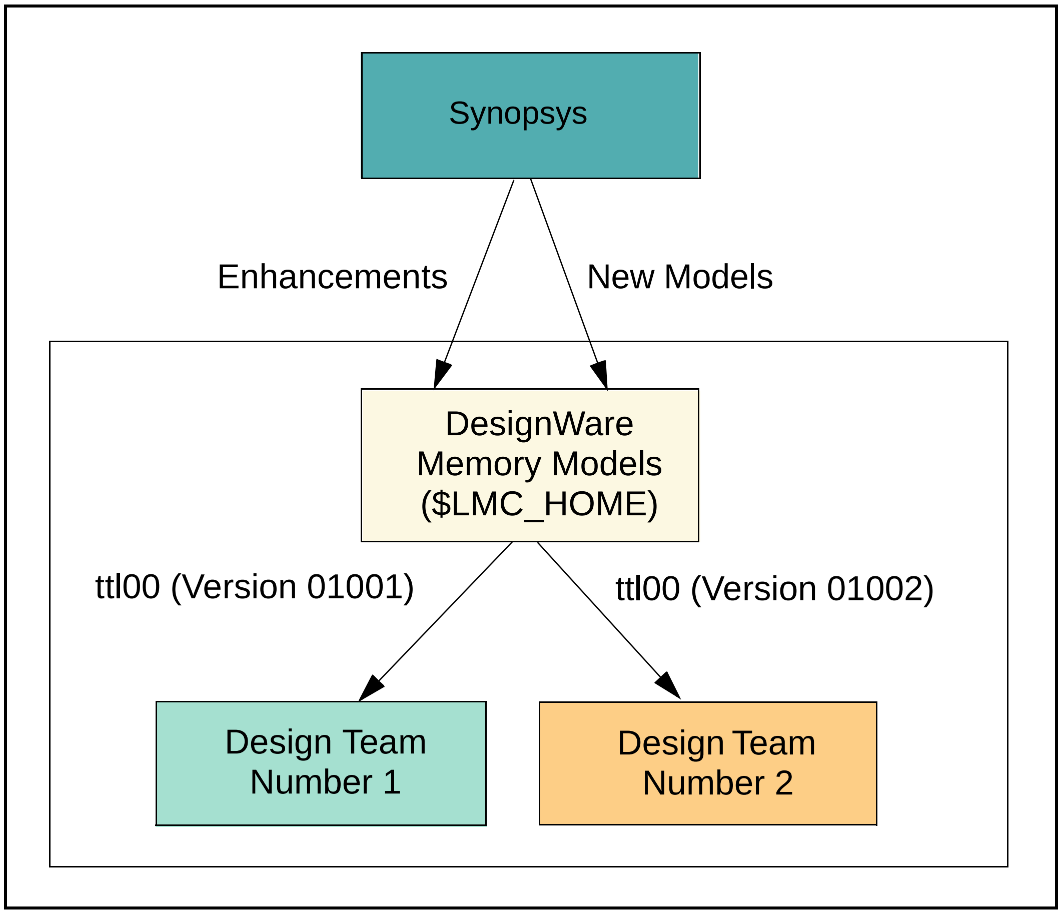


Figure 2: DesignWare Memory Model Versioning Environment

3 Using DesignWare Memory Models

This chapter provides information about using DesignWare Memory Models in your simulation, and contains the following topics:

- "Default Memory Values" on page 44
- "Controlling Model Messages" on page 48
- "Setting Timing Behavior" on page 50
- "Debugging Your Design" on page 51

Note
Before proceeding, make sure you have instantiated your DWMM models — see "Step 10 — Instantiate the Model" on page 34.

Note
See the *Simulator Configuration Guide for Synopsys Models* for information on the attributes used by the models.

Default Memory Values

You can specify default data for DWMM models by setting the DefaultData attribute in the model’s instantiation: see “Using DesignWare Memory Models with SWIFT Simulators” in the *Simulator Configuration Guide for Synopsys Models*.

The model returns the default data value whenever the simulation reads a memory location not previously loaded via an initialization file *and* not previously written with valid data.

The model also returns the default data value for reads to all memory locations deallocated with the mem_unload testbench command:

- “mem_unload” on page 135 (HDL)
- “slm_mem_unload” on page 165 (C)
- “inst.unload” on page 182 (VERA)

The following rule applies to the DefaultData value:

- The DefaultData value can contain Xs (unknowns). In VHDL simulations, the value can also contain Us (uninitialized).

  The length of DefaultData depends on the format in which it is specified:

  - In hexadecimal format, each X or U is considered to be 4 bits long.
  - In octal format, each X or U is 3 bits long.
  - In binary format, each X or U is 1 bit long.

Note

The default memory value for a memory instance is set with a Verilog defparam or a VHDL generic. If you use the defparam or generic to set a different default memory value, you must ensure that the width of the value is consistent with that of the model.

Binary Format

Syntax

value

Description

The data value is an ASCII string of valid binary characters, as follows:

- *value*: A string of binary characters supported for Verilog or VHDL simulators:

Verilog:	x | X | 0 | 1 | _
VHDL:	x | X | u | U | 0 | 1 | _

You can place underscore characters anywhere in the value string to improve readability.

Examples

```
0110_XX01_U0U1_1111
xxxxxxxx_1000000
uu1x
```

Note

The first and last examples are not valid in Verilog.

Verilog Format

Syntax

[*size*] *base_specifier value*

Description

The data value is an ASCII string made up of three elements, as follows:

1. *size*: An optional decimal value that specifies the data width.

   If *value* is smaller than the specified *size*, then *value* is padded to the left with zeros, unless the left-most bit in *value* is an unknown (x or X), in which case *value* is padded to the left with Xs.

2. *base_specifier*: A Verilog base identifier:

   ‘b | ‘B — Binary

   ‘o | ‘O — Octal

   ‘d | ‘D — Decimal

   ‘h | ‘H — Hexadecimal

3. *value*: A string of characters valid for the selected radix:

Binary:	x | X | 0 | 1 | _
Octal:	x | X | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _
Decimal:	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | _
Hexadecimal:	x | X | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F | _

No spaces are allowed, but you can use underscore characters to improve readability.

Examples

Hexadecimal

The following example specifies a 16-bit value with a binary representation of 1111XXXX11111111:

```
16’hfxff
```

Decimal

The following example specifies a value with a binary representation of 11111111:

```
’d255
```

Octal

The following example specifies a value with a binary representation of 010XXX101XXX:

```
’o2x5x
```

Binary

The following example specifies an 8-bit value with a binary representation of 010101X1:

```
8’b010101x1
```

VHDL Format

Syntax

base_specifier " *value* "

Description

The data value is an ASCII string made up of two elements:

1. *base_specifier*: A VHDL base identifier:

   b | B — Binary

   o | O — Octal

   x | X — Hexadecimal

2. " *value* ": A quoted string of characters valid for the selected radix:

Binary:	x | X | u | U | 0 | 1 | _
Octal:	x | X | u | U | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _

   Hexadecimal:

      x | X | u | U | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F | _

No spaces are allowed, though you can use underscore characters to improve readability.

Examples

Hexadecimal

The following example specifies a value with a binary representation of 1010XXXX0101UUUU:

```
X"ax5U"
```

Octal

The following example specifies a value with a binary representation of 111UUU111:

```
O"7u7"
```

Binary

The following example specifies a value with a binary representation of 10XU1100:

```
B"10xu1100"
```

Decimal Format

Syntax

< D | d > *value*

Description

The data value is an ASCII D or d followed by valid decimal characters, as follows:

- *value*: A string of decimal or underscore characters:

  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | _

Examples

The following example specifies a value with a binary representation of 10000001:

```
d129
```

The following example specifies a value with a binary representation of 11111111:

```
D255
```

Controlling Model Messages

Message Categories

Model messages are grouped into categories and each category can be individually enabled or disabled for each model instance. Several message categories are applicable to all models; additional categories can be defined for specific models or model types. The general categories are as follows.

Fatal

Fatal messages are always enabled. When a fatal error is detected, the simulation always stops immediately after reporting the message. For example, referencing an unknown DWMM model instance handle causes a fatal error.

Error

Error messages apply to situations that are erroneous, but the model is able to recover and simulation can continue. For example, an error message would be initiated when the model receives a command that would put it in an illegal state.

Warning

Warning messages apply to situations that users might want to check, but that are not obviously wrong. For example, a warning would be issued when significant bits of an address are ignored.

Info

Info messages inform you of the status or behavior of the model; for example, when a read or write occurs.

Timing

Timing messages are used to report timing constraint violations. Typical situations that cause timing messages are setup or pulse width violations.

X-Handling

X-handling messages report X values on input ports when valid data was expected.

Controlling Message Output

There are two ways to control messaging for DWMM models:

- Using the MessageLevel attribute. (For more information and instantiation examples, see “Using DesignWare Memory Models with SWIFT Simulators” in the *Simulator Configuration Guide for Synopsys Models*.)
- Using a testbench command. (For more information, see “mem_msg_level” on page 126 (HDL), “slm_set_message_level” on page 163 (C), or “inst.set_msg_level” on page 180 (VERA).)

By default, models display these general message categories: Fatal, Error, Warning, Timing, and X-handling. If you set an attribute for an instance, that setting overrides the default behavior. Similarly, if you use the testbench interface, it overrides the attribute.

For information on controlling messaging through a testbench command, see “mem_msg_level” on page 126 (HDL), “slm_set_message_level” on page 163 (C), or “inst.set_msg_level” on page 180 (VERA).

Setting Timing Behavior

Each DWMM model can be set to run in either timing-accurate mode or function-only mode by setting the model's TimingVersion and DelayRange attributes:

- TimingVersion — Specifies the timing version that a model instance uses when scheduling changes on its outputs, or checking setup and hold time on its inputs. Set to "none" for function-only mode. See the model datasheet for the default and allowable timing versions.
- DelayRange — Specifies a propagation delay range for a model instance: "Min", "Typ", or "Max" (default setting). If TimingVersion is set to "none" (so that the model runs in function-only mode), DelayRange is ignored.

For details about these attributes, see "Using DesignWare Memory Models with SWIFT Simulators" in the *Simulator Configuration Guide for Synopsys Models*.

Function-Only Mode

To run in function-only mode, set TimingVersion to "none".

In function-only mode, all timing checks are disabled, there is no propagation delay, and DelayRange is ignored.

Timing Checks

DWMM models support timing checks, with constraints such as setup, hold, recovery, skew, period, and pulse width supported. Propagation delays (including pin-to-pin delays, internal pin delays, and access delays) are embedded in each model's code.

Timing checks are performed automatically whenever TimingVersion is set to anything other than "none". See the model datasheet for information about setting TimingVersion.

Timing-Accurate Mode

To run in timing-accurate mode:

1. Set TimingVersion to a valid timing component. (If you set an invalid value, the model uses the default timing component and issues a warning.)
2. Set DelayRange to select the Min, Typ, and Max range values for the selected timing component. By default, the model uses the Max timing range.

   Table 4 shows the default values that DWMM models use for Min, Typ, and Max if all three values are not specified in the model datasheet.

Table 4: Delay Range Defaults

Given	Min	Typ	Max
Min and Max	Min	Max	Max
Max only	Max	Max	Max
Min only	Min	Min	Min

In addition, some devices specify a negative setup value for certain parameters: a value of zero (0 ns) is used for these.

Debugging Your Design

DWMM models support a wide range of debugging techniques, including model trace function, loading memory images and dumping memory data, and using MemScope to examine memory databases and transaction histories created during a simulation.

Model Messages

DWMM models include informational messages that indicate the model's current state, and can be useful for debugging. These messages are *not* turned on by default; to activate them, follow the procedure in "Using Synopsys Models with Simulators" in the *Simulator Configuration Guide for Synopsys Models*.

Testbench Commands

You can generally debug your model using standard testbench commands:

- If you use an HDL simulator, you can link into your simulation for access to the Verilog or VHDL commands. You can also use the C and VERA testbench interfaces.
- If you use a non-HDL simulator, you can use VERA or C to access the testbench commands.

For command listings and instructions on use, see "HDL Testbench Interface" on page 113, "C Testbench Interface" on page 139, and "VERA Testbench Interface" on page 167.

Loading Memory Images

You can load memory data from an external file (MIF or Verilog $readmemh format) using the MemoryFile attribute and mem_load testbench command. You can also preload a memory instance by overriding the MemoryFile attribute with the pathname of the memory file in a defparam in the Verilog testbench, or as a generic in the VHDL testbench. You can also use the existing mem_load and mem_unload testbench commands to load memory instances and unload specified ranges during simulation. (See "Comparison of HDL Commands, VERA Methods, and C Functions" on page 115 for the appropriate command names in HDL, C, and VERA.)

Dumping Memory Data

DWMM models support dumping memory data to a file (MIF or Verilog $readmemh format) — see "Dumping Memory Data" on page 214.

Access to Memory Data

You can use the MemScope tool to access memory data in your DWMM simulations. See "MemScope" on page 53 for details and instructions.

4 MemScope

This chapter introduces the MemScope interactive debugging tool, and explains how to use it.

MemScope can help you understand the behavior of designs that use DesignWare Memory Models. MemScope provides details about each memory model in the design, including a complete history of the transactions executed by each model. This information can help you track down design problems and monitor the verification process.

MemScope provides three primary functions you can use in debugging your design:

- **Examining transaction history files** — You can examine history files that detail all transactions that took place in your simulation.
- **Creating logical address maps** — You can create an abstract memory map — called a logical address map — of one or more physical memory devices in your simulation, which you can then use to view or modify the data in the system.
- **Accessing devices during simulation** — You can connect MemScope to a running simulation to view and modify memory contents dynamically.

Chapter Contents

A Quick Tour of the MemScope Graphical User Interface

This section introduces MemScope’s graphical user interface, which provides windows, menus, dialogs, and other visual elements to make MemScope more convenient to use.

Note

You may want to have MemScope running while you become familiar with its interface — see “Starting MemScope” on page 69.

Selecting Items in MemScope

As in many software applications, you can use the mouse or keyboard to select the various items displayed in MemScope.

Using the Mouse

The mouse functions much as in other software that uses a mouse:

- Choose items by pointing and left-clicking.
- Open items by double-clicking.
- Open context-sensitive help and pop-up menus by right-clicking.

Using the Keyboard

Most menus and commands contain underlined letters in their names (File, Edit, and so on) that enable you to choose them quickly using the keyboard.

- To open a menu, press the Alt key and the menu’s underlined letter.
- To choose an item from an open menu, press the item’s underlined letter, or highlight the item using the arrow keys and press Return or Enter.

In addition, many items have one- or two-key combinations you can use to select them — for example, you can press Ctrl-H to open a history file. These key combinations are shown in the menus to the right of the commands.

You can also use the keyboard to move around in other ways:

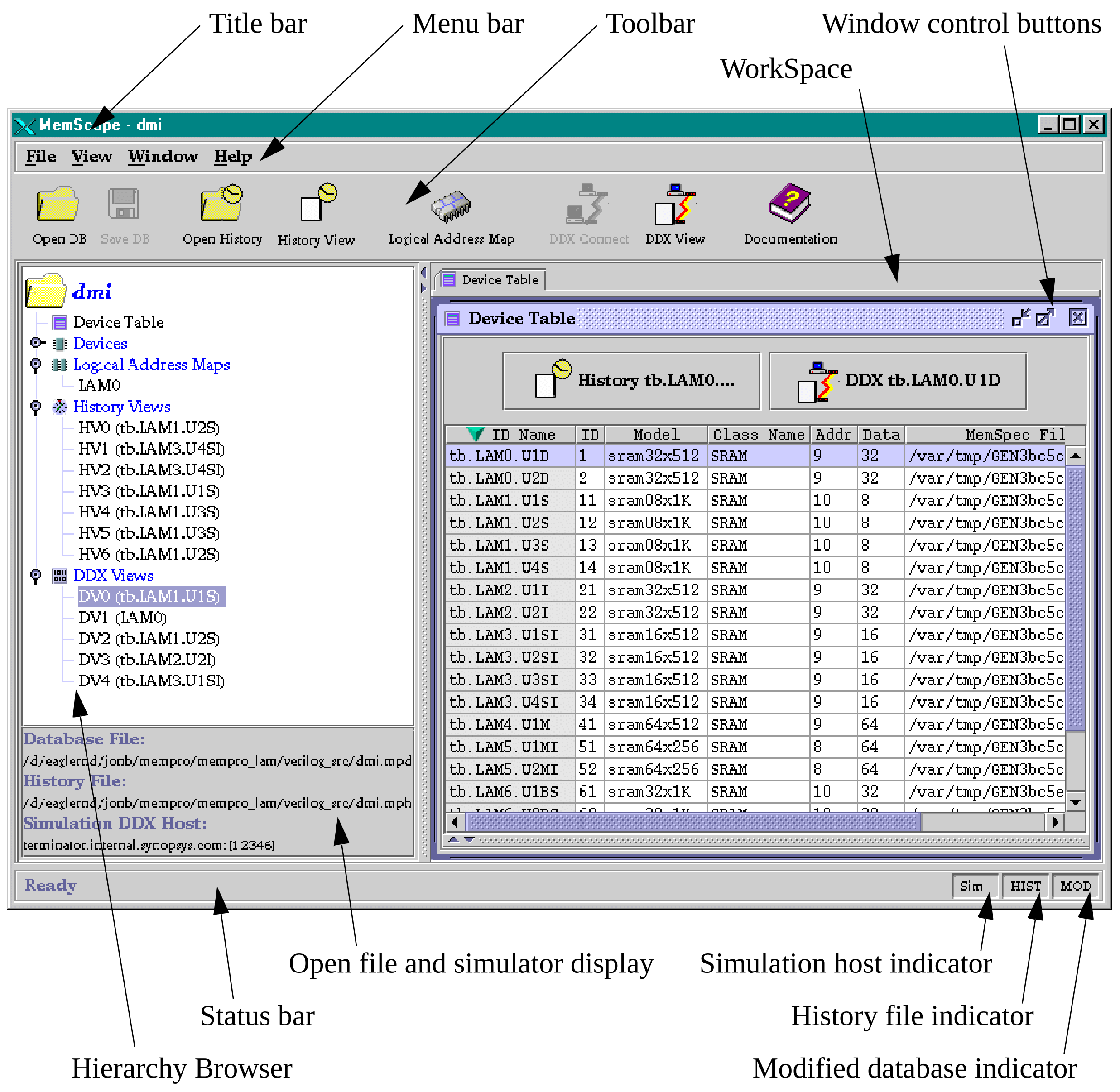
- Use the arrow keys to move left or right from an open menu to adjacent menus, or to open (right arrow) or close (left arrow) a “cascading” menu from an open menu.
- Use the Tab key to move among items on the screen.
- Use Ctrl-F to search, and Ctrl-G to repeat the search.

Note

The Alt key feature may not function if you run MemScope from an Exceed window under Windows NT.

The MemScope Graphical User Interface

MemScope’s graphical interface provides windows, menus, dialogs, and other graphic elements that you will use to work with the tool.



dmi
- Device Table
- Devices
- Logical Address Maps
  - LAM0
- History Views
  - HV0 (tb.LAM1.U2S)
  - HV1 (tb.LAM3.U4SI)
  - HV2 (tb.LAM3.U4SI)
  - HV3 (tb.LAM1.U1S)
  - HV4 (tb.LAM1.U3S)
  - HV5 (tb.LAM1.U3S)
  - HV6 (tb.LAM1.U2S)
- DDX Views
  - DV0 (tb.LAM1.U1S)
  - DV1 (LAM0)
  - DV2 (tb.LAM1.U2S)
  - DV3 (tb.LAM2.U2I)
  - DV4 (tb.LAM3.U1SI)

Database File:
/d/eaglernd/jonb/mempro/mempro_lam/verilog_src/dmi.mpd
History File:
/d/eaglernd/jonb/mempro/mempro_lam/verilog_src/dmi.mph
Simulation DDX Host:
terminator.internal.synopsys.com: [12346]

ID Name	ID	Model	Class Name	Addr	Data	MemSpec Fil
tb.LAM0.U1D	1	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM0.U2D	2	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM1.U1S	11	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U2S	12	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U3S	13	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U4S	14	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM2.U1I	21	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM2.U2I	22	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM3.U1SI	31	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U2SI	32	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U3SI	33	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U4SI	34	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM4.U1M	41	sram64x512	SRAM	9	64	/var/tmp/GEN3bc5c
tb.LAM5.U1MI	51	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c
tb.LAM5.U2MI	52	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c
tb.LAM6.U1BS	61	sram32x1K	SRAM	10	32	/var/tmp/GEN3bc5e

Figure 3: MemScope Graphical Interface and Elements

The primary components of the interface are as follows:

Table 5: Elements of the MemScope Interface

This element....	**Does this....**
Title bar	Displays the name of the open database.
Menu bar	Provides access to MemScope’s menus and their commands — see “Menu Bar” on page 57.
Toolbar	Provides convenient buttons for performing common tasks — see “Toolbar” on page 63.
WorkSpace	Contains windows for the items you’re viewing.
Window control buttons	Minimizes the window to a tab showing the window title; click to restore. Lets the window float within the WorkSpace. Maximizes the window within the WorkSpace. Closes the window.
Hierarchy Browser	Provides access to the memory models in your design, along with the logical address maps and views that you define — see “Hierarchy Browser” on page 64.
Open file and simulator display	Displays the names and paths of the open database and history file, as well as the host name and port number of any open simulator host connection.
Status bar	Displays messages about MemScope’s current operation.
Simulation host indicator	Displays **Sim** when a connection to a simulator host is established.
History file indicator	Displays **HIST** when a history file is open.
Modified database indicator	Displays **MOD** when the database has been modified since it was last saved.

Menu Bar

The menu bar appears at the top of the display, and contains the menus and commands you’ll use to operate MemScope.

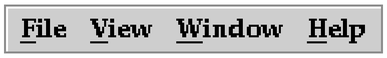


Figure 4: MemScope Menu Bar

Note
Menu items that are not available in a particular situation are greyed out.

File Menu

Use the commands in the File menu to open and close databases, history files, and DDX connections, and to import logical address maps from database files.

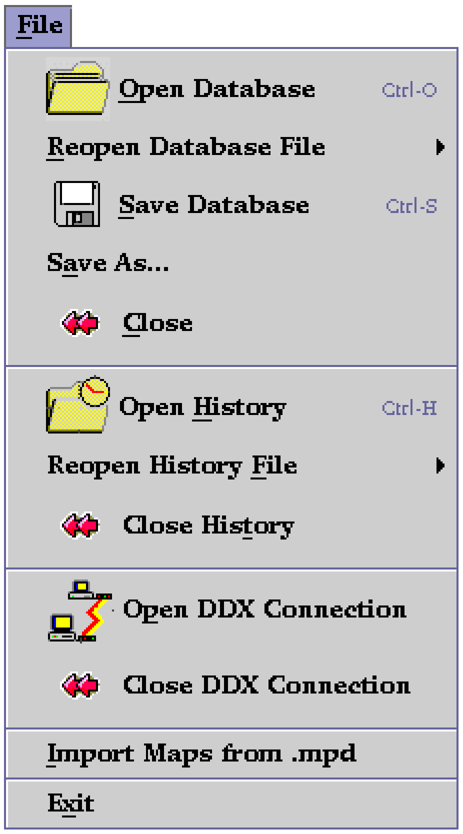


Figure 5: File Menu

Table 6: File Menu Commands

This command....	Does this....
Open Database	Opens a dialog that lets you choose and open a database (.mpd file), and closes the current database if one is open. (See page 69 for information about databases.)
Reopen Database File	Provides a cascading submenu that lists databases you've opened recently.
Save Database	Saves the open database. If you choose Save Database again later, the saved database is updated.
Save As...	Saves the open database under another name.
Close	Closes the open database and all associated windows. (If you modified but did not save the database, you'll be prompted to save it.)
Open History	Opens a history (.mph) file. (See page 69 for information about history files.)
Reopen History File	Provides a cascading menu that lists history files you've opened recently.
Close History	Closes the current history file.
Open DDX Connection	Opens a Dynamic Data Exchange (DDX) connection with a simulation. (See page 106 for information about using DDX.)
Close DDX Connection	Closes the current DDX connection.
Import Maps from .mpd	Loads logical address maps from a database (.mpd file).
Exit	Quits MemScope. (If you modified but did not save an open database, you'll be prompted to save it.)

View Menu

Use the commands in the View menu to create and validate logical address maps, create History Views and DDX Views, and configure several items in the MemScope display.

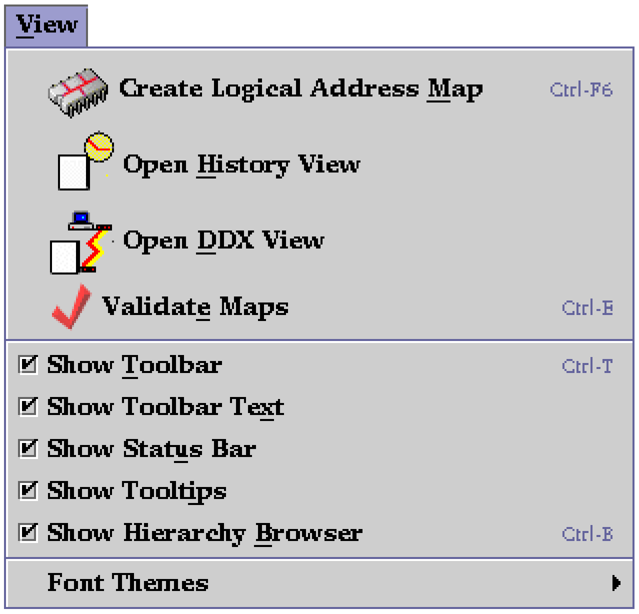
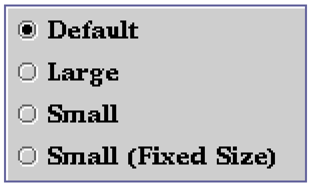


Figure 6: MemScope View Menu

Table 7: View Menu Commands

This command....	Does this....
Create Logical Address Map	Starts the process of creating a logical address map — see "Working with Logical Address Maps" on page 81.
Open History View	Opens a saved History View of a device — see "Examining Transaction History Files" on page 78.
Open DDX View	Opens a saved Dynamic Data Exchange (DDX) View of a device — see "Viewing and Editing Memory Data During Simulation" on page 106.
Validate Maps	Checks all open logical address maps for internal consistency and correctness — see "Working with Logical Address Maps" on page 81.

Table 7: View Menu Commands

This command....	Does this....
Show Toolbar	Displays the Toolbar in the MemScope window — see page 63.
Show Toolbar Text	Displays captions beneath the Toolbar buttons.
Show Status Bar	Displays the status bar in the MemScope window.
Show Tooltips	Displays brief descriptions of items in MemScope when the cursor is over them.
Show Hierarchy Browser	Displays the Hierarchy Browser — see "Hierarchy Browser" on page 64.
Font Themes	Selects the font size used in the MemScope display.  ● Default ○ Large ○ Small ○ Small (Fixed Size)

Window Menu

Use the commands in the Window menu to control how MemScope displays windows, and to move among the windows you have open.

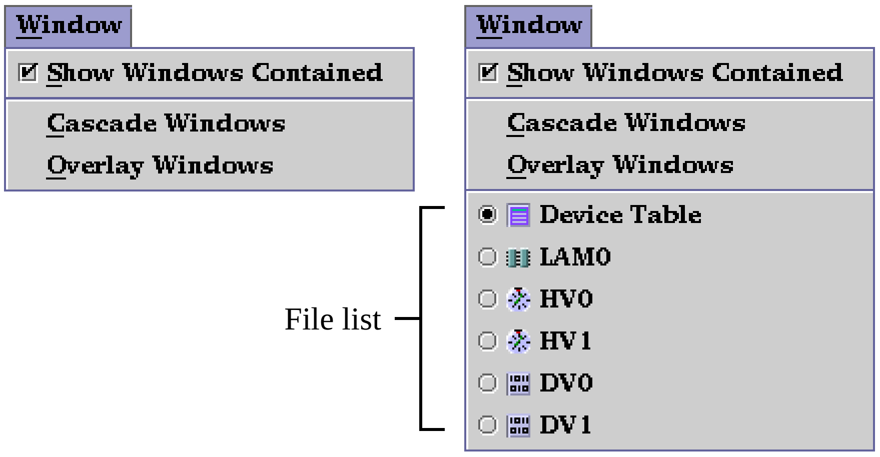
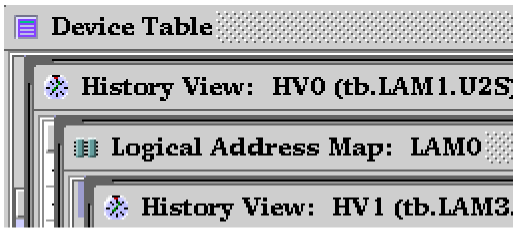
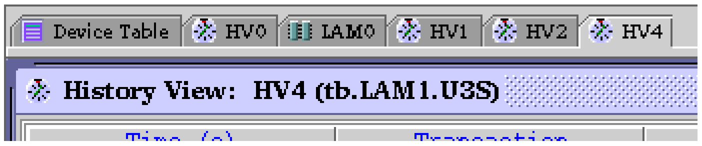


Figure 7: MemScope Window Menu

Table 8: Window Menu Commands

This command....	**Does this....**
Show Windows Contained	Contains all MemScope’s windows within the WorkSpace. If this is not selected, windows “float” independently along with the main MemScope window.
Cascade Windows	Cascades the windows in the WorkSpace.  Device Table History View: HV0 (tb.LAM1.U2S) Logical Address Map: LAM0 History View: HV1 (tb.LAM3.
Overlay Windows	Displays the windows in the WorkSpace in a single layer, with a selection tab at the top of each.  Device Table HV0 LAM0 HV1 HV2 HV4 History View: HV4 (tb.LAM1.U3S)
(File list)	Lists all open windows in the WorkSpace. To display a window, click its radio button.

Help Menu

The Help menu provides access to MemScope's built-in online help and reference material.

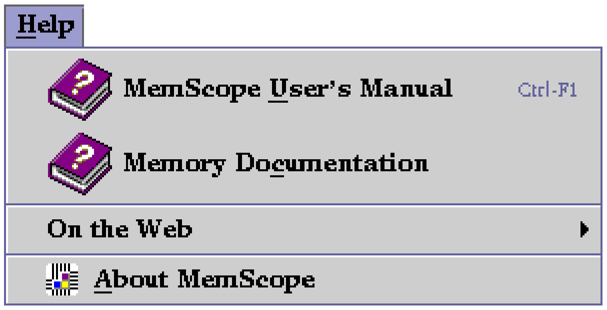


Figure 8: MemScope Help Menu

Table 9: Help Menu Commands

This command....	Does this....
MemScope User's Manual	Displays the *DesignWare Memory Model User's Manual*.
Memory Documentation	Displays the *Guide to DesignWare Memory Model Documentation*, which contains links to all documents in the set.
On the Web	Opens a browser link to one of several Synopsys web pages: Synopsys home, Memory Central, Models, or DesignWare. **Note:** Netscape must be included in your system's search path in order to use these links.
About MemScope	Displays the version of MemScope you're using.

Toolbar

The Toolbar provides quick access to a number of tasks you’ll commonly perform while using MemScope. Each of these buttons works exactly the same as the equivalent command from the pull-down menus listed in Table 10.

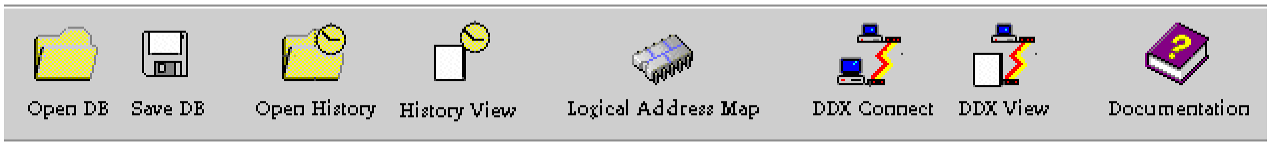


Figure 9: MemScope Toolbar

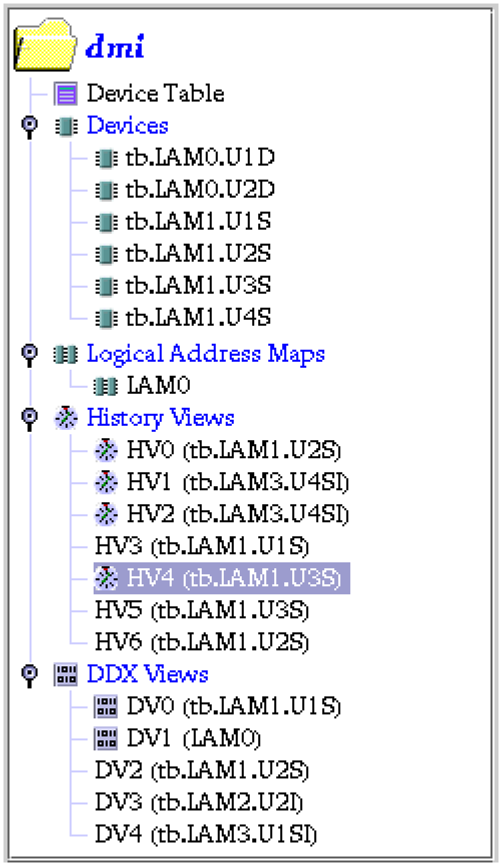
Note
The Toolbar in Figure 9 has “Show Toolbar Text” enabled (see page 60).

Table 10: Toolbar Buttons

This Toolbar button....	Works like this menu command....
	File > Open Database
	File > Save Database
	File > Open History
	View > Open History View
	View > Create Logical Address Map
	File > Open DDX Connection
	View > Open DDX View
	Help > MemScope User’s Manual

Hierarchy Browser

MemScope uses a hierarchical display called the Hierarchy Browser to show all memory model instances in your design, along with the logical address maps, History Views, and DDX Views that you define.



- dmi
  - Device Table
  - Devices
    - tb.LAM0.U1D
    - tb.LAM0.U2D
    - tb.LAM1.U1S
    - tb.LAM1.U2S
    - tb.LAM1.U3S
    - tb.LAM1.U4S
  - Logical Address Maps
    - LAM0
  - History Views
    - HV0 (tb.LAM1.U2S)
    - HV1 (tb.LAM3.U4SI)
    - HV2 (tb.LAM3.U4SI)
    - HV3 (tb.LAM1.U1S)
    - HV4 (tb.LAM1.U3S)
    - HV5 (tb.LAM1.U3S)
    - HV6 (tb.LAM1.U2S)
  - DDX Views
    - DV0 (tb.LAM1.U1S)
    - DV1 (LAM0)
    - DV2 (tb.LAM1.U2S)
    - DV3 (tb.LAM2.U2I)
    - DV4 (tb.LAM3.U1SI)

Figure 10: Hierarchy Browser

You can display any element in the Hierarchy Browser simply by clicking that element. A window for the element then appears in the WorkSpace.

Hierarchy Browser Pop-Up Menus

The Hierarchy Browser includes context-sensitive pop-up menus that provide quick access to commands you will commonly use in the Browser. Each menu also contains a Help command for the menu.

- **Device Table pop-up menu** — This appears when you right-click the Device Table icon in the Hierarchy Browser.

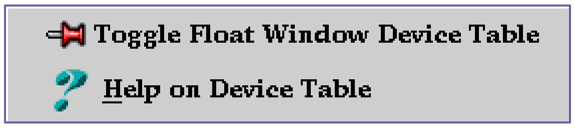


Figure 11: Device Table Pop-Up Menu

  - **Toggle Float Window Device Table** displays the Device Table as a separate, free-floating window, or displays it within the WorkSpace. (This command appears only when “Show Windows Contained” is checked in the Windows pull-down menu — see page 61.)

- **Device pop-up menu** — This appears when you right-click a device in the Devices list.

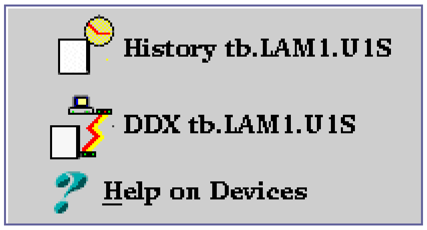


Figure 12: Device Pop-Up Menu

  - **History *device name*** creates a History View for the device. (This is greyed out if no history file is open.)
  - **DDX *device name*** creates a DDX View for the device. (This appears as “No Simulation Host” if you are not connected to a host.)

- **Logical Address Map pop-up menu** — This appears when you right-click a logical address map in the list.

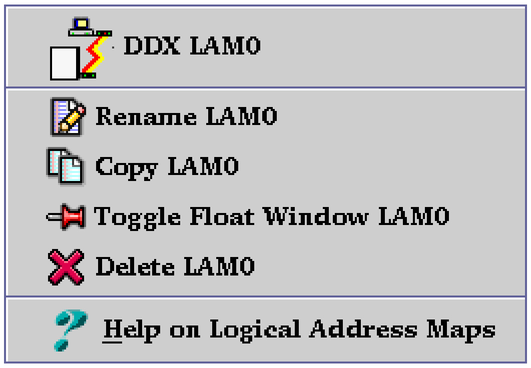


Figure 13: Logical Address Map Pop-Up Menu

  - **DDX *map name*** creates a DDX View for the logical address map. (This item appears as "No Simulation Host" if you are not connected to a host.)
  - **Rename *map name*** opens a dialog that lets you rename the map.
  - **Copy *map name*** opens a dialog so you can create a duplicate of the map under a new name.
  - **Toggle Float Window *map name*** displays the map as a separate, free-floating window, or displays it within the WorkSpace. (This command appears only when "Show Windows Contained" is checked in the Windows pull-down menu — see "View Menu" on page 59.)
  - **Delete *map name*** closes the map (if open) and deletes it from the database.

- **History Views pop-up menu** — This appears when you right-click a History View in the list.

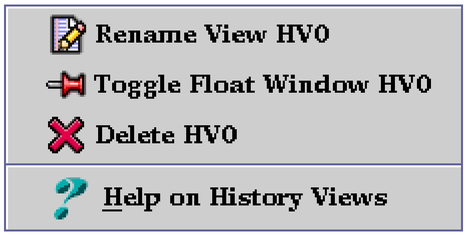


Figure 14: History Views Pop-Up Menu

  - **Rename *view name*** opens a dialog that lets you rename the History View.
  - **Toggle Float Window *view name*** displays the History View as a separate, free-floating window, or displays it within the WorkSpace. (This command appears only when “Show Windows Contained” is checked in the Windows pull-down menu — see “View Menu” on page 59.)
  - **Delete *view name*** closes the History View (if open), but does not delete the history file.

- **DDX Views pop-up menu** — This appears when you right-click a DDX View in the list.

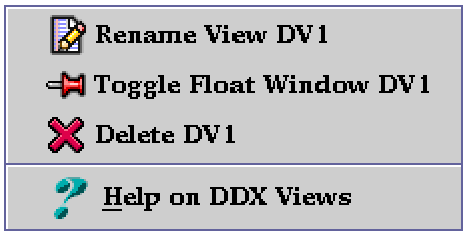


Figure 15: DDX Views Pop-Up Menu

  - **Rename *view name*** opens a dialog that lets you rename the DDX View.
  - **Toggle Float Window *view name*** displays the DDX View as a separate, free-floating window, or displays it within the WorkSpace. (This command appears only when “Show Windows Contained” is checked in the Windows pull-down menu — see “View Menu” on page 59.)
  - **Delete *view name*** closes the DDX View (if open), and deletes it from the database.

- **WorkSpace pop-up menu** — This appears when you right-click anywhere in the WorkSpace.

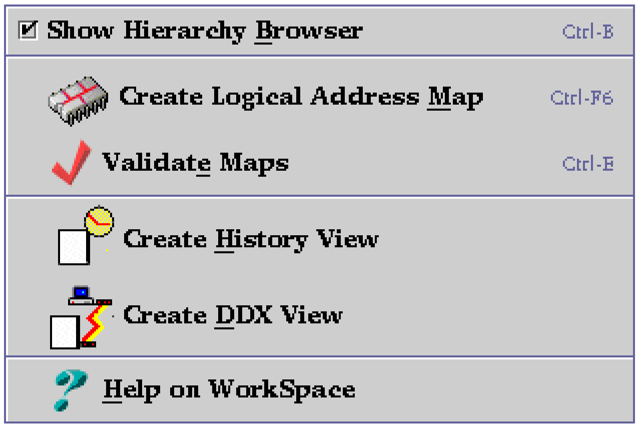


Figure 16: WorkSpace Pop-Up Menu

  - **Show Hierarchy Browser** displays or hides the Hierarchy Browser.
  - **Create Logical Address Map** starts the process of creating a logical address map — see “Working with Logical Address Maps” on page 81.
  - **Validate Maps** checks all open logical address maps for internal consistency and correctness — see “Working with Logical Address Maps” on page 81.
  - **Create History View** creates a History View of a selected device, or opens a saved view — see “Examining Transaction History Files” on page 78.
  - **Create DDX View** creates a Dynamic Data Exchange (DDX) View of a selected device, or opens a saved view — see “Viewing and Editing Memory Data During Simulation” on page 106.

Getting Help

MemScope provides extensive context-sensitive help throughout its interface. To get help for a particular item, simply right-click on it. (If context-sensitive help is not available, you get the online help Contents page.)

Starting MemScope

This section describes how to configure your testbench to run with MemScope, and how to invoke MemScope itself.

Step 1 — Prepare Your Testbench for MemScope

Before you can use MemScope's features, you first need to modify your testbench to create two files during simulation:

- **Database file** (.mpd) — contains data about each memory model instance in your design.
- **History file** (.mph) — records the transactions performed by each instance in the design.

To do this, edit your testbench as follows.

Note

The following example uses HDL testbench commands. For the corresponding commands in a C or VERA testbench, see the following:
• "HDL, VERA, and C Comparison" on page 115
• Chapter 6 on page 139 for C testbench information
• Chapter 7 on page 167 for VERA testbench information.

1. Insert a call to the database creation routine:

   mem_create_db (*filename*, *comment*, *status*)

   This command creates (or reads) a MemScope database, which stores device information on all memory model instances in the simulation. If the specified filename already exists, the database is read to get access to all models in the testbench.

   (For details about the command and a description of the command arguments, see "mem_create_db" on page 119.)

Note

Your testbench must contain one and only one mem_create_db command. Place it in an initial block following a delay statement.

2. Insert the command to begin the history logging routine:

   mem_begin_history (*filename*, *status*)

   This command creates a new history file or reopens an existing history file. The command also closes any currently open history file, and begins recording transactions on all memory model instances in the design. You can record history for the entire simulation run, or limit history recording to a specific time period to maximize performance.

   For details about the command and a description of the command arguments, see “mem_begin_history” on page 118.

3. Insert a command to end the history logging routine:

   mem_end_history (*status*)

   This command stops recording transactions and closes the history file. For details about the command and a description of the command arguments, see “mem_end_history” on page 122.

Note

Each mem_begin_history command in your testbench must have a matching mem_end_history command later in the testbench, or an empty history file will be generated.

You can also create multiple successive history files in a single simulation.

Your design simulation runs will now automatically create the database and history files needed by MemScope.

4. Run your simulation at least once to generate the database and history files.

Note

If you plan to view memory data during the simulation, you must also enable Dynamic Data Exchange before starting the simulator — see “Viewing and Editing Memory Data During Simulation” on page 106.

Step 2 — Invoke MemScope

To invoke MemScope, do one of the following according to your platform:

- On UNIX workstations, enter the following:

  ```
  $LMC_HOME/bin/memscope [database]
  ```

- From a command window, enter the following:

  ```
  [%LMC_HOME%\bin\]memscope [database]
  ```

The initial display then appears as shown in Figure 17.

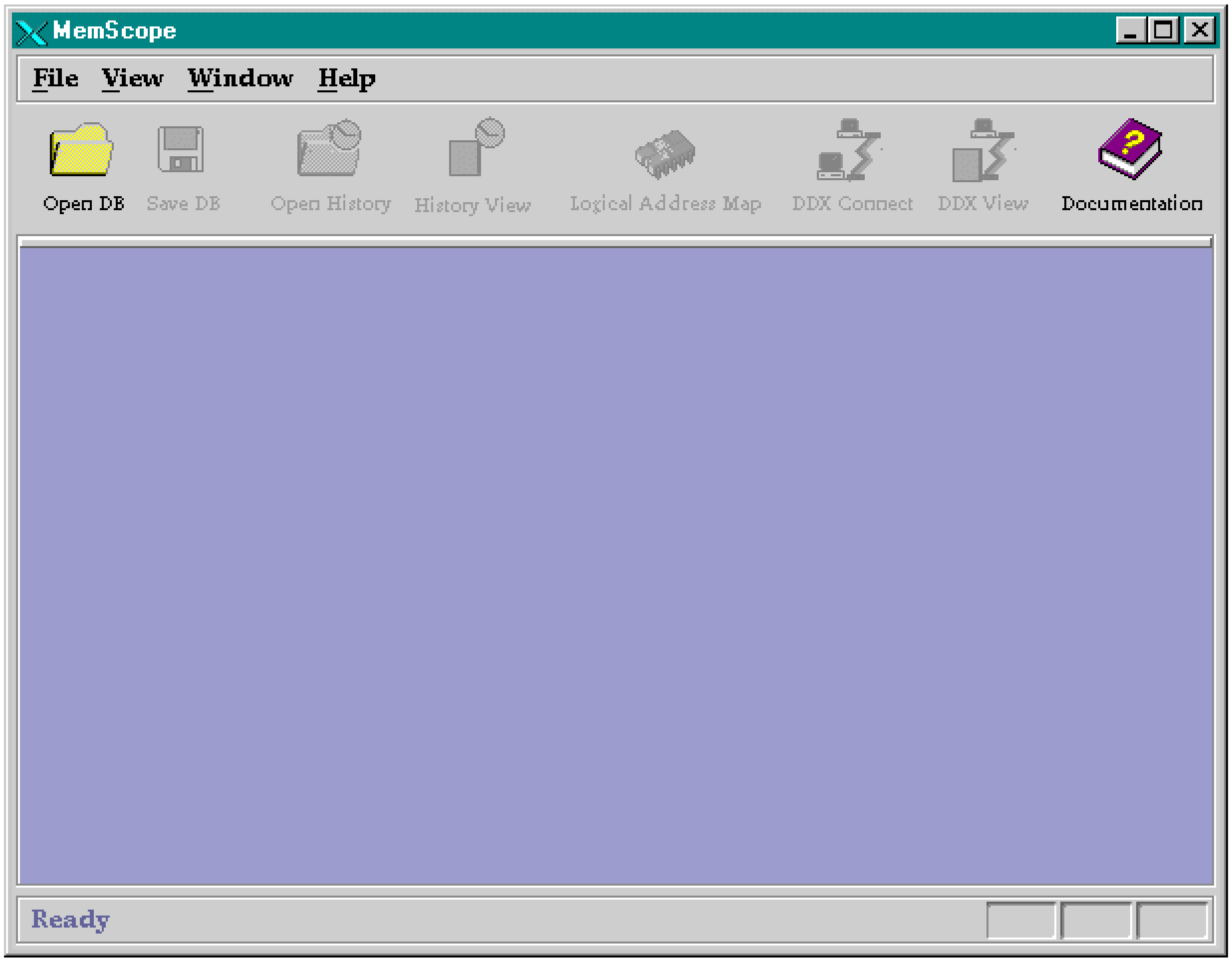


Figure 17: MemScope Initial Display

Step 3 — Open a Design Database and History File

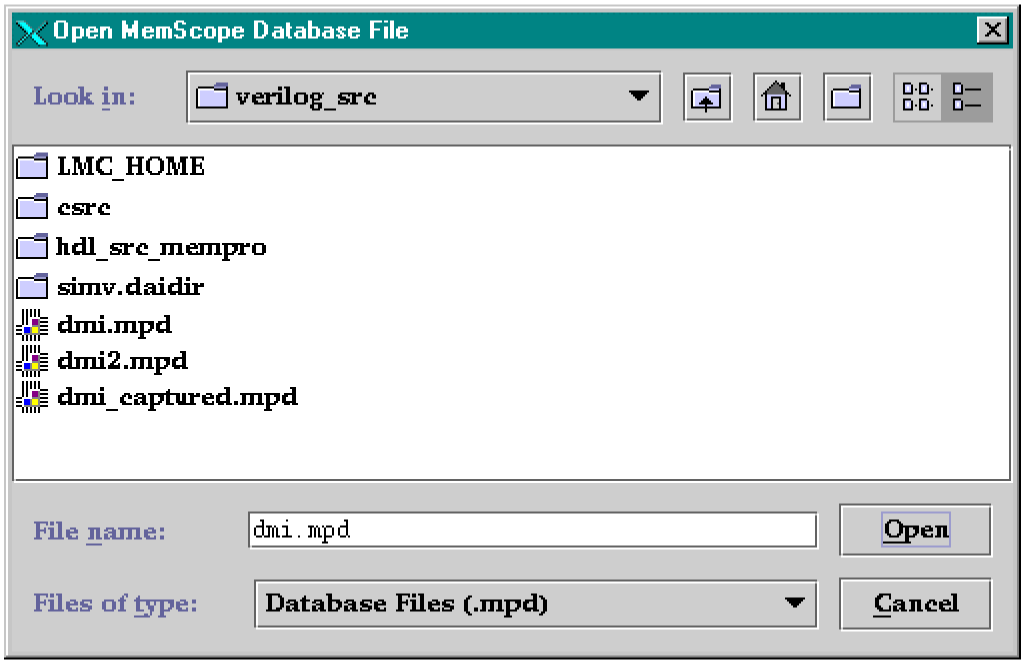
Once you’ve invoked MemScope, you need to load the database file (.mpd) and history file (.mph) for the design you want to work with.

Note

The database and history file are loaded automatically if you specify a database as a command line argument when you start MemScope.

1. To open a database, select **File** > **Open Database** from the pull-down menu, or click the Open DB button on the Toolbar.

   The dialog shown below then appears, listing database (.mpd) files by default.



LMC_HOME
csrc
hdl_src_mempro
simv.daidir
dmi.mpd
dmi2.mpd
dmi_captured.mpd

File name: dmi.mpd

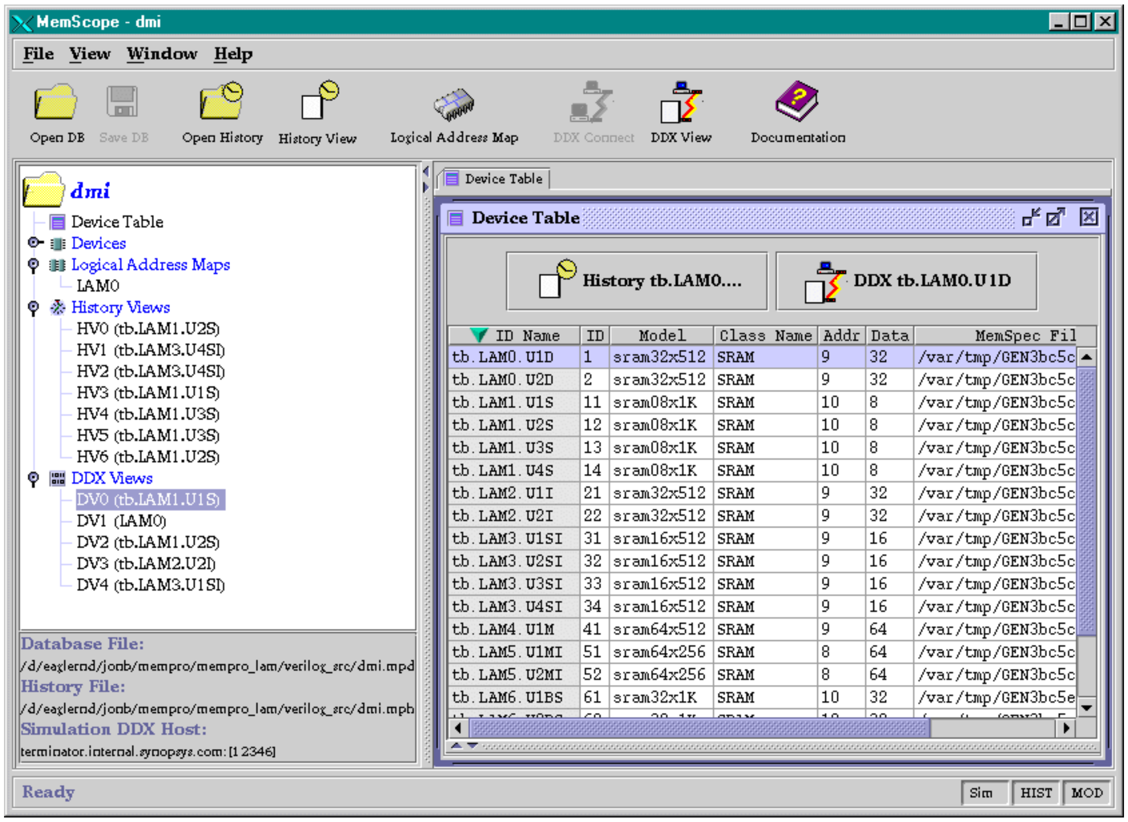
Files of type: Database Files (.mpd)

Figure 18: Open MemScope Database Dialog

Note

When you open a database, MemScope also automatically opens any history file that has the same name (not including the extension).

You can also manually open another history file if you prefer. However, you cannot load a history file until after you have loaded the associated database.



dmi
- Device Table
- Devices
- Logical Address Maps
  - LAM0
- History Views
  - HV0 (tb.LAM1.U2S)
  - HV1 (tb.LAM3.U4SI)
  - HV2 (tb.LAM3.U4SI)
  - HV3 (tb.LAM1.U1S)
  - HV4 (tb.LAM1.U3S)
  - HV5 (tb.LAM1.U3S)
  - HV6 (tb.LAM1.U2S)
- DDX Views
  - DV0 (tb.LAM1.U1S)
  - DV1 (LAM0)
  - DV2 (tb.LAM1.U2S)
  - DV3 (tb.LAM2.U2I)
  - DV4 (tb.LAM3.U1SI)

Database File:
/d/eaglernd/jonb/mempro/mempro_lam/verilog_src/dmi.mpd
History File:
/d/eaglernd/jonb/mempro/mempro_lam/verilog_src/dmi.mpb
Simulation DDX Host:
terminator.internal.synopsys.com: [12346]

ID Name	ID	Model	Class Name	Addr	Data	MemSpec Fil
tb.LAM0.U1D	1	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM0.U2D	2	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM1.U1S	11	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U2S	12	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U3S	13	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM1.U4S	14	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c
tb.LAM2.U1I	21	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM2.U2I	22	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c
tb.LAM3.U1SI	31	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U2SI	32	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U3SI	33	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM3.U4SI	34	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c
tb.LAM4.U1M	41	sram64x512	SRAM	9	64	/var/tmp/GEN3bc5c
tb.LAM5.U1MI	51	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c
tb.LAM5.U2MI	52	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c
tb.LAM6.U1BS	61	sram32x1K	SRAM	10	32	/var/tmp/GEN3bc5e

Figure 19: MemScope Main Window

2. If you need to open a history file, select **File** > **Open History** from the pull-down menu, or click the Open History button in the Toolbar.

When Using Database and History Files....

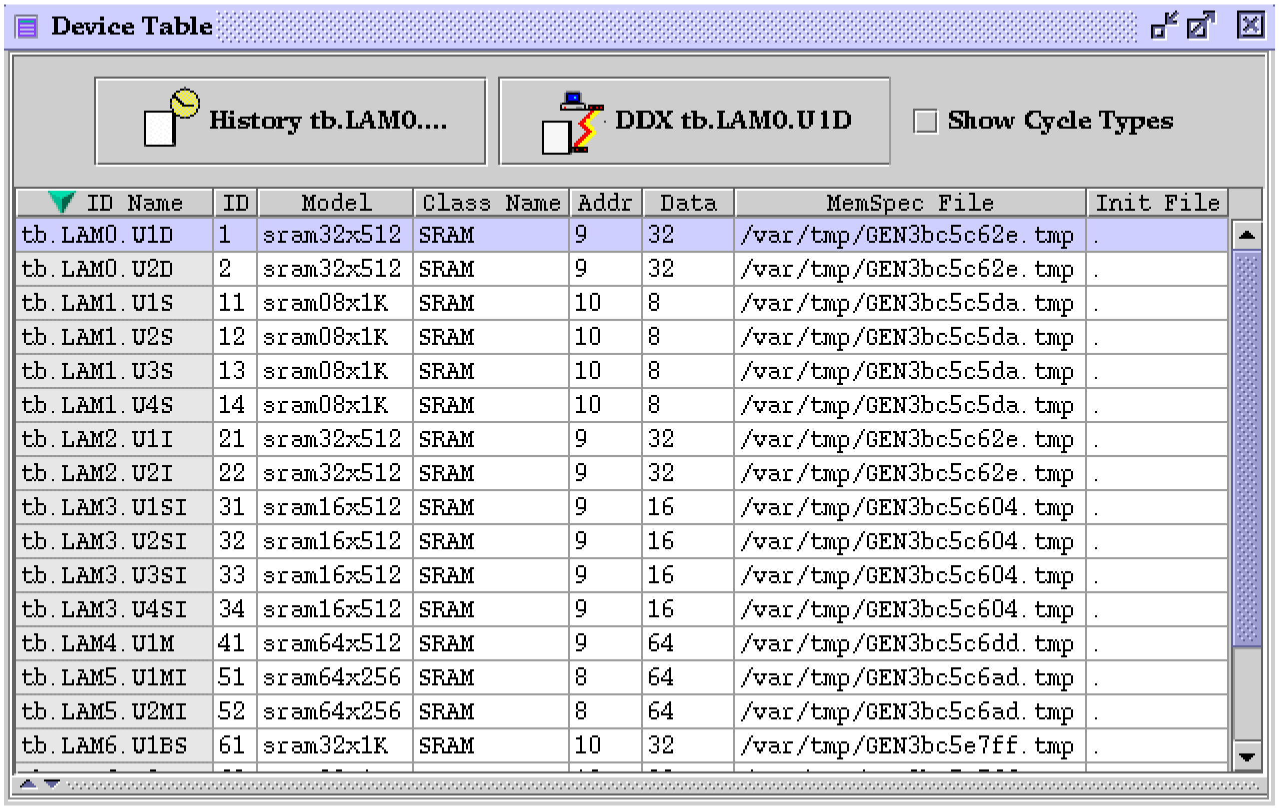
Keep the following in mind when you’re working with databases and history files in MemScope:

- The database and history file you open must be from the same simulation run.
- You cannot open a history file before opening a database. If you try, MemScope issues a warning and does not open the history file.
- When you open a database, MemScope closes any database and history file that were already open.

Device Views

When MemScope opens the database, it displays a Device Table that lists memory class, model, address and data widths, and other data about the devices in your design.

You can sort and search the information in the Device Table, as well as use it to create History Views and DDX Views for the listed devices.



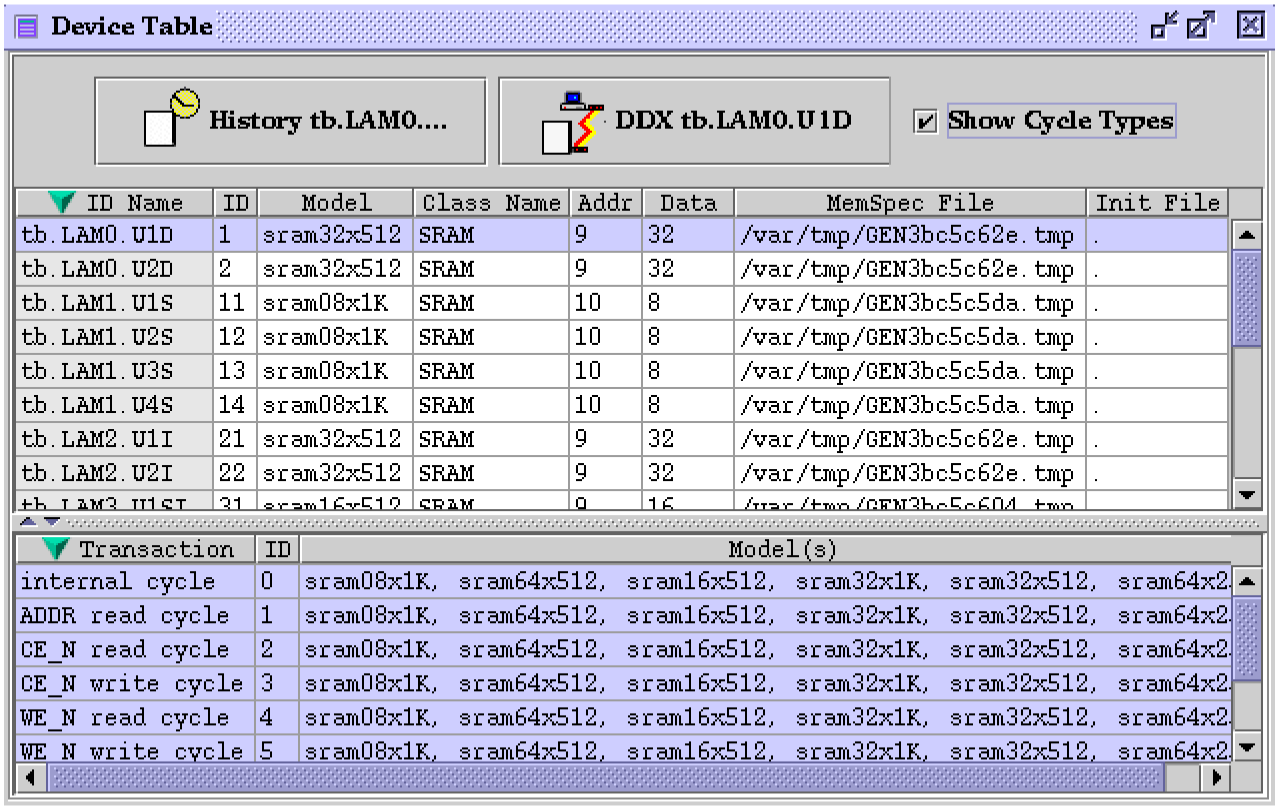
ID Name	ID	Model	Class Name	Addr	Data	MemSpec File	Init File
tb.LAM0.U1D	1	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM0.U2D	2	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM1.U1S	11	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U2S	12	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U3S	13	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U4S	14	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM2.U1I	21	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM2.U2I	22	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM3.U1SI	31	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c604.tmp	.
tb.LAM3.U2SI	32	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c604.tmp	.
tb.LAM3.U3SI	33	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c604.tmp	.
tb.LAM3.U4SI	34	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c604.tmp	.
tb.LAM4.U1M	41	sram64x512	SRAM	9	64	/var/tmp/GEN3bc5c6dd.tmp	.
tb.LAM5.U1MI	51	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c6ad.tmp	.
tb.LAM5.U2MI	52	sram64x256	SRAM	8	64	/var/tmp/GEN3bc5c6ad.tmp	.
tb.LAM6.U1BS	61	sram32x1K	SRAM	10	32	/var/tmp/GEN3bc5e7ff.tmp	.

Figure 20: Device Table

Whenever you select a device in the Hierarchy Browser's Devices list, the Device Table opens (if it wasn't previously) with the row for the selected device highlighted.

Displaying Cycle Type Information

You can display cycle type information in the Device Table by checking the Show Cycle Types box, as shown in Figure 21. When you select a device in the Device Table, the corresponding cycles are then highlighted in the cycle view.



ID Name	ID	Model	Class Name	Addr	Data	MemSpec File	Init File
tb.LAM0.U1D	1	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM0.U2D	2	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM1.U1S	11	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U2S	12	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U3S	13	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM1.U4S	14	sram08x1K	SRAM	10	8	/var/tmp/GEN3bc5c5da.tmp	.
tb.LAM2.U1I	21	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM2.U2I	22	sram32x512	SRAM	9	32	/var/tmp/GEN3bc5c62e.tmp	.
tb.LAM3.U1SI	31	sram16x512	SRAM	9	16	/var/tmp/GEN3bc5c604.tmp	

Transaction	ID	Model(s)
internal cycle	0	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2
ADDR read cycle	1	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2
CE_N read cycle	2	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2
CE_N write cycle	3	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2
WE_N read cycle	4	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2
WE_N write cycle	5	sram08x1K, sram64x512, sram16x512, sram32x1K, sram32x512, sram64x2

Figure 21: Device Table with Cycle Types

Sorting and Searching the Device Table

The Device Table provide several ways to sort the rows by column, and to find particular items within a column.

Sorting the Rows

You can sort the rows in the Device Table in two ways:

- Click the heading of the column you want to sort by. This highlights the column, and displays a sort indicator — ▼ or ▲ — to indicate ascending or descending order. To reverse the sort order, click the column heading again.
- Right-click the column you want to sort by. This opens a pop-up menu that lets you sort by the column heading, as well as search for particular items in the column. Figure 22 shows two of these pop-up menus.

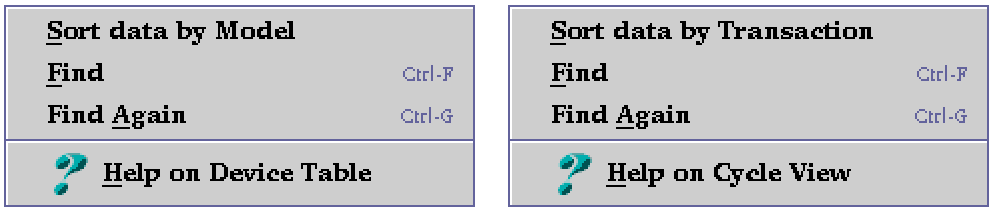


Figure 22: Device Table Pop-Up Menus

Finding Items in the Columns

Use the Find commands in the pop-up menus to find items in a selected column.

1. Select the column you want to search, or a particular cell in the column that contains a value you want to search for, and right-click. This displays a pop-up menu such as those shown in Figure 23.

   In each pair of menus shown, the upper menu appears when a *column* is selected, and the lower appears when a *cell* is selected (with the Find Down displaying the value in the cell).

   You can also use Ctrl-F and Ctrl-G to search through the device table, as shown in the menus.

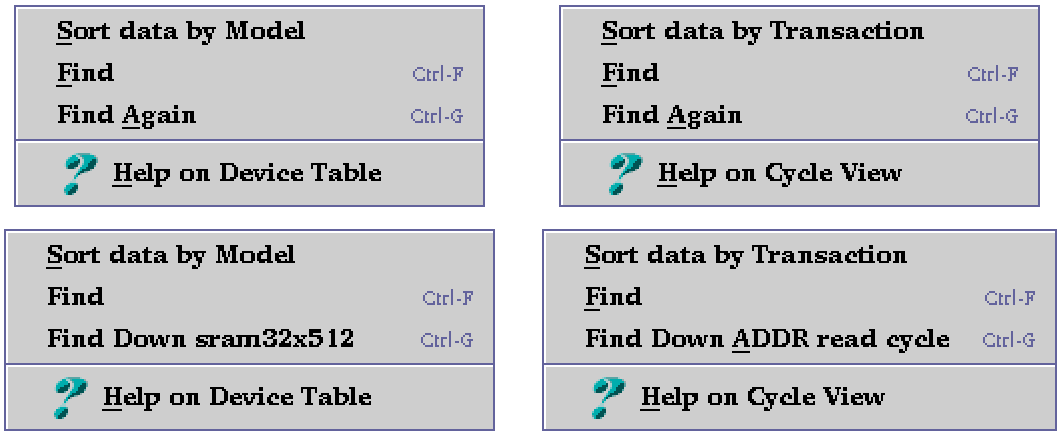
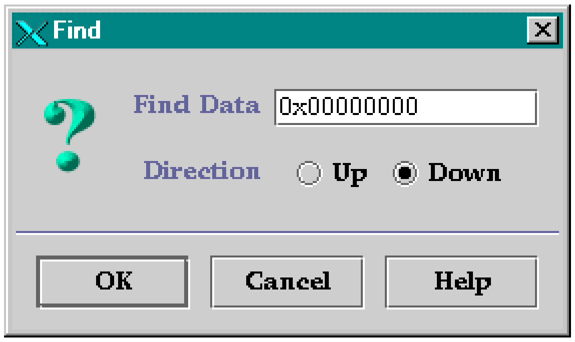


Figure 23: Device Table Pop-Up Menus with Find Commands

2. Select the Find, Find Again, or Find Down command:
   - ❍ **Find** opens a dialog such as that shown below that lets you search the column for a value you specify, or the value in the cell you selected.



Find Data: 0x00000000
Direction: Up [✓ Down]

Figure 24: Device View Find Dialog

   - ❍ **Find Again** repeats your last search.
   - ❍ **Find Down** appears when you selected a cell in the column, and finds the next occurrence of the value in that cell.

From Here On....

The remainder of this chapter describes how to use MemScope to perform its three primary debugging capabilities:

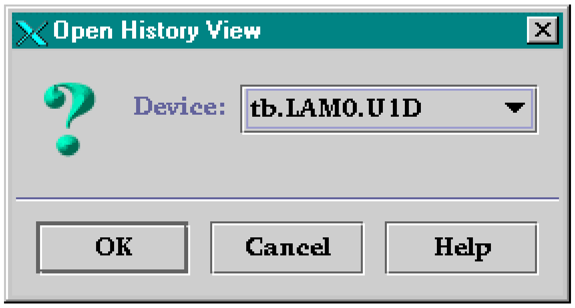
- Viewing transaction histories of the memory models in your simulation — see “Examining Transaction History Files” on page 78.
- Creating logical address maps of one or more of the physical memory devices in your simulation, which you can then use to view or modify the data stored in the system — see “Working with Logical Address Maps” on page 81.
- Connecting MemScope to a running simulation to view and modify memory contents dynamically — see “Viewing and Editing Memory Data During Simulation” on page 106.

Examining Transaction History Files

When your testbench is configured for MemScope, it automatically creates transaction histories for the memory models in your simulation. You can then use MemScope to create a "History View" of a particular device and examine its recorded transactions in detail.

To open a history file and create a History View for a particular device, do one of the following:

- Select a device in the Hierarchy Browser, then click the History View button on the Toolbar. (If you click the button without selecting a device, the dialog shown below opens to let you select a device from the pull-down menu.)

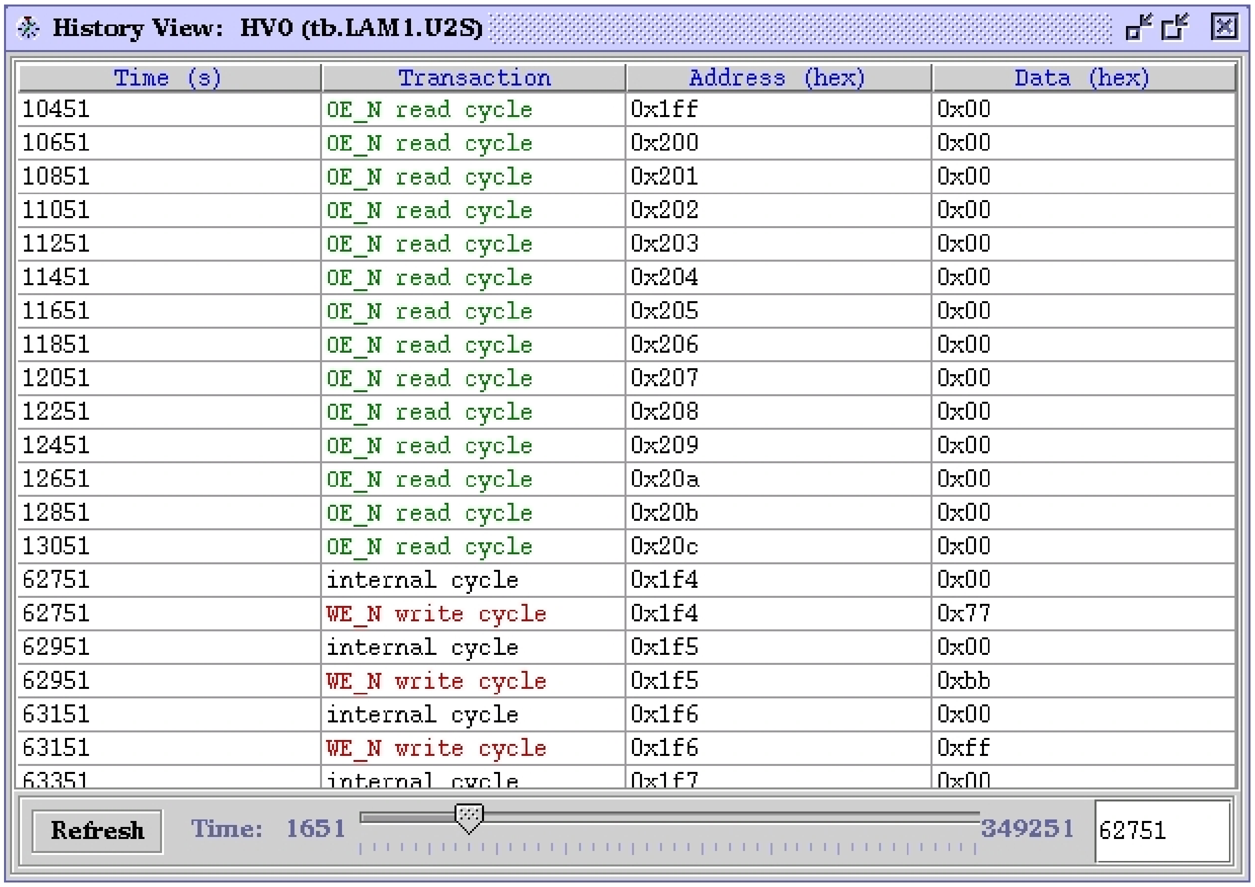


Device: tb.LAM0.U1D

Figure 25: Open History View Dialog

- Right-click a device in the Hierarchy Browser, then select History from the pop-up menu (see Figure 12 on page 65).
- Select a device in the Device Table, then click the History button in the table.

Each of these creates a History View for the device, such as that shown in Figure 26.



Time (s)	Transaction	Address (hex)	Data (hex)
10451	OE_N read cycle	0x1ff	0x00
10651	OE_N read cycle	0x200	0x00
10851	OE_N read cycle	0x201	0x00
11051	OE_N read cycle	0x202	0x00
11251	OE_N read cycle	0x203	0x00
11451	OE_N read cycle	0x204	0x00
11651	OE_N read cycle	0x205	0x00
11851	OE_N read cycle	0x206	0x00
12051	OE_N read cycle	0x207	0x00
12251	OE_N read cycle	0x208	0x00
12451	OE_N read cycle	0x209	0x00
12651	OE_N read cycle	0x20a	0x00
12851	OE_N read cycle	0x20b	0x00
13051	OE_N read cycle	0x20c	0x00
62751	internal cycle	0x1f4	0x00
62751	WE_N write cycle	0x1f4	0x77
62951	internal cycle	0x1f5	0x00
62951	WE_N write cycle	0x1f5	0xbb
63151	internal cycle	0x1f6	0x00
63151	WE_N write cycle	0x1f6	0xff
63351	internal cycle	0x1f7	0x00

Figure 26: History View

Once you’ve created and saved a History View, its name is listed in the Hierarchy Browser, and you can then re-open it by double-clicking.

Viewing a History File

A History View extracts the transaction history for a particular device from a history file, and displays the history as shown in Figure 26. The view includes the following information:

- **Time** — the simulator time at which the transaction occurred
- **Transaction** — the type of transaction that was performed. Reads are colored green, writes are red, and all others are black.
- **Address** — the memory address at which the transaction was performed.
- **Data** — the data written or read in the transaction.

You can browse the contents of the History View in several ways:

- Use the Page Up and Page Down keys to move one screen up or down.
- Use the up and down arrow keys to move one row up or down.
- Drag the slider at the bottom of the window.
- Type the time you want to view into the field in the lower-right corner of the window.

Changing the Time, Address, and Data Displays

You can select the units used to display the address, data, and time values in the History View by using pop-up menus. Right-click in the column you want to set, and select the units from the Format sub-menu:

- **Address** — select the display base: Hexadecimal, Octal, Decimal, or Binary.
- **Data** — select a number base: Hexadecimal, Octal, Decimal, or Binary.
- **Time** — select seconds (s), milliseconds (ms), microseconds (us), nanoseconds (ns), or picoseconds (ps).

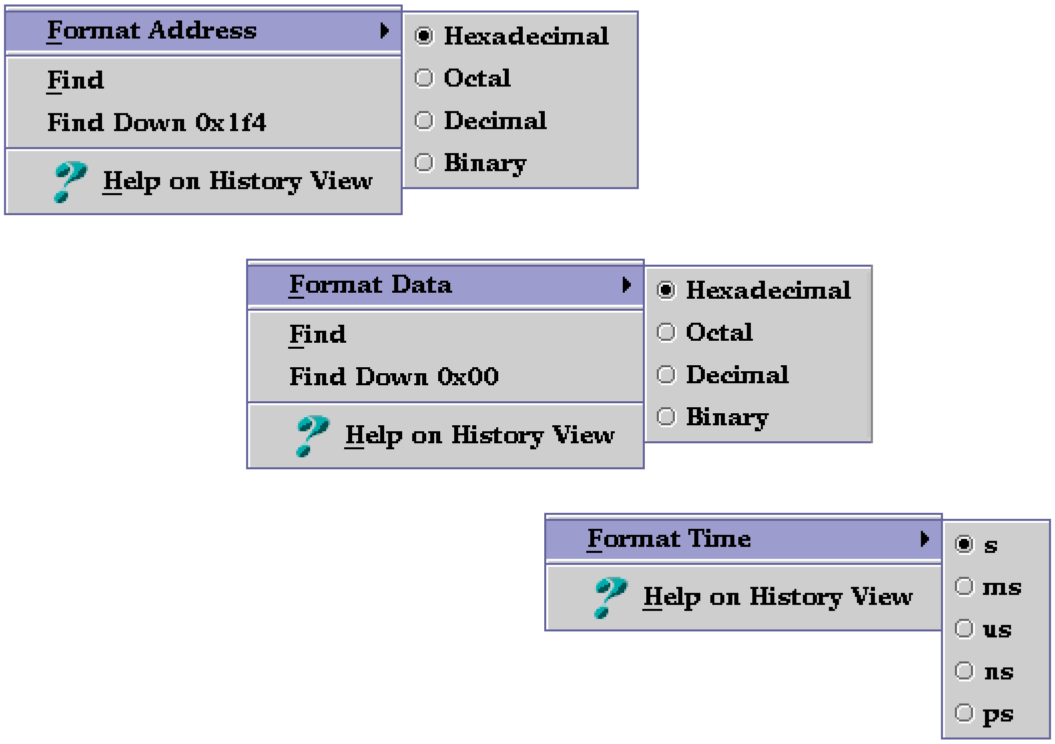


Figure 27: Format Address Pop-Up Menus

The unit you select is displayed in the column heading.

Working with Logical Address Maps

MemScope provides a capability called logical address mapping that enables you to create and view an abstract description of one or more of the memory devices in your simulation.

You can use logical address mapping to create a contiguous sequence of data words beginning at address zero, even though the physical implementation might have gaps or unmapped addresses in the map, especially if the memory data is distributed among two or more devices.

Once you have created a logical address map, you can use MemScope to observe or modify the data stored in the system as seen from the microprocessor's point of view.

Example of a Logical Address Map

Figure 28 shows a typical logical address map, in which two physical memory devices have been mapped together into a single logical address space. In this case, the map includes two devices:

- U1 has no offset, and stores the data for addresses 0 to 0xFFF.
- U2 has an offset of 0x1000, and stores data for addresses 0x1000 to 0x1FFF.

In the hardware, this mapping is implemented by connecting the lower 12 system address lines to the address lines of both devices, then connecting the 13th address line to each device's Chip Select pin. An inverter between $A_{[13]}$ and $\overline{CS}$ on U2 makes U1 active when $A_{[13]}$ is low, and U2 active when $A_{[13]}$ is high.

The data lines from the devices can be run together, since only one device can read or write at a time.

To map the memory system shown in Figure 28, you would add U1 to the logical address map with a base address of zero, using the device map type. You would then add U2 to the map with a base address of 0x1000 (hexadecimal) and the device type.

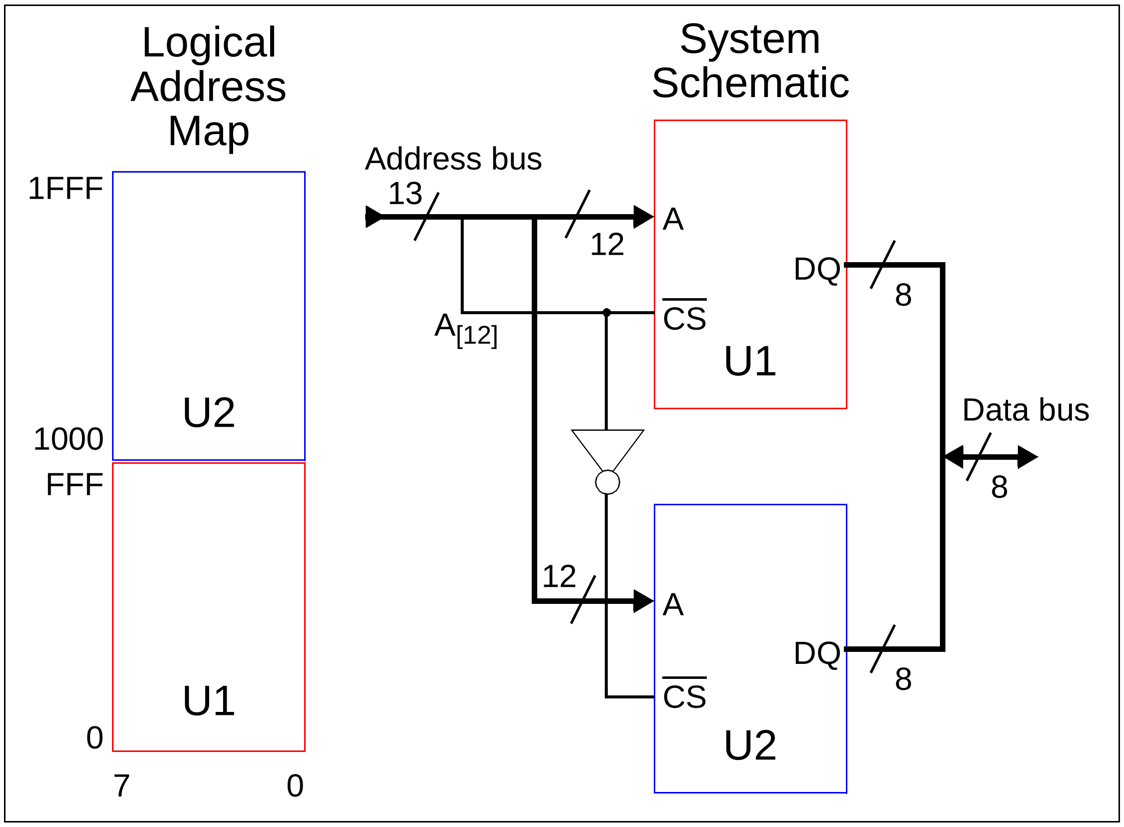


Figure 28: Logical Address Map and Corresponding Schematic Diagram

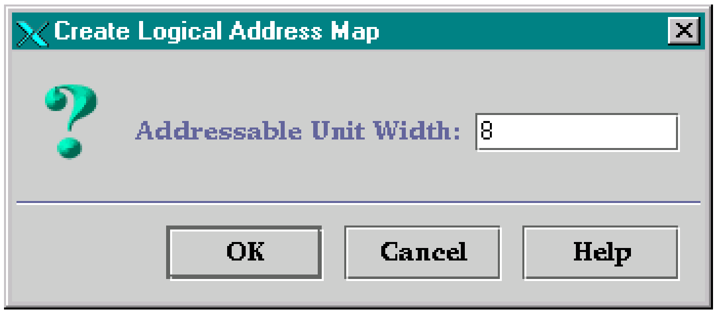
Creating a Logical Address Map

The following procedure describes how to create a logical address map for the devices in your simulation.

Step 1 — Open the Logical Address Map Window

1. Once your database is open, begin creating a logical address map by doing either of the following:
   - Select **View > Create Logical Address Map** from the pull-down menus.
   - Click the Logical Address Map button in the Toolbar.

   The dialog shown below then appears.



Addressable Unit Width: 8

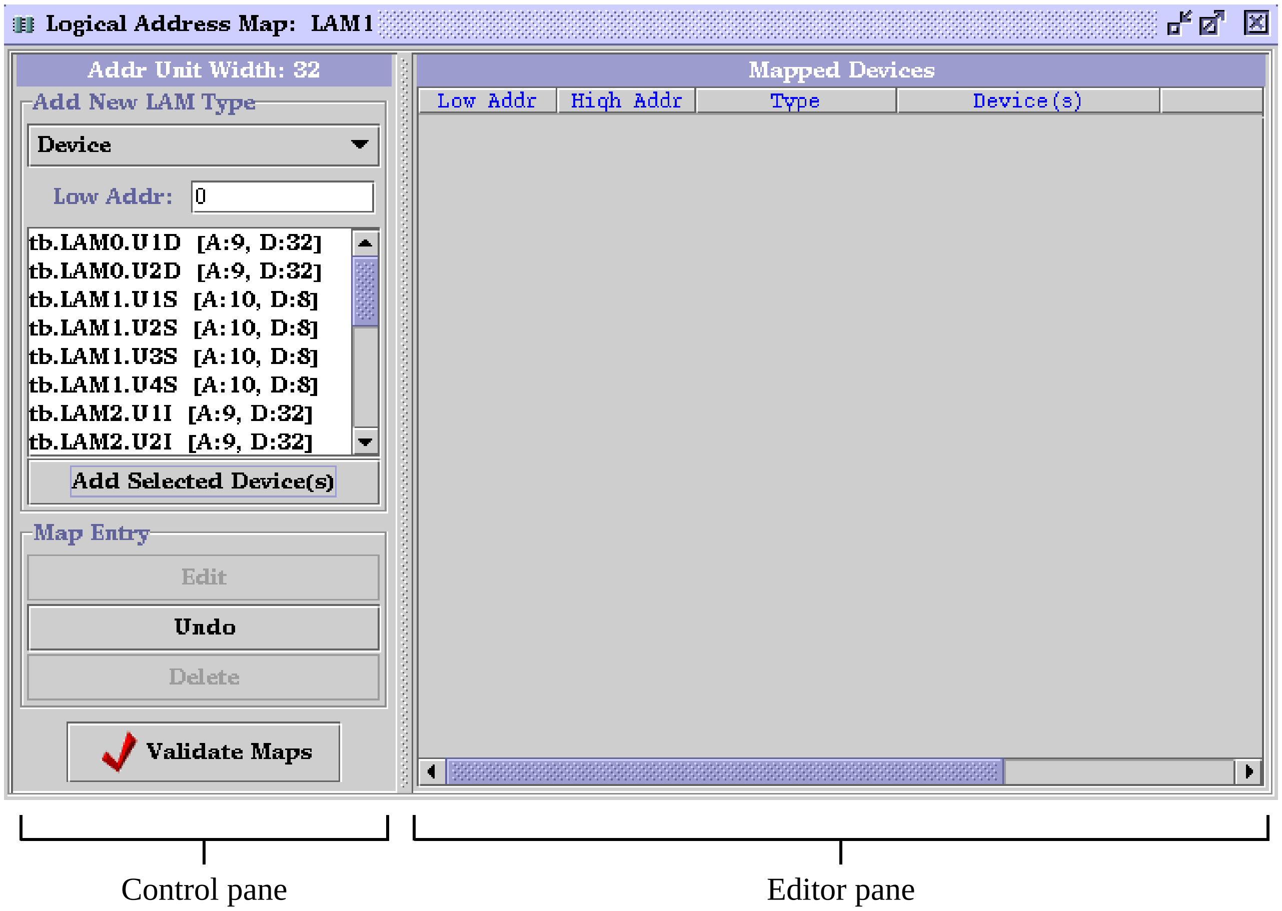
Figure 29: Create Logical Address Map Setup Dialog

2. Specify the map’s addressable unit width (the default is 8 bits).

Note

The addressable unit width can differ from the bus width. Many memory systems have 32-bit buses and might support 8-, 16-, 32-, and 64-bit transfers on the bus. However, if increasing the logical address value by one moves the storage referenced by 8 bits, then the addressable unit width is 8.

The Logical Address Map window then appears, as shown in Figure 30.



tb.LAM0.U1D [A:9, D:32]
tb.LAM0.U2D [A:9, D:32]
tb.LAM1.U1S [A:10, D:8]
tb.LAM1.U2S [A:10, D:8]
tb.LAM1.U3S [A:10, D:8]
tb.LAM1.U4S [A:10, D:8]
tb.LAM2.U1I [A:9, D:32]
tb.LAM2.U2I [A:9, D:32]

Figure 30: Initial Logical Address Map Window

The window is divided into two areas, as shown in Figure 30:

- **Control pane** — Use this to select the map type, address offset, and devices to add to the map, and to edit or delete devices you've added to the map. You can also filter the device list to make selection easier.
- **Editor pane** — Use this to move and edit devices in the map. When you create maps of certain types, this pane becomes a type-specific "Entry Editor" with additional controls for editing the devices in the entry.

Note also that the map displays the name and data width you specified.

Step 2 — Add Entries to the Logical Address Map

You’ll build the logical address map by specifying the map type and an address offset (if needed), and then adding entries for your memory devices. Each entry represents a range of addresses in the logical address space defined by the map.

1. Select the map type from the Add New LAM Type pull-down menu.

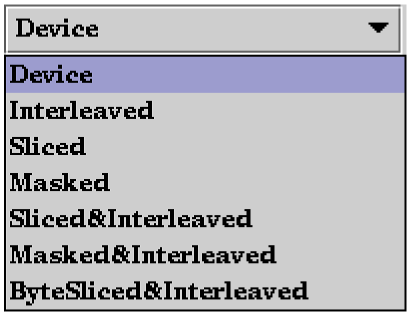


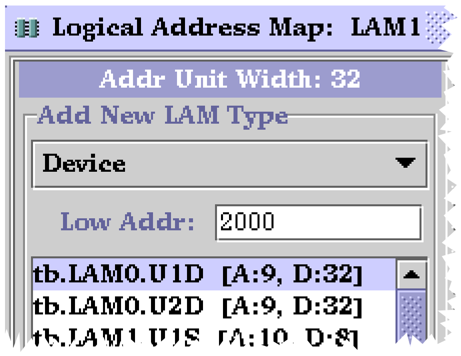
Figure 31: Add New LAM Type Menu

The menu includes the following map types:

- **Device** — Provides address offsets.
- **Interleaved** — Alternates memory addresses between two or more devices. Can include an address offset.
- **Sliced** — Splits memory data words between two or more devices, and can include an address offset.
- **Masked** — Allows sharing of two or more data words in a single device location (that is, a wide device data bus device supports a narrow system data bus), and can include an address offset.
- **Sliced and Interleaved** — Provides a combination of sliced and interleaved types to groups of four or more devices, and can include an address offset.
- **Masked and Interleaved** — Provides a combination of masked and interleaved types to groups of two or more devices, and can include an address offset.
- **Byte-Sliced and Interleaved** – Allows specific bytes to be enabled or disabled within a device, and can include an address offset. The addressable unit width is in bytes, and can span multiple devices.

For details about the map types, and how to select the appropriate type for modeling a particular physical device, see “Logical Address Map Types” on page 92.

2. If needed, add an address offset to the map by entering the offset value into the Low Addr field.



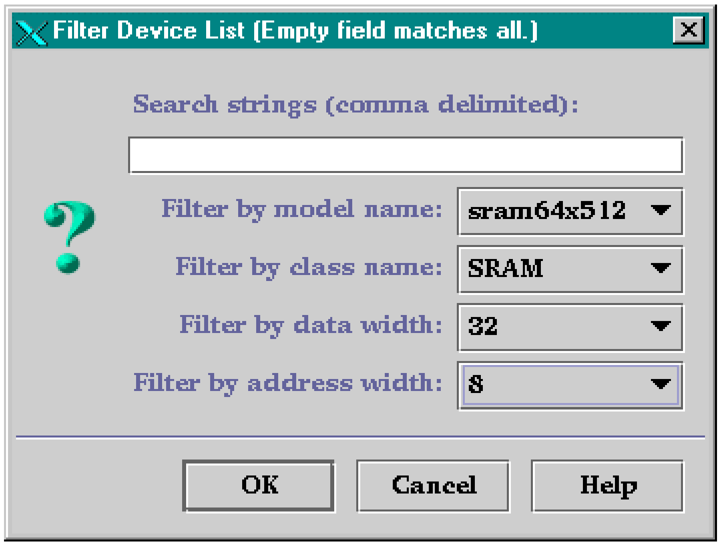
Low Addr: 2000

```
tb.LAM0.U1D  [A:9, D:32]
tb.LAM0.U2D  [A:9, D:32]
```

Figure 32: Address Offset Field

MemScope begins each new logical address map with zero as the default address offset for the first device or devices in each row, and keeps track of the offset as you add additional rows to the map.

3. To quickly select particular devices from the device list, you can right-click in the list to open the Filter Device List dialog. As shown below, the dialog provides several methods of filtering.

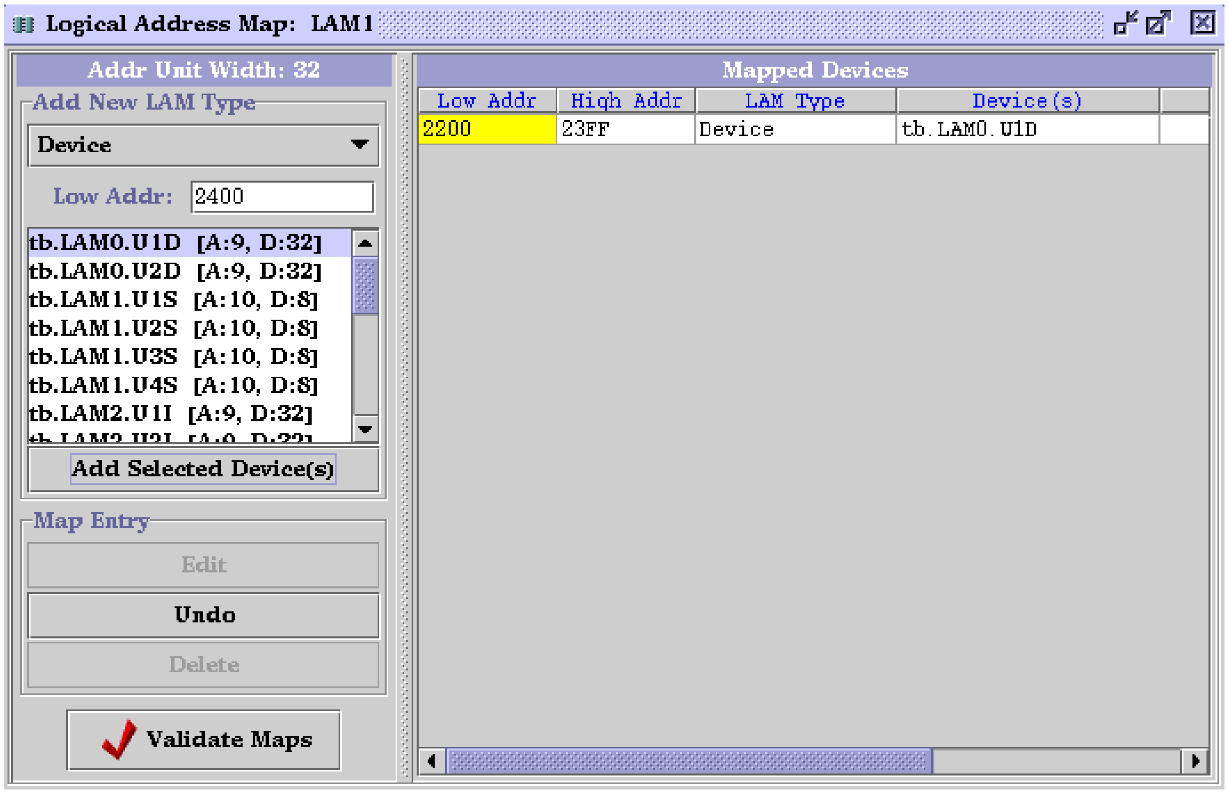

Filter Device List (Empty field matches all.)
Search strings (comma delimited):
OK
Cancel
Help

Filter by model name: sram64x512
Filter by class name: SRAM
Filter by data width: 32
Filter by address width: 8

Figure 33: Filter Device List Dialog

4. To add a device to the map, click the device in the device list, then click the Add Selected Device(s) button. The device then appears as an entry in the map, as shown in Figure 34.

   You can also press and hold Ctrl or Shift to select multiple devices at once.



Low Addr: 2400

tb.LAM0.U1D [A:9, D:32]
tb.LAM0.U2D [A:9, D:32]
tb.LAM1.U1S [A:10, D:8]
tb.LAM1.U2S [A:10, D:8]
tb.LAM1.U3S [A:10, D:8]
tb.LAM1.U4S [A:10, D:8]
tb.LAM2.U1I [A:9, D:32]

Low Addr	High Addr	LAM Type	Device(s)
2200	23FF	Device	tb.LAM0.U1D

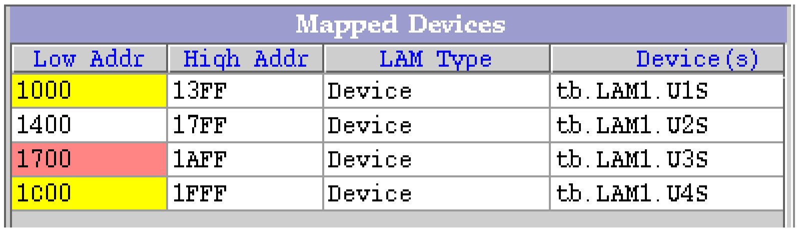
Figure 34: Adding a Device to the Logical Address Map

For certain map types, a type-specific Entry Editor window appears when you click Add Selected Device(s). Depending on the map type, the Entry Editor includes two or four buttons (such as ◀ and ▲) that you can use to rearrange the devices in the map. The buttons are active only when a device is selected.

5. Make additional entries as needed. As you do, you can modify the entries several ways:
   - To edit an entry (except in Device type maps), select the entry and click the Edit button, or double-click the entry’s LAM Type cell. The Entry Editor then opens.
   - To delete an entry from the map, select the entry, then click the Delete button.
   - To undo your last action, click the Undo button. (The button then changes to Redo, which reverses the action of the Undo button.)

Keep these guidelines in mind as you build the map:

- ❍ You can specify a different device type for each entry in the map.
- ❍ Each address in the map must belong to only one entry.
- ❍ You can specify gaps (unmapped addresses) between entries, but you cannot specify overlaps. Any gaps in the map are colored yellow, and overlaps are colored red, as shown below.



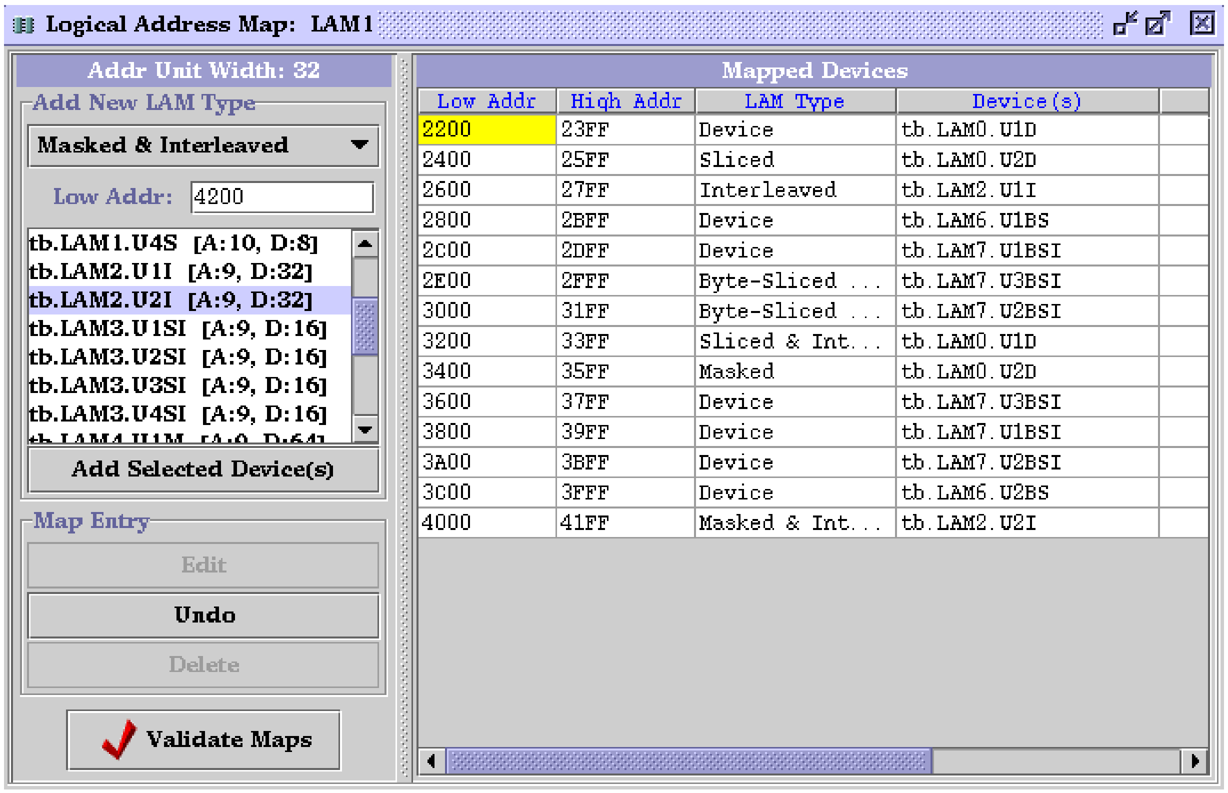
Mapped Devices

Low Addr	High Addr	LAM Type	Device(s)
1000	13FF	Device	tb.LAM1.U1S
1400	17FF	Device	tb.LAM1.U2S
1700	1AFF	Device	tb.LAM1.U3S
1C00	1FFF	Device	tb.LAM1.U4S

Figure 35: Address Gaps and Overlaps

- ❍ Order sliced devices so that the left-most slice (the first entry) corresponds to the left-hand portion of the associated data bus, and the right-most slice (the last entry) corresponds to the right-hand side of the data bus.
- ❍ If the map is byte-sliced and interleaved, the above left-to-right ordering also applies, though the constructed slice can also cover more than one address unit width.
- ❍ If a slice covers multiple addresses, add devices starting with the device at the highest address in the slice, and work your way down to the device at the lowest address in the slice.

6. When you finish building the map, click the Close Editor button (if present).



Low Addr: 4200

```
tb.LAM1.U4S  [A:10, D:8]
tb.LAM2.U1I  [A:9, D:32]
tb.LAM2.U2I  [A:9, D:32]
tb.LAM3.U1SI  [A:9, D:16]
tb.LAM3.U2SI  [A:9, D:16]
tb.LAM3.U3SI  [A:9, D:16]
tb.LAM3.U4SI  [A:9, D:16]
```

Low Addr	High Addr	LAM Type	Device(s)
2200	23FF	Device	tb.LAM0.U1D
2400	25FF	Sliced	tb.LAM0.U2D
2600	27FF	Interleaved	tb.LAM2.U1I
2800	2BFF	Device	tb.LAM6.U1BS
2C00	2DFF	Device	tb.LAM7.U1BSI
2E00	2FFF	Byte-Sliced ...	tb.LAM7.U3BSI
3000	31FF	Byte-Sliced ...	tb.LAM7.U2BSI
3200	33FF	Sliced & Int...	tb.LAM0.U1D
3400	35FF	Masked	tb.LAM0.U2D
3600	37FF	Device	tb.LAM7.U3BSI
3800	39FF	Device	tb.LAM7.U1BSI
3A00	3BFF	Device	tb.LAM7.U2BSI
3C00	3FFF	Device	tb.LAM6.U2BS
4000	41FF	Masked & Int...	tb.LAM2.U2I

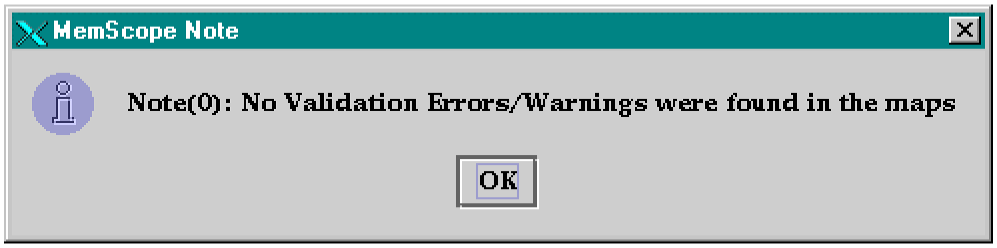
Figure 36: Completed Logical Address Map

Note

When you modify a logical address map, any DDX Views you've created of that map will also be affected, so you'll need to run your simulation again. (For details about DDX Views, see "Viewing and Editing Memory Data During Simulation" on page 106.)

7. Click the Validate Maps button. You must validate the map before your simulation will be able to run using the database that contains the map.

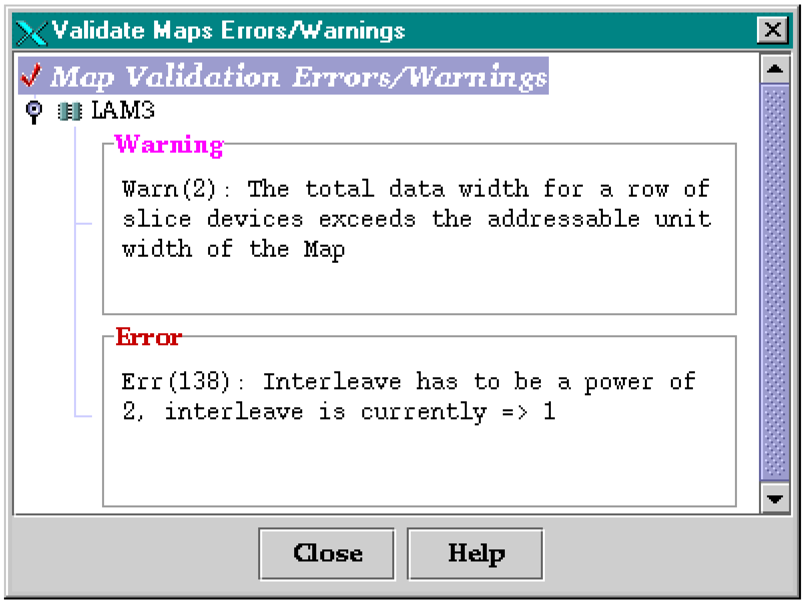
   If the map has no errors, a message like that shown below appears:



Note(0): No Validation Errors/Warnings were found in the maps

Figure 37: No Validation Error Message

If the map does contain errors, a message similar to that shown below appears:



Warn(2): The total data width for a row of slice devices exceeds the addressable unit width of the Map

Err(138): Interleave has to be a power of 2, interleave is currently => 1

Figure 38: Validation Error Message

Using Logical Address Maps in Testbenches

During simulation, you'll use a testbench command to load your database. You can then use other testbench commands to load, dump, peek at, and poke the maps in the database.

Once you've defined a logical address map, you can substitute the name of the map for the name of a single instance in most HDL testbench commands or C testbench functions that have an instance argument. This means you can refer to any location in any device in a complex memory system through its physical *or* its logical address.

HDL testbench commands and C testbench functions that can use a logical address map include the following:

- Memory instance information C functions (no HDL equivalents):
  - "slm_find_instancebyname" on page 153
  - "slm_mem_instance_info" on page 158
- Memory image file HDL commands and C functions:
  - "mem_load" on page 125 (HDL) and "slm_mem_load" on page 160 (C)
  - "mem_dump" on page 120 (HDL) and "slm_mem_dump" on page 156 (C)
  - "mem_unload" on page 135 (HDL) and "slm_set_message_level" on page 163 (C)

- Memory location access HDL commands and C functions:
  - "mem_peek" on page 128 (HDL) and "slm_mem_peek" on page 161 (C)
  - "mem_poke" on page 129 (HDL) and "slm_mem_poke" on page 162 (C)
- Memory address tracing HDL commands (no C equivalents):
  - "mem_trace" on page 130
  - "mem_untrace" on page 136
- Model message control HDL commands and C functions:
  - "mem_msg_level" on page 126 (HDL) and "slm_set_message_level" on page 163 (C)
  - "slm_get_message_level" on page 154 (C: no HDL equivalent)

The general procedure for using a logical address map with testbench commands is as follows:

Note

The following assumes that you have modified your testbench to create a database and history file during simulation, as described in "Step 1 — Prepare Your Testbench for MemScope" on page 69.

1. Load the entire memory system at once from a single initialization file, by referring to the logical address map name in a call to mem_load.
2. Use the map name to make calls to other commands (such as mem_peek and mem_poke) to access data in any mapped device, without needing to worry about how addresses are translated.

Using a logical address map eliminates the need to split initialization data between multiple load files. You no longer have to calculate the address offset each time you want to peek at or poke data in the testbench.

Logical Address Map Types

To successfully use MemScope's logical address mapping features, you must choose a type for your map that matches your memory system's architecture.

This section discusses the available types of logical address maps, explains their use, and shows schematic diagrams of typical systems for each.

Device Type

Use the Device map type to offset the base address of the device so that it fits into the logical address map. In this type, the data width of each device must equal the addressable unit width of the map, and the addressable unit width, group width, and device width are all equal:

addressable unit width = group width = device width

In a hardware implementation, the Device type is usually represented by connecting the lower bits of the address bus directly to the device address input. Most designs use decode logic on the most significant address bits to drive the Chip Select input or inputs. The device data bus directly connects to the system memory bus.

Figure 39 shows an example of a schematic and corresponding logical address map that would use the Device type. In the example, the addressable unit width is 16 bits, and each device width is 16 bits. Although there are two 16-bit devices, only one is enabled at a time through the address decoding, so the group width is also 16.

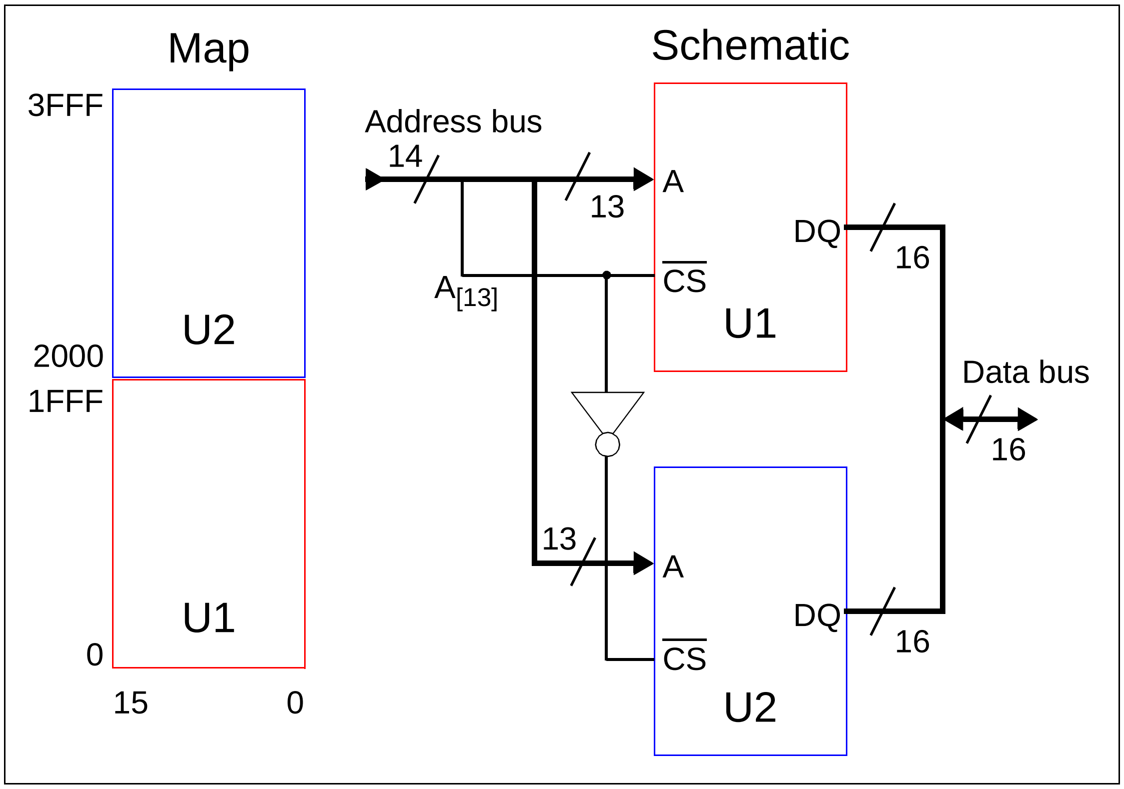


Figure 39: Device Map Type Design Example

Note
Device is the only map type that does not have its own specific Entry Editor.

Sliced Type

Use the Sliced map type when each data word in a logical address view is split between two or more physical devices (so that the data width of the device must be less than the addressable unit width of the map). In this type, the addressable unit width is equal to the group width, but greater than the device width by a factor of 2 or more:

addressable unit width = group width = (device width)* n (where n = 2, 3, 4, ...)

For example, if the system's addressable unit width is 8 bits, but it contains two 4-bit-wide devices accessed in parallel, you would use the Sliced type.

The example in Figure 40 shows the schematic of a memory space implemented in a sliced configuration, and the corresponding logical address map. The address and control lines of two or more memory devices are connected in parallel, and all devices in the group are reading and writing data using the same address. The data bus on each device in the group connects to a subset of the data lines for the complete system. In the example, the addressable unit width is 8 bits, the width of each device is 4 bits, and the group width is 4 * 2 = 8 bits (*n* = 2 in this case).

When you add devices to a Sliced type map, MemScope makes sure that all devices in the group have the same number of locations (that is, the same address bus width), and that the sum of all data bus widths is equal to the addressable unit width specified for the map.

Note
A Sliced map can include an address offset.

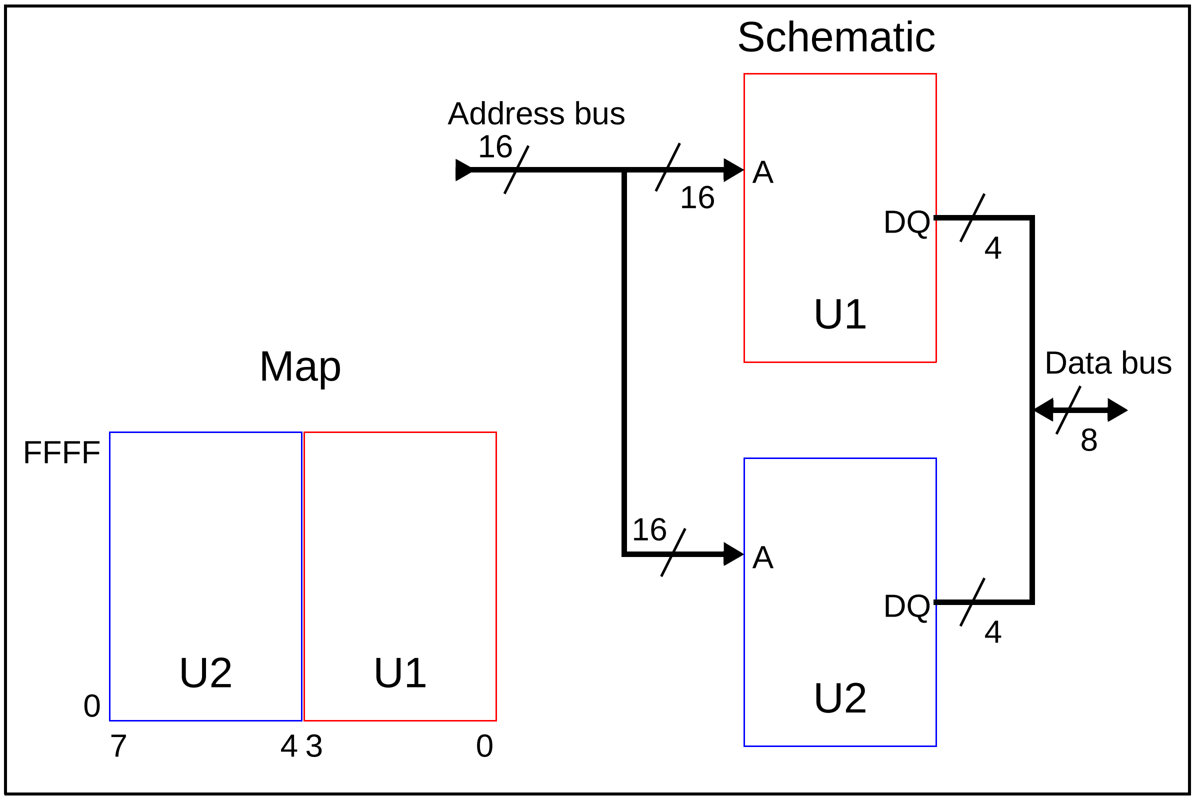


Figure 40: Sliced Map Type Design Example

Interleaved Type

Some memory systems alternate — or *interleave* — consecutive logical address locations between two or more memory devices. In the Interleaved map type you would use to map such a system, each device width must equal the addressable unit width of the map.

In the Interleaved map type, the addressable unit width is equal to the device width, which is less than the group width:

group width > addressable unit width = device width

In addition, all devices grouped together in an interleaved row of the map must have the same number of locations (address bus width), and the number of devices must be an integral power of two (2, 4, 8, 16, ...):

group width = (addressable unit width) * 2^n (n = 1, 2, ...)

In a hardware implementation, the least significant bits of the system address bus are decoded to derive the Chip Select pins for the devices in the group, as shown in Figure 41. The remaining address bits are connected to the address bus inputs of the devices. The device data buses are connected in parallel to the system data bus.

In the example, the addressable unit width is 8 bits, the width of each device is 8 bits, and the group width is 16 bits, so the relationship is satisfied if n = 1:

group width = (addressable unit width) * 2^1 = 16

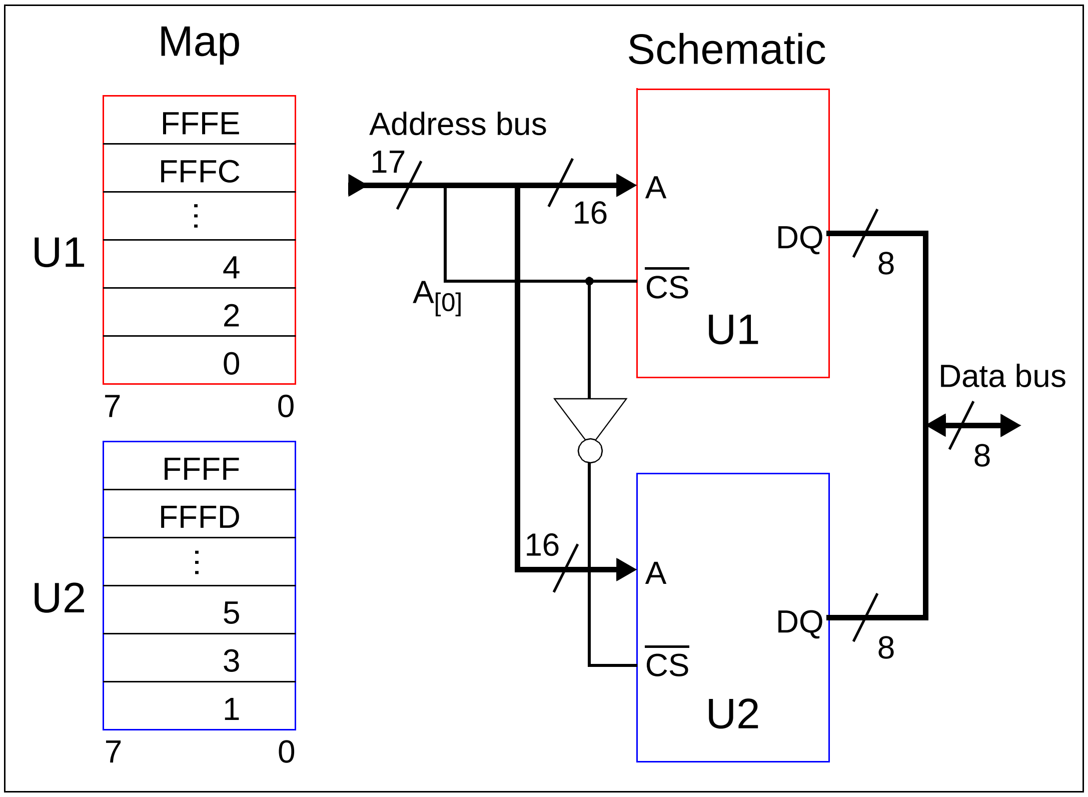


Figure 41: Interleaved Map Type Design Example

Note

An Interleaved map can include an address offset.

Sliced and Interleaved Type

Use the Sliced and Interleaved map type when the addressable unit width is less than the group width, but greater than the device width:

group width > addressable unit width > device width

and

addressable unit width = (device width) * n (where n = 2, 3, 4, ...)

In this type, consecutive logical addresses are alternated between two or more groups of devices.

All devices grouped in an interleaved row must have the same number of locations (address bus width). In addition, the group width must equal the addressable unit width times an integral power of two (2, 4, 8, 16, ...):

group width = addressable unit width * 2^k (k = 1, 2, ...)

The example in Figure 42 shows the schematic of a memory space physical implemented in a sliced and interleaved configuration, and the corresponding logical address map. In the example, the addressable unit width is 16 bits, the width of each device is 8 bits, and the group width is 16 bits, so the relationship is satisfied if k = 1:

group width = (addressable unit width) * 2^1 = 16

Notice that although the example has four 8-bit devices, the group width is 16 and not 32 because only two of the devices are enabled at a time. Devices U1 and U2 combine to form a data word with an even-numbered address, while devices U3 and U4 combine to form a data word with an odd-numbered address.

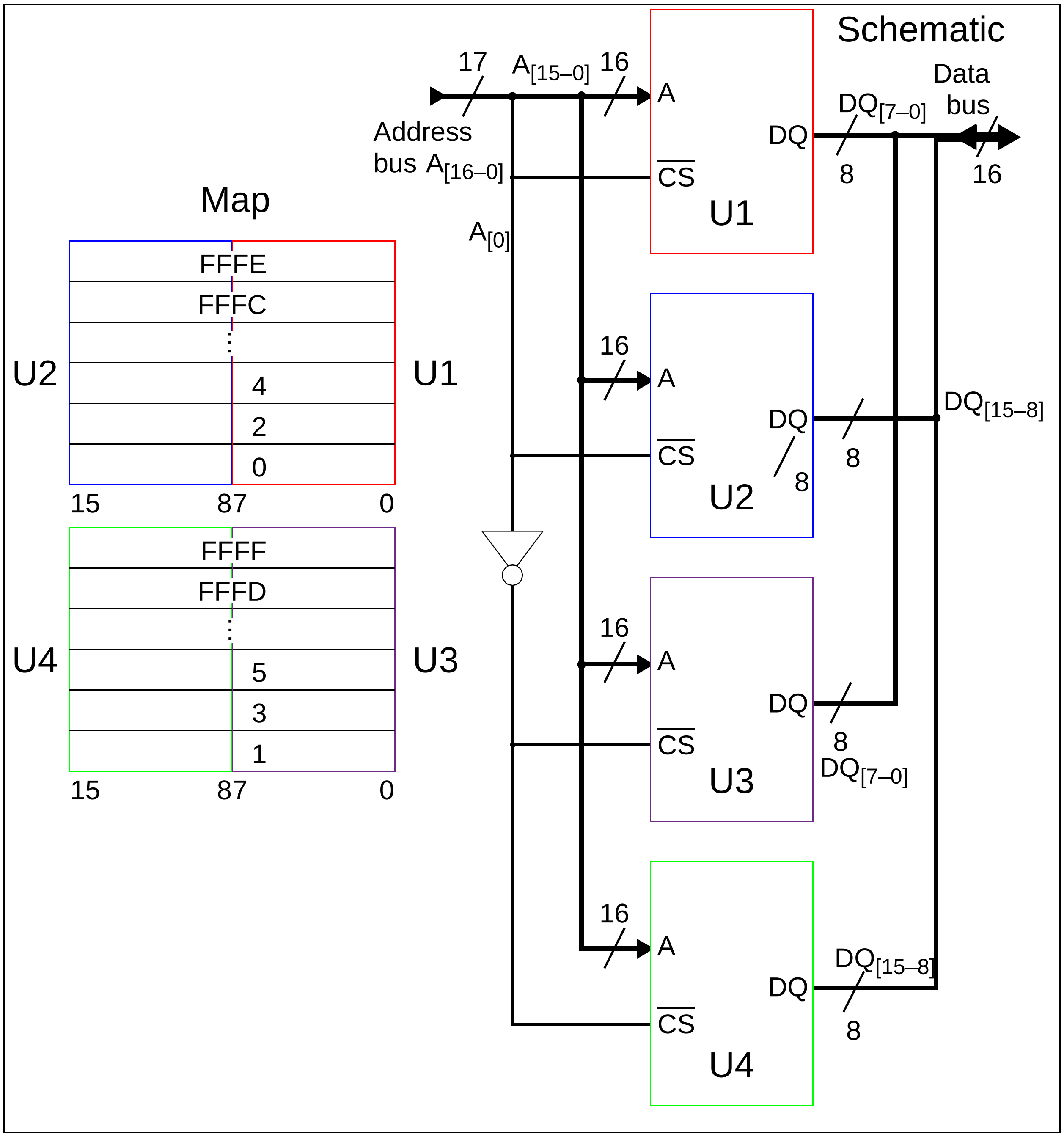


Figure 42: Sliced and Interleaved Map Type Design Example

Note
A Sliced and Interleaved map can include an address offset.

Masked Type

Use the Masked map type when each word in a memory device contains two or more logical words. In this type, the group width equals the device width, but is greater than the addressable unit width by a factor of 2 or more:

group width = device width = (addressable unit width) * n (where $n \geq 2$)

A common design technique is to use 32-bit memory chips to store byte-addressable data, so each word of the device stores four addressable units.

When you configure a masked row in your map, MemScope computes the number of mask elements based on the relationship between the device data width and the map addressable unit size.

In the example hardware implementation in Figure 43, the least significant address bit is used to control a multiplexer on the data bus. The remaining address bits are connected directly to the memory device. The addressable unit width is 8 bits, the device width is 16 bits, and the group width is 16 bits.

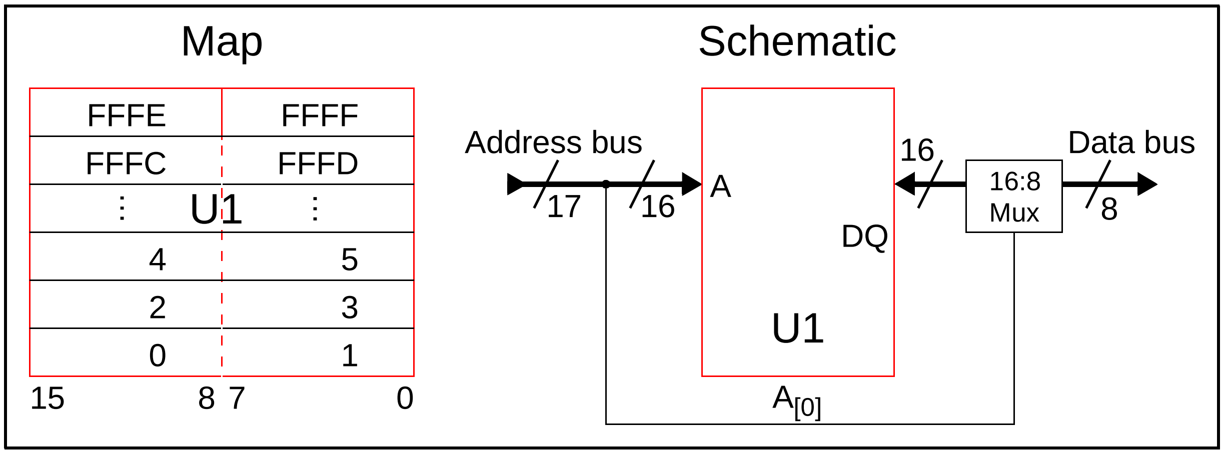


Figure 43: Masked Map Type Design Example

Note

A Masked map can include an address offset.

Masked and Interleaved Type

Use the Masked and Interleaved map type when both the following are true:

- The addressable unit width of the map is smaller than the device data width.
- Two or more devices take alternate sequences of address values.

In this map type, the group width is greater than the device width, which in turn is greater than the addressable unit width by a factor of 2 or more:

group width > device width = (addressable unit width) * n (where $n \geq 2$)

In addition, all devices grouped in an interleaved row must have the same number of locations (address bus width), and the number of devices must be an integral power of two (2, 4, 8, 16, etc.):

group width = (addressable unit width) * 2^k (k = 1, 2, ...)

The example in Figure 44 shows two interleaved devices with two-way masking. The logical address values are shown superimposed on data word locations within each device. In this example, the addressable unit width is 8 bits, the width of each device is 16 bits, and the group width is 32 bits, so the relationship is satisfied if $k = 2$:

group width = (addressable unit width) * $2^2 = 32$

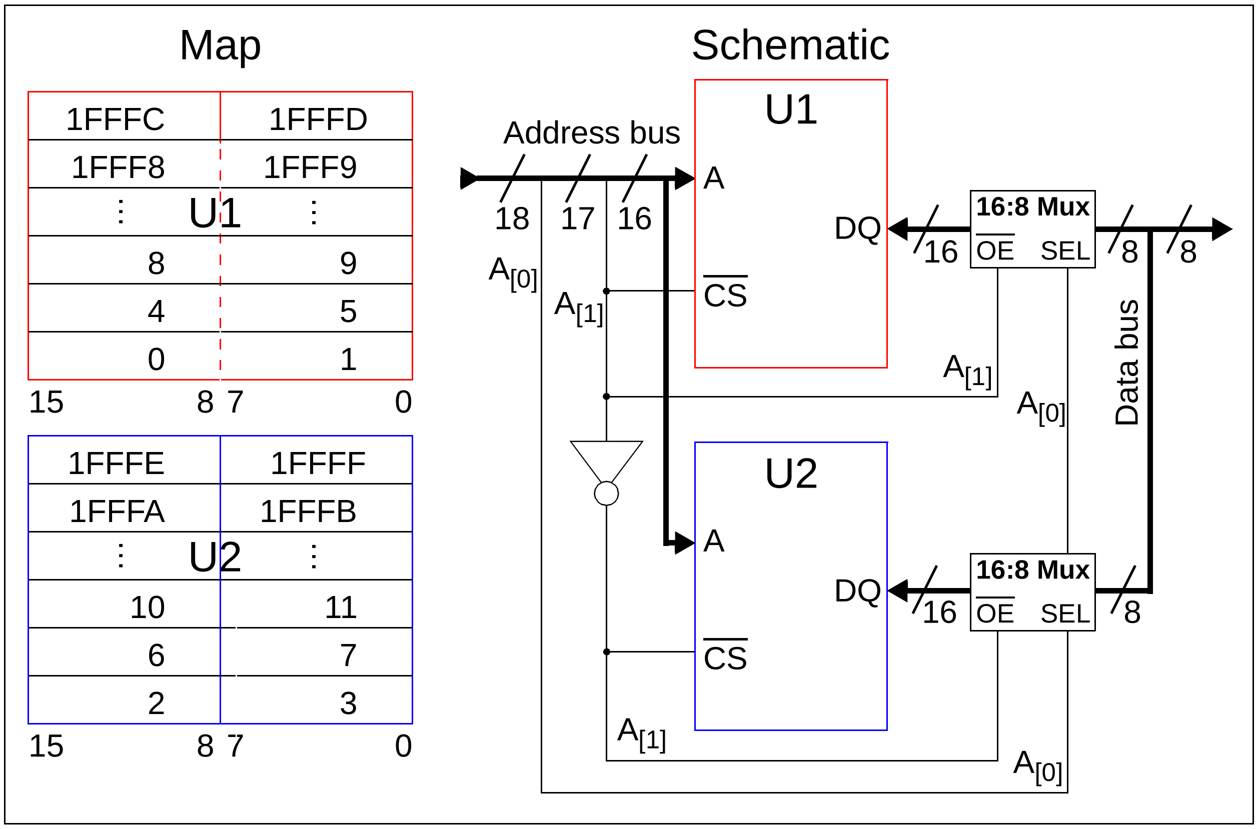


Figure 44: Masked and Interleaved Map Type Design Example

Note
A Masked and Interleaved map can include an address offset.

Byte-Sliced and Interleaved Type

Use the Byte-Sliced and Interleaved map type when both the following are true:

- Consecutive logical addresses alternate between two or more devices or combinations of devices.
- Some of the data bytes of the devices will be unused (disabled).

In this map type, the addressable unit width can be any number of bytes:

addressable unit width = (8 bits) * n (where n = 1, 2, 3, ...)

All devices grouped in an interleaved row must have the same number of locations (address bus width), and the number of devices must be an integral power of two (2, 4, 8, 16, ...):

group width = (addressable unit width) * 2^k (where k = 0, 1, 2, ...)

Also, the group width is the sum of the enabled bytes of all devices.

Figure 45 shows a schematic for a design example that uses three 16-bit devices. Bits 0–1 of the system address bus are decoded by the BSI decoder to derive the Chip Select pins for the group devices. In this type, Byte Slice 0 represents the Most Significant Byte (the left-most byte) in a device. Thus, each 16-bit device in the example has two byte slices: byte slice 0 (bits 15–8) and byte slice 1 (bits 7–0), at its DQ output.

In device U1, both byte slices are present at the DQ output, but only byte slice 1 (bits 7–0) is input to the multiplexer; byte slice 0 (bits 15–8) is discarded. Thus, for device U1, byte slice 1 (bits 7–0) is used (enabled), while byte slice 0 (bits 15–8) is unused (disabled or masked). Similarly, in device U2, byte slice 0 is enabled, while byte slice 1 is disabled. In device U3, byte slices 0 and 1 are both enabled.

As the data enters the multiplexer, byte slice 0 of U2 and byte slice 1 of U1 form a 2-byte data word, and byte slices 0 and 1 of device U3 form another 2-byte data word. The device group data width is the sum of all enabled bytes: 1 byte for U1, 1 byte for U2, and 2 bytes for U3, making 4 bytes. In this example, the addressable unit width is 16 bits (2 bytes), so only half of the device group data width can be addressed at a time. Through the multiplexer, the data word of device U3 is interleaved with the combined data word of devices U1 and U2, so that these have alternating addresses. In the example, the addressable unit width is 16 bits, the enabled width of each device is 8 bits, and the group width is 16 bits, so the relationship is satisfied if n = 0:

group width = (addressable unit width) * 2^0 = 16

On the memory map, the enabled bytes for each device are represented by shading. The data word of devices U1/U2 uses even-numbered addresses, while the data word of device U3 uses odd-numbered addresses.

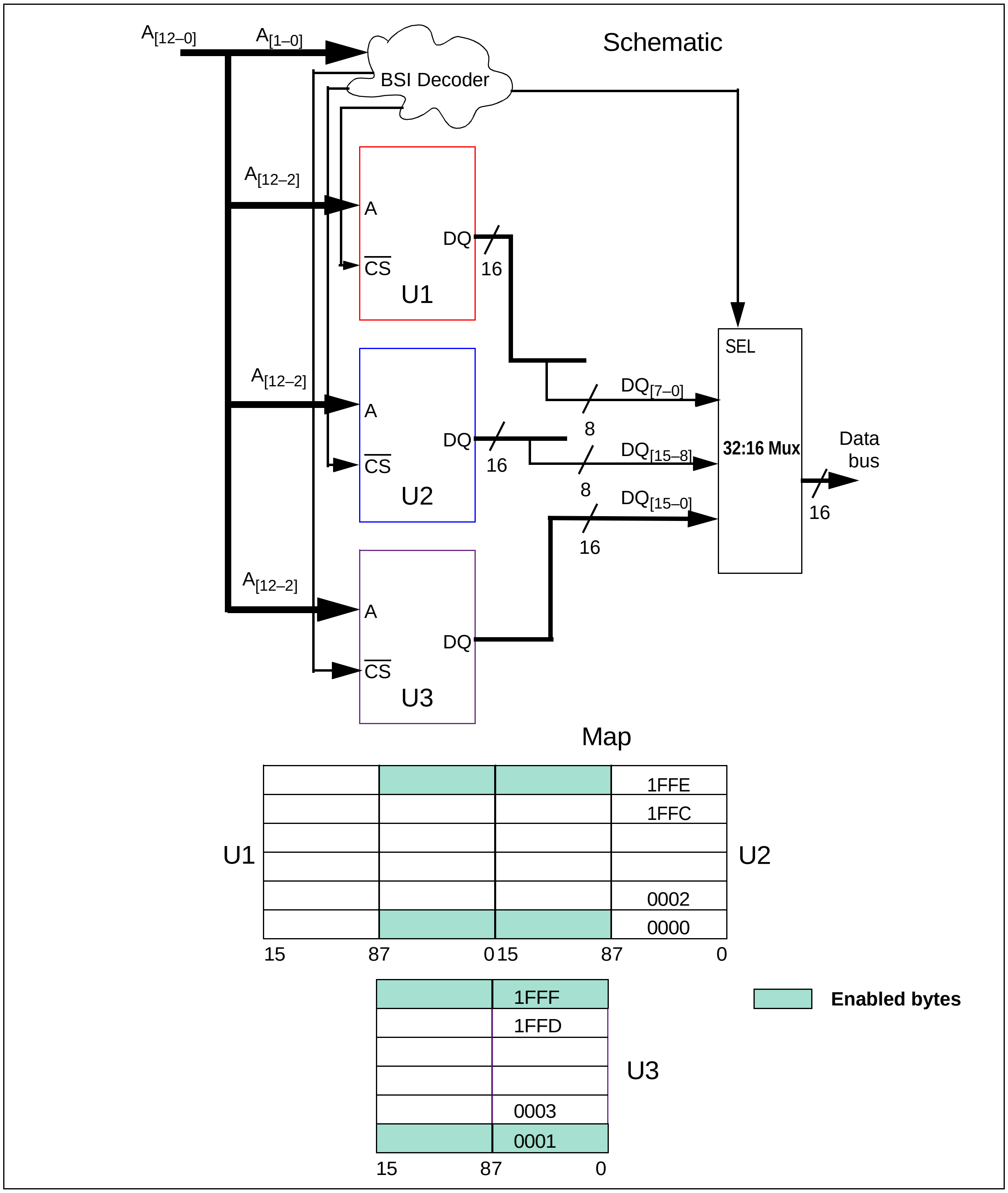


Figure 45: Byte-Sliced and Interleaved Mapping Design Example

Note

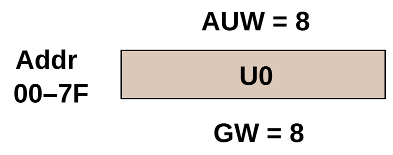
A Byte-Sliced and Interleaved map can include an address offset.

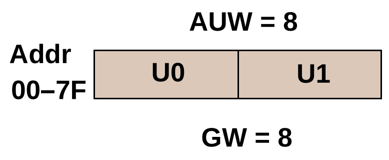
Summary

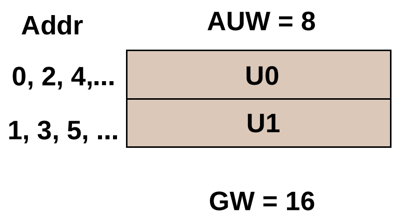
Table 11 summarizes the required relationships among the addressable unit width (AUW), device width (DW), and group width (GW) for the six map types.

Table 11: Required Relationships Among AUW, DW, and GW

Map Type	Required Relationship	Example

Device	AUW = GW = DW
	
	Example:
	AUW = GW = DW = 8 bits
		

Sliced	AUW = GW = DW* n
	(where n = 2, 3, 4, ...)
	
	Example:
	DW = 4 bits
	AUW = GW = 4*2 = 8 bits
	(n = 2)
		

Interleaved	GW > AUW = DW
	GW = AUW*2^n
	(n = 1, 2, 3, ...)
	Example:
	AUW = DW = 8 bits
	GW = 16 bits
		 Addr
		AUW = 8
		0, 2, 4,...
		U0
		1, 3, 5, ...
		U1
		GW = 16

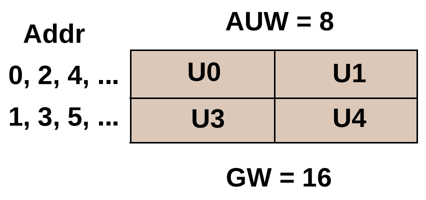
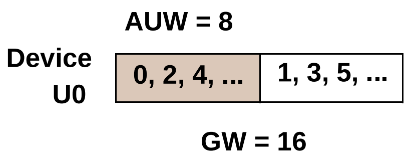
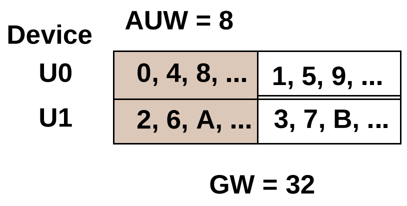
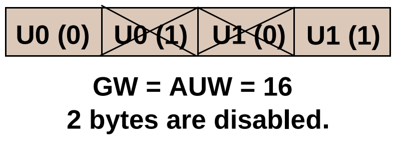
Sliced and Interleaved	GW > AUW > DW
	AUW = DW * n
	(n = 2, 3, 4, ...)
	GW = AUW*2^k
	(k = 1, 2, 3, ...)
	Example:
	DW = 4
	GW = 4 * 4 = 16 (n = 4)
	AUW = 8
	GW = 8 * 2^1 = 16 (k = 1)
		 Addr
		AUW = 8
		0, 2, 4, ...
		U0
		U1
		1, 3, 5, ...
		U3
		U4
		GW = 16

Table 11: Required Relationships Among AUW, DW, and GW

Map Type	Required Relationship	Example

Masked	GW = DW = AUW* n (n = 2, 3, 4, ...)
	Example:
	AUW = 8 bits
	GW = DW = 8 * 2 = 16 bits
	(n = 2)
		 AUW = 8
		Device U0
		0, 2, 4, ...
		1, 3, 5, ...
		GW = 16

Masked and Interleaved	GW > DW = AUW* n (n = 2, 3, 4, ...) $GW = AUW*2^k$ (k = 1, 2, 3, ...) Example: AUW = 8 DW = 8 * 2 = 16 bits (n = 2) $GW = 8 *2^2 = 32$ bits (k = 2)	 AUW = 8 Device U0 U1 0, 4, 8, ... 1, 5, 9, ... 2, 6, A, ... 3, 7, B, ... GW = 32
Byte-Sliced and Interleaved	AUW = (8 bits)* n (n = 1, 2, 3...) $GW = AUW * 2^k$ (k = 0, 1, 2...) Example: GW = AUW = 16. DW = 16 (2 bytes per device)	 U0 (0) U0 (1) U1 (0) U1 (1) GW = AUW = 16 2 bytes are disabled.

Viewing and Editing Memory Data During Simulation

MemScope enables you to connect to a running simulation and view and edit the memory contents of any device or logical address map during the simulation. You can also save these memory contents to a file that you can reload later.

Step 1 — Enable Dynamic Data Exchange (DDX)

Enable Dynamic Data Exchange (DDX) by setting environment variables in your simulator.

Note

Make sure you’ve prepared your testbench as described in “Step 1 — Prepare Your Testbench for MemScope” on page 69.

Do *not* start your simulator before completing steps 1 through 3, below.

1. Use the shell from which you will invoke your simulator to set the MEMPRO_ENABLE_DDX environment variable to any numeric value, as in this example:

```
% setenv MEMPRO_ENABLE_DDX 1
```

2. If you want to connect with MemScope using a port other than the default port of 9988, specify the port by setting the MEMPRO_DDX_PORT environment variable, as in this example:

```
% setenv MEMPRO_DDX_PORT 10027
```

Note

You can use any valid unused port number greater than 1024 (the default is 9988). If you choose a port that is not available, the system automatically selects an unused port. The port selected is recorded in the simulation transcript.

3. **If you are using Synopsys VCS on Solaris**, compile your design using -Xstrict=0x01 -syslib "-lpthread", as described in the *Simulator Configuration Guide for Synopsys Models*.

Caution

VCS users on Solaris: Your simulator could core dump if MEMPRO_ENABLE_DDX is set and you use a simv that was compiled without -Xstrict=0x01 -syslib "-lpthread".

4. Invoke the simulator.

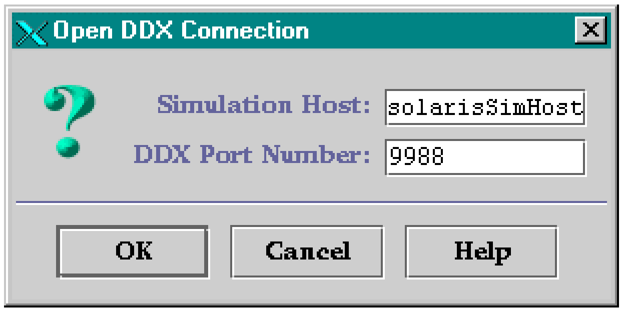
Note

If you receive the error message “unable to bind to socket” / “permission denied”, shut down the simulator, select a different port number, and restart. See step 2.

Step 2 — Open a DDX Connection

Connect MemScope to the simulation using DDX as follows:

1. Invoke MemScope and open the database file specified in your testbench (see “Step 1 — Prepare Your Testbench for MemScope” on page 69).
2. Select **File** > **Open DDX Connection** from the pull-down menu. The following dialog then appears.



Simulation Host: solarisSimHost
DDX Port Number: 9988

Figure 46: DDX Connection Setup Dialog

Note

You cannot connect through DDX until after the design has elaborated.

3. Enter the name of the simulation host, and the port number you specified using the MEMPRO_DDX_PORT command. (The simulator transcript shows which port was opened.) The port number defaults to 9988.

   A message appears when the connection is established. In addition, the host name and port number appear in the file and simulator display area below the Hierarchy Browser, and the Sim flag appears in the status bar.

Caution

VCS users on Solaris: Your simulator could core dump if MEMPRO_ENABLE_DDX is set and you use a simv that was compiled without -Xstrict=0x01 -syslib "-lpthread".

For more information about compiling your design for Dynamic Data Exchange, see the *Simulator Configuration Guide for Synopsys Models*.

Step 3 — Open a DDX View

Once you have opened a database and established a simulator host connection, you can create a DDX View to view and edit the contents of the simulation memory.

1. To create a new DDX View of a particular device or logical address map, do one of the following:
   - ❍ Click the DDX View button in the Toolbar.
   - ❍ Right-click a device in the Hierarchy Browser, then select DDX from the pop-up menu (see Figure 12 on page 65).
   - ❍ Select a device in the Device Table, then click the DDX button in the table.

   Each of these opens the dialog shown below. Use its pull-down menu to select the LAM or device.

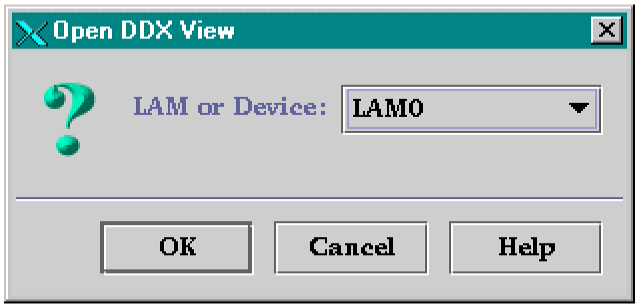
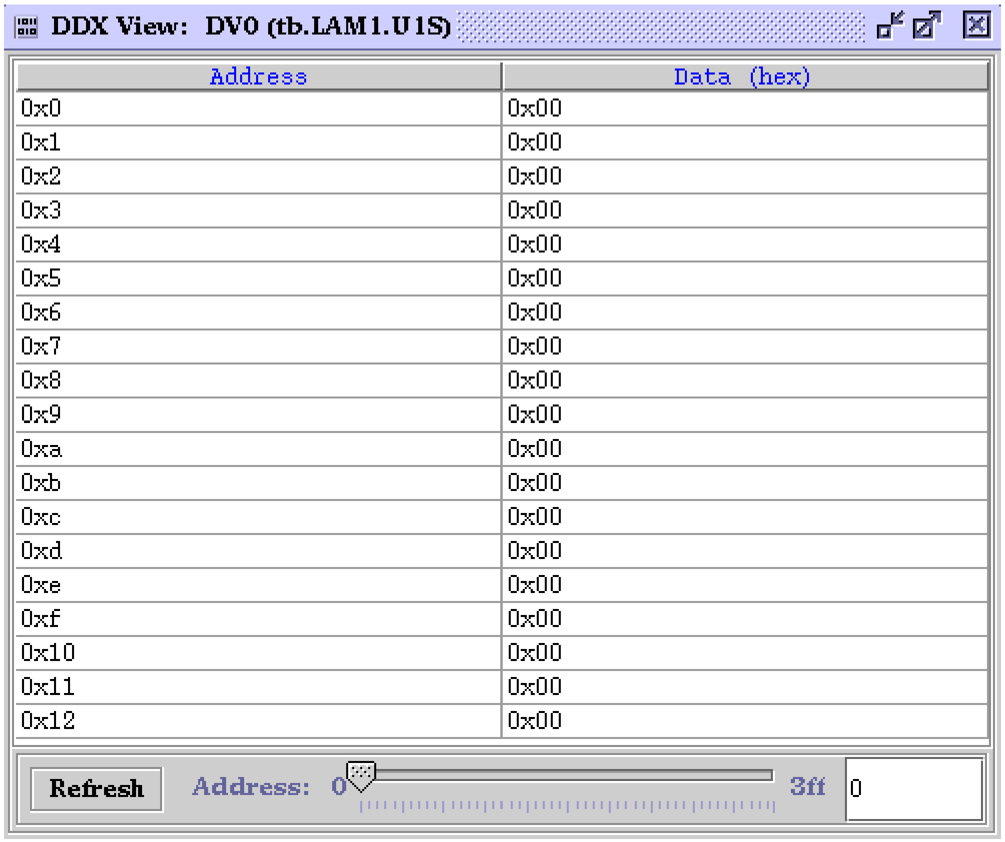


Figure 47: Open DDX View Dialog

The DDX View for the device then appears, such as that shown in Figure 48.



Address	Data (hex)
0x0	0x00
0x1	0x00
0x2	0x00
0x3	0x00
0x4	0x00
0x5	0x00
0x6	0x00
0x7	0x00
0x8	0x00
0x9	0x00
0xa	0x00
0xb	0x00
0xc	0x00
0xd	0x00
0xe	0x00
0xf	0x00
0x10	0x00
0x11	0x00
0x12	0x00

Figure 48: DDX View

Once you’ve created and saved a DDX View, its name is listed in the Hierarchy Browser, and you can then re-open it by double-clicking.

Browsing Memory Contents

You can browse the contents of the DDX View in several ways:

- Use the Page Up and Page Down keys to move one screen up or down.
- Use the up and down arrow keys to move one row up or down.
- Drag the slider at the bottom of the window.
- Type the time you want to view into the field in the lower-right corner of the window.

Updating the Data in the View

As you scroll through the DDX View, the message "fetching data …" occasionally appears under the Data column while MemScope updates the data in the display.

To update the display anytime, click the Refresh button.

Formatting the Data

You can change the format used to display the data in the DDX View:

1. Right-click in the Data column to display this pop-up menu:

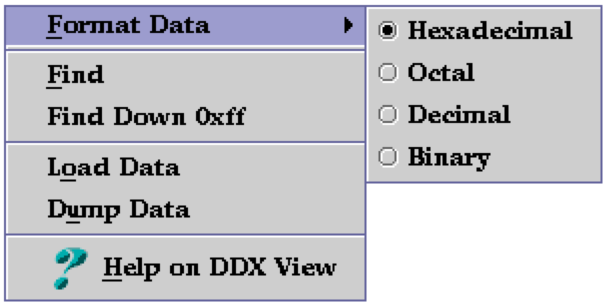
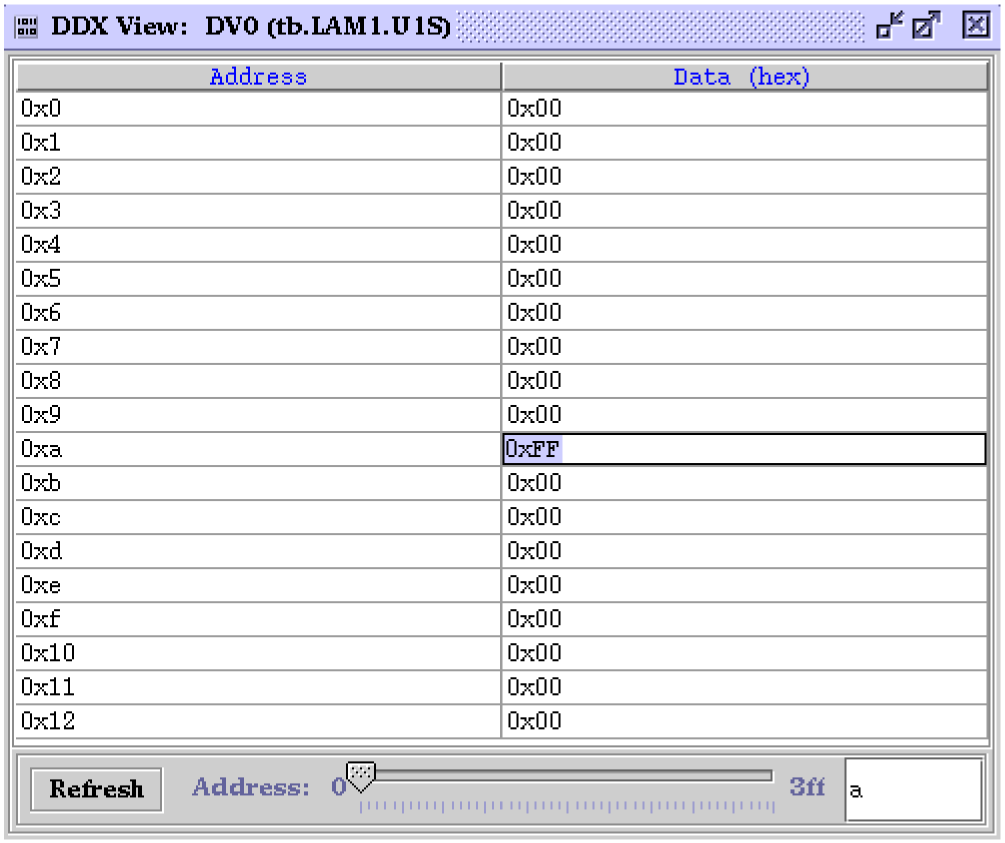


Figure 49: DDX View Pop-Up Menu

2. Select Format Data, then select a format.

Editing Memory Contents

In the DDX View, you can select and edit the value stored at any location in the memory, as shown in Figure 50. Any new value you enter is immediately sent to the simulator.



Address	Data (hex)
0x0	0x00
0x1	0x00
0x2	0x00
0x3	0x00
0x4	0x00
0x5	0x00
0x6	0x00
0x7	0x00
0x8	0x00
0x9	0x00
0xa	0xFF
0xb	0x00
0xc	0x00
0xd	0x00
0xe	0x00
0xf	0x00
0x10	0x00
0x11	0x00
0x12	0x00

Figure 50: DDX View with Modified Data

Note

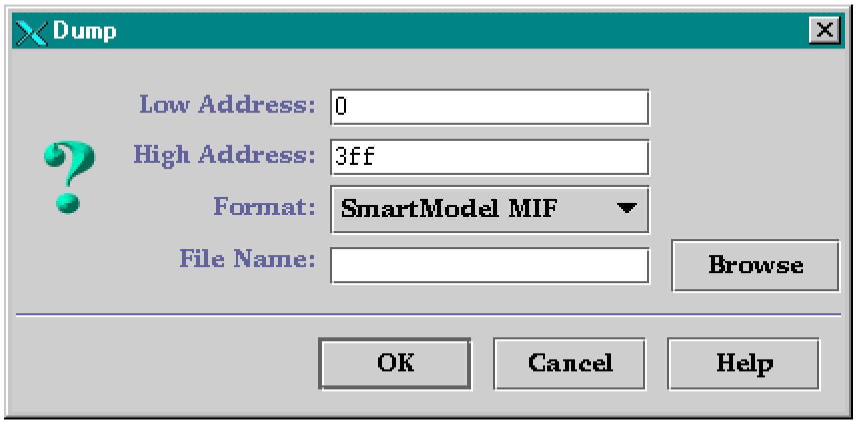
Make sure the values you enter are legal for the current data format. To format the data, see “Formatting the Data” on page 110.

You can use unknown (X) values only if data is displayed in binary format.

Dumping Memory to a File

From the DDX View, you can dump all or part of the displayed memory contents to a file.

1. Right-click in the DDX View to open the pop-up menu (see Figure 49), then select Dump Data. The dialog below then appears.


Dump

Low Address: 0

High Address: 3ff

Format: SmartModel MIF

File Name:

Figure 51: DDX Dump Dialog

2. Specify the memory range to dump by entering values in the Low Address and High Address fields.

   To dump the entire memory, use the default values that appear in the dialog.

3. Use the Format pull-down menu to choose the format of the dump file: SmartModel MIF (the default) or Verilog readmemh.
4. Click the Browse button to select a destination directory for the saved file.

Loading Memory from a File

You can also load previously-saved memory contents from a SmartModel MIF or Verilog readmemh file into a DDX View:

1. Right-click in the DDX View to open the pop-up menu (see Figure 49).
2. Select Load Data, and use the dialog that appears to select the file to load.

5 HDL Testbench Interface

The DesignWare Memory Model package includes a set of commands to interface between your testbench code and DWMM model instances. You can incorporate these commands in your design as global Verilog tasks, VHDL procedures, C functions, or VERA methods.

This chapter describes the HDL testbench commands, and contains the following sections:

Accessing the DWMM HDL Testbench Commands

Before an HDL Testbench can use the DWMM testbench commands, you must include the testbench interface in the simulation. These commands are contained in two sets of files, one for VHDL and another for Verilog, listed in Table 12 on page 114.

These files are provided in $LMC_HOME/sim/*simulator_interface*/src. For instructions on how to use the appropriate files for each supported simulator, see the *Simulator Configuration Guide for Synopsys Models*.

Command Summary

Table 12 lists testbench commands by operation. On the following reference pages, commands are arranged in alphabetical order.

Table 12: Testbench Commands by Operation

Command	Description
Memory Image File Commands	
mem_load	Initializes a memory instance with data from an external file.
mem_unload	Deallocates all or a portion of memory in an instance.
mem_dump	Writes the contents of a memory instance to an external file.
Memory Location Access Commands	
mem_peek	Reads data from a specified address in a memory instance.
mem_poke	Writes data to a specified address in a memory instance.
Memory Address Tracing Commands	
mem_trace	Sets a trace on a specified address range.
mem_untrace	Clears a trace on a specified address range.
mem_lastop	Shows the last traced operation on a memory location.
Model Message Control Command	
mem_msg_level	Sets the message mask for a memory instance.
MemScope Database and History Commands	
mem_create_db	Creates a new MemScope database (.mpd) file or reads an existing database file.
mem_begin_history	Starts saving in an .mph file the history of all instantiated DWMM models.
mem_end_history	Stops saving model history in the .mph file specified by the previous mem_begin_history command.

Comparison of HDL Commands, VERA Methods, and C Functions

The HDL commands, VERA methods, and C functions are very similar. However, there are some differences, as detailed in Table 13.

Table 13: HDL, VERA, and C Comparison

HDL Command	VERA Method	C Function	Comments
mem_begin_history	inst.begin_history	slm_begin_history	
mem_create_db	inst.create_db	slm_create_db	
mem_dump	inst.dump	slm_mem_dump	
mem_end_history	inst.end_history	slm_end_history	
—	—	slm_find_instance slm_find_instancebyname	These functions are required only in the C testbench because other C functions use a simulator-generated handle, rather than a user-defined instance name.
—	inst.get_msg_level	slm_get_message_level	Not supported in HDL
—	inst.instance_info	slm_mem_instance_info	Not supported in HDL
mem_lastop	—	—	Not supported in C or VERA.
mem_load	inst.load	slm_mem_load	—
mem_msg_level	inst.set_msg_level	slm_set_message_level	—
mem_peek	inst.peek	slm_mem_peek	—
mem_poke	inst.poke	slm_mem_poke	—
mem_trace	—	—	Not supported in C or VERA.
mem_unload	inst.unload	slm_mem_unload	—
mem_untrace	—	—	Not supported in C or VERA.

Identifying Instances in Testbench Commands

Most DWMM HDL testbench commands have instance arguments. You can specify the instance argument four ways when using Verilog simulators, and three ways when using VHDL simulators:

- As a string containing a hierarchical pathname to a DWMM model instance (Verilog only).
- As a unique, user-defined integer identifier, specified with a ModelId attribute statement at instantiation. ModelId must be a quoted string containing a positive integer.

Note

ModelId values passed to HDL testbench commands ***must*** be integers, not strings. For example:

Valid: mem_peek(5, *peek_address*, *status*);

Invalid: mem_peek("5", *peek_address*, *status*);

- As a unique, user-defined model alias string, specified with a ModelAlias attribute statement at instantiation. ModelAlias must be a valid alphanumeric string.

Note

For any instance, you can assign values to both ModelId and ModelAlias.

- As a string containing the name of a logical address map, created for logical memory mapping (VHDL and Verilog). For more information on logical address maps, see “Working with Logical Address Maps” on page 81.

Command Return Status

Each testbench command has a status argument, which returns one of three values, depending on the success of the operation. The command return status values are defined in mempro_pkg.v and mempro_pkg.vhd, and are listed in Table 14.

Table 14: Command Return Status Values

Value	VHDL Constant	Verilog define	Comment
1	SLM_TESTBENCH_WARNING	‘SLM_TESTBENCH_WARNING	An abnormal condition has been detected, but simulation can continue.
0	SLM_TESTBENCH_SUCCESS	‘SLM_TESTBENCH_SUCCESS	The operation completed successfully.
–1	SLM_TESTBENCH_FAILURE	‘SLM_TESTBENCH_FAILURE	The operation failed. Simulation can continue, but might not exhibit correct behavior.

Command Reference

The following pages list the HDL testbench interface commands for DWMM models, in alphabetical order.

mem_begin_history

Starts saving the history of all instantiated DWMM models.

Note

To improve performance, all writes to a history file are buffered. Because of this, the mem_end_history() function *must* be called at the end of simulation to make sure events are properly written to the history file. If mem_end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Syntax

mem_begin_history (filename, status)

Arguments

filename	Input — a Verilog character array or VHDL string that specifies the name of the file to which the history information is to be written. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To identify the file as a history file, and to more conveniently browse files in the MemScope File Open window, use the file suffix “.mph”.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command opens the specified history file in overwrite mode. The state of all instantiated DWMM models is written into the file, then MemScope information is appended to the file. Recording begins at the next memory access of a DWMM model.

To enable another history session and save model history to a different file, you must close the current history session with the mem_end_history command.

Examples

Verilog

```
mem_begin_history("hist02.mph",stat);
```

VHDL

```
mem_begin_history("hist02.mph",stat);
```

mem_create_db

Creates a new MemScope database file or reads an existing database file.

Syntax

mem_create_db (filename, comment, status)

Arguments

filename	Input — a Verilog character array or VHDL string that specifies the name of the file to which the history information is to be written. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To help identify the file as a database file, use the file suffix ".mpd".
comment	Input — a Verilog character array or VHDL string that specifies user comments to be displayed by MemScope.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

If the specified database file exists, this function reads it; otherwise, it creates a MemScope database file that contains device information about all instantiated DWMM models in the simulation. You can call this command only once in a simulation session. In your code, make sure you place this command *before* any testbench command that uses a logical address map as the instance argument, and *after* a delta delay statement, because some simulators do not instantiate DWMM models until time zero.

Examples

Verilog

```
mem_create_db("sim2_3-27-00.mpd","Simulation run 2, 27-Mar-00",stat);
```

VHDL

```
mem_create_db("sim2_3-27-00.mpd","Simulation run 2, 27-Mar-00", stat);
```

mem_dump

Writes to an external file the contents of a specified address range of an instance.

Syntax

mem_dump (instance, low_addr, high_addr, filename, format, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81.
low_addr	Input — a Verilog register or VHDL std_logic_vector that specifies the lowest memory address, inclusive, whose data is to be written.
high_addr	Input — a Verilog register or VHDL std_logic_vector that specifies the highest memory address, inclusive, whose data is to be written.
filename	Input — a Verilog character array or VHDL string that specifies the name of the file to which the memory instance contents are to be written. Legal filename characters are alphanumeric, underscore, hyphen, or period.
format	Input — a Verilog or VHDL integer that specifies the format in which the file is to be written. Allowed values are 0 and 3; for definitions of these integers, see Table 15 on page 121.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command writes the contents of the memory instance to the named file. If *filename* is an empty string, the file names are derived from the model instance names, one file per instance.

The mem_dump command writes only those memory locations that have been written. For example, if you specify a low address of 0x0000 and a high address of 0x0100, and the contents of addresses 0x0080 to 0x00af have not been written, the dump file will contain only addresses 0x0000 to 0x007f and 0x00b0 to 0x00ff.

To determine whether or not a memory location has been written, mem_dump compares its contents with the default data value for that memory instance. If the values match, the memory location is assumed not to have been written, although it could actually have been written with data matching the default memory value.

Note

When you load a memory instance with a dumped memory file using the mem_load command, ensure that the default memory value of the instance to be loaded matches that of the instance that originated the dump file. Otherwise, the load could introduce unexpected data into the instance.

Table 15 lists valid file formats to use with mem_dump. These values are defined in the mempro_pkg.vhd or mempro_pkg.v file in the $LMC_HOME/sim/*simulator_interface*/src directory for each supported simulator. For more information on memory file formats, see "Memory Image Files" on page 209.

Table 15: mem_dump File Formats

Format	Value	VHDL Constant	Verilog 'define
SmartModel MIF	0	SLM_FMT_MIF	'SLM_FMT_MIF
Verilog $readmemh	3	SLM_FMT_VLOG	'SLM_FMT_VLOG

Examples

Verilog

```
mem_dump("u1.bank1",'h0000,'hFFFF,"mem.dmp",'SLM_FMT_MIF,stat);
mem_dump(10,'h0000,'hFFFF,"mem.dmp",'SLM_FMT_MIF,stat);
```

VHDL

```
mem_dump("u1/bank1",X"0000",X"FFFF","mem.dat",SLM_FMT_MIF,stat);
mem_dump(10,X"0000",X"FFFF","mem.dat",SLM_FMT_MIF,stat);
```

mem_end_history

Stops saving model history in the .mph file specified by the previous mem_begin_history command.

Note

To improve performance, all writes to a history file are buffered. Because of this, the mem_end_history() function *must* be called at the end of simulation to make sure events are properly written to the history file. If mem_end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Syntax

mem_end_history (status)

Arguments

status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command ends the current history session and closes the history file previously opened by the mem_begin_history command.

Examples

Verilog

```
mem_end_history(stat);
```

VHDL

```
mem_end_history(stat);
```

mem_lastop

Shows the last traced operation on a memory location.

Syntax

mem_lastop (handle, address, data, write, status)

Arguments

handle	Output — a Verilog or VHDL integer that receives the handle of the last DWMM memory instance.
address	Output — a Verilog register or VHDL std_logic_vector that receives the memory address of the last read or write operation. Ensure that the register or std_logic_vector has enough bit width to accommodate the address width of any DWMM model instantiated.
data	Output — a Verilog register or VHDL std_logic_vector that receives the data last read from or written to the specified instance. Ensure that the register or std_logic_vector has enough bit width to accommodate the data width of any DWMM model instantiated.
write	Output — a Verilog or VHDL integer that receives read/write information. Values returned are 0 if the last operation was a read access, or 1 if the last operation was a write access.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command retrieves information about the last traced action on any DWMM model instance. It is typically called from a testbench process that is sensitive to a net specified by one or more calls to mem_trace().

Examples

Verilog

```
mem_lastop(inst,last_addr,last_data,lastop,stat);
```

VHDL

```
mem_lastop(inst,last_addr,last_data,lastop,stat);
```

Note

If you have multiple DWMM models instantiated, have used the mem_trace() command more than once on models with different address or data sizes, and are using the same signal net in the different calls to mem_trace(), be sure to call mem_lastop() with address and data containers at least as wide as the widest address and data fields of your memory models. mem_lastop() returns the latest traced operation for all model instances.

In the following example, assume that mem_inst1 has a 12-bit address range and 16 bits of data, while mem_inst2 has a 14-bit address range and 8 bits of data.

```
mem_trace( mem_inst1, lo_adr, hi_adr, "trigger_signal", status );
mem_trace( mem_inst2, lo_adr, hi_adr, "trigger_signal", status );
...
```

Note

When using the mem_trace command with NC Verilog, you must disable access protection for the net argument ("trigger_signal" in the above example) so that it is accessible from the PLI layer. For details, see "mem_trace" on page 130.

When mem_lastop() is called in the following code fragment, you cannot predict whether last_inst will be mem_inst1 or mem_inst2, so you must provide for either by defining last_addr to be at least 14 bits wide and last_data to be at least 16 bits wide.

```
always @trigger_signal
       begin
       mem_lastop( last_inst, last_addr, last_data, last_op, stat );
       $display("INST %d: A=%x D=%x Op=%d",
             last_inst, last_addr, last_data, last_op);
       end
```

mem_load

Initializes a memory instance with data from an external file.

Syntax

mem_load (instance, filename, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81.
filename	Input — a Verilog character array or VHDL string that specifies the name of the memory image file, in SmartModel MIF or Verilog $readmemh format, with which to load the memory instance.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command initializes locations in the specified memory instance, using the contents of the specified file. Any memory locations not loaded by the contents of the specified file retain their previous values.

Examples

Verilog

```
mem_load("u1.bank1","mem.dat",stat);
mem_load(10,"mem.dat",stat);
```

VHDL

```
mem_load(“u1/bank”,"mem.dat",stat)
mem_load(10,"mem.dat",stat)
```

mem_msg_level

Sets the message mask for a memory instance.

Syntax

mem_msg_level (instance, mask, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81. If you use a logical address map, the message mask is applied to all physical devices that are part of that map.
mask	Input — a Verilog integer bitmap or VHDL integer that specifies the message type or types to be reported by the specified model instance. Allowed values in Table 16.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command controls message reporting for messages in the specified memory instance. Fatal messages are always reported. The mask value is computed by any logical or arithmetic combination of constants as an integer bitmap interpreted as shown in the “Mask Bit” column of Table 16.

Table 16: mem_msg_level Values

Mask Bit (Value)	VHDL Constant	Verilog ‘define	Interpretation
– (0)	SLM_NO_MSGS	‘SLM_NO_MSGS	No messages (except Fatal)
0 (1)	SLM_ERROR	‘SLM_ERROR	Error messages
1 (2)	SLM_WARNING	‘SLM_WARNING	Warning messages
2 (4)	SLM_TIMING	‘SLM_TIMING	Timing messages
3 (8)	SLM_XHANDLING	‘SLM_XHANDLING	X-handling messages

Table 16: mem_msg_level Values (Continued)

Mask Bit (Value)	VHDL Constant	Verilog ‘define	Interpretation
4 (16)	SLM_INFO	‘SLM_INFO	Info messages
0–4 (31)	SLM_ALL_MSGS	‘SLM_ALL_MSGS	All messages
5–31	—	—	Reserved for future use

The constants listed in Table 16 are defined in mempro_pkg.v and mempro_pkg.vhd found in $LMC_HOME/sim/*simulator_interface*/src.

Note

When the message mask value is changed, the new message mask does not take effect until after the next read or write cycle executed by the specified instance.

For detailed descriptions of all message categories, see “Controlling Model Messages” on page 48.

Examples

The following examples cause the specified instance to report Fatal and Error messages.

Verilog

```
mem_msg_level(5,‘SLM_ERROR,stat);
mem_msg_level("u1.bank1",‘SLM_ERROR,stat);
```

VHDL

```
mem_msg_level(10,SLM_ERROR,stat);
mem_msg_level("u1/bank1",SLM_ERROR,stat);
```

mem_peek

Reads data from a specified address in a memory instance.

Syntax

mem_peek (instance, address, data, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81.
address	Input — a Verilog register or VHDL std_logic_vector that contains the memory address to be read.
data	Output — a Verilog register or VHDL std_logic_vector that receives the memory data value. Ensure that the register or std_logic_vector has enough bit width to accommodate the data width of the specified instance.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command reads data from a specified address in a specified memory instance.

Examples

Verilog

```
mem_peek(10,’h0100,rdata,stat);
mem_peek("u1.bank1",’h0100,rdata,stat);
```

VHDL

```
mem_peek(10,X"0100",rdata,stat);
mem_peek("u1/bank1",X"0100",rdata,stat);
```

mem_poke

Writes data to a specified address in a memory instance.

Syntax

mem_poke (instance, address, data, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81.
address	Input — a Verilog register or VHDL std_logic_vector that contains the memory address to be written to.
data	Input — A Verilog register or VHDL std_logic_vector that contains the data value to be written.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command writes specified data into the memory instance at the indicated address. The width of the data array must match the declared data width of all specified memory instances.

Examples

Verilog

```
mem_poke(10,’h0100,’h0101,stat);
mem_poke("u1.bank1",’h0100,’h0101,stat);
```

VHDL

```
mem_poke(10,X"0100",X"0101",stat);
mem_poke("u1/bank1",X"0100",X"0101",stat);
```

mem_trace

Sets a trace on a specified address range.

Note

The mem_trace command is not supported under VSS and Cyclone.

Syntax

mem_trace (instance, low_addr, high_addr, net, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in "Comparison of HDL Commands, VERA Methods, and C Functions" on page 115, or logical address map, as described in "Working with Logical Address Maps" on page 81.
low_addr	Input — a Verilog register or VHDL std_logic_vector that contains the lowest address in the address range to be traced.
high_addr	Input — a Verilog register or VHDL std_logic_vector that contains the highest address in the address range to be traced.
net	Input — a Verilog character array or VHDL string that specifies the name of a Verilog register or VHDL signal to be used to set the trace. This register or signal is toggled (from 1 to 0 and back) each time an access is made to the specified address range in the specified memory instance.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Simulators and the net Argument

Certain simulators impose additional requirements on the use of the net argument.

- When using Scirocco VHDL simulator version 2000.06 or later, you must provide the full path for the net argument, and the path separator must be a colon (for example: ":top:signal_to_toggle").
- In Scirocco version 2000.02, you must provide a partial path (for example: "signal_to_toggle").

- When using NC-VHDL, place the net argument in the top simulation level (for example: ":signal_to_toggle").

Note

NC-VHDL version 4 and later requires you to enable write access to the design from foreign language sub-programs. To do this, add "-ACCESS +w" to the ncelab command.

(For example: "ncelab -ACCESS +w -work work work.CFG_TEST_RTL".)

- When using NC-Verilog, you must disable access protection for the net argument so that it is accessible from the PLI layer.

  NC-Verilog provides read, write, and connectivity protection for all simulation objects. To enable DWMM to change the state of the net argument specified by a mem_trace command, that net must be read- and write-enabled. The simplest way to do this is by adding "+ncaccess+rw" to the simulator command line. For example:

  ```
  ncverilog +ncaccess+rw \
  ${VLOG_SRC_FILES} \
  +incdir+$LMC_HOME/sim/pli/src \
  +loadpli1=swiftpli:swift_boot >& ${0}.log
  ```

  This disables read and write protection for all simulation elements.

  You can also disable protection for a list of simulation elements. For details, refer to your NC-Verilog documentation regarding enabling read, write, or connectivity access to simulation objects.

Description

This command sets the trace flag on memory words in the specified range for the specified memory instance. When a read or write operation occurs at any of the traced addresses, the register or signal specified by the net argument is toggled from 1 to 0, then back to 1. To detect the event on the signal, your testbench should declare a process (VHDL) or an always block (Verilog) sensitive to the named signal. The mem_lastop() function determines which instance was triggered, along with the address, data, and operation type (write or read).

Keep the following in mind when using mem_trace:

- No more than one signal can be active on any one address at any given time. Ensure that address ranges do not overlap with those assigned to other signals.
- Only the most recently set signal is active on each address. When you execute mem_trace() on a range that includes addresses that are already traced, the command overwrites any previously set traces, no warning messages are generated.

Examples

This example illustrates the behavior of mem_trace for subsequent settings of the trace for various address ranges. Table 17 shows the registers that are set as trace flags for each address range as a result of each command in the following sequence for Verilog:

```
mem_trace ("SRAM_8Kx16", 'h00, 'hff, "sig1", status);
mem_trace ("SRAM_8Kx16", 'h00, 'h7f, "sig2", status);
mem_trace ("SRAM_8Kx16", 'h70, 'hff, "sig3", status);
mem_trace ("SRAM_8Kx16", 'h10, 'h1f, "sig1", status);
```

and in the analogous sequence for VHDL:

```
mem_trace ("SRAM_8Kx16", X"00", X"ff", "sig1", status);
mem_trace ("SRAM_8Kx16", X"00", X"7f", "sig2", status);
mem_trace ("SRAM_8Kx16", X"70", X"ff", "sig3", status);
mem_trace ("SRAM_8Kx16", X"10", X"1f", "sig1", status);
```

Table 17: Address Ranges and Trace Flags Set

Command	**Trace Flag, 00–0F**	**Trace Flag, 10–1F**	**Trace Flag, 20–2F**	**Trace Flag, 30–6F**	**Trace Flag, 70–7F**	**Trace Flag, 80–FF**
First command	sig1	sig1	sig1	sig1	sig1	sig1
Second command	sig2	sig2	sig2	sig2	sig2	
Third command					sig3	sig3
Fourth command		sig1				
Trace flags after all commands	sig2	sig1	sig2	sig2	sig3	sig3

Notice that each command overwrites any trace flag specification that already existed for that address range. Thus, the second command overwrites the sig1 specification for all addresses except 80–FF; the third command overwrites the sig2 specification for addresses 70–7F and the sig1 specification for 80–FF; and the fourth command overwrites the sig2 specification for addresses 10–1F.

After the first command, sig1 is the trace flag for addresses 00–FF.

After the second command, sig2 is the trace flag for addresses 00–7F, and sig1 is the trace flag for addresses 80–FF.

After the third command, sig2 is the trace flag for 00–6F, and sig3 is the trace flag for 70–FF.

After the fourth command, sig2 is the trace flag for 00–0f and 10–6F, sig1 is the trace flag for 10–1F, and sig3 is the trace flag for 70–FF.

VHDL Example

The following example is a VHDL code fragment showing the sequence of mem_trace() and mem_lastop() and the definitions of the required variables.

```
signal trigger1 : bit;

testbench: process

VARIABLE status: integer   := 0;

begin -- Don’t use mem_trace at simulator time 0.
    wait until (clk’event and clk = ’0’);
    mem_trace( ModelId, "000000000000", "000011111111", "trigger1", status );
    ...
    -- other user testbench code here
    ...
end process;


trigger_service: process

  VARIABLE last_addr: std_logic_vector( 11 downto 0);
  VARIABLE last_data: std_logic_vector( 15 downto 0);
  VARIABLE last_op: integer;
  VARIABLE inst: integer;
  VARIABLE stat: integer := 0;

begin

    -- Wait for the trigger set up with mem_trace() to fire...
    wait until (trigger1’event);-
    mem_lastop( handle, addr, data, rdwr, stat );

    --Test status returned by mem_lastop() here...
    assert stat=SLM_TESTBENCH_SUCCESS report " LASTOP FAILED " severity error;

    -- Process the inst, last_addr, last_data, and last_op values
    -- returned by mem_lastop() here...


end process;
```

Verilog Example

The following example is a Verilog code fragment showing the sequence of mem_trace() and mem_lastop() and the definitions of the required variables.

```
reg trigger1;
integer lo_adr, hi_adr;

initial
    begin
    // Model instantiation may not happen until end of simulator time tick 0.
    // Delay a tick to ensure the memory has been instantiated.
        #1;
        lo_adr = ’h0000
        hi_adr = ’h3fff;
        mem_trace( mem1.ModelId, lo_adr, hi_adr, "Test1Spec_tb.trigger1",
        status);
    end

// Wait for the trigger created with the mem_trace() to fire...
always @trigger1
   begin
    mem_lastop( last_inst, last_adr, last_data, last_op, status );

    // Test status returned by mem_lastop() here...
    if ( status != ‘SLM_TESTBENCH_SUCCESS )
    begin
        $display("ERROR: LASTOP FAILED");
    end

    // Process the inst. last_addr, last_data, and last_op values
    // returned by mem_lastop() here...
    if ( last_op == 0 ) begin
        $display("RD inst=%d: adr=%x data=%x", last_inst, last_adr, last_data);
    end else begin
        $display("WR inst %d: adr=%x data=%x", last_inst, last_adr, last_data);
    end
   end

initial
   begin
   ...
   // more user testbench code here
   ...
   end
```

Note

When using mem_trace with NC Verilog, you must disable access protection for the net argument (“Test1Spec_tb.trigger1” in the above example) so it is accessible from the PLI layer. For details, see “mem_trace” on page 130.

mem_unload

Deallocates all or a portion of a memory instance.

Syntax

mem_unload (instance, low_addr, high_addr, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in “Comparison of HDL Commands, VERA Methods, and C Functions” on page 115, or logical address map, as described in “Working with Logical Address Maps” on page 81.
low_addr	Input — a Verilog register or VHDL std_logic_vector that contains the lowest memory address to deallocate.
high_addr	Input — a Verilog register or VHDL std_logic_vector that contains the highest memory address to deallocate.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command marks memory words in the range of low_addr through high_addr, inclusive, as being UNWRITTEN for the specified memory instance; the unloaded memory words take on the default value. You use this command during a simulation run to recover system memory allocated for DWMM models once it is no longer needed. The unloaded memory is returned to the host computer’s free memory pool as soon as possible. If you have large areas of simulated memory that were used but will not be read or written again for the remainder of the simulation, using the mem_unload() command might provide a slight speed improvement. Once an address range is deallocated, any read operation within the deallocated address range reads the default value until that address is written again.

Examples

Verilog

```
mem_unload("u1.bank1",'h0000,'h008F,stat);
```

VHDL

```
mem_unload(10,X"0000",X"008F",stat);
```

mem_untrace

Clears a trace on a specified address range.

Syntax

mem_untrace (instance, low_addr, high_addr, status)

Arguments

instance	Input — a Verilog character array or integer value, or a VHDL string or integer that identifies the instance, as described in "Comparison of HDL Commands, VERA Methods, and C Functions" on page 115, or logical address map, as described in "Working with Logical Address Maps" on page 81.
low_addr	Input — a Verilog register or VHDL std_logic_vector that contains the lowest memory address from which to remove a trace.
high_addr	Input — a Verilog register or VHDL std_logic_vector that contains the highest memory address from which to remove a trace.
status	Output — a Verilog or VHDL integer that receives status information. Values returned are shown in Table 14 on page 117.

Description

This command clears the trace flag on memory words in the range of low_addr through high_addr, inclusive, for the specified memory instance.

Examples

Verilog

```
mem_untrace("u1.bank1",'h0000,'h00FF,stat);
```

VHDL

```
mem_untrace(10,X"0000",X"00FF",stat);
```

HDL Command Usage Notes

The following rules apply to all testbench commands.

1. All commands return status values: “SLM_TESTBENCH_SUCCESS” when the operation completed correctly, “SLM_TESTBENCH_FAILURE” when an error occurs, and “SLM_TESTBENCH_WARNING” when a warning occurs, as shown in Table 14 on page 117.
2. If you supply an illegal instance integer identifier, simulation continues and a warning message is returned.
3. All testbench commands convert Xs in address arguments to 0s. The testbench interface issues a warning message when it performs this conversion. For example, in a 9-bit address memory, the Verilog command

   ```
   mem_poke(device_id,9’b0011x001x,8’h55,status);
   ```

   or the VHDL command

   ```
   mem_poke(device_id,"0011X001X","01010101",status);
   ```

   writes to memory address

   ```
   001100010
   ```

4. If you specify an address argument that is shorter than the memory address width, the address is extended with zeros. For example, in a 9-bit address memory, the Verilog command

   ```
   mem_peek(device_id,4’b0011,8’haa,status);
   ```

   or the VHDL command

   ```
   mem_peek(device_id,"0011","10101010",status);
   ```

   reads from memory address

   ```
   000000011
   ```

5. If you specify an address argument that is longer than the memory address width:
   a. VHDL models report an error message; the command is ignored.
   b. Verilog truncates the address argument to the correct width. For example, in a 9-bit address memory, the Verilog command

      ```
      mem_poke(device_id,12’b100011110011,8’h99,status);
      ```

      writes to memory address

      ```
      011110011
      ```

6. In commands that require an upper and lower address, if you reverse the upper and lower addresses, the command exchanges the addresses and executes, but the model returns a warning message.

6
C Testbench Interface

The DesignWare Memory Model package includes a set of commands to interface between your testbench code and DWMM model instances. You can include these commands into a design as global Verilog tasks, VHDL procedures, C functions, or VERA methods.

This chapter describes the C testbench functions, and contains the following sections:

C Testbench Overview

The C testbench interface allows you to create C procedures that can be linked into your simulator to directly access data in the DWMM models in your design. Figure 52 shows how this C testbench interface integrates into your simulation environment. The FlexModel testbench interface integration, which is via an IPC channel, is also shown for comparison.

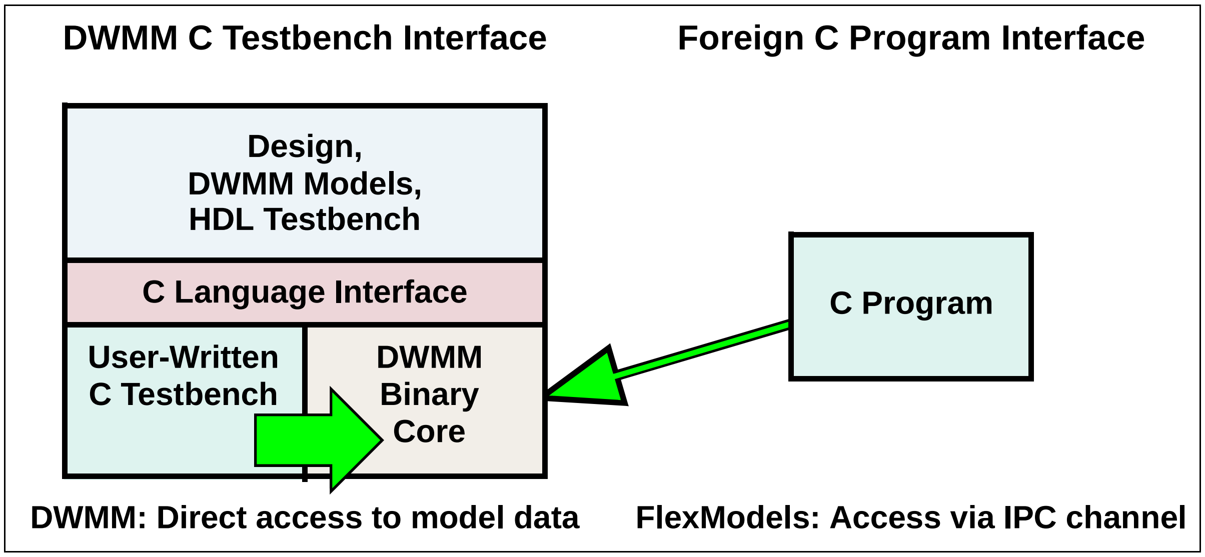

DWMM C Testbench Interface
Foreign C Program Interface
Design, DWMM Models, HDL Testbench
C Language Interface
User-Written C Testbench
DWMM Binary Core
C Program
DWMM: Direct access to model data
FlexModels: Access via IPC channel

Figure 52: C Testbench Interface and Foreign C Program Interface

Including C Programs in Your Testbench

The C routines you develop that call DWMM C interface functions must include the C testbench header file, which is located at $LMC_HOME/include/mempro_c_tb.h, and includes the following:

- Typedef definitions
- Macro definitions for status return codes, memory file formats, and message masks
- Function declarations

For additional information on setting up the C testbench with your specific simulator, see the *Simulator Configuration Guide for Synopsys Models*.

Example of User-Written C Code

The following program executes the slm_find_instance, slm_mem_instance_info and slm_mem_peek functions after the built-in HDL testbench commands execute.

```
/*
 *  This code is a simple "C" testbench that can be called as a task from
 *  the user’s Verilog code.  This code is designed to access a 256 by
 *  8-bit SRAM DWMM model. The first call to slm_mem_peek() below relies on
 *  the fact that the Verilog testbench has already loaded specific memory
 *  locations with certain values.
 */

#include <stdio.h>
#include "mempro_c_tb.h"

#define ADDR_WIDTH 8
#define DATA_WIDTH 8
#define STRING_5 "01010101"

/* Globals */
int test_failures = 0;
int test_number = 0;

/* Local function prototypes */
static void print_err_msg();


int example_DWMM_testbench()
{
  int i;
  int instance_id = 1;
  int data_width, addr_width;

  slm_handle_t handle;
  slm_status_t status;
  slm_message_t msg_mask;

  char instance_name[SLM_MAX_NAME_LENGTH];
  char    class_name[SLM_MAX_NAME_LENGTH];

  slm_addr_t lo_addr[2] = {0,0};
  slm_addr_t hi_addr[2] = {0,0};
  slm_addr_t addr[2];

  slm_data_t data[DATA_WIDTH+1]; /* Add 1 more location for NULL
termination */
  data[DATA_WIDTH] = ’\0’; /* NULL termination enables treatment as a
string */
```

```
/* Use sml_find_instance() to obtain a handle to the instance with id=1 */
/*-------------------------------------------------------------------*/
  slm_find_instance(instance_id, &handle, &status);
  ++test_number;
  if (status != SLM_TESTBENCH_SUCCESS) {
    print_err_msg();
  }
/* Check the instance info against expected values */
/*-------------------------------------------------------------------*/
  slm_mem_instance_info(handle, &data_width, &addr_width,
                instance_name, class_name, &status);
  ++test_number;
  if (status != SLM_TESTBENCH_SUCCESS) {
    print_err_msg();
  }
  ++test_number;
  if (data_width != DATA_WIDTH) {
    print_err_msg();
  }
  ++test_number;
  if (addr_width != ADDR_WIDTH) {
    print_err_msg();
  }
  ++test_number;
  if (strcmp(instance_name, "DWMM1_tb.model1")) {
    print_err_msg();
  }
/* Examine the data at address OxAA with slm_mem_peek () */
/*-------------------------------------------------------------------*/
  addr[0] = 0xAA;
  addr[1] = 0;

  slm_mem_peek(handle, addr, data, &status);
  ++test_number;
  if (status != SLM_TESTBENCH_SUCCESS) {
    print_err_msg();
  }
  ++test_number;
  if (strcmp(data, STRING_5))
    {
      print_err_msg();
      printf("ACT = %s\n", data);
      printf("EXP = %s\n", STRING_5);
    }
```

```
/* Error handling */
/*-------------------------------------------------------------------*/
  if (test_failures) {
    printf("****** Test failed with %d errors\n", test_failures);
  } else {
    printf("TEST PASSED!\n");
  }
  return test_failures;
}


static void print_err_msg()
{
  ++test_failures;
  printf("***FAILURE - test %d\n", test_number);
}
```

C Function Summary

Table 18 lists testbench functions by operation. In the “Function Reference” on page 148, functions are arranged in alphabetical order.

Table 18: Testbench Functions by Operation

Function	Description
Memory Instance Information Functions	
slm_find_instance	Gets the internal instance handle for a specified instance ID.
slm_find_instancebyname	Gets the internal instance handle for a specified instance name or MemScope logical address map name.
slm_mem_instance_info	Gets the address width, data width, name, and memory class of the instance.
Memory Image File Functions	
slm_mem_load	Initializes a memory instance with data from an external file.
slm_mem_unload	Deallocates all or a portion of memory in an instance.
slm_mem_dump	Writes the contents of a memory instance to an external file.
Memory Location Access Functions	
slm_mem_peek	Reads data from a specified address in a memory instance.
slm_mem_poke	Writes data to a specified address in a memory instance.

Table 18: Testbench Functions by Operation (Continued)

Function	Description
Model Message Control Functions	
slm_get_message_level	Gets the message mask for a memory instance.
slm_set_message_level	Sets the message mask for a memory instance.
MemScope Database and History Functions	
slm_create_db	Creates a new MemScope database file or reads an existing database file.
slm_begin_history	Starts saving the history of all instantiated DWMM models.
slm_end_history	Stops saving model history in the .mph file specified by the previous slm_begin_history function.

Comparison of C Functions, HDL Commands, and VERA Methods

The VERA methods, HDL commands, and C functions are very similar. However, there are some differences, as detailed in Table 19.

Table 19: C, HDL, and VERA Comparison

C Function	HDL Command	VERA Method	Comments
slm_begin_history	mem_begin_history	inst.begin_history	
slm_create_db	mem_create_db	inst.create_db	
slm_end_history	mem_end_history	inst.end_history	
slm_find_instance slm_find_instancebyname	—	—	These functions are required only in the C testbench because other C functions use a simulator-generated handle, rather than a user-defined instance name.
slm_get_message_level	—	inst.get_msg_level	Not supported in HDL
slm_mem_dump	mem_dump	inst.dump	
slm_mem_instance_info	—	inst.instance_info	Not supported in HDL
—	mem_lastop	—	Not supported in C or VERA.
slm_mem_load	mem_load	inst.load	
slm_mem_peek	mem_peek	inst.peek	
slm_mem_poke	mem_poke	inst.poke	
—	mem_trace	—	Not supported in C or VERA.
slm_mem_unload	mem_unload	inst.unload	
slm_set_message_level	mem_msg_level	inst.set_msg_level	
—	mem_untrace	—	Not supported in C or VERA.

C Testbench Datatypes

This section describes the datatypes used by the arguments in the C testbench functions.

slm_addr_t

Addresses in DWMM interface functions are represented by an array of elements of type slm_addr_t. For addresses with a width of 32 bits or less, the array contains a single element. For addresses larger than 32 bits, two elements are needed to represent an address (since the maximum address width is 64).

The zeroth element of the array is the least significant and subsequent elements are interpreted as if they were concatenated into a single wide binary word. For example, the least significant bit of the second word is actually the 33rd bit of the address. When representing addresses with widths that are not integral multiples of 32, the high order bits of the most significant 32-bit word must be set to zero on input to and are set to zero output from the DWMM functions.

Note

For all functions that use arguments of this type, the calling code must allocate and initialize an array of the correct size for the specific memory instance accessed.

slm_data_t

Memory data words passed into and retrieved from DWMM interface functions are represented by arrays of elements of type slm_data_t. Each element of the array represents a single bit of the memory word. The zeroth element contains the left-most bit of the memory word and the last element contains the right-most. DWMM memory words have no built-in concept of least or most significant bit; that interpretation is up to the application.

Note

For all functions that use arguments of this type, the calling code must allocate and initialize an array of the correct size for the specific memory instance that is to be accessed.

Note

Arrays of this datatype are *not* implicitly null-terminated.

On input, the DWMM functions recognize the character values 0, 1, x, and X, with the conventional meanings. On output, DWMM functions load the array with the characters 0, 1, X, and U. Memory instances that support only two state storage will convert X input values to 1.

Note
DWMM models can be initialized to the U state, but that state cannot be written.

slm_handle_t

All DWMM interface functions that deal with model instances take an argument of type slm_handle_t to specify which instance that the function will be applied to. These handles are assigned by the DWMM system; you can query them by calling slm_find_instance(). The calling code should not make any assumptions about the magnitude or ordering of the handle values.

slm_status_t

All DWMM interface functions have a status return argument of this type. Possible values returned are defined by the macros SLM_TESTBENCH_SUCCESS, SLM_TESTBENCH_WARNING, and SLM_TESTBENCH_FAILURE.

slm_memory_format_t

Interface functions that deal with data initialization files (for example, slm_mem_dump) take an argument specifying the file format. Supported values are defined by the macros SLM_MIF_FORMAT (SmartModel Memory Image Format) and SLM_VLOG_FORMAT (Verilog readmemh format).

slm_message_t

Interface functions that deal with messages, such as slm_message_level(), take an argument specifying the message mask. The mask value can be assembled from logical or arithmetic combinations of the following message level macros:

SLM_ERROR
SLM_WARNING
SLM_TIMING
SLM_XHANDLING
SLM_INFO
SLM_ALL_MESSAGES
SLM_NO_MESSAGES

Function Return Status

Each testbench function has a status argument, which returns one of three values, depending on the success of the operation. These values are listed in Table 20.

Table 20: Function Return Status Values

Value	Constant	Comment
1	SLM_TESTBENCH_WARNING	An abnormal condition has been detected, but simulation can continue.
0	SLM_TESTBENCH_SUCCESS	The operation completed successfully.
–1	SLM_TESTBENCH_FAILURE	The operation failed. Simulation can continue, but might not exhibit correct behavior.

Function Reference

The following pages list the C testbench interface functions for DWMM, in alphabetical order.

Note

The trace, untrace, and lastop functions, supported in the HDL testbenches, are not supported in the C testbench.

slm_begin_history

Starts saving the history of all instantiated DWMM models.

Note

To improve performance, all writes to a history file are buffered. Because of this, the slm_end_history() function *must* be called at the end of simulation to make sure events are properly written to the history file. If slm_end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Prototype

```
void slm_begin_history(
   char* filename,              /* History file */
   slm_status_t* status         /* Status return */
   );
```

Arguments

filename	Input; a pointer to a character string that specifies the name of the file to which the history information is to be written. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To identify the file as a history file, and to more conveniently browse files in the MemScope File Open window, use the file suffix “.mph”.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function opens the specified history file in overwrite mode. The current state of all instantiated DWMM models is first written into the file, then MemScope information is appended to the file. History recording begins at the next memory access involving a DWMM model.

To enable another history session and save model history to a different file, you must close the current history session with the slm_end_history function.

Example

```
slm_begin_history("hist02.mph",&status);
```

slm_create_db

Creates a new MemScope database file or reads an existing database file.

Prototype

```
void slm_create_db(
   char* filename,              /* Database file */
   char* comment,               /* User comments */
   slm_status_t* status         /* Status return */
   );
```

Arguments

filename	Input; a pointer to a character string that specifies the name of the file to be created or read. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To identify the file as a database file, use the file suffix ".mpd".
comment	Input; a pointer to a character string that specifies user comments to be displayed by MemScope.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

If the specified database file exists, this function reads it; otherwise, it creates a MemScope database file that contains device information about all instantiated DWMM memories in the simulation. You can call this command only once in a simulation session. In your code, make sure you place this command *before* any testbench command that uses a logical address map as the instance argument, and *after* a delta delay statement, because some simulators do not instantiate DWMM models until time zero.

Example

```
slm_create_db("sim2_3-27-00.mpd","Simulation run 2, 27-Mar-00",&status);
```

slm_end_history

Stops saving model history in the .mph file specified by the previous slm_begin_history command.

Note

To improve performance, all writes to a history file are buffered. Because of this, the slm_end_history() function *must* be called at the end of simulation to make sure events are properly written to the history file. If slm_end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Prototype

```
void slm_end_history(
   slm_status_t* status          /* Status return */
   );
```

Arguments

status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function ends the current history session and closes the history file previously opened by the slm_begin_history function.

Example

```
slm_end_history(&status);
```

slm_find_instance

Gets the internal instance handle for a specified instance ID.

Prototype

```
void slm_find_instance(
   int instance_id,             /* Instance id to find */
   slm_handle_t* inst,          /* Handle return */
   slm_status_t* status         /* Status return */
   );
```

Arguments

instance_id	Input; an integer element that specifies the instance ID for which the handle is to be retrieved.
handle	Output; a pointer to a variable of data type slm_handle_t that receives the instance handle, used by other C testbench functions.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function gets the instance handle of the instance specified by its instance ID. The instance handle is an internally-generated number assigned by the simulator at run time. All other C testbench functions require the instance handle as input.

Execute this function to get the "handle" argument required by the slm_get_message_level, slm_mem_dump, slm_mem_instance_info, slm_mem_load, slm_mem_peek, slm_mem_poke, slm_set_message_level, and slm_set_message_level C testbench functions.

Example

```
slm_find_instance(0x373, &handle, &status);
```

slm_find_instancebyname

Gets the internal instance handle for a specified instance name or MemScope logical address map name.

Prototype

```
void slm_find_instancebyname(
   char* instance_name,         /* Instance or MDM name to find */
   slm_handle_t* inst,          /* Handle return */
   slm_status_t* status         /* Status return */
   );
```

Arguments

instance_name	Input; a pointer to a character string that specifies the instance name or logical address map name for which the handle is to be retrieved.(For more information on logical address maps, see "Working with Logical Address Maps" on page 81.)
handle	Output; a pointer to a variable of data type slm_handle_t that receives the instance handle, required by other C testbench functions.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function gets the instance handle of the instance specified by its instance name or logical address map name. The instance handle is an internally-generated number assigned by the simulator at run time. All other C testbench functions require the instance handle as input.

You use this function to get the "handle" argument required by the slm_get_message_level, slm_mem_dump, slm_mem_instance_info, slm_mem_load, slm_mem_peek, slm_mem_poke, slm_set_message_level, and slm_set_message_level C testbench functions.

Example

```
slm_find_instancebyname(map23, &handle, &status);
```

slm_get_message_level

Gets the message mask for a specified memory instance.

Prototype

```
void slm_get_message_level(
   slm_handle_t handle,          /* Instance to access */
   slm_message_t* mask,          /* Message mask to return*/
   slm_status_t* status          /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
mask	Output; a pointer to a variable of data type slm_message_t that contains an integer bitmap specifying message type or types to be reported by the specified model instance. For possible retrieved values, see Table 21 on page 155.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function gets the current message reporting mask for the specified memory instance. Fatal messages are always reported. The mask value is computed as any logical or arithmetic combination of constants. The mask argument is an integer bitmap interpreted as shown in the “Mask Bit” column of Table 21 on page 155.

Table 21: Retrieved Values for slm_get_message_level

Mask Bit (Value)	Constant	Interpretation
– (0x00)	SLM_NO_MSGS	No messages (except Fatal)
0 (0x01)	SLM_ERROR	Error messages
1 (0x02)	SLM_WARNING	Warning messages
2 (0x04)	SLM_TIMING	Timing messages
3 (0x08)	SLM_XHANDLING	X-handling messages
4 (0x10)	SLM_INFO	Info messages
0-4 (0x1F)	SLM_ALL_MSGS	All messages
5–31	—	Reserved for future use

The constants listed in Table 21 are defined in mempro_c_tb.h.

Note
When the message mask value is changed, the new message mask does not take effect until after the next read or write cycle executed by the specified instance.

For detailed descriptions of all message categories, see "Controlling Model Messages" on page 48.

Example

```
slm_get_message_level(handle, &msg_mask, &status);
```

slm_mem_dump

Writes to an external file the contents of the specified address range of an instance.

Prototype

```
void slm_mem_dump(
   slm_handle_t handle,          /* Instance to dump */
   char* filename,               /* File to dump */
   slm_memory_format_t format,   /* Format to write */
   slm_addr_t* low_addr          /* First address to dump */
   slm_addr_t* high_addr         /* Last address to dump */
   slm_status_t* status          /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
filename	Input; a pointer to a character string that specifies the name of the file to which the memory instance contents are to be written. Legal filename characters are alphanumeric, underscore, hyphen, or period.
format	Input; a variable of data type slm_memory_format_t that specifies the format in which the file is to be written. Allowed values are 0 and 3; for definitions of these integers, see Table 22 on page 157.
low_addr	Input; a pointer to a variable of data type slm_addr_t that specifies the lowest memory address, inclusive, whose data is to be written.
high_addr	Input; a pointer to a variable of data type slm_addr_t that specifies the highest memory address, inclusive, whose data is to be written.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function writes to a specified external file the contents of a specified address range of an instance. If *filename* is an empty string, the filenames are derived from the model instance names, one file per instance.

Use the values or constants in Table 22 (defined in mempro_c_tb.h) for the format argument. For more information on memory file formats, see “Memory Image Files” on page 209.

Table 22: slm_mem_dump File Formats

Format	Value	Constant Name
SmartModel MIF	0	SLM_FMT_MIF
Verilog $readmemh	3	SLM_FMT_VLOG

Example

```
slm_mem_dump(handle, "dump_file", SLM_MIF_FORMAT,
        lo_addr, hi_addr, &status);
```

slm_mem_instance_info

Gets the address width, data width, name, and memory class of a specified instance.

Prototype

```
void slm_mem_instance_info(
   slm_handle_t handle,         /* Instance to query */
   int* data_width,             /* Data width return */
   int* addr_width,             /* Address width return */
   char* instance_name,         /* Instance name return */
   char* class_name,            /* Class name return */
   slm_status_t* status         /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
data_width	Output; a pointer to an integer variable that receives the data bus width of the specified instance.
addr_width	Output; a pointer to an integer variable that receives the address bus width of the specified instance.
inst_name	Output; a pointer to a character string of length SLM_MAX_NAME_LENGTH that receives the leaf name of the specified instance.
class_name	Output; a pointer to a character string of length SLM_MAX_NAME_LENGTH that receives the memory class name of the specified instance.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function gets the address width, data width, name, and memory class of a specified instance. The *data_width* and *addr_width* arguments contain the number of bits in their buses. The *inst_name* argument contains the leaf name of the instance. The *class_name* argument contains the model's selected memory type.

Example

```
slm_mem_instance_info(handle, &data_width, &addr_width,
       instance_name, class_name, &status);
```

slm_mem_load

Initializes a memory instance with data from an external file.

Prototype

```
void slm_mem_load(
   slm_handle_t handle,          /* Instance to load */
   char* filename,               /* File to load */
   slm_status_t* status          /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
filename	Input; a pointer to a character string that specifies the name of the file to be used to initialize the instance. The file must be in SmartModel MIF or Verilog $readmemh format.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

Initializes a memory instance with data from a specified external file. Any memory locations not loaded by the specified file retain their previous values.

Example

```
slm_mem_load(handle, "dump_file", &status);
```

slm_mem_peek

Reads data from a specified address in a memory instance.

Prototype

```
void slm_mem_peek(
   slm_handle_t handle,          /* Instance to access */
   slm_addr_t* address,          /* Address to examine */
   slm_data_t* data,             /* Data return */
   slm_status_t* status          /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
address	Input; a pointer to a variable of data type slm_addr_t that specifies the memory address to be read.
data	Output; a pointer to a variable of data type slm_data_t that receives an ASCII representation of the binary data value read. Ensure that the data array has enough bit width to accommodate the data width of the specified instance.

Note

The slm_data_t datatype is *not* null-terminated. Do not use functions that depend on a terminating null (for example, strcpy()) without first adding a null terminator.

status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function loads the array data with the value stored in specified address in the specified instance.

Example

```
slm_mem_peek(handle, addr, data, &status);
```

slm_mem_poke

Writes data to a specified address in a memory instance.

Prototype

```
void slm_mem_poke(
   slm_handle_t handle,         /* Instance to access */
   slm_addr_t* address,         /* Address to modify */
   slm_data_t* data,            /* Data to write */
   slm_status_t* status         /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
address	Input; a pointer to a variable of data type slm_addr_t. that contains the memory address to be written.
data	Input; a pointer to a variable of data type slm_data_t that contains the data value to be written.

Note

The data array does not need to be null-terminated.

status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function writes specified data into the memory instance at the indicated address. The length of the data array must match the declared data width of the memory instance.

Example

```
slm_mem_poke(handle, addr, "01010101", &status);
```

slm_set_message_level

Sets the message mask for a memory instance.

Prototype

```
void slm_set_message_level(
   slm_handle_t handle,         /* Instance to access */
   slm_message_t mask,          /* Message mask to set */
   slm_status_t* status         /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).

Note

If you call slm_set_message_level on a logical address map, the message mask is applied to all physical devices that are included in that map.

mask	Input; a variable of data type slm_message_t that specifies the message type or types to be reported by the specified model instance. Allowed values are in Table 23 on page 164.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function defines message reporting for messages in the specified memory instance. Fatal messages are always reported. The mask argument is an integer bitmap interpreted as shown in the "Mask Bit" column of Table 23. The mask value can be any combination of legal values or addition of constants.

Table 23: slm_set_message_level Values

Mask Bit (Value)	Constant	Interpretation
0 (0x01)	SLM_ERROR	Error messages
1 (0x02)	SLM_WARNING	Warning messages
2 (0x04)	SLM_TIMING	Timing messages
3 (0x08)	SLM_XHANDLING	X-handling messages
4 (0x10)	SLM_INFO	Info messages
5–31		Reserved for future use

The constants listed in Table 23 are defined in mempro_c_tb.h.

Note
When the message mask value is changed, the new message mask does not take effect until after the next read or write cycle executed by the specified instance.

For detailed descriptions of all message categories, see "Controlling Model Messages" on page 48.

Example

The following example causes the specified instance to report Fatal and Error messages.

```
slm_set_message_level(handle, msg_mask-SLM_ERROR, &status);
```

slm_mem_unload

Deallocates all or a portion of a memory instance.

Prototype

```
void slm_mem_unload(
   slm_handle_t handle,         /* Instance to unload */
   slm_addr_t* low_addr,        /* First address to free */
   slm_addr_t* high_addr,       /* Last address to free */
   slm_status_t* status         /* Status return */
   );
```

Arguments

handle	Input; a variable of data type slm_handle_t that specifies the instance handle, as retrieved by slm_find_instance (see page 152) or slm_find_instancebyname (see page 153).
low_addr	Input; a pointer to an array of data type slm_addr_t that contains the lowest memory address to deallocate.
high_addr	Input; a pointer to an array of data type slm_addr_t that contains the highest memory address to deallocate.
status	Output; a pointer to a variable of data type slm_status_t that receives status information. Values returned are shown in Table 20 on page 148.

Description

This function marks memory words in the range of low_addr through high_addr, inclusive, as being UNWRITTEN for the specified memory instance; the unloaded memory words take on the default value. You can use this function during a simulation run to recover system memory allocated for DWMM models once it is no longer needed. The unloaded memory is returned to the host computer’s free memory pool as soon as possible. If you have large areas of simulated memory that were used but will not be read or written again for the remainder of the simulation, using the slm_mem_unload() function might provide a slight speed improvement. Once an address range is deallocated, any read operation within the deallocated address range reads the default value until that address is written again.

Example

```
slm_mem_unload(handle, lo_addr, hi_addr, &status);
```

C Function Usage Notes

The following rules apply to all testbench functions.

1. All functions return status values: “SLM_TESTBENCH_SUCCESS” when the operation completed correctly, “SLM_TESTBENCH_FAILURE” when an error occurred, and “SLM_TESTBENCH_WARNING” when a warning occurred.
2. If you supply an illegal instance integer identifier, simulation continues and a warning message is returned.
3. In functions that require an upper and lower address, if you reverse the upper and lower addresses, the function executes, but the model returns a warning message.

7
VERA Testbench Interface

The DesignWare Memory Model software includes a set of commands to interface between your testbench code and DWMM model instances. You can include these commands your design as global Verilog tasks, VHDL procedures, C functions, or VERA methods. This chapter describes the VERA testbench methods.

VERA is a testbench automation tool that works as a front end to Verilog or VHDL simulators. For details about using VERA with DWMM models, see the *Simulator Configuration Guide for Synopsys Models*.

This chapter contains the following topics:

Method Return Status

Each testbench method has a status argument, which returns one of three values, depending on the success of the operation. The method return status values are listed in Table 24. The model object saves the message and the severity of the error.

Table 24: Command Return Status Values

Value	Constant	Comment
1	SLM_TESTBENCH_WARNING	An abnormal condition has been detected, but simulation can continue.
0	SLM_TESTBENCH_SUCCESS	The operation completed successfully.
–1	SLM_TESTBENCH_FAILURE	The operation failed; simulation can continue, but might not exhibit correct behavior.

You can use showStatus() to access the current error status, as in the following example:

```
program model_test
{
       MemPro inst1;

        inst1 = new(2);

        if ( inst1.showStatus() == SLM_TESTBENCH_FAILURE ) {;
            // Take suitable action
        }
        else {
            // No errors, proceed.
        }

} // program model_test
```

VERA Testbench Method Summary

Table 25 lists testbench methods by operation. In "Method Reference" on page 171, functions are arranged in alphabetical order.

Table 25: Testbench Methods by Operation

Function	Description
Memory Instance Information Method	
inst.instance_info	Gets the address width, data width, name, and memory class of the instance.
Memory Image File Methods	
inst.load	Initializes a memory instance with data from an external file.
inst.unload	Deallocates all or a portion of memory in an instance.
inst.dump	Writes the contents of a memory instance to an external file.
Memory Location Access Methods	
inst.peek	Reads data from a specified address in a memory instance.
inst.poke	Writes data to a specified address in a memory instance.
Model Message Control Methods	
inst.get_msg_level	Gets the message mask for a memory instance.
inst.set_msg_level	Sets the message mask for a memory instance.
MemScope Database and History Methods	
inst.create_db	Creates a new MemScope database file or reads an existing database file.
inst.begin_history	Starts saving the history of all instantiated DWMM models.
inst.end_history	Stops saving model history in the .mph file specified by the previous mem_begin_history command.

Comparison of VERA Methods, HDL Commands, and C Functions

The VERA methods, HDL commands, and C functions are very similar. However, there are some differences, as detailed in Table 26.

Table 26: VERA, HDL, and C Comparison

VERA Method	HDL Command	C Function	Comments
inst.begin_history	mem_begin_history	slm_begin_history	
inst.create_db	mem_create_db	slm_create_db	
inst.dump	mem_dump	slm_mem_dump	
inst.end_history	mem_end_history	slm_end_history	
—	—	slm_find_instance slm_find_instancebyname	These functions are required only in the C testbench because other C functions use a simulator-generated handle, rather than a user-defined instance name.
inst.get_msg_level	—	slm_get_message_level	Not supported in HDL
inst.instance_info	—	slm_mem_instance_info	Not supported in HDL
—	mem_lastop	—	Not supported in C or VERA.
inst.load	mem_load	slm_mem_load	
inst.peek	mem_peek	slm_mem_peek	
inst.poke	mem_poke	slm_mem_poke	
inst.set_msg_level	mem_msg_level	slm_set_message_level	
—	mem_trace	—	Not supported in C or VERA.
inst.unload	mem_unload	slm_mem_unload	
—	mem_untrace	—	Not supported in C or VERA.

Method Reference

The following pages list the VERA testbench interface methods, in alphabetical order.

Note

The trace, untrace, and lastop functions, supported in the HDL testbenches, are not supported in the VERA testbench.

inst.begin_history

Starts saving the history of all instantiated DWMM models.

Note

To improve performance, all writes to a history file are buffered. Because of this, the *inst*.end_history() method *must* be called at the end of simulation to make sure events are properly written to the history file. If *inst*.end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Syntax

***inst*.begin_history** (filename, status)

Arguments

filename	Input; a character string that specifies the name of the file to which the history information is to be written. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To identify the file as a history file, and to more conveniently browse files in the MemScope File Open window, use the file suffix ".mph".
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method opens the specified history file in overwrite mode. The state of all instantiated DWMM models is written into the file, then MemScope information is appended. Recording begins at the next memory access involving a DWMM model.

To enable another history session and save model history to a different file, you must close the current history session with the *inst*.end_history method.

Example

```
mem1.begin_history("hist02.mph",status);
```

inst.create_db

Creates a new MemScope database file or reads an existing database file.

Syntax

***inst*.create_db** (filename, comment, status)

Arguments

filename	Input; a character string that specifies the name of the file to be created or read. Legal characters for the filename are alphanumeric, underscore, hyphen, or period. To identify the file as a database file, use the file suffix ".mpd".
comment	Input; a character string that specifies user comments to be displayed by MemScope.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

If the specified database file exists, this function reads it; otherwise, it creates a MemScope database file that contains device information about all instantiated DWMM models in the simulation. You can call this command only once in a simulation session. In your code, ensure that you place this command *before* any testbench command that uses a logical address map as the instance argument, and *after* a delta delay statement, because some simulators do not instantiate DWMM models until time zero.

Example

```
mem1.create_db("sim2_3-27-00.mpd","Simulation run 2, 27-Mar-00",status);
```

inst.dump

Writes to an external file the contents of a specified address range of an instance.

Syntax

***inst*.dump** (filename, format, low_addr, high_addr, status)

Arguments

filename	Input; a character string that specifies the name of the file to which the memory instance contents are to be written. The file name can include alphanumeric characters, underscores, hyphens, or periods.
format	Input; an integer that specifies the format in which the file is to be written. Allowed values are 0 and 3; for definitions of these integers, see Table 27 on page 173.
low_addr	Input; a bit field that specifies the lowest memory address, inclusive, whose data is to be written.
high_addr	Input; a bit field that specifies the highest memory address, inclusive, whose data is to be written.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method writes the contents of the memory instance to the named file. If *filename* is an empty string, the filenames are derived from the model instance names, one file per instance.

Table 27 lists valid file formats to use with *inst*.dump. These values are defined in the mempromodel.vrh file located in the $LMC_HOME/sim/vera/src directory. For more information on memory file formats, see “Memory Image Files” on page 209.

Table 27: *inst*.dump File Formats

Format	**Value**	**Constant Name**
SmartModel MIF	0	SLM_FMT_MIF
Verilog $readmemh	3	SLM_FMT_VLOG

Example

```
mem1.dump(DUMP_FILE, SLM_MIF_FORMAT,
        lo_addr, hi_addr, status);
```

inst.end_history

Stops saving model history in the .mph file specified by the previous *inst*.begin_history command.

Note

To improve performance, all writes to a history file are buffered. Because of this, the *inst*.end_history() method *must* be called at the end of simulation to make sure events are properly written to the history file. If *inst*.end_history() is not called at simulation end (when history collection is enabled), some events may be omitted from the file.

Syntax

***inst*.end_history** (status)

Arguments

status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method ends the current history session and closes the history file previously opened by the inst.begin_history command.

Example

```
mem1.end_history(status);
```

inst.get_msg_level

Gets the message mask for the memory instance.

Syntax

***inst*.get_msg_level** (mask, status)

Arguments

mask	Input; an integer that contains the message mask currently used by the model. For possible retrieved values, see Table 29 on page 180.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method returns the current message reporting mask for its memory instance. Fatal messages are always reported. The mask value can computed by any logical or arithmetic combination of constants. The mask argument is an integer bitmap interpreted as shown in the “Mask Bit” column of Table 28.

Table 28: Return Values for *inst*.get_msg_level

Mask Bit (Value)	Constant	Interpretation
– (0)	SLM_NO_MSGS	No messages (except Fatal)
0 (1)	SLM_ERROR	Error messages
1 (2)	SLM_WARNING	Warning messages
2 (4)	SLM_TIMING	Timing messages
3 (8)	SLM_XHANDLING	X-handling messages
4 (16)	SLM_INFO	Info messages
0–4 (31)	SLM_ALL_MSGS	All messages

Note

When the message mask value is changed, the new mask does not take effect until after the next read or write cycle executed by the specified instance.

For descriptions of message categories, see “Controlling Model Messages” on page 48.

Example

```
mem1.get_msg_level(msg_mask, status);
```

inst.instance_info

Gets the address width, data width, name, and memory class of the instance.

Note
This method is not supported for the SDRAM Module memory class.

Syntax

***inst*.instance_info** (data_width, addr_width, inst_name, class_name, status)

Arguments

data_width	Output; an integer that receives the data bus width of this instance.
addr_width	Output; an integer that receives the address bus width of this instance.
inst_name	Output; a character string that receives the leaf name of this instance.
class_name	Output; a character string that receives the memory class name of this instance.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method returns the name, type, and size of the memory instance. The data_width and addr_width arguments contain the number of bits in their buses, expressed as a decimal number. The inst_name argument contains the leaf name of the instance. The class_name argument contains a text string that matches the model's selected Memory Type.

Example

```
mem1.instance_info(data_width, addr_width,
        instance_name, class_name, status);
```

***inst*.load**

Initializes a memory instance with data from an external file.

Syntax

***inst*.load** (filename, status)

Arguments

filename	Input; a character string that specifies the name of the memory image file, in SmartModel MIF or Verilog $readmemh format, with which to load the memory instance.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method initializes locations in the memory instance, using the contents of the specified file. Any memory locations not loaded by the contents of the specified file retain their previous values.

Example

```
mem1.load(DUMP_FILE, status);
```

inst.peek

Reads data from a specified address in a memory instance.

Syntax

***inst*.peek** (address, data, status)

Arguments

address	Input; a bit field that specifies the memory address to be read.
data	Output; a bit field with a minimum width of 2048, that receives the binary data value read.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method loads the array data with the value stored in the word at the specified address in the memory instance.

Example

```
mem1.peek(addr, data, status);
```

inst.poke

Writes data to a specified address in a memory instance.

Syntax

***inst*.poke** (address, data, status)

Arguments

address	Input; a bit field that contains the memory address to be written.
data	Input; a bit field with a minimum width of 2048, that contains the binary data value to be written.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method writes specified data into the memory instance at the specified address.

Example

```
mem1.poke(addr, STRING_C, status);
```

inst.set_msg_level

Sets the message mask for a memory instance.

Syntax

***inst*.set_msg_level** (mask, status)

Arguments

mask	Input; an integer bitmap specifying message type or types to be issued by the memory instance.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

Note

If you call *inst*.set_msg_level on a logical address map (LAM), the message mask is applied to all physical devices that are part of that LAM.

This method defines message reporting for messages in the memory instance. Fatal messages are always reported. The mask value can computed by any logical or arithmetic combination of constants. The mask argument is an integer bitmap interpreted as shown in the “Mask Bit” column of Table 29.

Table 29: *inst*.set_msg_level Values

Mask Bit (Value)	Constant	Interpretation
– (0)	SLM_NO_MSGS	No messages (except Fatal)
0 (1)	SLM_ERROR	Error messages
1 (2)	SLM_WARNING	Warning messages
2 (4)	SLM_TIMING	Timing messages
3 (8)	SLM_XHANDLING	X-handling messages
4 (16)	SLM_INFO	Info messages
0–4 (31)	SLM_ALL_MSGS	All messages
5–31	—	Reserved for future use

The constants listed in Table 29 are defined in mempromodel.vrh.

Note

When the message mask value is changed, the new message mask does not take effect until after the next read or write cycle executed by the specified instance.

For detailed descriptions of all message categories, see "Controlling Model Messages" on page 48.

Example

The following example causes the specified instance to report Fatal and Error messages.

```
mem1.set_msg_level(msg_mask-SLM_ERROR, status);
```

inst.unload

Deallocates all or a portion of a memory instance.

Syntax

***inst*.unload** (low_addr, high_addr, status)

Arguments

low_addr	Input; a bit field that contains the lowest memory address to deallocate.
high_addr	Input; a bit field that contains the highest memory address to deallocate.
status	Output; an integer that receives status information. Values returned are shown in Table 24 on page 168.

Description

This method marks memory words in the range of low_addr through high_addr, inclusive, as being UNWRITTEN for the memory instance; the unloaded memory words take on the default value. You use this function during a simulation run to recover system memory allocated for DWMM models once it is no longer needed. The unloaded memory is returned to the host computer’s free memory pool as soon as possible. If you have large areas of simulated memory that were used but will not be read or written again for the remainder of the simulation, using the *inst*_unload() method might provide a slight speed improvement. Once an address range is deallocated, any read operation within the deallocated address range reads the default value until that address is written again.

Example

```
mem1.unload(lo_addr, hi_addr, status);
```

VERA Method Usage Notes

The following rules apply to all testbench methods.

1. All methods return status values: “SLM_TESTBENCH_SUCCESS” when the operation completed correctly, “SLM_TESTBENCH_FAILURE” when an error occurred, and “SLM_TESTBENCH_WARNING” when a warning occurred.
2. In methods that require an upper and lower address, if you reverse the upper and lower addresses, the function executes, but the model returns a warning message.

8 Library Tools

This chapter discusses several library tools included with the DWMM technology software package, and includes the following topics:

- “Admin and Browser” on page 185
- “Translating Memory Image Files with cnvrt2mif” on page 186

Admin and Browser

You may often find that the only tools you need to make effective use of your DWMM models are the sl_admin (Admin) and sl_browser (Browser) tools, both available in $LMC_HOME/bin. The Admin and Browser tools constitute your main interface to the DWMM.

For information about the Admin tool, see “Admin Tool” in the *SmartModel Library Administrator’s Manual*. For details about the Browser tool, see the *SmartModel Library User’s Manual*.

Translating Memory Image Files with cnvrt2mif

DWMM models read memory image files (MIF) to configure themselves at simulation startup. The MIF file format does not match other memory image formats created by third parties; specifically, Intel Hex, and Motorola S-record formats.

You can use the cnvt2mif tool to convert memory data files to MIF file format. Using cnvt2mif, you can also specify the number, name, and bit width of output files and perform endian conversion. In addition, you can select data from within the input file by specifying an address range for the conversion. You can also specify a base address for indexed addressing, and control the verbosity and extent of message display. cnvt2mif reads hexadecimal digits in either upper or lower case.

cnvt2mif translates both Intel Hex and Motorola S-record files, as follows:

- **Intel Hex-checksum** – For Intel Hex-checksum translations, cnvt2mif handles both extended segment address records and extended linear address records.
- **Motorola S-record** – For Motorola S-record translations, cnvt2mif can process input files containing mixed data lengths in different records. The tool recognizes S0, S1, S2, S32, S5, S7, S8, and S9 records anywhere in the file. Only S1, S2, and S3 records contain data to place in the output files.

cnvrt2mif returns execution status values as shown in Table 30 on page 186.

Table 30: cnvrt2mif Execution Status Values

Returned Value	Status
0	cnvrt2mif executed successfully with no error or warning messages.
1	cnvrt2mif executed successfully with no errors, but warning messages were generated.
2	cnvrt2mif failed to execute; errors were detected in the command line syntax, the file I/O, or the input file content.

Syntax

```
% cnvrt2mif input_file input_file(s) [ -o out_file_name ] [-ml]
[-devwidth device_width] [-devs num_devices] [-flip16] [-flip32]
[-flip64] [-range lo_adr hi_adr ] [-baseadr base_adr] [-noxsum]
[-wnodup] [-v ] [-V] [-mape] [-h[elp]] [-u[sage]] [-e[xamples]]
```

Arguments and options are order-independent.

Arguments

input_file(s)	Specifies a list of one or more (up to 15) input files in Intel Hex-checksum or Motorola S-record format. You can mix input files of these two formats in the same command invocation, unless the files have different endianess. The endian option you use (-flip16, -flip32, -flip64, or none) applies to all input files specified in one command; you cannot specify endianism on a file-by-file basis. If files require different endian options, convert them individually, then concatenate them using (for example) the UNIX cat command, as follows:

```
% cat file1.mif file2.mif file3.mif > final_output.mif
```

Options

-baseadr *base_adr*	A hexadecimal number, without a leading “0x”, that specifies the input record address that corresponds to memory index 00000000 in the output files.
-devs *num_devices*	A positive nonzero integer that specifies the number of memory devices for which data files are to be generated. The default is 1.
-devwidth *device_width*	A positive nonzero integer that specifies the bit width of each memory device, and therefore the bit width of each output file. The default is 16.

Note

Only byte-aligned widths (that is, multiples of 8 bits) are currently supported; for example, 8, 16, 24, 32 bit widths.

-e[xamples]	Indicates that examples of cnvrt2mif usage are to be displayed.
-flip16	Indicates that a two-byte endian swap is to be performed on each 16-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.
-flip32	Indicates that a four-byte endian swap is to be performed on each 32-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.

-flip64	Indicates that an eight-byte endian swap is to be performed on each 64-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.
-h[elp]	Indicates that the syntax of cnvrt2mif is to be displayed.
-mape	Indicates that error, warning, and status messages are to be issued on stdout instead of stderr. By default, messages are issued on stderr.
-ml	Indicates that ASCII hex values in the output file are to be lower case. (The letters "ml" stand for MIF Lowercase.) By default, upper case is used.
-noxsum	Suppresses the validation of checksums of input records. By default, these checksums are validated.
-o *out_ file_name*	Specifies the base filename of the MIF format output file; the extension .mif is automatically appended to the name. Multiple output files are named *out_ file_name*_00.mif, *out_ file_name*_01.mif, and so on. By default, the output file is named "mem.mif" if this option is omitted.
-range *lo_adr hi_adr*	A pair of hexadecimal addresses, without a leading "0x", that specifies the limiting address range for data conversion. Only data within that address range, inclusive, is to be converted; data outside the range is ignored. For example, -range 2000 2fff indicates that only data in addresses 2000 through 2fff is to be converted.
-u[sage]	Indicates that basic usage of cnvrt2mif is to be displayed.
-v	Indicates that status messages are to be moderately verbose.
-V	Indicates that status messages are to be highly verbose.
-wnodup	Suppresses the generation of warning messages when multiple input records occupy the same address. By default, these messages are generated.

Example #1 — Converting S-records into a 32-bit wide single output file with no endian conversion

The input file srec4 contains two records with data from byte address 400 to 41f.

```
S3 15 00000400 101111231415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command specifies that the input file srec4 is to be converted to MIF format and written to a single 32-bit wide output file myoutput.mif, with no endian conversion.

```
% cnvrt2mif srec4 -devwidth32 -o myoutput
```

cnvrt2mif automatically recognizes the input file as being in Motorola S-record format. Because -noxsum was not specified, cnvrt2mif validates the checksums on each record. The output file myoutput.mif contains the following data:

```
100 / 10111213;
101 / 14151617;
102 / 18191A1B;
103 / 1C1D1E1F;
104 / 30313233;
105 / 34353637;
106 / 38393A3B;
107 / 3C3D3E3F;
```

Because this is a 4-byte (32-bit) device, the input record addresses (400-41f) have been adjusted to device indexes.

Example #2 — Converting S-records into two 16-bit wide output files with endian conversion

As in Example #1, the input file srec4 contains two records with data from byte address 400 to 41f.

```
S3 15 00000400 101112131415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command specifies that the input file srec4 is to be converted to MIF. The switch -devwidth 16 indicates that the output files are to be 16 bits wide; -devs 2 indicates that files are to be created for two devices (that is, two files are to be created). -flip32 indicates that there is to be a 32-bit (4-byte) endian conversion.

```
% cnvrt2mif srec4 -devwidth 16 -devs 2 -flip32 -o myoutput
```

The two output files are named myoutput_00.mif and myoutput_01.mif.

myoutput_00.mif contains the following data:

```
100 / 1312;
101 / 1716;
102 / 1B1A;
103 / 1F1E;
104 / 3332;
105 / 3736;
106 / 3B3A;
107 / 3F3E;
```

myoutput_01.mif contains the following data:

```
100 / 1110;
101 / 1514;
102 / 1918;
103 / 1D1C;
104 / 3130;
105 / 3534;
106 / 3938;
107 / 3D3C;
```

Because the two output files comprise a 4 byte wide memory image, the input record addresses (400–41f) have been adjusted to device indexes starting at 100.

Example #3 — Converting S-records into one 8-bit wide output file with offset

In this example, a memory device starts at 0x400 in the address space. The linker has generated an S-record file that contains the real addresses at which the processor in the design will address the contents of memory devices. The required outcome is that the indexes (that is, the internal memory relative addresses) must start at 0x0000, and only the first 8 bytes of the S-record file are to be placed in the MIF output file.

As in examples 1 and 2, the input file srec4 contains two records with data from byte address 400 to 41f, as follows:

```
S3 15 00000400 101111231415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command uses the -baseadr 400 option to specify that the image is to be offset by 0x400 bytes, and the -range 400 407 option to specify that only data in the range of input addresses 0x400 and 0x407 is to be included in the output file. The -devwidth 8 option specifies an 8-bit data width for the output file.

```
% cnvrt2mif srec4 -devwidth 8 -baseadr 400 -range 400 407 -o output
```

The MIF format output file is named output.mif and contains the following data:

```
0 / 10;
1 / 11;
2 / 12;
3 / 13;
4 / 14;
5 / 15;
6 / 16;
7 / 17;
```

Notice that the offsets in the output file start at 0 and only the data from input addresses 0x400 to 0x407 is included.

Example #4 — Converting S-records into one 32-bit wide output file with conversion to lower case

In this example, a post processing tool can read hexadecimal values of A through F only in lower case. Instead of using sed to filter the file, the -ml option of cnvrt2mif generates an output file in lower case.

Note

cnvrt2mif reads either upper or lower case hexadecimal characters from an input file record.

The input file infile2 contains the following data:

```
S1 13 0000 A0B1C2D3E4F5A6B7C8D9E0F1A2B3C4D5 7C
```

The following command generates a single 32-bit output file with alphabetic characters in lower case.

```
% cnvrt2mif infile2 -devwidth 32 -ml
```

Because no output filename was specified, the default filename, mem.mif, is used. The output file contains the following data:

```
0 / a0b1c2d3;
1 / e4f5a6b7;
2 / c8d9e0f1;
3 / a2b3c4d5;
```

Example #5 — Converting S-records into a single output file without checksum validation

In this example, the input file was generated by a tool that did not compute a correct checksum for each record. In its default mode, cnvrt2mif reads the file and generates a large number of checksum warning messages, which could obscure other, more serious messages that might be generated.

The following command uses the -noxsum option to suppress the checksum validation and the accompanying warning messages.

```
% cnvrt2mif input_7.srec -noxsum -o myoutput
```

If invalid checksums are the only warning conditions detected, using the -noxsum option causes cnvrt2mif to return an execution status value of 0, whereas with checksum validation enabled, cnvrt2mif returns an execution status value of 1.

Note

Although this example shows the use of the -noxsum option, you should use it only sparingly, and only after you have attempted to locate and correct the problem in the S-record or Intel Hex file generator. Disabling checksum validation could result in invalid data being placed in the output .mif file and subsequently loaded into your memories.

For additional information about cnvrt2mif, see “Translating Memory Image Files” in the *SmartModel Library User’s Manual*.

A
Reporting Problems

If you think your DesignWare Memory Model is not working correctly, check with your system administrator to see if you are using the latest version. It is possible that a more recent version of a model has the fix you need. Significant model changes are documented in the model history section at the end of each model's datasheet.

First, verify the version number of the model using the Browser tool ($LMC_HOME/bin/sl_browser) to access the model datasheet. The title banner at the top of the datasheet lists the model's MDL version number. Then compare reported fixes for subsequent versions of that model by reading the model history section at the end of the latest datasheet on the Model Directory:

http://www.synopsys.com/memorycentral

For more information on model history, refer to "Model History and Fixed Bugs" on page 195.

If you cannot find a more recent model version that solves the problem, contact Synopsys Customer Support — see "Getting Help" on page 17.

Checking Installation Integrity

If you suspect that your DWMM installation may be fault, you can use the swiftcheck tool to create an environment and status report for the model. See "Step 8 — Check Installation Integrity" on page 31 for instructions on using swiftcheck.

Using Model Logging

Before you contact Technical Support, generate a model logging file (mlog.log). Model logging captures all of a model's activity during simulation (that is, stimulus and response) in ASCII text format. Transmitting an mlog.log file to Technical Support will help ensure accurate diagnosis of the problem. Only one instance of one model can be logged at any one time and system performance degrades when you use model logging.

To generate a mlog.log file, create a file called mlog.cfg in the directory where you run the simulator. All models look for this file and read its contents, if it exists, to determine which model to log. If you need to log more than one model, see "Logging Multiple Instances" on page 194.

You can select a model for model logging in any of the following ways:

- Create an empty mlog.cfg file. If you do not specify a particular model, the first model loaded in a circuit is logged. (This can be handy if you have only one model in your design.)
- Specify a model by its model name. Put a line in the mlog.cfg file that follows this case-sensitive convention:

  ```
  %m <model_name>_mx
  ```

  For example:

  ```
  %m cy7c1324_mx
  ```

  This logs the first model of that name. This is a good method to use when the design has only one instance of a particular model type.
- Specify a model by its instance name. Put a line in the mlog.cfg file that follows this case-sensitive convention:

  ```
  %i <scope><instance_name>
  ```

  For example:

  ```
  %i u100
  ```

  The instance with instance name u100 is logged. Note that for some simulators the instance name includes the instance path.

During simulation, the specified model creates a file named mlog.log. This file contains all of the stimulus and response recorded at the model's ports during simulation.

Logging Multiple Instances

If you need to log more than one instance of a model in the design, reset the instance name specified in the mlog.cfg file and rerun the simulator for each instance you want to log. Save the mlog.log output file to another location before running the simulator.

Sending the Log Files to Customer Support

After you rerun your simulation to generate the model log files, tar those files, zip the tarball up using gzip, and send the zipped log files to Customer Support (sw_support@synopsys.com) as an e-mail attachment. Include your call number if you have one and a description of the problem in the body of your message.

Other Diagnostic Information

Depending on the type of model, you may also need to e-mail your memory image files in addition to the mlog.cfg file.

In all communications to the Synopsys Technical Support Center, please include a phone number where we can reach you.

Model History and Fixed Bugs

At the end of each DWMM datasheet is a model history section detailing significant model changes that occurred during the past year. If the model has not changed significantly in a year, its datasheet does not contain any model history entries. Significant changes cause the model to behave differently in simulation. Of course, this includes all model bug fixes. For more information on datasheets, see page 35.

Each change entry in the model history includes the following:

- Reference number
- MDL version of the model after the change
- MDL date of the change
- SRC version
- Problem and resolution descriptions

Model history entries look like the following example.

```
--------------------------------------------------------------------
Reference:: 41087

MDL Version:: 01002
MDL Date:: 13-June-1996

SRC Version:: v1.1

Problem::    The minimum high/low pulse width for CCLK in
             synchronous peripheral mode did not conform to revised
             vendor specifications.

Resolution:: Corrected the model.
--------------------------------------------------------------------
```

Model History Entry Field Descriptions

The “Reference::” Field

The “Reference::” field contains the internal number assigned to the specific change.

The “MDL Version::” Field

The “MDL Version::” field contains the model version after the change. Not all MDL version number changes are significant. Only changes such as bug fixes that affect model behavior are considered significant and generate model history entries. That’s why the model MDL version number listed in the title banner on the first page of a datasheet can be a higher number than the MDL version number listed in the latest model history entry for a model.

The “MDL Date::” Field

The “MDL Date::” field contains the publication date for the corresponding MDL Version of a model.

The “SRC Version::” Field

The “SRC Version::” field contains the internal model source code version after the change. Because not all MDL Version changes for a model involve changes to the source code, the same SRC Version number can appear in multiple model history entries for different MDL Version numbers.

The “Problem::” and “Resolution::” Fields

The “Problem::” and “Resolution::” fields briefly describe the user-visible symptoms of the problem and, if appropriate, what was changed to correct it.

B
Selecting the Model Version

Selecting Models in $LMC_HOME

You can install and maintain multiple versions of the same model can in the same $LMC_HOME. You select a specific version of a model to use in a design simulation using configuration (LMC) files.

The default model version is the most recently installed version. In cases where you do not want to use the latest installed version of a model, you can override the default model version by creating one or more custom configuration (LMC) files. The software locates the default and custom configuration files using the $LMC_HOME and $LMC_CONFIG environment variables. For more information about model versioning, refer to "Versioning" on page 42.

Selecting Tool Versions

For some DWMM tools, called model-versioned tools, the version of the tool that is used is determined by the model. You cannot change the versions of model-versioned tools because a particular version of a model may depend on a specific tool version to function properly.

For other tools, called user-versioned tools, you can select the version of the tool to use via the default and custom configuration (LMC) files, in the same way that you select different model versions.

For more information about tools, see "Library Tools" on page 185.

Configuration (LMC) Files

A configuration file — called an “LMC” file, for “List of Model Configurations” — contains a list of model names, tool names, and libraries, with a version number specified for each model and tool. There are two kinds of configuration files: default configuration files and custom configuration files. All configuration files must have the extension .lmc.

The following shows a typical configuration file:

```
%PLT hp700

# Models added by Sl_Admin: Fri Feb 23 15:56:24 1996

%EXE swiftcheck 01009
%EXE mi_trans 04059
%EXE ptm_make 01006
%MOD atv2500 01000
%MOD bt458 01000
%MOD c5c_c8c_2 01000
%MOD dm74s188 01000
%MOD ecl01 01000
%MOD gal18v10 01000
%MOD hm658128 01000
%MOD cy7c1324_mx 01002
...
%MOD ttl0 01000 7400 74LS00
 ...
%MOD windows 01000
%MOD z8536 01000
```

Default Configuration (LMC) File

The default configuration (LMC) file that comes with the DWMM package contains a list of all installed DWMM models, libraries, and their versions. The LMC file is platform-specific, and is typically named *platform*.lmc (for example, hp700.lmc). Normally, when a model version is installed in the library, the $LMC_HOME/data/*platform*.lmc file is updated with the most recently added model version. If, for example, Version 01002 has been more recently installed than Version 01004, then Version 01002 is the default version used even though Version 01004 has a higher version number.

Before using the Browser tool, you must specify the default configuration file for the Browser by setting your local $LMC_HOME environment variable to the install directory. The model versions specified in the default configuration file will be used by the simulator unless you define other model versions in one or more custom configuration (LMC) files.

For information about the Browser tool, see the *SmartModel Library User’s Manual*.

Configuration File Syntax

There are three commands that can appear in an LMC file, as follows:

%PLT *platform_name*

If this optional command is present, it indicates that a check is to be made to determine if the platform specified is the same as that on which the software is currently running. Examples of allowed values are hp700, sunos, and solaris. The first line in the above example,

```
%PLT hp700
```

indicates the hp700 platform. If PLT is absent, no checking is done.

Hint

If you want a single LMC file to be shared among several platforms, omit the PLT command.

%EXE *tool_name version*

This command specifies the versions of the most recently installed user-versioned DWMM tools. Examples of user-versioned tools include MemScope and swiftcheck:

For example, these lines:

```
%EXE swiftcheck 01009
%EXE memscope 01018
```

indicate that, when the tools are called, the versions that will be used are 01009 for swiftcheck, and 01018 for MemScope.

%MOD *model_name model_version* **[***alias***[***alias***]**

This command specifies the model name, model version, and any aliases that might apply to the model. In the example, the line

```
%MOD ttl00 01000 7400 74LS00
```

indicates model ttl00, version 01000, with aliases of 7400 and 74LS00.

Custom Configuration (LMC) Files

A custom configuration file contains a list of installed DWMM models and user-versioned tools to be used by the simulator. If you want to use tool or model versions that are different from those in the default configuration file, create one or more custom configuration files that specify the names and versions of those models and tools.

There are several reasons for creating and using a custom configuration file:

- Freezing your design, so that it always refers to the same model versions.
- Accessing a specialized model version that no one else in your group should use.
- Checking new models before releasing them.
- Archiving your design, along with the model versions used.
- Accessing an updated model version to use a new or revised function, when other design teams do not want to use the updated model.
- Reverting to a previous version of a tool.

LMC files are platform-specific and must have the extension .lmc. To use a custom configuration file, set the $LMC_CONFIG environment variable to the path of the .lmc file.

The model versions specified in custom configuration files are used by the simulator, overriding the versions of those same models that are specified in the default configuration file. However, the model versions you specify must be installed in the library at your location. To determine what versions of a specific model are installed in your library, use the model detail function.

Creating a Custom Configuration (LMC) File

To create a custom configuration file, follow these steps:

1. In your home directory, open a new file named (for example) *my_platform*.lmc. Although the file will be platform-specific, you do not have to use “*platform*.lmc”.
2. In another window, open a read only copy of the default configuration file, $LMC_HOME/data/*platform*.lmc.
3. In the default configuration file, search for the models and tools whose version numbers you want to change.
4. As you find each item, copy its version record (the entire line on which the item appears) into the new file.
5. In the new file, change the version numbers to the ones you want.
6. Save the file.

Using a Custom Configuration File

To use a custom configuration (LMC) file, set your $LMC_CONFIG environment variable to the path of your LMC file.

- If you have no other configuration file defined in $LMC_CONFIG, or if you have an older file defined but want the new file to replace the old one for this work session, enter the following on the command line:

```
% setenv LMC_CONFIG /user/johnq/newfilename.lmc
```

- If you have a configuration file defined in $LMC_CONFIG, and for this work session want to use the defined file *in addition to* the new configuration file, set $LMC_CONFIG to both file names, separated by a colon, in the order in which you want the files to be searched for models, as shown in the following example:

```
% setenv LMC_CONFIG/user/johnq/newfilename.lmc :
/user/johnq/oldfilename.lmc
```

C
Licensing Information

This appendix includes the following topics:

UNIX Licensing

DWMM applications use the Synopsys Common Licensing (SCL) scheme to control their usage. This section describes the elements that are unique to the DWMM implementation of SCL, and explains how to get licensing started.

Note

You cannot use a Linux workstation as a license server.

For details about SCL, see "Step 3 — Download and Install the FLEXlm License Software" on page 28. You can also find general SCL information on the web:

http://www.synopsys.com/keys

Obtaining License Server Hostid

In order to receive license keys from Synopsys, you must provide the hostid of your license server. Once the DWMM product is installed, perform the following steps to obtain license server hostid:

1. Set the LMC_HOME environment variable:

   ```
   % setenv LMC_HOME installation_path
   ```

   where *installation_path* is the location that was used to install an LMC_HOME directory structure on the license server.

2. Run the lmutil program:

   ```
   % $LMC_HOME/bin/lmutil hostid
   ```

   The program returns the hostid of the license server.

Creating a License File

The following procedure assumes that you have set the LMC_HOME environment variable to the DWMM installation location.

Create the license file as follows:

1. Create a file named memprolicense.dat in the $LMC_HOME/data directory using the format shown below (Synopsys Common Licensing daemon):

   ```
   SERVER hostname hostid 5305
   VENDOR snpslmd full_path_to_snpslmd
   INCREMENT feature_name snpslmd 1.000 expire_date token_count auth_code
      [options]
   ```

   Note

   full_path_to_snpslmd must be an absolute path name, *not* an environment variable.

   For a detailed description of the FLEXlm license file, refer to the FLEXlm documentation at $LMC_HOME/doc/flexlm/TOC.htm.

2. In the FEATURE or INCREMENT line, enter the *feature_name*, *expire_date*, *token_count*, and *auth_code* from your Authorization Certificate.

3. In the SERVER line, enter the *hostname* and *hostid*. The *hostid* is listed in your Authorization Certificate, but you need to provide the *hostname*. Also, note that some Authorization Certificates list a *portid* of 5305 (as shown in step 1) but you can substitute any unused *portid* for your license server.

4. In the VENDOR line, enter the full path to the license daemon:

   ```
   cl_root/platform/bin/snpslmd
   ```

   where *platform* is any valid platform supported by Globetrotter (see the FLEXlm documentation at $LMC_HOME/doc/flexlm/TOC.htm).

5. Include all your Synopsys snpslmd-based product and feature authorizations in a smartlicense.dat file.

6. Start the FLEXlm license server (DWMM requires FLEXlm v5.1+):

   ```
   $LMC_HOME/bin/lmgrd -c path_to_memprolicense.dat -l \
   path_to_memprolicense.log
   ```

   If you add the DWMM features to an existing license file, use the following command to shut down and restart the server to make sure a v5.1 server is running:

   ```
   $LMC_HOME/bin/lmutil lmdown -c path_to_memprolicense.dat
   ```

7. If you want to add features, edit the memprolicense.dat file and use the following command to make the server read the file again:

   ```
   $LMC_HOME/bin/lmutil lmreread -c path_to_memprolicense.dat
   ```

8. Before using the DWMM tools or models, you must set the SNPSLMD_LICENSE_FILE environment variable. This variable can contain the path to the license.dat file, or the port number and hostname of the server process. Port number selection is controlled by the SERVER record in license.dat.

   ```
   setenv SNPSLMD_LICENSE_FILE path_to_license.dat
   ```

   or

   ```
   setenv SNPSLMD_LICENSE_FILE port@hostname
   ```

How Licensing Works

- **License tokens used per simulation**

  License tokens are consumed per simulation, not per instance, so that multiple instances of a particular model in a given simulation consume only a single license token. Each simulation session requires license tokens for the models in that simulation. If you run multiple simulations concurrently, each simulation consumes one or more license tokens.

- **Licenses are release-independent**

  The feature lines related to DWMM do not use the feature version field. This means that new licenses are not required to license new releases of the software.

- **Merged licenses are supported on the same server**

  DWMM licenses can be merged with other FLEXlm features. However, all feature lines must have been generated for the same license server.

- **Six server hardware platforms**

  Synopsys ships the FLEXlm licensing software with the DWMM software. We ship FLEXlm binaries for six hardware platforms: Sun4 OS4 (Sun OS), Sun4 OS5 (Solaris), HP700, DEC Alpha, and IBM RS6000. You can run the license server on any of these platforms.

Note

Synopsys does not provide FLEXlm licensing software for Linux workstations.

- **Licenses expire at midnight**

  Licenses expire at midnight of the expiration date given in the feature line.

- **Number of tokens = sum of feature lines**

  DWMM features are additive. This means that the number of license tokens available for a particular feature is the sum of the individual feature lines in the license file.

- **License usage notes**
  - One license token is consumed per simulation process, not one license per instance.
  - In addition to the previously-mentioned license features, DWMM models can also be authorized by a MEMPRO-SIM license feature at simulation run-time.
- **License polling**

  You can use the environment variable DW_WAIT_LICENSE to turn license polling on. If DW_WAIT_LICENSE is set to 1, simulation waits until a license becomes available.

License Management Tools

The following table lists several FLEXlm licensing commands, and gives a short description of each.

Table 31: FLEXlm Command Descriptions

Command	Description
lmcksum	Performs checksum of the license file.
lmdown	Shuts down all license daemons.
lmgrd	Starts a flexible license manager daemon.
lmhostid	Reports the hostid of a system.
lmremove	Removes specific licenses and returns them to license pool.
lmreread	Tells the license daemon to reread the license file.
lmstat	Reports status on license manager daemons and feature usage.
lmutil	Runs a generic FLEXlm utility program.
lmver	Reports the FLEXlm version of a library or binary file.

See the *FLEXlm End User's Guide* for details, and for FLEXlm usage information. This manual is available at the Globetrotter web site:

http://www.globetrotter.com

License Unavailability

To prevent your simulation tool from continuing when no license is available for your DWMM models, use the NoLicenseFatal feature.

DWMM models (which are based on the SWIFT standard) take a command called NoLicenseFatal that can be set on or off (the default):

```
setenv LMC_COMMAND "NoLicenseFatal ON"
setenv LMC_COMMAND "NoLicenseFatal OFF"
```

When on, NoLicenseFatal causes a simulation session with DWMM models to send a fatal error message to the simulator if any DWMM model in the simulation fails to authorize, causing the simulation to exit when the first DWMM licensing error (or denial) occurs.

When off, the simulation will run, but any DWMM model without a license will have its outputs set to x.

Synopsys supports Scirocco, VCS, and MTI VHDL with this feature.

Because DWMM model licensing does not use SWIFT, licensing errors alone will not terminate the simulation even with NoLicenseFatal set to on. A testbench that contains a DWMM model will not terminate on a license failure in any condition.

Several simulators currently do not stop simulating when they encounter the fatal error caused by the license being unavailable when the NoLicenseFatal variable is set to on. This includes simulators which rely on the LMTV integration (MTI Verilog, NC-Verilog, and Verilog-XL) as well as at least one other simulator, NC-VHDL, which relies on a vendor provided integration. However, note that all simulators, including those that use LMTV, show a fatal error in the lmc_trace file when the NoLicenseFatal variable has been set to on.

The behavior of a memory model when licensing fails is normally to continue the simulation (although a number of “license-failure” messages may appear. You can change this behavior using the environment variables DW_NO_LICENSE_FATAL or LMC_COMMAND.

The simulation terminates immediately on memory model license failure in either of these circumstances:

- DW_NO_LICENSE_FATAL is set to “on”
- LMC_COMMAND contains the command “nolicensefatal on”

On (and off) are case-insensitive. Use these to terminate your simulation immediately whenever the DWMM models has a problem checking out a license.

D
Memory Image Files

You can preload DesignWare Memory Models from files, or dump model data to files, as long as the files have one of these supported formats:

- SmartModel Memory Image File (MIF)
- Verilog $readmemh (also readable by VHDL models)

The following formats are supported indirectly through the cnvrt2mif program. For details, see "Translating Memory Image Files with cnvrt2mif" on page 186.

- Motorola S-Record
- Intel Hex

For information on creating a data file in SmartModel MIF format, see "Creating a SmartModel MIF File." For information on creating a data file in Verilog $readmemh format, see IEEE Std 1364-1995 or the appropriate vendor documentation for your Verilog simulator. For information on creating a data file in Motorola S-Record or Intel Hex format, see the appropriate Motorola or Intel documentation, respectively.

Creating a SmartModel MIF File

This section describes how to create and work with SmartModel-format MIF files.

Memory Image File Format

Each MIF file contains one or more records, each of which specifies a data word to be written to a particular memory location (or range of locations).

Record Conventions and Rules

The records in the MIF file use these syntax conventions:

- Braces ({ }) indicate a list of one or more entries.
- Brackets ([ ]) indicate optional entries.
- *Italics* indicate variables for which you supply values.
- Fields are not case-sensitive.
- More than one record can appear on a line.
- The character "X" or "x" (case-insensitive) indicates an unknown value, and is legal only in a data word where the data is expressed in binary, octal, or hexadecimal (illegal in decimal).

MIF Record Syntax

Each record in a MIF file specifies an address, a data value to load at that address, and the number base for the data value:

```
{address1 [:address2] / base_specifier data_value;} [# comment]
```

where the arguments are as follows:

address1	Specifies the memory address (or the beginning address of a range) to which data will be written.
:*address2*	Specifies the ending address of a range. You can use a colon (:) or a hyphen (-) as a delimiter.

To translate an address in a MIF file to a column and row address, follow these steps.

1. Use Table 32 to find the total number of address bits (row and column bits combined) for your device:

Table 32: Bits in Row and Column Addresses

Device Size	Row Bits	Column Bits	Total Bits
4 MB	11	11	22
1 MB	10	10	20
256 KB	9	9	18
64 KB	8	8	16

2. Write the MIF address in binary form (converting if needed), and pad with leading zeroes so you have the correct *total* number of bits.
3. Divide the bits into two equal sets and convert each set to hexadecimal. The first set is the row address, and the second set is the column address.

Example 1

Suppose a 1-MB memory device uses a MIF address of 4834:

1. Convert 4834h to binary: 100 1000 0011 0100
2. Pad with zeroes to 20 bits: 0000 0100 1000 0011 0100
3. Divide into two equal groups, and convert to hexadecimal:

   00 0001 0010 (row) 00 0011 0100 (column)

The row address is thus 12h, and the column address is 34h.

Example 2

Suppose a 256-KB device uses a MIF address of 2405:

1. Convert 2405h to binary: 10 0100 0000 0101
2. Pad with zeroes to 18 bits: 00 0010 0100 0000 0101
3. Divide into two equal groups, and convert to hexadecimal:

   0 0001 0010 (row) 0 0000 0101 (column)

The row address is thus 12h, and the column address is 5h.

/base_specifier	A slash (/) separates the address specification from the data word. *base_specifier* is one of the following: `’b` = binary `’o` = octal `’d` = decimal `’h` = hexadecimal (default) You can also mix different base numbers within a record.
data_value;	Specifies the value of the data word to be written to the specified memory locations. A semicolon (;) defines the end of each record.
# ***comment***	A comment can be included in a record by using the pound sign (#). All information from the pound sign to the end of the line is treated as a comment.

Example 1

The following example shows how various constructs can be used or combined in a MIF. In the example, the width of the memory location is 8 bits.

```
0:3/0; #Colon separator for address range
4-6/0; #Hyphen separator for address range
’d7/’b10101110; #Address and data can use a different
                #numeric base
10/0; 11/’b10000000; #Two records on the same line
12:1e/’HxF; ’d31/’hX8; #Information is case-insensitive
20:7FF/4; #Load remaining addresses with 00000100
```

Example 2

When you specify the data value to load into memory, the safest practice is to specify values that match the width of the memory location; however, this is not required. If the data value has fewer bits than the memory location, the value is padded with leading zeros. If the data value is larger than the memory location, the model’s format error checking rejects the data.

The following example specifies that the hex value F (binary 1111) is to be loaded into memory location 0 (zero). If the memory location is 9 bits wide, the value entered is 000001111; if the location is 6 bits wide, the value is 001111; and so on.

```
0/F
```

Example 3

Unknown (x) values are most easily specified in binary; often the unknown represents a single bit.

In the following example, for an 8-bit memory location the binary value is loaded into 0F exactly as written; the hex value xF is loaded into FA as xxxx1111. For a 9-bit memory location, the binary value is loaded as 01010x0x1 and the hex value as 0xxxx1111.

```
OF/’b1010x0x1;
FA/xF;
```

Using cnvrt2mif

Because the MIF file format does not match other memory image formats created by third parties — specifically, Intel Hex and Motorola S-record formats — the cnvrt2mif command-line tool is provided so that you can quickly convert Intel Hex and Motorola S-record memory image files to MIF format.

cnrvrt2mif is located at $LMC_HOME/bin/cnvrt2mif, and contains extensive built-in help text, which you can access by executing cnvrt2mif with the -help option, as in the following example:

```
% $LMC_HOME/bin/cnvrt2mif -help
```

For cnvrt2mif syntax, description, and examples, see “Translating Memory Image Files with cnvrt2mif” on page 186.

Dumping Memory Data

Models that simulate internal memory locations can write their contents to an external system file called a dump file. You can write the contents of a model’s simulated memory locations to a dump file at any time during a simulation.

You can create a dump file by using MemScope (see “Dumping Memory to a File” on page 112) or a testbench command (see “mem_dump” on page 120 [HDL], “slm_mem_dump” on page 156 [C], or “inst.dump” on page 173 [VERA]).

The dump file format is the same as MIF file format — addresses and data are represented in hexadecimal, except that data is represented in binary if the data contains any unknown bits.

The memory dump allows you to eliminate the read cycles required to verify the success of a test. The size of the dump file is minimized by filtering data that remains in its initial or power-up state, and by writing out only one data line for contiguous addresses that contain the same data value.

For example, consider the following memory contents, 8 bits wide:

```
Addr 0 = 0
Addr 1 = 0
Addr 2 = 0
Addr 3 = 0
Addr 8 = 11111111
Addr 15 = 1100X1X0
```

All other addresses contain an initial value of X.

The dump file contents would then be this:

```
0:3/0; 8/FF; F/’b1100X1X0;
```

Note

If you subsequently load the dump file for the same instance of a memory model, you are guaranteed to put the memory back in exactly the same state it was in when it was dumped.

E
Writing Custom Models

Appendix Contents

Using Custom Models with DWMM

The DWMM software provides a custom model interface you can use to create a custom DWMM model, either from scratch, or by modifying a standard DWMM model.

The DWMM software also provides you with the following capabilities for working with custom models:

- **Dynamic Memory Manager** – Allocate system memory only when it is needed. Your testbench can also deallocate system storage when it is no longer required.
- **Testbench Interfaces** – Access your custom model with Verilog, VHDL, C, or VERA testbench commands.
- **MemScope Debugger** – Analyze the behavior of your custom model.

The DWMM software also includes a set of commands that enable you to access the model interface from your custom models. These commands are included in your custom model as global Verilog tasks and VHDL procedures.

Note

Before using any custom model interface functions, please review the standard testbench commands described in these chapters: “HDL Testbench Interface” on page 113, “C Testbench Interface” on page 139, or “VERA Testbench Interface” on page 167. These standard commands may solve your needs directly.

Writing a Custom Model

Your model will interact with the DWMM model interface through HDL task or procedure calls. To get access to the model interface, your model must contain the following:

Verilog

```
‘include "mempro_pkg.v"
```

VHDL

```
library SLM_LIB;
use SLM_LIB.mempro_pkg.all;
```

When the model first initializes (time=0 in the simulation), each instance of your model must register itself with the model interface:

```
memcore_instance_ext(addr_width, data_width, four_state, initial_state,
                     type, message_level, user_label, alias, handle);
```

This command passes information about the model to the model interface. The command returns an instance handle value which you store in a local variable and pass as an argument to all subsequent model interface calls.

Read and Write Cycles

The DWMM model interface contains two commands that support reading from and writing to memory addresses: memcore_read_ext and memcore_write_ext. These commands also provide information for transaction logging, as explained in “Working with MemScope” on page 218.

Preloading Data

If you want data in your model to be initialized from a file, use the optional memcore_load command. Like the HDL, C, or VERA testbench interfaces, the model interface supports load files in MIF or readmemh formats.

Message Level Control

If you want to enable control of messages from a testbench, you need to first declare a register that will contain the message mask:

Verilog

```
   reg [(‘SLM_MEMPRO_MSG_WIDTH - 1) : 0] msg_reg;
```

VHDL

```
variable msg_reg : std_logic_vector((SLM_MEMPRO_MSG_WIDTH - 1) downto 0);
```

The variable is passed as an argument to memcore_read_ext and memcore_write_ext where it is updated if a testbench command has changed the message control state. When your model issues a message, it should test the message state register to control message display:

Verilog

```
if (msg_reg[warning_bit] === 1’b1) begin
      $fdisplay(simOutputHandle,"WARNING: Address changed during write");
    end
```

VHDL

```
  if (msg_reg(warning_bit) = ‘1’) then
        assert("Illegal RAM address during Read: Read ignored", WARNING);
  end if;
```

Working with MemScope

You can record the transaction history of your custom models for use by MemScope just as you can for any DWMM model. (For details, see “Step 1 — Prepare Your Testbench for MemScope” on page 69.)

Use the memcore_info and memcore_register_cycle_type to provide the DWMM model interface with details about your model and the operations it can perform. You can define any number of cycles using memcore_register_cycle_type, but each must fall into one of three categories: SLM_READ, SLM_WRITE, or SLM_OTHER.

The memcore_info command allows you to input arguments that describe the memory class and initialization file, if any. If there is no memory initialization file, pass “.” for that argument; any string (including an empty one) can be provided for the class name.

A transaction history for MemScope is set up with multiple calls of memcore_register_cycle_type, which has arguments that specify the cycle descriptions to be registered. Then, the transaction history is accumulated with each call to memcore_read_ext or memcore_write_ext.

If you need to record a transaction that contains a cycle other than read or a write (such as a refresh, idle, or command cycle), use memcore_cycle_type.

Instantiating a Custom Model

Once you’ve created a custom model, you’ll need to instantiate it into your testbench and simulate the design.

Use the links below to view detailed instantiation instructions for your simulator in the *Simulator Configuration Guide for Synopsys Models*:

- “Using Custom DesignWare Memory Models with VHDL and Verilog Simulators”
- “Using Custom DesignWare Memory Models with VCS”
- “Using Custom DesignWare Memory Models with Verilog-XL”
- “Using Custom DesignWare Memory Models with NC-Verilog”
- “Using Custom DesignWare Memory Models with MTI Verilog”
- “Using Custom DesignWare Memory Models with Scirocco”
- “Using Custom DesignWare Memory Models with MTI VHDL”
- “Using Custom DesignWare Memory Models with NC-VHDL”
- “Using Custom DesignWare Memory Models with QuickSim II”
- “Using Custom DesignWare Memory Models with SystemC”
- “Using VERA with Custom DesignWare Memory Models”

Custom Model Interface Commands

This section describes the commands that make up the custom model interface.

Table 33 lists commands by operation. On the following reference pages, commands are arranged in alphabetical order.

Table 33: Custom Model Interface Commands by Operation

Function	Description
Configuration Commands	
memcore_info	Collects model configuration for MemScope reports.
memcore_register_cycle_type	Defines all read and write cycles used by the model.
memcore_cycle_type	Logs transactions not part of read or write cycles.
Creation and Initialization Commands	
memcore_init_complete	Indicates to MemCore that model initialization is complete.
memcore_instance_ext	Defines instance size, number of states, uninitialized value, type, messages returned, user ID, and alias; returns an instance handle.
Memory Location Access Commands	
memcore_read_ext	Reads data from a specified address in a memory; reads the message mask.
memcore_write_ext	Writes data to from a specified address in a memory; reads the message mask.
Memory Image File Commands	
memcore_load	Loads a memory instance from an external data file.
memcore_unload	Deallocates contents of a memory instance.

memcore_cycle_type

Logs transactions not part of read or write cycles.

Syntax

memcore_cycle_type (handle, cycle_number)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
cycle_number	Input — an integer that identifies the type of cycle, when you wish to log a transaction that is not part of a read or write cycle. The cycle_number must have been previously registered with the memcore_register_cycle_type command (see page 228).

Description

This command tells the DWMM model interface that the model is executing the specified cycle_number. This command *is not* required for basic model operation.

Examples

Verilog

```
memcore_cycle_type (sram8, 4);
```

VHDL

```
memcore_cycle_type (sram8, 4);
```

memcore_info

Collects model configuration for MemScope reports.

Syntax

memcore_info (handle, memoryfile, modelname, specfile, classfile)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
memoryfile	Input — a string containing the name of the memory load file, if any. If not used, this argument should contain a period (.).
modelname	Input — a string containing the model name. In DWMM models, this argument contains the text entered in the “Model Name” field and is used as the default name for the HDL model.
specfile	Input — a string containing the name of the file that defines the model. If not used, this argument may contain a null string. In DWMM models, this argument contains the name of the file that was used to generate the model.
classfile	Input — a string that identifies the model class (such as SRAM, SDRAM, FIFO, etc.). If not used, this argument can contain a null string.

Description

This command collects model information that MemScope can use to show model configuration: initialization file, name, model generation file, and memory class. Use memcore_register_cycle_type to define that types of transactions that this model performs. This command *is not* required for basic model operation.

Examples

Verilog

```
memcore_info(sram8, ".", "SRAM_Test4", "", "SRAM");
```

VHDL

```
memcore_info(sram8, ".", "SRAM_Test4", "", "SRAM");
```

memcore_init_complete

Indicates that model initialization is complete.

Syntax

memcore_init_complete (handle)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).

Description

This command indicates that initialization is complete. In the Verilog model, you must call this command from within a custom model; otherwise, the model hangs during simulation. You must place it in the initial block of the model, after memcore_instance_ext, after delta time, and before any memcore_load. If you use the optional command memcore_info for MemScope reports, you must place memcore_init_complete after memcore_info.

In the VHDL model, you must call this command from within a process statement, whose sensitivity list must contain a signal to be assigned the model's instance handle in the main model code. This signal assignment must occur after the call to memcore_instance_ext and before any call to memcore_load. If you use the optional command memcore_info for MemScope reports, the signal assignment must occur after the call to memcore_info.

Examples

Verilog

```
initial begin
   memcore_instance_ext(addr_width,data_width,four_state,default_data,
   memcore_type,message_level,model_id,model_alias,inst_handle);
   ...
   #0  memcore_init_complete (inst_handle);
   memcore_load(inst_handle, memoryfile);
   ...
end
```

VHDL

```
-- Signal declarations
signal initComplete : integer := -1;
...
process (initComplete)
begin
       if (initComplete /= -1) then
             memcore_init_complete(initComplete);
       end if;
end process;
...
-- Main model code
...
memcore_instance_ext(addr_width,data_width,four_state,default_data,
       memcore_type,message_level,model_id,model_alias,
       inst_handle);
...
initComplete <= inst_handle;
memcore_load(inst_handle, memoryfile);
...
```

memcore_instance_ext

Defines instance size, number of states, uninitialized value, type, messages returned, user ID, and alias; returns an instance handle.

Syntax

memcore_instance_ext (addr_width, data_width, four_state, initial_state, type, message_level, user_label, alias, handle)

Arguments

addr_width	Input — an integer containing the address bus width of the specified instance. The value must be between 2 and 64, inclusive.
data_width	Input — an integer containing the data bus width of the specified instance. The value must be between 2 and 2048, inclusive.
four_state	Input — an integer set to 1 for a four-state model (0, 1, X, and U states), or 0 for a two-state model (0 and 1 states).
initial_state	Input — a Verilog register or VHDL std_logic_vector containing the returned value when uninitialized memory locations are read. The value should have the same number of bits as data_width. If four_state is set to 1, initial_state may contain X and U bits.
type	Input — an integer value.

Note

The type argument can be any legal integer.

message_level	Input — an integer bitmap specifying messages to be reported by the model. See Table 34 on page 225 for a list of message levels.
user_label	Input — an integer used to identify the instance.
alias	Input — a string used to set an alias for the instance. Usually set to "." (period).
handle	Output — an integer containing an internally-generated identifier of the specified instance.

Description

This command defines address and data bus sizes, number of states (2 or 4), the uninitialized value (which must be compatible with the number of states), type, messages returned, user ID, and alias for a specified instance. This command also returns an internally-generated handle for the instance.

Attention
Do not call this command more than once for each DWMM model instance.

The message_level argument controls which messages the model returns, as specified by Table 34. The memcore_read_ext and memcore_write_ext commands can be used to read the returned messages bitmap. This command *is* required for all model interface operations and *must* be the first command issued by each model instance.

Table 34: The mask Argument and Associated Message Levels

Mask Value	Message Level
0 (1)	Error messages
1 (2)	Warning messages
2 (4)	Timing messages
3 (8)	X-handling messages
4 (16)	Info messages

To enable multiple message types, add the mask values for those types, and enter that sum for message_level.

For example, to enable error, warning, and X-handling messages, set message_level to 1 + 2 + 8 = 11.

Examples

Verilog

```
memcore_instance_ext(10,8,1,8'bxxxxxxxx,1,11,sram8,".",inst_handle);
```

VHDL

```
memcore_instance_ext(10,8,1,"UUUUUUUU",1,11,sram8,".",inst_handle);
```

memcore_load

Loads a memory instance from an external data file.

Syntax

memcore_load (handle, filename)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
filename	Input — a string containing the file name of a file with the memory image in SmartModel MIF or Verilog $readmemh format.

Description

This command initializes locations in the memory specified by instance, using the contents of filename. Any memory locations not loaded by filename retain their previous values. This command *is not* required for basic model operation.

Examples

Verilog

```
memcore_load(sram8, "/server2/user/jdoe/dwmm/sram8.mif");
```

VHDL

```
memcore_load(sram8, "/server2/user/jdoe/dwmm/sram8.mif");
```

memcore_read_ext

Reads data from a specified address in a memory; reads the message mask.

Syntax

memcore_read_ext (handle, address, data, cycle_number, message_level)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
address	Input — a Verilog register or VHDL std_logic_vector containing the memory address to be read.
data	Output — a Verilog register or VHDL std_logic_vector that receives the memory data value.
cycle_number	Input — an integer that identifies the type of read or write cycle. Created by memcore_register_cycle_type.
message_level	Output — a Verilog register or VHDL std_logic_vector containing a bitmap of enabled messages. Set with the memcore_instance_ext command. See Table 34 on page 225 for information on interpreting message_level values.

Description

This command returns the data stored at a specific location in the specified instance. This command also returns the current message mask. This command *is* required for basic model operation.

Examples

Verilog

```
memcore_read_ext(sram8, i_addr, i_data, 1, msg_reg);
```

VHDL

```
memcore_read_ext(sram8, i_addr, i_data, 1, msg_reg);
```

memcore_register_cycle_type

Defines all read and write cycles used by the model.

Syntax

memcore_register_cycle_type (handle, cycle_category, cycle_number, cycle_name)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
cycle_category	Input — an integer that identifies the cycle as read (0), write (1), or other (2).
cycle_number	Input — an integer that identifies the type of read or write cycle. Used by memcore_cycle_type, memcore_read_ext, and memcore_write_ext.
cycle_name	Input — a string that contains a description of cycle_define, used in transaction histories.

Description

This command declares names for cycles executed by the model. (Each cycle has a cycle_category of "read," "write," or "other.") An "other" cycle could be any cycle that isn't a read or a write, such as a refresh cycle.

Some examples are: Address-controlled read cycle, CE-controlled read cycle, WE-controlled read cycle, OE-controlled read cycle, CE-controlled write cycle, and WE-controlled write cycle.

This command *is not* required for basic model operation.

Examples

Verilog

```
memcore_register_cycle_type(sram8, 0, we_read_cycle, "WE Read Cycle");
```

VHDL

```
memcore_register_cycle_type(sram8, 0, we_read_cycle, "WE Read Cycle");
```

memcore_unload

Deallocates a specified address range.

Syntax

memcore_unload (handle, low_addr, high_addr)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
low_addr	Input — a Verilog register or VHDL std_logic_vector containing the lowest memory address to be dumped, inclusive.
high_addr	Input — a Verilog register or VHDL std_logic_vector containing the highest memory address to be dumped, inclusive.

Description

This command deallocates memory words in the range of low_address through high_address, inclusive for the specified memory instance. When deallocated, any read operation within the low_addr through high_addr range returns the default value.

This command can be used during a simulation run to recover system memory allocated for DWMM models when it is no longer needed. For example, memories that can be “reset” can use this command to enable that operation. This command *is not* required for basic model operation.

Examples

Verilog

```
memcore_unload (sram8, low_addr, high_addr);
```

VHDL

```
memcore_unload (sram8, low_addr, high_addr);
```

memcore_write_ext

Writes data to from a specified address in a memory; reads the message mask.

Syntax

memcore_write_ext (handle, address, data, cycle_number, message_level)

Arguments

handle	Input — an integer instance handle, as returned by memcore_instance_ext (see page 224).
address	Input — a Verilog register or VHDL std_logic_vector containing the memory address to be written.
data	Input — a Verilog register or VHDL std_logic_vector containing the data value to be written.
cycle_number	Input — an integer that identifies the type of read or write cycle. Created by memcore_register_cycle_type.
message_level	Output — a Verilog register or VHDL std_logic_vector containing a bitmap of enabled messages.

Description

This command writes data to a specific location in the specified instance. This command also returns the current message mask. This command *is* required for basic model operation.

Examples

Verilog

```
memcore_write_ext(sram8, i_addr, i_data, 1, msg_reg);
```

VHDL

```
memcore_write_ext(sram8, i_addr, i_data, 1, msg_reg);
```

Index