VCS®/VCSi™ Testbench Tutorial Suite

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (^{sm})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1

Introducing OpenVera Native Testbench

OpenVera Native Testbench (NTB-OV) is an HVL (hardware verification language) that enables you to set up constrained random testbench environment easily and helps you to detect bugs that you normally do not expect. By easily writing constrained random testbenches in NTB-OV environment you can solve hard verification problems faster and more effectively than by writing directed tests in Verilog.

The NTB-OV environment is plugged into VCS platform to make it a single unified platform or environment. Integration of this efficient and high-performance technology called Native testbench (NTB-OV) into VCS enables you to write testbenches in OpenVera verification language and simulate them in VCS along with the design(s). Using the integrated environment, you can both simulate your HDL designs and verify them with high-level testbench constructs.

NTB-OV is superior to directed tests because you can only find predictable bugs by writing directed tests whereas NTB-OV helps you find hard-to-find bugs using its constrained random simulation feature. Its seamless integration with VCS engine helps optimization of both your design and the testbench thereby providing superior performance. This single kernel solution for testbench and the design generally increases the performance by two times.

Note:

In order to use NTB with VCS you always have to specify the -ntb switch.

This tutorial introduces NTB-OV, followed by a description of some of its salient features.

High-Level Verification

Dramatic increase in the size and complexity of designs pose a significant challenge to traditional verification methodologies. The ever-increasing inter-module interactions and design interdependencies have made these verification methodologies inefficient and insufficient. Therefore, the current goal is to provide a means for achieving this functional validation in the most reliable, smart, efficient, and expeditious manner in order to deliver first-time-working silicon or systems on time.

The number one objective to reaching the above stated verification goal is to be able to find and fix all bugs in a design before tape-out. Synopsys' unified design and verification environment of VCS is uniquely geared towards catching such hard-to-find bugs efficiently

and early in the design cycle. The seamless integration with VCS's various smart-verification technologies makes the NTB-OV environment unmatched in its efficiency and speed.

NTB-OV is one of the many key technologies that constitute this environment. Through the use of complex synchronization and timing mechanisms, you can write concurrent processes, providing a mechanism to simulate a real and dynamic test environment. NTB-OV also supports object-oriented methodology, and provides the necessary abstraction level to develop reliable and reusable test environments. NTB-OV enables random stimulus generation and self checking, which helps increase the efficiency of the verification environment.

OpenVera NTB-OV

NTB-OV has several features built specifically to address your singular functional verification needs. They are:

- Concurrency and Control
- Random Stimulus Generation with Constraints
- Driving Stimulus and Self-checking
- Interfaces and Clock Domains
- Classes, Methods, and Properties: An Object-Oriented Methodology

The following section describes these features in detail.

Concurrency and Control

Concurrency enables you to spawn off multiple parallel processes from a parent process. It brings power and flexibility to your verification environment, which inherently requires the execution of many processes in parallel, both for efficiency and smarter inter-process interaction. A typical example would be to stimulate a design and check the outcome in parallel. This enables your testbench to react to the outcome in real time by enabling it to modify the stimulus (even before the simulation ends).

You enable concurrency through the fork-join construct, which spawns off multiple parallel processes. The type of join-back to the parent process depends on whether all, any, or none of the parallel processes are completed.

For inter-process communication, constructs such as mailboxes let any process send a message to any other process. A receiving process can also synchronize with a sending process by waiting for its message.

Constructs such as semaphores prevent several parallel processes from trying to access a common resource, such as a signal, by allowing access to only one process at any time.

Constructs such as triggers are used in process to trigger events. These triggered events are received by other processes with the help of constructs called syncs in order to synchronize with the triggering processes.

Other process-control mechanisms are provided to help the processes wait either on specific variables or until the completion of their child processes.

Random Stimulus Generation with Constraints

Verification is primarily concerned with testing unpredictable corner-case scenarios as opposed to obvious cases. Random stimulus inherently ensures testing these corner-case scenarios. Consequently, verification time and effort is spent in the most efficient and effective manner.

NTB-OV provides a very powerful mechanism to generate random stimulus. It is based on class-object randomization, which means random variables of a class-object are automatically randomized by a call to the predefined randomize method associated with the object.

Constraints further augment the randomization feature. Constraints are properties that define the boundaries within which the randomization feature works. They have two key advantages: (1) you can add weights, by changing the distribution function so that certain sets of stimuli occur more often than others, and (2) you can change constraints for the next state by using the current state of the design to change these weights.

Driving Stimulus and Self-checking

You use constructs called drives to drive an internal design signal or a primary input either directly into a design or through its top-level ports.

Similarly, checking the outputs from a design is achieved with constructs called expects which are concise one-line statements.

Expects are advantageous because they automatically cause verification error messages to be displayed when output values do not match expected values. They are designed to catch two typical errors:

- Detecting an expected value at an inappropriate time.
- Detecting an unexpected value at any given time.

Interfaces and Clock Domains

Interfaces in NTB-OV provide a mechanism to group design signals based on design functionality. You can have as many interfaces as needed as long as each is associated with a specific clock. Input signals are driven and output signals are sampled only at clock edges. Interfaces therefore provide separation and clarity between clock domains in a multi-clock design environment.

Classes, Methods and Properties: An Object-Oriented Methodology

You do not need any previous knowledge of or experience with object-oriented methodology to use NTB-OV, as object-oriented programming is very similar to HDL programming.

At its very basic level, object-oriented methodology compartmentalizes segments of code into what are called “Classes”. These classes are made up of two entities: “Properties” (data, variables) and “Methods” (function or task routines). Think of a class as an HDL module containing data and behavioral code. This methodology provides protection from other code segments that might be using the same data names.

You can instantiate a class just like an HDL module with the help of an instance called its “Object.” Methods of this object will operate on class data members. Private data members can only be accessed outside the class by these class methods. You can prevent accidental misuse or misapplication of class data in a particular segment of code by specifying methods that limit access to that data.

The implementation of object-oriented methodology in NTB-OV is clear and straightforward, assimilating and extending the best of the features found in C++ and Java. The methodology naturally lends itself to grouping related code and data of a testbench. These groups of data and related code are structured and thus easy to develop, understand, debug, maintain, and reuse. The result is a disciplined and systematic testbench structure that is not only useful for verifying small and medium designs, but also has the dramatic, powerful impact on your productivity as soon as designs become large and significantly more complex.

NTB-OV Tutorial Overview

This tutorial introduces you to the OpenVera Native Testbench (NTB-OV) technology. The tutorial uses a memory system design written in Verilog to explain the basic concepts of NTB-OV. This tutorial will take you through the major steps in writing a testbench in NTB-OV and running it with VCS along with the design. The tutorial:

1. Explains Memory system and its each component in detail.
2. Familiarizes you with the concepts of NTB-OV technology.
3. Teaches you how to write a basic testbench in NTB-OV for a design.

4. Explains how to verify the functionality of Memory system using NTB-OV.

This section contains the following sections:

- “Memory System” on page 43
- “Tutorial-design Directory Setup” on page 10

The approach adopted to verify the memory system in this tutorial is as follows:

1. First, you verify all sub-modules individually at the block level.
2. Then you integrate all sub-modules into a final design and verify it at the system-level.
3. To verify each block, the tutorial first teaches you how to write a corresponding testbench for the design.
4. You then compile the design and the testbench.
5. Finally, you perform a simulation run for verification.
6. Once block level verification is complete, you verify the complete chip functionality at the system level.

Memory System

This section introduces the design of a system comprising a memory, an arbiter, a controller, a system bus and CPUs that access the memory, all of which are used in this tutorial. It provides you an overview of the components of the system, describes how they interact with each other to complete the system and illustrates the basic structure of the files used for this tutorial.

The design used in this tutorial is a simple memory system for a two CPU machine. It consists of a system bus, a centralized round-robin arbiter, and a memory controller that controls four static SRAM devices. Figure 1-1 on page 9 shows the schematic of this system.

Figure 1-1 Memory System Schematic

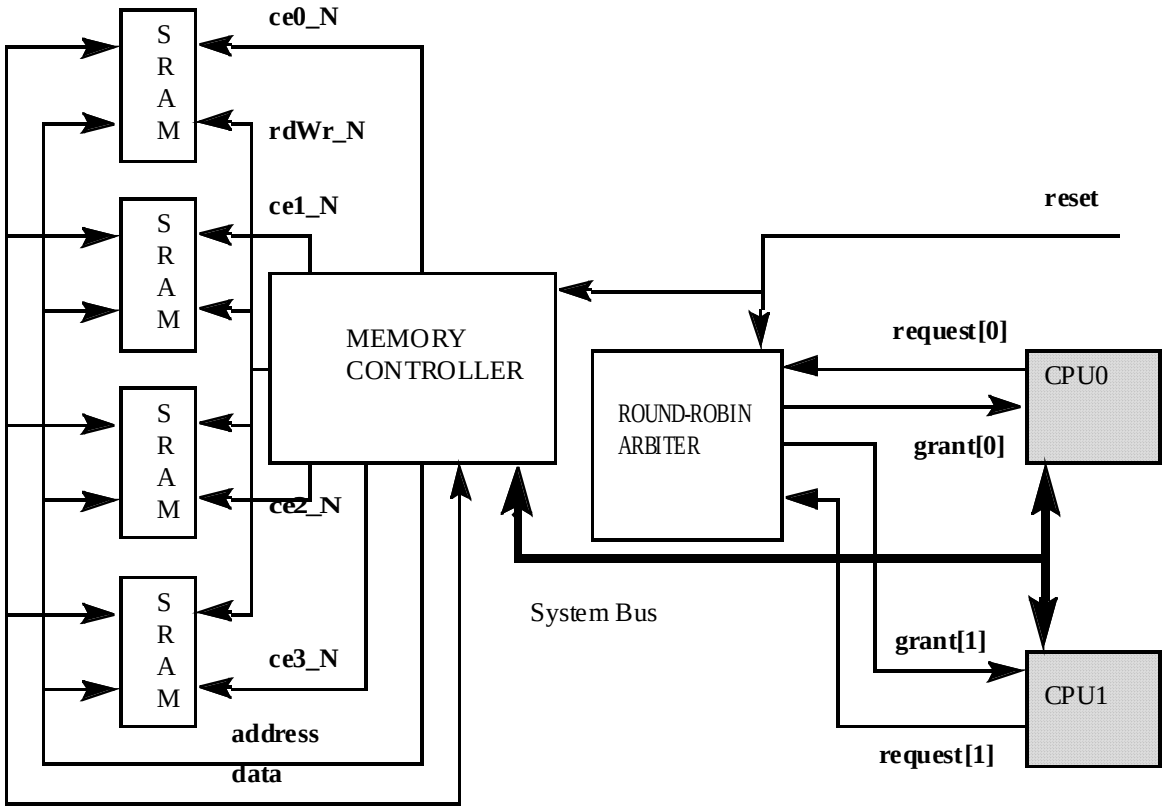


Figure 1-1 shows a schematic of the memory system. You may notice that the blocks labeled CPU0 and CPU1 are shaded. This is to indicate that they are not part of the system under test, but they will be modeled within the NTB-OV testbench. The signals between the CPUs and the rest of the system form the interface between the system under test and the “outside world.”

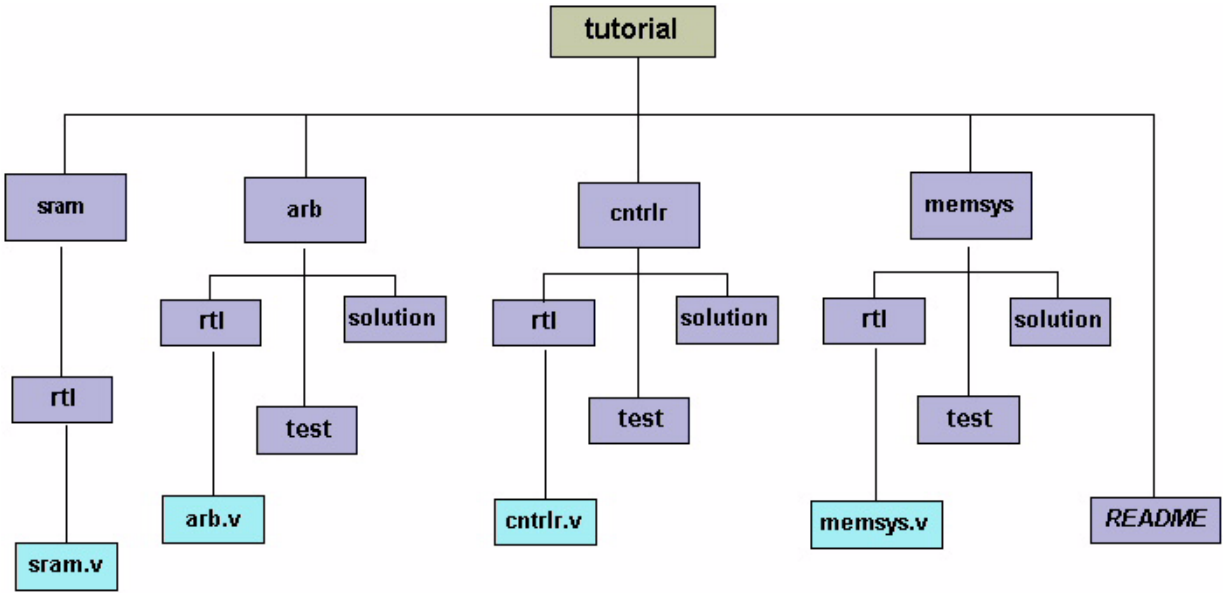
The memory system consists of the SRAMs, the Memory Controller, and the Arbiter. These are all defined in the HDL files of each sub-module.

Location of Files for This Tutorial

The files for this tutorial are in the following directory:

$VCS_HOME/doc/examples/testbench/ov/Tutorial

Tutorial-design Directory Setup



The NTB-OV tutorial tree contains the following directories:

- README — short description and file/directory index and listing of tools and versions used.
- sram — contains the memory RTL.
- arb — contains the submodule RTL and solution directory.

- cntrlr — contains the submodule RTL and solution directory.
- memsys — contains the top-level RTL netlist that integrates the entire memsys design and the solution directory.

Also note the following:

- Each `rtl` directory contains Verilog HDL code.
- Each `solution` directory contains the solution NTB-OV file, interface, header and script files. Refer to the `README` file in each directory to know how to run the scripts and see the expected simulation output of the solution files.

Each `test` directory inside the `arb`, `cntrlr`, and `memsys` directories is where you work on the tutorial exercises. You use this directory for creating your testbench, run scripts, compile and simulate the generated files and directories.

Arbiter

This section explains how to use NTB-OV to verify the arbiter in the example design. It briefly describes the functionality of the arbiter including a short timing and logic discussion. The section then describes the NTB-OV technology as it pertains to verifying the arbiter portion of the system.

This section explains how, specifically, using this technology, a testbench written in NTB-OV interacts with a Verilog design to drive signals, how the connections between the testbench and DUT are made, and how some of the basic signal operations behave. This section is divided into the following sections:

- “Testbench Overview” on page 14
- “Verifying the Arbiter” on page 18

Arbiter Overview

An arbiter is a basic building block that determines sharing of resources amongst many requesters.

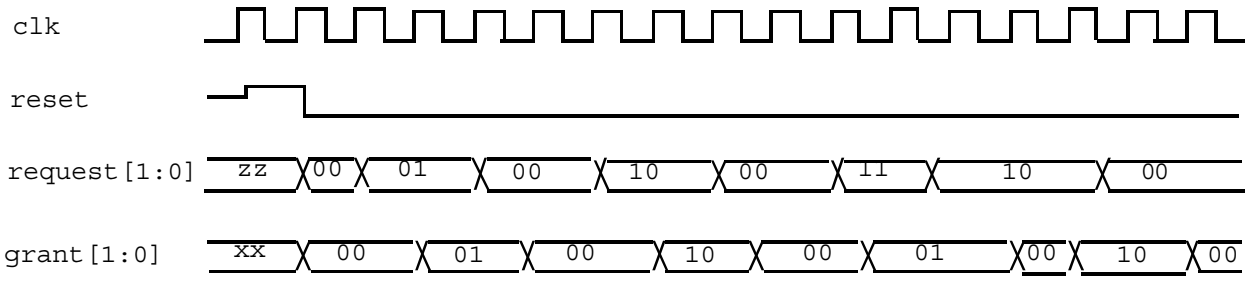
You will work inside the `tutorial/arb/test` directory.

Table 1-1 File and directory descriptions

File	Directory	Description
arb.v	tutorial/arb/rtl/	Contains RTL source code for arbiter.
arb.test_top.v	tutorial/arb/solution/	Contains solution top-level Verilog.
arb.vr	tutorial/arb/solution/	Contains solution testbench.
arb.if.vrh	tutorial/arb/solution/	Contains solution interface.
README	tutorial/arb/solution/	Contains details of running the solution testbench.
run.csh	tutorial/arb/solution/	Contains compile and run scripts for the solution.

Figure 1-2 illustrates the IO behavior using a timing waveform diagram.

Figure 1-2 Timing Diagram



The arbiter implements a round-robin arbitration algorithm between two CPUs. Each CPU can drive a request input signal (request[0] or request[1]). The arbiter queues the requests and determines which CPU will gain access to the system bus. The arbiter grants this access by asserting one of the grant output signals (grant[0] or grant[1]).

While the grant signal is asserted for a given CPU, the CPU continues to assert its request signal so that both the grant and the request signals for the CPU remain high while the CPU accesses the system bus. Once the CPU is done, it de-asserts its request signal and, on the subsequent clock cycle, the arbiter de-asserts its grant signal. With all the signals de-asserted, the arbiter can continue with the next request.

Getting Started

You just have arb.v file delivered to you as a part of the package. You need to use the template generator with the arb.v file to create additional files that you would modify them later to verify the arbiter.

You specify port and signal directions corresponding to OpenVera in arb.vrh file while you can connect Vera signals with Verilog signals in the test_top.v file. The clock generated by test_top.v is an input to both RTL and the testbench. Interface file contains information about port directions with respect to NTB-OV.

Testbench Overview

NTB-OV provides a Perl script called `ntb_template`. You use this script to generate template files that you later fill in with NTB-OV code as you complete the tutorial exercises.

First, you need to `cd` to the test directory, and use the following command line to use the template generator:

```
% ntb_template -t arb -c clk ../rtl/arb.v
```

In this example, the `-t` option defines the top-level name of the circuit under test, which is currently arb. The `-c` option defines the clock signal to be used in the generated interface. The specified file (../rtl/arb.v) is the RTL Verilog source code that is used with the Perl script to generate the template files. Note that the names of the generated files are derived from the top-level RTL Verilog filename.

A testbench suite comprises several key components:

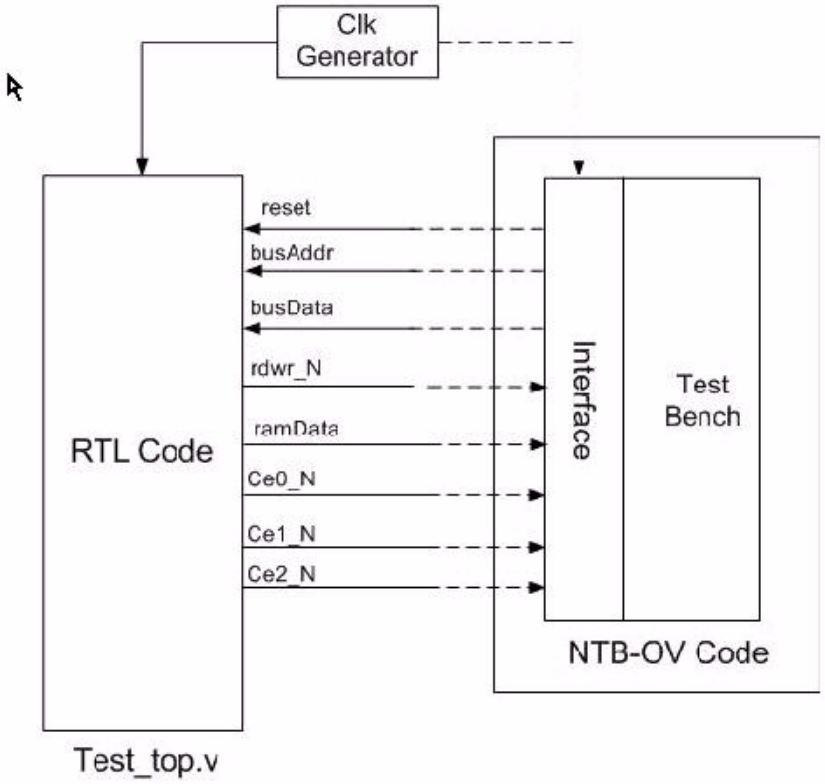
- **Testbench File** — Describes the testbench. In this example, this is just a single file. Based on the complexity of the design, you may have a number of files under the test environment as well as hundreds of test cases.
- **Interface Specification File** — Defines the signals for the testbench module. Typically the interface specification is contained in a file with the following name: filename.if.vrh.

- **Test_top File** (filename.test_top.v) — The top-level Verilog file that encapsulates the DUT and the testbench suite. It instantiates the DUT and the natively compiled testbench (as a shell), and handles system-clock generation and file dumping in Verilog. Figure 1-3 shows a basic schematic for this configuration.

Note:

Of the three generated files, you work only with the testbench file.

Figure 1-3 test_top.v Configuration Schematic



Note:

Alternatively, you can generate all these files manually. However, explaining how to generate these files manually is not within the scope of this document.

Invoking the NTB-OV template generator creates the following files:

- `arb.test_top.v`
- `arb.if.vrh`
- `arb.vr.tmp`

arb.test_top.v

The generated arb.test_top.v file contains the top-level Verilog code. The top-level code contains the signal and wire declarations that connect the testbench to the design. The declarations are made using the top-level RTL (arb.v). The top-level code also instantiates the testbench (a shell) and the design, and contains the clock-generator for the SystemClock (`clk`) that is passed both to the testbench interface and the design.

arb.if.vrh

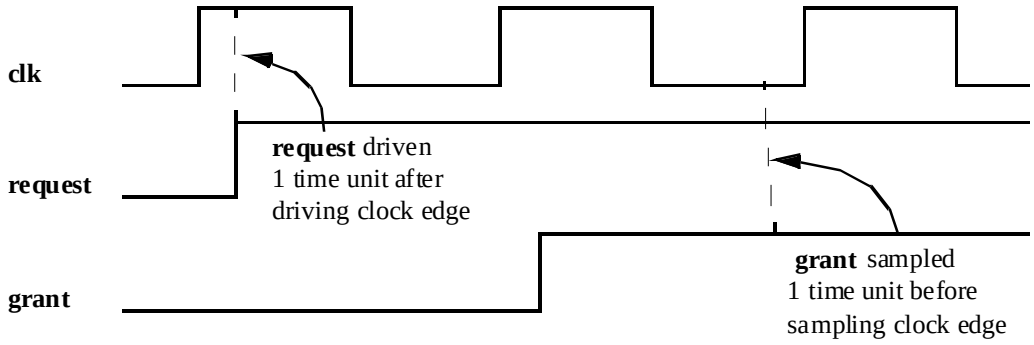
The generated arb.if.vrh file contains the testbench interface. The interface contains the testbench signal declarations for the arbiter. The signal names are taken from the top-level Verilog code of the arbiter (arb.v). Signals declared as outputs in the RTL are declared as inputs in the interface (and vice versa). Bidirectional signals remain bidirectional or inouts. Signals are given a standard skew of one time unit and are driven or sampled on the positive edge of the interface clock (`clk` in this example). You can customize the interface by editing this file.

Note that each interface has a clock associated with it. All signal operations occur on the corresponding interface clock edge.

All signal operations occur on the clock edge specified in the interface. If an output signal is marked PHOLD, all drives occur on the positive edge of the interface clock. Similarly, input signals marked PSAMPLE are sampled on the positive edge of the interface clock.

For example, given an interface with drives and samples (PSAMPLE and PHOLD) occurring on the positive clock edges and a skew of 1 and -1 respectively, the request and grant timing diagram (that shows only one request and one grant is) illustrated in the following diagram:

Figure 1-4 Timing Diagram



arb.vr.tmp

The generated arb.vr.tmp file contains the testbench template file. It also contains the interface-signal macros and preprocessor directives that include the arb.if.vrh interface file.

Verifying the Arbiter

First, verify the arbiter's reset. Then, verify that the arbiter handles simple requests appropriately and can grant access to one of the CPUs. Finally, check for the arbiter's proper handling of request sequences.

Verifying Reset

Verify if resets are working correctly. First assert the reset signal. After asserting the reset signal, hold the request signals inactive for each CPU (drive them to 0) and check that the grant signals are at their inactive states (0) after the reset.

Referencing Signals in the Testbench

To reference a signal, specify the interface name and the signal name. Using the `arb` interface, the `reset`, `request`, and `grant` signals are referenced as:

```
arb.reset
arb.request
arb.grant
```

Advancing Simulation to Next Value Change

To advance the simulation to the next value change of a specified signal, use the synchronize construct:

```
@(clock_edge signal_name); //clock_edge:posedge/negedge/no
edge specified
```

This advances the simulation to the specified edge of the signal. If the clock edge is omitted, it advances the simulation to the next signal-value change, which represents a sampling edge.

Asserting and Deasserting Signals

To assert and de-assert signals, use the NTB-OV drive construct:

```
@n signal_name = value;
```

The specified signal is driven to the appropriate value after *n* clock edges (defined as positive clock edges in the interface in our example). If the delay is omitted, the drive occurs on the next clock edge.

Checking Signal Values at Specified Times

To check that a signal has a specific value at a specified time, use the NTB-OV expect construct:

```
@n signal_name == value; //other operators except <= allowed
```

NTB-OV compares the specified signal to the given value after *n* clock edges. If the signal value is the same as the specified value, the simulation continues. If there is a mismatch, a verification error occurs with the following error message, but the simulation still continues:

```
VERIFICATION ERROR: Expect timeout Location: WAIT_ON_EXPECT
in program <shell_name>
```

where `shell_name` is `arb_test` in our example.

Verifying the Reset

First, rename `arb.vr.tmp` to `arb.vr`. In the `arb.vr` file, add the following code to verify the reset:

```
program arb_test{
arb.reset = 1; //assert reset
@2 arb.reset = 0; //de-assert reset after second posedge
of clock
@0 arb.request = 2'b00; // de-assert request
@1 arb.grant == 2'b00; //check grant is de-asserted after
```

```
next posedge
}
```

Note that the `request` and `grant` signals are 2-bit signals. Each bit of the signals must be de-asserted.

Compiling and Running Simulation with VCS

At this point, you should be in the tutorial/arb/test directory. With the code added to your arbiter testbench (arb.vr), run the simulation and test the results.

Compile the testbench using the following VCS command line:

```
% vcs -ntb arb.test_top.v ../rtl/arb.v arb.vr -Mupdate
+define+SYNOPSYS_NTB
```

Run the simulation by simply invoking the VCS `simv` executable as follows:

```
% simv
```

VCS automatically reports any errors found during simulation. For instance, change the line (expect) that checks the de-assertion of grant in the arb.vr file to the following:

```
@1 arb.grant == 2'b01;
```

Recompile and run the simulation again. The testbench now expects the `grant` signal to be asserted while the Verilog model continues to de-assert the signal as before. This results in an expect mismatch and a verification error as described on the previous page. The simulation will, however, proceed.

Note:

Remember to edit the testbench file arb.vr to correct this error before moving ahead with the tutorial.

Simple Request Verification

You can use NTB-OV to verify if the arbiter is handling simple requests correctly, monitor the `request` signals, check that the `grant` signal is set appropriately, and then check that the `grant` signal is de-asserted after the `request` is released.

Test for Simple Request by CPU0

To test that simple requests are handled correctly for CPU0, drive bit 0 of the `request` signal and then monitor bit 0 of the `grant` signal. Finally, de-assert both bits of the `request` signal and check that both bits of the `grant` signal are properly de-asserted, as shown in the following example.

```
program arb_test{
@0 arb.request = 2'b01; // assert bit 0 of request
@2 arb.grant == 2'b01; // check that bit 0 of grant is asserted
@0 arb.request = 2'b00; // de-assert bit 0 of request
@2 arb.grant == 2'b00; // check that both bits of grant are
de-asserted
}
```

Test for Simple Request by CPU1

To test that simple requests are handled correctly for CPU1, drive bit 1 of the `request` signal and then monitor bit 1 of the `grant` signal. Finally, de-assert both bits of the `request` signal and check that both bits of the `grant` signal are properly de-asserted, as shown in the following example.

```
program arb_test{
@0 arb.request = 2'b10; // assert bit 1 of request
@2 arb.grant == 2'b10; // check that bit 1 of grant is asserted
@0 arb.request = 2'b00; // de-assert bit 1' of request
@2 arb.grant == 2'b00; // check that both bits of grant are
de-asserted
}
```

Sequenced Request Verification

You can verify that sequences of requests are handled properly by using NTB-OV to verify a series of conditions:

- Assert both request signals and check that the correct grant is asserted (depends on which grant was previously asserted).
- Release the granted request and check that both grants are released.
- Then check that the other grant is asserted.
- Finally, release the other request and check that both grants are released.

Given this verification methodology, the code to check the arbiter behavior is the following:

```
program arb_test{
@0 arb.request = 2’b11; // assert both request signals
@2 arb.grant == 2’b01; // check for first grant
@0 arb.request = 2’b10; // de-assert corresponding request
@2 arb.grant == 2'b00;// check that grant de-asserts for 1
cycle
@1 arb.grant == 2'b10; // check that other grant is asserted
@0 arb.request = 2'b00; // de-assert corresponding request
@2 arb.grant == 2'b00; // check that both grant signals are
de-asserted
}
```

Doing Things in an Organized Manner with Tasks

After writing the above pieces of the testbench in the arb.vr file and verifying that they run, you probably have noticed that simulation takes minimum time to complete. Now, you can think of extending the simulation by checking repeatedly with sequences of reset

followed by request. This can be done by putting the code for reset and request in tasks and then calling these tasks from the main program in the testbench (`arb.vr` file).

The following code shows you how to use NTB-OV to write a task for verifying the reset:

```
task reset_test ()
{
     printf("Task reset_test: asserting and checking
          reset\n");
     arb.reset = 1; //assert reset
     @2 arb.reset = 0; // de-assert reset after second posedge
     @0 arb.request = 2'b00; // de-assert request
     @1 arb.grant == 2'b00; // check grant is de-asserted
                            // after next posedge
}
```

You can write a similar task called `request_grant_test` for verifying the request and grant sequence and then call both tasks from the main program of the testbench as follows:

```
program arb_test
{
integer count;
for (count = 0; count < 2000; count++)
     {
     reset_test();
     request_grant_test();
     }
}
```

Memory Controller

This section explains how to use NTB-OV to verify the memory controller in the example design. It gives you an overview of how the memory controller operates and describes some of the major NTB-OV features you can use to verify the controller. The section also describes virtual ports as well as synchronous and asynchronous events. These concepts are presented within the verification framework so that you can learn how to validate the memory controller. This section includes the following sections:

Memory Controller Overview

In the tutorial example design, the CPU accesses the bus through the arbiter. Once the CPU has access, it puts its request on the system bus. The memory controller acts on this request by reading data from the SRAM devices and returning data when necessary. To work with this part of the tutorial, first `cd` to the `tutorial/cntrlr/test` directory.

The following table describes the contents of the solution directory:

Table 1-2 File and directory descriptions

File	Directory	Description
cntrlr.v	tutorial/cntrlr/rtl/	Contains RTL source code for Verilog controller.
cntrlr.v	tutorial/cntrlr/solution/	Contains the solution top-level Verilog.
cntrlr.vr	tutorial/cntrlr/solution/	Contains solution testbench.
cntrlr.if.vrh	tutorial/cntrlr/solution/	Contains solution interface.
README	tutorial/cntrlr/solution/	Contains details of running the solution testbench.
run.csh	tutorial/cntrlr/solution/	Contains solution file to compile and run scripts.

The memory controller reads requests from the system bus and generates control signals for the SRAM devices attached to it. For read requests, the controller reads data and transfers it back to the bus and the CPU making the request. The address bus is 8 bits wide, which creates an address space of 256 bytes.

The controller supports up to 4 devices, allocating a maximum of 64 bytes of memory to each. The controller decodes the address and generates the chip enable for the corresponding device during a transaction. Figure 1-5 shows a diagram of how the testbench works with both the system bus and SRAM device signals.

Figure 1-5 Native Testbench/Memory Controller Interaction

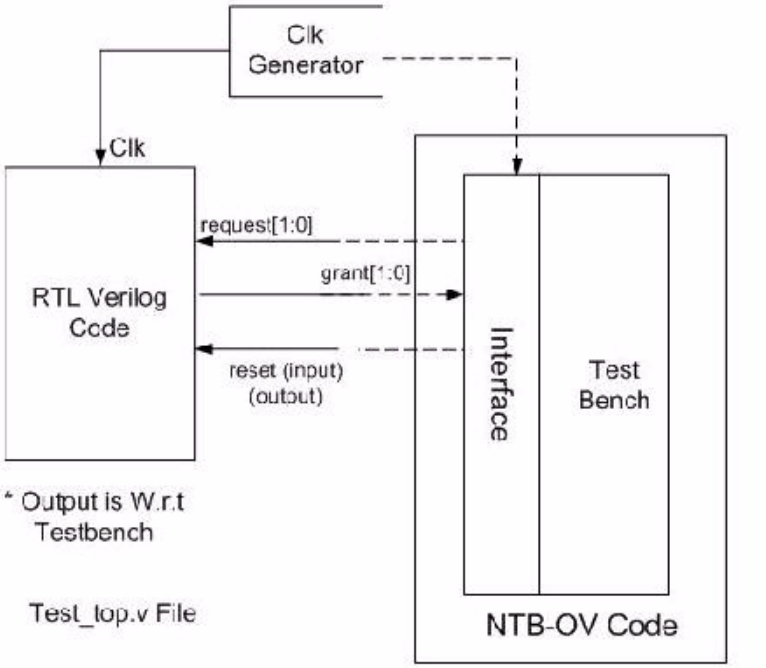
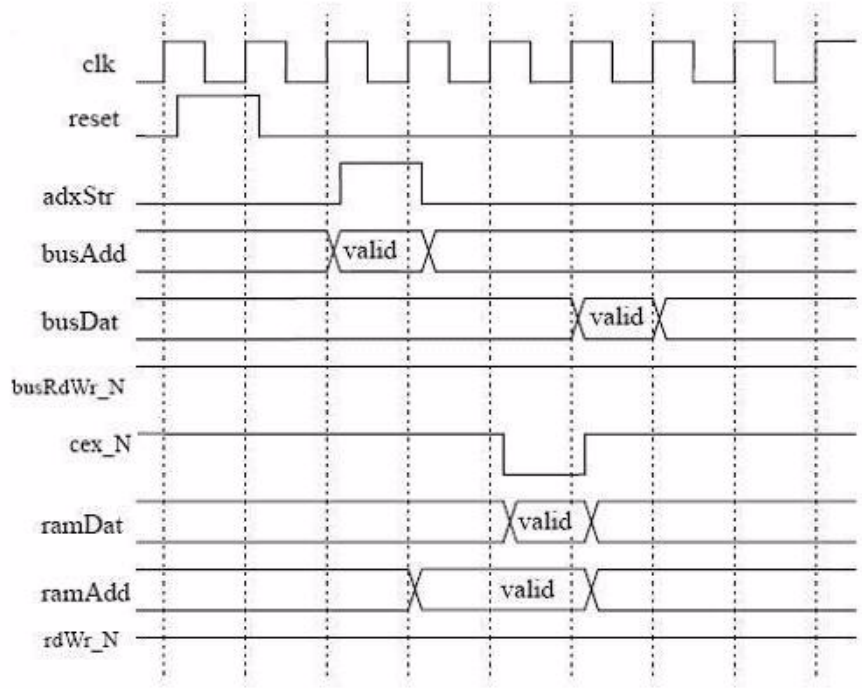


Figure 1-6 and Figure 1-7 show the timing diagrams for the memory controller's read and write operations respectively (note the signal names, as you will be using them in the verification process).

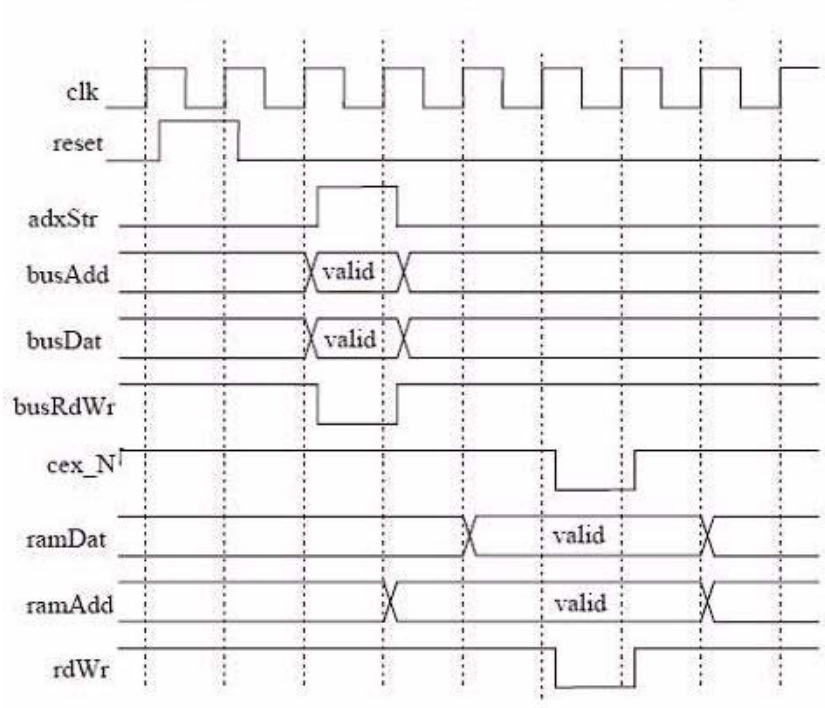
Figure 1-6 Memory Controller Read Operation Timing Diagram



Note:

In the case of cex_N, x refers to 0,1,2, and 3.

Figure 1-7 Memory Controller Write Operation Timing Diagram



Verifying the Memory Controller

First, verify the controller's reset by writing a task. Then, write tasks to check the read and write operations of the controller. Also check the integrity of the read and write operations. Finally, check the address map (all 256 addresses) exhaustively for the read and write functions.

To start the verification process, create the following template files using the template generator. Do a `cd` to the test directory, and use the following command line:

```
% ntb_template -tem -t cntrlr -c clk ../rtl/cntrlr.v
```

The template generator creates the following template files:

- `cntrlr.test_top.v` (top-level Verilog code)
- `cntrlr.vr.temp` (testbench file)
- `cntrlr.if.vrh` (interface file)

Controller Reset Verification

This section shows how you can use NTB-OV to verify the controller resets in the example design. First, assert the reset signal for at least one clock cycle. With the reset signal asserted, check that the memory enable signals are de-asserted.

Resetting the Controller

Rename `cntrlr.vr.tmp` to `cntrlr.vr`. Using simple drives, create a task as follows and add it to the `cntrlr.vr` file to reset the controller:

```
task resetSequence ()
{
         cntrlr.reset = 1'b1;
         @2 cntrlr.reset = 1'b0;
}
```

Verifying the Reset

Create another task as follows to verify that the memory enable signals (active_low) for all four SRAM devices are de-asserted (high) and add it to the `cntrlr.vr` file:

```
task resetCheck ()
{
         @0,10 cntrlr.ce0_N == 1'b1;
         cntrlr.ce1_N == 1'b1;
         cntrlr.ce2_N == 1'b1;
         cntrlr.ce3_N == 1'b1;
}
```

Invoking the Tasks

Having created the two tasks as described above, write calls to them in the main program in the `cntrlr.vr` file as follows:

```
program cntrlr_test
{
     @(posedge cntrlr.clk);
     resetSequence();
     resetCheck();
} // end of program cntrlr_test
```

`@(posedge cntrlr.clk)` advances the simulation to the posedge of the clock.

Compiling and Running Simulation with VCS

At this point, you should be in the tutorial/cntrlr/test directory. With the code added to your controller testbench (`cntrlr.vr`), run the simulation and check the results.

Compile the testbench using the following VCS command line:

```
% vcs -ntb cntrlr.test_top.v ../rtl/cntrlr.v cntrlr.vr -
Mupdate +define+SYNOPSYS_NTB
```

Run the simulation by simply invoking the VCS executable `simv` in the following way:

```
% simv
```

Driving System Bus for Read and Write Operations

To test the read and write capabilities of the controller, you can use NTB-OV to create two tasks that drive the bus for read and write operations.

Read Operation

Create a task that drives the read operation onto the system bus as specified in the timing diagram for the controller. The task should use an 8-bit bus address as an input. Given this requirement, the read operation task is the following:

```
task readOp (bit[7:0] adx)
{
     cntrlr.busAddr = adx;
     cntrlr.busRdWr_N = 1'b1;
     cntrlr.adxStrb = 1'b1;
     @1 cntrlr.adxStrb = 1'b0;
}
```

You pass the argument `adx` to the task. It then drives the `busAddr` signal to that value. Finally, it drives the `busRdWr_N` and `adxStrb` signals such that they match the timing diagram for the read operation of the controller.

Note:

Since this is a read operation, do not drive the data onto the bus and check for the expected data here.

Write Operation

Create a task that drives the write operation onto the system bus as specified in the timing diagram for the controller. The task should use 8-bit address and data buses as inputs. Finally, the task should leave the bus in an idle state (defined when `busData` is in high z and `busRdWr_N` is de-asserted). Given these requirements, the write operation task is the following:

```
task writeOp (bit[7:0] adx, bit[7:0] data)
{
     @1 cntrlr.busAddr = adx;
     cntrlr.busData = data;
     cntrlr.busRdWr_N = 1'b0;
```

```
    cntrlr.adxStrb = 1'b1;
    @1 cntrlr.busRdWr_N = 1'b1;
    cntrlr.busData = 8'bzzzzzzzz;
    cntrlr.adxStrb = 1'b0;
}
```

This task is passed with the argument `adx`. It then drives the `busAddr` signal to that value. Finally, it drives the `busData`, `busRdWr_N`, and `adxStrb` signals such that they match the timing diagram for the write operation of the controller.

You have just completed writing two tasks — Read and Write. Now, you need to call these two tasks in your testbench.

Implementing Virtual Ports

Using virtual ports, you can write a task or function once and reuse it many times with different interfaces of the design. In NTB-OV, a virtual port enables interface signals to be grouped into logical bundles. A virtual port is a set of generic port signal members that act as placeholders for interface signals. When needed, you can bind these ports to specific interface signals.

Defining Virtual Ports

Define virtual ports outside the main program block using this construct:

port `port_name`
```
{
port_signal_member1;
...;
port_signal_memberN;
}
```
port_name

Name of the port — must be a valid identifier.

port_signal_memberN

Name of the port signal — must be a valid identifier. Multiple port signal names are separated by semicolons (;).

Binding Virtual Port Signal Members to Interface Signals

You can use the NTB-OV `bind` construct to associate port signal members with interface signals. The bind defines a global static variable of type port_name to reference the interface signals. You must define binds outside of the main program block.

bind `port_name port_variable`

```
{
     port_signal_member1 interface_name.signal_name1;
  ........
  port_signal_memberN interface_name.signal_nameN;

}
```

port_name

The user-defined virtual port whose signal member names you want associated with interface signals.

port_variable

The name of the variable being declared.

port_signal_number

The name of the generic signal names you are including in the bind. Generally, all of the signals in the port are bound. However, you can bind selected signals if you want, and leave others unbound.

port_signal_number

The name of the interface to which you are binding the port signal members.

signal_name

The name of the signal you are binding to a particular port signal member. You can specify signal subfields using signal_name[x:y].

Referencing Ports and Binds

The global static variable defined in the bind construct is not directly passed to tasks/functions. First, you declare a port variable of the same type and initialize it to the global static variable.

port_name port_variable = initial_value;

port_name

The name of the port data type.

port_variable

The name of the port variable you are declaring.

initial_value

This is the static port variable defined in the bind construct. If it is not set, the *`port_variable`* has a NULL value.

You can pass global static variables defined in binds to tasks/functions through the port variable so that specific interface signals can be referenced and acted upon.

To reference individual interface signals within a tasks/functions, use the following construct:

`port_variable.$port_signal_member`

This references the port signal member that is associated with the interface signal.

Implementing Ports and Binds in the Memory Controller

Given the port/bind methodology presented here, let us define a device port for the SRAM parts (`ramAddr`, `ramData`, `rdWr_N`, and `ce_N`) as follows:

```
port device
{
     ramAddr;
     ramData;
     rdWr_N;
     ce_N;
}
```

After defining the virtual port, connect the port signals to the actual interface signals using the `bind` construct as follows:

```
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_SKEW #-1
#define INPUT_EDGE  PSAMPLE
#include "cntrlr.if.vrh"
port device {ce_N;
               ramAddr;
               ramData;
               rdWr_N;}
bind device device0
{
     ramAddr cntrlr.ramAddr;
     ramData cntrlr.ramData;
     rdWr_N cntrlr.rdWr_N;
     ce_N cntrlr.ce0_N;
}
program cntrlr_test
{
}
```

This bind construct results in the port variable `device0` of port type `device`. It connects the port signals to their corresponding interface signals. Note that the `ce_N` signal is connected to its device-specific signal. Similarly, construct binds for each device (`device1`, `device2`, and `device3`).

Verifying Read and Write Operations

The memory controller issues read and write operations to each of the four SRAM devices as shown in the earlier timing diagram. Create read and write tasks in your testbench file `(cntrlr.vr)` that check these operations. Earlier, you modeled the timing diagram exactly, cycle by cycle. Your approach now is to make use of timing windows, which enable you to specify ranges of time and event sequences.

Because of complex timing issues with the read operation, examine the write operation first. A discussion of the timing issues and the read operation follows.

Timing Windows

NTB-OV provides timing windows for its expect constructs. The syntax for the construct is:

```
@window signal_name == value;
```

The window of time for which the check is made must be in the form “x,y”. The check begins x clock edges after the current simulation time and continues for y clock edges after the check begins. If the x is omitted, that is, the window of time is of the form “,y”, the check begins immediately and lasts y clock edges after the check begins. However, NTB-OV disables the check as soon as the signal value

matches the expected value at any time within the window. This mechanism provides a means to evaluate a signal in a specified period of time.

Verifying the Write Operation

To verify the write operation, create a task that checks the SRAM write operation against the timing diagram provided. The task should have the `device_id` port variable as an argument so that we can pass in the signals on which we want the task to act. The task also has 6-bit address and 8-bit data buses as inputs. It must check that the SRAM signals are driven correctly, check that the address is the right address, and drive the data onto the `ramData` bus at the appropriate time. Given these requirements, the code for the write operation is the following:

```
task checkSramWrite (device device_id, bit [5:0] adx, bit
[7:0] data)
dx;

     @,2 device_id.$ramData == data;

     @1 device_id.$rdWr_N == 1'b0;
     device_id.$ce_N == 1'b0;
     device_id.$ramData == data;
     device_id.$ramAddr == adx;

     @1 device_id.$rdWr_N == 1'b1;
     device_id.$ce_N == 1'b1;
     device_id.$ramData == data;
     device_id.$ramAddr == adx;
}
```

The task first checks whether the address (`ramAddr`) is valid in a timing window that begins after the first rising clock edge and lasts for the next five rising clock edges. As soon as the address is found to be valid, the task checks whether the data (`ramData`) to be written is valid in a timing window that begins immediately and lasts for the

next two rising clock edges. At the rising edge after the data is found to be valid, the task checks whether the write (rdWr_N) and enable (ce_N) signals are asserted. At the same time, the task checks address and data again to see if they are still valid. Then, after another rising clock edge, the task checks whether the write and enable signals are de-asserted while checking the address and data for the third time to make sure they were valid.

Notice how the task accesses each SRAM instance by binding the port "device" to the interface signals specific to the instance using the task argument device_id. The SRAM instance that is accessed depends on the port variable passed to the task through the argument device_id. For example, if the port variable passed is device0, the fourth statement in the task is interpreted as cntrlr.ce1_N==1'b0, and so on.

Synchronous and Asynchronous Timing

By default, all NTB-OV signal operations are synchronous. That is, they occur on the clock edges specified in the interface specification. However, you can make NTB-OV signal operations asynchronous by adding the async keyword after the drive or expect operation as follows:

```
@(edge signal_name async); //advance to next edge of signal
     signal_name = value async; // drive new value immediately
     signal_name == value async;// execute expect expression
                                // immediately
```

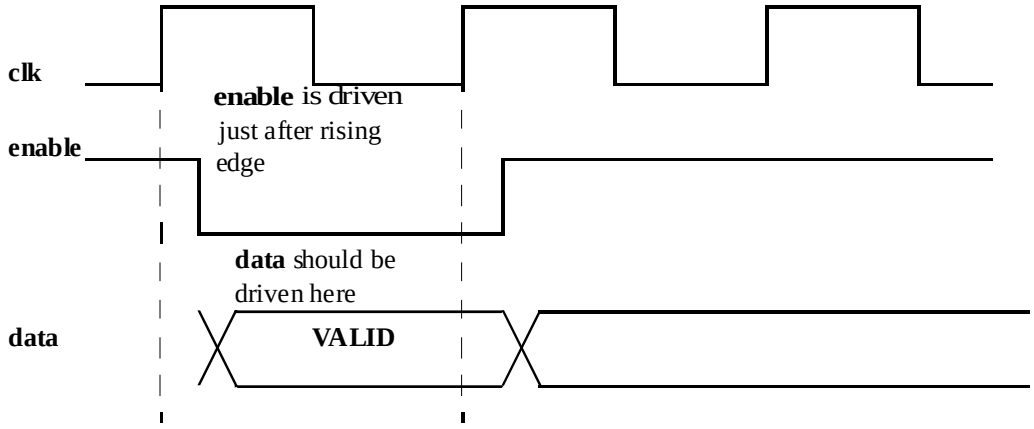
Note that the delays for the drive and expect operations are not used since they occur immediately.

The following code illustrates a few async drives and samples as you need to verify the read and write operations of SRAM using async drives and samples:

```
@(posedge main_bus.request async);
memsys.data[3:0] = 4'b1010 async;
data[2:0] = main_bus.data[2:0] async;
main_bus.data[7:4] == 4'b0101 async;
```

Verifying the Read Operation

You use NTB-OV to verify the read operation, check that the control signals are asserted, the correct address is driven by the memory controller, and the input data is driven as return data. However, an interesting timing issue arises in this case. Because of the driving skew, the `enable` signal is asserted just after the rising clock edge. This means that the data should be driven immediately and asynchronously upon sampling the enable signal. The following timing diagram illustrates this behavior:



Now, you can use an NTB-OV read task that checks the read operation against the timing diagram provided. The task must have the port variable `device_id` as an argument to allow it to access the enable signal of a particular SRAM instance. It also has 6-bit address and 8-bit data buses as inputs. The code for this task is the following:

```
task checkSramRead (device device_id, bit [5:0] adx, bit
[7:0] data)
{
```

```
    @(device_id.$ce_N async);
    device_id.$ce_N == 0 async;
    device_id.$rdWr_N == 1 async;
    device_id.$ramAddr == adx async;
    device_id.$ramData = data async;

    @1,3 cntrlr.busData == data;
    device_id.$ramData <= 8'bzzzzzzzz;
}
```

This task first advances the simulation to the change in the SRAM instance `enable` signal using the asynchronous construct. Next, it immediately checks that `enable` *is* 0, `rdWr_N` is de-asserted, and `ramAddr` has the appropriate value. After these checks, drive the data (`ramData`) immediately. Use the async construct here, so that the drive is executed immediately after the checks, and not on the next rising clock edge.

Next, check that the system bus has valid data in the specified window of time, which is three cycles after the next rising clock edge. Finally, one rising edge later, drive the data (`ramData`) back to high impedance. Note that the use of the `<=` drive operator indicates a non-blocking drive so that execution continues immediately.

Verifying SRAM Device

With the reset check completed, write the NTB-OV code to check the complete write operation of one of the devices. This includes driving the bus for the write operation in the `writeOp()` (Example on page 1-26) task, and checking the write operation in the `checkSramWrite()(Example  on page 1-30)` task. The code is the following:

```
    writeOp (8'h01, 8'h5A);
    checkSramWrite (device0, 6'b000001, 8'h5A);
```

This code drives the bus and then checks the write operation using the specified interface signals associated with `device0`. When checking other devices, remember that each device in the tutorial example design has a specific range of valid addresses as shown in the following table:

Device	Valid Address Range
0	0-63
1	64-127
2	128-191
3	192-255

Now add the NTB-OV code to check the write operations for the other devices. You can use the same tasks with different virtual ports and different address parameters.

Similarly, use the generic tasks to drive the bus for read operations and check the device read operations. Remember to check that the returned data matches the return data specified in the timing diagram.

The NTB-OV code for these checks is as follows:

```
readOp (8'h03);
checkSramRead (device0, 6'b000011, 8'h95);
```

This code drives the bus and then checks the read operation using the specified interface signals associated with `device0` (you can write similar code for other devices). Finally, the code checks the return data to see that it matches the correct value.

These tests only cover a subset of the valid addresses. To exhaustively test the entire range using these calls, you can write NTB-OV code that calls each task with every address. You can achieve this with the help of a for loop, as shown in the following code:

```
task checkAllAddresses () {
 device dev;
 integer i;
 bit [7:0] index;
 bit [7:0] data;

 printf("Task checkAllAddresses entered\n");

 for (i = 0; i < 256; i++) {
  printf ("Expect6: Index %0d time %0d\n", i, get_time(LO));
  index = i;
  data = 8'h5a;
  writeOp (index, data);
  case (index[7:6]) {
   2'b00: dev = device0;
   2'b01: dev = device1;
   2'b10: dev = device2;
   2'b11: dev = device3;
  }
  checkSramWrite (dev, index[5:0], data);
  readOp(index);
  checkSramRead (dev, index[5:0], data);
  //@1 cntrlr.busData == data;
  @0,1 cntrlr.busData == data;
  // @0,5 cntrlr.busData == data;
 }
}
```

Each iteration of this for loop acts on a different address. This loop drives the bus operation and then checks the SRAM operation using the subroutines defined previously. The case statement changes the interface signals on which the subroutines act so that they

correspond to the SRAM instance being accessed. This NTB-OV code then checks the bus data at the end of each iteration to monitor the return values.

Compiling and Running Simulation with VCS

With the code added to your controller testbench (cntrlr.vr), run the simulation and check results.

Compile the testbench using the following VCS command line:

```
% vcs -ntb cntrlr.test_top.v ../rtl/cntrlr.v cntrlr.vr -
Mupdate +define+SYNOPSYS_NTB
```

Run the simulation by simply invoking the VCS executable simv in the following way:

```
% simv
```

Memory System

In the previous sections, you verified the functionality of the arbiter and memory controller separately. Now, let’s examine how these components interact in a complete system. This section briefly describes the overview of the system, which includes the arbiter, controller, and SRAM devices.

The section also describes the higher level verification techniques used by NTB-OV. These include concurrency control mechanisms, such as semaphores, triggers, mailboxes, object-oriented methodology by way of classes, and random stimulus generation. Finally, this section shows you how to use these features to validate the memory system. This section includes the following sections:

- “Verifying the Memory System” on page 46

Memory System Overview

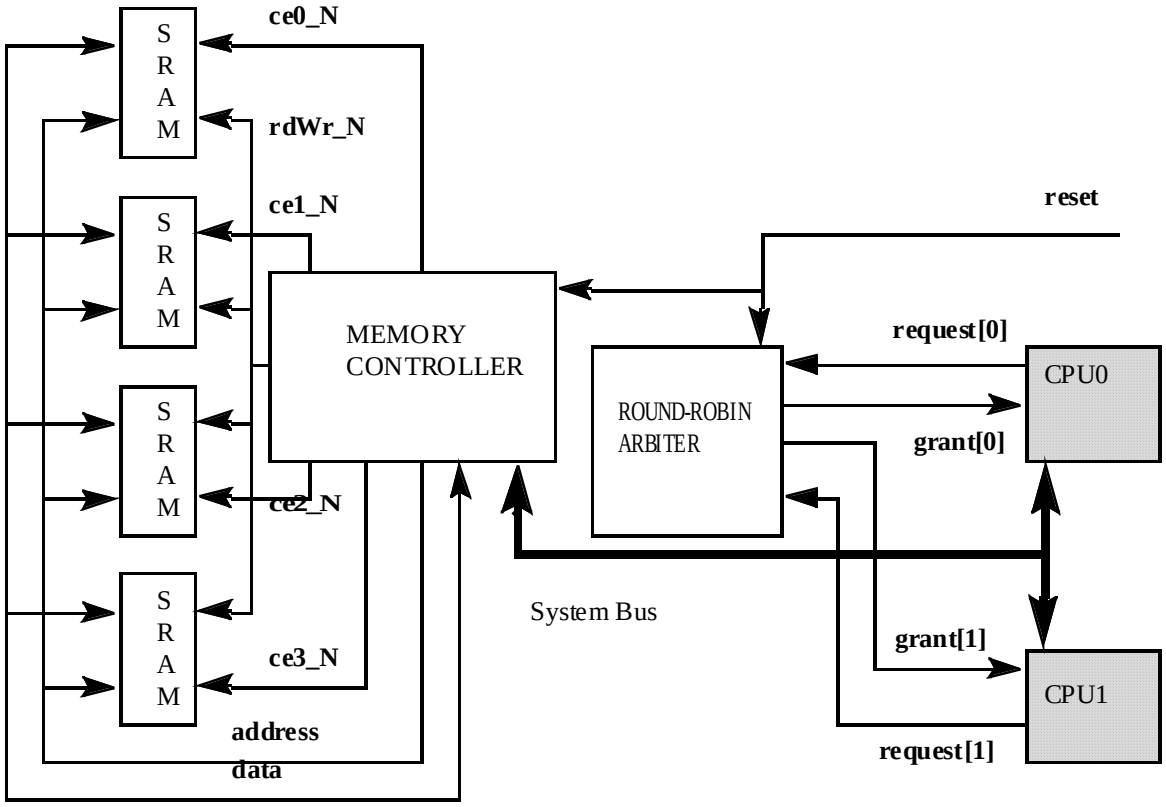
You will be working inside the tutorial/memsys/test directory, which includes the following files:

Table 1-3 File and directory descriptions

File	Directory	Description
memsys.v	tutorial/memsys/rtl/	Contains the Verilog memory system RTL source code.
memsys.f	tutorial/memsys/solution/	Contains links to all the RTL for sram, arb, cntrlr and memsys.
memsys.test_top.v	tutorial/memsys/solution/	Contains the solution top-level Verilog code.
memsys0.vr (semaphores) memsys1.vr (mailboxes/ triggers)	tutorial/memsys/solution/ tutorial/memsys/solution/	Contains the solution testbench.
memsys.if.vrh	tutorial/memsys/solution/	Contains the solution interface.
cpu.vr	tutorial/memsys/solution/	Contains the testbench code that mimics the CPUs.
README	tutorial/memsys/solution/	Contains the details of running the solution testbench.
run.csh	tutorial/memsys/solution/	Contains the solution compile and run script.

The memory system acts as a wrapper that instantiates the arbiter, memory controller, and four SRAM devices. In our system, the system bus is driven by two separate CPUs, with access granted through the arbiter. The memory controller handles the reading and writing of data to and from the system bus. A schematic of the complete system is provided in Figure 1-8.

Figure 1-8 Memory System Schematic



Verifying the Memory System

The methodology used to verify the entire memory system is broken down by the following tasks and concepts:

- **General Verification** — Port initialization, reset verification, and read/write operations.
- **Basic Concurrency and Control** — Concurrency between the two CPUs using fork/join, and control using semaphores to make sure the CPUs do not access the memory at the same time.

- **Object-oriented Programming** — Object-oriented programming using a class to model a CPU while making the model reusable.
- **Interprocess Communication** — Communication between the two CPUs using mailboxes and triggers to make sure that the simulation advances in lock-step fashion.

General Verification

To start the verification process, first create the following template files, as you did in the previous section, using the template generator:

- `memsys.test_top.v` (top-level Verilog code)
- `memsys.vr.tmp` (testbench file)
- `memsys.if.vrh` (interface file)

The command line for using the template generator is as follows:

```
% ntb_template -tem -t memsys -c clk ../rtl/memsys.v
```

Before you write any testbench code, rename the memsys.vr.tmp file to memsys.vr.

The general verification tasks include initializing ports, checking the reset procedure, and modifying the read and write operations previously developed for the memory controller. You can do this with the help of NTB-OV tasks. Finally, you can develop a testbench that checks both CPUs being run concurrently using multiple threads.

Port Initialization and Reset Verification

To check that the system is resetting correctly, we must first initialize ports and then go through the reset sequence. The NTB-OV code to do this is the following:

```
task init_ports ()
{
     @(posedge memsys.clk);
     memsys.request = 2'b00;
     memsys.busRdWr_N = 1'b1;
     memsys.adxStrb = 1'b0;
     memsys.reset = 1'b0;
}
task reset_sequence ()
{
     memsys.reset = 0;
     @1 memsys.reset = 1;
     @10 memsys.reset = 0;
     @1 memsys.grant == 2'b00;
}
```

Read and Write Operations

The bus used for accessing the system memory is very similar to the bus used in the memory controller in the previous section. Using the system bus, you can write the `writeOp()` task for the memory system as follows:

```
task writeOp (bit[7:0] adx, bit[7:0] data)
{
     @1 memsys.busAddr = adx;
     memsys.busData = data;
     memsys.busRdWr_N = 1'b0;
     memsys.adxStrb = 1'b1;
     @1 memsys.busRdWr_N = 1'b1;
     memsys.busData = 8'bzzzzzzzz;
     memsys.adxStrb = 1'b0;
}
```

During the read operation, the read operation task must also check for correct return data.

The new `readOp()` task with this data checking is the following:

```
task readOp (bit[7:0] adx, bit[7:0] data)
```

```
{
     @1 memsys.busAddr = adx;
     memsys.busRdWr_N = 1'b1;
     memsys.adxStrb = 1'b1;
     @1 memsys.adxStrb = 1'b0;
     @2,5 memsys.busData == data;
}
```

Multiple Threads or Concurrent Processes

You can use NTB-OV fork/join blocks to create concurrent processes. The syntax to declare a fork/join block is:

```
fork
     {statement1;}
     {statement2;}
     {...}
     {statementN;}
join wait_option
```

statementN

Can be any valid NTB-OV statement or sequence of statements.

wait_option

Specifies when the code after the fork/join block executes. The fork/join block can be either blocking or non-blocking. If it blocks, the code following the fork/join block does not execute until the code inside the fork/join thread returns. The *wait_option* must be one of the following:

all

When this option is used, the code after the fork/join block executes only after all of the concurrent processes have completed (default).

any

When this option is used, the code after the fork/join block executes right after any concurrent process within the fork/join completes.

none

When this option is used, the code after the fork/join block executes immediately without waiting for any of the fork/join processes to complete.

With the read and write operations defined, you want to set up your NTB-OV code such that each CPU issues a series of read and write requests to the memory system with random addresses and data. The two CPUs should operate concurrently or in parallel. Each CPU should use the `random()` system function to generate random addresses within the valid address space and an 8-bit data type. The CPUs should then request and access the bus, write the data to the bus, and release the bus (check for the release of the `grant` signal upon bus release). This sequence should be repeated 256 times using the `repeat()` flow control statement. Given these criteria, the NTB-OV code is the following:

```
random(12933);  // call random with seed
fork
{ // CPU0
     repeat(256)
     {
          randVar0 = random(); // get 32 bit random variable
          address0 = randVar0[13:6]; // get random 8-bit
                                     // address
          data0 = randVar0[29:22]; // get random 8-bit data
          @1 memsys.request[0] = 1'b1; // request the bus
          @2,20 memsys.grant == 2'b01; // check for grant
          writeOp(address0, data0); // issue write operation
          @1 memsys.request[0] = 1'b0; // release request
          @2,20 memsys.grant == 2'b00; // check for release
```

```
        @1 memsys.request[0] = 1'b1; // request again
        @2,20 memsys.grant == 2'b01; // check for grant
        readOp(address0, data0); // issue read operation
        @1 memsys.request[0] = 1'b0; // release request
        @2,20 memsys.grant == 2'b00; // check for grant
    }
}

{ // CPU1
    repeat(256)
    {
        randVar1 = random(); // get 32 bit random variable
        address1 = randVar1[13:6]; // get random 8-bit
                                   // address
        data1 = randVar1[29:22]; // get random 8-bit data
        @1 memsys.request[1] = 1'b1; // request the bus
        @2,20 memsys.grant == 2'b10; // check for grant
        writeOp(address1, data1); // issue write operation
        @1 memsys.request[1] = 1'b0; // release request
        @2,20 memsys.grant == 2'b00; // check for release
        @1 memsys.request[1] = 1'b1; // request again
        @2,20 memsys.grant == 2'b10; // check for grant
        readOp(address1, data1); // issue read operation
        @1 memsys.request[1] = 1'b0; // release request
        @2,20 memsys.grant == 2'b00; // check for grant
    }
}
join
```

This test works well in exhaustively checking the read and write operations for each CPU. However, because the CPUs are operating concurrently, problems can arise when each CPU accesses the same address space with different data. For instance, if CPU0 first writes to an address space and then CPU1 writes to the same address space, the data that CPU0 reads is different from what it expects (it reads the data that CPU1 wrote). This results in simulation failure because of the discrepancy between data read and

data expected. A solution to this issue is to use basic concurrency control as explained in “Introducing OpenVera Native Testbench” on page 1.

Object-Oriented Programming

Object-oriented programming (OOP) enables you to develop programs that are easier to debug and reuse by encapsulating related code (subroutines or methods) and data (properties) together into what is called a class. In this section, we show how you can implement NTB classes in the memory system using virtual ports to associate specific interface signals with each object (instance) of a class, how classes are constructed, how concurrency-control is achieved, how concurrent processes communicate, and how to get constrained random stimulus to drive the design.

Encapsulation

A class is a collection of data and a set of subroutines that act on that data. A class’s data is referred to as properties, and a class’s subroutines are referred to as methods. An instance of a class is called an object, and an object comprises the class’s properties and methods.

Class properties are instance-specific. Each instance of a class has its own copy of the variables declared in the class definition.

Because multiple instances of classes can exist, when calling a class method, you must identify the instance name for which the method is being called. This is because each method only accesses the properties associated with its object, or instance. So, when calling a method, you must use this syntax:

```
instance_name.method_name();
```

Constructors

Use the NTB-OV `new` construct to create objects and instantiate a class using the following syntax.

```
class_name instance_name = new;
```

This declaration creates an instance (called *`instance_name`*) of the class *class_name*. When this construction takes place, the method `new()`, if any exists within the class, is executed. By defining the task `new()` within a class, you can initialize the class upon construction or instantiation. Further, by passing arguments to the constructor, you can allow for runtime customizing of the object:

```
class_name instance_name = new(argument1, argument2, ...
argumentN);
```

Using this constructor, you pass the specified arguments to the task `new` within the class. The conventions for these arguments are the same as those of the usual NTB-OV subroutine calls.

Verifying Memory System using OOP

You need to create a .vr file and name it cpu.vr to work in it. You will be working in the cpu.vr file to implement the OOP (object-oriented programming) features.

You will define a class to represent a CPU. You instantiate this CPU twice because there are two CPUs. The next section illustrates with a code snippet how you must implement this class.

Implementing a Class

In the memsys example, since you have two CPUs (CPU0 and CPU1), you first declare a class called CPU, so that each CPU can be represented by an object of this class. This is done in the following manner:

```
class cpu
{

     //properties
     bus_arb localarb;
     local integer cpu_id;
     bit [7:0] address;
     bit [7:0] data;
     integer delay;
     //methods
     task new(bus_arb arb, integer id);
     task readOp();
     task writeOp();
     task request_bus();
     task release_bus();
     task delay_cycle();
     task randomize_tb();
```

You’ve just completed writing a skeleton of the CPU class. There are different tasks defined in this class. However, note that the file where you write your class skeleton must have the actual code there only. The actual code is discussed in the following section.

Interface-Signal Assignment

When implementing object-oriented concepts in your system, you should make specific interface-signal assignments to each instance or object of a class using virtual ports. Since each of the two CPUs is represented by an object of the class CPU and accesses the system bus through the common arbiter, we declare a virtual port

`bus_arb` and then, using it, declare two binds, one for use with each of the two class CPU objects. Thus, we ensure each CPU connects to the arbiter using its specific interface signals.

The NTB-OV code for declaring a virtual port is as follows:

```
port bus_arb
{
     arbreq;
     arbgrant;
}
```

Using this port declaration, we declare two binds, one for each CPU, as follows:

```
bind bus_arb arb0
{
     arbreq memsys.request[0];
     arbgrant memsys.grant[0];
}

bind bus_arb arb1
{
     arbreq memsys.request[1];
     arbgrant memsys.grant[1];
}
```

Depending on the CPU object that is invoked, the bind associated with that object gets passed to its class methods and determines which interface signals get affected.

Class Methods

In your class CPU, you must create the initialization method that is executed when the class is constructed. You must then create the read and write operation methods. It is also helpful to create methods to request and release the bus.

The initialization method should pass in the bind of type `bus_arb` (as declared above) and assign it to a local property. The code for the initialization method `new` is as follows:

```
task cpu::new (bus_arb arb, integer id)
{
     integer temp;
     printf("Constructing new CPU.\n");
     localarb = arb;
     cpu_id = id;
}
```

The read operation `readOp` must behave as before. Depending on the object of the class CPU that is invoked, the `readOp` class method only applies to the CPU associated with that object. The code for the `readOp` class method is the following:

```
task cpu::readOp ()
{
     printf("CPU %d readOp: address %h data %h\n", cpu_id,
          address, data);
     @1 memsys.busAddr = address;
     memsys.busRdWr_N = 1'b1;
     memsys.adxStrb = 1'b1;
     @1 memsys.adxStrb = 1'b0;
     @2,5 memsys.busData == data;
     printf("READ address = 0%H, data = 0%H \n", address,
          data);
}
```

The write operation `writeOp` must behave as before. Depending on the object of the class CPU that is invoked, the `writeOp` class method only applies to the CPU associated with that object. The conditional statement inside the method evaluates the local property `localarb` to which the bind was passed in the initialization process and thus the method is able to print which CPU is writing. The NTB-OV code for the `writeOp` class method is the following:

```
task cpu::writeOp()
```

```
{
     printf("CPU %d writeOp: address %h data %h\n", cpu_id,
          address, data);
     @1 memsys.busAddr = address;
     memsys.busData = data;
     memsys.busRdWr_N = 1'b0;
     memsys.adxStrb = 1'b1;
     @1 memsys.busRdWr_N = 1'b1;
     memsys.busData = 8'bzzzzzzzz;
     memsys.adxStrb = 1'b0;
     if (localarb == arb0)
          printf("CPU0 is writing \n");
     else if (localarb == arb1)
          printf("CPU1 is writing \n");
     printf("WRITE address = 0%H, data = 0%H \n", address,
          data);
}
```

Our `request_bus` method must assert the corresponding CPU's request line and then check for the appropriate grant line. You do this is using the associated bind, which was passed to the local property `localarb`. The code for the `request_bus` class method is the following:

```
task cpu::request_bus ()
{
     printf("CPU %d requests bus on %0s\n", cpu_id,
          localarb);
     @1 localarb.$arbreq = 1'b1; // request the bus
     @2,20 localarb.$arbgrant == 1'b1; // check for grant
}
```

Conversely, our `release_bus` method must release the corresponding CPU's request line and then check for the release of the appropriate grant line. You do this using the associated bind, which was passed to the local property `localarb`. The code for the `release_bus` class method is the following:

```
task cpu::release_bus ()
{
```

```
    printf("CPU %d releases bus on %0s\n", cpu_id,
        localarb);
    @1 localarb.$arbreq = 1'b0; // release the bus
    @1,2 localarb.$arbgrant == 1'b0; // check for grant
}
```

Random Stimulus Generation

Like before, you need to randomize the address that each CPU uses when accessing the SRAM. However, this time you can use the random stimuli for the address with a class method. You also want to introduce a randomized delay between CPU accesses, and further want to constrain this delay to within 10 clock cycles. So you also generate this constrained, randomized delay using modulo arithmetic with this class method. This is demonstrated by the following code for the `randomize_vlite()` class method:

```
task cpu::randomize_vlite ()
{
    bit [31:0] random_val;
    random_val = random();
    address = random_val[7:0];
    data = random_val[15:8];
    delay = {random_val[31:16] % 10}; // random delay with
                                      // constraint
    printf("CPU %d randomize_vlite: address %0h data %0h
        delay %0d \n", cpu_id, address, data, delay);
}
```

Note:

The method, `randomize_vlite`, is not a part of the CPU class method in the solutions directory. If you want to use `randomize_vlite()`, then you have to include the declaration as well as definition of `randomize_vlite()` task in the CPU class. Remember to remove the NTB-OV code that generates random addresses if you decide to use `randomize_vlite` method.

Finally, you write the method `delay_cycle()` to introduce the delay between CPU accesses. The code for the `delay_cycle` class method is the following:

```
task cpu::delay_cycle()
{
  printf("CPU %d Delay cycle value: %d\n", cpu_id, delay);
  repeat(delay) @(posedge memsys.clk);
  printf("delay = %d\n",delay);
}
```

Implementing Object-Oriented Programming

Before you use your objects, you must instantiate each CPU class object and invoke the initialization routines that specify the interfaces to be used by the objects, in the following manner:

```
cpu cpu0 = new (arb0, 0);
cpu cpu1 = new (arb1, 1);
```

Note:

> You must put these object instantiations inside the main program block in the memsys.vr file.

With your class CPU defined with the properties and methods described above, you can now write the same concurrent execution sequences for the two CPUs created earlier using NTB-OV fork/join constructs as follows:

```
fork
//fork CPU 0{
     repeat(256)
     {
          cpu0.randomize_vlite();
          cpu0.request_bus();
          cpu0.writeOp();
          cpu0.release_bus();
```

```
          cpu0.request_bus();
          cpu0.readOp();
          cpu0.release_bus();
          cpu0.delay_cycle();
     }
}

//fork CPU 1{
     repeat(256)
     {
          cpu1.randomize_vlite();
          cpu1.request_bus();
          cpu1.writeOp();
          cpu1.release_bus();
          cpu1.request_bus();
          cpu1.readOp();
          cpu1.release_bus();
          cpu1.delay_cycle();
     }
}
join
```

Note how the class property `address` is passed (using the instance name). Also, note the ease of reuse through invoking the class methods with the appropriate instance name.

Basic Concurrency Control

The two objects of the class CPU that were invoked concurrently in the previous subsection have the potential to access the same location in the SRAM memory if the random addresses generated for them happen to be the same. You can use semaphores to achieve concurrency control to prevent a potential conflict between the two CPU objects from leading to a data or resource hazard.

This section describes how you can synchronize and communicate between threads using semaphores and mailboxes. You can either choose a semaphore or a mailbox depending upon your requirement.

Note:

Use semaphores if you primarily want to synchronize the threads and control access to a shared resource. If you want to synchronize as well as send data across threads then use mailboxes.

The next section describes the following two topics:

- “Semaphores” on page 61.
- “Mailboxes” on page 65.

Semaphores

Conceptually, a semaphore is like a bucket containing a number of keys. No process can execute without first obtaining a key. Therefore, only as many processes as there are keys can execute at any time. All other processes must wait until the keys are returned.

To allocate a semaphore, you must use the `alloc` system function as follows:

```
function integer alloc(SEMAPHORE, integer semaphore_id,
integer semaphore_count, integer key_count);
```

semaphore_id

This is the ID number of the particular semaphore being created. It must be an integer. You should use 0, for the ID is then automatically generated by the simulator. Any other number explicitly assigns that number as the ID to the generated semaphore.

semaphore_count

This specifies the number of semaphores (buckets) you want to create. It must be an integer.

key_count

This specifies the number of keys initially allocated to each semaphore. It must be an integer.

The `alloc` function returns the base semaphore ID if the semaphores are successfully created. Otherwise, it returns 0.

To obtain a key, you must use the `semaphore_get` system task/function.

The prototype is:

```
task | function semaphore_get( integer wait_option ,
integer semaphore_id, integer key_count);
```

wait_option

The value of option must be one of the following predefined constants:

Table 1-4 Function and Task descriptions

Table 1-5

Subroutine type	Constant	Description
Function	NO_WAIT	This option specifies that the process will not wait for keys if there are no keys available and code execution will continue. Returns a 1 if there is a key, 0 otherwise.
Task	WAIT	This options specifies that the process will stop further code execution to wait for keys.

Semaphore_id

This ID specifies the semaphore from which to get keys.

Key_count

This specifies the number of keys being taken from the semaphore.

To return a key, you must use the `semaphore_put` system task as follows:

```
task semaphore_put(integer semaphore_id, integer
                   key_count);
```

Semaphore_id

This ID specifies the semaphore to which to returns keys.

Key_count

This specifies the number of keys being returned to the semaphore.

Implementing Semaphores

You can now modify the fork/join code that you wrote earlier to include semaphores as follows:

```
integer semaphoreId;
semaphoreId = alloc(SEMAPHORE, 0, 1, 1);
fork
{// fork process for CPU 0
     repeat(256)
     {
          cpu0.randomize_vlite();
          semaphore_get(WAIT, semaphoreId, 1);
          cpu0.request_bus();
          cpu0.writeOp();
          cpu0.release_bus();
          cpu0.request_bus();
          cpu0.readOp();
          cpu0.release_bus();
          semaphore_put(semaphoreId, 1);
          cpu0.delay_cycle();
     }
}

{// fork process for CPU 1
     repeat(256)
     {
          cpu1.randomize_vlite();
          semaphore_get(WAIT, semaphoreId, 1);
          cpu1.request_bus();
          cpu1.writeOp();
          cpu1.release_bus();
          cpu1.request_bus();
          cpu1.readOp();
          cpu1.release_bus();
          semaphore_put(semaphoreId, 1);
          cpu1.delay_cycle();
     }
}
join
```

Interprocess Communication and Synchronization

Up until now, you had both CPUs operating concurrently, with each going through the read and write cycles completely. Now, you may want to do things slightly differently by making CPU0 only write to the memory and CPU1 only read from the memory.

However, you must ensure that CPU0 writes before CPU1 reads and that the address generated by CPU0 is passed on to CPU1. Also, the data generated by CPU0 has to be passed on to CPU1 so that it can check for memory corruption by comparing the data read from the memory with that passed to it by CPU0. This interprocess communication in which CPU0 passes the address and data to the waiting CPU1 can be achieved with mailboxes. Further, you must also ensure that CPU1 finishes with its read operation and checks completely before CPU0 starts its next write cycle. This final synchronization can be achieved with triggers and syncs.

Mailboxes

A mailbox is an NTB-OV mechanism that you can use to exchange messages between processes. You can send data to a mailbox by one process and retrieve by another. Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, you can retrieve the letter (and any data stored within). However, if the letter has not been delivered when you check the mailbox, you must choose whether to wait for the letter or retrieve the letter on a subsequent trip to the mailbox. Similarly, NTB-OV mailboxes enable you to transfer and retrieve data in a controlled manner.

To allocate a mailbox, you must use the `alloc()` system function. The prototype is:

```
function integer alloc(MAILBOX, int mailbox_id, int
mailbox_count);
```

mailbox_id

The ID number of the particular mailbox being created. It must be an integer. You should generally use 0. When you use 0, NTB-OV automatically generates a mailbox ID.

mailbox_count

Specifies how many mailboxes you want to create. It must be an integer.

The `alloc` function returns the base mailbox ID if the mailboxes are successfully created. Otherwise, it returns 0.

The `mailbox_put` system task sends data to the mailbox. The prototype is:

```
task mailbox_put(int mailbox_id, scalar data);
```

mailbox_id

Specifies which mailbox receives the data.

data

Can be any general expression that evaluates to a scalar.

The `mailbox_put` system task stores data in a mailbox in a FIFO manner. Note that, when passing objects, only object handles are passed through the mailbox.

Use the `mailbox_get` function/task to retrieve data from a specified mailbox. The mailbox waiting queue is similar to the semaphore waiting queue as far as relative ordering of requests is concerned. The prototype is:

```
function|task scalar mailbox_get(integer wait_option, int
mailbox_id [, scalar dest_var [, keyword check_option]]);
```

wait_option

The option value must be one of the following predefined constants:

Table 1-6 Function and task descriptions

Subroutine type	Constant	Description
Function	NO_WAIT	De-queues mailbox data if it is available; else returns 0.
	COPY_NO_WAIT	Copies the mailbox data without de-queuing it if it is available; else returns 0.
Task	WAIT	Blocks the calling process until data is available in the mailbox; then de-queues the data and returns the number of entries in the mailbox.
	COPY_WAIT	Blocks the calling process until data is available in the mailbox; then copies the mailbox data without de-queuing it and returns the number of entries in the mailbox.

wait_option

Specifies from which mailbox data is being retrieved.

dest_var

The destination variable of the mailbox data.

Check_option

An optional argument that should be set to CHECK when used. It specifies whether type checking occurs between the mailbox data and the destination variable.

The `mailbox_get()` system task/function assigns any data stored in the mailbox to the destination variable and returns the number of entries in the mailbox, including the entry just received. If there is a type mismatch between the data sent to the mailbox and the destination variable, a runtime error occurs unless the CHECK option is used. If the CHECK option is active, a -1 is returned, and the message is left in the mailbox and is de-queued on the next `mailbox_get()` function call. If the mailbox is empty, the function waits for a message to be sent, depending on the wait option. If the wait option is NO_WAIT, the function returns a 0. If no destination variable is specified, the function returns the number of entries in the mailbox, but it does not dequeue an item from the mailbox.

Synchronization between Concurrent Processes

You can synchronize concurrent processes with each other using triggers and syncs. First, an event variable is declared. This event is then passed by one of the processes using a trigger. Synchronization of the other process to the one is achieved by using a sync.

Variables serve as the link between triggers and syncs. They are bidirectional as they can be used both to pass and receive triggers.

Triggers are used to pass or send events. A trigger is initiated by making a call to the system task trigger defined as follows:

```
task trigger (ON/OFF, event_name);
```

`event_name` is the name of the event that is to be passed. More than one event can be passed by any trigger.

The argument ON or OFF refers to the triggering on or off of the event. Refer to the *OpenVera LRM: Native Testbench* for other argument options instead of ON.

Syncs are used to receive events and thereby synchronize the receiving processes with the triggering processes. A sync is initiated by making a call to the system task sync defined as follows:

```
task sync (ALL, event_name);
```

event_name

> *`event_name`* is the name of the event that is to be received.More than one event can be received by any sync.

The argument ALL refers to the receiving of all events listed as arguments to the sync. Refer to the *OpenVera LRM Native Testbench* for other argument options instead of ALL.

Implementing Interprocess Communication and Synchronization

In order to avoid any conflicts between the two class CPU objects, the fork/join construct that was seen earlier is modified with the help of a mailbox, a trigger and a sync. Now, both objects CPU0 and CPU1 access the same memory location using the random address generated by CPU0. However, they do so not at the same time. While CPU0 writes to a random memory location, CPU1 waits for that random address and the data to be passed to it by CPU0 through a mailbox. CPU1 then accesses the same memory location

to read the data. Likewise, CPU0 waits to sync on a triggered event from CPU1 which indicates CPU1 has finished its write cycle and therefore CPU0 can start its next write cycle.

```
integer mboxId;
event CPU1done;
mboxId = alloc(MAILBOX, 0, 1);
fork
{// fork process for CPU 0
     repeat(256)
     {
          cpu0.randomize_vlite();
          cpu0.request_bus();
          cpu0.writeOp();
          cpu0.release_bus();
          mailbox_put(mboxId, cpu0.address);
          sync(ALL, CPU1done);
          trigger(OFF, CPU1done);
          cpu0.delay_cycle();
     }
}

{// fork process for CPU 1
     repeat(256)
     {
          mailbox_get(WAIT, mboxId, cpu1.address, CHECK);
          cpu1.request_bus();
          cpu1.readOp();
          cpu1.release_bus();
          trigger(ON, CPU1done);
          cpu1.delay_cycle();
     }
}
join
```

Constrained Randomization of Stimulus

Previously, you generated the `data,address` and `delay` properties of the class CPU using the **random()** function. Now, you will learn to do the same using the powerful automatic randomization

feature of NTB-OV for CPU class defined in the cpu.vr file. Further, you will also learn how to specify constraints to automatically constrain certain random properties (variables).

You can declare class properties as random using the `rand` declaration:

rand `data_type variable = initial_value;`

Variables declared as random within a class are randomized when the `randomize()` system function is called. Because `randomize()` acts as a class method, you must specify the instance for which the system function is called:

`function int object_name.`**randomize**`();`

object_name

The name of the object in which the random variables have been declared.

The `randomize()` class method generates random values for all random variables within the specified class instance. The `randomize()` method returns a 1 if it successfully sets all the random variables and objects to valid values. If it does not, it returns a 0. If an object has no random variables anywhere in its inheritance hierarchy (no random variables or sub-objects) or if all of its random variables are inactive, the `randomize()` function returns a 1.

Using random declarations, we declare our class properties `address`, `data` and `delay` as random as follows:

```
randc bit[7:0] address;
rand bit[7:0] data;
rand integer delay;
```

Each time an instance is randomized, the `address`, `data` and `delay` values for that instance are randomized. In particular, `address` has to be declared cyclic random (randc), for we want it to cycle through all the 256 memory addresses without repetition.

Note that there are no restrictions on the value that `delay` can assume because it is declared as a random integer. We can implement constraints on the values that random variables can assume using the **`constraint`** construct:

constraint `constraint_name { contraint_expressions }`

constraint_name

The name of the constraint block.

constraint_expression

The conditional expression that limits random values. It is a series of expressions that are enforced when the class is randomized.

To limit the values of delay to those between 0 and 10, we define this constraint within the class:

```
constraint del_lt10
{
     delay < 10;
     delay >=0;
}
```

Implementing Constraints and Randomization

First, introduce the changes described in the previous subsection in the declaration of the CPU class. This also involves removing the `randomize_vlite()` method. The CPU class would now appear as the following:

```
class cpu
{
     //properties
     bus_arb localarb;
     local integer cpu_id;
     randc bit [7:0] address;
     rand bit [7:0] data;
     rand integer delay;
     constraint del_lt10
     {
          delay < 10;
          delay >= 0;
     }
     //methods
     task new(bus_arb arb, integer id);
     task readOp();
     task writeOp();
     task request_bus();
     task release_bus();
     task delay_cycle();
}
```

Now, make changes to just one statement inside the fork-join construct from the subsection on “Implementing Interprocess Communication and Synchronization” on page 69. The task call to the `randomize_vlite()` class method is to be changed to a system call to the predefined system class method randomize to automatically randomize all variables declared as rand in the declaration of the CPU class. Any constraints defined on the rand

variables are also automatically taken into account when randomizing those variables. With these changes, the fork-join construct will appear as the following:

```
integer mboxId, randflag;
event CPU1done;
mboxId = alloc(MAILBOX, 0, 1);
fork
{// fork process for CPU 0
     repeat(256)
     {
          randflag = cpu0.randomize();
          cpu0.request_bus();
          cpu0.writeOp();
          cpu0.release_bus();
          mailbox_put(mboxId, cpu0.address);
          sync(ALL, CPU1done);
          trigger(OFF, CPU1done);
          cpu0.delay_cycle();
     }
}
{// fork process for CPU 1
     repeat(256)
     {
          mailbox_get(WAIT, mboxId, cpu1.address, CHECK);
          cpu1.request_bus();
          cpu1.readOp();
          cpu1.release_bus();
          trigger(ON, CPU1done);
          cpu1.delay_cycle();
     }
}
join
```

Finally, you can structurally make your code look more elegant by encasing the code for the fork/join in a task called `check_all` and then calling this and other tasks from inside the program `memsys_test` in the memsys.vr file as follows:

```
task check_all () {

  integer mboxId, randflag;
  event CPU1done;

  printf("Task check_all:\n");
  mboxId = alloc(MAILBOX, 0, 1);

  fork
    {// fork process for CPU 0
      repeat(256) {
        randflag = cpu0.randomize();
        cpu0.request_bus();
        cpu0.writeOp();
        cpu0.release_bus();
        mailbox_put(mboxId, cpu0.address);
        mailbox_put(mboxId, cpu0.data);
        mailbox_put(mboxId, cpu0.delay);
        sync(ALL, CPU1done);
        trigger(OFF, CPU1done);
        cpu0.delay_cycle();
      }
    }

    {// fork process for CPU 1
      repeat(256) {
        mailbox_get(WAIT, mboxId, cpu1.address, CHECK);
        mailbox_get(WAIT, mboxId, cpu1.data, CHECK);
        mailbox_get(WAIT, mboxId, cpu1.delay, CHECK);
        cpu1.request_bus();
        cpu1.readOp();
        if (memsys.busData == cpu1.data)
          printf("\nThe read and write cycles finished
successfully\n\n");
        else
          printf("\nThe memory has been corrupted\n\n");
        cpu1.release_bus();
        trigger(ON, CPU1done);
        cpu1.delay_cycle();
      }
    }
  join
```

```
}
```

The following code illustrates an example for semaphores. This is available in the `memsys0.vr` file in the solutions directory.

```
task check_all () {

  integer semaphoreId, randflag;
  event CPU1done;
  bit[7:0] mem_add0[], mem_add1[];

  printf("Task check_all:\n");
  semaphoreId = alloc(SEMAPHORE, 0, 1, 1);

  fork
    {// fork process for CPU 0
      repeat(256) {
        randflag = cpu0.randomize();
     printf("\n THE RAND MEM0 ADD IS %b \n\n", cpu0.address);
        if (mem_add0[cpu0.address] !== cpu0.address)
        {
            mem_add0[cpu0.address] = cpu0.address;
         printf("\n mem_add0[%b] is %b \n\n", cpu0.address,
mem_add0[cpu0.address]);
        }
        else
        {
            printf("\nThe memory0 address is being
repeated\n\n");
         printf("\n mem_add0[%b] is %b \n\n", cpu0.address,
mem_add0[cpu0.address]);
        }
        semaphore_get(WAIT, semaphoreId, 1);
        cpu0.request_bus();
        cpu0.writeOp();
        cpu0.release_bus();
        cpu0.request_bus();
        cpu0.readOp();
        cpu0.release_bus();
        semaphore_put(semaphoreId, 1);
        cpu0.delay_cycle();
```

```
      }
    }

    {// fork process for CPU 1
      repeat(256) {
        randflag = cpu1.randomize();
     printf("\n THE RAND MEM1 ADD IS %b \n\n", cpu1.address);
        if (mem_add1[cpu1.address] !== cpu1.address)
        {
            mem_add1[cpu1.address] = cpu1.address;
         printf("\n mem_add1[%b] is %b \n\n", cpu1.address,
mem_add1[cpu1.address]);
        }
        else
        {
            printf("\nThe memory1 address is being
repeated\n\n");
         printf("\n mem_add1[%b] is %b \n\n", cpu1.address,
mem_add1[cpu1.address]);
        }
        semaphore_get(WAIT, semaphoreId, 1);
        cpu1.request_bus();
        cpu1.writeOp();
        cpu1.release_bus();
        cpu1.request_bus();
        cpu1.readOp();
        cpu1.release_bus();
        semaphore_put(semaphoreId, 1);
        cpu1.delay_cycle();
      }
    }
  join
}
```

The program block for the top level testbench of the memory system will appear as follows:

```
program memsys_test
{ // Start of memsys_test
         cpu cpu0 = new (arb0, 0);
         cpu cpu1 = new (arb1, 1);
```

```
        init_ports();
        reset_sequence();
        check_all();

} // end of program memsys_test
```

The program block for the testbenches implementing mailbox and semaphores remains the same while the task check_all varies.

Compiling and Running Simulation with VCS

With the code added to your memory system testbench (`cpu.vr`), run the simulation and check the results.

Compile the testbench using the following VCS command line:

```
% vcs -ntb memsys.test_top.v ../rtl/memsys.v memsys.vr -
Mupdate +define+SYNOPSYS_NTB
```

Run the simulation by simply invoking the VCS executable `simv` in the following way:

```
% simv
```

Note:

If you use the run.csh from the solutions directory, you must specify the name of the testbench as an argument to the script. For example, run.csh memsys0 to run memsys0.vr testbench.

Congratulations! You've just completed writing testbenches in NTB-OV, and then compiling and simulating them using VCS.

2

Introducing SystemVerilog for Testbench

For quite some time now, design and verification engineers, alike, have felt the need for a single unified design and verification language that allows them to both *simulate* their HDL designs and *verify* them with high-level testbench constructs. To this end, Synopsys has implemented SystemVerilog, including SystemVerilog for design, assertions and testbench in its Verilog simulator, VCS. This unified language essentially enables engineers to write testbenches and simulate them in VCS along with their design in an efficient, high-performance environment.

Please contact vcs_support@synopsys.com for any questions or issues.

This section introduces SystemVerilog for Testbench, followed by a discussion of some of its verification-specific features. It includes the following sections:

High-Level Verification

Dramatic increases in the size and complexity of designs pose a significant challenge to traditional verification methodologies. The ever-increasing inter-module interactions and design interdependencies have made traditional verification methodologies inefficient (and in many cases, insufficient) for completing functional validation of designs with the desired degree of confidence and in the allotted amount of time. Therefore, the current need is to provide a means for achieving this functional validation in the most reliable, smart, efficient, and expeditious manner in order to deliver first-time-working silicon or systems on time.

Synopsys has become the leader in providing a single, unified design and verification environment based on VCS that not only enables completing functional validation of designs with the desired degree of confidence, but also helps achieving this goal in the most intelligent and efficient way and in the shortest time possible.

From our experience with customers, the number one objective on the way to reaching this verification goal is to be able to find and fix all bugs in a design before tape-out. The cost of fixing bugs grows exponentially with time as designs evolve. Not catching a few bugs early on in the design process inevitably leads to the dangerous scenario of their proliferation into more bugs, leading to very costly time-delays and design re-spins. In this regard, Synopsys’ unified design and verification environment of VCS is uniquely geared towards catching such hard-to-find bugs efficiently and early in the

design cycle. Additionally, the tight integration of the Synopsys Discovery platform makes this environment unmatched in its efficiency and speed.

SystemVerilog is one of the many key technologies that constitute this environment. Through the use of complex synchronization and timing mechanisms, concurrent processes can be written, providing a mechanism to simulate a real and dynamic test environment. SystemVerilog also supports the object-oriented methodology, and provides the necessary abstraction level to develop reliable and reusable test environments. SystemVerilog also enables random stimulus generation and self checking, which help increase the efficiency of the verification environment.

SystemVerilog for Testbench

SystemVerilog has several features built specifically to address functional verification needs. Please refer to the SystemVerilog Language Reference Manual (LRM) for the details on the language syntax, and the VCS User Guide for the usage model.

Concurrency and Control

Concurrency basically allows you to spawn off multiple parallel processes from a parent process. It brings power and flexibility to your verification environment, which inherently requires the execution of many processes in parallel, both for efficiency and smarter inter-process interaction. A typical example would be to stimulate a design and check the outcome in parallel. This allows your testbench to react decisively to the outcome in real time by enabling it to modify the stimulus (even before the simulation ends).

Concurrency is enabled via the fork-join construct, which spawns off multiple parallel processes. The type of join-back to the parent process depends on whether all, any, or none of the parallel processes have completed.

For inter-process communication, constructs called mailboxes allow any process to send a message to any other process. A receiving process can also synchronize with a sending process by waiting for its message.

Constructs called semaphores prevent several parallel processes from trying to access a common resource, such as a signal, by allowing access to only one process at any time.

Named events in processes are triggered using -> operator. These triggered events are received by other processes with the help of event control constructs in order to synchronize with the triggering processes.

Other process-control mechanisms are also provided to help processes wait either on specific variables or until the completion of their child processes.

Random Stimulus Generation with Constraints

Verification is the process of proving that your design is fully tested. Random stimulus inherently ensures corner-case scenarios are reached and tested. It is an important component to ensure that verification time and effort is spent in the most efficient and effective manner.

SystemVerilog provides a very powerful mechanism to generate random stimulus. It is based on class-object randomization, which means random variables of a class-object are automatically randomized by a call to the predefined randomize method associated with the object.

Constraints further augment the randomization feature. Constraints are properties that define the boundaries within which the randomization feature works. They have two key advantages: (1) You can add weights so that certain sets of stimuli occur more often than others, that is you can change the distribution function, and (2) you can use the current state of the design to change these weights and consequently the constraints for the next state.

Clocking Blocks

Clocking blocks encapsulate synchronous timing details. You can have as many clocking blocks as needed as long as each is associated with a specific clock. Input signals are driven and output signals are sampled only at clock edges. Clocking blocks provide separation and clarity between clock domains in a multi-clock design environment.

Classes, Methods, Properties: An Object-Oriented Methodology

You do not need any previous knowledge of or experience with object-oriented methodology in order to use SystemVerilog. You will be surprised to discover that object-oriented programming is very similar to HDL programming.

At its very basic level, object-oriented methodology compartmentalizes segments of code into what are called “Classes”. These classes are made up of two entities: “Properties” (data, variables) and “Methods” (function or task). Think of a class as an HDL module containing data and behavioral code. A class can be instantiated just like an HDL module with the help of an instance called its “Object”. Methods of this object will operate on class data members and a private data member can only be accessed outside the class by these methods. You can prevent accidental misuse or misapplication of its data in a particular segment of code by specifying methods that limit access to that data. Thus, this methodology provides protection from other code segments that might be using the same data names. However, classes provide a way of modeling the dynamic nature of environment and transaction which traditional HDL module can not.

The implementation of object-oriented methodology in SystemVerilog is clear, straightforward and elegant, assimilating and extending the best of the features found in C++ and Java. The methodology naturally lends itself to grouping related code and data of a testbench. These groups of data and related code are structured and thus easy to develop, understand, debug, maintain and reuse. The result is a disciplined and systematic testbench structure that is not only useful for verifying small and medium designs, but also has the dramatic, powerful impact on your productivity as soon as designs become large and significantly more complex.

Functional Coverage

Because of the complexity of chips and broad scope of functions that need tests for any given chip or system, the completeness of verification is a key issue. Functional coverage complements the

traditional code coverage by making sure all the functionalities of the chip or system are fully verified. SystemVerilog provides built-in functional coverage capability to address this need.

Design Overview

This section introduces the design of a system comprising of a memory, an arbiter, a controller, a system bus and CPUs that access the memory, all of which will be used in the remainder of the tutorial. It gives you an overview of the components of the system, describes how they interact with each other to complete the system and details the basic structure of the files used for this tutorial. It includes the following sections:

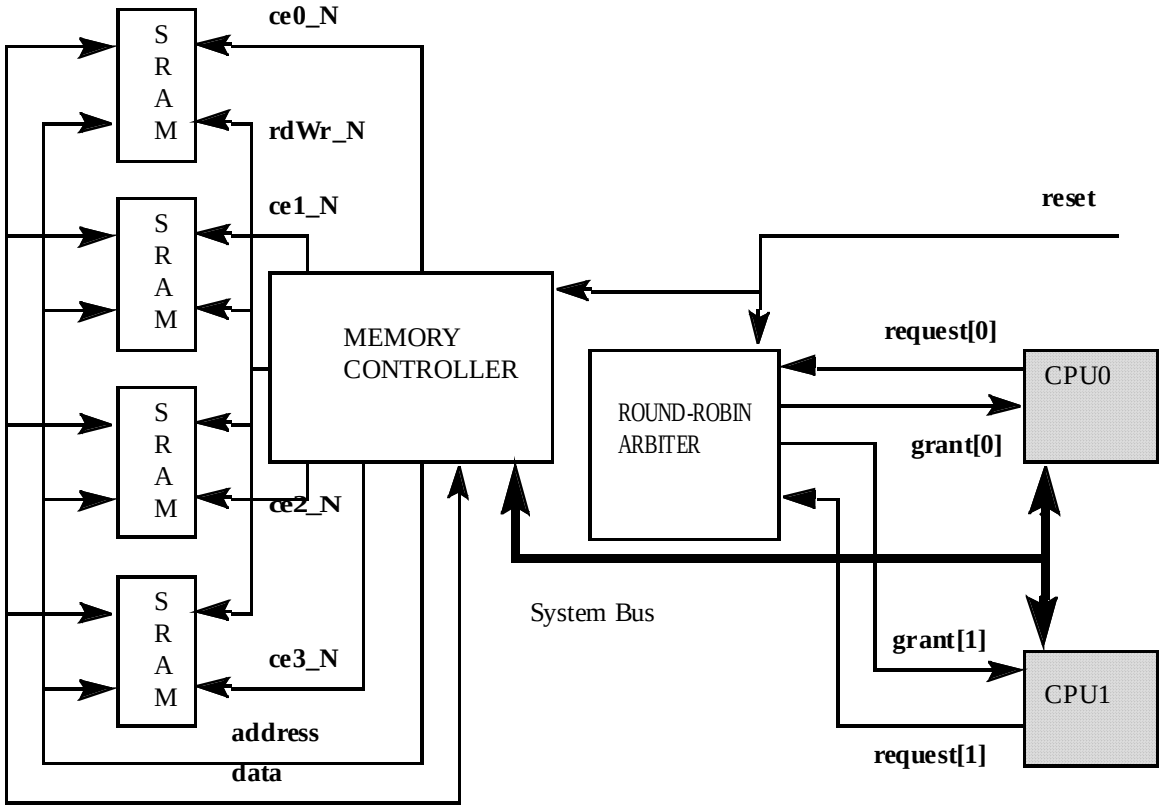
- "High-Level Verification" on page 2
- "Tutorial-design Directory Setup" on page 9

The files for this tutorial can be accessed in the following directory: $VCS_HOME/doc/examples/testbench/sv/tutorial.

Memory System

The design used in this tutorial is a simple memory system for a two CPU machine. It consists of a system bus, a centralized round-robin arbiter, and a memory controller that controls four static SRAM devices. Figure 2-1 shows the schematic of this system.

Figure 2-1 Memory System Schematic



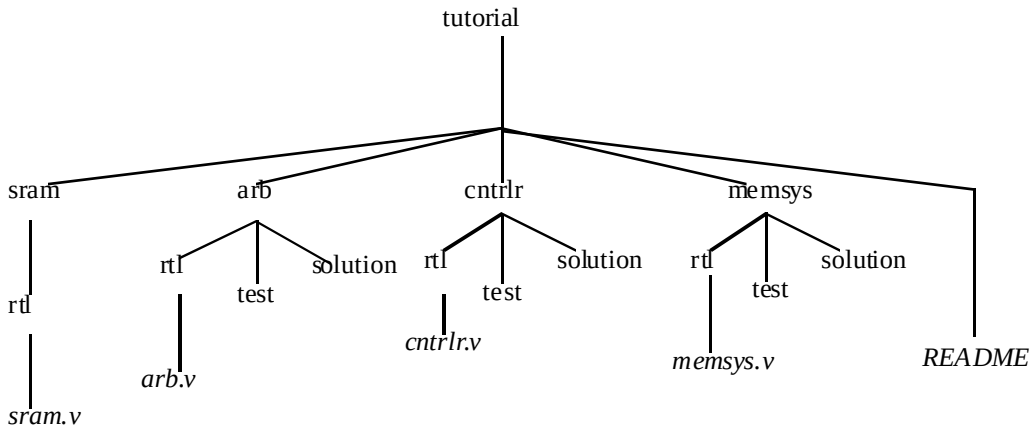
Notice that the blocks labeled CPU0 and CPU1 are shaded. This is to indicate that these blocks are not part of the system under test, but rather these blocks will be modeled within the SystemVerilog testbench. The signals shown between the CPUs and the rest of the system form the interface between the system under test and the “outside world.”

The memory system consists of the SRAMs, the Memory Controller, and the Arbiter. These are all defined in the HDL files of each sub-module. The approach used to verify the memsys system is similar to most project verification flows:

- Initially, all sub-modules are individually verified at the block level.

- Finally, all sub-modules are integrated into the final design for verification at the system level.
- This “full chip” functionality is verified at the system level.

Tutorial-design Directory Setup



The components of the design directory setup are described as follows:

- README — short description and file/directory index and listing of tools and versions used
- sram — contains the memory RTL
- arb — contains the submodule RTL and solution directory
- cntrlr — contains the submodule RTL and solution directory
- memsys — contains the top-level RTL netlist that integrates the entire memsys design and the solution directory

Also note the following:

- Each “rtl” directory contains Verilog HDL code.
- Each “solution” directory contains the solution SystemVerilog testbench, test-level module, header and script files. Please refer to the “README” file in this directory to see how to run the scripts and see the expected simulation output of the solution files.
- Each “test” directory inside the arb, cntrlr, and memsys directories is where you will be working. You will use this directory for creating your testbench, run scripts, compilation and simulation-generated files and directories.

Arbiter

This section focuses on the arbiter’s roll in the design. It briefly describes what the arbiter does, including a short timing and logic discussion. The section then describes the SystemVerilog for Testbench technology as it pertains to verifying the arbiter portion of the system. Specifically, this section explains how, using this technology, a testbench written in SystemVerilog interacts with a SystemVerilog design to drive signals, how the connections between the testbench and DUT are made, and how some of the basic signal operations behave. This section is divided into the following sections:

Arbiter Overview

You will be working inside the tutorial/arb/test directory.

- The Verilog arbiter RTL source code is in the following file:

  `tutorial/arb/rtl/arb.v`

- The solution top-level Verilog is in the following:

  `tutorial/arb/solution/arb.test_top.v`

- The solution testbench is in the following file:

  `tutorial/arb/solution/arb_test.v`

- The details of running the solution testbench are in the following file:

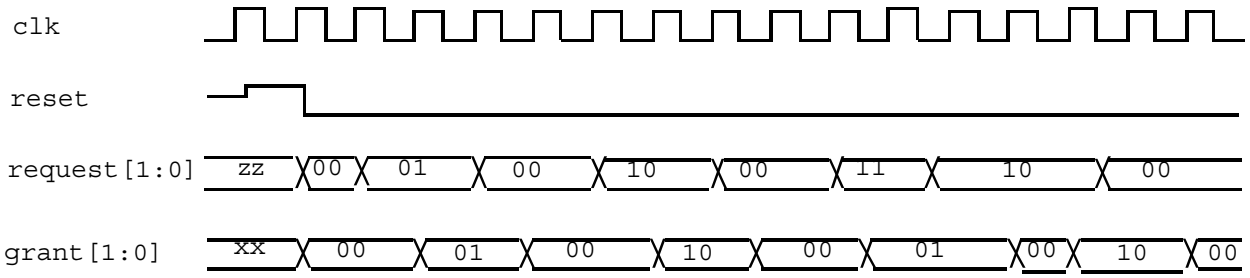
  `tutorial/arb/solution/README`

- The solution compile and run scripts are in the following file:

  `tutorial/arb/solution/run.csh`

The timing diagram that describes its IO behavior is now examined. Figure 2-2 shows the timing diagram.

Figure 2-2 Arbiter Timing Diagram



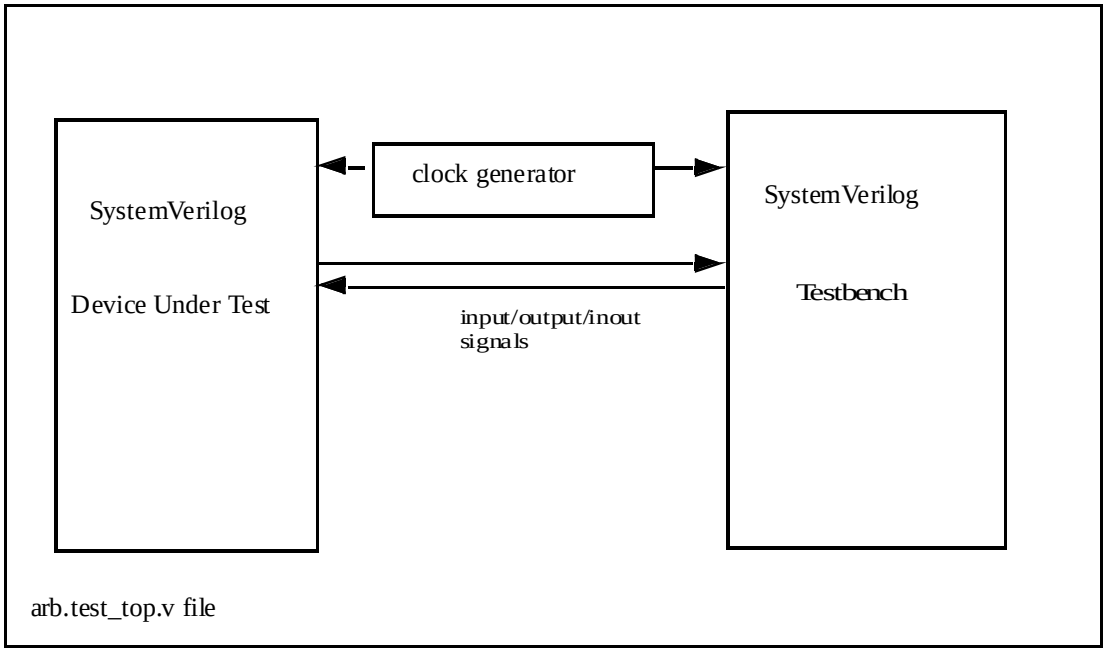
The arbiter implements a round-robin arbitration algorithm between two CPUs. Each CPU can drive a request input signal (**request**[0] or **request**[1]). The arbiter queues the requests and determines which CPU will gain access to the system bus. The arbiter grants this access by asserting one of the grant output signals (**grant**[0] or **grant**[1]). While the grant signal is asserted for a given CPU, the CPU continues to assert its request signal so that both the **grant** and **request** signals for the CPU remain high while the CPU accesses the system bus. Once the CPU is done, it de-asserts its request signal and, on the subsequent clock cycle, the arbiter de-asserts its grant signal. With all the signals de-asserted, the arbiter can continue with the next request.

Testbench Overview

A testbench suite is comprised of several key components:

- **`Testbench Module File`** — Describes the testbench.
- **`Test_top File (filename.test_top.v)`** — The top-level SystemVerilog file that encapsulates the DUT and the testbench suite. It instantiates the DUT and the SystemVerilog testbench, and handles system-clock generation and file-dumping in SystemVerilog. Figure 2-3 below shows basic schematic for this configuration.

Figure 2-3 arb.test_top.v Configuration Schematic



How to Get Going

arb.test_top.v

The *arb.test_top.v* file contains the top-level SystemVerilog code. The top-level code contains the signals that connects the testbench to the DUT. It also instantiates the testbench and the DUT, and contains the clock-generator for the system clock (clk) that is passed to the both DUT and testbench.

arb_test.v

The *arb_test.v* file shall contain the testbench program.

```
program arb_test(
        input  clk,
        input  [1:0]  grant_p,
        output [1:0] request_p,
```

```
        output reset_p);
endprogram
```

Verifying the Arbiter

First, verify the arbiter's reset. Then, verify that the arbiter handles simple requests appropriately and can grant access to one of the CPUs. Finally, check for the arbiter's proper handling of request sequences.

Reset Verification

Verify resets are working correctly. First assert the **`reset`** signal. With the **`reset`** signal asserted, hold the **`request`** signals inactive for each CPU (drive them to 0) and check that the **`grant`** signals are at their inactive states (0) after the reset.

Basic Signal Operation

All signal operations occur on the clock edge specified in the clocking block.

To advance the simulation to the next value change of a specified signal, use the **synchronize** construct:

```
@(clock_edge signal_name); //clock_edge:posedge/negedge/no
     edge specified
```

This advances the simulation to the specified edge of the signal. If the clock edge is omitted, it advances the simulation to the next signal-value change, which represents a sampling edge.

To assert and de-assert signals, use the SystemVerilog for Testbench **`drive`** construct:

```
signal_name <=value;
```

The specified signal is driven to the appropriate value after the current time.

Note:

Non-blocking Assignment (NBA) operator "<=" must be used for drives.

Using **`expect`** construct:

The expect operator is used to check for correct expected behavior.

```
label: expect (@(clock_sensitivity) property_spec);
```

Where, *clock_sensitivity* is the name of the clock and edge that should be used during the checking, and *property_spec* is the temporal expression. For example,

```
e1: expect(@(posedge clk) ##[1:3] xyz);
```

will check that `xyz` is asserted between 1 to 3 clock cycles.

If the signal value is the same as the specified value, the simulation continues. If there is a mismatch, a user defined error message in the action block in the else branch will be printed or a default error message without using action block, but the simulation still continues. User can stop or finish the simulation in the action block if needed.

Verifying the Reset

In the *arb_test.v* file, add the following code to verify the reset:

```
$write("Task reset_test: asserting and checking reset\n");
```

```
reset_p <= 1;
repeat (2) @(posedge clk);
reset_p <= 0;
request_p <= 2'b00;
expect(@(posedge clk) grant_p == 2'b00);
```

Note that the `request` and `grant` signals are 2-bit signals. Each bit of the signals must be de-asserted.

Compiling and Running the Simulation with VCS

(If you want to run the tutorial solution, go to the arb/solution directory and invoke the script run.csh after making sure you have your `$VCS_HOME` and license variables already set).

At this point, you should be in the *tutorial/arb/test* directory. With the code added to your arbiter testbench (*arb_test.v*), run the simulation and test the results.

Compile the testbench use the following VCS command line and fix any syntax errors during compilation.

```
% vcs -sverilog arb.test_top.v ../rtl/arb.v arb_test.v -l \
    comp.log
```

Run the simulation by simply invoking the VCS executable simv in the following way:

```
% simv -l sim.log
```

Any verification errors found during the simulation will be automatically reported. For instance, change the line (expect) that checks the de-assertion of grant in the *arb_test.v* file to the following:

```
expect(@(posedge clk) grant_p <= 2'b01););
```

Recompile and run the simulation again. The testbench now expects the `grant` signal to be asserted while the SystemVerilog model continues to de-assert the signal as before. This results in an expect mismatch and a user defined error as described on the previous page. The simulation will, however, proceed.

Note:

Remember to edit the testbench file *arb_test.v* to correct this error before moving ahead with the tutorial.

Simple Request Verification

To check if the arbiter is handling simple requests correctly, monitor the **`request`** signals, check that the **`grant`** signal is set appropriately, and then check that the **`grant`** signal is de-asserted after the **`request`** is released.

Test For Simple Request by CPU0

To test that simple requests are handled correctly for CPU0, drive bit 0 of the **`request`** signal and then monitor bit 0 of the **`grant`** signal. Finally, de-assert both bits of the **`request`** signal and check that both bits of the **`grant`** signal are properly de-asserted.

```
@(posedge clk) request_p <= 2'b01;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b01);
@(posedge clk) request_p <= 2'b00;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b00);
```

Test For Simple Request by CPU1

To test that simple requests are handled correctly for CPU1, drive bit 1 of the **`request`** signal and then monitor bit 1 of the **`grant`** signal. Finally, de-assert both bits of the **`request`** signal and check that both bits of the **`grant`** signal are properly de-asserted.

```
@(posedge clk) request_p <= 2'b10;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b10);
@(posedge clk) request_p <= 2'b00;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b00);
```

Sequenced Request Verification

Verify sequences of requests are handled properly by checking a series of conditions:

- Assert both request signals and check that the correct grant is asserted (depends on which grant was previously asserted)
- Release the granted request and check that both grants are released
- Then check that the other grant is asserted
- Finally release the other request and check that both grants are released

Given this verification methodology, the code to check the arbiter behavior is the following:

```
@(posedge clk) request_p <= 2'b11;
@(posedge clk);
expect(@(posedge clk) grant_p == 2'b01);
request_p <= 2'b10;
expect(@(posedge clk) ##[0:2] grant_p == 2'b10);
```

```
request_p <= 2'b00;
repeat (2) @(posedge clk);
assert(grant_p == 2'b00);
```

Doing Things in an Organized Manner with Tasks

After writing the above pieces of the testbench in the *arb_test.v* file and verifying that they run, you probably have noticed that the simulation takes hardly a fraction of a second to run and complete. Now, you can think of stretching the simulation by checking repeatedly with sequences of reset followed by request. This can be easily done by putting the code for reset and request in tasks and then calling these tasks from the main program in the testbench (*arb_test.v* file).

The following code shows you how to write a task for verifying the reset:

```
task reset_test;
  $write("Task reset_test: asserting and checking reset\n");
   reset_p <= 1;
   repeat (2) @(posedge clk);
   reset_p <= 0;
   request_p <= 2'b00;
   expect(@(posedge clk) grant_p == 2'b00) else
      $display($time, " Failed");
Endtask
```

You can write a similar task called request_grant_test for verifying the request and grant sequence and then call both the tasks from the main program of the testbench in the following manner:

```
repeat (2000)
begin
     reset_test();
     request_grant_test();
end
```

Memory Controller

This section discusses the memory controller in your design. It gives you an overview of how the memory controller operates. It discusses some of the major features of SystemVerilog for Testbench used to verify the controller, including a description of polymorphism and virtual task/function as well as synchronous events. These concepts are presented within the verification framework so that you can learn how to adequately validate the memory controller. This section includes the following sections:

Memory Controller Overview

In our system, the CPU accesses the bus through the arbiter. Once the CPU has access, it puts its request on the system bus. The memory controller acts on this request by reading data from the SRAM devices and returning data when necessary. You will be working inside the `tutorial/cntrlr/test` directory.

- The Verilog controller RTL source code is in the following file:

  `tutorial/cntrlr/rtl/cntrlr.v`

- The solution top-level Verilog is in the following file:

  `tutorial/cntrlr/solution/cntrlr.test_top.v`

- The solution testbench is in the following file:

  `tutorial/cntrlr/solution/cntrlr_test.v`

- The solution device class is in the following file:

  `tutorial/cntrlr/solution/device.v`

- The details of running the solution testbench are in the following file:

  `tutorial/cntrlr/solution/README`

- The solution compile and run scripts are in the following file:

  `tutorial/cntrlr/solution/run.csh`

The memory controller reads requests from the system bus and generates control signals for the SRAM devices attached to it. For read requests, the controller reads data and transfers it back to the bus and the CPU making the request. The address bus is 8 bits wide, which creates an address space of 256 bytes. The controller supports up to 4 devices, allocating a maximum of 64 bytes of memory to each. The controller decodes the address and generates the chip enable for the corresponding device during a transaction. Figure 2-4 shows a diagram of how the testbench works with both the system bus and SRAM device signals.

Figure 2-4 SystemVerilog Testbench/Memory Controller Interaction

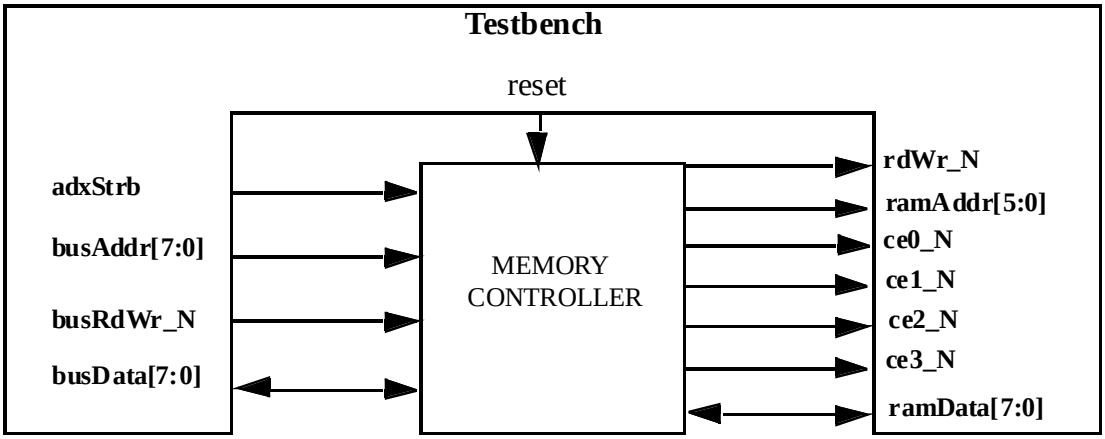
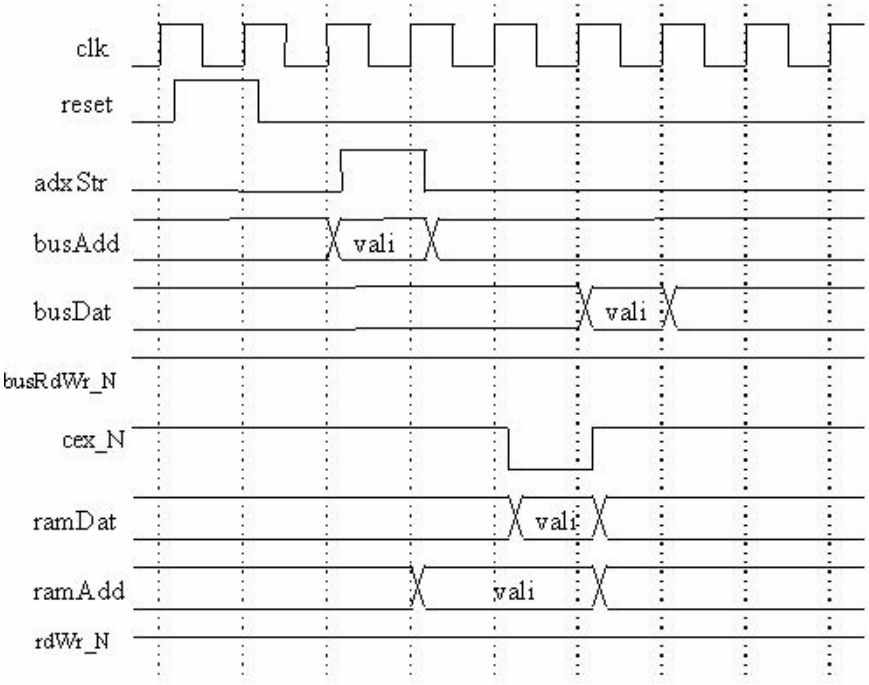


Figure 2-5 and Figure 2-6 show the timing diagrams for the memory controller's read and write operations respectively (note the signal names as you will be using them in the verification process).

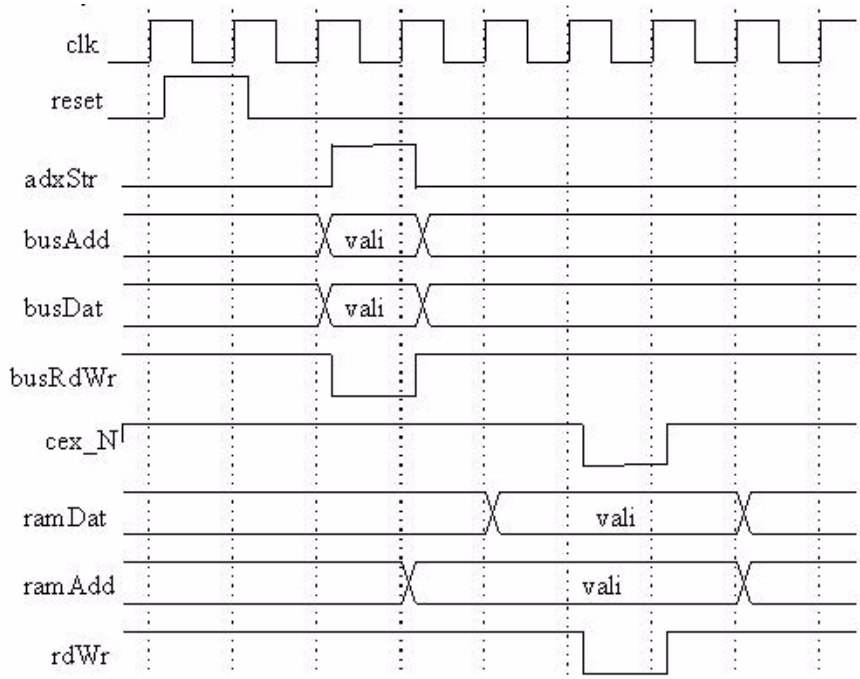
Figure 2-5 Memory Controller Read Operation Timing Diagram



Note:

In the case of cex_N, x refers to 0,1,2 and 3.

Figure 2-6 Memory Controller Write Operation Timing Diagram



Verifying the Memory Controller

To completely check the functionality of the memory controller, you have to perform a series of tests. First, verify the controller's reset by writing a task. Then, write tasks to check the read and write operations of the controller. Also check the integrity of the read and write operations. Finally, check the address map (all 256 addresses) exhaustively for the read and write functions.

This section focuses on checking the memory controller by emulating both the system bus and the memory bus behavior. Rather than connecting the RTL models of the memory to the controller, model the behavior of the four different memory devices in the testbench.

To start the verification process, first create the following files:

- `cntrlr.test_top.v` (top-level module file)
- `cntrlr_test.v` (testbench file)
- `device.v` (device class file)

Controller Interface

Let's create an interface and call it "cntrlr_intf". This interface will be instantiated in the cntrlr.test_top.v file and passed to the program block and the dut. The interface will encapsulate the interface signal in a single structure.

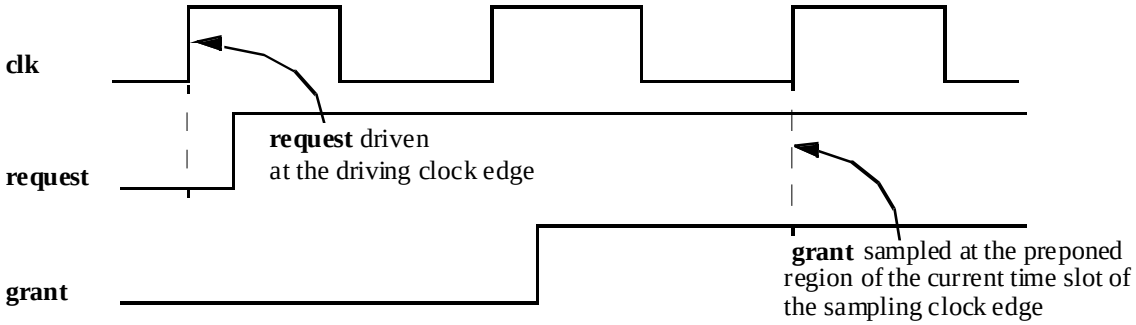
```
interface cntrlr_intf(input clk);
  wire  reset ;
  wire  [7:0]  busAddr ;
  wire  [7:0]  busData ;
  wire  busRdWr_N ;
  wire  adxStrb ;
  wire  rdWr_N ;
  wire  ce0_N ;
  wire  ce1_N ;
  wire  ce2_N ;
  wire  ce3_N ;
  wire  [5:0]  ramAddr ;
  wire  [7:0]  ramData ;

endinterface
```

Next we must instantiate the interface:

```
cntrlr_intf intf(clk);
```

SystemVerilog provides clocking block construct to specify the sampling and driving skews. Note that each clocking block has a clock associated with it. All signal operations occur on the corresponding clocking block clock-edge. For example, given an clocking block with drives and samples occurring on the positive clock edges and an input skew of #1step and an output skew of #0 respectively, the timing diagram is given by the following:



Let's put the clocking block definition within the interface:

```
clocking CBcntrlr @(posedge clk);
  output reset,busAddr,busRdWr_N,adxStrb;
  input rdWr_N,ce0_N,ce1_N,ce2_N,ce3_N,ramAddr;
  inout busData,ramData;
endclocking
```

We can pass the interface directly to the program block, but we must pass each element of the interface to the dut:

```
cntrlr_test testbench(intf);

cntrlr dut(
   .clk ( clk ),
   .reset ( intf.reset ),
   .busAddr ( intf.busAddr ),
   .busData ( intf.busData ),
   .busRdWr_N ( intf.busRdWr_N ),
   .adxStrb ( intf.adxStrb ),
   .rdWr_N ( intf.rdWr_N ),
```

```
   .ce0_N ( intf.ce0_N ),
   .ce1_N ( intf.ce1_N ),
   .ce2_N ( intf.ce2_N ),
   .ce3_N ( intf.ce3_N ),
   .ramAddr ( intf.ramAddr ),
   .ramData ( intf.ramData )
);
```

Controller Reset Verification

First assert the reset signal for at least one clock cycle. With the reset signal asserted, check that the memory enable signals are de-asserted.

Resetting the Controller

Using simple drives, create a task as follows and add it to the cntrlr_test.v file to reset the controller:

```
task resetSequence();
     vintf.CBcntrlr.reset <= 1'b1;
     vintf.CBcntrlr.ramData <= 8’bzzzzzzzz;
     repeat (2) @CBcntrlr;
     vintf.CBcntrlr.reset <= 1'b0;
endtask
```

Verifying the Reset

Create another task as follows to verify that the memory enable signals (active_low) for all the 4 SRAM devices are de-asserted (high) and add it to the cntrlr_test.v file.

```
task resetCheck ();
  $write("Task resetCheck entered to check reset values\n");
   //all chip enables must be deasserted
 expect(@(vintf.CBcntrlr) ##[0:10] vintf.CBcntrlr.ce0_N ===
1'b1);
   assert(vintf.CBcntrlr.ce1_N == 1'b1);
   assert(vintf.CBcntrlr.ce2_N == 1'b1);
```

```
   assert(vintf.CBcntrlr.ce3_N == 1'b1);
endtask
```

Invoking the Tasks

Having created the two tasks as described above, write calls to them in the main program with a clocking block in the `cntrlr_test.v` file as follows:

```
program cntrlr_test (cntrlr_intf intf);

virtual cntrlr_intf vintf;
initial begin
  vintf = intf;
  @vintf.CBcntrlr;
  resetSequence();
  resetCheck();

  $finish;
end
 //reset sequence
task resetSequence ();
  $write("Task resetSequence entered\n");
  vintf.CBcntrlr.reset <= 1'b1;
  vintf.CBcntrlr.ramData <= 8'bzzzzzzzz;
  repeat (2) @vintf.CBcntrlr;
  vintf.CBcntrlr.reset <= 1'b0;
endtask

//Check state of controller after reset
task resetCheck ();
  $write("Task resetCheck entered to check reset values\n");
  //all chip enables must be deasserted
 expect(@(vintf.CBcntrlr) ##[0:10] vintf.CBcntrlr.ce0_N ===
1'b1);
   assert(vintf.CBcntrlr.ce1_N == 1'b1);
   assert(vintf.CBcntrlr.ce2_N == 1'b1);
   assert(vintf.CBcntrlr.ce3_N == 1'b1);

endtask
```

```
endprogram
```

Compiling and running the Simulation with VCS

At this point, you should be in the `tutorial/cntrlr/test` directory. With the code added to your controller testbench (`cntrlr_test.v`), run the simulation and check the results.

Compile the testbench using the following VCS command line and fix any syntax errors that you may have:

```
% vcs -sverilog cntrlr.test_top.v ../rtl/cntrlr.v
cntrlr_test.v -l comp.log
```

Run the simulation by simply invoking the VCS executable simv in the following way:

```
% simv -l sim.log
```

Driving the System Bus For Read and Write Operations

In testing the read and write capabilities of the controller, create two tasks that drive the bus for read and write operations.

Read Operation

Create a task that drives the read operation onto the system bus as specified in the timing diagram for the controller. The task should use an 8-bit bus address as an input. Given this requirement, the read operation task is the following:

```
task readOp (bit [7:0] adx);
  $write("Task readOp : address %0h\n", adx);
  vintf.CBcntrlr.busAddr <= adx;
  vintf.CBcntrlr.busRdWr_N <= 1'b1;
  vintf.CBcntrlr.adxStrb <= 1'b1;
  @vintf.CBcntrlr vintf.CBcntrlr.adxStrb <= 1'b0;
```

```
endtask
```

This task is passed the argument `adx`. It then drives the `busAddr` signal to that value. Finally, it drives the `busRdWr_N` and `adxStrb` signals such that they match the timing diagram for the read operation of the controller.

Note:

Since this is a read operation, do not drive the data onto the bus and check for the expected data here.

Write Operation

Create a task that drives the write operation onto the system bus as specified in the timing diagram for the controller. The task should use 8-bit address and data busses as inputs. Finally, the task should leave the bus in an idle state (defined when `busData` is in high z and `busRdWr_N` is de-asserted). Given these requirements, the write operation task is the following:

```
task writeOp (bit [7:0] adx, bit [7:0] data);
  $write("Task writeOp : address %0h data %0h\n", adx, data);
   @vintf.CBcntrlr vintf.CBcntrlr.busAddr <= adx;
   vintf.CBcntrlr.busData <= data;
   vintf.CBcntrlr.adxStrb <= 1'b1;
   vintf.CBcntrlr.busRdWr_N <= 1'b0;
   @vintf.CBcntrlr vintf.CBcntrlr.adxStrb <= 1'b0;
   vintf.CBcntrlr.busRdWr_N <= 1'b1;
   vintf.CBcntrlr.busData <= 8'bzzzzzzzz;
endtask
```

This task is passed the argument `adx`. It then drives the `busAddr` signal to that value. Finally, it drives the `busData`, `busRdWr_N`, and `adxStrb` signals such that they match the timing diagram for the write operation of the controller.

Implementing Virtual Interfaces

In SystemVerilog for Testbench, virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same subprogram to operate on different portions of a design, and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals. Changes to the underlying design do not require the code using virtual interfaces to be re-written. By abstracting the connectivity and functionality of a set of blocks, virtual interfaces promote code-reuse. However, virtual interfaces are not implemented yet VCS8.0 alpha release.

Implementing 4 devices in the Memory Controller

Let us define a device base class for the SRAM parts (`ramAddr`, `ramData`, `rdWr_N`, and `ce_N`) which has methods to drive and sample these signals synchronously as follows:

```
virtual class device;
    task driveRamData(input logic [7:0] data);
        vintf.CBcntrlr.ramData <= data;
    endtask

    function  logic [7:0] getBusData();
        return vintf.CBcntrlr.busData;
    endfunction

    function  logic [5:0] getRamAddr();
        return vintf.CBcntrlr.ramAddr;
    endfunction

    function  logic [7:0] getRamData();
        return vintf.CBcntrlr.ramData;
    endfunction
```

```
    function logic getRdWr_N();
        return vintf.CBcntrlr.rdWr_N;
    endfunction

    // these methods must be defined in derived classes
    extern virtual function logic getCe_N();
    extern virtual task waitCe_N();

endclass
```

After defining the device base class, let's extend it to create 4 device classes, device0, device1, device2 and device3 as follows:

```
class device0 extends device;
    function logic getCe_N();
        return vintf.CBcntrlr.ce0_N;
    endfunction

    task waitCe_N();
        @vintf.CBcntrlr.ce0_N;
    endtask
endclass

class device1 extends device;
    function logic getCe_N();
        return vintf.CBcntrlr.ce1_N;
    endfunction

    task waitCe_N();
        @vintf.CBcntrlr.ce1_N;
    endtask
endclass

class device2 extends device;
    function logic getCe_N();
        return vintf.CBcntrlr.ce2_N;
    endfunction

    task waitCe_N();
        @vintf.CBcntrlr.ce2_N;
    endtask
```

```
endclass

class device3 extends device;
function logic getCe_N();
        return vintf.CBcntrlr.ce3_N;
    endfunction

    task waitCe_N();
        @vintf.CBcntrlr.ce3_N;
    endtask
endclass
```

In the extended class, the two methods `getCe_N` and `waitCe_N` is connected to its device-specific signal. These classes definitions will be included within the program block and instantiated as follows:

```
`include "device.v"
device0 d0 = new;
device1 d1 = new;
device2 d2 = new;
device3 d3 = new;
```

Verifying Read and Write Operations

The memory controller issues read and write operations to each of the four SRAM devices as shown in the earlier timing diagram. Create read and write tasks in your testbench that check these operations. Earlier, you modeled the timing diagram exactly, cycle by cycle. Your approach now is to make use of timing windows, which allow you to specify ranges of time and event sequences.

Because of complex timing issues with the read operation, examine the write operation first. A discussion of the timing issues and the read operation follows.

Verifying the Write Operation

To verify the write operation, create a task that checks the SRAM write operation against the timing diagram provided. The task should have the `device_id` port variable as an argument so that we can pass in the signals on which we want the task to act. The task also has 6-bit address and 8-bit data busses as inputs. It must check that the SRAM signals are driven correctly, check that the address is the right address, and drive the data onto the `ramData` bus at the appropriate time. Given these requirements, the code for the write operation is the following:

```
task checkSramWrite ( device device_id, bit [5:0] adx, bit
[7:0] data);
  expect (@(vintf.CBcntrlr) ##[1:5]device_id.getRamAddr()
         == adx);
  expect (@(vintf.CBcntrlr) ##[0:2] device_id.getRamData()
         == data);
   assert(device_id.getRdWr_N() == 1'b0);
   assert(device_id.getCe_N() == 1'b0);
   assert(device_id.getRamData() == data);
   assert(device_id.getRamAddr() == adx);
   @vintf.CBcntrlr assert(device_id.getRdWr_N() == 1'b1);
   $write("Task checkSramWrite: Address %0h data %0h\n",
   vintf.CBcntrlr.ramAddr, vintf.CBcntrlr.ramData);
   assert(device_id.getCe_N() == 1'b1);
   assert(device_id.getRamData() == data);
   assert(device_id.getRamAddr() == adx);
endtask
```

The task first checks whether the address (`ramAddr`) is valid in a timing window that begins after the first rising edge and lasts for the next five rising clock edges. As soon as the address is found to be valid, the task checks whether the data (`ramData`) to be written is valid in a timing window that begins immediately and lasts for the next two rising clock edges. At the rising edge after the data is found to be valid, the task checks whether the write (`rdWr_N`) and enable

(ce_N) signals are asserted. At the same time, the address and data are checked again to see if they are still valid. Then, after another rising clock edge, the task checks whether the write and enable signals are de-asserted while checking the address and data for the third time to make sure they were valid.

Notice how the task accesses each SRAM instance by passing "device" object to the task argument device_id. The SRAM instance that is accessed depends on the port variable passed to the task through the argument device_id. For example, if the port variable passed is d0, the fourth statement in the task would be interpreted as vintf.CBcntrlr.Ce0_N==1'b0.

Verifying the Read Operation

To verify the read operation, check that the control signals are asserted, the correct address is driven by the memory controller, and the input data is driven as return data within 0-2 clock cycles. The task must have the port variable device_id as an argument to allow it to access the enable signal of a particular SRAM instance. It also has 6-bit address and 8-bit data busses as inputs. The code for this task is the following:

```
task checkSramRead (device device_id, bit [5:0] adx, bit
[7:0] data);
  device_id.driveRamData(data);
  device_id.waitCe_N();
  assert(device_id.getCe_N() == 0);
  assert(device_id.getRdWr_N() == 1);
  assert(device_id.getRamAddr() == adx);
  $write("Task checkSramRead: Address %0h data %0h\n", adx,
data);
  printCycle();
  expect (@(vintf.CBcntrlr) ##[0:2] device_id.getBusData()
=== data);
  printCycle();
  device_id.driveRamData (8'bzzzzzzzz);
```

```
endtask
```

Running the Simulation

With the reset check completed, write the code to check the complete write operation of one of the devices. This includes the code to drive the bus for the write operation in the writeOp() task, and the code to check that write operation in the checkSramWrite() task. The code is the following:

```
writeOp (8'h01, 8'h5A);
checkSramWrite (d0, 6'b000001, 8'h5A);
```

This code drives the bus and then checks the write operation using the specified device signals associated with `device0`. When checking other devices, remember that each device has a specific range of valid addresses as shown in the following table:

Device	Valid Address Range
0	0-63
1	64-127
2	128-191
3	192-255

Now add in the code to check the write operations for the other devices. The same tasks can be used with different virtual ports and different address parameters.

Similarly, use the generic tasks to drive the bus for read operations and check the device read operations. Remember to check that the returned data matches the return data specified in the timing diagram. The code for these checks is the following:

```
    readOp (8'h03);
    checkSramRead (d0, 6'b000011, 8'h95);
    @vintf.CBcntrlr assert(vintf.CBcntrlr.busData ==
             8'h95);
```

This code drives the bus and then checks the read operation using the specified device signals associated with `d0` (similar code can be written for the other devices). Finally, the return data is checked to see that it matches the correct value.

These tests only cover a subset of the valid addresses. To exhaustively test the entire range using these calls, each task must be called with every address. This is achieved with the help of a for loop, as shown in the following code:

```
bit[7:0] index;
    ...
for (int i=0;i<=255;i++) begin
    index = i;
    writeOp(index, 8'h5A);
    case (index[7:6])
         2'b00: device_id = d0;
         2'b01: device_id = d1;
         2'b10: device_id = d2;
         2'b11: device_id = d3;
    endcase
    checkSramWrite (device_id, index[5:0], 8'h5A);
    readOp(index);
    checkSramRead (device_id, index[5:0], 8'h5A);
    @vintf.CBcntrlr assert(vintf.cntrlr.busData == 8'h5A);
end
```

Each iteration of this for loop acts on a different address. It drives the bus operation and then checks the SRAM operation using the tasks defined previously. The case statement picks the device on which the tasks act so that they correspond to the SRAM instance being accessed. The bus data is then checked at the end of each iteration to monitor the return values.

Memory System

Now that you've become familiar with the separate functionality of the arbiter and memory controller, you can examine the way these components interact in a complete system. This section briefly discusses the overview of the system, which includes the arbiter, controller, and SRAM devices.

Also discussed are some of the higher level verification techniques used by SystemVerilog Testbench. These include concurrency control mechanisms, such as semaphores, events, mailboxes, object-oriented methodology by way of classes, and random stimulus generation. Finally, this section will show you how to use these features to validate the memory system. This section includes the following sections:

Memory System Overview

You will be working inside the `tutorial/memsys/test` directory, which includes the following files:

- The Verilog memory system RTL source code is in the following file:

  `tutorial/memsys/rtl/memsys.v`

- Links to all the RTL for sram, arb, cntrlr and memsys are in the following file:

  `tutorial/memsys/solution/memsys.f`

- The solution top-level Verilog code is in the following file:

  `tutorial/memsys/solution/memsys.test_top.v`

- The solution testbench is in the following files:

  ```
  tutorial/memsys/solution/memsys0.v (semaphores)
  tutorial/memsys/solution/memsys1.v (mailboxes and
  events)
  ```

- The testbench code that mimics the CPUs is in the following file:

  `tutorial/memsys/solution/cpu.v`

- The two functional coverage groups for memory system is in the following file:

  `tutorial/memsys/solution/memsys_coverage.v`

- The details of running the solution testbench are in the following file:

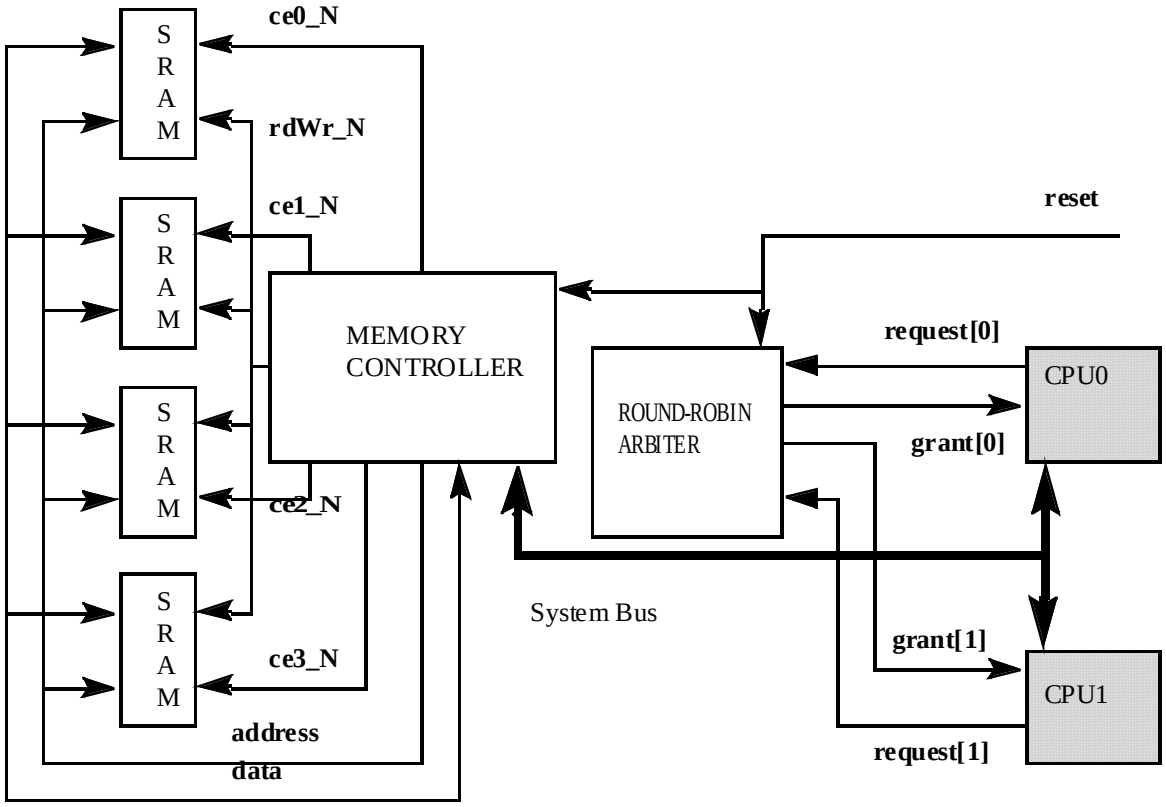
  `tutorial/memsys/solution/README`

- The solution compile and run script are in the following file:

  `tutorial/memsys/solution/run.csh`

The memory system acts as a wrapper that instantiates the arbiter, memory controller, and four SRAM devices. In our system, the system bus is driven by two separate CPUs, with access granted through the arbiter. The memory controller handles the reading and writing of data to and from the system bus. A schematic of the complete system is provided in Figure 2-7.

Figure 2-7 Memory System Schematic



Verifying the Memory System

The methodology used to verify the entire memory system is broken down by the following tasks and concepts:

- **General Verification** — Port initialization, reset verification and read/write operations.
- **Basic Concurrency and Control** — Concurrency between the two CPUs using fork/join, and Control using semaphores to make sure the CPUs do not access the memory at the same time.

- **Object-oriented Programming** — Object-oriented programming using a Class to model a CPU while making the model re-usable.
- **Interprocess Communication** — Communication between the two CPUs using Mailboxes and Events to make sure that the simulation advances in lock-step fashion.
- **Functional Coverage** — Coverage objects to make sure that the chip or system perform defined coverage goals.

General Verification

To start the verification process, first create the following files:

- `memsys.test_top.v` (top-level module file)
- `memsys0.v` and `memsys1.v` (testbench files)
- `cpu.v` (cpu file)
- `memsys_coverage.v` (coverage file)

The general verification tasks include initializing ports, checking the reset procedure and modifying the read and write operations previously developed for the memory controller. You can do this with the help of tasks. Finally, you can develop a testbench that checks both CPUs being run concurrently using multiple processes.

Setting Up the Top Level

In the file "`memsys.test_top.v`" create an interface with a single clocking block as we did in the previous section.

Port Initialization and Reset Verification

To check that the system is resetting correctly, we must first initialize ports and then go through the reset sequence. The code to do this is the following:

```
task init_ports () ;
  $write("Task init_ports\n");
  @vintf.CBmemsys vintf.CBmemsys.request <= 2'b00;
  vintf.CBmemsys.busRdWr_N <= 1'b1;
  vintf.CBmemsys.adxStrb <= 1'b0;
  vintf.CBmemsys.reset <= 1'b0;
endtask

task reset_sequence () ;
   $write("Task reset_sequence\n");
   vintf.CBmemsys.reset <= 0;
   @vintf.CBmemsys vintf.CBmemsys.reset <= 1;
   repeat (10) @vintf.CBmemsys;
   vintf.CBmemsys.reset <= 0;
   @vintf.CBmemsys assert(vintf.CBmemsys.grant == 2'b00); /
/ check if grants are
                                                  // 0's
Endtask
```

Read and Write Operations

The bus used for accessing the memory in the system is very similar to the bus used in the memory controller in the previous section. Using this system bus, the writeOp() task for the memory system is the following:

```
task writeOp (bit[7:0] address, bit[7:0] data);
  @vintf.CBmemsys vintf.CBmemsys.busAddr <= address;
  vintf.CBmemsys.busData <= data;
  vintf.CBmemsys.busRdWr_N <= 1'b0;
  vintf.CBmemsys.adxStrb <= 1'b1;
  @vintf.CBmemsys vintf.CBmemsys.busRdWr_N <= 1'b1;
  vintf.CBmemsys.busData <= 8'bzzzzzzzz;
  vintf.CBmemsys.adxStrb <= 1'b0;
```

```
  $write("WRITE address = 0%H, data = 0%H \n", address, data);
endtask
```

During the read operation, the read operation task must also check for correct return data. The new readOp() task with this data checking is the following:

```
task readOp (bit[7:0] address, bit[7:0] data);
  @vintf.CBmemsys vintf.CBmemsys.busAddr <= address;
  vintf.CBmemsys.busRdWr_N <= 1'b1;
  vintf.CBmemsys.adxStrb <= 1'b1;
  @vintf.CBmemsys vintf.CBmemsys.adxStrb <= 1'b0;
  expect (@(vintf.CBmemsys) ##[2:5] vintf.CBmemsys.busData
          == data);
  $write("READ address = 0%H, data = 0%H \n", address, data);
Endtask
```

Concurrent Processes

Fork/join blocks are the primary mechanism for creating concurrent processes. The syntax to declare a fork/join block is:

```
fork
     statement1;
     statement2;
     ...
     statementN;
```

join/join_none/join_any

statementN

Can be any valid SystemVerilog statement.

join

When this is used, the code after the fork/join block executes only after all of the concurrent processes have completed.

join_any

When this is used, the code after the fork/join_any block executes right after any concurrent process within the fork/join_any completes.

join_none

When this is used, the code after the fork/join_none block executes immediately without waiting for any of the fork/join_none processes to complete.

With the read and write operations defined, you want to set up your testbench such that each CPU issues a series of read and write requests to the memory system with random addresses and data. The two CPUs should operate concurrently or in parallel. Each CPU should use the **`random()`** system function to generate random addresses within the valid address space and an 8-bit data type. The CPUs should then request and access the bus, write the data to the bus, and release the bus (check for the release of the **`grant`** signal upon bus release). This sequence should be repeated 256 times using the **`repeat()`** flow control statement. Given these criteria, the testbench code is the following:

```
random (12933); //call random with seed
fork
// CPU0
repeat(256)
begin
     randVar0 = $random(); // get 32 bit random variable
     address0 = randVar0[13:6]; // get random 8-bit address
     data0 = randVar0[29:22]; // get random 8-bit data
     @vintf.CBmemsys vintf.CBmemsys.request[0] <= 1'b1;
          // request the bus

     expect(@(vintf.CBmemsys) ##[2:20]  (memsys.grant ==
          2'b01));
```

```
    writeOp(address0, data0); // issue write operation
    @vintf.CBmemsys vintf.CBmemsys.request[0] <= 1'b0;
        // release request
    expect(@(vintf.CBmemsys) ##[2:20] (memsys.grant ==
        2'b00));
    @vintf.CBmemsys vintf.CBmemsys.request[0] <= 1'b1;
        // request again
    expect (@(vintf.CBmemsys) (memsys.grant == 2'b01));
    readOp(address0, data0); // issue read operation
    @vintf.CBmemsys vintf.CBmemsys.request[0] <= 1'b0;
        // release request
    expect( @(vintf.CBmemsys) ##[2:20](memsys.grant ==
        2'b00));
end

repeat(256)
begin
    randVar1 = $random(); // get 32 bit random variable
    address1 = randVar1[13:6]; // get random 8-bit address
    data1 = randVar1[29:22]; // get random 8-bit data
    @vintf.CBmemsys vintf.CBmemsys.request[1] <= 1'b1;
        // request the bus
    expect(@(vintf.CBmemsys) ##[2:20]  (memsys.grant ==
        2'b10));
    writeOp(address1, data1); // issue write operation
    @vintf.CBmemsys vintf.CBmemsys.request[1] <= 1'b0;
        // release request
    expect(@(vintf.CBmemsys) ##[2:20] (memsys.grant ==
        2'b00));
    @vintf.CBmemsys vintf.CBmemsys.request[1] <= 1'b1;
        // request again
    expect (@(vintf.CBmemsys) (memsys.grant == 2'b10));
    readOp(address1, data1); // issue read operation
    @vintf.CBmemsys vintf.CBmemsys.request[1] <= 1'b0;
        // release request
    expect( @(vintf.CBmemsys) ##[2:20](memsys.grant ==
        2'b00));
end
join
```

This test works well in exhaustively checking the read and write operations for each CPU. However, because the CPUs are operating concurrently, problems can arise when each CPU accesses the same address space with different data. For instance, if CPU0 first writes to an address space and then CPU1 writes to the same address space, the data that CPU0 reads will be different from what it expects (it reads the data that CPU1 wrote). This will result in simulation failure because of the discrepancy between data read and data expected. A solution to this issue is to use basic concurrency control and this is discussed in the section following **`object-oriented programming and random stimulus generation.`**

Object-Oriented Programming

Object-oriented programming allows you to develop programs that are easier to debug and reuse by encapsulating related code (methods) and data (properties) together into what is called a class. In this section, you will examine how classes can be implemented in our memory system using polymorphism to associate specific signals with each object (instance) of a class, how classes are constructed, how concurrency-control is achieved, how concurrent processes communicate and finally how automatic constrained randomization of stimulus is achieved.

Encapsulation

A class is a collection of data and a set of methods that act on that data. A class’s data is referred to as properties, and a class’s methods are referred to as methods. An instance of a class is called an object, and an object is comprised of the class’s properties and methods.

Class properties are instance-specific. Each instance of a class has its own copy of the variables declared in the class definition.

Because multiple instances of classes can exist, when calling a class method, you must identify the instance name for which the method is being called. This is because each method only accesses the properties associated with its object, or instance. So, when calling a method, you must use this syntax:

```
instance_name.method_name();
```

Constructors

Objects, or instances, are created when a class is instantiated using the **`new`** statement:

```
class_name instance_name = new;
```

This declaration creates an instance (called *`instance_name`*) of the class *`class_name`*. When this construction takes place, the method *new()*, if any exists within the class, is executed. By defining the *task new()* within a class, you can initialize the class upon construction or instantiation. Further, by passing arguments to the constructor, you can allow for runtime customizing of the object:

```
class_name instance_name = new(argument1, argument2, ...
argumentN);
```

Using this constructor, the specified arguments are passed to the *function new* within the class. The conventions for these arguments are the same as those of the usual SystemVerilog function calls.

Implementing a Class

In the memsys example, since you have two CPUs (CPU0 and CPU1), you first declare a class called cpu, so that each CPU can be represented by an object of this class. This is done in the following manner:

```
class cpu;

     //properties
     arb localarb;
     local integer cpu_id;
     bit [7:0] address;
     bit [7:0] data;
     integer delay;
     string name;
     //methods
     function new(arb arb, integer id, string n);
       ...
     endfunction
     task readOp();
        ...
     endtask
     task writeOp();
       ...
      endtask
     task request_bus();
       ...
     endtask
     task release_bus();
       ...
      endtask
     task delay_cycle();
       ...
     endtask
endclass
```

Currently, all methods declaration and definitions have to be kept within the class. This limitation will be removed in the future releases.

Synchronous Signal Assignment

When implementing object-oriented concepts in your system, you should make specific Synchronous signal assignments to each instance or object of a class using virtual ports. Since each of the two CPUs is represented by an object of the class cpu and accesses the system bus through the common arbiter, we declare a base class arb and then, using it, declare two extended classes, arb0 and arb1, one for use with each of the two class cpu objects. Thus, we ensure each CPU connects to the arbiter using its specific device signals. The code for declaring a base class for arbiter as following:

```
//object of this base class can not be created directly
virtual class arb;
   extern virtual task driveRequest(input bit s);
   extern virtual function logic getGrant();
endclass
```

Using this class declaration, we declare two extended classes, one for each CPU, as follows:

```
class arb0 extends arb;
    task driveRequest(input bit s);
        vintf.CBmemsys.request[0] <= s;
    endtask

    function logic getGrant();
        return vintf.CBmemsys.grant[0];
    endfunction
endclass

class arb1 extends arb;
    task driveRequest(input bit s);
        vintf.CBmemsys.request[1] <= s;
    endtask
    function logic getGrant();
        return vintf.CBmemsys.grant[1];
    endfunction
endclass
```

Depending on the CPU object that is invoked, the corresponding arbiter object gets passed to its class methods and determines which device signals get affected.

Class Methods

In your class cpu, you must create the initialization method that is executed when the class is constructed. You must then create the read and write operation methods. It is also helpful to create methods to request and release the bus.

The initialization method should pass in the object of type `arb` (as declared above) and assign it to a local property. The code for the initialization method new is the following:

```
function new (integer id) ;
  $write("Constructing new CPU.\n");
  case (id)
    0: begin arb0 a = new; localarb = a; end
    1: begin arb1 a = new; localarb = a; end
    default : assert(0 && "unknown cpu id\n");
  endcase
  cpu_id = id;
endfunction
```

The read operation `readOp` must behave as before. Depending on the object of the class cpu that is invoked, the `readOp` class method only applies to the CPU associated with that object. The code for the `readOp` class method is the following:

```
task readOp () ;
  $write("CPU %0d readOp: address %h data %h\n", cpu_id,
     address, data);
  @vintf.CBmemsys vintf.CBmemsys.busAddr <= address;
  vintf.CBmemsys.busRdWr_N <= 1'b1;
  vintf.CBmemsys.adxStrb <= 1'b1;
```

```
  @vintf.CBmemsys vintf.CBmemsys.adxStrb <= 1'b0;
  expect (@(vintf.CBmemsys) ##[2:5] vintf.CBmemsys.busData
     == data);
  $write("READ address = 0%H, data = 0%H \n", address, data);
Endtask
```

The write operation `writeOp` must behave as before. Depending on the object of the class cpu that is invoked, the `writeOp` class method only applies to the CPU associated with that object. The conditional statement inside the method evaluates the cpu_id was passed in the initialization process and thus the method is able to print which CPU is writing. The code for the `writeOp` class method is the following:

```
task writeOp() ;
  $write("CPU %0d writeOp: address %h data %h\n", cpu_id,
address, data);
  @vintf.CBmemsys vintf.CBmemsys.busAddr <= address;
  vintf.CBmemsys.busData <= data;
  vintf.CBmemsys.busRdWr_N <= 1'b0;
  vintf.CBmemsys.adxStrb <= 1'b1;
  @vintf.CBmemsys vintf.CBmemsys.busRdWr_N <= 1'b1;
  vintf.CBmemsys.busData <= 8'bzzzzzzzz;
  vintf.CBmemsys.adxStrb <= 1'b0;
  $write("CPU%0d is writing \n", cpu_id);
 $write("WRITE address = 0%H, data = 0%H \n", address, data);
Endtask
```

Our request_bus method must assert the corresponding CPU's request line and then check for the appropriate grant line. This is done with the help of the associated bind, which was passed to the local property localarb. The code for the request_bus class method is the following:

```
task request_bus ();
  $write("CPU%0d requests bus on cpu%0d\n", cpu_id, cpu_id);
  @vintf.CBmemsys localarb.driveRequest(1'b1); // request
                                               // the bus
```

```
  expect (@(vintf.CBmemsys) ##[2:20] localarb.getGrant() ==
     1'b1);
endtask
```

Conversely, our release_bus method must release the corresponding CPU's request line and then check for the release of the appropriate grant line. This is done with the help of the associated object, which was passed to the local property localarb. The code for the *release_bus* class method is the following:

```
task release_bus ();
  $write("CPU%0d releases bus on cpu%0d\n", cpu_id, cpu_id);
   @vintf.CBmemsys localarb.driveRequest(1'b0); // request
                                                // the bus
  expect (@(vintf.CBmemsys) ##[1:2] localarb.getGrant() ==
     1'b0);
endtask
```

Finally, you write the method delay_cycle() to introduce the delay between CPU accesses. The code for the delay_cycle class method is the following:

```
task delay_cycle();
  $write("CPU%0d Delay cycle value: %d\n", cpu_id, delay);
  repeat(delay) @vintf.CBmemsys;
  $write("delay = %d \n",delay);
Endtask
```

Constrained Randomization of Stimulus

Previously, you had generated the `data`, `address` and `delay` properties of the class cpu using the **`random()`** function. Now, you will learn to do the same using the powerful automatic randomization feature of SystemVerilog for Testbench. Further, you will also learn how to specify constraints to automatically constrain certain random properties (variables).

You can declare class properties as random using the **`rand`** declaration:

rand *`data_type variable = initial_value;`*

Variables declared as random within a class are randomized when the **`randomize()`** system function is called. Because **`randomize()`** acts as a class method, you must specify the instance for which the system function is called:

```
function int object_name.randomize();
```

object_name

The name of the object in which the random variables have been declared.

The **`randomize()`** class method generates random values for all random variables within the specified class instance. The **`randomize()`** method returns a 1 if it successfully sets all the random variables and objects to valid values. If it does not, it returns a 0. If an object has no random variables anywhere in its inheritance hierarchy (no random variables or sub-objects) or if all of its random variables are inactive, the **`randomize()`** function returns a 1.

Using random declarations, we declare our class properties `address`, `data` and `delay` as random as follows:

```
randc bit[7:0] address;
rand bit[7:0] data;
rand integer delay;
```

Each time an instance is randomized, the `address`, `data` and `delay` values for that instance are randomized. In particular, `address` has to be declared cyclic random (randc), for we want it to cycle through all the 256 memory addresses without repetition.

Note that there are no restrictions on the value that `delay` can assume because it is declared as a random integer. We can implement constraints on the values that random variables can assume using the **`constraint`** construct:

constraint *`constraint_name`* { *`contraint_expressions`* }
`constraint_name`

The name of the constraint block.

constraint_expression

The conditional expression that limits random values. It is a series of expressions that are enforced when the class is randomized.

To limit the values of *delay* to those between 0 and 10, we define this constraint within the class:

```
constraint del_lt10
{
     delay < 10;
     delay >=0;
}
```

Implementing Constraints and Randomization

First, introduce the changes described in the previous subsection in the declaration of the cpu class. The cpu class would now appear as the following:

```
class cpu;
     //properties
     arb localarb;
     integer cpu_id;
     randc bit [7:0] address;
     rand bit [7:0] data;
     rand integer delay;
     constraint del_lt10
```

```
    {
        delay < 10;
        delay >= 0;
    }
    //methods
        ...
endclass
```

Implementing Object-Oriented Programming

Before we can use our objects, we must instantiate each cpu class object and invoke our initialization routines in the following manner:

```
cpu cpu0 = new (0);
cpu cpu1 = new (1);
```

With your class cpu defined with the properties and methods described above, the same concurrent execution sequences for the two CPUs created earlier using fork/join can now be written as follows:

```
fork
//fork CPU 0

    repeat(256)
    begin
        cpu0.randomize();
        cpu0.request_bus();
        cpu0.writeOp();
        cpu0.release_bus();
        cpu0.request_bus();
        cpu0.readOp();
        cpu0.release_bus();
        cpu0.delay_cycle();
    end

//fork CPU 1
    repeat(256)
    begin
```

```
            cpu1.randomize();
            cpu1.request_bus();
            cpu1.writeOp();
            cpu1.release_bus();
            cpu1.request_bus();
            cpu1.readOp();
            cpu1.release_bus();
            cpu1.delay_cycle();
        end
join
```

Note how the class property `address` is passed (using the instance name). Also, note the ease of reuse through invoking the class methods with the appropriate instance name.

Basic Concurrency Control

The two objects of the class cpu that were invoked concurrently in the previous subsection have the potential to access the same location in the SRAM memory if the random addresses generated for them happen to be the same. To prevent a potential conflict between the two cpu objects from leading to a data or resource hazard, concurrency control can be achieved using semaphores.

Semaphores

Conceptually, a semaphore is like a bucket containing a number of keys. No process can execute without first obtaining a key. Therefore only as many processes as there are keys can execute at any time. All other processes must wait until the keys are returned.

Semaphore is a built-in class that provides the following methods:

- Create a semaphore with a specified number of keys: new()
- Obtain one or more keys from the bucket: get()

- Return one or more keys into the bucket: put()
- Try to obtain one or more keys without blocking: try_get()

```
new (number_of_keys)
```

It specifies the initial number of keys in the semaphore.

```
put(number_of_keys)
```

Increments the number of keys in the semaphore.

```
get(number_of_keys)
```

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, VCS halts simulation of the process (initial block, task, etc.) until there the put method in another process increments the number of keys to the sufficient number.

```
try_get (number_of_keys)
```

Decrements the number of keys in the semaphore. If there aren't the specified number of keys in the semaphore, this method returns a 0. If the semaphore has the specified number of keys, this method returns 1. After returning the value, VCS executes the next statement.

Implementing Semaphores

The fork/join code that you wrote earlier should now be modified to include semaphores as follows:

```
task check_all () ;
  bit[7:0] mem_add0[*], mem_add1[*];
  semaphore semaphoreId = new(1);
```

```
  $write("Task check_all:\n");
  fork
     // fork process for CPU 0
      repeat(256) begin
        if(cpu0.randomize() == 0) begin
           $display("Fatal Error Randomization Failed.
Exiting\n");
           $finish;
        semaphoreId.get(1);
        cpu0.request_bus();
        cpu0.writeOp();
        cpu0.release_bus();
        cpu0.request_bus();
        cpu0.readOp();
        cpu0.release_bus();
        semaphoreId.put(1);
        cpu0.delay_cycle();
      end


    // fork process for CPU 1
      repeat(256) begin
        if(cpu1.randomize() == 0) begin
           $display("Fatal Error Randomization Failed.
Exiting\n");
           $finish;
        semaphoreId.get(1);
        cpu1.request_bus();
        cpu1.writeOp();
        cpu1.release_bus();
        cpu1.request_bus();
        cpu1.readOp();
        cpu1.release_bus();
        semaphoreId.put(1);
        cpu1.delay_cycle();
      end
  join
endtask
```

Interprocess Communication and Synchronization

Up until now, you had both CPUs operating concurrently, with each going through the read and write cycles completely. Now, you may want to do things slightly differently by making CPU0 only write to the memory and CPU1 only read from the memory.

However, you must ensure that CPU0 writes before CPU1 reads and that the address generated by CPU0 is passed on to CPU1. Also, the data generated by CPU0 has to be passed on to CPU1 so that it can check for memory corruption by comparing the data read from the memory with that passed to it by CPU0. This interprocess communication in which CPU0 passes the address and data to the waiting CPU1 can be achieved with **mailboxes**. Further, you must also ensure that CPU1 finishes with its read operation and checks completely before CPU0 starts its next write cycle. This final synchronization can be achieved with **events.**

Mailboxes

A mailbox is a mechanism to exchange messages between processes. Data can be sent to a mailbox by one process and retrieved by another. Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, you can retrieve the letter (and any data stored within). However, if the letter has not been delivered when you check the mailbox, you must choose whether to wait for the letter or retrieve the letter on subsequent trips to the mailbox. Similarly, SystemVerilog for Testbench’s mailboxes allow you to transfer and retrieve data in a very controlled manner

Mailbox is a built-in class that provides the following methods:

- Create a mailbox: new()

- Place a message in a mailbox: put()
- Retrieve a message from a mailbox: get() or peek()
- Try to retrieve a message from a mailbox without blocking: try_get() or try_peek() - Retrieve the number of messages in the mailbox: num()

put(expression)

Puts a message in the mailbox.

get(variable)

Assigns the value of the first message to the variable. VCS removes the first message so that the next message becomes the first method. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a put method put a message in the mailbox.

try_get(variable)

Assigns the value of the first message to the variable. If the mailbox is empty, this method returns the 0 value. If the variable is available, this method returns the 1 value. After returning the value, VCS executes the next statement.

peek(variable)

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, VCS suspends simulation of the process (initial block, task, etc.) until a put method put a message in the mailbox.

try_peek(variable)

Assigns the value of the first message to the variable without removing the message. If the mailbox is empty, this method returns the 0 value. If the variable is available, this method returns the 1 value. After returning the value, VCS executes the next statement.

Synchronization Between Concurrent Processes

Concurrent processes can be synchronized with each other using events. First, an event variable `CPU1done` is declared. This event is then triggered by one of the processes using event triggering operator ->. Synchronization of the other process to the one is achieved by using a wait or event control operator @.

Event variable serve as the link between triggering and synchronizing processes. They are bi-directional as they can be used both to pass and receive triggers.

Triggers are used to pass or send events. A trigger is initiated by using event triggering operator as follows:

```
->event_name;
```

Waits are used to receive events and thereby synchronize the receiving processes with the triggering processes. The triggered event property is used in the context of a wait construct:

```
wait (hierarchical_event_identifier.triggered)
```

Using this mechanism, an event trigger shall unblock the waiting process whether the wait executes before or at the same simulation time as the trigger operation. The triggered event property, thus, helps eliminate a common race condition that occurs when both the

trigger and the wait happen at the same time. A process that blocks waiting for an event might or might not unblock, depending on the execution order of the waiting and triggering processes. However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the wait and trigger operations.

Implementing Interprocess Communication and Synchronization

In order to avoid any conflicts between the two class cpu objects, the fork/join construct that was seen earlier is modified with the help of a mailbox, a trigger and a wait. Now, both objects CPU0 and CPU1 access the same memory location using the random address generated by CPU0. However, they do so not at the same time. While CPU0 writes to a random memory location, CPU1 waits for that random address and the data to be passed to it by CPU0 through a mailbox. CPU1 then accesses the same memory location to read the data. Like wise, CPU0 waits for a triggered event from CPU1 which indicates CPU1 has finished its write cycle and therefore CPU0 can start its next write cycle.

```
integer randflag;
event CPU1done;
mailbox mboxId = new;

$write("Task check_all:\n");

fork
  // fork process for CPU 0
    repeat(256) begin
       $write("\nthe start of block 1  at %t\n\n", $time);
      randflag = cpu0.randomize();
      cpu0.request_bus();
      cpu0.writeOp();
      cpu0.release_bus();
      mboxId.put(cpu0.address);
```

```
        mboxId.put(cpu0.data);
        mboxId.put(cpu0.delay);
        wait(CPU1done.triggered);
        CPU1done = 0;
        cpu0.delay_cycle();
      end

    // fork process for CPU 1
      repeat(256) begin
         $write("\nthe start of block 2  at %t\n\n", $time);
        mboxId.get(cpu1.address);
        mboxId.get(cpu1.data);
        mboxId.get(cpu1.delay);
        cpu1.request_bus();
        cpu1.readOp();
        if (CBmemsys.busData == cpu1.data)
          $write("\nThe read and write cycles finished
              successfully\n\n");
        else
          $write("\nThe memory has been corrupted\n\n");
        cpu1.release_bus();
        ->CPU1done;
        cpu1.delay_cycle();
      end
  join
```

Finally, you can structurally make your code more elegant by encasing the code for the fork/join in a task called check_all() and then calling this and other tasks from inside the program memsys_test in the memsys_test.v file as follows:

```
initial begin
   init_ports();
   reset_sequence();
   check_all();
end
```

Functional Coverage

In this tutorial, SystemVerilog functional coverage capabilities are built within the object-oriented framework. You specify what you want VCS to monitor for coverage with the **covergroup** construct. The definition inside **covergroup** includes valid and invalid states and transitions, a measure of coverage, the coverage goal, and coverage options. The official description of the **covergroup** construct begins on page 306 of the SystemVerilog 3.1a LRM.

Implementing Coverage Objects

We will create two coverage groups. We first want to check that the entire address space is tested. We monitor the state variable busAddr for the memory controller that it assumes all valid states between 0 and 255:

```
enum logic [1:0] {IDLE, START, WRITE0, WRITE1} st;

covergroup range @(negedge
memsys_test_top.dut.Umem.adxStrb);
   a: coverpoint memsys_test_top.dut.Umem.busAddr {
      bins m_state[] = {[0:255]};
   }
endgroup
```

We also want to monitor the state machine for the memory controller if all the valid state transitions have been covered:

```
covergroup cntlr_cov @vintf.CBmemsys;
     b: coverpoint memsys_test_top.dut.Umem.state {
          bins t0 = (IDLE => IDLE);
          bins t1 = (IDLE => START);
          bins t2 = (START => IDLE);
          bins t3 = (START => WRITE0);
          bins t4 = (WRITE0 => WRITE1);
          bins t5 = (WRITE1 => IDLE);
```

```
        bins bad_trans = default sequence;
    }
endgroup
```

We must instantiate these two coverage groups within our main program to create our coverage objects:

```
range cov1 = new;
cntrlr_cov cov2 = new;
```

The command line to generate the coverage ASCII text file after the coverage database is created is as follows:

```
% vcs -cov_text_report memsys_test.db
```

3

Introducing Reference Verification Methodology

This tutorial is a beginner's guide to using the Reference Verification Methodology (RVM), with the OpenVera language. You can simulate your testbenches with Vera or VCS's Native Testbench (NTB). With RVM, you can quickly build a layered testbench. These testbenches support high-level tests using constrained random stimulus and functional coverage to indicate which areas of the design have been checked.

In this tutorial, you will verify an ATM switch, starting with simple stimuli and then working up to a complex testbench.

Requirements

This section outlines the requirements that you must have in order to perform this tutorial.

Knowledge

- You should know a hardware design language such as Verilog or VHDL. This tutorial currently uses Verilog.
- You should have basic knowledge of the OpenVera language, especially concepts such as Object Oriented Programming (OOP) and synchronization between the testbench and Design Under Test (DUT). If you have not used virtual methods in OOP, you should first read Appendix A.
- You should be familiar with using either VCS NTB or Vera to compile and simulate designs plus testbenches.

Tools

This tutorial requires VCS with Native Testbench (7.2 and higher) or Vera (6.2 and higher) plus another Verilog simulator.

References

You should refer to documentation and examples contained in the releases of the tools used. The *Reference Verification Methodology User Guide* is a useful reference for these labs. This document is available through the HTML online documentation system.

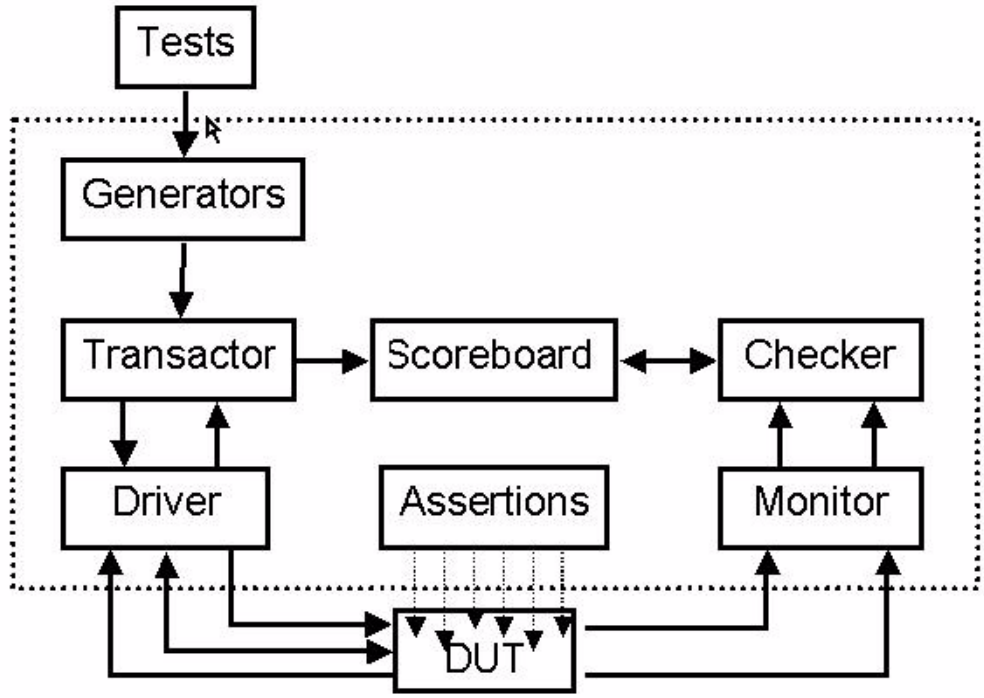
Overview

To get the most out of a Hardware Verification Language (HVL) such as OpenVera, you need to adopt a new methodology. If you use the same techniques from your old Verilog or VHDL testbenches (directed tests with little randomization), you will not find bugs in your design as quickly as if you tried a new approach. Switching will also help make your code easier to maintain and reuse.

RVM consists of coding guidelines and a set of base classes that allow you to develop reusable testbench components such as data models, transactors, and generators. RVM is based on years of experience verifying many different types of designs, enabling use of advanced techniques. Using RVM will give your project a consistent look and feel, so less time is spent creating verification infrastructure, and more time verifying your design.

The RVM methodology involves creating a robust, flexible testbench infrastructure once, and then creating many simple tests, while never changing the underlying testbench. This requires advanced techniques such as factory patterns and callbacks. For example, virtual methods allow you to write the testbench once, and then not have to modify it for every possible type of stimulus variant such as error injection, synchronization, and variable delays. While you can verify a design using a simple testbench, you would have to create many elaborate tests and continually update the testbench. This latter approach yields more code and reduces the readability and maintainability of your code.

Layered Environment



The three major parts of a verification environment are the DUT, the testbench (the inner box above), and the test which controls the testbench. Inside the testbench are the following parts:

- The Driver controls the signals into the DUT. You can write it in OpenVera or Verilog. It executes single commands such as a bus read or write, or driving a cell / packet / frame into the DUT
- The Monitor bundles signal changes from the DUT into transactions.
- Assertions constantly check the DUT for correctness. These look at external and internal signals. The testbench uses the results of these assertions to see if the DUT responded correctly.

- The Transactor takes high-level transactions such as a burst-read into individual read commands, or a single USB transaction into multiple USB packet TX/RX commands.
- The Scoreboard stores the transactions from the Transactor for later comparison.
- The Checker compares the output of the DUT, as seen by the Monitor, to the Scoreboard, using a predictor or reference model.
- The Generator generates transactions, either individually or in streams.

Note that each of these components may be instantiated several times or come in several flavors for different protocols.

A goal of RVM is that the testbench will not need to change for individual tests. Its components include hooks to allow the testbench to control the stimulus, without having to anticipate all possible conditions such as error injection.

Labs

The files for this tutorial are in the following directory:

$VCS_HOME/doc/UserGuide/examples-pdf/rvm_tutorial_ov/labs

Each lab exists in a separate directory, allowing you to complete the labs in any order. Each lab has some existing code and comments indicating where you need to complete code, as shown below:

```
// Lab1: ...  comments.
```

Solutions to all labs are in the solutions directory. You should examine the tips and hints sections that follow each lab before consulting the solutions.

During the labs, you will construct a verification environment to verify a very simple ATM switch.

- The switch has a single input port that accepts ATM cells using a 32-bit data bus.
- The switch has four output ports that send ATM cells using a 32-bit data bus.
- The switch algorithm is to look at the low 2 bits of the Generic Flow Control (GFC) field and route the packet to that port.
- Data is synchronous to the clock on the input port and is sampled on the rising edge.
- Each port has 32 data signals, data valid signal and a clock signal.

Messaging

A testbench produces messages of many types and severities. The rvm_log class lets you control which messages are displayed, what their format is, and even promote and demote them (useful for error testing). All messages are sent to standard output, i.e. displayed on the screen and also sent to the simulation log file, just like printf messages in OpenVera and $display in Verilog.

Type and Severity

In RVM, every message has a type and a severity. You may want to print a message to debug a piece of code, tell the user that simulation reached a notable state, or encountered a problem. The message type tells which of these is happening.

- Failure: Error has been detected
- Note: Simulation progress
- Debug: Optional simulation diagnostics
- Timing: Timing check or error

The severity field describes the message importance. The following list shows the severity levels and the type in parenthesis.

- Fatal
  - Functional correctness definitely compromised (Failure)
  - Example: Testbench failure
- Error
  - Functional correctness may be compromised (Failure)
  - Example: Actual model results don’t match expected results
- Warning
  - Functional correctness not compromised (Failure or Timing)
- Normal
  - Regular, expected message
- Trace
  - High-level simulation execution trace message (Debug)
  - Example: "Executing transaction"
- Debug
  - Detailed simulation execution trace message (Debug)

  - Example: "Waiting for acknowledge"
- Verbose
  - Very detailed simulation execution trace message (Debug)
  - Example: "Sending byte #5 (0x5A)"

rvm_log class

Each part of the testbench (test, generator, checker, etc.) uses its own instance of the rvm_log class to generate messages. Each instance is a separate message source with a descriptive name and an instance name. You can use regular expressions to select and control sources so use clear names. Usually the descriptive name is the name of the class instantiating rvm_log, and the instance name is the name of the object, or “class” if there is only a single instance.

Declaration and Instantiation

The rvm_log is usually instantiated inside a testbench object such as a generator or checker, or in a data object:

```
rvm_log log;
log = new("name", "instance");
```

The name string is the name of the class that contains the log, such as “USB Host”, or “MAC Frame”. The *instance* string is the name of this instance of the object such as “Generator 1”, or “Left side”. If there is only a single instance, just use the string “class”.

The easiest way to use an rvm_log object is with the macros:

```
rvm_fatal(rvm_log log, string msg);
rvm_error(rvm_log log, string msg);
```

```
rvm_warning(rvm_log log, string msg);
rvm_note(rvm_log log, string msg);
rvm_trace(rvm_log log, string msg);
rvm_debug(rvm_log log, string msg);
rvm_verbose(rvm_log log, string msg);
```

Here are two examples of using the above messages. The first displays a simple string. The second needs to print variable arguments, so it uses psprintf, which returns a formatted string:

```
rvm_verbose(log, "Checking rcvd byte");
if (byte != expect) {
  rvm_error(log, psprintf("Bad data: 0x%h vs. 0x%h",
                          byte, expect));
}
```

Note that these macros expand to several lines, so surround them with curly braces { } when used in an if-statement.

Coding Guideline:
Avoid declaring and instantiating an object all on one line. You will not be able to call any procedural code before the first call to new(). Instead of:

```
rvm_log log = new("name", "instance");  // Poor code
```

Use:

```
rvm_log log;                    // Separate declaration
log = new("name", "instance"); // from instantiation
```

Note that this tutorial occasionally skips this rule to make the examples more readable.

Message Handling

The messaging class handles each message according to its severity level. The default is that fatal messages cause the simulation to exit, error messages increment a global error count, and cause the simulation to exit after 10 errors, while all others just print to standard out. You can use the method rvm_log::modify() to change how messages are handled.

Controlling Verbosity

By default, only messages with a severity of NORMAL (rvm_fatal, rvm_error, rvm_warning, rvm_note) or higher are displayed. You can control this two ways:

Using +rvm_log_default

The command line switch +rvm_log_default=DEBUG will enable printing of all messages with the severity level DEBUG and higher. Other settings are WARNING, NORMAL, TRACE, or VERBOSE.

Using set_verbosity()

The method rvm_log::set_verbosity() allows you to set the level of printing on the fly. The following code sets the level to DEBUG for any rvm_log object with “Drv” in its name:

```
log.set_verbosity(log.DEBUG_SEV, "/Drv/", "/./", );
```

The regular expression "/./" matches any string, so the following matches all rvm_log objects, regardless of their name or instance name:

```
log.set_verbosity(log.DEBUG_SEV, "/./", "/./", );
```

Note that since set_verbosity and DEBUG_SEV are part of the rvm_log class, they must be prefixed with a handle to that class.

Also note that this call overrides the +rvm_log_default switch, and only applies to current rvm_log objects, not any created afterwards.

Further Exploration

- Call rvm_log::report() at the end of simulation to display the number of warnings and errors.
- You can change the formatting of rvm_log using the format() method.
- You can create complex, multi-line messages using the rvm_log methods start_msg(), text(), and end_msg().

See the RVM User Guide for more information and examples.

Lab 1 – Message Service

Testbench components create various types of output messages during each simulation. The ability to control the level of detail from each component in the testbench is critical during debugging of the design and test environment.

Recommended Time: 30 Minutes

References:

- Reference Verification Methodology User Guide: Common Message Service
- RVM Testbench Methodology User Guide :Appendix A Message Reporting Class, rvm_log class

The prepared material for this lab is in the "lab1" subdirectory. Please change your working directory to this directory before proceeding.

```
% cd lab1
```

To use the base classes, you must include the RVM library in your code:

```
#include <rvm_std_lib.vrh>
```

Step 1 – Log Messages

During this lab, you will create a program that issues verbose, debug, note, warning and error messages. This will demonstrate the severity and verbosity capabilities provided by the RVM environment.

Start by reviewing the test01.vr file, and adding code for the following elements:

- rvm_log log1

  ```
  log1 = new("Main", "Logger1");
  ```

- Issue the messages using the following logging macros:
  - "Error Message" using rvm_error()
  - "Warning Message" using rvm_warning()
  - "Note Message" using rvm_note()
  - "Debug Message" using rvm_debug()
  - "Verbose Message" using rvm_verbose()

Compile and run the test using the following command:

```
% gmake lab1 (or gmake ntb – for VCS NTB)
```

Note the format of each message. Each message contains a name and instance name that identifies the source of the message.

ERROR [Failure] on Main (Logger1) at 0 ns:

error message

WARNING [Failure] on Main (Logger1) at 0 ns:

warning message

Do all the messages appear? Y / N

If so, why, if not, why not?_____

_____

Step 2 – Global Verbosity

Experiment with the global verbosity level via the command line by using the commands:

```
% gmake lab1 (or gmake ntb – for VCS NTB)
% vera_cs +vera_load=obj/test01.vro
% vera_cs +vera_load= obj/test01.vro
+rvm_log_default=NORMAL
% vera_cs +vera_load= obj/test01.vro +rvm_log_default=DEBUG
% vera_cs +vera_load= obj/test01.vro
+rvm_log_default=VERBOSE
```

Note which messages are displayed depending on the debug level and complete the table below by adding ü where the message is displayed:

Cmd Line Severity	Log1				
	Error	Warn	Note	Dbg	Verb
None Specified	P	P	P		
Note					
Debug					
Verbose					

Step 3 – Instance Verbosity

RVM objects usually contain, at most, one log instance. Multiple log instances are used in this example to demonstrate filtering. These log instances are usually present in different RVM instances.

Edit the file test01.vr add a second logger by:

- Add code: rvm_log: log2: name = “Main”, instance = “Logger2”
- Copy the log macros “rvm_error(), rvm_warning(.) … etc” and change the log instance to log2

Run the simulation using the five commands of step 2, and verify that both loggers display all messages.

Edit the file test01.vr, and set the second logger severity by adding the following code just after log1 and log2 are declared and new’ed.

```
log1.set_verbosity(log1.WARNING_SEV);
```

Run the simulation using the four commands above, and note that the messages are different from each logger. Complete the table below by adding P where the message is displayed:

Cmd Line Severity	Log1					Log2				
	Error	Warn	Note	Dbg	Verb	Error	Warn	Note	Dbg	Verb
None Specified										
Normal										
Debug										
Verbose										

Modify the test again, adding the following statement directly below the lines just added

```
log2.set_verbosity(log2.DEBUG_SEV, “/M/”, "/2/");
```

Re-run the simulation, and note that the messages are different from each logger. Complete the table below by adding Pwhere the message is displayed:

Cmd Line Severity	Log1					Log2				
	Error	Warn	Note	Dbg	Verb	Error	Warn	Note	Dbg	Verb
None Specified										
Normal										
Debug										
Verbose										

Does the command line override the severity in the code? Y / N

Is this what you expected?

_____

_____

_____

Step 4 – Further Exploration

Advanced steps are optional.

Edit the test01.vr file and add one final message. This message should be issued using the log1 object, and be a multi-line message that displays as shown below.

Normal [Note] on Main(Logger1) at 0 ns:

Complex Message...

First line

Second line

Edit the test01.vr file to display the log messages on a single line. The default format settings are as follows:

```
void = log.format("%S [%T] on %N (%I) at %t ns:\n   %M", "
");
```

Verification Environment

Your testbench goes through many phases of execution, from initialization, simulation, and cleanup. The class rvm_env helps you manage these steps and ensures that all execute in the proper order.

Coding Guideline:
The new() method should only initialize values, and should never have any side effects such as spawning threads or consuming time. If a testbench object starts running as soon as new() is called, you will not be able to delay its start, or synchronize it with other testbench operations.

The Eight Steps

The rvm_env class divides a simulation into the following eight steps, with corresponding methods:

- gen_cfg() – Randomize test configuration descriptor
- build() – Allocate and connect test environment components
- cfg_dut_t() – Download test configuration into the DUT
- start_t() – Start components
- wait_for_end_t() – End of test detection
- stop_t() – Stop data generators and wait for DUT to drain
- cleanup_t() –Check recorded statistics and sweep for lost data

- report() – Print final report

Coding Guideline:
You can tell if a RVM method consumes time by the "_t" suffix. Thus, the gen_cfg() method will execute without any delays, while cfg_dut_t() may take several cycles to configure the DUT.

Simplest Example

Above all these methods is run_t() which keeps track of which steps have executed, and, when called, runs the remaining ones. For example, the following program runs all 8 steps automatically:

```
program test {
   verif_env env;
   env = new(...);
   env.run_t();
}
```

The class `verif_env` extends `rvm_env`. You call `run_t()` and it does the rest.

Basic Example

The next example runs the first step, makes a modification to the configuration, and then completes the test:

```
program test {
   verif_env env;
   env = new(...);
   env.gen_cfg();              // Create rand config
   env.rand_cfg.n_frames = 1; // Only run for 1 frame
   env.run_t();                // Run the other steps
}
```

Automatic Sequencing

run_t() is not the only method that executes the steps. As shown in the following example, if you call build() without calling gen_cfg(), the build() method will automatically execute the previous step:

```
class my_eth_fr extends eth_frame {
   rand bit [47:0] mac_address;
   constraint one_port_only {
      da == mac_address;          // Use fixed address
   }
}
program test {
   verif_env env
   env = new();
   env.build();                   // Config and build
   {
      my_eth_fr my_fr;
      my_fr = new();                // Use my own frame
      void = my_fr.randomize();
      env.src[0].rand_fr = my_fr; // Use to build more
   }
   env.run_t();
}
```

Using rvm_env

The following example defines the class verif_env. The virtual methods gen_cfg ( ) and build ( ) must call their super method as the first step. These calls to the base methods contain the sequencing code that ensures all previous steps have been called. If you leave out the calls, the rvm_env class will generate a fatal error at run time.

```
#include <rvm_std_lib.vrh>

class verif_env extends rvm_env {
   my_cfg cfg;
```

```
   my_gen gen[4];
   my_drv drv[4];
   my_mon mon[4];

   virtual task gen_cfg() {
      super.gen_cfg();
      // rest of gen_cfg method
   }
   virtual task build() {
      super.build();
      // rest of build method
   }
}
```

Detailed Explanation of Methods

gen_cfg()

This method creates a random configuration of the test environment and DUT. It may choose the number of input and output ports in the design and their speed, or the number of drivers on a bus and their type (master or slave). You can also randomly select the number of transactions, percent errors, and other parameters. The goal is that over many random runs, you will test every possible configuration, instead of the limited number chosen by directed test writers.

build()

This method builds the testbench configuration that you generated in the previous method: generators and checkers, drivers and monitors, and anything else not in the DUT.

cfg_dut_t()

In this method you download the configuration information into the DUT. This might be done by loading registers using bus transactions, or backdoor loading them using C code.

start_t()

This method starts the test components. This is usually done by starting the transactor objects, which will be described in section 8.

wait_for_end_t()

This method waits for the end of the test, usually done by waiting for a certain number of transactions or a maximum time limit.

The following example shows the wait_for_end_t ( ) method with its inner wait for the end of test event, plus a time-out statement. These are wrapped in a fork-join any that completes when either of these complete, and then a terminate command to stop the unfinished statement/threads.

```
class verif_env extends rvm_env {
   virtual task wait_for_end_t() {
      super.wait_for_end_t();   // Call super method
      fork                     // Limit scope of terminate
      {
         fork
           sync(ALL, this.sb.enough); // End of test event
            {
               delay(TIME_OUT);
               rvm_fatal(log, "Test did not complete");
            }
         join any
         terminate;
      }
```

```
    join
}
```

stop_t()

This method stops the data generators and waits for the transactions in the DUT to drain out.

cleanup_t()

Check recorded statistics and sweep for lost data.

report()

Print the final report. Note that rvm_log will automatically print its report at the end of simulation.

Further Exploration

Vera 6.4 and VCS after 7.2 offer an additional method,

rvm_env::reset_dut_t() between build() and cfg_dut_t so you can reset the design under test. Also explore class, rvm_watchdog which lets one monitor signals in the DUT, useful for wait_for_end_t().

Lab 2 – RVM Environment

The environment contains all permanent testbench elements or components. The environment also contains code to sequence the components through the well-defined steps during simulation. During this lab you will add debug code to each of the sequencing steps. You will then run a simple test to verify all steps execute correctly.

Recommended Time: 30 Minutes

References:

- Reference Verification Methodology User Guide: Design for Verification (DFV)
- Reference Verification Methodology User Guide: Common Message Service
- Reference Verification Methodology User Guide: Appendix A: Class Reference, rvm_env, rvm_log.

The prepared material for this lab is in the “lab2” subdirectory. Please change your working directory to this directory before proceeding.

```
% cd lab2
```

Step 1 – Review

The ATM environment contains several functions that implement the simulation phases or steps. Which of the following functions do you expect to be ‘virtual’?

new()	Virtual? Y / N
gen_cfg()	Virtual? Y / N
build()	Virtual? Y / N
cfg_dut_t()	Virtual? Y / N
start_t()	Virtual? Y / N
stop_t()	Virtual? Y / N
cleanup_t()	Virtual? Y / N

report() Virtual? Y / N

Review the lab2/atm_env.vr file, and see how many of these you answered correctly.

Check that all steps listed in the slides (and above) are present in the file. Are any steps missing in the list above?. (For solutions, see Appendix B: Answers to Lab Questions.)

Missing Functions:_____

Step 2 – Coding

The atm_cfg class contains the test and DUT configuration. The num_input_ports and num_output_ports data members set the number of driver and monitor transactors. This is randomized in `gen_cfg()` and the results are used to create testbench objects in `build()`.

Edit the lab2/atm_cfg.vr file to…

- Add a random integer data member (num_input_ports)
- Add a random integer data member (num_output_ports)
- Add a constraint “valid” that ensures:
  - num_input_ports is 1
  - rum_output_ports is 4

Edit the lab2/atm_env.vr file to…

- Add a rvm_log data member (log)
- Add a atm_cfg data member (cfg)
- In the new task, Allocate the cfg data member

- In the gen_cfg task, call super.gen_cfg(), then randomize the cfg data member
- In each of the remaining tasks, add a call to the parent (super) task.
- In each of the remaining tasks, add a rvm_debug message, displaying the class and function name.

In the test02.vr file, add code to:

- Instantiate the environment (atm_env)
- Set the environment debug level to VERBOSE_SEV
- Run the environment

Step 3 – Compile and Run

Compile the code using the makefile

```
% gmake lab2 (or gmake ntb – for VCS NTB)
```

Correct any compiler errors, and verify the output. Each step of the stimulation should print a debug message in the correct order. Check the messages match the steps in the RVM Class training slides. Each of the messages has a format similar to that shown below:

Debug [Debug] on rvm_env(class) at 0 ns:

```
    atm_env::gen_cfg()...
```

Logging Review: What does the “rvm_env” text refer to?_____

Logging Review: What does the “(class)” text refer to?_____

Step 4 – Advanced

If one (or more) of the super.method() calls in the atm_env.vr file are removed, what do you expect to happen? (This does not apply to super.report()).

_____
_____

Try commenting one of these lines out, and run the test.

Did the expected behavior happen? Y / N

If not – why?

_____

Hints

Review the RVM training slides, in particular the slide with all testbench environment execution steps listed.

Review the Vera User Guide for any Vera questions.

Constraints are defined using:constraint name { a == 0; }

Review the solution directory code briefly.

Data and Transactions

Traditionally, you implemented transactions as procedures, one per transaction. This caused the following problems:

- Code is not self-contained
- Code is not protected properly
- Cannot extend data types
- Cannot add constraints to an existing data type.

Instead, you should model transactions as objects. Their data values exist in a transaction class that can be randomized, copied, packed, and unpacked. The code to actually execute the transactions resides in the Driver.

Transaction Coding Guidelines

Properties in a transaction class should be public so they can be modified or constrained by other classes such as the testbench. Do not hide data values using set() & get() methods. methods. In hardware verification, you need access to all parts of the testbench for maximum control.

Properties should be random by default so that they can be randomized by other classes. You can always go back and use rand_mode() to turn this off.

Transactions vs. Transactors

The transaction class contains both physical values that are sent to the DUT (address, data, etc.) and meta-data that has extra information about the transaction, such as a “kind” field. Even though this might be encoded in the physical values, put it into a separate field that can be easily accessed and can be randomized.

```
enum kind_t = READ, WRITE;
class apb_transaction extends rvm_data {
   rand kind_t kind;
   rand bit [ 7:0] sel;
   rand bit [31:0] addr;
   rand bit [31:0] data;
}
```

The Transactor contains the code to send the transaction to the next testbench level. The following is a Driver to read and write to a real bus:

```
class apb_master {
   task do(apb_transaction tr) {
      case (tr.kind) {
         READ: {
            tr.data = this.read(tr.addr);
         }
         WRITE: {
            this.write(tr.addr, tr.data);
         }
      }
   }
```

Creating Your Own Transactions

The RVM recommends that you extend the RVM classes to create your own company-specific classes to build a buffer between the RVM and end users. In this buffer, you can customize the classes for your own best practices. This layer is most commonly added around rvm_data, though this tutorial uses the class directly.

ID Fields

Every transaction has three integer ID fields uniquely identifying it. The stream_id tells which stream created this object – useful when there are multiple generators. The sequence_id is used when a stream generator creates groups of related transactions, and identifies the groups. The object_id identifies individual transactions in a sequence. In the following example, the stream_id is used in a constraint block.

```
class rvm_data {
  integer stream_id;
  integer scenario_id;
  integer object_id;
  ...
}

class atm_cell extends rvm_data {
  rand integer has_vlan;
  …
}
class my_atm_cell extends atm_cell {
   ...
   constraint stream_0_is_vlan {
      if (stream_id == 0) has_vlan == 1;   }
}
```

Constraints

Every transaction should have one or more constraint blocks for the “must-obey” constraints that are never turned off or overridden. For example, they would make sure an integer field is never negative or that a length field is never 0. Name these constraints “class_name_valid”, such as “atm_cell_valid”.

You should have separate constraints for “should-obey”. You can turn these off later for injecting errors. Name these constraints “class_name_rule”.

Methods

The rvm_data class defines a set of virtual methods for manipulating its properties. (See Appendix A for a review of virtual methods.) Here are some of the ones used in this tutorial. You will need to make your own methods when you extend rvm_data.

display() & psdisplay()

These methods display the contents of the class, either to the screen or to a string.

```
virtual task              display (string prefix);
virtual function string   psdisplay(string prefix);
```

allocate()

This method allocates an rvm_data object and initializes required fields. This is a virtual method, unlike `new()` so the correct method is called regardless of the handle type.

```
virtual function rvm_data allocate();
```

copy()

This method makes a copy of an existing transaction. It has an optional “to” field so you can copy to a previously allocated object. Note that this method returns an `rvm_data` type, so you may need to `use cast_assign()` with it.

```
virtual function rvm_data my_data::copy(rvm_data to = null){
   my_data cpy;

   // Copying to a new instance?
   if (to == null) {
      cpy = new();
   } else

   // Copying to an existing instance. Correct type?
  // If destination handle is passed as argument during copy()
  //call,
   // use cast_assign to check that handle type is correct.
  // Argument handle has to be base type (rvm_data) or my_data
   // else abort.  IF argument is base type(rvm_data), cpy
   // handle is upcasted to rvm_data type (cpy -> "to").

   if (!cast_assign(cpy, to, CHECK)) {
      rvm_fatal(this.log,
          "Attempting to copy to a non my_data instance");
      copy = null;
      return;
   }
```

```
    // Copy ID's and any other properties
    cpy.stream_id   = this.stream_id;
    cpy.scenario_id = this.scenario_id;
    cpy.object_id   = this.object_id;

    // Assign the copy to the return handle
    copy = cpy;
}
```

There are two alternatives to the copy() method. The new() method does a shallow copy which may not be sufficient – encapsulated objects such as subfields and data arrays will not be copied. The object_copy() method does a deep copy which might be too much replication, especially if your transaction points to the testbench environment. Next, object_copy() is not available in SystemVerilog 3.1a. Lastly, object_copy() only duplicates existing values, and so you may want to increment pointers and check the validity of data. Both new and object_copy() force re-allocation of the destination handle, and therefore, do not allow for copying fields to an existing object.

The following example shows how to use the copy method:

```
my_data d1, d2, d3;  // Handles
d1 = new();          // Allocate object
d2 = d1;             // Both d1 & d2 point to same object
d2 = d1.copy();      // Error, copy returns rvm_data
cast_assign(d2, d1.copy()); // Good
void = d1.copy(d2);  // Copy d1 contents to d2 object
void = d1.copy(d3);  // Bug – d3 is still null
```

compare()

This method compares two objects and reports the difference.

```
virtual function bit compare (to, diff, kind);
```

The current object is compared with “to” using type “kind”. (See 4.3 for an example of “kind”.) The method returns 1 if the two objects are the same, 0 if not. The diff string gives a description of the difference.

Packing and unpacking

The following three methods are used for converting between the physical fields of an object and an array of bytes:

```
virtual function integer  byte_size  (kind);
virtual function integer  byte_pack  (bytes,offset,kind);
virtual function integer  byte_unpack(bytes,offset,kind);
```

The method byte_size tells how many bytes you need to pack an object of this kind. The method byte_pack packs the object of type kind into a dynamic array of bytes. The method byte_unpack unpacks the data from the dynamic array of bytes. The offset tells the methods where to start in the byte array.

Lab 3 – RVM Data

Data and transactions are modeled using classes in a RVM testbench. This lab will focus on creating an RVM compliant ATM cell class. (For solutions, see Appendix B: Answers to Lab Questions.)

Recommended Time: 30 Minutes

References:

- Reference Verification Methodology User Guide: Data and Transaction Models
- Reference Verification Methodology User Guide: Class Reference, rvm_data class

You can find the prepared material for this lab in the “lab3” subdirectory. Please change your working directory to this directory before proceeding.

```
% cd lab3
```

Step 1 – Data Members

The atm_cell.vr class contains the data members and methods for an ATM cell data object. The fields in an ATM cell (and bit widths) are as follows:

gfc	vpi	vci	pt	clp	hec	payload
Bits: 4	8	16	3	1	8	8*48

Review the file “`atm_cell.vr`”, and note the sections that require completion.

Edit the atm_cell.vr file to create a class as follows:

- Class Name: atm_cell, derives from: rvm_data
- Static Data Member: rvm_log
- Data Members: as above (gfc, vpi, vci, etc).

Why is the log data member static?

_____

_____

_____

Is the log data member static in all classes? Yes / No

Why?

_____
_

_____
_____

Step 2 - Methods

RVM requires certain methods be defined for rvm_data classes. RVM enforces this by:

1. Declaring these methods virtual in the rvm_data base class
2. Not defining a default implementation.

These functions are called ‘pure virtual’ and are defined this way in the base classes. As there is no default behavior defined in the base class, this forces the derived class to define their own behavior. Vera will issue compile time errors if any virtual functions are defined without a function body.

Complete the OpenVera code for atm_cell::copy() function. Add OpenVera code to copy one packet object to another by copying the individual data fields. Note the built-in copy method cannot be called, as the RVM copy method can copy data into a new instance. (this last sentence is convoluted. Should “as” be “since”?)

Review the code present that compares one cell with another.

Review the code for byte_pack and byte_unpack. Note how the OpenVera built-in tasks are called inside the RVM function names.

Review the display() task. Why is printf() used instead of a log instance? ___

_____

_____

_____

_____

Step 3 – Compilation & Testing

A small test suite is provided to test the atm_cell class. This test file creates packets, randomizes, copies packs and unpacks ATM cells. Note that this test uses a standalone environment, and does not use the ATM environment. Compile and test the class provided using command below.

```
% gmake lab3  (or gmake ntb – for VCS NTB)
```

How many “Pass (OK)” Messages were printed?

_____

How many “Fail (OK)” Messages were printed?

_____

Was this as expected?

_____

Why?

_____

_____

_____

_____

_____

_____

Hints

A foreach loop can be used instead of a ‘for’ loop, this simplifies the code and will work with arrays of any length.

```
foreach (array, i) { array[i] = . . . }
```

Variance is added by defining a discriminant property, and then testing this property in the tasks.

```
rand enum cell_kind = NORMAL, LONG;
task display() {
  // Existing code here…
  if (cell_kind == NORMAL) {
    for (i=0; i < 48; i++) printf(. . .);
  } else if (cell_kind == LONG) {
    for (i = 0; i < 96; i++) printf(. . . );
  }
```

Notification

RVM provides an event notification class that allows you to notify when an event notification is indicated and include data. These notifiers are based on integer identifiers that hold a symbolic value. For example, the following code creates three notification identifiers associated with the atm_driver class:

```
class atm_driver extends rvm_xactor {
   static integer TR_STARTED;
   static integer TR_ABORTED;
   static integer TR_SUCCESS;
}
```

You must configure a notifier before using it. A notifier can be ON_OFF_TRIGGER, ONE_SHOT_TRIGGER, HAND_SHAKE_TRIGGER or ONE_BLAST_TRIGGER. The

following example calls the configure() method in the notify object (pre-instantiated in rvm_xactor, described in a later section), which returns an integer value.

```
class apb_master extends rvm_xactor {
   task new(...) {
      this.TR_STARTED = this.notify.configure(*,
                                this.notify.ON_OFF_TRIGGER);
  }
 }
```

When you indicate an event, you can optionally attach an object derived from rvm_data. You use this to describe why you indicated the notification. You should not modify this object. If you need to change the object, use callbacks instead, shown later in this tutorial.

```
while (1) {
   atm_cell cell;
   ...
   this.pre_cell_tx_t(cell, drop);
   foreach (callbacks, i) {
      ...
   }
   if (drop) continue;
   this.notify.indicate(this.PRE_CELL_TX, cell);
   ...
}
```

Pre-defined Events

Many RVM classes include notification service interfaces, instances of rvm_notify. As shown above, rvm_data has an instance of rvm_notify and the events EXECUTE, STARTED and ENDED. The rvm_xactor class includes the notify properties XACTOR_IDLE, XACTOR_BUSY, XACTOR_STARTED, XACTOR_STOPPED, and XACTOR_RESET.

Further Exploration

When you use an RVM notification to pass an object, it should be treated as read-only. To pass an object that may be modified in more than one location in your testbench, use the rvm_broadcast class. This is able to buffer objects so you will not lose data.

Lab 7 – Notification

You can use RVM notifiers to communicate between objects in the testbench. Some standard notification identifiers are present in the RVM base classes, and are triggered by various testbench components.

During this lab, you will add a notification to the driver, to indicate when a cell has completed (ENDED). Add notification to the atomic generator to indicate when a cell is ready to be placed into the channel (GENERATED) and when enough cells have been generated (DONE).

Recommended Time: 30 Minutes

References:

- RVM Testbench Methodology User Guide: Class Reference: Appendix A rvm_notify class

The prepared material for this lab is in the "lab7" subdirectory. Please change your working directory to this directory before proceeding.

```
% cd lab7
```

Step 1 – Master Transactor

The master transactor will use a notification to indicate that a transaction has been sent to the DUT. RVM provides a building 'ENDED' notification inside all `rvm_data` objects.

Add an indicate call to the atm_driver.vr file, immediately after the process_cell() method has completed.

tr.notify.indicate(tr.ENDED);

Step 2 – Generator

Review and edit the "atm_generator.vr" file:

- Add a notification data member (integer) to the class – GENERATED
- Add a notification data member (integer) to the class - DONE
- Configure the GENERATED to be ONE_SHOT_TRIGGER
- Configure the DONE to be ON_OFF_TRIGGER
- Add a notification call for GENERATED just before the cell is placed into the output channel.
- Add a notification call for DONE to the end of the main_t() task

Step 3 – Update Environment

The atm_env must be updated to wait for the DONE notification produced by the generator.

In the atm_env.vr file, add the following code to the wait_for_end_t() task:

```
rvm_debug(log, "Waiting for atm_gen.DONE");
```

```
fork
{
  fork
  {
    void =
this.atm_gen.notify.wait_for_t(this.atm_gen.DONE);
  }
  {
    delay(TB_TIMEOUT_DELAY);
    rvm_fatal(log, "Test did not complete, TB timeout");
  }
  join any
  terminate;
}
join
rvm_debug(log, "Received atm_gen.DONE");
```

Step 4 – Compile and Test

Compile and run the test using the following command, and review the log produced.

```
% gmake lab7  (or gmake ntb – for VCS NTB)
```

Did the output change from lab6? Y / N

If so, how?

_____

_____

_____

_____

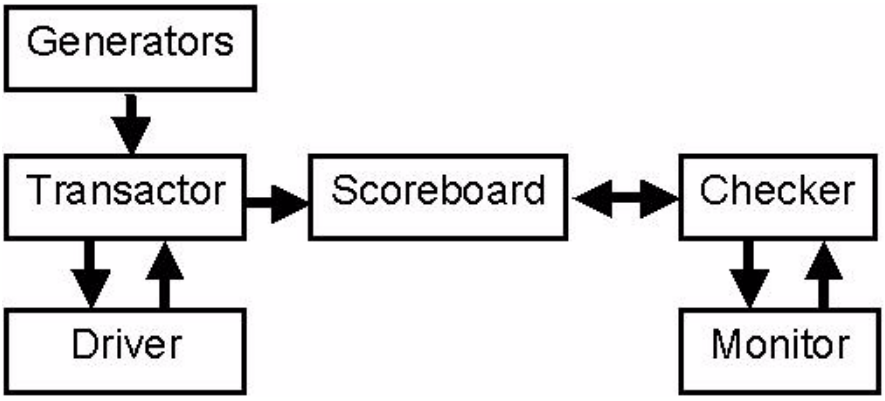
Hints:

The following task is used in the atm_driver to notify and pass a copy of the transaction:

```
tr.notify.indicate(tr.ENDED);
To initialize a notification data member, use:
NOTIFY_IDENTIFIER = notify.configure(*,
this.notify.TRIGGER_MODE);
```

Channels and Completion Models

Your testbench environment needs to exchange transactions between its components. For example, transactions flow from the Generator -> Transactor -> Driver, or from the Monitor -> Checker -> Scoreboard. As you saw in the last section, you should model transactions as objects that are then created and modified by the different testbench components.



The connection between these components is the RVM channel. One side is the producer (such as the Generator) putting transactions into the channel. The consumer side (Transactor) gets the transactions out of the channel, and executes them.

The RVM channel has several advantages over the OpenVera mailbox:

- Unlike mailboxes, channels are strongly typed which helps prevent coding errors.
- Channels allow flow control, so the put_t() method will block if the channel is full.
- A channel can have both high-water and low water marks to fine tune the interactions between the producer and consumer. The get_t() method removes the transaction from the end of the channel, while peek_t() give you a handle to it without removal. Both block if there are no transactions in the channel.
- The output of a channel can be replicated using the tee_t() method.

Definition and Creation

You define a channel for a specific type using a macro, and then create channels as shown:

```
class atm_cell extends rvm_data {
   ...
}
rvm_channel(atm_cell)
// atm_cell_channel data member declaration
     // rvm_channel(data_type_name)
    // macro automatically creates new data type by
// appending "_channel" to data_type_name

program test {
   atm_cell_channel chan;
   chan = new("ATM Cell channel", "class");
 }
```

```
Now two threads can create communicate as follows:
// Producer
while (1) {
  atm_cell cell = new();
  ch.put_t(cell);
}
and:
// Consumer
while (1) {
  atm_cell cell = ch.get_t();
  ...
}
```

Under the Hood

Just like an OpenVera mailbox, the channel contains handles to objects, not the object themselves. So you can modify an object after it has been put in the channel, leading to a common mistake:

```
// Producer
atm_cell_channel ch = new("ATM Cell channel", "class");
atm_cell         cell;
cell = new();
while (...) {
   void = cell.randomize();
   ch.put_t(cell);
}
```

This code only allocates a single cell. It then repeatedly randomizes this cell and puts it in the channel. The result is that the channel will contain many references to the same object. The solution is to allocate a new cell every time through the loop:

```
while (...) {
   cell = new();
   void = cell.randomize();
   ch.put_t(cell);
}
```

Using Channels to Connect Blocks

The most common way to use channels is to allocate them in the rvm_env object and then pass them into the testbench components as shown in the following example:

```
task dut_env::build() {
   chan = new(...);
   consumer = new(chan);
   producer = new(chan);
}
```

Transaction Completion

You can synchronize two testbench blocks using a channel. The easiest way is to configure the channel with full=1 (the default) so it works like a procedural interface. The producer thread blocks when it calls put_t(). When the consumer calls get_t() to remove the transaction from the channel, the producer unblocks so it can create a new transaction. In the following example, the consumer first calls peek_t() to read the transaction, but does not call get_t() until it is done, thus waking the producer:

```
// Consumer
while (1) {
   cell = ch.peek_t();      // Read the cell
   case (cell.kind) {
      ...                   // Process the cell
   }
   void = ch.get_t();       // Done, wake up producer
}
```

Further Exploration

See the RVM User Guide section on “Completion and Response Models” for more examples of synchronizing using channels. What if you have more than one producer or consumer? The rvm_broadcast class is used for one-to-many communication, while rvm_scheduler takes N inputs and combines them into a single output.

Transactors

By this point you have seen RVM transactors. In this section, you are going to learn more about how to create and use them in simulation.

At its simplest, an RVM transactor is just a while loop that reads in transactions from a previous testbench layer, does some processing, and sends out transactions to the next layer. The key is properly starting and stopping the transactors.

The RVM has several transactor types:

- Active Xactor – Master
  - Drives pins, blocks on channel get_t()
- Reactive Xactor – Slave
  - Monitors and drives pins, blocks on signal edges
- Passive Xactor – Monitor
  - Monitors pins, blocks on signal edges
- Also – Generator or other Xactors as needed

- Creates transactions, blocks using notification or channel put_t()

A Basic Transactor

Here is a basic transactor. You add code to method main_t() to process transactions. The other methods all start with a call to the base method to start and stop this method. For example, rvm_xactor::start_xactor() starts the virtual method main_t() – you don’t need to do this.

```
class driver extends rvm_xactor {
   //start_xactor starts the execution threads
   virtual task start_xactor{
      super.start_xactor();
      ...
   }
   //stops execution threads after currently executing
   //transaction had completed. Takes effect at next call
   //to ::wait_if_stopped_t()
   virtual task stop_xactor() {
      super.stop_xactor();
      ...
   }
   //resets the xactor’s state and execution threads
   virtual task reset_xactor(...) {
      super.reset_xactor(...);
      ...
   }
   virtual task main_t(){
      ...
   }
}
```

Stopping and Starting

The main_t() method periodically checks to see if the transactor has been stopped by calling wait_if_stopped_t() as shown below:

```
virtual task main_t() {
   while (1) {
      this.wait_if_stopped_t();
      atm_cell cell = to_driver.get_t();
      this.wait_if_stopped_t();
      ...
   }
}
```

The wait_if_stopped_t() method will block if stop_xactor() has been called. Different blocks in your testbench will define when to stop and what it means. You should check if the transactor needs to stop after every time-consuming action, such as the call to get_t() above.

Physical Interface

The OpenVera virtual port construct will group all relevant physical signals (ports) into a single virtual port, just as a C typedef combines several objects into a struct. You can then pass this virtual port into drivers and monitors. Now a single driver can be replicated to drive multiple physical ports.

A single virtual port can have signals from multiple clock domains, or even asynchronous signals. As a result, do not include the sampling clocks. If you need to synchronize on a clock edge, use:

```
@1 this.sigs.$rxd == void;  // Void expect
...
@1 this.sigs.$txd = void;   // Void drive
```

rather than:

```
@ (posedge this.sigs.$tx_clk);  // Wait for clock edge
...
@1 (posedge this.sigs.$rx_clk); // Or wait for the next
```

Note that void drives and expects do not have to code the active edge of the clock signal. If the designer wants to change the edge, you only have to change the interface definition, not every usage of the signal.

Reusable Transactors

Recall that one of the goals of RVM is that the testbench objects should not have to change. Test specific code goes in the test, not in the transactors such as the driver. The question then, is, how do you write a transactor that can meet all verification requirements such as injecting errors and delays, sampling data for functional coverage and connecting the scoreboard?

Before RVM, the testbench would have the “mother of all transactors” (MOT) that performed all these actions and more. However, any time one would dream up a new way to control the DUT, one would have to edit the MOT. All these edits make the MOT unstable and a bottleneck for the verification team.

Instead, a transactor should do the simplest things by default – no errors, no delays, but has hooks so that you can add test-specific extensions. You can accomplish this by using callback methods. These are extensible virtual methods that are empty by default.

```
task main_t() {
   ...
   while (1) {
      ...
      cb.callback();
      // Now drive out cell
      ...
```

```
    }
}
```

In the above code, the method cb.callback() is called just before that cell is driven into the design. By customizing your own method, you could insert delays, modify or even drop the cell, gather functional coverage information on the transmitted cell, and put this cell into the scoreboard.

Creating Callbacks

Once you have identified key points in the transactor flow to insert callbacks, you can define a callback façade class:

```
class atm_callbacks extends rvm_xactor_callbacks {
   virtual task pre_cell_tx_t(atm_driver xactor,
                              atm_cell   cell,
                              var bit    drop)
   {}
}
```

The driver now just calls pre_cell_tx_t() before driving the cell. Next, define your test-specific callback classes such as error_inject, scoreboard_insert, functional_cov that extend atm_callbacks. Each callback class is then registered at the test level. The callbacks are called in the order you registered them. So, be sure to inject errors first, then add the transaction to the scoreboard.

```
class stretch_ifg extends atm_callbacks {
   // This class stretches the inter-frame gap
   ...
}

program test {
   verif_env env = new();
   env.build();
   {
     stretch_ifg cb = new();
```

```
      env.driver[0].register_callback(cb, 1); // 1=prepend
      env.driver[3].register_callback(cb, 1);
   }
   env.run_t();
}
```

You can see that the code to invoke the callbacks is common across all transactors, so the RVM provides a macro:

```
class atm_driver extends rvm_xactor {
   ...
   while (1) {
      atm_cell cells = this.in_chan.get_t();
      bit drop = 0;

      rvm_OO_callback(atm_driver_callbacks,
                      pre_cell_tx_t(this, cell, drop));
      if (drop) continue;
      ...
   }
}
```

Lab 6 – RVM Transactors

RVM transactors can have several different formats

- Master
- Active (Slave)
- Passive (Monitor)

This lab will focus on creating some of these transactors for the atm_cell data type.

Recommended Time: 60 Minutes

References:

- RVM Testbench Methodology User Guide: Transactors
- RVM Testbench Methodology User Guide: Appendix A, Class Reference, rvm_xactor class

The prepared material for this lab is in the “lab6” subdirectory. Please change your working directory to this directory before proceeding.

```
%cd lab6
```

Step 1 – Master Transactor / Driver

The basic structure for a master driver is provided in the file “atm_driver.vr”. Edit the file and add the class definition and data members as follows:

- Add code to the class “atm_driver” to derive from rvm_xactor.
- Add an input atm_cell channel data member atm_cell_channel (in_chan)
- Add a local variable for the virtual port “atm_drv_port” variable (iport)

The constructor (new() task) is present but empty. Edit the file to add the following to the constructor:

- Call the super.new task
- Save the iport
- Allocate a channel if null is passed in
- Save the in_chan
- Drive the $valid and $data signals to 0 and 8'hZZ async

Note that async will not block or advance time, a requirement for a RVM new task. (This is also a SystemVerilog requirement. new() is a SystemVerilog function and cannot consume time.)

Add code to the process_cell task. The code can utilize the atm_cell::byte_pack task to pack the data fields into a byte stream. This byte stream/array can then be processed using a simple for loop.

Step 2– Update atm_env

The Master driver can now be added into the environment. During this step, the testbench environment will change significantly – HDL code will be added to the simulation.

- Include the atm_driver.vrh, atm_port.vri, atm_if.vri files
- Add a data member for the atm_driver (drv) instance
- Allocate the atm_cell_channel instance (new)
- Allocate the atm_driver instance (new)
- Start the driver in the start_t() task
- Stop the driver in the stop_t() task

Run the test, and examine the output.

```
% gmake lab6  (or gmake ntb – for VCS NTB)
```

Does the packet data displayed by the generator match the packet data displayed by the driver?

_____

Verify the output by examining the log and VCD produced.

```
% vcs –RPP &
```

Read the vcdplus.vpd file, and add the router/* signals to the wave window.

Was data driven onto the data lines? Yes / No

From the VCD file, how many packets were driven onto the data lines? _____

From the log file, how many packets were driven onto the data lines? _____

Does this match? Y / N

Why? _____

_____
_

Step 3 – Passive Transactor / Monitor

The basic structure for a passive transactor or monitor is provided in the file “atm_monitor.vr”. Edit the file and add the class definition and data members as follows:

- Add code to the class “atm_monitor” to derive from rvm_xactor.
- Add an atm_cell instance (allocated_tr)

Review the constructor task. It is similar to the driver new task. Note the task:

- Calls the super.new task with a fixed text-string name, instance and stream_id
- Saves the iport
- Configures the channel to have 64k entries

- Sets the cell count to 0

In the main_t() task most of the code is standard and will be in almost all monitors.

For the ATM monitor, add code to:

- allocate a new cell
- call the monitor_t() task

In the monitor_t() task, the code that monitors and extracts the cell data is present. Add code to:

- Unpack the array of data into the cell instance
- Set the cell.stream_id to this.stream_id
- Set the cell.scenario_id to 0
- Set the cell.object_id to this.object_id++

Step 4 – Update atm_env

The ATM Monitor transactors can now be added into the environment. A monitor will be added to the transmit port, and each of the 4 receive ports in the design. The transactors will log all legal packets detected on the ports.

- Add the #include line to insert the atm_monitor.vrh file
- Add four data members for the atm_monitor transactor instance (mon[4])
- Add four channel data members for the mon_to_sb channels (mon_to_sb[4])
- Allocate the four atm_cell_channel instances (four calls to new)

- Allocate the four atm_monitor transactor instance (four calls to new)
- Start the monitor transactor in the start_t() task (for loop 1..cfg.num_output_ports; call to start_xactor)
- Stop the monitor transactor in the stop_t() task (for loop 1..cfg.num_output_ports; call to stop_xactor)

Run the test, and verify the output by examining the log. This time, the data should be printed from the Monitors.

```
% gmake lab6  (or gmake ntb – for VCS NTB)
% gmake lab6 > log  (or gmake ntb > log  – for VCS NTB)
% grep Driver log
% grep Mon log
```

For the transmit port:

Do the master transactor packets match the monitor packets? Y / N

How many packets were transmitted?

_____

For each receive port:

Do the active transactor packets match the monitor packets? Y / N

How many packets were transmitted on:

Port0? _____

Port1? _____

Port2? _____

Port3? _____

Is the distribution of traffic between the ports as expected? Y / N

Does it look like there is a bug in the RTL? Y / N

If so, which functionality is incorrect?

_____

_____
___

_____
____

We will find this (and possibly more) RTL issues when the scoreboard is present.

Hints – Master Transactor

Check in_channel, if this is null, allocate a channel using new(“ATM Driver Input Channel”, instance);

Use ‘async’ drives in the new task, these do not block.

The atm_cell.byte_pack(stream) function returns the number of bits. The stream is an array of bit [7:0], as defined in process_cell() task.

Hints – Passive Transactor

To configure the channel depth, use “channel.configure(integer depth)”

The data can be unpacked using the atm_cell::byte_unpack() function to unpack the data into a byte-array.

Each field in the cell should be set individually, using the monitor data values for stream_id, and cell_count. Scenario ID should be set to 0 as it is unknown.

The Verbose printing loop is similar to the Debug loop, without the heading text. The severity to check for is VERBOSE_SEV, and data items 8 to 52 should be printed.

Hints – atm_env

In the atm_env class, add the data members. An array of four monitor and channels are needed.

In the build task, call the new task for the driver. For the channels and monitors, call the new task once for each of the four instances. Simply copy the code four times to eliminate string manipulation.

In the start_t task, start the transactors by calling the start_xactor task. Call this once for the driver, and use a for loop to start the four monitors.

In the stop_t task, copy the start_t code, and change the call to stop_xactor.

Atomic Generators

Tests should tune random generators, not completely rewrite them. This results in less code (each test is smaller), more randomness (all unspecified behavior is random) and more checking (extra randomness broadens the stimulus). Traditional testbenches create stimulus with generators that grow more and more complex as the project progresses, accommodating every variant of stimulus, error generation, synchronization, etc. This “mother of all generators” can be unstable because of the constant changes, as well as difficult to enhance and maintain. RVM recommends a different approach.

Adding Constraints

A typical generator might look like the following:

```
class cell_gen extends rvm_xactor{
   ...
   task main_t() {
      while (1) {
         atm_cell cell = new();
         void = cell.randomize();
         this.chan.put_t(tr);
      }
   }
}
```

The problem with this generator is that there is no easy way to randomize cells with different constraints.

CODING TIP:
While the above example ignores the result from cell.randomize(), you should never do this in real code. Always check the result and issue an rvm_error. Otherwise, a constraint failure may be missed, leading you to think that the test generated cases that it really did not.

- You could use the randomize() with{} construct, but this would require a separate generator for each test, just what you were trying to avoid.
- You could modify the transaction class (atm_cell), adding constraints for each test, but this moves the problem to a different file.

But, each of these requires every test writer to edit a common file, with the results applied to every generator / ATM cell. Some testbenches have knobs to control the different distributions and cases, but once again, the generator or transaction becomes the bottleneck, growing in complexity. In addition, the testcase is the testbench plus knob files, adding another file to the flow.

Factories

As an alternative, consider an automobile factory. It creates a stream of cars, each unique with varying options. The manufacturer accomplishes this by having a set of blueprints for the major variants (coupe, sedan, station wagon), and then allowing the user to tweak the details. You can use the same idea with creating transactions. Create a “factory” that stamps out transactions, then have individual tests feed it different blueprints. All the test-specific code is located in the test file, not the generators. In fact, you can have multiple factories running in a test, each generating a unique set of stimulus.

What does this look like? The following example uses a transaction class that extends rvm_data. The blueprint instance is randomized, and then copied to a new instance processed by the next testbench layer.

```
class factory {
   transaction blueprint = new();
```

```
   task run() {
      while (run) {
         transaction tr;
         void = blueprint.randomize()
         cast_assign(tr, blueprint.copy());
         process(tr);
      }
   }
}
```

You can change the blueprint from the test level, which is just an OpenVera program:

```
class my_transaction extends transaction {
   constraint address_even {
      addr[0] == 0;
   }
}
program test {
   verif_env env = new();

   env.build();
   {
      my_transaction my_tr = new();
      env.src[0].blueprint = my_tr;
   }

   env.run_t();
}
```

With this change, the generator src[0] will always create transactions with even addresses.

Benefits

Tests can modify constraints by

- Making variables non-random
- Turning constraint blocks off
- Add new constraint block

- Re-define constraint blocks
- Add random variables
- Supersede random results with directed data

Factory generators are:

- Good: Unmodified generic, reusable data class
- Good: Unmodified generic, reusable generator class
- Good: Different generators can have different constraints
- Good: Constraint set can be dynamically changed
- Good: Localized test-specific code
- Bad: Difficult to share constraint sets between testcases

Atomic Generator Macro

Vera and VCS NTB include a macro to create an atomic generator:

```
    //Macro: defines atm_cell_channel
rvm_channel(atm_cell)
//Macro: defines atm_atomic_gen
rvm_atomic_gen(atm_cell, “ATM_Cell”)

class atm_env extends rvm_env {
  atm_cell_atomic_gen gen1; // Generator instance
  atm_cell_channel chan; // Channel instance

  virtual task build() {
    chan = new(“Channel”, "from gen");
    gen1 = new(“Gen”, *, chan);      // Connect channel
    drv = new (“Driver”, “*”, chan); // to next level
  }
 }
```

The macro calls the blueprint object randomized_obj.

Lab 4 – Atomic Generation

An RVM generator is a class that creates random rvm_data objects. This lab will focus on creating an RVM compliant ATM generator

Recommended Time: 30 Minutes

References:

- Reference Verification Methodology User Guide: Stimulus and Generation
- Reference Verification Methodology Users Guide: Class Reference, rvm_xactor class

The prepared material for this lab is in the “lab4” subdirectory. Please change your working directory to this directory before proceeding.

```
% cd lab4
```

Step 1 – Data Members

The generator object may be instantiated in the testbench several times, once for each channel is common. For this reason, the generator class must store all settings and configurations on a per-channel basis. This will allow the test to configure each channel independently.

Edit the atm_generator.vr file as follows:

- Add the class “atm_cell_gen” (derives from rvm_xactor).
- Add the following data members to the class:

```
atm_cell          randomized_obj;
atm_cell_channel  out_chan;
local integer     scenario_id;
local integer     object_id;
integer           stop_after_n_insts;
```

Step 2 – Constructor

Add code to the constructor (new task) to…

- initialize scenario_id to 0
- initialize object_id to 0
- initialize stop_after_n_insts to 0

Step 3 – Functions

Add code into the main_t() task that…

- Sets the stream and scenario ID's, and increments the object ID

  randomized_obj.stream_id = this.stream_id;

  randomized_obj.scenario_id = this.scenario_id;

  randomized_obj.object_id = this.object_id++;

- Copy the randomized_obj into the local variable
- Put the local cell in the output channel

Step 4 – Update atm_env

The atm_atomic_gen and atm_cell classes can now be added to the testbench environment. Edit the atm_env.vr file and making the following changes:

- Add #include statements to include the atm_cell.vrh file
- Add #include statements to include the atm_generator.vrh file
- Create an instance of the atm_cell_atomic_gen (atm_gen)
- Create an instance of the atm_cell_channel (gen_to_drv)
- Initialize (call new) for the atm_gen data member and gen_to_drv channel
- Stop the generator in the stop_t() task by calling stop_xactor()

Note that the generator will be started in the test, since we want to enable either the atomic generator or the scenario generator during the lab exercise. Usually, the generator is started in the start_t task by calling gen.start_xactor().

Step 5 – Review the Test and Run

Review the test04.vr file, and note the following:

- A derived ATM Cell class is defined that contains an additional constraint on the data fields.
- This makes the test easier to debug, as cells with known content are applied to the design.
- In the test, a new scope is created with the { … } construct
- Inside the new scope, an instance of the constrained ATM cell is defined.
- The blueprint inside the atomic generator is replaced with the new constrained ATM instance
- The generator will now copy this constrained ATM cell instead of the default ATM cell

Build and run the default test using the following command.

```
% gmake lab4 (or gmake ntb – for VCS NTB)
```

Check the result by viewing the printed results.

_____

- Did you generate the correct number of packets?
  _____
- Are the packets headers random?
  _____
- Are the packets data random?
  _____
- Is this expected?
  _____

Step 5 – RVM Atomic Generator Macro (Advanced)

Create a second atomic generator using the rvm_atomic_gen() macro. Add this macro call to the end of the atm_cell.vr file, and create an instance of the atm_cell_atomic_gen generator. In the test04.vr file, start the generator created by the macro (atm_cell_atomic_gen) intead of the manually coded generator (atm_cell_gen). Run the test and observe the output.

Hints

For Loop:

```
while (this.cell_count < this.stop_after_n_cells ||
       this.stop_after_n_cells == 0 ) { . . .}
```

ID Setting:

Set stream_id, scenario_id and object_id fields in this.randomized_cell object. Add 3 calls, one for each data member.

```
this.randomized_obj.scenario_id = this.scenario_id;
```

Output Channel:

Place the output cell into the channel using the “put_t” task.

Callbacks (Advanced):

The callback class extends from rvm_xactor_callbacks and has the following task:

```
virtual task post_cell_gen_t(atm_atomic_gen gen,
                             atm_cell       cell,
                             var bit        drop);
```

The callback routine in the main_t() will be similar to …

```
cast_assign(cell, this.randomized_cell.copy());
drop = 0;
foreach (super.callbacks, i) {
   atm_atomic_gen_callbacks cb;
   if (!cast_assign(cb, super.callbacks[i], CHECK))
      continue;
   cb.post_cell_gen_t(this, cell, drop);
}
if (drop) continue;
```

OOP & Virtual Methods

With traditional procedural programming, you create structs to hold the data, and global procedures that manipulate the data. With Object Oriented Programming, you wrap the data and the procedures inside of a class:

```
class packet {
   reg [31:0] src, dst, crc;
   reg [31:0] data[8];

   task display() {
      ...
   }

   function reg [31:0] compute_crc() {
      ...
   }
}

program test {
   packet p;
   p = new();
   p.display();
   p.crc = p.compute_crc();
}
```

Inheritance

Now that you have a packet, how do you create different flavors? For example, you might want to add an is_good bit that tells if the class should always generate good CRC. In OOP, this is done by extending the original packet class (base class):

```
class my_packet extends packet {
   reg is_good;
   function compute_crc() {
```

```
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = urandom;
   }
}
```

Handles to Objects

When you make the following declaration:

```
packet p;
```

you are creating a handle that can reference objects of type packet. The handle is initialized to null. You can call the new() method to create an object:

```
p = new();
```

This allocates enough memory to hold the class packet, 11 longwords (sa, da, crc, and data[8]). If you allocate a my_packet object, 12 longwords will be allocated (the original 11 plus one more for the is_good bit).

What happens if you try to mix handles for base and extended classes? The easiest way to visualize this is to consider the is_good bit. It would be an error if you used a my_packet handle to access the is_good bit, but the handle points to only a packet object (11 longwords).

```
packet p;
my_packet mp;

mp = new();
p = mp;     // Good: Copies a handle

p = new();
mp = p;     // Error: mp.is_good won't exist
```

But what if you use a base handle to point to an extended object, then try to assign back to an extended handle? This is normally not allowed, so you will need to use cast_assign. This method checks the object type to make sure it matches the destination object.

```
packet p;
my_packet mp, m2;

     mp = new();
     p = mp;        // Still good
    cast_assign(m2, p); // Good: p points to my_packet object
```

Polymorphism

What happens if you call a method in a class. By default, if you use a packet handle to call the compute_crc() method, the packet::compute_crc() method is called, even if the object was actually of type my_packet. (See code in the Inheritance section above.) But if you use virtual methods, Vera will call the method based on the object type, not the handle type:

```
class packet {
   virtual function reg [31:0] compute_crc() {
      ...
   }
}
class my_packet extends packet {
   reg is_good;
   virtual function compute_crc() {
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = urandom;
   }
}

packet p;
my_packet mp;
```

```
mp = new();
p = mp;
p.crc = p.compute_crc();  // Calls my_packet method
```

Why is this useful? You can write a generic class such as packet and use it in a testbench:

```
class protocol {
   ...
   task transmit(packet pkt) {
      ...
      pkt.crc = pkt.compute_crc();
      off = vera_pack(bytes, off, pkt);
      // Transmit packet data
   }
}
```

If you call the transmit() method with a packet object, it will call packet::compute_crc(). Call it with a my_packet object and it will call my_packet::compute_crc() and inject errors.

Answers to Lab Questions:

Lab 1:

Step1:

Do all the messages appear? No

If so, why, if not, why not?*Messages are displayed depending on the severity level of each message and the severity level of the log object. Log object severity can be controlled at runtime*.

Step2:

Cmd Line Severity	Log1				
	Error	Warn	Note	Dbg	Verb
None Specified	P	P	P		
Note	P	P	P		
Debug	P	P	P	P	
Verbose	P	P	P	P	P

Step3:

Cmd Line Severity	Log1					Log2				
	Error	Warn	Note	Dbg	Verb	Error	Warn	Note	Dbg	Verb
None Specified	P	P				P	P	P		
Normal	P	P				P	P	P		
Debug	P	P				P	P	P	P	
Verbose	P	P				P	P	P	P	P

Cmd Line Severity	Log1					Log2				
	Error	**Warn**	**Note**	**Dbg**	**Verb**	**Error**	**Warn**	**Note**	**Dbg**	**Verb**
None Specified	P	P				P	P	P	P	
Normal	P	P				P	P	P	P	
Debug	P	P				P	P	P	P	
Verbose	P	P				P	P	P	P	

Does the command line override the severity in the code? No

Is this what you expected?*Perhaps? The command line is a default, and is overridden by any settings in the code*.

Lab 2:

Step 1:

new()	Virtual? No
gen_cfg()	Virtual? Yes
build()	Virtual? Yes
cfg_dut_t()	Virtual? Yes
start_t()	Virtual? Yes
stop_t()	Virtual? Yes
cleanup_t()	Virtual? Yes

report() Virtual? Yes

Missing Functions: *wait_for_end_t()*

Step 3:

Logging Review: What does the “rvm_env” text refer to? *A logger name, usually the name of the class.*

Logging Review: What does the “(class)” text refer to? *A logger instance, usually the instance name.*

Step 4:

If one (or more) of the super.method() calls in the atm_env.vr file are removed, what do you expect to happen?

An error will be produced by the RVM base class code. Try it out.

Did the expected behavior happen? Yes

If not – why? *If an error was not produced, ask the instructor to help*

Lab 3:

Step 1:

Why is the log data member static?*Each data object does not need a logger, one for all data*

objects is sufficient. This is mainly for speed and memory reasons.

Is the log data member static in all classes? No

Why? *In the permanent testbench objects (drivers, generators, monitors etc), a class and instance name is required for each logger. This is instance specific, so the logger cannot be static.*

Step 2:

Review the display() task. Why is printf() used instead of a log instance? *When printing a data object,*

Another object (driver, monitor, generator) calls the display routine. This object will print the messageType, time etc. If a logger was used in the data class, a second heading would be printed with the static Logger name ‘atm_cells(class)’ or similar. Using printf eliminates this second heading line.

Step 3:

How many “Pass (OK)” Messages were printed? 2

How many “Fail (OK)” Messages were printed? *1*

Was this as expected?*Yes.*

Why? *A fail was expected, the test compares two unequal packets to check the compare routine.*

The message should say “fail (expected)” to indicate the failure is expected.

Step 5:

Does the code behave as expected? Yes

If so, why, if not, what problems do you see? Please ask the instructor if you see problems

Lab 4:

Step 5:

Is the correct number of packets generated? Yes

Are the packets headers random? No

Are the packets data random? Yes

Is this expected? Yes – the header is constrained in the test04.vr file.

Lab 5:

Step 3:

Does the default output look like the atomic generator? *Yes*

Are the scenarios selected using a round-robin scheme? *Yes*

How could the default scenario be removed? *Replace scenario_set[0]*

What happens if the repeated data member is not constrained? *Repeated is randomized, and will probably cause many copies of the scenario to be copied to the output. (depending on the randomized value of repeated – it is an unconstrained integer, so chances that it is large are good)*.

Is there any built-in protection to warn you if this occurs?

Yes - The repeat_thresh warning will probably occur (depending on the randomized value of repeated.

Is the scenario length now random in the range 1…5? *Yes*

Was the incrementing scenario constrained atm cells? *Yes. allocate_scenario and atm_cell_cst::copy() are present*.

Was the decrementing scenario constrained atm cells? *No. We didn't call allocate_scenario.*

Lab 6:

Step 4:

Was data driven onto the data lines? Yes

From the VCD file, how many packets were driven onto the data lines? *25*

From the log file, how many packets were driven onto the data lines? *25*

Does this match? Yes

Why? *Each packet is logged as it is driven by the atm driver*

For the transmit port:

Do the master transactor packets match the monitor packets? Yes

How many packets were transmitted? *25*

Step 6:

For each receive port:

Do the active transactor packets match the monitor packets? Yes

How many packets were transmitted on:

Port0? *Depends on random seed*

Port1? *0 – constraint to use port 0 only*

Port2? *0 – constraint to use port 0 only*

Port3? *0 – constraint to use port 0 only*

Is the distribution of traffic between the ports as expected? Yes

Does it look like there is a bug in the RTL? No

If so, which functionality is incorrect? *No bug expected yet.*

Lab 7:

Step 4:

Did the output change from lab6? Yes

If so, how? *Generator print statements are present in lab6, otherwise no change.*

Lab 8:

Step 1:

Which function is used to receive ATM cells from the driver callback? *atm_sb::from_driver()*

How many threads are present in the scoreboard? *4*

How are the atm_cell objects freed? *Vera has automatic garbage collection when the last reference to an object goes out of scope, the object is freed*

Step 3:

How many packets were transmitted? *25*

How many packets were received? *25*

Did the Scoreboard report any messages? No

If so, is there an RTL bug? *It does not look like there is a bug yet.*

Step 5:

How much coverage was achieved? *55% (this may vary depending on the random seeds)*

Which items require additional coverage? *Many cell fields*

Edit the test07.vr file and increase the length of the test to 100 cells.

Did this help? *Yes*

Why? *More packets increases coverage. The design is in the broad spectrum testing phase*

Examine the VPI field coverage. Is this 100% ? No

If not, why not? There is a constraint on this item in atm_cfg

Create constraints to increase the coverage of the design. Did you find any additional bugs in the design?

If so, please document them here: *Yes. There is a bug in the RTL, output channel 3 and 4 both go to channel 4. Fixing the RTL by correcting the case statement where the “bug here” comment is should fix the problem.*

4

Introducing Verification Methodology Manual

This tutorial is a beginner’s guide to using the Verification Methodology Manual (VMM), with the SystemVerilog language. You can simulate your testbenches with VCS. With VMM, you can quickly build a layered testbench. These testbenches support high-level tests using constrained random stimulus and functional coverage to indicate which areas of the design you have checked.

In this tutorial, you will verify an ATM switch, starting with simple stimuli and then working up to a complex testbench.

Requirements

Knowledge

- You should know the Verilog hardware design language.
- You should have basic knowledge of the SystemVerilog language, especially concepts such as Object Oriented Programming (OOP) and synchronization between the testbench and Design Under Test (DUT). If you have not used virtual methods in OOP, you should first read Appendix A. The term “method” refers to tasks and functions.
- You should be familiar with VCS to compile and simulate designs plus testbenches.

Tools

This tutorial requires VCS X-2005.06 or higher.

References

You should refer to documentation and examples contained in the releases of the tools used. The Verification Methodology Manual (VMM) is a useful reference. You can buy it from Springer Press or use the online version at `$VCS_HOME/doc/UserGuide/pdf/vmm_sv.pdf`.

Just to clarify, the methodology is known as the VMM, while the book is the VMM. Synopsys uses the “vmm_” prefix for the classes and macros in SystemVerilog.

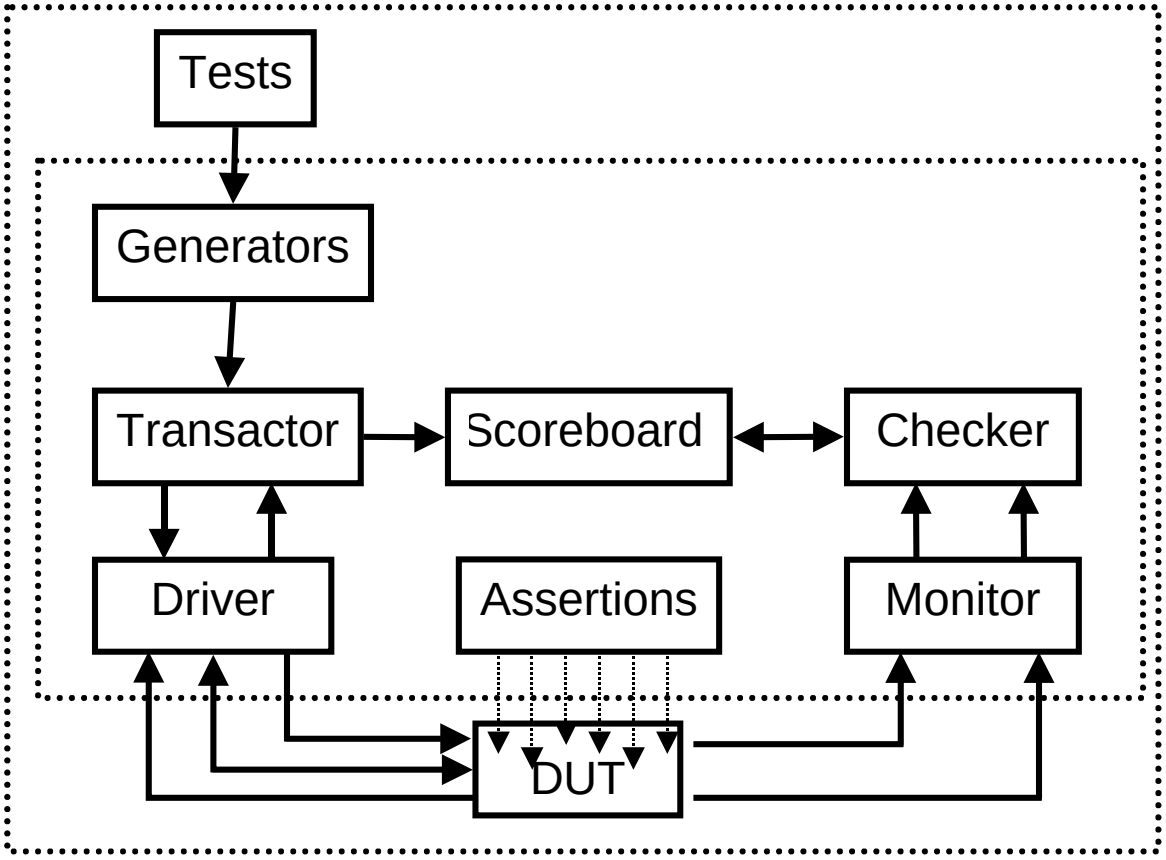
Overview

To get the most out of a Hardware Verification Language (HVL) such as SystemVerilog, you need to adopt a new methodology. If you use the same techniques from your old Verilog testbenches (directed tests with little randomization), you will not find bugs in your design as quickly as if you tried a new approach. Switching will also help make your code easier to maintain and reuse.

The VMM consists of coding guidelines and a set of base classes that allow you to develop reusable testbench components such as data models, transactors, and generators. Synopsys designed VMM based on years of experience verifying many different types of designs, enabling use of advanced techniques. Using VMM will give your project a consistent look and feel, so less time is spent creating verification infrastructure, and more time verifying your design.

The VMM involves creating a robust, flexible testbench infrastructure once, and then creating many simple tests, while never changing the underlying testbench. This requires advanced techniques such as factory patterns and callbacks. For example, virtual methods allow you to write the testbench once, and then not have to modify it for every possible type of stimulus variant such as error injection, synchronization, and variable delays. While you can verify a design using a simple testbench, you would have to create many elaborate tests and continually update the testbench. This latter approach yields more code and reduces the readability and maintainability of your code.

Layered Environment



The three major parts of a verification environment are the DUT, the testbench (the inner box above), and the test which controls the testbench. Inside the testbench are the following parts:

- The Driver controls the signals into the DUT. You can write it in SystemVerilog or Verilog. It executes single commands such as a bus read or write, or driving a cell / packet / frame into the DUT
- The Monitor bundles signal changes from the DUT into transactions.

- Assertions constantly check the DUT for correctness. These look at external and internal signals. The testbench uses the results of these assertions to see if the DUT responded correctly.
- The Transactor takes high-level transactions such as a burst-read into individual read commands, or a single USB transaction into multiple USB packet TX/RX commands.
- The Scoreboard stores the transactions from the Transactor for later comparison.
- The Checker compares the output of the DUT, as seen by the Monitor, to the Scoreboard, using a predictor or reference model.
- The Generator generates transactions, either individually or in streams.

Note that each of these components may be instantiated several times or come in several flavors for different protocols.

A goal of VMM is that the testbench will not need to change for individual tests. Its components include hooks to allow the testbench to control the stimulus, without having to anticipate all possible conditions such as error injection.

Labs

The files for this tutorial are in the following directory:

$VCS_HOME/doc/UserGuide/examples-pdf/vmm_tutorial_sv/

Each lab exists in a separate directory, allowing you to complete the labs in any order. Each lab has some existing code and comments indicating where you need to complete code, as shown below:

```
// Lab1 - ... comments
```

Solutions to all labs are in the solutions directory. You should examine the tips and hints sections that follow each lab before consulting the solutions.

During the labs, you will construct a verification environment to verify a very simple APB system. The testbench issues read/write commands, and the RTL is a simple memory. The APB protocol uses a simple address, write-data, read-data, Read/Write, Select, Enable and clock interface.

Lab instructions are provided in Appendix B of this document.

Messaging

A testbench produces messages of many types and severities. The vmm_log class lets you control which messages are displayed, what their format is, and even promote and demote them (useful for error testing). All messages are sent to standard output, i.e. displayed on the screen and sent to the simulation log file, just like $display.

Type and Severity

In VMM, every message has a type and a severity. You may want to print a message to debug a piece of code, tell the user that simulation reached a notable state, or encountered a problem. The message type tells which of these is happening.

- Failure: Error has been detected
- Note: Simulation progress
- Debug: Optional simulation diagnostics

- Timing: Timing check or error

The severity field describes the message importance. The following list shows the severity levels and the type in parenthesis.

- Fatal: Functional correctness definitely compromised (Failure)
  - Example: Testbench failure
- Error: Functional correctness may be compromised (Failure)
  - Example: Actual model results don’t match expected results
- Warning: Functional correctness not compromised (Failure or Timing)
- Normal: Regular, expected message
- Trace: High-level simulation execution trace message (Debug)
  - Example: "Executing transaction"
- Debug: Detailed simulation execution trace message (Debug)
  - Example: "Waiting for acknowledge"
- Verbose: Very detailed simulation execution trace message (Debug)
  - Example: "Sending byte #5 (0x5A)"

---

vmm_log class

Each part of the testbench (test, generator, checker, etc.) uses its own instance of the vmm_log class to generate messages. Each instance is a separate message source with a descriptive name and an instance name. You can use regular expressions to select and

control sources so use clear names. Usually the descriptive name is the name of the class instantiating vmm_log, and the instance name is the name of the object, or “class” if there is only a single instance.

Declaration and Instantiation

The vmm_log is usually instantiated inside a testbench object such as a generator or checker, or in a data object:

```
vmm_log log;
log = new(“name”, “instance”);
```

The name string is the name of the class that contains the log, such as “USB Host”, or “MAC Frame”. The instance string is the name of this instance of the object such as “Generator 1”, or “Left side”. If there is only a single instance, just use the string “class”.

The easiest way to use a vmm_log object is with the macros:

```
`vmm_fatal(vmm_log log, string msg);
`vmm_error(vmm_log log, string msg);
`vmm_warning(vmm_log log, string msg);
`vmm_note(vmm_log log, string msg);
`vmm_trace(vmm_log log, string msg);
`vmm_debug(vmm_log log, string msg);
`vmm_verbose(vmm_log log, string msg);
```

Here are two examples of using the above messages. The first displays a simple string. The second needs to print variable arguments, so it uses $psprintf, which returns a formatted string:

```
`vmm_verbose(log, "Checking rcvd byte");
if (byte != expect) begin
   `vmm_error(log, $psprintf("Bad data: 0x%h vs. 0x%h",
                             byte, expect));
end
```

Note that these macros expand to several lines, so surround them with begin-end when used in an if-statement.

```
Coding Guideline:
Avoid declaring and instantiating an object all on one line.
You will not be able to call any procedural code before the
first call to new(). Instead of:
        vmm_log log = new("name", "instance"); // Poor code
Use:
        vmm_log log;               // Separate declaration
        log = new("name", "instance"); // from
instantiation
Note that this tutorial occasionally skips this rule to
make the examples more readable.
```

Message Handling

The messaging class handles each message according to its severity level. The default is that fatal messages cause the simulation to exit, error messages increment a global error count, and cause the simulation to exit after 10 errors, while all others just print to standard out. You can use the method vmm_log::modify() to change how messages are handled.

Controlling Verbosity

By default, only messages with a severity of NORMAL (vmm_fatal, vmm_error, vmm_warning, vmm_note) or higher are displayed. You can control this two ways:

Using +vmm_log_default

The command line switch +vmm_log_default=DEBUG will enable printing of all messages with the severity level DEBUG and higher. Other settings are WARNING, NORMAL, TRACE, or VERBOSE.

Using set_verbosity()

The method vmm_log::set_verbosity() allows you to set the level of printing on the fly. The following code sets the level to DEBUG for any vmm_log object with “Drv” in its name:

```
log.set_verbosity(log.DEBUG_SEV, "/Drv/", "/./", );
```

The regular expression "/./" matches any string, so the following matches all vmm_log objects, regardless of their name or instance name:

```
log.set_verbosity(log.DEBUG_SEV, "/./", "/./", );
```

Note that since set_verbosity() and DEBUG_SEV are part of the vmm_log class, they must be prefixed with a handle to that class.

Also note that this call overrides the +vmm_log_default switch, and only applies to current vmm_log objects, not any created afterwards.

Further Exploration

- You can create complex, multi-line messages using the vmm_log methods start_msg(), text(), and end_msg().
- You can change the formatting of vmm_log by extending the vmm_log_format class and register an instance with the vmm_log::set_format method.

See the VMM for more information and examples.

Verification Environment

Your testbench goes through many phases of execution, from initialization, simulation, and cleanup. The class vmm_env helps you manage these steps and ensures that all execute in the proper order.

Coding Guideline:
The new() method should only initialize values, and should never have any side effects such as spawning threads or consuming time. If a testbench object starts running as soon as new() is called, you will not be able to delay its start, or synchronize it with other testbench operations.

The Nine Steps

The vmm_env class divides a simulation into the following nine steps, with corresponding methods:

- gen_cfg() – Randomize test configuration descriptor
- build() – Allocate and connect test environment components
- reset_dut() – Reset the DUT
- ·cfg_dut() – Download test configuration into the DUT
- start() – Start components
- wait_for_end() – End of test detection
- stop() – Stop data generators and wait for DUT to drain
- cleanup() – Check recorded statistics and sweep for lost data
- report() – Print final report

Simplest Example

Above all these methods is run() which keeps track of which steps have executed, and, when called, runs the remaining ones. For example, the following program runs all nine steps automatically:

```
program test;
 initial begin
    verif_env env;
    env = new(...);
    env.run();
  end
endprogram
```

The class verif_env extends vmm_env. You call run() and it will call all the steps which have not yet been run.

Basic Example

The next example runs the first step, makes a modification to the configuration, and then completes the test:

```
program test;
  initial begin
    verif_env env;
    env = new(...);

    env.gen_cfg();              // Create rand config
    env.rand_cfg.n_frames = 1; // Only run for 1 frame
    env.run();                 // Run the other steps
  end
endprogram
```

Automatic Sequencing

The run() task is not the only method that executes the steps. As shown in the following example, if you call build() without calling gen_cfg(), the build() method will automatically execute the previous step:

```
class my_eth_fr extends eth_frame;
  rand bit [47:0] mac_address;
  constraint one_port_only {
    da == mac_address;          // Use fixed address
  }
endclass

program test;
  initial begin
    verif_env env
    env = new();
    env.build();                  // Config and build
    begin
      my_eth_fr my_fr;
      my_fr = new();              // Use my own frame
      void = my_fr.randomize();
      env.src[0].rand_fr = my_fr; // Use to build more
    end
    env.run();
  end
endprogram
```

Using vmm_env

The following example defines the class verif_env. The virtual methods gen_cfg() and build() must call their super method as the first step. These calls to the base methods contain the sequencing

code that ensures all previous steps have been called. If you leave out the calls, the vmm_env class will generate a fatal error at run time.

```
`include "vmm.sv"

class verif_env extends vmm_env;
  my_cfg cfg;
  my_gen gen[4];
  my_drv drv[4];
  my_mon mon[4];

  virtual function void gen_cfg();
    super.gen_cfg();
    // rest of gen_cfg method
  endfunction

  virtual function void build();
    super.build();
    // rest of build method
  endfunction
endclass
```

Detailed Explanation of Methods

gen_cfg()

This method creates a random configuration of the test environment and DUT. It may choose the number of input and output ports in the design and their speed, or the number of drivers on a bus and their type (master or slave). You can also randomly select the number of transactions, percent errors, and other parameters. The goal is that over many random runs, you will test every possible configuration, instead of the limited number chosen by directed test writers.

build()

This method builds the testbench configuration that you generated in the previous method: generators and checkers, drivers and monitors, and anything else not in the DUT.

reset_dut()

This method resets the DUT to make it ready for configuration.

cfg_dut()

In this method you download the configuration information into the DUT. This might be done by loading registers using bus transactions, or backdoor loading them using C code.

start()

This method starts the test components. This is usually done by starting the transactor objects, which will be described in section 8.

hwait_for_end()

This method waits for the end of the test, usually done by waiting for a certain number of transactions or a maximum time limit.

The following example shows the wait_for_end() method with its inner wait for the end of test event, plus a time-out statement. These are wrapped in a fork-join_any that completes when either of these complete, and then a terminate command to stop the unfinished statement/threads.

```
class verif_env extends vmm_env {
   virtual task wait_for_end();
```

```
    super.wait_for_end();   // Call super method
    fork                     // Limit scope of terminate
    begin
       fork
          sync(ALL, this.sb.enough); // End of test event
          begin
             delay(TIME_OUT);
             `vmm_fatal(log, "Test did not complete");
          end
       join any
       terminate;
    end
    join
endtask
```

stop()

This method stops the data generators and waits for the transactions in the DUT

cleanup()

Check recorded statistics and sweep for lost data.

report()

Print the final report. Note that vmm_log will automatically print its report at the end of simulation so you do not have to write any special code for this.

Lab 1 vmm_env

Using the lab instructions (Appendix B), complete Lab 1 for vmm_env.

Data and Transactions

Traditionally, when you created a testbench in Verilog, you implemented transactions as procedures, one per transaction. This caused the following problems:

- Code is not self-contained
- Code is not protected properly
- Cannot extend data types
- Cannot add constraints to an existing data type

Instead, you should model transactions as objects. Their data values exist in a transaction class that can be randomized, copied, packed, and unpacked. The code that actually executes the transactions resides in the Driver.

Transaction Coding Guidelines

Properties in a transaction class should be public so they can be modified or constrained by other classes, such as the testbench. Do not hide data values using set() & get() methods. In hardware verification, you need access to all parts of the testbench for maximum control.

Properties should be random by default so that they can be randomized by other classes. You can always go back and use rand_mode() to turn this off.

Transactions vs. Transactors

The transaction class contains both physical values that are sent to the DUT (address, data, etc.) and meta-data that has extra information about the transaction, such as a “kind” field. Even though this might be encoded in the physical values, put it into a separate field that can be easily accessed and can be randomized.

```
enum kind_t = READ, WRITE;
class apb_transaction extends vmm_data;
  rand kind_t kind;
  rand bit [ 7:0] sel;
  rand bit [31:0] addr;
  rand bit [31:0] data;
endclass
```

The Transactor contains the code to send the transaction to the next testbench level. The following is a Driver to read and write to a real bus:

```
class apb_master;
  task do(apb_transaction tr);
    case (tr.kind)
      READ:
        tr.data = this.read(tr.addr);
      WRITE:
        this.write(tr.addr, tr.data);
    endcase
 endtask
endclass
```

Creating Your Own Transactions

The VMM recommends that you extend the VMM classes to create your own company-specific classes to build a buffer between the VMM and end users. In this buffer, you can customize the classes for your own best practices. This layer is most commonly added around vmm_data, though this tutorial uses the class directly.

ID Fields

Every transaction has three integer ID fields uniquely identifying it. The stream_id tells which stream created this object – useful when there are multiple generators. The sequence_id is used when a stream generator creates groups of related transactions, and identifies the group. The object_id identifies individual transactions in a sequence. In the following example, a constraint block uses the stream_id.

```
class vmm_data;
  integer stream_id;
  integer scenario_id;
  integer object_id;
  ...
endclass

class atm_cell extends vmm_data;
  rand integer has_vlan;
  …
endclass

class my_atm_cell extends atm_cell;
   ...
   constraint stream_0_is_vlan {
      if (stream_id == 0) has_vlan == 1;
   }
endclass
```

Constraints

Every transaction should have one or more constraint blocks for the “must-obey” constraints that are never turned off or overridden. For example, they would make sure an integer field is never negative or that a length field is never 0. Name these constraints “class_name_valid”, such as atm_cell_valid.

You should have separate constraints for “should-obey”. You can turn these off later for injecting errors. Name these constraints “class_name_rule”.

Methods

The vmm_data class defines a set of virtual methods for manipulating its properties. (See Appendix A for a review of virtual methods.) Here are some of the ones used in this tutorial. You will need to make your own methods when you extend vmm_data.

display() & psdisplay()

These methods display the contents of the class, either to the screen or to a string.

```
virtual task              display (string prefix);
virtual function string   psdisplay(string prefix);
```

allocate()

This method allocates a vmm_data object and initializes required fields. This is a virtual method, unlike new() so the correct method is called regardless of the handle type.

```
virtual function vmm_data allocate();
```

copy()

This method makes a copy of an existing transaction. It has an optional “to” field so you can copy to a previously allocated object. Note that this method returns a vmm_data type, so you may need to use $cast() with it.

```
virtual function vmm_data my_data::copy(vmm_data to = null);
   my_data cpy;

   // Copying to a new instance?
   if (to == null)
      cpy = new();
   else

   // Copy to an existing instance. Is it the correct type?
  // If destination handle is passed as argument during copy()
   // call, use $cast to check that handle type is correct.
  // Argument handle has to be base type (vmm_data) or my_data
   // else abort.  IF argument is base type(vmm_data), cpy
   // handle is upcasted to vmm_data type (cpy -> "to").

   if (!$cast(cpy, to, CHECK)) begin
      `vmm_fatal(this.log,
          "Attempting to copy to a non my_data instance");
      return;
   end

   // Copy ID’s and any other properties
```

```
    cpy.stream_id   = this.stream_id;
    cpy.scenario_id = this.scenario_id;
    cpy.object_id   = this.object_id;


    // Assign the copy to the return handle
    copy = cpy;
endfunction
```

There is an alternative to the copy() method. The new keyword does a shallow copy – more of a “photocopy” of variable values. Encapsulated objects are not copied; the handle value is just copied. Note that the new() method is not called, so the result will have the same ID fields as the original.

The following example shows how to use the copy method:

```
my_data d1, d2, d3;  // Handles
d1 = new();          // Allocate object
d2 = d1;             // Both d1 & d2 point to same object
d2 = d1.copy();      // Error, copy returns vmm_data
$cast(d2, d1.copy()); // Good
void = d1.copy(d2);  // Copy d1 contents to d2 object
void = d1.copy(d3);  // Bug – d3 is still null
```

compare()

This method compares two objects and reports the difference.

virtual function bit compare (to, diff, kind);

The current object is compared with “to” using type “kind”. (See 4.3 for an example of “kind”.) The method returns 1 if the two objects are the same, 0 if not. The diff string gives a description of the difference.

Packing and unpacking

The following three methods are used for converting between the physical fields of an object and an array of bytes:

```
virtual function integer  byte_size  (kind);
virtual function integer  byte_pack  (bytes,offset,kind);
virtual function integer  byte_unpack(bytes,offset,kind)
;
```

The method byte_size tells how many bytes you need to pack an object of this kind. The method byte_pack packs the object of type kind into a dynamic array of bytes. The method byte_unpack unpacks the data from the dynamic array of bytes. The offset tells the methods where to start in the byte array.

Lab 2 – the vmm_data class

Using the VMM Basic lab instructions, complete Lab 2 for vmm_data.

Notification

VMM provides an event notification class that allows you to notify when an event notification is indicated, and includes data. These notifiers are based on integer identifiers that hold a symbolic value. For example, the following code creates three notification identifiers associated with the atm_driver class:

```
class atm_driver extends vmm_xactor;
   static integer TR_STARTED;
   static integer TR_ABORTED;
   static integer TR_SUCCESS;
endclass
```

You must configure a notifier before using it. A notifier can be ON_OFF, ONE_SHOT, or BLAST. The following example calls the configure() method in the notify object (pre-instantiated in vmm_xactor, described in a later section), which returns an integer value.

```
class apb_master extends vmm_xactor;
  function new(...);
    this.TR_STARTED = this.notify.configure(*,
 this.notify.ON_OFF_TRIGGER);
  endtask
endclass
```

When you indicate an event, you can optionally attach an object derived from vmm_data. You use this to describe why you indicated the notification. You should not modify this object. If you need to change the object, use callbacks instead, shown later in this tutorial.

```
forever begin
   atm_cell cell;
   ...
   this.pre_cell_tx_t(cell, drop);
```

```
   foreach (callbacks[i]) begin
      ...
   end
   if (drop) continue;
   this.notify.indicate(this.PRE_CELL_TX, cell);
   ...
end
```

Pre-defined Events

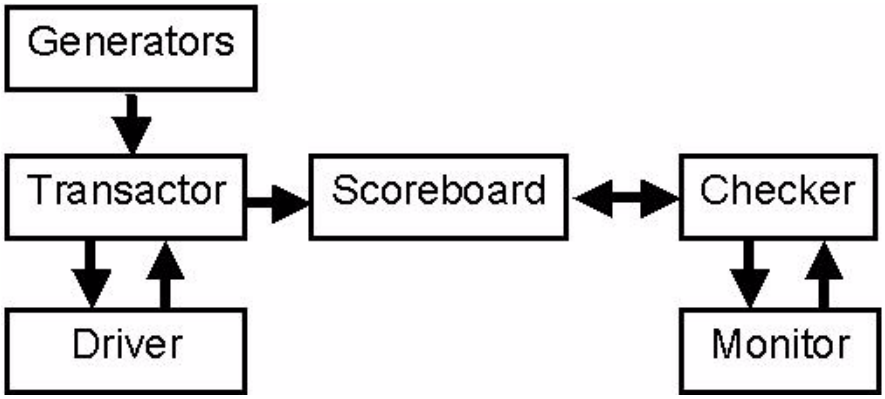
Many VMM classes include notification service interfaces, instances of vmm_notify. As shown above, vmm_data has an instance of vmm_notify and the events EXECUTE, STARTED and ENDED. The vmm_xactor class includes the notify properties XACTOR_IDLE, XACTOR_BUSY, XACTOR_STARTED, XACTOR_STOPPED, and XACTOR_RESET.

Further Exploration

When you use a VMM notification to pass an object, it should be treated as read-only. To pass an object that may be modified in more than one location in your testbench, use the vmm_broadcast class. This can buffer objects so you will not lose data.

Channels and Completion Models

Your testbench environment needs to exchange transactions between its components. For example, transactions flow from the Generator -> Transactor -> Driver, or from the Monitor -> Checker -> Scoreboard. As you saw in the last section, you should model transactions as objects that are then created and modified by the different testbench components.



The connection between these components is the VMM channel. One side is the producer (such as the Generator) putting transactions into the channel. The consumer side (Transactor) gets the transactions out of the channel, and executes them.

The VMM channel has several advantages over the SystemVerilog mailbox:

- Unlike mailboxes, channels are strongly typed which helps prevent coding errors.
- Channels allow flow control, so the put() method will block if the channel is full.

- A channel can have both high-water and low water marks to fine tune the interactions between the producer and consumer. The get() method removes the transaction from the end of the channel, while peek() give you a handle to it without removal. Both block if the channel is empty.
- The output of a channel can be replicated using the tee() method.

Definition and Creation

You define a channel for a specific type using a macro, and then create channels as shown:

```
class atm_cell extends vmm_data;
   ...
endclass

// macro automatically creates new data type by
// appending "_channel" to data_type_name
vmm_channel(atm_cell)    // atm_cell_channel declaration

program test;
  initial begin
    atm_cell_channel chan;
    chan = new("ATM Cell channel", "class");
  end
endprogram

Now two threads can create communicate as follows:
// Producer
forever begin
  atm_cell cell = new();
  ch.put_t(cell);
end
and:
// Consumer
forever begin
  atm_cell cell = ch.get();
```

```
  ...
end
```

Under the Hood

Just like a SystemVerilog mailbox, the channel contains handles to objects, not the object themselves. You can modify an object after it has been put in the channel, leading to a common mistake:

```
// Producer
atm_cell_channel ch = new("ATM Cell channel", "class");
atm_cell cell;
cell = new();
while (...) begin
   void = cell.randomize();
   ch.put_t(cell);
end
```

This code only allocates a single cell. It then repeatedly randomizes this cell and puts it in the channel. The result is that the channel will contain many references to the same object. The solution is to allocate a new cell every time through the loop.

```
while (...) begin
   cell = new();
   void = cell.randomize();
   ch.put_t(cell);
end
```

Using Channels to Connect Blocks

The most common way to use channels is to allocate them in the vmm_env object and then pass them into the testbench components as shown in the following example.

```
task dut_env::build() {
   chan = new(...);
   consumer = new(chan);
   producer = new(chan);
}
```

Transaction Completion

You can synchronize two testbench blocks using a channel. The easiest way is to configure the channel with full=1 (the default) so it works like a procedural interface. The producer thread blocks when it calls put(). When the consumer calls get() to remove the transaction from the channel, the producer unblocks so it can create a new transaction. In the following example, the consumer first calls peek() to read the transaction, but does not call get() until it is done, thus waking the producer:

```
// Consumer
forever begin
   cell = ch.peek();      // Read the cell
   case (cell.kind) {
      ...                 // Process the cell
   }
   void = ch.get();       // Done, wake up producer
end
```

Further Exploration

See the VMM section on “Completion and Response Models” for more examples of synchronizing using channels. What if you have more than one producer or consumer? The vmm_broadcast class is used for one-to-many communication, while vmm_scheduler takes N inputs and combines them into a single output.

Atomic Generators

Tests should tune random generators, not completely rewrite them. This results in less code (each test is smaller), more randomness (all unspecified behavior is random) and more checking (extra randomness broadens the stimulus). Traditional testbenches create stimulus with generators that grow more and more complex as the project progresses, accommodating every variant of stimulus, error generation, synchronization, etc. This “mother of all generators” can be unstable because of the constant changes, as well as difficult to enhance and maintain. VMM recommends a different approach.

Adding Constraints

A typical generator might look like the following:

```
class cell_gen extends vmm_xactor;
   ...
   task main();
      forever begin
         atm_cell cell = new();
         void = cell.randomize();
         this.chan.put_t(tr);
      end
   endtask
```

```
endclass
```

The problem with this generator is that there is no easy way to randomize cells with different constraints.

> **CODING TIP:** While the above example ignores the result from cell.randomize(), you should never do this in real code. Always check the result and issue a vmm_error. Otherwise, you may miss a constraint failure, leading you to think that the test generated cases that it really did not.

- You could use the randomize() with{} construct, but this would require a separate generator for each test, just what you were trying to avoid.
- You could modify the transaction class, atm_cell, adding constraints for each test, but this moves the problem to a different file.

Each of these requires every test writer to edit a common file, with the results applied to every generator / ATM cell. Some testbenches have knobs to control the different distributions and cases, but once again, the generator or transaction becomes the bottleneck, growing in complexity. In addition, the testcase is the testbench plus knob files, adding another file to the flow.

Factories

As an alternative, consider an automobile factory. It creates a stream of cars, each unique with varying options. The manufacturer accomplishes this by having a set of blueprints for the major variants (coupe, sedan, station wagon), and then allowing the user to tweak the details. You can use the same idea with creating transactions. Create a “factory” that stamps out transactions, then have individual

tests feed it different blueprints. All the test-specific code is located in the test file, not the generators. In fact, you can have multiple factories running in a test, each generating a unique set of stimulus.

What does this look like? The following example uses a transaction class that extends vmm_data. The blueprint instance is randomized, then copied to a new instance processed by the next testbench layer.

```
class factory;
   transaction blueprint = new();
   task run();
      while (run) begim
         transaction tr;
         void = blueprint.randomize()
         $cast(tr, blueprint.copy());
         process(tr);
      end
   endtask
endclass
```

You can change the blueprint from the test level, which is just a SystemVerilog program:

```
class my_transaction extends transaction;
   constraint address_even {
      addr[0] == 0;
   }
endclass

program test;
  verif_env env;

  initial begin
    env = new();
    env.build();
    begin
       my_transaction my_tr = new();
       env.src[0].blueprint = my_tr;
    end
```

```
    env.run();
  end
endprogram
```

With this change, the generator src[0] will always create transactions with even addresses.

Benefits

- Tests can modify constraints by
- Making variables non-random
- Turning constraint blocks off
- Add new constraint block
- Re-define constraint blocks
- Add random variables
- Supersede random results with directed data

Factory generators are:

- Good: Unmodified generic, reusable data class
- Good: Unmodified generic, reusable generator class
- Good: Different generators can have different constraints
- Good: Constraint set can be dynamically changed
- Good: Localized test-specific code
- Bad: Difficult to share constraint sets between testcases

Atomic Generator Macro

VCS includes a macro to create an atomic generator:

```
vmm_channel(atm_cell) //Macro: defines atm_cell_channel
vmm_atomic_gen(atm_cell, “ATM_Cell”) //Macro: defines
atm_atomic_gen

class atm_env extends vmm_env;
  atm_cell_atomic_gen gen1; // Generator instance
  atm_cell_channel chan;    // Channel instance

  virtual function void build();
    chan = new(“Channel”, "from gen");
    gen1 = new(“Gen”, *, chan);      // Connect channel
    drv = new (“Driver”, “*”, chan); // to next level
  endfunction
endclass
```

The macro uses the blueprint object randomized_obj.

Lab 3 – Channels and the Atomic Generator

Using the VMM Basic lab instructions, complete Lab 3 for Channels and the Atomic Generator

Transactors

By this point, you have seen VMM transactors. In this section, you are going to learn more about how to create and use them in simulation.

At its simplest, an VMM transactor is just a while loop that reads in transactions from a previous testbench layer, does some processing, and sends out transactions to the next layer. The key is properly starting and stopping the transactors.

The VMM has several transactor types:

- Active Xactor – Master
- Drives pins, blocks on channel get()
- Reactive Xactor – Slave
- Monitors and drives pins, blocks on signal edges
- Passive Xactor – Monitor
- Monitors pins, blocks on signal edges
- Also – Generator or other Xactors as needed
- Creates transactions, blocks using notification or channel put()

A Basic Transactor

Here is a basic transactor. You add code to method main() to process transactions. The other methods all start with a call to the base method to start and stop this method. For example, vmm_xactor::start_xactor() starts the virtual method main() – you don’t need to do this.

```
class driver extends vmm_xactor;
   //start_xactor starts the execution threads
   virtual task start_xactor;
      super.start_xactor();
      ...
   endtask

   //stops execution threads after currently executing
   //transaction had completed. Takes effect at next call
   //to ::wait_if_stopped()
   virtual task stop_xactor();
      super.stop_xactor();
      ...
   endtask

   //resets the xactor’s state and execution threads
   virtual task reset_xactor(...);
      super.reset_xactor(...);
      ...
   endtask
   virtual task main();
      ...
   endtask
endclass
```

Stopping and Starting

The main() method periodically checks to see if the transactor has been stopped by calling wait_if_stopped() as shown below:

```
virtual task main();
   forever begin
      this.wait_if_stopped();
      atm_cell cell = to_driver.get();
      this.wait_if_stopped();
      ...
   end
endtask
```

The wait_if_stopped() method will block if stop_xactor() has been called. Different blocks in your testbench will define when to stop and what it means. You should check if the transactor needs to stop after every time-consuming action, such as the call to get() above.

Physical and Virtual Interfaces

The SystemVerilog interface groups all relevant physical signals (ports) together, just as a C typedef combines several objects into a struct. A virtual interface is just a pointer to a physical interface. You can pass a virtual interface into drivers and monitors. Now the testbench can replicate a driver, with each instance using a separate virtual interface so as to drive multiple physical ports.

If you need to synchronize on a clock edge in an interface, use the clocking block in the interface.

```
@1 bus_ifc.cb;
```

Note that this form does not contain the active edge of the clock signal. If the designer changes the edge, you only have to change the interface definition, not every usage of the signal.

Reusable Transactors

Recall that one of the goals of VMM is that the testbench objects should not have to change. Test specific code goes in the test, not in the transactors such as the driver. The question then, is, how do you write a transactor that can meet all verification requirements such as injecting errors and delays, sampling data for functional coverage and connecting the scoreboard?

Before the VMM, the testbench would have the “mother of all transactors” (MOT) that performed all these actions and more. However, any time one would dream up a new way to control the DUT, one would have to edit the MOT. All these edits make the MOT unstable and a bottleneck for the verification team.

Instead, a transactor should do the simplest things by default – no errors, no delays, but has hooks so that you can add test-specific extensions. You can accomplish this by using callback methods. These are extensible virtual methods that are empty by default.

```
task main();
  ...
  forever begin
    ...
    cb.pre_drive_callback();

    // Drive out the transaction

    cb.post_drive_callback();
  end
endtask
```

In the above code, the method cb.pre_drive_callback() is called just before that transaction is driven into the design, and cb.post_drive_callback() is called just after the transaction is driven. By customizing your own method, you could insert delays, modify or

even drop the transaction, gather functional coverage information on the transmitted transaction, and put this transaction into the scoreboard.

Creating Callbacks

Once you have identified key points in the transactor flow to insert callbacks, you can define a callback façade class:

```
class atm_callbacks extends vmm_xactor_callbacks;
   virtual task pre_cell_tx_t(atm_driver xactor,
                              atm_cell   cell,
                              var bit    drop)
    …
  endtask
endclass
```

The driver now just calls pre_cell_tx() before driving the cell. Next, define your test-specific callback classes such as error_inject, scoreboard_insert, functional_cov that extend atm_callbacks. Each callback class is then registered at the test level. The callbacks are called in the order you registered them. So, be sure to inject errors first, and then add the transaction to the scoreboard.

```
class stretch_ifg extends atm_callbacks;
   // This class stretches the inter-frame gap
   ...
endclass

program test;
  verif_env env;
  initial begin
    env = new();
    env.build();
    begin
      stretch_ifg cb = new();
      env.driver[0].prepend_callback(cb);
```

```
      env.driver[3].prepend_callback(cb);
    end
    env.run();
  end
endprogram
```

You can see that the code to invoke the callbacks is common across all transactors, so the VMM provides a macro:

```
class atm_driver extends vmm_xactor;
  ...
   forever begin
      atm_cell cells = this.in_chan.get();
      bit drop = 0;

      `vmm_callback(atm_driver_callbacks,
                    pre_cell_tx_t(this, cell, drop));
      if (drop) continue;

      // Drive the cell here …

      `vmm_callback(atm_driver_callbacks,
                    pre_cell_tx_t(this, cell, drop));
   end
endclass
```

Labs 4, 5, 6 and 7

Using the VMM Basic lab instructions, complete Lab 4 for the APB Master Transactor, Lab 5 for the Monitor Transactor, and Lab 6, Scoreboard Integration

Lab 7, Functional Coverage, is optional. It shows an example of adjusting random constraints to increate functional coverage.

OOP & Virtual Methods

With traditional procedural programming, you create structs to hold the data, and global procedures that manipulate the data. With Object Oriented Programming, you wrap the data and the procedures inside of a class:

```
class packet;
   logic [31:0] src, dst, crc, data[8];

   task display();
      ...
   endtask

   function logic [31:0] compute_crc();
      ...
   endfunction
endclass

program test;
  packet p;
  initial begin
    p = new();
    p.display();
    p.crc = p.compute_crc();
  end
endprogram
```

Inheritance

Now that you have a packet, how do you create different flavors? For example, you might want to add an is_good bit that tells if the class should always generate good CRC. In OOP, this is done by extending the original packet class (base class):

```
class my_packet extends packet;
   bit is_good;
```

```
   function compute_crc();
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = urandom;
   endfunction
endclass
```

Handles to Objects

When you make the following declaration:

```
packet p;
```

you are creating a handle that can reference objects of type packet. The handle is initialized to null. You can call the new() method to create an object:

```
p = new();
```

This allocates enough memory to hold the class packet, 11 longwords (sa, da, crc, and data[8]). If you allocate a my_packet object, 12 longwords will be allocated (the original 11 plus one more for the is_good bit).

What happens if you try to mix handles for base and extended classes? The easiest way to visualize this is to consider the is_good bit. It would be an error if you used a my_packet handle to access the is_good bit, but the handle points to only a packet object (11 longwords).

```
packet p;
my_packet mp;

mp = new();
p = mp;     // Good: Copies a handle

p = new();
mp = p;     // Error: mp.is_good won't exist
```

But what if you use a base handle to point to an extended object, then try to assign back to an extended handle? This is normally not allowed, so you will need to use $cast(). This method checks the object type to make sure it matches the destination object.

```
packet p;
my_packet mp, m2;

     mp = new();
     p = mp;        // Still good
      $cast(m2, p); // Good: p points to my_packet object
```

Polymorphism

What happens if you call a method in a class. By default, if you use a packet handle to call the compute_crc() method, the packet::compute_crc() method is called, even if the object was actually of type my_packet. (See code in the Inheritance section above.) But if you use virtual methods, VCS will call the method based on the object type, not the handle type:

```
class packet;
   virtual function logic [31:0] compute_crc();
      ...
   endfunction
endclass

class my_packet extends packet;
   bit is_good;
   virtual function compute_crc();
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = ~compute_crc;
   endfunction
endclass

packet p;
```

```
my_packet mp;

initial begin
  mp = new();
  p = mp;
  p.crc = p.compute_crc();  // Calls my_packet method
end
```

Why is this useful? You can write a generic class such as packet and use it in a testbench:

```
class protocol;
   ...
   task transmit(packet pkt);
      ...
      pkt.crc = pkt.compute_crc();
      off = vera_pack(bytes, off, pkt);
      // Transmit packet data
   endtask
endclass
```

If you call the transmit() method with a packet object, it will call packet::compute_crc(). Call it with a my_packet object and it will call my_packet::compute_crc() and inject errors.

Labs

LAB 1: VMM Environment

Goal	Create a testbench environment classIssue a message
Location	lab1
Base Classes	vmm_env, vmm_log
Allocated Time	15 minutes

The DUT Environment class is the main top-level testbench class.

apb/apb_cfg.sv	APB Configuration Class
env/dut_env.sv	DUT Environment Class
tests/test_01.sv	Simple Test to run the environment

This class instantiates all the permanent testbench elements, and controls the sequence of testbench steps. The environment is DUT specific, so it is in the env/ directory.

1. DUT Environment Class Code Review

   Review the dut_env.sv file and answer the following.

   How many steps are present in the vmm_env flow?

   _____

   Are the tasks/functions all defined as virtual?_____

   Why? _____

2. APB Configuration Class Code Review

The apb_cfg class contains the random configuration details for the testbench. For simplicity, this consists of a single data item, to determine how many cycles to run before exiting the testbench.

Review the code for the testbench configuration object in apb_cfg.sv.

How many random integers are present in the class?_____

3. Adding APB Configuration Class to the APB Environment

   Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a “Lab1 – comment” to provide help in completing the task. The following steps add the config object to the APB environment:

   Add a vmm_log handle to the dut_env class

   Add a configuration handle to the dut_env class

   Construct the log handle using new(“dut”, “env”)

   Construct the configuration handle

   Randomize the configuration in the dut_env::gen_cfg() task

   Add a debug print statement to the end of the dut_env::gen_cfg() task to print the value of the cfg.trans_cnt data using the `vmm_note() macro. Hint: Use $psprintf or $sformat

4. Compile and Run the Testbench

   Compile the testbench using the following command in the lab1 directory.

```
csh% gmake lab1
```

Note the VCS command and options that are used by the Makefile here.

vcs

```
csh% gmake lab1
```

Run the testbench several times with random seeds.

```
Verify the trans_cnt value changes.
csh% ./simv +ntb_random_seed=1
csh% ./simv +ntb_random_seed=2
csh% ./simv +ntb_random_seed=3
```

Solutions for the above questions are at the end of this document.

LAB 2: Creating a vmm_data class

Goal	Create an APB transaction class
Location	lab2
Base Classes	vmm_data
Allocated Time	15 minutes

apb/apb_trans.sv	APB Transaction Class
Tests/test_02.sv	APB Transaction Test Program

The APB Data class will contain properties for an APB transaction. The properties include addr, data and an enumerated type for direction. This code is part of the reusable VIP, so it is in the apb/ directory. Each step has comments in the code to assist in editing.

1. Properties

   The transaction class contains a random property for each data field.

For the APB protocol, add the following properties.

typedef enum {READ, WRITE} dir_e;

rand dir_e dir;

rand logic [31:0] addr, data;

2. copy()

   The copy() function is used by a transaction to make a copy of itself.

   For the APB protocol, review the copy() function contents.

3. copy_data()

   The copy_data() function is used by a transaction to copy each data field.

   Complete the copy_data() function by adding a line for each property.

4. compare()

   The compare() function is used by a transaction to compare itself with another transaction. The pass/fail result is returned by the function, and a ‘diff’ string is used to return a compare message.

   For the APB protocol, complete the compare() function by adding a block to compare the property to the end of the compare() function.

5. psdisplay()

   The psdisplay() function is used to create a text string from a transaction for printing to the screen or log file.

Review the psdisplay() function and note the function uses several $sformat() statements to create a string. The string contains the transaction in a text format, including the stream_id, scenario_id and data_id values.

6. byte_size()

   This function calculates the number of bytes in a transaction.

   Review the code for byte_size(). In our simplified APB protocol, the size is always 4 bytes, so this function returns a constant.

7. byte_pack()

   This function packs the transaction object into a byte-array.

   Review the code for byte_pack(). In our simplified APB protocol, the transaction is always 4 bytes long, so the transaction can simply be packed with 4 assignment statements.

8. byte_unpack()

   This function unpacks a byte-array of data into a transaction object.

   Review the code for byte_unpack(). In our simplified APB protocol, the transaction is always 4 bytes long, so the transaction can be unpacked with an assignment statement.

9. Testing the apb_trans class

   A short test is supplied to test the apb_trans class. Briefly review the teststs/test02.sv and note that the test performs the following tasks.

   creates a random apb_trans

   prints this transaction to the screen

copies the apb_trans

prints the copied transaction to the screen

compares the two transactions

Compile and run the test using the following command in the lab2 directory.

```
csh% gmake lab2
```

LAB 3: Channels and Atomic Generator

Goal	Create an atomic generator and add it to the environment
Location	lab3
Base Classes	vmm_data, vmm_atomic_gen macro
Allocated Time	15 minutes

apb/apb_trans.sv	APB Transaction Class
apb/apb_cfg.sv	APB Configuration Class
env/dut_env.sv	DUT Environment Class
tests/test_03.sv	Simple Test Program

In this lab, an atomic generator and channel will be added to the environment. The atomic generator will create streams of individually-randomized transactions. The apb-channel will be used to connect the generator to other transactors in the environment.

The APB Transaction class contains the following two macro statements.

```
`vmm_channel(apb_trans)
`vmm_atomic_gen(apb_trans)
```

These macro statements cause the following classes to be created automatically when the apb_trans.sv file is compiled.

```
class apb_trans_channel;
class apb_trans_atomic_gen;
```

1. Addition of the Atomic Generator and Channel

   The macro statement creates the atm_trans_atomic_gen class when the apb_trans.sv file is compiled. This class can easily be used by the testbench to create a stream of individually-randomized transactions.

   Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a “Lab3 – comment” to provide help in completing the task. The following steps are required to add an atomic generator to the environment.

   Define an apb_trans_channel handle called gen2mas

   Define an apb_trans_atomic_gen handle called gen

   Create the channel by calling new() in the dut_env::build() task

   Create the generator by calling new() in the dut_env::build() task

   Configure the generator by adding gen.stop_after_n_inst = cfg.trans_cnt in the dut_env::build() task

   Start the generator by calling gen.start_xactor() in dut_env::start()

   Wait for the generator to signal completion (DONE) by calling gen.notify.wait_for() in the dut_env::wait_for_end() task

(hint: apb_trans_atomic_gen::DONE, Note: enum {DONE} is created when vmm_atomic_gen macro is called, please refer to VMM manual for more detail)

- Stop the generator by calling gen.stop_xactor() in dut_env::stop()

2. Compile and Run the Testbench

   Compile and run the testbench using the following command.

   csh% gmake lab3

   Note:

   All atomic generators contain the following properties.

   -randomized_obj (blueprint)

   -stop_after_n_insts(control)

   -DONE (notify)

   -GENERATED (notify)

LAB 4: APB Master Transactor

Goal	Create an APB Master Transactor
Location	Lab4
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	30 minutes

apb/apb_if.sv	APB Interface Definition
apb/apb_master.sv	APB Master Transactor
apb/apb_trans.sv	APB Transaction Class
env/dut_env.sv	Testbench Environment
tests/test_04.sv	Simple Test Program

A timing diagram for the APB interface is shown below for reference during this lab.

The APB master transactor extracts apb_trans objects from a channel, and executes read/write cycles on the APB bus. The goal of this lab is to create a VMM-compliant transactor, not to focus on the low-level physical protocol. For this reason, do_read(), do_write() and do_idle() tasks are provided to perform the bus cycles.

1. Create an APB Master Transactor

   Complete the new() task of the apb_master class, using the following arguments.

```
string instance;
integer stream_id = -1;
virtual apb_if.Master apb_master_mp;
apb_trans_channel in_chan = null;
```

2. Review the new() task

Review the code in the new() task. Note that if the in_chan argument is not specified (null), a channel will be created automatically.

3. Review the main() task

   Note the following:

   -super.main is called to perform any base-class actions

   -The body of main() is an infinite forever loop

   -A transaction is extracted from the channel

   - A pre-tx callback is called

   -The transaction is processed (read, write or idle cycles executed on the bus)

   -A post-tx callback is called

4. Add a post-tx callback

   Add a `vmm_callback() macro call to the main() task after the transaction is processed. The arguments to the macro are detailed in the comments.

5. Review the reset(), do_read(), do_write(),do_idle() tasks

   These tasks execute the various transactions on the physical bus. Note that these tasks contain code that is typically present inside a BFM model. These tasks block as needed, and are protocol specific.

6. Integration of APB Master into the APB Environment

   The following steps are required to add the apb_master into the environment in dut_env.sv.

   -Define an apb_master handle called mst

-Create the master by calling new() in the dut_env::build() task

-Reset the master by calling mst.reset() in dut_env::reset_dut() task

-Start the master by calling mst.start_xactor() in dut_env::start()

-Stop the master by calling mst.stop_xactor() in dut_env::stop()

Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a “// Lab4” comment to provide guidance.

7. Compile and Run the Testbench

   Compile and run the testbench using the following command in the lab4 directory.

   csh% gmake lab4

   Optional: View the waveforms using DVE.

   csh% dve &

   File > Open Database > vcdplus.vpd

LAB 5: APB Monitor Transactor

Goal	Create an APB Monitor Transactor
Location	Lab5
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

apb/apb_if.sv	APB Interface Definition
apb/apb_monitor.sv	APB Monitor Transactor
apb/apb_trans.sv	APB Transaction Class
apb/dut_env.sv	Testbench Environment
tests/test_05.sv	Simple Test Program

1. Review the APB Monitor Code

   Note that the Monitor is similar to a generator, as both create a stream of transactions. The monitor uses the factory pattern, to copy a blueprint in a similar fashion to the generator. This allows the test writer to replace the default transaction object with a custom object, containing additional tracking information or other properties.

   The monitor class contains a randomized_obj declaration that is constructed in the apb_monitor::new() task.

   The main() task contains an infinite loop, to constantly monitor the bus.

   The bus is monitored in the sample_bus() task containing BFM-like code.

2. Add the Factory/Prototype Pattern code

In the main() task, add code to copy the randomized_obj instance, and cast assign this to the local ‘tr’ data variable. Comments in the code provide hints on how to accomplish this.

3. Add a Post-Rx Callback

   After the sample_bus() task is called, add a Post-Rx callback macro call to invoke the VMM callback class tasks. Comments in the code help with the required syntax; this will be very similar to the Post-Rx callback in the APB Master.

   Integration of APB Monitor into the APB Environment

   The following steps are required to add the apb_monitor into the environment. Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps Each step has a “Lab5 – comment” to provide help in completing the task.

   -Define an apb_monitor handle called mon

   -Define a dut_scb handle called scb

   -Create the scoreboard by calling new() in the dut_env::build() task

   -Create the monitor by calling new() in the dut_env::build() task

   -Start the monitor by calling mon.start_xactor() in dut_env::start()

   -Stop the monitor by calling mon.stop_xactor() in dut_env::stop()

4. Compile and Run the Testbench

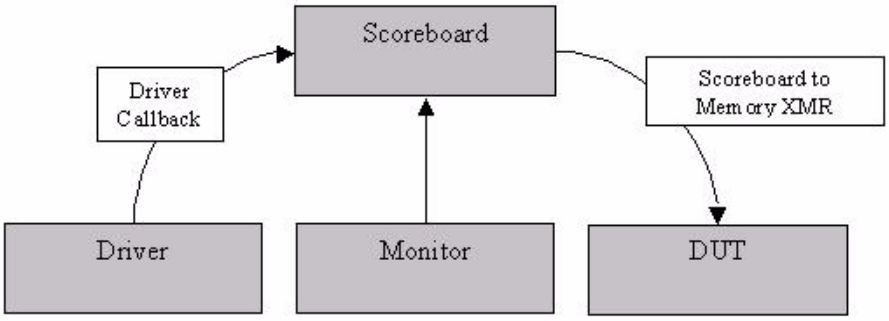
   Compile and run the testbench using the following command in the lab5 directory.

   csh% gmake lab5

LAB 6: Scoreboard Integration

Goal	Integrate the scoreboard using callbacks
Location	Lab6
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

env/apb_scb.sv	Scoreboard Class
env/scb_callbacks.sv	Scoreboard Callback Class
env/dut_env.sv	Test Bench Environment
tests/test_06.sv	Simple Test Program



The scoreboard is integrated using callbacks attached to the Driver and Monitor transactors. This allows the scoreboard to integrate with the APB VIP components in a passive manner, and requires no code changes or restrictions on the APB classes.

1. Review the APB Callback Class

   Review the apb/apb_master.sv file and verify following tasks are present.

   ```
   apb_master_callbacks::master_pre_tx()
   ```

```
apb_master_callbacks::master_post_tx()
```

As these classes are base-classes for the APB callbacks, the tasks are empty. The code to perform callback actions is DUT or test specific, so it does not appear in the APB VIP directory files.

2. Review the Scoreboard Callback Class and Integrate the Scoreboard

   Review the env/scb_callbacks.sv file.

   Verify the following class::tasks are present

   ```
   apb_master_callbacks::master_pre_tx()
   apb_master_callbacks::master_post_tx()
   apb_monitor_callbacks::monitor_pre_tx()
   apb_monitor_callbacks::monitor_post_tx()
   ```

   Add the following code

   - In master_post_tx, call scb.from_master() to add a transaction to the scoreboard
   - In monitor_post_tx, call scb.compare() to compare the received transaction to the expected one

3. Review the Scoreboard Class

   Briefly review the env/dut_scb.sv file.

   Note the following task to add data to the scoreboard from the master callback dut_scb::from_master(), and another task to compare the actual with expected dut_scb::compare().

   Integration of Scoreboard into the APB Environment

The following steps are required to add the scoreboard into the environment. Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a “Lab6 – comment” to provide help in completing the task.

- Create and construct an apb_master_scb_callbacks object called apb_mst_scb_cb, and append the callback to the APB Master object mst
- Create and construct an apb_monitor_scb_callbacks object called apb_mon_scb_cb, and append the callback to the APB Monitor object mst
- In wait_for_end, Wait for either the generator or scoreboard in a fork-join_any. The generator’s wait is already there, just call scb.notify.wait_for(scb.DONE)
- Clean up after the test by calling scb.cleanup() in dut_env::cleanup()
- Create a report by calling scb.report () in dut_env::report ()

4. Compile and Run the Testbench

   Compile and run the testbench using the following command in the lab6 directory.

   csh% gmake lab6

LAB 7: Functional Coverage

Goal	Add functional coverage using callbacks
Location	Lab7
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

env/cov_callback.sv	Coverage Callbacks & Coverage Code
env/dut_env.sv	Test Bench Environment
tests/test_07.sv	Simple Test Program

The coverage will be integrated using callbacks attached to the Master transactor. This allows coverage to be added in a passive manner, with no changes to the APB master.

1. Add Coverage Points

   The structure of the coverage callback class is similar to the scoreboard callback class. Significantly more code is present in the coverage callback, as the coverage model has been placed directly in the class.

   Review the env/cov_callbacks.sv file, and add the following coverage points:

```
data: coverpoint tr.data {
   bins zero = {0};
   bins onek = {1024};
   bins others = default;
}
```

2. Integration of Scoreboard into the APB Environment

   The following steps are required to add the scoreboard into the environment:

- Define an apb_master_cov_callback cov_cb and call new()
- Append the callback to the mst APB Master object

Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a “Lab7 – comment” to provide help in completing the task. This code will look very similar to the master scoreboard callback.

3. Compile and Run the Testbench

   Compile and run the test using the following command in the lab7 directory

   csh% gmake lab7

4. Create and View the Functional Coverage Reports

   The unified report generator is used to create the reports. Run the report using the following command:

   csh% gmake cov

   View the report by opening the HTML files in the urgReport/ directory with a browser.

   Or use the following commands for older-style HTML or text coverage reports:

```
csh% vcs –cov_report .
csh% vcs –cov__text_report.
```

5. Constrain the atomic generator to increase coverage

   It is unlikely that totally random APB transactions will hit the specified coverage points unless we run a huge number of simulation cycles. Adding constraints into the test to increase the chances that interesting stimulus will be generated will help.

Modify the tests/test_07.sv file as follows:

- Add a new class derived from apb_trans called my_apb_trans
- Add a constraint into the new class to constrain addr and data (see comments)
- In the environment initial block, create a new scope with begin … end
- Create a new my_apb_trans object
- Place this object into the atomic generator object gen
- Re-run the test, and examine the log output and coverage reports.

The addr and data fields should now be constrained, and the coverage numbers should be higher than before.

csh% gmake cov

csh% vcs –cov_report .

Question/ Answers

Lab1:

How many steps are present in the vmm_env flow? *9*

Are the tasks all defined as virtual? *Yes*

Why? So you can override the functionality in your own derived env class.

How many random integers are present in the class? *1*