VCS®/VCSi™
SystemVerilog Testbench Tutorial

Version X-2005.06
August 2005

Comments?
E-mail your comments about Synopsys documentation to doc@synopsys.com

SYNOPSYS®

SystemVerilog Testbench Tutorial Version X-2005.06

Contents

1

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

- High-Level Verification
- SystemVerilog for Testbench

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.

2

Design Overview

This chapter 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:

- Memory System
- Tutorial-design Directory Setup

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

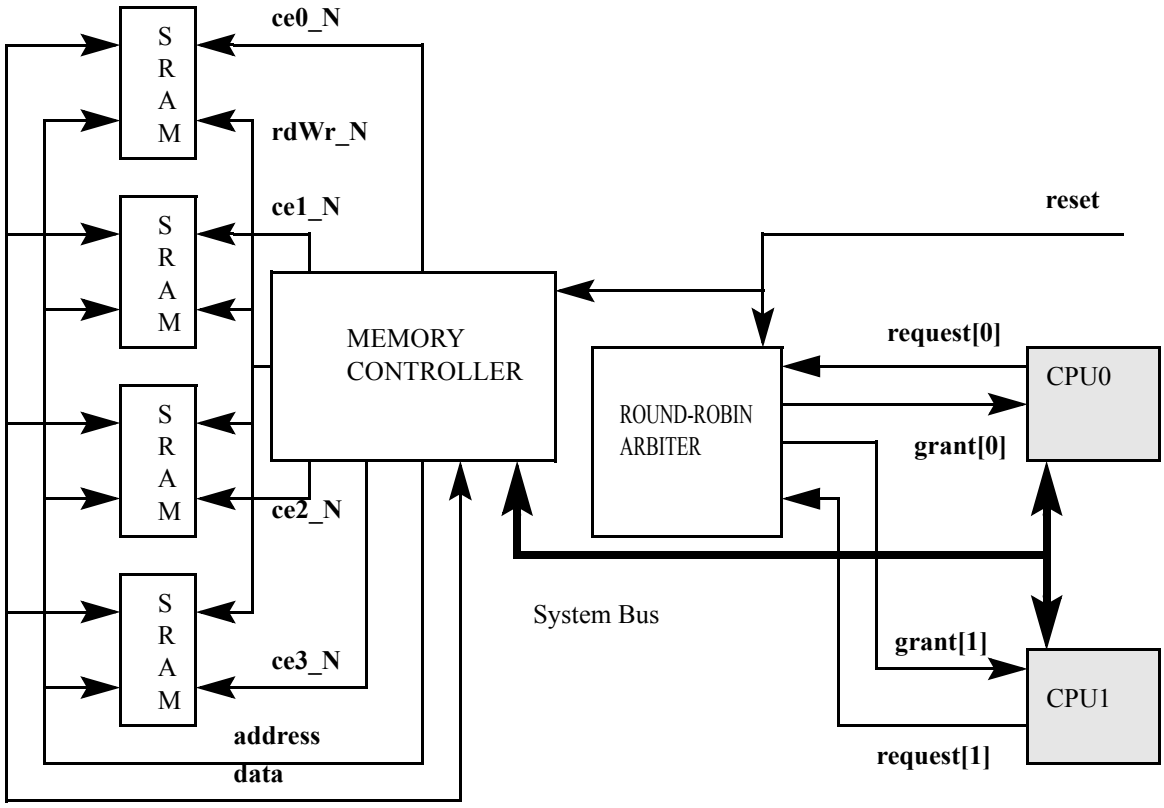


Figure 2-1 Memory System Schematic

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.

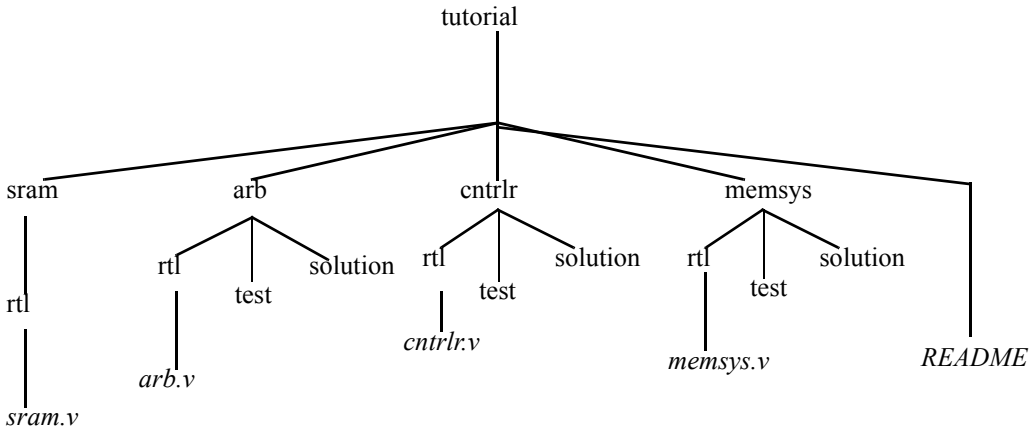
This “full chip” functionality is verified at the system level.

First, in Chapter 3 of the tutorial, the arbiter sub-module is verified. To do this, the surrounding blocks are modeled in the testbench.

Then, in Chapter 4, the memory controller sub-module is verified. For this, both the CPU and the memory are designed in the testbench. This gives you a chance to use some of the advanced features of SystemVerilog for Testbench that could be used to verify protocol based designs.

Finally, Chapter 5 of the tutorial shows you how to verify the complete system by integrating the arbiter and controller sub-modules as shown in the diagram above, with the SystemVerilog models of the CPUs instantiated in the testbench. Several different features of SystemVerilog for Testbench will be used in different approaches. We will also introduce object-oriented testbench design in this last chapter.

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.

Location of Files for Tutorial

The files for this tutorial are located at $VCS_HOME/doc/examples/nativetestbench/systemverilog/tutorial.

3

Arbiter

This chapter focuses on the arbiter's roll in the design. It briefly describes what the arbiter does, including a short timing and logic discussion. The chapter then describes the SystemVerilog for Testbench technology as it pertains to verifying the arbiter portion of the system. Specifically, this chapter 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 chapter is divided into the following sections:

- Arbiter Overview
- Testbench Overview
- Verifying the Arbiter

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:

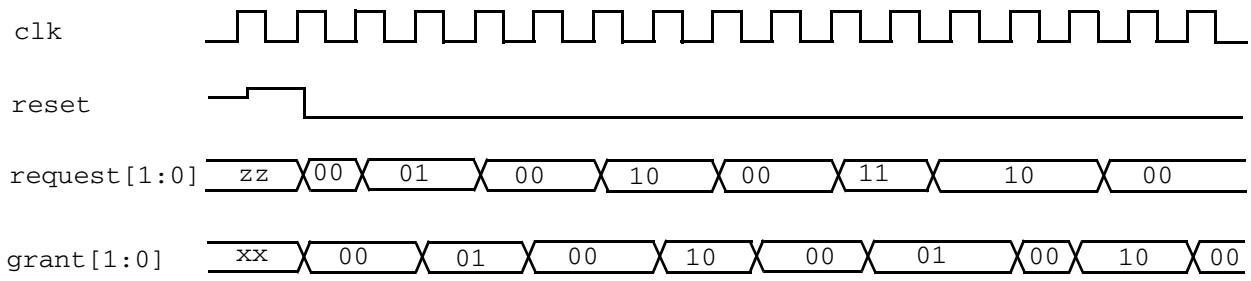
  `tutorial/arb/solution/README`

- 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 3-1 shows the timing diagram.

Figure 3-1 Arbiter Timing Diagram



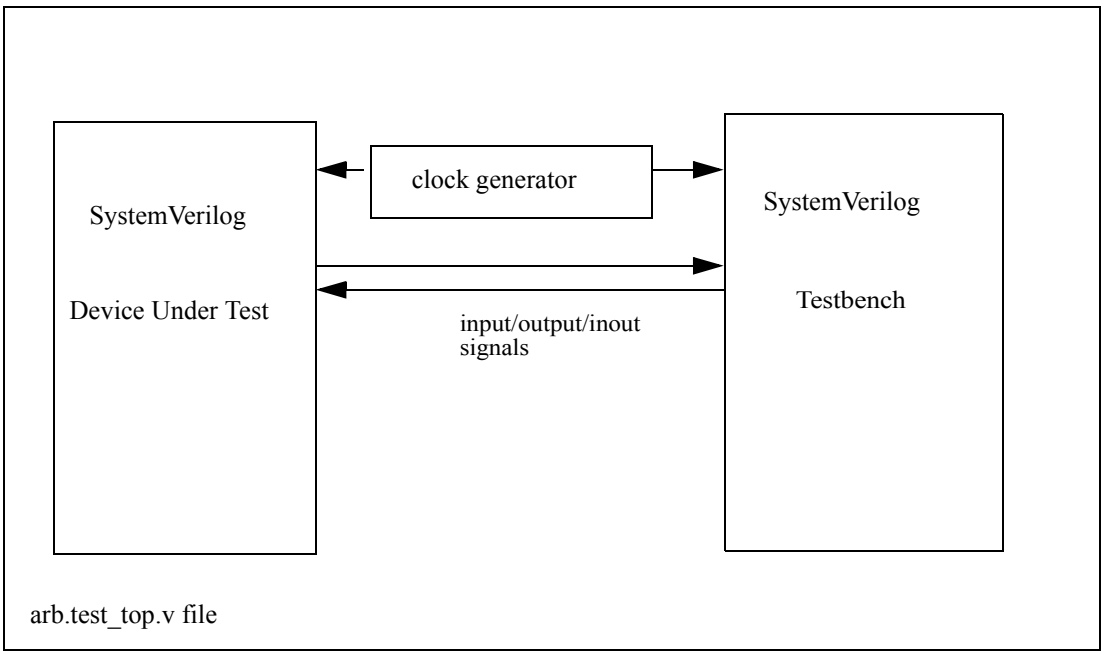
The arbiter implements a round-robin arbitration algorithm between two CPUs. Each CPU can drive a request input signal (**request**[0] or **request**[1]). The arbiter queues the requests and determines which CPU will gain access to the system bus. The arbiter grants this access by asserting one of the 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 3-2 below shows basic schematic for this configuration.

Figure 3-2 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
```

4

Memory Controller

This chapter 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 chapter includes the following sections:

- Memory Controller Overview
- Verifying the Memory Controller

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 4-1 shows a diagram of how the testbench works with both the system bus and SRAM device signals.

Figure 4-1 SystemVerilog Testbench/Memory Controller Interaction

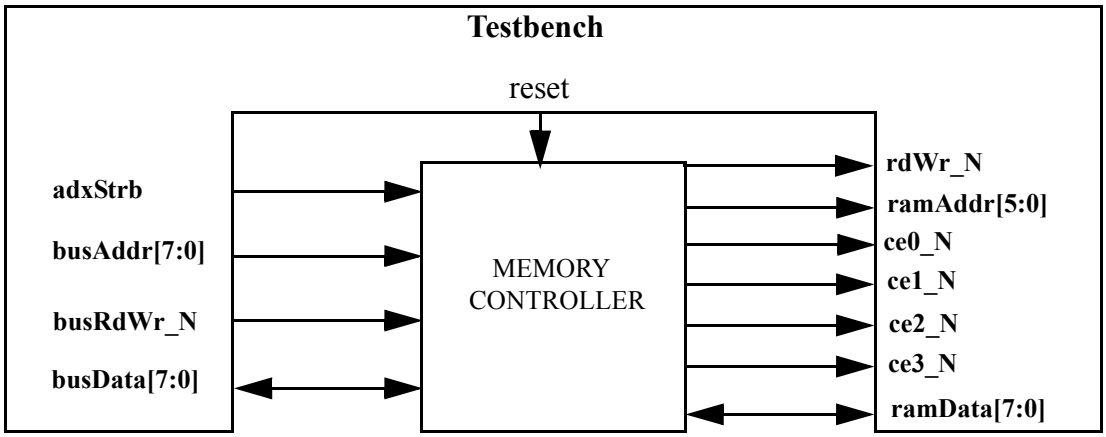
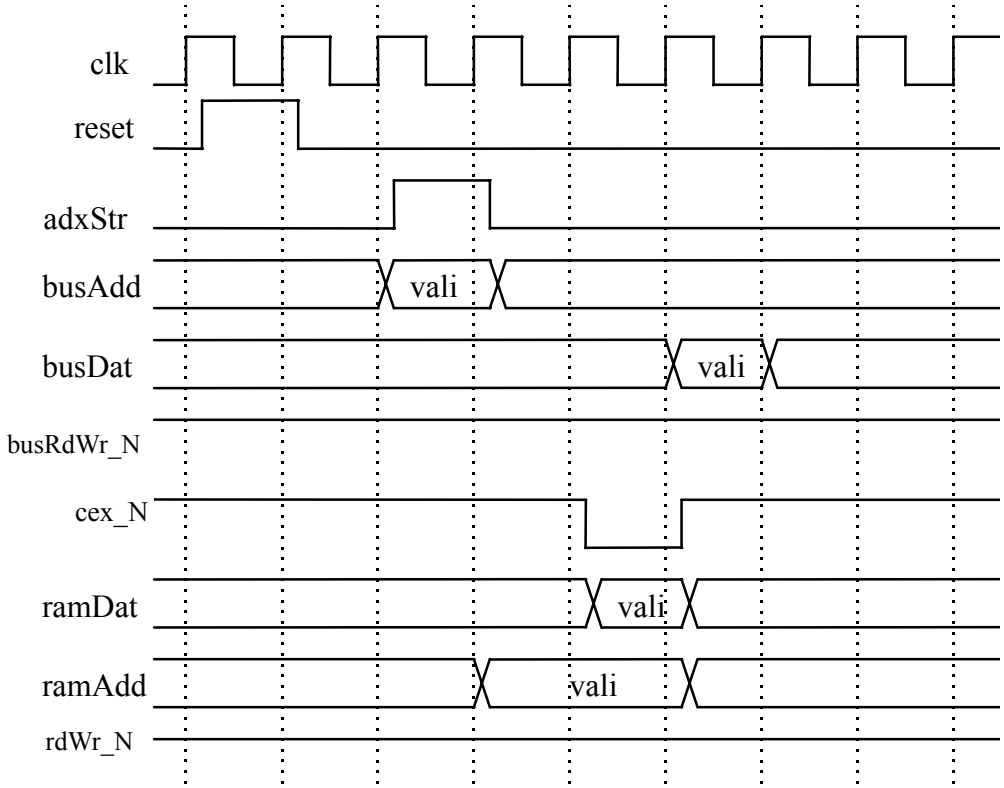


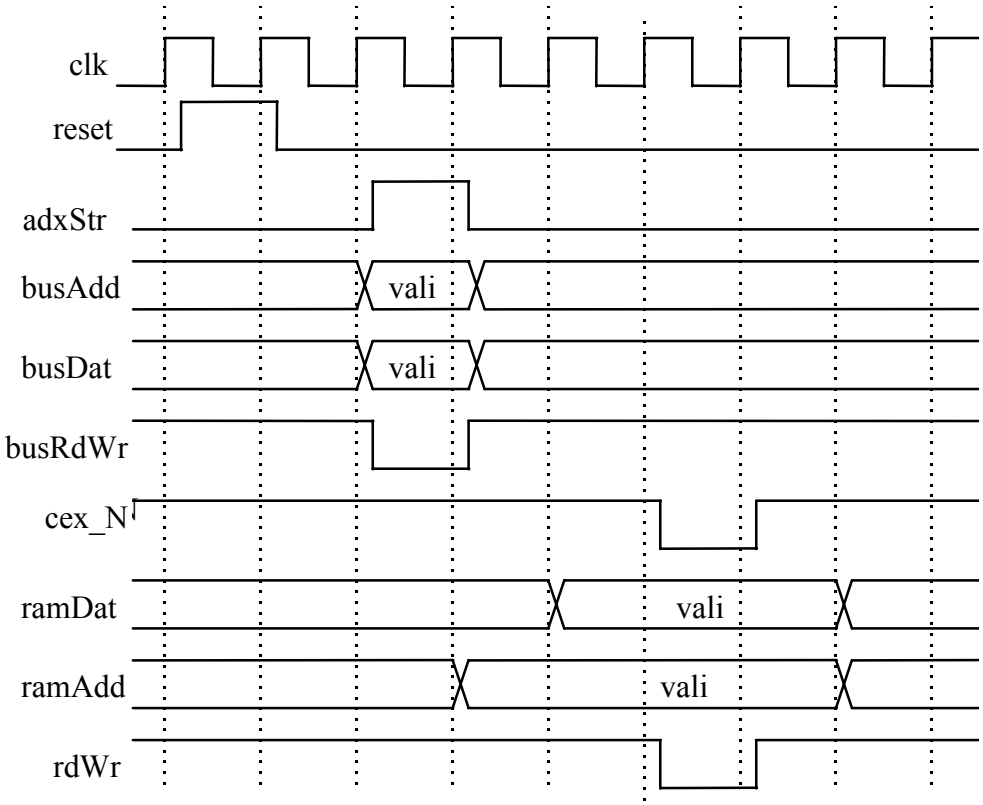
Figure 4-2 and Figure 4-3 show the timing diagrams for the memory controller's read and write operations respectively (note the signal names as you will be using them in the verification process).

Figure 4-2 Memory Controller Read Operation Timing Diagram



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

Figure 4-3 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 chapter 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.

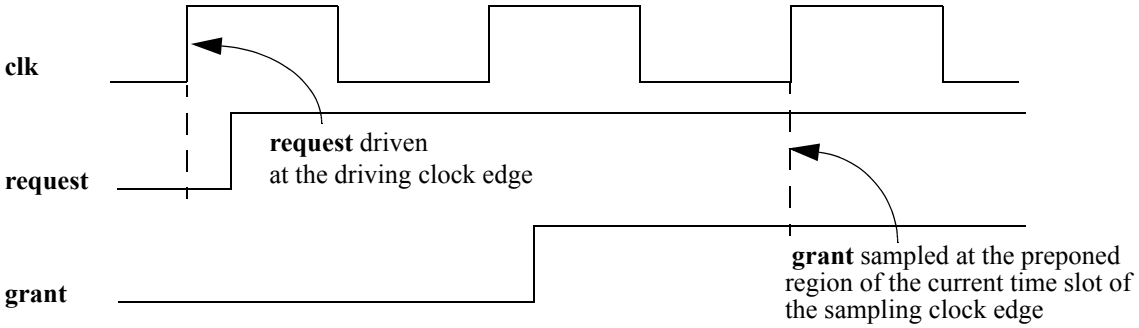
```
interface cntrlr_intf(input clk);
  wire  reset ;
  wire  [7:0]  busAddr ;
  wire  [7:0]  busData ;
  wire  busRdWr_N ;
  wire  adxStrb ;
  wire  rdWr_N ;
  wire  ce0_N ;
  wire  ce1_N ;
  wire  ce2_N ;
  wire  ce3_N ;
  wire  [5:0]  ramAddr ;
  wire  [7:0]  ramData ;

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 VCS8.0alpha 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.

5

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 chapter 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 chapter will show you how to use these features to validate the memory system. This chapter includes the following sections:

- Memory System Overview
- Verifying the Memory System

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:

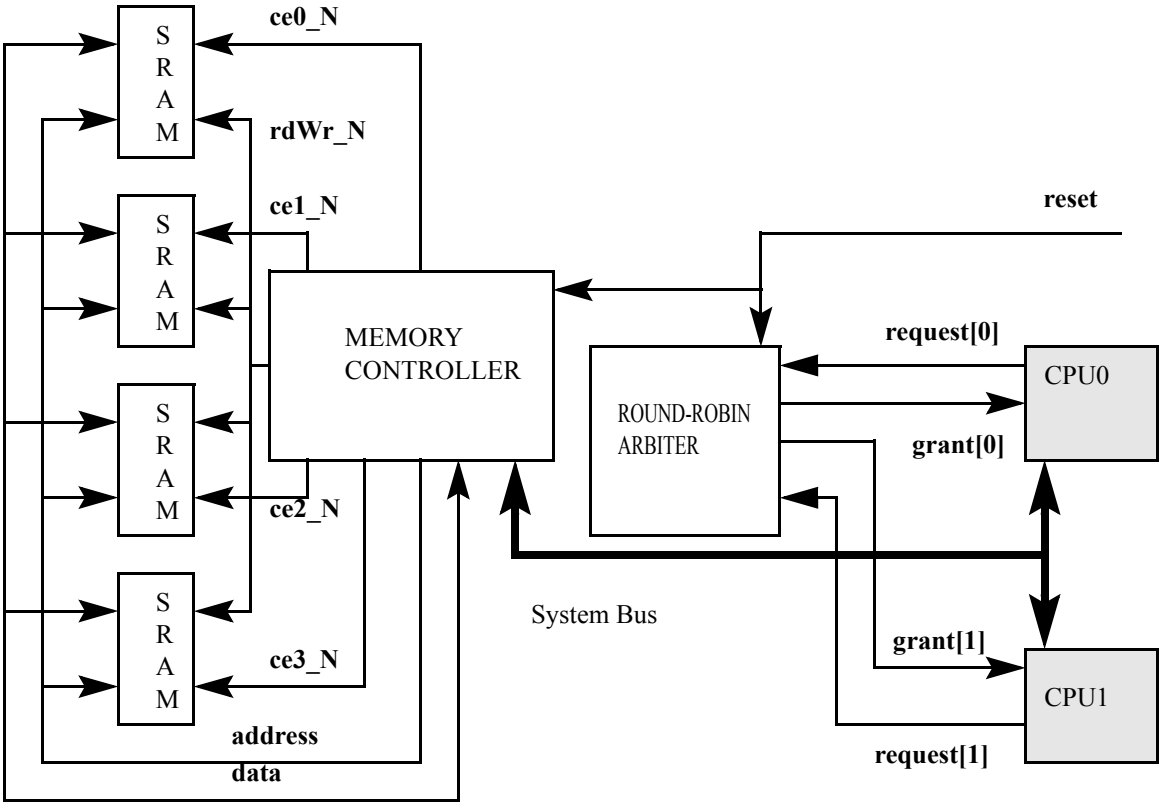
  `tutorial/memsys/solution/README`

- 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 5-1.

Figure 5-1 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 chapter. 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
```