VCS Testbench Quick Start Guide

Version E-2011.03
March 2011

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

Introduction

Simulation-based verification dominates the majority of the design validation process. The task for design and verification engineers together is to create an environment that stimulates the input to the design and watches for errors by checking the output from the design.

RTL design and verification teams use Verilog or VHDL hardware description languages to set up the verification environment for their designs, relying on directed test cases. With this approach the success of the design and verification teams depends on their ability to anticipate scenarios for test cases.

To improve the efficiency and comprehensiveness of verification, VCS supports built-in features for constrained random verification, and functional coverage.

With these built-in features, you can code valid constraints on data and types of activities to stimulate the Design Under Test (DUT). Once a constrained-random environment is set up, you can debug and validate the simulation results. You can also perform functional coverage analysis to find areas that have not been verified.

The built-in coverage features in VCS provide both functional coverage (stimulus coverage) and code coverage. You can find out where you need to steer the stimulus by modifying constraints in the verification environment. You can then develop directed tests to deal with the areas of interest that have not been covered yet.

Code coverage capabilities are supported for Verilog, VHDL, and mixed language designs. The VCS unified coverage metrics, which encompass functional coverage, assertion coverage, and code coverage, enable the merging of results from different kinds of coverage across all tests in a regression suite. This enables project engineers to measure the status of the verification process and its completeness.

The VCS verification solution also provides assertions to help you check design correctness throughout the design hierarchy as well as on the boundary of the design, thus enhancing visibility into the design by revealing bugs very close to the source. Ideally, as designers write the RTL they document assumptions about the rules for interfaces to adjoining blocks and how the design is expected to behave. This documentation is also reusable. SystemVerilog assertions (SVA) are fully supported in VCS and significantly enhance the development process. The SVA Checker library that is incorporated into VCS is also very useful in this process since you can easily plug in pre-written and pre-verified checks to help you with bug finding.

For larger designs under test, such as SOC, and systems that use industry standard protocols such as PCIExpress, Ethernet, etc., users have access to pre-built and pre-verified verification IP components (VIP) that are ready to plug into the testbench environment. VCS also incorporates pre-built and pre-verified assertion-based IPs (AIP) that provide protocol coverage and checks for dynamic simulations.

VCS includes the Discovery Visual Environment (DVE), an integrated graphical debug environment specially engineered to work with the VCS advanced bug-finding features. DVE enables easy access to design and verification data in an intuitive, menu-driven environment. The debug capabilities include features such as:

- Driver tracing
- Waveform comparison
- Schematic views
- Path schematics
- Support for VCD+ binary dump format

The graphical debugging visualizations also provide mixed-HDL (Verilog, VHDL, SystemVerilog) code debugging tools, as well as assertion tracing to help automate manual tracing of relevant signals and sequences.

VCS native support for verification can yield up to a 5x performance speedup and a 3x reduction in memory usage. VCS enables the thorough verification of complex SoC designs, in much less time than with other tools and methods. The result is a far higher likelihood of first silicon success.

In this quick start, we introduce the testbench and verification features in steps. We demonstrate how you can use the features of VCS to verify the FIFO block described in Appendix A. You do not have to re-architect your environment to take advantage of the verification features of VCS. We show you how you can incrementally use these features in your existing environment.

Each chapter introduces the next set of advanced constructs, starting with basic constructs in chapter 2. Chapter 3 introduces high-level abstraction using object-oriented programing concepts and interfaces, including the concepts of structured and layered testbench development using advanced VCS constructs.

2

Using Basic VCS Features

This chapter provides an introduction to SystemVerilog assertions, SystemVerilog testbench constructs, Discovery Visual Environment (DVE), and functional and code coverage. A FIFO module is used to demonstrate how to develop a verification environment using SystemVerilog testbench constructs for constraints and coverage.

This chapter describes the methodology, testbench constructs and assertion technology used in constructing a constrained random verification environment.

The FIFO Design

The FIFO that we want to verify is a synchronous (single-clock) FIFO with static flags. It is fully parameterized and has single-cycle push and pop operations. It has empty, half-full, and full flags, as well as parameterized almost full and almost empty flag thresholds with error flags. (A block-level schematic and timing diagram are shown in “The FIFO Design Block” on page 79”.)

The following sections provide guidelines and examples of how you can use the basic features of VCS to verify the FIFO block. The testbench and verification code provide you with the steps to simulate designs with VCS. You can use this structure and the code with your current design environment to increase the number of testcases as well as to create more complete and complex routines for testing the DUT.

Verification Architecture

In this section we introduce the methodology, testbench constructs and assertion technology used in constructing a constrained random verification environment.

An important aspect of VCS methodology is the ability to create a verification environment in a way that completely separates the testbench from the design under test (DUT). The connection to the DUT from the testbench is through port and boundary connections, thus avoiding unintended, unstructured access to the DUT. Creating this type of structure for the testbench reduces the number of errors and reduces the amount of time it takes to debug the testbench code and the design code.

The SystemVerilog program block facilitates this separation. A SystemVerilog program block is similar to a module in Verilog, but is specifically intended to distinguish between the testbench and the design code.

Components and verification-specific code blocks such as constraint blocks, functional coverage code, random data variables, and so on, are referenced within program blocks.

Using SystemVerilog program blocks in this way helps to avoid race conditions that are inherent in simulation tests.

Figure 2-1 shows a typical design hierarchy. The top-level shell contains the clock generator, which generates the clock for both the testbench and the DUT. The DUT can be Verilog, SystemVerilog or VHDL. The testbench contains the program block, which has additional verification features.

Figure 2-1 Verification Hierarchy

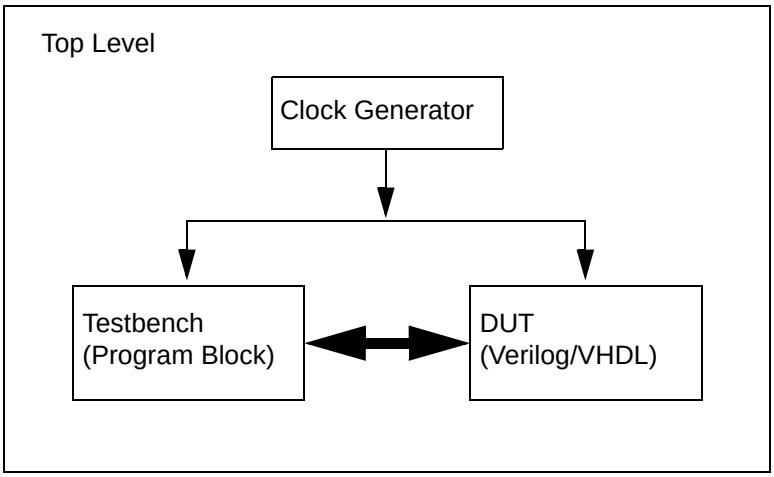
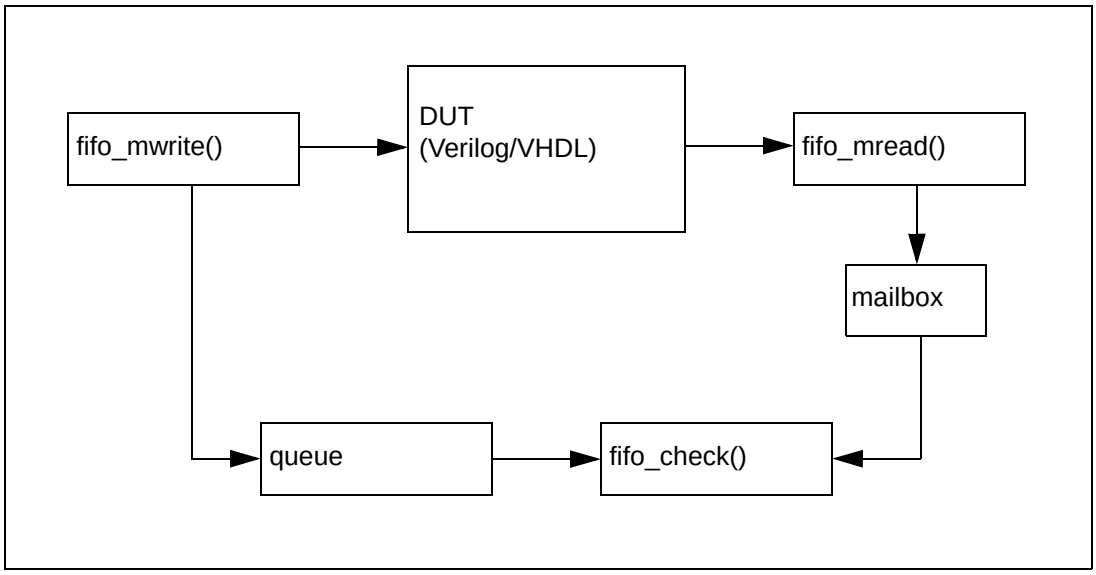


Figure 2-2 shows a high-level view of the verification architecture. The architecture contains three main tasks: `fifo_mwrite()`, `fifo_mread()`, and `fifo_check()`. The `fifo_mwrite()` task

generates random data to the DUT and places the data in a queue. The `fifo_mread()` task reads data from the DUT and sends the data to a mailbox for checking. The `fifo_check()` task compares the expected output in the queue with the actual output in the mailbox for correctness.

Figure 2-2 Verification Architecture



Testbench Files for the FIFO Example

Links to the testbench and design files are listed in “Testbench Files and Structure” on page 81.

Basic VCS Features and Constructs for Testbenches

This section describes some of the constructs added in VCS 2005.06 as part of SystemVerilog for Testbenches. These constructs are just a few of the many additional testbench constructs and SystemVerilog features in VCS.

Program Block

The program block is a SystemVerilog module that is specialized for verification purposes. Verification objects such as constraint blocks, functional coverage constructs, random signals, and so on, which comprise the test stimulus are referenced within program blocks. A program block contains the testbench components for a design. Usually the design is created in an HDL such as SystemVerilog, Verilog, or VHDL. The program block allows you to create a well defined boundary between the testbench code and the design code.

The abstraction and modeling constructs simplify the creation and maintenance of testbenches. The ability to instantiate and individually connect each instance of a program enables their use as generalized models.

The following is the `program` block for the FIFO module:

```
program fifo_test (
   rst_n, clk, data_in, data_out, push_req_n, pop_req_n,
   diag_n, empty, full, almost_empty, almost_full,
   half_full, error
);

   output logic rst_n;
   input clk;
   output logic [15:0] data_in;
```

```
   input [15:0] data_out;
   output logic push_req_n, pop_req_n, diag_n;
   input empty, full, almost_empty, almost_full, half_full,
      error;
   // declarations
   // instantiations
   // tasks...
   // initial block
endprogram
```

Since the program block drives data to the DUT input and reads data from the DUT output, the port directions in the `program` block are opposite to those of the DUT.

Constrained Random Stimulus

Constrained random stimulus generation is a powerful technique for use in verifying designs. You apply it by describing a set of constraints for the design, for which the VCS constraint solver finds a valid solution and generates tests bounded by the set of constraints. In addition to this method being much faster than creating tests manually, VCS is able to generate tests that check for unexpected behavior that you might not think of.

Assume the FIFO word length of the data field is 16 bits. The depth of the FIFO can be parameterized at the top level. The other field is called `data_rate` and it also is 16 bits wide. The data rate specifies how fast data is written to and read from the FIFO. A data rate of 8 means that the read or write operation occurs every eight clock cycles.

To generate random data for the `data` field and the `data_rate` field, you use the keyword `rand`. The `rand` keyword identifies these signals as randomizable data fields.

You must wrap the randomizable variables in a class. The full usage of a class is beyond the scope of this guide; it is sufficient to say that in this design the class is used as a construct to create random data.

The data rate can have a value between 1 and 1023. We do not want to test all combinations of data values and data rates, but only to check cases for low, medium and high data rates. Also, we want to generate data rates in the ranges 1 to 8, 128 to 135, and 512 to 519. We can use the `dist` construct to specify this as shown in the code below.

You can use the `constraint` construct to define the desired constraints. The following code defines two classes, one for the write operation and one for the read operation.

```
////////////////////////////////////////
// Definition: Random Write Data Class
////////////////////////////////////////
class random_write_data;
   rand logic [`WIDTH-1:0] data [`DEPTH];
   rand logic [15:0]  data_rate;

   constraint reasonable {
      data_rate > 0;
      data_rate dist  { [1:8] := 10 , [128:135] := 10,
         [512:519] :=1 } ;
   }

endclass

////////////////////////////////////////
// Definition: Random Read Data Class
////////////////////////////////////////
class random_read_data;
   rand logic [15:0]  data_rate;

   constraint reasonable {
      data_rate > 0;
```

```
      data_rate dist  { [1:8] := 10 , [128:135] := 10,
         [512:519] :=1 } ;
   }

endclass
```

On the `data_rate dist` line, the number after the `:=` specifies the weight. Items with higher distribution weights will be generated more often than items with a lower distribution weight. In this case, that means that data rates in the ranges with weights of 10 will be generated more often than data rates in the range with a weight of 1.

You can now use the `data_rate` variable to test different read and write data rates of the FIFO.

Verification Expect Checks

It is very useful to check for correct behavior of the DUT from within the testbench. VCS provides two mechanisms for doing this: the `assert` and `expect` constructs.

The following is the syntax for the `assert` and `expect` constructs:

```
[label:] assert (expression) [action block] [else statement]
[label:] expect (property spec) [action block]
   [else statement]
```

You can use the `assert` construct to check for behavior at the current simulation time. You can use the `expect` construct to check for behavior over many clock cycles.

For example, in the FIFO testbench, the `fifo_reset_check()` task checks to see that the outputs of the DUT are reset correctly to known values within a specified amount of time:

```
task fifo_reset_check();
   $write("FIFO_RESET_CHECK: Task reset_check started \n");
   //all reset signals must be at proper value
   E1: expect(@(posedge clk) ##[0:2]
      fifo_test_top.dut.empty === 1'b1);
   A1: assert(fifo_test_top.dut.almost_empty  == 1'b1);
   A2: assert(fifo_test_top.dut.half_full  == 1'b0);
   A3: assert(fifo_test_top.dut.almost_full  == 1'b0);
   A4: assert(fifo_test_top.dut.full  == 1'b0);
   A5: assert(fifo_test_top.dut.error  == 1'b0);
endtask
```

The `fifo_reset_check()` task first checks that `empty` becomes asserted within the first two clock cycles. If `empty` does not become asserted within two cycles, VCS will issue the following verification error:

```
Error: "fifo_test.v", 94:
fifo_test_top.test.fifo_reset_check.E1: at time 350
```

All verification checks should have labels for ease of debugging. The `E1: expect` statement blocks the task from executing the next line of code until the check is completed successfully or an error message is issued. The next five `A*` lines of code immediately check that the other output signals are properly set by the DUT. If they are not set properly, VCS issues a verification error.

Special Synchronization Constructs and Features

SystemVerilog provides advanced synchronization and concurrency control mechanisms through several constructs. You can use these features to develop test cases and checkers that mimic the true function of the design. For example, semaphores are constructs that

are used in bus arbitration mechanisms. Another construct, a mailbox, is used as a means for passing data and information between running processes.

The Mailbox Construct

The mailbox is a communication mechanism for passing messages and exchanging data between processes. In the FIFO example, we use the `mailbox` construct to pass data from the FIFO read task to the checker routine. Conceptually, the mailbox acts like a FIFO, and ensures race-free communication between processes. It also manages access to the data by various processes.

To use a mailbox, first you must instantiate and initialize the mailbox as follows:

```
mailbox mbox = new;
```

Next, in the `fifo_read()` task, you wait until there is data in the FIFO, then pop a word off of the FIFO and put it in the mailbox as follows:

```
while (empty) @(posedge clk);
   pop_req_n <= 1'b0;
   mbox.put(data_out);
   @(posedge clk);
   pop_req_n <= 1'b1;
```

The mailbox has some built-in methods; we are using the built-in `put()` method to place data into the mailbox.

Next, the `fifo_check()` task reads from the mailbox, and checks the expected data against the actual data. The statement that reads data from the mailbox is:

```
mbox.get(tempOut);
```

Queues and Arrays in SystemVerilog

A queue is a data type that is used to store sequences of data variables. The queue size can vary, by addition and deletion of elements. All stored elements can be accessed, and data elements can be inserted at the beginning of or removed from the end of a SystemVerilog queue.

A queue is similar to an unpacked array. As in an array, you can access the elements of a queue using an index. You can concatenate or slice the elements, and compare elements using comparison operators. The FIFO example uses a queue to store all the data generated from the `fifo_write()` task. The `fifo_check()` task reads from the queue for checking against the input data. The declaration of the queue is as follows, where the name of the queue is `cdata`:

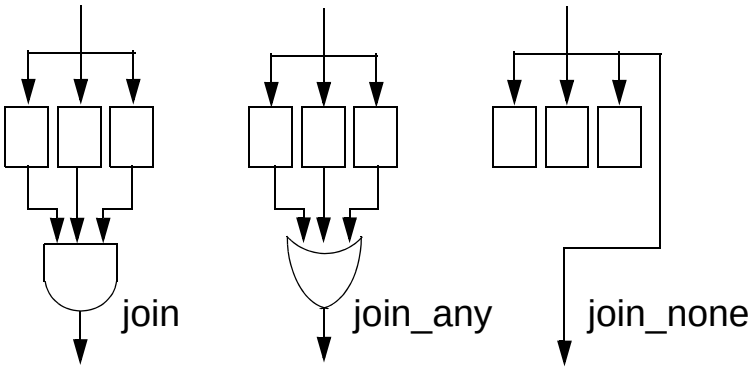
```
logic [‘WIDTH-1:0] cdata[$];
```

The `fifo_write()` code that uses the queue is as follows:

```
push_req_n <= 1'b0;
cdata.push_back(WRITE_DATA.data[j]);
data_in <= WRITE_DATA.data[j];
@(posedge clk);
push_req_n <= 1'b1;
```

The first line of code asserts the push control line of the FIFO. The second line puts the data generated into the queue. The generated data (`WRITE_DATA`) is described in the following sections.

Figure 2-3 Process and Thread Control Through Fork and Join



In the FIFO testbench, there is a FIFO checker that should always be active and waiting for data. The following code shows the `fifo_check()` task:

```
task fifo_check();
   logic [WIDTH-1:0] tempIn;
   logic [WIDTH-1:0] tempOut;
   while (1)
      begin
         mbox.get(tempOut);
         tempIn = cdata.pop_front();
         assert(tempIn == tempOut);
      end
endtask
```

When `fifo_check()` is called, it will continuously wait for data from the mailbox (which implies data was read from the FIFO), and then check the data against the expected output.

The `fifo_check()` task is called using the `fork...join_none` construct in the initial block of the program code, which will start the while loop, and then continue with the next line of code.

```
program fifo_test(....);
```

```
   // declarations
   // tasks
   //...
   initial begin : main_prog
      // instantiation of objects
      // ...
      fork
         fifo_check();
      join_none
      ...
      //
   end : main_prog
endprogram : fifo_test
```

Assertions

Assertions help designers and verification teams to catch bugs faster. With VCS, you can easily add assertions into your design to check for correct design behavior. You can write sophisticated assertions from scratch, or you can use the built-in Checker Library. VCS has a rich set of checkers in its library that you can easily instantiate in your design to identify bugs.

The checks in the library include:

- one hot/cold
- mutual exclusion
- even/odd parity
- handshake
- overflow
- underflow

- window
- arbiter
- many others

For a full description of the Checker Library, see the *SystemVerilog Checker Library Reference Manual*.

In addition to the Checker Library being pre-verified and ready to be instantiated, another advantage of using the Checker Library is that it has comprehensive assertion coverage built into each checker. It includes corner case coverage and range coverage. The checkers not only find bugs, but also report on coverage holes in your design.

For the FIFO design example, the FIFO word size is 16 bits with a depth of 128. We can use a FIFO assertion checker from the library. Within the checker, there is a useful category value for enabling and disabling assertions with the same category value.

You can instantiate the FIFO assertion directly into either the Verilog or the VHDL design. Alternatively, you can put the assertion into a separate file using the `bind` construct. This works for both Verilog and VHDL blocks. Here is how you place a FIFO assertion checker into the sample FIFO block design, with category level of 10:

In Verilog:

```
assert_fifo #(.depth(128), .elem_sz(16),
   .coverage_level_1(31),
   .coverage_level_2(0),
   .coverage_level_3(31) )
SVA_FIFO_inst (clk, rst_n, !push_req_n, data_in,
!pop_req_n, data_out);
```

In VHDL:

```
SVA_FIFO_inst : assert_fifo
generic map (depth => 128, elem_sz => 16,
   coverage_level_1 => 31,
   coverage_level_2 => 0,
   coverage_level_3 => 31)
port map(clk => TOP_CLOCK, reset_n => rst_n, enq => push_req,
   enq_data => data_in, deq => pop_req,
   deq_data => data_out);
```

The FIFO checker checks for the following failure modes: assertion `assert_fifo_overflow` reports a failure when enqueue is issued while the FIFO is full.

- Assertion `assert_fifo_underflow` reports a failure when the dequeue is issued, and the FIFO is empty at that time (and no simultaneous enqueue with pass_thru is enabled).
- Assertion `assert_fifo_value_chk` reports `(sva_v_q[sva_v_head_ptr] == deq_data))` as the failing expression when there is a dequeue, the FIFO is not empty, and the data does not correspond to the expected value. It reports `(enq_data == deq_data)` as the failing expression if there is a dequeue, and pass_thru is enabled (=1). Otherwise, if there is a dequeue on an empty FIFO it is a failure.
- Assertion `assert_fifo_hi_water_chk` reports a failure if the FIFO is filled above the high-water mark.

The assertion coverage that is reported is as follows:

- `cov_level_1_0` indicates there was an enqueue operation
- `cov_level_1_1` indicates there was a dequeue operation
- `cov_level_1_2` indicates there were simultaneous enqueue and dequeue operations

- `cov_level_1_3` indicates there was an enqueue followed eventually by a dequeue
- `cov_level_2_0` reports which FIFO fill levels were reached at least once
- `cov_level_3_0` indicates the high water mark was reached on an enqueue
- `cov_level_3_1` indicates there were simultaneous enqueue and dequeue operations on an empty queue
- `cov_level_3_2` indicates there were simultaneous enqueue and dequeue operations on a full queue
- `cov_level_3_3` indicates that empty condition was reached on dequeue
- `cov_level_3_4` indicates that full condition was reached on enqueue

Comprehensive Code Coverage

VCS has built-in comprehensive code coverage for both Verilog and VHDL designs. For full documentation, see the *VCS Coverage Metrics User Guide*.

The types of coverage are as follows:

- Line coverage reports which lines, statements, and blocks for any instance or module of the design were exercised during simulation.
- Condition coverage monitors values taken on by boolean and bitwise expressions, such as conditional expressions in if statements, or conditional concurrent signal assignments.
- Toggle coverage reports whether signals and signal bits had 0->1 and 1->0 transitions. A signal is considered fully covered if, and only if, it toggled in both directions: 0->1 and 1->0.
- Finite state machine (FSM) coverage recognizes some portion of sequential logic as an FSM, and reports which FSM states and which state transitions were executed. FSM coverage can determine which parts of the design are implemented as FSMs, and give specific information that other kinds of coverage do not provide regarding all possible sequences of state transitions.

In order to enable code coverage in the simulation, the following compile and runtime arguments are used with the `-cm` option:

- `-cm` *`coverage-type`* specifies the type of coverage to collect.

The *coverage-type* options are:

Option	Enables
line	statement (line) coverage
tgl	toggle coverage
cond	condition coverage
fsm	FSM coverage

Any combination of coverage types can be enabled simultaneously, as in the following for example:

```
-cm cond+tgl+line+fsm
```

Functional Coverage

Functional coverage provides a metric for measuring the progress of design verification, based on the design specification and the functional test plan. Functional coverage is used in conjunction with code coverage. The main purpose of functional coverage is to guide the verification process by identifying tested and untested areas of design and ensuring that corner cases are dealt with. The built-in functional coverage mechanisms in VCS support:

- Variable and expression coverage, as well as cross coverage
- Automatic and user-defined coverage bins
- Filtering conditions (illegal values, values to ignore, and so on)
- Automatic coverage sample triggering by events and sequences

The FIFO example should test all different combinations of read and write data rates. Because the data rate can take 1024 different values, the number of possible combinations of data rates is enormous. Instead of exhaustively testing all combinations, we define low, medium and high data rates, and test for their combinations. In the following code snippet, a LOW data rate is any `data_rate` between 1 and 127, a MED data read is any `data_rate` between 128 and 511, and a high data rate is a `data_rate` between 512 and 1023. In order to test for all combinations of LOW, MED and HIGH data rates we use the `cross` construct to specify the cross of RD data rates and WR data rates. This results in only nine different combinations:

```
covergroup Cvr;
    RD: coverpoint READ_DATA.data_rate {
        bins LOW = {[1:127]};
        bins MED = {[128:511]};
        bins HIGH = {[512:1023]};
    }
    WD: coverpoint WRITE_DATA.data_rate {
        bins LOW = {[1:127]};
        bins MED = {[128:511]};
        bins HIGH = {[512:1023]};
    }
    RDxWD: cross RD,WD;
endgroup
```

Running the Test Case

In the quick start directory there is a makefile, to run the Verilog version of the FIFO. Here is the compile command:

```
vcs +vc -cm line -sverilog -debug fifo_test.v \
    fifo.test_top.v DW_fifo_s1_sf.v DW_fifoctl_s1_sf.v \
    DW_ram_r_w_s_dff.v +define+ASSERT_ON+COVER_ON -y \
    $(VCS_HOME)/packages/sva/ +libext+.sv \
    +incdir+$(VCS_HOME)/packages/sva
```

Note:

Location of the makefile is specified in “Testbench Files and Structure” on page 81.

`-cm line` enables line coverage.The `-sverilog` option enables SystemVerilog compilation. The `-debug` option enables interactive debug.

To run the simulation for 50 tests use the “+NUM” options to simv:

```
simv -cm line +NUM+50
```

To generate assertion coverage report use the assertCovReport utility:

```
assertCovReport
```

To generate testbench coverage, do the following step:

```
vcs -cov_text_report fifo_test.db
```

To generate code coverage reports, do the following step:

```
urg -dir fifo_test.db
```

For more information on URG options, see the Coverage Technology Reference Manual.

To run the VHDL version of the FIFO, here is the compile command:

```
vlogan -q -sverilog fifo_test.v
   vlogan -q +vc +define+ASSERT_ON+COVER_ON -sverilog \
   $(VCS_HOME)/packages/sva/assert_fifo.v \
   -y $(VCS_HOME)/packages/sva/ +libext+.v \
   +incdir+$(VCS_HOME)/packages/sva
   vhdlan -q DWpackages.vhd
   vhdlan -q DW_fifo_s1_sf
   vhdlan -q DW_fifo_s1_sf_sim.vhd
   vhdlan -q fifo.test_top.vhd
```

Here is the elaboration step:

```
vcs -cm line tb -debug -mhdl
```

To run the simulation for 50 tests, use the `+NUM` options to `simv`:

```
simv -cm line -R -verilog "+NUM+50"
```

To generate assertion coverage report use the `assertCovReport` utility:

```
assertCovReport -cm_assert_dir simv.db.dir/simv.o.vdb/
```

To generate testbench coverage do the following step:

```
vcs -cov_text_report fifo_test.db
```

To generate code coverage reports, do the following step:

```
urg -dir fifo_test.db
```

FIFO Simulation Results

The command line below simulates the FIFO with five sets of data:

```
simv -cm line +NUM+5
```

The testbench reports the `data_rate` for each call of the tasks `fifo_mwrite()` and `fifo_mread()`.

The following is the output from the simulator:

```
Chronologic VCS simulator copyright 1991-2005 Contains Synopsys proprietary
information.
Compiler version X-2005.06; Runtime version X-2005.06; Jul 22 14:13 2005

 VCS Coverage Metrics Release X-2005.06 Copyright (c) 2003 Synopsys, Inc
FIFO_RESET: Task reset_fifo started
FIFO_RESET_CHECK: Task reset_check started
FIFO_CHECK: Task check started
FIFO_MWRITE: Sending 128 words with data_rate of    1 to fifo at:           350
FIFO_MREAD: Reading 128 words with data_rate of    1 from fifo at:           350
FIFO_MWRITE: Sending 128 words with data_rate of  135 to fifo at:         26050
FIFO_MREAD:  Reading 128 words with data_rate of  513 from fifo at:         26050
FIFO_MWRITE: Sending 128 words with data_rate of  134 to fifo at:       6605250
FIFO_MREAD:  Reading 128 words with data_rate of  135 from fifo at:       6605250
FIFO_MWRITE: Sending 128 words with data_rate of    7 to fifo at:       8346050
FIFO_MREAD:  Reading 128 words with data_rate of    3 from fifo at:       8346050
FIFO_MWRITE: Sending 128 words with data_rate of    1 to fifo at:       8448550
FIFO_MREAD:  Reading 128 words with data_rate of  135 from fifo at:       8448550
FIFO_MWRITE: Sending 128 words with data_rate of  515 to fifo at:      10189350
FIFO_MREAD:  Reading 128 words with data_rate of  130 from fifo at:      10189350
```

At the end of the simulation, VCS reports on the assertion coverage for the assert_fifo checker:

```
 "/VCS_MX/Linux/packages/sva/assert_fifo.sv", 565:
fifo_test_top.SVA_FIFO_inst.cov_level_1_0.cover_number_of_enqs, 167943
attempts, 768 match, 0 vacuous match

"/VCS_MX/Linux/packages/sva/assert_fifo.sv", 572:
fifo_test_top.SVA_FIFO_inst.cov_level_1_1.cover_number_of_deqs, 167943
attempts, 768 match, 0 vacuous match

"/VCS_MX/Linux/packages/sva/assert_fifo.sv", 579:
fifo_test_top.SVA_FIFO_inst.cov_level_1_2.cover_simultaneous_enq_deq, 167943
attempts, 1 match, 0 vacuous match

"/VCS_MX/Linux/packages/sva/assert_fifo.sv", 587:
fifo_test_top.SVA_FIFO_inst.cov_level_1_3.cover_enq_followed_eventually_by_deq
, 167943 attempts, 547 match, 0 vacuous match

"VCS_MX/Linux/packages/sva/assert_fifo.sv", 715:

fifo_test_top.SVA_FIFO_inst.cov_level_3_1.cover_simultaneous_enq_deq_when_empt
y, 167943 attempts, 0 match, 0 vacuous match

"VCS_MX/Linux/packages/sva/assert_fifo.sv", 724:

fifo_test_top.SVA_FIFO_inst.cov_level_3_2.cover_simultaneous_enq_deq_when_full
, 167943 attempts, 0 match, 0 vacuous match

"VCS_MX/Linux/packages/sva/assert_fifo.sv", 733:
 fifo_test_top.SVA_FIFO_inst.cov_level_3_3.cover_number_of_empty, 167943
attempts, 513 match, 0 vacuous match

"/VCS_MX/Linux/packages/sva/assert_fifo.sv", 744:
 fifo_test_top.SVA_FIFO_inst.cov_level_3_4.cover_number_of_full, 167943
attempts, 0 match, 0 vacuous match
```

You can use the assertCovReport utility to generate an HTML report of the assertion result, see Figure 2-4 on page 32.

The file: "simv.vdb/reports/report.index.html" includes the HTML report. As you can see there are eight cover properties, five of which are covered and three are not covered by this test. The three cover properties not covered are:

```
cover_simultaneous_enq_deq_when_empty
cover_simultaneous_enq_deq_when_full
```

```
cover_number_of_full
```

If you run the test longer, (250 tests),

```
simv -cm line +NUM+250
```

only two cover properties are not covered:

```
cover_simultaneous_enq_deq_when_empty
cover_simultaneous_enq_deq_when_full
```

The cover property `cover_simultaneous_enq_deq_when_empty` is not allowed in this version of the FIFO and results in an error condition.

The cover property `cover_simultaneous_enq_deq_when_full` is not an error condition, but because 128 words are being written into the FIFO, this condition will not be hit. This can be rectified by writing 128 words, and then reading and writing at the same time:

```
fifo_mwrite(1);
fork
   fifo_mwrite(1);
   fifo_mread(1);
   @(posedge clk) fifoCvr.sample();
join
```

The first line writes 128 words into the FIFO with a `data_rate` of 1. The next five lines of code perform a simultaneous read and write, ensuring the cover property `cover_simultaneous_enq_deq_when_full` is hit.

You can now look at the testbench coverage for 500 tests using this command:

```
vcs -cov_text_report fifo_test.db
```

The following is the generated report:

```
Functional Coverage Report

Coverage Summary
    Number of coverage types: 1
    Number of coverage Instances: 1
    Cumulative coverage: 100.00
    Instance coverage: 0.00

Coverage Groups Report

Coverage group                      # inst    weight    cum cov.
=================================================================
Cvr                                 1         1         100.00
=================================================================

=================================================================
= Cumulative report for Cvr
=================================================================

Summary:
    Coverage: 100.00
    Goal: 90

Coverpoint                  Coverage       Goal           Weight
=================================================================
RD                          100.00         90             1

WD                          100.00         90             1

=================================================================
Cross                       Coverage       Goal           Weight
=================================================================
RDxWD                        100.00         90             1

=================================================================

Cross Coverage report
CoverageGroup: Cvr
    Cross: RDxWD
```

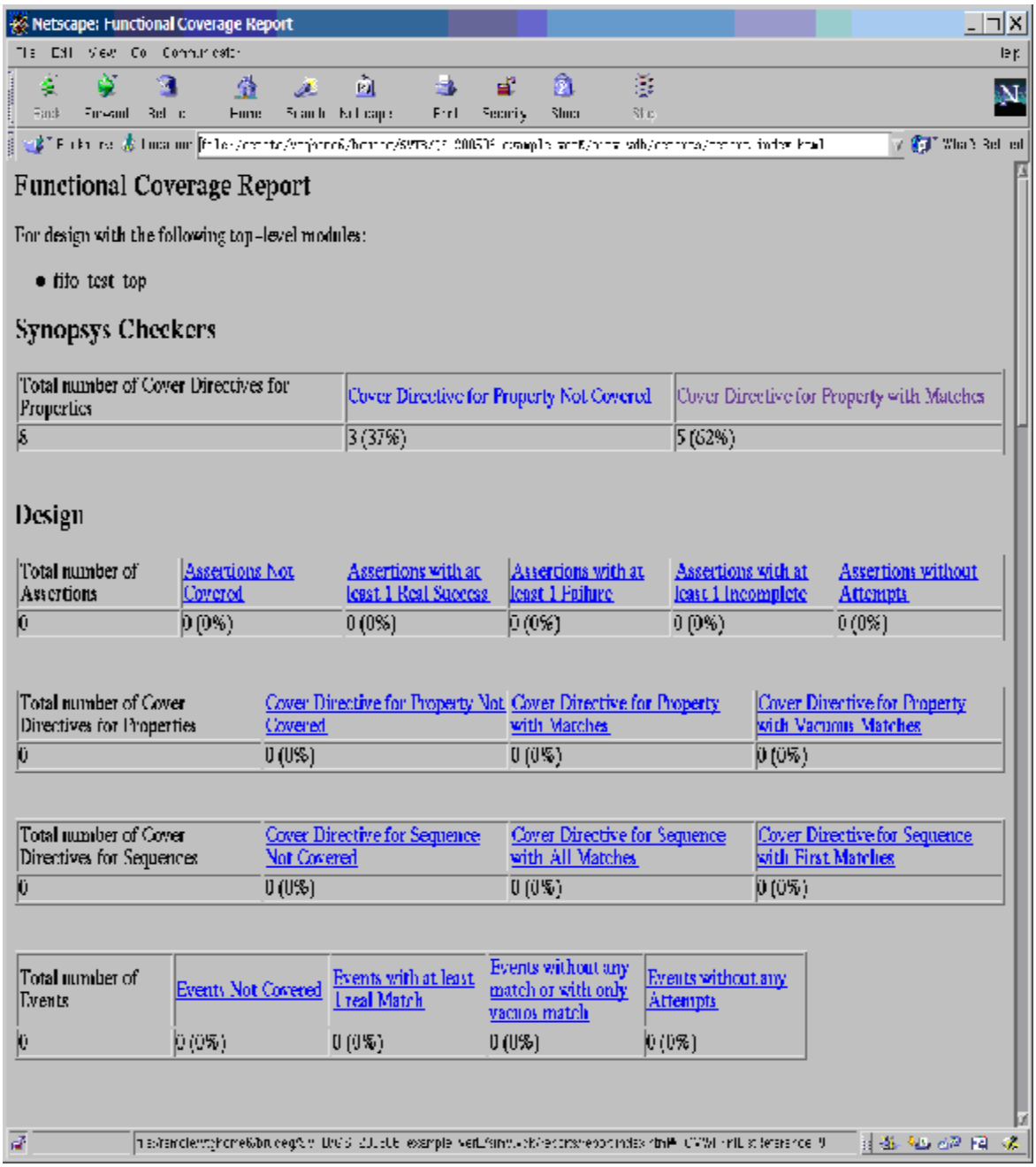
```
Summary
    Coverage: 100.00
    Goal: 90
    Number of Coverpoints Crossed: 2
    Coverpoints Crossed: RD WD
    Number of Expected Cross Bins: 9
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 9

    Automatically Generated Cross Bins

    RD              WD              # hits          at least
    ========================================================
    HIGH            HIGH            2               1
    HIGH            LOW             11              1
    HIGH            MED             7               1
    LOW             HIGH            7               1
    LOW             LOW             105             1
    LOW             MED             104             1
    MED             HIGH            10              1
    MED             LOW             124             1
    MED             MED             132             1
    ========================================================


Coverpoint Coverage report
CoverageGroup: Cvr
    Coverpoint: RD
Summary
    Coverage: 100.00
    Goal: 90
    Number of User Defined Bins: 3
    Number of Automatically Generated Bins: 0
    Number of User Defined Transitions: 0

    User Defined Bins

    Bin                                         # hits    at least
    ============================================================
    HIGH                                        20        1
    LOW                                         216       1
    MED                                         266       1
    ============================================================

Coverpoint Coverage report
CoverageGroup: Cvr
    Coverpoint: WD
Summary
```

```
    Coverage: 100.00
    Goal: 90
    Number of User Defined Bins: 3
    Number of Automatically Generated Bins: 0
    Number of User Defined Transitions: 0

    User Defined Bins

    Bin                                         # hits    at least
   ============================================================
    HIGH                                        19        1
    LOW                                         240       1
    MED                                         243       1
    ============================================================
Database Files Included in This Report:

    fifo_test.db


Test Bench Runs Included in This Report:

Test Bench   Simulation Time      Random Seed     Random48 Seed
================================================================
fifo_test.db::fifo_test 866446150000   0               0
================================================================
```

The code coverage reports are in the directory "simv.cm/reports".

Figure 2-4 shows an example of a functional coverage report.

Figure 2-4 Functional Coverage Report



Discovery Visual Environment

Starting with Version 2005.06, VCS comes with a comprehensive built-in graphical user interface named Discovery Visual Environment (DVE). DVE supports graphical debugging of mixed language designs including Verilog, VHDL, and SystemVerilog. DVE is an intuitive environment that also supports assertion debugging. Detailed documentation can be found in the *Discovery Visual Environment User Guide*.

Summary

The built-in verification features of VCS can be used to quickly ramp up new users who are unfamiliar with the standard languages such as SystemVerilog Hardware Design and Verification Language (HDVL). VCS incorporates testbench, assertion, and coverage capabilities needed for design and verification engineering tasks. With these technologies, VCS creates the framework and foundation for building verification environments at all levels, from module to cluster of modules, to chip and system level verification.

In the following chapters we introduce more advanced built-in VCS features such as interfaces, object-oriented programming concepts, and structured testbench development.

3

Using Advanced Features in VCS

In chapter 2 we used a FIFO design block example to discuss the basic testbench constructs supported by VCS, and created our verification environment. We explained the basic feature set for constrained random stimulus generation and functional coverage along with assertions, code coverage and debug features. The `program` block construct, process control mechanisms, queues and arrays as well as mailbox elements were also reviewed in more detail to provide support for testbench implementation for the FIFO.

In this chapter we introduce object-oriented programming concepts with high-level data abstractions and interfaces, and show the reader how to use these advanced SystemVerilog constructs as well as SystemC models to build a structured and layered testbench for the FIFO design example (see “The FIFO Design Block” on page 79”).

Verification Environment

In the previous chapter we introduced the program block as the basic building block that contains the SystemVerilog testbench. A program block allows creation of a well defined boundary between the test code and the design code, which reduces the time to debug the design and test code. The abstraction and modeling constructs simplify the creation and maintenance of testbenches. The ability to instantiate and individually connect each instance of a program enables their use as generalized models.

A similar design hierarchy is used in this chapter as well. In Figure 3-1, the top-level shell contains the clock generator that is passed to the testbench and the DUT. The DUT can be Verilog, SystemVerilog or VHDL. The program block incorporates the structured testbench environment and components.

Figure 3-1 Verification Hierarchy

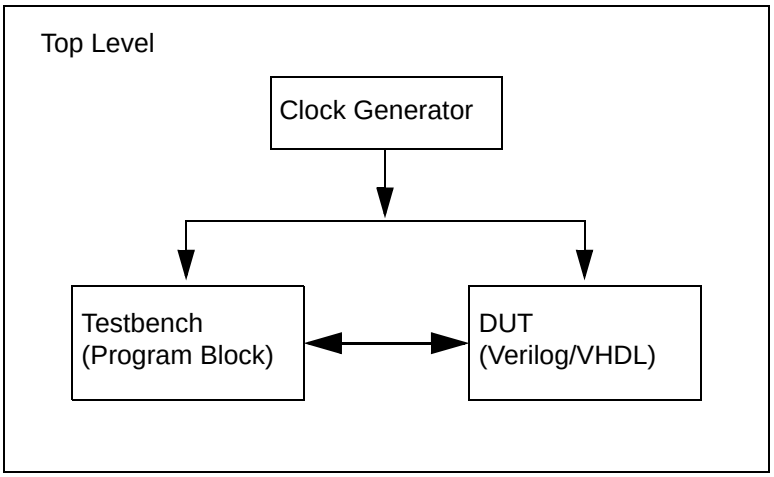
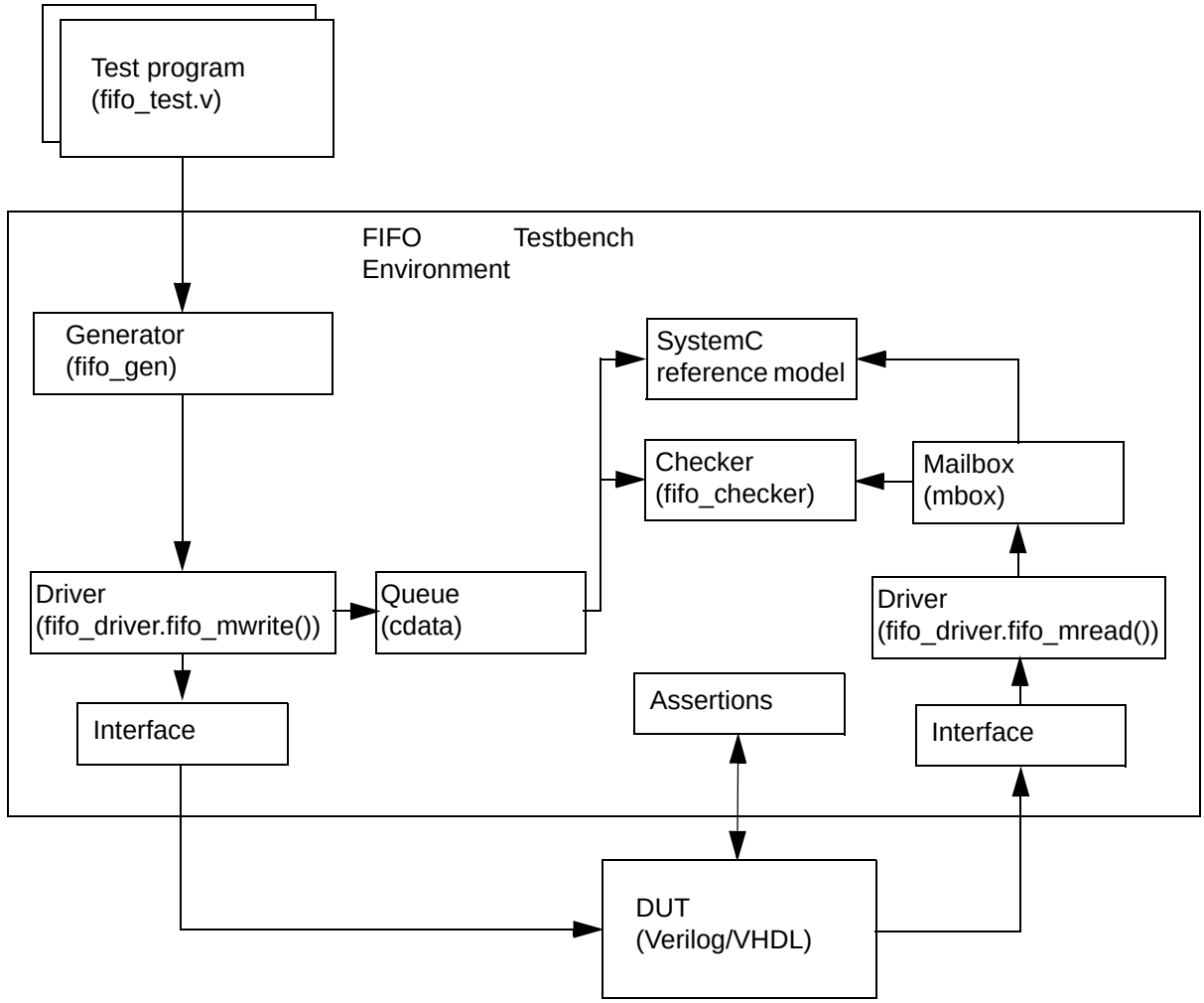


Figure 3-2 is a high-level view of the verification architecture of the DUT and testbench.

Figure 3-2 FIFO Verification Architecture



The architecture uses a hierarchy of classes to build a structured testbench that enables reusability and scalability. The top level program contains an environment object, which in turn contains generator driver and checker objects. We also include a connection to a SystemC reference model to demonstrate the VCS features supporting SystemC models.

Advanced Constructs for Testbench and Design in VCS

This section describes a few more of the advanced constructs for SystemVerilog for Testbench development, and their usage models in a structured testbench verification environment. There are many additional constructs and SystemVerilog features. For details about all of the SystemVerilog capabilities in VCS, refer to the online HTML documentation system.

In this chapter we use the basic structure developed for the testbench in the previous chapter, to show how easy it is to use the advanced object-oriented constructs in VCS to create a scalable and reusable verification environment.

Basic SystemVerilog Object-Oriented Constructs for Testbench

VCS enables data abstraction with class data types. Classes as self-contained components form the foundation of object-oriented programming. Object-oriented programming is different from the procedural programming with which people are familiar. There are three principles behind objects; *encapsulation*, *inheritance*, and *polymorphism*. Those principles are discussed in the following sections.

Objects, Declaration and Instantiation

An object is an instance of a defined and declared class.

```
Sensor TempSense;  //declare a variable of class Sensor
TempSense = new;  // initialize variable, object of Sensor
```

Instantiation is done by calling the constructor `new` for the object. Objects are stored dynamically: memory is allocated at the time of instantiation for that object and not at the time of definition and declaration.

Encapsulation and Data Hiding

The `class` data type encapsulates all attributes of coherent data in a system—the members that store data items and the member functions and tasks that manipulate the data and communicate with the rest of the system. Through modularization, classes efficiently manage the complexity of large programs. Once declared, a class can be instantiated and dynamically created as an object. An object is a container of variables and methods operating on them. We operate on the object by calling its methods.

Classes can contain public part and private parts. Members—variables and methods—declared in the private part of class are hidden from the rest of the system, using protected or local attributes. The public part is the section that connects the objects to the rest of the environment. This lets designers and verification engineers solve the details of their verification environment piece by piece, in a modular fashion. As each section is completed, it can be set aside with all its details of implementation from the rest of the system, so that only the public section can affect – or be affected by other segments. The following shows a simple data class for a sensor.

```
class Sensor;
   string model;
   integer address;
   bit state;  // 0 off, 1 on
   integer value;
   // constructor, initialization sub-routine
```

```
   function new();
      model = "Normal";
      value = 0;
      state = 0;
   endfunction : new
   //  methods, accessing the variable members
   task start_sensor();
      state = 1;
   endtask
   function bit current_state();
      current_state = state;
   endfunction
endclass: Sensor
```

In this example we have created:

- A constructor function, `new()`
- A task, `start_sensor()`, that starts the sensor
- A function, `current_state()`, that reports the state of the sensor

You can add other methods that operate on the sensor in this class.

Inheritance and Polymorphism

In order to be able to build reusable components, object-oriented technology provides a mechanism for creating hierarchies of classes through derived classes. The process of deriving classes is called inheritance. Inheritance is the mechanism that allows a class B to inherit properties of a class A. We say “B extends A”. Objects of class B thus have access to attributes and methods of class A without the need to redefine those attributes and methods in class B.

If class B inherits from class A, then A is called a superclass of B, and B is called a subclass of A. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behavior as objects of the superclass.

Polymorphism, meaning “having multiple forms,” means that an entity can have different meanings or different usages in different contexts.

In a subclass, explicit calls to the methods of a superclass are done by using `super`. For example, `super.new()` in the subclass constructor calls the superclass constructor and has to be the first call in the subclass constructor definition. In the following example, Adult class (the subclass) extends from Person class (the superclass).

```
class Person;
   //data or class properties
   string name;
   integer age;
   string gender;
   //initialization
   function new();
      Name = "";
      Age = 0;
   endfunction;
   virtual task speak();
      $display("This is a person \n");
   endtask
endclass

class Adult extends Person;
   function new();
      super.new();
   endfunction;

  virtual task speak();
```

```
   $display(" my name is %s \n",name);
   endtask
endclass: Adult
```

Note the usage of `virtual` attribute for the task `speak()`. Virtual methods are basic polymorphic constructs. Virtual methods provide prototypes for subroutines and override methods in all the base classes, whereas normal methods only override methods in their classes and descendants. Subclasses override the virtual methods by following the prototype exactly; the number of arguments, return type, and encapsulation of all versions of a virtual method are identical in all subclasses.

If we were to define a generic Person class type, knowing that the class will be extended to define the characteristics of the functions such as `speak()`, the function must be declared as virtual and then its behavior defined in derived classes.

```
class Person;
   //data or class properties
   ...
   virtual task speak();
   endtask
endclass: Person
```

Now we can define an adult person and by extension of the Person class. Note that in this case we are overriding the function in the base class.

```
class Adult extends Person;
   ...
   virtual task speak();
      $display(" my name is %s \n",name);
   endtask
endclass: Adult
```

We can also extend class Person to provide functionality for another entity, for example, Child:

```
class Child extends Person;
   virtual task speak();
      $display(“ bah-bah-bah %s \n”,name);
   endtask
endclass: Child
```

In addition to its usage as a means of representing both the common features and specialized aspects of different data structures and components, class derivation is also a tool for modularization. For example, a verification engineer can define a derived class based on an existing class library that was produced specifically for verification tasks.

The object-oriented (OO) class features provided in SystemVerilog and supported in VCS provide the inheritance mechanism to build reusable and modular verification environments to model complex systems. Using the OO elements of VCS provides the foundation for a layered architecture approach to implementing the verification environment.

In this chapter we review a simple object-based structure as an introduction to OO concepts.

Classes and objects benefit users in various ways; above all they encourage the reuse of software components that are developed for specific tasks. Users will also reduce development time and risk, because systematic and modularized development using objects provides more resilience and reduces code size.

Another new construct supported in VCS, `interface`, provides a container for signals used in the design and verification environment.

The interface Construct and Signal Connectivity

VCS provides a new construct, `interface`, as a container that acts as bundle of wires, encapsulating signal definitions and synchronization, and allows ease of connectivity between testbench and design components. The `interface` construct can be instantiated like a module. It provides flexibility through instantiation in a module or a program. The best approach to declare signals for connectivity is to declare interface signals as `wire` in SystemVerilog.

```
interface fifo_intf(input clk);
   parameter WIDTH = 16;
   wire rst_n;
   wire push_req_n;
   ..
   wire [WIDTH-1: 0] data_out;
 endinterface : fifo_intf
```

Once an interface container has been defined, variables of that interface type can be declared and instantiated.

```
 fifo_intf intfFifo(clk);
   // declaration and instantiation of of type fifo_intf
```

modports

Directional information for module ports is provided by using the `modport` construct in SystemVerilog. In a verification environment there are various views and uses for interface signals: some are driven, such as in driver transactors, and some are simply monitored, such as in monitor transactors. In order to compartmentalize these different views, modports are declared and defined for each of the transactor views.

```
interface fifo_intf(input clk);
```

```
   parameter WIDTH = 16;
   wire rst_n;
   wire push_req_n;
   ...
   wire [WIDTH-1: 0] data_out;
   modport Fifo_Driver(output rst_n;
      output push_req_n;
      ...
      input data_out);

   modport Fifo_Designer(input rst_n;
      input push_req_n;
      ...
      output data_out);
endinterface: fifo_intf
```

Now you can declare and instantiate the interface and pass the modport to the testbench:

```
 fifo_intf intfFifo(clk);
   // declaration and instantiation of type fifo_intf
   // pass modport to a program instantiation.
 fifo_test test(intfFifo.Fifo_Driver);
```

Note that in order to avoid duplication of effort and create a modular and flexible verification environment that allows extensibility, the actual task and function code for transactors is not defined inside the interface.

In the next sections we describe the usage of clocking blocks to allow synchronous sampling and driving of signals. This blocking removes any potential race conditions between the design and the verification testbench. Using such constructs in SystemVerilog eliminates unpredictability in the timing behavior of the connections between the testbench the design.

Virtual Interface

A virtual interface provides a mechanism for separating abstract models and test programs from the actual signals that make up the design, further promoting code reuse. For example, a network switch router has many identical ports that operate the same way. A single virtual interface declared inside a class that deals with these Ethernet signals can represent different interface instances at various times throughout the simulation. A virtual interface can be declared inside as a class property that can be initialized procedurally or by an argument to the class constructor.

In the following code example the class `fifo_driver` instantiates a virtual interface:

```
class fifo_driver
   ...
   virtual fifo_intf v_intf;  // full interface instance
   ...
endclass: fifo_driver
```

The instantiation of the interface can have several flavors in the testbench area, as in the examples below.

In a program block:

```
v_intf = interface_passed_to_program;
```

Or through the class constructor:

```
function new(virtual fifo_intf vIntf, ...);
   this.v_intf = vInft;
```

And with modport connections:

```
virtual fifo_intf.Fifo_Driver fifo_driver_if;
```

Similarly the assignment through the program block and class constructor is as follows:

Assignment to a virtual interface:

```
fifo_wr_if = interface_passed_to_program;
```

Through a class constructor:

```
function new(virtual fifo_intf.Fifo_Driver vIntfDR, ...);
   this.v_intf = vInftDR;
```

Referencing signals within an interface can be accessed using a relative hierarchical path name. For example:

```
...
initial
   intfFifo.rst_n <= 0;
   always @(posedge intfFifo.clk)
   $display("Cycle is %0 \n",cyc++);
...
```

Eliminating Race Conditions in Synchronous Designs

Clocking Blocks

One of the major contributions of SystemVerilog has been the introduction of a clocking block as a synchronization construct for signals between the testbench and the design. Usage of a clocking block inside interfaces helps remove race conditions between the two sides and handles signal delay differences between RTL and gate-level models. Clocking blocks used an in interface definition in conjunction with modports define proper directions for each modport and signals in it. The clocking block arranges signals that are synchronous to a particular clock and makes their timing explicit. The

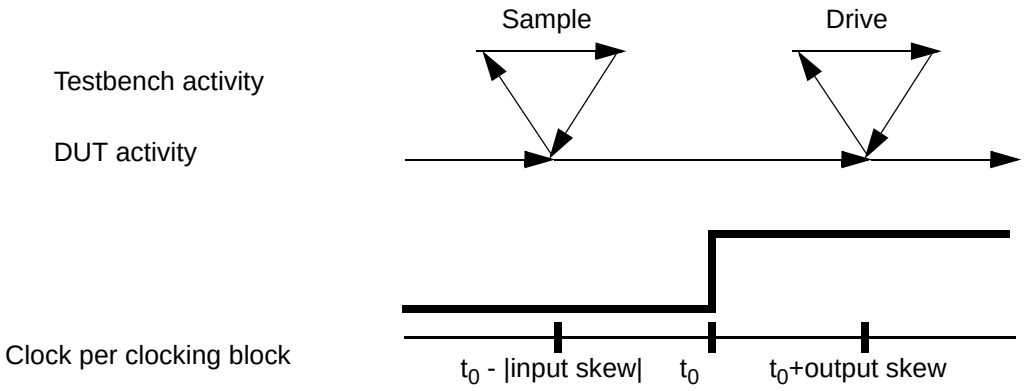
clocking block is key in allowing engineers to focus on test development rather than signals and transitions over time. Depending on the nature of the design and verification environment, there can be more than one clocking block.

An example of interface and clocking blocks follows.

```
interface fifo_intf(input clk,input clk2);
   ...
   parameter setup_t = 2ns;
   parameter hold_t = 3ns;
   default input #setup_t output #hold_t;
   clocking cb @(posedge clk);
      output rst_n;
      output push_req_n;
      ..
      input data_out;
   endclocking

   clocking cbDR @(posedge clk2);
      output rst_n;
      output push_req_n;
      ...
   endclocking
endinterface: fifo_intf
```

Signals within each clocking block are sampled and driven with respect to the specified clock edge, given appropriate setup and hold time. An interface can contain one or more clocking blocks.

Figure 3-3 Timing reference for sample and drive of synchronous signals



Modport and Clocking Block

In the interface definition, each modport has a reference to a clocking block. As discussed previously, the main transactors within the testbench program block connect to the signals through virtual interface modports to allow extensibility:

```
interface fifo_intf;
   modport fifo_Driver(clocking cbDR);
   modport fifo_Checker(clocking cbCheck)
   ...
endinterface: fifo_intf
```

Signals are then referenced with respect to interface modport names, and are synchronized with the appropriate clocking block timing:

```
v_intf.cb.push_req_n <= 1'b1;    //direct clocking block
fifo_wr_if.cbDR.push_req_n <= 1'b1;  //modport usage
```

Asynchronous Signals

The interface `modport` construct can be used to define and create asynchronous ports and signal connections. The signal is defined as an input or output in a modport declaration:

```
interface fifo_intf;
   modport fifo_Driver(clocking cbDR, output rst_n);
   modport fifo_Checker(clocking cbCheck)
   ...
endinterface: fifo_intf
```

In this case, the signal `rst_n` is defined as asynchronous with respect to the clock edges.

Testbench Structure for the FIFO

In this section we discuss the new testbench structure using classes for encapsulation of signals, data and transactions. We use an object hierarchy in a simple layered structure to introduce the use of more advanced object-oriented concepts. The testbench is comprised of data types that encapsulate signal connections (`interface`, `fifo_intf`), a fifo data class (`data_tr`), and an environment class (`fifo_env`). Within the environment class we incorporate classes for test configuration, a driver and checkers as well as coverage groups. The main program instantiates the global data object and environment object that run the test.

There are a few sample test files that help introduce various advanced topics such as using class extension to create new tests as well as incorporating a SystemC checker object to show the VCS SystemVerilog to SystemC connection features.

Interface

The FIFO interface encapsulates not only the signals that read and write from the FIFO, but also the access mechanisms of the FIFO.

This interface is named `fifo_intf` and it contains a single input clock named `clk` that is passed to the interface when it is instantiated at the top level of our environment.

The interface also contains a clocking block that defines the signal timing with respect to the positive edge of the clock. The clocking block is only used by the testbench, and is not referenced by the FIFO itself. The clocking block ensures proper driving and sampling from the testbench to the FIFO.

```
interface fifo_intf(input clk);
parameter WIDTH = 16;
   wire rst_n;
   wire push_req_n;
   wire pop_req_n;
   wire diag_n;
   wire [WIDTH-1 : 0] data_in;
   wire empty;
   wire almost_empty;
   wire half_full;
   wire almost_full;
   wire full;
   wire error;
   wire [WIDTH-1 : 0] data_out;

clocking cb @(posedge clk);
   output push_req_n;
   output pop_req_n;
   output diag_n;
   output data_in;
   input  empty;
   input  almost_empty;
   input  half_full;
```

```
   input  almost_full;
   input  full;
   input  error;
   input  data_out;
endclocking
modport tb (clocking cb, output rst_n);
modport dut( input  clk,  rst_n,  push_req_n,  pop_req_n,
   diag_n,data_in, output empty, almost_empty, half_full,
   almost_full, full, error, data_out);
endinterface
```

We have also defined two modports named `tb` and `dut`. These modports establish access mechanisms from the design and testbench perspective. From the design perspective, we can read the inputs directly and drive the outputs directly.

The `tb` modport establishes the access mechanism for the testbench. The testbench can only read and write from the FIFO through the clocking block, except for the `rst_n` signal. All reads and writes are synchronous with respect to the positive edge of the clock except for `rst_n`, which can be written to asynchronously.

The FIFO interface can be instantiated in the top level of my environment as shown below:

```
fifo_intf intf(clk);
```

The interface can be now passed to both the testbench `fifo_test` and to the design under test `DW_fifo_s1_sf_wrap`. We pass the modport version of the FIFO to its respective instance as shown below.

```
 //program instantiation
fifo_test test(intf.tb);

// dut instantiation
DW_fifo_s1_sf_wrap #(WIDTH,  DEPTH,  ae_level,  af_level,
```

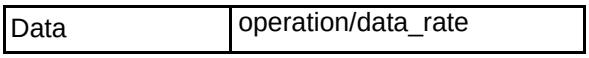
```
   err_mode, rst_mode)
dut (intf.dut);
```

Using modports ensures that the proper driving and sampling semantics are used. Trying to drive the `pop_req_n` signal asynchronously results in an error. This prevents incorrect usage of the interface.

Data Class

Referring to the example FIFO used in the first chapter, let us assume our FIFO word length is 16 bits. You can parameterize the depth of the FIFO at the top level. We have chosen the other fields for push (write) or pop (read) data rates.

Figure 3-4 FIFO data format



The `data` field is 16 bits wide. The data rates for write and read are also 16 bits wide. In this example, we want the `data` field to contain values in which the two highest order bits are set. The code for these constraints looks like this:

```
class data_tr;
   rand logic [15:0] data;
   rand logic [15:0] wr_data_rate;
   rand logic [15:0] rd_data_rate;
   constraint reasonable {
      data [15:14] == 2'b11;
      wr_data_rate > 0;
      wr_data_rate dist { [1:8] := 10, [60:80] := 10 }
      rd_data_rate > 0;
      rd_data_rate dist { [1:8] := 10, [60:80] := 10 }
   }
```

```
endclass
```

Any data type can be declared as random using the `rand` keyword. Here we have declared the fields of FIFO data as random. We must encapsulate our random variables within a class, and we have named the class `data_tr`.

We also declare integer variables named `wr_data_rate` and `rd_data_rate` as `rand`, and constrain them with a distribution list. We will use these random variables to test different read/write data rates of the FIFO. For very slow data rates, delays have large values and for fast data rates delays have low values. Finally, we create 10 different random values. Here is what the output looks like:

wr_data_rate	**rd_data_rate**	**data**
20	12	c82b
28	24	c852
40	30	fb72
5	5	dd2a
35	35	c3e7
25	20	cb56
10	8	f208
15	12	f4b3
25	25	d1df
12	6	d715

The distribution list allows interesting combinations of write and read data rates for FIFO access.

FIFO Environment Classes and Transactors

In the previous chapter, the functionality that wrote, read and reset the FIFO, was contained within four separate procedural tasks: `fifo_mwrite()`, `fifo_mread()`, `fifo_reset()`, and `check_fifo_reset()`. Assume that we have two different FIFOs, which would require four additional tasks (one for read, one for write, and two for reset), doubling the amount of code that we would have to write and maintain.

However, applying object-oriented principles, the read and write tasks of the FIFO can be considered as common functionality that can be encapsulated in a class. Then, to instantiate more than one FIFO we only have to declare a new instance of our class.

Here is a code snippet of the FIFO driver class:

```
class driver;
   ...
   task fifo_mwrite();
      ...
   endtask

   task fifo_mread();
      ...
   endtask

endclass: driver
```

A similar class is created for the FIFO checker. Self-checking is discussed further in the next section.

In addition to these transactors, we create a new class named `generator` that produces randomized FIFO data. The generator class has functions and tasks declared as virtual so they can be

extended in inherited classes. In this way many constrained random tests can be created by simply extending the `generator` class in the main test program file.

```
class generator ;
   rand data_tr rand_tr;
   //holder randomized FIFO data(transaction)
   virtual task main();
      ...
      FIFO_DATA = get_transaction();
      ...
   endtask: main

////////////////////////////////////////
// get_transaction method
////////////////////////////////////////
   virtual function data_tr get_transaction();
      rand_tr.randomize();
      get_transaction = rand_tr;
   endfunction
   ...
endclass: generator
```

The `get_transaction()` function returns a randomized transaction of the FIFO data type (in this case, `data_tr`) and is declared as virtual so that it can be modified by extending the generator class. We will show an example of this extension and show how easy it is to create new tests. When `main()` is called in the generator, the actual randomized data transaction is returned to the global `FIFO_DATA` for simplicity.

Through the `generator` class we have pushed the stimulus generation up in the hierarchy, separating the generation from the actual driving and checking segments of the testbench. We now can easily change the data pattern (transaction) that is sent to the DUT at the top layer and pass it to the lower layer drivers and checker transactors.

We also create a configuration class that helps us configure tests. That class, named `test_cfg`, contains a random number that specifies how many times to write to the FIFO. Here is the code for this class:

```
class test_cfg;
   rand int NUM;
   constraint num_runs {
      NUM > 10; NUM < 50;
   }
endclass: test_cfg
```

To make the design of these classes more systematic, we incorporate methods that are a part of all transactor classes and can call other specific actions within the class. The main task that starts the transactor is an example of this systematic approach. The top level environment class contains instances of the generator, driver, and checker classes. Our environment class, named `fifo_gen`, is the class that we instantiate in our top-level program. Note that the reset related tasks are now part of the environment class.

```
/// environment class for the FIFO testbench,
//  instantiates the transactors and generator
class fifo_env ;
   //declaration of other objects/transactors
   generator fifo_gen;
   checker fifo_checker;
   driver fifo_driver;
   test_cfg cfg;
   int NUM;
   // Coverage group definition
   // Constructor

   function new();
      cfg = new();
      cfg.randomize();
      NUM = cfg.NUM;
      $write(" Test configuration for FIFO  \n");
```

```
      Cvr = new();
      fifo_gen = new();
   endfunction: new

   virtual function void build();
      begin
      fifo_driver = new();
      fifo_checker = new();
      end
   endfunction: build
////////////////////////////////////////
   task fifo_reset();
      ...
   endtask

   task fifo_reset_check();
      ...
   endtask

   task pre_test();
      fork
      fifo_reset();
      fifo_reset_check();
      join_none
      start_checker();
   endtask: pre_test

   task post_test();
      @(vintf.cb) Cvr.sample();
   endtask: post_test

   virtual task test();
      repeat(NUM) begin
      fifo_gen.main();
      fork
      fifo_driver.fifo_mwrite();
      fifo_driver.fifo_mread();
      join
      post_test();
      end
   endtask: test
```

```
   task run();
      build();
      pre_test();
      test();
   endtask: run

endclass: fifo_env
```

The `new()` function of `fifo_env()` accomplishes a number of things:

- It initializes test configuration (how many sets of data to push into the FIFO).
- It creates the functional coverage object.
- It creates the data object that will be randomized.

The `build()` function of `fifo_env()` creates the driver and checker objects.

The `pre_test()` function resets the FIFO, and then starts the checker running in the background.

The `test()` task randomizes the data object, writes data to the FIFO, and then reads back the data.

The `post_test()` task is called after a single test to sample the data rates for functional coverage.

The `run()` function of `fifo_env()` calls the build function, `pre_test()`, and then the `test()` task. The `fifo_env()` task is the task that is called from the top level program.

Checking for Functional Correctness Using a Self-Checking Technique

It is important to check the functional correctness of the design under test. Because we are generating random stimulus and sending this stimulus to the FIFO, we need to be able to determine whether the FIFO is operating correctly. In the next section we will illustrate the following two techniques:

- “SystemVerilog Checker” on page 60
- “SystemC Reference Model Checker” on page 61

SystemVerilog Checker

The first way we will check for functional correctness is by comparing the output of the FIFO to the data sent to the input of the FIFO. Because this is a FIFO, we know that data sent to the FIFO will come out the in same order.

In task `fifo_mwrite()` that writes to the FIFO, we take the input data and push it onto a SystemVerilog queue:

```
cdata.push_back(FIFO_DATA.data[j]);
```

In task `fifo_mread()`, whenever data is popped off of the FIFO we push it into a mailbox:

```
mbox.put(vintf.cb.data_out);
```

Now, in the checker class we will pop data off the mailbox and off the queue and compare the two values, which should be the same:

```
mbox.get(tempOut);
```

```
   tempIn = cdata.pop_front();
   assert( tempIn == tempOut )
   else $write("TEST: FIFO read data 'h%0h DID NOT
      match expected data 'h%0h\n",tempOut, tempIn);
```

If our DUT was more complicated and transmitted out-of-order data, then it would be necessary to search through the queue to find the correct value, using the built-in methods of the queue.

SystemC Reference Model Checker

VCS has incorporated transaction level interface support for SystemVerilog and SystemC. This allows a SystemC reference model to connect to SystemVerilog Design and Testbench for Verification re-use. In the following example, we included a call to a FIFO reference model in SystemC that runs in parallel with the checker in the SystemVerilog transaction level model described in previous sections.

To export our FIFO SystemC model into HDL, we must define our access mechanism, which is derived from the `sc_interface` SystemC class as follows:

```
  //////  systemC code reference model ///////
class fifo_if : public virtual sc_interface
{
   public:
      virtual void nb_push(int data) = 0;
      virtual void pop(int *data) = 0;
      virtual int status() = 0;
}; // end class fifo_if
```

We have defined the `nb_push()` method to be non-blocking, and the `pop()` method to be blocking. This is defined in the “idf” file, which is used to automatically create the helper files.

```
interface fifo_if
direction sv_calls_sc
verilog_module fifo_helper
systemc_module fifo_sysc_helper
#include "fifo_if.h"
function nb_push
input int data

task pop
output int data

function int status
```

In this file format, you can specify whether SystemVerilog calls SystemC, or SystemC calls SystemVerilog. Also, you must define either functions (nonblocking) or tasks (blocking). See the HTML VCS documentation for more detailed information.

The SystemC reference model includes three different function calls:

```
SC_MODULE(fifo), public fifo_if
{
   public:
      sc_in<sc_logic>  dummy;
      sc_fifo<int> my_fifo;
      int tmp;
      SC_CTOR(fifo) : my_fifo(128)   {
}
   void pop(int *data) {
      if (my_fifo.num_free() == 128)
      printf("ERROR within SystemC.. pop when fifo is
         empty\n");
      my_fifo.read(tmp);
      *data = tmp;
   }
   void nb_push(int data) {
      if (my_fifo.num_free() == 0)
      printf("ERROR within SystemC.. push when fifo is
         full\n");
      my_fifo.nb_write(data);
```

```
   }
   int status() {
      return (my_fifo.num_free() );
   }

};
```

The `nb_push()` method is used when we want to push data into our FIFO, and the `pop()` method is used when we want to pop data from our FIFO. We have also included a method named `status()` that returns the number of free entries in the FIFO.

In the testbench, in the `fifo_mwrite()` task, whenever we write data into our DUT we will also call the `nb_push()` method of the reference model:

```
fifo_test_top.fifo_reference_model.nb_push(FIFO_DATA.
data[j]);
```

Also, in the `checker` class, we can compare the output of the SystemC reference model with the actual output of the DUT:

```
class checker;
   virtual task main();
   logic [`WIDTH-1:0] tempIn;
   logic [`WIDTH-1:0] tempOut;
   logic [`WIDTH-1:0] syscOut;
   $write("FIFO_CHECK: Task check started \n");
   forever begin
      mbox.get(tempOut);
      tempIn = cdata.pop_front();
////////// Checker #1 ////////////////////
      assert( tempIn == tempOut )
      else $write("CHECKER: FIFO read data 'h%0h DID NOT
         match expected data 'h%0h\n",tempOut, tempIn);

///////////// Checker #2 /////////////////
      fifo_test_top.fifo_reference_model.pop(syscOut);
```

```
      assert( syscOut == tempOut )
      else $write("SYSC_CHECKER: FIFO read data 'h%0h DID
         NOT match expected data 'h%0h\n",tempOut, syscOut);

      end
   endtask: main
endclass: checker
```

How to Run Tests

The main test program instantiates the FIFO environment and calls the main run routine. This is the default test case set in `fifo_test.v`. We can extend the generator and transactions to create new test cases.

Default Run fifo_test_00_default.v

The following is the default test, which is contained in `fifo_test.v`. The program is declared with a single interface in the port list. A virtual interface is declared with the name `vintf`. All of our base classes are also included here:

```
program fifo_test ( fifo_intf intf);
   virtual fifo_intf.tb vintf;
`include "fifo_env.v"

mailbox mbox;
   data_tr FIFO_DATA;
   fifo_env fifo_tb_env;
   logic [`WIDTH-1:0] cdata[$];
```

Here all our global data is declared, our mailbox, our random data, our environment class, and our queue:

```
   initial begin : main_prog
      vintf = intf;
      mbox=new();
      FIFO_DATA = new();
      fifo_tb_env = new();
      vintf.rst_n <= 1'b0;
      @(vintf.cb);
      fifo_tb_env.run();
   end : main_prog
endprogram : fifo_test
```

The code snippet above shows the main part of our test: Our virtual interface `vintf` is initialized from the actual interface; all of our objects are created using `new()`, and our reset signal `rst_n` is asserted. Then the `run()` method is called, which runs the test and sends a random number of data sets to the FIFO.

Extended Run fifo_test_01_extend.v

The benefit of this methodology is that it easily allows one to change the definition of a testbench entity. If we want to change the generator to write with a data rate equal to the read data rate of 128, we can easily implement this as follows:

```
class my_gen extends generator;
   function data_tr get_transaction();
      $write("this is from extended generator \n");
      rand_tr.randomize() with {
         wr_data_rate == 128;
         rd_data_rate == wr_data_rate;
      };
      get_transaction = rand_tr;
   endfunction
endclass
```

We then “plug” this generator back into our environment as follows:

```
program fifo_test( fifo_intf intf);
   ...
   class my_gen extends generator;
      ...
   endclass

   my_gen fifo_new_generator;
   ...
   fifo_new_generator= new();
   fifo_tb_env = new();

   fifo_tb_env.fifo_gen = fifo_new_generator;
   fifo_tb_env.run();

   end: main_prog
endprogram: fifo_test
```

The key is that we do not need to modify any of our base classes. This is all done in the test-specific file `fifo_test_01_extend.v`.

To create another test with different constraints, we can copy the file to `fifo_test_02_my_test.v` and edit that file.

Assertions

Assertions help designers and verification teams to catch bugs faster. With VCS, you can easily add assertions into your design to check for correct design behavior. Using VCS, you can write sophisticated assertions from scratch, or you can use the built-in Checker Library. VCS has a rich set of checkers in its library that you can easily instantiate in your design to identify bugs.

For a full description of the Checker Library, see the *SystemVerilog Assertions Checker Library Reference Manual*. The checks in the library include:

- one hot/ cold
- mutual exclusion
- even/odd parity
- handshake
- overflow
- underflow
- window
- arbiter
- many others

The advantage of using the Checker Library is that it is pre-written, pre-verified, and ready to be instantiated. Another advantage of using the Checker Library is that it has comprehensive assertion

coverage built into each checker. It includes corner case coverage and range coverage. So the checkers not only find bugs, but also report on coverage holes in your design.

You can instantiate the FIFO assertion directly into either the Verilog, or the VHDL design. The assertion can also be put in a separate file using the `bind` construct. This works for both Verilog and VHDL blocks.

For a more detailed description, see “Assertions” on page 17.

Comprehensive Code Coverage

VCS has a built-in comprehensive code coverage for both Verilog and VHDL designs. See the *VCS/VCS MX Coverage Metrics User Guide* for full documentation. For types of coverage supported, see “Comprehensive Code Coverage” on page 21.

Functional Coverage

Functional coverage provides a metric for measuring the progress of design verification, based on the functional test plan and the design functional specification. Functional coverage is used in conjunction with design code coverage. The main purpose of functional coverage is to guide the verification process by identifying tested and untested areas of the design and ensuring that corner cases are tested. For more details see “Functional Coverage” on page 22.

Running the Test Case

A makefile is provided in the directory with rest of the code for this quick start. Use the following compile command to run the Verilog version of the FIFO with the default constraints:

```
make verilog_00
```

For a VHDL version of the test case, use the following command:

```
make vhdl_00
```

To run with the extended generator, type the following two commands:

```
make verilog_01
make vhdl_01
```

To generate testbench coverage, run the following command:

```
vcs -cov_text_report fifo_test.db
```

FIFO Results

Use the following command to simulate the FIFO with the default configuration:

```
make verilog_00
```

The testbench reports the `data_rate` for each call in `fifo_driver()` to the methods `fifo_mwrite()` and `fifo_mread()`.

The following is the output from the simulator:

```
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version X-2005.06-SP1-9; Runtime version X-2005.06-
SP1-9;  Jan 10 14:38 2006

 Test configuration for FIFO
Sending          17 transactions to DUT
FIFO_RESET: Task reset_fifo started
FIFO_RESET_CHECK: Task reset_check started
FIFO_CHECK: Task check started
FIFO_MWRITE: Sending 128 words with data_rate of  130 to
fifo at:             50
FIFO_MREAD:  Reading 128 words with data_rate of  128 from
fifo at:           50
FIFO_MWRITE: Sending 128 words with data_rate of    4 to
fifo at:        1677150
FIFO_MREAD:  Reading 128 words with data_rate of    7 from
fifo at:      1677150
FIFO_MWRITE: Sending 128 words with data_rate of    2 to
fifo at:        1779650
FIFO_MREAD:  Reading 128 words with data_rate of    6 from
fifo at:      1779650
FIFO_MWRITE: Sending 128 words with data_rate of    2 to
fifo at:        1869350
FIFO_MREAD:  Reading 128 words with data_rate of  129 from
fifo at:      1869350
```

```
FIFO_MWRITE: Sending 128 words with data_rate of  514 to
fifo at:       3533450
FIFO_MREAD:  Reading 128 words with data_rate of  133 from
fifo at:      3533450
FIFO_MWRITE: Sending 128 words with data_rate of    3 to
fifo at:      10125750
FIFO_MREAD:  Reading 128 words with data_rate of  514 from
fifo at:     10125750
FIFO_MWRITE: Sending 128 words with data_rate of    4 to
fifo at:      16717850
FIFO_MREAD:  Reading 128 words with data_rate of    5 from
fifo at:     16717850
FIFO_MWRITE: Sending 128 words with data_rate of  513 to
fifo at:      16794850
FIFO_MREAD:  Reading 128 words with data_rate of  512 from
fifo at:     16794850
FIFO_MWRITE: Sending 128 words with data_rate of  132 to
fifo at:      23374350
FIFO_MREAD:  Reading 128 words with data_rate of  519 from
fifo at:     23374350
FIFO_MWRITE: Sending 128 words with data_rate of  518 to
fifo at:      30030450
FIFO_MREAD:  Reading 128 words with data_rate of  516 from
fifo at:     30030450
FIFO_MWRITE: Sending 128 words with data_rate of  517 to
fifo at:      36673950
FIFO_MREAD:  Reading 128 words with data_rate of  519 from
fifo at:     36673950
FIFO_MWRITE: Sending 128 words with data_rate of  515 to
fifo at:      43330050
FIFO_MREAD:  Reading 128 words with data_rate of    3 from
fifo at:     43330050
FIFO_MWRITE: Sending 128 words with data_rate of  128 to
fifo at:      49935150
FIFO_MREAD:  Reading 128 words with data_rate of    3 from
fifo at:     49935150
FIFO_MWRITE: Sending 128 words with data_rate of  134 to
fifo at:      51586650
FIFO_MREAD:  Reading 128 words with data_rate of  132 from
fifo at:     51586650
FIFO_MWRITE: Sending 128 words with data_rate of    7 to
fifo at:      53314950
FIFO_MREAD:  Reading 128 words with data_rate of  133 from
fifo at:     53314950
```

```
FIFO_MWRITE: Sending 128 words with data_rate of  519 to
fifo at:       55030250
FIFO_MREAD:  Reading 128 words with data_rate of  131 from
fifo at:     55030250
FIFO_MWRITE: Sending 128 words with data_rate of    4 to
fifo at:       61686550
FIFO_MREAD:  Reading 128 words with data_rate of  133 from
fifo at:     61686550
$finish at simulation time          63401850000
```

You can now look at the testbench coverage for 500 tests using this command:

```
vcs -cov_text_report fifo_test.db
```

The following is the generated report:

```
Functional Coverage Report

Coverage Summary
    Number of coverage types: 1
    Number of coverage Instances: 1
    Cumulative coverage: 100.00
    Instance coverage: 0.00

Coverage Groups Report

Coverage group                    # inst   weight   cum cov.
=============================================================
fifo_env::Cvr                       1       1       100.00
=============================================================


=============================================================
= Cumulative report for fifo_env::Cvr
=============================================================

Summary:
    Coverage: 100.00
    Goal: 100
```

```
Coverpoint                    Coverage     Goal       Weight
=============================================================
RD                          100.00      100        1
WD                          100.00      100        1
=============================================================
Cross                         Coverage     Goal       Weight
=============================================================
RDxWD                        100.00      100        1
=============================================================


Cross Coverage report
CoverageGroup: fifo_env::Cvr
    Cross: RDxWD
Summary
    Coverage: 100.00
    Goal: 100
    Number of Coverpoints Crossed: 2
    Coverpoints Crossed: RD WD
    Number of Expected Cross Bins: 9
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 9

    Automatically Generated Cross Bins

    RD              WD              # hits         at least
   ============================================================
    HIGH            HIGH            3              1
    HIGH            LOW             1              1
    HIGH            MED             1              1
    LOW             HIGH            1              1
    LOW             LOW             3              1
    LOW             MED             1              1
    MED             HIGH            2              1
    MED             LOW             3              1
    MED             MED             2              1
   ============================================================

Coverpoint Coverage report
CoverageGroup: fifo_env::Cvr
    Coverpoint: RD
```

```
Summary
    Coverage: 100.00
    Goal: 100
    Number of User Defined Bins: 3
    Number of Automatically Generated Bins: 0
    Number of User Defined Transitions: 0

    User Defined Bins

    Bin                                     # hits    at least

=============================================================
    HIGH                                    5         1
    LOW                                     5         1
    MED                                     7         1

=============================================================

Coverpoint Coverage report
CoverageGroup: fifo_env::Cvr
    Coverpoint: WD
Summary
    Coverage: 100.00
    Goal: 100
    Number of User Defined Bins: 3
    Number of Automatically Generated Bins: 0
    Number of User Defined Transitions: 0
    User Defined Bins

    Bin                                     # hits    at least

=============================================================
    HIGH                                    6         1
    LOW                                     7         1
    MED                                     4         1

=============================================================

Database Files Included in This Report:

    fifo_test.db
```

```
Test Bench Runs Included in This Report:

Test Bench          Simulation Time     Random Seed
Random48 Seed
============================================================
fifo_test.db::fifo_test 63401850000       0               0
============================================================
```

Debug

VCS 2005.06 and later versions come with a comprehensive built-in graphical user interface named Discovery Visual Environment (DVE). DVE supports graphical debugging of mixed-language designs including Verilog, VHDL, and SystemVerilog. DVE also supports assertion debugging. Detailed documentation can be found in the *Discovery Visual Environment User Guide*.

DVE can be run interactively or post-processing. Use the following command to invoke the debugger:

```
simv -gui
```

After the debugger starts you can set breakpoints in the design code and the testbench code, trace drivers, and navigate easily through the hierarchy.

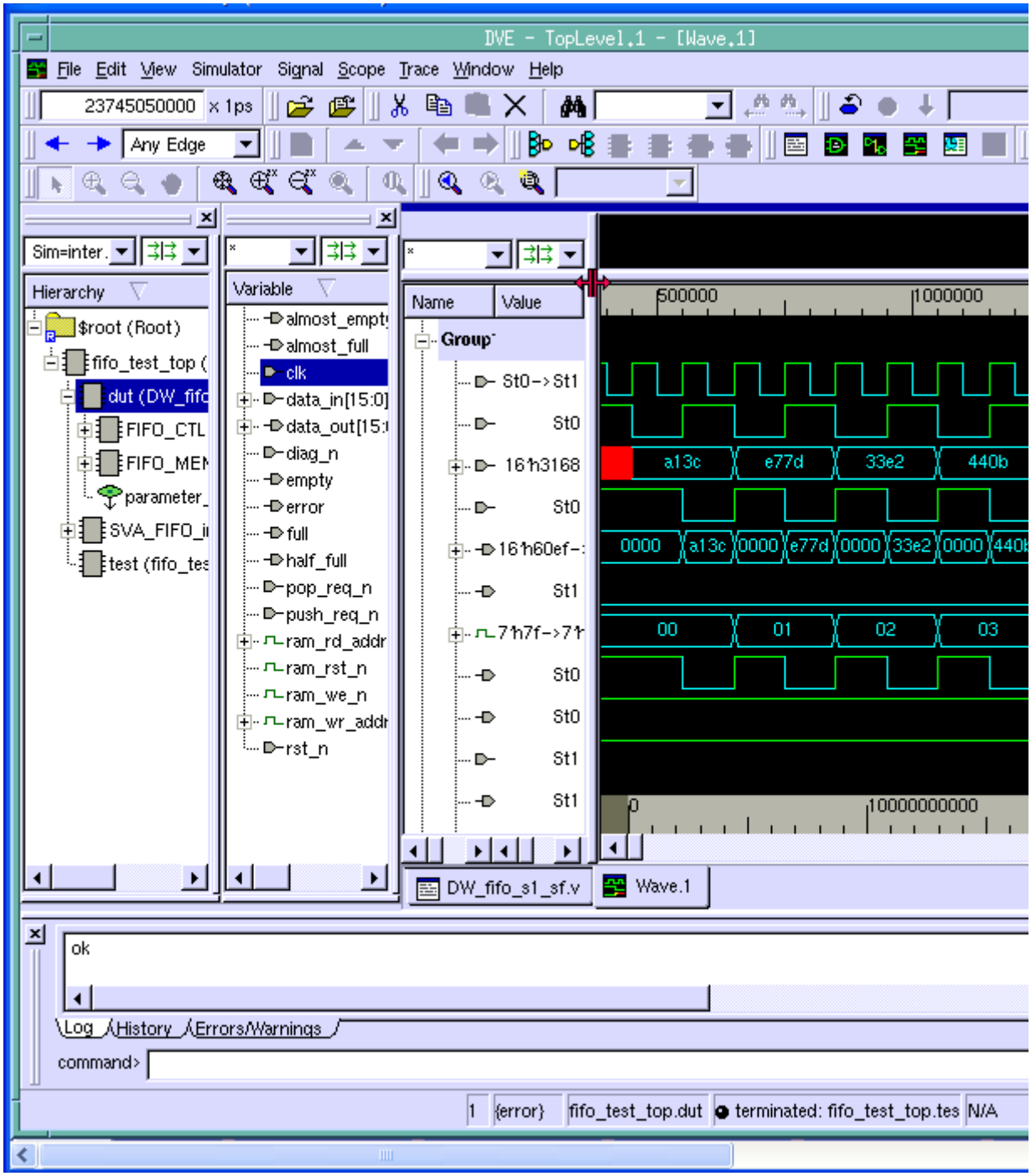
Figure 3-5 is a waveform trace showing an assertion failure at time 650 due to memory corruption in the FIFO. Assume there is a design bug in the FIFO that affects the data read from the FIFO. During simulation, there will be an assertion failure from the `assert_fifo` checker library you have already instantiated. The error message will look something like this:

```
"fifo_test.v", 162:
   fifo_test_top.test.fifo_check.unnamed$$_1: started at
```

```
   550000ps failed at 550000ps
   Offending '(tempIn == tempOut)'
TEST: FIFO read data 'h5ec3 DID NOT match expected data 'ha13c
```

You can use the VCS GUI to help you debug the problem.

Figure 3-5 Waveform Showing Assertion Failure Due to Memory Corruption



Summary

We have introduced the new constructs for testbench and verification, interfaces, and clocking blocks, as well as object-oriented design constructs in VCS.

VCS is leading the industry to incorporate testbench, assertion and coverage capabilities needed for design and verification engineering tasks. With these technologies encapsulated, VCS creates the framework and foundation for building a structured layered verification environments at all levels, from module to cluster-of-modules, chip and system level verification.

VCS enables the thorough verification of complex SoC designs, in much less time than with other tools and methods. The result is a greater likelihood of first-silicon success.

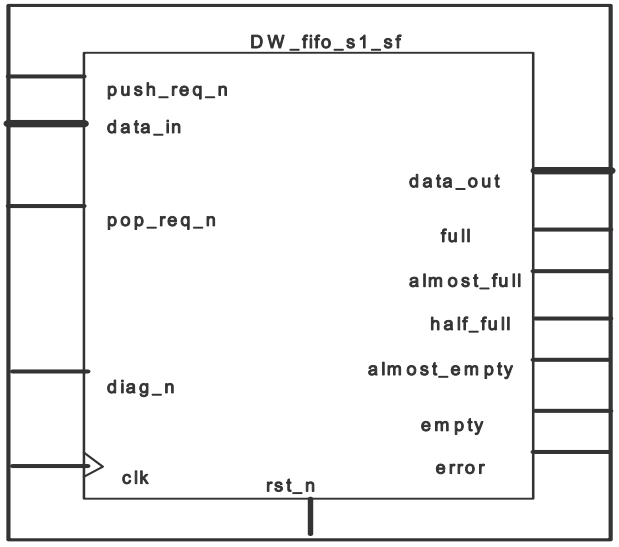
A

The FIFO Design Block

The FIFO design is a synchronous (single-clock) FIFO with static flags. It is fully parameterized and has single-cycle push and pop operations. It also has empty, half-full, and full flags, as well as parameterized almost full and almost empty flag thresholds with error flags.

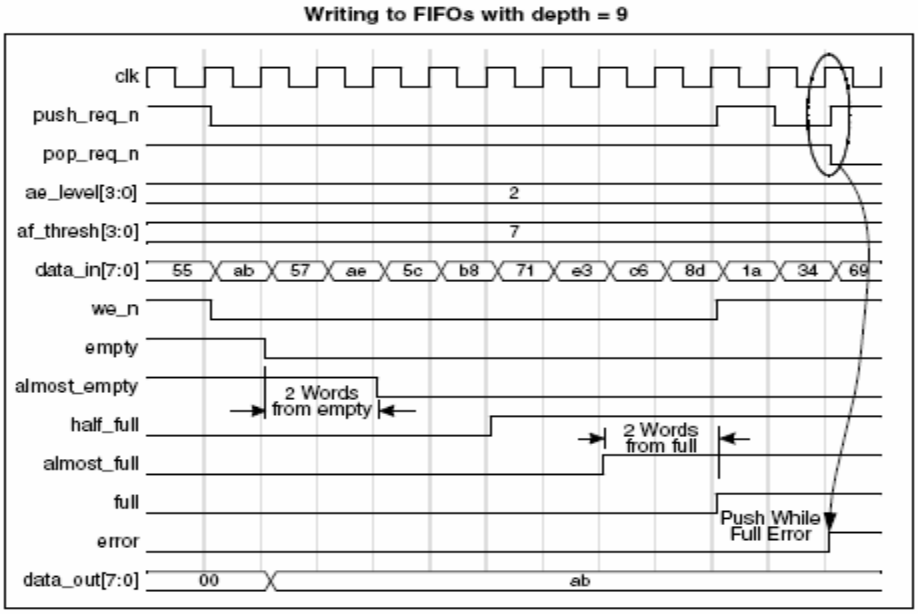
Figure A-1 shows a block diagram for the FIFO design.

Figure A-1 FIFO Design Block Diagram



The sample timing diagram in Figure A-2 illustrates how the FIFO block works.

Figure A-2 FIFO Timing Diagram



B

Testbench Files and Structure

You can find the testbench and design files in the following locations in the VCS installation.

- Basic features:

  ```
  $VCS_HOME/doc/examples/nativetestbench/systemverilog/
  vcs_quickstart/basic
  ```

- Advanced features:

  ```
  $VCS_HOME/doc/examples/nativetestbench/systemverilog/
  vcs_quickstart/advanced
  ```

Files for Using Basic Features in VCS

README

DW_fifo_s1_sf.v—Verilog FIFO model

DW_fifoctl_s1_sf.v —Verilog FIFO controller

DW_ram_r_w_s_dff.v— Verilog memory

DW_fifo_s1_sf.vhd — VHDL FIFO entity

DW_fifo_s1_sf_sim.vhd—VHDL FIFO architecture

DWpackages.vhd—VHDL packages

synopsys_sim.setup—Setup file

fifo.test_top.v—Verilog top level instantiating program and

fifo.test_top.vhd—VHDL top level instantiating program and

fifo_test.v—program testbench

Files for Using Advanced Features in VCS

README

DW_fifo_s1_sf.v —Verilog FIFO model

DW_fifoctl_s1_sf.v—Verilog FIFO controller

DW_ram_r_w_s_dff.v—Verilog memory

DW_fifo_s1_sf.vhd—VHDL FIFO entity

DW_fifo_s1_sf_sim.vhd—VHDL FIFO architecture

DWpackages.vhd—VHDL packages

synopsys_sim.setup—VHDL setup file

fifo.test_top.v— Verilog top level instantiating program and

fifo.test_top.vhd—VHDL top level instantiating program and

fifo_env.v—environment, generator, transactor file

fifo_test_00_default.v—program testbench

fifo_test_01_extend.v—extend program testbench

fifo.h—SystemC Models of FIFO

scu_dut.h—SystemC wrapper file

fifo_sysc_helper.h—SystemC helper files (automatically created)

fifo.idf—SystemC interconnect definition file

C

Verification IP

Verification components help to automate the creation of a verification environment. The VCS solution, besides fully supporting DesignWare verification IP, has added an Assertion IP Library of components for standard protocols to help with verification tasks.

Assertion IP Library

Digital designs must implement complex functional specifications. Many of the design components use industry-standard protocols in order to easily plug and play with other system components. However these complex protocols such PCIX standard usually require sophisticated verification environments to validate the designs.

To fully test compliance with the standard protocol, the Assertion IP Library supported by VCS provides a set of checkers that can be used to verify complex protocols. The VCS Assertion IP Library allows designers to perform functional checks during simulation, identify and report protocol violations, and to capture assertion coverage data. You can easily augment your verification environment, the constrained-random stimulus generation and coverage routines, with the assertion IP at the block-level to full chip-level environment. These components are pre-built and pre-verified to comply with the standard protocols. You can configure the assertion IP according to the functionality of your DUT. Each IP contains a set of assertions to monitor the protocol behavior and coverage targets to aid in finding out more about how thoroughly your test cases exercise the respective protocol functionality.

Included as a standard feature starting with the VCS 2005.06 release, the VCS Assertion IP is provided for the following interface and protocol standards:

- PCI and PCI-X® 2.0 interface
- AMBA 2 AHB and APB, 802.11a-g
- AGP, SMIA, DDR2, OCP 2.0 and LPC

Look in the $VCS_HOME/aip/ directory for a complete list of components.

The VCS Verification IP (VIP) Library

VCS Verification IP packages provide the framework for building constrained-random stimulus for many complex standard protocols. These components can be plugged into the verification environment to create random scenarios based on the configuration of the DUT. For complete functionality, these VIP components provide full monitors and coverage to aid in verification tasks. The library contains many popular standard protocols such as PCI, PCI-X, PCI Express, USB 2.0, Ethernet, Serial ATA, AMBA, as well as memory models. DesignWare Verification is fully functional in VCS NTB 2005.06.

For more information on verification and implementation IPs, please check the Verification IP site: http://www.synopsys.com/products/designware/designware.html.