

# verificationavenue

The Synopsys technical bulletin for design and verification engineers

issue 3, February 2002

## general manager's column

### To Our Valued Customers

Welcome to the third issue of the Verification Avenue. Since the first newsletter last May we have seen a steady rise in both the readership and the scope of the publication. Many of you have suggested topics for future newsletters and we are consciously addressing these topics. Please continue to send your suggestions and thank you for your support and guidance.

In October Synopsys announced the availability of VCS™ 6.0.1. Coverage Metrics and DirectC were the main highlights of this release. Included in this issue are articles on these capabilities. VCS's new coverage capability simplify the verification flow significantly improve performance. DirectC enables the import C functions without PLI performance penalties. Our simulation articles continue with an article on Mixed-Language simulation and another that suggests Verilog coding style changes to improve simulation throughput.

We are also happy to include a verification methodology article from Synopsys Professional Services team. This article shows a best-practice approach to improve your OpenVera testbench. I think you will find this article particularly helpful. The article

lays out a methodical approach towards improving testbench automation to realize higher productivity.

The OpenVera™ Community is pleased to announce an online forum for posting questions, exchanging ideas and getting expert advice from the growing number of OpenVera users. Its purpose is to allow the easy sharing of information relating to hardware verification language OpenVera; verification methodology; testbench automation tool VERA®; interfaces between VERA and other EDA products; and general verification issues. Join the OpenVera Discussion Forum today by registering as a new user.

As we start work on the fourth issue of Verification Avenue I want to encourage your continued feedback. It's only by listening to our customers that we can continue to improve this newsletter. To thank you for your feedback we are giving away a [Nikon Coolpix 885](#) digital camera and copies of the book [The Art of Verification with VERA](#). Please drop by [our website](#) and tell us what you think.

*Best regards,  
Manoj Gandhi, Senior Vice President  
and General Manager  
Verification Technology Group*

## Contents

### General Managers Column

To Our Valued Customers .....1

### Product Updates

VERA and VCS Tightly Integrated  
for Faster and Smarter Verification ..... 2

Synopsys Further Improves Performance  
of VHDL Simulation with the Latest  
Release of Scirocco ..... 3

New Release of VCS Verilog Simulator  
Incorporates Breakthrough  
Verification Capabilities ..... 4

### Concepts

Testbench Design, a Systematic Approach ..... 5

Mixed-Language Simulation with VCS ..... 9

Synchronous Coding Styles for  
Maximum Performance ..... 13

OpenVera Graphical Code Generation ..... 15

Using C/C++ function in the VCS  
Verification Flow ..... 16

Coverage Metrics built into VCS for  
Smarter Verification ..... 19

### Questions and Answers

What is the new VCS coverage metrics  
feature and how do I enable it? ..... 21

How do I take advantage of the new,  
tight integration between VERA 5.0  
and VCS 6.0.1? ..... 21

I am simulating a mix of Verilog and VHDL.  
What kind of performance should I expect  
and what factors impacts it? ..... 21

How is the VERA profiling feature used? ..... 21

How do I take advantage of using the  
simplified UDF feature in VERA 5.0? ..... 22

### News & Events Calendar

Events Calendar ..... 21

News ..... 22

Documentation ..... 22

Feedback ..... 22

Synopsys, Inc.  
700 East Middlefield Road  
Mountain View, CA 94043

Email  
[vtg-news@synopsys.com](mailto:vtg-news@synopsys.com)

contact

# product updates

## VERA and VCS Tightly Integrated For Faster and Smarter Verification

Synopsys announced the release of VERA® 5.0, the latest version of the testbench generation tool, on November 12.

VERA 5.0 with VCS™ 6.0.1 offers a solution that is deeply integrated using the VCS Direct Kernel Interface and more, to address engineers' most pressing verification problems. This unique integration cuts down the overall verification time and improves testbench efficiency by providing real-time access to VCS simulation coverage metrics.

### Problem:

Verification speed, creating efficient testbenches and achieving coverage objectives quickly are among the top challenges faced by customers today.

Until now, testbench and simulation tools have viewed each other as external point tools and have relied on standard interfaces, such as Verilog Programming Language Interface, to exchange information. PLI based communication impacts the *content* and the *speed* of information exchanged between external tools.

### Solution:

VERA 5.0 + VCS 6.0.1 implement a unique integration, providing internal access to each other's engines through VCS Direct Kernel Interface and other unique communication mechanisms. This deep integration enables verification engineers to create faster and smarter verification environment. Faster verification delivers more simulation cycles in a given time and smarter verification delivers testbenches that are coverage-aware and hence able to react to simulation coverage goals. Combined, these capabilities enable customers to find more bugs in less time.

In addition, VERA 5.0 offers unification of design and testbench waveform analysis. VERA 5.0 supports viewing of VERA components in VCS VirSim waveform viewer. This eliminates the need for engineers to use separate tools for testbench and design analysis and results in faster debug of simulation mismatches.

VERA 5.0 also provides a performance profiler that can be used to analyze the amount of simulation time used up by different components of the testbench. This provides visibility into slow-executing testbench blocks, which can be optimized to create efficient testbenches.

For further questions please send email to [vera-support@synopsys.com](mailto:vera-support@synopsys.com)

### **Synopsys further improves performance of VHDL simulation with the latest release of Scirocco**

Last month Synopsys released the new version of Scirocco™, its high performance VHDL simulator. This release focuses on higher performance and ease of use, and enables designers to adopt Scirocco quicker and to benefit from highest runtime performance to speed up their verification task.

Out-of-the-box performance has been greatly improved in Scirocco 2001.10. Our customers have been consistently reporting faster simulations, with 1.5 to 2X performance improvements over the previous version 2000.12-2.

By enabling cycle compiler optimizations, Scirocco customers benefit from even higher simulation performance. In the past, cycle optimizations have been limited to RTL descriptions. But many designers have blocks embedded in their synthesizable designs, that are not described at RTL: behavioral IPs, in VHDL or Verilog, memory models provided by their vendor in VITAL, gate-level legacy blocks...

This new release of Scirocco now enables designers to apply cycle optimizations beyond RTL code. It is now possible to accelerate the VITAL descriptions of memory, pads, or any legacy block, thus improving the overall simulation performance of the complete design.

Behavioral IPs (in VHDL or Verilog) which cannot be cycle-optimized, can now be instantiated in an optimized RTL description. This results in faster runtime performance and quicker activation of cycle optimizations.

The Graphical User Interface has also been enhanced to facilitate the adoption of Scirocco. A project browser enables designers to manage their projects through intuitive graphical views of their source files, libraries and simulation options. And the new release of Scirocco, combined with VCS 6.1 now offers a language neutral GUI for interactive debugging.

## Announcing VCS, Version 6.0.1

VCS™ 6.0.1, which began shipping in October 2001, provides a major boost to overall verification productivity of our customers. This new release contains built-in comprehensive coverage analysis, enabling design teams using VCS to determine their verification quality before tapeout. In addition, Synopsys has added VCS DirectC, a new interface to accommodate the use of C/C++ models within a Verilog verification environment.

### Built-in Coverage

Coverage metrics are an industry-accepted measure of simulation effectiveness. As a standard part of VCS, users will now have access to comprehensive built-in coverage analysis, including condition, toggle, line and finite-state-machine coverage. Using these capabilities built into the VCS engine, design teams can easily determine the quality or "coverage" of their verification tests. With the latest VCS release, designers only need to compile once to run both simulation and coverage analysis. As a result of this single compilation, users will see substantially better compile and run-time performance than with previous point tool solutions that use the Verilog Programming Language Interface (PLI). VCS combined with built-in coverage metrics allows design teams quickly achieve quality objectives within a single Verilog-based flow.

from / to	FETCH	DECODE	EXECUTE	MEMORY	WRITE_BACK
FETCH	1/4	0/0	0/0	0/0	0/0
DECODE	1/4	1/4	0/0	0/0	0/0
EXECUTE	1/3	1/3	1/3	0/0	0/0
MEMORY	1/2	1/2	1/2	1/2	0/0
WRITE_BACK	1/1	1/1	1/1	1/1	1/1

Is Covered	Length	Sequences
✓	3	MEMORY -> WRITE_BACK -> FETCH
✗	2	MEMORY -> FETCH

*Figure 1 – VCS 6.0.1 provides high performance simulation with coverage metrics and advanced debug capabilities through its integrated graphical interface.*

### VCS DirectC

The VCS DirectC interface significantly improves ease-of-use and performance over existing PLI-based methods by enabling designers to directly embed C/C++ functions within their Verilog design description. VCS automatically recognizes these C/C++ function calls and integrates them into the simulation run, in contrast to interfacing with them via manually created PLI files. Furthermore, using this interface eliminates debugging often associated with PLIs. As a result, VCS DirectC users can expect up to 2x simulation performance improvement over PLI.

### Package Review

VCS 6.0.1 is currently available. Current customers will receive the new version as a maintenance upgrade. A single license enables VCS to run on any supported platform.

# concepts

## Testbench Design, a Systematic Approach

### 1 Introduction

There are many technical publications in the electronic industry that have recognized and brought awareness to the key issue of functional verification that faces companies designing System on a Chip (SoC). This task is estimated, depending on the complexity and size of the chip, to take more than 50- percent of the total effort to bring the SoC to market. Table 1 below shows the main tasks related to the functional verification of a typical SoC.

Typically the verification engineer has to first study and understand the specification of the Design Under Test (DUT) in detail before any tests or testbenches can be developed. This paper suggests that if functional tests can be written in terms of the activities described in the specification of the DUT, these tests will be much easier to write, debug, and maintain. This can reduce the effort of developing and debugging tests by 25-percent (see Table 1), which currently at 60-percent is the largest effort required in SoC functional verification. Using the suggested architecture, the engineer can end up with a powerful what-if scenario engine that can lead to an increased level of confidence in the functional correctness of the SoC.

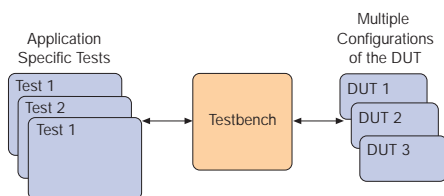


Figure 1. Generic Relationship between Tests, Testbench and DUT.

ID	Task Description for Functional Verification Only	Typical Effort Distribution for Main Tasks in SoC Verification *	Improved Effort Distribution Main Tasks in SoC Verification**
1	Study DUT specification and develop verification strategy	10 %	10 %
2	Develop detailed test plan	10 %	10 %
3	Develop testbench environment	10 %	35 % ***
4	Develop and debug tests	60 %	10 %
5	Run regression test suite	10 %	10 %
	Total	100%	75%
6	Effort savings	0	25 %

Table 1

\* Typical here implies functional verification using directed tests with non-layered approach and none to minimal use of self checking and randomization.

\*\* Improved here implies using the proposed testbench architecture methodology to achieve same functional coverage as stated in test plan.

\*\*\* This effort will be further reduced after the first project that uses this testbench methodology.

### 2 Testbench Architecture

Before discussing testbench architecture solutions, let's review the main goals of devising an improved testbench architecture:

1. Reduce the effort of test development.
2. Reduce the effort of debugging to isolate bugs easily
3. Enable re-use of tests from sub-system level testing in system-level testing
4. Provide quality metrics of the test through code and functional coverage
5. Maximize re-use of the testbench across multiple projects

To resolve this open-ended problem, it helps to begin by thinking at higher level. Tests are connected to the DUTs by the testbench, which in turn is connected to the DUT configurations as shown in Figure 1. Note that different applications require different tests and DUTs. Even within the same application, the same tests typically have multiple DUT representations.

The testbench needs a common architecture that takes into account the variations of the DUT as well as the variations in the required tests.

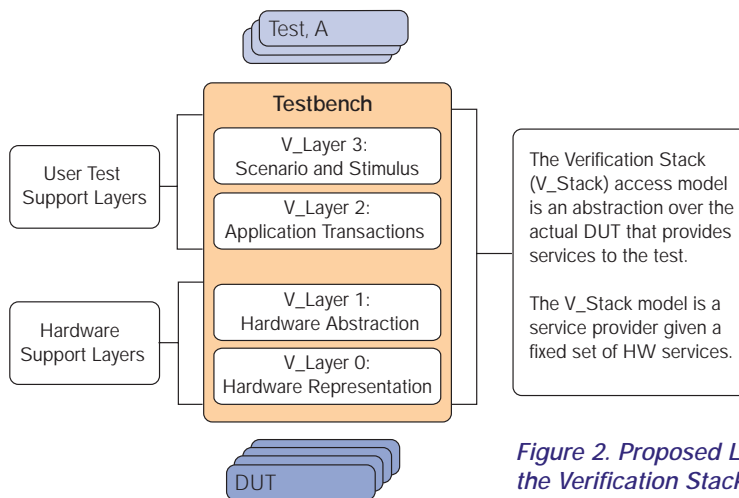


Figure 2. Proposed Layered Solution: the Verification Stack (V\_Stack model), with its four layers.

# concepts

## 2.1 A Layered Approach

The proposed solution suggests a layered approach with clearly defined functionality for each of the layers as shown in Figure 2. Each layer then provides a certain set of services to the upper layer, while shielding it from lower-level details. Each of these layers will be referred to as a Verification Layer (V\_Layer), and collectively, the four V\_Layers are known as the Verification Stack (V\_Stack).

### 2.1.1 V\_Layer 0: Hardware Representation Layer

This V\_Layer 0 provides signal-level connectivity into the physical representation of the DUT (typically described in HDL or SystemC™). For example, if the DUT is implemented in HDL and the rest of the testbench is in VERA®, this layer would consist of the VERA interface file. In it one would define the signals (of the DUT) that are of interest. In short, this layer provides signal name-abstraction and connectivity to the event-driven worlds of most simulation engines. Here is an example of the data in an interface file that constitutes this layer:

Signal Direction	Virtual Signal Name	Signal Sampling Specification	Corresponding Signal Name in HDL
Input	Core_clk	CLOCK	HDL_NODE "TB.dut.clk";
Output [31:0]	Xm_addr	Output_edge Output_skew	HDL_NODE "TB.dut.addr";

The concept here is that the test developer uses virtual signal names instead of real (implementation specific) names when implementing tests. This has the immediate and significant benefit that if the signal names or paths change in the DUT (or HDL part of the testbench), the tests would not be affected. It is not unusual for the interface layer to define several hundred signals, and for the (existing) environments to contain hundreds of tests. From a maintenance point of view alone, the verification engineer would have one less complication to deal with.

### 2.1.2 V\_Layer 1: Hardware Abstraction Layer

V\_Layer 1 provides a HW bus-abstraction view of the hardware. Its role is to make sure that the bus transactions issued by upper V\_Layers will reach the DUT, regardless of how the DUT is represented. Furthermore, this V\_Layer is responsible for configuring the DUT. Typically, this V\_Layer would include bus transactors, bus monitors, protocol checkers, and simple behavioral models for the case when the DUT is partially represented because RTL is not yet available.

Typical commands that are present in this layer usually correlate to bus transactions. Here is an example of a HW transaction that can be issued from this layer (PCI write, then read):

**Pci\_transaction (WE\_CMMD, ADRS, DATA, BYTE\_ENABLE);**

**WE\_CMMD:** a PCI bus command for write, and read.  
**ADRS:** an address in PCI address space.  
**DATA, and BYTE\_ENABLE:** data and the "byte\_enable" flags

### 2.1.3 V\_Layer 2: Application Transactions

V\_layer 2 provides the abstraction needed to carry out the operations of the SoC from the application point of view. This is the first layer that can reflect the actual services required by the application. Again, these services are defined from the application point of view and not from the test point of view. They also include the checking of results. With results checking built into the application transactions, an abstraction is provided that enables the test developer to focus solely on creating

the right stimulus, which can be either explicit or pseudo random based. This assures that once a transaction completes, the SoC also has processed it correctly.

For example, for the Security Processor chip, the SoC under test provides well-defined encryption services. Data encryption is handled by packaging the data inside a descriptor (similar to a packet). Descriptors can be dynamically dispatched to any one of the 9 channels, or statically dispatched to a dedicated channel. Therefore, from the application point of view, the application transactions consist of:

#### SET 1:

- Create & send a dynamic descriptor (Checking implied)
- Create & send a static descriptor (Checking implied)

When one does not include the checking in the application transaction, one gets a set of functions on a lower level of abstraction, for example:

#### SET 2:

- Create dynamic descriptor
- Create static descriptor
- Send descriptor
- Check dynamic descriptor
- Check static descriptor

This set of functions results in a situation where all of the complexities of checking the behavior of the SoC are moved up into the test, and therefore does not provide a clear self-containing abstraction. Having the checking embedded in the application transaction offers a clear interface that enables the test developer to focus on generating stimulus to the DUT.

# concepts

Unlike the bus-based transactions of V\_Layer 1, the transactions in V\_Layer 2 do not have a 1-to-1 correspondence with a HW unit. Instead, the application transaction is an abstraction that is carried out by the whole DUT and often the whole testbench. In the encryption SoC example, the create & send & check descriptor single transaction would generate a series of 16 PCI write operations that would be issued by accessing bus transactions from V\_Layer 1.

### 2.1.4 V\_Layer 3: Scenario and Stimulus

This V\_Layer provides high-level interface to configure the DUT, testbench, and the test. It also enables stimulus generation (manual or automatic streams). Streams are defined in terms of a series of application transactions (V\_layer 2 transactions). The test writer has control over various streams of application transactions by applying constraints to each stream, and/or using inter-stream synchronization based on system events.

Inter-stream synchronization is essential for defining complex test scenarios where the engineer needs to synchronize the flow of a stream with one or more activities in the DUT, or in another stimulus stream. This synchronization is done by System Events where the engineer can use well-known system events in the design. For example, an event can be defined by the PCI monitor to announce detection of bus abort.

This layer supports built-in self-checking at the inter-stream level. This stream-level self-checking augments the self-checking that takes place in each of the application transactions defined in V\_Layer 2.

### 2.2 The Testbench Architecture

With the above description of the intended functionality of the V\_Layers in mind, Figure 3 shows a general architecture of the V\_Stack. The figure

shows the objects that are typically present in each of the layers. The User Test is part of the object called Scenario Manager. Note how the User Test only interacts with the entire testbench

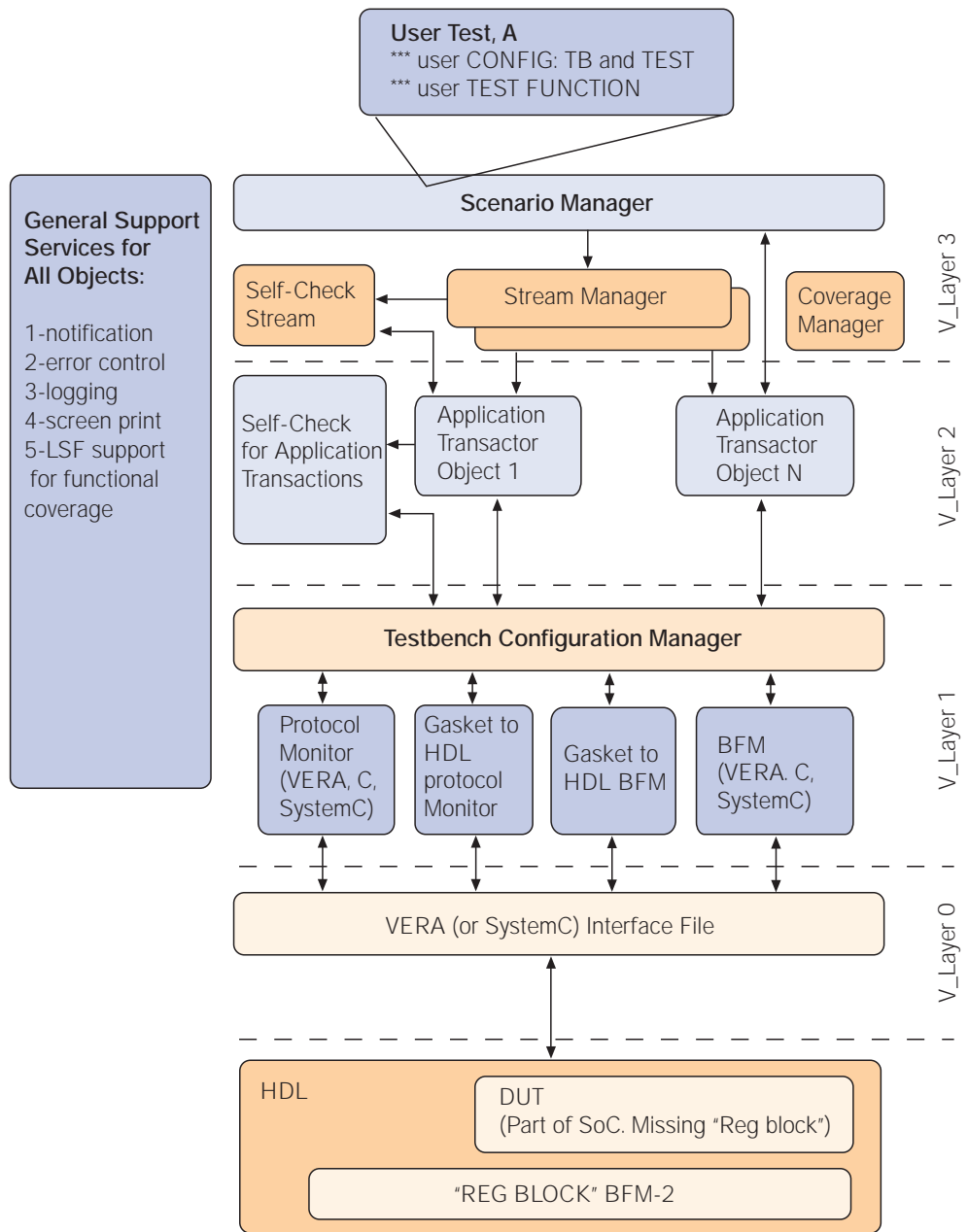


Figure 3. A Proposed Testbench Design that Takes Advantage of the Layered Approach.

# concepts

environment by the Scenario Manager. This object is the main user interface to this testbench environment. The User Test usually consists of two short sections: the configuration section (a set of knobs to adjust) and the test function.

At simulation runtime, based on the user configuration, the Scenario Manager instantiates all the objects necessary to carry out the test. Inside the test function, the primary activities are

- Issue application transactions
- Control streams of application transaction
- Request notification to any system vent

This architecture has been successfully used on two different projects, and other projects have begun to deploy it. Readers interested in a more in-depth treatment of the concepts presented and the proposed testbench architecture can refer to the web for a [white paper on this architecture](#).

## 3 Conclusion

The proposed testbench architecture is not an out-of-the box solution for SoC verification. It is however a systematic blueprint that substantially reduces the design and implementation time of the testbench. (Our experiences find more than a 25-percent reduction). Specifically, the architecture includes:

- The Scenario Manager (V\_Layer 3) that reduces the effort of creating tests.
- The TCM (in V\_layer 1) that allows re-use of sub-system tests at system level testing.
- The distributed self-checking techniques (V\_Layer 1, 2, and 3) to make it easier to isolate bugs.
- The functional coverage manager that provides the mechanism to track functional behavior against detailed test plan.
- The support services (with object-oriented programming techniques) which maximize the benefit of re-use across multiple projects.

This layered testbench architecture can be adapted to fit most digital designs. To gain the most benefit in effort savings and maximize re-use, the use of a powerful HVL like VERA with object-oriented capabilities and powerful concurrent thread (and corresponding synchronization) capabilities is highly recommended.

# concepts

## Mixed-Language Simulation with VCS

Here is a common problem that might be coming to your desk sometime soon. As a Synopsys VCS™ user, you have just received a new IP from your vendor. However, it's delivered in VHDL and no Verilog version is available. As usual, you are on a tight project schedule and learning a VHDL simulator to verify the IP does not look attractive. How do you simulate a VHDL IP in the quickest way possible?

The solution is VCS with its new extended capabilities. Along with Synopsys VHDL Simulator Scirocco™, VCS now enables you to simulate mixed-HDL (mixed-language) designs. With this solution, you have the convenience of staying in your existing verification environment, which helps reduce your verification risks.

You also maintain the same superior performance of VCS. The only thing you need to run mixed-language simulation is to have access to a Scirocco license in addition to your VCS license. This article covers the three basic steps to simulate a mixed-language design with VCS:

- Simulator Setup
- VHDL Analysis
- VCS Compilation and Simulation

Each of these steps above is covered in more detail below with further explanation and examples. Also covered in this article is debugging, performance, and VCS mixed-language integration with other tools.

### Setup for VCS Mixed-Language Simulation

To use VCS for mixed-language simulation, you setup VCS as normal. The following is a quick overview of the commands used for mixed-language simulation setup in VCS. For more

detailed information on these commands and steps, see the VCS and Scirocco user guide and reference manuals.

```
setenv VCS_HOME <vcs_install_dir>
set path = ($VCS_HOME/bin $path)
```

In addition, you need to setup the Scirocco VHDL simulator

```
setenv SYNOPSIS_SIM <sc_install_dir>
source $SYNOPSIS_SIM/admin/envIRON.csh
```

To simulate a mixed-language design, you also need to setup a Scirocco license. This procedure follows the Synopsys Common Licensing Program (CLP) for Scirocco (and VCS). For more information on this step, see additional information documented in the Common Licensing Program user guide.

```
setenv SNPSLMD_LICENSE_FILE/
<path_to_license.dat or port@host>
```

### VHDL Libraries

In addition to setting up the simulators, you also need to set up the VHDL design environment for correct simulation, which includes setting up the libraries. The following is a brief overview of the process.

In VHDL, there is a concept of logical libraries and physical libraries that must be understood for successful simulation of your IP in VHDL. Logical and physical library settings are made in the `synopsys_sim.setup` file described below.

The logical libraries in VHDL function like containers for you to pick up components (similar to modules in Verilog). Logical libraries are means of grouping various components together in an intuitive way. The default logical library is WORK. Physical libraries on the other hand are directories where the Scirocco VHDL analyzer, `vhdlan`, keeps intermediate files. This is similar to the concept of `csrc` directory that VCS creates.

Setting up the libraries involves only a few steps:

1. Check for the logical libraries

These are specified in the logical library statement in the VHDL source file for your IP. If your IP is delivered in multiple files, you'll need to check the library statement in each file. The statement is found at the beginning of the file and looks something like the following:

```
library my_asic123;
```

2. Check to make sure that the physical library directories exist. Create them if necessary.

3. Map the logical libraries to the physical directories

The logical to physical library mapping is specified in the `synopsys_sim.setup` file. Make sure you have `synopsys_sim.setup` setup file in or linked to the current working directory. Note: You do not have to map default logical libraries to physical logical directories. These are IEEE, STD, or GTECH libraries.

The following is an example of a logical-to-physical mapping entry in `synopsys_sim.setup` (setup file).

```
WORK > DEFAULT
DEFAULT : ./work
my_asic123: ./xyz/my_lib1
```

The last line specifies mapping the logical library (`my_asic123`) to the physical directory (`./xyz/my_lib1`).

In addition to logical-to-physical mapping, you can also specify other settings for VHDL simulation. For example, you can set the timebase for VHDL as follows:

```
timebase = ns
```

# concepts

Once you have setup your libraries, you are now ready to analyze your VHDL IP code.

## VHDL Analyze

After setting up your simulators, you are ready to analyze the source of your VHDL IP. In VHDL, by default the source files are analyzed, in a directory called WORK, which was created in the library setup mentioned above.

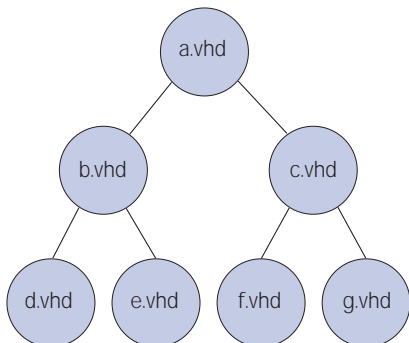
In general, VHDL requires that source files need to be analyzed in a certain hierarchical order, starting with source files describing leaf-level modules, then moving up to the source files describing the top-level module. This is different than Verilog compilation, where the user gives the entire netlist on the compile (or analyze) command line. In practice, follow the order specified by your IP provider to analyze your VHDL source files. The command that you use to analyze the VHDL source files with Scirocco is `vhdlan`.

### vhdlan VHDL\_Source\_files

There are options available for `vhdlan`, which are documented in the Scirocco Users Guide.

As mentioned above, VHDL requires a hierarchical order from leaf-to-top level. Note that this is contrary to Verilog

```
vhdlan a.vhd, b.vhd, c.vhd, e.vhd, f.vhd g.vhd
```



compilation, where the user gives the entire netlist on the compile (or analyze) command line. VHDL compilation requires that you analyze (using the `vhdlan` command) all source files, starting from leaf-level components up to the top-level module (see figure 1).

The following commands and code example uses a VHDL design entity DEC and analyzes it into default logical library **WORK**.

```
vhdlan DEC.vhd
```

The following is the VHDL IP code:

## VCS Compile and Simulate

Once your VHDL source files have been analyzed, you can now follow the VCS flow to compile and simulate your complete design. Your VHDL IP is instantiated in your Verilog design just as a Verilog component would. By default, VCS will pick up the pre-compiled VHDL component from the logical WORK library.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DEC is
port (A, B, ENABLE: in std_logic; Z: out std_logic_vector(0 to 3));
end DEC;

architecture DEC_SEQ of DEC is
begin
process (A, B, ENABLE)
variable ABAR, BBAR: std_logic;
begin
ABAR := not A; BBAR := not B;
if ENABLE = '1' then
Z(3) <= not (A and B); Z(0) <= not (ABAR and BBAR);
Z(2) <= not (A and BBAR); Z(1) <= not (ABAR and B);
else
Z <= "1111";
end if;
end process;
end DEC_SEQ;

configuration CDEC of DEC is
for DEC_SEQ
end for;
end CDEC;
  
```

# concepts

In order to simulate a mixed-language design you need to specify the VCS option **-mhdl** at compile time. For example, to compile this design with a Verilog testbench, a Verilog design under test (DUT), and a VHDL IP run the following

**vcs -mhdl top.v**

```
//The following is top.v
`timescale 1 ns / 1 ps;
module top;
wire [0:3] zz;
reg aa, bb, ee;

DEC i1(aa, bb, ee, zz); //instantiation of the VHDL DEC component

initial
begin
aa = 1'b0; bb = 1'b1;
repeat (10)
begin
#10 aa = ~aa; bb = ~bb; #5 ee = 1'b1; #2 ee = 1'b0;
end
end
initial
begin
$monitor($time,,aa, bb, ee, zz[0], zz[1], zz[2], zz[3]);
$dumpvars;
$vcddpluson;
$vcddplustraceon;
end
endmodule
```

You can use the advanced Scirocco options for the elaboration of the VHDL IP, (invoked during VCS compile phase), with

**-vhdllelab "VHDL elaboration options"**

Advanced Scirocco options for the simulation of the VHDL IP is also available with

**-vhdlrun "VHDL run-time options"**

Note: Use of double quotes around the above options is required. For a list of options see the Scirocco Users Guide.

## Debugging Verilog and VHDL

For your mixed-language design debugging, you have access to the same capability as with pure Verilog designs. You can debug your mixed-language design both in interactive and in post-processing mode in the VCS Graphical User Interface. You have full visibility into both portions (Verilog design and VHDL IP) of the design and can use the same GUI or CLI prompt to debug and analyze the entire design.

The VCS common textual debug environment, (UI), commands work on the entire design seamlessly for hierarchy traversal, signal and line breakpoints, and display/

setting of values of HDL signals, nets, or variables. You can use the same set of commands you are familiar with to analyze Verilog and VHDL blocks. With the *changemode* option at the UI prompt, you can change from the VCS command line interface to the Scirocco command line interface.

In the VCS GUI, you can view both VHDL and Verilog blocks and can set all nets in Verilog and VHDL. You can use the interactive GUI for the entire design by using **-RI** option. This allows you to set break points in either Verilog or VHDL sources, assuming correct use of options for line tracing in both the flows. Also you can drag and drop signals from both hierarchies.

## Performance

Simulating a mixed-language design with VCS, you maintain the same superior performance VCS has for pure Verilog designs. You have access to all optimizations available with VCS (such as radify with **+rad**).

In addition, if your IP is described at RTL you can benefit from the superior run time performance of cycle optimizations that provides unsurpassed performance for your regression.

## Integration with other tools

If your complete verification environment goes beyond the VCS Verilog simulator, you do not have to modify it when you verify mixed-language designs. In mixed-language mode, VCS supports the standard interfaces that are available for pure Verilog designs:

- VERA
- CoverMeter (Verilog portion supported only for this release)
- PLI and VHPI

There is no need to give up your complex verification environment when you plug in a VHDL IP into your Verilog design. If you have a VERA testbench, it will work correctly with your design once you instantiate your VHDL IP. In mixed-language simulation in VCS, CoverMeter is available as usual to verify the coverage of your Verilog code.

# concepts

If you have developed a C environment around your simulator using PLI, it will continue to work properly in your mixed-language environment. You also have access to the Scirocco C language interface VHPI (IEEE standard for VHDL API) to drive the VHDL portion of design.

PLI is supported fully in mixed-language simulation. If you need to access VHDL signals from your PLI application, Synopsys provides a Mixed-language API, which allows you to access your Verilog nets from PLI calls and VHDL signals from VHPI providing complete control on the entire design. These publicly available routines allow you to obtain information whether the probed net is a VHDL object or a Verilog object, so that you can call the appropriate VHPI or PLI routine.

## Links to More Information

- [Mixed-HDL white paper](#) and data sheet are available on the Synopsys web site.
- [SolvNet VCS Mixed-HDL Articles](#)
- [Synopsys Documentation](#) (with part numbers)
  - [Scirocco Reference Manual](#) (37081-000 JB)
  - [VCS/VCSi Documentation Package](#) (34043-000 KB)
  - [VCS/VCSi User Guide](#) (34174-000 KB)

# concepts

## Synchronous Coding Styles For Maximum Performance

Today, VCS™ is used for a wide variety of design types. Coding styles may differ between FPGAs, ASICs, and System-on-a-Chip designs, but most are dominated by synchronous logic. This means clouds of combinational logic surrounded by banks of flip-flops or latches. If VCS understands the coding style used for these synchronous devices, it can be much more aggressive at optimizing the entire design. This article will describe common coding styles for many synchronous devices and show how minor changes can have big effects on simulation performance.

### VCS Optimizations:

VCS has many algorithms to speed up simulation and many of these algorithms key on specific coding styles to achieve greater speedups. Over the years we have added to these algorithms to cover the most commonly used coding styles. These algorithms are split into two camps. The first are language and front-end optimizations that optimize logic to be evaluated and thereby minimize events in the event queue. All event-driven simulators use an event queue to schedule and propagate events. Event queues use a methodical process which makes them very flexible for any type of code, like asynchronous behavior, but they are inherently slow due to the overhead of this flexibility.

The second type of optimization within VCS is called Cycle optimizations. This optimization can be thought of as cycle-based algorithms that are applied to synchronous logic. Many synchronous devices only need a clock edge to propagate events so they do not need the true flexibility of the event queue. The concept is to minimize the flexibility for these types of constructs in order to get a bigger runtime speedup.

Cycle-based algorithms are on by default (i.e. no special compile-time flag needed). A much more aggressive family of optimizations within VCS, called High Level optimizations, can result in larger speedups, but are not the focus of this article. Cycle optimizations are enabled with the VCS compile switch +rad.

### Flop coding styles:

It is very important that the coding style of synchronous elements in the design are race free and coded in a way that VCS can understand them. If VCS recognizes the latches/FFs in the design, then it will automatically turn on a Cycle optimization and provide additional speedup. VCS will accelerate most flip-flop coding styles accepted by Design Compiler. Details are documented in the on-line VCS UsersGuide. Here are the general rules for coding synchronous devices in VCS.

\* Standard flip-flop coding looks like:

```
always @(posedge clk)
  a <= b;
```

This is a perfect coding style for VCS. Non-blocking assignments (NBA) ensure no race conditions exist between flip flops, and since there is no delay, VCS can simulate this flop using cycle-based algorithms.

\* Adding a delay slows down evaluations:

```
always @(posedge clk)
  a <= #1 b;
```

This example adds an unnecessary delay on the right-hand side of the NBA. Some users like to use this delay so waveforms are staggered and you can see which outputs are caused by the flops. Others believe that the delay helps avoid race conditions. This is not true. VCS' propagating algorithm guarantees that all NBA's execute in the

correct order. Therefore, delay is not needed. The main problem with this delay is that it inhibits VCS from using cycle-based optimizations. When VCS sees the delay, it assumes that the designer used it because some other logic relies on the delay to work properly (asynchronous feedback).

For regressions, you can force VCS to ignore any delay on the right-hand side of NBA's by using the compile-time flag: vcs +nbaopt This will allow more RoadRunner optimizations and result in a good speedup, but may expose race conditions if NBA's were used in inappropriate places such as within initial blocks.

\* Blocking assigns without delays are prone to races:

```
always @(posedge clk)
  a = b;
```

This coding style is prone to race conditions. The IEEE LRM states that all always blocks act in parallel and you cannot guarantee the order of execution. Therefore you cannot guarantee that b will or will not be updated before it is propagated to a.

\* Blocking assigns with delays are not recommended:

```
always @(posedge clk)
  a = #1 b;
```

Blocking assignments do not ensure proper ordering of events in daisy-chained flip-flops, so they require a #1 on the RHS to avoid race conditions. Since this inhibits VCS cycle-based optimizations, this coding style is not recommended.

# concepts

\* Adding asynchronous resets

```
always @(posedge clk or negedge rst)
begin
if (rst == 0) a <= 0;
else a <= b;
end;
```

Adding resets to flops generally will not adversely affect VCS optimizations.

## Latch Coding styles:

There are many ways to code a latch in Verilog. These examples all work well and will be properly accelerated in VCS:

```
1. always @(clk or d)
if(clk) q <= d;
2. always @(clk or d)
q <= clk ? d : q;
```

## UDPs coding style

Many vendor supplied UDPs are well written and thus VCS can infer the latches and flops properly. Sometimes inefficiencies arise when a user only includes table entries for particular input combinations of her interest. Uncovered input combinations in a UDP definition is defined by the Verilog LRM to output X. Since VCS needs to conform to all the possibilities of the LRM, this can result in a non-optimized UDP. If VCS is unable to recognize the clock input to the UDP, then performance can be significantly diminished. The preferred coding style is to always define "no change" explicitly.

## Use the VCS built-in profiler

Since synchronous devices are often instantiated thousands of times in a

typical design they can have a major impact on performance. If a flop model does not get accelerated in VCS it will show up high in the profile list. It is recommended to occasionally compile and run VCS with the flag `vcs +prof` to see which constructs are taking the most CPU time in the simulation. This is often the best way to find an offending module.

## Clock Drivers

There are many ways to drive clocks in the Verilog language. VCS is usually happy with any coding style used. Since VCS cycle-based optimizations occur at the block level, there is no performance penalty for having multiple clocks (even if they share paths or are asynchronous). An important rule to follow is this: clock signals should never be driven by a non-blocking assignment. This nullifies the benefits of using NBAs inside of flops and is prone to race conditions.

## Coding styles to avoid

A few inefficient constructs usually don't impact the performance of a Simulator; i.e. use whatever you need on a local level to achieve the needed functionality. But blocks that get instantiated thousands of times or always blocks that get triggered thousands of times have a much larger impact on overall performance.

## Other coding styles to avoid:

- 1) Use of delays inside always blocks
- 2) Use of signal display/monitor inside always blocks that are not really necessary; i.e. a display to flag an ERROR message is ok but displaying signal values at every edge can hurt performance.
- 3) Use of fork/join constructs
- 4) Force/release assign/deassign
- 5) Multiple event controls - multiple "@" or triggers inside the always block
- 6) Use of verilog task calls that have event/delay control in them.
- 7) Use of memories inside always blocks
- 8) Don't drive a clock with an NBA
- 9) Avoid modeling flops with transistors
- 10) Avoid any strength-based modeling
- 11) Avoid named blocks

## Summary

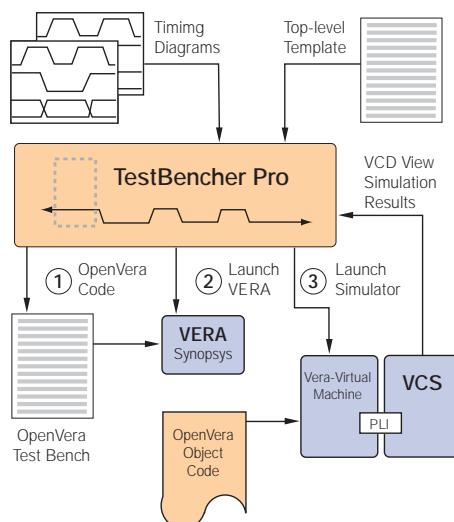
Synopsys is striving to make it easier for you to realize the maximum speed of VCS. By following these simple coding rules, VCS will aggressively optimize your design to achieve faster simulation and avoid race conditions.

## OpenVera Graphical Code Generation

TestBencher Pro™, by SynaptiCAD, provides designers with a graphical environment for generating OpenVera™ cycle-based or time-based bus-functional models from language-independent timing diagrams. TestBencher Pro's graphical interface speeds up test bench development for both expert and novice users. TestBencher generates all of the low-level transaction code, verification code, sequence detection, error reporting and file I/O code. The graphical representation also enhances the ability of engineers to share data across projects, even though new engineers might not be familiar with the OpenVera language. In addition to generating the OpenVera code, TestBencher can also serve as a graphical development environment for VERA® and your VHDL and Verilog simulators.

### System Level Design

TestBencher Pro generates the entire OpenVera test bench using graphical timing diagrams, information extracted from the model under test, and the top-level test bench file. The only code that the user writes is at the system level; all of the other coding details are automatically generated. The system level code specifies the order and logic in which to apply the timing diagram transaction calls to the model under test. TestBencher can automatically generate the transaction calls and the OpenVera data structures needed for the transactions, so users can concentrate on the functionality of the top-level test-bench. The generated test benches are compiled with the Synopsys' VERA tool and simulated using all major VHDL and Verilog simulators.



TestBencher OpenVera Design Flow

### Graphical Samples Generate OpenVera Expects

TestBencher Pro can generate state and timing protocol checking code for verifying the response of the model being tested. The protocol checkers use TestBencher's graphical samples that can verify a sequence of events or check for either a state change or for stability during a sampling window. These graphical samples generate OpenVera restricted, simple, and full expects. The type of OpenVera *Expect* generated depends upon the underlying drawn waveform. Defining an *Expect's* behavior this way makes it very easy to read a timing diagram and to be sure that the functionality is conveyed.

### OpenVera Advanced Data Structures Supported

TestBencher Pro's generates the code for the definitions and instances of OpenVera User *Defined Types* (UDTs) which can contain queues, arrays, associative arrays, and single elements of basic OpenVera types and other UDTs. Each field can be optionally defined as random, so that the OpenVera randomize function can fill the packet with data.

Each field can also be defined as static so that one memory location is shared by several instances of the class. OpenVera's *Mailboxes* are used to implement queues, so all of the functionality of *Mailboxes* is made easily accessible. Information about a data structure is entered once and then TestBencher can use the information to pass data to transactions or files.

### VERA Graphical Interface

TestBencher Pro can control external simulators through its graphical interface, so that compilation and simulation of the project can be handled without having to exit TestBencher. This is particularly useful for OpenVera because multiple tools are needed to compile and simulate a project. TestBencher can automatically launch VERA to compile the test bench and then launch your VHDL or Verilog simulator with the correct commands to dynamically link to the OpenVera compiled code. TestBencher can handle remote execution of VCS and Vera running on different computers and even under different operating systems. This means you can use a Windows PC as your desktop machine and transparently run your simulations as batch jobs on a Solaris box.

### Summary

The innovative combination of TestBencher Pro and VERA delivers a consistent and quick solution to the most demanding verification problems. This integration allows designers to focus on the verification of the device under test and not on the implementation of the test bench. Contact [SynaptiCAD](http://www.synopsys.com) for an evaluation license of TestBencher Pro.

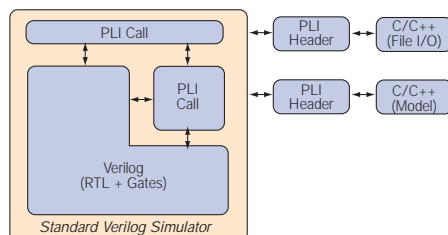
## Using C/C++ Functions In The VCS Verification Flow

Verifying today's complex ASICs and systems-on-a-chip (SoCs) takes up a major portion of the design cycle. Synopsys helps customer's speed up the verification process by relentlessly providing new technology updates to its simulation tools. While these updates speed up the simulation, they often contain new features that shorten the verification process by other means. The latest release of VCS™ contains a new capability called VCS DirectC. VCS DirectC is a unique interface that allows VCS users to quickly embed high-level C/C++ functions within a Verilog design description. It enables design team that use C or C++ to model portions of their design, to easily embed these C functions in their Verilog simulation environment.

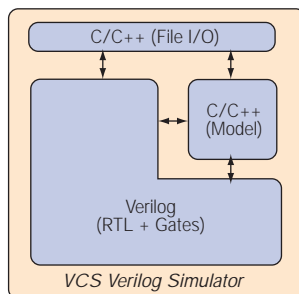
This interface minimizes the risks and learning curve typically associated with mixing C/C++ and Verilog in the verification flow.

Until now, hardware designers who wanted to use C/C++ functions during Verilog simulations relied on the Verilog Programming Language Interface (PLI).

Design teams developing PLI applications are painfully aware of the difficulty of creating C/C++ interfaces to Verilog simulators and the scarcity of hardware designers who have such PLI experience. PLI application development is far from user friendly; it often takes more time to debug the PLI interface than to debug the integrated code itself. In addition, PLIs add a layer between the external user code and the simulation kernel, which creates additional overhead and slows down the simulation. Nonetheless, PLIs are popular because of their flexibility, security and portability.



To address these issues, Synopsys developed the VCS DirectC interface. VCS DirectC has been successfully used in a wide range of designs such as microprocessor, image compression and networking applications. Unlike other Verilog/C/C++ solutions, users do not need any PLI knowledge to be productive with VCS DirectC. The interface extends Verilog functionality while avoiding the design and performance overhead associated with PLI.



It is important to note that the VCS DirectC interface is not a PLI replacement. VCS DirectC does not provide synchronization capabilities to the added C/C++ code and does not offer the ability to walk through the Verilog design hierarchy. Instead, the interface serves as a complement to the PLI. Verification engineers can use VCS DirectC instead of PLI wherever C/C++ functions are needed.

### VCS DirectC Applications

VCS DirectC implements a very efficient communication link between C/C++ and Verilog that avoids the

overhead introduced by the PLI. In VCS, C/C++ functions are referred to as external functions and are declared using the new VCS keyword `extern` in the Verilog function declaration. VCS treats external functions just like Verilog functions in that no simulation time passes while they execute.

C/C++ functions can be invoked anywhere within the Verilog code. It is possible to get a value returned by an external function (even a pointer to a structure) and pass it to another external function: there is no need to modify the Verilog syntax to add powerful and useful constructs.

In addition to unleashing the power of extensive C math libraries, VCS DirectC provides easy access to many applications that are traditionally difficult to achieve with the Verilog language: for example string manipulation and file I/O are common operations easily and efficiently implemented with `cfunctions`.

### A Faster and Debugged Verification Environment

In this section we will show how the VCS DirectC Interface can be used to speed up the development of the verification environment.

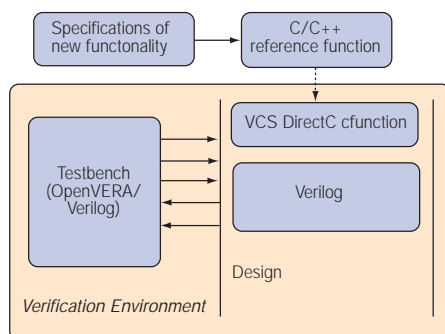
Today most verification flows take advantage of the use of behavioral models in order to expose parallelism between the design and testbench creation tasks: this methodology is known to help decreasing the overall development cycle (that is, finding more bugs in less time).

Behavioral models can be written quickly and in general simulate faster than their equivalent synthesizable counterparts: it takes more time to develop and more computational efforts to simulate the hardware implementation details of the latter.

# concepts

Accurate behavioral models of new design functionality could be easily and quickly devised by adopting VCS DirectC.

In this example, we assume that the design's architecture and verification environment are already established: the following diagram could represent the next generation of a given chip. As shown in the diagram, new functionality is added to the Verilog design in the form of a C/C++ representation.



With VCS DirectC, the C/C++ models that might have been created soon after the original specifications can be used as reference models to describe the new added design functionality. At this design stage these C/C++ models have already been thoroughly and independently debugged: they represent some of the behavioral aspects of the system well enough to be used now to debug the verification environment written in OpenVera™, Synopsys' Hardware Verification Language.

We are thus able to debug our testbench independently from the availability of the RTL representation of the design.

By the time the RTL design is released and integrated in the verification environment, the likelihood of finding design bugs (not testbench bugs) is higher since, thanks to VCS DirectC,

the testbench has already been debugged using the C/C++ reference code.

VCS DirectC saves valuable development time because it uses existing C/C++ reference design models to ensure that both the testbench and the design's hardware implementation are consistent with the original specification.

## Using the VCS DirectC Interface

Using VCS DirectC involves three simple steps:

- 1) Write C/C++ code to develop the desired function and verify it.
- 2) Invoke the appropriate C/C++ function from anywhere within the Verilog code.
- 3) Compile VCS using the new +vc flag.

Suppose a designer wants to use the functionality of a C math library in a Verilog design. With VCS DirectC this can be done in a few steps.

1. In the example below, the user first creates a C/C++ function with the desired functionality.

```

// ipow.c (This is pure ANSI C)
#include <math.h>
#define MAX_INT 32767
#define MIN_INT -32768
int ipow (int a , int b) {
double c;
  c = pow(a,b);
  if((c < MAX_INT) & (c > MIN_INT))
    return((int)c);
  else {
    printf("C: overflow for %d ** %d == %f\n", a,b,c);
    return(MIN_INT);
  }
}
  
```

2. After creating the C/C++ function, it is called into the Verilog environment with the new extern keyword. This is shown in the example below.

```

// Verilog code to instance the C function
extern int ipow(int a, int b);
// C function define within Verilog source
module tb_ipow;
  parameter N=100;
  integer i,j;
  initial
    for(i=-N; i<N; i=i+1)
      for(j=-N; j<N; j=j+1)
        $display("%d", ipow(i, j) );
endmodule
  
```

3. After creating the C function (ipow.c) and Verilog (ipow.v), users just compile VCS by typing

```
VCS -R +vc tb_ipow.v ipow.c
```

The entire process of using VCS DirectC to leverage the functionality of a C/C++ library such as math.h takes a short amount of time to complete. The resulting code is very easy to read and to maintain. The elimination of the cumbersome PLI layer results in better performance. The following table illustrates the performance difference between the DirectC and PLI implementations of the example above: in this case while varying the parameter N DirectC performs consistently twice as fast than PLI (tf routines).

# concepts

No. of \$display calls	10K	1M	25M	100M
PLI	0.7	13	350	1260
DirectC	0.6	7	160	660

When necessary, design engineers can easily convert many PLI applications to VCS DirectC to gain performance and enhance maintainability. In a specific application, a designer was able to double the simulation performance of file I/O operations in just 15 minutes by using VCS DirectC instead of a PLI.

## Conclusion

Today's complex designs and tight time-to-market windows make the verification bottleneck a growing concern. Speeding up the verification process, without affecting quality of results, is therefore of paramount importance. With VCS DirectC, designers can take advantage of C/C++ functions without the complexity and performance overhead of PLI.

# concepts

## Coverage Metrics Built Into VCS For Smarter Verification

With gate counts soaring and time-to-market windows shrinking, ensuring that your design is completely verified is more important than ever. Though coverage metrics are the best way to ensure that all of your design has been exercised, traditional coverage tools add overhead to simulation run time performance. In addition, code coverage tools have been somewhat awkward to use because they are linked to the verification environment through programming language interfaces (PLIs), which further slow down the simulation process. With many companies facing verification as the bottleneck in the flow, designers often decide to avoid coverage analysis in exchange for meeting verification schedules. Those who skip using coverage metrics often pay the price with costly respins and missed market opportunities.

Aware of this dilemma, Synopsys has seamlessly integrated coverage metrics capabilities into the latest version of its Verilog simulator, VCS™ 6.0.1. The feature, called VCS coverage metrics (VCM), is a standard part of VCS, so users will now have access to comprehensive built-in coverage analysis, including condition, toggle, line and finite-state-machine coverage. Using these capabilities built into the VCS engine, design teams can easily determine the quality of coverage of their verification tests and realize the benefits of smarter verification. With this built-in capability, designers only need to compile once to run both simulation and coverage analysis. As a result of this single compilation, users will see faster compile and run-time performance than with previous point tool solutions that used the Verilog PLI.

Because overhead is minimum, users can run VCM throughout a design cycle without significant performance impact, in every VCS simulation and all its phases. Furthermore, the use of a single VCS command line, both to instrument (an additional step normally required with a separate coverage tool) the design and compile it, makes coverage with VCS a seamless step in the simulation process. VCS's incremental coverage automatically makes sure that analyzed portions of a design do not get monitored again in subsequent coverage runs as the design evolves, further reducing any overhead and eliminating the need for any re-instrumentation to only monitor portions of code not already covered.

Using the coverage metrics feature requires using only a few switch options in VCS. Coupled with little runtime overhead, VCS users can now routinely ensure that all portions of their designs are verified.

### What VCM Does

VCS provides four coverage metrics, which are comprehensive as well as exhaustive in their analysis of a design. Each metric not only varies with respect to the aspect of a design it addresses, but also with respect to the phase of the design cycle in which it is used. This variation of metrics ensures appropriate use of coverage in every phase of the design cycle. The metrics are statement coverage, condition or expression coverage, toggle coverage, and finite-state-machine (FSM) coverage. The chart below indicates how the metrics provide coverage in all phases of the design cycle:

Phase/Metric	Statement	Condition & Expression	Toggle	FSM
Behavioral	X	X		X
RTL	X	X	X	X
Gate			X	

**Statement Coverage:** This indicates which statements of a design are not executed during simulation. Statement coverage provides much useful information very quickly with little simulation overhead. Statistics are provided that indicate the total number of statements, blocks and branches executed. An annotated listing of a design is provided, pin-pointing those portions of the design that did not get executed.

**Condition/Expression Coverage:** This ensures that conditional statements in a design are fully tested. Once statement-coverage analysis indicates that a majority of the statements in a design have been executed, condition coverage can then be used to make sure the conditional paths are being executed for the correct conditions. VCM supports multiple levels of condition coverage: sub-expressions with regard to a sensitizing or exhaustive set of vectors, as well as event triggers. Path Coverage is extracted by combining condition-coverage data with that of statement coverage.

**Toggle Coverage:** This is typically used at the gate level to monitor nodes in a design for any toggle activity. Both half (1 → 0 or 0 → 1) and full (1 → 0 and back, or 0 → 1 and back) toggle activities are reported. Statistics are generated on a module-by-module basis for toggle activity of nets and registers. Missing toggle activity can provide conclusions about inactive elements and unexecuted portions of the design, and statistics produced for each module can be examined quickly to determine areas of low coverage. A special metric, average toggle per clock cycle, can also be applied to a module or hierarchy of choice for an early estimate of power consumed by that module or hierarchy. A glitch elimination mechanism is available to ensure suppression of glitches or false toggles.

# concepts

**FSM Coverage:** This statically extracts states of an FSM, transitions between these states, and possible sequences (reachability) between any two of these states. Dynamic coverage of transitions and reachability is then presented in the context of the statically extracted picture, giving the user a crystal-clear view of an FSM's functionality.

**Coverage in Regressions:** VCM has two features that make it indispensable to regressions: incremental coverage and automatic test grading. Incremental coverage ensures that analyzed portions of designs are not monitored again during periodic runs. Automatic test grading, on the other hand, grades coverage results of any given set of tests to obtain a minimum set of least-overlapping tests for a desired coverage level. It ranks tests for a regression environment by enabling running of tests with most coverage first and then those that cover difficult corner cases and provides additional coverage later, thus maximizing the incremental coverage. Automatic test-grading can also give designers early feedback on whether committed design changes have caused any new simulation failures, and can further be used to identify redundant tests that do not improve the overall coverage of the design.

## Use Mode for Simulation with Coverage

**Compile:** With the new tightly integrated coverage metrics, the traditional two steps of compiling the source for coverage instrumentation and then the instrumentation itself are removed. VCS compiles the code once and automatically builds the required monitors for coverage into the code and applies its high-performance optimization.

**Simulate:** Simulation generates and stores binary intermediate coverage data for later analysis.

**Analyze:** This phase uses the graphical user-interface (GUI) or generates ASCII report files to present the data to the designer.

VCS Coverage Metrics' GUI (denoted on the title as "cmView") brings your attention directly to portions of your design that did not get covered. Besides the pertinent statistics, cmView displays annotated Verilog design code listings for statement coverage, analysis of sub-expressions of conditional statements for condition coverage, and data on half, full, and missed toggle-activities of nets and registers for toggle coverage. Besides state listings of an FSM and a matrix view of its transitions, cmView has an especially innovative and intuitive sequence-analysis matrix for analyzing sequences between any two states of an FSM, a feature beyond the scope of a bubble diagram. It is also a tool for test grading of regression tests graphically. Users can customize cmView in terms of color schemes for highlighting specific coverage data, or in terms of weights to the various coverage types for ranking tests.

VCM provides flexibility. For example, a designer can choose to accumulate intermediate data from each test in a separate file and then merge the intermediate data files using cmView into a consolidated ASCII report file. This method also allows a user to choose any combination of tests for the consolidated report file. Another way is to merge intermediate data, as it is generated, from each test into a single intermediate data file.

With all of these coverage capabilities, VCS users can now ensure that their designs have undergone complete verification with the industry's premier Verilog simulation tool. Seamless integration with the VCS simulation engine means that coverage metrics are easy

to use, with little runtime penalty. VCS not only provides features that cover every aspect of a design, but also fits perfectly into the top-down design methodology, making coverage an integral part of the design cycle. Coverage metrics are but the latest advance in the continuing evolution of VCS. Synopsys is committed to constantly improving VCS and all of its other offerings. More capabilities, performance enhancements and ease-of-use features are coming in the future.

# q&a

## Questions & Answers

**Q** *What is the new VCS coverage metrics feature and how do I enable it?*

**A** Beginning with the new 6.0.1 version, VCS™ has built-in, full-featured coverage capability, including line coverage, condition coverage, toggle coverage, FSM coverage, and test grading. These new code coverage measurements are collectively referred to as "coverage metrics".

You need to simply add the switch +CM on the VCS command line to enable all of these coverages. Then, include one of the following to simv to generate coverage data as desired:

+CM+ALL	Specifies all four types of coverage.
+CM+COND	Specifies condition coverage.
+CM+DFLT	Specifies line, condition, and FSM coverage at compile-time.
+CM+FSM	Specifies FSM coverage.
+CM+LINE	Specifies line coverage.
+CM+TGL	Specifies toggle coverage.

You can specify more than one of the above options together, if required. Coverage reports can be generated and tests can be graded using the window titled "cmView", which is a coverage user interface that is now part of VCS. Please refer to "VCS 6.0.1 Release Note" and "VCM User Guide" for additional details.

**Q** *How do I take advantage of the new, tight integration between VERA 5.0 and VCS 6.0.1?*

**A** VERA 5.0 and VCS 6.0.1 contain unique integration, with internal access to each other's engines via the

VCS™ Direct Kernel Interface and other unique program communication. You can take advantage of this integration by using the new option +vera on the VCS command line.

The option +vera links the VERA objects with VCS at compile time, enabling VCS to exploit all possible optimizations for speeding the simulation including parts of the design which VERA accesses.

You can expect to see up to 2x overall speed-up when using +vera option with VCS 6.0.1 and VERA 5.0.

**Q** *I am simulating a mix of Verilog and VHDL. What kind of performance should I expect and what factors impact it?*

reduce code visible to the simulator and thereby increase simulation performance. Individual language semantics, (VHDL delta cycle vs. Verilog no delta), restrict the scope of most of these optimizations to a specific language. As a result, the more you mix and match the languages, the less that optimizations can be applied resulting in slower performance.

Therefore, in order to maximize performance, try to keep the interaction and boundaries between the languages to a minimum. Be careful regarding where and why you plug in code written in a different language. Try to restrict code written in second languages for large design blocks that are not easily rewritten. Mix the languages closer to the top of the design in order to allow more optimizations to be applied in each language region. Mixed-language simulation is still quite efficient if used for a few large IP blocks, compared to many modules such as plugging in some VHDL vital modules into a Verilog gate-level netlist.

**Q** *How is the VERA profiling feature used?*

**A** The VERA Profiling feature aids users in writing more efficient OpenVera™ code. When Profiling is turned on, VERA tracks the percentage of CPU time for each task, function, and program.

When the "+vera\_profile\_start" runtime option is used in a simulation, a profiling log file with the default name "profile.log" is generated. By default, a total of 500 task, function and program calls are individually tracked. At the beginning of the log file, the sum of CPU time spent in task, function and program calls above 500 is recorded. In the log file,

# Q&A

the 500 individually tracked calls are sorted by the percentage of CPU time usage in descending order. The profile data in the log file is organized as follows.

- sum of percentage of CPU time spent on task, function, and program calls above 500
- percentage of CPU time spent in the HDL simulator
- percentage of CPU time spent in OpenVera initialization
- percentage of CPU time spent in each task, function, and program calls up to a total of 500

Here is an example of a profiling log file.

38.10% of the time was spent in the task/function calls beyond the limit of 500.	
HDL Simulator	0.00%
VERA Initialization	2.41%
mem_wr	26.56%
mem_rd	20.35%
reset	0.16%

From the profiling log file, a user can identify the task and function calls that consume the majority of the simulation time and make modifications accordingly.

Three additional runtime options can be used to control the profiling feature.

#### +vera\_profile\_limit:

increases the limit of 500 individually tracked task, function, and program calls.

#### +vera\_profile\_filename:

replaces the default profiling log file name with a user defined name.

#### + vera\_profile\_sort\_by\_name:

sorts the 500 individually tracked task, function, and program calls by name alphabetically instead of by percentage of CPU time usage

Usage notes:

- 1) VERA Profiling cannot be used in conjunction with the VCS Profiling feature (+prof switch).
- 2) VERA Profiling does not report task, function, and program calls that consume less than 0.01% of the total CPU time.
- 3) The Profile runtime options may be specified in the vera.ini file.

### Q How do I take advantage of using the simplified UDF feature in VERA 5.0?

**A** The new +vera\_udf feature in VERA 5.0 simplifies the flow of hooking up VCS, VERA and C /C++ UDF code. A very important advantage to the +vera\_udf feature is that it enables using the new -vera switch in VCS 6.0.1, which has improved linking and simulation speed.

In prior versions of VERA, many steps were required to hook up and run UDF (C and C++) code. Customers would sometimes forget they needed to set an environment variable pointing to the UDF library (the UDF .dl file). These two issues are addressed in VERA 5.0 by simplifying the process and removing the need for the SSI\_LIB\_FILES environment variable.

In VERA 5.0 the old flow is still supported, but the new vera\_udf flow is

simpler and provides more flexibility as well. The UDF library no longer needs to be included the libVERA.a file. This means that the UDF library can be reused in future versions of VERA without needing to recompile each time using a different libVERA.a. The new vera\_udf flow allows customers to specify the filename of the UDF library on the command line of simulation. Furthermore, using more than one UDF library at the same time is now supported.

Assuming the VERA\_UDF\_Table[] is inside the c file with the UDF code, the flow is much shorter. All of the platforms have slightly different switches, but as an example, for Solaris, the compile and link commands are as shown below.

VERA 5.0 has also implemented a new wrapper in C to hook up VERA to UDFs. The old method of VERA PLI calls is still supported, but the new wrapper is much easier to use and requires less code.

An example of using the vera\_udf and the new C wrappers on all supported platforms (Linux, HP-UX, AIX, DEC Alpha, Solaris) can be found in the VERA 5.0 distribution in **\$VERA\_HOME/vum\_examples/example\_new\_udf**.

Documentation can be found in Chapter 14 of the VERA 5.0 User Guide.

```
cc -DVERA_UDF -K PIC -I. -I$VERA_HOME/lib -c udfdll.c
—or—
gcc -DVERA_UDF -fPIC -I. -I$VERA_HOME/lib -c udfdll.c
ld -G -z text udfdll.o -o udfdll.dl
simv +vera_load=udfdll.vro +vera_udf=./udfdll.dl
—or—
```

For VHDL simulators, you can add "vera\_udf=./udfdll.dl" to the vera.ini file

Multiple dl files can be used in a colon separated list, for example:

```
+vera_udf=./udfdll1.dl:./udfdll2.dl:./udfdll3.dl:./udfdll4.dl
```

# news and events

## Calendar

The following is a list of events that users can attend to learn the latest tips & tricks on Synopsys Verification.

Event	Dates
<a href="#">VERA® Workshops</a>	Ongoing
<a href="#">Verification w/VCS Workshops</a>	Ongoing
<a href="#">DesignCon, Santa Clara, CA</a>	January 28-31
<a href="#">DATE, Paris, France</a>	March 5-7
<a href="#">SNUG Europe</a>	March 7-8
<a href="#">HDLCon, San Jose, CA</a>	March 11-12
<a href="#">SNUG San Jose</a>	March 13-15
<a href="#">Verification Seminars</a>	February 19, 2001 - Irvine, CA February 21, 2001 - Santa Clara, CA February 26, 2001 - Dallas, Texas February 28, 2001 - Austin, Texas March 5, 2001 - Marlboro, MA March 7, 2001 - Ontario, Canada April 4, 2001 - Grenoble, France
<a href="#">Platform Users Conference, Dallas, TX</a>	May 5-8
<a href="#">SNUG Taiwan</a>	May 21-22
<a href="#">SNUG Korea</a>	May 24
<a href="#">SNUG Shanghai</a>	May 28
<a href="#">SNUG India</a>	May 30
<a href="#">SNUG Singapore</a>	June 4
<a href="#">DAC, New Orleans, LA</a>	June 10-14
<a href="#">MiniSNUG Dallas</a>	July
<a href="#">SNUG Boston</a>	September
<a href="#">SNUG Tokyo</a>	November
<a href="#">SNUG Osaka</a>	November
<a href="#">SNUG Israel</a>	December

### \*Share your experiences at SNUG!

SNUG is an open forum that provides Synopsys users the opportunity to exchange ideas, discuss problems and explore solutions. If you'd like to submit an abstract for any of the above SNUG events, please visit the [SNUG website](#).

\*For a complete listing of available workshops please visit [Customer Education](#) on the web.

# news and events

## News and Feedback

### News

[Synopsys Acquires Technology Assets From C Level Design, Inc.](#)

[Synopsys Integrates VERA and VCS to Boost Verification Performance](#)

[Synopsys VCS Verilog Simulator Incorporates Breakthrough Verification Capabilities](#)

[Synopsys' VCS Verilog Simulator Delivers Up to 5X Faster Performance](#)

[Synopsys Opens Vera Language to Provide an Open Platform To Unify The Verification Market](#)

[Intel pushes assertion language as EDA standard](#)

[Cognigine Sets High Design Standards for Its New Intelligent Network Processor™](#)

[Synopsys Launches OpenVera Catalyst Program](#)

[Synopsys Introduces Formality With Advanced Hier-IQ Technology](#)

[Verification Central and Synopsys Announce The Art of Verification with Vera, A New Book on Verification Using the OpenVera Language](#)

### Join the **OpenVera™ Discussion Forum** to:

- Keep abreast of the latest OpenVera developments and solutions
- Ask questions and discuss experiences
- Share new techniques and code
- Interact with OpenVera experts

### Documentation

NEW! Order *Art of Verification with VERA*, a user written book on verifying large complex SoCs with the Synopsys VERA testbench automation tool.

### Feedback

We welcome your comments and suggestions to improve future issues of Verification Avenue. If you would like to [subscribe](#), [unsubscribe](#) or [tell us what you think](#).