

# Employ advanced logging techniques for SystemVerilog

By Bindesh Patel

Technical Marketing Manager

E-mail: bindesh\_patel@springsoft.com

Amanda Hsiao

Technical Marketing Manager

E-mail: amanda\_hsiao@springsoft.com

Verification Group

SpringSoft USA

SystemVerilog provides an advantage in addressing the verification complexity challenge—not simply as a new language for describing complex structures, but as a platform for driving a more efficient, realistic test of the design. It is no surprise then that the adoption of the language for verification purposes has been rapid. However, there is a gap when it comes to the debug and analysis of SystemVerilog testbench code. The accepted “dumpvars”-based techniques are not practical for the software-like object-oriented testbench code, and their benefits in this realm are also questionable. But, at the end of the day, engineers do need to know what the testbench is doing at any given point in time. Thus far, engineers have been forced to revert to low-level, text-based message logging and subsequent manual analysis of the resulting text log files.

Logging—the process of recording history—has been widely used in systems and software environments. And most SystemVerilog libraries used today provide some built-in utilities for logging information from the testbench to low-level text files that can be analyzed after simulation. Engineers then manually correlate the testbench data to the design activity along the time axis. This is a painfully low-tech process with inherent disparity in the design

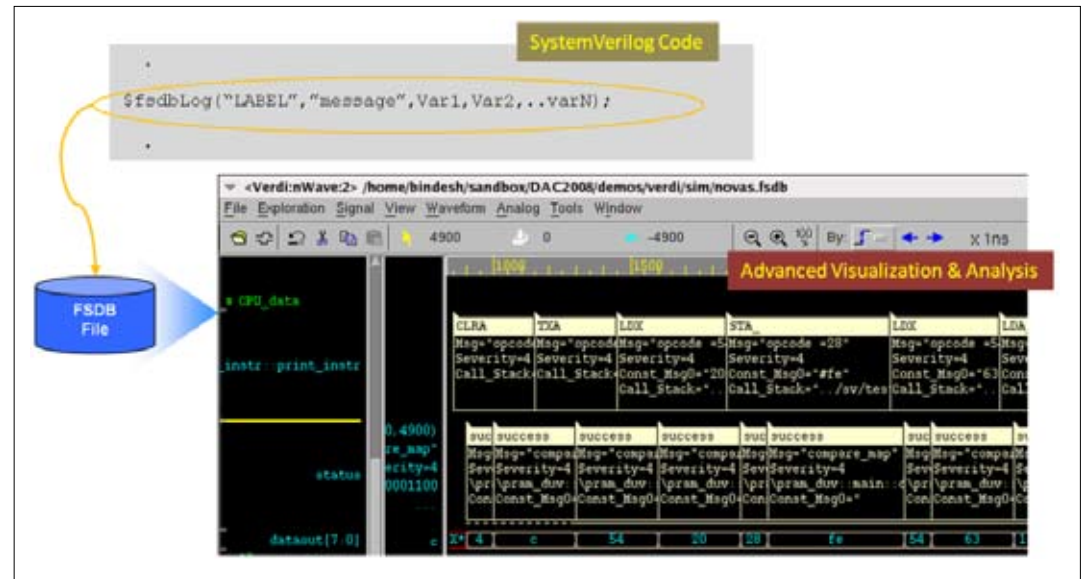


Figure 1: Illustrated is a flow based on logging user-instrumented information into FSDB database accessed by Springsoft’s Verdi Automated Debug and Siloti Visibility Automation systems.

and testbench debug flows, in which the visualization “tool” is often “vi” or “emacs”.

A new automated logging system-level task is introduced that not only captures messages, but also captures severities and variable states as properties or attributes of the message, as well as the call-stack to leverage later in debug. Since all the data goes into the same debug database as the one used for HDL recording, the logged information can be visualized alongside other data, such as HDL value-change and assertion states. Additionally, advanced visualization techniques allow engineers to observe what is going on in the entire environment in standard waveforms or specialized applications such as time-synchronized table (spreadsheet-like) views, which can be filtered and configured. Special-purpose features such as advanced filtering and highlighting can be used to identify or colorize specific messages based on some condition or to quickly find matches based on user-specified search criteria. The automatic capture of the call-stack during logging

provides unique opportunities for further automating debug. For example, a logged message can be synchronized with the source code using drag-and-drop from the waveform to the source code, which could then jump to where the message originated. It can also be used to quickly set breakpoints at the right place to drive interactive simulation from the debug environment.

## Logging to interactive link

While logging provides a coarse high-level view of testbench activity, interactive simulation of testbenches provides the GDB-like data that is occasionally required to understand their behavior, such as the values of variables at a specified point in time and detailed thread information. By bridging the ability to log messages with an integrated design-testbench debug flow, engineers can effectively use logging at the outset to determine the testbench code (location and time) that needs to be analyzed in more detail. The ability to drive interactive testbench simulation from the debug environment

allows for more user-friendly setup, visualization and analysis of the behavior of the design, the testbench message logs and the testbench itself, all in the same environment.

Strategies used in the software domain can help engineers meet head on the challenges of testbench verification and debug. It is clear that simply extending the traditional hardware debug techniques to testbench debug is not sufficient or even feasible. Gaining insight into what is going on in the testbench during simulation requires a new approach that builds upon the logging and interactive concepts previously discussed. The key is to make the logging process much more sophisticated and automated so that most of the debug and analysis of testbench activity can be done at that level. The goal is to use an advanced logging mechanism to pinpoint the location of a problem. If the problem is identified to be on the testbench side and more details are needed, engineers would then go into a tightly integrated interactive mode.

## Key pillar

Logging has been widely used in systems and software. For example, operating systems log information all the time for later analysis and debug if needed. Similarly, most software systems log information. So it is no surprise that logging is a key pillar in SystemVerilog testbench debug and analysis.

The dominant SystemVerilog methodologies in use today provide some basic libraries that enable users to log information from their testbench. However, the problem has been in visualizing the information, whether instrumented using raw SVTB syntax like \$display and printf or specialized base classes. All logging done through these mechanisms typically end up in text files.

To make debug of the design and testbench together a practical, efficient process, the logging mechanism must be flexible in terms of usage and the resulting output automatically captured in the same debug database as the design results (such as the de-facto standard FSDB format). This is fundamental to enabling advanced visualization, debug and analysis functionality. The proposed flow and usage are shown in **Figure 1**.

The task to log information—for example \$fsdbLog—needs to be highly flexible, allowing engineers to insert it anywhere in their code, including existing base class libraries that are intended for logging. The logging task must not only capture messages, but also severities, variable states etc. as properties or attributes of the message. In addition, the call-stack must be automatically captured to leverage in further debug automation. The upside of this approach is that since all the data goes into the same debug database as the one used for HDL recording, visualization support can be added to the debug system to analyze this logged information alongside other data, such as HDL value-change and assertion states. The net result is a unified system that enables engineers to observe what is going on in the entire en-



Figure 2: The use of a unified and full-featured debug system to drive interactive testbench simulation can allow for more user-friendly set-up and visualization and analysis of results.

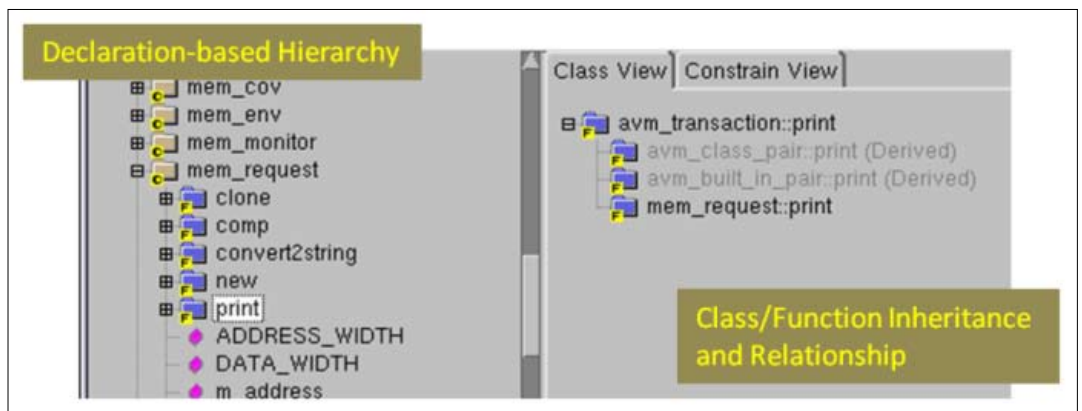


Figure 3: An instance-based hierarchy representation and UML-like class inheritance, and relationship view are critical to SystemVerilog testbench code comprehension.

vironment. The data is visualized in standard waveforms and via specialized applications such as a time-synchronized table view which, like a spreadsheet, can be filtered and configured.

Special-purpose features can be added to these views to help engineers easily identify messages of interest among the logged data.

## Building bridges

The automatic capture of the call-stack during logging provides unique opportunities for further automating debug. For example, a logged message can be synchronized with the source code using drag-and-drop from

the waveform to the source code, which could then jump to where the message originated. In addition to the obvious comprehension advantages of this capability, it can also be used to quickly set breakpoints at the right place to drive interactive simulation from the debugger.

Interactive simulation is often the only mechanism available for delving into the details of testbench code. While logging can provide a coarse high-level view of testbench activity, interactive simulation of testbenches can provide the GDB-like data that is required to understand their behavior, such as the values of variables at a specified point in time

and detailed thread information. Most simulators, when invoked in interactive mode, typically have access to all this information, albeit in a more primitive manner.

By bridging the ability to log messages with a unified design-testbench debug system, engineers can effectively use logging at the outset to determine the testbench code (location and time) that needs to be analyzed in more detail. With such a flow (**Figure 2**), a logged message can be dragged-and-dropped into the source code view so engineers can set a breakpoint, and then invoke interactive simulation in the background with the source-code view of the debugger serving as

the master cockpit. In this way, engineers can drive the simulator to a specific time or breakpoint, so that values, call stacks and thread information can be inspected (automatically or user-driven). This mode of operation is very similar to the GDB use model deployed by C/C++ programmers.

There are several compelling advantages of using the debugger to drive the simulator and display its results. Engineers can use the same environment to debug and analyze the behavior of the design, and the testbench message logs. Additionally, debug environments provide a more user-friendly and familiar environment to drive, view and analyze the testbench itself. For example, as shown in Figure 2, having variable watch and stack views alongside the source code can greatly enhance the user experience when debugging testbench code.

The task of understanding the structure and function of such complex testbenches can be daunting. Debuggers have always excelled at providing a platform for comprehending HDL source code. Commonly-used features, such as design browsing with an instance-based hierarchical representation and tracing of loads and drivers, are built upon a knowledge database that is automatically extracted from the source code. While some of this same functionality can be extended to testbench code, the more exciting opportunity lies in building on this knowledge-driven foundation to take testbench comprehension even further. Again, many of the ideas proposed here take advantage of practices that have already proven to be successful in the software domain.

Design code is typically built hierarchically with lower level modules instantiated at

higher levels and some modules instantiated multiple times. Conceptually, this can be represented in a tree-like fashion from the top-level module all the way down to the lower-level modules. Testbench code, however, like C++ and other object-oriented languages, is primarily made up of declarations of classes, functions and variables. During testbench debug and analysis, engineers want a quick way to navigate to a class, function, variable, or the newer SystemVerilog constraint and coverage code. Debug and analysis tools have to be able to import this type of code and display a meaningful representation that takes into account the declaration-centric nature of testbench code (**Figure 3**). This hierarchical representation must also be linked to the actual source code so that when a class, function or other entity

is selected, the corresponding source code is also displayed.

Given the object-oriented nature of SVTB code, engineers can easily reuse existing code and create reusable code themselves. Classes are often derived from existing base or parent classes. This inheritance allows them to retain all the capabilities of the parent while at the same time allowing for variables or functions to be replaced with new ones, or entirely new ones to be added. While declaration-based views can be enhanced to show some class hierarchy, most classes have complex relationships with other classes, particularly as engineers understandably take advantage of SVTB object-oriented-ness (reusability) in its purest sense. To represent this “organic” nature of classes, the concept of UML class diagrams can be borrowed from the software world.