

# SystemVerilog enhances assertion-based verification

By **Thomas Anderson**

Director of Technical Marketing  
Synopsys Inc.

Ben Cohen

Independent Consultant  
VHDL Cohen Publishing

The ever-increasing size and complexity of today's SoC devices put constant pressure on chip verification teams. Taping out a design without critical bugs is a huge challenge. One of the best-established approaches is assertion-based verification (ABV), which leverages designer knowledge and automatic verification methods to stress-test the design before tape-out. This article reviews this approach and discusses how the SystemVerilog language fosters more effective, widespread usage of assertions and ABV.

Assertions are statements of designer intent expressed in executable form. Ideally, designers capture their assertions within their RTL code as they write it. Assertions provide many advantages for the SoC development process. By documenting designer intent, they make the RTL code more readable and easily reusable. They capture elements of the required document or specification, thus providing a cross-check against the specific RTL implementation.

Given appropriate assertion language or constructs, assertions can capture complex temporal behaviors with relatively simple declarative statements. They enhance the verification process by adding value to simulation and enabling formal property analysis. They facilitate the development of verification IP by checking if design protocols are properly met.

In simulation, assertions are effective at detecting bugs at their sources. It is generally easier to diagnose and fix a bug when an assertion fails than when a simulation test failure is detected at chip outputs. Thousands of simulated cycles may elapse between the time a bug

is triggered and the time it is detectable at the outputs of a large SoC. In chip-level simulation, assertions catch the bug earlier, making it possible to isolate the problem to a specific block and hand it off to the correct designer for diagnosis and repair.

Both simulation and formal methods can generate various forms of coverage for the assertions. At a minimum, assertion coverage reports which assertions failed, passed in simulation and were proven by formal analysis. More advanced coverage metrics also provide infor-

extensions, pragma-based in-RTL methods and assertion-checker libraries available—many of them, vendor-specific. For the new assertion user, it is hard to select one method.

Finally, adoption of ABV has been hampered by a rather academic view of assertions and formal analysis. Adherents preach that all design behaviors should be captured in the form of properties (or assertions) and that design correctness should be proven exclusively by formal analysis with no need for simulation.

The greatest value of ABV arises in a verification flow that combines simulation and formal analysis. For example, formal might be used on blocks of 50-500K gates, a combination of simulation and formal at the functional-unit (1-2Mgate) level and simulation at the full-chip level. Furthermore, the advanced technology of hybrid formal verification combines the best elements of simulation and formal analysis in a single tool. As long as the same assertions can be leveraged throughout the entire verification flow, the development team will reap the benefits of ABV in terms of thoroughness and efficiency.

The SystemVerilog language standard, among its other advancements, fosters easier adoption and use of assertions and ABV. It includes rich assertion constructs that allow rapid specification of complex design behaviors, even those involving overlapping temporal sequences of multiple signals. SystemVerilog assertion features include basic Boolean expressions, sequences with specific or arbitrary delays, matching of regular expressions, disabling of assertions under specified conditions and specification of functional coverage points (**Figure 2**).

SystemVerilog assertions can be specified within the RTL code in the context of the surrounding design. For example, different assertions specified for “if” and “else” branches are evaluated only when their re-

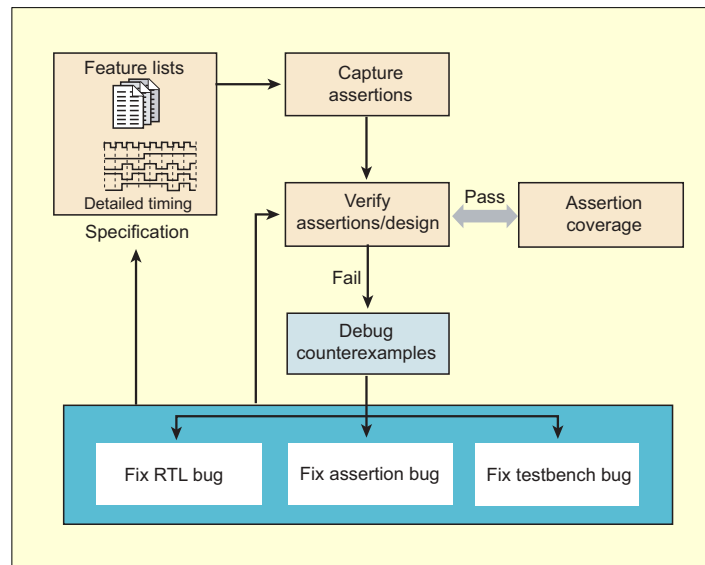


Figure 1: A combination of simulation and formal verification checks the design against the assertions.

Formal property analysis provides a way to leverage assertions without simulation. The assertions are treated as assumptions for stimulus constraints and as properties that are either proved correct (safe) or disproved with “counterexamples” showing how the assertions may fail. A disproof indicates an inconsistency between the assertion and the design, either an RTL design bug or a mistake in the assertion specification, possibly due to a misinterpretation of the requirements. Either way, the designer can fix the problem and rerun formal analysis to obtain a proof. Complete proofs guarantee that the assertion can never fail, but they may not be attainable for large designs or complex assertions. In such cases, bounded proofs quantify the amount of verification confidence.

**Figure 1** shows a typical ABV flow, in which simulation, formal property analysis, or both, are used to verify the assertions and design together.

mation on which assertions are passing “vacuously” because they have not been well-exercised. One common example is a FIFO overflow assertion passing in simulation only because the FIFO was never filled.

## Adopting ABV

Today, ABV is a mainstream approach that has been demonstrated many times to improve verification quality and reduce development time. Despite this, there are several reasons why not every design team has adopted ABV.

One challenge in adopting ABV is education. Writing high-quality assertions requires design and verification engineers to think about important behaviors of the design orthogonally from the specific RTL implementation. The second challenge is deciding how to specify the assertions. VHDL has a built-in assertion construct, although it is rather limited. There are many different property languages, language

```

property p_handshake;
  int v_data;
  @ (posedge clk) disable iff (!reset_n)
    ($rose(req), v_data=data) |=>
      ack ##[*1:5] data_out==v_data;
endproperty : p_handshake

ap_handshake : assert property (p_handshake) else
  $fatal;

```

Listing 1: The assertion references the property p\_handshake, which checks a specified sequence.

spective branches are active. This simplifies assertion specification, since enabling/disabling conditions can be leveraged from the RTL code rather than re-specified within the assertions. SystemVerilog assertions can also be specified in separate files and then “bound” to the RTL code. Commonly, designers specify assertions within their RTL, while verification engineers insert additional assertions in the testbench code or other separate files.

SystemVerilog provides a rich set of assertion-based functions, including:

- Those that sample values at specific times, such as \$fell, \$rose and \$past;
- Vector-analysis functions, such as \$onehot and \$countones;

- Severity-level system functions, such as \$fatal, \$error, \$warning and \$info, used to classify assertion errors and

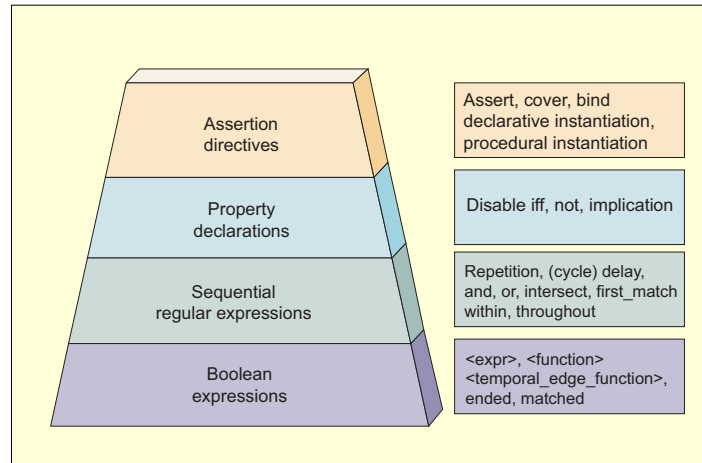


Figure 2: SystemVerilog provides a range of assertion constructs, from basic Boolean expressions to specification of functional coverage points.

- take appropriate action;
- Assertion-control system tasks, such as \$asserton, \$assertoff and \$assertkill, that can disable assertions during initialization, or at any other time during simulation, for individual assertions or all assertions in a hierarchy.

Listing 1 shows one example of a SystemVerilog assertion.

This code uses the assert property directive to specify an assertion. This assertion references the property p\_handshake, which checks a speci-

fied sequence. It samples all signals on the positive edge of clk and disables the assertion whenever the reset signal is active. If the design is not in the reset state, the property looks for req rising (asserted) and stores the value of data into the local variable v\_data. The property then checks if ack is asserted in the next cycle, followed within a range of 1-5 cycles later by data\_out equal to the stored v\_data. If this sequence occurs, the assertion passes successfully. If req is never asserted, then the assertion passes vacuously. If req is asserted, but is not followed by the specified sequence, then the assertion fails.

SystemVerilog assertions work in simulation and formal property analysis, with support available today in tools from numerous EDA vendors. SystemVerilog meets the key requirement of specifying assertions only once and then leveraging these assertions with multiple tools from multiple vendors. Furthermore, SystemVerilog assertions can be bound to any RTL design, whether it is SystemVerilog, an older version of Verilog, or even VHDL. □