Assertive Verification:
A Ten-Minute Primer

As published on 8/16/02 in EEDesign.com

And Written by
Saeed Coates,
Paradigm Works, Inc.
www.paradigm-works.com

Table of Contents

1.1 Introduction: The Verification Dilemma 3
1.2 Assertive Verification 4
  1.2.1 Assertion Checks 4
  1.2.2 Property Checking 5
1.3 Conclusion: Add Assertions Today and for Free 6

1.1 Introduction: The Verification Dilemma

By now it's well established that functional verification is the long pole to tapeout and if it's not thorough, could be a costly setback for a project. The move towards Systems on Chips (SoCs) contributes significantly to the exponential increase in verification effort compared to design effort. Time to market constraints also intensify this verification dilemma.

Most companies apply brute force methods to increase their verification success in minimal time. They staff projects with roughly three verification engineers for every designer. Using compute farms is a common method to mitigate exorbitant simulation run times. But even with more human resources and compute farms, improvements still need to be made in how verification is done. Simulations are extremely time consuming, especially for gate-level simulations, and they only cover a small subset of the possible states/events. Directed tests only verify the expected functionality, and test case randomization may find some unexpected bugs. Corner cases are becoming increasingly elusive in SoCs, however, and verification tests cannot cover all states. Therefore, fatal bugs are often found late in the development cycle, costing more time and money than if they were found earlier. The worst case, though, is when bugs make it into the ASIC and/or the field.

However one chooses to increase the effectiveness and timeliness of their verification effort, defining when the effort is complete is also a difficult task. The overall idea is to find as many bugs as early as possible and get to an experience-based confidence level. This intangible confidence level can be derived from combinations of several tangibles:

- A complete design specification
- Fulfilling a verification plan
- Directed tests that cover all intended functionality
- Random test cases
- System-level tests
- Code coverage metrics
- Applying formal verification methods
- A low and stabilized bug rate

Increased use of third-party IP, such as cores, accompanies this move towards SoCs. Verifying and distributing IP to minimize the integration effort poses an array of challenges. The IP distributor has to create their design to work in an extremely wide variety of environments and maintain compliance to a standard, if applicable. The quality and exhaustiveness of the IP distributor's verification effort directly impacts the customer's experience when integrating the IP.

In general, functional verification is a bottleneck in the design development process and commands a lot more attention and respect than in the past.

1.2 Assertive Verification

1.2.1 Assertion Checks

Every design has fundamental characteristics that must hold true during operation in order for the design to work properly. For instance, a bus may have a SignalB that must always goes low three to five clocks cycles after SignalA goes high; a bug exists if SignalB does not adhere to this property at any time during operation. This fundamental characteristic can be defined as an individual code entity known as a property. Properties can be implemented in either Verilog, VHDL, or an FPL (Formal Property Language). Additional code can be wrapped around basic properties and/or combined properties to build more complex properties. Entire monitors/checkers can even be built this way. A collection of basic properties, complex properties, and monitors are usually stored together in a property library. Assertions call out properties and check that the properties hold true at a give time (temporal) or at all times (static).

For whitebox verification, which is design implementation specific, design engineers can embed assertion calls inline with their RTL design. The assertion calls are written in the same language as the property library or as comments, similar to a Synopsys directive. Either way, the assertions do not get synthesized. For blackbox verification, verification engineers can instantiate assertion calls in a testbench. Assertions can be inserted at various places in or around a design:

- Around multiply instantiated blocks (eg. shipping 3rd party IP/reusable designs)
- Inline w/ code (eg. ensuring that a fundamental condition is always met)
- At the system level (eg. bus protocol monitors)
- At block boundaries (eg. ensuring designers adhere to a negotiated interface between blocks designed by different designers)

Assertions can be simulated with the RTL to act as watchdogs throughout the simulation. IP vendors can significantly leverage this by embedding assertions in their designs. The assertions ensure the quality and proper use of the IP, while staying invisible to the customer except to flag a bug.

When used with HDL simulators, assertions can significantly reduce debug time by more directly pointing engineers to where a fault has occurred in the design. Additionally, designers can do more debug on their own designs; thus, flushing out bugs earlier in the development process.

If the assertions are written in an FPL, an appropriate FPL compiler must compile the assertions before simulation. There is not yet a standardized FPL, but Accellera, a major electronic industry standards initiatives organization, is working toward proposing an FPL for IEEE standardization. Accellera selected IBM's internally developed FPL called "Sugar" from the four FPL submissions from members:

- Temporal e (Verisity)
- Sugar (IBM)
- CBV (Motorola)
- ForSpec (Intel)

After Accellera rejected ForSpec, Synopsys quickly joined with Intel and is now commercially packaging the language as part of its OpenVera hardware verification language. 0-In Design Automation has its own FPL variant which uses pragmas (ie. comment directives in the RTL) to declare assertions. OpenVera assertions call upon properties from the OVA (OpenVera Assertion) library, and O-In uses properties from its 0-In CheckWare library. Both libraries contain predefined properties and are user expandable. The OVL (Open Verification Library) is another assertion property library, but it's written in Verilog and VHDL rather than in an FPL. HDLs are not optimized for expressing temporal statements, but OVL is free and easily accessible as a download on the Web.

1.2.2 Property Checking

As gate counts continue to increase, the number of simulation cycles required to test an SoC's basic functionality increases by orders of magnitude. Still, though, the simulations only reach a limited subset of a design's entire state space (See Figure 1). Formal verification tools statically verify a design without using test vectors, testbenches, or other forms of stimulus. Compared to simulation, formal techniques can cover a larger area of a design's state space in a fraction of the time, and therefore, uncover hard to find bugs earlier in the design process.

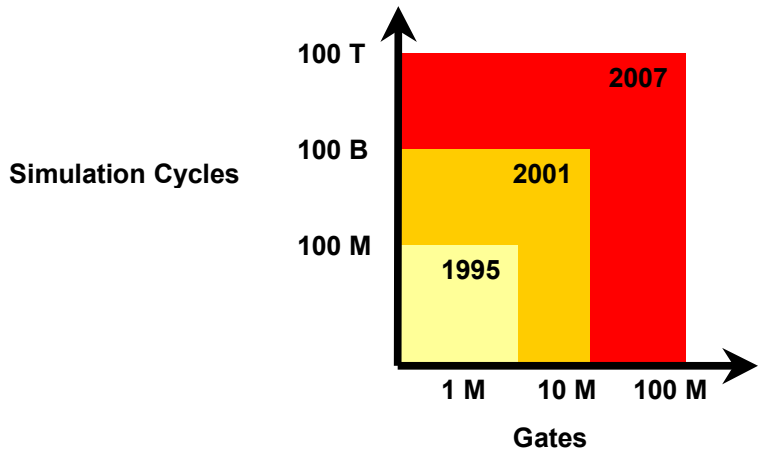

Figure 1: Simulation Cycles vs. Gate Count (Source: Faster and Smarter Verification; Manoj Gandhi, Sr. VP/GM Verification Technology Group, Synopsys, Inc.)

Equivalence checkers and property checkers are the two types of formal verification tools. Equivalence checkers compare that two representations of a design are functionally the same. Property checkers step through the different combinations at a design's inputs and concurrently run various checks against the design:

- Assertion checks
- Semantic inconsistency checks
    - Synopsys full case and parallel case directives
    - X-assignment
    - Casex
    - Casez
    - Range overflow
    - Uninitialized registers
- Structural inconsistency checks
    - Bus contention and floating bus issues
    - Sequential analysis
    - Stuck tri-state enables
    - Clock domain crossing checks

Assertions add significant value to simulations, but they add the most value when combined with property checkers. By stepping through a significant part, if not all, of a design's state space, property checkers overcome the well-known code and event coverage limitations of simulating. Property checkers significantly reduce the verification time by decreasing the need to simulate. This increased event/state coverage uncovers hard-to-find bugs, but sometimes property checkers can flag false negatives by checking unreachable states. Therefore, the tool user must supply adequate design constraints to the property checker.

1.3 Conclusion: Add Assertions Today and for Free

Leading verification experts have already generally accepted assertion-based verification as the next weapon against the SoC verification challenge. Unfortunately, it will take some time before a standard FPL is established and becomes ubiquitous. Hopefully, the varying proprietary FPLs won't hamper the adoption of assertions. However, the variety of choices in formal property checkers can only be a healthy sign:

- BlackTie (Verplex)
- 0-In Search (0-In Design Automation)
- Ketchum (Synopsys)
- Formal Model Checker (Avanti/Synopsys)
- @Verifier (@HDL)
- Solidify (Averant)
- Real Intent (Verix)
- ImProve HDL (Valiosys)
- Design Verity Check (Veritable)
- Formal Check (Cadence)

If adopting new tools into your flow is an issue, the best option would be to try the OVL (Open Verification Library); this is also the best option for keeping your RTL tool-independent until an IEEE standard FPL is released. It's available in both Verilog and VHDL and, best of all, it's downloadable for free (www.verificationlib.org). The assertions require no additional tools and work with any HDL simulator.