SYNOPSYS®

Design for Verification

Blueprint for Productivity and Product Quality

Rindert Schutten
Tom Fitzpatrick
Synopsys, Inc.

April 2003

Overview

Emerging design and verification technologies have shown great promise towards bridging the verification divide that exists in today's complex-chip design projects. However the lack of a cohesive, efficient methodology to allow these tools and techniques to be utilized efficiently has held back much of the potential progress. Addressing the root causes of today's verification problems, the design-for-verification (DFV) methodology described in this paper, is a comprehensive, structured design and verification methodology based on the following cornerstones:

- Comprehensive design flow covering multiple levels of abstraction
- Advanced technologies combining simulation and formal analysis
- Unified language to ensure concise and consistent specification, design, and verification descriptions
- Intellectual property to accelerate the design and verification of today's chips

We show how the DFV methodology enables project teams to design and verify the most complex system-on-chip (SoC) designs in a very effective manner. We describe what type of design artifacts to create to help eliminate bugs and facilitate their detection. We further discuss the requirements for simulation and analysis tools to support this methodology and ensure a predictable verification process that leads to the highest possible confidence in the correctness of the designs. As such, the DFV methodology described in this article aims to drive the verification solutions for the next generation of IC designs.

Introduction

In the last couple of years we have seen a plethora of new technologies, each focusing on specific verification bottlenecks experienced in today's complex chip-design projects. Technologies range from the introduction of dedicated verification languages like Vera® and e, various C/C++ based approaches, random stimulus-generation techniques, transaction-level modeling techniques, coverage metrics, formal analysis, template libraries, temporal assertions, and many others. However, fewer first-pass silicon successes indicate the verification gap is still widening despite advances in these technologies.

A 2002 study by Collett International Research revealed that first silicon success rate has fallen to 39 percent from about 50% two years ago. With the total cost of re-spins costing $100,000s and requiring months of additional development time, companies that are able to curb this trend have a huge advantage over their competitors, both in terms of the ensuing reduction in engineering cost and the business advantage of being to market sooner with high-quality products. To formulate a solution, we will take a step back and investigate the root causes of this problem and address the question: Why do chips fail?

Chips Fail for Numerous Reasons

Chips fail for many reasons ranging from physical effects like IR drop, to mixed-signal issues, power issues, and logic/functional flaws. However, logic/functional flaws are the biggest cause of flawed silicon. Of all tapeouts that required a silicon re-spin, the Collett International Research study shows more than 60 percent contained logic or functional flaws. The "band-aid" approaches of the past have not kept up with Moore's law. Therefore, companies need a more comprehensive approach to design and verification to master the complexities of tomorrow's designs and dramatically increase the effectiveness of verification. For companies designing the latest ASICs, ASSPs and FPGAs this is not a choice, but a requirement. For most it is a business necessity to increase verification productivity and improve the quality of results. This is exactly the promise of the design-for-verification (DFV) methodology that we describe in this article.

Therefore, let's ask the question again, but now more specifically related to logic/functional flaws: What types of logic/functional flaws are present in chips that nevertheless make it all the way to tapeout? The data that Collett International Research collected from North American design engineers is revealing. The top three categories are the following:

1. *Design error.* 82% of designs with re-spins resulting from logic/functional flows had design errors. This means that particular corner cases simply were not covered during the verification process, and bugs remained hidden in the design all the way through tapeout.
2. *Specification error.* 47% of designs with re-spins resulting from logic/functional flaws had incorrect/incomplete specs. 32% of designs with re-spins resulting from logic/functional flaws had changes in specifications. Clearly, improved specification techniques are required.
3. *Re-used modules and imported IP.* 14 percent of all chips that failed had bugs in reused components or imported IP.

There are two major reasons for the presence of design errors (bugs) in a design. First, the sheer complexity of a module, often including multiple-state machines, makes it virtually impossible to anticipate all possible conditions to which the module can be subjected in the context of an application. Typically the state space is very large and bugs can be buried very deep into the logic. Hence, some corner cases may simply not have been anticipated in the implementation. Second, designing a module often requires the designer to assume a particular behavior on the interface, that is, make assumptions on the behavior of modules physically connected to the module under design. These assumptions are needed to assure minimum area/maximum speed micro architectures to be designed. Analyzing the bugs indeed shows that incorrect or imprecise assumptions are a major cause of design flaws.

To improve the quality of the design process we clearly have to address specification, design, and verification in a concerted manner. Similar to other seemingly independent tasks in the past such as manufacturing test, design quality needs to become the whole team's concern, and the methodology employed must support this notion. That is exactly what the DFV methodology offers—coherent methodology to find design errors through constrained-random stimulus generation and advanced tools for formal state-space analysis, powerful means to eliminate ambiguity in specifications, and improved conformance checking capabilities to ensure that the imported design IP complies with required standards.

When deployed effectively, this methodology reuses work performed upstream in the process, combines tool benefits to increase verification coverage, finds the toughest bugs, introduces a sense of "verifiability" into the design phase, and makes use of up-front specifications as a representation of design intent to drive the design and verification process. Such a methodology brings together all the elements of good design and verification practice in a manner that provides significant and scalable productivity improvement.

Design for Verification

DFV is a cohesive methodology for design and verification of today's (and tomorrow's) complex SoCs to effectively do the following:

- Leverage a designer's intent and knowledge to strengthen the verification effort
  - Specify and refine design modules on multiple levels of abstraction
  - Maximize correctness of functionality of individual modules
  - Ensure the correctness of integration of these modules

These four elements enable a design flow that eliminates bugs systematically. DFV involves a small investment early in the design cycle focused on obtaining clear, unambiguous specifications, which provides a substantial return in the form of higher quality verification in less time.

The scope of DFV is depicted in Figure 1. The levels of abstractions that designers typically deal with in a design project span the universe of this methodology:

- Functional/transaction-level (TL)
- Synthesizable RTL
- Gate and transistor-level

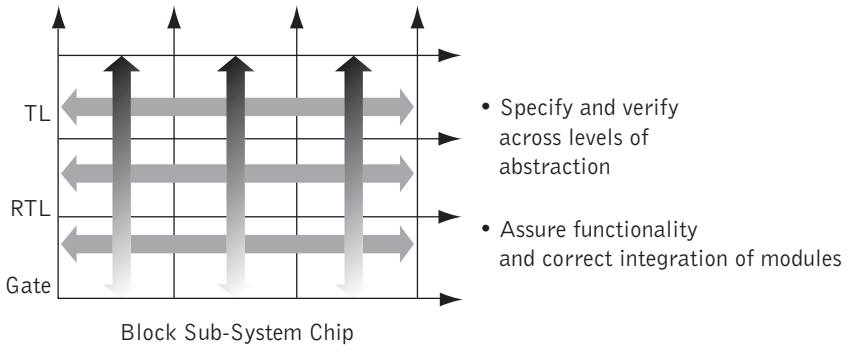

Figure 1. Design-for-verification methodology covers multiple levels of abstraction, block, sub-system and full-chip verification

Executing a design project requires design artifacts to be created on all three levels, and relationships between these levels to be maintained and assumptions propagated. For example, a transaction-level model assumes a particular latency between a request/response pair of transactions. This assumption must be carried forward when refining this block into an RTL implementation. If this assumption is "lost", some of the analysis rendered on the transaction-level model may be invalidated by the RTL implementation and indeed cause a fatal flaw. Other assumptions, for example between two RTL blocks and their communication protocols, are equally important and need to be captured and utilized in the design and verification process.

The Two Foundation Concepts of DFV

The two concepts that comprise the foundation of the DFV methodology are the following:

1. Use of design assertions as a focal point of specification to capture design assumptions, properties and interface constraints. Assertions add powerful capabilities to the verification process as follows:
   a. Assertions can be dynamically checked during simulations ensuring immediate notification of violations.
   b. Assertions can be analyzed in conjunction with the RTL to prove certain properties given a set of interface constraints.
   c. Assertions can be used to describe environment constraints that are used to drive automatic stimulus generation.
   d. Assertions yield coverage metrics that are directly linked back to the specification.

2. Use of multi-level interface design to ensure the following:
   a. Consistency between transaction-level. specification/assumptions and RT-level implementations, and
   b. Integration integrity on transaction-level, RT-level and gate-level models.

Central to these two concepts is the desire to allow designers to convey their design assumptions and intent into the verification process, as well as provide a means to compartmentalize the verification efforts. First, these capabilities separate the myriad-verification tasks into managed pieces while guaranteeing that these efforts can consolidate into a single consistent view of the overall design that is verified to be correct. Second, both concepts support and automate design practices that engineers handle every day, but in a fashion that makes these practices formalized through a comprehensive design language, and automated through a powerful verification platform. Ultimately, it is the effectiveness of the language that supports these concepts and the effectiveness of this platform on which the success of the DFV methodology depends.

As an evolutionary methodology, DFV enables designers to capture (formally) the specifications, assumptions and behavior properties of the design. Capturing design intent in a concise format enables this information to be used to automate the verification process and eliminate bugs from entering the flow. An equally important feature of a coherent DFV flow is that it improves the communication between design and verification tasks. Designers can leverage verification knowledge to improve the process, while verification engineers have a set of unambiguous design properties to verify. In fact, when designers write assertions (about the design) during the design process, assumptions and properties are made explicit, which leads to better code. Additionally, by taking into account the "verifiability" of a design block, it is further possible for the designer to make implementation choices that reduce the overall "verification cost" of the system, yielding higher-quality designs that are easier to verify.

Assertions as a Focal Point of Specification

Because the ultimate goal of verification is to obtain verified, synthesizable RTL code, the heart of any modern-verification methodology is an integrated toolset that includes simulation, coverage, advanced constrained-random test generation, and formal analysis, that provides the detailed verification of the RTL design implementation. Assertions bring all these capabilities together from a use-model perspective. Assertions are used to drive formal verification, monitor simulation behavior, control stimulus generation and provide the basis for comprehensive functional coverage metrics. The key role that assertions play is shown in Figure 2.

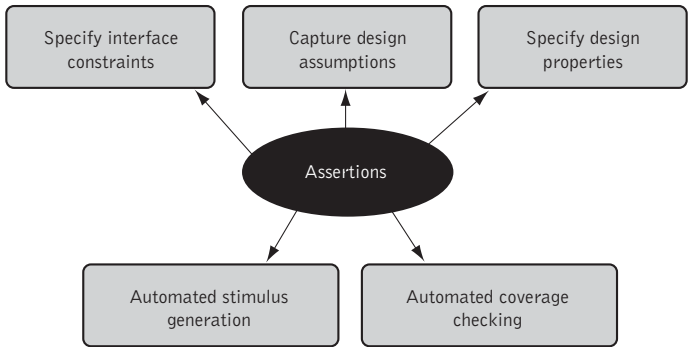

Figure 2. Assertions play a central role in the DFV methodology and are included in the SystemVerilog standard

Assertions capture constraints on interfaces, precisely defining the set of possible behaviors that are compliant with the protocols executed on the interface. These assertions then are continuously checked during simulation, while the same assertions are analyzed jointly with the RTL to explore the reachability of specific-design properties. Designers—developing the RTL code implementing a specific function—often make assumptions on how particular sub-modules behave to achieve the design targets. These assumptions are directly expressed through assertions and continuously monitored for validity. In certain cases verification tools can identify conflicts with these assumptions through mixed-dynamic and formal-analysis tools.

Adding up all the assertions that are associated with a module, designers can quickly reach hundreds of assertions that need to be tracked. Automated coverage tools to perform this tracking are essential to ensure that all of the assertions are exercised and systematically tracked to guarantee the completeness of the verification. Combining the individual modules during the integration phase of the project shows the value of having these assertions embedded in the RTL code—they automatically carry forward when the full-chip RTL is created. Companies that have followed this methodology have reported thousands of assertions associated with a design that are monitored, in many cases on a cycle-by-cycle basis, while running full-chip simulations.

Three aspects that are essential for the successful application of assertions as a central point of specification are the following:

1. *Native execution of assertions by the simulation engine.* Assertions are statements about design elements—signals, registers, transactions, and so forth. Checking an assertion requires access to these design elements, in many cases on a cycle-by-cycle basis. Minimizing the overhead of this access is required to ensure sufficient simulation throughput. Attempting to monitor even hundreds of assertions through, for example, a PLI interface severely degrades this performance. To minimize the overhead, assertion evaluation needs to be natively supported within the simulation kernel. Anticipating thousands of assertions to be "live" in comprehensive chip-level simulations makes such native execution a must.
2. *Synthesizability of assertions.* Having assertions synthesizable expands the applicability of the assertion-based methodology into formal-analysis based and hardware-assisted verification. The RTL code together with the assertions becomes a single module to which formal analysis technology can be applied, and even when this module is mapped into hardware the designer is still being alerted when assumptions and/or other design properties are being violated.
3. *Debug of assertions.* Assertions have the expressive power to enable concise descriptions of complex behaviors. When assertion violations are found, debugging aids are required to help analyze variable values contributing to the assertion result over time. Additionally, coverage results need to be debugged so that coverage holes are understood and tied back to missing functional tests.

Considering the above, the potential impact of using assertions to capture interface constraints, design properties and design assumptions is tremendous. It offers both RTL design and verification engineers the means to express design attributes that in current verification flows often are lost and lead to functional flaws in tapeouts. Moreover, now that these design attributes are captured formally, simulation and formal analysis tools dramatically increase the verification effectiveness, and even support project management by enabling extensive coverage metrics for these specifications.

Multi-Level Interface Design

Multi-level interface design is the second key concept behind the DFV methodology. Because most system-level bugs occur at the interfaces between blocks, designer investment in the form of well-defined interfaces greatly facilitates the DFV methodology. In traditional HDL design, the interface between two (or more) blocks is an agreed upon set of signals and behaviors to communicate data between the blocks. It is therefore not possible to test the interface fully until all blocks attached to it are assembled into a subsystem. At this level, the blocks are connected via a set of signals in the parent block, and the entire subsystem must be tested to ensure both the correct operation of the blocks together and the proper connection of the blocks.

At this level, designers spend a substantial amount of verification effort trying to stimulate the subsystem and exercise every low-level detail of the interface. It may even be possible, for example, to transfer data across the interface, even though there may be a subtle protocol error in one of the blocks, or even if the blocks are connected incorrectly. It would be beneficial if the interface could be reliably and correctly verified before the blocks are assembled into the subsystem.

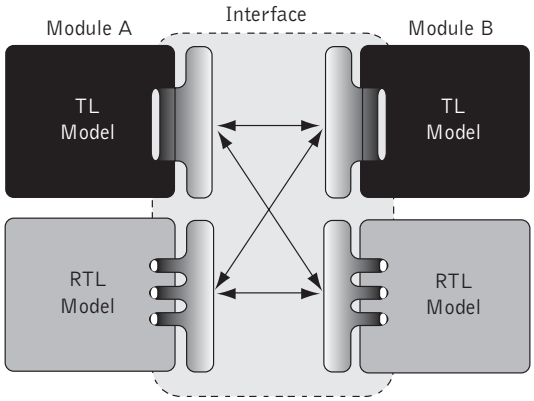

Figure 3. Interfaces capture how modules communicate and enable independent verification and multi-level simulation

Defining an interface as a standalone component of a system provides a twofold benefit. First, if the interface encapsulates the protocol and signals required for communicating across it, then a designer can verify the protocol in isolation. Second, it guarantees that blocks that use the interface are connected correctly, because the only connection is through the pre-verified interface. Hence, an improved divide-and-conquer flow results; at the block-level the focus is on complete testing of all detailed functionality, while at the top-level the focus is on the functionality of the overall system, not the detailed "wiring" between the blocks, because the interconnect is now correct by construction.

Interfaces, as outlined above and shown in Figure 3, are highly synergistic with the concept of assertions previously discussed. By including a set of assertions that completely defines the protocol in the interface, any block that connects to the interface will automatically be checked for protocol compliance. Interface assertions, similar to assertions embedded elsewhere in the design, can be used both in simulation and in formal verification to monitor or prove compliance, respectively. Because all modules connected to the interface share the same definition, the interface assertions represent a "single point of specification" for the protocol behavior.

By encapsulating protocols in this way, interfaces add the benefit of abstraction. Starting at the transaction-level (TL), both the "transmit" and "receive" sides of an interface are defined as high-level transactions enabling the creation of a true transaction-level model of the target chip. The interface then can include assertions that define qualities about the transactions, such as cycle latency, address/data relationships, and so forth. Moving downward in abstraction, by replacing the TL block with the corresponding RTL descriptions, the interface implementation is refined accordingly while ensuring that these assertions are maintained. At this level, additional RT-level assertions can be embedded in the interface. These RTL assertions further refine the protocol that was initially described at the transaction-level, providing more detail and finer granularity for defining the protocol and finding bugs.

Allowing the interface to define adapters to provide a bridge between the abstraction and RTL-transaction level enables chip-level simulations where parts of the design, including possibly the chip infrastructure, are "implemented" at the transaction-level, whereas other modules are represented as RTL. Therefore, setting up an interface-based infrastructure for the transaction-level model, allows a smooth transition from TL to RTL for each block in the system. In addition, the refinement of assertions in the interface ensures that the transaction-level behaviors remain consistent with the more detailed behaviors of the RTL.

With the increasing complexity of chips, the verification of a chip infrastructure is becoming a significant portion of chip-level verification. Representing this infrastructure explicitly by a set of interfaces that can be verified independently dramatically decreases the complexities of chip-level verification.

Simulation and Formal Technologies

The DFV methodology places specific requirements on the tools that support this flow. In addition to traditional design and verification capabilities, simulation and analysis tools need to incorporate assertions and multiple levels of abstraction, from both an execution and debugging perspective. High performance and high capacity are obvious requirements. Using higher levels of abstraction modeling techniques improves both by eliminating low-level details. However, as the design moves toward implementation these details can no longer be ignored, and therefore performance and capacity at the RT and gate levels continue to be a fundamental requirement. Irrespective of the abstraction level of the design, the ability to handle faster and larger designs is a key driver behind tool innovations.

Dynamic Verification Flow

Fully exploiting the benefits of design and verification on multiple levels of abstraction, the flexibility and scalability (through, for example, compute farms) of a software-based simulation flow, and the debugging capabilities it can provide, requires the simulation capability to be mainly software based. Existing verification flows focus on the effective creation of testbenches, random stimulus generation, coverage metrics, and core RTL simulation technology. DFV requires these flows to be extended to include assertions and multi-level interface design.

Full-chip simulations—even if part of the chip is simulated at the transaction-level– can have hundreds, if not thousands of embedded assertions. This requires that access to the design objects is direct and does not incur any overhead. The only way to achieve this is to have assertion execution directly incorporated in the simulation kernel. An added benefit is that this enables an integrated debugging environment, so that designers face only a single environment to debug and analyze both the design and the assertions. Similarly, the capability to gather both code coverage and assertion/functional coverage metrics, which is a required element of any verification methodology, needs to be supported with very low overhead to not slow down the simulation.

The second key technology of the DFV methodology, multi-level interface design, needs to be directly supported by system-level tools and simulators. From a simulation perspective the main requirement relates to debugging. Transactions need to be visualized and shown in the same environment as the RTL and the assertions, so that a designer, debugging a problem, can analyze them concurrently.

Some design/verification teams incorporate hardware-based acceleration or rapid prototyping in their flows. Typically these techniques can only be used late in a typical project timeline because all the RTL has to be available in a synthesizable form. The DFV methodology accommodates these hardware-assisted flows by requiring assertions to be synthesizable. Interface constraints, assumptions and design properties, can be leveraged in, for example, an FPGA-based prototyping flow. Simply including the assertions about the design and synthesizing them onto the FPGA fabric (at the cost of extra real estate) improves designers visibility into the design and gives them the ability to monitor results and be notified when they are violated, thereby substantially aiding the designer in finding the remaining bugs in the design when the design is placed into a (near) at-speed operating environment.

Finally, a very important aspect of simulation is the API into the simulator, especially the requirement of including C/C++/SystemC™ functionality into the (transaction-level) design. Another requirement is a low overhead, fast, standard interface covering the multiple levels of abstraction.

Formal-Analysis Flow

Assertions are an essential element in a formal-analysis flow. Interface constraints precisely define the set of possible behaviors of the interface when trying to analyze design properties. Aspects that determine the effectiveness of the flow are the following:

- *Completeness and correctness of the interface constraints.* An accurate set eliminates the potential of false-negatives/positives of the subsequent formal verification.
- *Formal engines underneath the analysis.* Depending on the type/location of a bug, alternative formal-analysis engines have different performance. To maximize the effectiveness of formal analysis, powerful formal analysis needs to be orchestrated with simulation technology to explore the state space of an entire module effectively. Moreover, a close link to simulation—specifically the ability to generate and validate accurate stimulus traces that cause a specific property or assumption to be violated—is essential when debugging the design error.
- *Capacity and performance.* The formal analysis needs to have high capacity to find bugs on large-scale designs. Capacity refers to both design size and property complexity in terms of state depth. The most advanced techniques use multiple formal engines integrated with simulation technology to effectively find deep corner-case bugs on complex designs.

Formal analysis (of RTL and assertions) is a promising new technology because it changes the balance of module verification from a simulation-dominated flow to a specification and analysis-based flow. Because formal analysis is specification driven and does not rely on testbenches, it holds great promise to increase verification productivity. In fact, many believe that a formal analysis-based methodology is a necessity to break the verification barrier of tomorrow's SoCs. Formal analysis technology transforms the "design first, then verify" methodologies of the past into the "specify first, then design and verify" DFV methodology. Incorporating assertions in a dynamic verification flow first, enables an evolutionary path towards increased use of formal analysis techniques.

Increasingly, formal analysis will be used to supplement existing module and block-level verification. This is why a cohesive flow, combining the formal analysis capabilities with the proven reliability of simulation technology, is the only practical solution. To best use the capabilities of each, there is also a requirement for multi-metric coverage capabilities to serve as a means of determining the completeness of the verification and as a way of communicating verification results across tools to avoid re-verifying features that have already been covered. To bring all these capabilities together from a use-model perspective, assertions form the single point of specification to drive formal verification, monitor simulation behaviors, control stimulus generation, and comprehensive functional-coverage metrics.

Verification IP

Most designs these days need to incorporate standard protocols. For example, AMBA®, PCI, USB (in all it's varieties), are standards one finds in many chips. Assurance that these standards are implemented correctly is essential for the success of the chip. Access to trusted verification IP to verify these standards increases verification productivity considerably. The nature of verification IP, however, is changing.

Most design and verification engineers are familiar with bus functional models (BFM). These models play an important role in quickly creating effective stimulus environments for a simulation-based verification flow. The DFV methodology extends this concept of Verification IP in the following ways:

- *Assertion-based verification IP.* A set of assertions that precisely captures the constraints of an interface and the corresponding transaction-level API's is the fuel for the DFV verification methodology. They combine the higher-level transaction interface with a precise specification of the intricacies of the signal protocols. They enable complete chip infrastructures to be designed and verified, while providing the means to ensure that the modules hooked up to this interface behave correctly. In turn, this provides a divide-and- conquer strategy, enabling a flow where the chip architecture can be designed and fully explored on the transaction-level, ensuring—through the DFV methodology—an effective path to fully-verified RTL. Additional design-level assertion IP, in the form of a library of pre-verified assertion templates, simplifies the designer's job by allowing specification of complex behaviors (such as FIFO over/underflow, arbiter fairness, and so forth) with minimum effort.
- *Transaction-level models.* In addition to the assertion-based verification IP, a second breed of IP is emerging. Now that an effective path is available to verify the refinements of transaction-level descriptions into RTL implementations, fast, cycle-accurate transaction-level processor models are appearing. They boost the ability to incorporate software earlier in the design process. Other models will follow to help designers populate executable architectures with transaction-level models of popular algorithms in multimedia, wireless, and core networking functions.

A well-defined DFV methodology enables an ecosystem of this new breed of verification IP. As BFM's are ubiquitous in current verification flows and will continue to play a crucial role in creating effective stimulus environments, assertion-based IP forms the cornerstone for high-productivity, formal analysis-based verification. These assertions need to be written in a common, well structured, standardized language that promotes reuse—a language that natively supports simulation tools (to maximize simulation performance) and is well understood by the analysis tools.

Unified Design and Verification Language

Just as language development, in the form of Verilog and VHDL, was critical for bridging the design-productivity gap from schematics to synthesis, language development is also a critical component of bridging the verification-productivity gap by enabling DFV. Aside from enabling synthesis, Verilog and VHDL had the clear advantage over previous languages in that they allowed the specification of both stimulus and the design in the same language. In the same way, SystemVerilog allows today's complex designs and testbenches to be described in a unified design and verification language. SystemVerilog includes the syntax and semantics to support assertions and multi-level interface design, while fully leveraging today's design and verification practices, making it the first true Hardware Description and Verification Language. Making the language evolutionary, through it's backward compatibility with Verilog, improves ease-of-use, learning curves, and teamwork, making adoption straightforward. Additional capabilities of SystemVerilog that complement the DFV methodology include the following:

- Powerful constrained random-data generation for testbench. Having the testbench and assertions part of a common language allows them to be consistent with each other, and to share a common semantic meaning across the full spectrum of the verification process. By making these concepts part of a standard language, it allows the industry to focus on improving methodologies and tools. Although SystemVerilog is based on Verilog, it also operates effectively as a public standard to unify the verification features with VHDL designs in a mixed-language verification environment, allowing all of its verification capability to be leveraged efficiently with VHDL designs.

- Improved capabilities for design. In addition to the encapsulated interface feature, SystemVerilog provides features such as data structures, enumerated types, multi-dimensional arrays, and more powerful procedural and control statements, making it possible to describe more complex synthesizable designs in fewer lines of code than is possible with Verilog or VHDL.
- Object-oriented features for testbench. To support today's existing verification infrastructures, SystemVerilog includes critical object-oriented programming features, such as classes, polymorphism and inheritance, to allow testbench environments to be assembled incrementally and extended in a clear and well-specified manner.

The built-in support for assertions and multi-level interface design together with the key additions to Verilog mentioned above, make for a language that truly unifies design and verification, not only in the traditional sense, but also as a foundation for the DFV methodology. SystemVerilog has the momentum and support from both users and vendors alike, enabling multiple suppliers of tools and IP to assemble the ecosystem in which the DFV methodology will flourish to become mainstream in the industry.

Powerful Verification Flow

With the key ingredients of the functional verification environment identified, a complete concept-to-gates verification flow, as depicted in Figure 4, emerges – a flow that eliminates bugs from entering the design. The first step, creation of a transaction-level model of the target chip, focuses on developing the right architecture. This architecture forms the starting point for the RTL implementation. Concurrently, the inclusion of a cycle-accurate processor model creates a virtual-system prototype that can be passed on to the software development team, enabling critical software components and hardware/software integration issues to be addressed early in the project.

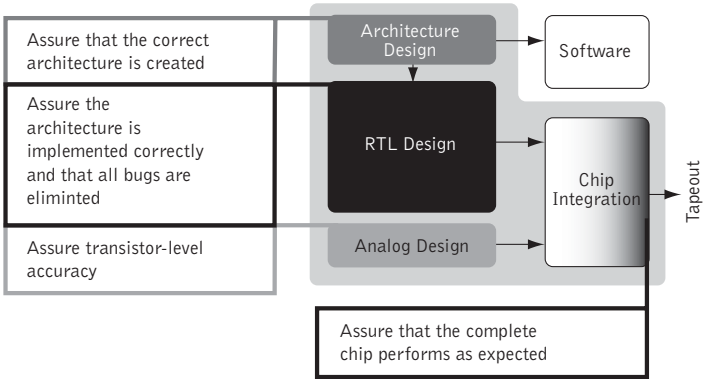

Figure 4. An integrated verification platform that covers the design from concept through system to transistor is required to meet the full demands of SoC verification.

The DFV methodology outlines how this transaction-level model can gradually be refined to a full RT-level model, extensively utilizing assertions to ensure that bugs do not enter the flow. Ultimately, this will ensure that the architecture is implemented correctly.

To meet the full demands of SoC verification, designers also need to be able to simulate and analyze analog components in the context of the system. This requires a co-simulation platform that can provide Spice-level accuracy for those modules that require detailed analog modeling. This environment must provide robust capacity and performance to handle the full system at the RTL level where potentially large portions of the chip are analyzed at transistor-level accuracy.

Furthermore, interfaces and API's need to be in place to include "foreign" models into this flow. Even though SystemVerilog, as a "concept to gates" design and verification language, supports the above flow, many valuable pieces of design and verification IP are in use today and need to be re-used tomorrow. Models in VHDL, SystemC, and so forth, therefore need to be incorporated into this flow in a seamless and high performance fashion to truly enable the design-for-verification methodology to become tomorrow's standard.

Conclusions

Many chips fail because of intricate design and/or specification errors that slip undetected through the verification process. The design-for-verification methodology describes a coherent solution comprising methodology, language, and technology to systematically prevent bugs from entering the design flow. This methodology benefits design and verification engineers by providing them the means to specify constraints, assumptions and properties unambiguously and incorporate multi-level interface design that enables a smooth flow from transaction-level down to RTL.

To bring the DFV concepts together in a single language is essential and SystemVerilog has the right ingredients to support this flow. Designers can use SystemVerilog to concisely describe more complex designs and design assertions, both at the transaction-level and in RTL. Testbenches can be effectively created in SystemVerilog and, together with assertions, ensure a complete solution for verification engineers. With SystemVerilog as a unified design and verification language, design and verification engineers speak the same language, providing indisputable advantages for team communication. Enabling a single language to capture the essential elements for a powerful verification methodology required for today's complex chips ensures maximum design productivity and the best chance for first-pass silicon success.

Finally, as a methodology DFV builds on the best practices used by many design and verification engineers who are successfully taping out chips today. DFV provides the methodology and sets the direction for tool and language development to best support engineers facing today and tomorrow's challenges. Early users that have embraced the DFV concepts report that they have achieved significant improvements in overall productivity and design quality. And this is exactly the motivation behind the DFV methodology: enabling design teams to create increasingly complex chips in less time, with first-pass silicon success.

700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com
For product related training, call 1-800-793-3448 or visit the Web at www.synopsys.com/services