Elements of Verification

March 25, 2005 -- ASIC design is a complex, multistage, time-consuming process, with huge associated development costs. It's important to get the design right at the first attempt and also meet time-to-marke criteria. Verification demonstrates the functional correctness of a design but it consumes almost 60 to 70% the total resources in a typical chip design cycle. With silicon design limits increasing 2x every 18 months, is important to verify functionality throughout the design to minimize the escalating cost of making a mista One error can decide success or failure of a product.

An effective verification environment (VE) is just as essential as a basic design environment. An effective V can aid in the development of a single chip and is imperative for creating multiple systems-on-chip (SoCs). value in the future will only increase as chips become larger, more complex, and more difficult to verify. Verification elements should be readily available in a predefined, easy-to-use form. Like any IP, they must modular, scalable, and reusable. Present day SoC sizes run in hundreds of millions of gates, for instance, Pentium IV comprises 55 million gates and network and backbone processors, in excess of 100 million gate They have become more sophisticated and complex to verify. Today's SoCs can have integrated processor cores, memories (RAMs, Cache, register banks), pipelines and parallel processing blocks, customized modu and multiple interfaces.

Since the design complexity has drastically increased, verification at different stages of ASIC development become absolutely essential. Various techniques and tools have evolved to support verification. There are different types of verification depending on the development stage:

- Functional verification
- Assertion-based verification
- Formal verification
- Gate-level verification
- Wafer testing
- Chip bring-up verification
- Manufacturing verification

This paper will discuss functional, assertion-based and formal verification types that deal with RTL and netl verification.

Functional verification

Functional verification determines correctness of logic in design before it is implemented on a device. It is used to check design compliance with specifications. Functional verification takes place during the early stages of the design flow so as to detect design flaws early in the chip development process. With the ever increasing complexities and an increasing use of reusable IP blocks, each new system IC and IP core has to be verified to eliminate discrepancies between the design and its specifications to ensure that the manufactured device functions properly. The test environment is developed and designed as per specifications. It generates and drives the defined stimulus to DUT (Device Under Test). Because timing information for the implemented design is not available, simulation tests the logic in the design using unit delays. Following are methods of carrying out functional verification.

Conventional Verification

Conventional verification is a BFM-based (bus functional model) approach where test benches are HDL (Verilog/VHDL) or 'C'-based. The verification engineer using this approach targets one bug/scenario at a ti Here, verification is restricted by the verification approach and the verification engineer's visibility of the te scenarios. But, attaining mere code coverage is no longer sufficient. Directed test cases have lost relevanc now that designs have become very complex and SoCs have millions of gates. They are written only to fill remnants. And there aren't enough hits with a single test run. Thousands of iterations are required for full functional and code coverage. The major drawbacks of HDL-based verification are:

- Test cases, as opposed to coveraged, are defined.
- Absence of random environment and random test case generation required to fill maximum coverag

Hardware verification language (HVL) based verification

Randomization, coverage, and analysis-based environments are implemented using HVL-based verification decreases the time and effort to verify and increases the verification strength. Only coverage groups need

be defined and the compliance checklist mapped in them. Using HVLs, the verification engineer is able to:

- Implement checkers, sequences.
- Generate signals, temporals.
- Inject and generate error/invalid scenarios and illegal conditions.

HVLs support use of coverage-driven verification methodology that enables complete verification of DUT us automatically generated real world scenarios. They also help create an integrated verification environment that supports constrained random traffic generation and functional coverage. This increases test range significantly and enhances maximum possibility for all kinds of corner cases. Automatic generation of transactions implies automatic coverage of corner cases and real world scenarios.

Benefits of HVL-based verification:

- Automatic, Random and Constrained Random generation
  - Automatic generation of transactions possible.
  - Automatic coverage of corner case scenarios and unknown corner cases.
  - Random packet generation and transmission resulting in greater possibility of hitting unexpected cases.
  - Random and constrained random generation supported.
  - Ranges, rules on generation possible for directed random packet generation.
  - Unknown corner cases covered .
- Coverage Driven Verification
  - Automated random generation functionality coverage.
  - Measure of verification completeness provided by functional coverage.
  - Corner cases exercised, with addition of a few directed test cases.

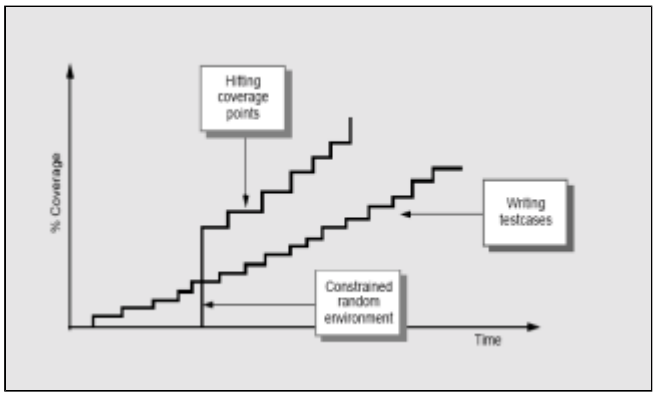

Figure 1. Constrained random generation improves coverage productivity

VIP-based verification

Most designs today incorporate standard protocols. For example, PCI Express, GBE, and USB are industry standards used in many chips. These standards have to be implemented correctly for the success of the ch Testing designs against such verification IP to verify these standards increases verification productivity considerably. IP/design reuse provides only 30 to 40% effort savings. The *big* ROI lies in verification reuse Verification consumes up to 70% of the time and effort and if users are focused on design IP, they address only 30% of the issue. Verification IP offers remarkable value to teams trying to integrate an IP core into t device. Until recently, VIP often only consisted of a bus-functional model (BFM), and possibly included a monitor for protocol checking. For VIP to be effective in enabling reuse of complex design IP, it needs to provide many more features and functions. The main advantages of using verification IP for verification are reduced time-to-verify, reduced costs, a scalable verification environment that enables bottom-up verificat of complex SoCs, and fast ramp-up to functional testing stage.

Layered, object-oriented verification methodology for next generation SoCs

Layer-based, object oriented verification environments are required to verify today's complex chips/SoCs. Layered verification environments (VEs) are the key to handling design modifications. VEs implemented layerwise, are modular, scalable, and reusable for different design configurations. Environments developed

using languages that support layered, object oriented verification (LOOV) provide scalability, modularity, a reusability. LOOV makes each layer and module in the VE interoperable and interdependent. Information sharing between modules and between layers becomes possible.

The elements of a verification environment are:

- Layers
  - Signaling layer (Interface/BFM)
  - Operational layer (Data manipulation / Data processing)
- Modules
  - Signaling modules (Interface/BFM)
  - Operational modules (Data manipulation / Data processing)
    - Data modules
    - Control module
- Test paths
  - Data paths
  - Control paths
- Test cases

Reusable VEs serve both module and system-level verification. Scalable VEs immediately acknowledge changes in functional modes or interfaces. For layer-wise VE implementation, design changes require changing only the corresponding VE layer. For example, if a new payload type is added, only the Data Protocol layer must be modified.

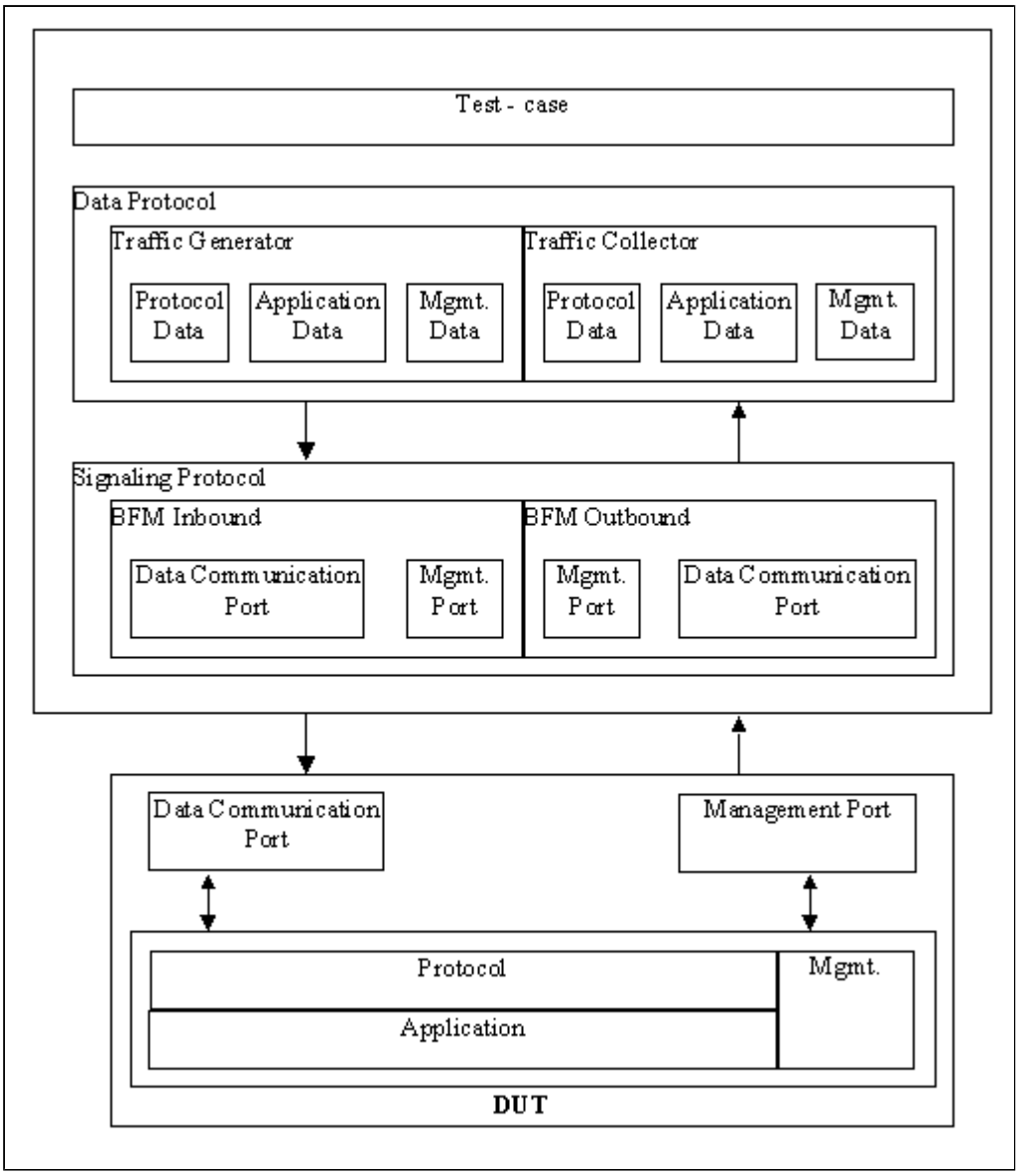

Figure 2. Object-oriented verification-based verification environment

Assertion-based verification

Assertion-based verification is a methodology that allows assertions to be specified. Assertions are design checks, either embedded deep in the design by a designer to catch the bugs early on or some pin-checks written by a verification engineer for black box testing of the code. These assertions detect bugs and implement coverage that guide test case writers to develop relevant and more effective tests. Their basic function is:

- Monitor all the interfaces to DUT in order to validate correctness of stimulus.
- Write protocol monitors in the DUT so that it is continuously monitored internally for the correct behavior.
- Write coverage properties to make sure current test suite covers all required stimulus scenarios needed to test DUT functionality.

Assertions provide information about the functional/behavioral nature of the block as the designer intende to be used. They can also be thought of as internal software test points that wait for a particular predefine condition to occur and then notify the designer about the occurrence. This is much better since the bugs ar reported at the cause point and not the effect point, saving the tedious process of tracing waveforms back their original causes. Without assertions, test vectors have to be much longer to ensure that triggered bug are propagated to observable outputs, else the errors remained undetected. By adding assertions, behavio inside the design could be checked and bugs observed instantly, at their source.

Assertion-based verification benefits users by:

- Simplifying diagnosis and detecting bugs by localizing occurrence of a suspected bug, thereby reducing simulation-debug time significantly.
- Self-checking code helps in design reuse.
- Interface assertions help find interface issues early on.

Formal verification

Formal verification is an algorithmic-based, mathematical approach to logic verification that thoroughly establishes the functional properties of a design. It's an alternative approach to simulation for functional verification. It proves that the origin and output are logically equivalent and that the functionality matches The two major types of formal verification used are equivalency checking and formal RTL verification. Once the design is functionally verified, formal verification is used to compare and verify the design at various levels of abstraction. As the device size and complexity increase, functional simulation becomes very time-consuming for both test bench writing and simulation. Formal verification has a short run time and provide complete functional coverage. Formal verification is a proven technology/flow in an ASIC design environme

Equivalency checking

Equivalency checking is the most successful and easiest to implement formal technique. Equivalency check prove that one gate or RTL design representation is equivalent to another gate or RTL representation. This technique is used to compare a design's synthesized gate-level netlist. Formal equivalence checking compa RTL, gate, gate with clock trees, gate with scan, and gate with engineering change orders to the "golden" I (an RTL design that has been fully verified to meet the design's functional requirements). The logical mode of design primitives are formally compared with their transistor implementations, and they, in turn, are compared with the final layout representations. The primary use of equivalency checkers is to establish tha successive design iterations still adhere to functionality of the golden RTL .

Formal RTL verification or model checking

Formal RTL verification tools are used to verify the correctness of the RTL. The verified RTL is called the "Golden RTL." Functional formal verification uses formal techniques to prove that a condition (often called property or assertion) can or cannot exist for a given design implementation. In other words, it's used to prove (or disprove) the assertions or characteristics of the design. Functional formal verification tools assu that each check is proven for the entire input space. Also, it overcomes the incompleteness issue in simulation methods.

Conclusion

However, there still is much more to verification than meets the eye. As mentioned before, verification consumes almost 60 to 70% of the total resources in a typical chip design cycle. Chip design flow requires verification at each level of abstraction beginning from architecture to silicon prototyping. The challenge fo engineers here is to verify spec-adherence for multiple abstractions. The verification process depends on various parameters such as the HVL used, the methodology employed, and the complexity of SoCs in term size, functionality and application.

There are a plethora of new tools, methodologies and languages in the semiconductor industry today. Eacl user segment has it's own preference. The combination of languages, tools, IPs and methodologies has morphed traditional verification into a "Hybrid Model" process. Since time-to-verify has become a major concern, moving forward, verification engineers may now have to depend on the "Hybrid Verification Mode that addresses all parameters of the verification process.

The future of design verification does not belong to any particular tool, language or methodology, but a combination of them all.

***By Rohit Dubey**. Marketing Manager, ASIC Solutions & Services, eInfochips.*