Chapter 0

ASSERTION BASED VERIFICATION
Use of assertions justified

The growing complexity and size of digital designs have made functional verification a huge challenge. In the last decade several new technologies have emerged in the area of verification and some of them have captured their place as a requirement in the verification process.

Figure 0-1 shows a block diagram of a verification environment that is adopted by a vast majority of verification teams. There are two significant pieces of technology that are used by almost all verification engineers:

1. A constrained random testbench
2. Code coverage tool

The objective is to verify the design under test (DUT) thoroughly and make sure there are no functional bugs. While doing this, there should be a way of measuring the completeness of verification. Code coverage tools provide a first level measure on the verification completeness. The data collected during code coverage has no knowledge of the functionality of the design but provides information on the execution of the code line by line. By guaranteeing that every line of the DUT executed at least once during simulations, a certain level of confidence can be achieved and code coverage tolls can help achieve that. Last but not the least, the process of verification should be completed in a timely fashion. It is a well-known fact that the worst bottleneck for any verification environment is performance.

Traditionally, designs are tested with stimulus that verifies a specific functionality of the design. The complexity of the designs forces verification engineers to use a random testbench to create more realistic verification scenarios. High-level verification languages like OPEN VERA are used extensively in creating complex testbenches.

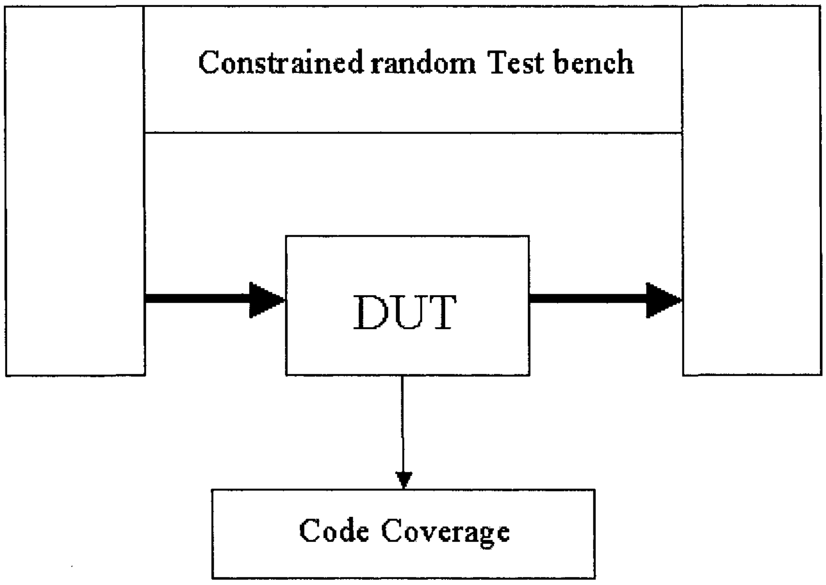

Figure 0-1. Before Assertion based verification

The testbenches normally perform three different tasks:

1. Stimulus generation.
2. Self-checking mechanisms.
3. Functional coverage measurement.

The first and foremost aim of a testbench is to create good quality stimulus. Advanced languages like OPEN VERA provide built-in mechanisms to create complex stimulus patterns with ease. These languages support object-oriented programming constructs that help improve the stimulus generation process and also the re-use of the testbench models.

A testbench should also provide excellent self-checking mechanisms. It is not always possible to debug the design in post-processing mode.

Mechanisms like waveform debugging are prone to human error and are also not very feasible with the complex designs of today. Every test should have a way of checking the expected results automatically and dynamically. This will make the debugging process easy and also make the regression tests more efficient. Self-checking processes usually targets two specific areas:

1. Protocol checking
2. Data checking

Protocol checking targets the control signals. The validity of the control signals is the heart of any design verification. Data checking deals with the integrity of the data being dealt with. For example, are the packets getting transferred without corruption in a networking design? Data-checking normally requires some level of formatting and massaging that is usually taken care of within the testbench environment effectively.

Functional coverage provides a measure of verification completeness. The measurement should contain information on two specific items:

1. Protocol coverage
2. Test plan coverage

Protocol coverage gives a measure on exercising the design for all valid and invalid design conditions. In other words, it is a measure against the functional specification of the design that confirms that all possible functionality has been tested. Test plan coverage, on the other hand, measures the exhaustiveness of the testbench. For example, did the testbench create all possible packet sizes, did the CPU write or read to all possible memory address spaces? Protocol coverage is measured directly from the design signals, and the test plan coverage can be easily measured with built-in methods within the testbench environment.

SystemVerilog assertions modify the verification environment in a manner such that the strengths of different entities are leveraged to the maximum. Figure 0-2 shows the modified block diagram for the verification environment that includes Assertion Based Verification (ABV).

There are two categories discussed in the different pieces of the testbench, which are addressed in detail by SystemVerilog assertions (SVA):

1. Protocol checking
2. Protocol coverage

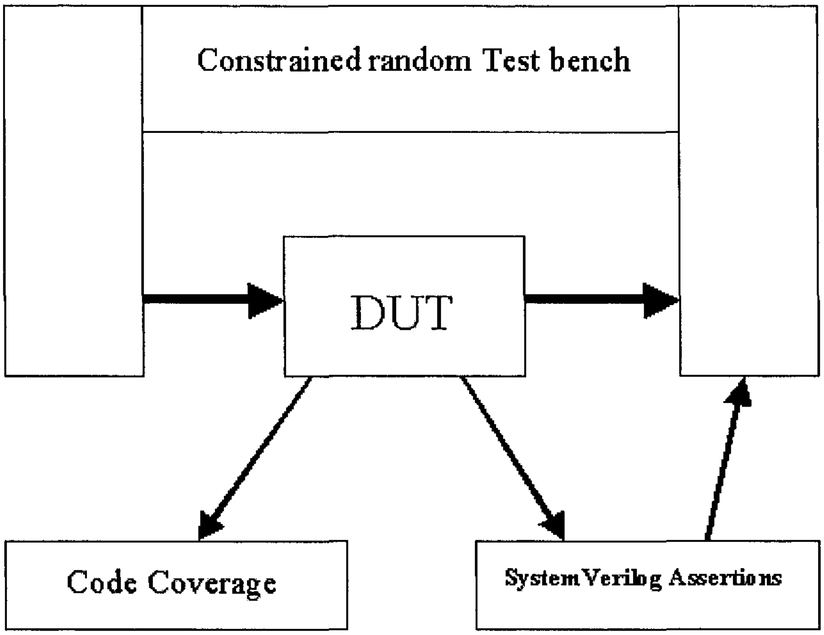

Figure 0-2. After SystemVerilog assertions

These two categories are closer to the design signals and can be managed more efficiently within SVA than by the testbench. By connecting these assertions directly to the design, the performance of the simulation environment increases tremendously as does the productivity. Table 0-1 summarizes the re-alignment of a verification environment based on SVA.

Though SVA interacts with the design signals directly, it can be used very effectively to share information with the testbenches. By sharing information dynamically during a simulation, very efficient reactive testbench environments can be developed. The completeness of the verification process can be measured more effectively by combining the code coverage and the functional coverage information collected during simulation.

Table 0-1. New verification environment

	Testbench	SVA
Before SVA	Stimulus generation Protocol checking Data checking Protocol coverage Test plan coverage	N/A
After SVA	Stimulus generation Data checking Test plan coverage	Protocol checking Protocol coverage

The book will introduce the SVA language, its use model and its benefits in an elaborate fashion with examples. It will show how to find bugs early by writing good quality assertions. Real design examples and the process of writing assertions to verify the design will be discussed. Measuring functional coverage on real designs and also how to use the functional coverage information dynamically to create more sophisticated testbenches will be discussed. Coding guidelines and simulation methodology practices will be discussed wherever relevant.