

Assertion-Based Verification Shortens Project Design Time

A popular verification methodology helps users detect bugs by localizing their occurrence and reducing simulation runs.

By Shailesh Dave

Designs are getting bigger by the day. The complexity of the testbench and the number of test cases are increasing exponentially with this design complexity. Many interfaces, for example, need various testbench architectures and a plethora of test cases. The result is a complex verification environment combined with the need for a huge amount of verification time. Such difficulties occur because there's no effective way of specifying what is to be exercised and verified against the intended functionality. Close to 70% of project time for chip design is spent on verification. Assertion-based verification is a methodology that helps the designer avoid this tiring, tedious, error-prone, and long-time problem.

Essentially, the methodology allows assertions that detect bugs. It also writes coverage that guides the test writers to write better. In addition, more effective tests can be specified. The basic idea follows:

- Monitor all of the interfaces to the device under test (DUT) in order to validate the correctness of the stimulus.
- Write protocol monitors in the DUT so that it is continuously monitored internally for the correct behavior.
- Write coverage properties to make sure that the current test suites cover all of the interested stimulus scenarios that are needed to test the functionality of the DUT.

As a result, designs can be verified thoroughly and effectively. In addition, bugs are found at the cause point. These techniques support two methods: dynamic verification using simulation and formal or semi-formal verification.

Although this method wasn't standardized or named, the industry has used assertions for verification for a long time. In its simplest form, Verilog users coded conditional loops to check for certain erroneous conditions or used a display for monitoring. VHDL programmers, on the other hand, used an assert statement to check for erroneous conditions. To effectively use assertions for verification, however, one needs a basic understanding of assertions and how they work.

WHAT ARE ASSERTIONS?

Assertions are design checks. To catch bugs early on, a designer may embed them deep down in the design. Or a verification engineer may write some as pin-checks for the black-box testing of code. Assertions express all of the information about the functional/behavioral nature of the block just as the designer intended it to be used. They also can be thought of as internal software test points that wait for a particular predefined condition to occur. They then notify the designer about the condition's occurrence. Such bugs are reported at the cause point rather than the effect point, saving the designer from the tedious backtracing of waveforms.

Without assertions, test vectors had to be much longer in order to ensure that triggered bugs were propagated to observable outputs. Otherwise, errors remained undetected. By adding assertions, bad behavior inside the design could be checked. In addition, the bugs were observed instantly at their source. An assertion-based interface specification also can be used to constrain pseudo-random simulation and formal analysis. This approach ensures that the circuit will work properly under all legal input stimuli while characterizing behavior in the presence of illegal input sequences.

The biggest advantage of assertions is that they become part of the design. Modern assertion methods enable the designer to specify assertions and monitors in line with the module's HDL code. Placing assertions in line is key, as in-line assertions cannot get lost.

Assertions also were used at design boundaries to provide checks for interface behavior. This function proved useful when design modules from different designers were integrated. Such interface-checking assertions made the modules more portable by preserving the intimate design knowledge needed to verify them as part of a larger system. As a result, modules with embedded assertions became self-checking wherever they were later reused. The net benefits resulted in finding problems sooner, spending less time analyzing them, reaching a stable design-verification point sooner, and simplifying integration and design reuse.

Generally, there are two kinds of assertions. Concurrent assertions state that a given property must always be true throughout a simulation. In contrast, procedural (sometimes called temporal) assertions apply only for a limited time or under specified conditions.

A proper approach to assertions is just as necessary as their use. It isn't always advisable or correct to write assertions from scratch for each project. They also must be modularized so that the same assertions can be reused for different applications. Protocol assertions are the typical classes of assertions that are the most suitable candidates for re-usability.

WHY ASSERTION-BASED VERIFICATION?

Normally, verification environments carry out black-box testing with many fairly complex emulation techniques, reference models, and scripts. The environment then uses thousands of test vectors to apply stimulus through hundreds of test cases. The idea is to match the functional testing goal. At times, even these complex scripts don't help to pinpoint the problem. For instance, say a problem is found at the interface. The tedious process of looking at waveforms for the in-module bug starts when the design engineer--and many times a verification engineer--spends all day looking at those 1 and 0 toggles on the screen. The engineers relate the signal's

previous state to the current state in order to find out what went wrong at what level. Many risks are involved in this verification approach:

- The time that it takes to set up the emulation environment, code reference models, and write scripts is huge.
- One needs to be sure of the reference model that's being used to compare the simulation results. These reference models normally check for the data signals and often tend to ignore the control signals. Once the reference model is made golden, the designer starts catching errors. But he or she still may not know the exact cause. The designer may end up doing a patch that can solve the problem temporarily. But the source of the problem may still be hidden.
- If the testing is done at the interface level, it's probable that a bug is hidden somewhere inside the module. Because it doesn't get propagated to the output, it remains hidden.

One cannot be sure of whether the sequence of inputs required to achieve a functional goal has occurred unless a highly directed test case is written. As a result, a plethora of test cases are required to achieve functional coverage. Assertions can help the designer to write internal/interface checkers and simultaneously write coverage to make sure that a specific scenario has been hit. The random generation of test cases will then be possible. In addition, 10 to 20 test cases could do what would otherwise have required hundreds of them.

Using assertions to specify interfaces is far superior to natural language and waveforms. After all, assertions are unambiguous and executable. The clarity of assertions comes from having well-defined syntax and semantics. Assertion languages allow the designer to clearly specify any interface. Yet they often require significant coding to describe complex interfaces and register-transfer-level (RTL) components. Such coding is definitely worth it, as it would take months off the project schedule. An added advantage is coverage facility, which acts as a watchdog for a specific desired scenario.

HOW ASSERTIONS WORK

Assertions are design checks that continuously get evaluated in the form of an expression. When any expression doesn't hold true, an assertion flag is raised. The verification engineer can then go directly to the problem instead of tediously backtracing the waveforms or analyzing a Megabyte-large log file.

Assertions can be utilized in various ways. They can be included directly within the hardware-description-language (HDL) code that comprises the RTL description of the design. They also can be applied from outside in the form of a testbench or collections of test vectors in order to check the design's response to stimulus.

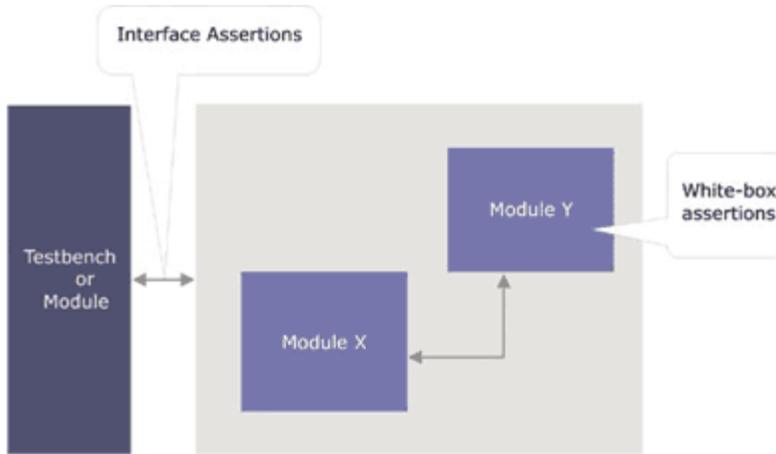


Figure 1: Interface-level assertions are used to check for defined relationships between all of the inputs and outputs on a chip.

Assertions can be developed by the designer and embedded in the design so that typical logic bugs can be found early in the project. Typically, the verification engineer creates interface-level assertions (black-box testing) to check for the defined relationship between all of the chip's inputs and outputs (see Figure 1).

Assertions		Coverage Points				
Total: 8 Inactive: 2 (25%) Active: 3 (37%) Finished: 1 (12%) Failed: 2 (25%) Disabled: 0 (0%)						
Assertion Name	Module	Instance	Current State	Checked	Finished Count	Failed Count
__sugar_assert_1	stimulus	stimulus	active	3	0	0
__sugar_assert_2	stimulus	stimulus	active	3	0	0
__sugar_assert_3	stimulus	stimulus	finished	3	2	0
__sugar_assert_4	stimulus	stimulus	active	3	0	0
__sugar_assert_5	stimulus	stimulus	inactive	never	0	0
__sugar_assert_6	stimulus	stimulus	inactive	never	0	0
req_gnt_seq	stimulus	stimulus	failed	4	0	1
req_seq	stimulus	stimulus	failed	4	0	1

Displayed: 8 Inactive: 2 (25%) Active: 3 (37%) Finished: 1 (12%) Failed: 2 (25%) Disabled: 0 (0%)

Filters:
 Name Filter: * Module Filter: * Instance Filter: *
 Display assertions with current state: Inactive Active Finished Failed Disabled

Figure 2: A reporting tool highlights the results from one simulation run with previously discussed assertions.

With embedded assertions, the bug shows up at the point of cause rather than a million clocks later when it propagates to the output of the chip. There is a chance that the bug will be nipped in the bud by the sub-module in it and therefore not propagated to the output. If a bug is detected at the later stages of the design, it could prove lethal. Because embedded assertions go with the design, they also make the reused-IP verification very easy.

As previously mentioned, assertions are basically protocol checkers that keep watch on specified signals for a specific transition. If a transition doesn't follow the expected sequence, the assertion reports an error. Assertions keep track of failing signals. Yet they also keep an eye on the

sequence of desired events and can be used to record a transaction. Take the example of a simple bus protocol that has an arbiter and more than one client talking to each other on a databus. All of the clients can talk to each other using the same bus, which the arbiter decides. The protocol uses the legendary req, gnt, busy, and done signals.

In this protocol, the req is followed by gnt and then busy is followed by done (once the requesting client gets the grant from the arbiter). Here are some simple assertions written in PSL:

```
//If Grant follows Request, then next is busy,  
and next is done.  
  
//psl property req_seq = always (reqA -> next  
(gntA -> next (busy && next done))) @posedge clk;  
  
//psl assert req_seq;  
  
//If request is followed by Grant, then next  
busy is high until done.  
  
//psl assert always (reqB -> next (gntB -> next  
(busy until done))) @posedge clk;  
  
//A Grant is always followed by busy until, and  
overlapping with, done.  
  
//psl assert always (gntA || gntB) -> next (busy  
until_ done) @posedge clk;  
  
//If A has a request outstanding when B receives a  
Grant, then A will receive a Grant before B  
receives another Grant.  
  
//psl assert always (reqA && gntB) -> next (gntA  
before gntB) @posedge clk;
```

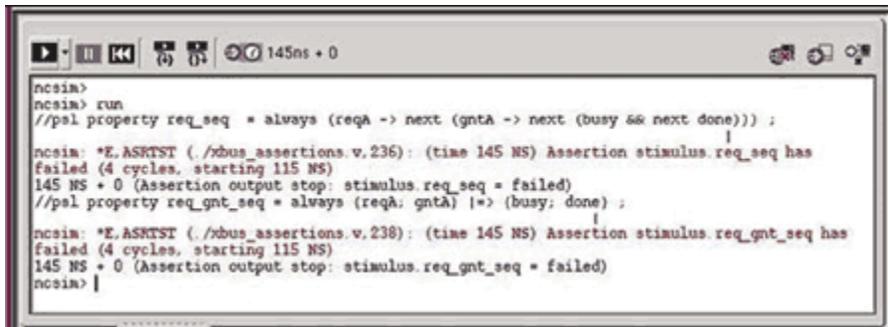
Clearly, it's easy to write an assertion and let the tool report the bug instead of studying and relating the waveforms to find out what went wrong. Once an engineer becomes familiar with writing assertions, shorthand assertions also can be used. These assertions are even more convenient.

The req_gnt_seq can be written as:

```
//psl property req_gnt_seq = always {reqA; gntA}
```

```
|=> {busy; done} ;  
  
//psl assert req_gnt_seq;
```

Informative reporting tools, which are built into the simulator, can compile information for the user. They will provide tables or text information with assertion text and the time of the failure. Figure 2 shows the results from one simulation run with the previously written assertions. Figure 3 illustrates a few common simulation errors.



```
ncsim>  
ncsim> run  
//psl property req_seq = always (reqA -> next (gntA -> next (busy && next done))) ;  
ncsim: *E. ASRTST ( ./bus_assertions.v.236): (time 145 NS) Assertion stimulus.req_seq has  
failed (4 cycles, starting 115 NS)  
145 NS + 0 (Assertion output stop: stimulus.req_seq = failed)  
//psl property req_gnt_seq = always (reqA: gntA) |=> (busy; done) ;  
ncsim: *E. ASRTST ( ./bus_assertions.v.238): (time 145 NS) Assertion stimulus.req_gnt_seq has  
failed (4 cycles, starting 115 NS)  
145 NS + 0 (Assertion output stop: stimulus.req_gnt_seq = failed)  
ncsim> |
```

Figure 3: A good reporting tool will capture typical simulation errors.

Another important aspect of assertions is functional coverage, which is far better and much more helpful than code coverage. These days, the designs are too complex. As a result, there's a need to make sure that all of the corner condition stimulus are generated and fed to the chip along with numerous normal test scenarios. Obviously, verification engineers rely on a random approach for test-vector generation. It's therefore even more important to make sure that a specific desired scenario is achieved. Using functional coverage is the best way to achieve this goal.

Although code coverage provides an indication of what piece of code has been tested and which has not, it fails to cover the sequence of events. For a verification engineer, the coverage for the event sequence is of more interest because that sequence simulates the happenings in the real world. With the increase in both the complexity and functionality of chips, the test plans are now at a higher level of abstraction. It becomes relatively easy to track that all desired functional scenarios are covered using functional coverage.

Although functional coverage can be written like an assertion, it is different in concept. Assertions are written to catch sequences that shouldn't occur in a simulation run, whereas coverage is used to make sure that the written sequence has occurred. Assertions therefore allow the user to exactly specify both what is needed and what is not with the help of coverage and assertion definitions.

Assertion-based verification benefits users by simplifying the diagnosis and detection of bugs. It localizes the occurrence of a suspected bug and thereby reduces simulation-debug time significantly. Another benefit is the capability to self-check code to help in the reuse of previous designs. Finally, interface assertions help to find the interface issues early on while detecting more bugs earlier and getting to the root cause of the problem more quickly.

Different languages are available for assertions, such as Sugar from IBM, Open Vera (OVA) from Synopsys, and Open Verification Library (OVL) from Accellera. Recently, Accellera accepted Sugar as a basis for a standardized property specification language--PSL (www.pslsugar.org). Since the IEEE has standardized PSL, it will be an added benefit for assertion users.

Shailesh Dave is project manager for ASIC design & Verification Group at eInfochips (www.einfochips.com). He holds bachelor's degrees in electronics and communications. He has over 5 years of experience in chip design and verification. He started his career in chip design as a verification engineer. He has worked as Design and Verification engineer for designs like PCI Express, graphic designs, DSP blocks, UMTS based receivers using HighLevel Verification Languages, Assertions and emulators like Palladium. Shailesh can be reached at: shailesh.dave@einfochips.com.