Practical Approaches to Deployment of SystemVerilog Assertions

Faisal Haque, Jon Michelson - April 03, 2007

As ASICs continue to grow in size and complexity, traditional verification techniques relying on procedural testbench languages are no longer sufficient. Stimulus generation needs to be further automated to accelerate coverage closure. Checkers and coverage monitors need to be more concisely specified, and bugs need to be more quickly isolated to decrease debugging time. More than fifty percent of the verification effort is spent on debugging errors. Debugging using end-to-end checkers requires tracing the error from the external interface back to the source. In complex designs, debugging is almost impossible without the help of intermediate checkers. Since assertions pinpoint errors at the source, they can be used as intermediate checkers and debug becomes much easier and faster. Figure 1 shows the productivity improvements that can be gained by using assertions.

Figure 1 shows the productivity improvements that can be gained by using assertions.

1. Assertions improve Coverage and Verification productivity. Results with directed testing method are shown in red; coverage driven, directed random results in blue; assertion and coverage driven, directed random in green.

As figure one shows assertions enable productivity improvements and reduce the risk of bugs by concisely specifying temporal behaviors for portable checkers and coverage points, pinpointing bugs closer to their source, and enabling formal model checking techniques to automate stimulus.
SystemVerilog Assertions (SVA) is a part of SystemVerilog and is being used in the verification of designs. To deploy SVA, guidelines need to be established which define where assertions should be added. The guidelines should also specify who writes the assertions, how they are controlled and how to ensure there are a sufficient number of assertions for each design.

SVA is a very concise yet expressive language in that each of its constructs can describe complex behaviors. There are subtle semantics of the language, which might not be fully understood, causing the resulting behavior to be different from what was intended. Therefore, it is recommended that coding guidelines be specified for SVA.

Temporal languages are different from procedural languages and thus have a higher learning curve. The key to successfully deploying assertions lies in solid language training from the start. Another key to successful deployment of assertions is to keep things simple, so starting with predefined assertions is easiest. Properties and sequences for custom SVA assertions can be created later. Predefined assertions can be created using assertion checker libraries, which are a predefined set of Verilog checker modules that check common behaviors. Accellera has a standardized set of checker modules known as Open Verification Library (OVL). Most tool vendors also supply checker libraries, which usually extend the OVL functionality adding some proprietary extensions and checkers.

Checker libraries can be used by instantiating the checker module in the RTL. For example, the `assert_always` checker module from the OVL library can be used to specify a condition that must always be true. The example below shows a counter that uses `assert_always` to verify that count never exceeds 143. For more details on the OVL checker library, the user should refer to the Accellera Open Verification Library reference manual.

```verilog
module test_assert_always(reset_n, clk);
  input reset_n, clk;
  reg [7:0] count;
  wire clr;

  always @(posedge clk)
    count <=(~reset_n || clr) && 8'b0 == count; 
    assign clr = (count >= 8'd143) ? 1'b1 : 1'b0;

  assert_always
    #("OVL_ERROR,"OVL_ASSERT,"ERROR: count > 143",'OVL_COVER_ALL)
    chk_cnt (clk, reset_n, ((count > 8'd0) && (count <= 8'd0)) && (count <= 8'd0))
endmodule
```

Checker library modules are a good way to start using assertions; however, they do not cover all of the different checks or coverage points needed for the entire design. A significant number of behaviors in a given design require the user to write custom SVA assertions. Therefore, to ensure a sufficient number of assertions are written for a given design, the user must write custom SVA assertions. In addition, since checker modules are written for generic usage, they contain a lot of additional code to support different configurations, to control different behaviors and to control the various levels of checking and coverage. A custom SVA assertion because it is written for a specific behavior has far less overhead than a checker module so it will provide better performance.
The example below shows a simple SVA based custom assertion that checks for Xs or Zs on control signals. Most SVA assertions will be more complicated but compared to a generic checker library module they will provide less overhead and maybe better suited to describe certain behaviors.

```
property PnotXorZ (expr);
@ (posedge CLK)
disable iff (!reset_done)
!$isunknown (expr);
endproperty
Ap_check_unknown : assert property (PnotXorZ({req1,gnt1,req2,gnt2}));
```

Effective use of custom SVA assertions requires usage and coding guidelines. A complete discussion of the usage issues and coding guidelines can take up an entire volume; therefore, we have limited ourselves a brief overview of the issues in this article. For more details on these topics, please refer to "The Art of Verification with SystemVerilog Assertions."

**Where to Add Assertions**

Assertions should be used in both design and testbenches. In design, they should be added in the RTL and used to check design intent, invariants, behaviors of various higher-level logic elements, or to specify interesting coverage points. In testbenches, assertions can be used as temporal checkers as well as to specify coverage points.

*Table 1* shows where assertions should be used in RTL to check behaviors and to specify coverage points. *Table 2* shows how assertions can be used in testbenches.
<table>
<thead>
<tr>
<th>RTL Functionality</th>
<th>Assertion</th>
<th>Coverage Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Invariants</td>
<td>Value is always within the legal range</td>
<td>Minimum and maximum values of the range</td>
</tr>
<tr>
<td>Temporal Invariant</td>
<td>A signal must be asserted some cycle after another signal</td>
<td>Boundaries of the temporal range have been hit</td>
</tr>
<tr>
<td>Check for X’s and Z’s on controls signals</td>
<td>Control signals should never be X or Z after reset</td>
<td>Control signal was asserted</td>
</tr>
<tr>
<td>Check for temporal windows</td>
<td>Ack/Grant delays or transaction delays</td>
<td>Minimum and maximum delays have been hit</td>
</tr>
<tr>
<td>RTL Elements</td>
<td>Memories, FMs, arbiters, FIFOs, queues, and math functions</td>
<td>Simultaneous reads and writes, Overflow, under run conditions Out of memory conditions</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Protocol checks, flow control, data integrity</td>
<td>Back to back transactions, Flow control at fastest and slowest rate Transaction errors</td>
</tr>
</tbody>
</table>

*Table 1. How to use checker assertions in RTL.*

[Click here for a larger version](#)
Table 2. How assertions can be used in testbenches.

Writing and controlling assertions
Who writes the assertions
We are frequently asked who writes the assertions—the designer or the verification engineer. The short answer is both. Generally, a designer will write assertions that go in the RTL, while the verification engineer will write assertions that are external to the RTL. For example, designers write assertions that are embedded in the RTL, while the verification engineer writes assertions on the interfaces of the design-under-test (DUT) and creates coverage points, checkers and monitors for the testbench. Verification engineers may also add assertions to fill any holes in the RTL checks left by the designer.

Controlling assertions
In any given DUT, there can be many assertions each consisting of one or more evaluation threads. Sometimes it is necessary to enable or disable certain sets of assertions. For example, during reset, all assertions not related to reset must be disabled, and during exception testing, the assertions related to the condition being violated must be disabled.

This means that a fine-grained mechanism must be defined for assertion control. One way to do this is to group assertions logically into categories. One or more categories can then be enabled or disabled at a time.
There are many different mechanisms available for assertion control. Each of the mechanisms has different trade-offs. `$asserton/$assertoff` system tasks are global mechanisms and can be used to control all assertions or specific named assertions. Compiler directives are compile time directives and allow assertions to be enabled or disabled at compile time. They do not allow assertions to be enabled or disabled dynamically during simulation. The SVA `disable iff` and `throughout` operators create vacuous matches or unnecessary failure respectively when the property is disabled. Therefore, these functions need to be used with care. The Verilog attribute construct allows an arbitrary tag such as a category to be attached to any Verilog code. Proprietary system tasks can then be used to enable or disable specific categories. The category attributes provides the best granularity of control for assertions.

**Ensuring that there are Enough Assertions**
The other major issue faced by users is how to make sure that there are enough assertions provided for a given DUT. It is important to make sure there are an adequate number of coverage and checker assertions instantiated in the DUT. Insufficient coverage assertions and checker assertions can lead to coverage holes, especially in corner cases, which in turn lead to bugs in the design. The question of whether or not there are enough assertions for the DUT can be addressed by the following:

- Assertion density
- Minimum sequential distance
- Cone of logic

Assertion density provides an approximate gauge of the observability and coverage of assertions within a particular region of the design. It is easy to implement with scripts and is usually measured in terms of number of assertions per 100 lines of RTL. Assertion density can be anywhere from 1-10 assertions per 100 lines of RTL depending upon the complexity and type of design. Sequential depth is a static measure of the number of levels of flip-flops between each register in a design and an assertion. The depth of the path with the least number of flip-flops between the register and an assertion is known as the minimum sequential distance (MSD). Typically, MSD should be less than five for most designs; however, this number varies with the type of design. The cone of logic is defined as the logic that fans into an assertion and is projected all the way back to the design inputs that fan into the assertion. The cone of logic infers the logic that is potentially covered by the assertion. A tool could aggregate the cones of logic for all the assertions and pinpoint any "holes" in assertion coverage.

Of these, assertion density is relatively easy to measure but provides the least precision. Minimum sequential depth (MSD) is more difficult to measure and requires tool support, but is more precise than assertion density. MSD identifies registers that might not be touched by assertions indicating logic areas that are low in assertions. Cone of logic-based coverage is more complex to implement but is also the most precise and can pinpoint logic areas that are low in assertions. All of these metrics should in fact be combined to provide feedback on how well a DUT is covered by assertions.

**SVA Coding guidelines**
SVA is a temporal language and has subtle side effects, which can prevent assertions from behaving as intended and leave undetected errors in the DUT. The actual coding guidelines are too numerous to list in this article, but Table 3 summarizes some guidelines.
<table>
<thead>
<tr>
<th>Coding and usage guideline</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify implementation-specific temporal corner cases using</td>
<td>Helps ensure that the DUT has been exercised properly</td>
</tr>
<tr>
<td>cover properties</td>
<td></td>
</tr>
<tr>
<td>Name each assertion</td>
<td>Identifies the assertion which fired</td>
</tr>
<tr>
<td>Keep things simple and modular</td>
<td>Easier to debug and reuse assertion code</td>
</tr>
<tr>
<td>Use sequence layer as much as possible</td>
<td>Sequences are instantiable within other sequences and properties while properties cannot be instantiated in a sequence</td>
</tr>
<tr>
<td>Always try to constrain unbounded sequences with other operators</td>
<td>A property using unconstrained unbounded sequences may never fail.</td>
</tr>
<tr>
<td>Be careful of implications that only pass vacuously</td>
<td>Assertions which are always vacuously true actually have not checked anything providing a false sense of security</td>
</tr>
</tbody>
</table>

Table 3. These are some of the coding guidelines.

Click here for a larger version.
Table 4. The table summarizes the deployment steps for assertions.

<table>
<thead>
<tr>
<th>Deployment stage</th>
<th>Where to Add Assertions</th>
<th>Type of assertions to Use</th>
<th>How to ensure that there are enough assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Check for Xs or Zs and simple invariants</td>
<td>Checker libraries and a few simple predefined custom assertions</td>
<td>Use assertion density</td>
</tr>
<tr>
<td>Later</td>
<td>Temporal windows</td>
<td>Checker libraries and custom SVA based assertions</td>
<td>Combine assertion density, MSD and core of logic</td>
</tr>
<tr>
<td></td>
<td>RTL Elements</td>
<td>Protocol checks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testbenches</td>
<td>Data transfer integrity</td>
<td></td>
</tr>
</tbody>
</table>

In summary, assertions are critical for verification of complex designs and if properly deployed can improve collection of coverage data, simplify checkers, and make debug much easier. Start with good language training and checker libraries, and keep things simple. Custom SVA assertions are essential to making sure all the important behaviors are covered and checked. Therefore, it is important to enable custom assertions by providing usage and coding guidelines. We have provided a quick overview of the all usage issues here for more details please refer to publications on SystemVerilog Assertions. We provide a complete list of references at www.verificationcentral.com.

About the Authors:
Faisal Haque is currently involved in architecture, design and verification of multi-tera bit systems. He co-authored the books titled "The Art of Verification with SystemVerilog Assertions" and "The Art of Verification with VERA". In addition, he is also the former chair of the SystemVerilog Assertions committee and is the current chair of the Unified Coverage Interoperability committee. Faisal has a Bachelors degree in Electrical Engineering and Computer Science from the University of NotreDame.

Jon Michelson leads efforts focused on the architecture, design and verification of multi-tera bit systems. He is a co-author of "The Art of Verification with SystemVerilog Assertions" and "The Art of Verification with Vera". He also co-designed and co-implemented a custom Hardware Verification Language (HVL) and methodology at SGI. He holds Masters and Bachelors degrees in Electrical Engineering and Computer Science from MIT.