

Assertions in SystemVerilog: A Unified Language for More Efficient Verification

Tom Fitzpatrick, Synopsys Inc.

October 2003

Abstract

The SystemVerilog standard is the result of an industry-wide effort extending the Verilog language in a consistent way to include enhanced modeling and verification features. By adding verification features to Verilog, an important goal of SystemVerilog has been achieved— a unified language which brings together enhanced design, verification and assertion features that deliver increased designer productivity and smarter verification that design and verification engineers can use to get their ever more challenging jobs done.

A key feature of SystemVerilog is SystemVerilog assertions (SVA), which unite simulation and formal verification semantics to drive the design for verification methodology. This article provides an overview of SVA and explores the unique advantages of unifying assertions with design and testbench constructs in the SystemVerilog language versus other approaches that use a separate language for assertions.

Introduction

SystemVerilog became an industry standard when it was approved in May 2002, by the Accellera standards committee. That version, SystemVerilog 3.0, focused mainly on enhancing the design modeling capabilities of Verilog to allow complex designs to be described concisely. In conjunction with the approval of SystemVerilog 3.0, several companies donated well-established technologies to Accellera in order to have SystemVerilog also address the most pressing needs of verification. Much of the effort of the SystemVerilog 3.1 standardization process was spent taking these technology donations and unifying them syntactically and semantically with the rest of SystemVerilog (and therefore Verilog). This unification provides a single environment in which both design and verification engineers can work together, cooperatively, to get working chips out to the market as quickly as possible. The SystemVerilog committee met this objective by delivering SystemVerilog 3.1 that was approved by Accellera this past May 2003.

SystemVerilog Assertions Benefits

SystemVerilog built assertions directly into the design and verification language, and that provides clear benefits. In fact, SystemVerilog has effectively unified assertions with the design and verification language for use in both simulation and formal verification. The key benefits of SVA can be summarized as follows:

- Ease of adoption because it is built on a familiar language and syntax
- Less assertion code due to automatic contextual understanding of design control logic
- Simple hookup and interaction between assertions and the testbench without special interfaces
- Customization and control of messaging and error severity levels
- Ability to interact with Verilog and C functions
- Avoidance of mismatching results between simulation and formal evaluations with clearly defined scheduling semantics
- Ability to improve verification performance by eliminating assertion co-simulation overhead and reusing simulation optimization algorithms

These benefits will be described in the following sections.

SystemVerilog Assertions Overview

SystemVerilog assertions were developed to provide design and verification engineers the means to describe complex behaviors about their designs in a clear and concise manner, building on concepts with which users are already familiar. With SVA unified syntactically with the rest of SystemVerilog, the user is able to embed assertions directly in-line with the design and other verification code, allowing the tools to infer a great deal of information from the context of the surrounding code. This reduces, in many cases substantially, the amount of code the user must write to specify the behavior, and simplifies the usage model since this information does not have to be duplicated, as it would with a separate assertion language.

The semantics of SVA are defined such that the evaluation of the assertions is guaranteed to be equivalent between simulation, which is event-based, and formal verification, which is cycle-based. This equivalence ensures that multiple tools will all interpret the behaviors specified in SVA in the same way. Moreover, the unification of assertions with design and verification code streamlines interaction to augment the power of assertions. In particular, SystemVerilog allows assertions to communicate information to the testbench and allows the testbench to react to the status of assertions without requiring a separate application programming interface (API) of any kind.

SystemVerilog provides two types of assertions: *immediate* and *concurrent*. Both assertion types are intended to convey the intent of the design engineer and to identify the source of a problem as quickly and directly as possible. Immediate assertions are procedural statements that can occur anywhere within **always** or **initial** blocks, and include a conditional expression to be tested and a set of statements to be executed depending on the result of the expression evaluation. The syntax of an immediate assertion is shown here:

```
immediate_assert_statement ::=
    assert ( expression ) [[ pass_stmt ] else fail_stmt ]1
```

The expression is evaluated immediately when the statement is executed, exactly as it would be for an **if** statement. The `pass_stmt` is executed if the expression evaluates to true, otherwise the `fail_stmt` is executed. The pass and fail statements, if present, are executed immediately after the expression is evaluated. Because an assertion carries with it the implication that the expression will be true, the failure of an assertion has a severity associated with it. SystemVerilog includes four system tasks (`$fatal`, `$error`, `$warning` and `$info`) that can be included in the fail statement block to indicate the severity of the failure and print additional user-defined debug messages, if desired. This is roughly the same functionality as the VHDL **assert** statement.

Concurrent Assertions

The real power of SVA, both for simulation and formal verification, is the ability to specify behavior over time, which VHDL assertions cannot do. Concurrent assertions provide the ability to specify such sequential behavior concisely and to evaluate that behavior at discrete points in time, usually clock ticks (e.g. "posedge clk"). The concepts and components that make up concurrent assertions can best be understood as a set of layers, each building on the layer(s) below:

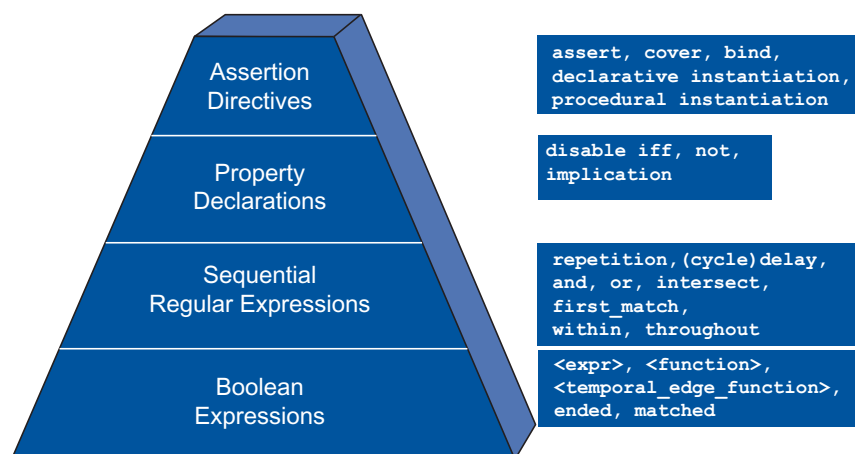


Figure 1. The layers of SystemVerilog Assertions.

¹ In syntax examples, such as this one, bold text indicates keywords or required punctuation; square brackets ("[]") indicate an optional item, and curly braces ("{}") indicate zero or more occurrences of an item.

The basic function of an assertion is to specify a set of behaviors that is expected to hold true for a given design or component. The Boolean expressions layer is the most basic, and specifies the values of elements at a particular point in time, while the sequential regular expressions layer builds on the Boolean layer to specify the temporal relationship between elements over a period of time. The property declarations layer builds on sequences to specify the actual behaviors of interest, and the assertion directives layer explicitly associates these behaviors with the design and guides verification tools about how to use them.

Sampling

To ensure consistency between simulation and formal verification tools, which apply a cycle-based view of the design, concurrent assertions in SystemVerilog use sampled values of signals to evaluate expressions. The sampled value of signals is defined to be the value of the signal at the end (i.e. at read-only synchronization time as defined by the PLI²) of the last simulation timestep before the clock occurs. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering and evaluating events.

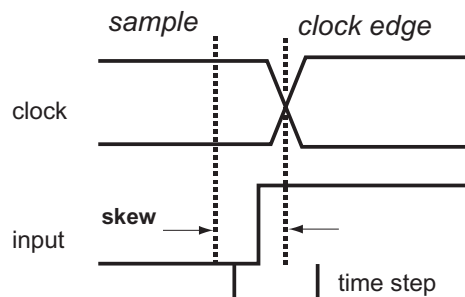


Figure 2. The sampling of values in SystemVerilog Assertions: The values are sampled at the beginning of a time step, before any other activity occurs.

This explicit notion of sampling signals relative to a clock edge is an integral part of the behavior of SystemVerilog assertions and is made possible in part because the assertions are simply another part of the SystemVerilog language. The existing Verilog scheduling mechanism has been extended to support value sampling and to provide “hooks” for the assertions to work seamlessly with the rest of the language. These scheduling extensions allow simulators and other tools to apply existing optimization algorithms to the execution of assertions as well as the modeling of the design and the testbench.

Separate assertion languages cannot define this explicit sampling mechanism because the required hooks are not part of Verilog. The behavior can be approximated in Verilog in some cases if the user adheres to a restricted coding style. But the possibility of race conditions and multi-clock systems makes it impossible to guarantee consistency between formal verification and simulation tools in all cases, without specifically enhancing the Verilog scheduling explicitly to support this particular goal.

Race conditions are an artifact of the event-based scheduling semantics of Verilog, which allow multiple events to occur at a given simulation time. Since synthesis and formal verification tools take a cycle-based view of the design, race conditions are always resolved according to the same sampling semantics that SystemVerilog assertions use. This means that the assertions, in effect, give the user a post-synthesis view of the design during pre-synthesis RTL verification, eliminating pre- and post-synthesis mismatches.

² PLI – Programming Language Interface

Concurrent assertions differ from immediate assertions in two important ways. First, in addition to being instantiated *procedurally* in a design as a statement in an **always** or **initial** block, concurrent assertions can also be instantiated *declaratively* as a module-level statement (similar to a continuous assignment) outside of procedural blocks. The second difference is that concurrent assertions allow the specification of a temporal behavior to be checked, instead of just a combinational condition as immediate assertions do. The syntax of a concurrent assertion is:

```
concurrent_assert_statement ::=
    assert property ( sequential_expr_or_property )
    [ [ pass_stmt ] else fail_stmt ]
```

The sequential expression is evaluated using sampled values of the signals, and the pass/fail statements allow the assertion to communicate with the testbench. Because the assertions are an integral part of the language, these statements can use the full breadth of SystemVerilog to trigger events, record coverage information or otherwise affect the flow of the verification code, including calling C code. Separate assertion languages are effectively “read-only” in that they can only monitor design behavior but cannot affect elements in either the design or the testbench.

Sequential Expressions

Having defined the proper sampling semantics for signals in assertions, SystemVerilog also includes the ability to specify *sequential expressions* or sequences of Boolean expressions with clear temporal relationships between them. To determine a match of the sequence, the Boolean expressions are evaluated at each successive sample point, defined by a clock that gets associated with the sequence. If all expressions are true, then a match of the sequence occurs. The most basic sequential expression is something like “a followed by b on the next clock” which is represented in SystemVerilog as

```
a ##1 b
```

In this example, the “##1” indicates a one-clock delay between successive Boolean expressions in the sequence.

Notice the similarity between the “##” cycle delay operator and the “#” delay operator, which specifies a number of time units to delay. This natural extension of an existing Verilog construct preserves the notion of delay, as Verilog users are used to, and is used elsewhere in SystemVerilog to indicate an explicit clock-cycle delay. This is but one instance where including the definition of assertions and sequences as part of SystemVerilog allows for common concepts to use the same syntax throughout different areas of the language.

It is important to understand that in SystemVerilog, each element of a sequence may be either a Boolean expression or another sequence. In terms of sequences, a Boolean expression is simply the degenerate case of a sequence of length 1. Thus, the expression

```
s1 ##1 s2
```

means that sequence s2 begins on the clock after sequence s1 ends.

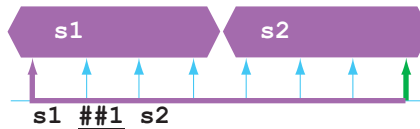


Figure 3. Sequence Concatenation: Sequence s2 starts on the cycle after s1 completes.

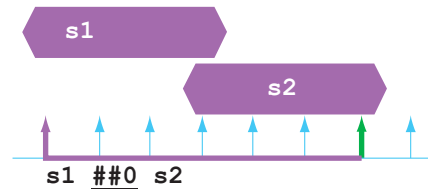


Figure 4. Sequence Overlap: Sequence s2 starts on the same cycle at which s1 completes.

While sequential expressions are useful for specifying temporal relationships between expressions, it is important to be able to capture sequential expressions as elements in the language so that they can be reused and referenced. SystemVerilog thus introduces the notion of a *sequence* which can be declared and reused elsewhere in the language, either to build other sequential expressions or, as we shall see, as part of properties to be asserted.

Sequences are declared in SystemVerilog using the following syntax:

```
sequence_declaration ::=
    sequence name [ ( formal_item {, formal_item } ) ] ;
    { assertion_variable_declaration }
    sequence_spec ;
endsequence
```

The optional list of formal arguments allows for specification of sequences as a generic temporal relationship that is applied to the actual arguments passed in when the sequence is instantiated. For example, the sequence

```
sequence seq1 (a,b);
    a ##2 b;
endsequence
```

represents a sequence of two expressions. When this sequence is instantiated as

```
seq1(e,f)
```

the actual arguments e and f are substituted for the formal arguments (a and b, respectively) in the sequence definition, so that the temporal relationship between e and f is

```
e ##2 f
```

The following table summarizes some of the operations that can be performed on sequences.

Operation	Syntax	Explanation
Concatenation	seq1 ##1 seq2	seq2 begins on the clock after seq1 completes
Overlap	seq1 ##0 seq2	seq2 begins on the same clock on which seq1 completes
Ended Detection	seq1 ##1 seq2.ended	seq2 completes on the clock after seq1 completes, regardless of when seq2 started.
Repetition	seq1[*n:m]	Repeat seq1 a minimum of n and maximum of m times. May result in multiple matching sequences
First Match Detection	first_match(seq1)	If seq1 has multiple matches, use the first and ignore the rest
Or	seq1 or seq2	Compound sequence that matches when either seq1 or seq2 matches
And	seq1 and seq2	Matches when one sequence matches on or after the other sequence also matches
Length-matching And	seq1 intersect seq2	Matches on cycles at which both seq1 and seq2 match
Condition qualification	cond throughout seq	cond is true for every cycle of seq
Within	seq1 within seq2	seq1 starts on or after seq2 and ends on or before the end of seq2

Table 1: Sequence Operations.

These sequence operations are shown visually in the following figures:

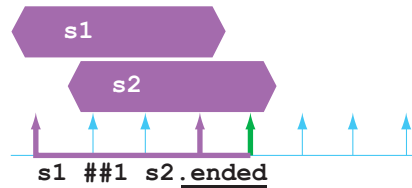


Figure 5: Sequence Ended: The ended method returns a Boolean that is true in the cycle at which the associated sequence achieves a match, regardless of when the sequence started.

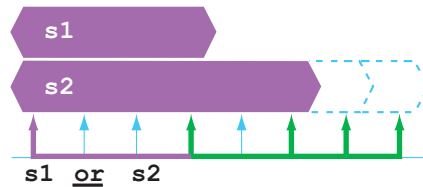


Figure 6: Sequence “or”: The sequence “s1 or s2” has multiple matches – when s1 matches and each of the samples on which s2 matches. If s1 matches, the “or” sequence also matches, regardless of whether s2 matches and vice versa.

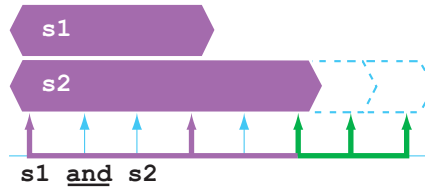


Figure 7: Sequence “and”: Although s1 matches at the 4th cycle, the and sequence does not match until s2 also matches, at the 6th, 7th and 8th cycles.

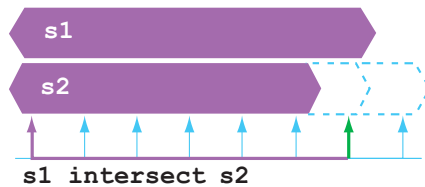


Figure 8: Sequence intersect: Sequence s2 matches on the 6th, 7th and 8th cycles. The intersection sequence only matches on the 7th cycle, at which both s2 and s1 match.

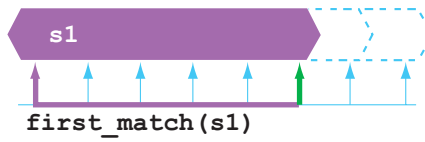


Figure 9: First Match: Sequence s1 has multiple matches at the 6th, 7th and 8th cycles. The first_match operator returns only the sequence that matches on the 6th cycle.

Local Variables in Sequences

Sequences specify temporal relationships between signals. However, to describe some complex behaviors, it is also necessary to be able to save values explicitly at some point in a sequence so it can be referenced later in the sequence. Consider a sequence such as “when data goes into a pipeline, it will come out between 3 and 5 clocks later, and will be incremented by one.”

```
sequence s5;
  int d;
  @(posedge clk) (d = data, valid) ##[3:5] (dout == (d+1));
endsequence
```

The valid signal marks when data goes into the pipeline, and 3 to 5 clocks later, the output of the pipeline, dout, will be equal to the input data incremented by one.

Capturing the sequence-specific data directly in the sequence eliminates the need for the user to write a separate state machine or other auxiliary code to capture the data, and eliminates the need of having to correlate this auxiliary code to each execution attempt of the sequence. It also avoids the problem of having to tell other tools that the auxiliary code is not really part of the design, thus simplifying the tool usage flow as well.

Properties

In many cases, sequences by themselves are sufficiently expressive to cover explicit behaviors. However, the property layer allows for more general behaviors to be specified. In particular, properties allow you to invert the sense of a sequence (i.e. the sequence should *not* happen), disable the sequence evaluation, or specify that a sequence be implied by some other occurrence. The *property* construct allows these capabilities using the following syntax:

```
property_declaration ::=
    property name [ ( formal_item {, formal_item } ) ] ;
    { assertion_variable_declaration }
    property_spec ;
    endproperty
property_spec ::=
    [clocking_event]
    [disable iff( expression )] [ not ] property_expr
property_expr ::= sequence_expr | implication_expr
```

A property definition is identified by the **property-endproperty** keyword pair, as one would expect, and supports formal arguments and argument passing exactly as sequence definitions do. Consider:

```
property p1;
    @(posedge clk) disable iff (test) not abort_seq;
endproperty
```

The **not** operator inverts the sense of the abort_seq sequence, so the sequential expression fails when abort_seq occurs. The “**disable iff**” clause disables the evaluation of the sequence if and only if the test signal is high. Notice the reuse of the keywords **disable** and **iff**, which are used elsewhere in SystemVerilog. This property could be read as “as long as the test signal is low, check that the abort_seq sequence does not occur.”

Another useful behavioral concept is that of *implication*. Suppose we want to check for an abort_seq sequence, but only after a valid init_seq sequence has occurred. This property would be written as

```
@(posedge clk) init_seq |=> abort_seq;
```

where $|=>$ is the *nonoverlapping implication* operator. This property expression states that each successful completion of the init_seq *antecedent* sequence *implies* that an abort_seq consequent sequence will occur on the next clock³. For clock cycles in which the antecedent does not occur, the consequent is not evaluated and the property is considered *vacuously true*.

Multiple Clock Support

All of the examples up to this point have used a single clock to sample all signals in the sequence. For single-clock systems in which the user is careful to avoid all race conditions, the semantics of Verilog allow for assertion languages to connect to the simulation and get the right answer⁴. However, as systems grow larger, including IP blocks from other groups or vendors, the likelihood that these strict guidelines can be met grows exceedingly small. As discussed, part of the advantage of unifying assertions with SystemVerilog is that the semantics are defined in such a way as to eliminate these unduly harsh usage restrictions. The race condition issue has already been resolved by the sampling mechanism discussed earlier.

³ The overlapping implication operator, “ $|>$ ” indicates that the consequent sequence begins on the same clock at which the antecedent sequence matches.

⁴ Accellera’s Open Verification Library is an example of such a single-clock assertion language.

When concatenating sequences with different clocks, the *multi-clock concatenation* operator, “##” is used:

```
@(clk1) s1 ## @(clk2) s2;
```

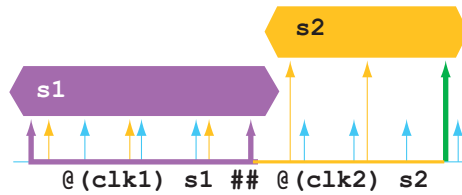


Figure 10: Multi-Clock Sequence Concatenation: After sequence s1 is detected on clk1, s2 begins on the next occurrence of clk2.

The first sequence is evaluated on each cycle of clk1 and when it matches, the ## operator moves to the next occurrence of clk2 to begin evaluating sequence s2. If the two clocks are identical, then this is equivalent to:

```
@(clk) s1 ##1 s2;
```

As with single-clock sequences, in multi-clock systems it is also useful to track the end of a subsequence, without regard to when it started. The *matched* method for sequences allows this functionality, similar to the *ended* method for the single-clock case. Unlike *ended*, however, *matched* stores the result of the sequence evaluation until the first occurrence of the alternate clock.

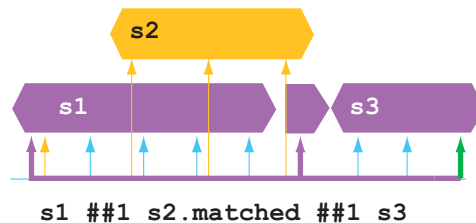


Figure 11: Multiple-Clock Sequence Matching: The completion of sequence s2 on clk2 is latched until the next occurrence of clk1. Sequence s3 begins on the next clk1 event after s2.matched.

By including multi-clock sequence concatenation and the **matched** method, sequences and properties can be written to specify behavior across clock domains. The sampling semantics and the latched behavior of **matched** ensure that the clock-crossing behavior is race-free, even in the case where the two clock edges are coincident, which is otherwise a situation fraught with race condition possibilities.

Procedural Assertions

In practice, most sequential assertions are expressed as some form of implication (“when this happens, then that will happen”), and thus require the assertion writer to specify the antecedent expression to trigger the assertion. As mentioned previously, one of the key advantages of unifying assertions with the design language is to make it easier for the design engineer to embed assertions in the design. However, declarative assertions, by their very nature, often require additional work by the designer to use them effectively.

Consider a finite state machine design. Assertions in this type of design are often of the form “when in *this* state, then *that* should happen,” such as “when in state ACK, if foo is high, then req should be held low for 5 clocks.” This assertion could be coded declaratively as:

```
P4: assert property(
    @(posedge clk) (st == ACK) && (foo == 1) |-> !req[*5]);
```

Now, consider what the RTL code would look like for the companion state machine:

```
always @(posedge clk)
    case (st)
    ACK:
        if (foo == 1)
            begin // when in this state, req should be low for 5 clocks
                ...
            end
        ...
    endcase
```

It is interesting to note that there are two specific pieces of information that are duplicated in the design and the assertion: the clock and the trigger state. Instead of requiring the designer to duplicate this information, the fact that the assertions are syntactically part of the design language allows them to be embedded *procedurally* in the RTL code and automatically infer this information from the usage⁵, as in:

```
always @(posedge clk)
    case (st)
    ACK:
        if (foo == 1)
            begin
                P5: assert property (!req[*5]);
                ...
            end
        ...
    end
```

This procedural concurrent assertion (P5) is defined to be semantically equivalent to the declarative concurrent assertion (P4) above, but is much easier to use and maintain. Procedurally embedded assertions use the sampled values of signals to evaluate the inferred trigger condition, just as declarative assertions do. The inferencing rules for the antecedent include both **case** and **if-else** statements, allowing arbitrarily complex triggers to be inferred for procedural assertions. The resulting advantage for procedural assertions is simple maintenance. That is, if the finite state machine code is modified, the triggering condition for the assertion is automatically updated, whereas the user would have to modify the code manually to update the trigger of the corresponding declarative assertion, introducing the possibility that the assertion will no longer accurately reflect the design.

Assertion and Testbench Interaction

As stated previously, a significant advantage of unifying assertions with the design and verification language is the ability for the assertions to communicate information to the testbench. In a standalone assertion language there is no such communication mechanism. In SystemVerilog, the pass and fail statements of an assertion may execute any procedural statement, allowing the user to notify the testbench that a sequence occurred, update coverage counters, or anything else that may contribute to the effectiveness of the verification, including setting signal values, calling class object methods, or even calling C code.

⁵ Notice the documentation benefits of the embedded assertion, which conveys unambiguously the same information to the reader as the comment in the previous example.

Consider a bus protocol that includes the property “a new bus cycle may not start for 2 clock cycles after an abort cycle occurs.” This property could be coded as:

```
property wait_after_abort;
  @(posedge clk) abort_cycle |=> !cycle_start[*2];
endproperty
P6: assert property (wait_after_abort);
```

When the *abort_cycle* sequence is detected, the property dictates that the *cycle_start* signal will be low for two clocks. The property P6 codifies this behavior, and can be used as a monitor in simulation to check the behavior, or in a formal tool to try to prove that the behavior will never occur.

However, if this property is defining the behavior of a top-level interface, the testbench will be generating the bus transactions that must conform to this behavior. The verification engineer writing the testbench can reuse the assertion information as a constraint to help ensure that the generated stimulus does not violate the behavior.

Such a testbench might look something like:

```
program manual_stimulus_generator;
repeat(1000) begin
  generate_transaction(addr,data);
  while(wait_cnt > 0)
    @(posedge clk) wait_cnt--;
end
endprogram
```

The *wait_cnt* counter is used to specify the delay between bus transactions generated by the testbench. The *abort_cycle* sequence can be used to set this counter by using a **cover** directive:

```
cover property( abort_cycle ) wait_cnt = 2;
```

Using the **cover** directive allows the *abort_cycle* sequence to be detected, without producing an error if it is not. When the sequence is detected, the *wait_cnt* counter is set to 2, causing the testbench to wait 2 cycles before generating the next transaction. The scheduling semantics of SystemVerilog ensure that there will not be a race condition between the execution of the pass statement and the testbench, which could not be guaranteed if the assertion were in a different language.

SystemVerilog includes the ability to specify an explicit synchronous interface between the testbench and the design. The *clocking domain* construct defines when the testbench will sample and drive device-under-test (DUT) signals relative to a clock. By default, the sampling of signals by the testbench is the same as the sampling used by assertions, ensuring that the testbench has access to the same values as the assertions. As mentioned previously, this not only avoids race conditions between the testbench/assertions and the DUT, but it also affords the testbench a “post-synthesis” view of the design behavior, with race conditions resolved according to synthesis rules, during pre-synthesis RTL verification.

SystemVerilog Assertions for Designers

Assertions are only of value if they are included with the design in some way. A substantial portion of the typical verification process is spent validating designers’ assumptions about the environment in which the design will operate and about the design functional specification. How many times have we, as designers, said “it should work,” only to realize that we had misread the specification (or at least interpreted it differently from a colleague) or had made some silly mistake in the RTL code such that the functionality described was subtly different from what we intended? It is therefore critically important to enable designers to capture assumptions easily in a way that is familiar and easily fits with the rest of the design and verification methodology.

Unifying SystemVerilog assertions with the rest of the language that designers already use allows designers to take advantage of this paradigm, and is perhaps one of the biggest advantages that SystemVerilog has over separate assertion languages. Because SystemVerilog assertions can be embedded directly in the design code, the incremental effort of adding assertions to the design is small. Designers may choose to write the properties directly, either declaratively or procedurally, or they may instantiate pre-verified properties from a library⁶. Either way, the assertions become part of the design, which makes designers more comfortable and more likely to use them.

With appropriate mixed-language tool support, the VHDL user is able to write concurrent assertions in SystemVerilog and use the **bind** statement to connect the assertions to VHDL components or instances, just as with SystemVerilog. The Boolean expressions in SVA can support VHDL signals, just as mixed-language tools support writing Verilog expressions that include VHDL signals.

SystemVerilog Assertions for Verification

Verification engineers often use different means and tools to ensure thorough functionality checking. For this approach to be effective, results between different verification tools must be consistent. Otherwise much effort is spent uncovering tool issues rather than design issues. Users of SVA obtain consistent results because SystemVerilog explicitly extends the scheduling semantics of Verilog to allow for the sampling of values and execution of assertions to ensure that simulation and formal verification tools produce the same answer.

It is interesting to note that to ensure this semantic understanding of assertions, the SystemVerilog standard defines several new callback points within a simulation timeslice exactly to support this requirement. These extensions, by definition, do not exist in Verilog making it difficult, if not impossible, to define a separate language to interact with Verilog in the same predictable way⁷.

The scheduling semantics and the unification of assertions in SystemVerilog also allow the testbench and assertions to work together, which means that the testbench can recognize the existence of assertions and react to their state. If assertions are described using a separate language, then there is nothing to which the testbench can refer. The scheduling extensions allow the verification code to execute at a later point within the timeslice after the assertions have been evaluated. In fact, the pass/fail statement block of a concurrent assertion is an example of such verification code and is a convenient mechanism for the assertion to communicate information to the testbench. Neither the scheduling nor communication mechanism exists for foreign assertion languages.

The unified language also provides greater flexibility for verification environments. SystemVerilog, for example, includes the ability to call C functions directly without requiring the PLI. Therefore, C functions can be included in the pass/fail block of assertions to be called automatically when an assertion passes or fails:

```
assert property (myprop) call_Cpass(); else call_Cfail();
```

Taking a specific action via direct calls to C (or Verilog) functions based on the assertion failure or success is vital to producing diagnostic and coverage information on the assertions. SystemVerilog provides the customization needed to produce this verification information.

In addition to the power and flexibility benefits of language unification in SystemVerilog, there are considerable performance advantages as well. The unification allows simulation optimization algorithms to be applied to the assertions, testbench and design together, eliminating the communication overhead that has long been a bottleneck between HDL simulation and separate verification tools. In fact, the assertion

⁶ Because assertions are unified, they can be encapsulated in modules, programs or interfaces, which are all supported by the existing library mechanisms already familiar to Verilog users.

⁷ The new callbacks could be used to allow a separate assertion language to interact with SystemVerilog, but since the assertions are already built into the language, there is no need to go to this trouble.

scheduling mechanism mentioned earlier was designed as an extension to existing Verilog scheduling exactly to allow such optimizations to carry over. It is clear that the usability and performance benefits of SystemVerilog, including assertions, are achieved mainly because these concepts are unified in a single language, allowing a single simulation platform to bring all of these concepts together from an implementation perspective as well⁸.

Conclusion

There is consensus among EDA users and vendors that assertion technology is a key asset in addressing the verification challenge, especially as an increasing number of engineering teams report success with assertions for increasing verification productivity as well as finding functional bugs rapidly. However, as discussed in this paper, the multitude of benefits, including ease-of-use, predictability and portability, can only be achieved when the assertions are part of the same language that describes both the design and the testbench: that language is SystemVerilog and is built on the familiar Verilog language.

Any serious examination of assertion language choices for adoption into a verification methodology must include an examination of all the design and verification languages that touch the methodology along with an understanding of how these languages will interact. Indeed, the means to provide the interaction itself may be the subtlest yet most important requirement for successful deployment of assertions.

SystemVerilog is a unified language that contains design and verification constructs, including assertions, and explicitly defines how they interact. As a result, engineering teams using SystemVerilog will know how to:

- make the assertions communicate to the testbench and vice-versa;
- customize messaging resulting from the assertions;
- create calls to Verilog and C functions dependent on the success or failure of the assertions; and
- minimize assertion code by inferring design control structures.

Furthermore, SystemVerilog assertions will eliminate the debugging of mismatches between simulation and formal tools because of the consistent understanding by such tools on how to evaluate assertions.

From the user's perspective, the unification of assertion capabilities with the design and verification language provides many benefits. On the other hand, from the industry perspective, unification means consolidating disparate assertion language efforts to provide a single language standard that encompasses the functionality required by users. Unified assertions in the SystemVerilog language allow EDA vendors to focus on advancing tools with powerful functionality for streamlined flows, rather than assigning resources to support multiple languages. As a result, users benefit from the availability of better tools and are able to create verification environments that are based on a single standard. Such environments can be based on tools from multiple vendors because of the consistent interpretation of design and verification behavior provided by the SystemVerilog standard. And because these SystemVerilog-based environments support all aspects of the design and verification flow, they enable unparalleled cooperation and communication among design and verification engineers to get their designs working correctly.

⁸ The Synopsys VCS™ HDL simulator extends the Verilog and VHDL native code generation algorithms to SystemVerilog to maximize the performance of the testbench and assertions together with the design.

SYNOPSYS®

700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com

Synopsys and the Synopsys logo are registered trademarks of Synopsys, Inc. All other brands, products or service names mentioned herein are trademarks of their respective holders and should be treated as such. All rights reserved.

Printed in the U.S.A.

©2003 Synopsys, Inc. 11/03.TM. WO 03-11787