cadence

SystemVerilog Assertions (SVA) EZ-Start Guide

August 2006

SystemVerilog Assertions (SVA) EZ-Start Guide

In general, there are three components to efficient verification: stimulus generation, coverage, and checking. Cadence Design Systems, Inc. provides companion EZ-Start packages for each of these three areas. The focus of this EZ-Start package and its accompanying executable example is assertion-based verification (ABV). Specifically, dynamic ABV simulation using the SystemVerilog assertion language (SVA).

This document is a self-guided introduction to using dynamic ABV and writing SVA. The following sections describe the three major steps involved in ABV:

- Picking a focus area
- Identifying behavior and mapping to property types
- Implementing assertions

Picking a Focus Area

The first step in ABV is to pick a focus area for your assertions. In general, you want to start with areas that are likely to contain bugs or that will benefit from additional visibility. For example, the following lists areas that typically have a good ROI for assertions and, therefore, represent good starting points for assertions.

- Interfaces between blocks
- Control logic that contains corner cases
- Critical functions within the design

Note: If the schedule permits, this list can be expanded to include other areas within the design.

Identifying Behavior and Mapping to Property Types

After you choose a focus area, you can then identify interesting behavior within the focus area and then express them in natural language form. Once you have expressed the behavior in natural language form, you can map the behavior to various property types. A *property* is a description of the functional behavior of signals over time. An assertion states that a given property is an important aspect of the behavior of the design, and that the design must behave consistently with this property. Writing assertions starts with defining the properties of a design.

The following table lists questions that can help identify the different types of properties in a design. Each of these questions map to a property type that can be used to create templates for your assertions.

Table 1 Basic Questions and Property Types

Question	**Property Type**
Are there signals that have a set behavior that must occur independent of time?	Invariants
Are there signals that have a set behavior that must occur within a certain time frame?	Bounded Invariants
Does the design contain boundary conditions that must trigger a set behavior?	Boundary Cases
Are there ways to specify values or sequences that would describe an error condition?	Bug Identification
Is the behavior of certain signals critical to the functionality of the design?	Signal Values

Note: Although this set of questions is not exhaustive, it is a good starting point for developing assertions.

The following sections describe each of these property types.

Invariants

An invariant is a condition that must always (or never) occur. Invariants are one of the easier property types to describe, because they do not involve time. Examples of invariants are:
- One-hot signals
- Boolean conditions that must always occur, or never occur
- Special cases where an object must hold true, given a particular condition

Bounded Invariants

A bounded invariant is a condition that must be invariant during a bounded period of time. Most often there are signals within the register transfer level (RTL) that can help identify this window of time. Otherwise, you can use auxiliary code to create a *start* and *end* signal that will identify this window and improve readability.

Boundary Cases

Bugs often hide in boundary cases. You can develop an assertion that ensures a boundary condition produces the expected behavior.

Bug Identification

Bug identification assertions describe behavior that must never occur in a design. The advantage of this type of assertion, opposed to describing behavior that must happen, is that RTL code does not get duplicated, which is usually a concern when you are writing RTL assertions.

Signal Values

One of the most basic ways to describe a signal's behavior is to identify when the signal:
- Must have a value of 0 or 1
- Transitions from 0 to 1, and from 1 to 0
- Retains its current value
- Changes value

If this can be done without duplicating the RTL (or by a person other than the designer), then this is the most complete way to develop an assertion. This technique might be best for critical signals.

Implementing Assertions

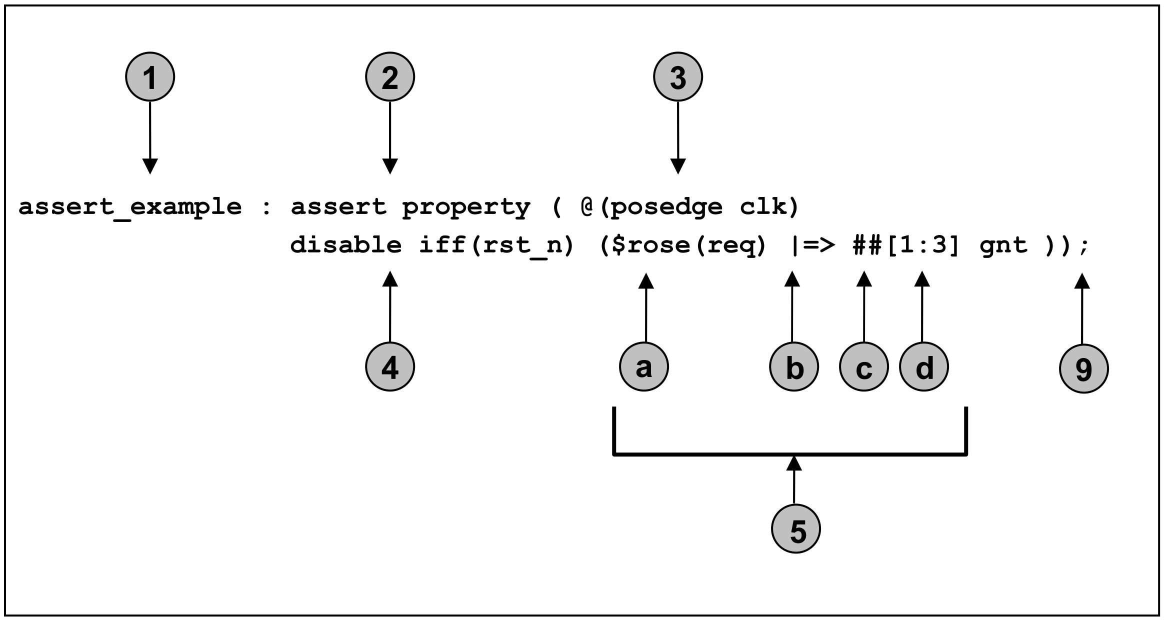
Now that you have selected your focus areas, extracted interesting behavior, translated that behavior into natural language form, and mapped the behavior to the various property types, you can express these property types in actual SVA syntax. Before doing this, you need to understand some basic SVA syntax.

Basic SVA Syntax

This section uses a sample Signal Value assertion to illustrate the fundamental SVA constructs.

The following assertion is derived from the following natural-language statement: *"If there is a rising edge on the request, then a grant must be issued within 1 to 3 cycles."*

Figure 1 Sample Signal Value Assertion



```
assert_example : assert property ( @(posedge clk)
                 disable iff(rst_n) ($rose(req) |=> ##[1:3] gnt ));
```

You can then break down the example into the following:

1. Assertion Label—Identifies the assertion. This is used for reports and debugging. (required)
2. Directive— Specifies how the assertion is used during the verification process--as an assertion or constraint, or for collecting coverage information (required)
3. Clocking—Indicates how or when the signals in the assertion are sampled (required)
4. Disabling Condition—Disables the assertion during certain conditions (optional)
5. Property Expressions (required)

   a. System Functions—Indicates one of the SVA system functions that are used to automate some common operations. For example: $rose, $fell, $past, and $onehot. Refer to the “Writing SystemVerilog Assertions” chapter of the *Assertion Writing Guide* for information on the SVA system and sampled-value functions that are supported in the current release.

   b. Implication Operator (|-> or |=>)—Specifies that the behavior defined on the right-hand side of the operator be checked only if the condition on the left-hand side occurs. There are two forms of the implication operator: overlapping (|->) and non-overlapping (|=>).

   The overlapping operator indicates that the last cycle of the LHS sequence overlaps the first cycle of the RHS sequence.

   The non-overlapping operator indicates that there is no overlap and that the first cycle of the RHS sequence must occur one cycle after the LHS sequence completes.

Note: When you are trying to capture an assertion in the standard written form, the implication operator typically maps to the word "then".

c. Cycle Operator (##)—Distinguishes between cycles of a sequence. Cycles are relative to the clock defined in the clocking statement. Use a fixed number (`##n`) to specify a specific delay or a range operator (as specified in the next sub-bullet) to specify a range.

d. Non-consecutive repetition operator ([*])—Enables the repetition of signals. Use the form `[*n]` to represent a fixed repetition, or `[*n:m]` to specify a range of repetition from `n` to `m`. You can also apply the range operator to the cycle operator.

6. End of Statement Delimiter—Indicates the end of an assertion (required)

Implementing Property Types

You can leverage the syntax described in the previous section to create templates for each of the property types discussed earlier. The following table maps the different property types to SVA constructs that you can use in property expressions.

Note: This table suggests SVA syntax that can be used in property expressions. This table does not cover the other aspects on an assertion, which were discussed in the previous section (such as assertion labels, directives, clocking, disabling conditions, and so on).

Table 2 Property Types and SVA Syntax

Question	**Property Type**	**Useful SVA Syntax**
Are there signals that are supposed to behave a certain way independent of time?	Invariant (onehot)	`$onehot(), $onehot0()`
	Invariant (Illegal Combinations)	`! (Boolean condition)`

	Invariant (Special Cases)	`(special case) |-> (expected behavior)`
		`(special case) |=> (expected behavior)`

Are there signals that are supposed to behave a certain way within some window of time?	Bounded Invariants	`(`*`start`*`) |-> ((!`*`finish`*` && `*`cond`*`)[*n:m] ##1 `*`finish`*` && `*`cond`*`)`
		`(`*`start`*`) |-> ((!`*`finish`*` && `*`cond`*`)[*n:m] ##1 `*`finish`*`)`
		`(`*`start`*` && !`*`finish`*`) |=> ((!`*`finish`*` && `*`cond`*`)[*n:m] ##1 `*`finish`*`)`
		
		Note: `##1` indicates a delimiter—not a 1-cycle delay

Are there any boundary conditions in the design in which some behavior needs to be exhibited when that condition is reached?	Boundary Cases	`(boundary case sequence) |-> (expected sequence)`
Are there ways to describe values or sequences that would describe an error condition?	Bug Identification	`Not (sequence)`

Are there signals that are critical to the functionality of the design?	Signal Values	`(condition |=> $rose(sig))`
		`(condition |=> $fell(sig))`
		`(condition |=> sig == $past(sig))`
		`(condition |=> sig != $past(sig))`

Arbiter Example

You can now apply this ABV process to the arbiter example described in the *Arbiter Specification*,.which is included with this document. The arbiter example is typically a very good focus area for assertions, because it is control logic that often contains many corner case conditions.

Mapping the Behavior

The following lists arbiter example's interesting behavior and maps this behavior to the various property types.

Table 3 Arbiter Behavior Mapping

Behavior	**Property Type**
Grant signals are zero one hot.	Invariant
Once granted, the Busy signal must be asserted until Done.	Bounded invariant
The arbiter must be fair for B.	Boundary Cases

You must never see a GntB, when only ReqA is asserted.	Bug Identification
Busy must be asserted one cycle after a grant.	Signal Values

Creating the Assertion Syntax

After you map the behavior to a property type (Table 3), you can create the actual syntax for the assertion.

- **Invariant**—Grant signals are zero one hot.

```
assert_grant_zeroonehot : assert property (@(posedge clk)
              disable iff (!ResetN) $onehot0({GntA, GntB}));
```

- **Bounded invariant:** Once granted, the Busy signal must be asserted until Done.

```
assert_Busy_until_Done : assert property (@(posedge clk)
  disable iff (!ResetN) (GntA || GntB) |=> Busy[*0:$] ##1 Busy && Done);
```

- **Boundary cases**—The arbiter should be fair for B.

```
assert_fair_for_B : assert property (@(posedge clk) disable iff
 (!ResetN)(GntA ##1 Busy[*0:$] ##1 Done&&ReqA&&ReqB |=> GntB));
```

- **Bug identification**—You must never see a GntB, when only ReqA is asserted.

```
assert_never_grant_wo_req : assert property (@(posedge clk)
           disable iff (!ResetN) not (ReqA && !ReqB ##1 GntB));
```

- **Signal values**—Busy must be asserted one cycle after a grant.

```
assert_Busy_active : assert property (@(posedge clk)
          disable iff (!ResetN) (GntA || GntB) |=> $rose(Busy));
```

Running the Assertions in the Simulator

Use the following steps to run these assertions in the Incisive Design Team Simulator:

1. Go to the directory that contains the arbiter example:

   ```
   cd <example_install_dir>/ius
   ```

2. Use the following command to run the example:

   ```
   run_sim
   ```

   In the console window, observe that the `assertion -summary` command shows that all the assertions have been executed, but one assertion fails.

3. From the toolbar, click the *Assertion Browser* button . The Assertion Browser displays a flat list of all of the assertions in a design and provides the assertion's name,

module, instance, class, finished count, failed count, and so on. The assertions are listed in table format.

4. Double-click any assertion. This displays the source code for the assertion. Notice how the assertions are embedded directly into the arbiter RTL file.

5. Close the Source Browser.

6. Select the `assert_fair_for_B` assertion. Add it to the waveform by selecting it and clicking on the *Waveform* button from the toolbar.

   Note: The waveform displays the state of the given assertion. The red circle indicates where the assertion fails and represents the starting point for debug.

7. Using the steps discussed in this document and in the arbiter specification, you can add more assertions to the `<example install dir>/rtl/arbiter.v` file and rerun.

Conclusion

Through the course of this document, you have been exposed to an ABV process that uses SVA. This includes picking focus areas for assertions, identifying properties that are characteristic to the design, implementing assertions using SVA, and viewing assertions in a real example. After completing these steps, you can try implementing SVA in your own designs.