cadence®

Assertion Writing Guide

Product Version 14.2

January 2015

Document Last Updated: March 2015

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

1

Introduction to Assertions

Assertions in Verification

Assertion checking is a simple addition to your simulation-based verification methodology. All you need to do is add monitors to your design. These monitors are called assertions. During simulation, these monitors watch to see

- If a specific condition occurs, or
- If a specific sequence of events occurs

The monitors generate warnings or errors if required conditions or sequences fail, or if forbidden conditions or sequences occur.

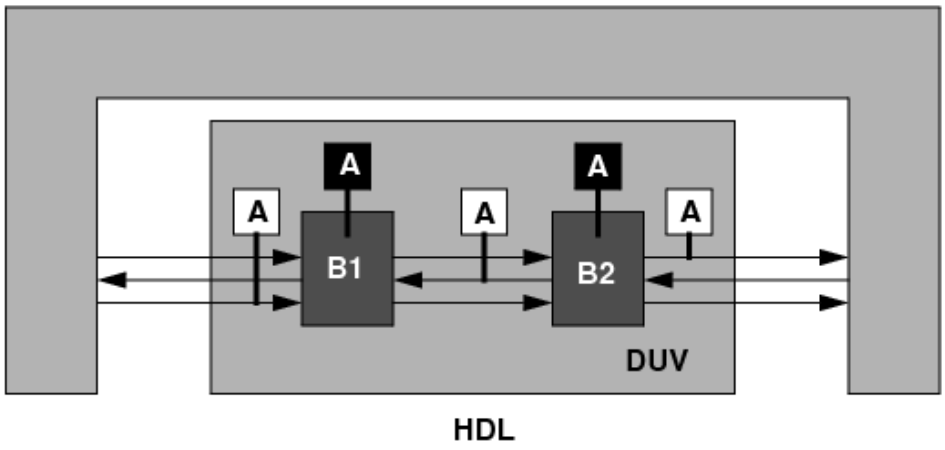
Using Assertions

Assertions written to detect errors have two main uses:

- You can use assertions to monitor the interface of a module.
  Placing monitors on a module interface can help when integrating a block into a design, because the monitors will quickly identify when the protocols or sequences of signals are incorrect. This feature can also be very useful when reusing a design, or when providing IP to other organizations.

  In Figure 1-1, the white boxes marked "A" are the assertions in a design that monitor the inputs to a module.

Figure 1-1 Assertions in a Design



- You can use assertions to monitor signals within a block. These monitors help verify that the signal is behaving properly during circuit operation.
  Placing monitors on a signal within a block can help track down intermittent problems within a design, because the monitor is always watching the signal for correct operation.

  In Figure 1-1, the black boxes marked "A" are the assertions in a design that monitor signals within a block.

ABV Documentation

For information that will help you quickly get started with assertion checking, see the following Cadence manuals:

- Assertion Writing Quick Start
  Provides hands-on exercises to get you started with dynamic simulation of Property Specification Language (PSL) assertions in the Verilog, SystemVerilog, VHDL, and SystemC flavors; and with SystemVerilog Assertions (SVA). Also includes printable quick reference pages for PSL, SVA and ABV Tool Commands.

For detailed information about writing and using assertions, see the following Cadence manuals:

- *Assertion Writing Guide*
  (This manual) Describes how to write PSL assertions for Verilog, SystemVerilog, VHDL, and SystemC designs, and how to write SVA assertions for SystemVerilog designs.

- Assertion Checking in Simulation
  Describes how to enable assertion checking, how to control it, and how to interpret the results.
- *SystemC Simulation Reference*
  Chapter 6, "Using SystemC PSL," describes how to use the SystemC flavor of PSL.
- Incisive Assertion Library Reference
  Describes the contents and use of the Incisive Assertion Library (IAL), a set of predefined verification modules provided to facilitate adoption of assertion-based verification.
- ICC User Guide
  Describes how to use PSL and SVA constructs for control-oriented functional coverage analysis.

For quick-reference guides to Cadence assertion support, see the following Cadence manuals:

- ICC Quick Reference and ICC Quick Start
  Provides a quick-reference guide to the Incisive comprehensive coverage solution.

2

Basic Assertion Concepts

About Assertions

An assertion is a behavioral description that can function as any or all of the following:

- A specification of required behavior
- A statement of assumptions about system inputs
- An active element that checks for design errors during simulation or emulation
- A formal statement that can be verified exhaustively using a formal analysis tool
- A means of precisely documenting interface requirements for IP users
- A means by which designers can efficiently communicate with verification engineers
- A source of implicit coverage points for simulation coverage analysis
- A means of capturing and recording transactions within a design
- An element of a test plan

Assertions are written by various people at various times during the development of a product, from conception through design and implementation, and during verification. Assertions can be written at different levels of abstraction, in various hardware design language contexts.

Assertions take some effort to write. At the same time, they can make design verification much more efficient, can simplify debugging, and can significantly improve overall productivity. This guide explains how to write assertions most effectively, in order to achieve increased productivity.

Assertion Languages

Assertions can be written in many ways, even in general-purpose languages such as C. However, it is more efficient to use languages designed specifically to describe behavior over time in a manner that fits well within the typical design verification flows used today. This guide provides an introduction to two such languages, PSL and SVA.

- PSL is the 1850 IEEE standard for PSL. Chapter 3, "Writing PSL Assertions," provides an overview of PSL.
- SVA is the SystemVerilog Assertions subset of the 1800 IEEE standard for SystemVerilog. Chapter 5, "Writing SystemVerilog Assertions," provides an overview of SVA.

Assertions and Coverage

Assertions are related to monitors used for functional coverage. In fact, assertions can even be used as coverage monitors, or implicit coverage monitors can be inferred from assertions. Both assertions and coverage monitors track the behavior of the design during simulation, and detect when a specified behavior occurs. However, assertions and coverage monitors differ in how they are interpreted.

A design with assertions must always exhibit the behavior specified by those assertions. If it does not, the assertions will *fail*, indicating an error in the design's behavior. In contrast, a design with coverage monitors might or might not exhibit the behavior specified by the coverage monitors. If it does, the coverage monitors *finish*, indicating that the testbench has successfully stimulated the design to test the specified behavior. Assertions can also finish when the behavior they require occurs in the design, so assertions can often be used as coverage monitors as well.

Although assertions contribute to coverage analysis, coverage involves many other elements as well. For more information about coverage analysis capabilities within Incisive, including the use of assertions for monitoring coverage, refer to

- In your Incisive installation--"*ICC Overview* " in the *ICC User Guide*
- In your IES installation--"Coverage-Driven Verification with Enterprise Manager" in the *Enterprise Manager User Guide*

Assertion Language Concepts

Assertions specify required behavior, by

- Describing the behavior of interest
- Specifying that the described behavior is required

Describing Behavior

PSL and SVA model the behavior of a system in terms of a hierarchy of concepts:

- Conditions--A condition is a relationship among signals in a particular state of the design--that is, at a particular instant in time. Conditions are represented as Boolean expressions, which are expressions that can be interpreted as either True or False.
- Sequences--A sequence is a series of conditions over time. Sequences are represented using an extended form of regular expression syntax, which makes it possible to describe a variety of sequential behaviors in a concise manner.
- Properties--A property is a relationship among conditions, sequences, and subordinate properties over time. Properties are represented using logical implication operators, which allow if/then checking of an assertion, and temporal operators, which check the temporal relationship among a set of design elements.
- Assertion statements--Executable statements that describe a behavior to be monitored. Assertion statements contain a directive, such as `cover`, `assert`, `assume`, or `restrict`, that indicates what type of monitoring is desired. Assertion statements can contain instances of property and sequence declarations.

To enable reuse of behavioral descriptions, PSL and SVA also include declarations, which allow you to describe a behavior as a sequence or a property, and to give that description a name. The sequence or property name can then be used elsewhere to represent the described behavior. Declarations can be parameterized to increase their reusability.

Specifying Requirements using Assertions

PSL and SVA impose requirements through the use of *directives*. The most commonly used directive is the `assert` directive, but there are other directives as well.

There are various ways to interpret the requirements imposed by directives. The most literal view is that directives impose requirements on the behavior of the design and its environment--they specify what the design must do, and the ways in which the design can be used. An alternative view is that directives impose requirements on the verification process--they specify how tools must use behavioral specifications to verify the design.

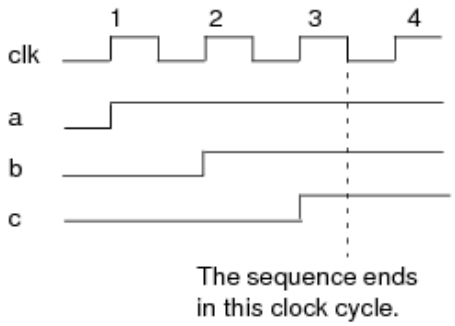
Which view you adopt will depend upon the methodology you select for verification. A more manual, user-directed methodology might suggest the second view; a more automatic methodology might suggest the first view. This manual adopts the first view, interpreting directives as requirements on the behavior of the design and its environment.

Comparing Specified and Actual Behavior with Assertions

Assertion checking compares the behavior described by assertions to the actual behavior of the design, and reports the results. The comparison determines whether or not the behavioral description in the assertion has been satisfied by the design behavior as follows:

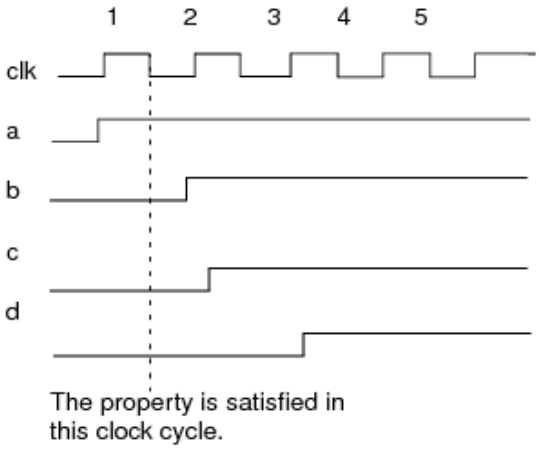
- Booleans
  A Boolean is satisfied when its evaluation returns True.
- Sequences
  A sequence is satisfied when every Boolean in the sequence is satisfied. Most sequences are expressed relative to some clock, rather than the simulation cycle. For example, a sequence states that *if* `a` *is true, then* `b` *must be true in the next negative edge of the clock, and* `c` *must be true in the cycle after that*. The sequence is tightly satisfied in the clock cycle in which `c` evaluates to true (Figure 2-1).

Figure 2-1 Satisfying a Sequence



- Properties
  A property is satisfied, or *holds*, in the *first cycle* in which every sequence and Boolean in the property is satisfied.
  For example, a property states that *If* `a` *is true at the negative edge of the clock, then* `b` *must be true on the next negative edge of the clock. When this behavior occurs, check that* `c` *is true on the negative edge of the clock in which* `b` *is true, and that* `d` *is true on the negative edge of the clock after that.* The sequence takes place over a span of three clock cycles. If all of the behaviors hold, it is said to be satisfied in the first clock cycle, as shown in Figure 2-2.

Figure 2-2 Satisfying a Property



Properties can be classified into Single Cycle and Multi Cycle properties as explained below:

- Single Cycle properties are simple boolean/same cycle implication/one cycle sequence properties which require just one clock cycle to evaluate.

  Following are some examples of the single cycle properties:

```
property single_cycle_bool;
  @(posedge clk) disable iff(reset) a && b;
endproperty

property single_cycle_impl;
  @(posedge clk) disable iff(reset) a |-> b;
endproperty

property single_cycle_seq;
  @(posedge clk) disable iff(reset) a ##0 b;
endproperty
```

  All the above properties require 'a' and 'b' to be true in the same clock cycle.

- Multi Cycle properties span over more than one clock cycle and require two or more clock cycles to activate and evaluate. Following are some simple examples of multi cycle properties:

```
property multi_cycle_impl;
  @(posedge clk) disable iff(reset) a |=> b;
endproperty

property multi_cycle_seq;
  @(posedge clk) disable iff(reset) a ##1 b ##1 c |=> d ##5 e;
endproperty
```

- Multi cycle properties include ranges and repetitions across number of clock cycles.

Assertion Categories

The following describes the different ways in which assertions can be activated and evaluated during processing.

Concurrent Assertions

Concurrent as applied to assertions means that sequence detection runs concurrently with the simulation; that is, the simulation continuously monitors design behaviors. The simulation evaluates concurrent assertions when signals in the design that are referenced in the assertions change values.

Procedural Assertions

A *procedural* assertion is evaluated when the flow of control in a procedural block of code reaches the assertion.

You can use procedural assertions in a Verilog `always` block with procedural code, such as a `case` statement or an `if-then-else` block. They will be evaluated depending on which branch is taken; assertions will be activated only in the context in which they are important.

Immediate Assertions

Immediate assertions are procedural assertions that can check only a single combinational condition; they cannot involve temporal (sequential) operators. These assertions are also referred to as simple immediate assertions.

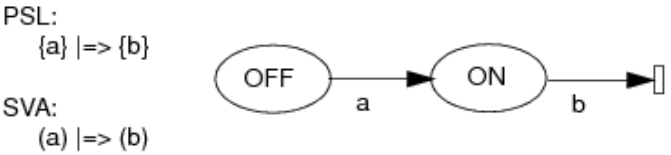
Deferred Assertions

Deferred assertions are immediate assertions that are used to suppress errors that occur due to glitching activity on combinational inputs to immediate assertions. These assertions are also referred to as deferred immediate assertions.

Enabling, Fulfilling, and Discharging Clauses

The core part of an assertion is the *property*. The property describes the condition or sequence of events for which you want the simulation to check. A property can contain an optional enabling condition, a fulfilling condition, an optional discharging condition, and an optional clocking condition.

As shown in the following figure, `a` is the enabling condition and `b` is the fulfilling condition.



- The *enabling condition* can be any Boolean or sequential expression. You can string multiple enabling conditions and implication operators together to form a complex enable. The assertion is considered to begin when the *first* sequence of the enabling condition evaluates to true.
- The *fulfilling condition* defines the behavior to be checked. It is checked on every verification cycle by default. If it has an associated enabling or clocking condition, that condition defines when to check it.
  In the following PSL example, the simulation continually monitors the `in1` signal at every positive edge of the default clock to see if it is true; this is the enabling condition.

  ```
  default clock = (posedge clk);

  assert always (in1 -> next in2);
  ```

  The `next in2` clause specifies a fulfilling condition. If `in2` becomes true in the next simulation cycle after `in1` becomes true, the fulfilling condition has evaluated to true. If it does not, the assertion fails.
- The *discharging condition* indicates when to stop checking the behavior. When a discharging condition becomes true, the simulation stops checking the fulfilling condition. In

PSL, for example, discharging conditions can be `until`, `until_`, `async_abort`, `sync_abort`, and `abort` followed by a Boolean expression. In SVA, the discharging condition is specified using the `disable iff` construct.

In the PSL example that follows, the `until` keyword specifies a discharging condition. When `in1` is true, `in2` must be true in the next cycle, and remain true until the cycle in which `in1` goes false, even if `in2` is true forever and `in1` is never false.

```
// psl IN2_UNTIL_NOT_IN1: assert always (in1 -> next (in2 until !in1));
```

- The *clocking condition* specifies when the conditions are to be checked, such as a particular edge of the clock.

A property that is asserted will be checked during simulation. When the property's enabling sequence occurs--when the endpoint of the enabling sequence is reached--the assertion expects the fulfilling sequence to occur next. To see if this is true, the assertion is checked:

- If the fulfilling sequence does occur, the assertion *finishes*.
- If the fulfilling sequence does not occur, the assertion *fails*.

At any given time during simulation, the number of times each assertion has been checked, has finished, or has failed is reported.

In simulation, properties are checked along a particular *trace*, or sequence of states through which the design progresses as a result of its inputs.

- If an assertion is not checked at all in a given simulation run, its enabling condition never occurred during that run.
- If it has been checked in a given run and has finished, the design's behavior during the run conformed to the behavior described by the asserted property.
  However, the assertion was checked only for that specific input stimulus, and only up until the simulation ended. It is always possible that an assertion that did not fail in a given simulation might fail if that simulation ran longer, or it might fail in another simulation.

In formal analysis, assertions are checked for all possible traces, starting at an initial state. If an assertion does not fail in formal analysis, it will not fail in any simulation run, provided that the constraints used to verify the property are satisfied by the context in which the design appears.

Common Assertion Forms

Invariant Assertions

An invariant condition is a condition that must hold throughout the operation of the design. The condition must always or never be true.

For example:

- A Grant never occurs without a Request.
- The `enable_n` line must always be high on the falling edge of `write_n`.
- The `read_n` and `write_n` lines must never be low at the same time.

Conditional Assertions

Conditional assertions specify that one condition must occur if another condition occurs. These assertions specify activity that is required or disallowed as a consequence of some other activity. For example:

- If `m_task` = `read`, it must always be followed by `read_n` low for two samples, followed by `read_n` high for one sample.
- If A receives a Grant, then B does not.

Temporal Assertions

You can use assertions to set bounds on the behavior of a design. Temporal assertions can be bounded or unbounded.

- Bounded
  Bounded assertions describe a condition that must occur within certain limits. For example, if `in2` is true, `out1` must become true in the next cycle.
- Unbounded
  Unbounded assertions describe a condition that must eventually occur. For example, the `IOReq` signal must be followed by an `IOGrant` at some indeterminate time in the future.

3

Writing PSL Assertions

PSL Characteristics

PSL Domain

PSL provides comprehensive property specification and assertion capabilities for VHDL, Verilog, SystemVerilog, and SystemC.

Each flavor of PSL adopts some of the syntax and semantics of the corresponding HDL. In particular, each flavor allows the use of HDL expressions with the same syntax and semantics as in the corresponding HDL. Such expressions are part of the Boolean layer of PSL.

For a summary list of the Property Specification Language (PSL) constructs supported by the Incisive simulator, see the Assertion Writing Quick Start guide.

For detailed information about PSL, refer to the *1850 IEEE Standard for Property Specification Language (PSL)*, 17 October 2005.

Note: PSL declarations share the design name space. All naming rules of the respective design languages apply to PSL declarations as well.

PSL Structure

PSL is used to define *properties*--the temporal relationship among a set of design elements. Properties are built from Booleans, sequences, and subordinate properties. There are two common styles of properties in PSL:

- Sequence-based ("suffix implication") properties

  Sequence-based properties are built using the suffix implication operators, `|->` and `|=>`. For example:

  ```
  {a[+];b} |=> {c[*];d}

  {{[*]; a} : {b[*3]}} |-> c;
  ```

  These properties describe behavior patterns using sequential extended regular expression (SERE) notation. The left operand must be a sequence; the right operand can be either a sequence or another property.

For more information, see "The PSL |-> Suffix Implication Operator" and "The PSL |=> Suffix Next Implication Operator" .

- Keyword-based properties
  Keyword-based properties are built using keyword operators and the Boolean implication operator, `->`. For example:

  ```
  always a -> next b until c
  ```

  These properties describe behavior patterns using English words that represent temporal relationships. The keyword operators include `always` , `never` , `until` , `until_` , `before` , `before _`, `eventually!` , `next` , `next_a` , `next_e` , and the `abort` operators.

  Keyword-based and sequence-based properties can be intermixed. For example:

  ```
  always {a; b} |=> {c[*]; d} abort e
  ```

  For more information, see "PSL Property Operators" .

  Properties can be nested to express more complex ideas. For example:

  ```
  always ({a; {b[*];c};d} |-> next ((e before f) until g) abort stop) abort reset
  ```

PSL Categories

PSL properties always act as concurrent elements in the design. They monitor the behavior of the inputs and outputs of other concurrent elements in the design. They act like a Verilog `always` block or a VHDL `process`, reacting to changes on input signals.

- Clocked properties react to the clock.
- Unclocked properties react to changes on inputs [1] .

> Although you can embed a PSL assertion within a Verilog `always` block or VHDL `process`, it is not affected by the sensitivity list of the `always` block or `process`, and is not executed as a sequential statement. Any PSL assertion is executed as a separate process, regardless of where you write it.

1	This behavior is a Cadence-specific interpretation of the LRM.

PSL Lexical Structure

The elements of the PSL lexical structure are the following:

Identifiers

Identifiers are names of your choice to uniquely identify properties, sequences, and endpoints. An identifier must start with an alphabetic character, followed by zero or more alphanumeric characters, each of which can optionally be preceded by a single underscore. Identifiers are case-sensitive in the Verilog, SystemVerilog, and SystemC flavors, and case-insensitive in the VHDL flavor.

Keywords

If you have an HDL name that is also a PSL keyword, you cannot reference it directly by its simple name in a PSL property. You can reference the name indirectly, by using its HDL hierarchical or qualified name. PSL keywords that are supported by the Incisive simulator are listed in the following table.

abort	cover	isunknown	restrict
async_abort	default	never	rose
always	ended	next	sequence
assert	endpoint	next_a	stable
assume	eventually!	next_e	sync_abort
before	fell	onecold	until
before_	forall	onehot	until_
boolean	hdltype	onehot0	vmode
clock	in	property	vprop
const	inf	prev	vunit
countones	inherit	report	within

Operators

For any flavor of PSL, the HDL operators, including logical, relational, and arithmetic operators, have the highest precedence. The PSL operators and their precedence are listed in the following table.

Highest	HDL operators	Associativity
|	@	left
	[*] [+] [=] [->]	left
	within	left
	SERE & SERE &&	left
	SERE |	left
	:	left
	;	left
	abort async_abort sync_abort	left
	eventually! next*	right
	until* before*	right
	|-> |=>	right
	->	right
Lowest	always never	right

Assertions are intended to monitor the HDL, not change the values of signals. The increment and decrement operators are not allowed in PSL properties.

PSL Data Types

PSL defines five data type classes: Bit, BitVector, Boolean, Numeric, and String. For each HDL, PSL defines which data types in that HDL can be used where one of these type classes is required.

The following table shows, for each language, the data types (and, for Verilog, objects that contain the implicit data type) that are compatible with each of the PSL type classes. These type classes are used to define

- What kind of arguments are used with built-in functions
- What kind of expressions are allowed to appear as Boolean or Numeric expressions in a sequence or property, or as arguments of a sequence or property

To use a data type that is not mentioned in Table 3-1, you need to either convert it to a supported data type, or use it in an expression that results in a supported data type. For example, to use a variable of type T in a Boolean expression, if T is not one of the types that are listed as compatible with the Boolean type class, you can compare the variable to a value, provided that the comparison operation returns a value of a type that is listed as compatible with the Boolean type class.

The data types listed in this table can be used as formal arguments to PSL properties and sequences. Additionally, any HDL or user-defined data type can also be used.

Table 3-1 PSL Data Type Classes Mapped to HDL Types

PSL Type Classes	Verilog	SystemVerilog	VHDL	SystemC
Bit	reg net wire	bit logic[1]	bit std_ulogic	sc_bit sc_logic
BitVector	reg net wire integer	bit logic[1] integer	bit_vector std_logic_vector std_ulogic_vector signed unsigned	sc_bv sc_lv sc_int sc_uint sc_bigint sc_biguint
Boolean	reg net wire	bit logic	boolean std_ulogic	bool sc_bit sc_logic
Numeric	reg net wire integer	bit logic[1] integer	standard.integer	bool char short int long long long sc_bit sc_logic sc_int sc_uint sc_bigint sc_biguint
String	string literal	string	string	char* string

1	Includes the Verilog objects that contain values of the built-in logic types, such as reg and wire

PSL Conditions

A *condition* is a relationship among signal values at a given time. For example:

```
(x <= y)
(a && b || c)
onehot(sel)
```

The PSL Boolean layer consists of expressions that represent these conditions. Expressions can be

- HDL Expressions in PSL
- PSL Expressions
- PSL Built-In Functions
- PSL Clock Expressions

HDL Expressions in PSL

Any HDL expression that is legal in an HDL module can also be used in a PSL declaration or directive associated with that module, whether the PSL is embedded in the module or appears in a verification unit bound to the module.

You can also use Verilog and VHDL functions in PSL declarations and directives. For details, see "Using HDL Functions in PSL".

Boolean Expressions

The following guidelines apply to using Boolean expressions in PSL:

- You can use any HDL operator, including comparison operators, in a Boolean expression.
- Signals that can take on the bit values `0' or `1' are usually allowed to appear where a Boolean expression is required in PSL. This includes any signal in Verilog, and signals of certain types in VHDL, SystemVerilog, and SystemC (see Table 3-1).

A Boolean expression is interpreted as True or False according to the rules of Verilog, SystemVerilog, or SystemC, respectively, for interpreting an expression that appears as the condition of an `if` statement, such as a statement that uses an implication operator. Unknown values will be handled according to the rules of the HDL. For example, Verilog does not respond to an X value in an `if` statement by taking both paths, so the Verilog flavor of PSL responds to an X value in the same way. For details, see section 5.1.2 in the *1850 IEEE Standard for Property Specification Language (PSL)*.

SystemC Expressions in PSL

You can include arbitrary SystemC/C++ expressions in PSL properties by enclosing the expressions within an underscore/parentheses combination, called the *literal inclusion construct*:

```
__( SystemC/C++_expressions )
```

When the SystemC PSL parser encounters this literal-inclusion construct, it performs limited data type checking, then passes everything between the parentheses. If an expression in the `SystemC/C++_expressions` is not Boolean, you must use it in an expression that is Boolean--for example, as an argument to a Boolean built-in function, or as an operand of a relational expression.

> As the SystemC PSL parser does not parse for signal names in a literal inclusion construct, value changes in an unclocked assertion will not be detected.

For detailed information, see "Boolean Expressions in SystemC PSL," in " Using SystemC PSL ," of the *SystemC Simulation Reference* .

> The literal inclusion function is Cadence-specific.

PSL Expressions

PSL defines the `->` operator as an additional combinational operator, so that for Booleans A and B, the expression `A->B` is also a Boolean expression, one that is true if either `A` is false or `B` is true.

If the operands are both Boolean, the `->` operator is interpreted as the combinational implication operator. Otherwise, it is interpreted as the temporal "logical if" operator.

For more information, see "The PSL -> Implication Operator".

PSL Built-In Functions

PSL defines a set of built-in functions to detect conditions that are frequently of interest in verification. There are two types of built-in functions:

- Those that determine the values of expressions over time: `rose()`, `fell()`, `prev()`, and

`stable()`.

- Those that determine the values of bits in a vector at a given time: `isunknown()`, `countones()`, `onehot()`, and `onehot0()`.

The built-in functions are governed by the same clocking as the context in which they appear. For example, `prev(x)` is true if `x` was true the last time this property was evaluated.

Data Types for PSL Built-In Functions

"PSL Data Types" specifies Cadence support for the PSL data types used with the built-in functions. For details about the data types, see Section 5.1 of the *1850 IEEE Standard for Property Specification Language (PSL)*, 2005.

Out-of-module references (OOMRs) and external names are supported as arguments to the built-in functions.

For VHDL external names, use the `<< external . name : type >>` syntax and the `-v200X` compile option. For example:

```
entity e is
port(Q: in bit_vector(0 to 3));
end e;

architecture e_a of e is
signal S: bit_vector(0 to 3);
signal Y: integer := 10;
signal X: integer := 50;
begin
    S <= "1010";
    process(Q)
    begin
    --psl property P1(const m) is always (S(0)->Q(m))
     @(rising_edge(<<signal.top.clk:std_logic>>));
    -- psl assert P1(2);
    -- psl assert always {countones(<<signal.top.S:bit_vector(0 to 3)>>) =2};
    end process;
    process(Y)
    begin
     -- psl assert always {<<signal.top.topY:integer>> + Y < X};
    end process;
end e_a;

entity top is
end top;

architecture top_a of top is
signal clk: std_logic:='1';
signal topY: integer := 5;
signal S: bit_vector(0 to 3);
begin
    clk <= not clk after 1 ns;
    S <= "1010" after 1 ns;
    I1: entity work.e;
end top_a;
```

VHDL external name references work only in a pure VHDL context. You cannot reference external names in other languages, such as Verilog. External name references cannot pass through non-VHDL hierarchies, even if they terminate in a VHDL item.

Memories and multidimensional arrays are not supported as arguments to the built-in functions.

For the VHDL flavor of PSL, the following data types are supported for the built-in functions:

```
std.standard.boolean
std.standard.bit
std.standard.integer
std.standard.real
std.standard.bit_vector
IEEE.std_logic_1164.std_logic
IEEE.std_logic_1164.std_logic_vector
IEEE.std_logic_1164.std_ulogic_vector
IEEE.numeric_std.signed
IEEE.numeric_std.unsigned
IEEE.numeric_bit.signed
IEEE.numeric_bit.unsigned
IEEE.std_logic_arith.signed
IEEE.std_logic_arith.unsigned
```

Limitations

- Package referenced signal given as input argument to built-in function
  A signal of any type, is member of package PACK, and cannot be passed as an input argument to a built-in function prev/stable using dot operator (.) whereas built-in function prev is used inside the architecture body.

  Example:

```
-------PACKAGE------------
package pack is
TYPE STATE_TYPE is (
  idle,
  gnt0,
  gnt1
  );
SIGNAL state : STATE_TYPE := idle;
end;

----------ARCH-----------------
use work.pack.all;
architecture arch of dut is
--psl property P2 is always {prev(work.pack.state) = idle}; //state is an enumerated signal
defined inside package scope
--psl assert P2;
begin
end arch;
```

- Use of built-in function inside PSL Sequence along with fusion as well as delay operator
  The fusion operator `:' and delay operator `;' are used inside a PSL Sequence. Built-in functions along with any type of VHDL signal as input arguments are present with fusion and delay operator inside PSL Sequence.

Example:

```
Signal std1 : std_logic := `0';

Signal std2 : std_logic := `0';

--psl sequence S3 is {prev(std1) = jump : prev(std1,2) = transition;

stable(std2)};
```

- Use of nested built-in functions inside PSL Sequence where stable/prev appears as outer built-in and rose/fell appear as inner built-in functions.
- Multi-Dimention array or MD array with index given as input to PSL built-in function.
  A scenario where signal `arr2_rec` of a MD array has been passed to a built-in function stable, is a limitation for this feature.

Built-In Functions for Values over Time

You can use the following PSL built-in functions in your assertions to determine the value of an expression over time.

```
    rose( Bit
)
```

This function indicates the rising edge of a signal. For example, the expression `rose(in1)` is true (has the value 1) if `in1` had the value 0 in the previous time tick and has the value 1 in the current time tick. Otherwise, the expression is false.

In the following example, a specified condition must occur some time after the `intc` signal rises:

```
assert always
(rose(intc) -> eventually! (mw && add==3'b001));
```

```
    fell( Bit
)
```

This function indicates the falling edge of a signal. For example, the expression `fell(in1)` is true (has the value 1) if `in1` had the value one in the previous time tick and has the value 0 in the current time tick. Otherwise, the expression is false.

The value of `fell()` is affected only by the sequence of time as seen by the verification tool. In the following example, a specified condition must hold in the same time tick in which `intc` transitions

from 1 to 0:

```
assert always (fell(intc) -> (data==8'b0000_0000));
```

prev(*Any_Type* [**,** *i*] **)**

This function returns the value of an expression at the previous time tick as seen by the verification tool. If you include a cycle index, *i* , this function returns the value of the expression in the *i* th previous time tick.

For VHDL, the simulation tool must be able to determine the type of the expression at parse time, so it must be a local identifier, not a complex expression of ambiguous width, such as a math operation.

You can use Verilog OOMRs and VHDL external names for the expression. For VHDL external names, use the << *external* . *name* : *type* >> syntax and the `-v200X` compiler option; see "Data Types for PSL Built-In Functions" for an example and limitations. For Verilog and SystemVerilog, you can also use packed structures and enumerated types.

The *i* value is the number of cycles in the past at which to sample the expression. This value can be a statically computable positive integer, or a `generate` statement index variable. For Verilog/SystemVerilog, the *i* value can be a generic, mathematical, or parameter expression. Parameters and mathematical expressions are not supported for VHDL and SystemC.

The following example checks whether the `address` signal has the same value that it had in the previous time tick:

```
assume always
({req && !ack} |=> address == prev(address))
@(posedge clk);
```

To improve performance, avoid large numbers in the argument of the `prev() built-in` function. For example, prev(x,100) must find the value that x had 100 clocks previous to the current clock cycle. Instead, it is more efficient to implement this function with some auxiliary HDL.

stable(*Any_Type* **)**

This function returns True if the value of *Any_Type* is the same as it was in the previous time tick.

In the following example, the `address` signal must be stable in the time tick after the `req && !ack` sequence finishes:

```
assume always ({req && !ack} |=> stable(address)) @(posedge clk);
```

For VHDL, the signed and unsigned types are not currently supported for this function.

ended(*Sequence* **)**

This function is used to detect a sequence match. It returns True for the cycle in which the last component of the sequence is detected, and False otherwise. If the second argument is specified, it is equivalent to

```
ended({Sequence }@Clock_Expression )
```

The PSL `ended()` function can be used anywhere a Boolean can be used:

- In property and sequence definitions
- In modeling layer code (verification units)
- In modeling layer code as an event

Use of `ended()` in HDL is specific to the Cadence implementation, and requires the `assert` compiler option.

When `ended()` is used as an event, the event will trigger when the result changes state, not each time the `ended()` expression evaluates to True. In a Verilog event expression, you can qualify `ended()` with the `posedge` or `negedge` qualifiers.

The `ended()` function is often used for reactive tests. For general information about reactive tests, see Chapter 9, "Writing Reactive Tests using Assertions." For information about using the PSL `ended()` function for reactive tests, see "Using PSL Reactive Test Techniques".

This function is not supported for VHDL and SystemC.

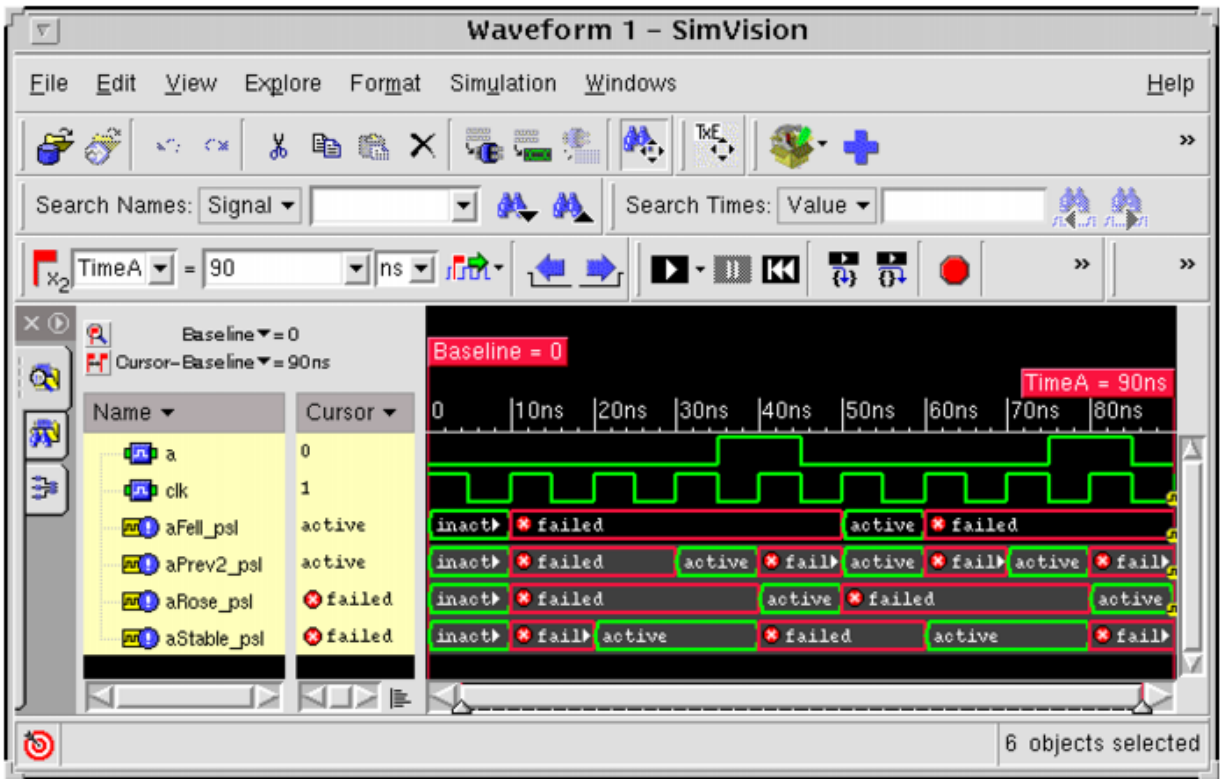
Examples of PSL Values over Time

Figure 3-1 shows the waveforms generated by the following assertions that use the built-in `rose`, `fell`, `prev`, and `stable` functions:

```
aFell_psl:   assert always (fell(a)) @(posedge clk);
```

```
aPrev2_psl:  assert always ( a == (prev(a, 2)) ) @(posedge clk);
aRose_psl:   assert always (rose(a)) @(posedge clk);
aStable_psl: assert always (stable(a)) @(posedge clk);
```

Figure 3-1 Values over Time: rose, fell, prev, and stable



PSL Built-In Functions for Values of Bits in a Vector

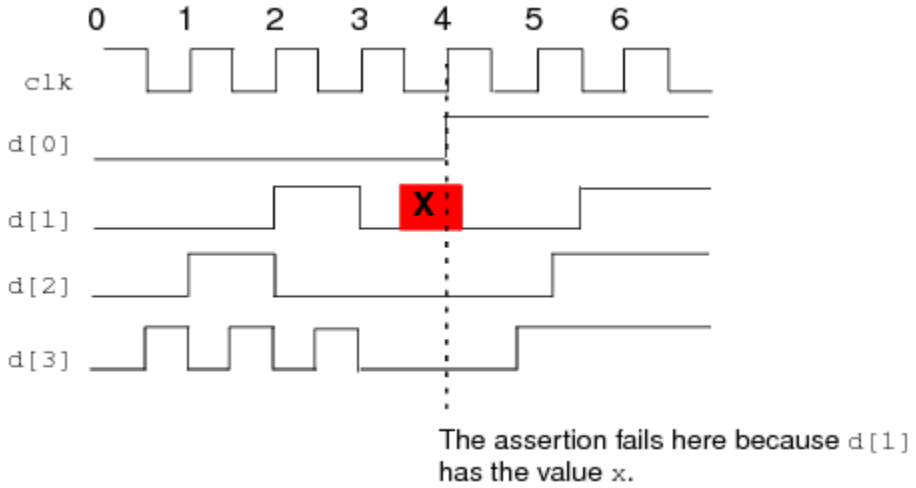
You can use the following PSL functions in your assertions to determine the values of bits in a vector.

```
isunknown ( BitVector )
```

This function returns True if *BitVector* contains any bits that have a value of "unknown"--that is, values other than 0 or 1.

In the following example, the `A_unknown` assertion calls `$isunknown` at each positive edge of `clk`:

```
A_unknown: assert always !isunknown(d) @(posedge clk);
```



countones(*BitVector* **)**

This function returns the number of bits in *BitVector* that have a value of 1. For example:

```
assert always (countones(memword) % 2 == 1);
```

onehot(*BitVector* **)**

This function returns True if *BitVector* contains exactly one bit that has a value of 1. For example:

```
assert always onehot(gnt);
```

onehot0(*BitVector* **)**

This function returns True if *BitVector* contains at most one bit that has a value of 1. The argument can be an out-of-module reference (OOMR). For example:

```
assert always onehot0(enable[115:0]);
```

Differences between PSL rose/fell and Verilog posedge/negedge

The differences between the PSL `rose`/`fell` built-in functions and Verilog `posedge`/`negedge` are the following:

- The `rose` and `fell` functions ignore transitions to and from X and Z values; `posedge` and `negedge` take them into account.

- In the presence of race conditions, `posedge` and `negedge` can produce different results from `rose` and `fell`.
- You can use `rose` and `fell` within a sequence to detect data edges; for example:

  ```
  {rose(a); fell(b)}
  ```

  You cannot use `posedge` and `negedge` in this way.
- The `rose` and `fell` functions are portable between PSL dialects, whereas `posedge` and `negedge` are not.

PSL Clock Expressions

You can ensure that an assertion will be evaluated only at certain times by specifying a clock expression. A clock expression can be one of the following:

- A condition represented by a Boolean expression
- An event represented by an HDL event expression
- An expression using the `rose` or `fell` built-in function

PSL Unclocked Assertions

In the Incisive PSL implementation, an unclocked assertion is evaluated at any change on any signal or bitvector that is referenced by the asserted property, similar to a combinational process or `always` block. When a slice of a bit vector is referenced in an assertion, any change to that bit vector will result in the assertion being evaluated, whether that change is in the referenced slice or not.

For unclocked combinational assertions, this implementation is completely consistent with the PSL LRM definition. For unclocked sequential assertions, this implementation might be different from some other implementation's interpretation of the PSL LRM.

Unclocked sequences, especially those with repetition, are not recommended. For example, the unclocked assertion `assert never in1[*4];` will never evaluate to true, because it requires a change in the value of `in1` to trigger evaluation of the assertion. Therefore, it is likely that the use of unclocked sequences will lead to unexpected results.

Specifying a PSL Clocking Event

@(*event_expression* **)**

In many cases, you might want an assertion to be evaluated only at a specific event, like a clock edge. To specify an event for evaluating assertions, use the clock expression at the end of your property declaration.

The *event_expression* is the same as the event expression that you can use in the sensitivity list of a Verilog `always` block or a VHDL `process` block.

Always use parentheses to associate the clock, because the clock has the highest precedence, and applies only to the property to its immediate left. For example:

```
assert always GNT -> next BSY @(posedge clk);
```

is equivalent to

```
assert always GNT -> next (BSY @(posedge clk));
```

To get the expected behavior, use the following instead:

```
assert always (GNT -> next BSY) @(posedge clk);
```

HDL Clocking Events in PSL

The following declaration will evaluate the assertion on every rising edge of `clk`:

```
memcontrol1: assert never (write && read) @(posedge clk) ;
```

This assertion is evaluated while the enable signal is low:

```
memcontrol2: assert never (write || enable) @(!enable) ;
```

In the following example, the `mem_control1` assertion will be evaluated each time the write or read lines change value:

```
memcontrol1: assert never ( write && read ) ;
```

This declaration will evaluate the assertion on every rising edge of `clk`:

```
memcontrol1: assert never (write='1' AND read='1') @(rising_edge(clk)) ;
```

For Verilog/SystemVerilog and VHDL, you can specify multiple clocks for a property. This feature allows you to specify assertions that cross clock domains.

SystemC Clocking Events in PSL

There are special considerations with respect to clocking in SystemC PSL. For further information, see "Clock Expressions in SystemC PSL" in "Using SystemC PSL ," of the *SystemC Simulation Reference*.

Declaring Default Clocks in PSL

default clock =|is [1] **(** *Clock_Expression* **) ;**

1 = for Verilog/SystemVerilog and SystemC, `is` for VHDL

A default clock declaration defines a clock expression to be applied to any directives within its scope that do not have an explicit clock expression.

When several assertions in a module share the same edge expression or clock, you can define a default clock to use for all of these assertions. The default clock is applied to any directive whose property is not clocked at the highest level--the outermost property in the directive--as though the property were parenthesized. You must define the default clock before the assertions that use it.

The Incisive implementation of PSL lets you define multiple default clocks. Only one default clock is effective at any point in the code. This feature is not part of the PSL standard. The default clock is used by all assertions that follow that default clock definition, until another default clock definition is encountered.

A Boolean expression is evaluated

- When the associated explicit clock expression is true, if there is one; otherwise
- When the clock expression in the most recent default clock declaration is true, if there is one; otherwise
- When any signal in the Boolean expression changes

If you do not want a property definition to be clocked, you must put it before the first default clock definition.

Notes about Default Clocks in PSL:

- Default clock specifications affect the assertions derived from synthesis pragmas, as well as the explicit assertions. For more information about pragma processing, see "Enabling Synthesis Pragma Checking for ABV," in *Assertion Checking in Simulation*.
- In property files, the default clock's effect is limited to the verification unit in which it is defined. It is not inherited. It affects only the `assert` and other directives in the verification unit in which it appears.

PSL Default Clock Examples

The following example illustrates clocking in PSL using default clocks:

```
check_on_signal_change: assert always ( a='1' AND b='1' ) ;

default clock is (rising_edge( clk )) ;
pos_edge_clk: assert always ( a='1' AND b='1' ) ;

my_clock: assert always ( a='1' AND b='1' ) @(rising_edge( my_clk )) ;

default clock is (falling_edge(clk)) ;
neg_edge_clk: assert always ( a='1' AND b='1' ) ;
```

In this example

- The `check_on_signal_change` assertion is an unclocked assertion. It has no clock defined in the assertion, and it is declared before any default clock. This assertion is evaluated whenever `a` or `b` changes value.
- The `pos_edge_clk` assertion is clocked on the positive edge of `clk`. It has no clock defined in the assertion, and it appears after the first `default clock` declaration, so it uses that default clock. This is equivalent to
  ```
  pos_edge_clk: assert always ( a='1' AND b='1' ) @(rising_edge(clk)) ;
  ```
- The `my_clock` assertion is clocked on the positive edge of `my_clk`, as specified in the assertion.
- The `neg_edge_clk` assertion is clocked on the negative edge of `clk`. It has no clock expression

specified in the assertion, and it appears after the second `default clock` declaration, so it uses the clock expression of that default clock declaration. This is equivalent to

```
neg_edge_clk: assert always ( a='1' AND b='1' ) @(falling_edge(clk)) ;
```

PSL Sequences

In general, a sequence is a series of conditions that occur at successive times. For example:

```
{req; ack; read; !req; !ack}
{state==s1; state==s3}
```

> A sequence can be clocked or unclocked. The examples that follow assume that the sequences occur in a context in which the default clock applies.

A PSL *sequential extended regular expression* (*SERE*) is composed of Boolean expressions. It describes single-cycle or multiple-cycle behavior.

A PSL *sequence* is a SERE that can appear at the top level of a declaration, directive, or property. It is composed of Boolean expressions and other sequences. The simplest sequence is a single Boolean expression, enclosed in braces. For example, the following is a sequence:

```
{ req }
```

A sequence can also contain a series of elements, separated by the concatenation operator (**;**) (see "Concatenation".). For example, the following is a sequence

```
{ req ; ack ; !req }
```

Each Boolean expression in the sequence represents a condition, and each condition occurs in the cycle after the previous condition occurs.
A sequence can also contain other sequences. For example, the following sequence

```
{ { req ; ack } ; busy ; { !req ; !ack } }
```

contains three elements:

- A sequence, (`{ req ; ack }`)
- A Boolean expression, (`busy`)
- Another sequence, (`{ !req ; !ack }`).

This expression is equivalent to the following flattened sequence, which says that `req` occurs in cycle 1, then `ack` occurs in cycle 2, then `busy` occurs in cycle 3, then `!req` occurs in cycle 4, and `!ack` occurs in cycle 5.

```
{ req ; ack ; busy ; !req ; !ack }
```

PSL Declarations

PSL includes two forms of declaration related to sequences: the `sequence` declaration, and the `endpoint` declaration. Both define a name associated with a sequence.

A PSL declaration ends with a semicolon.

PSL Sequence Declaration

```
sequence Identifier [( Formal_Argument_List )] =|is [1] { SERE };
```

1 = for Verilog and SystemC, `is` for VHDL

A `sequence` declaration defines a name to represent a whole sequence.

The *`Formal_Argument_List`* contains one or more argument declarations, separated by semicolons. An argument can be `const`, `hdltype`, `boolean`, `bit`, `bitvector`, `numeric`, `string`, `sequence`, or `property`, followed by one or more argument names. For more information about arguments, see "PSL Sequence Arguments".

For each formal argument in the sequence, a sequence instantiation provides a corresponding actual argument. For more information about instantiation, see "PSL Sequence Instantiation".

For example:

```
sequence BusGnt(boolean R,G) = {R[+]; G};
sequence BusRls(boolean R,G) = {!R; !G};
BusGntRls: assert always BusGnt(Rq,Ak) |=> {{BusRls(Rq,Ak)} within [*5]};
```

You can improve simulation performance by using sequences with more specific, and less frequent, conditions. The following example shows an inefficient use of a sequence:

```
sequence command_silence is
{((m_command_en_n='0') or (m_command_in='1')) [*8 to inf]};
```

On the second and later cycles of this sequence, not only does the simulation begin to count, but it creates new instances of the sequence. A more efficient solution is to use auxiliary HDL code, as follows:

```
process (m_clk) begin
    if (falling_edge(m_clk)) then
     if ((m_command_en_n='0') or (m_command_in='1')) then
         m_counter <= m_counter + 1;
     else
         m_counter <= 0;
     end if;
    end if;
end process;
```

The sequence can then be defined as follows:

```
sequence command_silence is {m_counter >= 8};
```

PSL Endpoint Declaration

endpoint *Identifier* [**(** *Formal_Argument_List* **)**] **=|is** [1] **{** *SERE* **};**

1 = for Verilog and SystemC, `is` for VHDL

The `endpoint` construct has been replaced by the built-in `ended()` function in the *IEEE 1850 PSL standard*. The `endpoint` declaration is still supported in the Incisive simulator for backward compatibility.

An `endpoint` declaration defines a name that represents the end of a sequence.

For example, the following endpoint declaration defines the name `RunPath` to mean the end of the specified sequence. Any reference to `RunPath` is the same as a reference to a Boolean signal that is True in the last cycle of any trace that tightly satisfies the sequence in the `RunPath` definition (see "Satisfaction and Endpoints in PSL").

```
  endpoint RunPath is { (state=idle; state=start; (state=run)[*3]; state=done;
state=idle) };
```

The rules for formal and actual arguments and instantiation are the same for `endpoint` as they are for the `sequence` declaration ("PSL Sequence Declaration").

> Endpoints in Boolean expressions and the modeling layer are not supported. You can use the built-in `ended()` function instead.

> You can define a name for a single condition by declaring an endpoint for the sequence that contains a single Boolean expression representing that condition. The endpoint name can then be used wherever a Boolean expression is allowed within a sequence or property.

> It takes time to construct an endpoint model, which can affect simulation performance. To avoid unnecessary performance degradation, do not create endpoints that will not be used.

Satisfaction and Endpoints in PSL

Boolean expressions, sequences, and properties are said to be *satisfied* as described in "Comparing Specified and Actual Behavior with Assertions".
An endpoint is satisfied--evaluates to True--in the *last cycle* in which the last Boolean in its sequence is satisfied.

PSL Sequence Arguments

Sequence declarations can include arguments that make the meaning of the declaration more generic and available for reuse. A formal argument type for a sequence declaration can be

- `const`--An integer constant
- `boolean`--A Boolean [2] expression
- `bit`--A Bit[1] expression
- `bitvector`--A BitVector[1] expression
- `numeric`--A Numeric[1] expression

- `string`--A String[1] expression
- `sequence`--A sequence
- `property`--A property
- `hdltype`--An HDL or user-defined data type

> SystemC PSL supports only `const`, `boolean`, `sequence`, and `property`.

The argument type is followed by one or more argument names. The `hdltype` is followed by the HDL or user-defined data type, then the argument name. Arguments in a list are separated by semicolons. For example, the following sequence declaration defines the name `GenericOp` to mean a sequence in which the `start` and `done` Boolean arguments bracket a series of *n* repetitions of the `op` sequence:

```
sequence GenericOp (boolean start, done; sequence op; const n) =
{start; op[*n]; done};
```

Similarly, the following endpoint declaration defines the name `end_seq` to mean the endpoint of whatever sequence is passed in as argument `S`.

```
endpoint end_seq (sequence S) is { S };
```

> Using formal arguments enables semantic checks.

1 This behavior is a Cadence-specific interpretation of the LRM.
2 See Table 3- 1

PSL Sequence Instantiation

```
Name [( Actual_Argument_List )];
```

You can define sequences and use instances of them as components in complex assertions, as shown in the following examples:

```
sequence RW_DATA_EN = {!RcvDataEn; RcvDataEn[*4]};
rcv_write_RcvDataEn: assert always {triggerRW} |-> {RW_DATA_EN};

  sequence RR_DATA_LD = { !BfrDataLd; BfrDataLd; !BfrDataLd };
rcv_read_BfrDataLd: assert always { read_en } |=> {RR_DATA_LD };
```

The next example shows several sequences and their instantiations.

```
sequence BusGnt(boolean R,G) = {R[+]; G};
```

```
sequence BusRls(boolean R,G) = {!R; !G};
BusGntRls: assert always BusGnt(Rq,Ak) |=> {{BusRls(Rq,Ak)} within [*5]};
```

PSL Sequence Operators

Sequence operators are used to build descriptions of single-cycle or multiple-cycle behavior.

Concatenation

```
{ SERE ; SERE }
```

A sequence is made up of Boolean expressions separated by semicolons (**;**). Each Boolean represents a step in the sequence. The semicolon, which is the concatenation operator, indicates the cycle separation relative to the edge condition. Braces (**{ }**) enclose the entire sequence.

For example, the following assertion

```
first_SERE: assert always {d} |=> { a; b; c } @(posedge clk)
```

looks for

1. `d` equal to 1 on the first positive edge of `clk`. If this is not true, the assertion fails. If it is true, then it looks for
2. `a` equal to 1 on the second positive edge of `clk`. If this is not true, the assertion fails. If it is true, it looks for
3. `b` equal to 1 on the third positive edge of `clk`. If this is not true, the assertion fails. If it is true, it looks for
4. `c` equal to 1 on the fourth positive edge of `clk`. If this is not true, the assertion passes.

Fusion

```
Sequence : Sequence
```

A sequence can involve the fusion operator (**:**), which is similar to the concatenation operator but without a cycle delay. For example:

```
{ { req ; ack } : busy }
```

is a sequence that says that `busy` occurs in the last cycle of the `{ req ; ack }` sequence. Generally, the fusion operator means that the second operand overlaps the first operand by one cycle. If the

two operands are both sequences, the last condition of the left operand and the first condition of the right operand both occur in the same cycle.

For example:

```
{ { req ; ack } : { busy ; done } }
```

is equivalent to

```
{ req ; (ack && busy) ; done }
```

OR

```
Sequence | Sequence
```

You can use the sequence OR operator, **|** , between two sequences, when either the first sequence or the second sequence, or both, must start and not fail. For example:

```
header_i_o: assert always {header_in_seq}  |=>
{{[*]; {header_out_seq}}  | {cancel}}
@(posedge clk);
```

Non-Length-Matching AND

```
Sequence & Sequence
```

You can use the sequence AND operator, **&** , between two sequences, when both sequences must start in the same cycle and not fail. The two sequences might finish in different cycles. For example:

```
sequence data_out = {{header_out_seq} &
{data_out==data_aux1; data_out==data_aux2;
data_out==data_aux3; data_out==data_aux4}}
@(posedge clk);
```

Length-Matching AND

```
Sequence && Sequence
```

You can use the sequence AND operator, **&&** , between two sequences, when both sequences

must start in the same cycle and finish in the same, usually later, cycle. For example:

```
rd_and_wr: cover
{ {rd_seq} && {wr_seq} } @(posedge clk);
```

within

Sequence **within** *Sequence*

You can use the `within` operator between two sequences, when the first sequence must start at or after the cycle in which the second sequence starts, and finish at or before the cycle in which the second sequence finishes. This operator is equivalent to `{[*];Sequence1;[*]} && {Sequence2}`. For example:

```
Ack_After_Req: assert always
{  {ack[*2]} within {req; [*]; grant} } @(posedge clk);
```

PSL Sequence Clocking

Braced_SERE **@** *Clock_Expression*

A sequence can be clocked by adding a clock expression. The clock expression specifies when the Boolean expressions in the sequence are evaluated. For example:

```
{ req ; ack ; busy[*3] : done ; !req ; !ack } @(rose(clk))
```

is a sequence clocked by `rose(clk)`, a PSL built-in function that returns True when its argument (`clk`) has just risen. Each Boolean expression in the sequence will be evaluated in turn at successive rising edges of `clk`. If a sequence has no clock expression--is an unclocked sequence--it inherits the clock expression from the context in which it is contained, if any.

For more information about clocking, see "PSL Clock Expressions".

PSL Repetition

Repetition operators define a sequence in which the operand occurs repeatedly. For example:

```
req[*3]
```

is equivalent to the sequence

```
{ req ; req ; req } .
```

Similarly:

```
{ clk ; !clk }[*3]
```

is equivalent to

```
{ { clk ; !clk } ; { clk ; !clk } ; { clk ; !clk } }
```

which is equivalent to

```
{ clk ; !clk ; clk ; !clk ; clk ; !clk }
```

There are three classes of repetition operators:

- Consecutive repetition operators: `[*]` and `[+]`

  These operators apply to either a Boolean or a sequence. The resulting sequence matches any trace in which the operand occurs for a specified number or range of consecutive cycles. The `[+]` notation is a shortcut for a repetition of one or more times.

- (Possibly) non-consecutive repetition operator `[=]`

  This operator applies to a Boolean. The resulting sequence matches any trace in which the operand occurs for a specified number or range of cycles, which might or might not be consecutive. That is, the trace might contain other cycles in which the Boolean does not occur, either before, in between, or following the cycles in which the Boolean does occur.

- GoTo repetition operator `[->]`

  This operator applies to a Boolean. The resulting sequence matches any trace in which the operand occurs for a specified number or range of cycles, which might or might not be consecutive, provided that the last occurrence is in the last cycle in the trace. That is, the trace might contain other cycles in which the Boolean does not occur, either before, or in between, the cycles in which the Boolean does occur, but not following the cycle of the last occurrence of the Boolean.

Consecutive	(Possibly) Non- Consecutive	Goto
R **[*** *n* **]**	*B* **[=** *n* **]**	*B* **[->** *n* **]**
R **[*** *n* **:** *m* **]**	*B* **[=** *n* **:** *m* **]**	*B* **[->** *n* **:** *m* **]**
R **[*** *n* **:inf]**	*B* **[=** *n* **:inf]**	*B* **[->** *n* **:inf]**

For example:

- `a[*3]` matches the trace `(` **a** `,` **a** `,` **a** `)`
- `a[=3]` matches the traces `(` **a** `,` **a** `,` **a** `),(` **a** `, b, b,` **a** `, b,` **a** `),` and `(b,` **a** `, b, b,` **a** `, b,` **a** `, b, b, b)`

- `a[->3]` matches the traces `( a , a , a )`, `( a , b, b, a , b, a )`, and `(b, a , b, b, a , b, a )`

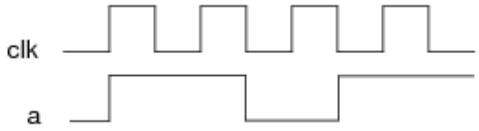
If the repeat count is a range, the resulting sequence matches any trace in which the operand is repeated some number of times in that range. If the upper bound in such a range is `inf`, the upper end of the range is unbounded--that is, the resulting sequence will match any number of repetitions equal to or greater than the lower bound of the range.

> The upper limit for a repeat count is about 64K. Using very large repeat values, such as `ack[*20000000]`, uses a large amount of memory, but unbounded repeat values, such as `ack[*]`, do not.

There are several useful shorthand forms:

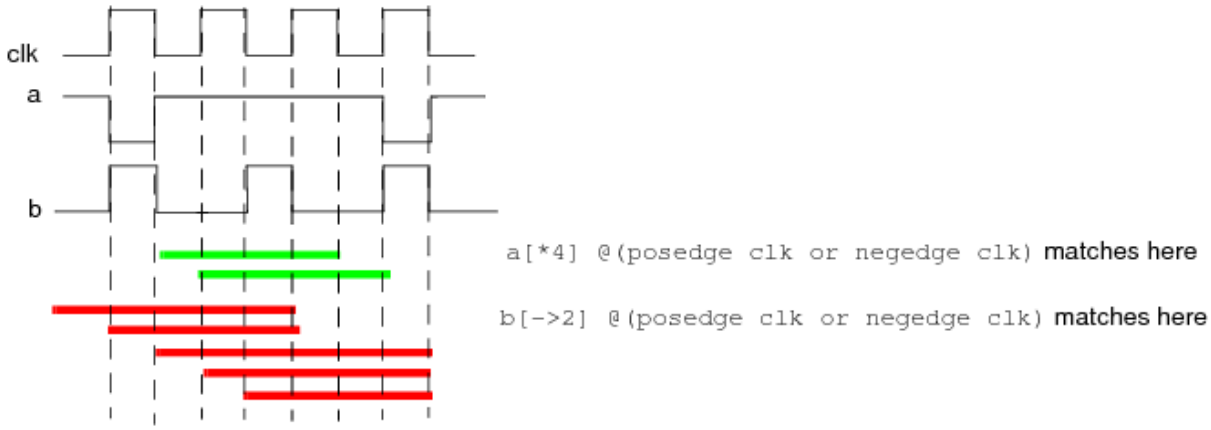
R **[*]**	Equivalent to *R* `[*0:inf]`.
R **[+]**	Equivalent to *R* `[*1:inf]`.
R **[*0]**	Matches an empty trace--a trace of length zero. For example, `{a; b[*0]; c}` matches the trace `(a, c)`, because `b[*0]` matches the *empty*, or zero-length, trace between `a` and `c`.
B **[=0]**	Matches any trace in which B does not occur. That is, `B[=0]` is equivalent to `!B[*]`.
B **[->]**	Equivalent to `B[->1]`.
[*] **[*** *n* **]** **[*** *n* **:** *m* **]** **[+]**	For any consecutive repetition operator, if the operator appears with no operand, the default operand is True, which matches any cycle. So `[*]` matches a trace of any length, including the empty trace; `[*5]` matches any trace of length 5; `[*2:3]` matches any trace of length 2 or 3; and `[+]` matches any trace of length 1 or more.

A repeated sequence might or might not match a trace, depending on how it is clocked. A sequence can be clocked, either by an explicit clock expression or by inheriting a clock context. A clocked sequence is evaluated only when its clock expression is True. For example, given the following traces:



sequence `a[*3] @(posedge clk or negedge clk)` matches twice, but sequence `a[*3]@(rose(clk))` does not match.

A given sequence can match overlapping portions of the same trace. For example, given the following traces, and assuming a clock of `@(posedge clk or negedge clk)`:



The true Value

You can use `true` for don't-care cycles. If you do not care what happens for zero or more cycles between two events, you can use the word `true` to take you forward to the next event.

You can use repetition with `true`. For example, `true[*3]` represents three cycles in which anything can happen, and `true[*]` represents any number of cycles in which anything can happen. For example:

```
A1_slc: assert never {true; (in1->in2); clk; true[*2]; out1};
```

In this example, the sequence that must never be true is:

- One cycle occurs in which anything can happen.
- The `in1` and `in2` signals are high in the next cycle.

- `clk` is high in the next cycle.
- Anything can happen for the next two cycles.
- `out1` is true in the next cycle.

Specifying `[*3]` between semicolons is the same as specifying `true[*3]`. Similarly, `;[*];` is the same as `;true[*];`, and `;[+];` is the same as `;true[+];`.

PSL Properties

A *property* is a temporal relationship among conditions and sequences. For example:

```
always (a)
never {b; c; d}
{req; ack} |=> {read; !req; !ack}
```

PSL Property Declarations

property *Identifier* **[(** *Formal_Argument_List* **)]=|is** *Property* **;**

A property declaration describes the behavior of the design or its environment in terms of temporal operators.

A PSL property declaration ends with a semicolon. For example:

```
property req_evnt_gnt = always
(req -> eventually! gnt) @(posedge clk);
```

For a property to be checked in simulation, you must explicitly specify it in a directive.

You can define PSL sequences, properties, and endpoints in VHDL and SystemVerilog packages. Notes about PSL in packages:

- If a package containing PSL is used by an architecture, any verification unit bound to that architecture can reference the package PSL.
- PSL sequences and endpoints declared in package A can be used in other PSL sequences, endpoints, and properties declared in package B, as long as package B imports package A to make the PSL visible.
- Properties declared in a package can be asserted in an entity or architecture by using a selective use or use all clause.
- You can use both the pragma and native forms of VHDL PSL in packages.
- You cannot define a default clock in a package, because it applies only to directives, not to properties and sequences.
- If you want properties and sequences in a Verilog package to reference properties and sequences in another package that is compiled into a different library, you must import the definitions of all of the referenced properties and sequences from both packages. Similarly, for VHDL packages, you must specify a use clause to access referenced properties and sequences in another package.
- You must use the -assert compiler option to enable PSL pragmas in packages.
- You must use the -v200x option for native PSL in VHDL packages.

You can define a PSL property in a VHDL entity definition, so that it applies to all architectures associated with that entity. For example:

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder_3to8 is
    port (
     clk     : in std_logic;
     input   : in std_logic_vector(2 downto 0);
     output  : out std_logic_vector(0 to 7) );
    begin
     --psl default clock is (falling_edge(clk));
     -- psl p1: cover {output(0); output(1); output(2) };
end decoder_3to8;
```

This feature will not work in a mixed-language model where the VHDL architecture is actually a Verilog model.

PSL Property Arguments

A property can optionally declare a list of formal arguments that can be referenced within the property. Formal arguments for a property declaration can be of the following types:

- `const`--An integer constant
- `boolean`--A Boolean [3] expression
- `bit`--A Bit[1] expression
- `bitvector`--A BitVector[3] expression
- `numeric`--A Numeric[3] expression
- `string`--A String[3] expression
- `sequence`--A sequence
- `property`--A property
- `hdltype`--An HDL or user-defined data type

SystemC PSL supports only `const`, `boolean`, `sequence`, and `property`.

The argument type is followed by one or more argument names, separated by commas. The `hdltype` is followed by the HDL or user-defined data type, then the argument name. Arguments in a list are separated by semicolons.

For example:

```
  property ResultAfterN (boolean start, stop; property result; const n) =  always
((start -> next[n] (result)) @ (posedge clk) abort stop);
```

> Using formal arguments enables semantic checks.

3 See Table 3- 1.

PSL Property Instantiation

An instance of a named property provides actual expressions for the property in place of the formal arguments, if any. For this example

```
  property ResultAfterN (boolean start, stop; property result; const n) =  always
((start -> next[n] (result)) @ (posedge clk) abort stop);
```

an instantiation might look like the following:

```
  assert ResultAfterN(start_time, end_time, result_prop, 7);
```

> The Incisive simulator does not support recursive instantiation of properties. You cannot instantiate a declared property within itself.

PSL Property Operators

The PSL property operators supported in the Incisive simulator are listed in the *PSL Quick Reference* .

> **Note:** With the exception of `eventually!`, `before!`, and `before!_`, the strong (`!`) operators are currently not supported.

The PSL always and never Operators

```
label : assert always condition ;
label : assert never condition ;
```

For most assertions, properties must start with the `always` or `never` operator. These operators declare when the property must be true. They specify invariant conditions that must hold throughout the operation of the design. The invariant condition is defined by the appropriate HDL expression.

> For a description of assertions that do not use the `always` or `never` keywords, see "Writing One-Time Checks in PSL".

The simplest way to describe conditions in a property construct is to use Boolean expressions. For example, the following property states that the `write` and `read` signals must never be true at the same time:

```
mem_control1: assert never ( write='1' and read='1' ) ;
```

Both `always` and `never` specify a fulfilling condition. The expression is repetitively checked at the edge condition. When the edge is not explicitly specified, the signals in the property form the edge condition, similar to a sensitivity list (this is an Incisive simulator interpretation, and might not be portable). The following are examples of invariant properties.

- The `enable_n` line must always be high on the falling edge of `write_n`.

  ```
  WRITE_N_AND_ENABLE_N: assert
    always (enable_n) @(negedge write_n);
  ```

- The `read_n` and `write_n` lines must never be low at the same time.

  ```
  READ_N_AND_WRITE_N: assert
    never (!read_n && !write_n);
  ```

For more information about using the `always` and `never` operators, see "Using the PSL always and never Operators".

The PSL -> Implication Operator

```
left_operand -> right_operand
```

The `->` ("logical if") operator allows if/then checking of an assertion. For example, the following assertion

label **:** assert always *left_operand* **->** *right_operand* **;**

has the following behavior:

- When *left_operand* is true, the *right_operand* will be checked in the same cycle.
- If the *right_operand* is also true, the assertion passes. If the *right_operand* is false, the assertion fails.
- When *left_operand* is false, the assertion is ignored for this cycle.

The left operand must be a Boolean expression. The right operand can be a Boolean, sequence, or property.

The following assertion uses the `->` operator to specify that if `gntA` is true, `gntB` must be false in the same cycle. There is no delay involved.

```
If_GntA_No_GntB: assert always
 (gntA -> !gntB) @(posedge clk);
```

You can combine `->` operators into a complex expression. In the following example, the *right_operand* will be checked only when *in_1* , *in_2* , and *in_3* are true:

```
complex_if: assert always in_1 -> in_2 -> in_3 -> right_operand ;
```

The PSL |-> Suffix Implication Operator

{ *left_operand* **}** **|->** *right_operand*

The `|->` ("suffix implication") operator also allows if/then checking of an assertion. Unlike the `->` operator, which requires a Boolean expression as the left operand, the `|->` operator requires a sequence as the left operand. The `|->` operator specifies that evaluation of the right operand starts in the same cycle in which the left operand sequence finishes.

> As the two operands must overlap by exactly one cycle, they cannot have different clocks--the overlap cycle cannot be spread out over two different clocks.

For example, this assertion

label **:** assert always { *left_operand* } **|->** *right_operand* **;**

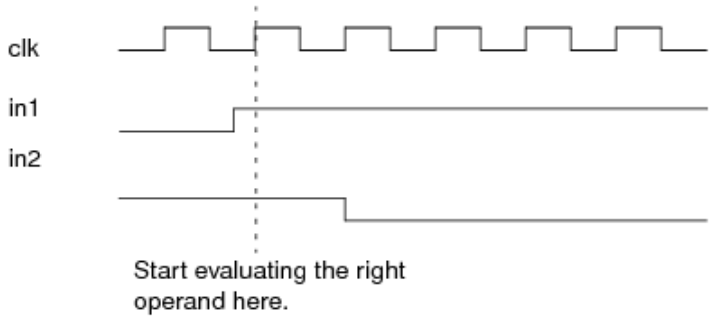
has the following behavior:

- When the sequence that is the *left_operand* occurs, the *right_operand* will be checked in the same cycle in which the sequence finishes.
- If the *right_operand* is true, the assertion passes. If the *right_operand* is false, the assertion fails.

The left operand must be a sequence. The right operand can be a Boolean, sequence, or property.

The following assertion uses the `|->` operator to specify that `in2` is true in the same cycle in which `in1` goes high. There is no delay between the completion of the left operand sequence and checking of the right operand.

```
IF_Rising_IN1_then_IN2: assert always ({!in1; in1} |-> in2);
```



> For information about the performance issues involved in using an open-ended repetition operator in a suffix implication operation, see in "Using PSL Repetition in Suffix-Implication Operations".

You can use the `|->` operator in place of the `->` operator by putting braces around the Boolean expression that is the left operand of the `->` operator. For example:

```
fill_up_fifo: assert always (seq_fill_up -> full) @(posedge clk);
```

can also be written as

```
fill_up_fifo: assert always ({ seq_fill_up } |-> full ) @(posedge clk);
```

The PSL |=> Suffix Next Implication Operator

{ *left_operand* **}** **|=>** *right_operand*

The |=> ("suffix next implication") operator is yet another operator that allows if/then checking of an assertion. Like the |-> operator, it requires a sequence as the left operand. Unlike the |-> operator, the |=> operator specifies that evaluation of the right operand must start in the next cycle after the one in which the left operand sequence finishes.

For example, the following assertion

label **:** assert always { *left_operand* } **|=>** *right_operand* **;**

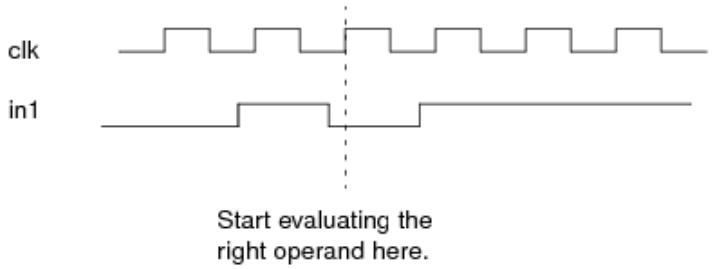
has the following behavior:

- When the sequence that is the *left_operand* occurs, the *right_operand* will be checked in the next cycle after the cycle in which the sequence finishes.
- If the *right_operand* is true, the assertion passes. If the *right_operand* is false, the assertion fails.

The left operand must be a sequence. The right operand can be a Boolean, sequence, or property.

The following assertion uses the |=> operator to specify that a rising edge on in1 is followed by another rising edge on in1. There is a one-cycle delay from the end of the left operand sequence to the beginning of the right operand sequence.

```
Rising_IN1_Repeats: assert always ({!in1; in1} |=> {!in1; in1}) @(posedge clk);
```



In this example, the right operand sequence is the same as the left operand sequence, so each occurrence of a rising edge on in1 that satisfies the right operand sequence also satisfies the left operand sequence. Therefore, the assertion requires an infinite series of rising edges to occur.

For information about the performance issues involved in using an open-ended repetition operator in a suffix implication operation, see in "Using PSL Repetition in Suffix-Implication Operations".

You can use the `|=>` operator in place of the operator pair `|->` `next` without changing the meaning. For example:

```
assert always {Req} |-> next Ack;
```

can also be written as

```
assert always {Req} |=> Ack;
```

The following example uses the `|=>` operator to specify that `READ_PULSE` must start in the first cycle after `m_task` goes to 2'b10. In the following example, if `m_task` = 2'b10, it must always be followed by `read_n` low for two samples, followed by `read_n` high for one sample. This assertion is sampled on the positive edge of `clk`.

```
sequence READ_PULSE = {!read_n[*2]; read_n};
READ_MEM_READ_N: assert
always ( {m_task == 2'b10} |=> {READ_PULSE} )
@(posedge clk);
```

The PSL until Operators

left_operand **until** *right_operand*
left_operand **until_** *right_operand*

The `until` operators specify that the left operand holds until the right operand holds.

For many bus protocols, you expect a specific condition to hold until you receive a response. Once the response is received, the protocol is complete. PSL has two operators to address these kinds of protocols: `until` and `until_`. These operators specify a discharging condition--when this condition occurs, the simulation stops checking the fulfilling condition.

The following example uses the `until` operator to model a bus protocol where `req` must hold until a `gnt` is received. Once the response is received, the protocol is complete, and the assertion that models the protocol ends successfully. For example:

```
hold_req_until_gnt: assert always
(req) -> (req until gnt) @(posedge clk);
```

Use `until` to model situations in which in-flight, multi-cycle behavior continues, but no new

behaviors start. For example, the `until` property is useful for checking that a Boolean condition stays true until an event occurs in your system.

Use `until` rather than `abort` if the behavior is a single-cycle Boolean condition, even though, in this case, the two operators have the same meaning. Reserve `abort` for multi-cycle behavior.

Because `until` and `until_` are weak operators (not followed by an exclamation mark, `!`), they do not specify that the terminating property must eventually hold.

Using the until Operator

The `until` operator specifies that the left operand holds up to, but not necessarily including, the cycle in which the right operand holds.

In the following example, when `in1` is true, `in2` must be true in the next sample, and remain true until `in1` becomes false. This implies that `in2` must stay high if `in1` never becomes false.

```
IN2_UNTIL_NOT_IN1: assert always in1 -> next (in2 until !in1);
```

For more information about the `until` operator, see "Example of until Usage," following, and "Using the PSL until and abort Operators".

Using the until_ Operator

The `until_` operator specifies that the left operand holds up to and *including* the cycle in which the right operand holds.

The following example is similar to the previous one, except that `in2` must be true up to and including the cycle in which `in1` goes false.

```
IN2_UNTIL_NOT_IN1: assert always in1 -> next (in2 until_ !in1);
```

Example of until Usage

When you use the `until` operator, the simulation continually checks the behavior specified by the left-hand operand--everything to the left of the `until` operator. When it encounters the condition specified as the right-hand operand--the discharging condition--it stops checking the left operand (See "Enabling, Fulfilling, and Discharging Clauses").

For example:

```
// psl default clock = (negedge clk);
// psl assert_until: assert always ((!REQ && !GNT) -> next !GNT) until REQ;
```

Using the signal values shown in the following table, on the negative edge of the clock, the first trace for this assertion is

- `active` (A) when the enabling condition begins--`REQ` and `GNT` are 0 in clock tick 1.

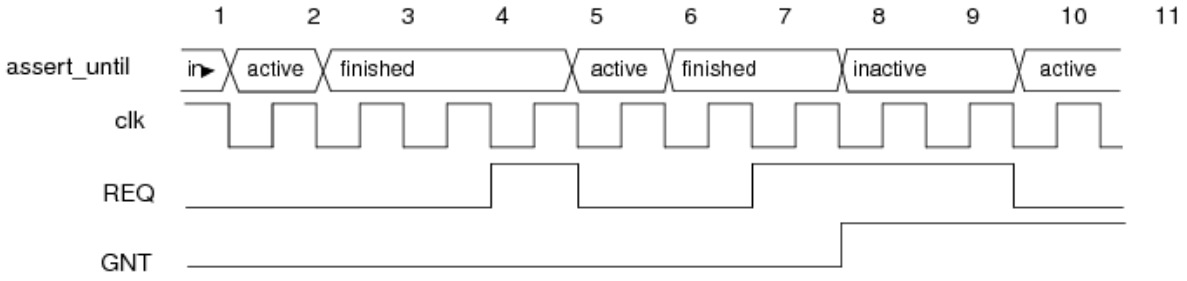
- `finished` (F) when the fulfilling condition is completed--`GNT` is 0 in clock tick 2.
- `inactive` (I) when the discharging condition occurs--`REQ` is 1 in clock tick 4.

clk	0	1	2	3	4	5	6	7	8	9	10
REQ	x	0	0	0	1	0	0	1	1	1	0
GNT	x	0	0	0	0	0	0	0	1	1	1
assert_until	I	A	F	F	F	A	F	F	I	I	A

This assertion creates overlapping traces. The behavior being checked begins in clock tick 1 and finishes successfully in clock tick 2--the behavior occurred as specified. However, the discharging condition has not been encountered, so the simulation continues checking for the behavior. A second trace begins in clock tick 2 and finishes successfully in clock tick 3. The third trace begins in clock tick 3, but the simulation stops checking the behavior when it encounters the discharging condition in clock tick 4. If any other traces were being checked when the discharging condition occurred, the simulation stops checking them as well.

Similarly, the simulation begins checking new traces in clock tick 5, which complete in clock ticks 6 and 7. However, when the enabling condition becomes false, assertion checking becomes inactive. When `REQ` becomes 0 in clock tick 10, assertion checking begins again, and the assertion becomes active.

```
assert_until: assert always (((!REQ && !GNT) -> next !GNT) until REQ) @(negedge clk);
```



For more information about `until`, see "Using the PSL until and abort Operators".

The PSL before Operators

```
left_operand before right_operand
left_operand before! right_operand
left_operand before_ right_operand
left_operand before!_ right_operand
```

The `before` operators specify that one Boolean must hold before a second Boolean holds.

- For the `before/before!` forms, a Boolean is True in some cycle before the cycle in which a second Boolean is True. In the following example, `GntA` must be high before `GntB` goes high:

```
Rnd_Rbn_Gnt: assert always
(ReqA && GntB) -> next (GntA before GntB))
@(posedge clk);
```

- For the `before_/before!_` forms, a Boolean is True in some cycle before or *in the same cycle* in which a second Boolean is True. In the next example, which uses the `before_` form of the operator, `IOReq` must be high either before or in the same cycle in which `IOGrant` goes high:

```
Req_To_Gnt_B_: assert always
(IOReq before_ IOGrant) @(posedge clk);
```

Because `before` and `before_` are weak operators (not followed by an exclamation mark, `!`), they do not specify that the left operand must eventually hold. The `before!` and `before!_` forms specify that the left operand must eventually hold.

In the following example, `REQ` must be high before `GNT` goes high:

```
// psl default clock = (negedge clk);
// psl assert_before: assert always (!REQ && !GNT) -> next (REQ before GNT);
```

On the negative edge of the clock, this assertion is

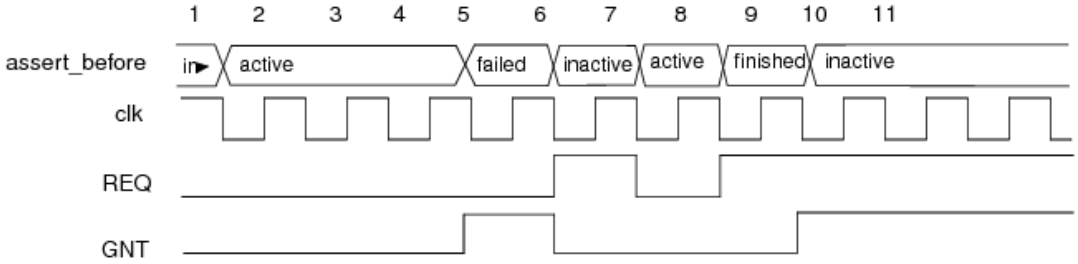
- `active` (A) when the enabling condition begins--`REQ` and `GNT` are 0 in clock tick 1.

  It stays active as long as
  - The enabling condition is true and
  - The fulfilling condition has not occurred
- `failed` (FA) when the fulfilling condition is false--`GNT` is 1 in clock tick 4, but `REQ` remains at 0, so `REQ` is not true before `GNT` is true.
- `inactive` (I) when the enabling condition is not true.
- `finished` (FI) when the fulfilling condition occurs.

clk	0	1	2	3	4	5	6	7	8	9	10
REQ	x	0	0	0	0	1	0	1	1	1	1
GNT	x	0	0	0	1	0	0	0	1	1	1
assert_before	I	A	A	A	FA	I	A	FI	I	I	I

Note: These and other states are described in "Understanding Assertion States," in *Assertion Checking in Simulation*.

```
assert_before: assert always (!REQ && !GNT) -> next (REQ before GNT);
```



The PSL next Operators

```
left_operand next right_operand
left_operand next [ n ] right_operand

next_a [ Range ] ( Property )
next_e [ Range ] ( Property )
```

One of the operators you can use to check a sequence of events is the `next` operator.

If the *`left_operand`* holds in the current cycle, the *`right_operand`* must hold in the next cycle, or the assertion fails.

In this example, if `gnt[0] && !req[0]` holds in a cycle, then `!gnt[0]` must hold in the next cycle, or the assertion fails:

```
no_consec_gnt: assert always ((gnt[0] && !req[0]) -> next !gnt[0])
@(posedge clk);
```

If you use the `next[ n ]` form and the *`left_operand`* holds in the current cycle, the *`right_operand`* must hold in the *`n`* th next cycle, or the assertion fails. The *`n`* value can be 0, which means that the *`right_operand`* must hold in the current cycle. Using `next` *`right_operand`* is the same as using `next [1]` *`right_operand`*.

You can also use extended `next` operators as follows:

- **next_a** -- *Property* is True in every cycle indicated by the *Range* , which must be a finite value (`inf` is not supported). For example, if `RWreq` holds in a cycle, then `RcvDataWrtAck` must hold in all of the next three cycles:

```
ack_1_to_3_after_req: assert always (RWreq ->
   next_a[1:3] (RcvDataWrtAck)) @(posedge clk);
```

- **next_e** -- *Property* is True in some cycle indicated by the *Range* , which must be a finite value (`inf` is not supported). For example, if `RWreq` holds in a cycle, then `RcvDataWrtAck` must hold at least once in the next three cycles:

```
ack_sometime_after_req: assert always (RWreq ->
   next_e[1:3] (RcvDataWrtAck)) @(posedge clk);
```

The PSL eventually! Operator

left_operand **eventually!** *right_operand*

The `eventually!` operator is similar to the `next` operator, but does not specify exactly when the *right_operand* must occur. With the `eventually!` operator, the *right_operand* can occur in the current cycle, or in any later cycle.

The following is an example of specifying temporal nondeterminism, where the `IOReq` signal must be followed by an `IOGrant` at some indeterminate time in the future. The `IOGrant` can happen in the same cycle as the `IOReq`, or in a future cycle.

```
Req_Eventually_Gnt: assert always
(IOReq -> eventually! IOGrant) @(posedge clk);
```

To debug stalled transactions, you can place `eventually!` assertions on all events for which your design waits, and write timeout code to terminate an abnormally long simulation. When the simulator reports unsatisfied eventually! constructs, you can use the simulator report to determine the reason for the timeout.

The PSL abort Operators

Property **sync_abort** *Boolean*
Property **async_abort** *Boolean*
Property **abort** *Boolean*

For some assertions, you might want to terminate assertion checking when a specific event occurs. To do this, use one of the `abo rt` operators. It is not necessary for the enabling condition to be satisfied for the `abort` condition to cancel checking of the assertion.

- For the `sync_abort` operator, the cancellation signal is sampled at the specified clock event; if it is active, checking is terminated.
- For the `abort` and `async_abort` operators, checking is terminated when the cancellation signal becomes active. The `abort` and `async_abort` operators are identical.

  Note: IUS 5.7 and previous releases implemented the `abort` operator as a synchronous abort.

Using the abort Operators

The abort operators--`abort`, `async_abort`, and `sync_abort`--return true if their right operand (the abort condition) becomes true. Abort operators can therefore be used to terminate evaluation of a property, or a portion of a property, when an abort condition occurs, without reporting an error.

Consider the following example:

```
D4_AFTER_A_B:
assert always ({a; b} |=> d[*4]) @(posedge clk);
```

This assertion requires `d` to be true (high) for four consecutive cycles after sequence `{a; b}` occurs. Signals `a`, `b`, and `d` are all sampled at the positive edge of the clock signal, `clk`.

Now consider this variation of the assertion:

```
D4_AFTER_A_B_ABORT_E:
assert always ({a; b} |=> d[*4] abort e) @(posedge clk);
```

In this case, if `e` becomes true at any time, whether at a positive edge of `clk` or not, the subproperty `(d[*4] abort e)` is satisfied, so the implication is satisfied.

The operator precedence rules of PSL imply that this assertion is evaluated as if it was written with parentheses, as follows:

```
assert always ({a; b} |=> (d[*4] abort e)) @(posedge clk);
```

In this case, the `abort` operator applies to the sequence `d[*4]`. So an occurrence of `e` does not terminate the parent property--the implication--or any on-going recognition of `{a;b}`. It does not abort

the entire property; it only aborts the right-hand side of the implication. If, in the same or next cycle, the parent property detects the end of sequence `{a;b}`, the right-hand side of the implication will be checked again.

This assertion can also be written with explicit parentheses, as follows:

```
assert always (({a; b} |=> d[*4]) abort e) @(posedge clk);
```

In this case, an occurrence of `e` aborts the entire implication, so any recognition of `{a;b}` that is in progress when `e` occurs is also terminated. However, because the parent `always` property is not affected by the `abort` condition, the assertion will start checking for `{a;b}` again in the next cycle.

The assertion can also be written as follows:

```
assert ((always {a; b} |=> d[*4]) abort e) @(posedge clk);
```

In this case, an occurrence of `e` will abort the entire `always` property. Because it is the topmost property in the assertion, there is no outer property that will continue to evaluate after the abort, so the abort condition will cause the assertion to stop evaluating for the rest of the verification run.

The `abort` operator is shorthand for the more explicitly-named operator, `async_abort`. Use either `abort` or `async_abort` when the abort condition must have an effect independent of any clock edge. Use the `sync_abort` operator when the abort condition can only have an effect at a clock edge.

For more information about the `abort` operator, see "Using the PSL until and abort Operators".

Applications for the abort Operator

Use the `abort` operator to model situations in which in-flight, multi-cycle behavior is abruptly terminated. For example, the `abort` operator is useful to terminate checking of your properties if you get a reset in your system. Also, you can use it any time that you need a control signal to terminate all incomplete traces of an assertion that were being checked for the behavior specified by the left operand.

The PSL Property Conjunction Operator

```
Property AND_OP Property
```

A property evaluates to true if both *`Property`* expressions evaluate to true. The AND_OP is one of the following:

- Verilog/SystemVerilog--`&&`
- VHDL--`and`

PSL Property Clocking

```
Property @ Clock_Expression
```

A property can be clocked by using the clock operator, `@`.

For example:

```
Req_To_Gnt_B_: assert always (IOReq before_ IOGrant) @clk2;
```

The clock expression must be a Boolean expression.

For more information about clocking, see "PSL Clock Expressions".

PSL Property Replication

```
forall name in value_set : property
forall name in boolean : property
```

The `forall` operator replicates a property a specified number of times.

The *name* is a scalar replicator variable. This variable must be used in the *property* expression.

The *property* expression is replicated once for each value in *value_set* , or twice (true, false) for `boolean` . The *value_set* must be an ascending range, from lower bound to upper bound.

For example:

```
property no_gnt_wo_req =  forall i in {0:ARB_W-1} : never (gnt[i] && !req[i]);
assert no_gnt_wo_req;
```

Notes about the `forall` replicator:

- You can use scalar replicator variables only; array variables are not currently supported. For example, the following is not supported:

  ```
  // NOT SUPPORTED
  // psl property P1 = forall i[1:0] in boolean : ...
  ```

- Transaction recording for replicated properties is not supported.
- The `forall` replicator variable cannot be used in SERE repeat counts.
- For VHDL, when a Boolean replicator is used in a `forall` operation, it is interpreted as a member of the `std.boolean` type. The replicator must be used in expressions as a member of the Boolean type, or type mismatch errors can result. The following example

shows illegal and legal uses of the `forall` replicator in VHDL:

```
library ieee;
use ieee.std_logic_1164.all;
architecture behave of TT is
    signal a, b, c, d : std_logic;
    signal e, f : boolean;
begin
-- ILLEGAL
-- This property uses the equality operator with arguments of two
-- different types, boolean (replicator i) and std_logic (signal a).
-- A type mismatch error will result.
-- psl property illegal_rep_use is forall i in boolean :
-- always {a = i; b} |=> {d};
--
--
-- LEGAL
-- psl property P1 is forall i in boolean :
-- always {e = i; f[*2]} |=> {c};
--
end architecture behave;
```

For more information about using `forall`, see "Using forall, %for, and for/generate".

PSL Directives

Verification directives tell a simulation what to do with the PSL declarations. For simulation, the verification directives are the following:

- `assert`--Expresses required design behavior. The simulator will check that the design conforms to the specified behavior.
- `assume`--Expresses required behavior of the design environment, expressed as a property.

The simulator will check that the environment conforms to the specified behavior.

- `restrict`--Describes required behavior of the design environment, expressed as a sequence. The simulator will check that the environment conforms to the specified behavior.
- `cover`--Expresses possible behavior of the design or the environment, expressed as a sequence. In simulation, the simulator will note when the described behavior occurs.

The PSL assert Directive

```
label : assert property [report " message "
      [severity  note|warning|error|failure]];
```

For the `assert` directive, you can use the optional `severity` and `report` keywords to specify a severity level and error message text to use when the property fails.

For example:

```
assert Clr_Mem_Write_N report "Memory failure"
severity warning ;
```

If you specify both the message and the severity level, the `report` keyword must appear first.

Assertions with a severity of `note` or `warning` do not count as errors with regard to the simulator exit status.

> To prevent any PSL property from being counted as an error relative to the simulator exit status, regardless of severity, use the Tcl `assertion -logging -error off` command.

report

The `report` keyword changes the simulation reporting. Ordinarily, as the simulation runs, assertion results are printed in the SimVision I/O area, as shown in this example:

```
ncsim: *E,ASRTST (./ctr.v,55): (time 120 NS) Assertion top.ctr.memctrl2  has failed
```

If you use `report` to specify an error message, this message replaces the `has failed` portion of the output. For example, if you include `report" RW was high 2 cycles in a row"` with the `RWcycleFail` property, the output prints as follows:

```
ncsim: *E,ASRTST (./ctr.v,55): (time 120 NS) Assertion top.ctr.memctrl2
```

```
RW was high 2 cycles in a row
```

The *message* can be a string literal, Verilog parameter, or VHDL generic or constant. For example:

```
parameter err_msg = "Memory failure" ;
// psl assert Clr_Mem_Write_N report err_msg ;
  generic (AWIDTH : integer := 12; DWIDTH : integer := 32) ;
-- psl assert Addr_Width report "AWIDTH" ;
```

For more information about the content of simulation error messages, see "Understanding Simulator Error Messages " in *Assertion Checking in Simulation*.

> The `report` option can only report when a failure occurs. It cannot report when a sequence occurs, or when a condition has been fulfilled. Use the ended() construct to report sequence matches.

severity

The default severity is `error`. If you change it to `failure`, the simulator treats an assertion violation of that property as a fatal error, and terminates the simulation.

- The severity specification is not supported for cover.
- The severity specification is not a PSL standard.

The PSL assume Directive

```
label : assume property ;
```

In simulation, `assume` directives are treated as if they were `assert` directives. This means that assumptions about the design's primary inputs, if expressed using `assume` directives, will be checked during simulation, and any failure of those assumptions will be reported. This check can help ensure that the testbench is driving the design correctly during simulation.

In formal analysis, `assume` directives are used as constraints during verification of the assertions, because they describe the required behavior of the design environment. For example:

```
assume_no_underflow: assume never (read && empty);
```

The PSL cover Directive

```
label : cover sequence [report " message "];
```

The `cover` directive checks whether the design behavior ever satisfies a sequence. For example:

```
packet_1cyc_overlap: cover
{sopl && !eop[*]; sop && eop}
report "One cycle packet overlap occurred." ;
```

For the `cover` directive, the `report` message is printed only if:

- Logging for assertions in the `finished` state is enabled
- The covered assertion finishes

The PSL restrict Directive

```
label : restrict sequence ;
```

The `restrict` directive provides a way to constrain design inputs to a specified sequence. You can use this directive to initialize the design to a specific state before starting the assertion checks.

The `restrict` directive constrains the environment so that the sequence holds throughout the entire simulation trace, as compared to `assume` , which constrains the environment so that the property holds in a prefix of the simulation trace.

The following example specifies that `test_mode` must be low for the duration of the simulation:

```
no_test_mode: restrict {!test_mode[*]};
```

The simulator interprets this directive as

```
no_test_mode: assert {!test_mode[*]; false};
```

and checks that the sequence holds tightly throughout the entire simulation. If the `restrict` sequence stops holding, the simulator issues an assertion failure message.

No error is issued if the end of the sequence is never reached.

PSL Directive Labels

To provide a user-specified name for a statement, rather than using the software-generated name, you can specify a label for the `assert`, `assume`, `cover`, and `restrict` directives. The label is followed by a colon (:), and precedes the directive.

In the following example, the label is `request` :

```
request: assert always ( bus_request -> eventually! bus_grant )
@(rising_edge( clk )) ;
```

If a directive does not have a label, it is assigned a default name.

Using Labels

You can use labels in embedded PSL, property files, and simulator Tcl commands.

Labels share the design name space. All naming rules of the respective design languages apply to label names.

It is best to give your properties meaningful names or labels, which makes it easier to

- Locate and understand failed properties when you are interactively debugging your simulation
- Identify the purpose of the properties when the design is used later in another context

Combining a Directive with a Declaration by Using Labels

Using labels, you can combine a verification directive and a property declaration into a single statement. This is the simplest way to put assertions into your design. The following illustrates the syntax for combining a property declaration with a directive to create an assertion:

```
assertion_label : assert  interesting event  or sequence of events  ;
```

or, more formally:

```
assertion_label : assert Property ;
```

> The `assertion_label` must be a legal Verilog/SystemVerilog, VHDL, or SystemC name that does not conflict with any other name in the design namespace.

4

Using PSL

Placing PSL Assertions

When you write assertions in PSL, you can embed them as comments in your design code, or include them in a separate file.

General Rules for Embedding PSL Assertions in the Design

To guarantee that assertions travel with the design, you can embed them in the HDL. This technique is particularly recommended for assertions that verify low-level internal structures within an implementation, and are specific to a design.

When you include embedded PSL assertion constructs in your HDL or SystemC design, they must be in pragma form; that is, they must obey the following rules:

- All assertions appear within a consecutive series of comments appropriate for the context, delimited by
  - Verilog/SystemVerilog and SystemC--The `//` or `/* */` characters

    Note: If you use the /* */ characters, the entire assertion must be enclosed within only one set of the characters.
  - VHDL--The `--` characters

    Note: You can also embed PSL in VHDL without having to use the pragma form; see "Using PSL as Native Code in VHDL".
- The first line of the assertion must start with `psl`.

  Because assertions are specified as comments, they will be ignored by tools that do not support PSL.
- Both `psl` and the word that follows it must be on the same line.
- All of the PSL language constructs are case-sensitive.
- Each assertion must end with a semicolon (`;`).
- As your assertions become more complex, you might have difficulty reading the entire construct if it is on a single line. To write a PSL construct that spans multiple lines, write your assertions as follows:

- Start the PSL construct with `psl`.
- Make each line in the construct a comment.
- End the PSL construct with a semicolon (`;`).

  For example, you can write a complex property as follows:

```
// Verilog, SystemVerilog, and SystemC
// psl complex_if: assert always
//  in_1 ->
//      in_2 ->
//          in_3 ->
//              right_operand ;
```

```
-- VHDL
-- psl complex_if: assert always
--  in_1 ->
--      in_2 ->
--          in_3 ->
--              right_operand ;
```

For temporal assertions--assertions that span multiple clock cycles--an error message will display the line of source code that contains the term that failed. If you keep the entire assertion on one line, instead of using multiple lines, the entire assertion will be printed when a failure occurs.

Using PSL as Native Code in VHDL

Native PSL is different from pragma PSL, in the sense that pragma PSL can be written anywhere in the RTL design, whereas native PSL must be written as specified in the VHDL 200X LRM:

- All PSL sequences and properties must be declared in the declarative regions of an entity, architecture, or package.
- All directives must be declared in the body part of an entity or architecture.

You can use the simple subset of PSL as native VHDL code, without having to use the pragma form.

These PSL keywords are treated as VHDL reserved words:

assert	cover	property	sequence	vprop
assume	default	restrict	vmode	vunit

Note: The `endpoint` keyword is supported in PSL pragma form only.

Native PSL is placed in the declarative regions of entities, architectures, packages, block statements and `generate` statements, before the `begin` statement. The assertion statement containing the directive is then placed in the executable statement region. For example:

```
-- Embedded PSL in entity
entity top is
   generic(AB : integer := 1; BC : integer :=2);
   port (clk: inout std_logic := '1';
       in1: inout std_logic := '1';
       in2: inout std_logic := '1';
       a: in bit := '0';
       b: in bit := '0';
       c: in bit := '0';
       out1: inout std_logic := '1');
   -- PSL declarations in entity scope declarative region
   sequence seq0 is {{ clk = '0'}[*AB]};
   property A1 is {in1} |=> {in2};
   -- Default PSL clock
   default clock is falling_edge(clk);
   begin
   -- PSL assertion statements in statement region
   assert A1;
   assume (clk) report "message";
   restrict {in1[*4]};
   cover {in1[*4]};
   cover {a;b;c};
end top;



-- Embedded PSL in architecture
entity top is
    generic(AA:integer:=5;
        clk:inout std_logic:='1';
        in1:inout std_logic:='1';
        in2:inout std_logic:='1';
        AB: integer bit:=1;
        BC: integer:=2;
        BB: positive:=4);
end entity top;

use std.textio.all;
architecture a of top is
    signal wordin : bit_vector(32 downto 1);
    signal wordout :bit_vector(32 downto 1);
    -- PSL declarations in architecture declaration region
    property prop_param(const m; const n) is always
       (wordout(m)->next wordout(n));
    -- Default PSL clock
    default clock is falling_edge(clk);
    begin
    -- PSL assertion statements in architecture statement region
    assert prop_param(AA, BB);;
    assert eventually! {in1; in2};;
    assert never {in2; in1; in2} @(clk);
end architecture;
```

PSL as native VHDL code uses the same general syntax as embedded pragma PSL, except without the `-- psl` pragma construct, and with the normal VHDL restrictions on placement. If a concurrent statement is

ambiguous, and can be interpreted either as a concurrent assertion statement or as a PSL assertion directive, it is interpreted as a concurrent assertion statement.

Notes about using PSL as native VHDL:

- An `assert` *`condition`* statement with no temporal operators is treated as a VHDL concurrent assertion.
- An `assert` *`condition`* statement followed by `report` *`string`* is treated as a VHDL concurrent assertion specifying a condition that must hold at all times.
- Calls to PSL built-in functions from VHDL expressions that are outside a PSL declaration, directive, or verification unit construct are not supported.
- PSL macro preprocessor directives ("PSL Macros") are not supported.
- You cannot mix native and pragma forms. For example, you cannot define a sequence using pragma form and reference it from a property using native form:

```
-- ILLEGAL: NATIVE PROPERTY USES PRAGMA SEQUENCE
--psl sequence s1 is {x; y};
property p1 is always enable -> {s1};
```

- The pragma and native forms use the same namespace. For example:

```
-- ILLEGAL: NATIVE AND PRAGMA USE SAME NAME
--psl sequence s1 is {x; y};
sequence s1 is {a; b; c};
```

- The pragma and native forms can co-exist, both embedded in the VHDL code and in verification units. For example:

```
-- LEGAL
--psl property p1 is always enable -> x;
property p2 is always enable -> y;
```

- You must use the `-v200X` compiler option to enable native PSL.

Putting PSL Assertions in Verification Units

To add assertions to an existing design without modifying the design source code, you can specify them in a separate file, called a *property file*. They are placed within a *veri fication unit* associated with the relevant portion of the design. This technique is particularly recommended for assertions that are added to designs under source code control, because it does not affect the source code itself.

Some additional uses for verification units include

- Adding assertions to an existing design without modifying the source text, as with legacy IP
- Experimenting with assertions before embedding them in the source file
- Adding assertions when you are working in teams where the HDL author does not create the assertions
- Reusing assertions from a previous design
- Selectively including groups of assertions, such as restrictions

Syntax of Assertions in a Property File

```
vunit_type verif_unit_name [ ( design_unit ) ] {
      [ inherit verif_unit_name [ , verif_unit_name ...] ; ]
      [ default clock = clock_edge ; ]
      [ PSL_Declarations ] [ SVA_Declarations ]
      [ PSL_Directives ] [ SVA_Directives ]
      [ HDL_Code ]
}
```

The components of this construct are as follows:

- The *vunit_type* is one of the following:
  - `vprop`--Contains different sets of assertions to verify; cannot contain a directive that is not an `assert` directive, and cannot inherit a `vmode` or `vunit`
  - `vmode`--Contains different sets of constraints; cannot contain an `assert` directive, and cannot inherit a `vprop` or `vunit`
  - `vunit`--Contains assertions, assumptions, and coverage points to verify, and applicable restrictions

SVA that is included in a `vprop` or `vmode` verification unit is treated as HDL code. Therefore, the restrictions on directives in these unit types applies only to PSL, and does not apply to SVA.

- The optional *design_unit* value is the name of the module (Verilog, SystemVerilog, SystemC), instance (Verilog, SystemVerilog), or entity (VHDL) in which to interpret the construct.

- If bound to a module or entity, the assertion applies to every instance of the module or entity.
- If bound to a Verilog or SystemVerilog instance, the HDL names and operators defined in that context can be used within the verification unit. In the following example, the `READ_N_AND_WRITE_N` assertion applies to the `read_n` and `write_n` signals in the `memtest2.mctl.mem8x256.i2` instance:

```
vunit bus_assert(memtest2.mctl.mem8x256.i2) {
    READ_N_AND_WRITE_N: assert
    never (!read_n && !write_n);
}
```

For details about instance binding, see "Using Verification Unit Instance Binding (Verilog only)".

Note: Instance binding is not supported for VHDL and SystemC.

- If a *`design_unit`* is omitted, the verification unit will be *unbound*. This feature lets you group commonly-used PSL declarations so they can be inherited by other PSL verification units.

- The *`verif_unit_name`* is a name of your choice to describe this collection of assertions.

  A verification unit shares the namespace of the module to which it is bound. To avoid conflict with other named objects within *`design_unit_name`* , this name must be unique.The `inherit` option specifies one or more other verification units from which this verification unit inherits the contents.

- The `default clock` construct used in a verification unit is the same as the standard PSL default clock construct. For more information, see "Declaring Default Clocks in PSL".

- Verification units can contain PSL declarations and statements, as well as auxiliary HDL code, including SVA.

Note: For SystemC PSL, you cannot include arbitrary HDL code in a verification unit.

For example:

```
  vunit Interface_Assertions (counter) {
inherit bus_if;
default clock = (posedge clk);
assert_no_gnta_wo_reqa: assert never (GntA && !ReqA);
cover_fair_for_A : cover {GntB;Busy[*];Done && ReqA && ReqB};
}
```

Tips for using property files:

- A single property file can contain both verification units bound to modules and verification units bound to instances.

- Verilog users can include the `` `define ``, `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `endif ``, `` `undef ``, and `` `include `` preprocessor directives in their property files.

Note: The scope of the macros is applicable to that particular property file only.

- VHDL users can include `library` and `use` clauses in a property file, if placed just before the verification unit declaration to which they apply. Each `library/use` clause affects only one verification unit--the one before which it is placed. The `use` clause is also allowed inside a verification unit. For example:

```
-- Use std_logic_1164 from the IEEE library
library ieee;
vunit vu1 (test) {
    use ieee.std_logic_1164.all;
... }
```

- Putting a `default clock` construct in the verification unit can help make the code self-documenting.
- The `protect` pragma is permitted in verification units.
- Synthesis pragmas are not permitted in verification units.
- The pragma form of PSL (// `psl`) can be used in a verification unit the same as other pragmas. However, for clarity, it is recommended that the non-pragma form of PSL be used in a verification unit.
- The contents of any verification unit are visible to other verification units associated with the same module, unless the verification unit is bound to a specific module instance.

Using Verification Unit Instance Binding (Verilog only)

The verification unit instance binding feature lets you group PSL objects--sequences, properties, endpoints--and associate them with specific instances of a module. A wrapper module is automatically created for the verification unit that contains the required ports to communicate between the verification unit content and the target module. This new module is instantiated as an instance of the target module instance, creating a new hierarchy. The new hierarchy offers the advantage of isolating verification unit content from the content of the module, and from other verification units associated with that module.

Some notes about verification unit instance binding:

- You can specify the target instance in two formats: *`module_name . instance_name , or full.hierarchical.path.to.instance`*.

- The verification unit instance name in the Design Browser will be *verif_unit_name*. The hierarchical name to use in Tcl commands is *full_path_to_instance . verif_unit_name*.
- An array of instances is not supported for the *design_unit*.
- You cannot bind to a protected instance.
- A verification unit bound to an instance cannot contain assertions on unpacked structures. Unpacked structures cannot be passed as ports, so they cannot be referenced from within a verification unit when verification unit instance binding is used. Use a packed structure that is defined in a package, or in the associated module.
- You cannot reference a PSL object defined in a module from a verification unit that is bound to an instance of that module. Instead, define the PSL objects in a package that is then referenced by both the module and the verification unit.
- Classes that are defined in a module are not visible to instances within that module. So class objects cannot be referenced in a verification unit that is bound to an instance of that module. To reference a class object from a verification unit, define that class in a package that is globally visible. Class members of the associated module can be accessed, but they cannot be modified.
- Multidimensional nets, or parts of a multidimensional net, that are declared in a module cannot be referenced in a verification unit that is bound to an instance. Use a reg instead of a net.
- If your PSL property file contains a verification unit that binds to an instance, you must specify the property file when you compile the topmost module referenced by that instance. This permits the compiler to locate the instance specified in the binding. For example, if a verification unit in `prop.psl` binds to instance `top.ex1`:

  ```
  ncvlog -propfile prop.psl top.v
  ```

- Support files for instance binding are created in the *worklib* `/.cdssvbind` directory for three-step mode and `./INCA_libs/.cdssvbind` for irun.

Using Verification Unit Properties in generate Constructs

The Incisive simulator inserts the contents of a verification unit at the end of the referenced *design_unit*. For example, the following property will be inserted at the end of the `dut` module:

```
vunit dut_psl(dut) {

  // psl property A2 = always ({a} |-> {b});
```

```
}
```

However, a `generate` scope within the `dut` module references property A2 before it is declared. This reference causes an error, because properties referenced in a `generate` scope must be declared before the `generate`:

```
module dut (clk, counter);
    input clk;


    input counter;
   ...
   generate
       if (1) begin: assert_gen
           wire c;
           wire d;

           assign c = (counter[2] & counter[3]);
           assign d = (counter[3] | counter[4]);

          // psl assert (A2);
       end
   endgenerate

   // The contents of the dut_psl vunit are inserted here.

   endmodule
```

To prevent this error, you can include the `generate` block in the verification unit:

```
vunit dut_psl(dut) {

   // psl property A2 = always ({a} |-> {b});

   generate
       if (1) begin: assert_gen2
       ...
       // psl assert (A2);
       end
   endgenerate

   }
```

SystemC PSL in Verification Units

For SystemC, only default clock definitions, comments, and verification directives can be included in this file. Comments are allowed. The following are ***not*** supported for SystemC PSL in property files:

- Arbitrary SystemC code

- Unbound and instance-bound verification units

Interpretation of PSL Constructs in a Property File

PSL constructs in a property file are interpreted as if the text of the constructs was inserted into the text of the module, immediately before the end of the module. An expression contained in an external PSL construct must be interpretable at that point as a valid expression.

A PSL construct in a property file can refer to

- Objects in the design
- Names defined by embedded PSL constructs in the design
- Names defined by other external PSL constructs that are effectively embedded in the design, subject to the scope and visibility rules of the context in which it is effectively embedded

Modeling Layer Support of HDL Code in PSL Verification Units

Cadence supports the PSL modeling layer, so you can use HDL code within a verification unit. You can add auxiliary variables and monitoring logic to the PSL without modifying the HDL design.

In addition to regular HDL, PSL also defines PSL also defines macros for use in verification units. These macros provide support for conditionally instantiating or replicating code. For details, see "PSL Macros".

For VHDL, any construct that you can write in the architecture to which the verification unit is bound, except for `library` clauses, is allowed in the verification unit. The `library` clause can be placed directly before the verification unit declaration to apply to the verification unit.

Because the PSL modeling layer is supported in verification units, it is possible to instantiate monitors (IAL and OVL, for example) in a verification unit (see "Including IAL Assertions in Verification Units").

Note: This feature is not supported for SystemC PSL.

PSL Macros

You can use PSL macro processing to define properties in a Verilog verification unit. For the Cadence implementation, the Verilog flavor supports the PSL `%for` and `%if` macros, which you can use to conditionally or iteratively generate PSL statements.

You do not need to use the comment form (// `psl`) for these macros. When you include the `-assert` or `-propfile` compiler option, the macros are recognized as PSL.

You can use any valid Verilog or SystemVerilog code within the macros.

Although Cadence has implemented extensions to support PSL declarations and directives embedded in the design, it has not implemented extensions to support the `%for` and `%if` macros. These macros can be used only in verification units.

The Incisive simulator first processes Verilog compiler directives--`` `define ``, `` `ifdef ``, `` `else ``, `` `include ``, and `` `undef ``--then `%if` and `%for`.

An example of using these macros together is the following:

```
%for index in 0..3 do
    %if %{index} == 0 %then
        // psl assert_%{index}: assert always fifo[%{index}] !== 8`bx;
    %else
        // psl assert_%{index+1}: assert always fifo[%{index+1}] !== 8`bx;
    %end
%end
```

This example produces the following results:

```
// psl assert_0: assert always fifo[0] !== 8`bx;
// psl assert_1: assert always fifo[1] !== 8`bx;
// psl assert_2: assert always fifo[2] !== 8`bx;
// psl assert_3: assert always fifo[3] !== 8`bx;
```

For more information about these macros, see "The %for Construct (Verilog)" and "The %if Construct (Verilog)".

The %for Construct (Verilog)

```
%for var in expr1 .. expr2 do      code
%end

%for var in { item , item , ... , item } do      code
%end
```

The `%for` construct replicates *code* a specified number of times. Each replication is made unique by parameter substitution.

- *var* --The replicator variable name; any legal PSL identifier name. It must be a unique name, with the exception that it can be the same as another non-enclosing PSL replicator variable. The value of the replicator variable must be statically computable.
- *expr1* , *expr2* --Replication expressions; statically computed expressions that result in a legal PSL range.

- *code* --Can be any code, not just assertion code.
- *item* --Replication item; any legal PSL alphanumeric string or previously defined preprocessor-style macro. Items are separated by commas, and must be enclosed in curly braces.

For the first form of the construct, the code between `%for` and `%end` will be replicated *expr2* `-` *expr1* `+1` times, assuming that *expr2* is greater than or equal to *expr1* .

For the second form of the construct, the code will be replicated according to the number of items in the list. During each replication of the code, the loop variable value is substituted into the code. For example, for a loop variable called `i`, you can access the current value of the loop variable from the loop body using one of these methods:

- If `i` is a separate token in the code, access its value by using `i`:

  ```
  %for i in 0 .. 3 do
       define aa(i) := i > 2;
   %end
  ```

  This code is processed as follows:

  ```
  define aa(0) := 0 > 2;
  define aa(1) := 1 > 2;
  define aa(2) := 2 > 2;
  define aa(3) := 3 > 2;
  ```

- If `i` is part of an identifier, access its value by using `%{i}`:

  ```
  %for i in 0 .. 3 do
       define aa%{i} := i > 2;
   %end
  ```

  This code is processed as follows:

  ```
  define aa0 := 0 > 2;
  define aa1 := 1 > 2;
  define aa2 := 2 > 2;
  define aa3 := 3 > 2;
  ```

- If `i` must be used as part of an expression, access its value by using `%{i-1}`:

```
%for i in 1 .. 4 do
    define aa%{i-1} := %{i-1} > 2;
%end
```

This code is processed as follows:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

You can use the following operators in preprocessor expressions:

=	*	-
<	!=	/
<=	>	%
+	>=	

For more information about using the `%for` construct, see "Using forall, %for, and for/generate".

The %if Construct (Verilog)

```
%if expr %then
code
%end

%if expr %then
   code
%else     code
%end
```

The `%if` construct generates the enclosed code if the condition specified in the *expr* is true.

This macro is similar to the `#if` preprocessor directive, except that it can be made conditional on variables defined in an enclosing `%for` construct.

The `%if` condition must be statically computable.

Including IAL Assertions in Verification Units

Because Incisive Assertion Library (IAL) components are Verilog HDL models with PSL assertions embedded within them, you can include them in a verification unit that is associated with a Verilog module.

The following example includes the IAL `ial_mclk_mport_fifo` component in a verification unit bound to the `fifo_16x64` module.

```
vunit fifo16x64Test (fifo_16x64)
{
ial_mclk_mport_fifo # (64, 54, 10, 1) ial_fifo16x64_inst (rd_clk, read_rstn,
wr_clk, write_rstn, write, enb_net);
}
```

> Many IAL components also contain coverage points.

Another method of including IAL in a verification unit is to place the IAL components within a module in a monitor file. For example:

```
module Qcheck (.. );
    input qClk, qInsert,qRemove;
    input qFull, qEmpty;
    input reset;
...
ial_never NoSimAccess (
    qClk, reset, (qInsert && qRemove), enable);

endmodule
```

You can then instantiate the monitor module in the verification unit:

```
vunit Qtest (Queue)
{
     // Instantiate IAL monitor
     Qcheck QChk(qClk, reset, qInsert, qRemove, qFull, qEmpty);
}
```

> For more information about using IAL components, see "The IAL Use Model" in "Assertion-Based Verification Using the IAL ," in the *Incisive Assertion Library Reference*.

Putting SVA Assertions in a PSL Verilog Verification Unit

To add assertions to an existing design without modifying the design source code, you can specify them in a separate file, and associate them with the relevant portion of the design. To do this, there are several techniques available:

- A PSL verification unit--You can bind SVA in a PSL verification unit to the design.
- An SVA `bind` directive--You can bind PSL to the design by referencing it in an SVA `bind` directive.

For details about putting SVA in a PSL verification unit, and for a comparison of the advantages and disadvantages of `bind` and verification unit techniques, see "Using SVA in a PSL Verilog Verification Unit".

Using PSL Sequences

The PSL language lets you declare a property of the form

```
assert always { sequence1 };
```

where *sequence1* extends over more than one clock cycle. However, this syntax probably does not describe a useful property of a design.

Many people, when they write a property of this form, assume that PSL will interpret it as an infinite number of copies of the sequence concatenated together. They assume that the tool will look for { *sequence1* ; *sequence1* ; *sequence1* ; }.

The PSL language says that `always { sequence1 }` implies that the sequence has to hold at every clock in the simulation. However, instead of concatenating copies of the sequence to be checked one after another, the simulator creates a new copy of the sequence at every clock and begins a new check, which produces *overlapping assertion traces*.

For example, you might want a property to ensure that the clock has a regular waveform. You might try using the following assertion:

```
assert always { clk; !clk };
```

PSL does not concatenate this sequence into an infinite series of { `clk; !clk; clk; !clk;` }. Instead, it starts a new copy of the sequence with each cycle in the design:

```
 Cycle # 1     2     3      4

Copy 1  clk  !clk

Copy 2        clk   !clk
```

```
Copy 3              clk   !clk
```

In cycle 1 of simulation or formal analysis, if the clock is high, the property passes. But in cycle 2, the first copy of the sequence says that the clock must be low, while the second copy of the sequence says that the clock must be high. No matter what the actual state of the clock, one of those properties is guaranteed to fail.

Using HDL Functions in PSL

To simplify a complex property, you can invoke a Boolean HDL function anywhere the PSL syntax allows a Boolean. For example:

```
function undriven(vec:std_logic_vector)
return boolean is
    begin
        for i in vec`range loop
            if vec(i) = 'Z' then
                return true;
            end if;
        end loop;
        return false;
    end;

A1: assert never undriven(data);
```

Using SystemC PSL

For detailed information about features that are unique to SystemC PSL, see " Using SystemC PSL ," of the *SystemC Simulation Reference*.

Referring to Signals in SystemC PSL

The Boolean layer for SystemC consists of syntactically valid C++ expressions that are legal in the context of the enclosing SystemC module. These expressions must evaluate to a value that can be interpreted as a Boolean value (see section 5.1.2, "Boolean expressions," in the *1850 IEEE Standard for Property Specification Language*).

A SystemC signal can be referred to directly in a PSL assertion. The PSL parser automatically recognizes references to ports and signals, and implicitly adds a call to the `.read()` method where needed. For example, a signal is declared as follows:

```
sc_out<sc_lv<4> > countVal;
```

and it is referred to in the PSL code as

```
invert1: assert always (countVal[0] -> next !countVal[0]);
```

Its value is converted to SystemC code as `countVal.read()[0]`.

Note: This simple assertion is portable to Verilog/SystemVerilog and VHDL. Assertions with expressions within the SystemC literal inclusion function, `_( SystemC/ C++_expressions )`, are not portable. See "SystemC Expressions in PSL".

Accessing Assertions from a SystemC Testbench

The Cadence Incisive Enterprise Simulator-XL provides an API that gives you uniform access to assertions in Verilog (PSL), VHDL (PSL), SystemC (PSL), and SystemVerilog (PSL, SVA) from a SystemC testbench. This capability lets you create reactive tests that can access assertion information.

For details, see "Access to Assertions from SystemC Testbench" in " Using SystemC PSL ," of the *SystemC Simulation Reference*.

Using the PSL always and never Operators

PSL allows you to create a property that contains

- Both `always` and `never`
- Neither `always` nor `never`

Using always and never in the Same PSL Property

According to the PSL grammar, you can create a property of the form

```
bad: assert always a -> next ( never b );
```

Although syntactically correct, this approach is not recommended. With this construct, the simulator will create nested loops for checking the assertion: one loop for the `always` construct and, for each cycle in which `a` is true, a nested loop for the never construct to check that `b` is not true. Because this property creates multiple parallel traces, it can significantly affect simulation performance.

To correct the problem in the previous example, use the following property instead:

```
good: assert always a -> next (( never b ) until_ a);
```

This property creates loops like the previous version, but the nested loop for this version checks that `b` is not true only until `a` is true again. This reduces the large number of simultaneous traces.

Writing One-Time Checks in PSL

According to the PSL grammar, you can create properties without the `always` or `never` directives. For example, you can create the following property:

```
not_a_and_b: assert !( a && b ) ;
```

This property will check that signals `a` and `b` are not both high. However, this property will be evaluated *only during the initial simulation cycle*. The property will not be checked during the rest of simulation. However, it might be useful for checking initial conditions in a simulation.

> Assertion libraries often use these one-time checks to validate the parameters that must be defined when the module is instantiated.

Similarly, the grammar description shows that properties can be started with the `next` or `eventually!` keywords. For example:

```
not_a_and_b: assert next !( a && b ) ;
```

This property will check that signals `a` and `b` are not both high. This property will be evaluated *only on the first simulation cycle after initialization*. The property will not be checked during the rest of simulation. However, it might be useful for checking conditions right after initialization.

This property will check that signals `a` and `b` are not both high:

```
not_a_and_b: assert eventually! !( a && b ) ;
```

This property will be evaluated during the entire simulation.

- If the condition ever occurs, the assertion will finish with no error messages. It will not be checked during the rest of simulation.
- If the condition does not occur, an assertion error will be recorded at the end of simulation.

You might use this property to watch for an event that must happen at least once during simulation. However, it will not check for any events after the first event.

Using PSL abort with never

From a language definition point of view, `abort` can be used with `never`, but the particular assertion might have a different meaning from what you expect. For example, the following assertion produces an `ILLFRM Illegal context for a PSL formula` error:

```
assert never bar abort(rst);
```

Because of PSL operator precedence, the parser interprets the assertion as if it was written with the following parentheses:

```
assert never ( bar abort(rst) );
```

The term `bar abort(rst)` is a PSL property. PSL allows only a Boolean or a sequence as the operand of the `never` operator, not a property. Adding parentheses still does not result in the desired behavior:

```
assert ( never bar ) abort(rst);
```

When `rst` is true, the `never bar` term is aborted, which cancels all future checking of the `bar` signal. When `rst` is deasserted, checking of the `never bar` term is not resumed.

One way to encode an abort term into a never property is to make the term part of the never property:

```
assert never (bar && !rst);
```

Using the PSL until and abort Operators

The `until` and `abort` operators are similar, but different. They both allow the property that is the left operand to stop holding after the Boolean right operand becomes true, but they differ with respect to *when* the left operand is allowed to stop holding.

Note: You can use SEREs on the left-hand side of `until`, `until_`, and `abort` statements. For example:

```
-- psl COMPLEX_SERE: assert
--  always { state_bits = State1 ; state_bits = State2 }
--      |=> { state_bits = State3 ; state_bits = State4 }
--      abort reset
--  @(rising_edge( clk )) ;
```

Example of PSL until and abort Operators

If `f` is a Boolean condition, the following assertions have the same meaning:

```
f until b
f abort b
```

However, there is a difference if the left operand is a multi-cycle property. For example:

property **f** = ({a} |=> {b;c;d})
property **f_until_e** = (f until e)
property **f_abort_e** = (f abort e)

The table that follows shows the behavior on signals `a`, `b`, `c`, `d`, and `e`.

For these three properties

- Y indicates that the assertion trace beginning at this cycle will complete

- N indicates that the assertion trace will fail at some future time with an error message
- . indicates that the property is irrelevant in that cycle

Note: The `f` property, as written, will start a new trace of the assertion at every cycle where `a` is true.

Cycle:	1	2	3	4	5	6	7
a	**1**	1	**1**	1	**0**	0	**0**
b	-	**1**	1	**1**	1	**0**	0
c	-	-	**1**	1	**1**	1	**0**
d	-	-	-	**1**	1	**1**	1
e	0	0	0	0	0	0	0
f	Y	Y	Y	Y	.	.	.
f_until_e	Y	Y	Y	Y	.	.	.
f_abort_e	Y	Y	Y	Y	.	.	.

For these values, `f` holds in cycles 1, 2, 3, and 4 because, in each of these cycles, the `{a}` condition occurs, followed in the next three cycles by `{b;c;d}`. Because f holds in those cycles, `f_until_e` and `f_abort_e` also hold in those cycles.

In the following case, `f` holds in cycles 1 and 2 as before, but it does not hold in cycles 3 and 4, because the {`a;b;c;d`} sequence does not complete in cycles 6 and 7. Again, `f_until_e` and `f_abort_e` hold in the same cycles in which `f` holds.

Cycle:	1	2	3	4	5	6	7
a	**1**	1	**1**	1	**0**	0	**0**
b	-	**1**	1	**1**	1	**0**	0
c	-	-	**1**	1	**1**	0	**0**
d	-	-	-	**1**	1	**0**	0
e	0	0	0	0	0	0	0
f	Y	Y	N	N	.	.	.
f_until_e	Y	Y	N	N	.	.	.

f_abort_e	Y	Y	N	N	.	.	.

The following case shows the difference between `abort` and `until` when `e` goes high:

Cycle:	1	2	3	4	5	6	7
a	**1**	1	**1**	1	**0**	0	**0**
b	-	**1**	1	**1**	1	**0**	0
c	-	-	**1**	1	**1**	0	**0**
d	-	-	-	**1**	1	**0**	0
e	0	0	0	0	**1**	0	0
f	Y	Y	N	N	.	.	.
f_until_e	Y	Y	N	N	Y	Y	Y
f_abort_e	Y	Y	Y	Y	Y	Y	Y

Here, signal `e` goes high in cycle 5. This removes the obligation for `f` to hold after cycle 5. However, the requirements for `f_until_e` and `f_abort_e` are different:

- For `f_until_e`, the occurrence of `e` removes the obligation for any *new occurrences* of `{a}` to be followed by `{b;c;d}`. The `{a}`s that have already occurred must still be followed by `{b;c;d}`. That is, it obligates any "in-flight" behavior to complete.
- For `f_abort_e`, the occurrence of `e` removes the obligation for all "in-flight" behavior to complete, so the traces that started in cycles 3 and 4 are not required to complete in cycles 6 and 7. Therefore, `f_abort_e` holds in cycles 3 and 4 as well.

Analogy for PSL abort and until Operators

A restaurant is open until midnight, but its kitchen closes at 10 pm. An assertion that describes the behavior of diners in this restaurant is

```
({arrive,seated,order} |=> {salad,dinner,dessert}) until kitchen-closes;
```

That is, as long as they have arrived before the kitchen closes, the diners will complete their meals.

In contrast, to describe the behavior of diners in the event that the restaurant catches fire, the assertion is

```
({arrive,seated,order} |=> {salad,dinner,dessert}) abort fire;
```

That is, regardless of what stage they are in, diners will abandon their meals if a fire breaks out.

These conditions can be combined, as follows:

```
property normal_dinner = {arrive,seated,order} |=> {salad,dinner,dessert};

property actual_dinner = ((normal_dinner) until kitchen-closes) abort fire);
```

Using `forall`, `%for`, and `for/generate`

The `forall`, `%for`, and `for/generate` constructs are similar. The following are the advantages and disadvantages of each:

- `forall` generates one assertion for all values. For example:

  ```
  // psl check_init: assert
  //      forall i in {0,1,2} : always (!fifo[i]) @(rose(rstn));
  ```

This example creates the following in the Incisive simulator:

```
// psl check_init: assert always (!fifo[0] && !fifo[1] && !fifo[2])
   @(rose(rstn));
```

The advantage of using forall is that there is only one assertion to track and monitor. The disadvantage is that it is more difficult to debug, because more analysis is needed to understand which case failed.

Note: In IFV, `forall` acts like `for/generate`, except that a new scope is not created.

- `for/generate` generates one assertion for each value; for example:

  ```
  for i in {0,1,2} generate
      // psl check_init: assert always (!fifo[i]) @(rose(rstn));
  endgenerate;
  ```

This example creates the following:

```
// psl check_init: assert always (!fifo[0]) @(rose(rstn));
// psl check_init: assert always (!fifo[1]) @(rose(rstn));
// psl check_init: assert always (!fifo[2]) @(rose(rstn));
```

The `for/generate` construct creates a new scope for each of these assertions. For this example, the assertion name, `check_init`, becomes part of the hierarchical path that you specify in Tcl commands. The default clock from the parent scope applies to this scope.

The advantage of using `for/generate` is that all of the assertions can have the same name. The disadvantage is that coverage reports, which list results by scope, can be lengthy.

- `%for` generates one assertion for each value; for example:

```
%for i in 0..2 do
    // psl check_init%{i}: assert always (!fifo[%{i}]) @(rose(rstn));
%end
```

This example creates the following:

```
// psl check_init0: assert always (!fifo[0]) @(rose(rstn));
// psl check_init1: assert always (!fifo[1]) @(rose(rstn));
// psl check_init2: assert always (!fifo[2]) @(rose(rstn));
```

The advantage of using `%for` is that it can be used to distinguish the assertion names while keeping them all in one scope, because the index variable is a preprocessor step. The disadvantage is that the `%for` construct can be used only in a verification unit.

The following table compares `forall`, `for/generate`, and `%for`.

Function	Location	Preprocessor step	Number of assertions generated	Creates new scope
`%for`	Verification unit	Yes	Many	No
`forall`	Anywhere	No	One	No
`for/generate`	Anywhere	No	Many	Yes

Using the assume and restrict Directives

The Incisive simulator implements the `restrict` and `assume` directives similar to the `assert` directive, because there is no significant distinction between a constraint and an assertion to be checked in dynamic simulation.

An assume is treated exactly like an assert, although the name of the directive has different implications on where the root of the problem is when a failure occurs--`assert` indicates that the outputs have a problem, whereas `assume` indicates that the stimulus is wrong.

The `restrict` directive is treated differently when the rightmost sequence is open-ended. A single `restrict` can--if it is open-ended--cover the whole simulation. For example:

```
assume {!rst; rst[*3]; !rst[*]};
```

is satisfied and finishes after four cycles, because the trailing `!rst[*]` has no effect, as for the right-hand

side of any property. The corresponding restrict:

```
restrict {!rst; rst[*3]; !rst[*]};
```

imposes a constraint on the entire simulation that `rst` must stay low.

Using PSL Repetition in Suffix-Implication Operations

There is a fundamental difference between the left and right sides of a suffix-implication operation. These sides are sometimes described as "all match" and "first match," respectively.

In English, the suffix implication might read "for all matches of the left operand, there must be a match of the right operand." If you use an open-ended repeat operator at the end of the left operand, the property is tightly constrained, because it must satisfy all cases.

When you shift the open-ended repetition to the right operand, where the meaning is "there must be," the constraint described by the property is loosened, by allowing the final fulfillment of the right operand sequence to be delayed.

For example:

```
// psl Y_after_X_before_X_or_Z_3: assert always
//     ( {boolX; (!boolX && !boolZ)[*0:2]} |=> seqY ) abort rst;
```

This example is checked as follows, which results in many consecutive failures:

```
assert always ( {boolX; (!boolX && !boolZ)[0]} |=> seqY ) abort rst;
assert always ( {boolX; (!boolX && !boolZ)[1]} |=> seqY ) abort rst;
assert always ( {boolX; (!boolX && !boolZ)[2]} |=> seqY ) abort rst;
```

As shown in this example, many overlapping assertions have resulted. A large number of overlaps can result in performance degradation.

Sampling Signals in Assertions

This section describes how to specify clocking for your assertions to avoid simulation problems.

Preventing Assertion Failures at Time 0

If you used `posedge clk` as the sampling clock for your assertions, and the clock starts high, the assertions will be tested at time 0. This is because an X-to-1 transition is considered a positive edge. Similarly, an X-to-0 transition is a negative edge.

To check assertions only when a 0-to-1 transition occurs, you can use `rose(clk)` as your clock. Similarly, you can use `fell(clk)` for the 1-to-0 transition. Using `rose(clk)` or `fell(clk)` will eliminate time 0

failures.

> Synthesis pragmas use the PSL default clock. If you define the default clock to be `rose(clk)`, assertions created from synthesis pragmas will not fire at time 0.

Avoiding BOOLOP Messages for PSL Assertions

To avoid error messages due to operator precedence, you might need to parenthesize expressions to achieve the desired behavior. For example, in earlier versions of PSL, the property

```
// psl P1: assert always (in1 -> next in2 -> next !in1);
```

had the following meaning, because `next` had a lower precedence relative to `->`.

```
// psl P1: assert always ((in1 -> next (in2 -> next !in1)));
```

However, with the introduction of PSL1.1, `next` has a higher precedence relative to `->`:

```
// psl P1: assert always (in1 -> (next in2) -> (next !in1));
```

Because the left-hand-side of the `->` operator must be Boolean, the new precedence rules cause a fatal `BOOLOP` error message when the parentheses are missing.

For more information about PSL operator precedence, see Operators precedence table.

Using PSL Reactive Test Techniques

For general information about reactive test architecture and tradeoffs, see Chapter 9, "Writing Reactive Tests using Assertions."

The PSL construct that can be used for reactive tests is the following built-in function:

```
ended( sequence_spec [, clock_spec ])
```

For more information, see the description of the `ended()` construct in this manual, and section 5.2.3.6 in the PSL IEEE 1850 standard.

Using the PSL ended() Construct

You can use the `ended()` function anywhere a signal can be used--for example, in a conditional statement, or as an event control. When used as an event control, the `ended()` function behaves as if it returns an event--it triggers the block on each edge.

The Cadence implementation of the `ended()` function allows this construct anywhere in the HDL, rather than only in the modeling layer.

The following is an example of using the `ended()` function:

```
vunit test_triggers(top.TX_Monitor) {
       reg gframe = 0;
       enum (DA, SA, Length, Payload, CRC, Err) state;

       sequence good_frame_transmitted = {
           state==DA[*12]; state==SA[*12]; state==Length[*2];
           state==Payload[*min:max]; state==CRC[*8]; state!=Err};

       always @(ended(good_frame_transmitted, rose(clk)))
           gframe = ~gframe;
}
```

For more information about using the `ended()` function, see "Using PSL ended() in HDL".

PSL Event-Based Reactive Test Example

Reactive tests can respond to legal or illegal activity or inactivity. A generic example is a test that reacts to a successfully-transmitted frame by adding another frame to the queue. It can also add the frame to the receive queue of the appropriate response checker.

The reactive test constructs are bound to a monitor that has an HDL-based state machine and many associated compliance and coverage assertions.

A sequence is defined that detects when a good frame has been transmitted. When the sequence is detected--for example, the `ended( sequence )` construct can be used to detect that the sequence occurred--an event is triggered.

The sequence and detection code is placed in a PSL verification unit. The verification unit is bound to an instance of a monitor component. The verification unit binding essentially instantiates the verification unit into the monitor instance.

HDL can then be used to change the state of a signal. The test is defined to react to changes on that signal using the full hierarchical path.

If the monitor is written in VHDL instead of Verilog, you use the VHDL flavor of PSL instead of the Verilog flavor. In addition, because events are not supported in VHDL, you must set a signal that can be accessed by the test.

SystemC Reactive Test with PSL

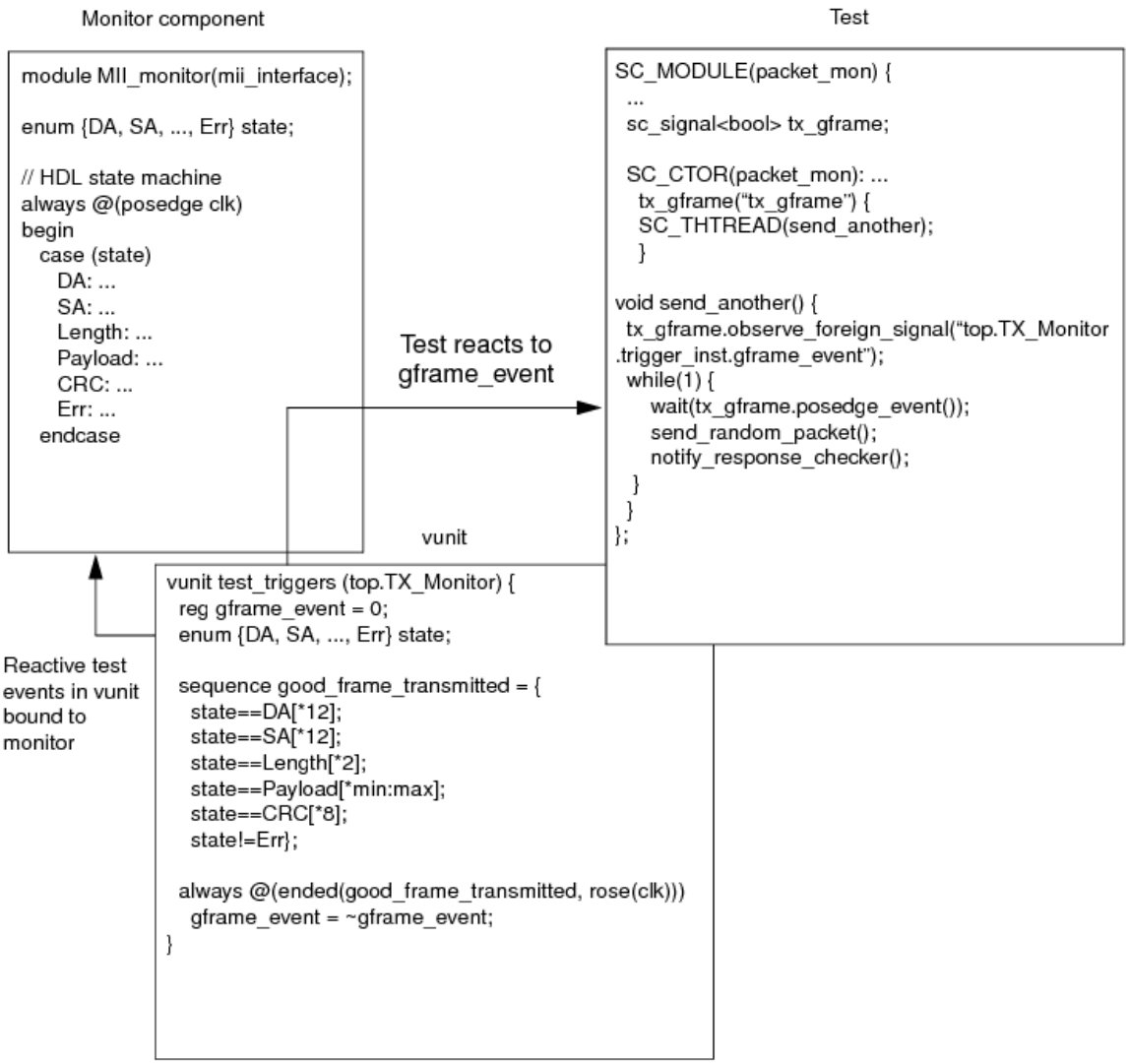
Example 4-1 shows how the test example can be implemented for a SystemC test with PSL assertions.

The `test_triggers vunit` defines the `good_frame_transmitted` sequence and uses the `ended( sequence )` construct to detect that the sequence occurred. HDL is then used to change a signal state. When the state change occurs, the `gframe_event` signal is toggled.

The test is defined to react to changes on that signal using the full hierarchical path. When a change occurs, the code in the `send_another()` method is executed, which adds another randomized packet to the processing and response checker queues.

> The Cadence Incisive Enterprise Simulator-XL also provides SystemC classes that permit access to PSL and SVA assertions for reactive tests. For details, see "Access to Assertions from SystemC Testbench" in " Using SystemC PSL ," of the *SystemC Simulation Reference*.

Example 4-1 SystemC Test with PSL Assertions

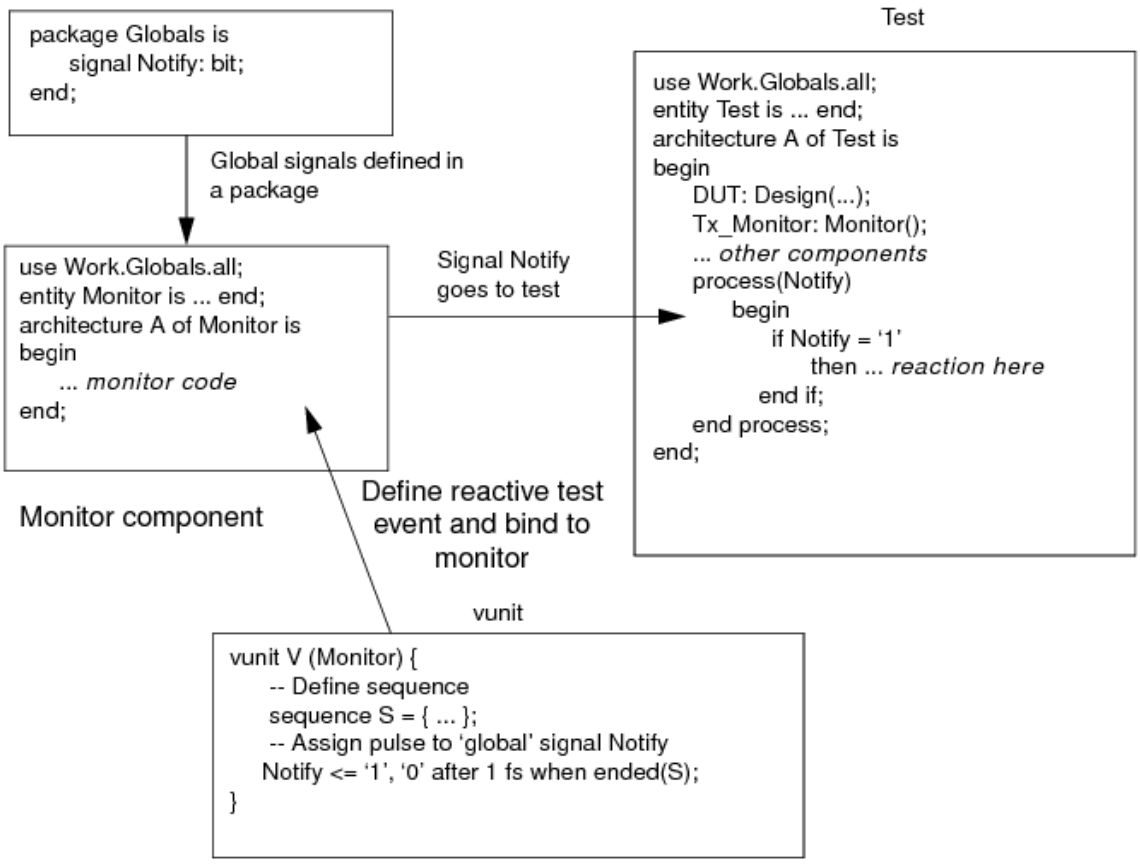


VHDL Reactive Test with PSL

In this example, PSL is used for a VHDL test and VHDL monitor. In this case, it is recommended that the sequence be defined as an adjunct to the monitor rather than in the test, because there is no support for OOMRs in VHDL. Instead, global signals are used to communicate between the test and the monitor, so you need to minimize the number of signals that go between the two.

Example 4-2 shows an implementation of this technique.

Example 4-2 VHDL Test with PSL Assertions



Using PSL in VHDL

Using PSL Assertions in VHDL generate Statements

The VHDL `generate` statement allows conditional elaboration of a portion of a description. For example:

```
if (boolean condition) generate;
     begin
          -- psl label: assert my_property(i+1, j-1);
     end_generate
```

When `condition` is true, the following is executed:

```
-- psl label: assert my_property(i+1, j-1);
```

A default clock in a `generate` scope affects directives only within that `generate` scope. If there is no default

clock in the `generate` scope, a default clock in the design unit scope applies, if it exists.

Using VHDL Outputs in PSL Assertions

Because the VHDL code written in PSL must follow the same language rules as the architecture body with which it is associated, using VHDL outputs in PSL assertions can result in the following error:

```
*E, WSSNEX static readable signal name expected
```

Some restrictions on VHDL output types have been relaxed in the latest version of the VHDL standard. To prevent the `WSSNEX` error, you can compile with the `-v200x` option to the `ncvhdl` compiler, which uses this newer set of rules.

Using PSL ended() in HDL

Although the PSL standard states that the `ended()` function can be used in the modeling layer, the Cadence implementation extends this capability to any HDL code. There are two common ways that this feature might be used. In the following example, the `ended()` function is evaluated on the negative edge of the clock to avoid a race condition.

```
always @(posedge clk)
     if ( ended({a;b}, negedge clk) )
         $display ("Sequence a;b detected at %t", $time);
```

Another way that `ended()` can be used in HDL, which is not recommended, is as an event control. For example:

```
always @(ended({a;b}, posedge clk ) )
     $display ("Sequence a;b detected at %t", $time);
```

This code will execute twice each time the sequence `a;b` is detected--once when the Boolean return value is asserted, and once when the Boolean return value is deasserted. The Boolean return value is active for one clock interval so, for Verilog level-sensitive event controls, the statements are executed on any change. Although you might have intended the statement to execute once, when the sequence ends, it will always execute twice.

Another type of unexpected behavior can result if the sequence ends multiple times on consecutive clock cycles. You might have intended for the statement to be executed each time a sequence match is true, but it will only be executed twice.

When considering the use of `ended()`:

- To avoid race conditions, always specify a clock for `ended()` that is different from that of the property or HDL code.

- Remember that when `ended()` is used as a Verilog event control, execution occurs when the result of the `ended()` evaluation changes value--both when it becomes true and when it becomes false.

How Synthesis Pragmas Convert to PSL

When you enable synthesis pragma processing by using the `-genassert_synth_pragma` compiler option, the parser converts the `one_hot`, `one_cold`, `full_case`, and `parallel_case` synthesis pragmas in `cadence`, `synopsys`, and `ambit` forms to PSL assertions for simulation (for details, see "Enabling Synthesis Pragma Checking for ABV" in the *Assertion Checking in Simulation* guide).

Synthesis pragmas are converted to PSL as follows:

- `full_case`

  The PSL assertion generated for a `full_case` statement checks whether the control signal of the `case` statement can get a value other than the values set in the `case` statement. Consider the following `case` statement:

```
wire [1:0] ctrl;
case (ctrl) // cadence full_case
    2'b00: ...
    2'b01: ...
    2'b10: ...
endcase
```

During simulation, if `ctrl` gets any value other than 0, 1, or 2, the assertion fires. Although this `case` statement does not have 2'b11 as one of its options, it can still comply with the full-case check if 2'b11 was never exercised during simulation.

- `one_hot`

  The PSL assertion generated for the `one_hot` pragma checks whether the associated bus to the pragma is one-hot encoded. One-hot encoding specifies that only one of the bits of the bus can have a 1 value at any given time. For example, a four-bit bus with one-hot encoding can only have a value of 0001, 0010, 0100, or 1000.

- `one_cold`

  The PSL assertion generated for the `one_cold` pragma checks whether the associated bus to the pragma is one-cold encoded. One-cold encoding specifies that only one of the bits of the bus can

have a 0 value at any given time. For example, a four-bit bus with one-cold encoding can only have a value of 0111,1011, 1101, or 1110.

- `parallel_case`

  The PSL assertion generated for a `parallel_case` pragma ensures that two or more choices of a `caseX` or `caseZ` statement are not selected at the same time. The assertion fires if more than one choice is activated during simulation.

Enabling/Disabling Assertions in HDL

You can use the SVA `$asserton`, `$assertoff`, and `$assertkill` assertion control system tasks to enable and disable SVA and PSL assertions in your code.

For details, see "Assertion Control System Tasks".

Debugging Assertions in Protected IP

For details about how Incisive simulation treats assertions in protected IP, see Chapter 10, "Writing Assertions for Protected IP."

Using hdltype Arguments

When you use `hdltype` data types as formal arguments to sequences and properties, the actual arguments must be the same type as the formal argument. For example:

```
typedef reg [3:0] foo;
foo f;
reg [4:1] g;

// psl property P1(hdltype reg [3:0] a) = always (a[4]);
// psl property P2(hdltype foo b) = always (b[4]);
// psl A1: assert P1(f);
// psl A2: assert P1(g);
// psl A3: assert P2(f);
// psl A4: assert P2(g);
```

All of these examples will report a BNDWRN warning, "bit-select or part-select index out of declared bounds," because

- A1 and A3--The formal argument type is `reg [3:0]`, and the actual argument type is `reg [3:0]`, but always `(a[4])` specifies a bit-select out of the `[3:0]` range.

- A2 and A4--The formal argument type is `reg [3:0]`, but the actual argument type, `reg [4:1]`, is different.

Using PSL Assertions in AMS Designs

For information about using PSL assertions in an analog/mixed-signal design, see "Applying Assertions to real, wreal, and electrical Nets" in Chapter 7, " Preparing the Design: Using Mixed Languages " in the *Virtuoso® AMS Designer simulator User Guide*.

5

Writing SystemVerilog Assertions

The SystemVerilog assertion language, SVA, lets you specify assertions for SystemVerilog designs.

For more information about SystemVerilog assertions, see

- *IEEE 1800 Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language*.

For more information about the Cadence implementation of SystemVerilog, see

- Cadence *SystemVerilog Reference*
- Cadence *Debugging SystemVerilog*

SVA Immediate Assertions

```
[ label ] assertboolean_expression [ action_block ] ;
```

Immediate assertions are executed like procedural code during simulation.

An immediate assertion tests an expression when the statement is executed in the procedural code. If the expression is interpreted as true, the assertion passes. If the expression is interpreted as false, the assertion fails.

By default, when an assertion fails, the simulator calls the `$error` severity task, which displays an error message. No action is taken by default when an assertion passes. However, you can include an action block ("SVA Action Blocks") in an immediate assertion to specify the action to take on success, and to change the default behavior to take when the assertion fails.

The `assert` directive causes an immediate assertion to be checked during simulation. You place this statement in the procedural code wherever you want the assertion to be checked. The *`label`* becomes the name of the assertion.

The following examples define several styles of immediate assertions inside `always` blocks. Comments in the code describe each style.

```
always @(negedge clk) begin
    // state_is_valid defines no pass or fail actions, so the defaults are used.
    // It checks that state is between ready and finish_burst state.
    // The "state_is_valid" label creates a named block around the
    // assert statement, and becomes a part of the design hierarchy.
    //
    state_is_valid : assert (state >= `ready && state <= `finish_burst)

    if(!enable_n)

        // enable_set_during_read_op_only has fail code only.
        // It checks that enable is not set during a read operation.
        // If it evaluates to true, no action is taken.
        // If false, a warning message is printed.
        //
        enable_set_during_read_op_only :
        assert (state >= `start_read && state <= `finish_read)
                else $warning("Enable set when state => %b", state);

        // The following unlabeled assertion also defines a fail action.
        // It checks that the memory task register, m_task, has a valid value.
        // It displays a message if m_task has an invalid value.
        //
        assert(m_task === `clear_memory || m_task === `write_memory ||
               m_task === `read_memory  || m_task === `read_burst)
            begin
                if(done) // "done" is a register
                    $display("%t Finished command %b",$time,last_task);
            end
                else $error("Unsupported memory task command %b",m_task);
end

always @(read_n || address) begin
    case (bank)
     "RAM_BANK_0" : mem_delay = 5;
     "ROM_BANK_0" : mem_delay = 10;
     "RAM_BANK_1" : mem_delay = 15;
     default :
       begin
         // Here, an assertion is used inside a case statement.
         // It checks that the correct names of memory bank variables are used.
         //
         assert(0) else $error("Unexpected bank setting. %s\n",bank);
         mem_delay = 0; /* Default */
       end
 endcase
end
```

Notes about immediate assertions:

- Immediate assertions can appear in any procedural code, including `initial` blocks, `always` blocks, tasks, class methods, functions, and program blocks.

  Note: Immediate assertions cannot be declared directly in compilation units or in program blocks. They can be defined inside any procedural code (`initial` and `always` blocks), and in tasks and functions inside packages, in compilation unit scopes, and in class methods like functions and tasks.

- Immediate assertions that appear in classes are shown in the Assertion Browser, and reported in the assertion summary. The action blocks of immediate assertions that appear in classes will be executed, and the standard error messages will be logged to the log file. A good example of the use of an immediate assertion in a class is with the randomization task:

  ```
  varRandomizationOK: assert (randomize(var))
      else $fatal(1, "Randomization failed: conflicting constraints");
  ```

  Simulation stops and the message goes to the log file when the randomization is not successful.

- The `assertion -redirect` command does not work for immediate assertions.

- In accordance with the IEEE 1800 SystemVerilog standard, `vpi_control` does not apply to immediate assertions. Only system-level VPI commands apply.

SVA Deferred Assertions

```
[ label:] directive#0(expression)[ action_block ] ;
```

A deferred assertion is used to suppress the errors that occur due to glitching activity on combinational inputs to immediate assertions. A deferred immediate assertion is similar to simple immediate assertion with the following key differences:

- A #0 delay is specified after the assertion directive
- Reporting is delayed rather than immediate
- Action blocks can contain only a single subroutine call
- Assertions can be used outside procedural blocks as a module_common_item

In a deferred assertion, a #0 delay control is specified after the directive to delay reporting. This is followed by an expression and an action block. For example:

```
always@(clk)
begin
    x=foo();
    a1: assert #0 (x>0) $display("a1 passed");
    else $error("a1 failed as x=%d",x);
end
```

If the action block contains a pass or a fail statement, then each of the statements can contain only one subroutine call, which can be a task, a task method, a void function, a void function method, or a system task. In this case, `begin-end` statements cannot surround the pass or fail statements as `begin` is not a subroutine call.

In an action block, a subroutine argument can be passed either by value or by reference as a ref and a const ref. If an argument is passed by value, the expressions use the values of the underlying variables when the deferred assertion expression was evaluated. However, if the argument is passed by reference, expressions use the current values of the underlying variables in the Reactive region. Further, it is not allowed to pass automatic or dynamic variables as actual to a ref or a const ref formal.

Note: As a limitation in a subroutine call in an action block, saved values are not used when actuals are out of scope or wires.

Reporting in a Deferred Assertion

In a deferred assertion, though the deferred assertion's expression is evaluated when the deferred assertion statement is processed but the reporting or action blocks are scheduled in the Reactive region in the current time step.

The action block subroutine call and the current values of its input arguments are placed in a deferred assertion reporting queue with the currently executing process. When a deferred assertion flush point is reached, the deferred assertion reporting queue is cleared and any pending assertion reports are not executed. After a queue reaches the deferred assertion flush point, each pending report that is in the Observed region of each simulation time step and has not been flushed either matures or is confirmed for reporting. The associated subroutine call in the pending report is executed in the Reactive region and finally, the report is cleared from the specific deferred assertion report queue. Remember that any report that matures cannot be flushed.

In case a deferred assertion expression is evaluated in the Observed region, the deferred assertion matures immediately. For example:

```
wait (S.triggered);//Here, S is a sequence.
 A1: assert #0 (expr);
```

In the given example, `S.triggered` stands true only in the Observed region. Therefore, the deferred assertion `A1` will hit only the Observed region and will definitely report.

Note: At times, code in the Reactive region may modify a signal and lead to another pass to the Active region. In this case, the Active region may re-execute some of the deferred assertions with different reported results and may lead to glitching behavior. Though deferred assertions prevent glitches due to order of procedural execution but do not prevent glitches caused by execution loops between regions due to the assignments from the Reactive region.

Deferred Assertion Flush Point

A deferred assertion flush point is reached when:

- A process that was earlier suspended resumes execution on reaching an event control or wait statement.
- A process that was declared by an always_comb or always_latch resumes execution due to a transition on one of its dependent signals.
- The outermost scope of the process is disabled by a disable statement. However, when you disable a task or a non-outermost scope of a procedure any pending reports are not flushed.

Note: In IES, the flushing of deferred assertions in case the outermost scope of the process is disabled by a `disable` statement is not supported and the results are reported.

Disabling a Deferred Assertion

You can disable a deferred assertion by using a `disable` statement. When you disable a specific deferred assertion, all pending assertion reports are cancelled.

Usage of a Deferred Assertion

You can specify a deferred assertion either inside or outside a procedural code. The following example illustrates specifying a deferred assertion inside a procedural code:

```
always@(clk)
begin
    x = foo();
    a1: assert #0 (x>0) $display("a1 passed");
    else $error("a1 failed as x=%d",x);
end
```

When you specify the deferred assertion outside the procedural code as a *module_common_item*, it is treated as if it is in an *always_comb* procedure. For example:

```
module m (input a, b);
    a1: assert #0 (a == b);
endmodule
```

is equivalent to:

```
module m (input a, b);
    always_comb begin
      a1: assert #0 (a == b);
    end
endmodule
```

Deferred Assertion in Multiple Processes

As deferred assertions are associated with the processes in which they are executed, a deferred assertion in a function is executed whenever a function is called in any process. A function can be executed multiple times by different processes and each of these executions is independent. Consider the following example of execution of deferred assertion in multiple processes:

```
function void foo (int x);
    a1: assert #0 (x != 0);
endfunction

always_comb
begin:b1
    foo(a);
end
always_comb
begin:b2
    foo(b);
end
```

In the given example, if the function `foo` is called from processes `b1` and `b2` at the same time step, then the execution of assertion `a1` through process `b1` will never flush the reporting of assertion through process `b2`, and vice versa.

SVA Concurrent Assertions

```
  [ label ] directiveproperty (property_expression)
     [ action_block ] ;
```

Concurrent assertions can describe Boolean behaviors or patterns of behavior that span clock cycles. They can also specify assumptions about the environment, or monitor behavior for functional coverage. Concurrent assertions are distinguished by the `property` and `sequence` keywords; for example, `assert property`.

Concurrent assertions are triggered by a clock, which must be explicit.

You can place a concurrent assertion outside of procedural code in a module, interface, clocking

block, or program. You can also place a concurrent assertion in simple `initial` and `always` blocks. Assertions in `initial` blocks are evaluated only once ("Embedding Concurrent Assertions in Procedural Code").

The following example defines a property, `P1`, which checks that if `a` is true, and `b` is true two clock cycles later, then `c` must be true one clock cycle after that. The concurrent assertion, `A1`, causes this property to be checked during simulation or formal analysis.

```
property P1;
    @(negedge clk) (a ##2 b) |=> (c);
endproperty

A1: assert property (P1);
```

> You can only use static class members in concurrent assertions. You cannot use a member variable unless the variable is initialized at time 0. For example, you cannot define a clock as a member of a class, then use it to clock a property. Doing so results in a run-time error.

Labels in SVA

```
identifier:
```

A label gives a name to an assertion, assumption, or coverage statement. You can use a label in other properties and in simulator Tcl commands to refer to the assertion, assumption, or coverage statement.

Labels are optional, but strongly recommended. The label you provide is the name that will be associated with this assertion in assertion and coverage analysis reports, and in the Assertion Browser and coverage GUI. They can make it easier to locate and understand failed properties when debugging your simulation.

If you do not provide a label, the tool generates one.

The *identifier* must be a legal Verilog or SystemVerilog name, and it must be unique within the design name space.

This example defines an assertion labeled `a_or_b_then_not_c`:

```
a_or_b_then_not_c: assert property (@(posedge clk) ((a || b) |-> ##1 !c));
```

Boolean Expressions in SVA

operandopoperand

A Boolean expression describes a behavior that might be true during a single clock cycle. It can be any SystemVerilog expression that evaluates to `0`, `1`, `X`, or `Z`. The value `1` is interpreted as true; the values `0`, `X`, and `Z` are interpreted as false.

A Boolean expression is satisfied in the clock cycle in which it evaluates to true.

The operands can be any data type except `shortreal`, `real`, `realtime`, `string`, `event`, `chandle`, `class`, `associative arrays`, and `dynamic arrays`. An operand can include function calls, including calls to system functions, such as `$onehot` or `$countones`. System functions are described in Bit-Vector System Functions.

For example, the following expression evaluates to true if `a` is `1` or `b` is `0`:

```
(a || !b)
```

Formal Arguments of Sequences and Properties

You can declare a sequence or property with optional formal arguments. The syntax of formal arguments to sequences and properties is identical. When you instantiate the sequence or assert the property, you pass the actual arguments to it.

Untyped Formal Arguments

A formal argument that is not prefixed with a data type and also does not follow a set of arguments with a specific data type is treated as untyped. You can also use the untyped keyword to specify a data type as untyped. When you use an untyped keyword, you can mix typed and untyped arguments in the same sequence or property definition.

The following example declares untyped formal arguments. In this example, the `S1` sequence is defined with two formal arguments, `term2` and `term3`. When the sequence is used in the `S1_assertion` assertion, actual arguments `b` and `c` are substituted for the formal arguments. The `S2` sequence is used in the same way. In the `S2` sequence, the formal arguments are declared as untyped using the untyped keyword.

```
// Sequence
sequence S1 (term2, term3);
    (a ##1 term2 ##1 term3);
endsequence

sequence S2 (untyped CC, DD);
    (a ##1 CC) or (b ##1 DD);
endsequence

S1_assertion: assert property
    (@(negedge clk) (S1(b,c)) |=> S2((e||f), g));
```

Typed Formal Arguments

You can optionally specify the data type for a formal argument. This technique is recommended, because it enables compile-time semantic checks. Consider the given example in which sequence S1 defines two arguments of type `bit`.

```
sequence S1 (bit AA, bit BB);
    (a ##1 AA ##1 BB);
endsequence
```

When an argument follows a set of arguments with a specific data type, it takes up the data type of the preceeding set of arguments. For example, Property P5 shows a set of arguments in which the argument `AA` is untyped, the argument `BB` that is of the type `bit`, the argument `CC` takes up the `bit` data type, and `DD` is specified as `untyped`.

```
property P5 (AA, bit BB, CC, untyped DD);
    @(negedge clk)
    (BB ##1 CC) |=> (AA ##[1:2] (DD||AA));
endproperty
```

When using typed formal arguments, all arguments can be initialized and all integral types are automatically truncated or sign/zero-extended for width mismatches before type checking.

IES supports sequence, property and event data types in typed formal argument. Declaraing a data type as a sequence or property restricts the actual argument to a sequence or property instance or expression. For example:

```
Property P5( sequence S, property P)
    @(posegde clk)
      S or P;
Endproperty
```

```
A1: assert property ( P5 ( a ##1 b, c |-> d );
```

In the given example, the actual arguments `S` and `P` are restricted to a sequence or property.

Note: Property data type is only allowed in property declaration.

Limitation of Typed Formal Arguments

- You can pass a signal that is used as an event argument (`clk`), but not the event (`@clk`). For example, you can use

  ```
  property exampl1(clk,x,y);
      @(clk) x |=> y;
  endproperty
  ```

  but you cannot use

  ```
  property exampl1(@clk,x,y);
      @(clk) x |=> y;
  endproperty
  ```

Actual Arguments

When you instantiate a sequence or property, the argument list can be either name-associated or positional. For example, P5 can be instantiated in either of the following ways:

```
positional_association: assert property (P5 (my_AA, my_BB));
named_association: assert property (P5 (.AA(my_AA), .BB(my_BB)));
```

In both cases, `AA` is assigned `my_AA`, and `BB` is assigned `my_BB`.

Defaults for SVA formal arguments are also supported. In the following example, when P6 is instantiated, `en` takes the default value of 1. Arguments with a default value do not have to be specified in the instance. The `positional_association1` assertion relies on the default value of `EN`, while `positional_association2` relies on the default value of `BB`:

```
parameter true = 1;
property P6 (bit AA, BB=`true, EN=1);
    @(negedge clk)
    EN |-> (BB ##1 c) |=> (AA ##[1:2] (d||AA));
endproperty
named_association: assert property (P6(.AA(my_AA),.BB(my_BB)));
positional_association1: assert property (P6(my_AA, my_BB));
positional_association2: assert property (P6(my_AA, ,1));
```

For this sequence:

```
sequence BusReq (REQ, ACK);
```

```
    REQ ##[1:3] ACK;
endsequence
```

`BusReq` can be instantiated in either of the following ways:

```
  BusReq(rq, ak)                // Positional
  BusReq(.REQ(rq), .ACK(ak))    // Name-associated
```

In both cases, `REQ` is assigned `rq`, and `ACK` is assigned `ak`.

You can also declare defaults for SVA formal arguments. In the following example, when `BusReq` is instantiated, both `min` and `max` take the default value of `0`:

```
sequence BusReq (min=0, max=0);
    REQ ##[min:max] ACK;
endsequence
```

Clocking in SVA

SVA Clock Expressions

```
@(identifier | event_expression)
```

A clock expression specifies the clock or event that controls the evaluation of a sequence or property. The values are sampled as follows:

- *identifier* --On both edges of the expression for level expressions
- *event_expression* --When the edge expression occurs

If a clock expression specifies `@(negedge clk)`, for example, the values in the assertions are sampled when the `clk` signal transitions to low.

For example:

```
property P0;
    @(posedge clk)
    (a ##1 b ##1 c) |=> (!a ##[0:2] !b);
endproperty
```

If asserted, the property `P0` is evaluated on the rising edge of `clk`.

Note: You cannot use the `$rose` and `$fell` sampled-value functions as clock expressions.

SVA Default Clocking Blocks

```
default clocking [ clk_identifier ] [ clocking_event ];
                 clocking_items
endclocking [ :clk_identifier ]
```

A default clocking block specifies the clock or event that controls the evaluation of sequences or properties within the block. For example:

```
default clocking master_clk @(posedge clk);
    property p4; (a |=> ##2 b); endproperty
    assert property (p4);
endclocking : master_clk
```

The *clocking_event* can be a clock event expression or a *clk_identifier* that was defined by using a previous `clocking/end clocking` block. For example:

```
clocking pos_clk @(posedge clk);
     ...
 endclocking : pos_clk

 default clocking pos_clk;
```

The *clocking_items* can be properties, sequences, and assertions. You cannot use hierarchical names to refer to properties in clocking blocks.

Note: In the Cadence implementation, assertion statements are permitted in clocking blocks. However, assertions in clocking blocks are not permitted by the IEEE 1800 standard.

Notes about default clocking blocks:

- A contextually-inferred clocking event ("Embedding Concurrent Assertions in Procedural Code") in a procedural block takes precedence over a default clock.
- The default clock applies only when there is no explicit or inferred clock. An explicitly specified leading clocking event takes precedence over a default clock.
- Only one default clock per module is permitted.
- You cannot use a clocking block name as an explicit clock.
- No explicit clocking event is allowed within a sequence or property declaration in a clocking block.
- A property or sequence declared outside a clocking block that is instantiated in a clocking

block must be singly-clocked, and its clocking event must be identical to that of the clocking block.

- General clocking blocks are not supported for SVA; only the default clock is supported.
- Clocking blocks and default clocking cannot be specified in a `generate` block.

Embedding Concurrent Assertions in Procedural Code

A concurrent assertion statement can also be embedded in a procedural block. This is shown in the following example:

```
property prop;
    a ##1 b |-> c;
endproperty

always @(posedge clk) begin
  <statements>
assert property (prop);
end
```

If no clocking event is specified in a procedural concurrent assertion, the leading clocking event of the assertion shall be inferred from the procedural context, if possible. If no clock can be inferred from the procedural context, then the clocks are inferred from the default clocking, as if the assertion were instantiated immediately before the procedure.

A clock is inferred for the context of an always or initial procedure that satisfies the following requirements:

1. There is no blocking timing control in the procedure.
2. There is exactly one event control in the procedure.
3. Within the event control of the procedure, there is exactly one event expression that satisfies both of the following conditions:

- The event expression is of the form edge identifier expression1 [iff expression2] and is not a proper subexpression of an event expression of this form.
- No term in expression1 appears anywhere else in the body of the procedure.

Examples of Inferring a Clock:

1. Here the leading clock for assertion will be "posedge clk", as no assertion clock is specified.

```
always @(posedge clk)
      A1: assert property ( a ##1 b);
```

2. Here the leading clock for assertion will be "negedge clk", as procedural clock 'clk' does not have any edge identifier.

```
default clocking def_clk @ (negedge clk);
endclocking

always @(clk)
       A1: assert property ( a ##1 b);
```

3. Here the leading clock for assertion will be NONE.

```
// No default clocking present

always @(clk)
       A1: assert property ( a ##1 b);
```

This will lead to an error at parse time.

Procedural Assertion Flush Points

A process is defined to have reached a procedural assertion flush point if any of the following occur:

- The process, having been suspended earlier due to reaching an event control or wait statement resumes execution.

When a Procedural Assertion Flush Points is reached, the assertion reporting queue is cleared and any pending assertion reports are not executed.This functionality is enabled under -procsva option and is a compile time option. The -procsva option currently supporting procedural assertions in always and initial block will be deprecated in 13.2. The concurrent assertions in procedural blocks will now be supported by default.

Limitations:

- Automatic variables are not supported for concurrent assertions in procedural code.
- The flushing of concurrent assertions in procedural code in case the outermost scope of the process is disabled by a disable statement is not supported.
- Disabling of procedural concurrent assertion
- Flushing in fork/join

SVA Multiple Clocks

You can specify multiple explicit clocks in a sequential expression. For example:

```
assert property (@(posedge clk1) (a ##1 b) |=> @(posedge clk2) (c ##1 d));
```

Multiple-clock sequences must be built by concatenating single-clocked sub-sequences using the single-delay concatenation operator, `##1`, or the `|=>` implication operator. The delay is understood to be from the endpoint of the first sequence that occurs at the tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

All properties must have a single leading clock. The statistics counter for the number of attempts will be based on this leading clock.

Clock flow and multi clock support changes are compliant with the SV 1800-2009 LRM, however, with the following limitations:

- ##0 operator is not supported in the multiclock context, that is, when two operands to ##0 are differently clocked. (This support is as per 2005 LRM - and not as per 2009 LRM).
- For if/else operations, the if and else clause properties must begin on the same clock as the test of the if Boolean condition. (This support is as per 2005 LRM - and not as per 2009 LRM).
- Empty match is not handled correctly in multiclock properties. For example, the given code is not handled correctly:
  ```
  @(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1];
  ```
- Recursive properties and property or are not supported in multiclock ontext

Notes about multiple clocks:

- For multiply-clocked properties, the number of attempts to match the specified behavior is based on the leading clock.
- The scope of a clocking event
  - Flows from left to right across linear operators--repetition, concatenation, negation, implication
  - Distributes to the operands of branching operators--conjunction, disjunction, intersection, `if/else`

    until it is replaced by a new clocking event.
- The scope of a clocking event flows into parenthesized subexpressions, but does not flow out of the enclosing parentheses.
  If the subexpression is a sequence, if flows left to right across the parenthesized subexpression.
- The clocking events of sequence and property definitions do not flow out of the definitions.

This definition:	Is equivalent to:
`@(c) x |-> @(c) y ##1 @(d) z`	`@(c) x |-> y ##1 @(d) z`
`@(c) x |=> @(d) y ##1 z`	`@(c) x |=> @(d) (y ##1 z)`
`@(d) @(c) x`	`@(c) x`

SVA Local Variables

You can use dynamically created local variables to manipulate data within named sequences and properties. Local variables can be dynamically created and assigned a value using an assertion variable declaration within the declaration of a named sequence or property. Local variables are useful for modeling data propagation through a pipeline.

Local Variables Declaration

```
  assertion_variable_declaration::= var_data_type
list_of_variable_decl_assignments
```

In the given local variable declaration, the arguments are as follows:

- The `assertion_variable_declaration` declares a local variable.
- The `var_data_type` is the data type of the local variable. This data type can be one of the supported data types allowed within assertions.

  Note: The data supported as local variables include `bit`, `byte`, `int`, `integer`, `logic`, `longint`, `reg`, `shortint`, `time`, Packed `struct`, `class`, signed and unsigned variants of these types, `reg`, `logic`, and `bit` vectors, and arrays of supported types.
- The `list_of_variable_decl_assignments` is a comma-separated list of one or more identifiers with optional declaration assignments. A declaration assignment is used to define the initial value of the local variable. This initial value is defined using an expression, which is not a constant.

SVA Local Variable Usage

Local variables in both sequence and property declarations enable you to pass data from one stage in a sequential expression to a later stage. These variables are dynamically created within an instance of a sequence and are deleted when the end of the sequence is reached.

```
(expression {, sequence_match_item }) [repetition_op]
```

You can initialize local variables in sequence and property declarations (Section 16.10 of system verilog LRM, IEEE 1800-2009). You can use local variable with input mode in argument list of parameterized sequence and properties. The following example illustrates the initialization of a local variable with input mode:

```
property P2 (local input int arg1 = 4 + P, local input int arg2 = 0);
    (a ##2 b) |=> (arg1 != 0) && (arg1 != arg2);
endproperty
```

You can specify local variables in a sequence or property expression that will be executed when its associated sequence is matched. Local variables can be assigned multiple times within a sequence. Local variables can be used within a check that spans an arbitrary interval of time, and that overlaps with other checks. These variables are used primarily to check data, especially when the latency is variable and out of order.

As shown in the following example, when `valid_in` is true, the `x` variable is assigned the value of `data_in`. If `data_out` is equal to `x` + 1 five cycles later, the property is true. Otherwise, the property is false. When `valid_in` is false, this property evaluates vacuously to true.

```
property pipeline;
      int x;
      @(posedge clk) disable iff (reset)
      (valid_in, x = data_in) |=> ##5 (data_out == (x+1));
  endproperty
```

The following example shows the use of a packed struct as a local variable:

```
// Pipeline data check
typedef struct packed signed {reg[7:0] x, y;} data_struct;

property ex1;
    data_struct ldata;
    @(posedge clk)
    (valid_in, ldata.x=data_in) |-> ##5 (data_out == ldata.x);
endproperty
```

Note: Local variables:

- Are dynamically created for every attempt to match the specified behavior, so copies of the variable are continually being created and destroyed.
  Because local variable values are so volatile, you cannot access these values with Tcl commands. Attempts to do so will produce an "object not found" message.
- Do not show up in the Design Browser, because they are dynamically created.

Limitations of Local Variable

- Local variable with `inout` and `output` mode in argument list of parameterized sequences and properties is unsupported and will be reported as an error.
- Local variables cannot be used in sampled value functions and system tasks such as `$past`.
- You cannot export the values of local variables through typed formal arguments.
- Local variables cannot be displayed using `$monitor`.
- The following data types are not currently supported as local variables:
  - `enum`
  - Non-integer types: `shortreal`, `real`, `realtime`
  - `string`
  - Tagged `union`
  - Unpacked `struct`

SVA Sequence Match Items

You can specify a *`sequence_match_item`* to be called at the end of a successful sequence match. You can call tasks, task methods, void functions, void function methods, and system tasks. You can also make assignments to local variables, and increment and decrement local variables. A *`sequence_match_item`* can be placed at any point in the sequence. If there are multiple match items, they are executed in the order of occurrence. Sequence evaluations do not wait for any task delays that might occur.

Subroutines can be called more than once at the same time. Multiple subroutine calls are executed in the order in which they are listed.

The result is similar to the operation of an action block. The difference is that the *`sequence_match_item`* is executed while the property or sequence is being evaluated, rather than

being executed only when the property or sequence passes or fails. You can use this feature to get more visibility into the inner workings of a sequential expression. For example:

```
sequence s1;
     logic v, w;
     (a, v=e) ##1 (b[->1], w=f,
     $display("b after a with v= %h, w = %h\n", v, w));
 endsequence
```

For details about this feature, see section 16.8, "Manipulating data in a sequence," and section 16.9, "Calling subroutines on match of a sequence," in the *1800 - 2009 IEEE Standard for SystemVerilog*.

SVA Sequences

A sequence describes a behavior that can span time. It is made up of sequence expressions separated by time delays. The simplest type of sequence expression is a single Boolean expression. However, sequence expressions can be more complex, including the use of repetition operators and sequence instantiation.

A sequence can be declared in a module, interface, program block, clocking block, compilation unit scope, or package.

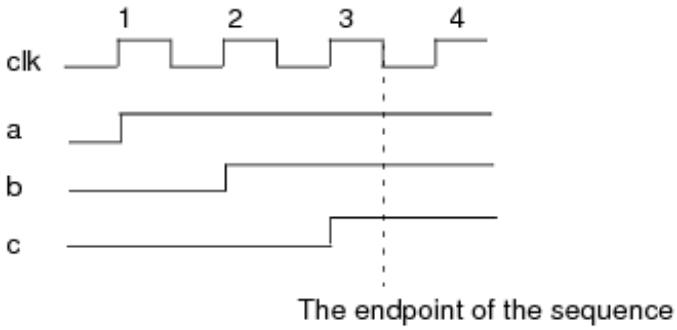
Sequences can be evaluated over several clock cycles. If each sequence expression evaluates to true in successive clock cycles, the entire sequence is said to be *tightly satisfied*.

Because they can span clock cycles, sequences have an *endpoint*--the clock cycle in which the last sequence expression is satisfied. For example, suppose you defined a sequence, which states

At the negative edge of `clk` *, if* `a` is true, then `b` must be true in the next clock *cycle, and* `c` must be true in the cycle after that.

The sequence ends in the last cycle of a series of cycles that tightly satisfy the sequence, as follows:

```
a ##1 b ##1 c;
```



The Incisive simulator uses "weak" semantics when evaluating SVA sequences. If the enabling condition of a sequential assertion has been satisfied, but it is incomplete at the end of simulation, the Incisive simulator does not report it as a failure.

Starting with 13.2 release, the default assertion evaluation semantics is weak for SVA assertions.

The default value of `assert_report_incompletes` is set to "`0`". Use `assert_report_incompletes 1` to get strong semantics.

For details, see the description of the Failed state in "Understanding Assertion States" in *Assertion Checking in Simulation* user guide.

SVA Sequence Declarations

```
  sequenceidentifier [ argument_list ] ;
[]    [ clock_expression] sequence_expression
    [ sequence_opsequence_expression ] ... ;
 endsequence [ :identifier ]
```

You can declare a sequence in a module, interface, package.

Note: Sequences referenced in a package must be defined in that package.

Sequence declarations are not checked; they merely describe a behavior, which is then used in another sequence or property.

You can instantiate a declared sequence in other sequences or properties. This technique makes a sequence or property definition more readable when you are checking for complex behaviors.

The following are supported for sequences:

- Typed Formal Arguments
- Clocking in SVA
- SVA Local Variables
- SVA Sequence Match Items

Note: Declaring `typedefs` within a sequence declaration is not permitted; only variable declarations are permitted.

SVA Sequence Instantiation

```
identifier [ ( actual_arguments ) ]
```

You instantiate a sequence by using its name in an expression. If the sequence declaration has formal arguments, you specify the actual argument values when you instantiate the sequence. The actual argument list can be either name-associated or positional.

In the following example, the `ReadTransaction` sequence contains an instance of the `BusReq` sequence, and passes the actual arguments, `rq` and `ak`, to the sequence:

```
sequence BusReq (req, ack);
    req ##[1:3] ack;
endsequence

sequence ReadTransaction ;
    BusReq(rq, ak) ##1 Transfer ##1 BusRls ;
endsequence
```

Note: Sequences cannot be referenced by using heirarchical names.

SVA Sequence Expressions

```
[delay] { boolean_expression [ repetition_operator ]
        | sequence_instantiation }
[sequence_expression] ...
```

A sequence expression is made up of Boolean expressions and instances of other sequences. All expressions in a sequence expression are evaluated on the clock edge of the property in which the

sequence is instantiated. You place time delays between sequence expressions to describe behaviors that span clock cycles.

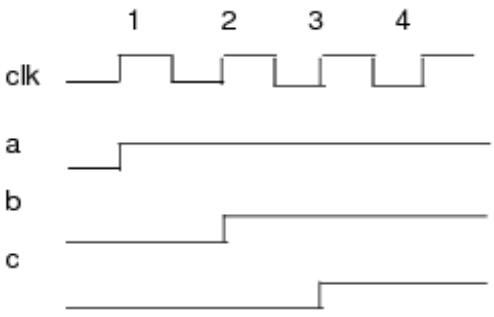
The *delay* value in a sequence expression can be any of the following:

##n	Specifies the number of clock cycles that occur between sequence expressions. The number of clock cycles can be `0`, to specify no delay.
##identifier	Specifies any named object that has a fixed integer value greater than or equal to `0` at simulation time.
##(constant_expression)	Specifies an expression that resolves to an integer value greater than or equal to `0` at compile time.
##[start:end]	Specifies a range of clock cycles. The *start* and *end* arguments can be integers, constant identifiers, constant expressions, or the $ token. Use $ to indicate an open-ended range. You can also use ##[*] for [0:$], and ##[+] for [1:$].

Note: The delay value can be an untyped formal argument. When passed as a delay, the value must be a compile-time constant. It cannot be an HDL variable value, or any expression using an HDL variable.

In the following example, assume the sequence is instantiated in a property that is clocked on the negative edge of `clk`. The sequence is satisfied when `a` holds, then `b` holds in the next clock cycle, and `c` holds in the clock cycle after that.

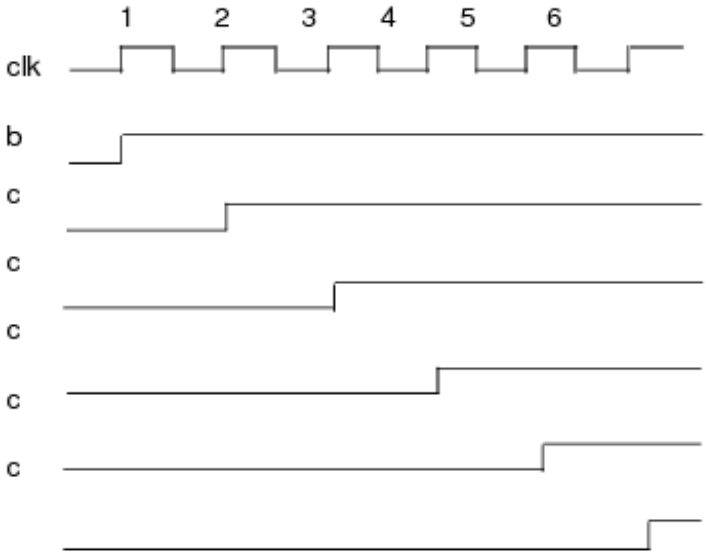
```
a ##1 b ##1 c;
```



The next example also assumes that the sequence is instantiated in a property that is clocked on

the negative edge of `clk`. In this sequence, `c` must hold in any clock cycle from the first through the fourth cycle after `b` holds, matching any of the possible waveforms shown for `c`.

```
b ##[1:4] c;
```



This example uses the identifier `max` to specify the number of clock cycles after `c` holds that `d` must hold:

```
c ##(max) d;
```

This example defines a sequence, `S1`, that takes two arguments. `S1` specifies that after the first argument occurs, the second argument must occur from zero to two clock cycles later. The sequence `S2` instantiates `S1` and passes the arguments `b` and `a` to `S1`.

```
sequence S1(AA, BB);
   (AA ##[0:2] BB);
endsequence

   sequence S2;
    S1(b,a);
endsequence
```

SVA Repetition

Repetition operators define a sequence in which the operand occurs repeatedly. For example:

```
req[*3]
```

is equivalent to the sequence

```
 ( req ##1 req ##1 req )
```

Similarly:

```
( clk ##1 !clk )[*3]
```

is equivalent to

```
( ( clk ##1 !clk ) ##1 ( clk ##1 !clk ) ##1 ( clk ##1 !clk ) )
```

which is equivalent to

```
( clk ##1 !clk ##1 clk ##1 !clk ##1 clk ##1 !clk )
```

When used as a formal argument, the repetition value must be untyped. Repetition values must be compile-time constants.

There are three classes of repetition operators:

- Consecutive repetition operator: `[*n]` and `[+]`

  These operators apply to either a Boolean or a sequence. The resulting sequence matches any trace in which the operand occurs for a specified number or range of consecutive cycles.

  You can also use [*] for [0:$], and [+] for [1:$].

- (Possibly) non-consecutive repetition operator `[=]`

  This operator applies to a Boolean. The resulting sequence matches any trace in which the operand occurs for a specified number or range of cycles, which might or might not be consecutive. That is, the trace might contain other cycles in which the Boolean does not occur, either before, in between, or following the cycles in which the Boolean does occur.

- GoTo repetition operator `[->]`

  This operator applies to a Boolean. The resulting sequence matches any trace in which the operand occurs for a specified number or range of cycles, which might or might not be consecutive, provided that the last occurrence is in the last cycle in the trace. That is, the trace might contain other cycles in which the Boolean does not occur, either before, or in between,

the cycles in which the Boolean does occur, but not following the cycle of the last occurrence of the Boolean.

Consecutive	(Possibly) Non- Consecutive	Goto
R **[*** *n* **]**	*B* **[=** *n* **]**	*B* **[->** *n* **]**
R **[*** *n* **:** *m* **]**	*B* **[=** *n* **:** *m* **]**	*B* **[->** *n* **:** *m* **]**
R **[*** *n* **:$]**	*B* **[=** *n* **:$]**	*B* **[->** *n* **:$]**

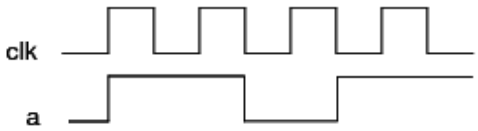
For example:

- `a[*3]` matches the trace (**a, a, a**)
- `a[=3]` matches the traces (**a, a, a**), (**a,** `b, b` **, a,** `b` **, a**), and (**b, a,** `b, b` **, a,** `b` **, a,** `b, b, b`)
- `a[->3]` matches the traces (**a, a, a**), (**a,** `b, b` **, a,** `b` **, a**), and (**b, a,** `b, b` **, a,** `b` **, a**)

If the repeat count is a range, the resulting sequence matches any trace in which the operand is repeated some number of times in that range. If the upper bound in such a range is `$`, the upper end of the range is unbounded--that is, the resulting sequence will match any number of repetitions equal to or greater than the lower bound of the range.

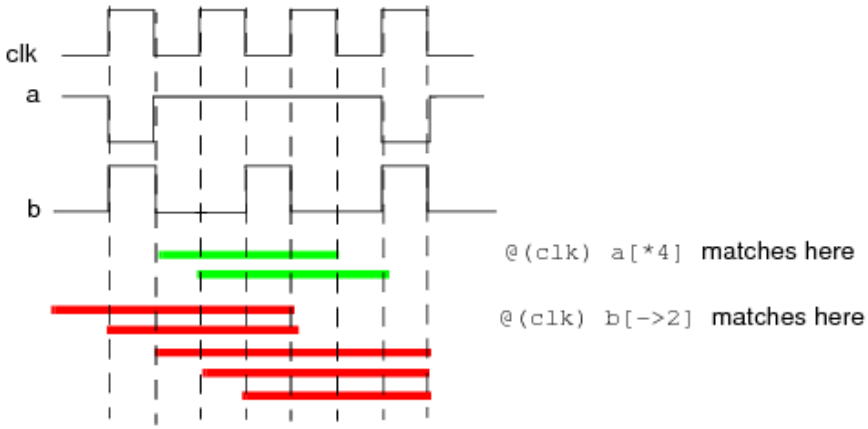
> The upper limit for a repeat count is about 64K. Using very large repeat values, such as `ack[*20000000]`, uses a large amount of memory, but unbounded repeat values, such as `ack[*]`, do not.

A repeated sequence might or might not match a trace, depending on how it is clocked. A sequence can be clocked by an explicit clock expression, or by inheriting a clock context. A clocked sequence is evaluated only when its clock expression is True. For example, given the following traces, sequence `@(clk) a[*3]` matches twice, but sequence `@(posedge clk) a[*3]` does not match:



A given sequence can match overlapping portions of the same trace. For example, given the

following traces:



SVA Sequence Operators

Sequence operators let you specify temporal relationships between sequence expressions.

> Assertions are intended to monitor the HDL, not change the values of signals. The increment and decrement operators are not allowed in SystemVerilog properties.

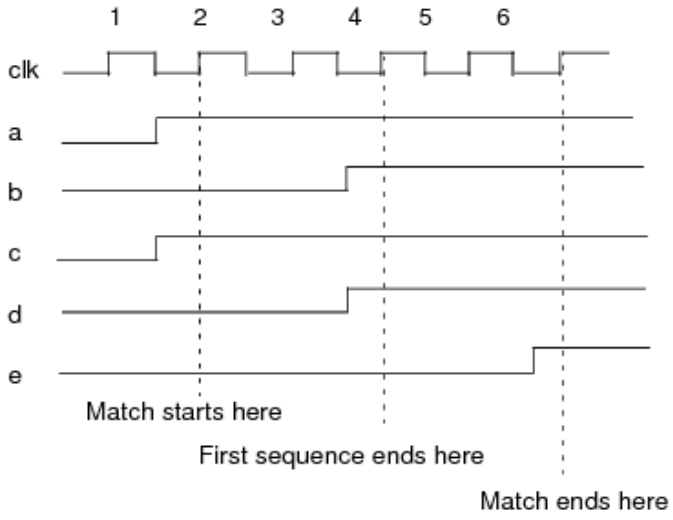
SVA and Operator

```
sequence_expression1 and sequence_expression2
```

Specifies that two sequence expressions must begin in the same clock cycle, but they do not need to end during the same cycle.

In the following example, the first sequence spans more time than the second sequence. The match starts when `a` and `c` are true, assuming that the sequence is evaluated on the positive edge of the clock. It ends when the second sequence completes; that is, when `e` holds.

```
(a ##2 b) and (c ##2 d ##2 e)
```



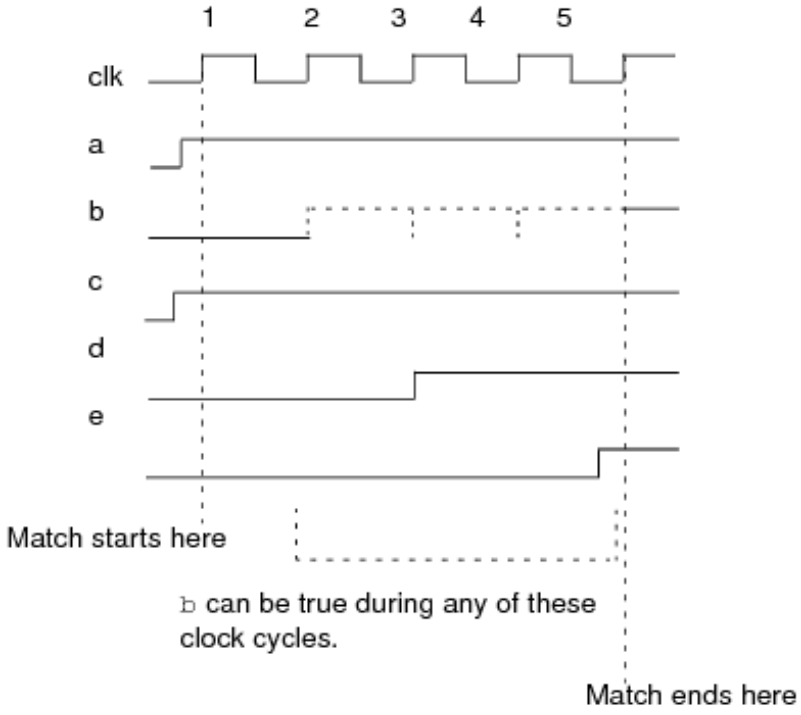
SVA intersect Operator

```
sequence_expression1 intersect sequence_expression2
```

Specifies that both sequences must end in the same clock cycle.

In this example, the match begins when `a` and `c` are true and ends when both `b` and `e` are true:

```
(a ##[1:5] b) intersect (c ##2 d ##2 e)
```



SVA or Operator

```
sequence_expression1 or sequence_expression2
```

Specifies that either the first or the second sequence expression must match, or both.

For example, this sequence matches if `b` holds, followed by `c` in the next clock cycle; or if `d` holds for one or two cycles, followed by `e` in the next cycle; or if `f` holds for two cycles. Any of these occurrences match the compound sequence:

```
(b ##1 c) or (d[*1:2] ##1 e) or f[*2]
```

SVA throughout Operator

```
boolean_expression throughout sequence_expression
```

Specifies that a condition must hold for the duration of a sequence expression.

For example, this expression ensures that an interrupt, `irq`, does not become true during the specified sequence.:

```
!irq throughout (b ##1 c) or (d[*1:2] ##1 e) or f[*2]
```

Note: In IES, in addition to boolean expression, Sequence Match Item (SMI) on boolean expression is also supported as the first operand of the "throughout" operator.

SVA within Operator

Specifies that a sequence must match at some point within the duration of another sequence.

For example, this expression checks whether the first sequence expression (`!a` for one to three cycles followed by `b`) occurs during the time when the second sequence expression (`c` followed by `!c` after five cycles) occurs:

```
(!a[1:3] ##1 b) within (c ##5 !c)
```

SVA first_match Operator

first_match(*sequence_expression* [, *sequence_match_item*]**)**

Specifies that the evaluation of a sequence stops when the first match of the sequence expression is found. If more than one sequence expression matches at that time, they are both first matches.

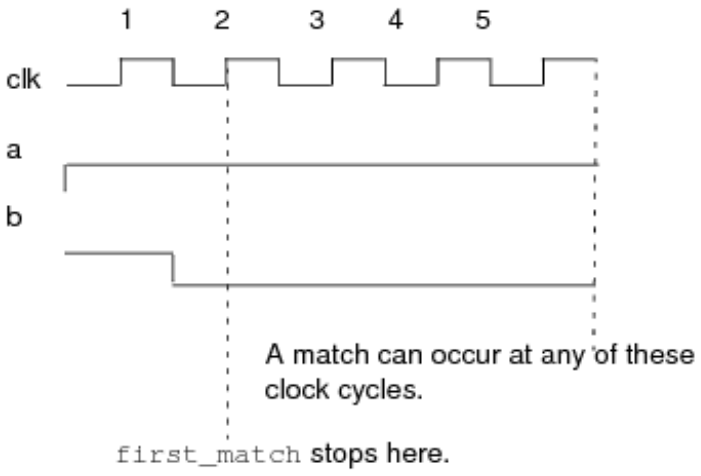
When the sequence expression is matched, the sequence match item is executed. This optional match item can be used to assign values to local variables or execute a $display statement. For example:

```
property P6;
   reg AA;
   @(negedge clk)
   first_match(seq1, AA = 0);
endproperty
```

If you use `first_match` in the enable condition of a property, it limits the instantiation of overlapping properties. If you use `first_match` in the fulfilling condition of a property, it has no effect, because the first match of a sequence is implicit in this context.

In this example, for the sequence evaluation beginning at time 1, only the match that ends at time 2 is used; matches ending at time 3 and beyond are ignored:

```
first_match(a ##[1:3] !b);
```



SVA Sequence Methods

You can use the `ended`, `triggered`, and `matched` sequence methods to identify the endpoint of a sequence. The endpoint occurs in any cycle in which the sequence completes, regardless of when the sequence started. When the endpoint occurs, the sequence evaluates to true; it evaluates to false otherwise. The syntax is:

```
sequence_instance.method
```

Because sequence method values are not available in the Preponed region, you cannot use sequence methods in sampled-value functions.

The following table compares the three available sequence methods.

Method	Duration	Use
`ended`	Set in the Observed region. Does not persist.	In a sequence
`triggered`	Set in the Observed region. Persists throughout the timestep.	In HDL, `disable iff`, and assertion context
`matched`	Set in the Observed region. Persists until the Observed region of the first clock tick of the destination sequence after the match.	To synchronize sequences with different clocks

For details about individual sequence methods, see:

- ended Sequence Method
- triggered Sequence Method
- matched Sequence Method

Limitations of Sequence Methods

- Sequence methods are not currently supported for:
  - Sequences as events, such as `@(seq1)`, although you can use `@(seq1.triggered)`
  - VPI interface
  - Sequences to which local variables are passed as an actual argument. For example:

```
sequence s1(x);
    @(clk) x ##1 b;
endsequence
property p1;
    int lv1;
    @(clk) s1(lv).triggered; // not supported
endproperty
assert property (p1);
```

ended Sequence Method

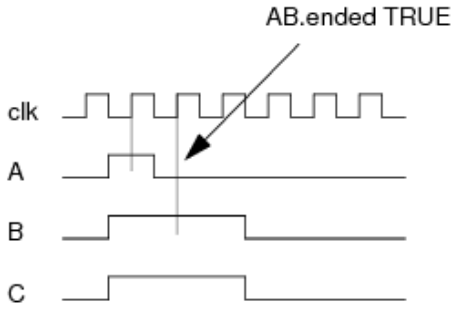
```
sequence_instance.ended
```

The `ended` method evaluates to true in a given clock tick if the sequence has reached its endpoint, and false otherwise. The ended status of a sequence is set in the Observed region, and persists throughout the Observed region.

The `ended` method can be used only to detect the endpoint of a sequence that is used in another sequence. It cannot be used in `disable iff` Boolean expressions for properties. Also, an error is reported when the use of `ended` causes circular dependencies between sequences. You cannot use this method in a sampled value function, because the sequence values are not available in the Preponed region.

An example of using the `ended` method is the following:

```
sequence AB;
   @(posedge clock) A ##1 B;
endsequence

A1: assert property (
   @(posedge clk)
   AB.ended |=> C);
```



Evaluation

When the `ended` method is evaluated in an expression, it tests whether its operand sequence has reached its endpoint at that particular clock tick. The `ended` method tests only for the endpoint of a sequence, not its starting point. In the following example, the endpoint of the `BusReq` sequence, when used in the `ReadTransaction` sequence, must occur one clock tick after Enable.

```
sequence BusReq;
    @(posedge clk) req ##[1:3] ack;
endsequence

sequence ReadTransaction;
    @(posedge clk) Enable ##1 BusReq.ended ##1 Transfer ##1 BusRls;
endsequence
```

If the `ended` call is removed from the `BusReq` instance, a match of `BusReq` must start one clock tick after `Enable`.

Using ended with Multiple Clocks

The `ended` method can be used with multiple clocks. However, the ending clock of the sequence instance on which `ended` is called must always be the same as the clock that immediately follows.

triggered Sequence Method

```
sequence_instance.triggered
```

The `triggered` method evaluates to true in a given clock tick if the sequence has reached its endpoint, and false otherwise. The triggered status of the sequence is set in the Observed region, and persists throughout the remainder of the time step--that is, until simulation time advances. You cannot use this method

- In a sampled value function, because the sequence values are not available in the Preponed

region.

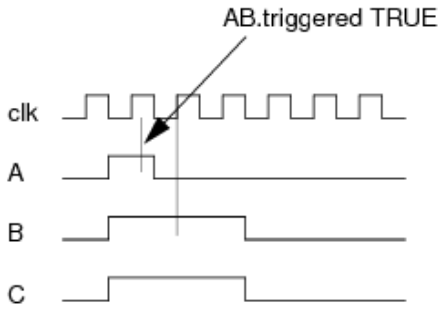
- On sequences that treat their formal arguments as local variables--that is, the formal argument is used as an lvalue in an operator assignment, or as an increment/decrement expression in a sequence match item.
- In action blocks.

The `triggered` method can be used only

- In `wait` and `if` statements of Boolean expressions outside of a sequence context
- In a `disable iff` Boolean expression for properties
- In assertion context

An example of using the `triggered` method is the following:

```
sequence AB;
   @(posedge clock) A ##1 B;
endsequence


wait (AB.triggered ||
BC.triggered);
   if (AB.triggered)
   $display("AB
triggered");
```



The `triggered` sequence method can be used to construct reactive tests. For details, see "SVA Reactive Tests using Sequence Methods".

Using triggered with Multiple Clocks

The `triggered` method can be used with multiple clocks. However, the ending clock of the sequence instance to which `triggered` is applied shall be the same as the clock in the context where the application of method ended appears.

matched Sequence Method

```
sequence_instance.matched
```

The `matched` method detects the endpoint of a sequence used in another sequence. The `matched` method can be used only in sequence expressions. This method is used when the clock of the source sequence, `sequence_instance`, is different from the clock of the destination sequence, which

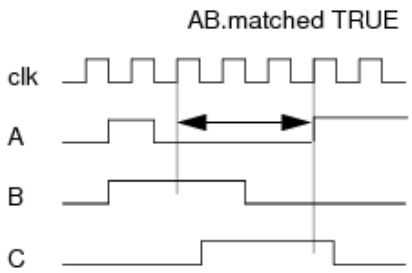
is the sequence in which matched is called on `sequence_instance`.

The `matched` method detects the endpoint of the source sequence, which is reached whenever there is a match of its expression. This method synchronizes the two clocks by storing the result of the source sequence until the first clock tick of the destination sequence after the match.

The matched status of the sequence is set in the Observed region, and persists until the Observed region following the first clock tick of the destination sequence after the match.

An example of using the `matched` method is the following:

```
sequence AB;
   @(posedge clock) A ##1 B;
endsequence

A1: assert property (
   AB.matched |=> @(A) C);
```



In the following example, the source sequence, `s1`, is evaluated at the positive edge of `clk`, while the destination sequence, `s2`, is evaluated at the positive edge of `sysclk`. In `s2`, the endpoint of the `s1` instance is tested to occur sometime after the occurrence of `inst`. The `matched` method tests only for the endpoint of a sequence, not its starting point.

```
sequence s1;
    @ (posedge clk) $rose(a) ##1 b ##1 c;
endsequence

sequence s2;
    @(posedge sysclk) reset ##1 inst ##1 s1.matched [->1] ##1 branch_back;
endsequence
```

A sequence instance on which `matched` is called can have multiple matches in a single cycle of the destination sequence clock. The multiple matches are treated semantically the same way as matching both disjuncts of an or. In other words, the thread evaluating the destination sequence will fork to account for such distinct local variable valuations.

Sequence Events

```
@sequence_name statement

@(sequence_name) statement
```

You can use a sequence instance in an event expression to control the execution of procedural statements, based on the successful match of the sequence. The process is resumed following the Observed region in which the endpoint of the match is detected.

Arguments to the sequence must be static. Using variables as sequence arguments results in an error.

The *sequence_name* cannot be an out-of-module reference (OOMR). You cannot use sequence events in classes and tasks.

A sequence event is a good alternative to an action block when only the pass condition is significant, and when the coverage statistics that come automatically with assertions are not needed.

In the following example, a sequence is defined once in the design under test, or in a design unit bound to the DUT, and it sends an event to the test when the sequence is detected:

```
event notify_test_of_bad_crc;

sequence bad_crc;
 @(posedge clk)
 data_state[*min:max] ##1 bad_crc_state[*2];
endsequence

@(bad_crc) -> notify_test_of_bad_crc;
```

One or more tests can detect the event by using an OOMR, as follows:

```
@(top.dut.notify_test_of_bad_crc) $display("Processing bad CRC frame");
```

Similarly, if an event is specific to a given test, you can use an OOMR to define the sequence in the test, as follows:

```
sequence bad_crc;
    @(posedge clk)
    top.dut.data_state[*min:max] ##1 top.dut.bad_crc_state[*2];
endsequence
```

The test can use a direct reference to the sequence, as follows:

```
@(bad_crc) $display("Processing bad CRC frame")
```

SVA Properties

A property describes a behavior that you want to check during simulation. It can be a simple behavior described by a single expression, or a complex behavior described by several expressions and sequences with specific relationships between them. A property is distinguished from a sequence in that it often contains implications.

Properties can be declared in a module, interface, program, package, or compilation unit scope.

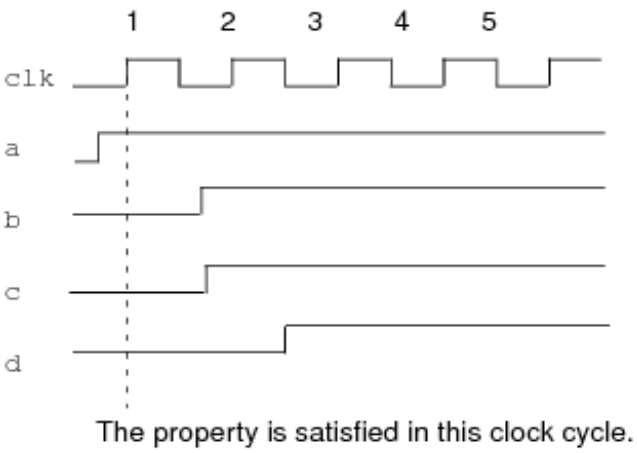
A property is similar to a Boolean expression, in that it holds at the beginning of the behavior that matches. However, unlike a Boolean expression, a property can span any number of clock cycles; it can even span an infinite length of time.

For example, suppose you have a property that states

If `a` *is true, and* `b` *is true on the next positive edge of* `clk`, then `c` must also be *true, and* `d` *must be true in the next clock.*

The property takes place over a span of three clock cycles. If all of the behaviors hold, it is said to be satisfied in the first clock cycle, as shown:

```
a ##1 b |-> c ##1 d;
```



The property is satisfied in this clock cycle.

It is also possible that a property will never be evaluated. During simulation, this can occur if the test vectors do not trigger the behaviors being checked. The property does not fail because it is never checked.

Note: Per the IEEE 1800 SystemVerilog standard, you cannot call the following types of functions from within a property:

- Functions that have output or inout arguments

- Functions that have non-constant reference arguments
- Functions that have static data declarations
- Functions that modify data outside of the function scope

SVA Property Declarations

```
 propertyidentifier [ argument_list ] ;[]
   [ clock_expression ] [ disable_clause ] property_expression ;
endproperty [ :identifier ]
```

You can declare a property in a module, interface, program, package, or compilation unit scope.

Note: Properties referenced in a package must be defined in that package.

The following are supported for properties:

- Typed Formal Arguments
- Clocking in SVA
- SVA Local Variables
- SVA Sequence Match Items

Note: Declaring `typedefs` within a property declaration is not permitted; only variable declarations are permitted.

SVA Property Instantiation

```
identifier [ ( actual_arguments ) ]
```

You instantiate a property by using its name in an expression. If the property declaration has formal arguments, you specify the actual argument values when you instantiate the property. The actual argument list can be either name-associated or positional.

In this example, the P6 property passes the actual arguments, `my_AA` and `my_BB`, to the property:

```
parameter true = 1;
property P6 (bit AA, BB=`true, EN=1);
    @(negedge clk)
    EN |-> (BB ##1 c) |=> (AA ##[1:2] (d||AA));
endproperty
assert_P6: assert property P6(.AA(my_AA),.BB(my_BB));
```

Note: Properties cannot be referenced by using heirarchical names.

SVA Disable Clause

```
 disable iff boolean_expression
default disable iff boolean_expression
```

The `disable iff` clause cancels any current obligations of the property to hold. You can use the `disable iff` clause to terminate checking of an assertion at any time.

In this example, if `rst` goes high, the current evaluation of the property is canceled, and the property cannot fail. No new evaluations start until `rst` goes low, at which point the implication is once again evaluated normally.

```
property P4;
    @(negedge clk)
    disable iff (rst)
    (c) |-> (##[max-1:$] d);
endproperty
```

The `default disable iff` clause specifies the default reset condition that will apply to all assertion statements that follow. For example:

```
default disable iff rst;
```

Semantic of `disable iff` has been changed to align with 2012 LRM. In IES 13.1 and earlier releases, assertion is disabled if the `disable iff` signal becomes active anytime between sampling and observed region. In IES 13.2, this behavior has been changed to use the value in the observed region for disabling the assertion.

Notes about `disable iff`:

- The `disable iff` clause is evaluated asynchronously--that is, independent of attempts to match the specified property.

- The value of the *boolean_expression* is its value in the current simulation cycle, not the sampled value.
- Per the IEEE 1800 SystemVerilog standard, nesting of `disable iff` clauses, either explicitly or through property instantiation, is not legal.
- The disable counter is incremented when an assertion transitions to the disabled state.

SVA Property Expressions

Property expressions can be of the following types:

```
{  sequence_property
   | implication_property
   | property_conjunction
   | property_disjunction
   | negation_property
   | until property
   | nexttime property
   | eventually property
   | always property
   | iff property
   | implies property
   | conditional_property
   | instance_property }
```

SVA Sequence Property

```
sequence_expression ...
```

The body of a sequence property contains sequence expressions.

For example, the following defines a property made up of two sequence expressions:

```
sequence Seq;
   (a ##1 b && c);
endsequence

property Prop;
   Seq[*3]##1(d && e);
endproperty
```

The SVA sequence property contains the following three forms:

- Strong
  Evaluates to true if, and only if, there is a nonempty match of the *sequence_expr* . An evaluation attempt of a strong (*sequence_expr*) in progress will be reported as FAILED at the end of simulation (EOS). All the evaluation attempts of a particular assertion in flight at EOS will be reported.
- Weak
  Evaluates to true if, and only if, there is no finite prefix that witnesses inability to match the *sequence_expr* . An evaluation attempt of a weak (*sequence_expr*) in progress will be reported as FINISHED at the end of simulation.

The following example is a valid expression:

```
assert property ( @(posedge clk) strong (a ##1 b) );
```

whereas, the following example is an invalid expression since strong/weak cannot operate upon *property_expr* :

```
assert property ( @(posedge clk) strong (a |=> b) );
```

- If the strong or weak operator is omitted, then the evaluation of the *sequence_expr* depends on the assertion statement in which it is used. If the assertion statement is assert property or assume property, then the *sequence_expr* is evaluated as weak (sequence_expr). Otherwise, the *sequence_expr* is evaluated as strong (sequence_expr).

  ```
  s_nexttime ( a |=> b )
  ```

Note: As per IEEE 1800-2012 LRM, starting 13.2 release, the default assertion evaluation semantics is weak. Use `assert_report_incompletes 1` to get strong semantics.

SVA Implication Property

```
sequence_expression { |-> | |=> } property_expression
```

An implication property specifies a sequence that is checked only when a certain precondition is satisfied.

The implication operators, `|->` and `|=>`, differ as follows:

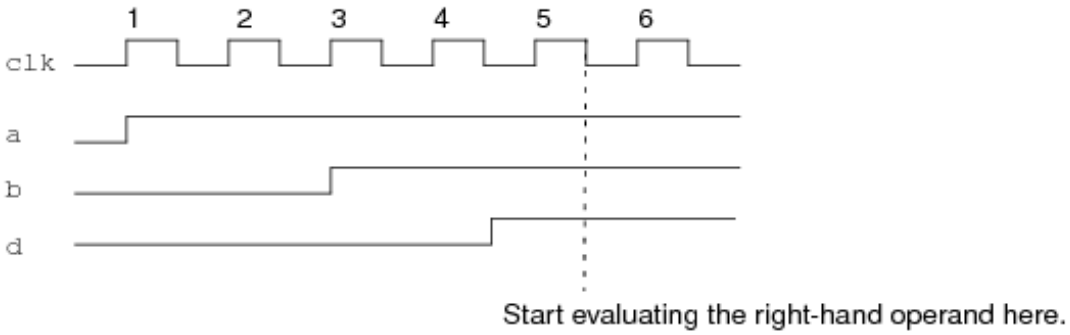
`|->`	If the left-hand operand matches the specified sequence, the endpoint of the match is the starting point for evaluating the right-hand operand.
`|=>`	If the left-hand operand matches the specified sequence, the starting point for evaluating the right-hand operand is one clock cycle after the endpoint of the match.

The following example defines a property named `P1`, which is triggered on the negative edge of `clk`. If `a` holds true for two cycles, and two cycles later `b` holds true for two cycles, then `d` must hold in the next clock cycle after `b` ends.

```
property P1;
   @(negedge clk)
    (a[*2] ##2 b[*2]) |=> (d);
endproperty
```

The `P2` example is similar, except that `d` is evaluated during the clock cycle in which the right-hand expression completes.

```
property P2;
    @(negedge clk)
    (a[*2] ##2 b[*2]) |-> (d);
endproperty
```



You can nest implication operators to construct complex sequential properties.

The `P4` example uses nested implication operators. It states that if `a` is followed by `b`, then in the next clock cycle, `c` or `d` must hold, followed by `d` or `e`. In the next clock cycle, if `f` or `e` holds, `g` must hold in the same clock cycle.

```
property P4;
    @(negedge clk)
    a ##1 b |=> (c || d) ##1 (d || e) |=> (f || e) |-> g ;
endproperty
```

SVA followed_by Property

*sequence_expression { **#-#** | **#=#** } property_expression*

For the followed-by property to succeed, the following conditions must be met:

- From a given start point `sequence_expression` should have at least one successful match.
- Starting from the end point of some successful match of `sequence_expression`, `property_expression` shall be successfully evaluated.

From a given start point, evaluation of the followed-by property succeeds and returns true if, and only if, there is a match of the antecedent `sequence_expression` beginning at the start point, and the evaluation of the consequent `property_expresion` beginning at the end point of the match succeeds and returns true.

These variants are classified based on:
Overlapping and non overlapping operators

Overlapping form

`sequence_expression` **`#-#`** `property_expression`

In overlapping form, the evaluation of consequent `property_expression` starts in the same cycle where the antecedent `sequence_expression` matches.

Non-overlapping form

`sequence_expression` **`#=#`** `property_expression`

In non-overlapping form, the evaluation of consequent `property_expression` starts a cycle after the antecedent `sequence_expression` matches.

The followed-by operators are the duals of the implication operators, "`|->`" and "`|=>`". Therefore, `sequence_expression #-# property_expresion` is equivalent to the following:

```
not (sequence_expression |-> not property_expression)
```

While `sequence_expression #=# property_expression` is equivalent to the following:

```
not (sequence_expression |=> not property_expression)
```

The following example defines a property named `P1`, where `done` shall be asserted at some clock tick during the first 6 clock ticks, and `rst` shall always be low starting from one of the clock ticks when `done` is asserted.

```
property p1;
    ##[0:5] done #-# always !rst;
endproperty
```

The `P2` example is similar, except that `done` shall be asserted at some clock tick during the first 6 clock ticks, and `rst` shall always be low starting from the clock tick after one of the clock ticks when `done` is asserted.

```
property p2;
    ##[0:5] done #=# always !rst;
endproperty
```

SVA Property Conjunction

```
property_expression1 and  property_expression2
```

A property evaluates to true if both *property_expression1* and *property_expression2* evaluate to true.

SVA Property Disjunction

```
property_expression1 or  property_expression2
```

A property evaluates to true if either *property_expression1* or *property_expression2* evaluate to true.

SVA Property Negation

```
notproperty_expression
```

A negation property defines a condition that must not occur. If the expression being checked holds, the negation property returns false. If it does not hold, the property returns true.

Do not negate an implication property, because it will fail when the property passes vacuously.

This example defines a property, N1, which returns false if a and rst are `1` at the same time:

```
property N1(a);
   @(posedge clk)
   not (a && rst);
endproperty
```

SVA Until Property

```
property_expression1 { until
|s_until
|until_with
|s_until_with } property_expression2
```

These variants are classified based on:

- Overlapping and nonoverlapping operators
- Weak and Strong operators

Weak Non overlapping form

```
property_expr1 until property_expr2
```

The weak non overlapping form of until evaluates to true if and only if, `property_expr1' evaluates to true at every clock tick beginning with starting clock tick of evaluation attempt and continue until atleast a tick before a clock tick where `property_expr2' is true. If `property_expr2' is true at starting clock tick of evaluation attempt, *`property_expr1'* need not be true at that clock tick.

Weak Overlapping form

```
property_expr1 until_with property_expr2
```

The weak overlapping form of until evaluates to true if and only if, `property_expr1' evaluates to true at every clock tick beginning with starting clock tick of evaluation attempt and continue until and including a clock tick where *`property_expr2'* is true.

Strong overlapping form

```
property_expr1 s_until property_expr2
```

The strong non overlapping form of until requires that a current or future clock tick must exist at which *property_expr2* evaluates to true.

Note: The default assertion semantics in 13.1 is still strong:

- As per 1800-2009, default semantics should be weak.
- For the operators which will have support for strong variant, the other variant will be treated as weak.

SVA Nexttime Property

```
{ nexttime
|s_nexttime } [ constant_expression ] property_expression2
```

Weak nexttime

```
nexttime property_expr
```

The weak nexttime property evaluates to true if, and only if, either the *property_expr* evaluates to true beginning at the next clock tick, or there is no further clock tick.

Indexed form of weak nexttime

```
nexttime [ constant_expression ] property_expr
```

The indexed weak nexttime property evaluates to true if, and only if, either there are not *constant_expression* clock ticks or *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

Strong nexttime

```
s_nexttime property_expr
```

The strong nexttime property evaluates to true if, and only if, there exists a next clock tick and *property_expr* evaluates to true beginning at that clock tick.

Indexed form of strong nexttime

```
s_nexttime [ constant_expression ] property_expr
```

The indexed strong nexttime property evaluates to true if, and only if, there exists *constant_expression* clock ticks and *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

Note: The number of clock ticks given by *constant_expression* shall be a non-negative integer constant expression.

Note: The default assertion semantics in 13.1 is still strong:

- As per IEEE 1800-2009 LRM, default semantics should be weak.
- For the operators which will have support for strong variant, the other variant will be treated as weak.

SVA Abort Property

```
accept_on ( expression_or_dist ) property_expression
reject_on ( expression_or_dist ) property_expression
sync_accept_on ( expression_or_dist ) property_expression
sync_reject_on ( expression_or_dist ) property_expression
```

The abort property operators are used to specify abort or exit condition for the evaluation of any `property_expression`. Using different abort operators, the result of terminated evaluation is specified as success or failure.

These variants are classified based on:

Asynchronous and synchronous abort properties

Asynchronous Abort Properties

accept_on (expression) property_expression

To evaluate `accept_on`, the underlying `property_expression` is evaluated. If during the evaluation process, the abort condition becomes true, then the overall evaluation of the property also evaluates to true. Otherwise, the overall evaluation of the property is equal to the evaluation of the underlying `property_expression`.

reject_on (expression) property_expression

To evaluate `reject_on`, the underlying `property_expression` is evaluated. If during the evaluation process, the abort condition becomes true, then the overall evaluation of the property evaluates to

false. Otherwise, the overall evaluation of the property is equal to the evaluation of the underlying `property_expression`.

Note:

- Similar to `disable iff` clause, the operators `accept_on` and `reject_on` are evaluated at every simulation time step.
- Their abort condition is evaluated using sampled value as a regular Boolean expression in assertions, unlike `disable iff` which uses current value. The abort operators `accept_on` and `reject_on` represent asynchronous reset.
- Evaluation attempt of abort operators `accept_on` and `reject_on` is nonvacuous if the evaluation attempt of `property_expression` is nonvacuous and the `expression_or_dist` abort condition evaluates to zero.
- In both the asynchronous abort properties, the abort condition takes precedence when the abort condition occurs at the same time step where the evaluation of `property_expression` ends.

Synchronous Abort Properties

`sync_accept_on` `(expression) property_expression`

To evaluate `sync_accept_on`, the underlying `property_expression` is evaluated. If during the evaluation process, the abort condition becomes true, then the overall evaluation of the property evaluates to true. Otherwise, the overall evaluation of the property is equal to the evaluation of the underlying `property_expression`.

`sync_reject_on` `(expression) property_expression`

To evaluate `sync_reject_on`, the underlying `property_expression` is evaluated. If during the evaluation process, the abort condition becomes true, then the overall evaluation of the property evaluates to false. Otherwise, the overall evaluation of the property is equal to the evaluation of the underlying `property_expression`.

Note:

- Unlike `disable iff` clause, `accept_on`, and `reject_on`, the operators `sync_accept_on` and `sync_reject_on` are evaluated at the simulation time step when the clocking event happens.
- Their abort condition is evaluated using sampled value as is done for `accept_on` and `reject_on,` unlike `disable iff` which uses current value. The operators `sync_accept_on` and `sync_reject_on` represent synchronous reset.

- In both the synchronous abort properties, the abort condition takes precedence when the abort condition occurs at the same time step where the evaluation of `property_expression` ends.

Abort properties (synchronous or asynchronous) affects single clock cycle or Boolean properties when the abort condition evaluates to true at the same clock tick when the `property_expression` is evaluated.

Any nesting of abort operators is allowed and these nested operators are evaluated in the lexical order.

SVA Eventually Property

```
{ eventually [ constant_expression ] property_expression1
s_eventually } [ cycle_delay_const_range_expression ]
property_expression2
```

Strong eventually

```
s_eventually cycle_delay_const_range_expression
```

A property *s_eventually [cycle_delay_const_range_expression] property_expr* evaluates to true if, and only if, there exists a current or future clock tick within the range specified by `cycle_delay_const_range_expression` at which *property_expr* evaluates to true. The range for a strong eventually may be unbounded.

A strong form of eventually shall report FAILURE at the end of simulation (EOS) if it does not conclude its evaluation till then. Strong eventually can also be classified in indexed and non-indexed form as weak forml.

Ranged form of weak eventually

```
eventually [ constant_range ] property_expr
```

A property*eventually [constant_range] property_expr* evaluates to true if, and only if, either there exists a current or future clock tick within the range specified by constant_range at which *property_expr* evaluates to true or not all the current or future clock ticks within the range specified by constant_range exist. The range for a weak eventually shall be bounded.

A ranged form of *eventually*, *s_eventually[n:m]* specifies the range of `n+1` to `m+1` clock ticks, where counting starts at the current time step.

Note: The strong variant *s_eventually* cannot be applied to any property expression that instantiates a recursive property [IEEE 1800-2012 LRM Section 16.13.17].

Ranged form of strong eventually

```
s_eventually [ cycle_delay_const_range_expression ] property_expr
```

A property *s_eventually [cycle_delay_const_range_expression] property_expr* evaluates to true if, and only if, there exists a current or future clock tick within the range specified by *cycle_delay_const_range_expression* at which *property_expr* evaluates to true. The range for a strong eventually may be unbounded.

SVA Always Property

```
always property_expr

always [ cycle_delay_const_range_expression ] property_expr

s_always [ constant_range] property_expr
```

Weak always

```
always  property_expr
```

A property **always** *property_expr* evaluates to true if, and only if, *property_expr* holds at every current or future clock tick.

Ranged form of weak always

```
always  [ cycle_delay_const_range_expression ]  property_expr
```

A property **always** [*cycle_delay_const_range_expression*] *property_expr* evaluates to true if and only if, *property_expr* holds at every current or future clock tick that is within the range of clock ticks specified by *cycle_delay_const_range_expression* . It is not required that all clock ticks within this range exist.

Strong always

```
s_always  [ constant_range] property_expr
```

A property **s_always** [*constant_range*] *property_expr* evaluates to true if, and only if, all the current or future clock ticks specified by *constant_range* exist and *property_expr* holds at each of these clock ticks.

SVA Iff Property

```
property_expr1 iff property_expr2
```

A property evaluates to true if and only if, either both *property_expr1* and *property_expr2* evaluate to false or both *property_expr1* and *property_expr2* evaluate to true.

SVA Implies Property

```
property_expression1 { implies
_____________________________________} property_expression2
```

A property is an implies if it has the following form:

```
property_expr1 implies property_expr2
```

A property of this form evaluates to true if, and only if, either *property_expr1* evaluates to false or *property_expr2* evaluates to true.

A property expression used in implies operator can also contain an implies operator within
`property_expr1 implies property_expr2 implies property_expr3`

Example:

```
reg a=1, b=0, c=1, d=0, clk=0;
initial #4 b = 1 ;
              d = 1;
#7 d = 0;
always #5 clk= ~clk ;
 A1: assert property ( @(clk) a ##1 b implies c ##1 d );
Result:FINISHED @10 (2 cycles, starting at 5 NS)
```

SVA Conditional Property

```
 if(expression)
  property_expression
[ else property_expression ]
```

The `if`, `else if`, and `else` constructs return true if a value holds when a given condition is met.

In this example, property `P1` uses a nested `if` statement. This property checks `b` if `a` is true. If `b` is

true, `c` is checked one clock cycle later. If `c` is true, the assertion passes if `d` is true and, one clock cycle later, `e` is true.

```
property P1;
    @(negedge clk)
        if(a) (b |=> if (c) (d |=> e));
endproperty
```

Property `P2` uses an `if/else` statement. Like `P1`, this property checks `(b |=> c)` if `a` is true. If `a` is not true, the property checks `(d |=> e)`.

```
property P2;
    @(negedge clk)
        if(a) (b |=> c)
        else (d |=> e);
endproperty
```

Property `P3` uses the `if/else` if statement to check different values, depending upon the value of the `rst_count` variable.

```
property P3;
    @(negedge clk)
         if(rst_count <= 0)
             (a&&(b||c) |=> d ##[1:2] e)
         else if (rst_count == 1)
             (e |-> f ##[1:2] g)
         else if(rst_count == 2)
             ((a&&b&&c) |-> ##[0:$] (d&&e&&f));
endproperty
```

SVA Instance Property

```
identifier [ ( argument_list ) ]
```

An instance property instantiates a property within another property or assertion. If the property declaration has an argument list, the formal arguments are instantiated with the actual arguments in the instance property.

For example, property `P` specifies that if `a` or `b` is true, then `c` must be true in the same clock cycle. The `I1` assertion instantiates `P` and specifies that if `a` is `0`, then `P` must hold in the next clock cycle:

```
property P;
```

```
    @(posedge clk) (a || b) |-> c;
endproperty

I1: assert property (@(posedge clk) !a |=> P);
```

SVA Recursive Property

A recursive property is a named property that includes instantiation of itself. Recursion provides a flexible framework for coding properties to serve as ongoing assertions, assumptions, or coverage monitors.

Example 1:

The following code illustrates a recursive property.

```
property prop_always(p);
    p and (1'b1 |=> prop_always(p));
endproperty
```

In the given recursive property, if the formal argument `p` holds, then recursively, in the next cycle `p` must also hold. In summary, `p` must hold at every cycle.

Example 2:

The given code illustrates a mutually recursive property.

```
property x;
    1'b1 |=> y;
endproperty

property y;
    1'b1 |=> x;
endproperty
```

Notes about recursive property:

- In the recursive property, all the restrictions apply as per the LRM, except the restrictions related to handling of local variables as formal arguments. The local variables as formal arguments of properties are not supported.
- Recursive properties are supported inside a module, interface, and package. However, recursive properties are not supported in program, clocking blocks, and compilation-unit scope.

Supporting Hierarchical Reference to SVA Properties and Sequences

In an assertion, you can hierarchically reference (out-of-module reference) any of the following:

- Property
- Sequence
- Sequence method

This means that the scope of referenced SystemVerilog assertion sequence, property, or sequence method may be different from the scope where it is actually used.

Following are the different scopes where out-of-module reference (OOMR) to assertion properties or assertion sequences is supported:

- Module
- Program
- Interface (Excluding virtual interface)

This usage is explained with the help of few examples:

Example 1: The following example demonstrates how sequence **s1** defined in module **test** is used in module **top**. The hierarchical path **itest.s1** in assertion A1 within module top points to sequence **s1** defined in module test.

```
module top;

    reg clk = 0, a=1,b=0;

    test itest();

    A1:assert property (@(posedge clk) itest.s1(a,b));

endmodule

module test;

    sequence s1(x,y);

        x ##1 y;

    endsequence

endmodule
```

Example 2: The following example demonstrates how property **p1** defined in interface **inf** is used in module **top**. The hierarchical path **inf_i.p1** in property **tp1** within module **top** points to property **p1** defined in interface **inf**.

```
module top;

     …
```

```
    inf inf_i();
    property tp1;
        disable iff (~reset) inf_i.p1 ('h00, ctl);
    endproperty
    assert property (@(clk) tp1);
endmodule
    interface inf;
        property p1(a,b);
           a ##1 b;
        endproperty
    endinterface
```

Example 3: The following example demonstrates the use of sequence method in procedural block and in if generate block.

```
module top;
   …
   test itest();
   always
   begin
       #2 wait(itest.s2(a).triggered);
       #1 $display($time, " %m hierarchical referenced sequence method s2 success in
       always");
   end
   generate
       if(i>5)
           begin: gen_blk1
               assert #0(itest.s1.triggered);
           end
   endgenerate
```

```
endmodule
```

Example 4: The following example below demonstrates how SV property defined in for-generate block within a module test is referenced using hierarchical path **itest.loop[1].p1** from module top.

```
module top;

    …

    test itest();

    A1: assert property(itest.loop[1].p1);

endmodule

module test();

    …

    genvar i;

       generate

         for (i=0; i<4; i=i+1)

           begin : loop

              property p1;

                  @(posedge clk) 1'b1 |-> a ;

              endproperty

           end

       endgenerate

endmodule
```

Limitations:

- An assertion within a procedural block, that references a property or a sequence defined in an out-of-module scope is not supported.
- Any reference to compilation unit scope (cu-scope) variable, let construct, and user-defined data types in hierarchical referenced properties or sequences is not supported.
- The failure message for assertions containing hierarchical reference to SVA properties or sequences, does not point to the exact source code in assertion logging message.

SVA Directives

Directives specify how an assertion is used during the verification process--as a behavior to be checked, or for collecting coverage information. Only properties with a verification directive are evaluated. The evaluation result will be one of the following:

- Finished
- Failed
- Vacuous
- Disabled

The SVA assert Directive

```
[label: ] assert property( property_expression )[action_block];
```

Specifies a property to be checked during simulation, and any actions to take if the property passes or fails. You can use the `assert` directive to declare an immediate assertion; use the assert property directive to declare a concurrent assertion.

In this example, property `p3` is declared as a concurrent assertion:

```
property p3 ; @(posedge d) (a ##2 b); endproperty
assert property (p3) else -> error_detected;
```

The SVA assume Directive

```
[label: ] assume property( property_expression )[action_block];
```

Specifies a property that must hold true throughout verification, and any actions to take if the property passes or fails. When used in formal analysis, the `assume` function constrains the conditions that are proven. When used in simulation, the function is used to check that only valid stimuli are tested.

In this example, property `P3` defines an assumption that, at every positive edge of `D`, `A` will be true, and two clock ticks later, `B` will be true:

```
property p3 ; @(posedge D) (A ##2 B); endproperty
p3_label: assume property (p3);
```

The SVA cover Directives

```
[ label: ] cover property( property_spec ) [pass_statement];

[ label: ] cover sequence([disable iff (expr)]
                          seq_expression) [pass_statement] ;
```

Specifies a property or sequence to be monitored during simulation; the results are used as coverage information. The `cover` statement can optionally specify a pass statement--an action to take when the property or sequence passes. The pass statement cannot include any concurrent `assert`, `assume`, or `cover` statements.

In this example, property `C1` is used to generate coverage information:

```
property C1;
@(posedge clk)
a |=> b ##2 c;
endproperty
cover property (C1) $display("Covering C1");
```

The `cover property` directive counts at most one match of the property for each evaluation attempt. The `cover sequence` directive counts all matches of the sequence, rather than one per attempt. The counts for `cover property` and `cover sequence` will differ for a variable-length sequence. For example:

```
cover sequence (a ##1 b[*1:3] ##1 c);
```

For any one attempt, `cover property` will count only the first match, but `cover sequence` can result in up to three matches for the sequence.

Note: The `cover sequence` directive is not currently part of the IEEE 1800-2005 standard for SystemVerilog, but is proposed for a future release.

The SVA expect Statement

```
expect ( property_expression )[ action_block ];
```

Specifies a procedural blocking statement that lets you wait on a property evaluation. The next statement is not evaluated until the expected property passes or fails. The structure of the `expect` statement is like an `assert` statement, because you can include an action block.

The `expect` statement can be located in any procedural blocking statement, including tasks and

class methods.

Note: The `expect` statement can refer only to static variables--automatic variables are not supported.

The statement following the `expect` statement is executed after the Observed region in which the property completes its evaluation is processed. When the property succeeds or fails, the process unblocks, and the property stops being evaluated--that is, no property evaluation is started until that expect statement is executed again.

If the property fails at its clocking event, the optional `else` clause of the action block is executed. If it succeeds, the optional pass statement is executed.

You cannot use the `expect` statement within an action block.

The following example shows how the `expect` action block can be used:

```
initial begin
    expect ((@ ... ) ##[1:10] data == value)
        $info ("Value occurred. Continuing with test.");
    else
        begin
            $error("Value did not occur within 10 clock cycles.");
            $finish;
        end
end
```

The `expect` statement can be used to construct reactive tests. For details, see "SVA Reactive Tests using the expect Statement".

Note: Because the `expect` statement can fail, it is treated like an assertion in the Incisive simulator--it appears in the Assertion Browser and assertion reports, and it increments finished and failed counters in the coverage database.

SVA Action Blocks

```
[ pass_statement ] else fail_statement
```

An action block defines the actions to take when an assertion passes or fails.

The action blocks of concurrent assertions can contain sampled value functions ("SVA Sampled Value Functions"), but the action blocks of immediate assertions cannot.

Note: The Incisive simulator does not support delays in action blocks, including the action block of

the `expect` statement.

The following is an example of an action block. This assertion displays the message `A1 passes` when the property passes, and `A1 fails` when the property fails.

```
A1: assert property (@(posedge clk) a || b -> c || d)
        $display("%m passes");
        else $error("%m fails");
```

Note: The *1800 IEEE Standard for SystemVerilog* specifies that the pass statement runs when a vacuous pass occurs, as well as when an assertion finishes. By default, the Incisive simulator runs the pass statement only when the status is finished, not when a vacuous pass occurs. Use the -strict option in Running an Assertion Simulation of *Assertion Checking in Simulation* to execute the pass statement when a vacuous pass occurs.

Known Limitation

If an assertion contains an action block that has a message display function, a message is reported as a result of the execution of the action block. This message is displayed in addition to the internal message of completion that is reported from the tool. However, in some scenarios, these messages from the action blocks are not reported alongside the internal messages from the tool.

SVA Severity System Tasks

The severity system tasks are used in the pass or fail action block to report the severity of an error detected by an assertion. When an assertion calls one of these tasks, the simulator displays a message, as described in "Understanding Simulator Error Messages" in *Assertion Checking in Simulation*. You can also specify a message to display when the task is called. This message is displayed on a separate line below the message returned by the task.

Note:

- The behavior of Severity System Tasks in pass action block will be the same as that in fail action block in both, batch mode as well as interactive mode.
- Immediate assertions (simple and deferred) would have similar expected behavior as for concurrent assertions.

SVA $fatal System Task

```
$fatal ( level, message[ , args ])
```

Returns a message with an `F` severity level (fatal), and stops the simulation. With the support in pass block, executed even when the assertion passes. Also, twice mentioned that it stops the simulation.

A call to `$fatal` implicitly calls the Verilog `$finish` system task. As for the `$finish` task, you can specify a *level* of 0, 1, or 2, where

- `0` prints nothing
- `1` prints the simulation time
- `2` prints the simulation time plus memory and CPU usage statistics

Note: If you include a *`message`* argument, the *`level`* argument is required. Otherwise, the *`level`* argument is optional and a simple `$fatal()` may be used.

The *`message`* argument is an optional string to be displayed when the task is called. The message can contain format strings, similar to the `$display` function. The %m does not return the name of the assertion, but returns the scope from which the task is called .

For example, this assertion checks that `c` and `d` are high at the same time:

```
assert_block: assert (c & d) $display ("PASS STMT");
    else begin $fatal (0,"%m failed %b",c); end
```

If the assertion fails, simulation ends, and the following messages are displayed:

```
ncsim: *F,ASRTST (./test.v,87): (time 0 NS) Assertion test.assert_block_2 has failed
test.assert_block failed 1
```

The first message is returned by the simulator. The second message is specified in the call to `$fatal`.

SVA $error System Task

```
$error (message[ , args ])
```

Returns a simulator message with an `E` severity level (error). Whether simulation continues

depends on other configuration settings. By default, it stops when you are running the simulation in interactive mode and continues in non-interactive mode.

For example, this assertion fails and calls the `$error` severity task if `c` or `d` is low:

```
assert_block: assert (c & d) $display ("PASS STMT");
    else begin $error ("%m failed, c = %b",c); end
```

The following message is displayed when `c`=1, which indicates that `d` caused the assertion to fail:

```
ncsim: *E,ASRTST (./test.v,87): (time 0 NS) Assertion test.assert_block has failed
test.assert_block failed, c = 1
```

The simulation error count is optionally incremented. For details, see "assertion -logging" in *Assertion Checking in Simulation*.

SVA $warning System Task

$warning (*message*[,*args*])

Returns a simulator message with a `W` severity level (warning).This severity task does not increment the simulation error count.

For example, this assertion calls the `$warning` severity task if `c` or `d` is low:

```
assert_block: assert (c & d) $display ("PASS STMT");
    else begin $warning ("%m failed %b",c); end
```

If the assertion fails, the following message is displayed:

```
ncsim: *W,ASRTST (./test.v,87): (time 0 NS) Assertion test.assert_block has failed
test.assert_block failed 1
```

SVA $info System Task

$info (*message*[,*args*])

Returns a simulator message with an `N` severity level (note). This severity task does not increment the simulation error count.

For example, this assertion calls the `$info` severity task if either `c` or `d` is low:

```
assert_block: assert (c & d) $display ("PASS STMT");
    else begin $info (0,"%m failed %b",c); end
```

If the assertion fails, the following message is displayed:

```
ncsim: *N,ASRTST (./test.v,87): (time 0 NS) Assertion test.assert_block has failed
test.assert_block failed 1
```

SVA Sampled Value Functions

The sampled-value functions are used to access sampled values of an expression either at the current clock cycle or at a specified number of cycles in the past. You can also use sampled-value functions to detect changes in sampled value of an expression.

You can use sampled-value functions in a sequence or a property expression, and in the action blocks of concurrent assertions. In addition to using sampled-value functions in assertions, you can also use them in your SystemVerilog procedural code. For details, see "Sampled Value Functions in Procedural Blocks" in the *SystemVerilog Reference*.

When you use sampled-values in assertions, each assertion must specify a clock expression to be used by the sampled-value function. This clock expression can be an assertion clock, a default clock, or an explicit clock. The explicit clock can be specified as an argument to the sampled-value function. It is the 4th argument to `$past` and the 2nd argument to `$stable`, `$changed`, `$rose`, and `$fell`. As an exception, $sampled does not have a clocking event.

The following Sample Value Functions are supported in IES:

```
$sampled ( expression )
$rose ( expression [, [clocking_event] ] )
$fell ( expression [, [clocking_event] ] )
$stable ( expression [, [clocking_event] ] )
$changed ( expression [ , [ clocking_event ] ] )
$past ( expression [, [number_of_ticks ] [, [expression2 ] [, [clocking_event]]] ] )
```

The use of these functions is not limited to assertion features; they may be used as expressions in procedural code as well. All variables referenced in the actual argument expressions passed to these functions shall be static. The clocking event, although optional as an explicit argument to the functions, `$past`, `$rose`, `$stable`, `$changed`, and `$fell`, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The following rules are used to infer the clocking event:

- If called in an assertion, the appropriate clocking event from the assertion is used.
- If called in an action block of a singly clocked assertion, the clock of the assertion is used.
- If called in an action block of a multiclocked assertion, the leading clock of the assertion is

used.

- If called in a procedural block, the inferred clock, if any, for the procedural code is used.

$sampled

```
$sampled(expression)
```

Returns the sampled value of an expression at the current clock cycle.

The function `$sampled` does not use a clocking event.

For a sampled value function other than `$sampled`, the clocking event is explicitly specified as an argument or inferred from the code where the function is called.

$rose

```
$rose( expression[, clocking_event])]
```

Returns true if the least significant bit of an expression is changed to `1`. Otherwise, it returns false.

$fell

```
$fell( expression[, clocking_event])]
```

Returns true if the least significant bit of an expression is changed to `0`. Otherwise, it returns false.

$stable

```
$stable( expression[, clocking_event])]
```

Returns true if the value of an expression did not change during the current clock cycle.

$changed

```
$changed( expression[, clocking_event])]
```

Returns true if the value of an expression is changed. Otherwise, it returns false.

$past

```
$past( expression [, [number_of_ticks ] [, [expression2 ] [,
[clocking_event]]] ])
```

Returns the sampled value of an expression at a previous time.

- *expression* specifies the object whose value you want to obtain. This expression cannot contain dynamic objects, unpacked structures, and unpacked unions. Also, expression cannot contain variables or index variables in bit or part select expressions.
- *number_of_ticks* is an optional number of clock cycles in the past. This value can be an integer literal, a `generate` statement index variable, or a generic, mathematical, or parameter expression. If you do not specify *number_of_ticks* , the function returns the value at the previous clock cycle.
- *expression2* is used as a gating expression for the *clocking event* .
- *clocking_event* specifies the clocking event for sampling *expression* .

Limitations of Sampled Value Functions:

The support of sampled-value functions in Incisive Enterprise Simulator is as per Section 16.9.3 of the SV 1800-2009 LRM but with the following limitations. You cannot use sampled-value functions:

- In the action blocks of immediate assertions
- On variables defined in the scope of `always/initial' blocks
- Inside event expressions
- Inside a generate, where a default clock is inferred for sampled-value functions used in always_comb/always_latch
- With unpacked structures and OOMRs to unpacked arrays as arguments
- With unpacked arrays as arguments in nested sampled-value functions
- With unpacked arrays as argument in nested sampled-value functions, where inner SVF is

$past or $sampled(), $SVF($past(unpack-arr))

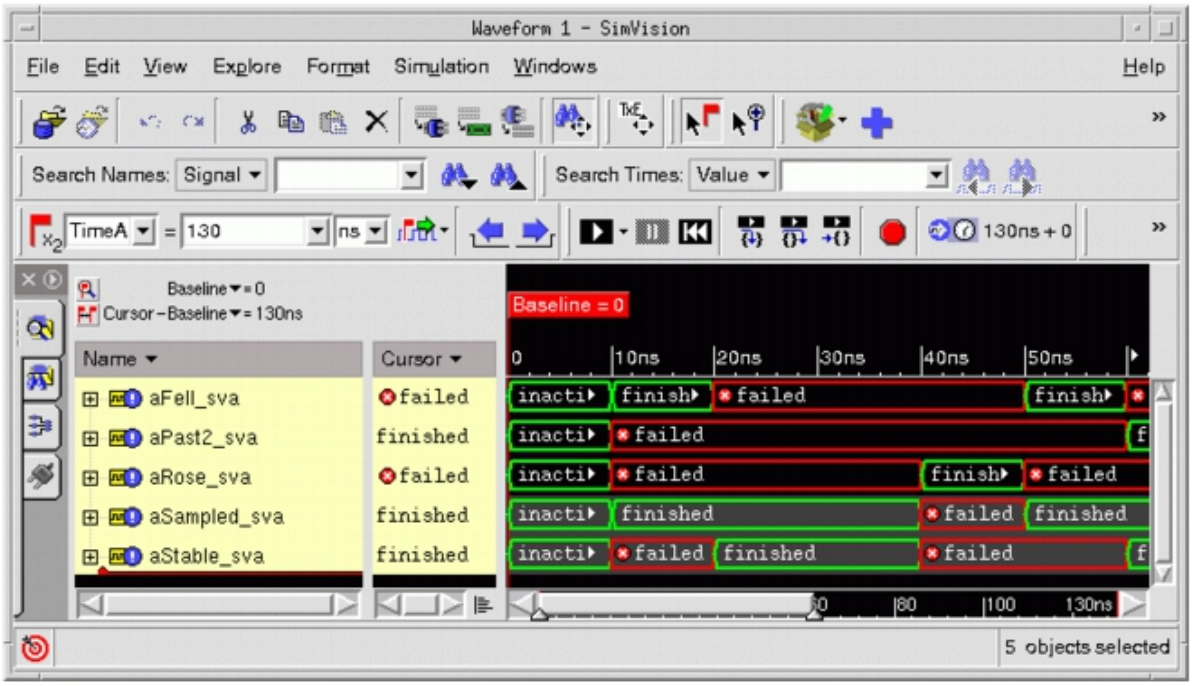
- With unpacked array as argument in $past when parameter is used to define `number_of_ticks`.
- With OOMR argument in nested $past
- Inside classes

Examples of SVA Sampled-Value Functions

Figure 5-1 shows the waveforms generated by the following assertions that use the built-in `$sampled`, `$rose`, `$fell`, `$stable`, and `$past` functions:

```
aFell_sva: assert property ( @(posedge clk) ($fell(a)));
aPast2_sva: assert property ( @(posedge clk) ($past(a, 2) == 1));
aRose_sva: assert property ( @(posedge clk) ($rose(a)));
aSampled_sva: assert property ( @(posedge clk) ($sampled(a)));
aStable_sva: assert property ( @(posedge clk) ($stable(a)));
```

Figure 5-1 Sampled-Value Functions



Bit-Vector System Functions

You can call system functions in assertions in any Boolean expression to perform common tests of signal values, such as whether only one bit of a bit vector has the value `1`.

The IEEE 1800-2012 LRM extends the parameter types of the bit-vector system functions to be bit-stream types instead of just bit or bit-vectors. This is supported in IES.

$onehot

```
$onehot( expression )
```

Evaluates a bit vector and returns true if only one bit is high. Otherwise, the function returns false.

The following example defines a sequence, `S3_4`, which states that whenever `a` and `b` are `0` and `c` is `1`, then `a` and `c` must be `0` and `b` must be `1` in the next clock cycle. The `P34_onehot_ABCD` property calls the `$onehot` system function when `S3_4` is detected, checking that, in the next clock cycle, only one bit in `d` has the value `1`.

```
sequence s3_4; (!a && !b && c) ##1 (!a && b && !c); endsequence
property P34_onehot_ABCD;
    @(posedge clk) (s3_4 |=> ##1 $onehot(d));
endproperty
```

$onehot0

```
$onehot0( expression )
```

Evaluates a bit vector and returns true if zero or one bit is high. Otherwise, the function returns false.

This example defines a sequence, `S3_4`, which states that whenever `a` and `b` are `0` and `C` is `1`, then `a` and `c` must be `0` and `b` must be `1` in the next clock cycle. The `P34_onehot0_ABCD` property calls the `$onehot0` system function when `S3_4` is detected, checking that in the next clock cycle, either all of the bits in `d` have the value `0`, or only one bit has the value `1`.

```
sequence s3_4; (!a && !b && c) ##1 (!a && b && !c); endsequence
property P34_onehot0_ABCD;
    @(posedge clk) (s3_4 |-> ##1 $onehot(d));
endproperty
```

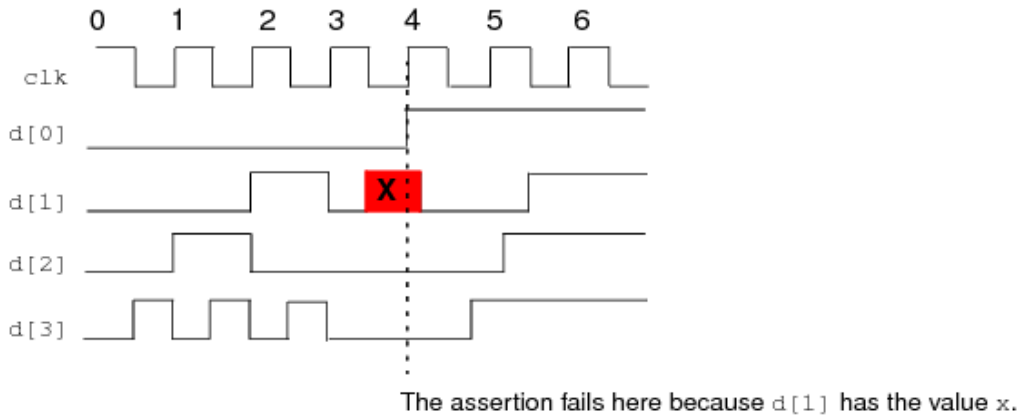
$isunknown

```
$isunknown( expression )
```

Evaluates a bit vector and returns true if any bit is `x` or `z`. Otherwise, the function returns false.

In this example, the `A_unknown` assertion calls `$isunknown` at each positive edge of `clk`:

```
A_unknown: assert property (@(posedge clk) $isunknown(d));
```



The assertion fails here because `d[1]` has the value `x`.

$countones

```
$countones( expression )
```

Evaluates a bit vector and returns the number of bits whose value is `1`; `x` and `z` values are not counted.

For example, this assertion counts the number of ones in vector `d`:

```
A_countones: assert property (@(posedge clk) $countones(d) > 0)
   $display("Found ones");
   else $error("Did not find ones");
```

Limitation:

- Support of streaming concatenation operators({>>{sig}}) in Assertion context and SVFs.
- Support of System verilog logical equality operators(==, !=) in assertion context, where operands are 'unpacked arrays' or 'unpacked structures'.

Assertion Control System Tasks

```
$ asrt_ctrl_task[( level[, list_of_modules_or_assertions])];
```

The assertion control system tasks allow the HDL to control assertion checking from simulation. Arguments are interpreted in the same way as for `$dumpvars`. If an assertion control task is used without arguments, it affects all assertions in the hierarchy.

- *`asrt_ctrl_task`*

  `$asserton`, `$assertoff`, or `$assertkill`; these tasks are described individually in the sections that follow.

- *`level`*

  Specifies the number of hierarchy levels below each specified module instance that are affected. The level is interpreted relative to all tops in the design, regardless of where the task is located in the design, similar to the way `$dumpvars` works.

  Note: The 0 value can be used only if subsequent arguments specify module instances. Setting this value to 0 affects all assertions in the specified module, and all module instances below the specified module. It cannot be used to specify individual assertions.

- *`list_of_modules_or_assertions`*

  Specifies the scopes of the model to which the command applies. The *`list_of_modules_or_assertions`* argument can specify entire modules or individual assertions within a module. Instance names, including hierarchical instance names, are supported. Only hierarchical references to properties are allowed; references to sequences are not permitted.

  The following are not supported for this argument:
  - Out-of-module references to individual assertions
  - Wildcards

For example:

```
$assertoff (0, top.mod1);
```

Note: These tasks affect both SystemVerilog assertions and PSL assertions.

Assertion Control System Tasks Usage

You can use the assertion control system tasks to control

- Assertions local to the assertion control system task by using the assertion name
- All assertions under a hierarchy, by using the *level* argument and an instance name
- All assertions in the design, by using the *level* argument only
- All assertions in a module, by using the *list_of_modules_or_assertions* argument only to specify a module

Note: Assertion control system tasks have no effect on assertions in protected code.

$assertoff

```
$assertoff [( level[, list_of_modules_or_assertions])];
```

Suspends checking of all specified assertions until `$asserton` is encountered. An assertion that is already executing, including assertion action blocks, will continue executing.

$asserton

```
$asserton [( levels[, list_of_modules_or_assertions])];
```

Resumes checking of all specified assertions that were disabled by a previous `$assertoff`.

Note: You cannot turn on permanently switched off assertions. An assertion is permanently turned off in the following scenarios:

- When command line option "`-noassert`" is used at elaboration time.
- When cover's finish count reaches the value specified on using simulation command line switch, "`-abvfinishlimit`" or its tcl variant, "`-finish_limit`".
- When the total failure count from all the assertions reaches the value specified on using simulation command line switch, "`-abvglobalfailurelimit`" or its tcl variant, "`-global_failure_limit`".
- When the assertion's failure count reaches the value specified on using simulation command line switch, "`-abvfailurelimit`" or its tcl variant, "`-failure_limit`".
- When "`assertion -off -always`" tcl command or "`-abvoff`" command line option is used.
- By default for cover properties.

To turn on cover properties, enable it using simulation command line switch "`-abvcoveron`" or the elaboration command line switch, "`-coverage u`" or "`-coverage all`".

$assertkill

```
$assertkill [( levels[, list_of_modules_or_assertions])];
```

Halts checking of all specified assertions that are currently executing, then suspends checking of all specified assertions until `$asserton` is encountered.

Per the IEEE 1800 SystemVerilog standard, the `$assertkill` task has no effect on immediate assertions and non-temporal concurrent assertions, due to scheduling issues.

Alternatives to Assertion Control System Tasks

If these tasks do not meet your needs, one or more of the following approaches might work for you instead:

- Use the Tcl interface to control assertions
  - Individually
  - By directive
  - By hierarchy
    See "Enabling and Disabling Assertions" in "Running an Assertion Simulation," in *Assertion Checking in Simulation*.
- Use VPI controls. For details, see "SVA VPI Extensions".
- Use the Incisive Enterprise Simulator-XL SystemC assertion API, which allows test visibility and control of all assertions in the design, regardless of the language in which they are written.
  See "Access to Assertions from SystemC Testbench " in "Using SystemC PSL," of the *SystemC Simulation Reference*.

When you use multiple assertion-control approaches, the last command received is executed. For example, if you use `$assertoff` in your SystemVerilog code, then turn assertions on with a Tcl command during simulation, the assertions will be on, because the Tcl command was executed last.

SVA VPI Extensions

VPI is the Verilog programming interface to the simulator environment. It enables you to compile custom C routines with the VPI routine libraries and simulator core, to create a customized version of the simulator. The VPI routine libraries are defined by the IEEE 1364 standard. Extensions to these VPI routines for SystemVerilog assertions are defined in the IEEE 1800 standard. These extensions include access, callback, and utility routines that you call from your C programming language function. Some things that you can do with the VPI extensions include the following:

- Determine what assertions are contained in the design, where they are, and what they are
- Enable/disable/kill assertions in the design
- Register callbacks and associated routines to run when an assertion state changes
- Access UNIX utilities, do file I/O, and communicate with third-party tools and models.

Note: The following are not supported for immediate assertions, because they are combinatorial:

- Controls--`vpiAssertionReset` and `vpiAssertioKill`
- Callbacks--`cbAssertionReset` and `cbAssertionKill`

Several files in the installation are needed for the VPI SVA extensions. Include these files as headers in the file that contains custom VPI calls. All of these files are located in `` `ncroot`/tools/include ``.

- `vpi_user.h`

  Contains the constant definitions, structure definitions, and routine declarations and prototypes used by the access routines. You must include this header at the beginning of every file of C routines that use VPI calls.

- `sv_vpi_user.h`

  IEEE 1800 SystemVerilog VPI extensions. Contains the constant definitions, structure

definitions, and routine declarations and prototypes used by the access routines, as defined in the standard.

- `vpi_user_cds.h`

  Contains Cadence extensions to the Verilog HDL specification.

- `stdio.h`

  Contains the definition of NULL.

Note: A file named `vpi_abv_cds.h` in `` `ncroot`/tools/include `` contains non-standard, assertion-specific VPI routines for SVA and PSL. This file contains the constant definitions, and routine declarations and prototypes, used by the access routines. These definitions are not needed if you use `sv_vpi_user.h`.

There are three approaches to coding user routines. For more information, see "Creating a Debug Image" in the *Cadence VPI User Guide and Reference*. The three approaches include:

- Use the bootstrap function to dynamically load the library, by using the -LOADVPI *`libraryname:bootstrapname`* option to irun or ncsim, or *`+LOADVPI=libraryname:bootstrapname`* for ncverilog. The *`libraryname`* value is the name of the shared library containing the VPI application. The *`bootstrapname`* is the name of the function in that library that will register the VPI system tasks and functions. For example, in the following, `abv_vpi_test.c` contains the VPI code, and `test.v` contains the Verilog code; 32-bit ELF binaries are created:

  ```
  > gcc -c -g -fPIC -m32 -I. -I`ncroot`/tools/include
      -o abv_vpi_test.o abv_vpi_test.c

    > ld -shared -m elf_i386 -o abv_vpi_test.so abv_vpi_test.o

    > ncsim
  ```

  To create 64-bit ELF binaries, use `-m64` instead of `-m32` in the gcc command, and `-m elf_x86_64` in the ld command.

- You can modify and re-compile the vlog_startup_routines array in the dynamic shared library that you are going to load. This array is in the `vpi_user.c` file.

- When defining your registration routine, use `vpi_systtf_data_structure.tfname` to define the

name of the system task, which can then be called from Verilog HDL. The name must begin with a `$` sign.

The sections that follow discuss the custom VPI routines that you can create. Both dynamic and static VPI is supported. This chapter describes how to

- Get a list of all of the assertions in the design or scope of interest,
- Get static information about the assertion
- Set dynamic callbacks to specified changes on the assertion

It also discusses control actions that allow you to enable, disable, or kill assertions.

Listing Assertions in the Design

Generally, the first step is to get a list of the assertions. Assertions are identified by an *assertion handle*. Key assertion-related VPI routines include the following:

- *assertion_iterator* = vpi_iterate(vpiAssertion,scope);

  The *assertion_iterator* initializes to point to the first SVA assertion in the specified scope, because the type is `vpiAssertion`. A scope of 0 or NULL lists all SVA assertions in the design. The scope can also be an instance.

- *assertion_handle* = vpi_scan(assertion_iterator);

  The *assertion_handle* points to the next such item every time you call `vpi_scan`. Memory is allocated upon each call to `vpi_iterate`, and deallocated when the scan completes. Always free the iterator if you do not finish scanning the list. Never free or use the iterator if you do finish scanning the list--that is, if `vpi_scan` returned NULL.

- vpi_free_object(*assertion_iterator*);

  Frees the iterator *assertion_iterator*.

- vpi_handle_by_name(*assertName, scope*);

  Returns the handle for the named assertion.

For example, these simple commands provide the basis for doing an operation on all assertions in the design:

```
if ((assertion_iterator= vpi_iterate(vpiAssertion, 0)) != NULL )
    while ( (assertion_handle = vpi_scan(assertion_iterator) ) != NULL )
```

```
    vpi_printf ( "%s is an assertion.\n",vpi_get_str(vpiName,assertion_handle)
);
```

The IEEE 1800 standard uses object-definition diagrams to show:

- The simulation objects you can access
- The properties of those objects that can be accessed
- The relationships between objects

The properties and relationships between objects are often used for static RTL checks after elaboration, such as HAL checks.

Dynamic operations can also be performed during simulation. Dynamic operations consist of control actions for enabling, disabling, and killing assertions; and setting callbacks.

Obtaining Static Information about Assertions

You can obtain static information about assertions by using VPI. This information is useful for selecting assertions, and for such things as RTL checks on assertion constructs. The following static information is available:

- Assertion name
- Instance in which the assertion occurs
- Module definition containing the assertion
- Assertion type
  - Assert
  - Assume
  - Cover
  - Immediate Assertion
- Assertion source information--the file, line, and column where the assertion is defined
- Assertion clocking blocks and expressions

No value is returned for objects in protected code.

Syntax

```
vpi_get(vpiType, assertion_handle)
```

Returns the integer or Boolean of the specified property for the specified object.

Where `vpiType` can be:

`vpiAssert`

`vpiAssume`

`vpiCover`

`vpiImmediateAssert`

`vpiPropertyDecl`

`vpiSequenceDecl`

Example

Once you have a list of all assertions, you can filter them on a specified type:

```
if ((assertion_iterator= vpi_iterate(vpiAssertion, 0)) != NULL ) {
    while ( (assertion_handle = vpi_scan(assertion_iterator) ) != NULL ) {
     if (vpi_get(vpiType, assertion_handle) == vpiCover) {
         /* process cover type assertions */
     }
    }
}
```

Controlling Assertions through VPI

VPI can control both the assertion system and individual assertions.

VPI Assertion System Control

Use the `vpi_control` command to control all assertions in the simulation.

Syntax

```
vpi_control(control_reason, handle)
```

Where:

`control_reason` can be:

- `vpiAssertionSysReset`--Discards all attempts in progress for all assertions, and restores the

assertion system to its initial state. All assertion callbacks remain.

- `vpiAssertionSysOff`--Disables any new assertions from starting. Assertions that are already executing are not affected. Existing callbacks are not affected.
- `vpiAssertionSysKill`--Disables any new assertions from starting. Assertions that are already executing are terminated. Existing callbacks are not affected.
- `vpiAssertionSysOn`--Restarts the assertions system after it was stopped by `vpiAssertionSysOff` or `vpiAssertionSysEnd`.
- `vpiAssertionSysEnd`--Discards all attempts in progress, and disables any further assertions from starting. All assertion callbacks currently installed are removed. Once this control is issued, no further assertion-related actions are permitted.

`handle` can be:

- `vpiHandle` for a scope
- `vpiCollection` of handles for a list of scopes
- NULL handle signifies that the control applies to all assertions

VPI Individual Assertion Control

Use the `vpi_control` command to control individual assertions.

Syntax

```
vpi_control(control_reason, assertion_handle, attemptStartTime)
```

where:

- *`control_reason`* can be:
  - `vpiAssertionReset`--Discards all attempts in progress for this assertion, and resets this assertion to its initial state. The `attemptStartTime` argument is invalid.
  - `vpiAssertionDisable`--Disables the starting of new attempts for this assertion. The `attemptStartTime` argument is invalid.
  - `vpiAssertionEnable`--Enables the starting of new attempts for this assertion. The

`attemptStartTime` argument is invalid.

- `vpiAssertionKill`--Discards the attempt that started at `attemptStartTime`, but leaves the assertion enabled. The state of the assertion is not affected. The *`assertion_handle`* can be any immediate or concurrent assertion; however, there is no effect on immediate assertions or Boolean concurrent assertions, because they are completed before the VPI is scheduled.

- *`assertion_handle`* --Specific to one assertion

Setting Callbacks on Assertions

You can set callbacks to occur when a condition is met, by using an instance of a predefined data structure. The system uses another instance of the data structure to pass information back. The predefined data structures are included in the `sv_vpi_user.h` file.

When you register the callback, you define what condition is significant, and what routine to execute when the callback occurs. A handle to the callback is returned when registration is successful. If an error occurs during registration, NULL is returned.

Callbacks on assertions, like controls, can be set on the assertion system, or on individual assertions.

Setting Callbacks on the Assertion System

Use the `vpi_register_cb` command to set a callback on all assertions in the simulation.

Syntax

```
callback_handle = vpi_register_cb(
     callback_reason,
     pointer_to_callback_function,
     user_data)
```

where:

- *`callback_reason`* can be:
  - `cbAssertionSysInitialized`)--This callback occurs after the system has initialized. No assertion-specific actions can be performed until this callback completes.
  - `cbAssertionSysOn` --The assertion system has become active, and starts processing

assertion attempts. By default, this callback occurs on simulation startup, but can be delayed with system control tasks, or by using $asserton with no arguments, or by using the Tcl assertion -on -all command.

- `cbAssertionSysOff`--The assertion system is temporarily suspended. No new attempts are processed, and no new callbacks occur. Assertions already executing are not affected. This callback occurs as a result of a VPI control, $assertoff with no arguments, or a Tcl disable command with the -all option.
- `cbAssertionSysKill`--The assertion system is temporarily suspended. No new attempts are processed, and no new callbacks occur. Assertions already executing are terminated. This callback occurs as a result of a VPI control, or an $assertkill command with no arguments.
- `cbAssertionSysEnd` --All assertion processing has completed, and will have no further affect. This callback normally occurs at the end of simulation.
- `cbAssertSysReset` --This callback occurs when the assertion system is reset.

- *`pointer_to_callback_function`* --Follows the normal VPI callback prototype, and is passed an `s_cb_data` structure containing the callback reason and any user data, *`user_data`* . For more information about the callback function, see "Callback Functions".
- *`user_data`* --User-supplied data to be passed to the callback function when the callback occurs.

Setting Callbacks on Individual Assertions

Use the `vpi_register_assertion_cb` command to set a callback on all assertions in the simulation.

Syntax

```
callback_handle = vpi_register_assertion_cb(
assertion_handle
     callback_reason,
     pointer_to_callback_function,
     user_data)
```

where:

- The *`assertion_handle`* can be:
  - Any concurrent, cover, assert, or assume statement

  - An immediate assertion.
    Invalid assertion names are sequence or property names.

- The *callback_reason* can be:
  - cbAssertionStart (606)--An attempt has started.
  - cbAssertionSuccess (607)--An attempt reaches the finished state; vacuous successes are not reported.
  - cbAssertionFailure (608)--An attempt fails.
  - cbAssertionStepSuccess (609)--Not supported.
  - cbAssertionStepFailure (610)--Not supported.
  - cbAssertionDisable (611)--Assertion is disabled as a result of a VPI control action, $assertoff with arguments, $assertkill with arguments, or Tcl without -all)
  - cbAssertionEnable (612)--The assertion is enabled as a result of VPI control or $asserton with arguments, or by a Tcl command without the -all option.
  - cbAssertionReset (613)--The assertion is reset. This callback occurs as a result of a VPI control or $assertkill command with arguments.
  - cbAssertionKill (614)--An attempt is killed as a result of a control action. This callback occurs as a result of a VPI control or an $assertkill command with arguments.

- *pointer_to_callback_function* --Follows the normal VPI callback prototype, and is passed an s_cb_data structure containing the callback reason and any user data, user_data.

- *user_data* --User-supplied data to be passed to the callback function when the callback occurs.

Callback Functions

The callback function is executed when a registered callback occurs. Each callback can define a unique callback function. This function is defined when the callback is registered.

The VPI prototype is passed an s_cb_data structure containing the callback reason and any user data.

Syntax

```
my_cbname(
callback_reason
     callback_time,
     assertion_handle,
     pointer_to_attempt_info,
     reference_to_user_data)
```

where:

- *callback_reason* is the reason specified with `vpi_register_cb` or `vpi_register_assertion_cb`.
- *callback_time* can be:
  - `cbAssertionStart`--Time when the assertion attempt started
  - `cbAssertionSuccess`--Time when the assertion succeeded non-vacuously
  - `cbAssertionFailure`--Time when the assertion fails
  - `cbAssertionStepSuccess`--Not supported
  - `cbAssertionStepFailure`--Not supported
  - `cbAssertionDisable`--Time when the assertion was disabled
  - `cbAssertionEnable`--Time when the assertion was enabled
  - `cbAssertionReset`--Time when the assertion was reset
  - `cbAssertionKill`--Time when the assertion was killed
- *pointer_to_attempt_info* can be:
  - For disable, enable, reset, and kill callbacks, NULL is returned
  - For start and success callbacks, only `attemptStartTime` is valid
  - For a `cbAssertionFailure` callback, the `attemptStartTime` and `detailfailExpr` fields are valid
  - Step callbacks are not supported.

The Incisive simulator provides the attempt and callback time in `vpiScaledRealTime` format. The IEEE 1800 standard does not specify which format to use, so it is up to the implementation. The VPI code must check which format is in use to be portable.

The following example sets a callback to occur any time a concurrent assertion with the `assert`

directive fails.

```
if ((assertion_iterator= vpi_iterate(vpiAssertion, 0)) != NULL ) {
    while ((assert_handle = vpi_scan(assertion_iterator)) != NULL ) {
     if (vpi_get(vpiType, assert_handle) == vpiAssert) {
         /* setup a callback on failure */
         callback_handle = vpi_register_assertion_cb (
             assert_handle,
             cbAssertionFailure,
             Pointer to the callback function,
             User data)
     }
    }
}
```

Removing Callbacks

You can use the `vpi_remove_cb` command to remove an assertion system callback or an individual assertion callback.

Syntax

```
vpi_remove_cb(callback_handle)
```

where the *`callback_handle`* is the value that was returned when the callback was registered.

Using VPI to Query Assertion Statistics

You can use VPI to query SVA assertion counter statistics and other information.

Syntax

```
vpi_get(statistic, assertion_handle)
```

where *`statistic`* can be:

- `vpiCovered`--True if the assertion has been attempted, has succeeded at least once, and has never failed.
- `vpiAssertAttemptCovered`--Returns the number of attempts to match the assertion.
- `vpiAssertSuccessCovered`--Returns the number of times the assertion has succeeded non-vacuously or, if the assertion handle corresponds to a cover sequence, the number of times

the sequence has been matched.

- `vpiAssertVacuousSuccessCovered`--Returns the number of times the assertion has succeeded non-vacuously.
- `vpiAssertFailureCovered`--Returns the number of times the assertion has failed.

Known Limitations

- Support for assertion controls on specific scopes is deferred for a future release.
- vpiHandles for property and sequence instances are deferred for a future release.
- VPI step controls are deferred for a future release.
- The following VPI dynamic assertion step controls and callbacks: `vpiAssertionDisableStep`, `vpiAssertionEnableStep`, `cbAssertionStepSuccess`, and `cbAssertionStepFailure`.

6

PSL Language Support Limitations

The Cadence-supported constructs of the PSL property specification language are limited to the Verilog, SystemVerilog, VHDL, and SystemC flavors of the Boolean layer, the LTL segment of the temporal layer, and some elements from the modeling and verification layers.
This subset is further restricted to the subset of PSL that can be checked on-the-fly, according to the definition in section 4.4.4 of the *1850™ IEEE Standard for Property Specification Language (PSL)*, 17 October 2005.

Unsupported PSL Constructs

An X in the following table indicates that the PSL construct is **not supported** for a given language.

PSL Construct	LRM	Unsupported for				
		Verilog	Verilog-AMS	VHDL	SystemVerilog	SystemC
Macros (`%for`, `%if`)	4.2.4			X		X
`HDL_DECL`, `HDL_STMT` in verification units	4.3.2.2					X
`HDL` variables as formal arguments	4.3.2.4					X

The built-in `next()` function, as opposed to the LTL `next` operator, which is supported	5.2.3.2	X	X	X	X	X
The `ended()` function	5.2.3.6			X		X
`nondet()`, `nondet_vector()`	5.2.3.9 5.2.3.10	X	X	X	X	X
The `union` operator on Verilog, VHDL, or SystemC expressions	5.2.4	X	X	X	X	X
Multiple/nested clocks	6.1.2.5					X
The strong forms of `next`	6.2.1.3.4	X	X	X	X	X
The `next_event` operators	6.2.1.4.3 6.2.1.4.4 6.2.1.4.5	X	X	X	X	X
The strong forms of `before`:`before!` and `before!_`	6.2.1.5.2					X
The strong forms of `until`:`until!` and `until!_`	6.2.1.5.3	X	X	X	X	X
Parameterized properties	6.2.1.7.1	X	X	X	X	X
Logical iff, `<->`	6.2.1.7.3	X	X	X	X	X
Property OR	6.2.1.7.5	X	X	X	X	X

Base LTL operators (`X`, `G`, `F`, `U`, `W`)	6.2.1.8	X	X	X	X	X
The OBE operators, including `<->` (iff)	6.2.2	X	X	X	X	X
Formal parameters to property and sequence declarations: `bit`, `bitvector`, `numeric`, `string`, and `hdltype`	6.3.1					X
The `assume_guarantee`, `restrict_guarantee`, `fairness`, and `strong_fairness` verification directives	7.1.3 7.1.5 7.1.7	X	X	X	X	X
Unbound verification units	7.2.1					X
Verification unit binding to instances, as opposed to modules, or to other scopes	7.2.1		X	X		X
Verilog type extensions (`integer` range, `struct`) [1]	8.1 8.2	X	X	n/a	n/a	n/a

1 The SystemVerilog equivalents can be used instead of the Verilog extensions.

Only the pragma form of PSL is supported in SystemVerilog packages.

Other Known Limitations in PSL Support

- For the SystemVerilog flavor of PSL, you cannot call the following types of functions from within a property:
  - Functions that have output or inout arguments
  - Functions that have non-constant reference arguments
  - Functions that have static data declarations
  - Functions that modify data outside of the function scope
- For the VHDL flavor of PSL:
  - You cannot use an endpoint declared in an upper scope.
    For example, a property declared in a `generate` scope cannot use an endpoint in the architecture body.
  - You cannot use parameters, such as `generate` variables, in the repetition expression of a SERE.
- There are some restrictions on the use of the `forall` replicator; for details, see "PSL Property Replication".
- In a verification unit, if you make an assignment to a signal declared in the design, that assignment is added to the design. When there are multiple assignments to the same signal, standard HDL rules apply. The Cadence implementation does not support the override semantics described in the PSL LRM, so assignment in the verification unit does not override assignments in the design. A VHDL signal defined in a verification unit does not override a signal of the same name in the design.
- Although you can bind a verification unit to a blackboxed module, both the module and the verification unit properties are ignored during simulation. To apply PSL properties to the inputs and outputs of a blackboxed module, bind the verification unit to the parent module or hierarchy.
- Memories and multidimensional arrays are not supported as arguments to the built-in functions.
- Verification unit instance binding is not supported for VHDL and SystemC.
- Synthesis pragmas are not permitted in verification units.

7

Using SVA

The `bind` directive binds properties to design units. It can be specified in

- Any SystemVerilog interface or module scope
- An external text file specified to the elaborator
- A compilation unit scope

> When using this `bind` solution, it is not necessary to specify whether the target is Verilog or VHDL. To ensure that binding occurs correctly, the `bind` specification must adhere to the Incisive simulator mixed-language binding rules. For details, see "Mixed Verilog/VHDL Simulation" in the *Verilog Simulation User Guide*.

Generic Bind Syntax

Following is the generic `bind` syntax:

```
bind target bind_obj [(params)] bind_inst (ports);
```

The argument values you use depend on whether the target is SystemVerilog or VHDL.

SystemVerilog Target	VHDL Target

target	The name or hierarchical name of the module or interface instance to which the *bind_obj* will be attached. When `bind` is specified in a text file, any hierarchical names for target must begin with $root, because the text file does not have the context to resolve the hierarchical identifier correctly. For a target specified as *module:instance* , the *bind_obj* will be bound to all instances of module that have the name instance throughout the design. Multiple instances are separated by commas.	The module name of the VHDL object to which the *bind_obj* will be attached, which can be a VHDL • Entity--Applies to all instances of all architectures of this entity • Architecture--Applies to all instances of this architecture • Instance • Configuration instance The *target* can also be specified as a Cell(View) name; for details, see "Limitations on Binding to a VHDL Target". For a target specified as *module:instance* , the *bind_obj* will be bound to all instances of module that have the name instance throughout the design. Multiple instances are separated by commas.
bind_obj	The name of the SystemVerilog object to bind to the SystemVerilog target. The *bind_obj* can be a module or interface. Binding to multiple instances is supported. The bind statement cannot be specified in the *bind_obj* .	The name of the SystemVerilog object to bind to the VHDL target. The *bind_obj* can be a module or interface. Binding to multiple instances is supported. The `bind` statement cannot be specified in the *bind_obj* . A variable or a user-defined function call cannot be used as a *bind_obj* .

	A list of actual argument values. Full Verilog expression syntax is supported for arguments.	Used for generics only. A list of parameter values mapped to VHDL generics or literals. Only simple names are allowed for parameters.
bind_inst	The instance name that the SystemVerilog object will have when it is attached to the module. This name becomes part of the hierarchical reference for assertions and signals in the bound module.	The instance name that the SystemVerilog object will have when it is attached to the VHDL object. This name becomes part of the hierarchical reference for assertions and signals in the bound module. The instance created for this binding adds another level of hierarchy to the elaborated design under the VHDL target.

	The ports to bind to. Full Verilog expression syntax is supported for port connections. Ports can be connected by order or name. The `.*` notation and out-of-module references (OOMRs) are allowed in the port connection list when binding to SystemVerilog. The `.*` notation is not supported when binding to a Verilog or VHDL design file.	The port map aspect that connects the SystemVerilog formal ports used in the assertions to the VHDL actual ports. Ports can be connected by order or name. Multiple port connections are separated by commas. VHDL ports of mode `out` can be read and used in the bound SystemVerilog module. The VHDL actual port can be an external port or internal signal declared in • The architecture declarative part • The entity declarative part • A package • A package body--deferred constants **Note:** Only simple names are supported for port connections. No expressions are allowed. Connecting to VHDL records or two-dimensional arrays, or any part of them, is not supported.

Note: Support files for SystemVerilog binding are created in the *worklib*`/.cdssvbind` directory for three-step mode, and `./INCA_libs/.cdssvbind` for irun.

SystemVerilog-to-SystemVerilog Binding Example

binds the fifo_full assertion module to the fifo1 instance in the design, with an instance name of v2. The following `bind` statements are valid for this example:

```
 // Instantiates fifo_full in all instances of fifo.
bind fifo fifo_full v1(clk, empty, full);
```

```
// Gives a hierarchical path to the instance.
bind top.dut.fifo1 fifo_full v2();


// Binds to a list of instances.
bind fifo:fifo1,fifo2 fifo_full v3();
```

SystemVerilog-to-VHDL Binding Example

The following example shows how a SystemVerilog assertion module can be bound to a VHDL entity/architecture.

The following `bind` statements are valid for this example:

```
// Instantiates fifo_full in all instances of fifo.
bind fifo fifo_full v1(clk, empty, full);


// Binds to an instance.
bind fifo:fifo1 fifo_full v2();


// Binds to an instance.
bind cell(v2_arch) fifo_full v3();
```

Note: Do not specify the entity/architecture using a hierarchical name starting with a Verilog top. Verilog hierarchical names ending in VHDL scopes are not permitted.

Recommended Use Models for Binding

In general, it is best to consolidate all of the `bind` statements into one or a few places, to make binding easier to manage. There are two recommended use models:

- Place all binds in a separate text file.
- Include binds in a compilation unit scope.
- Embed binds in a design file.

Binds in an External Text File

The advantage of placing all `bind` statements in an external text file is that you can place all Verilog and VHDL binds together in one file, and all you see in the Design Browser is the modified hierarchy of the design. The disadvantage is that SimVision does not facilitate viewing the file that contains the `bind` statements. You can use the File - Open menu item to view the file, but you need to know where it is located.

Following is an example of a bind file:

```
// bindfile.txt
bind target1 property_file1 #(1) instance_name(arg1,arg2);
bind target2 property_file2 #(1) instance_name(arg1,arg2);
```

The `//` and `/* */` Verilog-style comments only are supported.

You must specify the name of the external file using the `-extbind` *`filename`* option; for example:

```
ncelab -extbind bind_file
```

```
irun -extbind bind_file
```

Notes about external text files:

- You can use multiple `-extbind` options.
- External files can contain a mixture of SystemVerilog and VHDL.
- If you are using irun with the `-top` option, you must also specify `-top binds_in_textfile` as a second top module name. For example:

  ```
  irun -extbind bind_file -top dut -top binds_in_textfile
  ```

  Using `-top binds_in_textfile` is only necessary if you use `-top` already, which is a requirement for VHDL.

Binds in a Compilation Unit

You can include the `bind` statement in the same file as the module to be bound, but it must be outside of the module definition, in the compilation unit scope. For example:

```
bind top.dut.fifo1 fifo_full v2();
```

Another approach is to have all `bind` statements in a file, and provide that file in the same

compilation as the related design files.

Binds Embedded in a File

If you want to embed `bind` statements in your design or test, you can include them in any module, interface, or program provided. This technique can be useful, for example, when a bind is associated with a specific test.

Binding and Protected IP

If a `bind` directive appears within a protected portion of code, an error is generated.

If a `bind` directive is not protected, the binding is executed.

If the target of a `bind` directive is protected, the bound instance will exist, but it will not be visible in SimVision.

For details, see Chapter 10, "Writing Assertions for Protected IP."

Binding and ncdc

When you use the ncdc utility:

- ncdc will generate the elaborated hierarchy.
- Bound instances will appear in the bound hierarchy.
- The original bind files and instantiations will not appear.

Known SVA bind Limitations

General bind Limitations

This implementation of the `bind` statement does not support the following:

- Binding to an array of instances
- Binding to an AMS, ***e*** , or SystemC module
- Specifying as the target a Verilog hierarchical name that starts or ends in a VHDL scope

- The following table lists the Verilog and VHDL data types that are supported for port connections.

Verilog	VHDL
`bit`	`std_logic`
`bit`	`std_ulogic`
`bit vector`	`std_logic_vector`
`bit vector`	`std_ulogic_vector`
`bit vector`	`signed`
`bit vector`	`unsigned`

If a vector is used, the VHDL actual must be a constrained one-dimensional array subtype of a `std_logic` vector, and the entire array must be passed. VHDL record members are not supported as actuals.

- Multiple variants of a target

  Verilog and VHDL configurations have no effect in resolving the target of binds specified in a text file.

  Configurations can be used to resolve multiple variants of a target of binds specified in a module scope.

Limitations on Binding to a VHDL Target

The following are limitations specific to binding to a VHDL target:

- The `bind` statement must be specified in any Verilog context where a regular SystemVerilog-to-SystemVerilog binding is valid.
- Only the following VHDL object classes can be accessed by SystemVerilog assertions:
  - Signals
  - Ports
  - Constants
  - Package constants
  - Package signals
  - Shared variables

    - Generics
- When you bind VHDL generic parameters to Verilog parameters, you can use the following VHDL generics:
    - `STD.STANDARD.INTEGER`
    - `STD.STANDARD.REAL`
    - `STD.STANDARD.TIME`
    - `STD.STANDARD.STRING`
    - `STD.STANDARD.BOOLEAN`
    - Any user-defined enumerated type
- VHDL process variables are not supported.
- When you map a VHDL actual directly to a Verilog module parameter through a generic map aspect, the actual must be one of the following types:
    - `INTEGER_LITERAL`
    - `REAL_LITERAL`
    - `STRING_LITERAL`
    - `ENUMERATION_LITERAL`
- The syntax for binding to a compiled VHDL cell view is slightly different from binding to other VHDL objects. The target, *`Cell(View)`* , must be the cell view instance name of the architecture, and the *`bind_obj`* must be the SystemVerilog module name and parameter map: ();

When to Use SVA assert versus cover

The following applies to simulations using attempt-based counting only; it does not apply to trace-based counting. For details, see "Setting Up the Assertion State Counters."

If you need comprehensive information about property failures, use the `assert` directive. When the `assert` directive is used, by default property failures are

- Reported to the log file
- Reflected on waveform failure probes
- Affect the pass/fail exit code of the simulation
- Affect breaks on assertion failures

An `assert` directive can have an action block associated with it that can execute a statement when the property passes, and another when it fails. You also get coverage information when you use this directive--the Assertion Browser and summary report give you the disabled, finished, and failed counts for each assertion.

Use of the `cover` directive does not provide comprehensive failure information like an `assert` directive does. Specifically

- It does not log failure messages to the log file.
- Waveform failure probes will not show these failures.
- It does not affect the pass/fail exit code of the simulator.
- It does not affect breaks on failures.
- No failure counter is provided.

The `cover` directives--`cover sequence` and `cover property`--are intended to provide only coverage information. The `cover sequence` directive provides more comprehensive coverage information than the assert directive or the `cover property` directive. Specifically, overlapping traces created by matching design behavior to a sequence are all tracked to completion by `cover sequence`. The action block statistics and finished count increment once for each match. When you use the `assert property` or `cover property` directive, however, the property is finished as soon as one trace completes.

Turning on Synthesis Pragma Checks

When you turn on synthesis pragma processing with the `-genassert_synth_pragma` compiler option, synthesis pragmas are translated to PSL assertions and checked.

The clocking of these internally-generated assertions is based on the default PSL clock. If you do not have a clock defined, the pragmas are checked every simulation cycle. To ensure proper clocking of the translated pragmas, you can define a default clock in PSL, as described in "Declaring Default Clocks in PSL".

Avoiding Race Conditions

When assertions sample signals at the same time that the signals are changing, a race condition can result, and the assertion might report an erroneous failure. To avoid race conditions of this kind, clock assertions when the signals are stable.

Using SVA in a PSL Verilog Verification Unit

Similar to using an SVA `bind` directive, you can add SVA assertions to an existing design without modifying the design source code, by placing it within a *verification unit* associated with the relevant portion of the design.

Note: For more information about PSL verification units, see "Putting PSL Assertions in Verification Units".

There are advantages and disadvantages of using PSL verification units versus SVA `bind` statements:

	PSL Verification Unit	SVA bind
Advantages	• No need to change the design file. • No port list is required.	• No need to change the design file. • Mixed-language binding is supported. • The bound module or interface has a unique hierarchy.
Disadvantages	• The scope is not unique when module binding is used, so there can be name conflicts between verification units. • Mixed-language binding is not supported. • Instance binding is currently available only for Verilog/SystemVerilog.	• A complete port list is required for the module or instance to be bound.

Mechanics	• External file name provided at compile time with the `-propfile` option.	• Must supply a bind directive.

The following is an example of using SVA in a PSL verification unit. The `READ_BURST_READ_N` assertion is written in the Verilog flavor of PSL, and the `END_OF_BURST` sequence and `COUNT_BURST_READ_N` assertion are SVA:

```
vunit Controller_Assertions (memctl) {

// PSL
READ_BURST_READ_N: assert
  always ( {state == ready && m_req == '1' && m_task == read_burst} |=>
  {READ_PULSE[*]; (addr_counter == (m_data[4:0]) - 1) && state == ready})
  @(posedge clk);

// SVA
sequence END_OF_BURST;
  addr_counter == (m_data[4:0] - 1 && read_n ##1
  (addr_counter == (m_data[4:0] - 1) && !read_n)[*2] ##1
  (state == ready)
);
endsequence

COUNT_BURST_READ_N: assert property (
  @(posedge clk)
  (state == ready && m_req && m_task == burst_read) |=>
  (READ_PULSE[*0:$] ##1 END_OF_BURST) );

}
```

If it is unclear whether an assertion is PSL or SVA, the Incisive simulator assumes that it is PSL.

Although you can put SVA and Verilog PSL in a verification unit, you cannot define a sequence or property in one language and use it in the other.

This capability is not part of either the PSL or the SystemVerilog standard.

Using Packages for SVA

SVA properties can be defined in packages and imported for use. This is very useful for generic properties that are commonly used.

Use Model

The use model is as follows:

1. Define the properties in the package.

   Note: You can define properties in a package, but you cannot use assertion statements in a package.
2. In the program, module, or interface into which the package is imported, instantiate the property in an assertion statement.

You can declare PSL properties and sequences in SystemVerilog packages, and reference them from outside the package. PSL sequences and properties can also be imported from one package into another package.

Examples

The following examples show how to create properties in a package.

The first property, `area_not_preferred`, shows one approach. In this example, all of the signals that are used in the property must be declared in the package for the package to compile, although these declarations might not be used when the property is imported. Also, the signal names used in the package must match the signal names used in the design into which they are imported.

The second property, `area_preferred`, shows a better approach. This property takes advantage of the ability to specify property arguments in the package, and pass the signals in as arguments when the property is instantiated in a module. This preferred approach is shown by the `area_preferred` property.

```
`ifndef my_properties
package example_properties;

    byte width, length;
    int area;
    wire reset, clk;

    property area_not_preferred;
     @(posedge clk) disable iff (!reset)
     (area == length * width);
    endproperty

    property area_preferred(clk2, reset2, length2, width2, area2);
     @(posedge clk2) disable iff (!reset2)
     (area2 == length2 * width2);
    endproperty

endpackage
`define my_properties
`endif

// Module that imports the property from the package
module calculate_area;
    byte width, length;
    int area;
    wire reset, clk;

    Code here

    import example_properties::area_preferred;
    area_check: assert property (area_preferred(clk, reset, length, width,     area));
endmodule
```

If you include PSL in your package, you must use the pragma syntax:

```
`ifndef sv_props_pkg
package sv_props_pkg;
    reg X, Y, Z, clk, rst;
    // psl sequence SY = {S;Y};
    // psl property XY_Z = (always
     ended({XY}) -> next Z) @(posedge clk);
endpackage
`define sv_props_pkg;
`endif

module test;
...
import sv_props_pkg::*;
    // psl assert_XY_Z: assert XY_Z;
endmodule
```

Packages and Compilation Units

Although a compilation unit is an implicit package, and variables from one package can be imported into another, you cannot import variables from a compilation unit scope into a package. A compilation unit scope can, however, import variables from a package.

Contexts within a compilation unit, such as modules, programs, and interfaces, can directly access variables within the compilation unit. In cases where a reference might be ambiguous, the variable can be prefixed with `$unit::`.

Using SVA Repetition in Implication Operations

There is a fundamental difference between the left and right sides of a suffix-implication operation. These sides are sometimes described as "all match" and "first match," respectively.

In English, the suffix implication might read "for all matches of the left operand, there must be a match of the right operand." If you use an open-ended repetition operator at the end of the left operand, the property is tightly constrained, because it must satisfy all cases.

When you shift the open-ended repetition to the right operand, where the meaning is "first match," the constraint described by the property is loosened, by allowing the final fulfillment of the right operand sequence to be delayed.

For example:

```
Y_after_X_before_X_or_Z_3: assert property (
   @(posedge clk)
   ( {boolX; (!boolX && !boolZ)[*0:2]} |=> seqY ) abort rst);
```

This example is checked as follows, which results in many consecutive failures:

```
    ( {boolX; (!boolX && !boolZ)[0]} |=> seqY ) abort rst;

    ( {boolX; (!boolX && !boolZ)[1]} |=> seqY ) abort rst;

    ( {boolX; (!boolX && !boolZ)[2]} |=> seqY ) abort rst;
```

As shown in this example, many overlapping assertions have resulted. A large number of overlaps can result in performance degradation.

Using SVA Reactive Test Techniques

You can use SVA to implement reactive tests by using

- The pass statement of an assertion action block
- The `expect` statement
- The end of a sequence as a level-sensitive wait control
- Sequences used directly as event controls:
  `@(sequence) begin ... end`
- VPI to integrate C code into the test

For general information about reactive test architecture and tradeoffs, see Chapter 9, "Writing Reactive Tests using Assertions."

> You can also use the SystemC API to access assertions. For details, see "Access to Assertions from SystemC Testbench," in "Using SystemC PSL ," of the *SystemC Simulation Reference*.

SVA Reactive Tests using the Pass Statement

The syntax of the pass statement of an assertion action block is

```
assert property | cover property (property_expr) action_block;
```

Because the pass statement of an action block is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench. For example:

```
assert property (myfunc(a,b)) count1 = count + 1; else -> event1;
```

Action blocks are associated with both concurrent and immediate assertions. All assertions, with the exception of immediate assertions in tasks and functions, create entries in the Assertion Browser and assertion reports. Concurrent assertions are also recorded in the coverage reports.

SVA Reactive Tests using the expect Statement

An `expect` statement is procedural, so only one thread is active. It is blocking in nature, so the next statement is not evaluated until the expected property passes or fails. The structure of the `expect` statement is like an `assert` statement, because it can have an action block.

Because the `expect` statement can fail, it is treated like an assertion, in the sense that there is an entry in the Assertion Browser and assertion reports, and in the coverage reports.

One technique that can be used in SystemVerilog is to consolidate sequence generation and detection into the test itself. Not only is this architecture useful for simulation, it is necessary for acceleration, to minimize the synchronization needs of signals that pass between the simulator and the hardware emulator.

The following code shows the use of the `expect` construct in a SystemVerilog test that waits for ten cycles for a signal to have a certain value, then reports success or failure:

```
initial begin
    expect (@(clk) ##[1:10] data == value)
        $info ("value occurred. Continuing with test");
    else
        begin
            $error("value did not occur within 10 clock cycles");
            $finish;
        end
end
```

SVA Reactive Tests using Sequence Methods

One of the ways you can use SVA to implement reactive tests is by using the end of a sequence as a level-sensitive wait control.

You can use the end of a sequence as a control by combining the level-sensitive `wait` statement in conjunction with the `triggered` sequence method ("triggered Sequence Method"). The syntax is

```
wait ( sequence_instance.triggered );
```

You can delay the execution of procedural code by using `wait`, then use this function to communicate sequence completion to a test.

In the following example, the `initial` block in the `check` program waits for the endpoint of either sequence `abc` or sequence `de`. When either condition evaluates to true, the `wait` statement unblocks the process, displays the sequence that unblocked the process, and executes the statement labeled L2.

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence

   sequence de;
    @(negedge clk) d ##[2:5] e;
endsequence

   program check;
    initial begin
     wait (abc.triggered || de.triggered);
     if (abc.triggered)
         $display ("abc succeeded");
     if (de.triggered)
         $display ("de succeeded");
     L2: ...
    end
endprogram
```

SVA Reactive Tests using Sequence Events

When a sequence instance is specified in an event expression, the process that is executing will block until there is a match for the entire sequence. The process resumes execution when the match is detected.

Reactive tests can respond to legal or illegal activity or inactivity. A generic example is a test that reacts to a successfully-transmitted frame by adding another frame to the queue. It can also add the frame to the receive queue of the appropriate response checker:

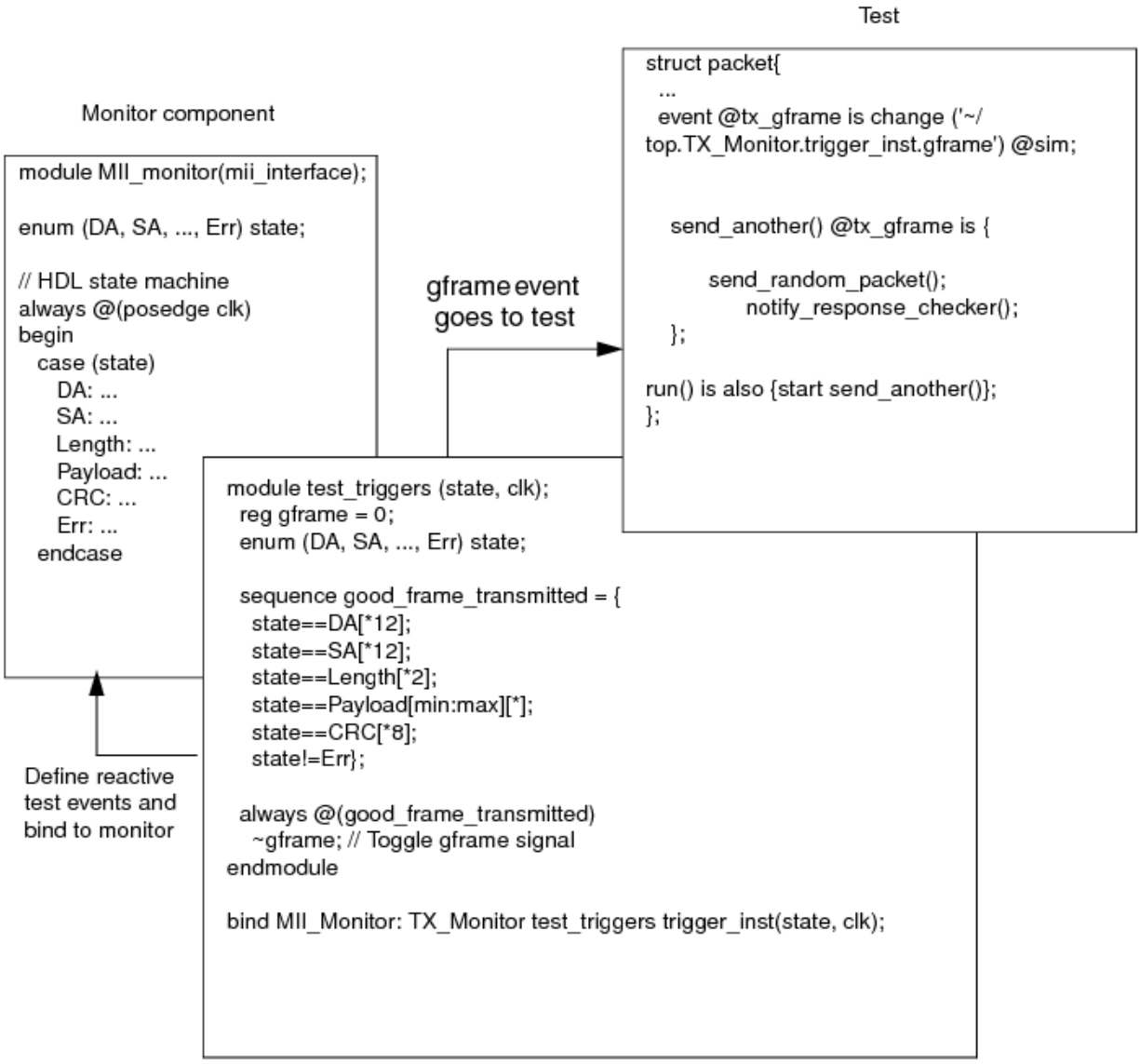
- The reactive test constructs are bound to a monitor that has an HDL-based state machine, and many associated compliance and coverage assertions.
- A sequence is defined that detects when a good frame has been transmitted. When the sequence is detected, an event is triggered.
- The sequence and detection code is placed in a module called `test_triggers`. The `bind` statement is used to instantiate the `test_triggers` module into the `TX_Monitor` instance of the `MII_Monitor`.

  The `bind` statement in essence instantiates `test_triggers` into the `TX_Monitor` instance with an instance name of `trigger_inst`.
- The `state` and `clk` variables are passed in, then the test references the name of the event using its full hierarchical path.

This example assumes that the testbench module is named `top`, and the monitor is instantiated with the name `TX_Monitor`.

Example 7-1 shows an implementation of this technique.

Example 7-1 Example *e* Test with SVA Assertions



SVA Reactive Tests using VPI

The following example uses VPI to determine whether the test encountered any unexpected assertion failures, and uses that information to print a pass/fail test at the end.

```
#include <vpi_user.h>
```

```
#include <sv_vpi_user.h> // Contains assertion VPI declarations
#include <stdio.h> // Contains definition of NULL

static p_vpi_time p_start_time = 0;
static s_vpi_time s_start_time;
static vpiHandle cbArray[100] = {0};

char * reason_to_str(int reason)
{
    switch(reason) {
     case cbAssertionStart:
         return("Start");
case cbAssertionFailure:
     return("Failure");
    case cbAssertionDisable:
     return("Disable");
    case cbAssertionEnable:
     return("Enable");
    default:
     break;
    }
    return("Unknown");
}


/*
 * 'asrtcb' is registered as a callback for all the assertion
 * callbacks attached to assertions. It prints various details
 * about the callback.
*/
PLI_INT32 asrtcb(PLI_INT32 reason,
              p_vpi_time cb_time,
              vpiHandle assertion,
              p_vpi_attempt_info info,
              PLI_BYTE8 * user_data)
{
    p_start_time = 0;
    /* get the handle of the assertion_status signal defined in the design */
    vpiHandle assertion_status=vpi_handle_by_name("test.assertion_status",NULL);
    s_vpi_value value_1;
    vpi_printf("\n Assertion: %s", vpi_get_str(vpiName, assertion));
    vpi_printf("\tTime: %e", cb_time->real);
    vpi_printf("\tReason: %s\t Data: %d", reason_to_str(reason), user_data);

    if(info) {
     vpi_printf("\tStartTime:%e", info->attemptStartTime.real);
     if(info->detail.failExpr)
         vpi_printf("\n\tFAIL EXPR: %s", vpi_get_str(vpiDecompile, info->detail.failExpr));
    }
    if (strcmp( vpi_get_str(vpiFullName, assertion), "test.A1_sva" ) == 0)
     vpi_printf("\nEXPECTED FAILURE OCCURRED!!\n");
    else {
     vpi_printf("\nUNEXPECTED ASSERTION FAILURE - TEST WILL FAIL!!\n");
     value_1.format = vpiIntVal;
```

```
     value_1.value.integer = 1;
     vpi_put_value(assertion_status, &value_1, cb_time, vpiForceFlag);
    }
    vpi_printf("\n");
    switch(reason) {
     case cbAssertionStart:
         s_start_time.type = cb_time->type;
         s_start_time.real = cb_time->real;
         p_start_time = &s_start_time;
     case cbAssertionSuccess:
break;

case cbAssertionFailure:
break;


     default:
         break;
    }
    return(0);
}

/*
 * Register callbacks
 * This routine is called only once when 'ncsim' loads this library
 */
void abv_vpi_test() {
    vpiHandle hdl, asrts;


/* Iterate over all the assertions in the design and print the name */
    asrts = vpi_iterate(vpiAssertion, NULL);
    if(asrts) {
     while(hdl = vpi_scan(asrts)) {
         switch (vpi_get(vpiType,hdl)) {
             case vpiAssume:
                 break;
             case vpiAssert:
                 vpi_printf("\nPlacing callback on failure of assertion: %s",
vpi_get_str(vpiName, hdl));
                 vpi_register_assertion_cb(hdl, cbAssertionFailure, asrtcb, (PLI_BYTE8*)3);
                 break;
              default:
                 break;
     }
    }
}
    vpi_printf("\n\n");
}
```

When using VPI, it is important to have read and/or write access to the design variables that are being referenced and driven by the VPI code.

The following code is located in the HDL part of the testbench:

```
module test;
    bit assertion_status=0;

   ....

final
    begin
    if (assertion_status == 1'b1)
     $display ("***Simulation Failed due to unexpected assertion failures");
    else
     $display ("***Simulation Passed %d",assertion_status);
    end
endmodule
```

Using SVA in AMS Designs

For information about using SystemVerilog assertions in an analog/mixed-signal design, see "Applying Assertions to real, wreal, and electrical Nets" in Chapter 7, "Preparing the Design: Using Mixed Languages " in the *Virtuoso® AMS Designer simulator User Guide*.

8

Maximizing Assertion Performance

When assertions are enabled in simulation, there is an overhead (both memory and runtime) during compilation, elaboration and simulation. The most significant is the runtime overhead during simulation.

There is no standard measure of the overhead which assertions introduce. Overhead depends on a number of factors which include assertion density and coding styles.

However, there are several basic guidelines that, if followed, will help minimize the performance cost of assertions. The sections that follow, describe the command-line options to help maximize performance, switches to turn off some optimizations which are enabled by default, and assertion writing guidelines that will help you minimize the overhead of assertions in your design and maximize simulation performance.

`Assertions' as referred to in this chapter include `assert', `assume' and `cover' properties.

Command-line options

1. Limiting the failure/finish counts - You can disable the evaluation of assertions after a certain failure/finish count is reached. You can specify this limit on the command-line. During simulation once this limit is reached, assertion evaluation will be completely disabled, thus reducing the simulation run time overhead of running with assertions. Following are the options which you can use:
   - `-abvfailurelimit <Number>`
   - `-abvfinishlimit <Number>`
   - `-abvglobalfailurelimit <Number>`

   For more details on these options, see "Setting Assertions finished/failed limits" in *Assertion Checking in Simulation.*

2. Disable evaluation when assertion state is not changing - You can disable the evaluation of assertions if the assertion state on consecutive clock cycle is the same or if the variables sampled in the assertion are not changing. This will give you a runtime performance benefit because of the reduced number of assertion evaluations. The following are the options which you can use to enable this optimization:
   - `-abvnostatechange <state>|all`

     This is a run time option which can be specified with ncsim/irun to evaluate assertions and report results only when the assertion state changes.

     If the result of an assertion is same at any consecutive execution, the evaluation at the current cycle can be disabled and thus reduce the simulation runtime overhead of running with assertions. This will mainly impact assertions without any sequence or property operators. A particular state can also be suppressed. You can also choose to just stop evaluation for particular assertion state.

     You can also use the following Tcl command to suppress the assertion evaluation for same consecutive state:

Tcl command	assertion -nostatechange <state>|all

     This option is only applicable for SV Assertions.
3. Changing the evaluation mode - You can change the assertion evaluation mode (either attempt-based or trace-based). This choice can have a large effect on assertion enabled simulation performance. This can be done using the following option:
   - `-assert_count_traces`

     In the default mode i.e the attempt-based counting mode, each attempt must either finish or fail, and can finish or fail only once. Trace-based counting, on the other hand, always counts assertion traces, where a trace is defined as a path from start to end. For details about attempt-based and trace-based counting, see "Setting Up the Assertion Statistics Counters" in *Assertion Checking in Simulation*.

Tcl command	set assert_count_attempts

     Assertion simulation performance with attempt-based counting is inefficient when there are many attempts in progress simultaneously--that is, if the overlapping range of the enabling condition is very long. For any assertion that does not start a lot of parallel attempts, the performance will not degrade noticeably.

     In addition, the finish and fail counts might be different, depending on which count mechanism you choose:

     - Attempt-based counting reports when different attempts of an assertion finish and fail simultaneously, which makes the number of attempts greater.

- Trace-based counting reports multiple traces of an attempt, which can make the number of traces greater.

4. Optimizing two cycle assertions - You can optimize the evaluation of two cycle assertion in a design. This can be done using the following option:

   - `-abv2copt`

     When this option is used, failure/finish reporting gets modified as successive failure and finishes and cycle information are not reported.

5. Turning off assertion counters - You can optimize the assertion overhead during simulation runtime by removing assertion counters. As a result of this optimization, finished, failed, and disabled counts are collated and reported only when the user enables the counts using commands that supersede optimization. This feature is restricted to SystemVerilog assertions and is available under the following simulation time switch:

   - `-abvoptreporting`

     You can also use the following Tcl command to optimize counts at anytime during the simulation:

Tcl command	assertion -report opt

     If you use this switch, then it will report passing of the assertion but will not maintain any count. Thus `assertion -summary` will show "0" under disabled, finished, and failed column. This optimization switch affects the functionality of some tcl commands. For instance, if optimization is enabled, then `assertion -logging` command will not update global failure error count while `assertion -on` command will turn on assertion, but will not do any book keeping of counters.

     > If you run any one of the following commands, then the optimization will be disabled for those assertions on which these commands have been applied:
     >
     > abvglobalfailurelimit, errormax, abvfailurelimit, tcl, coverage u/all, assert_count_traces, abvnostatechange, and gui.

Default optimizations and switches to turn off optimizations

In IES, certain Assertion Optimizations are enabled by default. These include:

1. Optimizing single - cycle assertion evaluation when the assertion states are not changing - By default on an assertion clock, if the assertion state remains same (as in the previous clock

cycle), IES will ignore that assertion evaluation. The assertion state count will not be changed, no message will be reported, and action block will not be executed. Ignoring of assertion evaluation (when the evaluated state is same as in the previous clock cycle) is also applicable in the scenario where variable/expression has changed, but the resultant state is still the same. This can result in less number of finished/failed counts as compared to the full verbose functionality where assertion is evaluated at each clock cycle.

2. Optimizations for assertions having unbounded ranges - For assertions having unbounded ranges, if there are number of parallel attempts, concluding at the same simulation time, only the longest (primary) attempt is reported (in some cases, newest attempt might also get reported). All the secondary attempts are optimized out. This can result in less number of finished/failed counts as compared to the full verbose functionality where all the parallel attempts are evaluated and reported.
3. Single finished counts for cover property - As a result of this, only one finished count is reported for cover property and after one finish, the evaluation of that cover property is turned off.
4. Optimization for multiple identically clocked assertions - By default, IES will combine the evaluation of multiple identically clocked assertions in a given scope. This leads to simulation run-time optimization but, at the same time, can lead to skewed entries in the profile reports. The evaluation load of all assertions combined for evaluation, will show up against one single assertion.
5. Optimizations for vacuous counts and number of attempts for assertions - By default, IES does not count vacuous pass, number of assertion attempts and cumulative pass (vacuous pass + finished counts). This also implies that assertion -strict run-time tcl option will not work by default.
6. Optimizations for Single Cycle assertions reporting - As a result of this, the debug information related to number of cycles, is not reported with assertion failure message in the simulation log.

The following section describes the options/switches to reverse each of these optimizations.

- `-abvevalnochange`

  This is a compile time option that turns off the optimization for single cycle assertions. By default, such assertion evaluations are ignored if the assertion state remains same as in the previous clock cycle.

- `-abvnorangeopt`

  This is a run time option that evaluates all attempts for properties having unbounded ranges

([n:$]).

- `-abvrecordcoverall`

  This is a run time option that enables counting of all finishes for cover properties.
  In case of single cycle cover properties, just using this option is not enough. We also need to add the `-abvevalnochange` option to disable the single cycle optimization before the cover option can take effect.

- `-abvnoassertamalg`

  This is an elaboration time option that disables the merging of multiple identically clocked assertions.

- `-abvrecordvacuous`

  This is a compile time option that enables the counting of vacuous pass, assertion attempts and cumulative pass. This also enables the reporting of these counts when assertion -strict tcl option is used at run-time.

  > All the above-mentioned switches are meant for SVA only.

- `-abvrecorddebuginfo`

  This is a compile time option that enables additional debugging info for single cycle assertions (this switch will also imply -abvevalnochange), and is applicable to ncvlog and irun.

Coding Style Guidelines for Maximizing Assertion Performance

The following section describes the Coding style guidelines which are targeted towards minimizing the overhead of assertions and thus maximizing assertion performance.

- Minimize the Number of Attempts
- Minimize False Starts
- Minimize Overlapping Attempts
- Avoid Nesting always in Assertions

Minimize the Number of Attempts

Properties with enabling conditions that are true a lot of the time can slow down the simulator. When you use attempt-based counting for this type of assertion, a new attempt is started in each cycle while waiting for the fulfilling condition, because the enabling condition is still true. This result might not be what you intended. You can improve performance by being more specific about the enabling condition.

Use a signal transition to start tracking a new request. Change an assertion so that a different property is enabled when the key Boolean term--for example, request is high and grant is low--*becomes* true, rather than *is* true. The statistic results are different, because the assertion is activated fewer times, but errors will still be caught. The following example uses the low-to-high transition on `busreq` to start tracking a new attempt.

```
// SVA
property holdReq(i);
   @(posedge clk)
   ( rst && $rose(busreq[i] && !gnt[i]) ) |-> ##1
   ((busreq[i]) throughout (!gnt[i] [*0:$] ##1 gnt[i]) );
endproperty


// PSL
property holdReq(i) = always
   (( rst && rose(busreq[i] && !gnt[i]) ) |->
   ((busreq[i]) throughout {!gnt[i]; [*]; gnt[i]} ))
   @(posedge clk);
```

Minimize False Starts

In addition to minimizing the number of attempts for an assertion, try to start sequences or property enabling conditions with

- A condition that is rarely true
- A condition that, if true, usually indicates that the sequence of interest has begun

The simulator can optimize more aggressively for the case where the initial condition is false and there is no attempt in flight. If the initial condition is practically always true, this optimization cannot occur, both because the condition is true, and because there will almost always be an attempt in flight from the previous cycle.

For example, the following assertion starts a new attempt on every clock in which `a` is true:

```
  // Inefficient SVA assertion
```

```
  cover sequence ( @(posedge clk) a ##1 b ##1 c ) ;
// Inefficient PSL assertion
  S1: cover { a; b; c } @(posedge clk);
```

You can rewrite this assertion more efficiently as follows:

```
// More efficient equivalent SVA assertion
  cover sequence ( @(posedge clk) $past(a) && b ##1 c ) ;
// More efficient equivalent PSL assertion
  S1: cover { prev(a) && b; c } @(posedge clk);
```

The difference is that now a new attempt will start only when `a` is followed by `b`. Here is another example:

```
// Inefficient for SVA--starting condition is usually true
sequence dowrite;
  (state == idle) ##1 (state == write) ##1 (state == idle);
endsequence

// Inefficient for PSL--starting condition is usually true
sequence S2 =
  {state == idle; state == write; state == idle};
```

The interesting condition for this assertion is not the `idle` state, which is true most of the time, but the change from `idle` to `write`. This assertion will run faster if the starting condition is true much less frequently, which avoids false matches at an earlier cycle:

```
// More efficient equivalent SVA assertion
sequence dowrite;
  (state == write && $past(state) == idle) ##1 (state == idle);
endsequence

// More efficient equivalent PSL assertion
sequence S2 =
  {state == write && prev(state) == idle; state == idle};
```

Minimize Overlapping Attempts

When you use attempt-based counting, you need to minimize the number of overlapping attempts, because each attempt must be tracked independently. This is not a problem with trace-based counting, because multiple attempts are merged into one trace. The following is an example of an assertion that is not efficient when attempt-based counting is used:

```
// Inefficient for SVA when attempt-based counting is used
  assert property ( @(posedge clk) a ##[0:$] b |=> c );


// Inefficient for PSL when attempt-based counting is used
```

```
A1: assert always ( {a; [*]; b} |=> c ) @(posedge clk);
```

This assertion specifies that, when `b` follows `a`, `c` must be true on the next cycle. So for every attempt in which `a` is true, the simulator must check, every clock cycle, that if `b` is true, `c` follows on the next cycle. This definition prevents any attempt from ever finishing, because

- It is not possible to rule out a future failure of that attempt
- There can be a new attempt starting on every clock

In this case, put the large repetition on the right-hand side of the implication, so that the attempt can complete when `b` is followed by `c`. For example:

```
// More efficient way to write SVA for attempt-based counting
  assert property ( @(posedge clk) a |-> ##[0:$]b ##1 c );


// More efficient way to write PSL for attempt-based counting
  A1: assert always ( a |=> {[*]; b; c} ) @(posedge clk);
```

Depending on the application, it is even better to reduce the number of valid attempts that are started, as follows:

```
// Even better SVA--also minimizes the number of attempts that start
  assert property ( @(posedge clk) $rose(a) |-> ##[0:$]b ##1 c );

// Even better PSL--also minimizes the number of attempts that start
  A1: assert always ( rose(a) |=> {[*]; b; c} ) @(posedge clk);
```

Avoid Nesting always in Assertions

For efficient performance, avoid placing an `always` property within another `always` property. When attempt-based counting is used on properties with nested `always`, many different attempts are forked off and finish at the same time. A large number of fail/finish counts are reported, and performance is adversely affected. Following is an example of an assertion that is inefficient when attempt-based counting is used:

```
//PSL
A1: assert always( always ({x} |=> {y})@(posedge clk));
```

In addition to following these guidelines, you can selectively disable assertions if they are not required for a given simulation run. Disabling assertions improves performance. Assertions can be disabled at

- Run time--Provides some benefit; see "Enabling and Disabling Assertions " in *Assertion Checking in Simulation*.
- Elaboration time--Provides more benefit; see "Disabling Assertion Checking at Elaboration Time " in *Assertion Checking in Simulation*.
- Compile time--Provides the maximum benefit; see "Enabling/Disabling Assertion Checking at Compile Time " in *Assertion Checking in Simulation*.

9

Writing Reactive Tests using Assertions

Reactive tests--tests that use feedback from design behavior during simulation to drive the behavior of the test--can eliminate redundancy and dead time from a test, making it more efficient and meaningful.

Assertion languages are used in reactive tests because they let you easily and concisely express complex temporal conditions of interest--conditions that are often cumbersome to describe in HDL. The PSL and SVA assertion languages can efficiently monitor the signals in the device under test (DUT), and notify the test when a condition of interest is Observed.

For examples of reactive test implementations, see

- "Using PSL Reactive Test Techniques"
- "Using SVA Reactive Test Techniques"

Types of Reactive Tests

A reactive test responds to observed behaviors in the DUT. There are several behaviors to which tests can react:

- Event-oriented activity that is expected or not expected to occur--Conditions that occur in the DUT.

  HDL code detects the condition or sequence of conditions of interest, and notifies the test of the activity.

- Event-oriented inactivity--Conditions that do not occur.

  Watchdog timers can detect inactivity.

- Coverage-oriented activity or inactivity

Example behaviors might include arbitration being granted, a coverage goal being met, or some sequence of events completing. The test might react by

- Changing the constraints or configuration and continuing

- Changing which assertions or coverage monitors are enabled or disabled
- Stopping the test
- Evaluating and printing the test results

Creating Reactive Tests with Assertions

To create a reactive test, it seems reasonable to define a named sequence, property, or coverage point, and reference it in your HDL, or use it to trigger an event. However, this approach is not possible because, to prevent scheduling issues, assertion languages limit access to assertion variables.

Special language constructs enable communication with the associated HDL. To create a reactive test, you will need to use these constructs to communicate with the test:

- PSL--`ended()`
- SVA
  - The pass statement of an assertion action block
  - The `expect` statement
  - The end of a sequence as a level-sensitive wait control
  - Sequences used directly as event controls:
    `@(`*`sequence`*` ) begin ... end`
  - VPI to integrate C code into the test

Considerations for Reactive Tests

Tradeoffs to consider when creating your reactive tests include the following:

- Overhead for a given test, versus sharing event monitors across all tests
  Is a sequence used only when referenced by the test, or is it used all the time?
- Synchronization between the test and the simulator when the test language is not native to the simulator or accelerator
  For performance reasons, it is important to minimize the number of signals and events that pass from the simulator to the test.
- Partitioning code for reuse and stability
  - Should the test react to interface activity, or to activity in the DUT's internal functionality?

    - Should the reactivity code be associated with the test, the verification components, or the DUT?

- Maintenance
    - How does the user define and maintain a large collection of event names, and connect them correctly in each test?
    - How are the various external files managed?

- Handling of sequence overlap
  Sequence overlap might be more of a coding style issue, but it needs to be considered.

- Effect on report output
  To determine what constructs to use, consider whether an entry in the Assertion Browser and coverage report are desired for this condition.

- Methodology
  It is useful to have a common methodology that is independent of language, and portable across tools and vendors.

- Reactivity types
    - Do you want to be able to end or disable assertions?
    - Do you want access to assertion status or counts?

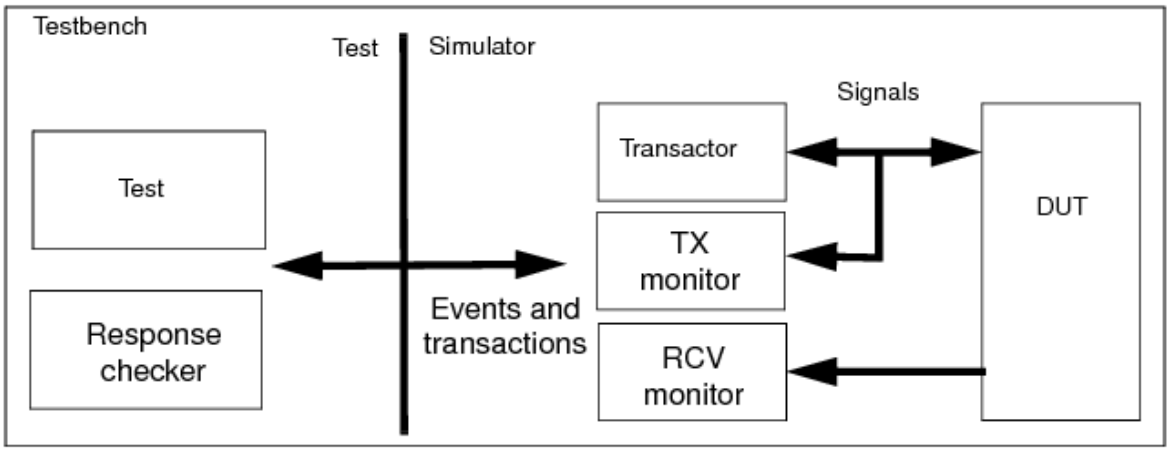
Reactive Test Architectures

In addition to understanding the constructs that can be used in reactive tests, it is important to understand where to use these constructs in a design. The following describes an example of a basic verification architecture with a worst-case scenario--most of the test running outside the simulator.

Figure 9-1 shows a recommended architecture. This is the worst case, in the sense that you need to minimize the amount of synchronization required between the test and simulator. This kind of architecture will work equally well across other verification configurations.

This figure shows a top-level testbench that instantiates the DUT and the verification components--the test, a response checker, a transactor that converts transactions to signal values, and several monitors, which are general-purpose monitors of the type that is commonly used in a verification environment. Because there will likely be many sequences that are specific to one or more of the tests but not to all tests, the reactive test is placed in a separate file that is selectively included.

Figure 9-1 Verification Architecture



Reuse and stability are key considerations in the placement of the reactive test constructs. You must consider whether you want the reactive test to react to interface activity or to internal functionality within the DUT:

- Internal functionality--Referencing the internal signals and states of a DUT is not recommended, because they are not as generic as the interface, and therefore not as re-usable. More importantly, because the design is in development, names might not be stable.
- Interface--An interface monitor is the recommended verification component to which to attach reactive test components, because monitors are designed to be reused, and they become stable sooner.

Reactive tests use assertion constructs to detect a condition of interest using specific signal activity. The assertions communicate these occurrences to the testbench using abstract testbench events, so that one event passes from the simulator to the test instead of all the signals that make up the sequence of interest.

In the monitor, or adjunct to the monitor, you need to

- Define the behavior of interest using the signals and/or states of the monitor.
- Send an abstract signal to the test when the behavior of interest is detected.

In the test, you want to:

- Detect the abstract event or signal change coming from the monitor.
- React as desired.

It is important to remember that

- The condition that is being monitored must be defined as a sequence or property.
- In the current Incisive simulator implementation, a sequence or property must be referenced in some other assertion construct, such as an `assert` or `cover` directive, or a built-in PSL function such as `ended()`. Otherwise, the sequence will be ignored by the compiler, because it will be perceived as not being used.
- The definition of a sequence and a reference to that sequence must be contained within the same module.
- You cannot use OOMRs or global signals to reference a named sequence or property.

You must define and detect the condition in the same module, then notify the test as follows:

- For Verilog or SystemVerilog, trigger an event and reference the event from the test by using OOMRs.
- For VHDL, set a global signal that the test can reference.

When the test language is native to the simulator or accelerator, the sequence and property definitions and detection can all be wholly contained in the test, where the property or sequence definitions can use OOMRs to reference signals.

Other Types of Reactive Test Responses

Most reactive tests respond to legal activity. You might also need a reactive test that responds to illegal activity, or to legal or illegal inactivity:

- Illegal activity
  An SVA assertion action block is the recommended approach to respond to illegal activity, so that the error is logged in the assertion error reports. The fail statement of the action block can trigger an event to the test for processing.

  For PSL, the SystemC assertion API or assertion breakpoints are useful.
- Inactivity
  When responding to inactivity, It is important to distinguish between inactivity that is a result of an error and inactivity that is permissible behavior. An example of legal event-oriented inactivity might be that a grant did not occur within a specified amount of time. The grant might not occur due to a bug in the chip, or it might not occur because the priority of the request was not sufficient. The response often depends on the protocol at the interface. Using an HDL-

based watchdog timer is the underlying technique for this type of test:

- For cases where the activity might be legal, use the action block to determine why the timer fired, and respond accordingly.
- For cases where the inactivity is likely to signal some unknown bug, the test should assert that the timer does not time out. This allows the behavior to be captured by error reports.

Examples of Reactive Tests

For examples of reactive tests, see the following:

- PSL
  - "SystemC Reactive Test with PSL"
  - "VHDL Reactive Test with PSL"
- SVA
  - "SVA Reactive Tests using the Pass Statement"
  - "SVA Reactive Tests using the expect Statement"
  - "SVA Reactive Tests using Sequence Methods"
  - "SVA Reactive Tests using Sequence Events"
  - "SVA Reactive Tests using VPI"

10

Writing Assertions for Protected IP

You might be writing assertions about a design that will be used as protected intellectual property (IP). If so, you need to make sure that end users can debug simulation results for their designs that contain your IP.

Selected Visibility

Use models for assertion protection are defined by the protected part of the assertion definition. For the greatest control, the property and verification directive are split. For example:

```
// Block to be encrypted by ncprotect
// pragma protect
// pragma protect begin
  `define property my_property
    (@(posedge clk)
    (a |-> b))
// pragma protect end
A_DEF:`my_property;
```

This technique, called *selected visibility*, protects the property definition, while keeping the assertion visible. The most common use model is selected visibility. In this case

- Messages about the assertion are printed to the log file.
- The summary report lists statistics for the assertion.
- The `assert`/`cover` statement named by the directive is visible in the Design Browser and Assertion Browser.
- The unprotected part of the assertion is visible in the Source Browser.
- The `assert`/`cover` statement is affected by Tcl commands.

Table 10-1 summarizes the level of interaction for end users of protected assertions.

Table 10-1 Reporting for Protected Assertions

Conditions	Log File Messages	Summary Report	Browser[1] Support
Total visibility: Unprotected assertions in protected IP	Normal	Normal	Normal
No visibility: Properties/sequences and their assertion directives are protected	Minimal, on failures only	No	No
Selected visibility: The property/sequence definition is protected, but the assertion directive is not	Message but no property references	Assertion only	Assertion only

1 Assertion Browser, Design Browser, and Source Browser

Log file run-time messages are as shown in the following examples:

- Total visibility
  Prints the source text of the assertion that failed; the file, line number, time of failure, and hierarchical path to the assertion; the number of cycles involved in the assertion failure; and the assertion start time.

```
(state == start_write)
  |
ncsim: *E,ASRTST (./memctl.v,59): (time 1525 NS) Assertion
memtest2.mctl.START_WRITE_MEMORY has failed (2 cycles, starting 1475 NS)
```

- No visibility
  Only assertion failures are reported; no filename, line number, or hierarchical path is printed.

```
ncsim: *E,ASRTST (): (time 1525 NS) Assertion has failed (2 cycles, starting 1475
NS)
```

- Selected visibility
  Prints all of the information about the assertion, but suppresses the source text of the assertion.

```
ncsim: *E,ASRTST (./memctl.v,59): (time 1525 NS) Assertion
memtest2.mctl.START_WRITE_MEMORY has failed (2 cycles, starting 1475 NS)
```

> The `assertion` and `probe` Tcl commands work for assertions with selected visibility.

For more information about the assertion summary, Assertion Browser, Design Browser, and Source Browser, see *Assertion Checking in Simulation* .

> Logging commands, such as `assertion -logging` and `assertion -summary`, are ignored for protected assertions.

Exceptions

There are several scenarios in which assertions, or any other variables, are unprotected, but you cannot access them:

- If a VHDL entity declaration is partially--starting from the beginning--or completely protected, the complete entity cannot be accessed.
  Only the text between the pragmas is encrypted. The corresponding bound architecture cannot be accessed, even if it is not protected. The following protects the complete ROM entity:

```
-- pragma protect

-- pragma protect begin

entity ROM is

-- pragma protect end
```

- If a VHDL architecture declaration is partially or completely protected, the complete architecture cannot be accessed.
  Only the text between the pragmas is encrypted. The entity corresponding to the architecture cannot be accessed, even if it is not protected. The following example protects the complete architecture:

```
-- pragma protect

-- pragma protect begin

architecture DataFlow of Full_Adder is

-- pragma protect end
```

Similar scenarios apply for Verilog. To protect a Verilog module so that the complete module cannot be accessed, you can protect

- The declaration of the module
- The keyword module
- The module name
- The entire module

Using IP Protection for Assertions

Notes about IP protection of assertions:

- Logging and probe commands are ignored for protected assertions.
  If all assertions are protected, a Tcl error occurs.
- If a `bind` directive appears within a protected portion of code, an error is generated.
- If a `bind` directive is not protected, the binding is executed.
- If the target of a `bind` directive is protected, the bound instance will not be visible in SimVision.
- If SVA inline code appears within a protected portion of VHDL code, an error is generated.
- If you use the `probe -signals` command on assertions that have contributing ports and signals that are protected, only the unprotected ports and signals are probed.
- If the path to an assertion is protected, the assertion is considered to be protected, even if the property/sequence definitions and directive are both unprotected. Such assertions are treated as if they have no visibility.
- Assertion system tasks, such as `$assertoff`, have no effect on assertions in protected code.

Limitations

The following limitations apply to IP protection when assertions are provided in a separate file:

- If the `bind` statement is used within a protected portion of code, an error is generated, because the module name must be visible.
  Only unprotected `bind` directives are executed.
- If the target of a `bind` directive is protected, the bound instance will not be visible in SimVision.
- A verification unit that is compiled into a protected design unit is also protected; Tcl commands will not affect assertions in such a verification unit.

11

PSL and SVA: Similarities and Differences

PSL was designed to work with many languages and support hierarchical properties. SVA was designed to work with SystemVerilog and express linear properties.

Where their capabilities overlap, the formal semantics of PSL and SVA are identical. Their syntax is the same in many cases, but is different at the top-level.

The primary characteristics of these assertion languages are the following:

- SVA
  - Defined for SystemVerilog HDL
  - Assertions are synchronous, with one asynchronous reset
  - Assertions are linear and implementation-oriented
- PSL
  - Defined for Verilog, SystemVerilog, VHDL, and SystemC
  - Assertions can be both synchronous and asynchronous, with multiple reset conditions
  - Assertions can be both abstract and hierarchical

Common PSL and SVA Capabilities

PSL and SVA can both describe the following:

- Conditions

  A relationship that exists among signals in a particular state of the design--that is, at a particular instant in time. Conditions are represented as Boolean expressions that can be interpreted as either true or false.

- Sequences

  A series of conditions over time. Sequences are represented using an extended form of regular expression syntax, which makes it possible to describe a variety of sequential

behaviors in a concise manner. See "Sequences".

- Properties

  A relationship that exists among conditions, sequences, and subordinate properties over time. Properties are represented by using logical implication and temporal operators. See "Assertion Property Declarations and Directives".

Both PSL and SVA use the following operators:

- Repetition

  Repetition operators make it possible to describe repetitive sequences of activity that must or must not occur. See "Sequence Operators".

- Implication

  Implication describes if/then conditions where, if a specified condition occurs, it must be followed by a second specified condition. Both PSL and SVA support the `|=>` and `|->` suffix implication operators, which have the same semantics.

Expressions

Both PSL and SVA use the underlying HDL expression syntax. PSL accepts Verilog, SystemVerilog, VHDL, and SystemC expressions. SVA accepts SystemVerilog expressions.

Built-In Functions

The SVA and PSL xbuilt-in functions are similar: `rose`, `fell`, `prev`, `stable`, `onehot`, `onehot0`, `countones`, and `is_unknown`. However, the notation varies, where PSL uses `rose()`, for example, and SVA uses `$rose()`.

PSL and SVA Similarities and Differences

Sequences

This section describes the similarities and differences between PSL and SVA sequence operators and semantics.

Sequence Operators

PSL and SVA have an equivalent set of sequence operators, with equivalent precedence, as shown in the following table:

Operator	PSL and SVA
Consecutive repetition	`[*count ] [*range ]`
Arbitrary repetition	`[=count ] [=range ]`
Goto repetition	`[->count ] [->range ]`
Sequence same-cycle implication	`|->`
Sequence next-cycle implication	`|=>`
`a` occurs in some cycle(s) during `b`	`a within b`

The differences between PSL and SVA operators are shown in the following table:

Operator	PSL	SVA
Sequence delimiters	`{ }`	`( )`
Sequence concatenation (non-overlapping)	`;`	`##1`
Sequence concatenation (overlapping)	`:`	`##0`
Sequence disjunction	`|`	`or`
Sequence conjunction	`&`	`and`
Sequence conjunction (length-matching)	`&&`	`intersect`
Sequence negation	`!`	`not`
`a` occurs in every cycle during `b`	`a[*] && b[*]`	`a throughout b[*]`
Zero or more	`b[*]`	`b[*]`
One or more	`a[+]`	`a[+]`
Unbounded ranges	`[0:inf]`	`[0:$]`

The following table gives examples of equivalent PSL and SVA sequence operator usage:

Type	PSL	SVA
Basic	`{ a; b; c }`	`( a ##1 b ##1 c )`

Repetition	`{ a[*1:3]; b[->2]; c[=3] }`
	`{ [*]; b }`
	`a[+]`
		`( a[*1:3] ##1 b[->2] ##1 c[=3] )`
		`( [*] ##1 b )`
		`a[+]`

Overlapping	`{ a; b } : { c; d }`	`( a ##1 b ) ##0 ( c ##1 d )`

Logical combination	`{ a; b } && { c; d } | { e; f }`
	`{ {a; b} within c[*] }`
	`{ {a} && c[*] }`
		`( a ##1 b ) intersect ( c ##1 d ) or ( e ##1 f )`
		`( (a ##1 b) within c[*] )`
		`( a throughout c[*] )`

Sequence Semantics

Both PSL and SVA imply nothing about unmentioned signals. If the assertion does not refer to a Boolean, for example, it is not constrained. For example, `a ##1 b` does not imply that `b` is either true or false in the first cycle, only that it must be true in the second cycle.

Both languages permit overlapping matches. For example, `{a; b; c}` starts a trace in a cycle if a is true. If a is true in the second cycle, another trace starts before the first has finished or failed.

Both PSL and SVA sequences can be used to express behavior using regular-expression style notation. A regular expression generally consists of a sequence of terms with optional repeat counts, so that each term is required to occur 0 or more times in order to match the regular expression. The PSL and SVA repetition operators provide this capability. For example:

PSL: `{ req; wait[*0:2]; ack }`

SVA: `( req ##1 wait[*0:2] ##1 ack )`

> Both PSL and SVA have repetition and delay shortcuts, where `[*]` means zero or more, and `[+]` means one or more.

This form assumes that each condition is equally important. In some cases, you might want to ignore certain conditions, and think of the behavior instead as a series of important conditions separated by delays. Both PSL and SVA sequences can be used to express behavior in a delay-

until-next form also. For example:

PSL: `{ req; [*0:2]: ack }`

SVA: `( req ##[0:2] ack )`

In this case, the two languages use slightly different operations to achieve the same goal. In the above example, PSL uses the repetition operation `[*0:2]`--which, when it stands by itself, is a shorthand for `true[*0:2]`, which will match anything for 0 to 2 cycles--together with the fusion operator, `:`, to represent the delay until `ack` occurs. SVA uses an extended form of the concatenation operator, `##`, to accomplish the same thing. Both have the same meaning.

Assertion Temporal Operators

PSL supports abstract temporal operators--`never`, `next*`, `eventually!`, `before*`, `until*`--for which there is no equivalent in SVA.

PSL uses `abort`, `async_abort`, and `sync_abort`; SVA uses `disable iff`. PSL puts the abort at the end of the property, SVA at the beginning.

Assertion Property Declarations and Directives

PSL supports an explicit `always` directive. SVA does not have a corresponding directive, but an `always` is implicit in every assertion.

The following PSL and SVA property examples are equivalent:

PSL	SVA
`always { req[*]; ack } |=> { read; drdy; !req; !ack }`	`( req[*] ##1 ack |=> read ##1 drdy ##1 !req ##1 !ack )`
`always { go } |=> { [*]; done } abort reset`	`( disable iff ( reset ) go |=> ##[*] done )`

The PSL `cover` directive and the SVA `cover sequence` directive both take a sequence argument. All matches of the sequence result in a match. SVA also has a `cover property` directive, which takes a property argument. It will have a maximum of one match per attempt. The `cover property` directive is similar to the `assert property` directive, except that failures are not reported.

Assertion Clocking

The way in which clock expressions are specified in PSL and SVA are similar, but there are some minor differences:

Clocking	PSL	SVA
Verilog event expressions	`@(posedge clk)` `@(clk, gate)`	`@(posedge clk)` `@(clk, gate)`
Boolean expressions	`@(clk)` (interpreted as level-sensitive clock)	`@(clk)` (interpreted as sensitive to both edges of the clock)

SVA infers clocking from the context of the procedural code; PSL does not. For example, the following PSL assertion will be evaluated at every simulator cycle, not on the positive edge of `clk`:

```
always @(posedge clk)
begin
 // psl example always a=1;
end
```

Embedding and External Modules for Assertions

Both PSL and SVA can be applied to a design in one of two ways. One approach is to embed PSL or SVA declarations and directives within the source text of the HDL description. Another approach is to place PSL or SVA declarations and directives in a separate file that is associated with an HDL module.

Embedding

PSL constructs can be embedded within HDL source text by using special comments in which the keyword `psl` is the first token after the comment symbol. For example:

Verilog or SystemVerilog PSL

```
module M (a,b,reset_n)
   ...
  // psl property P =
  //    never (a && b) abort reset_n;

  /* psl assert P @(posedge clk); */
   ...
endmodule
```

VHDL PSL

```
entity E is
   port (a,b,reset_n: bit);
    ...
   -- psl property P =
   --    never (a and b) abort reset_n;

   -- psl assert P @(posedge clk);
    ...
end;
```

Both line and block comments can be used, if the HDL supports both. A PSL declaration or directive in a line comment can be continued in a subsequent comment line without the `psl` token.

Because PSL can be embedded using comments, tools that do not support PSL assertions can be used together with those that do.

SVA constructs are inherently part of the SystemVerilog language, so SVA assertions can be embedded within any SystemVerilog design. For example:

SystemVerilog Assertions

```
module M (a,b,reset_n)
  ...
 property P;
  @(posedge clk)
  disable iff (reset_n) !(a&&b);
 endproperty: P

 assert property (P);
  ...
endmodule
```

External Modules

PSL constructs can be embedded within a *verification unit* that is associated with an HDL module. The associated verification unit is pulled into the compilation process whenever the HDL module with which it is associated is compiled. For example:

Verilog or SystemVerilog PSL

```
module M (a,b,reset_n)
   ...
endmodule

vunit V1 (M) {
    property P =
        never (a && b);
    assert P abort reset_n
      @(posedge clk);
}
```

VHDL PSL

```
...
    port (a,b,reset_n: bit);
end;

vunit V1 (E) {
    property P =
        never (a and b);
    assert P abort reset_n
      @(posedge clk);
}
```

The Incisive simulator allows SVA constructs to be included in verification units, as described in "Putting SVA Assertions in a PSL Verilog Verification Unit". This capability is not a SystemVerilog or PSL standard.

In a similar fashion, SVA constructs can be placed in a separate module that is then bound to the design module. For example:

```
module M (a,b,reset_n)
     ...
endmodule

module Properties (a,b,reset_n)
    property P;
     disable iff (reset_n) !(a&&b);
    endproperty: P
    assert property (@(posedge clk) P);
endmodule

bind M Properties my_props(a,b,reset_n);
```

For SVA, the module to be bound becomes an instance of the target module--that is, it becomes a new level of hierarchy. For PSL, if the `vunit` is bound to an instance, it appears as an instance in the module to which it is bound.

12

Dynamic ABV Implementation Guidelines

The following examples demonstrate how to get from a requirements specification to assertions. You can check many requirements by taking advantage of a small subset of assertion constructs.

This appendix provides examples of how to create assertions based on commonly-used phrases that appear in specifications for a device under verification. It shows efficient implementations for both PSL and SVA, making it a good reference for comparing and contrasting the PSL and SVA assertion languages.

This appendix starts by summarizing the steps for constructing an assertion. It then lists many common phrases that appear in requirement specifications and, for each one, shows how to construct an assertion that checks the specification. It also notes the implicit property coverage that is provided by the Incisive tools, and points out where additional coverage metrics are useful.

After showing how to construct properties, this appendix provides a checklist of the types of requirements to consider checking. It also provides coverage considerations, reuse considerations, performance implications, and other general tips.

To make adoption easier, you can take advantage of a small subset of assertion constructs right away, using these templates, then add more assertions later. It is also recommended that you use component checks from the Incisive Assertion Library (IAL), such as the ones for FIFO behavior and arbitration. For more information about IAL, see the *Incisive Assertion Library Reference* .

Creating an Assertion

Creating assertions is easy if you use the following steps:

1. Restate the requirement in the form of "if <>, then <>".
   This statement specifies a condition to be checked.
2. Decide when to check the condition.
   The condition you defined becomes the enabling condition for the check, and will be the expression on the left of the implication operator--that is, it is the `if` clause of the requirement. The simulator increments the Checked count when this condition is true. Consider overlapping conditions that might slow performance. For details, see "Assertion Coding Guidelines".
3. Decide how to express the condition to be checked.
4. Name your assertion.

Create a name that describes the correct behavior that the assertion is designed to check. Naming conventions can be useful for readability, and for configuring their operation in different tools. For details, refer to *ABV Methodology* in your Incisive verification kit.

5. Decide which verification directive to use.
   For checks, you can use either `assert` or `assume`:
   - Use the assert directive when the property expresses the behavior of the device under test.
   - Use the assume directive when the property expresses the behavior of the environment that generates stimulus to the device under test.
     Use the `cover` directive when the goal is to determine whether the behavior has been tested by the simulation stimulus or, in the case of formal verification, to determine if the behavior can possibly occur.

     For more information, see "When to Use SVA assert versus cover".

     Note: For formal verification, only assertions at the top level of the block being formally verified can use assume, because these assumptions specify environment constraints for the formal analysis. Because the top level often varies, the `assert` directive is recommended for all assertions, and a naming convention can be used to distinguish whether it is an assertion or an assumption, relative to the module in which it is embedded. You can then override the verification directive independently from the code for that top level when you run formal verification.
6. Decide what type of check to use:
   - For PSL, decide whether to use `always`, `never`, or a one-time check (see "Writing One-Time Checks in PSL").
   - For SVA, decide whether to use a one-time check, a concurrent assertion that is checked on every clock, or an immediate assertion in an `always`block.

     Note: SVA immediate assertions and PSL `never` assertions do not provide implicit coverage.
7. Consider how to clock the assertion.
   Often, defining a default clock for the module is easiest. An assertion that is strictly combinatorial--for example, has no clock and is never disabled or aborted--will be checked whenever one of the signals it references changes value. The Checked count will always be 0. Incisive coverage tools filter out these assertions.
8. Consider when the assertion must *not* be checked:
   - Use the `abort` or `until` clause in PSL.
   - Use the `disable iff` clause in SVA.

Assertion Templates for Checking Common Requirements

This section shows how to implement PSL and SVA checks for common phrases that are found in requirements specifications. For simplicity, unless otherwise noted

- All assertions are clocked on the rising edge of clk.
- The property is not checked when reset is high.
- A cycle is defined by (posedge clk).

Implicit coverage--that is, Checked and Finished counts--are also described.

The Checked count indicates the number of times the property was enabled, which includes any aborts or disables.

The following terminology is used in the examples:

X, Y, Z	`X`, `Y`, or `Z` can be a Boolean or sequence expression
BoolX, BoolY	`X` or `Y` is a Boolean expression
SeqX, SeqY	`X` or `Y` is a sequence expression
BitX, BitY	`X` or `Y` is a single-bit signal
VecX, VecY	`X` or `Y` is a multi-bit signal
SigX, SigY	`X` or `Y` is a single-bit or multi-bit signal
clk	Clock
n , *n1* , *n2* eval_a	Predefined constants Evaluation of expression `a`
	Expression, property, or sequence evaluates to TRUE. Green is used to show when a property finishes successfully. Black is used for expressions or sequences.

	Property evaluates to FALSE.

When a Boolean expression is required, the `BoolX` and `BoolY` terms are used. When a sequence expression is required, the `SeqX` and `SeqY` terms are used. The `X`, `Y`, and `Z` terms can be Booleans or sequence expressions, and are used in the drawings to represent both. It is important to understand which alternative is applicable, especially for PSL, where type checking determines which implication operator is used.

The following is a list of the templates that are provided:

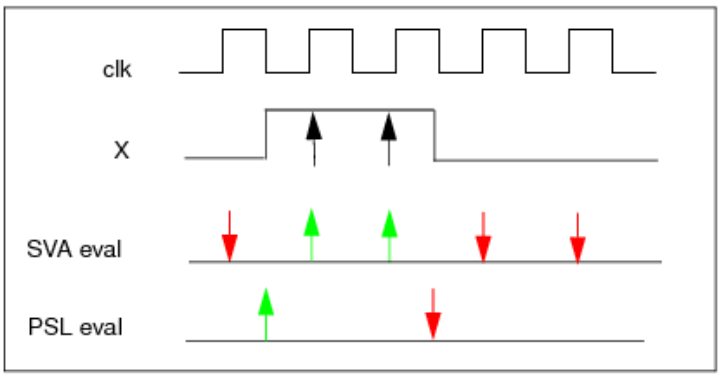
- Invariants
  - X must always be 1.
  - Control signals are always legal values.
  - X must never occur.
  - X must always be equal to Y .
  - Configuration mode must always be valid.
- Simple implications
  - Y occurs when X occurs.
  - Data signals are valid when the read or write signal transitions low.
  - Legal status is read from the status register.
- Next-cycle response
  - Y occurs one cycle after X occurs.
- Time-bounded responses
  - Y occurs n cycles after X occurs.
  - Y must not occur for n cycles after X occurs.
  - sigX is high for a maximum of n cycles.
  - sigX is high for a minimum of n cycles.
  - Y occurs within n cycles of X .
  - Y must occur within n1 to n2 cycles after X occurs.
  - Z occurs within n1 to n2 cycles after X or Y occurs.
  - Y occurs after n cycles with no occurrence of X .
- Event-bounded responses
  - Y occurs before Z after X occurs.
  - Y occurs until Z after X occurs.
- Unbounded responses
  - Y occurs sometime after X occurs.

    - Y occurs sometime after X occurs, with no occurrence of Z or X in between.
- One-time checks
    - The value of parameter X is less than 3.

Invariants

`X` must always be 1.

PSL Alternatives	SVA Alternatives
`BoolX == 1`	`BoolX == 1`



For PSL, assertions that have no clock are evaluated when a signal of the assertion changes value.

PSL Assertion

`X` is a Boolean:

```
// psl BoolX_is_1: assert always (BoolX == 1) abort reset;
```

SVA Assertion

```
BoolX_is_1: assert property (
@(posedge clk) disable iff (reset)
(BoolX == 1));
```

Implicit Coverage

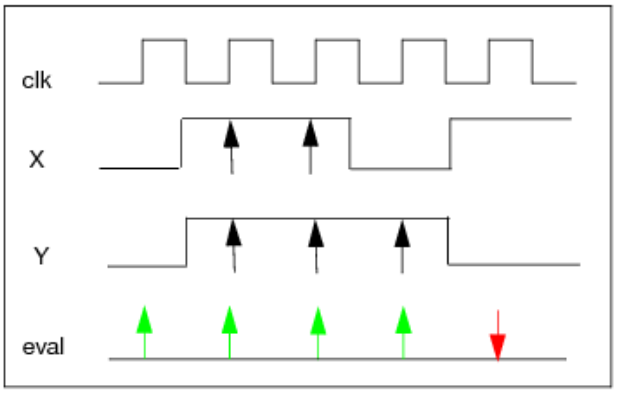
For PSL, the Checked count is incremented every time BoolX or reset change value. The Finished count is incremented each time the property is checked while `BoolX` is 1 and reset is low.

For SVA, because all assertions are clocked, the Checked count is incremented at every `posedge clk`

while `reset` is low. The Finished count is incremented at every posedge clk where reset is low and BoolX is one.

Control signals are always legal values.
→ Restated: A signal has a legal value, where legal means it is a known value.

PSL Alternatives	SVA Alternatives
`!$isunknown(X) && !$isunknown(Y)` `never(isunknown(X)|| isunknown(Y))`	`!$isunknown(X) && !$isunknown(Y)`



Lump as many of these valid data assertions together as possible, because they are generally test issues that get fixed early on, and are not likely to reoccur. Also, if there is more than one assertion, choose label names that will group them together in the Assertion Browser and summary reports.

PSL Assertion

```
// psl valid_X_and_Y: assert never
((isunknown(sigX) || isunknown(sigY)) && !reset) @(rose clk);
```

SVA Assertion

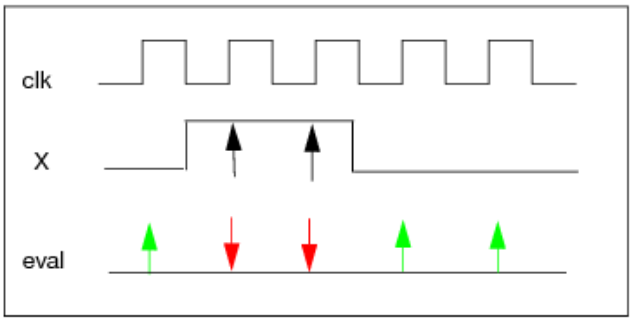
```
valid_X_and_Y: assert property (
@(posedge clk) disable iff(reset)
(!$isunknown(sigX) && !$isunknown(sigY)) );
```

Implicit Coverage

When PSL `never` is used, coverage information is not provided. The Checked count is the number of posedge clks when reset is low. The Finished count is 0. For SVA, and for PSL when `always` is used, the Checked count is incremented at every posedge clk when reset is low. The Finished count is incremented at every posedge clk when reset is low and X and Y are not unknown.

X must never occur.

PSL Alternatives	SVA Alternatives
`never X`	`not (X)`



PSL Assertion

```
// psl X_must_never_occur: assert
 never (X && !reset) @(posedge clk);
```

Note: If **`X`** is a sequence, the term of the sequence must be ANDed with **`reset`** .

SVA Assertion

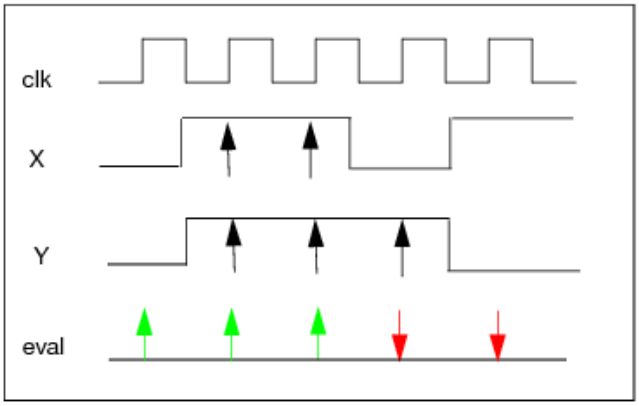
```
X_must_never_occur: assert property (
 @(posedge clk) disable iff (reset)
 not(X) );
```

Implicit Coverage

The Checked count and the Finished count are the number of posedge clks when reset is low.

`X` must always be equal to `Y`.

PSL Alternatives	SVA Alternatives
`BoolX == BoolY`	`BoolX == BoolY`



Note: To avoid errors when `X` and `Y` change value, it is important to clock this assertion.

PSL Assertion

`X` is a Boolean:

```
// psl BoolX_eq_BoolY:
assert always ((BoolX == BoolY) abort reset) @(posedge clk);
```

SVA Assertion

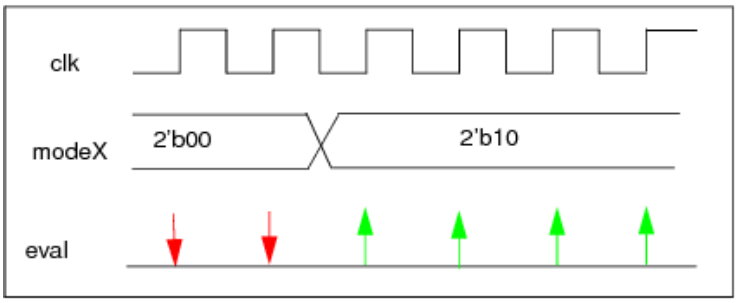
```
BoolX_eq_BoolY: assert property (
@(posedge clk) disable iff(reset) (BoolX == BoolY) );
```

Implicit Coverage

The Checked count is the number of posedge clks when reset is low. The Finished count is incremented at every posedge clk when reset is low and BoolX equals BoolY.

Configuration mode must always be valid.
→ Restated: `modeX` must always be valid, assuming that valid values are `01` and `10`.

PSL Alternatives	SVA Alternatives
`(modeX == 2'b01 || modeX == 2'b10)`	`(modeX == 2'b01 || modeX == 2'b10)`



Note: Additional coverage might be desired to determine which modes were exercised.

PSL Assertion

```
// psl valid_mode_X: assert always (
(modeX == 2'b01 || modeX == 2'b10) abort reset @(posedge clk);
```

SVA Assertion

```
valid_mode_X: assert property
(@(posedge clk) disable iff(reset)
(modeX == 2'b01 || modeX == 2'b10));
```

Implicit Coverage

The Checked count is the number of posedge clks when reset is low. The Finished count is incremented at every posedge clk when reset is low and modeX is 2'b01 or 2'b10.

PSL Additional Recommended Coverage

```
// psl mode_01_cover: cover {modeX == 2'b01 && !reset} @(posedge clk);
// psl mode_10_cover: cover {modeX == 2'b10 && !reset} @(posedge clk);
```

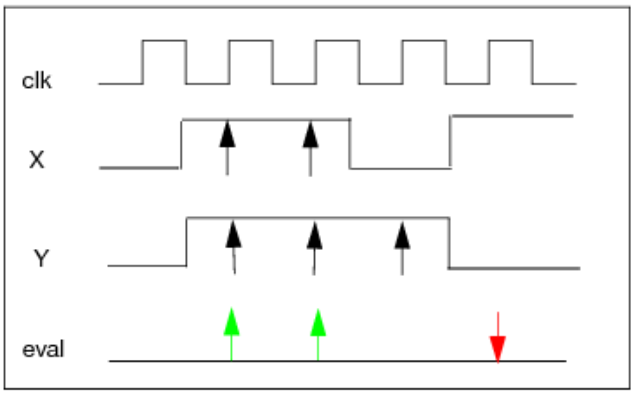
SVA Additional Recommended Coverage

```
cover_mode_01: cover property (
@(posedge clk) disable iff (reset) modeX == 2'b01);
cover_mode_10: cover property (
@(posedge clk) disable iff (reset) modeX == 2'b10);
```

Simple Implications

`Y` occurs when `X` occurs.
→ Restated: If `X`, then `Y`.

PSL Alternatives	SVA Alternatives
`BoolX -> Y` `{X} |-> Y`	`X |-> Y`



PSL Assertion

```
// psl if_X_then_Y:
assert always (({X} |-> Y) abort reset) @(posedge clk);
```

SVA Assertion

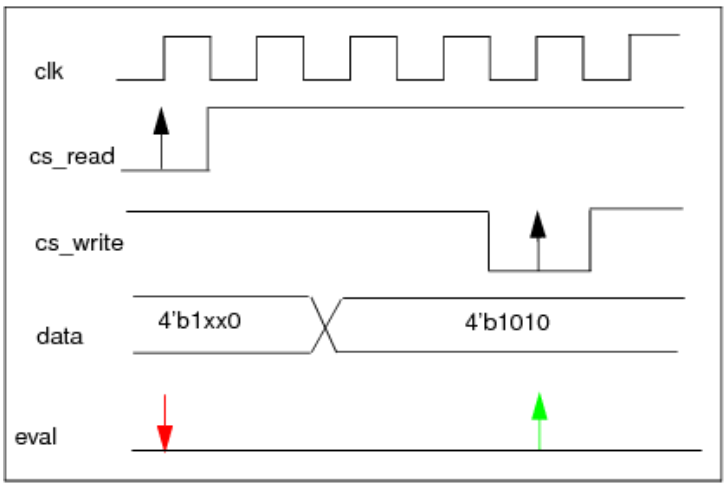
```
if_X_then_Y: assert property (
  @(posedge clk) disable iff(reset) (X |-> Y) );
```

Implicit Coverage

The Checked count is incremented at every posedge clk when X is true and reset is low. The Finished count is incremented at every posedge clk when X and Y are both true and reset is low.

Data signals are valid when the read or write signal transitions low.
→ Restated: If `cs_read` or `cs_write` goes low, `data` is valid.

PSL Alternatives	SVA Alternatives
`(fell(cs_write) || fell(cs_read)) -> !isunknown(data)`	`($fell(cs_write) || $fell(cs_read)) \|-> (!$isunknown(data))`



If you can ensure that the read and write signals will never go unknown and will never have glitches, you can define this behavior with an invariant assertion that is clocked when read or write goes low.

PSL Assertion

```
// psl valid_data: assert always
((fell(cs_write) || fell(cs_read)) -> !isunkown(data)
abort reset) @(posedge clk);
```

SVA Assertion

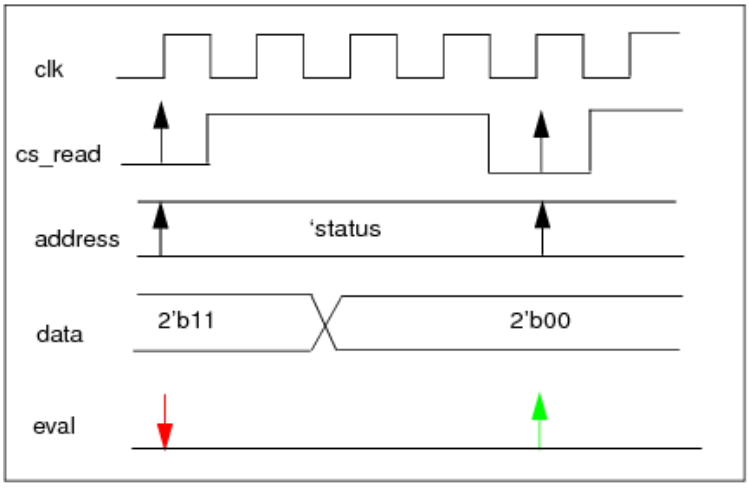
```
valid_data: assert property( @(posedge clk) disable iff (reset)
($fell(cs_write) || $fell(cs_read)) |-> (!$isunknown(data)) );
```

Implicit Coverage

Implicit coverage tells you how many read and write operations occurred. The Checked count is incremented at every posedge clk when the read or write signal transitions to low and reset is low. The Finished count is incremented at every posedge clk when the cs_read or cs_write signal transitions to low, reset is low, and `data` is a known value.

Legal status is read from the status register.
→ Restated: If the status register is read, data is valid, assuming that valid <2, read is 0.

PSL Alternatives	SVA Alternatives
`(address == `status) && (cs_read == 0) -> (data < 2)`	`(address == `status) && (cs_read == 0) |->(data < 2)`



Note: Implicit coverage indicates the number of times the status register was read.

Additional coverage might be used to determine if all legal status values occurred.

PSL Assertion

```
// psl valid_status: assert always (
(address == `status) && (cs_read == 0) -> (data < 2)
abort(reset)) @(posedge clk);
```

SVA Assertion

```
status_valid: assert property
  (@(posedge clk) disable iff (reset)
 ((address == `Qstatus) && (cs_read == 0)) |-> (data < 2 ) );
```

Implicit Coverage

Implicit coverage indicates the number of times the status register is read. The Checked count is incremented at every `posedge clk` when `reset` is low and a read of the status register occurs. The Finished count is incremented at every posedge clk when reset is low, a read of the status register occurs, and the data value is less than 2.

Additional Recommended PSL Coverage

```
// psl status_vecX_01_cover: cover {
 address == `stat && cs_read == 0 && vecX == 2'b01 && !reset} @(posedge clk);
// psl status_vecX_00_cover: cover {
 address == `stat && cs_read == 0 && vecX == 2'b00 && !reset} @(posedge clk);
```

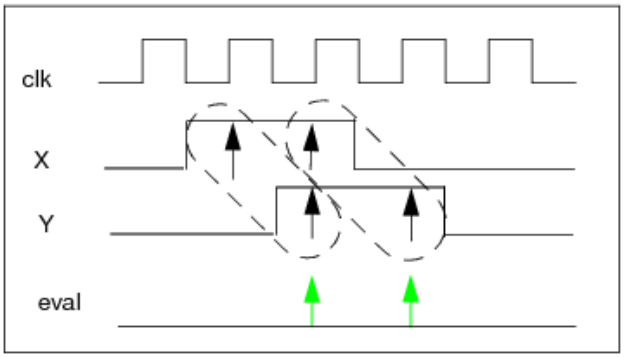
Additional Recommended SVA Coverage

```
status_vecX_01_cover: cover property (
 @(posedge clk) disable iff (reset)
 address == `stat && cs_read == 0 && vecX == 2'b01);
status_vecX_00_cover: cover property (
 @(posedge clk) disable iff (reset)
 address == `stat && cs_read == 0 && vecX == 2'b00);
```

Next-Cycle Response

`Y` occurs one cycle after `X` occurs.
→ Restated: If `X`, then `Y` occurs one cycle after.

PSL Alternatives	SVA Alternatives
`X -> next Y` `{X} |=> Y`	`X |=> Y`



PSL Assertions

X is a Boolean:

```
// psl if_BoolX_then_Next_Y:
  assert always ((BoolX -> next Y) abort reset) @(posedge clk);
```

X is a sequence:

```
// psl if_SeqX_then_Next_Y:
  assert always ((SeqX |=> Y) abort reset) @(posedge clk);
```

SVA Assertion

```
if_X_then_next_Y: assert property (
  @(posedge clk) disable iff(reset) (X |=> Y));
```

Implicit Coverage

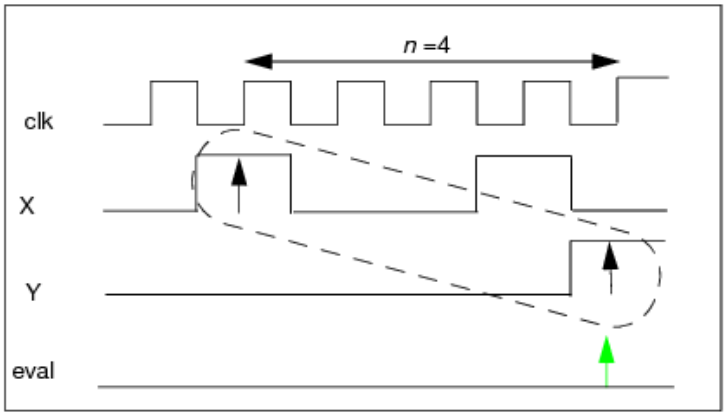
The Checked count is incremented when (`X` and `!reset`) is true. The Finished count is incremented when `Y` and `!reset` occurs on the cycle after `X and !reset`.

Time-Bounded Responses

Y occurs *n* cycles after X occurs.
→ Restated: If X occurs, then Y occurs *n* cycles after.

PSL Alternatives	SVA Alternatives

`BoolX -> next[n ] Y`
`BoolX -> { [*n] ; Y }`
`SeqX |-> { [*n] ; Y }`
`{X} |-> { [*n] ; Y }`
	`X |-> ##n Y`



Note: For *n*=0, the last cycle of X and the first cycle of Y are simultaneous.

PSL Assertions

X is a Boolean:

```
// psl if_BoolX_then_Next_n__Y:
  assert always ((BoolX -> next [n] Y) abort reset) @(rose(clk));
```

X is a sequence:

```
// psl if_SeqX_then_Next_n_Y:
  assert always ((SeqX |-> { [*n] ; Y } abort reset) @(rose(clk));
```

SVA Assertion

```
if_X_then_next_Y: assert property (
  @(posedge clk) disable iff(reset) (X |-> ##n Y));
```

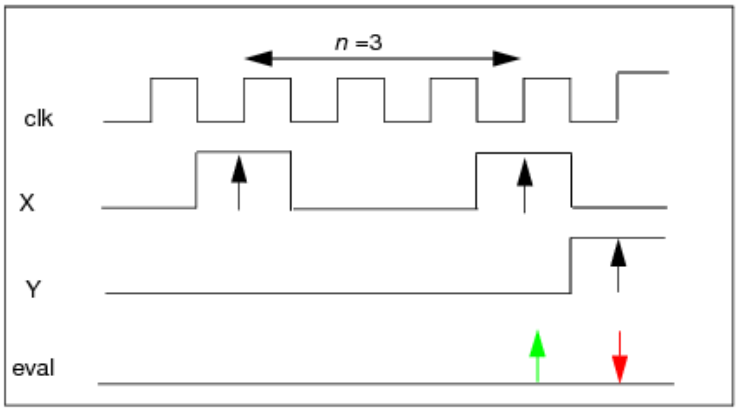
Implicit Coverage

The Checked count is incremented when (`X` and `!reset`) is true. The Finished count is incremented when `Y` occurs on the *n* th cycle after `X`, while !reset is true throughout.

`Y` must not occur for *n* cycles after `X` occurs.
→ Restated: if `X` occurs, then `Y` must not occur for *n* cycles.

PSL Alternatives	SVA Alternatives

`(boolX -> next !boolY[*n ])`
 	
`(seqX |=> !boolY[*n ])`
	`X |=> !Y[*n ]`



PSL Assertions

`X` is a Boolean

```
// psl if_X_then_noY_for_n: assert always (
    (boolX -> next !boolY[*n]) abort reset) @(posedge clk);
```

`X` is a sequence

```
// psl if_X_then_noY_for_n: assert always (
    (seqX |=> !boolY[*n]) abort reset) @(posedge clk);
```

SVA Assertion

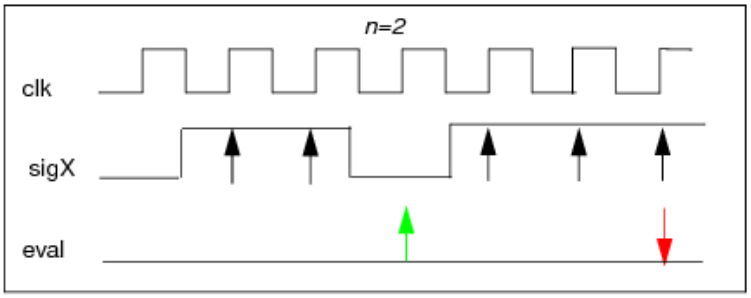
```
if_X_then_noY_for_n: assert property (
 @(posedge clk) disable iff(reset) (X |=> !Y[*n ]) );
```

Implicit Coverage

The Checked count is incremented on the posedge clk when `X` is true while `reset` is low. The Finished count is incremented on the posedge clk when `Y` does not occur for *n* cycles.

`sigX` is high for a maximum of *n* cycles.
→ Restated: if `X` rises, it must be high for a maximum of *n* .

PSL Alternatives	SVA Alternatives
`rose(sigX) -> {sigX[*1:n]; !sigX}`	`$rose(sigX) |-> (sigX[*1:n] ##1 !sigX)`



Note: Implicit coverage tells you how many times `sigX` rose. Additional recommended coverage is the number of times `sigX` was high for the maximum number of times.

PSL Assertion

```
// psl sigX_high_max_n: assert always (
 (rose(sigX) -> {sigX[*1:n]; !sigX}) abort reset) @(posedge clk);
```

SVA Assertion

```
sigX_high_max_n: assert property ( @(posedge clk) disable iff(reset)
  ($rose(sigX) |-> ( sigX[*1:n] ##1 !sigX )));
```

Implicit Coverage

The Checked count is incremented when sig`X` rises and reset is low. The Finished count is incremented at the first `posedge clk` when `X` goes low, assuming X was high for n cycles or less.

Additional PSL Recommended Coverage

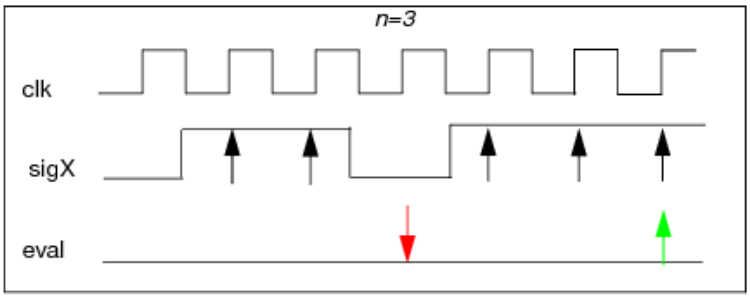
```
// psl corner_case_sigX_is_max: cover {
     {rose(sigX && !reset)}; sigX[*n-1]; !sigX} @(posedge clk);
```

Additional SVA Recommended Coverage

```
corner_case_sigX_is_max: cover property (
   @(posedge clk) disable iff (reset)
   $rose(sigX) ##1 sigX[*n-1] ##1 !sigX );
```

`sigX` is high for a minimum of *n* cycles.
→ Restated: if `X` rises, it must be high for a minimum of *n* cycles.

PSL Alternatives	SVA Alternatives
`rose(sigX) -> sigX[*n]`	`$rose(sigX) |-> sigX[*n]`



Note: Implicit coverage tells you how many times `sigX` rose. Additional recommended coverage is the number of times `sigX` was high for the minimum number of times.

PSL Assertion

```
// psl sigX_high_min_n: assert always (rose(sigX) -> sigX[*n] abort reset) @(posedge
clk);
```

SVA Assertion

```
sigX_high_min_n: assert property ( @(posedge clk) disable iff(reset)
$rose(sigX) |-> sigX[*n] );
```

Implicit Coverage

The Checked count is incremented on the `posedge clk` when `sigX` rises and `reset` is low. The Finished count is incremented when `sigX` is active *n* times, assuming that `reset` remained low throughout.

Additional PSL Recommended Coverage

```
// psl corner_case_sigX_is_min: cover {{!reset && rose(sigX)};X[*n-1];!sigX}
```

Additional SVA Recommended Coverage

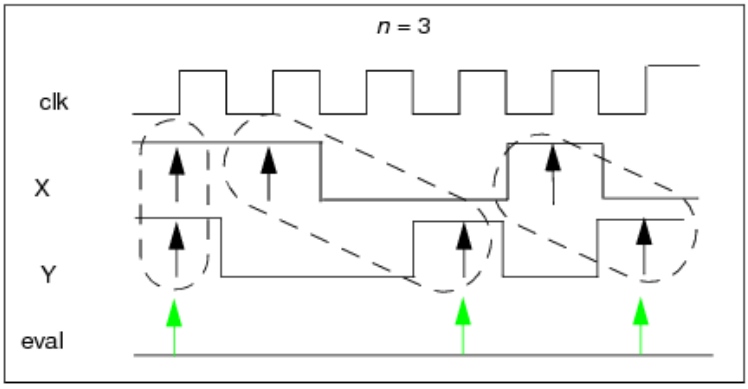
```
corner_case_sigX_is_min: cover property (
@(posedge clk) disable iff (reset)
($rose(sigX) ##1 sigX[*n-1] ##1 !sigX));
```

Implicit Coverage

The Checked count is incremented on the `posedge clk` when (`X` or `Y`) is true and `reset` is low. The Finished count is incremented when `Z` occurs exactly *n* cycles after either `X` or `Y` occurs while `reset` remains low.

`Y` occurs within *n* cycles of `X`.
→ Restated: If `X`, then `Y` occurs within *n* cycles.

PSL Alternatives	SVA Alternatives
`BoolX -> {[*0:n];BoolY}  n >= 0`	`X |-> ##[0:n] boolY  n >= 0`
`SeqX |-> {[*0:n];BoolY}  n >= 0`	`X |-> (Y within 1'b1[*n])  n > 0`
`BoolX -> {SeqY within [*n]}  n > 0`	
`SeqX |-> {SeqY within [*n]}  n > 0`	



PSL Assertion

```
// psl if_X_then_Y_within_n:
  assert always (({X} |-> {Y within [*n]}) abort reset) @(rose(clk));
```

SVA Assertion

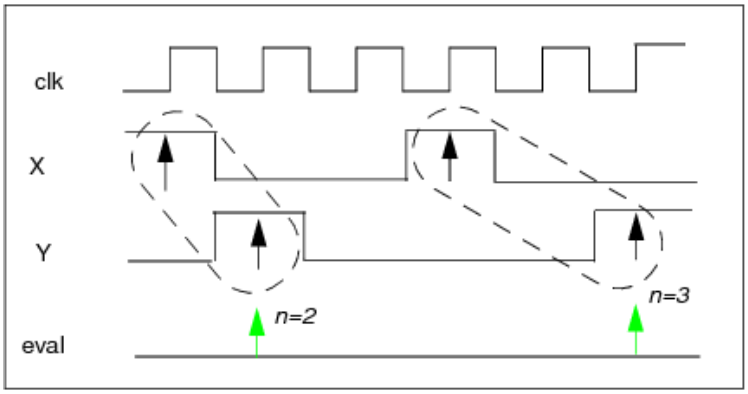
```
if_X_then_Y_within_n: assert property
  (@(posedge clk) disable iff(reset)
  X |-> (Y within 1'b1[*n]));
```

Implicit Coverage

The Checked count is incremented on the posedge clk when (`X` and `!reset`) is true. The Finished count is incremented on the posedge clk when (`Y` and `!reset`) occurs on the *n* th cycle after `X`.

`Y` must occur within *n1* to *n2* cycles after `X` occurs.
→ Restated: If X, then Y must occur within n1 to n2 cycles.

PSL	SVA
`boolX -> {[*n1 :n2 ]; Y}` *n1* >=	`X |-> ##[n1 :n2 ] Y` *n1* >=0
`seqX |-> {[*n1 :n2 ]; Y}` *n1* >= 0	`X |-> (Y within 1'b1[*n])` *n1* > 0
`BoolX -> {SeqY within [*n1:n2]}` *n1* > 0	
`SeqX |-> {SeqY within [*n1:n2]}` *n1* > 0	



PSL Assertions

X is a Boolean

```
// psl if_BoolX_then_Y_within_n1_to_n2: assert always (
  (boolX -> {[*n1 :n2 ]; Y}) abort reset) @ rose(clk);
```

X is a sequence:

```
// psl if_SeqX_then_Y_within_n1_to_n2: assert always (
  (seqX |-> {[*n1 :n2 ]; Y}) abort reset) @ rose(clk);
```

SVA Assertion

```
if_X_then_Y_within_n1_to_n2: assert property(
  @(posedge clk) disable iff(reset) (X |-> ##[n1 :n2 ] Y));
```

Implicit Coverage

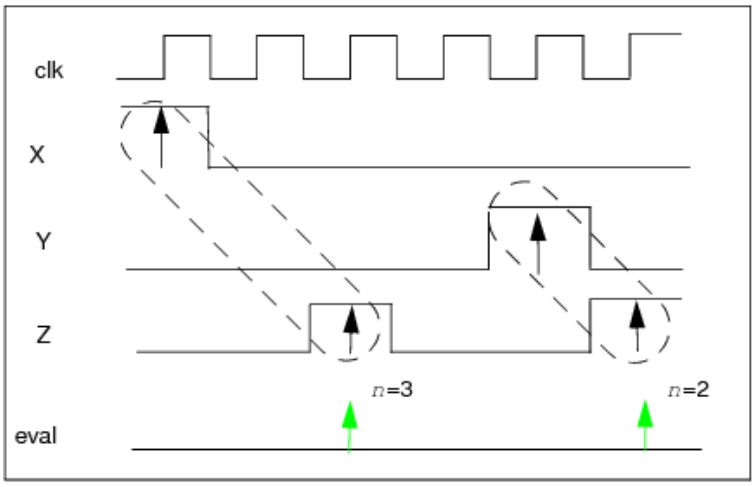
The Checked count is incremented on the posedge clk when `X` is true while `reset` is low. The Finished count is incremented on the posedge clk if `Y` occurs when the assertion is active, assuming that `reset` remains low.

`Z` occurs within *n1* to *n2* cycles after `X` or `Y` occurs.
→ Restated: If `X` or `Y` occurs, `Z` occurs within *n1* to *n2* cycles after.

PSL Alternatives **SVA Alternatives**

```
(BoolX || BoolY) -> {[*n1:n2]; Z}

{{SeqX} | {SeqY}} |-> {[*n1:n2]; Z}

(SeqX or SeqY) |->
{Z within [*n1:n2]}
```

```
(BoolX || BoolY) |-> ##[n1:n2] Z

(SeqX or SeqY) |-> ##[n1:n2] Z

(SeqX or SeqY) |->
  Z within 1'b1[*n1:n2]
```



PSL Assertions

X and Y are Booleans

```
// psl assert_if_XorY_then_Z_n1_to_n2_cycles_after:   assert
always ((BoolX || BoolY) -> {[*n1:n2]; Z}) abort reset) @(rose(clk));
```

X or Y is a sequence

```
// psl assert_if_XorY_then_Z_n1_to_n2_cycles_after:   assert
always ({{SeqX} | {SeqY}} |-> {[*n1:n2]; Z}) abort reset) @(rose(clk));
```

SVA Assertions

X and Y are Booleans

```
assert_if_BoolXorY_then_Z_n1_to_n2_cycles_after: assert property (
  @(posedge clk) disable iff(reset) ((BoolX || BoolY) |-> ##[n1:n2] Z));
```

X or Y is a sequence

```
assert_if_SeqXorY_then_Z_n1_to_n2_cycles_after: assert property (
  @(posedge clk) disable iff(reset) ((SeqX or SeqY) |-> ##[n1:n2] Z));
```

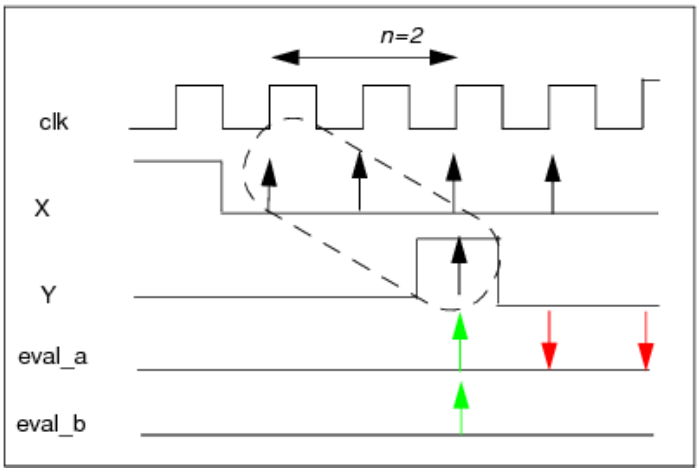
Implicit Coverage

The Checked count is incremented on the posedge clk when (`X` and `!reset`) or (`Y` and `!reset`) is true.

The Finished count is incremented on the posedge clk when `Z` occurs the first time between the n_1 st and n_2 nd time, assuming `!reset` throughout.

`Y` occurs after *n* cycles with no occurrence of `X`.
→ Restated: If `X` does not occur for *n* cycles, then `Y` must occur.

PSL Alternatives	SVA Alternatives
`a: (!boolX)[*n] |=> Y`	`a: (!boolX)[*n] |=> Y`
`b: {rose(!boolX); !boolX[*n-1]} |=> Y`	`b: $rose(!boolX) ##1 !boolX[*n-1] |=> Y`



Note: Using `fell` in expression b to qualify the start prevents overlapping assertions.

For expression a, the assertion to the left of the operator passes once and fails twice, as shown in `eval_a`. Expression b will pass as shown in `eval_b`.

PSL Assertion

`X` is a Boolean

```
// psl assert_if_notX_for_n_then_Y:   assert
always (({rose(!boolX);!boolX[*n-1]} |=> Y) abort reset) @(posedge clk);
```

SVA Assertion

`X` is a Boolean

```
assert_if_notX_for_n_then_Y: assert property (
  @(posedge clk) disable iff(reset)
  $rose(!boolX) ##1 !boolX[*n-1] |=> Y );
```

Implicit Coverage

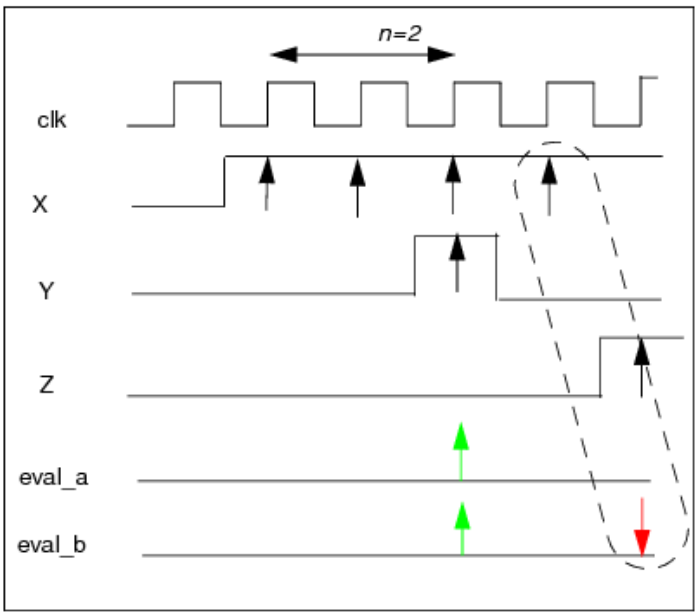
The Checked count is incremented on the posedge clk when `X` transitions low and stays low for *`n`* cycles. The Finished count is incremented on the posedge clk when `X` remains low for n clocks followed by `Y`, while `!reset` is true.

Event-Bounded Responses

Y occurs before Z after X occurs.
→ Restated: If `x` occurs, then Y occurs before Z

PSL Alternatives	SVA Alternatives

`a: rose(boolX) -> boolY before Z`
`b: boolX -> boolY before Z`
`c: {seqX} |-> boolY before Z`
	`a: boolX |-> !boolZ[*0:$] ##1 boolY`



Note: Using `rose` in expression `a` to qualify the start prevents overlapping assertions. You can miss failures as shown in `eval_b` if the protocol does not ensure that `boolX` returns to 0 before the start of the next evaluation.

PSL Assertion

`x` is a Boolean

```
// psl assert_if_X_then_Y_before_Z:   assert
always (({rose(boolX)} |-> Y before Z) abort reset) @(posedge clk);
```

SVA Assertion

`x` is a Boolean

```
assert_if_X_then_Y_before_Z: assert property (
```

```
@(posedge clk) disable iff(reset)
$rose(boolX) |-> !boolZ[*0:$] ##1 boolY );
```

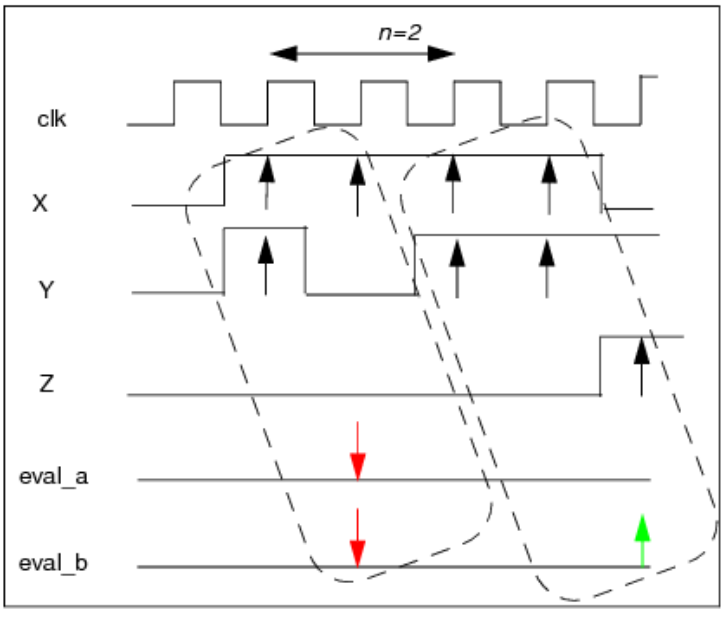
Implicit Coverage

The Checked count is incremented on the posedge clk when `X` transitions high while `reset=0`. The Finished count is incremented on the posedge clk after `X` transitions high and `Y && !Z` occurs, assuming `reset=0` and `Z=0`, because `X` transitioned high. The Failed count is incremented on the posedge clk when `X` transitions high and `Z=1` occurs before or at the same time that `Y=1`, assuming that `reset=0` and `Z=0`, because `X` transitioned high.

Y occurs until Z after X occurs.
→ Restated: If `X` occurs, then Y occurs until Z

PSL Alternatives	SVA Alternatives

`a: rose(boolX) -> boolY until Z`
`b: boolX -> boolY until Z`
`c: {seqX} |-> boolY until Z`
	`boolX |-> boolY[*0:$] ##1 boolZ`



Note: Using `rose` in expression `b` to qualify the start prevents overlapping assertions. You can miss failures as shown in `eval_b` if the protocol does not ensure that `boolX` returns to 0 before the start of the next evaluation.

PSL Assertion

X is a Boolean

```
// psl assert_if_X_then_Y_until_Z:   assert
always (({rose(boolX)} |-> Y until Z) abort reset) @(posedge clk);
```

SVA Assertion

X is a Boolean

```
assert_if_X_then_Y_until_Z: assert property (
  @(posedge clk) disable iff(reset)
  $rose(boolX) |-> boolY[*0:$] ##1 boolZ );
```

Implicit Coverage

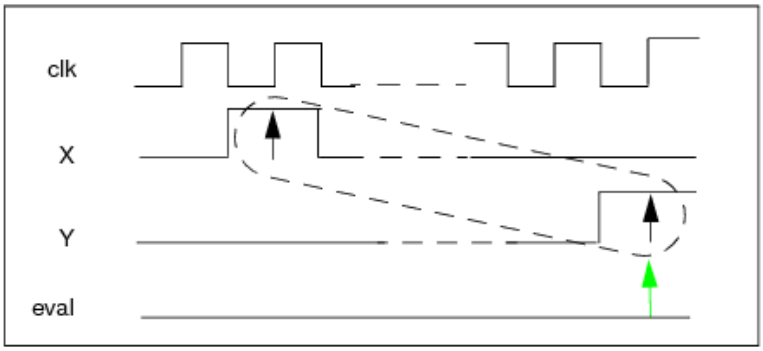
The Checked count is incremented on the posedge clk when X transitions high while reset=0. The Finished count is incremented on the posedge clk after X transitions high and Z occurs, assuming reset=0 and Y=1, because X transitioned high. The Failed count is incremented on the posedge clk when X transitions high and Y=0 occurs before or at the same time that Z=1 occurs, assuming reset=0 and Z=0, because X transitioned high.

Unbounded Responses

Y occurs sometime after X occurs.
→ Restated: If X, then eventually Y.

PSL Alternatives	SVA Alternatives

`BoolX -> next eventually! Y`
`SeqX |=> eventually! Y`
`{X} |=> eventually! Y`
	`X |-> ##[1:$] Y`



Note: With the PSL `eventually!` operator, an error occurs if the condition is not met by the end of simulation. There is currently no SVA equivalent in the Incisive implementation.

PSL Assertions

`X` is a Boolean

```
// psl assert_if_BoolX_eventually_Y:
  assert always ((BoolX -> next eventually! Y) abort reset) @(posedge clk);
```

`X` is a sequence

```
// psl assert_if_SeqX_eventually_Y:
  assert always ((SeqX |=> eventually! Y) abort reset) @(posedge clk);
```

SVA Assertion

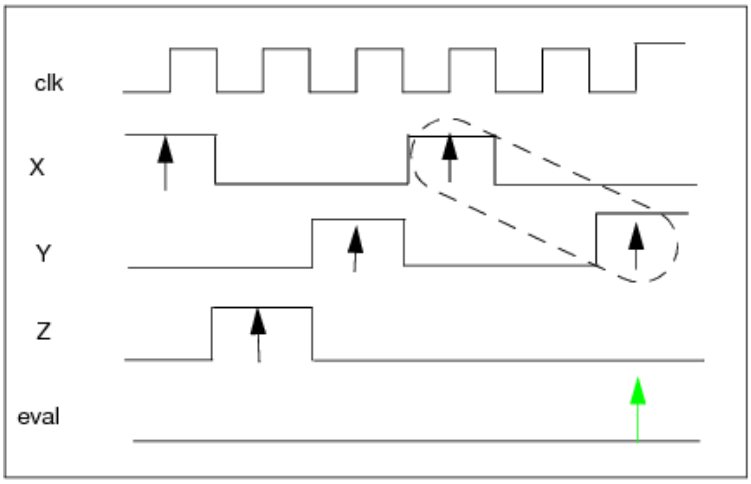
```
assert_if_X_then_next_Y: assert property (
  @(posedge clk) disable iff(reset) (X |-> ##[1:$] Y));
```

Implicit Coverage

The Checked count is incremented on the posedge clk when (`X` and `!reset`) is true. The Finished count is incremented on the posedge clk when `Y` occurs, assuming `!reset` throughout.

`Y` occurs sometime after `X` occurs, with no occurrence of `Z` or `X` in between.
→ Restated: If `X`, then no `X` or `Z` until `Y`.

PSL	SVA
`BoolX -> next{(!boolX && !boolZ)[*];Y}`	`X |=> (!BoolZ && !BoolX)[*0:$] ##1 Y`



PSL Assertion

```
// psl if_X_then_no_XorZ_until_Y:   assert always(
   (BoolX -> next{(!boolX && !boolZ)[*];Y}) abort reset) @(rose(clk));
```

SVA Assertion

```
if_X_then_no_XorZ_until_Y: assert property (
  @(posedge clk) disable iff(reset)
  (BoolX |=> (!BoolZ && !BoolX)[*0:$] ##1 Y) );
```

Implicit Coverage

The Checked count is incremented on the posedge clk when (X and `!reset)` is true. The Finished count is incremented on the posedge clk when the assertion is active and Y occurs.

One-Time Checks

The value of parameter `X` is less than 3.
→ Restated: Parameter `X` < 3.

PSL Alternatives	SVA Alternatives
`(parameter < 3)`	`(parameter < 3)`

PSL Assertion

```
// psl ParamXlessthan3: assert (X < 3) @rose(clk);
```

SVA Assertion

```
Initial
 ParamXlessthan3: assert(X < 3);
```

Implicit Coverage

Checked and Finished counts will be 1.

Suggested Behavior to Check with Assertions

Checks must be derived both from the specification and from the low-level implementation. Assertions can be used to express assumed input behavior, expected output behavior, and forbidden behavior. Behaviors to check for include the following:

- Check that parameters of a component are valid by using a one-time check.
- Check that all registers can be read from and/or written to, and, where possible, check that they operate correctly.
- Check that inputs and outputs behave as described in the specification, especially handshakes. You can derive relationships to test from the waveforms provided in the specification.
- Check bus protocols--for example, required patterns, disallowed patterns, and transaction modes.
- Check for valid data.
- Check that FSMs implement the desired response by writing assertions that describe the behavior of their outputs. These assertions are especially useful for formal analysis.
  For example, if an FSM is designed to ensure that the minimium packet size is 64 bytes, check that the resulting packet is minimally 64 bytes.

  Note: Do not write control-oriented assertions about the states of a state machine. States can generally be verified by a visual inspection of the code, so such assertions add little value. Derive assertions from the functional specification, not the implementation.
- Check that all corner cases, such as minimum and maximum sizes, waits, and minimum IFG, are handled properly.

- Check that illegal inputs can be handled properly.
- Check that forbidden behavior does not occur.
- Use IAL library components to check all common components of the design.

The *Incisive Plan-to-Closure Methodology: Assertion-Based Verification* document also suggests categories of assertions to write, templates to use, and naming conventions based on the category.

Suggested Coverage Monitors for ABV

Coverage points indicate whether the stimulus was able to create the conditions necessary to test the design's behavior. This information is critical to ensuring that the design has been sufficiently tested, with code coverage as a complementary approach. High functional coverage will help to minimize code coverage iterations, and code coverage holes indicate holes in the functional coverage.

Consider using the following functional coverage points:

- Create coverage assertions that represent all the paths through the state machine as the state machine is created.
  FSM transition checks and state checks are not necessary, because they can be done automatically by using code coverage. These assertions can be recorded as transactions for debugging purposes.
- Describe all corner cases and tricky areas that need to be tested, such as minimum and maximum sizes, maximum wait states, minimum gaps, and full and empty FIFO levels.
  It is best that designers write these assertions while they are creating the code, when possible corner cases are fresh in their minds.

  Many of these assertions are implicit from related assertions.
- Make sure that all legal modes of the chip have been exercised
  - When the device is not in reset condition
  - After the device has been configured
- Make sure that all legal register bit combinations have been exercised in normal operation, for all registers, including control and status registers.
  Control register coverage might already be provided by the implicit coverage of related assertion checks.
- Ensure that all types of data transactions, handshakes, and arbitration have occurred.
- Check that there is a good distribution of data values.
  It might be more convenient to use covergroups for this type of check.

- Check important state machine interactions.

For more information about writing assertions for functional coverage, see

- IUS--The *ICC User Guide*
- IES--"Coverage-Driven Verification with Enterprise Manager" in the *Enterprise Manager User Guide*

  Note: You can annotate the coverage measured by PSL and SVA directives into specific sections of your Enterprise Manager verification plan. For details, see "Implementation-Based Mapping" within this topic.

Assertion Reuse Considerations

When adding assertions, it is important to consider reuse--reuse across tools, across technologies (for example, simulation, formal verification, and acceleration), from block level to full chip, from one design to another design, and at different levels of abstraction. For effective reuse, consider the placement of assertions as well as coding guidelines.

Coding Guidelines for Assertion Reuse

Coding guidelines can help to ensure that assertions can be reused across tools, across technologies--simulation, formal verification, acceleration--from block level to full chip, from one design to another design, and at different levels of abstraction. For details, see "Assertion Reuse Tips".

Placing Assertions for Reuse

When deciding where to place an assertion, consider whether the assertion is specific to a single design, or can be reused elsewhere. There are three options for placing assertions:

- Embedded in the design
- In a property file that is bound to the design (PSL `vunit` or SVA `bind`)
- As a standalone component that is instantiated in the design

The advantages and disadvantages of each option are shown in Table B-1.

Table B-1 Techniques for Adding Assertions

	Embedded	Binding Property File	Standalone Verification Component
Pro	Always travels with the design. Assertions that document behavior can be placed near the logic with which it is associated. One file contains all of the information.	No need to modify the design. Easy to categorize assertions for performance optimizations. Easy to separate auxiliary code from design code.	Component is portable. Parser complains if the file is missing, because it is instantiated.
Con	Must guard that supporting HDL code does not get synthesized or affect code coverage results. Assertions can clutter the RTL.	Multiple files are required. Parser will not complain if the file is missing.	More work to instantiate the assertions in the design. An extra level of hierarchy adds a little complexity to any analysis. Multiple files are required.

In general, embed in the design those assertions that are specific to a design and are added by the designer, so they travel with the design. Place interface assertions in a separate verification component, so they can be used across designs.

Assertion Coding Guidelines

This section provides coding guidelines to improve performance and reuse.

ABV Performance Tips

Assertions will increase your simulation run times, but will reduce the number of simulations necessary to debug a design. To make your assertions as efficient as possible, remember the following tips:

- Prevent multiple in-flight assertions--that is, overlapping assertions. Tracking overlapping behaviors is one of the most significant sources of additional computation, and it also affects capacity when the assertion is synthesized for acceleration.
  - Avoid overlapping sequences in the enabling condition of a property.

Sequences that involve `[=] repetitions or ranges of repetition such as [*1:$] might continue matching indefinitely, starting new checks.`

- Avoid extraneous enables--use the `rose`, `fell`, or `stable` functions to enable an assertion check.

  Sequential assertions typically have an enabling condition that must be true in order to check the sequence. When the enabling condition is true in successive cycles, the assertion tracks the overlapping behavior, which takes unnecessary time, and might produce unexpected results.

  For example, the following assertion starts tracking the behavior for every clock in which `request` is high, although all checks will be satisfied by the same grant:

  ```
  assert always (request -> eventually! grant) @(posedge clk);
  ```

  It is more efficient to write the assertion as follows:

  ```
  assert always (rose(request) -> eventually! grant) @(posedge clk);
  ```

  or in SVA:

  ```
  assert property @(posedge clk)($rose(request) -> ##[0:$] grant);
  ```

  Beware of signals that might start high and never fall when using `rose to start the checking of a property. One safeguard is to use $rose(request && !reset) instead of $rose(request).`

- Avoid compound non-determinism--sequences that involve successive terms that have variable repeats, such as `b[*3]` followed by `c[*3]`.

  In this example, if `b` and `c` are high simultaneously for three cycles, the directive needs to track the following, separately and in parallel:

  b[*3] followed by c[*0]
  b[*2] followed by c[*1]
  b[*1] followed by c[*2]
  b[*0] followed by c[*3]

  If the intent is to look for `c` only after `b` goes low, the following is much more efficient:
  `b[*3] followed by (not b and c[*3])`

  - In PSL, do not use `always x -> never y`. For details, see "Using the PSL always and never Operators".
  - Avoid large numbers in the argument of the `prev()` built-in function. For example,

`prev(x,100)` means the value of `x` at 100 clocks previous to the current clock cycle. This can alternately be implemented with a little auxiliary HDL.

- Consider ahead of time whether you want all assertions to be enabled all of the time. Assertions can be enabled and disabled by module (scope), by name, by verification directive, and by omitting an external file. These techniques are generally sufficient, but Verilog `ifdef`s and VHDL `generate`s can allow you to further group categories of assertions.

- Here are some tips about enabling and disabling assertions:
  - Coverage is only important after the bugs are out. You can place coverage directives in external files that are selectively included, or disable all `cover` directives by using the Tcl controls (see "Enabling and Disabling Assertions " in *Assertion Checking in Simulation*).
  - For mature designs that are reused frequently, you will need to enable only the interface assertions that make sure the device is connected properly.
    Check assumptions by using the assume `directive`, so they can be easily distinguished from assertions that check the outputs.
  - Assertions that are created for transaction-viewing purposes need special consideration. A `cover` directive is easiest, but assertion transactions are generally used during debugging, not coverage analysis. It is a good idea to create assertions for transaction viewing by using an assertion that always evaluates to true when the sequence is matched.

Assertion Reuse Tips

- Know the common subset of constructs supported across the tools used in your flow. When possible, constrain all property specifications to these constructs. For assertions containing constructs that are not supported across the tools
  - Use `ifdef`s to optionally enable them, or
  - Place them in an external file that is optionally included
- To reuse properties at different levels of abstraction, specify them in terms of interface signals, so they do not depend on implementation-dependent signals.
- Create parameterized properties and sequences, and place them in SystemVerilog packages that can be used throughout a design team and across projects. See "Using Packages for SVA".
- Use descriptive names for properties and sequences. It is often beneficial to define commonly used sequence elements, and create properties by combining them.

Other Tips for Assertion-Based Verification

A few miscellaneous tips:

- You do not have to select one assertion language to use.
  Both PSL and SVA are very powerful, and can be used simultaneously in a design. PSL has the advantage that it can be used in Verilog, SystemVerilog, VHDL, and SystemC. SVA has the advantage of immediate assertions and action blocks in SystemVerilog code. SVA might also be preferred for SystemVerilog testbenches.

- Use assertion libraries for commonly-used components in the design, like FIFOs and arbiters. IAL is one such library. Libraries provide prepackaged assertions and coverage points. These libraries are also a good reference for learning how to use assertion languages. For more information about IAL, see the Incisive Assertion Library Reference.

- Simplify your assertions by using auxiliary code. It provides a simpler learning curve and the result is sometimes more readable to others. For example:

```
      -- Do not overflow FIFO.

--

overflow <= (write = '1' AND Bfull AND read = '0');


-- psl ASSUME_PSL_NO_OVERFLOW: assume (never overflow) abort(rst);
```

- Where to start adding assertions, and how many to add, often depend on the maturity of the design. Here are some guidelines:
  - For mature designs that are well tested, consider adding interface assertions that check to see whether the device is connected properly.
  - For mostly verified designs, start adding assertions where you have seen problems in the past, and in tricky areas, where there will be the most value.
  - For new designs, all assertions provide value.

- To avoid false assertion firing in simulation and formal analysis, make storage elements (flip-flops) resettable.

- In PSL, use a default clock of `rose(clk)` in simulation to avoid issues at time 0 when signals transition from X. Change the default clock to `posedge(clk)` for formal analysis.
  In SVA, use `$assertoff()` at the start of simulation to prevent firings at t=0.

- Some people find it convenient to use naming conventions to identify assertion categories. For more recommendations, refer to the *Incisive Plan-to-Closure Methodology: Assertion-Based Verification* document.

13

Glossary of ABV Terms

Assertion statement

A statement that a design must exhibit the behavior specified by a given property. Also, a directive to verification tools to verify that the design does not fail to satisfy the specified property, for some or all possible behaviors of the design.

Attempt

In SVA, the number of top-level clocks.

Behavior

The function of a design over time. Design behavior is often partitioned into combinational and sequential behavior. In this context, timing is typically abstracted, so that in various situations instantaneous, cycle-level, or higher-level abstractions of time are considered.

Concurrent assertion

An assertion that runs concurrently with design components, so that it continuously monitors design behavior.

Condition

A situation in which a set of variables or signals satisfy some predicate that specifies a relationship among their respective values at some instant in time. A condition is often represented by a Boolean expression, which evaluates to True when the specified relationship is satisfied. For example, the expression (A==B) is satisfied when the values of A and B are the same.

DUV

Design under verification. Also called DUT for design under test.

Fails

After the enabling condition of an assertion occurs, the fulfilling condition does not occur, or is incomplete.

False

The values 1'b0, 1'bx, 1'bz, or any value interpreted as False when it is the value of an if statement condition.

Finishes

After the enabling condition of an assertion occurs, the fulfilling condition occurs.

Immediate assertion

A procedural assertion that can check only a single combinational condition; it cannot involve temporal operators.

Match

A sequence matches a trace if and only if the trace satisfies the sequence.

Occurs

A Boolean occurs in any cycle in which it evaluates to a True value.

A sequence occurs starting in some cycle and finishing in a (usually later) cycle if the trace starting and ending in the respective cycles satisfies the sequence.

Property

A specification of the temporal relationship among a set of elements, each of which can be a condition, sequence, or (sub)property. A property specifies the behavior of a portion of a design or its environment.

Examples:

```
(ctr==0 before ctr==1)
never {ctr==1; ctr==3}
```

Procedural assertion

An assertion that is evaluated when the flow of control in a procedural block of code reaches the

assertion.

Satisfy

A state satisfies a condition if and only if the condition is True in that state.

A trace satisfies a sequence if and only if some behavior described by the sequence occurs in the trace.

A trace satisfies a property if and only if the behavior described by the property occurs in the trace.

Example:
```
state (a==1, b==2, c==3) satisfies condition (a+b <= c)
```

This trace

- Satisfies the sequence `{ctr==1; ctr==2}`.
- Satisfies the property `(ctr==0 before ctr==1)`.

SERE

Sequential extended regular expression; an expression composed of Boolean expressions describing single-cycle or multiple-cycle behavior.

Sequence

A series of conditions that occur at successive times, particularly when the times are indicated by active clock edges. Also, a form of extended regular expression that defines such a series of conditions, involving operators that indicate concatenation, repetition, and co-occurrence of elements, each of which can be a condition or a (sub)sequence. A sequence typically specifies the behavior of a set of signals or variables, as viewed at successive clock edges.

Example:
```
{ctr==0; ctr==1; ctr==2; ctr==3}[*]
```

State

A unique combination of values of the variables/signals in a design at an instant in time.

Trace (or behavior)

A series of successive states of the design.

The following figure shows a trace for a mod 4 counter, `ctr`:

True

The value 1'b1, or any value interpreted as True when it is the value of an If statement condition.

A Boolean is True in any cycle in which it evaluates to a True value.

A Property is True in the first cycle of a trace that satisfies the Property.

For implication and conditional properties, the value is vacuously true when the left-hand side of the implication is true.

Vacuous

A vacuous pass occurs when a property is not checked, either because it is not enabled or because it is discharged.