Accellera Standard OVL V1
Library Reference Manual

Software Version 1.8

October 18, 2006

STATEMENT OF USE OF ACCELLERA STANDARDS

Accellera Standards documents are developed within Accellera and the Technical Committees of Accellera Organization, Inc. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document. By using an Accellera standard, you agree to defend, indemnify and hold harmless Accellera and their directors, officers, employees and agents from and against all claims and expenses, including attorneys' fees, arising out of your use of an Accellera Standard.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied ?AS IS.?

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Accellera may change the terms and conditions of this Statement of Use from time to time as we see fit and in our sole discretion. Such changes will be effective immediately upon posting, and you agree to the posted changes by continuing your access to or use of an Accellera Standard or any of its content in whatever form.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Organization, 1370 Trancas Street #163, Napa, CA 94558 USA
E-mail: interpret-request@lists.accellera.org

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Overview of this standard

This section describes the purpose and organization of this standard, the Accellera Standard V1 Open Verification Library (Std. OVL) libraries implemented in IEEE Std. 1364-1995 Verilog and SystemVerilog 3.1a, Accellera's extensions to IEEE Std. 1364-2001 Verilog Hardware Description Language and Library Reference Manual (LRM)

Intent and scope of this document

The intent of this standard is to define Std. OVL accurately. Its primary audience is designers, integrators and verification engineers to check for good/bad behavior, and provides a single and vendor-independent interface for design validation using simulation, semi-formal and formal verification techniques. By using a single well-defined interface, the OVL bridges the gap between the different types of verification, making more advanced verification tools and techniques available for non-expert users.

From time to time, it may become necessary to correct and/or clarify portions of this standard. Such corrections and clarifications may be published in separate documents. Such documents modify this standard at the time of their publication and remain in effect until superseded by subsequent documents or until the standard is officially revised.

ACKNOWLEDGEMENTS

These Accellera Standard OVL Libraries and Library Reference Manual (LRM) were specified and developed by experts from many different fields, including design and verification engineers, Electronic Design Automation companies and members of the OVL VSVA technical committee.

The following contributors were involved in the creation of previous versions of the OVL: Shalom Bresticker, Bryan Bullis, Ben Cohen, Harry Foster, Himanshu Goel, Vijay Gupta, Brent Hayhoe, Richard Ho, Narayanan Krishnamurthy, David Lacey, Jim Lewis, Andrew MacCormack, Erich Marschner, Paul Menchini, Torkil Oelgaard, Joseph Richards, Vinaya Singh, Sean Smith, Andy Tsay and others.

The OVL VSVA technical committee and chair reports to Accellera TCC Chairman:

TCC Chairman Johny Srouji / IBM

The following individuals contributed to the creation, editing and review of the Accellera Standard OVL V1 Libraries and LRM

Eduard Cerny/Synopsys

Harry Foster/Jasper Design Automation

Dmitry Korchemny/Intel

Kenneth Elmkjær Larsen/Mentor Graphics (OVL-VSVA Chair)

David Lacey/Hewlett Packard

Uma Polisetti/Agilent

Ramesh Sathianathan/Mentor Graphics

Chris Shaw/Mentor Graphics

Sundaram Subramanian/Mentor Graphics

Bipul Talukdar/Mentor Graphics

Manoj Kumar Thottasseri/Synopsys

Mike Turpin/ARM

Minor version 1.1 released June 2005
Minor version 1.1a released August 2005
Minor version 1.1b released August 2005
Minor version 1.1c released September 2005
Minor version 1.5 released December 2005
Minor version 1.6 released February 2006
Minor version 1.7 released July 2006

Table of Contents

Chapter 1
Introduction

Welcome to the Accellera standard Open Verification Library V1 (OVL). The OVL V1 is composed of a set of assertion checkers that verify specific properties of a design. These assertion checkers are instantiated in the design establishing a single interface for design validation.

The OVL provides designers, integrators and verification engineers with a single, vendor-independent interface for design validation using simulation, hardware acceleration or emulation, formal verification and semi-/hybrid-/dynamic-formal verification tools. By using a single, well defined, interface, the OVL bridges the gap between different types of verification, making more advanced verification tools and techniques available for non-expert users.

This document provides the reader with a set of data sheets that describe the functionality of each assertion checker in the OVL V1, as well as examples that show how to embed these assertion checkers into a design.

About this Manual

It is assumed the reader is familiar with hardware description languages and conventional simulation environments.

This document targets designers, integrators and verification engineers who intend to use the OVL in their verification flow and to tool developers interested in integrating the OVL in their products.

This document has the following chapters:

- OVL Basics

  Fundamental information about the OVL library, including usage and examples.

- OVL Assertion Data Sheets

  Data sheet for each type of OVL assertion checker.

- OVL Defines

  Information about the define values used in general and for configuring the checkers.

Notational Conventions

The following textual conventions are used in this manual:

emphasis	Italics in plain text are used for two purposes: (1) titles of manual chapters and appendixes, and (2) terminology used inside defining sentences.
variable	Italics in courier text indicate a meta-variable. You must replace the meta-variable with a literal value when you use the associated statement.
literal	Regular courier text indicates literal words used in syntax statements or in output.

Syntax statements appear in sans-serif typeface as shown here. In syntax statements, words in italics are meta-variables. You must replace them with relevant literal values. Words in regular (non-italic) sans-serif type are literals. Type them as they appear. Except for the following meta-characters, regular characters in syntax statements are literals. The following meta-characters have the given syntactical meanings. **You do not type these characters.**

[ ]	Square brackets indicate an optional entry.

Verilog Assertion Syntax Format

All Verilog assertion checkers defined by the Open Verification Library initiative observe the following BNF format, defined in compliance with Verilog Module instantiation of the IEEE Standard 1364-1995 *Verilog Hardware Description Language*.

```
assertion_instantiation ::= assert_identifier
   [ parameter_value_assignment ] module_instance ;

parameter_value_assignment ::= #(severity_level
   [,other parameter expressions ], property_type, msg, coverage_level )

module_instance ::= name_of_instance ([list_of_module_connections])

name_of_instance ::= module_instance_identifier

list_of_module_connections ::=
   ordered_port_connection [, ordered_port_connection ]
   | named_port_connection [, named_port_connection ]

ordered_port_connection ::= [expression ]

named_port_connection ::= .port_identifier ([ expression ])

assert_identifier ::= assert_type_identifier

type_identifier ::= identifier
```

References

The following is a list of resources related to design verification and assertion checkers.

- Bening, L. and Foster, H., *Principles of Verifiable RTL Design, a Functional Coding Style Supporting Verification Processes in Verilog*, 2nd Ed., Kluwer Academic Publishers, 2001.
- Bergeron, J., *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers, 2000.
- *CheckerWare Data Book*, Release 2.4, 0-In Functional Verification Group, Mentor Graphics, 2006.
- *Assertions in Simulation User Guide*, Release 2.4, 0-In Functional Verification Group, Mentor Graphics, 2006.
- *Formal Verification User Guide*, Release 2.4, 0-In Functional Verification Group, Mentor Graphics, 2006.

Chapter 2
OVL Basics

The OVL is composed of a set of assertion checkers that verify specific properties of a design. These assertion checkers are instantiated in the design establishing a single interface for design validation.

OVL assertion checkers are instances of modules whose purpose in the design is to guarantee that some conditions hold true. Assertion checkers are composed of one or more properties, a message, a severity and coverage.

- Properties are design attributes that are being verified by an assertion. A property can be classified as a combinational or temporal property.

  A combinational property defines relations between signals during the same clock cycle while a temporal property describes the relation between the signals over several (possibly infinitely many) cycles.
- Message is the string that is displayed in the case of an assertion failure.
- Severity represents whether the error captured by the assertion library is a major or minor problem.
- Coverage consists of one or more flags that indicate whether or not specific corner-case or statistical events occur.

Assertion checkers benefit users by:

- Testing internal points of the design, thus increasing observability of the design.
- Simplifying the diagnosis and detection of bugs by constraining the occurrence of a bug to the assertion checker being checked.
- Allowing designers to use the same assertions for both simulation and formal verification.

OVL Assertion Checker Implementation

Assertion checkers address design verification concerns and can be used as follows to increase design confidence:

- Combine assertion checkers to increase the coverage of the design (for example, in interface circuits and corner cases).
- Include assertion checkers when a module has an external interface. In this case, assumptions on the correct input and output behavior should be guarded and verified.
- Include assertion checkers when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores), or may not completely understand the module. In these cases, guarding the module with assertion checkers may prevent incorrect use of the module.

Usually there is a specific assertion checker suited to cover a potential problem. In other cases, even though a specific assertion checker might not exist, a combination of two or three assertion checkers can provide the desired coverage. The number of actual assertions that must be added to a specific design may vary from a few to thousands, depending on the complexity of the design and the complexity of the properties that must be checked.

Writing assertion checkers for a given design requires careful analysis and planning for maximum efficiency. While writing too few assertions might not increase the coverage on a design, writing too many assertions may increase verification time, sometimes without increasing the coverage. In most cases, however, the runtime penalty incurred by adding assertion checkers is relatively small.

OVL Assertion Checker Characteristics

Checker Class

OVL assertion checkers are partitioned into the following checker classes:

- Combinational assertions — behavior checked with combinational logic.
- Single-cycle assertions — behavior checked in the current cycle.
- 2-cycle assertions — behavior checked for transitions from the current cycle to the next.
- *n*-cycle assertions — behavior checked for transitions over a fixed number of cycles.
- Event-bounded assertions — behavior is checked between two events.

Clock and Reset

All edge-triggered assertion checkers have a clock port named *clk*. All sampling and assertion checking of these checkers is performed on the rising-edge of *clk*. All checkers have an active-low reset port named *reset_n*. Reset on all edge-triggered assertion checkers is active-low, and is synchronous to *clk*. The reset assignments of all assertion checkers can be overridden and controlled by the following global variable:

`'OVL_GLOBAL_RESET=reset_signal`	Overrides the *reset_n* port assignments of all assertion checkers with the specified global reset signal. Default: each checker's reset is specified by the *reset_n* port.

Checker Parameters

Each OVL assertion checker has its own set of parameters as described in its corresponding data sheet. The following parameters are common to all checkers.

severity_level

The severity level determines how to handle an assertion violation. Possible values are:

`'OVL_FATAL`	Runtime fatal error.
`'OVL_ERROR`	(default) Runtime error.
`'OVL_WARNING`	Runtime warning.
`'OVL_INFO`	No improper design functionality.

If *severity_level* is not one of these values, the checker issues the following message:

```
Illegal option used in parameter 'severity_level'
```

property_type

The property type determines whether to use the assertion as an assert property or an assume property (for example, a property that a formal tool uses to determine legal stimulii). Possible values are:

`'OVL_ASSERT`	(default) Assert property.
`'OVL_ASSUME`	Assume property.
`'OVL_IGNORE`	Ignore property.

If *property_type* is not one of these values, an assertion violation occurs and the checker issues the following message:

```
Illegal option used in parameter 'property_type'
```

msg

The default message issued when an assertion fails is “VIOLATION”. The *msg* parameter changes the message for the checker.

coverage_level

The coverage level determines the cover point information reported by the individual assertion. This parameter can be any logical bitwise-OR of the defined cover point type values (“Cover Points” on page 14 and “Monitoring Coverage” on page 16):

`'OVL_COVER_SANITY`	Report SANITY cover points.
`'OVL_COVER_BASIC`	Report BASIC cover points.
`'OVL_COVER_CORNER`	Report CORNER cover points.
`'OVL_COVER_STATISTIC`	*Reserved for future use.*

For example, if the *coverage_level* parameter for an instance of the assert_range checker is:

```
'OVL_COVER_BASIC | 'OVL_COVER_CORNER
```

then the checker reports all three assert_range cover points (`cover_cover_test_expr_change`, `cover_test_expr_at_min` and `cover_test_expr_at_max`).

To simplify instance specifications, two additional cover point values are defined:

`'OVL_COVER_NONE`	Disable coverage reporting.
`'OVL_COVER_ALL`	(default) Report information for all cover points.

Assertion Checks

Each assertion checker verifies that its parameter values are legal. If an illegal option is specified, the assertion fails. The assertion checker also checks at least one assertion. Violation of any of these assertions is an assertion failure. The data sheet for the assertion shows the various failure types for the assertion checker (except for incorrect option values for severity_level, property_type and coverage_level).

For example, the assert_frame checker data sheet shows the following types of assertion failures:

`ASSERT_FRAME`	The value of *test_expr* was TRUE before *min_cks* cycles after *start_event* was sampled TRUE or its value was not TRUE before max_cks cycles transpired after the rising edge of *start_event*.
`illegal start event`	The *action_on_new_start* parameter is set to ‘OVL_ERROR_ON_NEW_START and *start_event* expression evaluated to TRUE while the checker was monitoring *test_expr*.
`min_cks > max_cks`	The *min_cks* parameter is greater than the *max_cks* parameter (and *max_cks* >0). Unless the violation is fatal, either the minimum or maximum check will fail.

X/Z Checks

Assertion checkers can produce indeterminate results if a checker port value contains an X or Z bit when the checker samples the port. (Note that a checker does not necessarily sample every port at every active clock edge.) To assure determinate results, assertion checkers have special assertions for X/Z checks. These assertions fall into two groups: explicit X/Z checks and implicit X/Z checks.

Explicit X/Z Checks

Two assertion checker types are specifically designed to verify that their associated expressions have known and driven values: assert_never_unknown and assert_never_unknown_async. Each has a single assertion check:

`test_expr contains X/Z value`	Expression evaluated to a value with an X or Z bit, and ‘OVL_XCHECK_OFF is not set.

Explicit X/Z checking is implemented when instances of these checkers are added explicitly to verify relevant expressions. Setting ‘OVL_XCHECK_OFF turns off all X/Z checks, both explicit and implicit (in particular, all assert_never_unknown and assert_never_unknown_async checkers are excluded).

Implicit X/Z Checks

All assertion checker types — except assert_never_unknown and assert_never_unknown_async — have implicit X/Z checks. These are assertions that ensure specific checker ports have known and driven values when the checker samples the ports. For example, the assert_frame checker type as the following implicit X/Z checks:

`test_expr contains X or Z`	Expression value was X or Z.
`start_event contains X or Z`	Start event value was X or Z.

Implicit checking is implemented inside the checker logic itself. Setting ‘OVL_IMPLICIT_XCHECK_OFF turns off the implicit X/Z checks, but not the explicit X/Z checks.

Cover Points

Each assertion type (typically) has a set of cover points and each cover point is categorized by its cover point type. For example, the assert_range assertion type has the following cover points:

`cover_cover_test_expr_change`	BASIC — Expression changed value.
`cover_test_expr_at_min`	CORNER — Expression evaluated to *min*.
`cover_test_expr_at_max`	CORNER — Expression evaluated to *max*.

The various cover point types are:

`SANITY`	Event that indicates that the logic monitored by the assertion checker was activated at least at a minimal level.
`BASIC`	Event that indicates that the logic monitored by the assertion checker assumed a state requisite for relevant assertion checking to occur.
`CORNER`	Event that indicates that the logic monitored by the assertion checker assumed a state that represents a corner-case behavior.
`STATISTIC`	*Reserved for future use.*

OVL Use Model

An Accellera Standard OVL library user specifies preferred control settings with standard global variables defined in the following:

- A Verilog file loaded in before the libraries.
- Specifies settings using the standard `+define` options in Verilog verification engines (via a setup file or at the command line).

Setting the Implementation Language

The Accellera Standard OVL is implemented in the following HDL languages: Verilog 95, SVA 3.1a and PSL 1.1. The following global variables select the implementation language:

`‘OVL_VERILOG`	(default) Creates assertion checkers defined in Verilog.
`‘OVL_SVA`	Creates assertion checkers defined in System Verilog.
`‘OVL_PSL`	Creates assertion checkers defined in PSL (Verilog flavor).

In the case a user of the library does not specify a language, by default the library is automatically set to ‘OVL_VERILOG.

Note

Only one library can be selected. If the user specifies both ‘OVL_VERILOG and ‘OVL_SVA (or ‘OVL_PSL), the ‘OVL_VERILOG is undefined in the header file. Editing the header file to disable this behavior will result in compile errors.

Instantiation in an SVA Interface Construct

If an OVL checker is instantiated in a System Verilog interface construct, the user should define the following global variable:

`‘OVL_SVA_INTERFACE`	Ensures OVL assertion checkers can be instantiated in a System Verilog interface construct. Default: not defined.

Limitations for PSL

The PSL implementation does not support modifying the *severity_level* and *msg* parameters. These parameters are ignored and the default values are used:

`severity_level`	‘OVL_ERROR
`msg`	“VIOLATION”

Generating Synthesizable Logic

The following global variable ensures all generated OVL logic is synthesizable:

`'OVL_SYNTHESIS_OFF`	Ensures OVL logic is synthesizable. Default: not defined.

Enabling Assertion and Coverage Logic

The Accellera Standard OVL consists of two types of logic: assertion logic and coverage logic. These capabilities are controlled via the following standard global variables:

`'OVL_ASSERT_ON`	Activates assertion logic. Default: not defined.
`'OVL_COVER_ON`	Activates coverage logic. Default: not defined.

If neither of these variables is defined, the assertion checkers are not activated. The instantiations of these checkers will have no influence on the verification performed.

Asserting, Assuming and Ignoring Properties

The OVL checkers' assertion logic—if activated (by the 'OVL_ASSERT_ON global variable)—identifies a design's legal properties. Each particular checker instance can verify one or more assertion checks (depending on the checker type and the checker's configuration).

Whether all of a checker's properties are asserts (i.e., checks) or assumes (i.e., constraints) is controlled by the checker's *property_type* parameter:

`'OVL_ASSERT`	(default) All the assertion checker's checks are asserts.
`'OVL_ASSUME`	All the assertion checker's checks are assumes.
`'OVL_IGNORE`	All the assertion checker's checks are ignored.

A single assertion checker cannot have some checks asserts and other checks assumes. However, you often can implement this behavior by specifying two checkers.

Monitoring Coverage

The 'OVL_COVER_ON define activates coverage logic in the checkers. This is a global switch that turns coverage monitoring on.

Reporting Assertion Information

By default, (if the assertion logic is active) every assertion violation is reported and (if the coverage logic is active) every captured coverage point is reported. The user can limit this reporting and can also initiate special reporting at the start and end of simulation.

Limiting a Checker's Reporting

Limits on the number of times assertion violations and captured coverage points are reported are controlled by the following global variables:

`'OVL_MAX_REPORT_ERROR`	Discontinues reporting a checker's assertion violations if the number of times the checker has reported one or more violations reaches this limit. Default: unlimited reporting.
`'OVL_MAX_REPORT_COVER_POINT`	Discontinues reporting a checker's cover points if the number of times the checker has reported one or more cover points reaches this limit.Default: unlimited reporting.

These maximum limits are for the number of times a checker instance issues a message. If a checker issues multiple violation messages in a cycle, each message is counted as a single error report. Similarly, if a checker issues multiple coverage messages in a cycle, each message is counted as a single cover report.

Reporting Initialization Messages

The checkers' configuration information is reported at initialization time if the following global variable is defined:

`'OVL_INIT_MSG`	Reports configuration information for each checker when it is instantiated at the start of simulation. Default: no initialization messages reported.

For each assertion checker instance, the following message is reported:

```
OVL_NOTE: V1.7: instance_name initialized @ hierarchy Severity:
severity_level, Message: msg
```

End-of-simulation Signal to assert_quiescent_state Checkers

The assert_quiescent_state assertion checker checks that the value of a state expression equals a check value when a sample event occurs. These checkers also can perform this check at the end of simulation by setting the following global variable:

`'OVL_END_OF_SIMULATION=`*`eos_signal`*	Performs quiescent state checking at end of simulation when the *eos_signal* asserts. Default: not defined.

Fatal Error Processing

When a checker reports a runtime fatal error (*severity_level* is ‘OVL_FATAL), simulation continues for a certain amount of time and then the simulation ends. This time limit is controlled by the following global variable:

`‘OVL_RUNTIME_AFTER_FATAL`	Number of time units from a fatal error to end of simulation. Default: 100.

Checking of X and Z Values

By default, OVL assertion checker logic includes logic implementing assertion checks for X and Z bits in the values of checker ports when they are sampled. To exclude part or all of this X/Z checking logic, specify one of the following global variables:

`‘OVL_IMPLICIT_XCHECK_OFF`	Turns off implicit X/Z checks.
`‘OVL_XCHECK_OFF`	Turns off all X/Z checks (implicit and explicit).

OVL Verilog/SVA Library

Library Characteristics

The OVL library has the following characteristics:

- All Verilog assertion checkers conform to Verilog IEEE Standard 1364-1995.
- All System Verilog assertion checkers conform to Accellera SVA 3.1a.
- Header files use file extension .h.
- Verilog files with assertion module/interfaces use extension .vlib and include assertion logic files in the language specified by the user.
- Verilog files with assertion logic use file extension `_logic.v`.
- System Verilog files with assertion logic use file extension `_logic.sv`.
- The name of an OVL assertion checker is `assert_`*name*, where the *name* is a descriptive identifier.
- Parameter settings are passed via literals to make configuration of assertion checkers consistent and simple to use by end users.
- Parameters passed to assertion checkers are checked for legal values
- Each assertion checker includes std_ovl_defines.h defining all global variables and `std_ovl_task.h` defining all OVL system tasks.
- Global variables are named OVL_*name*.
- System tasks are named `ovl_`*taskname*`_t`.
- Assertion checkers are initialized explicitly so that they work in a deterministic way without reset.
- Assertion checkers are backward compatible in behavior with existing OVL Verilog libraries (to the extent it is possible).

Library Layout

The Accellera OVL standard library has the following structure:

`$STD_OVL_DIR`	Installation directory of Accellera OVL library.
`$STD_OVL_DIR/vlog95`	Directory with assertion logic described in Verilog 95.
`$STD_OVL_DIR/sva31a`	Directory with assertion logic described in SVA 3.1a.
`$STD_OVL_DIR/psl11`	Directory with assertion logic described in PSL 1.1.
`$STD_OVL_DIR/psl11/vunits`	Directory with PSL1.1 vunits for binding with the assertion logic.

For example:

```
shell prompt> ls -l $STD_OVL_DIR
std_ovl/assert_always.vlib
std_ovl/assert_always_on_edge.vlib
.  .  .
std_ovl/std_ovl_defines.h
std_ovl/std_ovl_task.h
.  .  .
std_ovl/psl11:
std_ovl/psl11/assert_always_logic.vlib
std_ovl/psl11/assert_always_on_edge_logic.vlib
.  .  .
std_ovl/psl11/vunits:
std_ovl/psl11/vunits/assert_always.psl
std_ovl/psl11/vunits/assert_always_on_edge.psl
.  .  .
std_ovl/sva31a:
std_ovl/sva31a/assert_always_logic.vlib
std_ovl/sva31a/assert_always_on_edge_logic.vlib
.  .  .
std_ovl/vlog95:
std_ovl/vlog95/assert_always_logic.v
std_ovl/vlog95/assert_always_on_edge_logic.v
.  .  .
```

Examples

Header File

Figure 2-1. $STD_OVL_DIR/std_ovl_defines.h

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

`ifdef OVL_STD_DEFINES_H
// do nothing
`else
`define OVL_STD_DEFINES_H

`define OVL_VERSION “V1.7”
`ifdef OVL_ASSERT_ON
  `ifdef OVL_PSL
     `ifdef OVL_VERILOG
        `undef OVL_PSL
     `endif
     `ifdef OVL_SVA
        `ifdef OVL_PSL
          `undef OVL_PSL
        `endif
     `endif
  `else
    `ifdef OVL_VERILOG
    `else
      `define OVL_VERILOG
    `endif
    `ifdef OVL_SVA
       `undef OVL_VERILOG
    `endif
  `endif
`endif
`ifdef OVL_COVER_ON
`ifdef OVL_PSL
     `ifdef OVL_VERILOG
        `undef OVL_PSL
     `endif
     `ifdef OVL_SVA
        `ifdef OVL_PSL
          `undef OVL_PSL
        `endif
     `endif
  `else
    `ifdef OVL_VERILOG
    `else
      `define OVL_VERILOG
    `endif
    `ifdef OVL_SVA
       `undef OVL_VERILOG
    `endif
  `endif
`endif
```

```
`ifdef OVL_ASSERT_ON
  `ifdef OVL_SHARED_CODE
  `else
    `define OVL_SHARED_CODE
  `endif
`else
  `ifdef OVL_COVER_ON
    `ifdef OVL_SHARED_CODE
    `else
      `define OVL_SHARED_CODE
    `endif
  `endif
`endif

// specifying interface for System Verilog

`ifdef OVL_SVA_INTERFACE
  `define module interface
  `define endmodule endinterface
`else
  `define module module
  `define endmodule endmodule
`endif

// Selecting global reset or local reset for the checker reset signal

`ifdef OVL_GLOBAL_RESET
  `define OVL_RESET_SIGNAL `OVL_GLOBAL_RESET
`else
  `define OVL_RESET_SIGNAL reset_n
`endif

// active edges

`define OVL_NOEDGE  0
`define OVL_POSEDGE 1
`define OVL_NEGEDGE 2
`define OVL_ANYEDGE 3

// severity levels

`define OVL_FATAL   0
`define OVL_ERROR   1
`define OVL_WARNING 2
`define OVL_INFO    3

// coverage levels

`define OVL_COVER_NONE      0
`define OVL_COVER_SANITY    1
`define OVL_COVER_BASIC     2
`define OVL_COVER_CORNER    4
`define OVL_COVER_STATISTIC 8
`define OVL_COVER_ALL       15

// default coverage level
`define OVL_COVER_DEFAULT `OVL_COVER_BASIC
```

```
// property type

`define OVL_ASSERT 0
`define OVL_ASSUME 1
`define OVL_IGNORE 2

// necessary condition

`define OVL_TRIGGER_ON_MOST_PIPE    0
`define OVL_TRIGGER_ON_FIRST_PIPE   1
`define OVL_TRIGGER_ON_FIRST_NOPIPE 2

// action on new start

`define OVL_IGNORE_NEW_START   0
`define OVL_RESET_ON_NEW_START 1
`define OVL_ERROR_ON_NEW_START 2

// inactive levels

`define OVL_ALL_ZEROS 0
`define OVL_ALL_ONES  1
`define OVL_ONE_COLD  2

// ovl runtime after fatal error

‘define OVL_RUNTIME_AFTER_FATAL 100

`endif // OVL_STD_DEFINES_H
```

Assertion Checker Interface Files

Figure 2-2. $STD_OVL_DIR/assert_implication.vlib

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

`include “std_ovl_defines.h”
`module assert_implication (clk, reset_n, antecedent_expr,
            consequent_expr);
  input clk, reset_n, antecedent_expr, consequent_expr;
  parameter severity_level = `OVL_ERROR;
  parameter property_type = `OVL_ASSERT;
  parameter msg = “VIOLATION”;
  parameter coverage_level = `OVL_COVER_DEFAULT;
`ifdef OVL_COVER_ON
  parameter OVL_COVER_SANITY_ON = (coverage_level & `OVL_COVER_SANITY);
  parameter OVL_COVER_BASIC_ON = (coverage_level & `OVL_COVER_BASIC);
  parameter OVL_COVER_CORNER_ON = (coverage_level & `OVL_COVER_CORNER);
  parameter OVL_COVER_STATISTIC_ON = (coverage_level &
              `OVL_COVER_STATISTIC);
`endif
`ifdef OVL_VERILOG
  `include “./vlog95/assert_implication_logic.v”
`endif
```

```
`ifdef OVL_SVA
  `include "./sva31a/assert_implication_logic.sv"
`endif
`ifdef OVL_PSL
  `include "./psl11/assert_implication_psl_logic.v"
`else
  `endmodule
`endif
```

Assertion Checker Logic Files (Verilog 95)

Figure 2-3. $STD_OVL_DIR/vlog95/assert_implication_logic.v

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

  parameter assert_name = "ASSERT_IMPLICATION";

  `include "std_ovl_task.h"

`ifdef OVL_XCHECK_OFF
   //Do nothing
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    //Do nothing
  `else
  wire valid_antecedent_expr;
  wire valid_consequent_expr;

  assign valid_antecedent_expr = ~(antecedent_expr ^ antecedent_expr);
  assign valid_consequent_expr = ~(consequent_expr ^ consequent_expr);
 `endif // OVL_IMPLICIT_XCHECK_OFF
`endif // OVL_XCHECK_OFF

`ifdef OVL_INIT_MSG
    initial
      ovl_init_msg_t; // Call the User Defined Init Message Routine
`endif //OVL_INIT_MSG

`ifdef OVL_ASSERT_ON
  always @(posedge clk) begin
    if (`OVL_RESET_SIGNAL != 1'b0) begin
      if (antecedent_expr  == 1'b1 && consequent_expr == 1'b0) begin
        ovl_error_t("Antecedent does not have consequent");
      end
    end
  end
`endif // OVL_ASSERT_ON

`ifdef OVL_XCHECK_OFF
   //Do nothing
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    //Do nothing
  `else
 `ifdef OVL_ASSERT_ON
```

```
  always @(posedge clk)
    begin
      if (`OVL_RESET_SIGNAL != 1'b0)
        begin
          if (valid_antecedent_expr == 1'b1)
            begin
              // Do nothing
            end
          else
            ovl_error_t("antecedent_expr contains X or Z");

          if (antecedent_expr == 1'b1)
            begin
              if (valid_consequent_expr == 1'b1)
                begin
                  // Do nothing
                end
              else
                ovl_error_t("consequent_expr contains X or Z");
            end
       end
    end
 `endif // OVL_ASSERT_ON
 `endif // OVL_IMPLICIT_XCHECK_OFF
`endif // OVL_XCHECK_OFF

`ifdef OVL_COVER_ON
  always @ (posedge clk) begin
    if (coverage_level != `OVL_COVER_NONE) begin
     if ((OVL_COVER_BASIC_ON) && (`OVL_RESET_SIGNAL != 1'b0)) begin
      //basic coverage
      if (antecedent_expr == 1'b1) begin
       ovl_cover_t("antecedent covered");
      end
     end //basic coverage
    end //OVL_COVER_NONE
  end //always
`endif //OVL_COVER_ON
```

Assertion Checker Logic Files (System Verilog 3.1a)

Figure 2-4. $STD_OVL_DIR/sva31a/assert_implication_logic.sv

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

  parameter assert_name = "ASSERT_IMPLICATION";
  `include "std_ovl_task.h"

`ifdef OVL_INIT_MSG
    initial
      ovl_init_msg_t; // Call the User Defined Init Message Routine
`endif //OVL_INIT_MSG
```

```
  property ASSERT_IMPLICATION_P;
  @(posedge clk)
  disable iff (`OVL_RESET_SIGNAL != 1'b1)
  antecedent_expr |-> consequent_expr;
  endproperty

`ifdef OVL_XCHECK_OFF
  //Do nothing
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    //Do nothing
  `else
  property ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P;
  @ (posedge clk)
  disable iff (`OVL_RESET_SIGNAL != 1'b1)
  (!($isunknown(antecedent_expr)));
  endproperty

  property ASSERT_IMPLICATION_XZ_ON_CON_EXP_P;
  @ (posedge clk)
  disable iff (`OVL_RESET_SIGNAL != 1'b1)
  antecedent_expr |-> (!($isunknown(consequent_expr)));
  endproperty
  `endif //OVL_IMPLICIT_XCHECK_OFF
`endif //OVL_XCHECK_OFF

`ifdef OVL_ASSERT_ON

  generate

    case (property_type)
      `OVL_ASSERT : begin : ovl_assert
        A_ASSERT_IMPLICATION_P: assert property (ASSERT_IMPLICATION_P)
            else ovl_error_t("Antecedent does not have consequent");
`ifdef OVL_XCHECK_OFF
  //Do nothing
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    //Do nothing
  `else
        A_ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P:
        assert property (ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P)
        else ovl_error_t("antecedent_expr contains X or Z");
        A_ASSERT_IMPLICATION_XZ_ON_CON_EXP_P:
        assert property (ASSERT_IMPLICATION_XZ_ON_CON_EXP_P)
        else ovl_error_t("consequent_expr contains X or Z");
  `endif //OVL_IMPLICIT_XCHECK_OFF
`endif //OVL_XCHECK_OFF
      end
      `OVL_ASSUME : begin : ovl_assume
        M_ASSERT_IMPLICATION_P: assume property (ASSERT_IMPLICATION_P);
```

```
`ifdef OVL_XCHECK_OFF
  //Do nothing
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    //Do nothing
  `else
        M_ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P:
        assume property (ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P);

        M_ASSERT_IMPLICATION_XZ_ON_CON_EXP_P:
        assume property (ASSERT_IMPLICATION_XZ_ON_CON_EXP_P);
  `endif //OVL_IMPLICIT_XCHECK_OFF
`endif //OVL_XCHECK_OFF

      end
      `OVL_IGNORE : begin : ovl_ignore
        // do nothing
      end

      default     : initial ovl_error_t(“”);
    endcase

  endgenerate

`endif // OVL_ASSERT_ON

`ifdef OVL_COVER_ON
  generate
    if (coverage_level != `OVL_COVER_NONE) begin
     if (OVL_COVER_BASIC_ON) begin //basic coverage
      cover_antecedent:
      cover property (@(posedge clk) ( (`OVL_RESET_SIGNAL != 1’b0) &&
                     antecedent_expr) )
                     ovl_cover_t(“antecedent covered”);
     end
    end
  endgenerate
`endif // OVL_COVER_ON
```

Assertion Checker Logic Files (PSL 1.1)

Figure 2-5. $STD_OVL_DIR/psl11/assert_implication_psl_logic.v

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

//This file is included in assert_implication.vlib

`include “std_ovl_task.h”
  parameter assert_name = “ASSERT_IMPLICATION”;

 `ifdef OVL_INIT_MSG
  initial
    ovl_init_msg_t; // Call the User Defined Init Message Routine
`endif

`ifdef OVL_ASSERT_ON
 wire xzcheck_enable;

`ifdef OVL_XCHECK_OFF
  assign xzcheck_enable = 1’b0;
`else
  `ifdef OVL_IMPLICIT_XCHECK_OFF
    assign xzcheck_enable = 1’b0;
  `else
    assign xzcheck_enable = 1’b1;
  `endif //OVL_IMPLICIT_XCHECK_OFF
`endif //OVL_XCHECK_OFF

 generate
   case (property_type)
     `OVL_ASSERT: begin: assert_checks
                   assert_implication_assert
                   assert_implication_assert (
                       .clk(clk),
                       .reset_n(`OVL_RESET_SIGNAL),
                       .antecedent_expr(antecedent_expr),
                       .consequent_expr(consequent_expr),
                       .xzcheck_enable(xzcheck_enable));
                  end
     `OVL_ASSUME: begin: assume_checks
                   assert_implication_assume
                   assert_implication_assume (
                       .clk(clk),
                       .reset_n(`OVL_RESET_SIGNAL),
                       .antecedent_expr(antecedent_expr),
                       .consequent_expr(consequent_expr),
                       .xzcheck_enable(xzcheck_enable));
                  end
     `OVL_IGNORE: begin: ovl_ignore
                    //do nothing
                  end
     default: initial ovl_error_t(“”);
   endcase
 endgenerate
`endif
```

```
`ifdef OVL_COVER_ON
 generate
  if (coverage_level != `OVL_COVER_NONE)
   begin: cover_checks
          assert_implication_cover #(
                       .OVL_COVER_BASIC_ON(OVL_COVER_BASIC_ON))
          assert_implication_cover (
                       .clk(clk),
                       .reset_n(`OVL_RESET_SIGNAL),
                       .antecedent_expr(antecedent_expr));
   end
 endgenerate
`endif

`endmodule //Required to pair up with already used “`module” in file
assert_implication.vlib

//Module to be replicated for assert checks
//This module is bound to a PSL vunits with assert checks
module assert_implication_assert (clk, reset_n, antecedent_expr,
consequent_expr, xzcheck_enable);
       input clk, reset_n, antecedent_expr, consequent_expr,
xzcheck_enable;
endmodule

//Module to be replicated for assume checks
//This module is bound to a PSL vunits with assume checks
module assert_implication_assume (clk, reset_n, antecedent_expr,
consequent_expr, xzcheck_enable);
       input clk, reset_n, antecedent_expr, consequent_expr,
xzcheck_enable;
endmodule

//Module to be replicated for cover properties
//This module is bound to a PSL vunit with cover properties
module assert_implication_cover (clk, reset_n, antecedent_expr);
       parameter OVL_COVER_BASIC_ON = 1;
       input clk, reset_n, antecedent_expr;
endmodule
```

Assertion Checker vunit Files (PSL 1.1)

Figure 2-6. $STD_OVL_DIR/psl11/vunits/assert_implication.psl

```
// Accellera Standard V1.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2006. All rights reserved.

vunit assert_implication_assert_vunit (assert_implication_assert)
{
  default clock = (posedge clk);
  property ASSERT_IMPLICATION_P = always (
           reset_n && antecedent_expr -> consequent_expr);

  //Properties for X/Z checking
  property ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P =
           always( xzcheck_enable -> !isunknown(antecedent_expr)
               abort(!reset_n) );
  property ASSERT_IMPLICATION_XZ_ON_CON_EXP_P =
           always( xzcheck_enable && antecedent_expr ->
               !isunknown(consequent_expr)
                   abort(!reset_n) );
  A_ASSERT_IMPLICATION_P:
  assert ASSERT_IMPLICATION_P
  report “VIOLATION: ASSERT_IMPLICATION Checker Fires : Antecedent does
               not have consequent”;
  A_ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P:
  assert ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P
  report “VIOLATION: ASSERT_IMPLICATION Checker Fires: antecedent_expr
               contains X or Z”;
  A_ASSERT_IMPLICATION_XZ_ON_CON_EXP_P:
  assert ASSERT_IMPLICATION_XZ_ON_CON_EXP_P
  report “VIOLATION: ASSERT_IMPLICATION Checker Fires: consequent_expr
               contains X or Z”;
}

vunit assert_implication_assume_vunit (assert_implication_assume)
{
  default clock = (posedge clk);
  property ASSERT_IMPLICATION_P = always (
           reset_n && antecedent_expr -> consequent_expr);

  //Properties for X/Z checking
  property ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P =
           always( xzcheck_enable -> !isunknown(antecedent_expr)
               abort(!reset_n) );
  property ASSERT_IMPLICATION_XZ_ON_CON_EXP_P =
           always( xzcheck_enable && antecedent_expr ->
               !isunknown(consequent_expr)
                   abort(!reset_n) );
  M_ASSERT_IMPLICATION_P:
  assume ASSERT_IMPLICATION_P;
  M_ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P:
  assume ASSERT_IMPLICATION_XZ_ON_ANT_EXP_P;
  M_ASSERT_IMPLICATION_XZ_ON_CON_EXP_P:
  assume ASSERT_IMPLICATION_XZ_ON_CON_EXP_P;
}
```

```
vunit assert_implication_cover_vunit (assert_implication_cover)
{
  default clock = (posedge clk);
  cover_antecedent:
  cover {OVL_COVER_CORNER_ON && reset_n && antecedent_expr}
  report “COVERAGE REPORT: ASSERT_IMPLICATION Checker: antecedent
covered”;
}
```

Chapter 3
OVL Checker Data Sheets

Each OVL assertion checker type has a data sheet that provides the specification for checkers of that type. This chapter lists the checker data sheets in alphabetical order by checker type. Data sheets contain the following information:

- **Syntax**

  Syntax statement for specifying a checker of the type, with:

  - Parameters — parameters that configure the checker.
  - Ports — checker ports.

- **Description**

  Description of the functionality and usage of checkers of the type, with:

  - Assertion Checks — violation types (or messages) with descriptions of failures.
  - Cover Points — cover messages with descriptions.
  - Errors* — possible errors that are not assertion failures.

- **Notes***

  Notes describing any special features or requirements.

- **See also**

  List of other similar checker types.

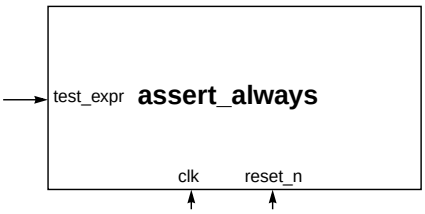
- **Examples**

  Examples of directives and checker applications.

* not applicable to all checker types.

assert_always

Ensures that the value of a specified expression is TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_always
      [#(severity_level, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should evaluate to TRUE on the rising clock edge.

Description

The assert_always assertion checker checks the single-bit expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to TRUE.

Assertion Checks

ASSERT_ALWAYS	Expression did not evaluate to TRUE.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.

Cover Points

none

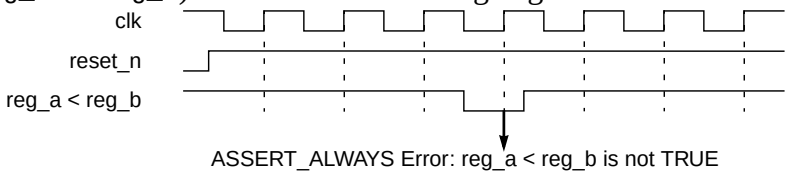
See also

assert_always_on_edge
assert_implication
assert_never
assert_proposition

Example

```
assert_always #(

   ‘OVL_ERROR,                                    // severity_level
   ‘OVL_ASSERT,                                   // property_type
   “Error: reg_a < reg_b is not TRUE”,            // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   reg_a_lt_reg_b (

      clk,                                        // clock
      reset_n,                                    // reset
      reg_a < reg_b );                            // test_expr
```

Ensures that (`reg_a < reg_b`) is TRUE at each rising edge of `clk`.



assert_always_on_edge

Ensures that the value of a specified expression is TRUE when a sampling event undergoes a specified transition.

Parameters:
severity_level
edge_type
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_always_on_edge
      [#(severity_level, edge_type, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, sampling_event, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
edge_type	Transition type for sampling event: ‘OVL_NOEDGE, ‘OVL_POSEDGE, ‘OVL_NEGEDGE or ‘OVL_ANYEDGE. Default: ‘OVL_NOEDGE.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
sampling_event	Expression that (along with *edge_type*) identifies when to evaluate and test *test_expr*.
test_expr	Expression that should evaluate to TRUE on the rising clock edge.

Description

The assert_always_on_edge assertion checker checks the single-bit expression *sampling_event* for a particular type of transition. If the specified transition of the sampling event occurs, the single-bit expression *test_expr* is evaluated at the rising edge of *clk* to verify the expression does not evaluate to FALSE.

The *edge_type* parameter determines which type of transition of *sampling_event* initiates the check:

- ‘OVL_POSEDGE performs the check if *sampling_event* transitions from FALSE to TRUE.
- ‘OVL_NEGEDGE performs the check if *sampling_event* transitions from TRUE to FALSE.
- ‘OVL_ANYEDGE performs the check if *sampling_event* transitions from TRUE to FALSE or from FALSE to TRUE.
- ‘OVL_NOEDGE always initiates the check. This is the default value of *edge_type*. In this case, *sampling_event* is never sampled and the checker has the same functionality as assert_always.

The checker is a variant of assert_always, with the added capability of qualifying the assertion with a sampling event transition. This checker is useful when events are identified by their transition in addition to their logical state.

Assertion Checks

`ASSERT_ALWAYS_ON_EDGE`	Expression evaluated to FALSE when the sampling event transitioned as specified by *edge_type*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.
`sampling_event contains X or Z`	Sampling event value was X or Z.

Cover Points

none

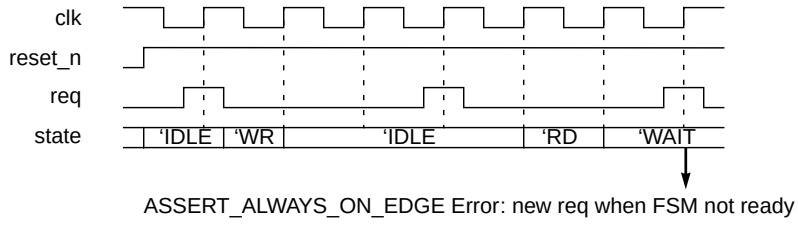
See also

assert_always
assert_implication
assert_never
assert_proposition

Examples

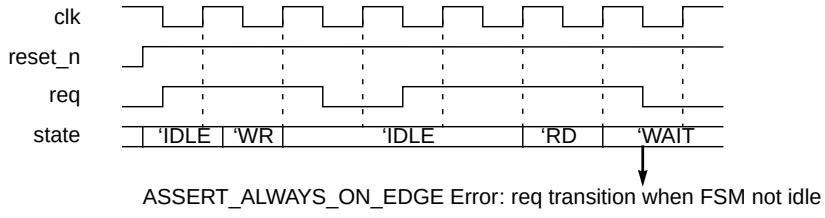
```
assert_always_on_edge #(

   ‘OVL_ERROR,                                    // severity_level
   ‘OVL_POSEDGE,                                  // edge_type
   ‘OVL_ASSERT,                                   // property_type
   “Error: new req when FSM not ready”,           // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   request_when_FSM_idle (

      clk,                                        // clock
      reset_n,                                    // reset
      req,                                        // sampling_event
      state == ‘IDLE);                            // test_expr
```

Ensures that `(state == ‘IDLE)` is TRUE at each rising edge of `clk` when `req` transitions from FALSE to TRUE.



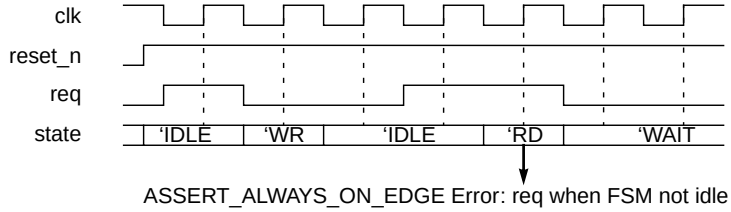

```
assert_always_on_edge #(

   ‘OVL_ERROR,                                    // severity_level
   ‘OVL_ANYEDGE,                                  // edge_type
   ‘OVL_ASSERT,                                   // property_type
   “Error: req transition when FSM not idle”,     // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   req_transition_when_FSM_idle (

      clk,                                        // clock
      reset_n,                                    // reset
      req,                                        // sampling_event
      state == ‘IDLE);                            // test_expr
```

Ensures that `(state == ‘IDLE)` is TRUE at each rising edge of `clk` when `req` transitions from TRUE to FALSE or from FALSE to TRUE.



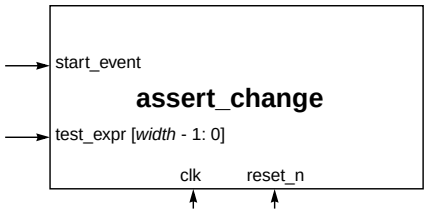
```
assert_always_on_edge #(

   ‘OVL_ERROR,                                  // severity_level
   ‘OVL_NOEDGE,                                 // edge_type
   ‘OVL_ASSERT,                                 // property_type
   “Error: req when FSM not idle”,              // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   req_when_FSM_idle (

      clk,                                      // clock
      reset_n,                                  // reset
      1’b0,                                     // sampling_event
      !req || (state == ‘IDLE) );               // test_expr
```

Ensures that `(!req || (state == ‘IDLE))` is TRUE at each rising edge of `clk`.



assert_change

Ensures that the value of a specified expression changes within a specified number of cycles after a start event initiates checking.



Parameters:
severity_level
width
num_cks
action_on_new_start
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_change
      [#(severity_level, width, num_cks, action_on_new_start,
      property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
num_cks	Number of cycles to check for a change in the value of *test_expr*. Default: 1.
action_on_new_start	Method for handling a new start event that occurs before *test_expr* changes value or *num_cks* clock cycles transpire without a change. Values are: ‘OVL_IGNORE_NEW_START, ‘OVL_RESET_ON_NEW_START and ‘OVL_ERROR_ON_NEW_START. Default: ‘OVL_IGNORE_NEW_START.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.

`start_event`	Expression that (along with *action_on_new_start*) identifies when to start checking *test_expr* .
`test_expr[width-1:0]`	Expression that should change value within *num_cks* cycles from the start event unless the check is interrupted by a valid new start event.

Description

The assert_change assertion checker checks the expression *start_event* at each rising edge of *clk* to determine if it should check for a change in the value of *test_expr*. If *start_event* is sampled TRUE, the checker evaluates *test_expr* and re-evaluates *test_expr* at each of the subsequent *num_cks* rising edges of *clk*. If the value of *test_expr* has not changed from its start value by the last of the *num_cks* cycles, the assertion fails.

The method used to determine how to handle a new start event, when the checker is in the state of checking for a change in *test_expr*, is controlled by the *action_on_new_start* parameter. The checker has the following actions:

- ‘OVL_IGNORE_NEW_START

  The checker does not sample *start_event* for the next *num_cks* cycles after a start event.

- ‘OVL_RESET_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the checker terminates the check and initiates a new check with the current value of *test_expr* (even on the last cycle of a check).

- ‘OVL_ERROR_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the assertion fails with an illegal start event violation. In this case, the checker does not initiate a new check and does not terminate a pending check.

The checker is useful for ensuring proper changes in structures after various events, such as verifying synchronization circuits respond after initial stimuli. For example, it can be used to check the protocol that an “acknowledge” occurs within a certain number of cycles after a “request”. It also can be used to check that a finite-state machine changes state after an initial stimulus.

Assertion Checks

`ASSERT_CHANGE`	The *test_expr* expression did not change value for *num_cks* cycles after *start_event* was sampled TRUE.

`illegal start event`	The *action_on_new_start* parameter is set to ‘OVL_ERROR_ON_NEW_START and *start_event* expression evaluated to TRUE while the checker was in the state of checking for a change in the value of *test_expr*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.
`start_event contains X or Z`	Start event value was X or Z.

Cover Points

`cover_window_open`	BASIC — A change check was initiated.
`cover_window_close`	BASIC — A change check lasted the full *num_cks* cycles. If no assertion failure occurred, the value of *test_expr* changed in the last cycle.
`cover_window_resets`	CORNER — The *action_on_new_start* parameter is ‘OVL_RESET_ON_NEW_START, and *start_event* was sampled TRUE while the checker was monitoring *test_expr,* but it had not changed value.

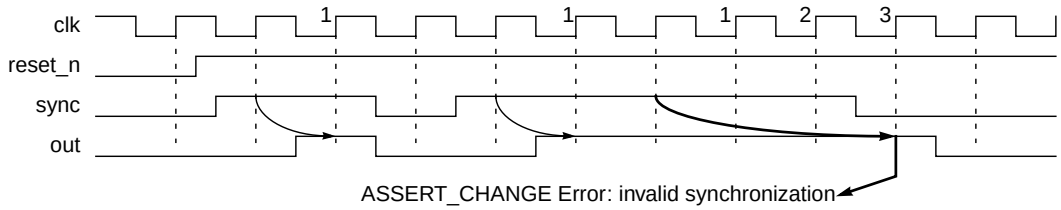
See also

assert_time
assert_unchange
assert_win_change
assert_win_unchange
assert_window

Examples

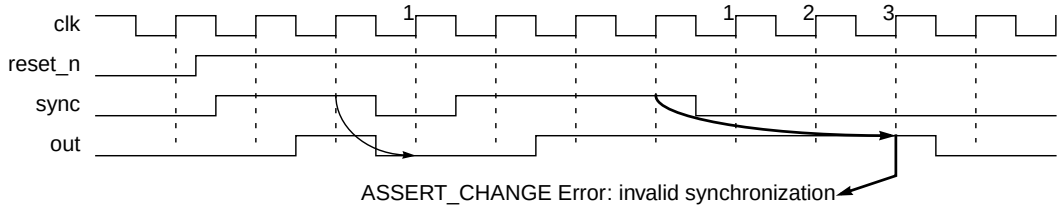
```
assert_change #(
   ‘OVL_ERROR,                                  // severity_level
   1,                                           // width
   3,                                           // num_cks
   ‘OVL_IGNORE_NEW_START,                       // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid synchronization”,            // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sync_out (

      clk,                                      // clock
      reset_n,                                  // reset
      sync == 1,                                // start_event
      out );                                    // test_expr
```

Ensures that `out` changes within 3 cycles after `sync` asserts. New starts are ignored.



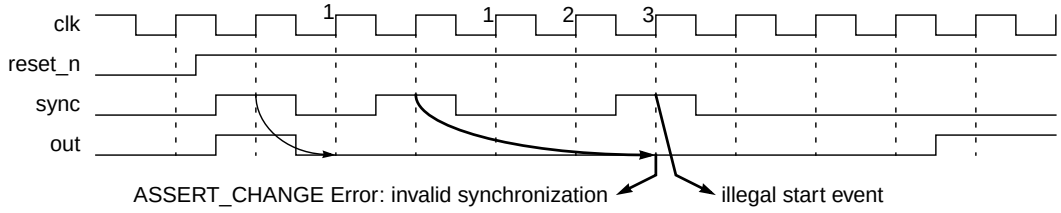

```
assert_change #(

   ‘OVL_ERROR,                                  // severity_level
   1,                                           // width
   3,                                           // num_cks
   ‘OVL_RESET_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid synchronization”,            // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sync_out (

      clk,                                      // clock
      reset_n,                                  // reset
      sync == 1,                                // start_event
      out );                                    // test_expr
```

Ensures that `out` changes within 3 cycles after `sync` asserts. A new start terminates the pending check and initiates a new check.



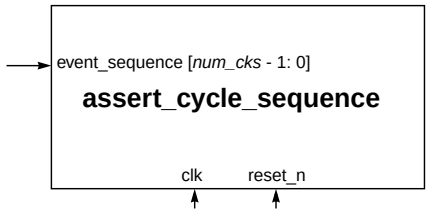

```
assert_change #(

   ‘OVL_ERROR,                                  // severity_level
   1,                                           // width
   3,                                           // num_cks
   ‘OVL_ERROR_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid synchronization”,            // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sync_out (

      clk,                                      // clock
      reset_n,                                  // reset
      sync == 1,                                // start_event
      out );                                    // test_expr
```

Ensures that `out` changes within 3 cycles after `sync` asserts. A new start reports an `illegal start event` violation (without initiating a new check) but any pending check is retained (even on the last check cycle).



assert_cycle_sequence

Ensures that if a specified necessary condition occurs, it is followed by a specified sequence of events.



Parameters:
severity_level
num_cks
necessary_condition
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_cycle_sequence
      [#(severity_level, num_cks, necessary_condition, property_type,
      msg, coverage_level )]
      instance_name (clk, reset_n, event_sequence );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
num_cks	Width of the *event_sequence* argument. This parameter must not be less than 2. Default: 2.
necessary_condition	Method for determining the necessary condition that initiates the sequence check and whether or not to pipeline checking. Values are: ‘OVL_TRIGGER_ON_MOST_PIPE, ‘OVL_TRIGGER_ON_FIRST_PIPE and ‘OVL_TRIGGER_ON_FIRST_NOPIPE. Default: ‘OVL_TRIGGER_ON_MOST_PIPE.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
event_sequence [*num_cks*-1:0]	Expression that is a concatenation where each bit represents an event.

Description

The assert_cycle_sequence assertion checker checks the expression *event_sequence* at the rising edges of *clk* to identify whether or not the bits in *event_sequence* assert sequentially on successive rising edges of *clk*. For example, the following series of 4-bit values (where *b* is any bit value) is a valid sequence:

```
1bbb –> b1bb –> bb1b –> bbb1
```

This series corresponds to the following series of events on successive rising edges of *clk*:

cycle 1	event_sequence[3] == 1
cycle 2	event_sequence[2] == 1
cycle 3	event_sequence[1] == 1
cycle 4	event_sequence[0] == 1

The checker also has the ability to pipeline its analysis. Here, one or more new sequences can be initiated and recognized while a sequence is in progress. For example, the following series of 4-bit values (where *b* is any bit value) constitutes two overlapping valid sequences:

```
1bbb –> b1bb –> 1b1b –> b1b1 –> bb1b –> bbb1
```

This series corresponds to the following sequences of events on successive rising edges of clk:

cycle 1	event_sequence[3] == 1	
cycle 2	event_sequence[2] == 1	
cycle 3	event_sequence[1] == 1	event_sequence[3] == 1
cycle 4	event_sequence[0] == 1	event_sequence[2] == 1
cycle 5		event_sequence[1] == 1
cycle 6		event_sequence[0] == 1

When the checker determines that a specified necessary condition has occurred, it subsequently verifies that a specified event or event sequence occurs and if not, the assertion fails.

The method used to determine what constitutes the necessary condition and the resulting trigger event or event sequence is controlled by the *necessary_condition* parameter. The checker has the following actions:

- ‘OVL_TRIGGER_ON_MOST_PIPE

  The necessary condition is that the bits:

  ```
  event_sequence [num_cks -1], . . . ,event_sequence [1]
  ```

  are sampled equal to 1 sequentially on successive rising edges of *clk*. When this condition occurs, the checker verifies that the value of *event_sequence*[0] is 1 at the next rising edge of *clk*. If not, the assertion fails.

  The checking is pipelined, which means that if *event_sequence*[*num_cks* -1] is sampled equal to 1 while a sequence (including *event_sequence*[0]) is in progress and subsequently the necessary condition is satisfied, the check of *event_sequence*[0] is performed (unless the first sequence resulted in a fatal assertion violation).

- ‘OVL_TRIGGER_ON_FIRST_PIPE

  The necessary condition is that the *event_sequence* [*num_cks* -1] bit is sampled equal to 1 on a rising edge of *clk*. When this condition occurs, the checker verifies that the bits:

  ```
  event_sequence [num_cks -2], . . . ,event_sequence [0]
  ```

  are sampled equal to 1 sequentially on successive rising edges of *clk*. If not, the assertion fails.

  The checking is pipelined, which means that if *event_sequence*[*num_cks* -1] is sampled equal to 1 while a check is in progress, an additional check is initiated.

- ‘OVL_TRIGGER_ON_FIRST_NOPIPE

  The necessary condition is that the *event_sequence* [*num_cks* -1] bit is sampled equal to 1 on a rising edge of *clk*. When this condition occurs, the checker verifies that the bits:

  ```
  event_sequence [num_cks -2], . . . ,event_sequence [0]
  ```

  are sampled equal to 1 sequentially on successive rising edges of *clk*. If not, the assertion fails.

  The checking is not pipelined, which means that if *event_sequence*[*num_cks* -1] is sampled equal to 1 while a check is in progress, it is ignored, even if the check is verifying the last bit of the sequence (*event_sequence* [0]).

Assertion Checks

`ASSERT_CYCLE_SEQUENCE`	The necessary condition occurred, but it was not followed by the event or event sequence.
`illegal num_cks parameter`	The *num_cks* parameter is less than 2.

Implicit X/Z Checks

`First event in the sequence contains X or Z`	Value of the first event in the sequence was X or Z.
`Subsequent events in the sequence contain X or Z`	Value of a subsequent event in the sequence was X or Z.
`First num_cks-1 events in the sequence contain X or Z`	Values of the events in the sequence (except the last event) were X or Z.
`Last event in the sequence contains X or Z`	Value of the last event in the sequence was X or Z.

Cover Points

`cover_sequence_trigger`	BASIC — The trigger sequence occurred.

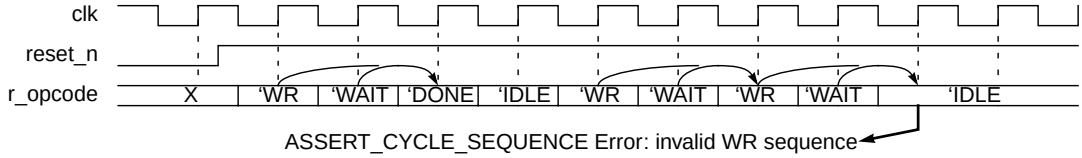
See also

assert_change assert_unchange

Examples

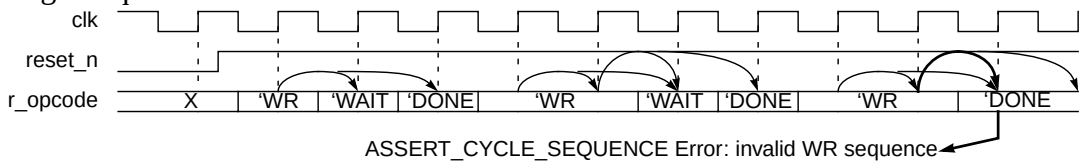
```
assert_cycle_sequence #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_TRIGGER_ON_MOST_PIPE,                   // necessary_condition
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid WR sequence”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_write_sequence (

      clk,                                      // clock
      reset_n,                                  // reset
      { r_opcode == ‘WR,                        // event_sequence
      r_opcode == ‘WAIT,
      (r_opcode == ‘WR) ||
      (r_opcode == ‘DONE)}  );
```

Ensures that a ‘WR, ‘WAIT sequence in consecutive cycles is followed by a ‘DONE or ‘WR. The sequence checking is pipelined.



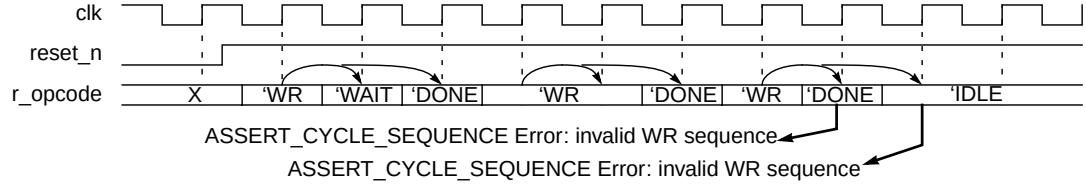

```
assert_cycle_sequence #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_TRIGGER_ON_FIRST_PIPE,                  // necessary_condition
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid WR sequence”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_write_sequence (

      clk,                                      // clock
      reset_n,                                  // reset
      { r_opcode == ‘WR,                        // event_sequence
      (r_opcode == ‘WAIT) ||
      (r_opcode == ‘WR),
      (r_opcode == ‘WAIT) ||
      (r_opcode == ‘DONE)}  );
```

Ensures that a ‘WR is followed by a ‘WAIT or another ‘WR, which is then followed by a ‘WAIT or a ‘DONE (in consecutive cycles). The sequence checking is pipelined: a new ‘WR during a sequence check initiates an additional check.



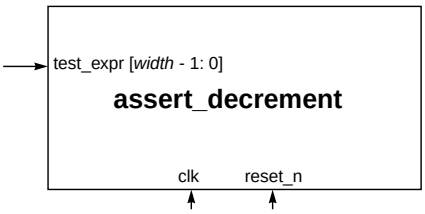

```
assert_cycle_sequence #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_TRIGGER_ON_FIRST_NOPIPE,                // necessary_condition
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid WR sequence”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_write_sequence (

      clk,                                      // clock
      reset_n,                                  // reset
      { r_opcode == ‘WR,                        // event_sequence
      (r_opcode == ‘WAIT) ||
      (r_opcode == ‘WR),
      (r_opcode == ‘DONE)}  );
```

Ensures that a ‘WR is followed by a ‘WAIT or another ‘WR, which is then followed by a ‘DONE (in consecutive cycles). The sequence checking is not pipelined: a new ‘WR during a sequence check does not initiate an additional check.



assert_decrement

Ensures that the value of a specified expression changes only by the specified decrement value.



Parameters:
severity_level
width
value
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_decrement
      [#(severity_level, width, value, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
value	Decrement value for *test_expr*. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should decrement by *value* whenever its value changes from the rising edge of clk to the next rising edge of clk.

Description

The assert_decrement assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the new value equals the previous value decremented by *value*. The

checker allows the value of *test_expr* to wrap, if the total change equals the decrement *value*. For example, if width is 5 and value is 4, then the following change in *test_expr* is valid:

```
5'b00010 –> 5'b11110
```

The checker is useful for ensuring proper changes in structures such as counters and finite-state machines. For example, the checker is useful for circular queue structures with address counters that can wrap. Do not use this checker for variables or expressions that can increment. Instead consider using the assert_delta checker.

Assertion Checks

`ASSERT_DECREMENT`	Expression evaluated to a value that is not its previous value decremented by *value*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	BASIC — Expression changed value.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

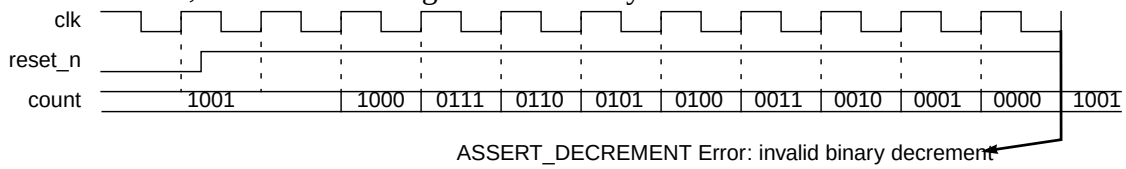
See also

assert_delta
assert_increment
assert_no_underflow

Example

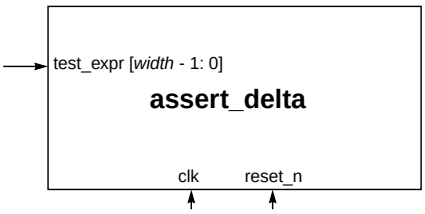
```
assert_decrement #(

   ‘OVL_ERROR,                                  // severity_level
   4,                                           // width
   1,                                           // value
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid binary decrement”,           // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_count (

      clk,                                      // clock
      reset_n,                                  // reset
      count  );                                 // test_expr
```

Ensures that the programmable counter’s `count` variable only decrements by 1. If `count` wraps, the assertion fails, because the change is not a binary decrement.



assert_delta

Ensures that the value of a specified expression changes only by a value in the specified range.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_delta
      [#(severity_level, width, min, max, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
min	Minimum delta value allowed for *test_expr*. Default: 1.
max	Maximum delta value allowed for *test_expr*. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should only change by a delta value in the range min to max.

Description

The assert_delta assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the difference between the new value and the previous value (i.e., the delta value) is in the range from *min* to *max*, inclusive. If the delta value is less than *min* or greater than *max*, the assertion fails.

The checker is useful for ensuring proper changes in control structures such as up-down counters. For these structures, assert_delta can check for underflow and overflow. In datapath and arithmetic circuits, assert_delta can check for "smooth" transitions of the values of various variables (for example, for a variable that controls a physical variable that cannot detect a severe change from its previous value).

Assertion Checks

`ASSERT_DELTA`	Expression changed value by a delta value not in the range *min* to *max*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	BASIC — Expression changed value.
`cover_test_expr_delta_at_min`	CORNER — Expression changed value by a delta equal to *min*.
`cover_test_expr_delta_at_max`	CORNER — Expression changed value by a delta equal to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.
2. The assertion check allows the value of *test_expr* to wrap. The overflow or underflow amount is included in the delta value calculation.

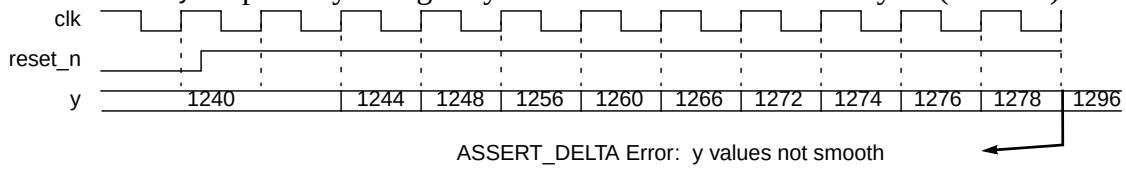
See also

assert_decrement
assert_increment
assert_no_overflow
assert_no_underflow
assert_range

Example

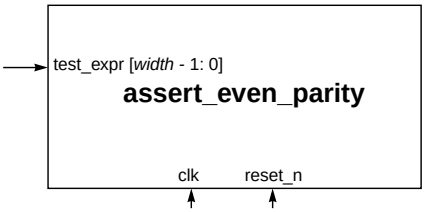
```
assert_delta #(

   ‘OVL_ERROR,                                  // severity_level
   16,                                          // width
   0,                                           // min
   8,                                           // max
   ‘OVL_ASSERT,                                 // property_type
   “Error: y values not smooth”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_smooth (

      clk,                                      // clock
      reset_n,                                  // reset
      y  );                                     // test_expr
```

Ensures that the y output only changes by a maximum of 8 units each cycle (*min* is 0).



assert_even_parity

Ensures that the value of a specified expression has even parity.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_even_parity
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to a value with even parity on the rising clock edge.

Description

The assert_even_parity assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to a value that has even parity. A value has even parity if it is 0 or if the number of bits set to 1 is even.

The checker is useful for verifying control circuits, for example, it can be used to verify a finite-state machine with error detection. In a datapath circuit the checker can perform parity error checking of address and data buses.

Assertion Checks

`ASSERT_EVEN_PARITY`	Expression evaluated to a value whose parity is not even.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	SANITY — Expression has changed value.

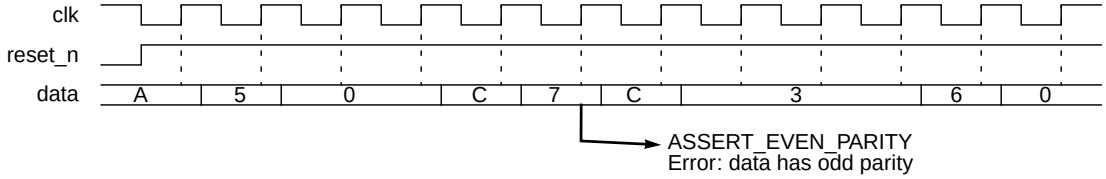
See also

assert_odd_parity

Example

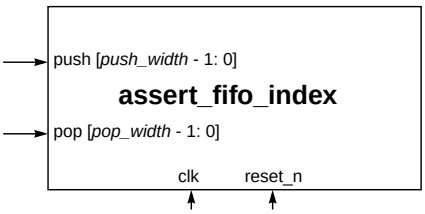
```
assert_even_parity #(

   ‘OVL_ERROR,                                  // severity_level
   8,                                           // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: data has odd parity”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_data_even_parity (

      clk,                                      // clock
      reset_n,                                  // reset
      data );                                   // test_expr
```

Ensures that `data` has even parity at each rising edge of `clk`.



assert_fifo_index

Ensures that a FIFO-type structure never overflows or underflows. This checker can be configured to support multiple pushes (FIFO writes) and pops (FIFO reads) during the same clock cycle.



Parameters:
severity_level
depth
push_width
pop_width
property_type
msg
coverage_level
simultaneous_push_pop

Class:
n-cycle assertion

Syntax

```
assert_fifo_index
      [#(severity_level, depth, push_width, pop_width, property_type,
      msg, coverage_level, simultaneous_push_pop )]
      instance_name (clk, reset_n, push, pop );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
depth	Maximum number of elements in the FIFO or queue structure. This parameter must be > 0. Default: 1.
push_width	Width of the *push* argument. Default: 1.
pop_width	Width of the *pop* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.
simultaneous_push_pop	Whether or not to allow simultaneous push/pop operations in the same clock cycle. When set to 0, if push and pop operations occur in the same cycle, the assertion fails. Default: 1 (simultaneous push/pop operations are allowed).

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
push[*push_width*-1:0]	Expression that indicates the number of push operations that will occur during the current cycle.
pop[*pop_width*-1:0]	Expression that indicates the number of pop operations that will occur during the current cycle.

Description

The assert_fifo_index assertion checker tracks the numbers of pushes (writes) and pops (reads) that occur for a FIFO or queue memory structure. This checker does permit simultaneous pushes/pops on the queue within the same clock cycle. It ensures the FIFO never overflows (i.e., too many pushes occur without enough pops) and never underflows (i.e., too many pops occur without enough pushes). This checker is more complex than the assert_no_overflow and assert_no_underflow checkers, which check only the boundary conditions (overflow and underflow respectively).

Assertion Checks

OVERLOW	Push operation overflowed the FIFO.
UNDERFLOW	Pop operation underflowed the FIFO.
ILLEGAL PUSH AND POP	Push and pop operations performed in the same clock cycle, but the simultaneous_push_pop parameter is set to 0.

Implicit X/Z Checks

push contains X or Z	Push expression value contained X or Z bits.
pop contains X or Z	Pop expression value contained X or Z bits.

Cover Points

cover_fifo_push	BASIC — Push operation occurred.
cover_fifo_pop	BASIC — Pop operation occurred.
cover_fifo_full	CORNER — FIFO was full.
cover_fifo_empty	CORNER — FIFO was empty.
cover_fifo_ simultaneous_push_pop	CORNER — Push and pop operations occurred in the same clock cycle.

Errors

`Depth parameter value must be > 0`	Depth parameter is set to 0.

Notes

1. The checker checks the values of the *push* and *pop* expressions. By default, (i.e., simultaneous_push_pop is 1), “simultaneous” push/pop operations are allowed. In this case, the checker assumes the design properly handles simultaneous push/pop operations, so it only ensures that the FIFO buffer index at the *end of the cycle* has not overflowed or underflowed. The assertion cannot ensure the FIFO buffer index does not overflow between a push and pop performed in the same cycle. Similarly, the assertion cannot ensure the FIFO buffer index does not underflow between a pop and push performed in the same cycle.

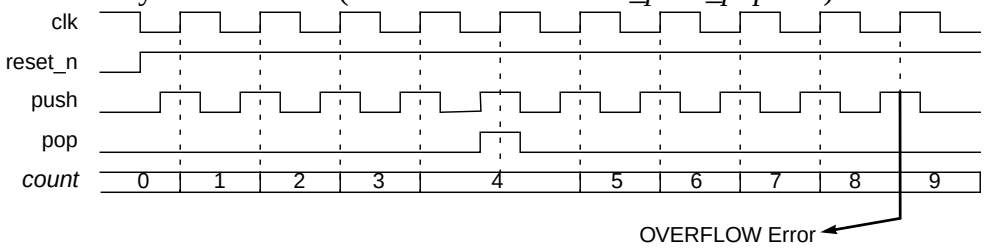
See also

assert_no_overflow assert_no_underflow

Examples

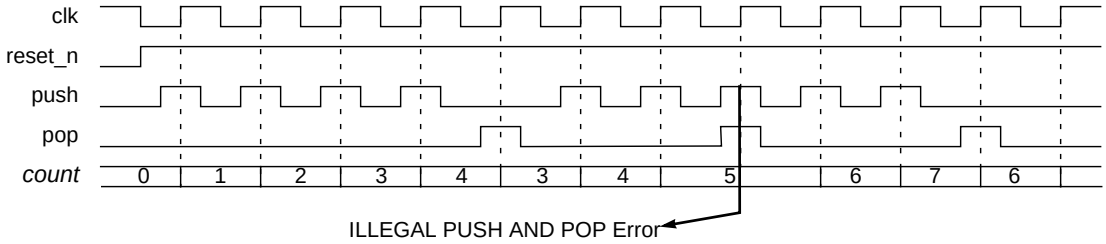
```
assert_fifo_index #(
   ‘OVL_ERROR,                          // severity_level
   8,                                   // depth
   1,                                   // push_width
   1,                                   // pop_width
   ‘OVL_ASSERT,                         // property_type
   “Error”,                             // msg
   ‘OVL_COVER_ALL,                      // coverage_level
   1)                                   // simultaneous_push_pop

   no_over_underflow (

      clk,                              // clock
      reset_n,                          // reset
      push,                             // push
      pop);                             // pop
```

Ensures that an 8-element FIFO never overflows or underflows. Only single pushes and pops can occur in a clock cycle (*push_width* and *pop_width* values are 1). A push and pop operation in the same clock cycle is allowed (value of *simultaneous_push_pop* is 1).



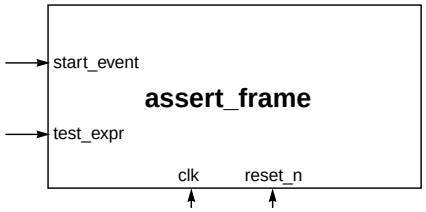
```
assert_fifo_index #(

   ‘OVL_ERROR,                                  // severity_level
   8,                                           // depth
   1,                                           // push_width
   1,                                           // pop_width
   ‘OVL_ASSERT,                                 // property_type
   “violation”,                                 // msg
   ‘OVL_COVER_ALL                               // coverage_level
   0)                                           // simultaneous_push_pop

   no_over_underflow (

      clk,                                      // clock
      reset_n,                                  // reset
      push,                                     // push
      pop);                                     // pop
```

Ensures that an 8-element FIFO never overflows or underflows and that in no cycle do both push and pop operations occur.



assert_frame

Ensures that when a specified start event is TRUE, then a specified expression must not evaluate TRUE before a minimum number of clock cycles and must transition to TRUE no later than a maximum number of clock cycles.



Parameters:
severity_level
min_cks
max_cks
action_on_new_start
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_frame
      [#(severity_level, min_cks, max_cks, action_on_new_start,
      property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
min_cks	Number of cycles after the start event that *test_expr* must not evaluate to TRUE. The special case where *min_cks* is 0 turns off minimum checking (i.e., *test_expr* can be TRUE in the same clock cycle as the start event). Default: 0.
max_cks	Number of cycles after the start event that during which *test_expr* must transition to TRUE. The special case where *max_cks* is 0 turns off maximum checking (i.e., *test_expr* does not need to transition to TRUE). Default: 0.
action_on_new_start	Method for handling a new start event that occurs while a check is pending. Values are: ‘OVL_IGNORE_NEW_START, ‘OVL_RESET_ON_NEW_START and ‘OVL_ERROR_ON_NEW_START. Default: ‘OVL_IGNORE_NEW_START.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

`clk`	Clock event for the assertion. The checker samples on the rising edge of the clock.
`start_event`	Expression that (along with *action_on_new_start*) identifies when to initiate checking of *test_expr*.
`test_expr`	Expression that should not evaluate to TRUE for *min_cks -1* cycles after *start_event* initiates a check (unless *min_cks* is 0) and that should evaluate to TRUE before *max_cks* cycles transpire (unless *max_cks* is 0).

Description

The assert_frame assertion checker checks for a start event at each rising edge of *clk*. A start event occurs if *start_event* has transitioned to TRUE, either at the clock edge or in the previous cycle. A start event also occurs if *start_event* is TRUE at the rising clock edge after a checker reset.

When a start event occurs, the checker performs the following steps:

1. Unless it is disabled by setting *min_cks* to 0, a minimum check is initiated. The check evaluates *test_expr* at each subsequent rising edge of *clk* for the next *min_cks* cycles. However, if a sampled value of *test_expr* is TRUE, the minimum check fails and the checker returns to the state of waiting for a start event.
2. Unless it is disabled by setting *max_cks* to 0 (or a minimum violation has occurred), a maximum check is initiated. The check evaluates *test_expr* at each subsequent rising edge of *clk* for the next (*max_cks - min_cks*) cycles. However, if a sampled value of *test_expr* is TRUE, the checker returns to the state of waiting for a start event. If its value does not transition to TRUE by the time *max_cks* cycles transpire (from the start of checking), the maximum check fails at cycle *max_cks*.
3. The checker returns to the state of waiting for a start event.

The method used to determine how to handle *start_event* when the checker is in the state of checking *test_expr* is controlled by the *action_on_new_start* parameter. The checker has the following actions:

- ‘OVL_IGNORE_NEW_START

  The checker does not sample *start_event* until it returns to the state of waiting for a start event.

- ‘OVL_RESET_ON_NEW_START

  Each time the checker samples *test_expr*, it also samples *start_event*. If *start_event* is TRUE, the checker first checks whether a pending minimum check is just failing. If so,

the assertion failed. Then—unless the assertion failed and it was fatal—the checker terminates the current checks and initiates a new pair of checks.

- ‘OVL_ERROR_ON_NEW_START

  Each time the checker samples *test_expr*, it also samples *start_event*. If *start_event* is TRUE, the assertion fails with an illegal start event error. If the error is not fatal, the checker returns to the state of waiting for a start event at the next rising clock edge.

Assertion Checks

`ASSERT_FRAME`	The value of *test_expr* was TRUE before *min_cks* cycles after *start_event* was sampled TRUE or its value was not TRUE before *max_cks* cycles transpired after the rising edge of *start_event*.
`illegal start event`	The *action_on_new_start* parameter is set to ‘OVL_ERROR_ON_NEW_START and *start_event* expression evaluated to TRUE while the checker was monitoring *test_expr*.
`min_cks > max_cks`	The *min_cks* parameter is greater than the *max_cks* parameter (and *max_cks* >0). Unless the violation is fatal, either the minimum or maximum check will fail.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.
`start_event contains X or Z`	Start event value was X or Z.

Cover Points

`start_event`	BASIC — The value of *start_event* was TRUE on a rising edge of *clk*.

Notes

1. The special case where *min_cks* and *max_cks* are both 0 is the default. Here, *test_expr* must be TRUE every cycle there is a start event.

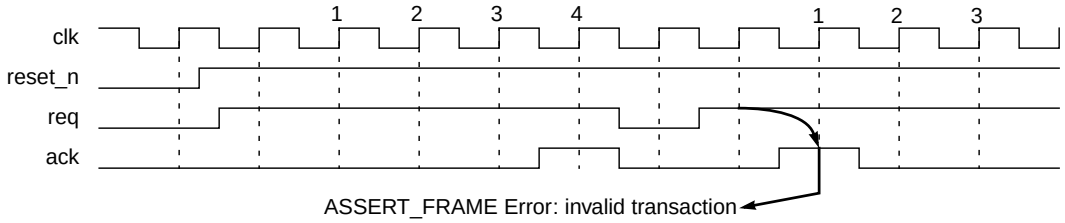
See also

assert_change
assert_next
assert_time
assert_unchange
assert_width

Examples

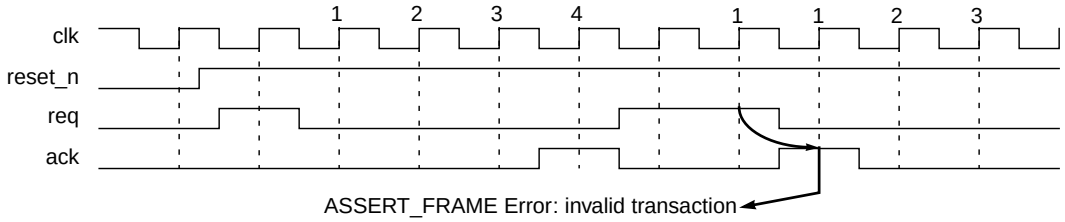
```
assert_frame #(

   ‘OVL_ERROR,                                  // severity_level
   2,                                           // min_cks
   4,                                           // max_cks
   ‘OVL_IGNORE_NEW_START,                       // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req,                                      // start_event
      ack);                                     // test_expr
```

Ensures that after a rising edge of `req`, `ack` goes high between 2 and 4 cycles later. New start events during transactions are not considered to be new transactions and are ignored.



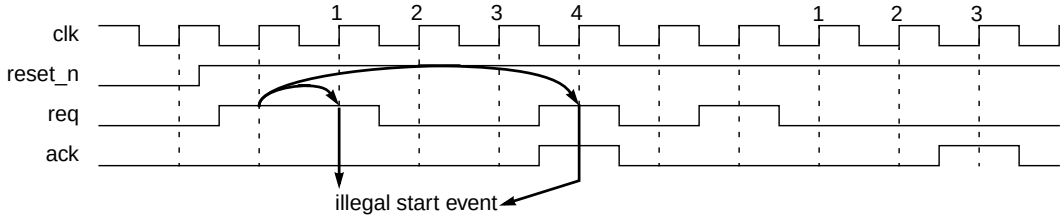

```
assert_frame #(

   ‘OVL_ERROR,                                  // severity_level
   2,                                           // min_cks
   4,                                           // max_cks
   ‘OVL_RESET_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req,                                      // start_event
      ack);                                     // test_expr
```

Ensures that after a rising edge of `req`, `ack` goes high between 2 and 4 cycles later. A new start event during a transaction restarts the transaction.



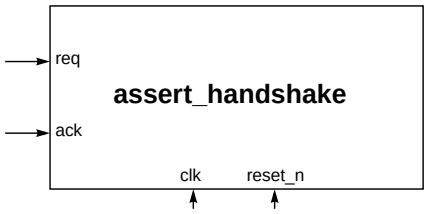
```
assert_frame #(

   ‘OVL_ERROR,                                  // severity_level
   2,                                           // min_cks
   4,                                           // max_cks
   ‘OVL_ERROR_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req,                                      // start_event
      ack);                                     // test_expr
```

Ensures that after a rising edge of `req`, `ack` goes high between 2 and 4 cycles later. Also ensures that a new transaction does not start before the previous transaction is acknowledged. If a start event occurs during a transaction, the checker does does not initiate a new check.



assert_handshake

Ensures that specified request and acknowledge signals follow a specified handshake protocol.



Parameters:
severity_level
min_ack_cycle
max_ack_cycle
req_drop
deassert_count
max_ack_length
property_type
msg
coverage_level

Class:
event-bounded assertion

Syntax

```
assert_handshake
      [#(severity_level, min_ack_cycle, max_ack_cycle, req_drop,
      deassert_count, max_ack_length, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, req, ack );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
min_ack_cycle	Minimum number of clock cycles before acknowledge. A value of 0 turns off the ack min cycle check. Default: 0.
max_ack_cycle	Maximum number of clock cycles before acknowledge. A value of 0 turns off the ack max cycle check. Default: 0.
req_drop	If greater than 0, value of *req* must remain TRUE until acknowledge. A value of 0 turns off the req drop check. Default: 0.
deassert_count	Maximum number of clock cycles after acknowledge that *req* can remain TRUE (i.e., *req* must not be stuck active). A value of 0 turns off the req deassert check. Default: 0.
max_ack_length	Maximum number of clock cycles that *ack* can be TRUE. A value of 0 turns off the max ack length check. Default: 0.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
req	Expression that starts a transaction.
ack	Expression that indicates the transaction is complete.

Description

The assert_handshake assertion checker checks the single-bit expressions *req* and *ack* at each rising edge of *clk* to verify their values conform to the request-acknowledge handshake protocol specified by the checker parameters. A request event (where *req* transitions to TRUE) initiates a transaction on the rising edge of the clock and an acknowledge event (where *ack* transitions to TRUE) signals the transaction is complete on the rising edge of the clock. The transaction must not include multiple request events and every acknowledge must have a pending request. Other checks—to ensure the acknowledge is received in a specified window, the request is held active until the acknowledge, the requests and acknowledges are not stuck active and the pulse length is not too long—are enabled and controlled by the checker's parameters.

When a violation occurs, the checker discards any pending request. Checking is restarted the next cycle that *ack* is sampled FALSE.

Assertion Checks

`multiple req violation`	The value of *req* transitioned to TRUE while waiting for an acknowledge or while acknowledge was asserted. Extra requests do not initiate new transactions.
`ack without req violation`	The value of *ack* transitioned to TRUE without a pending request.
`ack min cycle violation`	The value of ack transitioned to TRUE before *min_ack_cycle* clock cycles transpired after the request.
`ack max cycle violation`	The value of *ack* did not transition to TRUE before *max_ack_cycle* clock cycles transpired after the request.
`req drop violation`	The value of *req* transitioned from TRUE before an acknowledge.
`req deassert violation`	The value of req did not transition from TRUE before *deassert_count* clock cycles transpired after an acknowledge.
`ack max length violation`	The value of ack did not transition from TRUE before *max_ack_length* clock cycles transpired after an acknowledge.

Implicit X/Z Checks

`req contains X or Z`	Req expression value was X or Z.
`ack contains X or Z`	Ack expression value was X or Z.

Cover Points

`cover_req_asserted`	BASIC — A transaction initiated.
`cover_ack_asserted`	BASIC — A transaction completed.

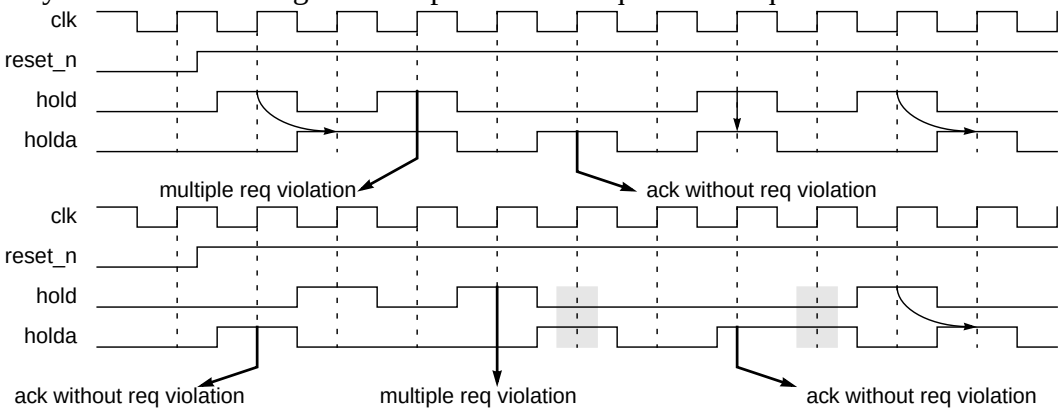
See also

assert_win_change
assert_win_unchange
assert_window

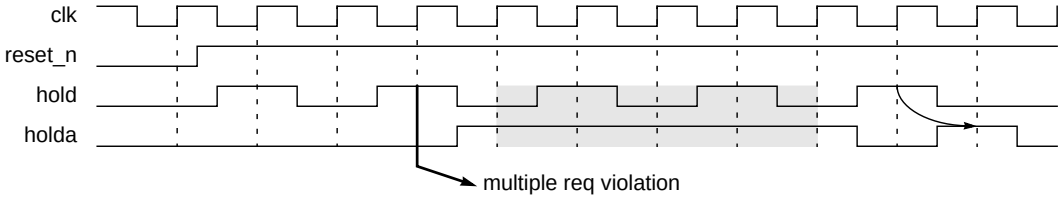
Examples

```
assert_handshake #(
   'OVL_ERROR,                                  // severity_level
   0,                                           // min_ack_cycle
   0,                                           // max_ack_cycle
   0,                                           // req_drop
   0,                                           // deassert_count
   0,                                           // max_ack_length
   'OVL_ASSERT,                                 // property_type
   "hold-holda handshake error",                // msg
   'OVL_COVER_ALL)                              // coverage_level

   valid_hold_holda (

      clk,                                      // clock
      reset_n,                                  // reset
      hold,                                     // req
      holda);                                   // ack
```

Ensures that multiple `hold` requests are not made while waiting for a `holda` acknowledge and that every `holda` acknowledge is in response to a unique `hold` request.

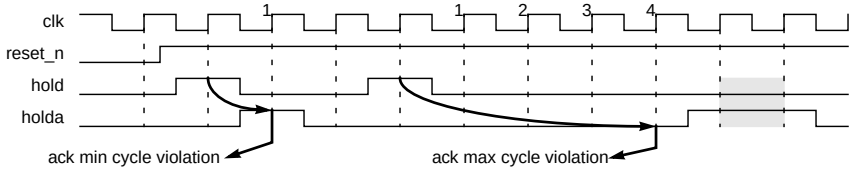


After a violation, checking is turned off until `holda` acknowledge is sampled deasserted.



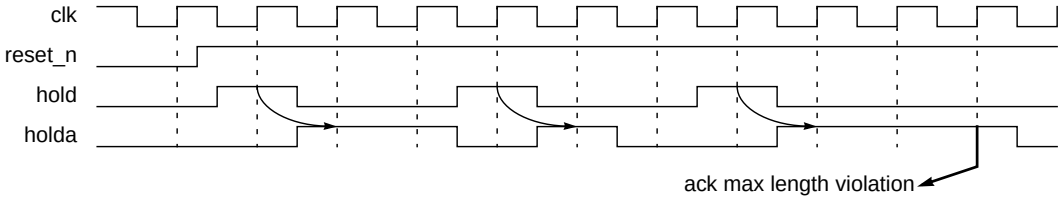

```
assert_handshake #(

   ‘OVL_ERROR,                                  // severity_level
   2,                                           // min_ack_cycle
   3,                                           // max_ack_cycle
   0,                                           // req_drop
   0,                                           // deassert_count
   0,                                           // max_ack_length
   ‘OVL_ASSERT,                                 // property_type
   “hold-holda handshake error”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_hold_holda (

      clk,                                      // clock
      reset_n,                                  // reset
      hold,                                     // req
      holda);                                   // ack
```

Ensures that multiple `hold` requests are not made while waiting for a `holda` acknowledge and that every `holda` acknowledge is in response to a unique `hold` request. Ensures `holda` acknowledge asserts 2 to 3 cycles after each hold request.



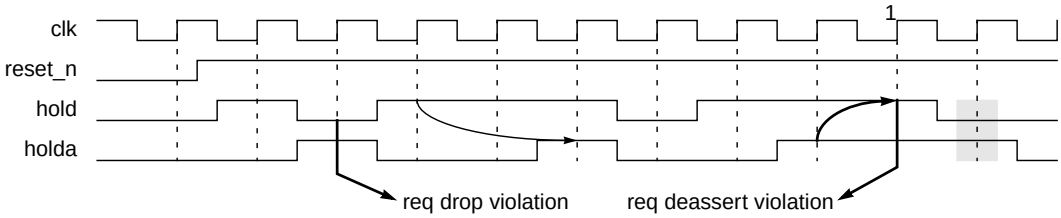
```
assert_handshake #(

   ‘OVL_ERROR,                                  // severity_level
   0,                                           // min_ack_cycle
   0,                                           // max_ack_cycle
   0,                                           // req_drop
   0,                                           // deassert_count
   2,                                           // max_ack_length
   ‘OVL_ASSERT,                                 // property_type
   “hold-holda handshake error”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_hold_holda (

      clk,                                      // clock
      reset_n,                                  // reset
      hold,                                     // req
      holda);                                   // ack
```

Ensures that multiple `hold` requests are not made while waiting for a `holda` acknowledge and that every `holda` acknowledge is in response to a unique `hold` request. Ensures `holda` acknowledge asserts for 2 cycles.



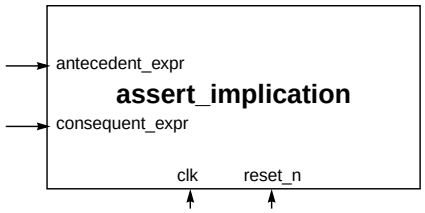

```
assert_handshake #(

   ‘OVL_ERROR,                                  // severity_level
   0,                                           // min_ack_cycle
   0,                                           // max_ack_cycle
   1,                                           // req_drop
   1,                                           // deassert_count
   0,                                           // max_ack_length
   ‘OVL_ASSERT,                                 // property_type
   “hold-holda handshake error”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_hold_holda (

      clk,                                      // clock
      reset_n,                                  // reset
      hold,                                     // req
      holda);                                   // ack
```

Ensures that multiple `hold` requests are not made while waiting for a `holda` acknowledge and that every `holda` acknowledge is in response to a unique `hold` request. Ensures `hold` request remains asserted until its `holda` acknowledge and then deasserts in the next cycle.



assert_implication

Ensures that a specified consequent expression is TRUE if the specified antecedent expression is TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_implication
      [#(severity_level, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, antecedent_expr, consequent_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
antecedent_expr	Antecedent expression that is tested at the clock event.
consequent_expr	Consequent expression that should evaluate to TRUE if *antecedent_expr* evaluates to TRUE when tested.

Description

The assert_implication assertion checker checks the single-bit expression *antecedent_expr* at each rising edge of *clk*. If *antecedent_expr* is TRUE, then the checker verifies that the value of *consequent_expr* is also TRUE. If *antecedent_expr* is not TRUE, then the assertion is valid regardless of the value of *consequent_expr*.

Assertion Checks

`ASSERT_IMPLICATION`	Expression evaluated to FALSE.

Implicit X/Z Checks

`antecedent_expr contains X or Z`	Antecedent expression value was X or Z.
`consequent_expr contains X or Z`	Consequent expression value was X or Z.

Cover Points

`cover_antecedent`	BASIC — The *antecedent_expr* evaluated to TRUE.

Notes

1. This assertion checker is equivalent to:

```
assert_always
   [#(severity_level, property_type, msg, coverage_level )]
   instance_name (clk, reset_n,
   (antecedent_expr  ? consequent_expr  : 1'b1 ));
```

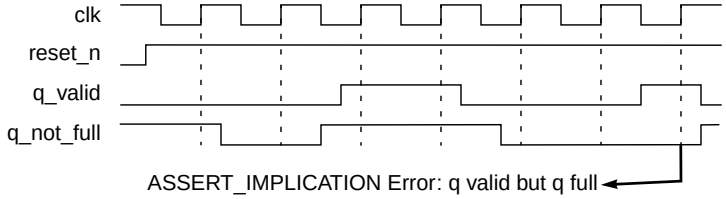
See also

assert_always
assert_always_on_edge
assert_never
assert_proposition

Example

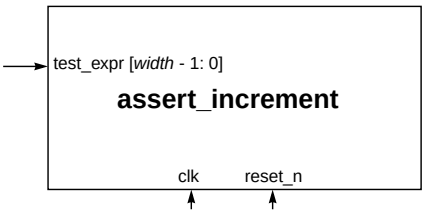
```
assert_implication #(

   `OVL_ERROR,                                  // severity_level
   `OVL_ASSERT,                                 // property_type
   "Error: q valid but q full",                 // msg
   `OVL_COVER_ALL)                              // coverage_level

   not_full (

      clk,                                      // clock
      reset_n,                                  // reset
      q_valid,                                  // antecedent_expr
      q_not_full );                             // consequent_expr
```

Ensures that `q_not_full` is TRUE at each rising edge of `clk` for which `q_valid` is TRUE.



assert_increment

Ensures that the value of a specified expression changes only by the specified increment value.



Parameters:
severity_level
width
value
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_increment
      [#(severity_level, width, value, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
value	Increment value for *test_expr*. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
`test_expr[width-1:0]`	Expression that should increment by *value* whenever its value changes from the rising edge of clk to the next rising edge of clk.

Description

The assert_increment assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the new value equals the previous value incremented by *value*. The checker allows the value of *test_expr* to wrap, if the total change equals the increment *value*. For example, if *width* is 5 and *value* is 4, then the following change in *test_expr* is valid:

```
5'b11110 –> 5'b00010
```

The checker is useful for ensuring proper changes in structures such as counters and finite-state machines. For example, the checker is useful for circular queue structures with address counters that can wrap. Do not use this checker for variables or expressions that can decrement. Instead consider using the assert_delta checker.

Assertion Checks

`ASSERT_INCREMENT`	Expression evaluated to a value that is not its previous value incremented by *value*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	BASIC — Expression changed value.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

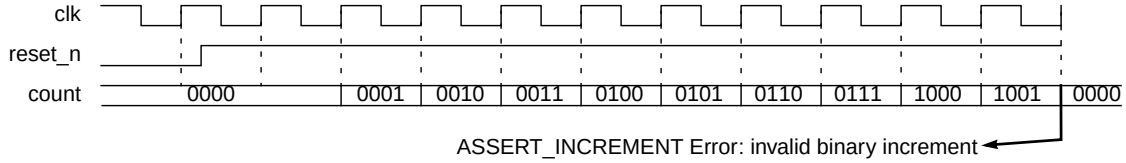
See also

assert_decrement
assert_delta
assert_no_overflow

Example

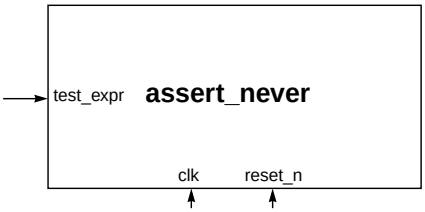
```
assert_increment #(

   `OVL_ERROR,                                  // severity_level
   4,                                           // width
   1,                                           // value
   `OVL_ASSERT,                                 // property_type
   "Error: invalid binary increment",           // msg
   `OVL_COVER_ALL)                              // coverage_level

   valid_count (

      clk,                                      // clock
      reset_n,                                  // reset
      count  );                                 // test_expr
```

Ensures that the programmable counter's `count` variable only increments by 1. If `count` wraps, the assertion fails, because the change is not a binary increment.



assert_never

Ensures that the value of a specified expression is not TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_never
      [#(severity_level, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should not evaluate to TRUE on the rising clock edge.

Description

The assert_never assertion checker checks the single-bit expression *test_expr* at each rising edge of *clk* to verify the expression does not evaluate to TRUE.

Assertion Checks

ASSERT_NEVER	Expression evaluated to TRUE.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

none

Notes

1. By default, the assert_never assertion is pessimistic and the assertion fails if *test_expr* is not 0 (i.e.equals 1, X, Z, etc.). However, if ‘OVL_XCHECK_OFF is set, the assertion fails if and only if *test_expr* is 1.

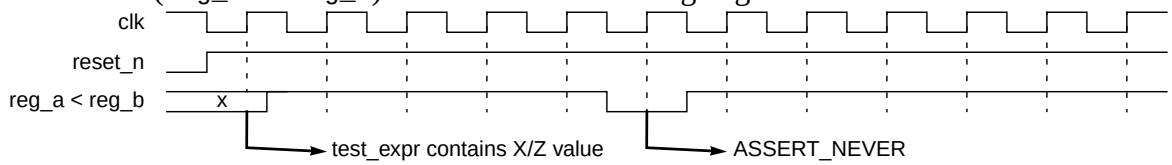
See also

assert_always
assert_always_on_edge
assert_implication
assert_proposition

Example

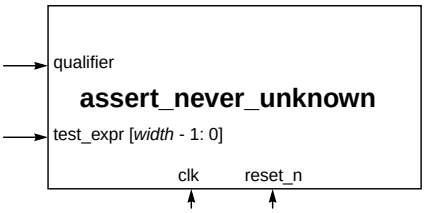
```
assert_never #(

   ‘OVL_ERROR,                          // severity_level
   ‘OVL_ASSERT,                         // property_type
   “”,                                  // msg
   ‘OVL_COVER_ALL)                      // coverage_level

   valid_count (

      clk,                              // clock
      reset_n,                          // reset
      reg_a < reg_b );                  // test_expr
```

Ensures that (`reg_a < reg_b`) is FALSE at each rising edge of `clk`.



assert_never_unknown

Ensures that the value of a specified expression contains only 0 and 1 bits when a qualifying expression is TRUE.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_never_unknown
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, qualifier, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
qualifier	Expression that indicates whether or not to check *test_expr* .
test_expr[*width*-1:0]	Expression that should contain only 0 or 1 bits when qualifier is TRUE.

Description

The assert_never_unknown assertion checker checks the expression *qualifier* at each rising edge of *clk* to determine if it should check *test_expr*. If *qualifier* is sampled TRUE, the checker evaluates *test_expr* and if the value of *test_expr* contains a bit that is not 0 or 1, the assertion fails.

The checker is useful for ensuring certain data have only known values following a reset sequence. It also can be used to verify tristate input ports are driven and tristate output ports drive known values when necessary.

Assertion Checks

`test_expr contains X/Z value`	The *test_expr* expression contained at least one bit that was not 0 or 1; *qualifier* was sampled TRUE; and ‘OVL_XCHECK_OFF is not set.

Cover Points

`cover_qualifier`	BASIC — A never_unknown check was initiated.
`cover_test_expr_change`	SANITY — Expression changed value.

Notes

1. If ‘OVL_XCHECK_OFF is set, all assert_never_unknown checkers are turned off.

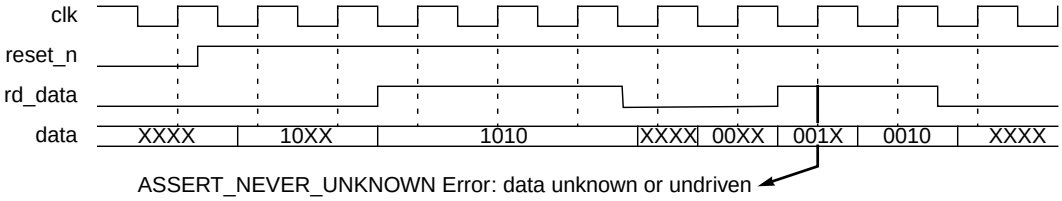
See also

assert_never
assert_never_unknown_async
assert_one_cold
assert_one_hot
assert_zero_one_hot

Example

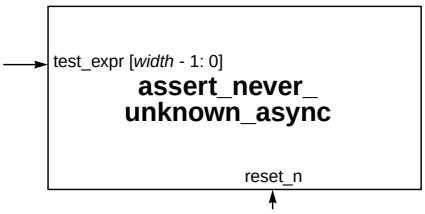
```
assert_never_unknown #(

  ‘OVL_ERROR,                                  // severity_level
  8,                                           // width
  ‘OVL_ASSERT,                                 // property_type
  “Error: data unknown or undriven”,           // msg
  ‘OVL_COVER_ALL)                              // coverage_level

  valid_data (

     clk,                                      // clock
     reset_n,                                  // reset
     rd_data,                                  // qualifier
     data);                                    // test_expr
```

Ensures that values of `data` are known and driven when `rd_data` is TRUE.



assert_never_unknown_async

Ensures that the value of a specified expression combinationally contains only 0 and 1 bits.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
combinational
assertion

Syntax

```
assert_never_unknown_async
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should contain only 0 or 1 bits when qualifier is TRUE.

Description

The assert_never_unknown_async assertion checker combinationally evaluates *test_expr* and if the value of *test_expr* contains a bit that is not 0 or 1, the assertion fails.

The checker is useful for ensuring certain data have only known values following a reset sequence. It also can be used to verify tristate input ports are driven and tristate output ports drive known values when necessary.

Assertion Checks

`test_expr contains X/Z value`	The *test_expr* expression contained at least one bit that was not 0 or 1 and ‘OVL_XCHECK_OFF is not set.

Cover Points

`cover_test_expr_change`	SANITY — Expression changed value.

Notes

1. If ‘OVL_XCHECK_OFF is set, all assert_never_unknown_async checkers are turned off.

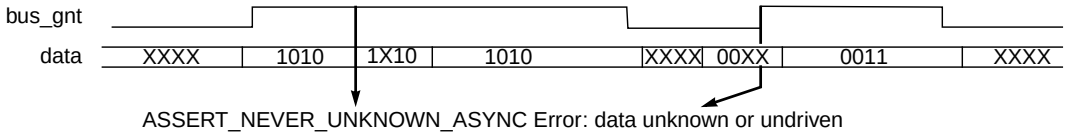
See also

assert_never

Example

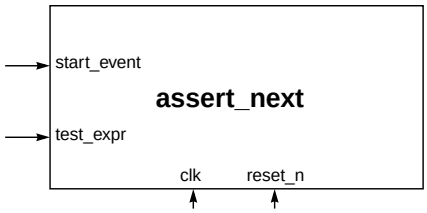
```
assert_never_unknown_async #(

  ‘OVL_ERROR,                                    // severity_level
  8,                                             // width
  ‘OVL_ASSERT,                                   // property_type
  “Error: data unknown or undriven”,             // msg
  ‘OVL_COVER_ALL)                                // coverage_level

  valid_data (

     bus_gnt,                                    // reset
     data);                                      // test_expr
```

Ensures that values of `data` are known and driven while `bus_gnt` is TRUE.



assert_next

Ensures that the value of a specified expression is TRUE a specified number of cycles after a start event.



Parameters:
severity_level
num_cks
check_overlapping
check_missing_start
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_next
      [#(severity_level, num_cks, check_overlapping,
      check_missing_start, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
num_cks	Number of cycles after *start_event* is TRUE to wait to check that the value of *test_expr* is TRUE. Default: 1.
check_overlapping	Whether or not to perform overlap checking. Default: 1 (overlap checking off). • If set to 0, overlap checking is performed. From the rising edge of *clk* after *start_event* is sampled TRUE to the rising edge of *clk* of the cycle before *test_expr* is sampled for the current next check, the checker performs an overlap check. During this interval, if *start_event* is TRUE at a rising edge of *clk*, then the overlap check fails (illegal overlapping condition). The current next check continues but a new next check is not initiated. • If set to 1, overlap checking is not performed. A separate next check is initiated each time *start_event* is sampled TRUE (overlapping start events are allowed).

check_missing_start	Whether or not to perform missing-start checking. Default: 0 (missing-start checking off). • If set to 0, missing start checks are not performed. • If set to 1, missing start checks are performed. The checker samples *test_expr* every rising edge of *clk*. If the value of *test_expr* is TRUE, then *num_cks* rising edges of *clk* prior to the current time, *start_event* must have been TRUE (initiating a next check). If not, the missing-start check fails (*start_event* without *test_expr*).
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
start_event	Expression that (along with *num_cks*) identifies when to check *test_expr*.
test_expr	Expression that should evaluate to TRUE *num_cks* cycles after *start_event* initiates a next check.

Description

The assert_next assertion checker checks the expression *start_event* at each rising edge of *clk*. If *start_event* is TRUE, a check is initiated. The check waits for *num_cks* cycles (i.e., for *num_cks* additional rising edges of *clk*) and evaluates *test_expr*. If *test_expr* is not TRUE, the assertion fails.

If overlap checking is off (*check_overlapping* is 1), additional checks can start while a current check is pending. If overlap checking is on, the assertion fails if *start_event* is sampled TRUE while a check is pending (except on the last clock).

If missing-start checking is off (*check_missing_start* is 0), *test_expr* can be TRUE any time. If missing-start checking is on, the assertion fails if *test_expr* is TRUE without a corresponding start event (*num_cks* cycles previously). However, if *test_expr* is TRUE in the interval of *num_cks* - 1 cycles after a reset and has no corresponding start event, the result is indeterminate (i.e., the missing-start check might or might not fail).

Assertion Checks

`start_event without test_expr`	The value of *start_event* was TRUE on a rising edge of *clk*, but *num_cks* cycles later the value of *test_expr* was not TRUE.
`illegal overlapping condition detected`	The *check_overlapping* parameter is set to 0 and *start_event* was TRUE on the rising edge of *clk*, but a previous check was pending.
`test_expr without start_event`	The *check_missing_start* parameter is set to 1 and *start_event* was not TRUE on the rising edge of *clk*, but *num_cks* cycles later *test_expr* was TRUE.
`num_cks parameter<=0`	The *num_cks* parameter is less than 2.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.
`start_event contains X or Z`	Start event value was X or Z.

Cover Points

`cover_start_event`	BASIC — The value of *start_event* was TRUE on a rising edge of *clk*.
`cover_overlapping_ start_events`	CORNER — The value of *start_event* was TRUE on a rising edge of *clk* while a check was pending.

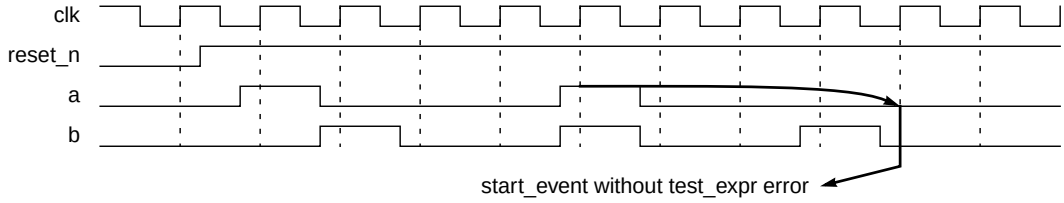
See also

assert_change
assert_frame
assert_time
assert_unchange

Examples

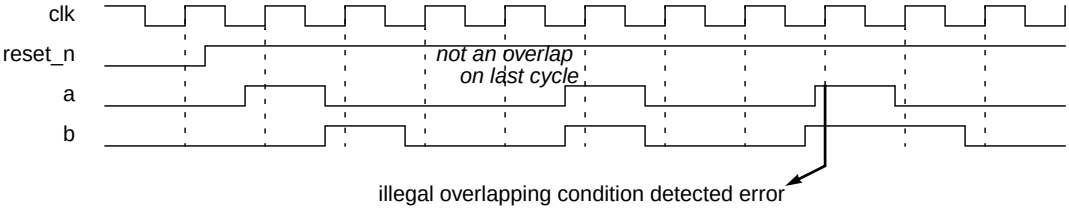
```
assert_next #(
   ‘OVL_ERROR,                          // severity_level
   4,                                   // num_cks
   1,                                   // check_overlapping (off)
   0,                                   // check_missing_start (off)
   ‘OVL_ASSERT,                         // property_type
   “error:”,                            // msg
   ‘OVL_COVER_ALL)                      // coverage_level

   valid_next_a_b (

      clk,                              // clock
      reset_n,                          // reset
      a,                                // start_event
      b );                              // test_expr
```

Ensures that b is TRUE 4 cycles after a is TRUE.



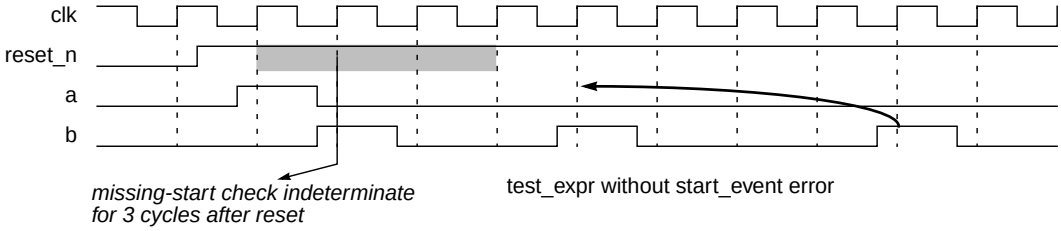

```
assert_next #(

   ‘OVL_ERROR,                                  // severity_level
   4,                                           // num_cks
   0,                                           // check_overlapping (on)
   0,                                           // check_missing_start (off)
   ‘OVL_ASSERT,                                 // property_type
   “error:”,                                    // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_next_a_b (

      clk,                                      // clock
      reset_n,                                  // reset
      a,                                        // start_event
      b );                                      // test_expr
```

Ensures that b is TRUE 4 cycles after a is TRUE. Overlaps are not allowed



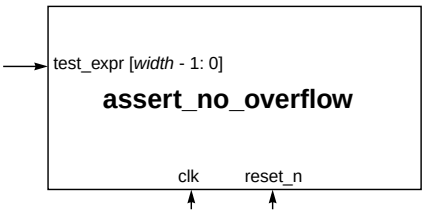

```
assert_next #(

   ‘OVL_ERROR,                                  // severity_level
   4,                                           // num_cks
   1,                                           // check_overlapping (off)
   1,                                           // check_missing_start (on)
   ‘OVL_ASSERT,                                 // property_type
   “error:”,                                    // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_next_a_b (

      clk,                                      // clock
      reset_n,                                  // reset
      a,                                        // start_event
      b );                                      // test_expr
```

Ensures that b is TRUE 4 cycles after a is TRUE. Missing-start check is on.



assert_no_overflow

Ensures that the value of a specified expression does not overflow.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_no_overflow
      [#(severity_level, width, min, max, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Width must be less than or equal to 32. Default: 1.
min	Minimum value in the test range of *test_expr*. Default: 0.
max	Maximum value in the test range of *test_expr*. Default: 2***width* - 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should not change from a value of *max* to a value out of the test range or to a value equal to *min*.

Description

The assert_no_overflow assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from a value (at the previous rising edge of *clk*) that was equal to *max*. If so, the checker verifies that the new value has not overflowed *max*. That is, it verifies the value of *test_expr* is not greater than *max* or less than or equal to *min* (in which case, the assertion fails).

The checker is useful for verifying counters, where it can ensure the counter does not wrap from the highest value to the lowest value in a specified range. For example, it can be used to check that memory structure pointers do not wrap around. For a more general test for overflow, use assert_delta or assert_fifo_index.

Assertion Checks

`ASSERT_NO_OVERFLOW`	Expression changed value from *max* to a value not in the range *min* + 1 to *max* - 1.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_at_min`	CORNER — Expression evaluated to *min*.
`cover_test_expr_at_max`	BASIC — Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle for which *test_expr* changed from *max*.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

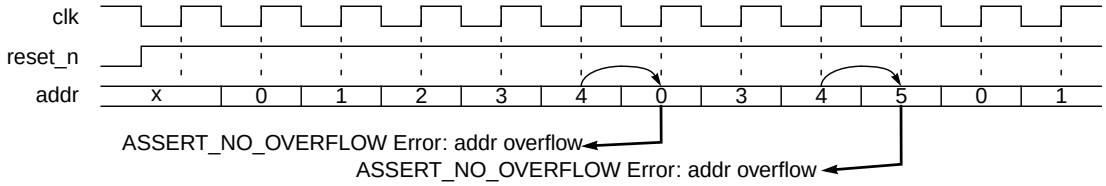
See also

assert_delta
assert_fifo_index
assert_increment
assert_no_overflow

Example

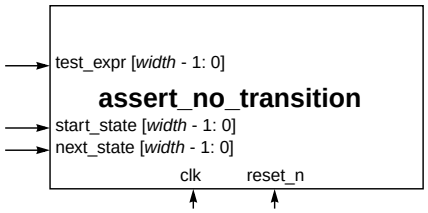
```
assert_no_overflow #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // width
   0,                                           // min
   4,                                           // max
   ‘OVL_ASSERT,                                 // property_type
   “Error: addr overflow”,                      // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   addr_with_overflow (

      clk,                                      // clock
      reset_n,                                  // reset
      addr  );                                  // test_expr
```

Ensures that `addr` does not overflow (i.e., change from a value of 4 at the rising edge of `clk` to a value of 0 or a value greater than 4 at the next rising edge of `clk`).



assert_no_transition

Ensures that the value of a specified expression does not transition from a start state to the specified next state.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_no_transition
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr, start_state, next_state );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should not transition to *next_state* on the rising edge of *clk* if its value at the previous rising edge of *clk* is the same as the current value of *start_state*.
start_state[*width*-1:0]	Expression that indicates the start state for the assertion check. If the start state matches the value of *test_expr* on the previous rising edge of *clk*, the check is performed.

next_state[*width*-1:0]	Expression that indicates the invalid next state for the assertion check. If the value of *test_expr* was *start_state* at the previous rising edge of *clk*, then the value of *test_expr* should not equal *next_state* on the current rising edge of *clk*.

Description

The assert_no_transition assertion checker checks the expression *test_expr* and *start_state* at each rising edge of *clk* to see if they are the same. If so, the checker evaluates and stores the current value of *next_state*. At the next rising edge of *clk*, the checker re-evaluates *test_expr* to see if its value equals the stored value of *next_state*. If so, the assertion fails. The checker returns to checking *start_state* in the current cycle (unless a fatal failure occurred)

The *start_state* and *next_state* expressions are verification events that can change. In particular, the same assertion checker can be coded to verify multiple types of transitions of *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) do not transition to invalid values.

Assertion Checks

ASSERT_no_transition	Expression transitioned from *start_state* to a value equal to *next_state*.

Implicit X/Z Checks

test_expr contains X or Z	Expression value contained X or Z bits.
start_state contains X or Z	Start state value contained X or Z bits.
next_state contains X or Z	Next state value contained X or Z bits.

Cover Points

cover_start_state	BASIC — Expression assumed a start state value.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

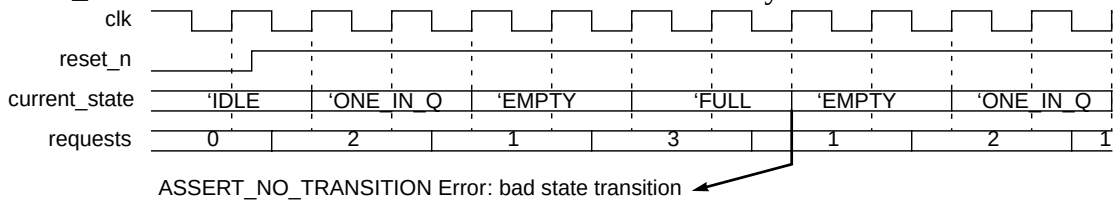
See also

assert_transition

Example

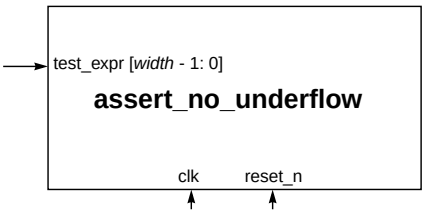
```
assert_no_transition #(

   ‘OVL_ERROR,                                    // severity_level
   3,                                             // width
   ‘OVL_ASSERT,                                   // property_type
   “Error: bad state transition”,                 // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   valid_transition (

      clk,                                        // clock
      reset_n,                                    // reset
      current_state,                              // test_expr
      requests > 2 ? ‘FULL : ‘ONE_IN_Q,           // start_state
      ‘EMPTY;                                     // next_state
```

Ensures that `current_state` does not transition to ‘EMPTY improperly. If `requests` is greater than 2 and the current_state is ‘FULL, `current_state` should not transition to ‘EMPTY in the next cycle. If `requests` is not greater than 2 and `current_state` is ‘ONE_IN_Q, `current_state` should not transition to ‘EMPTY in the next cycle.



assert_no_underflow

Ensures that the value of a specified expression does not underflow.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_no_underflow
      [#(severity_level, width, min, max, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Width must be less than or equal to 32. Default: 1.
min	Minimum value in the test range of *test_expr*. Default: 0.
max	Maximum value in the test range of *test_expr*. Default: 2***width* - 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should not change from a value of *min* to a value out of range or to a value equal to *max*.

Description

The assert_no_underflow assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from a value (at the previous rising edge of *clk*) that was equal to *min*. If so, the checker verifies that the new value has not underflowed *min*. That is, it verifies the value of *test_expr* is not less than *min* or greater than or equal to *max* (in which case, the assertion fails).

The checker is useful for verifying counters, where it can ensure the counter does not wrap from the lowest value to the highest value in a specified range. For example, it can be used to check that memory structure pointers do not wrap around. For a more general test for underflow, use assert_delta or assert_fifo_index.

Assertion Checks

`ASSERT_NO_UNDERFLOW`	Expression changed value from *min* to a value not in the range *min* + 1 to *max* - 1.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_at_min`	BASIC — Expression evaluated to *min*.
`cover_test_expr_at_max`	CORNER — Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle for which *test_expr* changed from *max*.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

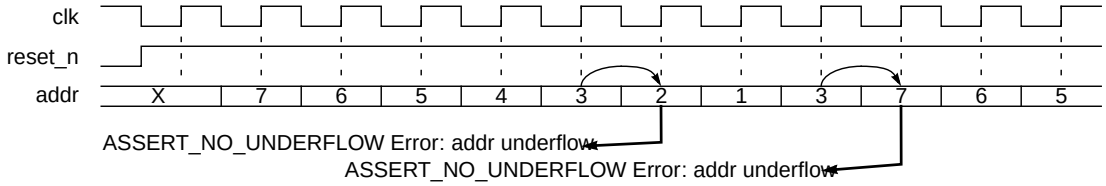
See also

assert_delta
assert_decrement
assert_fifo_index
assert_no_overflow

Example

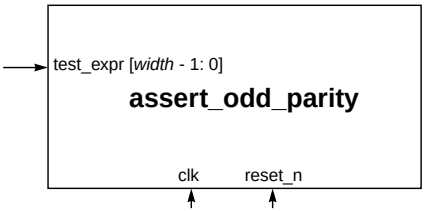
```
assert_no_underflow #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // width
   3,                                           // min
   7,                                           // max
   ‘OVL_ASSERT,                                 // property_type
   “Error: addr underflow”,                     // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   addr_with_underflow (

      clk,                                      // clock
      reset_n,                                  // reset
      addr  );                                  // test_expr
```

Ensures that `addr` does not underflow (i.e., change from a value of 3 at the rising edge of `clk` to a value of 7 or a value less than 3 at the next rising edge of `clk`).



assert_odd_parity

Ensures that the value of a specified expression has odd parity.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_odd_parity
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to a value with odd parity on the rising clock edge.

Description

The assert_odd_parity assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to a value that has odd parity. A value has odd parity if the number of bits set to 1 is odd.

The checker is useful for verifying control circuits, for example, it can be used to verify a finite-state machine with error detection. In a datapath circuit the checker can perform parity error checking of address and data buses.

Assertion Checks

`ASSERT_ODD_PARITY`	Expression evaluated to a value whose parity is not odd.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	SANITY — Expression has changed value.

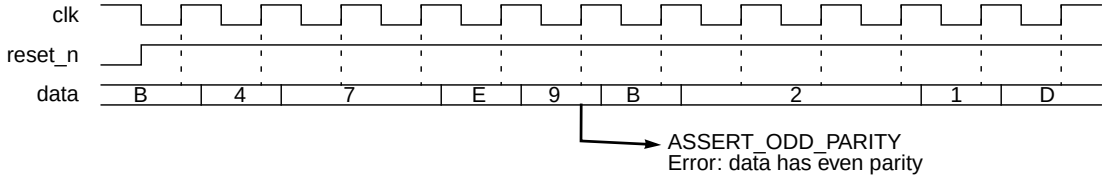
See also

assert_even_parity

Example

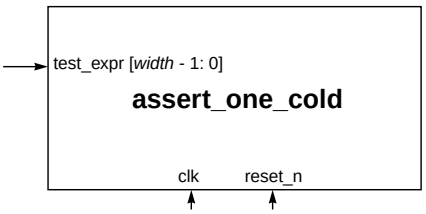
```
assert_odd_parity #(

   ‘OVL_ERROR,                                  // severity_level
   8,                                           // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: data has even parity”,               // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_data_odd_parity (

      clk,                                      // clock
      reset_n,                                  // reset
      data );                                   // test_expr
```

Ensures that `data` has odd parity at each rising edge of `clk`.



assert_one_cold

Ensures that the value of a specified expression is one-cold (or equals an inactive state value, if specified).



Parameters:
severity_level
width
inactive
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_one_cold
      [#(severity_level, width, inactive, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 32.
inactive	Inactive state of *test_expr*: ‘OVL_ALL_ZEROS, ‘OVL_ALL_ONES or ‘OVL_ONE_COLD. Default: ‘OVL_ONE_COLD.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to a one-cold or inactive value on the rising clock edge.

Description

The assert_one_cold assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to a one-cold or inactive state value. A one-cold value has exactly one bit set to 0. The inactive state value for the checker is set by the *inactive* parameter. Choices are: ‘OVL_ALL_ZEROS (e.g., 4‘b0000), ‘OVL_ALL_ONES (e.g.,4‘b1111) or ‘OVL_ONE_COLD. The default *inactive* parameter value is ‘OVL_ONE_COLD, which indicates *test_expr* has no inactive state (so only a one-cold value is valid for each check).

The checker is useful for verifying control circuits, for example, it can ensure that a finite-state machine with one-cold encoding operates properly and has exactly one bit asserted low. In a datapath circuit the checker can ensure that the enabling conditions for a bus do not result in bus contention.

Assertion Checks

`ASSERT_ONE_COLD`	Expression assumed an active state with multiple bits set to 0.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	SANITY — Expression has changed value.
`cover_all_one_colds_checked`	CORNER — Expression evaluated to all possible combinations of one-cold values.
`cover_test_expr_all_zeros`	CORNER — Expression evaluated to the inactive state and the *inactive* parameter was set to ‘OVL_ALL_ZEROS.
`cover_test_expr_all_ones`	CORNER — Expression evaluated to the inactive state and the *inactive* parameter was set to ‘OVL_ALL_ONES.

Notes

1. By default, the assert_one_cold assertion is pessimistic and the assertion fails if test_expr is active and multiple bits are not 1 (i.e.equals 0, X, Z, etc.). However, if ‘OVL_XCHECK_OFF is set, the assertion fails if and only if test_expr is active and multiple bits are 0.

See also

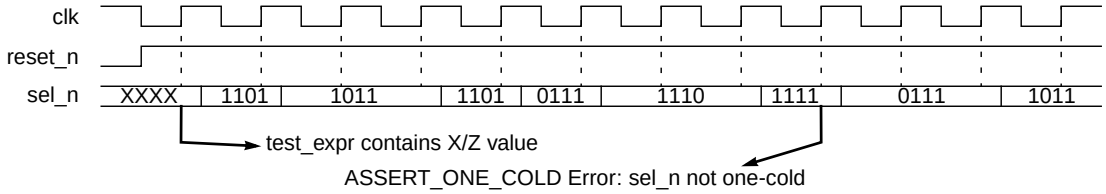
assert_one_hot

assert_zero_one_hot

Examples

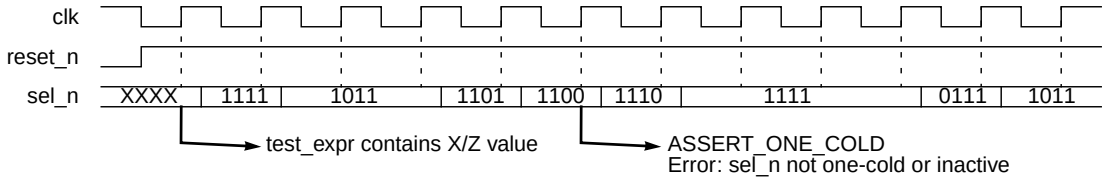
```
assert_one_cold #(

   `OVL_ERROR,                                  // severity_level
   4,                                           // width
   `OVL_ONE_COLD,                               // inactive (no inactive state)
   `OVL_ASSERT,                                 // property_type
   "Error: sel_n not one-cold",                 // msg
   `OVL_COVER_ALL)                              // coverage_level

   valid_sel_n_one_cold (

      clk,                                      // clock
      reset_n,                                  // reset
      sel_n );                                  // test_expr
```

Ensures that `sel_n` is one-cold at each rising edge of `clk`.



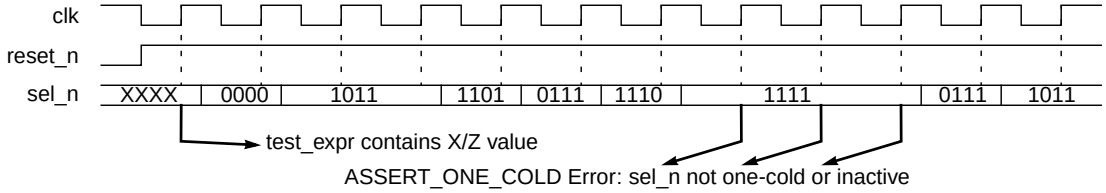

```
assert_one_cold #(

   `OVL_ERROR,                                  // severity_level
   4,                                           // width
   `OVL_ALL_ONES,                               // inactive
   `OVL_ASSERT,                                 // property_type
   "Error: sel_n not one-cold or inactive",     // msg
   `OVL_COVER_ALL)                              // coverage_level

   valid_sel_n_one_cold (

      clk,                                      // clock
      reset_n,                                  // reset
      sel_n );                                  // test_expr
```

Ensures that `sel_n` is one-cold or inactive (4'b1111) at each rising edge of `clk`.



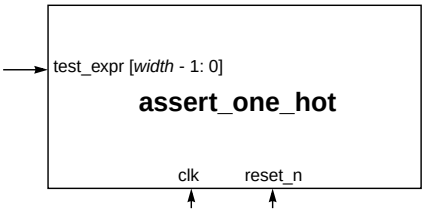
```
assert_one_cold #(

   ‘OVL_ERROR,                                  // severity_level
   4,                                           // width
   ‘OVL_ALL_ZEROS,                              // inactive
   ‘OVL_ASSERT,                                 // property_type
   “Error: sel_n not one-cold”,                 // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sel_n_one_cold (

      clk,                                      // clock
      reset_n,                                  // reset
      sel_n );                                  // test_expr
```

Ensures that `sel_n` is one-cold or inactive (4’b0000) at each rising edge of `clk`.



assert_one_hot

Ensures that the value of a specified expression is one-hot.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_one_hot
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 32.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to a one-hot value on the rising clock edge.

Description

The assert_one_hot assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to a one-hot value. A one-hot value has exactly one bit set to 1.

The checker is useful for verifying control circuits, for example, it can ensure that a finite-state machine with one-hot encoding operates properly and has exactly one bit asserted high. In a datapath circuit the checker can ensure that the enabling conditions for a bus do not result in bus contention.

Assertion Checks

`ASSERT_ONE_HOT`	Expression evaluated to zero or to a value with multiple bits set to 1.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	SANITY — Expression has changed value.
`cover_all_one_hots_checked`	CORNER — Expression evaluated to all possible combinations of one-hot values.

Notes

1. By default, the assert_one_hot assertion is optimistic and the assertion fails if *test_expr* is zero or has multiple bits not set to 0 (i.e.equals 1, X, Z, etc.). However, if ‘OVL_XCHECK_OFF is set, the ASSERT_ONE_HOT assertion fails if and only if *test_expr* is zero or has multiple bits that are 1.

See also

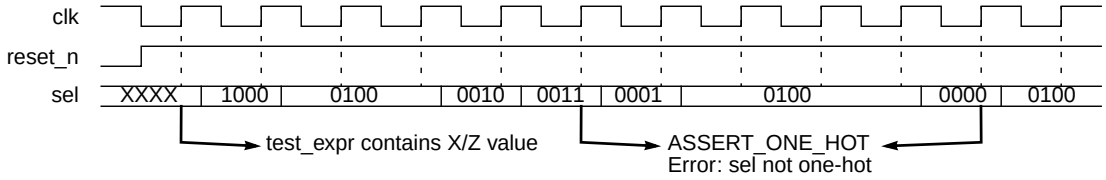
assert_one_cold

assert_zero_one_hot

Example

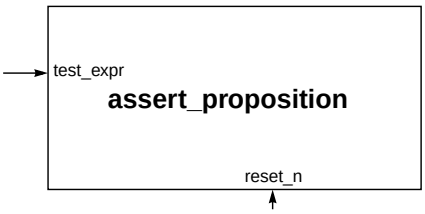
```
assert_one_hot #(
   ‘OVL_ERROR,                                  // severity_level
   4,                                           // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: sel not one-hot”,                    // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sel_one_hot (
      clk,                                      // clock
      reset_n,                                  // reset
      sel );                                    // test_expr
```

Ensures that `sel` is one-hot at each rising edge of `clk`.



assert_proposition

Ensures that the value of a specified expression is always combinationally TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
combinational
assertion

Syntax

```
assert_proposition
      [#(severity_level, property_type, msg, coverage_level )]
      instance_name (reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should always evaluate to TRUE.

Description

The assert_proposition assertion checker checks the single-bit expression *test_expr* when it changes value to verify the expression evaluates to TRUE.

Assertion Checks

ASSERT_PROPOSITION	Expression evaluated to FALSE.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.

Cover Points

none

Notes

1. Formal verification tools and hardware emulation/acceleration systems might ignore this checker. To verify propositional properties with these tools, consider using assert_always.

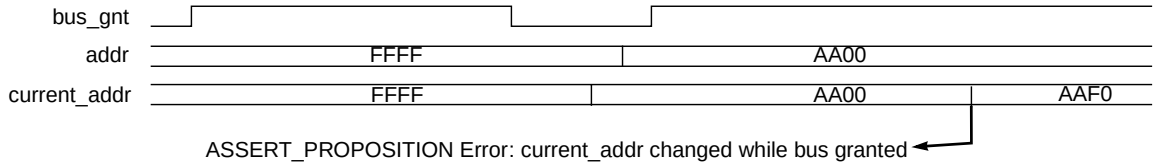
See also

assert_always
assert_always_on_edge
assert_implication
assert_never

Example

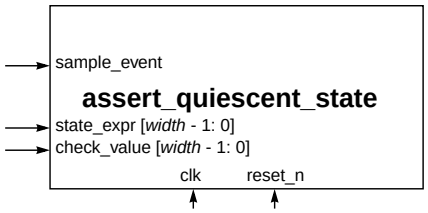
```
assert_proposition #(

   ‘OVL_ERROR,                                  // severity_level
   ‘OVL_ASSERT,                                 // property_type
   “Error: current_addr changed while bus       // msg
   granted”,                                    // coverage_level
   ‘OVL_COVER_ALL)

   valid_current_addr (

      bus_gnt,                                  // reset
      current_addr == addr );                   // test_expr
```

Ensures that `current_addr` equals `addr` while `bus_gnt` is TRUE.



assert_quiescent_state

Ensures that the value of a specified state expression equals a corresponding check value if a specified sample event has transitioned to TRUE.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_quiescent_state
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, state_expr, check_value,
      sample_event );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *state_expr* and *check_value* arguments. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
state_expr[*width*-1:0]	Expression that should have the same value as *check_value* on the rising edge of *clk* if *sample_event* transitioned to TRUE in the previous clock cycle (or is currently transitioning to TRUE).
check_value[*width*-1:0]	Expression that indicates the value *state_expr* should have on the rising edge of *clk* if *sample_event* transitioned to TRUE in the previous clock cycle (or is currently transitioning to TRUE).
sample_event	Expression that initiates the quiescent state check when its value transitions to TRUE.

Description

The assert_quiescent_state assertion checker checks the expression *sample_event* at each rising edge of *clk* to see if its value has transitioned to TRUE (i.e., its current value is TRUE and its value on the previous rising edge of *clk* is not TRUE). If so, the checker verifies that the current value of *state_expr* equals the current value of *check_value*. The assertion fails if *state_expr* is not equal to *check_value*.

The *state_expr* and *check_value* expressions are verification events that can change. In particular, the same assertion checker can be coded to compare different check values (if they are checked in different cycles).

The checker is useful for verifying the states of state machines when transactions complete.

Assertion Checks

`ASSERT_QUIESCENT_STATE`	The *sample_event* expression transitioned to TRUE, but the values of *state_expr* and *check_value* were not the same.

Implicit X/Z Checks

`state_expr contains X or Z`	State expression value contained X or Z bits.
`check_value contains X or Z`	Check vale expression value contained X or Z bits.
`sample_event contains X or Z`	Sample event value was X or Z.
`‘OVL_END_OF_SIMULATION contains X or Z`	State expression value contained X or Z bits at the end of simulation (‘OVL_END_OF_SIMULATION asserted).

Cover Points

none

Notes

1. The assertion check compares the current value of *sample_event* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.
2. The checker recognizes the Verilog macro ‘OVL_END_OF_SIMULATION=*eos_signal*. If set, the quiescent state check is also performed at the end of simulation, when *eos_signal* asserts (regardless of the value of *sample_event*).
3. Formal verification tools and hardware emulation/acceleration systems might ignore this checker.

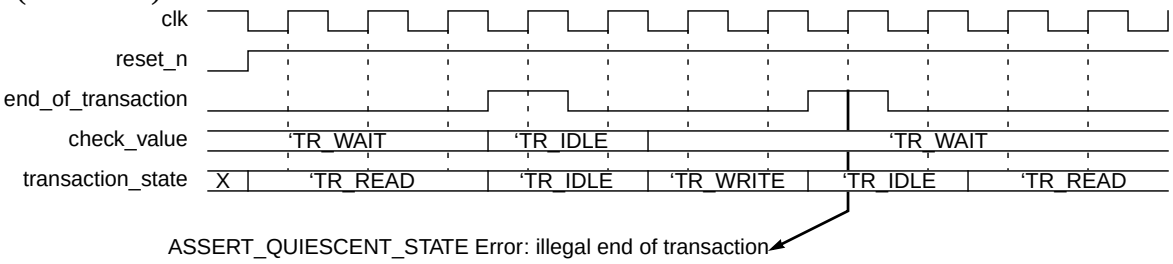
See also

assert_no_transition assert_transition

Example

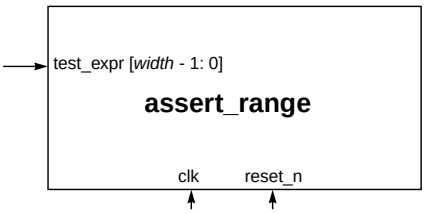
```
assert_quiescent_state #(

   ‘OVL_ERROR,                                    // severity_level
   4,                                             // width
   ‘OVL_ASSERT,                                   // property_type
   “Error: illegal end of transaction”,           // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   valid_end_of_transaction_state (

      clk,                                        // clock
      reset_n,                                    // reset
      transaction_state,                          // state_expr
      prev_tr == ‘TR_READ ? ‘TR_IDLE :            // check_value
      ‘TR_WAIT                                    // sample_event
      end_of_transaction);
```

Ensures that whenever `end_of_transaction` asserts at the completion of each transaction, the value of `transaction_state` is ‘TR_IDLE (if prev_tr is ‘TR_READ) or ‘TR_WAIT (otherwise).



assert_range

Ensures that the value of a specified expression is in a specified range.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_range
      [#(severity_level, width, min, max, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
min	Minimum value allowed for *test_expr*. Default: 0.
max	Maximum value allowed for *test_expr*. Default: 2***width* - 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to a value in the range from *min* to *max* (inclusive) on the rising clock edge.

Description

The assert_range assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression falls in the range from *min* to *max*, inclusive. The assertion fails if *test_expr* < *min* or *max* < *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) are within their proper ranges. The checker is also useful for ensuring datapath variables and expressions are in legal ranges.

Assertion Checks

`ASSERT_RANGE`	Expression evaluated outside the range *min* to *max*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_cover_test_expr_change`	BASIC — Expression changed value.
`cover_test_expr_at_min`	CORNER — Expression evaluated to *min*.
`cover_test_expr_at_max`	CORNER — Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle.

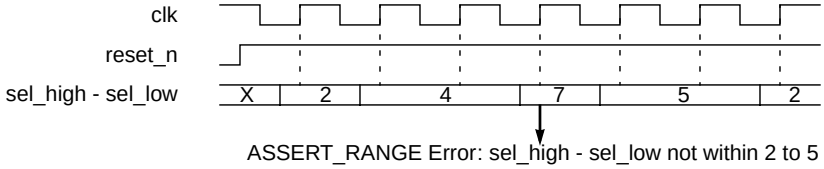
See also

assert_always
assert_implication
assert_never
assert_proposition

Example

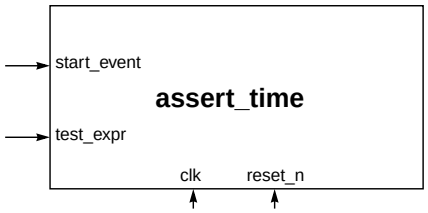
```
assert_range #(

   ‘OVL_ERROR,                                          // severity_level
   3,                                                   // width
   2,                                                   // min
   5,                                                   // max
   ‘OVL_ASSERT,                                         // property_type
   “Error: sel_high - sel_low not within 2 to 5”,       // msg
   ‘OVL_COVER_ALL)                                      // coverage_level

   valid_sel (

      clk,                                              // clock
      reset_n,                                          // reset
      sel_high - sel_low );                             // test_expr
```

Ensures that (`sel_high - sel_low`) is in the range 2 to 5 at each rising edge of `clk`.



assert_time

Ensures that the value of a specified expression remains TRUE for a specified number of cycles after a start event.



Parameters:
severity_level
num_cks
action_on_new_start
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_time
      [#(severity_level, num_cks, action_on_new_start, property_type,
      msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr );
```

Parameters

`severity_level`	Severity of the failure. Default: ‘OVL_ERROR.
`num_cks`	Number of cycles after *start_event* is TRUE that *test_expr* must be held TRUE. Default: 1.
`action_on_new_start`	Method for handling a new start event that occurs while a check is pending. Values are: ‘OVL_IGNORE_NEW_START, ‘OVL_RESET_ON_NEW_START and ‘OVL_ERROR_ON_NEW_START. Default: ‘OVL_IGNORE_NEW_START.
`property_type`	Property type. Default: ‘OVL_ASSERT.
`msg`	Error message printed when assertion fails. Default: “VIOLATION”.
`coverage_level`	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

`clk`	Clock event for the assertion. The checker samples on the rising edge of the clock.
`reset_n`	Active low synchronous reset signal indicating completed initialization.
`start_event`	Expression that (along with *num_cks*) identifies when to check *test_expr*.

`test_expr`	Expression that should evaluate to TRUE for *num_cks* cycles after *start_event* initiates a check.

Description

The assert_time assertion checker checks the expression *start_event* at each rising edge of *clk* to determine whether or not to initiate a check. Once initiated, the check evaluates *test_expr* each rising edge of *clk* for *num_cks* cycles to verify that its value is TRUE. During that time, the assertion fails each cycle a sampled value of *test_expr* is not TRUE.

The method used to determine what constitutes a start event for initiating a check is controlled by the *action_on_new_start* parameter. If no check is in progress when *start_event* is sampled TRUE, a new check is initiated. But, if a check is in progress when *start_event* is sampled TRUE, the checker has the following actions:

- ‘OVL_IGNORE_NEW_START

  The checker does not sample *start_event* for the next *num_cks* cycles after a start event.

- ‘OVL_RESET_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the checker terminates the check and initiates a new check without sampling *test_expr*.

- ‘OVL_ERROR_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the assertion fails with an illegal start event violation. In this case, the checker does not initiate a new check, does not terminate a pending check and reports an additional assertion violation if *test_expr* is FALSE.

Assertion Checks

`ASSERT_TIME`	The value of *test_expr* was not TRUE within *num_cks* cycles after *start_event* was sampled TRUE.
`illegal start event`	The *action_on_new_start* parameter is set to ‘OVL_ERROR_ON_NEW_START and *start_event* expression evaluated to TRUE while the checker was monitoring *test_expr*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.
`start_event contains X or Z`	Start event value was X or Z.

Cover Points

`cover_window_open`	BASIC — A time check was initiated.
`cover_window_close`	BASIC — A time check lasted the full *num_cks* cycles.
`cover_window_resets`	CORNER — The *action_on_new_start* parameter is ‘OVL_RESET_ON_NEW_START, and *start_event* was sampled TRUE while the checker was monitoring *test_expr*.

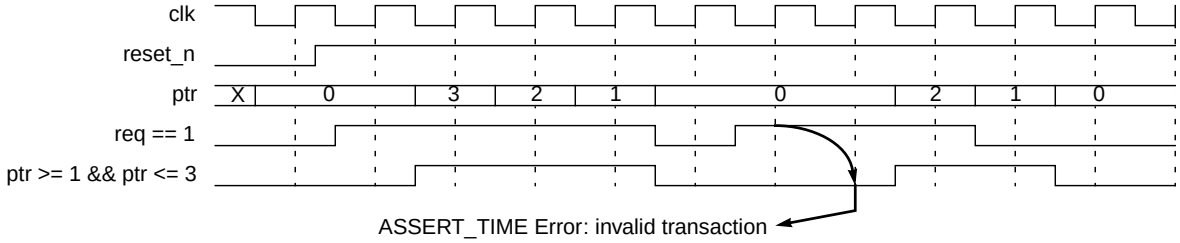
See also

assert_change
assert_next
assert_frame
assert_unchange
assert_win_change
assert_win_unchange
assert_window

Examples

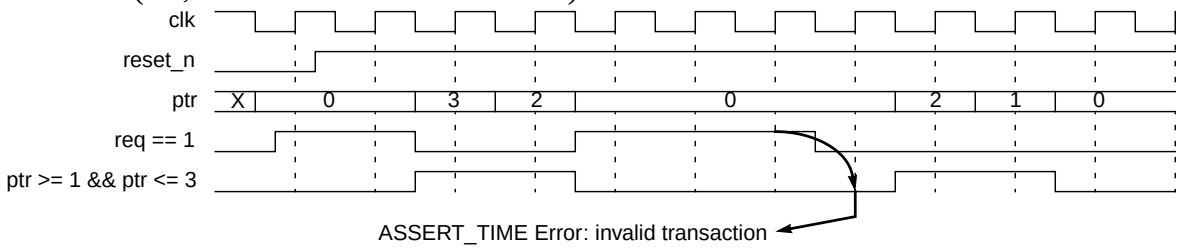
```
assert_time #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_IGNORE_NEW_START,                       // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req == 1,                                 // start_event
      ptr >= 1 && ptr <= 3);                    // test_expr
```

Ensures that `ptr` is sampled in the range 1 to 3 for three cycles after `req` is sampled equal to 1 at the rising edge of `clk`. If `req` is sampled equal to 1 when the checker samples `ptr`, a new check is not initiated (i.e., the new start is ignored).



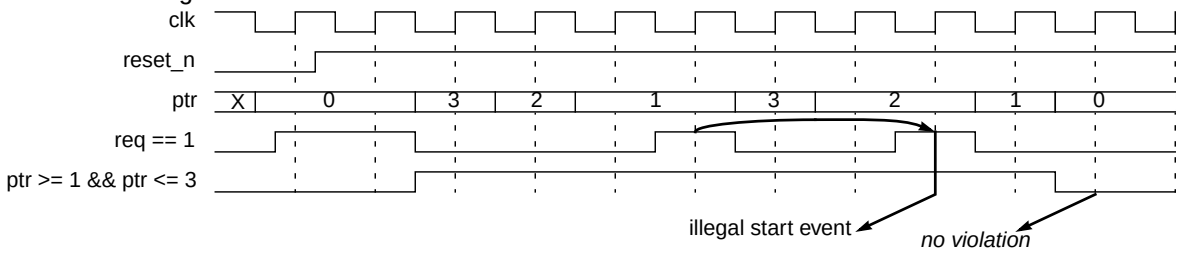
```
assert_time #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_RESET_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req == 1,                                 // start_event
      ptr >= 1 && ptr <= 3);                    // test_expr
```

Ensures that `ptr` is sampled in the range 1 to 3 for three cycles after `req` is sampled equal to 1 at the rising edge of `clk`. If `req` is sampled equal to 1 when the checker samples `ptr`, a new check is initiated (i.e., the new start restarts a check).



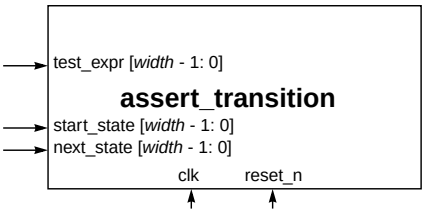

```
assert_time #(

   ‘OVL_ERROR,                                  // severity_level
   3,                                           // num_cks
   ‘OVL_ERROR_ON_NEW_START,                     // action_on_new_start
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid transaction”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_transaction (

      clk,                                      // clock
      reset_n,                                  // reset
      req == 1,                                 // start_event
      ptr >= 1 && ptr <= 3);                    // test_expr
```

Ensures that `ptr` is sampled in the range 1 to 3 for three cycles after `req` is sampled equal to 1 at the rising edge of `clk`. If `req` is sampled equal to 1 when the checker samples `ptr`, the checker issues an `illegal start event` violation and does not start a new check.



assert_transition

Ensures that the value of a specified expression transitions properly from a start state to the specified next state.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_transition
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr, start_state, next_state );
```

Parameters

`severity_level`	Severity of the failure. Default: ‘OVL_ERROR.
`width`	Width of the *test_expr* argument. Default: 1.
`property_type`	Property type. Default: ‘OVL_ASSERT.
`msg`	Error message printed when assertion fails. Default: “VIOLATION”.
`coverage_level`	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

`clk`	Clock event for the assertion. The checker samples on the rising edge of the clock.
`reset_n`	Active low synchronous reset signal indicating completed initialization.
`test_expr[width-1:0]`	Expression that should transition to *next_state* on the rising edge of *clk* if its value at the previous rising edge of *clk* is the same as the current value of *start_state*.
`start_state[width-1:0]`	Expression that indicates the start state for the assertion check. If the start state matches the value of *test_expr* on the previous rising edge of *clk*, the check is performed.

next_state[*width*-1:0]	Expression that indicates the only valid next state for the assertion check. If the value of *test_expr* was *start_state* at the previous rising edge of *clk*, then the value of *test_expr* should equal *next_state* on the current rising edge of *clk*.

Description

The assert_transition assertion checker checks the expression *test_expr* and *start_state* at each rising edge of *clk* to see if they are the same. If so, the checker evaluates and stores the current value of *next_state*. At the next rising edge of *clk*, the checker re-evaluates *test_expr* to see if its value equals the stored value of *next_state*. If not, the assertion fails. The checker returns to checking *start_state* in the current cycle (unless a fatal failure occurred)

The *start_state* and *next_state* expressions are verification events that can change. In particular, the same assertion checker can be coded to verify multiple types of transitions of *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) transition properly.

Assertion Checks

ASSERT_TRANSITION	Expression transitioned from *start_state* to a value different from *next_state*.

Implicit X/Z Checks

test_expr contains X or Z	Expression value contained X or Z bits.
start_state contains X or Z	Start state value contained X or Z bits.
next_state contains X or Z	Next state value contained X or Z bits.

Cover Points

cover_start_state	BASIC — Expression assumed a start state value.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.

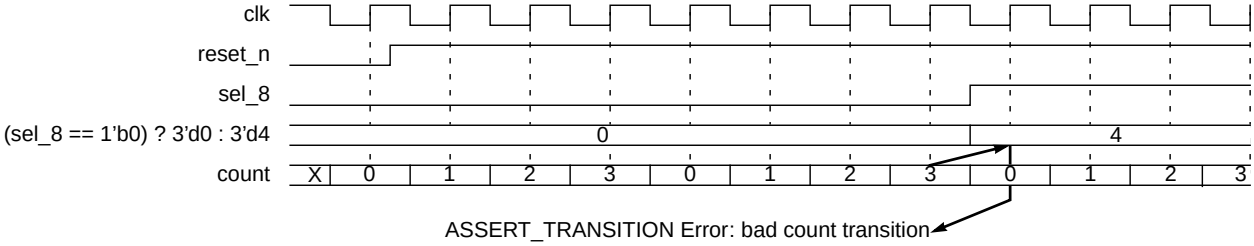
See also

assert_no_transition

Example

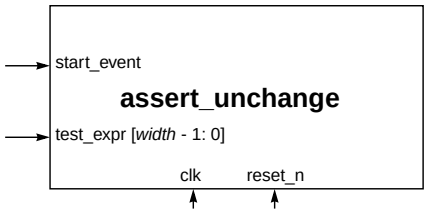
```
assert_transition #(

   ‘OVL_ERROR,                                    // severity_level
   3,                                             // width
   ‘OVL_ASSERT,                                   // property_type
   “Error: bad count transition”,                 // msg
   ‘OVL_COVER_ALL)                                // coverage_level

   valid_count (

      clk,                                        // clock
      reset_n,                                    // reset
      count,                                      // test_expr
      3’d3,                                       // start_state
      (sel_8 == 1’b0) ? 3’d0 : 3’d4  );           // next_state
```

Ensures that `count` transitions from 3’d3 properly. If `sel_8` is 0, `count` should have transitioned to 3’d0. Otherwise, `count` should have transitioned to 3’d4.



assert_unchange

Ensures that the value of a specified expression does not change for a specified number of cycles after a start event initiates checking.



Parameters:
severity_level
width
num_cks
action_on_new_start
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_unchange
      [#(severity_level, width, num_cks, action_on_new_start,
      property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr );
```

Parameters

`severity_level`	Severity of the failure. Default: ‘OVL_ERROR.
`width`	Width of the *test_expr* argument. Default: 1.
`num_cks`	Number of cycles *test_expr* should remain unchanged after a start event. Default: 1.
`action_on_new_start`	Method for handling a new start event that occurs before *num_cks* clock cycles transpire without a change in the value of *test_expr*. Values are: ‘OVL_IGNORE_NEW_START, ‘OVL_RESET_ON_NEW_START and ‘OVL_ERROR_ON_NEW_START. Default: ‘OVL_IGNORE_NEW_START.
`property_type`	Property type. Default: ‘OVL_ASSERT.
`msg`	Error message printed when assertion fails. Default: “VIOLATION”.
`coverage_level`	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

`clk`	Clock event for the assertion. The checker samples on the rising edge of the clock.
`reset_n`	Active low synchronous reset signal indicating completed initialization.

start_event	Expression that (along with *action_on_new_start*) identifies when to start checking *test_expr*.
test_expr[*width*-1:0]	Expression that should not change value for *num_cks* cycles from the start event unless the check is interrupted by a valid new start event.

Description

The assert_unchange assertion checker checks the expression *start_event* at each rising edge of *clk* to determine if it should check for a change in the value of *test_expr*. If *start_event* is sampled TRUE, the checker evaluates *test_expr* and re-evaluates *test_expr* at each of the subsequent *num_cks* rising edges of *clk*. Each time the checker re-evaluates *test_expr*, if its value has changed from its value in the previous cycle, the assertion fails.

The method used to determine how to handle a new start event, when the checker is in the state of checking for a change in *test_expr*, is controlled by the *action_on_new_start* parameter. The checker has the following actions:

- ‘OVL_IGNORE_NEW_START

  The checker does not sample *start_event* for the next *num_cks* cycles after a start event.

- ‘OVL_RESET_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the checker terminates the check and initiates a new check.

- ‘OVL_ERROR_ON_NEW_START

  The checker samples *start_event* every cycle. If a check is pending and the value of *start_event* is TRUE, the assertion fails with an illegal start event violation. In this case, the checker does not initiate a new check and does not terminate a pending check.

The checker is useful for ensuring proper changes in structures after various events. For example, it can be used to check that multiple-cycle operations with enabling conditions function properly with the same data. It can be used to check that single-cycle operations function correctly with data loaded at different cycles. It also can be used to verify synchronizing conditions that require date to be stable after an initial triggering event.

Assertion Checks

ASSERT_UNCHANGE	The *test_expr* expression changed value within *num_cks* cycles after *start_event* was sampled TRUE.
illegal start event	The *action_on_new_start* parameter is set to ‘OVL_ERROR_ON_NEW_START and *start_event* expression evaluated to TRUE while the checker was in the state of checking for a change in the value of *test_expr*.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.
`start_event contains X or Z`	Start event value was X or Z.

Cover Points

`cover_window_open`	BASIC — A change check was initiated.
`cover_window_close`	BASIC — A change check lasted the full *num_cks* cycles.
`cover_window_resets`	CORNER — The *action_on_new_start* parameter is ‘OVL_RESET_ON_NEW_START, and *start_event* was sampled TRUE while the checker was monitoring *test_expr* without detecting a changed value.

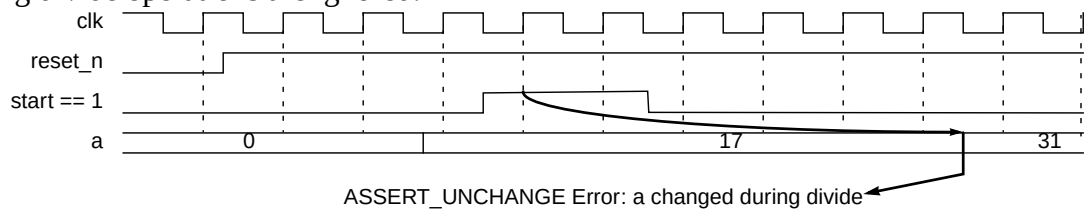
See also

assert_change
assert_time
assert_win_change
assert_win_unchange
assert_window

Examples

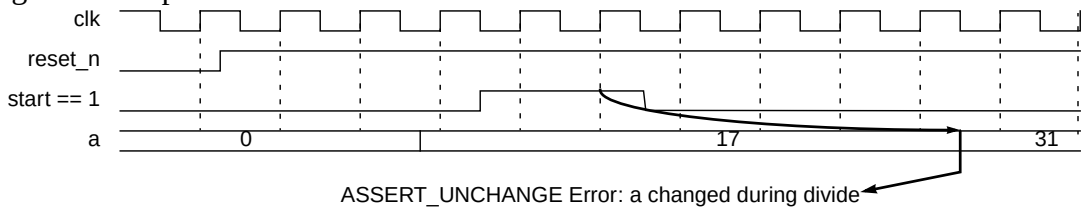
```
assert_unchange #(
  ‘OVL_ERROR,                                  // severity_level
  8,                                           // width
  8,                                           // num_cks
  ‘OVL_IGNORE_NEW_START,                       // action_on_new_start
  ‘OVL_ASSERT,                                 // property_type
  “Error: a changed during divide”,            // msg
  ‘OVL_COVER_ALL)                              // coverage_level

  valid_div_unchange_a (

     clk,                                      // clock
     reset_n,                                  // reset
     start == 1,                               // start_event
     a);                                       // test_expr
```

Ensures that a remains unchanged while a divide operation is performed (8 cycles). Restarts during divide operations are ignored.



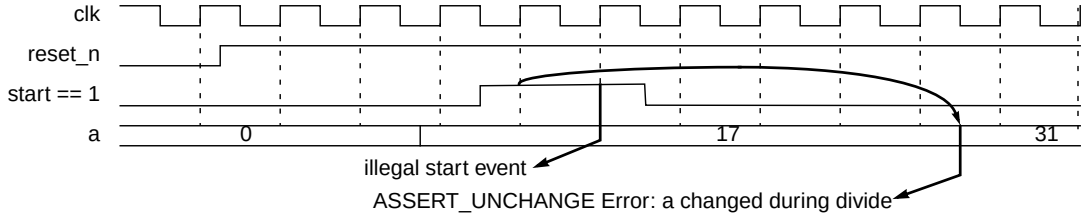
```
assert_unchange #(

  ‘OVL_ERROR,                                  // severity_level
  8,                                           // width
  8,                                           // num_cks
  ‘OVL_RESET_ON_NEW_START,                     // action_on_new_start
  ‘OVL_ASSERT,                                 // property_type
  “Error: a changed during divide”,            // msg
  ‘OVL_COVER_ALL)                              // coverage_level

  valid_div_unchange_a (

     clk,                                      // clock
     reset_n,                                  // reset
     start == 1,                               // start_event
     a);                                       // test_expr
```

Ensures that a remains unchanged while a divide operation is performed (8 cycles). A restart during a divide operation starts the check over.



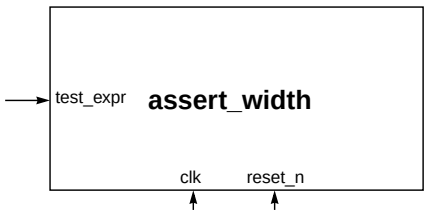

```
assert_unchange #(

  ‘OVL_ERROR,                                  // severity_level
  8,                                           // width
  8,                                           // num_cks
  ‘OVL_ERROR_ON_NEW_START,                     // action_on_new_start
  ‘OVL_ASSERT,                                 // property_type
  “Error: a changed during divide”,            // msg
  ‘OVL_COVER_ALL)                              // coverage_level

  valid_div_unchange_a (

     clk,                                      // clock
     reset_n,                                  // reset
     start == 1,                               // start_event
     a);                                       // test_expr
```

Ensures that a remains unchanged while a divide operation is performed (8 cycles). A restart during a divide operation is a violation.



assert_width

Ensures that when value of a specified expression is TRUE, it remains TRUE for a minimum number of clock cycles and transitions from TRUE no later than a maximum number of clock cycles.



Parameters:
severity_level
min_cks
max_cks
property_type
msg
coverage_level

Class:
n-cycle assertion

Syntax

```
assert_width
      [#(severity_level, min_cks, max_cks, property_type, msg,
      coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
min_cks	Minimum number of clock edges *test_expr* must remain TRUE once it is sampled TRUE. The special case where *min_cks* is 0 turns off minimum checking (i.e., *test_expr* can transition from TRUE in the next clock cycle). Default: 1 (i.e., same as 0).
max_cks	Maximum number of clock edges *test_expr* can remain TRUE once it is sampled TRUE. The special case where *max_cks* is 0 turns off maximum checking (i.e., *test_expr* can remain TRUE for any number of cycles). Default: 1 (i.e., *test_expr* must transition from TRUE in the next clock cycle).
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
test_expr	Expression that should evaluate to TRUE for at least *min_cks* cycles and at most *max_cks* cycles after it is sampled TRUE.

Description

The assert_width assertion checker checks the single-bit expression *test_expr* at each rising edge of *clk*. If the value of *test_expr* is TRUE, the checker performs the following steps:

1. Unless it is disabled by setting *min_cks* to 0, a minimum check is initiated. The check evaluates *test_expr* at each subsequent rising edge of *clk*. If its value is not TRUE, the minimum check fails. Otherwise, after *min_cks* -1 cycles transpire, the minimum check terminates.
2. Unless it is disabled by setting *max_cks* to 0, a maximum check is initiated. The check evaluates *test_expr* at each subsequent rising edge of *clk*. If its value does not transition from TRUE by the time *max_cks* cycles transpire (from the start of checking), the maximum check fails.
3. The checker returns to checking *test_expr* in the next cycle. In particular if *test_expr* is TRUE, a new set of checks is initiated.

Assertion Checks

`MIN_CHECK`	The value of *test_expr* was held TRUE for less than *min_cks* cycles.
`MAX_CHECK`	The value of *test_expr* was held TRUE for more than *max_cks* cycles.
`min_cks > max_cks`	The *min_cks* parameter is greater than the *max_cks* parameter (and *max_cks* >0). Unless the violation is fatal, either the minimum or maximum check will fail.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.

Cover Points

`cover_test_expr_asserts`	BASIC — A check was initiated (i.e., *test_expr* was sampled TRUE).
`cover_test_expr_asserted_for_min_cks`	CORNER — The expression *test_expr* was held TRUE for exactly *min_cks* cycles (*min_cks* > 0).
`cover_test_expr_asserted_for_max_cks`	CORNER — The expression *test_expr* was held TRUE for exactly *max_cks* cycles (*max_cks* > 0).

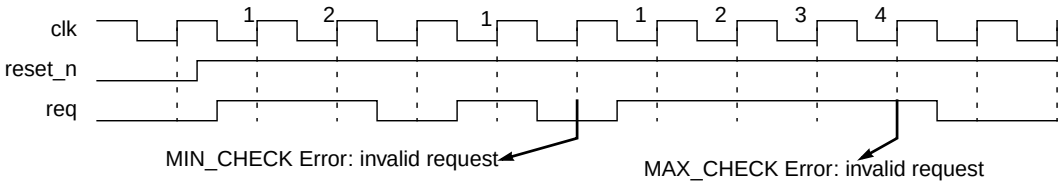
See also

assert_change
assert_time
assert_unchange

Example

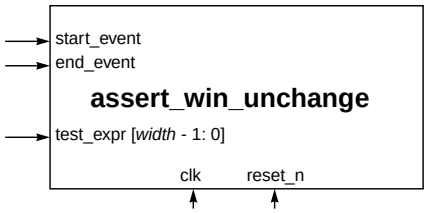
```
assert_width #(

   ‘OVL_ERROR,                                  // severity_level
   2,                                           // min_cks
   3,                                           // max_cks
   ‘OVL_ASSERT,                                 // property_type
   “Error: invalid request”,                    // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_request (

      clk,                                      // clock
      reset_n,                                  // reset
      req == 1);                                // test_expr
```

Ensures `req` asserts for 2 or 3 cycles.



assert_win_change

Ensures that the value of a specified expression changes in a specified window between a start event and an end event.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
event-bounded assertion

Syntax

```
assert_win_change
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr, end_event );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
start_event	Expression that opens an event window.
test_expr[*width*-1:0]	Expression that should change value in the event window
end_event	Expression that closes an event window.

Description

The assert_win_change assertion checker checks the expression *start_event* at each rising edge of *clk* to determine if it should open an event window at the start of the next cycle. If *start_event* is sampled TRUE, the checker evaluates *test_expr*. At each subsequent rising edge of *clk*, the checker evaluates *end_event* and re-evaluates *test_expr*. If *end_event* is TRUE, the checker closes the event window and if all sampled values of *test_expr* equal its value at the start of the window, then the assertion fails. The checker returns to the state of monitoring *start_event* at the next rising edge of *clk* after the event window is closed.

The checker is useful for ensuring proper changes in structures in various event windows. A typical use is to verify that synchronization logic responds after a stimulus (for example, bus transactions occurs without interrupts or write commands are not issued during read cycles). Another typical use is verifying a finite-state machine responds correctly in event windows.

Assertion Checks

`ASSERT_WIN_CHANGE`	The *test_expr* expression did not change value during an open event window.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.
`start_event contains X or Z`	Start event value was X or Z.
`end_event contains X or Z`	End event value was X or Z.

Cover Points

`cover_window_open`	BASIC — An event window opened (*start_event* was TRUE).
`cover_window_close`	BASIC — An event window closed (*end_event* was TRUE in an open event window).

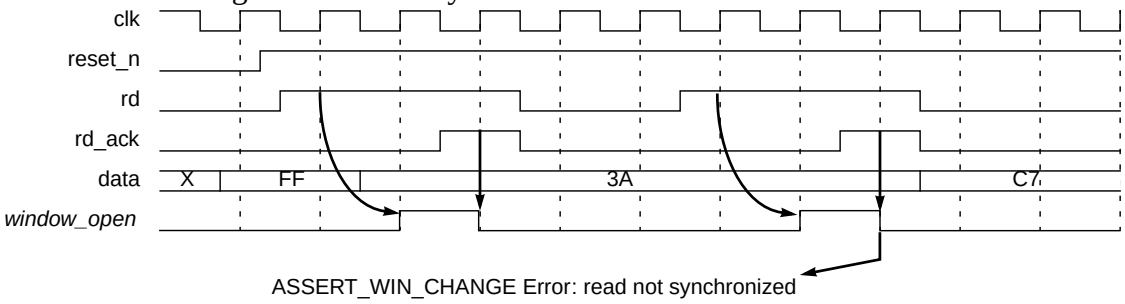
See also

assert_change
assert_time
assert_unchange
assert_win_unchange
assert_window

Example

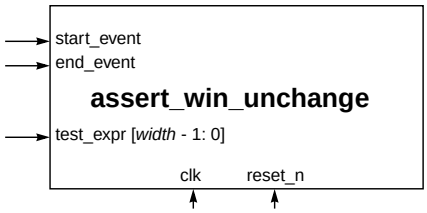
```
assert_win_change #(

   ‘OVL_ERROR,                                  // severity_level
   32,                                          // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: read not synchronized”,              // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sync_data_bus_rd (

      clk,                                      // clock
      reset_n,                                  // reset
      rd,                                       // start_event
      data,                                     // test_expr
      rd_ack );                                 // end_event
```

Ensures that `data` changes value in every data read window.



assert_win_unchange

Ensures that the value of a specified expression does not change in a specified window between a start event and an end event.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
event-bounded assertion

Syntax

```
assert_win_unchange
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr, end_event );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
start_event	Expression that opens an event window.
test_expr[*width*-1:0]	Expression that should not change value in the event window
end_event	Expression that closes an event window.

Description

The assert_win_unchange assertion checker checks the expression *start_event* at each rising edge of *clk* to determine if it should open an event window at the start of the next cycle. If *start_event* is sampled TRUE, the checker evaluates *test_expr*. At each subsequent rising edge of *clk*, the checker evaluates end_event and re-evaluates *test_expr*. If a sampled value of *test_expr* is changed from its value in the previous cycle, then the assertion fails. If *end_event* is TRUE, the checker closes the event window and returns to the state of monitoring *start_event* at the next rising edge of *clk*.

The checker is useful for ensuring certain variables and expressions do not change in various event windows. A typical use is to verify that synchronization logic responds after a stimulus (for example, bus transactions occurs without interrupts or write commands are not issued during read cycles). Another typical use is to verify that non-deterministic multiple-cycle operations with enabling conditions function properly with the same data.

Assertion Checks

`ASSERT_WIN_UNCHANGE`	The *test_expr* expression changed value during an open event window.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.
`start_event contains X or Z`	Start event value was X or Z.
`end_event contains X or Z`	End event value was X or Z.

Cover Points

`cover_window_open`	BASIC — An event window opened (*start_event* was TRUE).
`cover_window_close`	BASIC — An event window closed (*end_event* was TRUE in an open event window).

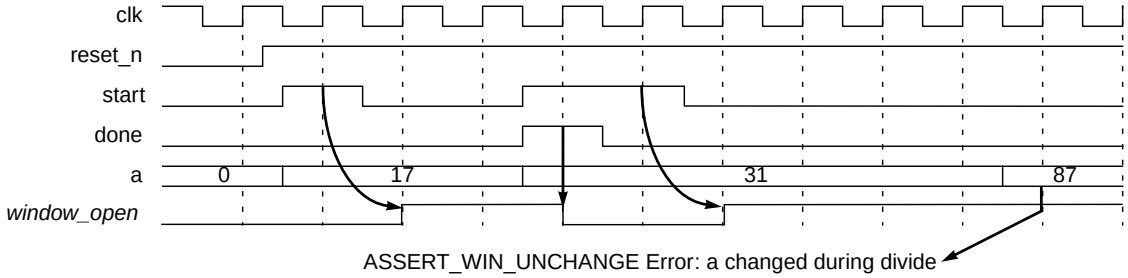
See also

assert_change
assert_time
assert_unchange
assert_win_change
assert_window

Example

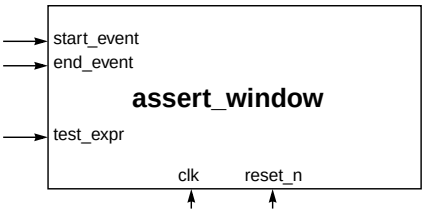
```
assert_win_unchange #(

   ‘OVL_ERROR,                                  // severity_level
   8,                                           // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: a changed during divide”,            // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_div_win_unchange_a (

      clk,                                      // clock
      reset_n,                                  // reset
      start,                                    // start_event
      a,                                        // test_expr
      done);                                    // end_event
```

Ensures that the a input to the divider remains unchanged while a divide operation is performed (i.e., in the window from `start` to `done`).



assert_window

Ensures that the value of a specified expression is TRUE in a specified window between a start event and an end event.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
event-bounded assertion

Syntax

```
assert_window
      [#(severity_level, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, start_event, test_expr, end_event );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
start_event	Expression that opens an event window.
test_expr	Expression that should be TRUE in the event window
end_event	Expression that closes an event window.

Description

The assert_window assertion checker checks the expression *start_event* at each rising edge of *clk* to determine if it should open an event window at the start of the next cycle. If *start_event* is sampled TRUE, at each subsequent rising edge of *clk*, the checker evaluates end_event and *test_expr*. If a sampled value of *test_expr* is not TRUE, then the assertion fails. If *end_event* is TRUE, the checker closes the event window and returns to the state of monitoring *start_event* at the next rising edge of *clk*.

The checker is useful for ensuring proper changes in structures after various events. For example, it can be used to check that multiple-cycle operations with enabling conditions function properly with the same data. It can be used to check that single-cycle operations function correctly with data loaded at different cycles. It also can be used to verify synchronizing conditions that require date to be stable after an initial triggering event.

Assertion Checks

`ASSERT_WINDOW`	The *test_expr* expression changed value during an open event window.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value was X or Z.
`start_event contains X or Z`	Start event value was X or Z.
`end_event contains X or Z`	End event value was X or Z.

Cover Points

`cover_window_open`	BASIC — A change check was initiated.
`cover_window_close`	BASIC — A change check lasted the full *num_cks* cycles.

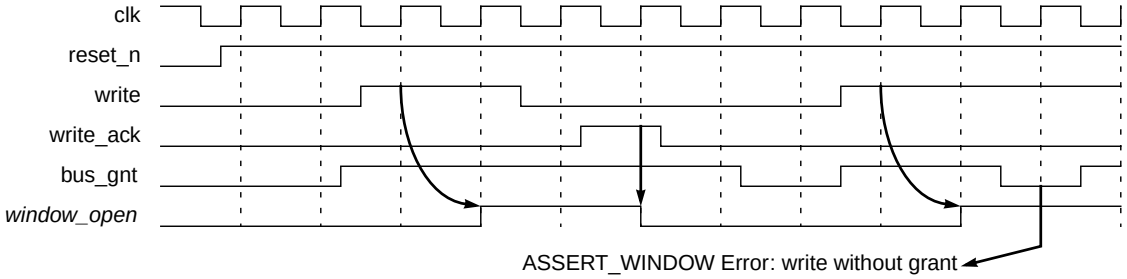
See also

assert_change
assert_time
assert_unchange
assert_win_change
assert_win_unchange

Example

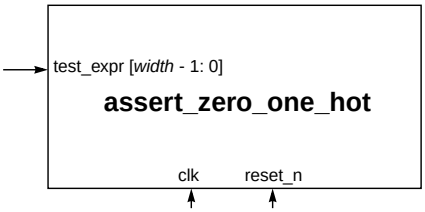
```
assert_window #(

   ‘OVL_ERROR,                                  // severity_level
   ‘OVL_ASSERT,                                 // property_type
   “Error: write without grant”,                // msg
   ‘OVL_COVER_ALL)                              // coverage_level

   valid_sync_data_bus_write (

      clk,                                      // clock
      reset_n,                                  // reset
      write,                                    // start_event
      bus_gnt,                                  // test_expr
      write_ack );                              // end_event
```

Ensures that the bus grant is not deasserted during a write cycle.



assert_zero_one_hot

Ensures that the value of a specified expression is zero or one-hot.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_zero_one_hot
      [#(severity_level, width, property_type, msg, coverage_level )]
      instance_name (clk, reset_n, test_expr );
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 32.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker samples on the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr[*width*-1:0]	Expression that should evaluate to either 0 or a one-hot value on the rising clock edge.

Description

The assert_zero_one_hot assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression evaluates to a one-hot value or is zero. A one-hot value has exactly one bit set to 1.

The checker is useful for verifying control circuits, circuit enabling logic and arbitration logic. For example, it can ensure that a finite-state machine with zero-one-cold encoding operates properly and has exactly one bit asserted high—or else is zero. In a datapath circuit the checker can ensure that the enabling conditions for a bus do not result in bus contention.

Assertion Checks

`ASSERT_ZERO_ONE_HOT`	Expression evaluated to a value with multiple bits set to 1.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression value contained X or Z bits.

Cover Points

`cover_test_expr_change`	SANITY — Expression has changed value.
`cover_all_one_hots_checked`	CORNER — Expression evaluated to all possible combinations of one-hot values.
`cover_test_expr_all_zeros`	CORNER — Expression evaluated to 0.

Notes

1. By default, the assert_zero_one_hot assertion is optimistic and the assertion fails if *test_expr* has multiple bits not set to 0 (i.e.equals 1, X, Z, etc.). However, if ‘OVL_XCHECK_OFF is set, the assertion fails if and only if *test_expr* has multiple bits that are 1.

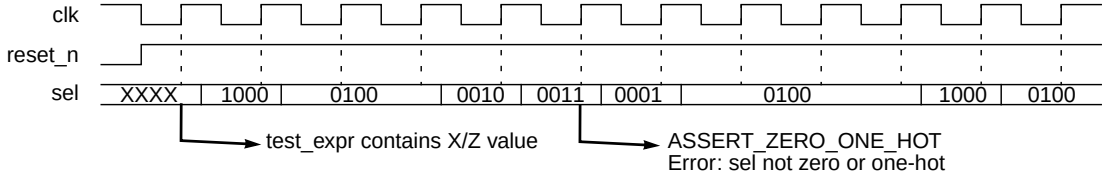
See also

assert_one_cold assert_one_hot

Example

```
assert_zero_one_hot #(

   `OVL_ERROR,                                    // severity_level
   4,                                             // width
   `OVL_ASSERT,                                   // property_type
   "Error: sel not zero or one-hot",              // msg
   `OVL_COVER_ALL)                                // coverage_level

   valid_sel_zero_one_hot (

      clk,                                        // clock
      reset_n,                                    // reset
      sel );                                      // test_expr
```

Ensures that `sel` is zero or one-hot at each rising edge of `clk`.



Chapter 4
OVL Defines

Global Defines

Type	DEFINE	Description
Language	`‘OVL_VERILOG`	(default) Creates assertion checkers defined in Verilog.
	`‘OVL_SVA`	Creates assertion checkers defined in System Verilog.
	`‘OVL_SVA_INTERFACE`	Ensures OVL assertion checkers can be instantiated in an SVA interface construct. Default: not defined.
	`‘OVL_PSL`	Creates assertion checkers defined in PSL. Default: not defined.
Synthesizable Logic	`‘OVL_SYNTHESIS_OFF`	Ensures OVL logic is synthesizable. Default: not defined.
Function	`‘OVL_ASSERT_ON`	Activates assertion logic. Default: not defined.
	`‘OVL_COVER_ON`	Activates coverage logic. Default: not defined.
Reset	`‘OVL_GLOBAL_RESET=`*`reset_signal`*	Overrides the *reset_n* port assignments of all assertion checkers with the specified global reset signal. Default: each checker’s reset is specified by the *reset_n* port.
Reporting	`‘OVL_MAX_REPORT_ERROR`	Discontinues reporting a checker’s assertion violations if the number of times the checker has reported one or more violations reaches this limit. Default: unlimited reporting.
	`‘OVL_MAX_REPORT_COVER_POINT`	Discontinues reporting a checker’s cover points if the number of times the checker has reported one or more cover points reaches this limit.Default: unlimited reporting.

Type	DEFINE	Description
	`‘OVL_INIT_MSG`	Reports configuration information for each checker when it is instantiated at the start of simulation. Default: no initialization messages reported.
	`‘OVL_END_OF_SIMULATION=`*`eos_signal`*	Performs quiescent state checking at end of simulation when the *eos_signal* asserts. Default: not defined.
Fatal Error Runtime	`‘OVL_RUNTIME_AFTER_FATAL`	Number of time units from a fatal error to end of simulation. Default: 100.
X/Z Values	`‘OVL_IMPLICIT_XCHECK_OFF`	Turns off implicit X/Z checks. Default: not defined.
	`‘OVL_XCHECK_OFF`	Turns off all X/Z checks. Default: not defined.

Internal Global Defines

The following global variables are for internal use and the user should not redefine them:

```
‘endmodule
‘module
‘OVL_RESET_SIGNAL
‘OVL_SHARED_CODE
‘OVL_STD_DEFINES_H
‘OVL_VERSION
```

Defines Common to All Assertions

Parameter	DEFINE	Description
severity_level	`‘OVL_FATAL`	Runtime fatal error.
	`‘OVL_ERROR`	(default) Runtime error.
	`‘OVL_WARNING`	Runtime Warning.
	`‘OVL_INFO`	Assertion failure has no specific severity.
property_type	`‘OVL_ASSERT`	(default) All the assertion checker’s checks are asserts.
	`‘OVL_ASSUME`	All the assertion checker’s checks are assumes.
	`‘OVL_IGNORE`	All the assertion checker’s checks are ignored.
coverage_level	`‘OVL_COVER_ALL`	(default) Includes coverage logic for all of the checker’s cover points if ‘OVL_COVER_ON is defined.
	`‘OVL_COVER_NONE`	Excludes coverage logic for all of the checker’s cover points.
	`‘OVL_COVER_SANITY`	Includes coverage logic for the checker’s SANITY cover points if ‘OVL_COVER_ON is defined. Can be bitwise-ORed with ‘OVL_COVER_BASIC and ‘OVL_COVER_CORNER.
	`‘OVL_COVER_BASIC`	Includes coverage logic for the checker’s BASIC cover points if ‘OVL_COVER_ON is defined. Can be bitwise-ORed with ‘OVL_COVER_SANITY and ‘OVL_COVER_CORNER.
	`‘OVL_COVER_CORNER`	Includes coverage logic for the checker’s CORNER cover points if ‘OVL_COVER_ON is defined. Can be bitwise-ORed with ‘OVL_COVER_SANITY and ‘OVL_COVER_BASIC.
	`‘OVL_COVER_STATISTIC`	*Reserved for future use.*

Defines for Specific Assertions

Parameter	Checkers	DEFINE	Description
action_on_new_start	assert_change assert_frame assert_time assert_unchange	‘OVL_IGNORE_NEW_START	(default) Ignore new start events.
		‘OVL_RESET_ON_NEW_START	Restart check on new start events.
		‘OVL_ERROR_ON_NEW_START	Assert fail on new start events.
edge_type	assert_always_on_edge	‘OVL_NOEDGE	(default) Always initiate check.
		‘OVL_POSEDGE	Initiate check on rising edge of sampling event.
		‘OVL_NEGEDGE	Initiate check on falling edge of sampling event.
		‘OVL_ANYEDGE	Initiate check on both edges of sampling event.
necessary_condition	assert_cycle_sequence	‘OVL_TRIGGER_ON_MOST_PIPE	(default) Necessary condition is full sequence. Pipelining enabled.
		‘OVL_TRIGGER_ON_FIRST_PIPE	Necessary condition is first in sequence. Pipelining enabled.
		‘OVL_TRIGGER_ON_FIRST_NOPIPE	Necessary condition is first in sequence. Pipelining disabled.
inactive	assert_one_cold	‘OVL_ALL_ZEROS	Inactive state is all 0’s.
		‘OVL_ALL_ONES	Inactive state is all 1’s.
		‘OVL_ONE_COLD	(default) No inactive state.

Chapter 5
OVL Backward Compatibility

V1.8

In V1.8, aside from bug fixes, all functionality is backward compatible.

V1.7

In V1.7, implicit X/Z checking was implemented. Implicit X/Z checks were added to many checkers, so additional assertion violations might be generated. Also, cover point typing (SANITY, BASIC and COVER types) was implemented which allows for finer-grained control over cover point data collection. Finally, the ‘OVL_RUNTIME_AFTER_FATAL global variable was added to allow modification of the runtime period from a fatal error to end of simulation (which was previously fixed at 100 time units).

Aside from bug fixes, all other functionality is backward compatible.

V1.6

In V1.6, aside from bug fixes, all functionality is backward compatible.

V1.5

In V1.5, PSL versions of checkers were added. The ‘OVL_IGNORE property_type value was added. Aside from bug fixes, all functionality is backward compatible.

V1.1

In V1.1, a typo was corrected in the port list of the assert_implication checker type. The port name *antecendent_expr* was changed to *antecedent_expr*.

V1.0

Backward compatibility with the non-standard OVL library is important and no changes were made for the V1.0 release in the following areas: naming of module names, naming of port names and to the extent possible the existing Verilog use model.

The name of the *options* parameter was changed to *property_type*. The only checker type that is not backward compatible in this respect is the assert_fifo_index checker.

assert_change and assert_unchange

In previous OVL versions, the window for these checkers closed when an assertion violation was detected, which in effect made the durations of these windows variable. In V1.0, these assertion checkers were recoded to use windows of fixed length (*num_cks* cycles).

assert_fifo_index

In previous OVL versions, the assert_fifo_index checker used the second bit of the *option* parameter to prohibit simultaneous pushes-pops in the same cycle. In V1.0, the *property_type* parameter is compatible with the first bit of previous *options* parameter. But, the second bit (if defined) is ignored. To enable the check for simultaneous pushes-pops, use the *simultaneous_push_pop* parameter (at the end of the parameter list).