Accellera Standard OVL V2
Library Reference Manual

Software Version 2.0rc

June 2007

STATEMENT OF USE OF ACCELLERA STANDARDS

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 Such documents modify this standard at the time of their publication and remain in effect until superseded by subsequent documents or until the standard is officially revised.

Major version 2.0 released April 2007

Table of Contents

Chapter 4
OVL Defines **241**

Chapter 1
Introduction

Welcome to the Accellera standard Open Verification Library V2 (OVL). 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 V2 is an extension of OVL V1. The OVL V2 augments the structure of the V1 original checkers by adding parameters, ports and control logic. These new checker versions are similar, but not completely identical to their V1 counterparts. The V1 checker types were named with an "assert_" prefix and their V2 counterparts are named with an "ovl_" prefix, with the same base names. For backward compatibility, all OVL V1 checkers (assert_* checkers) are available and supported in OVL V2. So, all existing code utilizing OVL V1 will function the same with OVL V2 (except for bug fixes and enhancements).

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 V2, 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.

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 ::= ovl_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 ])

ovl_identifier ::= ovl_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.

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 a 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 flags that indicate whether or not specific corner-case events occur and counts that tote up the occurrences of specific common events.

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.

- 1-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.

Checker Parameters

Each OVL assertion checker has its own set of parameters as described in its corresponding data sheet. The following parameters are (typically) common to all checkers: *severity_level, property_type, msg, coverage_level, clock_edge, reset_polarity* and *gating_type*. Each of these types of parameters has a default value used when the corresponding checker parameter is unspecified in the checker instance specification. These defaults are set by the following global defines (which can be modified): 'OVL_SEVERITY_DEFAULT, 'OVL_PROPERTY_ DEFAULT, 'OVL_MSG_DEFAULT, 'OVL_COVER_DEFAULT, 'OVL_CLOCK_EDGE_ DEFAULT, OVL_RESET_POLARITY_DEFAULT and 'OVL_GATING_TYPE_DEFAULT (see "Setting Checker Parameter Defaults" on page 21).

severity_level

A checker's "severity level" determines how to handle an assertion violation. The *severity_level* parameter sets the checker's severity level and can have one of the following values:

`'OVL_FATAL`	Runtime fatal error.
`'OVL_ERROR`	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

A checker's "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). The property type also selects whether to assert/assume X/Z value checks or not. The *property_type* parameter sets the checker's property type and can have one of the following values:

`'OVL_ASSERT`	Assert assertion check and X/Z check properties.
`'OVL_ASSUME`	Assume assertion check and X/Z check properties.
`'OVL_ASSERT_2STATE`	Assert assertion check properties. Ignore X/Z check properties.
`'OVL_ASSUME_2STATE`	Assume assertion check properties. Ignore X/Z check properties.
`'OVL_IGNORE`	Ignore assertion check and X/Z check properties.

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 value of 'OVL_MSG_DEFAULT is "VIOLATION". Changing this define provides a default message printed when a checker assertion is violated. To override this default message for an individual checker, set the checker's *msg* parameter.

coverage_level

A checker's "coverage level" determines the cover point information reported by the individual checker. The *coverage_level* parameter sets the checker's coverage level. This parameter can be any logical bitwise-OR of the defined cover point type values ("Cover Points" on page 18 and "Monitoring Coverage" on page 20):

`'OVL_COVER_SANITY`	Report SANITY cover points.
`'OVL_COVER_BASIC`	Report BASIC cover points.
`'OVL_COVER_CORNER`	Report CORNER cover points.
`'OVL_COVER_STATISTIC`	Report STATISTIC cover points.

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`	Report information for all cover points.

clock_edge

A checker's "clock edge" selects the active edges for the *clock* input to the checker. Edge-triggered checkers perform their analyses—which include evaluating inputs, checking assertions and updating counters—at the active edges of their clocks. The elapsed time from one active clock edge to the next is referred to as a *clock cycle* (or simply *cycle*). The *clock_edge* parameter specifies the checker's active clock edges and can have one of the following values:

`'OVL_POSEDGE`	Rising edges are active clock edges.
`'OVL_NEGEDGE`	Falling edges are active clock edges.

reset_polarity

A checker's "reset polarity" selects the *active level* of the checker *reset* input. When reset becomes active, the checker clears pending properties and internal values (coverage point values remain unchanged). A subsequent edge of the *reset* signal makes *reset* inactive, which initializes and activates the checker. The *reset_polarity* parameter sets the checker's reset polarity and can have one of the following values:

`'OVL_ACTIVE_LOW`	*Reset* is active when FALSE.
`'OVL_ACTIVE_HIGH`	*Reset* is active when TRUE.

gating_type

A checker's "gating type" selects the signal gated by the *enable* input. The *gating_type* parameter can be set to one of the following values:

`'OVL_GATE_NONE`	Checker ignores the *enable* input.
`'OVL_GATE_CLOCK`	Checker pauses when *enable* is FALSE. The checker treats the current cycle as a NOP. Checks, counters and internal values remain unchanged.
`'OVL_GATE_RESET`	Checker resets (as if the *reset* input became active) when *enable* is FALSE.

Checker Ports

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

clock

Each "edge-triggered" assertion checker has a clocking input port named *clock*. All of the checker's sampling, assertion checking and coverage collection tasks are performed at "active" edges of the checker's *clock* input. The active clock edges are set by the checker's *clock_edge* parameter (page 13): 'OVL_POSEDGE (rising edges) or 'OVL_NEGEDGE (falling edges). The default *clock_edge* parameter is set by the following global variable:

`'OVL_CLOCK_EDGE_` `DEFAULT`	Sets the default *clock_edge* parameter value for checkers. Default: 'OVL_POSEDGE.

Gating *clock*

If a checker's *gating_type* parameter (page 13) is set to 'OVL_GATE_CLOCK, the checker's *enable* signal 'gates' the *clock* input to the checker. Here the actual clock signal used internally by the checker is the gated clock formed combinationally from *clock* and *enable*. Deasserting *enable* in effect pauses the checker at the current state. No data ports are sampled; no checking is performed; no counters are incremented; and no coverage data are collected. When *enable* asserts again, the checker continues from the state it was "paused" by *enable*.

The internal clock for a checker (called *clk*) is formed combinationally from *clock* and possibly *enable* (based on the gating type and active clock edge for the checker) using the following logic:

```
wire gclk, clk;
'ifdef OVL_GATING_OFF
   assign gclk = clock; // Globally disabled gating
'else
   // LATCH based gated clock
   reg  clken;
   always @ (clock or enable) begin
      if (clock == 1'b0)
         clken <= enable;
   end
   assign gclk = (gating_type == 'OVL_GATE_CLOCK) ? clock & clken
                          : clock; // Locally disabled gating
'endif // OVL_GATING_OFF
// clk (programmable edge & optional gating)
assign clk = (clock_edge == 'OVL_POSEDGE) ? gclk : ~gclk;
```

Note that setting the 'OVL_GATING_OFF define disables clock (and reset) gating for all checkers.

reset

Each assertion checker has a reset input port named *reset*. Associated with the *reset* port is the checker's *reset_polarity* parameter: 'OVL_ACTIVE_LOW (*reset* active when FALSE) or 'OVL_ACTIVE_HIGH (*reset* active when TRUE). The default *reset_polarity* parameter is set by the following global variable:

`'OVL_RESET_POLARITY_` `DEFAULT`	Sets the default *reset_polarity* parameter value for checkers. Default: 'OVL_ACTIVE_LOW.

When a checker that is not in reset mode samples an active *reset*, the checker enters reset mode. The checker cancels pending assertion checks and freezes coverage data at their current values. At the next active clock edge that *reset* is not active, the checker exits reset mode. The checker initializes assertion properties and the checker behaves as it started from its initialized state—except coverage data continues from the values frozen during the reset interval.

Gating *reset*

If a checker's *gating_type* parameter is set to 'OVL_GATE_RESET, its *enable* signal 'gates' the *reset* input to the checker. Here the reset signal used internally by the checker is the gated input formed combinationally from *reset* and *enable* (and inverted if *reset* is active high). The *enable* input acts as a second, active-low reset.

The internal reset for a checker (called *reset_n*) is formed combinationally from *reset* and possibly *enable* using the following logic:

```
wire greset, reset_n;
'ifdef OVL_GATING_OFF
   assign greset = reset; // Globally disabled gating
'else
   assign greset = (gating_type == 'OVL_GATE_RESET) ? reset & enable
                   : reset; // Locally disabled gating
'endif // OVL_GATING_OFF
// reset_n (programmable polarity & optional gating)
assign reset_n = (reset_polarity == 'OVL_ACTIVE_LOW) ? greset : ~greset;
```

Global Reset

The *reset* port assignments of all assertion checkers can be overridden and controlled by the following global variable:

`'OVL_GLOBAL_RESET=` *reset_signal*	Overrides the *reset* port assignments of all assertion checkers with the specified global *reset_signal*. Checkers ignore their *reset_polarity* parameters and treat the global reset as an active-low reset. Default: each checker's reset is specified by the *reset* port and *reset_polarity* parameters.

Internally, each checker uses the reset signal defined by 'OVL_RESET_SIGNAL:

```
// 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
```

enable

Each assertion checker has an enabling input port named *enable*. This input is used to gate either the *clock* or *reset* signals for the checker (effectively pausing or resetting the checker). The effect of the enable port on the checker is determined by the checker's *gating_type* parameter (page 13):

- 'OVL_GATE_NONE (no effect),

- 'OVL_GATE_CLOCK (gate *clock*, see "Gating clock" on page 14) or

- 'OVL_GATE_RESET (gate *reset*, see "Gating reset" on page 15).

The default *gating_type* parameter is set by the following global variable:
'OVL_GATING_TYPE_DEFAULT (default: 'OVL_GATE_CLOCK).

fire

Each assertion checker has a fire signal output port named *fire*. Future OVL releases might extend this output, so extra bits are reserved for future use. For the V2.0 release of OVL, this is a 3-bit port:

```
'define OVL_FIRE_WIDTH 3
```

The *fire* output port has the following bits:

`fire[0]`	Assertion fired in 2-state mode (an assertion check violation).
`fire[1]`	X/Z check fired in non-2-state mode.
`fire[2]`	Coverage fired.

Each *fire* output bit is a registered output (i.e., it is implemented in a clocked process) that goes TRUE for one 'cycle' after the event and then resets to FALSE. If clock-gating is enabled (i.e., the default case) and *enable* deasserts at a clock edge where a *fire* bit asserts, then the *fire* bit remains TRUE while the checker is paused (i.e., until *enable* asserts again).

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*, *coverage_level*, *clock_edge*, *reset_polarity* and *gating_type*).

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

FRAME	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 (see "Using a Global Reset" on page 22).

Explicit X/Z Checks

Two assertion checker types are specifically designed to verify that their associated expressions have known and driven values: ovl_never_unknown and ovl_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 ovl_never_unknown and ovl_never_unknown_async checkers are excluded).

Implicit X/Z Checks

All assertion checker types — except ovl_never_unknown and ovl_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 ovl_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 ovl_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`	Counts of relevant states assumed by the logic monitored by the assertion checker.

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`   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 both of these variables are undefined, 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 a checker's properties are asserts (i.e., checks) or assumes (i.e., constraints) is controlled by the checker's *property_type* parameter. In addition, property_type can turn on and off X/Z checks:

`'OVL_ASSERT`   Assertion checks and X/Z checks are asserts.

`'OVL_ASSUME`   Assertion checks and X/Z checks are assumes.

`'OVL_ASSERT_2STATE`   Assertion checks are asserts. X/Z checks are excluded.

`'OVL_ASSUME_2STATE`   Assertion checks are assumes. X/Z checks are excluded.

`'OVL_IGNORE`   Ignore (exclude) assertion checks and X/Z check properties.

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.

Setting Checker Parameter Defaults

All common parameters for checkers and some parameters common to specific checker types have default parameter values. These are the parameter values assumed by the checker when the parameter is not specified. The std_ovl_defines.h sets the values of these defaults (i.e., to default default values), but the default values can be overridden by redefining them. The following Verilog defines set the values of these default parameter values for the common checker parameters:

`` `OVL_SEVERITY_DEFAULT``	Value of *severity_level* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_ERROR.
`` `OVL_PROPERTY_DEFAULT``	Value of *property_type* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_ASSERT.
`` `OVL_MSG_DEFAULT``	Value of *msg* to use when it is not specified. The value defined in std_ovl_defines.h is "VIOLATION".
`` `OVL_COVER_DEFAULT``	Value of *coverage_level* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_COVER_BASIC.
`` `OVL_CLOCK_EDGE_ DEFAULT``	Value of *clock_edge* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_POSEDGE.
`` `OVL_RESET_POLARITY_ DEFAULT``	Value of *reset_polarity* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_ACTIVE_LOW.
`` `OVL_GATING_TYPE_ DEFAULT``	Value of *gating_type* to use when it is not specified. The value defined in std_ovl_defines.h is `OVL_GATE_CLOCK.

Disabling Clock/Reset Gating

By default, if a checker's *gating_type* parameter is `OVL_GATE_CLOCK, the checker's internal clock logic is gated by the checker's *enable* input. Similarly, by default, if a checker's *gating_type* parameter is `OVL_GATE_RESET, the checker's internal reset logic is gated by the checker *enable* input. Setting the following define, overrides this behavior:

`` `OVL_GATING_OFF``	Turns off clock/reset gating, effectively setting all gating_type parameters to `OVL_GATE_NONE, so checkers ignore their enable inputs. Default: gating type specified by each checker's gating_type parameter.

Using a Global Reset

The *reset* port assignments of all assertion checkers can be overridden and controlled by the following global variable:

`OVL_GLOBAL_RESET=`
reset_signal

Overrides the *reset* port assignments of all assertion checkers with the specified global *reset_signal*. Checkers ignore their *reset_polarity* parameters and treat the global reset as an active-low reset. Default: each checker's reset is specified by the *reset* port and *reset_polarity* parameters.

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).

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: V2.0: instance_name initialized @ hierarchy Severity:
severity_level, Message: msg
```

End-of-simulation Signal to ovl_quiescent_state Checkers

The ovl_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 typically continues for a certain amount of time and then the simulation ends. However, the OVL logic can be configured so that runtime fatal errors do not end simulation. These behaviors are controlled by the following global variables:

`'OVL_RUNTIME_AFTER_`
`FATAL=time`   Number of time units from a fatal error to end of simulation. Default: 100.

`'OVL_FINISH_OFF`   Fatal errors do not stop simulation. Default: fatal error ends simulation after 'OVL_RUNTIME_AFTER_FATAL time units.

OVL 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
 .   .   .
```

Header Files

std_ovl_defines.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

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

```
`define OVL_VERSION "V2.0"

`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

   // default edge_type (ovl_always_on_edge)
`ifdef OVL_EDGE_TYPE_DEFAULT
   // do nothing
`else
   `define OVL_EDGE_TYPE_DEFAULT `OVL_NOEDGE
`endif


   // severity levels
`define OVL_FATAL   0
`define OVL_ERROR   1
`define OVL_WARNING 2
`define OVL_INFO    3

   // default severity level
`ifdef OVL_SEVERITY_DEFAULT
   // do nothing
`else
   `define OVL_SEVERITY_DEFAULT `OVL_ERROR
`endif

   // coverage levels (note that 3 would set both SANITY & BASIC)
`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
`ifdef OVL_COVER_DEFAULT
   // do nothing
`else
   `define OVL_COVER_DEFAULT `OVL_COVER_BASIC
`endif
```

```
// property type
`define OVL_ASSERT        0
`define OVL_ASSUME        1
`define OVL_IGNORE        2
`define OVL_ASSERT_2STATE 3
`define OVL_ASSUME_2STATE 4

// fire bit positions (first two also used for xcheck input to error_t)
`define OVL_FIRE_2STATE 0
`define OVL_FIRE_XCHECK 1
`define OVL_FIRE_COVER  2

// default property type
`ifdef OVL_PROPERTY_DEFAULT
  // do nothing
`else
  `define OVL_PROPERTY_DEFAULT `OVL_ASSERT
`endif

// default message
`ifdef OVL_MSG_DEFAULT
  // do nothing
`else
  `define OVL_MSG_DEFAULT "VIOLATION"
`endif

// necessary condition
`define OVL_TRIGGER_ON_MOST_PIPE    0
`define OVL_TRIGGER_ON_FIRST_PIPE   1
`define OVL_TRIGGER_ON_FIRST_NOPIPE 2

// default necessary_condition (ovl_cycle_sequence)
`ifdef OVL_NECESSARY_CONDITION_DEFAULT
  // do nothing
`else
  `define OVL_NECESSARY_CONDITION_DEFAULT `OVL_TRIGGER_ON_MOST_PIPE
`endif

// action on new start
`define OVL_IGNORE_NEW_START   0
`define OVL_RESET_ON_NEW_START 1
`define OVL_ERROR_ON_NEW_START 2

// default action_on_new_start (e.g. ovl_change)
`ifdef OVL_ACTION_ON_NEW_START_DEFAULT
  // do nothing
`else
  `define OVL_ACTION_ON_NEW_START_DEFAULT `OVL_IGNORE_NEW_START
`endif

// inactive levels
`define OVL_ALL_ZEROS 0
`define OVL_ALL_ONES  1
`define OVL_ONE_COLD  2
```

```
// default inactive (ovl_one_cold)
`ifdef OVL_INACTIVE_DEFAULT
  // do nothing
`else
  `define OVL_INACTIVE_DEFAULT `OVL_ONE_COLD
`endif

// ovl 2.0 new interface
`define OVL_ACTIVE_LOW  0
`define OVL_ACTIVE_HIGH 1

`define OVL_GATE_NONE   0
`define OVL_GATE_CLOCK 1
`define OVL_GATE_RESET 2

`define OVL_FIRE_WIDTH   3

`ifdef OVL_CLOCK_EDGE_DEFAULT
  // do nothing
`else
  `define OVL_CLOCK_EDGE_DEFAULT `OVL_POSEDGE
`endif

`ifdef OVL_RESET_POLARITY_DEFAULT
  // do nothing
`else
`define OVL_RESET_POLARITY_DEFAULT `OVL_ACTIVE_LOW
`endif

`ifdef OVL_GATING_TYPE_DEFAULT
  // do nothing
`else
`define OVL_GATING_TYPE_DEFAULT `OVL_GATE_CLOCK
`endif

// ovl runtime after fatal error
`define OVL_RUNTIME_AFTER_FATAL 100

// Covergroup define
`ifdef OVL_COVER_ON
  `ifdef OVL_COVERGROUP_OFF
  `else
    `define OVL_COVERGROUP_ON
  `endif // OVL_COVERGROUP_OFF
`endif // OVL_COVER_ON

// Ensure x-checking logic disabled if ASSERTs are off
`ifdef OVL_ASSERT_ON
`else
  `define OVL_XCHECK_OFF
  `define OVL_IMPLICIT_XCHECK_OFF
`endif

`endif // OVL_STD_DEFINES_H
```

std_ovl_init.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

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

std_ovl_clock.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

`ifdef OVL_SHARED_CODE
  wire gclk, clk;
  `ifdef OVL_GATING_OFF
    assign gclk = clock; // Globally disabled gating
  `else
    // LATCH based gated clock
    reg  clken;
    always @ (clock or enable) begin
      if (clock == 1'b0)
        clken <= enable;
    end
    assign gclk = (gating_type == `OVL_GATE_CLOCK) ? clock & clken
                                   : clock; // Locally disabled gating
  `endif // OVL_GATING_OFF
  // clk (programmable edge & optional gating)
  assign clk = (clock_edge == `OVL_POSEDGE) ? gclk : ~gclk;
`endif // OVL_SHARED_CODE
```

std_ovl_reset.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

`ifdef OVL_SHARED_CODE
  wire greset, reset_n;
  `ifdef OVL_GATING_OFF
    assign greset = reset; // Globally disabled gating
  `else
    assign greset = (gating_type == `OVL_GATE_RESET) ? reset & enable
                                   : reset; // Locally disabled gating
  `endif // OVL_GATING_OFF
  // reset_n (programmable polarity & optional gating)
  assign reset_n = (reset_polarity == `OVL_ACTIVE_LOW) ? greset : ~greset;
`endif // OVL_SHARED_CODE
```

std_ovl_count.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

// Support for printing of count of OVL assertions
`ifdef OVL_INIT_MSG
`ifdef OVL_INIT_COUNT
  integer ovl_init_count;
  initial begin
    // Reset, prior to counting
    ovl_init_count = 0;
    // Display total number of OVL instances, just after initialization
    $monitor("\nOVL_METRICS: %d OVL assertions initialized\n"\
                            ,ovl_init_count);
  end
`endif
`endif
```

std_ovl_cover.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

// Parameters that should not be edited

  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);
```

std_ovl_task.h

```
// Accellera Standard V2.0 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2007. All rights reserved.

`ifdef OVL_SYNTHESIS
`else
  integer error_count;
  integer cover_count;
  initial error_count = 0;
  initial cover_count = 0;
`endif // OVL_SYNTHESIS
```

```
     task ovl_error_t;
       input              xcheck;
       input [8*128-1:0] err_msg;
       reg   [8*16-1:0]  err_typ;
     begin
     `ifdef OVL_SYNTHESIS
     `else
       case (severity_level)
         `OVL_FATAL   : err_typ = "OVL_FATAL";
         `OVL_ERROR   : err_typ = "OVL_ERROR";
         `OVL_WARNING : err_typ = "OVL_WARNING";
         `OVL_INFO    : err_typ = "OVL_INFO";
         default      :
           begin
             err_typ = "OVL_ERROR";
             $display("OVL_ERROR: Illegal option used in parameter
severity_level, setting message type to OVL_ERROR : time %0t : %m",
$time);
           end
       endcase

       error_count = error_count + 1;
       `ifdef OVL_MAX_REPORT_ERROR
         if (error_count <= `OVL_MAX_REPORT_ERROR) begin
       `endif
           case (property_type)
             `OVL_ASSERT,
             `OVL_ASSUME         : begin
               $display("%s : %s : %s : %0s : severity %0d : time %0t : %m",
err_typ, assert_name, msg, err_msg, severity_level, $time);
             end
             `OVL_ASSERT_2STATE,
             `OVL_ASSUME_2STATE  : begin
               if (xcheck == `OVL_FIRE_2STATE) begin
               $display("%s : %s : %s : %0s : severity %0d : time %0t : %m",
err_typ, assert_name, msg, err_msg, severity_level, $time);
               end
             end
             `OVL_IGNORE         : begin end
             default             : begin end
           endcase
       `ifdef OVL_MAX_REPORT_ERROR
         end
       `endif

       `ifdef OVL_FINISH_OFF
       `else
         if (severity_level == `OVL_FATAL) ovl_finish_t;
       `endif // OVL_FINISH_OFF
     `endif // OVL_SYNTHESIS
     end
     endtask // ovl_error_t
```

```
task ovl_finish_t;
begin
  `ifdef OVL_SYNTHESIS
  `else
    #`OVL_RUNTIME_AFTER_FATAL $finish;
  `endif // OVL_SYNTHESIS
end
endtask // ovl_finish_t


task ovl_init_msg_t;
begin
  `ifdef OVL_SYNTHESIS
  `else
    case (property_type)
      `OVL_ASSERT,
      `OVL_ASSUME,
      `OVL_ASSERT_2STATE,
      `OVL_ASSUME_2STATE : begin
        `ifdef OVL_SYNTHESIS
        `else
          `ifdef OVL_INIT_COUNT
            #0.1 `OVL_INIT_COUNT = `OVL_INIT_COUNT + 1;
          `else
            $display("OVL_NOTE: %s: %s initialized @ %m Severity: %0d,
                  Message: %s", `OVL_VERSION, assert_name,severity_level,
                  msg);
          `endif
        `endif // OVL_SYNTHESIS
      end
      `OVL_IGNORE : begin
          // do nothing
      end
     default : $display("OVL_ERROR: Illegal option used in parameter
                  property_type : %m");
    endcase
  `endif // OVL_SYNTHESIS
end
endtask // ovl_init_msg_t
```

```
task ovl_cover_t;
  input [8*64-1:0] cvr_msg;
begin
 `ifdef OVL_SYNTHESIS
 `else
   cover_count = cover_count + 1;
    `ifdef OVL_MAX_REPORT_COVER_POINT
      if (cover_count <= `OVL_MAX_REPORT_COVER_POINT) begin
    `endif
       if (coverage_level > `OVL_COVER_ALL)
         $display("OVL_ERROR: Illegal option used in parameter
                  coverage_level : time %0t : %m", $time);
       else
         $display("OVL_COVER_POINT : %s : %0s : time %0t : %m",
                  assert_name, cvr_msg, $time);
    `ifdef OVL_MAX_REPORT_COVER_POINT
      end
    `endif
 `endif // OVL_SYNTHESIS
end
endtask // ovl_cover_t


`ifdef OVL_SYNTHESIS
`else
  // FUNCTION THAT CALCULATES THE LOG BASE 2 OF A NUMBER
  // ====================================================
  // NOTE: only used in sva31a
  function integer log2;
    input integer x;
    integer i;
    integer result;
  begin
    result = 1;
    if (x <= 0) result = -1;
    else
      for (i = 0; (1<<i) <= x; i=i+1) result = i+1;
    log2 = result;
  end
  endfunction
`endif // OVL_SYNTHESIS


  function ovl_fire_2state_f;
    input   property_type;
    integer property_type;
  begin
    case (property_type)
      `OVL_ASSERT,
      `OVL_ASSUME        : ovl_fire_2state_f = 1'b1;
      `OVL_ASSERT_2STATE,
      `OVL_ASSUME_2STATE : ovl_fire_2state_f = 1'b1;
      `OVL_IGNORE        : ovl_fire_2state_f = 1'b0;
      default            : ovl_fire_2state_f = 1'b0;
    endcase
  end
  endfunction // ovl_fire_2state_f
```

```
function ovl_fire_xcheck_f;
  input   property_type;
  integer property_type;
begin
`ifdef OVL_SYNTHESIS
  // fire_xcheck is not synthesizable
  ovl_fire_xcheck_f = 1'b0;
`else
  case (property_type)
    `OVL_ASSERT,
    `OVL_ASSUME        : ovl_fire_xcheck_f = 1'b1;
    `OVL_ASSERT_2STATE,
    `OVL_ASSUME_2STATE : ovl_fire_xcheck_f = 1'b0;
    `OVL_IGNORE        : ovl_fire_xcheck_f = 1'b0;
    default            : ovl_fire_xcheck_f = 1'b0;
  endcase
`endif // OVL_SYNTHESIS
end
endfunction // ovl_fire_xcheck_f
```

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.

ovl_always

Ensures that the value of an expression is TRUE.

```
                              fire['OVL_FIRE_WIDTH-1:0]

   test_expr              ovl_always


                    clock   reset   enable
```

Parameters:
severity_level
property_type
msg

coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion
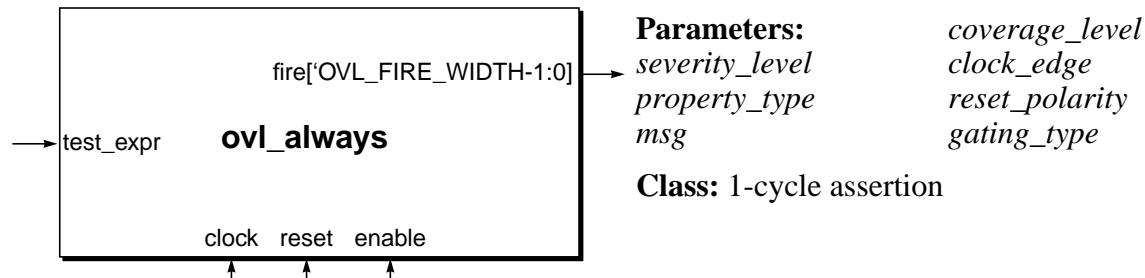
Syntax

```
ovl_always
    [#(severity_level, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr	Expression that should evaluate to TRUE on the rising clock edge.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_always assertion checker checks the single-bit expression *test_expr* at each active edge of *clock.* If *test_expr* is not TRUE, an always check violation occurs.

Assertion Checks

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

See also

ovl_always_on_edge   ovl_never
ovl_implication   ovl_proposition

Example

```
ovl_always #(

    'OVL_ERROR,                              // severity_level
    'OVL_ASSERT,                             // property_type
    "Error: reg_a < reg_b is not TRUE",      // msg
    'OVL_COVER_NONE,                         // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    reg_a_lt_reg_b (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        reg_a < reg_b,                       // test_expr
        fire);                               // fire
```

Ensures that ($reg\_a < reg\_b$) is TRUE at each rising edge of *clock*.

ALWAYS Error: reg_a < reg_b is not TRUE
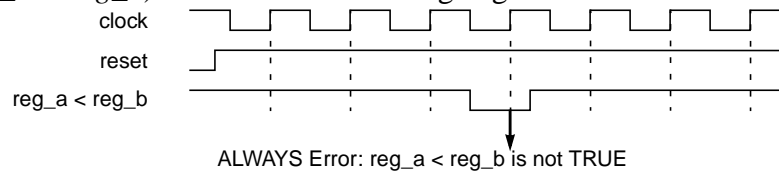
ovl_always_on_edge

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

```
                          fire['OVL_FIRE_WIDTH-1:0]

   sampling_event
            ovl_always_on_edge

   test_expr

            clock  reset  enable
```

Parameters:

severity_level   *coverage_level*
edge_type   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: 2-cycle assertion
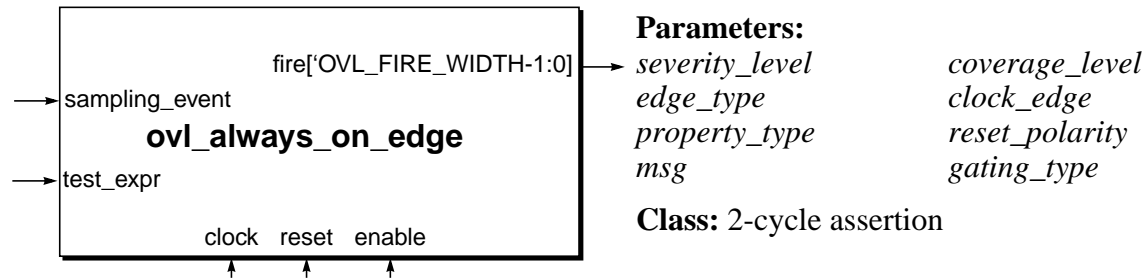
Syntax

```
ovl_always_on_edge
      [#(severity_level, edge_type, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_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 active edge of *clock* 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 ovl_always.

The checker is a variant of ovl_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

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   Expression value was X or Z.
or Z

sampling_event   Sampling event value was X or Z.
contains X or Z

Cover Points

See also

ovl_always   ovl_never
ovl_implication   ovl_proposition

Examples

```
ovl_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_NONE,                          // coverage_level
    `OVL_POSEDGE,                             // clock_edge
    `OVL_ACTIVE_LOW,                          // reset_polarity
    `OVL_GATE_CLOCK )                         // gating_type

    request_when_FSM_idle (

        clock,                                // clock
        reset,                                // reset
        enable,                               // enable
        req,                                  // sampling_event
        state == `IDLE,                       // test_expr
        fire_request_when_FSM_idle);          // fire
```

Ensures that *(state == 'IDLE)* is TRUE at each rising edge of *clock* when *req* transitions from
FALSE to TRUE.

ALWAYS_ON_EDGE Error: new req when FSM not ready
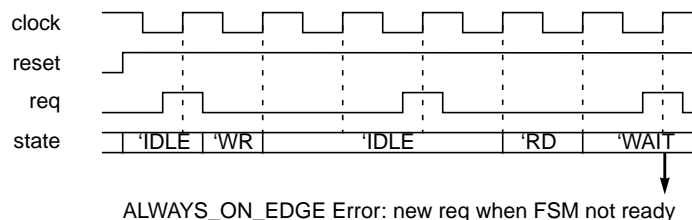
```
ovl_always_on_edge #(

    'OVL_ERROR,                                  // severity_level
    'OVL_ANYEDGE,                                // edge_type
    'OVL_ASSERT,                                 // property_type
    "Error: req transition when FSM not idle",   // msg
    'OVL_COVER_NONE,                             // coverage_level
    'OVL_POSEDGE,                                // clock_edge
    'OVL_ACTIVE_LOW,                             // reset_polarity
    'OVL_GATE_CLOCK )                            // gating_type

    req_transition_when_FSM_idle (

        clock,                                   // clock
        reset,                                   // reset
        enable,                                  // enable
        req,                                     // sampling_event
        state == 'IDLE,                          // test_expr
        fire_req_transition_when_FSM_idle);      // fire
```

Ensures that *(state == 'IDLE)* is TRUE at each rising edge of *clock* when *req* transitions from TRUE to FALSE or from FALSE to TRUE.

ALWAYS_ON_EDGE Error: req transition when FSM not idle
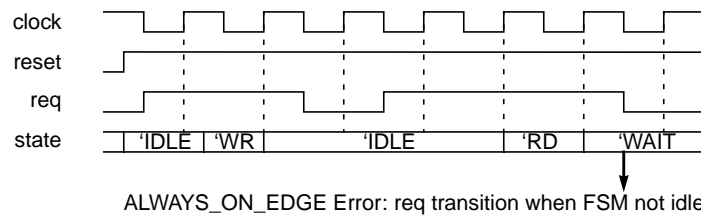
```
ovl_always_on_edge #(

    'OVL_ERROR,                                  // severity_level
    'OVL_NOEDGE,                                 // edge_type
    'OVL_ASSERT,                                 // property_type
    "Error: req when FSM not idle",              // msg
    'OVL_COVER_NONE,                             // coverage_level
    'OVL_POSEDGE,                                // clock_edge
    'OVL_ACTIVE_LOW,                             // reset_polarity
    'OVL_GATE_CLOCK )                            // gating_type

    req_when_FSM_idle (

        clock,                                   // clock
        reset,                                   // reset
        enable,                                  // enable
        1'b0,                                    // sampling_event
        !req || (state == 'IDLE),                // test_expr
        fire_req_when_FSM_idle);                 // fire
```

Ensures that *(!req || (state == 'IDLE))* is TRUE at each rising edge of *clock*.

ALWAYS_ON_EDGE Error: req when FSM not idle

---
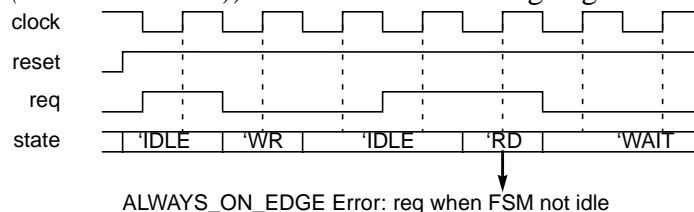
ovl_arbiter

Ensures that a resource arbiter provides grants to corresponding requests according to a specified arbitration scheme and within a specified time window.

Parameters:

severity_level	*single_gnt_check*
width	*property_type*
priority_width	*msg*
min_cks	*coverage_level*
max_cks	*clock_edge*
arbitration_rule	*reset_polarity*
priority_check	*gating_type*
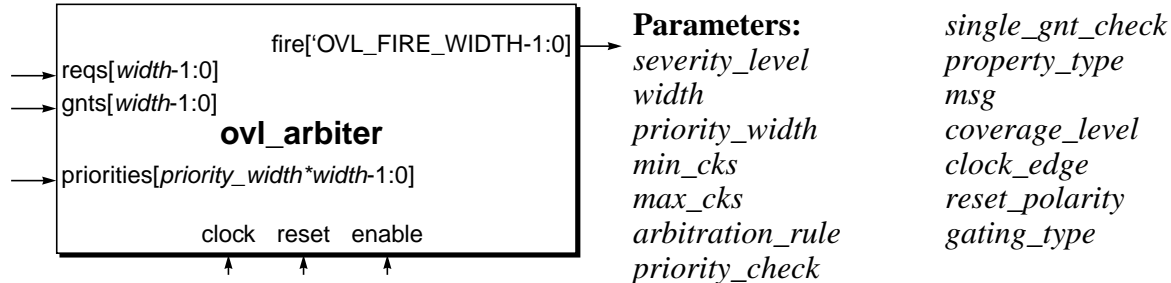
Class: event-bounded assertion

Syntax

```
ovl_arbiter
    [#(severity_level, width, priority_width, min_cks, max_cks,
        arbitration_rule, priority_check, single_gnt_check,
        property_type, msg, coverage_level, clock_edge, reset_polarity,
        gating_type)]
    instance_name (clock, reset, enable, reqs, gnts, priorities, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *reqs* and *gnts* ports (number of channels). Default: 2.
priority_width	Number of bits to encode a priority value in *priorities*. Default: 1.
min_cks	Minimum number of clock cycles after a request that its grant can be issued. If *min_cks* is 0, a grant can be issued in the same cycle the request is made. Default: 1
max_cks	Maximum number of clock cycles after a request that its grant can be issued. A value of 0 indicates no upper bound for grants. Default: 0.
arbitration_rule	Arbitration scheme used by the arbiter. This parameter turns on the corresponding check for the arbitration scheme. *arbitration_rule = 0* (Default) no scheme *arbitration_rule = 1* fair (round robin) *arbitration_rule = 2* FIFO *arbitration_rule = 3* least-recently used

priority_check	Whether or not to perform priority checks.
	priority_check = 0 (Default)
	    Turns off the priority check.
	priority_check = 1
	    Turns on the priority check. The *min_cks* parameter must be 0
	    or 1.

single_gnt_check	Whether or not to perform grant_one checks.
	single_gnt_check = 0
	    Turns off the grant_one check.
	single_gnt_check = 1 (Default)
	    Turns on the grant_one check.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
reqs[*width*-1:0]	Concatenation of request signals to the arbiter. Each bit in the vector is a request from the corresponding channel.
gnts[*width*-1:0]	Concatenation of grant signals from the arbiter. Each bit in the vector is a grant to the corresponding channel.

`priorities`
`[priority_width*width`
`-1:0]`
	Concatenation of non-negative integer values corresponding to the request priorities of the corresponding *req* channels (0 is the lowest priority). If the priority check is on, *priorities* must not change while any channel is waiting for a grant (otherwise certain checks might produce incorrect results). If the priority check is off, this port is ignored (however, the port must be configured with the specified width).

`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_arbiter checker ensures an arbiter follows a specified arbitration process. The checker checks *reqs* and *gnts* at each active edge of *clock*. These are two bit vectors representing respectively requests from the channels and grants from the arbiter. Both vectors have the same size (width), which is the same as the number of channels.

A request from a channel is signaled by asserting its corresponding *reqs* bit, which should be followed (according to the configured arbitration rules) by a responding assertion of the same bit in *gnts*. If a request deasserts before the arbiter issues the corresponding grant, all checks for that request are cancelled. If a request remains asserted in the cycle its grant is issued, a new request is assumed.

The ovl_arbiter checker checks the following rules:

- A grant should not be issued to a channel without a request.

- A grant asserts for one cycle (unless the grant is for consecutive requests).

- A grant should be issued in the time window specified by [*min_cks*:*max_cks*] after its request.

The ovl_arbiter checker can be configured to check that at most one grant is issued each cycle (i.e., a single grant at a time).

The ovl_arbiter checker also can be configured to check a specific arbitration scheme by turning the priority check on or off and selecting a value for *arbitration_rule*. The combination of the two selections determines the expected arbitration scheme.

- Primary rule.

  If the priority check is on, priority arbitration is the primary rule. When a request is made, the values in *priorities* are the priorities of the corresponding channels in ascending priority order (a value of 0 is the lowest priority). If multiple requests are pending, the grant should be issued to the channel with the highest priority. If more than one channel has the highest priority, the grant is made according to the secondary rule (applied to the channels with that priority).

If the priority check is off, only the secondary rule is used to arbitrate the grant.

- Secondary rule.

  The secondary rule is determined by the *arbitration_rule* parameter. This rule applies to the channels with the highest priority if the priority check is on and to all channels if the priority check is off. If *arbitration_rule* is 0, no secondary rule is assumed (if the priority check is on and multiple channels have the highest priority, any of them can receive the grant). If the priority check is off, no arbitration scheme checks are performed.

  If *arbitration_rule* is not 0, the secondary rule is one of the following:

  - Fairness or round-robin rule (*arbitration_rule* is 1).

    Grant is not issued to a (high-priority) channel that has received a grant while another channel's request is pending.

  - First-in first-out (FIFO) rule (*arbitration_rule* is 2).

    Grant is issued to a (high-priority) channel with the longest pending request.

  - Least-recently used (LRU) rule (*arbitration_rule* is 3).

    Grant is issued to a (high-priority) channel whose previous grant was issued the longest time before the current cycle.

Assertion Checks

`GNT_ONLY_IF_REQ`	Grant was issued without a request. *Gnt* bit was TRUE, but the corresponding *req* bit was not TRUE or transitioning from TRUE.
`ONE_CYCLE_GNT`	Grant was asserted for longer than 1 cycle. Grant was TRUE for 2 cycles in response to only one request.
`GNT_IN_WINDOW`	Grant was not issued within the specified time window. Grant was issued before *min_cks* cycles or no grant was issued by *max_cks* cycles.
`HIGHEST_PRIORITY`	Grant was issued for a request other than the highest priority request. *priority_check* = 1 Grant was issued, but another pending request had higher priority than all the requests that received grants.
`FAIRNESS`	Two grants were issued to the same channel while another channel's request was pending. *arbitration_rule* = 1 Two grants were issued to a channel while a request from another channel was pending (violating the fairness rule).

FIFO   Grant was issued for a request that was not the
                            longest pending request.
                            arbitration_rule = 2
                            Grant was issued, but one or more other (high priority)
                            requests were pending longer than the granted request
                            (violating the FIFO rule).

LRU   Grant was issued to a channel that was more-recently
                            used than another channel with a pending request.
                            arbitration_rule = 3
                            Grant was issued, but another channel with a pending (high
                            priority) request received its previous grant before the granted
                            channel received its previous grant (violating the fairness
                            rule).

SINGLE_GRANT   Multiple grants were issued in the same clock cycle.
                            single_gnt_check = 1
                            More than one *gnts* bit was TRUE in the same clock cycle.

Implicit X/Z Checks

reqs contains X or Z   Requests contained X or Z bits.

grants contains X or Z   Grants contained X or Z bits.

priorities contains X   Priorities contained X or Z bits.
or Z

Cover Points

cover_req_granted   BASIC — Number of granted requests for each channel.

cover_req_aborted   BASIC — Number of aborted requests for each channel.

cover_req_granted_at_   CORNER — Number of times grant was issued *min_cks* cycles
min_cks   after its request was asserted.

cover_req_granted_at_   CORNER — Number of times grant was issued *max_cks* cycles
max_cks   after its request was asserted.

time_to_grant   STATISTIC — Reports the number of requests granted at each
                            cycle in the time window.

concurrent_requests   STATISTIC — Reports for each channel, the number of times
                            each other channel had requests concurrent with that channel.

See also

ovl_bits

Ensures that the number of asserted (or deasserted) bits of the value of an expression is within a specified range.

fire['OVL_FIRE_WIDTH-1:0]

ovl_bits

test_expr[*width*-1:0]

clock reset enable

Parameters:

severity_level	*msg*
width	*coverage_level*
asserted	*clock_edge*
min	*oreset_polarity*
max	*gating_type*
property_type	

Class: 1-cycle assertion
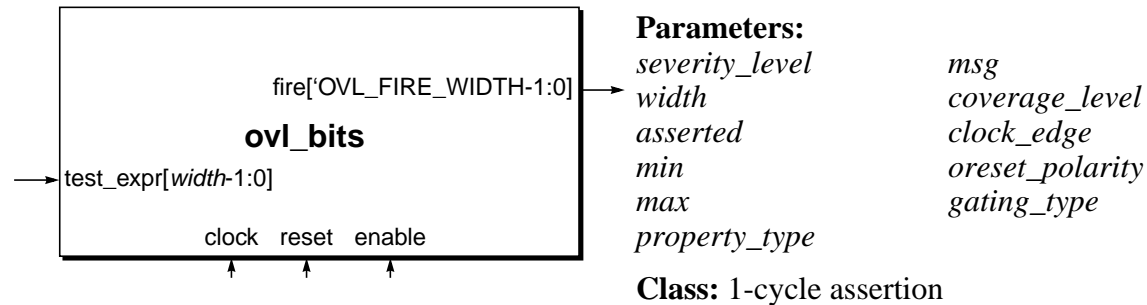
Syntax

```
ovl_bits
    [#(severity_level, width, asserted, min, max, property_type, msg,
        coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.

asserted	Whether to count asserted or deasserted bits.
	asserted = 0
	    Counts FALSE (deasserted) bits.
	asserted = 1 (Default)
	    Counts TRUE (asserted) bits.

min	Whether or not to perform min checks. Default: 1.
	min = 0
	    Turns off the min check.
	min ≥ 1
	    Minimum number of bits in *test_expr* that should be asserted (or deasserted).

max	Whether or not to perform max checks. Default: 1.
	max = 0
	    Turns off the max check.
	max ≥ 1
	    Maximum number of bits in *test_expr* that should be asserted (or deasserted). *Max* must be ≥ *min*.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).

msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Variable or expression to check.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_bits checker checks the multiple-bit expression *test_expr* at each active edge of *clock* and counts the number of TRUE bits (if *assert* is 1) or FALSE bits (if *assert* is 0). If the count is < *min* a min violation occurs and if the count is > *max*, a max violation occurs.

Assertion Checks

MIN	Fewer than 'min' bits were asserted.
	  min > 0 and *asserted* = 1
	  The number of TRUE bits in the value of *test_expr* was less than the minimum specified by *min*.
	Fewer than 'min' bits were deasserted.
	  min > 0 and *asserted* = 0
	  The number of FALSE bits in the value of *test_expr* was less than the minimum specified by *min*.

MAX
: More than 'max' bits were asserted.
  max > 0 and *asserted* = 1
    The number of TRUE bits in the value of *test_expr* was more than the maximum specified by *max*.
  More than 'max' bits were deasserted.
  max > 0 and *asserted* = 0
    The number of FALSE bits in the value of *test_expr* was more than the maximum specified by *max*.

Implicit X/Z Checks

test_expr contains X or Z
: Expression contained X or Z bits.

Cover Points

cover_values_checked
: SANITY — Number of cycles *test_expr* changed value.

cover_bits_within_limit
: BASIC — Number of cycles the number of counted *test_expr* bits was in range.

cover_bits_at_min
: CORNER — Number of cycles the number of counted *test_expr* bits was *min*.

cover_bits_at_max
: CORNER — Number of cycles the number of counted *test_expr* bits was *max*.

ovl_change

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

fire['OVL_FIRE_WIDTH-1:0]

start_event

ovl_change

test_expr[*width*-1:0]

clock   reset   enable

Parameters:

severity_level	*msg*
width	*coverage_level*
num_cks	*clock_edge*
action_on_new_start	*reset_polarity*
property_type	*gating_type*

Class: 2-cycle assertion
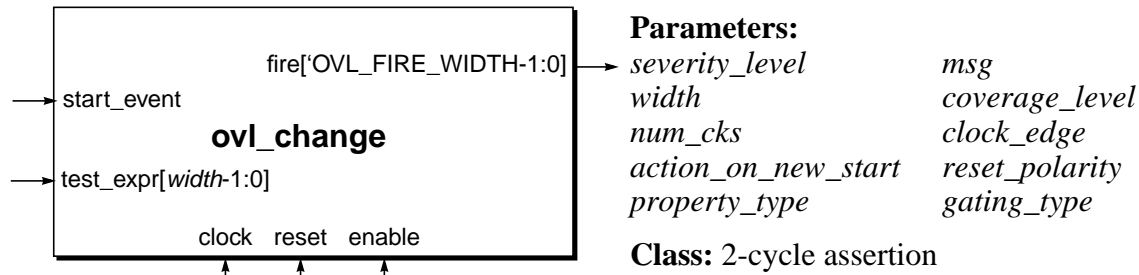
Syntax

```
ovl_change
    [#(severity_level, width, num_cks, action_on_new_start,
       property_type, msg, coverage_level, clock_edge, reset_polarity,
       gating_type)]
  instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_change assertion checker checks the expression *start_event* at each active edge of *clock* 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* active edges of *clock*. 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

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

ovl_time   ovl_win_unchange
ovl_unchange   ovl_window
ovl_win_change

Examples

```
ovl_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_DEFAULT,                          // coverage_level
    'OVL_POSEDGE,                                // clock_edge
    'OVL_ACTIVE_LOW,                             // reset_polarity
    'OVL_GATE_CLOCK)                             // gating_type

    valid_sync_out (

        clock,                                  // clock
        reset,                                  // reset
        enable,                                 // enable
        sync == 1,                              // start_event
        out,                                    // test_expr
        fire_valid_sync_out);                   // fire
```

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

CHANGE Error: invalid synchronization

```
ovl_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_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK )                        // gating_type

    valid_sync_out (

       clock,                               // clock
       reset,                               // reset
       enable,                              // enable
       sync == 1,                           // start_event
       out,                                 // test_expr
       fire_valid_sync_out);                // fire
```

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

CHANGE Error: invalid synchronization

```
ovl_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_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK )                        // gating_type

    valid_sync_out (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        sync == 1,                           // start_event
        out,                                 // test_expr
        fire_valid_sync_out );               // fire
```

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).

CHANGE Error: invalid synchronization   illegal start event
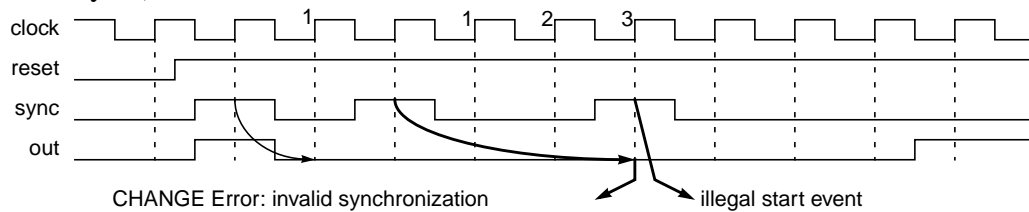
ovl_code_distance

Ensures that when an expression changes value, the number of bits in the new value that are different from the bits in the value of a second expression is within a specified range.

```
                    fire['OVL_FIRE_WIDTH-1:0]

test_expr1[width-1:0]
            ovl_code_distance

test_expr2[width-1:0]

            clock   reset   enable
```

Parameters:

severity_level	*msg*
width	*coverage_level*
min	*clock_edge*
max	*reset_polarity*
property_type	*gating_type*

Class: 1-cycle assertion
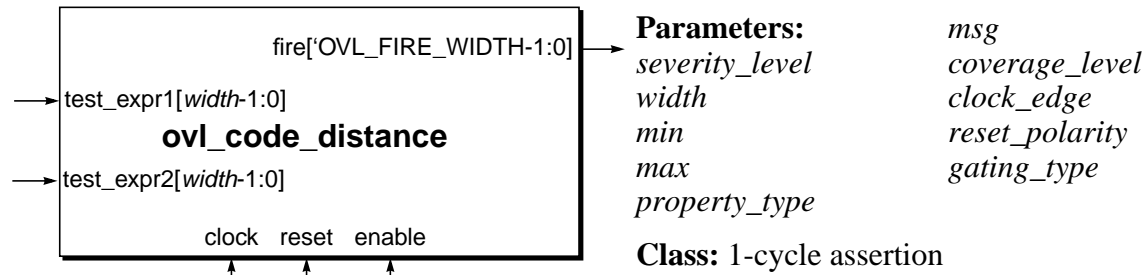
Syntax

```
ovl_code_distance
      [#(severity_level, width, min, max, property_type, msg,
         coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, test_expr1, test_expr2, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr* and *test_expr2*. Default: 1.
min	Minimum code distance. Default: 1.
max	Maximum code distance. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr1[width-1:0]	Variable or expression to check when its value changes.
test_expr2[width-1:0]	Variable or expression from which the code distance from *test_expr1* is calculated.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_code_distance assertion checker checks the expression *test_expr1* at each active edge of *clock* to determine if *test_expr* has changed value. If so, the checker evaluates a second expression *test_expr2* and calculates the absolute value of the difference between the two values (called the *code distance*). If the code distance is < *min* or > *max*, the assertion fails and a code_distance violation occurs.

Assertion Checks

CODE_DISTANCE	Code distance was not within specified limits. Code distance from test_expr1 to test_expr2 is less than min or greater than max.

Implicit X/Z Checks

test_expr1 contains X or Z	Expression contained X or Z bits.
test_expr2 contains X or Z	Second expression contained X or Z bits.

Cover Points

cover_test_expr_ changes	SANITY — Number of cycles *test_expr1* changed value.
cover_code_distance_ within_limit	BASIC — Number of cycles *test_expr1* changed to a value whose code distance from *test_expr2* was in the range from *min* to *max*.
observed_code_ distance	BASIC — Reports the code distances that occurred at least once.

`cover_code_distance_at_min`	CORNER — Number of cycles *test_expr1* changed to a value whose code distance from *test_expr2* was *min*.
`cover_code_distance_at_max`	CORNER — Number of cycles *test_expr1* changed to a value whose code distance from *test_expr2* was *max*.

ovl_cycle_sequence

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

```
                                          Parameters:          msg
        fire['OVL_FIRE_WIDTH-1:0]         severity_level       coverage_level
                                          num_cks              clock_edge
        ovl_cycle_sequence                necessary_condition  reset_polarity
                                          property_type        gating_type
   event_sequence[num_cks-1:0]
                                          Class: n-cycle assertion
        clock   reset   enable
```
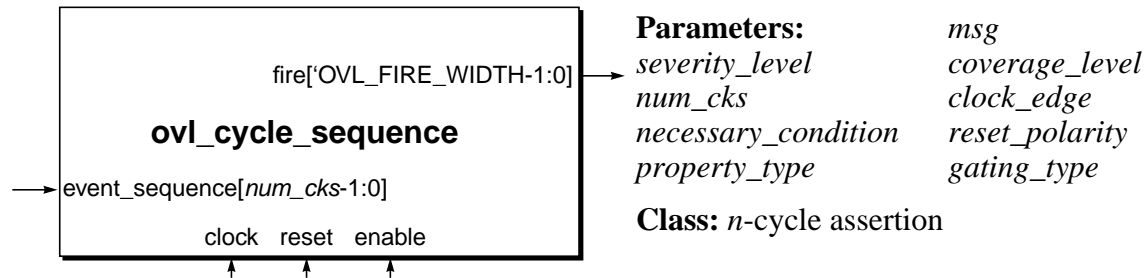
Syntax

```
ovl_cycle_sequence
        [#(severity_level, num_cks, necessary_condition, property_type,
           msg, coverage_level, clock_edge, reset_polarity, gating_type)]
    instance_name (clock, reset, enable, event_sequence, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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 (default), 'OVL_TRIGGER_ON_FIRST_PIPE and 'OVL_TRIGGER_ON_FIRST_NOPIPE.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).

`gating_type`	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`event_sequence [num_cks-1:0]`	Expression that is a concatenation where each bit represents an event.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

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

        1*bbb* —> *b*1*bb* —> *bb*1*b* —> *bbb*1

This series corresponds to the following series of events on successive active edges of *clock*:

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:

        1*bbb* —> *b*1*bb* —> 1*b*1*b* —> *b*1*b*1 —> *bb*1*b* —> *bbb*1

This series corresponds to the following sequences of events on successive active edges of *clock*:

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 active edges of *clock*. When this condition occurs, the checker verifies that the value of *event_sequence*[0] is 1 at the next active edge of *clock*. 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.

- 'OVL_TRIGGER_ON_FIRST_PIPE

  The necessary condition is that the *event_sequence* [*num_cks* -1] bit is sampled equal to 1 on an active edge of *clock*. 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 active edges of *clock*. If not, the assertion fails and the checker cancels the current check of subsequent events in the sequence.

  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 an active edge of *clock*. 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 active edges of *clock*. If not, the assertion fails and the checker cancels the current check of subsequent events in the sequence.

  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

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

ovl_change   ovl_unchange

Examples

```
ovl_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_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK )                           // gating_type

  valid_write_sequence (

    clock,                                      // clock
    reset,                                      // reset
    enable,                                     // enable
    { r_opcode =='WR,                           // event_sequence
    r_opcode =='WAIT,
    (r_opcode == 'WR) ||
    (r_opcode =='DONE)},
    fire_valid_write_sequence );                // fire
```

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

CYCLE_SEQUENCE Error: invalid WR sequence

```
ovl_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_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK )                           // gating_type

  valid_write_sequence (

    clock,                                      // clock
    reset,                                      // reset
    enable,                                     // enable
    { r_opcode == 'WR,                          // event_sequence
    (r_opcode == 'WAIT) ||
    (r_opcode == 'WR),
    (r_opcode == 'WAIT) ||
    (r_opcode == 'DONE)},
    fire_valid_write_sequence );                // fire
```

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.

CYCLE_SEQUENCE Error: invalid WR sequence

```
ovl_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_DEFAULT,                        // coverage_level
   'OVL_POSEDGE,                              // clock_edge
   'OVL_ACTIVE_LOW,                           // reset_polarity
   'OVL_GATE_CLOCK)                           // gating_type

   valid_write_sequence (

      clock,                                  // clock
      reset,                                  // reset
      enable,                                 // enable
      { r_opcode == 'WR,                      // event_sequence
      (r_opcode == 'WAIT) ||
      (r_opcode == 'WR),
      (r_opcode == 'DONE)},
      fire_valid_write_sequence );            // fire
```

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.

CYCLE_SEQUENCE Error: invalid WR sequence
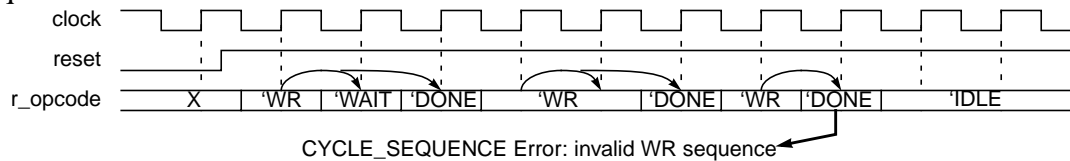
ovl_decrement

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

```
                              ┌─────────────────────────────┐
              fire['OVL_FIRE_WIDTH-1:0] ──→
                              │     ovl_decrement           │
     ──→  test_expr[width-1:0]│                             │
                              │   clock   reset   enable    │
                              └─────↑───────↑───────↑───────┘
```

Parameters: *msg*
severity_level *coverage_level*
width *clock_edge*
value *reset_polarity*
property_type *gating_type*

Class: 2-cycle assertion
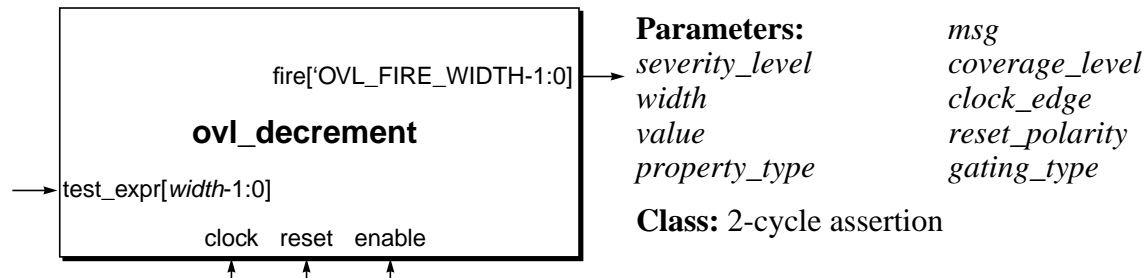
Syntax

```
ovl_decrement
      [#(severity_level, width, value, property_type, msg, coverage_level,
         clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should decrement by *value* whenever its value changes from the active edge of *clock* to the next active edge of clock.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_decrement assertion checker checks the expression *test_expr* at each active edge of *clock* to determine if its value has changed from its value at the previous active edge of *clock*. 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 ovl_delta checker.

Assertion Checks

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 *clock* after *reset* deasserts.

See also

ovl_delta   ovl_no_underflow
ovl_increment

Example

```
ovl_decrement #(

    'OVL_ERROR,                              // severity_level
    4,                                       // width
    1,                                       // value
    'OVL_ASSERT,                             // property_type
    "Error: invalid binary decrement",       // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_count (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        count,                               // test_expr
        fire_valid_count );                  // fire
```

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.

DECREMENT Error: invalid binary decrement
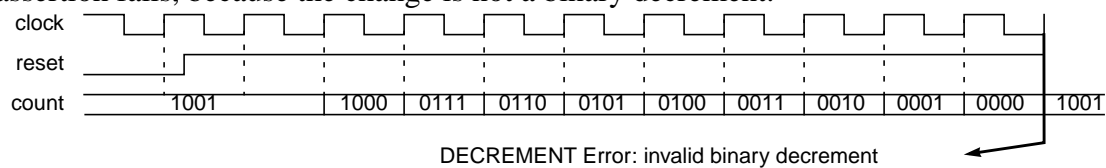
ovl_delta

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

```
              fire['OVL_FIRE_WIDTH-1:0]  →

                    ovl_delta

→  test_expr[width-1:0]

              clock   reset   enable
                ↑       ↑       ↑
```

Parameters:

severity_level	*msg*
width	*coverage_level*
min	*clock_edge*
max	*reset_polarity*
property_type	*gating_type*

Class: 2-cycle assertion

Syntax

```
ovl_delta
    [#(severity_level, width, min, max, property_type, msg,
       coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should only change by a delta value in the range min to max.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_delta assertion checker checks the expression *test_expr* at each active edge of *clock* to determine if its value has changed from its value at the previous active edge of *clock*. 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, ovl_delta can check for underflow and overflow. In datapath and arithmetic circuits, ovl_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

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 *clock* after *reset* 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

ovl_decrement   ovl_no_underflow
ovl_increment   ovl_range
ovl_no_overflow

Example

```
ovl_delta #(

    `OVL_ERROR,                          // severity_level
    16,                                  // width
    0,                                   // min
    8,                                   // max
    `OVL_ASSERT,                         // property_type
    "Error: y values not smooth",        // msg
    `OVL_COVER_DEFAULT,                  // coverage_level
    `OVL_POSEDGE,                        // clock_edge
    `OVL_ACTIVE_LOW,                     // reset_polarity
    `OVL_GATE_CLOCK)                     // gating_type

    valid_smooth (

        clock,                           // clock
        reset,                           // reset
        enable,                          // enable
        y,                               // test_expr
        fire_valid_smooth );             // fire
```

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

DELTA Error: y values not smooth
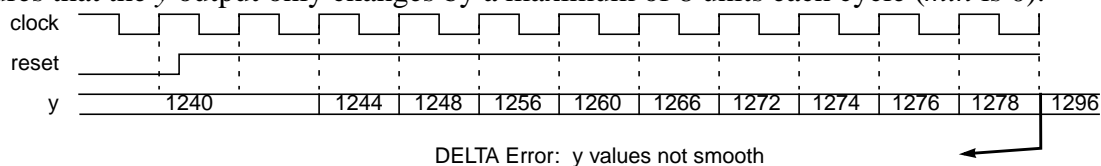
ovl_even_parity

Ensures that the value of an expression has even parity.

```
                    fire['OVL_FIRE_WIDTH-1:0]
              ovl_even_parity

      test_expr[width-1:0]

              clock   reset   enable
```

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
property_type	*reset_polarity*
msg	*gating_type*

Class: 1-cycle assertion
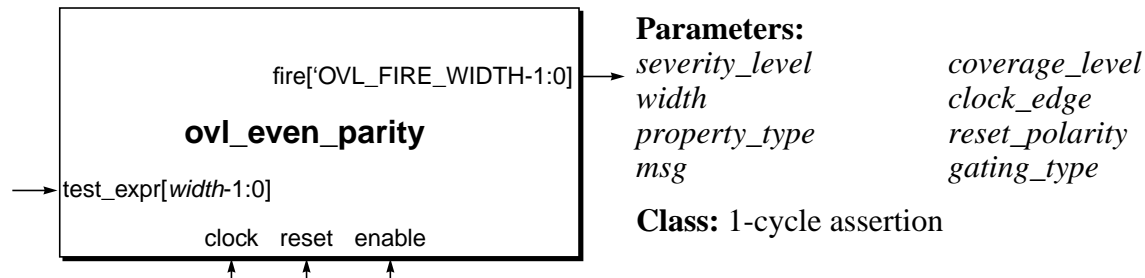
Syntax

```
ovl_even_parity
      [#(severity_level, width, property_type, msg, coverage_level,
         clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should evaluate to a value with even parity on the rising clock edge.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_even_parity assertion checker checks the expression *test_expr* at each active edge of *clock* 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

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

ovl_odd_parity

Example

```
ovl_even_parity #(

    'OVL_ERROR,                                // severity_level
    8,                                         // width
    'OVL_ASSERT,                               // property_type
    "Error: data has odd parity",              // msg
    'OVL_COVER_DEFAULT,                        // coverage_level
    'OVL_POSEDGE,                              // clock_edge
    'OVL_ACTIVE_LOW,                           // reset_polarity
    'OVL_GATE_CLOCK)                           // gating_type

    valid_data_even_parity (

        clock,                                 // clock
        reset,                                 // reset
        enable,                                // enable
        data,                                  // test_expr
        fire_valid_data_even_parity );         // fire
```
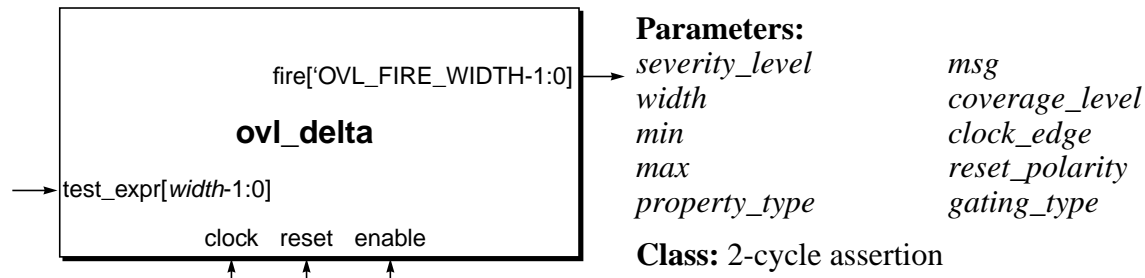
Ensures that *data* has even parity at each rising edge of *clock*.

EVEN_PARITY
Error: data has odd parity

---
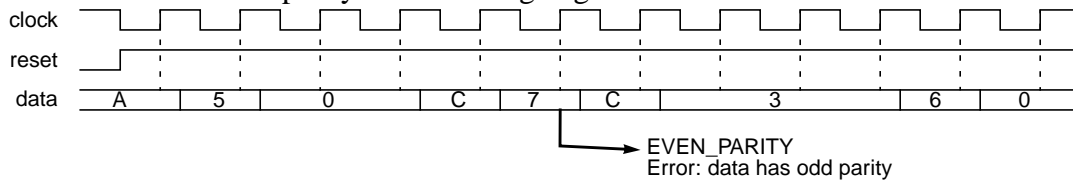
ovl_fifo

Ensures the data integrity of a FIFO and ensures that the FIFO does not overflow or underflow.

```
     →  enq
     →  deq
     →  full          fire ['OVL_FIRE_WIDTH-1:0] →
     →  empty
                     ovl_fifo
     →  enq_data[width-1:0]
     →  deq_data[width-1:0]
     →  preload[preload_count*width-1:0]*
            clock  reset  enable
               ↑      ↑      ↑
```

Parameters:

severity_level	*high_water_mark*
width	*value_check*
depth	*property_type*
pass_thru	*msg*
registered	*coverage_level*
enq_latency	*clock_edge*
deq_latency	*reset_polarity*
preload_count	*gating_type*
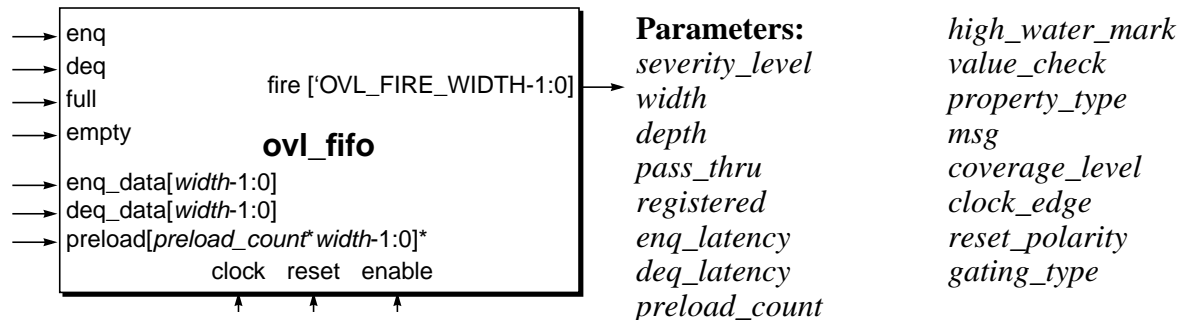
*if `preload_count` = 0:
   preload is `width` bits wide

Class: event-bounded assertion

Syntax

```
ovl_fifo
      [#(severity_level, width, depth, pass_thru, registered, enq_latency,
         deq_latency, preload_count, high_water_mark, value_check,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, enq, deq, full, empty, enq_data, \
      deq_data, preload, fire);
```

Parameters

`severity_level`	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
`width`	Width of a data item. Default: 1.
`depth`	FIFO depth. The *depth* must be > 0. Default: 2.
`pass_thru`	How the FIFO handles a dequeue and enqueue in the same cycle if the FIFO is empty.

`pass_thru` = 0 (Default)

No pass-through mode. Simultaneous dequeue/enqueue of an empty FIFO is an dequeue violation.

`pass_thru` = 1

Pass-through mode. Enqueue happens before the dequeue. Simultaneous enqueue/dequeue of an empty FIFO is not a dequeue violation.

registered	How the FIFO handles an enqueue and dequeue in the same cycle if the FIFO is full.
	
	registered = 0 (Default)
	    No registered mode. Simultaneous enqueue/dequeue of a full FIFO is an enqueue violation.
	
	registered = 1
	    Registered mode. Dequeue happens before the enqueue. Simultaneous enqueue/dequeue of a full FIFO is not an enqueue violation.

enq_latency	Latency for enqueue data.
	
	enq_latency = 0 (Default)
	    Checks and coverage assume *enq_data* is valid and the enqueue operation is performed in the same cycle *enq* asserts.
	
	enq_latency > 0
	    Checks and coverage assume *enq_data* is valid and the enqueue operation is performed *enq_latency* cycles after *enq* asserts.

deq_latency	Latency for dequeued data.
	
	deq_latency = 0 (Default)
	    Checks and coverage assume *deq_data* is valid and the dequeue operation is performed in the same cycle *deq* asserts.
	
	deq_latency > 0
	    Checks and coverage assume *deq_data* is valid and the dequeue operation is performed *deq_latency* cycles after *deq* asserts.

preload_count	Number of items to preload the FIFO on reset. The preload port is a concatenated list of items to be preloaded into the FIFO. Default: 0 (FIFO empty on reset).
high_water_mark	FIFO high-water mark. Must be < *depth*. A value of 0 disables the high-water mark cover point. Default: 0.

value_check	Whether or not to perform value checks.
	
	value_check = 0 (Default)
	    Turns off the value check.
	
	value_check = 1
	    Turns on the value check.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
enq	FIFO enqueue input. When *enq* asserts, the FIFO performs an enqueue operation. A data item is enqueued onto the FIFO and the FIFO counter increments by 1. If *enq_latency* is 0, the enqueue is performed in the same cycle *enq* asserts. Otherwise, the enqueue and counter increment occur *enq_latency* cycles later.
deq	FIFO dequeue input. When *deq* asserts, the FIFO performs a dequeue operation. A data item is dequeued from the FIFO and the FIFO counter decrements by 1. If *deq_latency* is 0, the dequeue is performed in the same cycle *deq* asserts. Otherwise, the dequeue and counter decrement occur *deq_latency* cycles later.

full	Output status flag from the FIFO.
	full = 0
	    FIFO not full.
	full = 1
	    FIFO full.

empty	Output status flag from the FIFO.
	empty = 0
	    FIFO not empty.
	empty = 1
	    FIFO empty.

enq_data[width-1:0]	Enqueue data input to the FIFO. Contains the data item to enqueue in that cycle (if *enq_latency* = 0) or to enqueue in the cycle *enq_latency* cycles later (if *enq_latency* > 0).
deq_data[width-1:0]	Dequeue data output from the FIFO. Contains the dequeued data item in that cycle (if *deq_latency* = 0) or in the cycle *enq_latency* cycles later (if *enq_latency* > 0).

preload
[*preload_count*width*-1
:0]
	Concatenated preload data to enqueue on reset.
	preload_count = 0
	  No preload of the FIFO is assumed. The width of preload should be *width*, however no values from *preload* are used. The FIFO is assumed to be empty on reset.
	preload_count > 0
	  Checker assumes the value of *preload* is a concatenated list of items that were all enqueued on the FIFO on reset (or simulation start). The width of preload should be *preload_count * width* (preload items are the same width). Preload values are enqueued from the low order item to the high order item.

fire [`OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_fifo assertion checker ensures a FIFO functions legally. A FIFO is a memory structure that stores and retrieves data items based on a first-in first-out queueing protocol. The FIFO has configured properties specified as parameters to the ovl_fifo checker: width of the data items (*width*), capacity of the FIFO (*depth*), and the high-water mark that identifies the point at which the FIFO is almost full (*high_water_mark*). Control and data signals to and from the FIFO are connected to the ovl_fifo checker.

The checker checks *enq* and *deq* at the active edge of *clock* each cycle the checker is active. If *enq* is TRUE, the FIFO is enqueuing a data item onto the FIFO. If *deq* is TRUE, the FIFO is in the process of dequeuing a data item. Both enqueue and dequeue operations can each take more than one cycle. If the *enq_latency* parameter is defined > 0, then *enq_data* is ready *enq_latency* clock cycles after the *enq* signal asserts. Similarly, if the *deq_latency* parameter is defined > 0, then *deq_data* is ready *deq_latency* clock cycles after the *deq* signal asserts. All assertion checks and coverage are based on enqueue/dequeue data after the latency periods.

The checker ensures the FIFO does not enqueue an item when it is supposed to be full (enqueue check) and the FIFO does not dequeue an item when it is supposed to be empty (dequeue check). The checker also ensures that the FIFO's *full* and *empty* status flags operate correctly (full and empty checks). The checker also can verify the data integrity of dequeued FIFO data (value check).

The checker also can be configured to handle other FIFO characteristics such as preloading items on reset and allowing pass-through operations and registered enqueue/dequeues.

Assertion Checks

ENQUEUE
Enqueue occurred that would overflow the FIFO.
registered = 0
> *Enq* was TRUE, but *enq_latency* cycles later, FIFO contained *depth* items.

registered = 1
> *Enq* was TRUE, but *enq_latency* cycles later, FIFO contained *depth* items and no item was to be dequeued that cycle.

DEQUEUE
Dequeue occurred that would underflow the FIFO.
pass_thru = 0
> *Deq* was TRUE, but *deq_latency* cycles later, FIFO contained no items.

pass_thru = 1
> *Deq* was TRUE, but *enq_latency* cycles later, FIFO contained no items and no item was to be enqueued that cycle.

FULL
FIFO 'full' signal asserted or deasserted in the wrong cycle.
> FIFO contained fewer than *depth* items but *full* was TRUE or FIFO contained *depth* items but *full* was FALSE.

EMPTY
FIFO 'empty' signal asserted or deasserted in the wrong cycle.
> FIFO contained one or more items but *empty* was TRUE or FIFO contained no items but *empty* was FALSE.

VALUE
Dequeued FIFO value did not equal the corresponding enqueued value.
deq_latency = 0
> *Deq* was TRUE, but *deq_data* did not equal the corresponding enqueued item.

deq_latency > 0
> *Deq* was TRUE, but *deq_latency* cycles later *deq_data* did not equal the corresponding enqueued item.

This check automatically turns off if an enqueue or dequeue check violation occurs since it is no longer possible to correspond enqueued with dequeued values. The check turns back on when the checker resets.

Implicit X/Z Checks

enq contains X or Z
Enqueue signal was X or Z.

deq contains X or Z
Dequeue signal was X or Z.

full contains X or Z
FIFO full signal was X or Z.

empty contains X or Z
FIFO empty signal was X or Z.

enq_data contains X or Z
Enqueue data expression contained X or Z bits.

`deq_data contains X or Z`	Dequeue data expression contained X or Z bits.

Cover Points

`cover_enqueues`	SANITY — Number of data items enqueued on the FIFO.
`cover_dequeues`	SANITY — Number of data items dequeued from the FIFO.
`cover_simultaneous_ enq_deq`	BASIC — Number of cycles *enq* and *deq* asserted together.
`cover_enq_followed_by_ deq`	BASIC — Number of times *enq* asserted, then deasserted in the next cycle and stayed deasserted until eventually *deq* asserted.
`cover_high_water_mark`	CORNER — Number of times the FIFO count transitioned from $< high\_water\_mark$ to $\geq high\_water\_mark$. Not reported if *high_water_mark* is 0.
`cover_simultaneous_ deq_enq_when_empty`	CORNER — Number of cycles the FIFO was enqueued and dequeued simultaneously when it was empty.
`cover_simultaneous_ deq_enq_when_full`	CORNER — Number of cycles the FIFO was enqueued and dequeued simultaneously when it was full.
`cover_fifo_empty`	CORNER —Number of cycles FIFO was empty after processing enqueues and dequeues for the cycle.
`cover_fifo_full`	CORNER — Number of cycles FIFO was full after processing enqueues and dequeues for the cycle.
`cover_observed_counts`	STATISTIC — Reports the FIFO counts that occurred at least once.

ovl_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.

```
                    fire['OVL_FIRE_WIDTH-1:0]
push[push_width-1:0]
                ovl_fifo_index

pop[pop_width-1:0]

        clock   reset   enable
```

Parameters: *property_type*
severity_level *msg*
depth *coverage_level*
push_width *clock_edge*
pop_width *reset_polarity*
simultaneous_push_pop *gating_type*

Class: *n*-cycle assertion
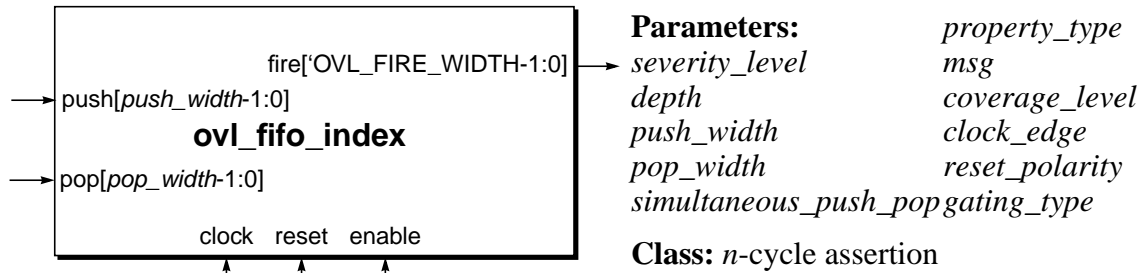
Syntax

```
ovl_fifo_index
    [#(severity_level, depth, push_width, pop_width,
       simultaneous_push_pop, property_type, msg, coverage_level,
       clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, push, pop, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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.
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).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_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 ovl_no_overflow and ovl_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

ovl_no_overflow   ovl_no_underflow

Examples

```
ovl_fifo_index #(

    'OVL_ERROR,                              // severity_level
    8,                                       // depth
    1,                                       // push_width
    1,                                       // pop_width
    1,                                       // simultaneous_push_pop
    'OVL_ASSERT,                             // property_type
    "Error",                                 // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type
                                             // simultaneous_push_pop

    no_over_underflow (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        push,                                // push
        pop,                                 // pop
        fire_fifo_no_over_underflow );       // fire
```

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).
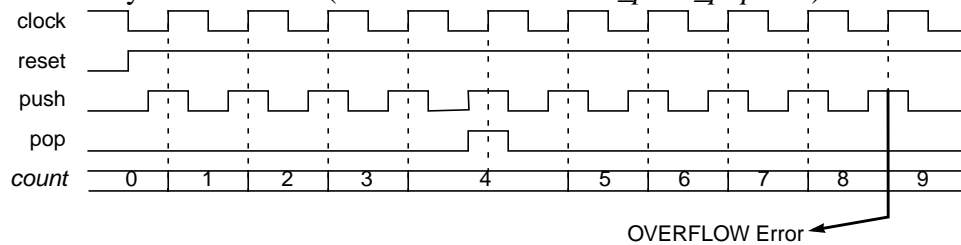
```
ovl_fifo_index #(

    'OVL_ERROR,                              // severity_level
    8,                                       // depth
    1,                                       // push_width
    1,                                       // pop_width
    0,                                       // simultaneous_push_pop
    'OVL_ASSERT,                             // property_type
    "violation",                             // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    no_over_underflow (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        push,                                // push
        pop,                                 // pop
        fire_fifo_no_over_underflow );       // fire
```

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

ILLEGAL PUSH AND POP Error
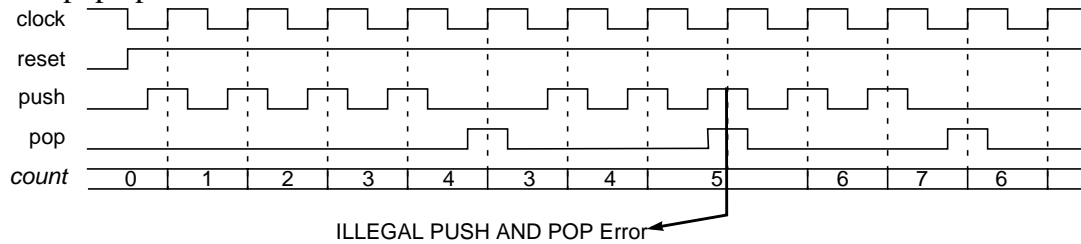
ovl_frame

Ensures that when a specified start event is TRUE, then an 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.

```
                    fire['OVL_FIRE_WIDTH-1:0]

  start_event

                    ovl_frame

  test_expr

            clock  reset  enable
```

Parameters:

severity_level	*msg*
min_cks	*coverage_level*
max_cks	*clock_edge*
action_on_new_start	*reset_polarity*
property_type	*gating_type*

Class: *n*-cycle assertion

Syntax

```
ovl_frame
    [#(severity_level, min_cks, max_cks, action_on_new_start,
       property_type, msg, coverage_level, clock_edge, reset_polarity,
       gating_type)]
  instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").

coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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).
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_frame assertion checker checks for a start event at each active edge of *clock.* 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 active edge of *clock* 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 active edge of *clock* 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

FRAME	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 active 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 an active edge of *clock*.

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

ovl_change   ovl_unchange
ovl_next   ovl_width
ovl_time

Examples

```
ovl_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_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK)                         // gating_type

    valid_transaction (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        req,                                 // start_event
        ack,                                 // test_expr
        fire_valid_transaction );            // fire
```
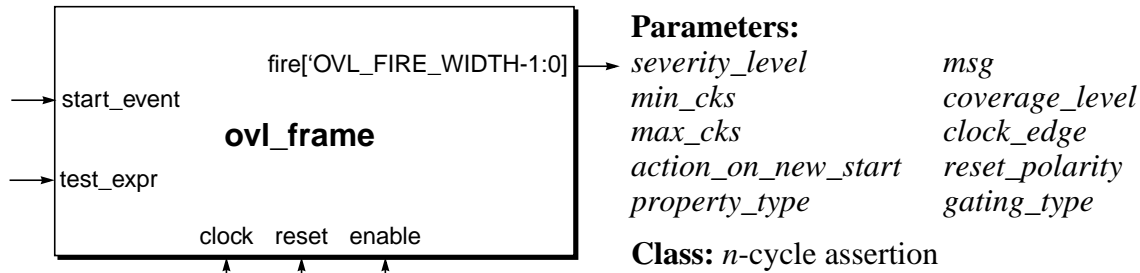
Ensures that after a rising edge of *req*, *ack* goes high between 2 and 4 cycles later. New start events during transactions are not considered to be new transactions and are ignored.

FRAME Error: invalid transaction
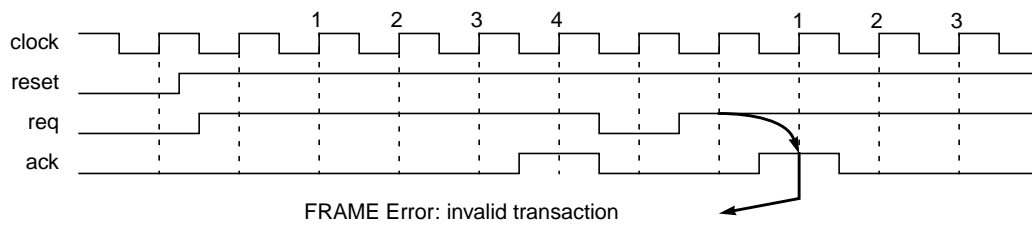
```
ovl_frame #(

    `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_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK )                        // gating_type

    valid_transaction (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        req,                                 // start_event
        ack,                                 // test_expr
        fire_valid_transaction );            // fire
```

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.

FRAME Error: invalid transaction
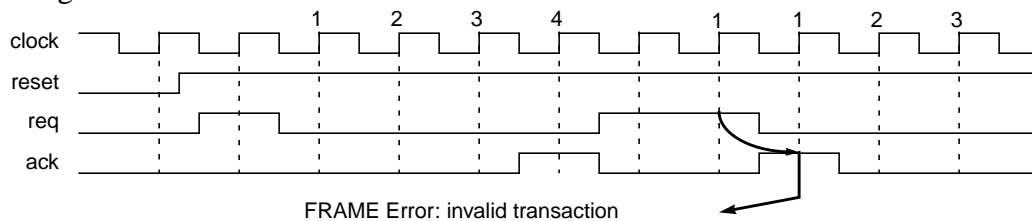
```
ovl_frame #(

   '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_DEFAULT,                         // coverage_level
   'OVL_POSEDGE,                               // clock_edge
   'OVL_ACTIVE_LOW,                            // reset_polarity
   'OVL_GATE_CLOCK)                            // gating_type

   valid_transaction (

      clock,                                   // clock
      reset,                                   // reset
      enable,                                  // enable
      req,                                     // start_event
      ack,                                     // test_expr
      fire_valid_transaction );                // fire
```

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.
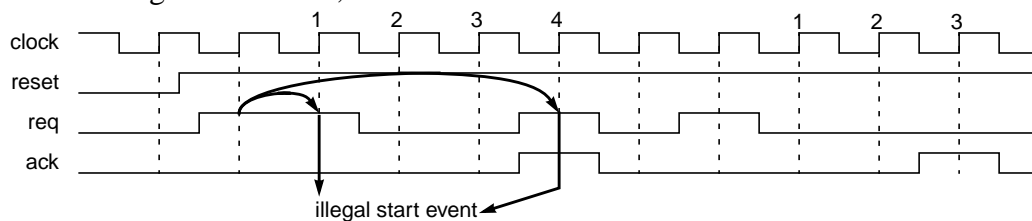
ovl_handshake

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

```
                  fire['OVL_FIRE_WIDTH-1:0]

  req

            ovl_handshake

  rack


         clock   reset   enable
```

Parameters:

severity_level	*property_type*
min_ack_cycle	*msg*
max_ack_cycle	*coverage_level*
req_drop	*clock_edge*
deassert_count	*reset_polarity*
max_ack_length	*gating_type*

Class: event-bounded assertion

Syntax

```
ovl_handshake
    [#(severity_level, min_ack_cycle, max_ack_cycle, req_drop,
        deassert_count, max_ack_length, property_type, msg,
        coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, req, ack, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).

clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
req	Expression that starts a transaction.
ack	Expression that indicates the transaction is complete.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_handshake assertion checker checks the single-bit expressions *req* and *ack* at each active edge of *clock* 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 active edge of *clock* and an acknowledge event (where *ack* transitions to TRUE) signals the transaction is complete on the active edge of *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

ovl_win_change   ovl_window
ovl_win_unchange

Examples

```
ovl_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_DEFAULT,                       // coverage_level
    'OVL_POSEDGE,                             // clock_edge
    'OVL_ACTIVE_LOW,                          // reset_polarity
    'OVL_GATE_CLOCK)                          // gating_type

    valid_hold_holda (

        clock,                                // clock
        reset,                                // reset
        enable,                               // enable
        hold,                                 // req
        holda,                                // ack
        fire_valid_hold_holda );              // fire
```
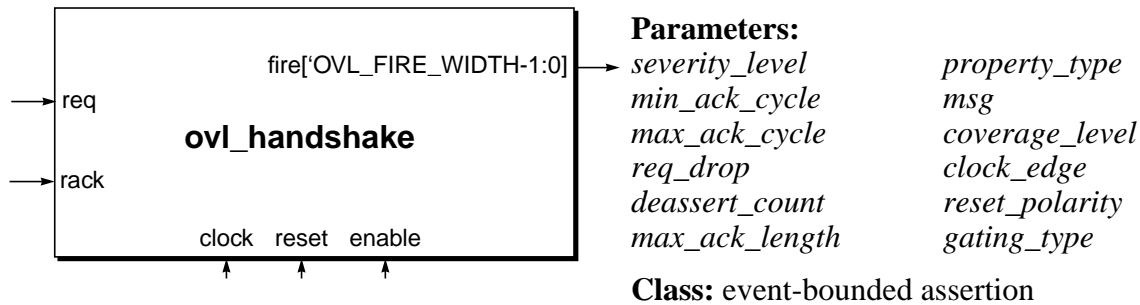
Ensures that multiple *hold* requests are not made while waiting for a *holda* acknowledge and that every *holda* acknowledge is in response to a unique *hold* request.

After a violation, checking is turned off until *holda* acknowledge is sampled deasserted.
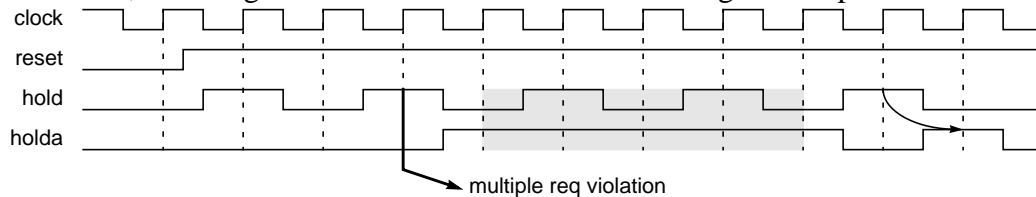
```
ovl_handshake #(

    '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_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_hold_holda (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        hold,                                // req
        holda,                               // ack
        fire_valid_hold_holda );             // fire
```

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.
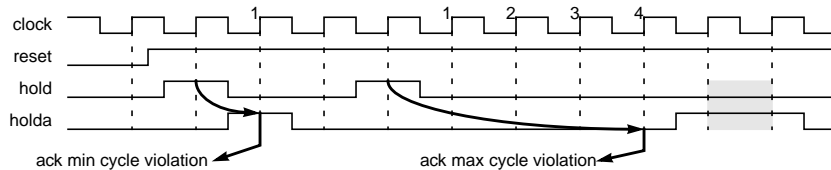
```
ovl_handshake #(

   '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_DEFAULT,                      // coverage_level
   'OVL_POSEDGE,                            // clock_edge
   'OVL_ACTIVE_LOW,                         // reset_polarity
   'OVL_GATE_CLOCK)                         // gating_type

   valid_hold_holda (

      clock,                               // clock
      reset,                               // reset
      enable,                              // enable
      hold,                                // req
      holda,                               // ack
      fire_valid_hold_holda );             // fire
```

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.

ack max length violation
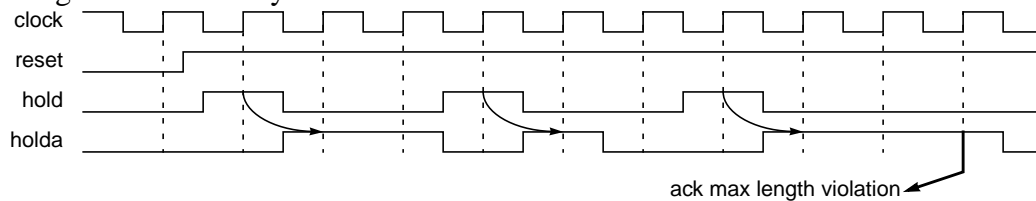
```
ovl_handshake #(

    `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_DEFAULT,                         // coverage_level
    `OVL_POSEDGE,                               // clock_edge
    `OVL_ACTIVE_LOW,                            // reset_polarity
    `OVL_GATE_CLOCK)                            // gating_type

    valid_hold_holda (

        clock,                                  // clock
        reset,                                  // reset
        enable,                                 // enable
        hold,                                   // req
        holda,                                  // ack
        fire_valid_hold_holda );                // fire
```

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.
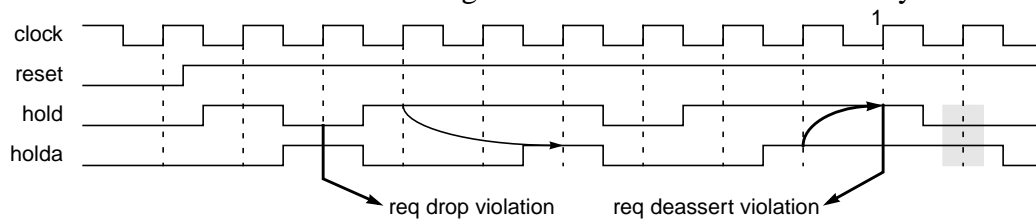
ovl_hold_value

Ensures that once an expression matches the value of a second expression, the first expression does not change value until a specified event window arrives and then changes value some time in that window.

```
                    fire ['OVL_FIRE_WIDTH-1:0]

test_expr[width-1:0]
              ovl_hold_value

value[width-1:0]

          clock  reset  enable
```

Parameters: *msg*
severity_level   *coverage_level*
width   *clock_edge*
min   *reset_polarity*
max   *gating_type*
property_type

Class: *n*-cycle assertion
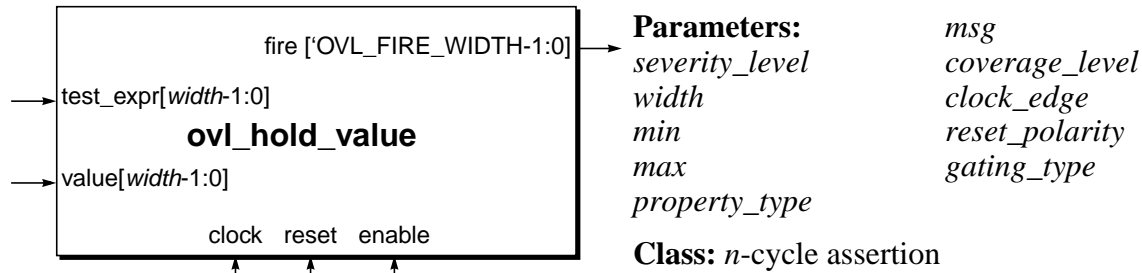
Syntax

```
ovl_hold_value
    [#(severity_level, width, min, max, property_type, msg,
       coverage_level, clock_edge, reset_polarity, gating_type)]
 instance_name (clock, reset, enable, test_expr, value, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr* and *value*. Default: 2.
min	Number of cycles after the value match that the event window opens. Default: 0 (*test_expr* can change value in any cycle).

max	Number of cycles after the value match that the event window closes. But if *max* = 0, no event window opens and there are the following special cases:
	min = 0 and *max* = 0
	  When *test_expr* and *value* match, *test_expr* must change value in the next cycle.
	min > 0 and *max* = 0
	  When *test_expr* and *value* match, *test_expr* must not change value in the next *min*-1 cycles.
	Default: 0.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).

clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Variable or expression to check.
value[width-1:0]	Value to match with *test_expr*.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_hold_value assertion checker checks *test_expr* and *value* at the active edge of *clock*. If *test_expr* has changed value and the values of *test_expr* and *value* match, the checker verifies that the value of *test_expr* holds as follows:

- `0 = min = max` (default)

  If the value of *test_expr* does not change in the next cycle, a hold_value violation occurs.

- `0 = min < max`

  If the value of *test_expr* has not changed within the next *max* cycles, a hold_value violation occurs.

- `0 < min ≤ max`

  If the value of *test_expr* changes before an event window opens *min* cycles later, a hold_value violation occurs. Then, if the value of *test_expr* changes, the event window closes. However if *test_expr* still has not changed value *max* cycles after the value match, the event window closes and a hold_value violation occurs.

- `0 = max < min`

    If the value of *test_expr* changes within the next *min*-1 cycles a hold_value violation occurs.

The checker returns to the state of checking *test_expr* and *value* in the next cycle.

Assertion Checks

`HOLD_VALUE`

A match occurred and the expression had the same value in the next cycle.
`0 = min = max`
After matching *value*, *test_expr* held the same value in the next cycle.

A match occurred and the expression held the same value for the next 'max' cycles.
`0 = min < max`
After matching *value*, *test_expr* held the same value for the next *max* cycles.

A match occurred and the expression changed value before the event window or held the same value through the event window.
`0 < min ≤ max`
After matching *value*, *test_expr* did not hold the same value for the next *min*-1 cycles or *test_expr* held the same value for the next *max* cycles.

A match occurred and the expression changed value before the event window opened.
`0 = max < min`
After matching *value*, *test_expr* did not hold the same value for the next *min*-1 cycles.

Implicit X/Z Checks

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

Cover Points

`cover_test_expr_changes`	SANITY — Number of cycles *test_expr* changed value.
`cover_hold_value_for_min_cks`	CORNER — Number of times *test_expr* held value for exactly *min* cycles.
`cover_hold_value_for_max_cks`	CORNER — Number of times *test_expr* held value for exactly *max*+1 cycles.

`cover_hold_value_for_max_cks`	CORNER — indicates that the test_expr was held exactly equal to value for specified max clocks. Not reported if *max* = 0 and *min* > 0.
`observed_hold_time`	STATISTIC — Reports the hold times (in cycles) that occurred at least once.

ovl_implication

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

```
                        fire['OVL_FIRE_WIDTH-1:0]

        antecedent_expr
                ovl_implication

        consequent_expr

            clock   reset   enable
```

Parameters: *coverage_level*
severity_level   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: 1-cycle assertion
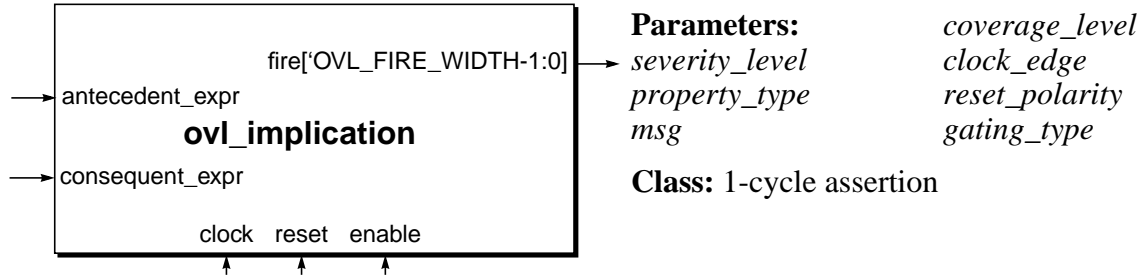
Syntax

```
ovl_implication
    [#(severity_level, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
  instance_name (clock, reset, enable, antecedent_expr, consequent_expr,
      fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.

`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`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.
`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_implication assertion checker checks the single-bit expression *antecedent_expr* at each active edge of *clock*. 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

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:

```
ovl_always
   [#(severity_level, property_type, msg, coverage_level, clock_edge,
      reset_polarity, gating_type)]
   instance_name (clock, reset, enable,
      (antecedent_expr  ? consequent_expr  : 1'b1 ), fire);
```

See also

ovl_always   ovl_never
ovl_always_on_edge   ovl_proposition

Example

```
ovl_implication #(

    'OVL_ERROR,                              // severity_level
    'OVL_ASSERT,                             // property_type
    "Error: q valid but q full",            // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    not_full (

        clock,                              // clock
        reset,                              // reset
        enable,                             // enable
        q_valid,                            // antecedent_expr
        q_not_full,                         // consequent_expr
        fire_not_full );                    // fire
```

Ensures that *q_not_full* is TRUE at each rising edge of *clock* for which *q_valid* is TRUE.
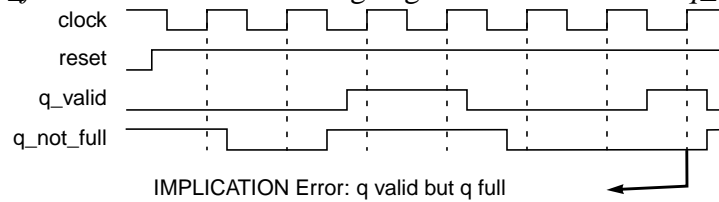
ovl_increment

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

```
                                          Parameters:          msg
      fire['OVL_FIRE_WIDTH-1:0]           severity_level       coverage_level
                                          width                clock_edge
         ovl_increment                    value                reset_polarity
                                          property_type        gating_type
  test_expr[width-1:0]
                                          Class: 2-cycle assertion
         clock  reset  enable
```
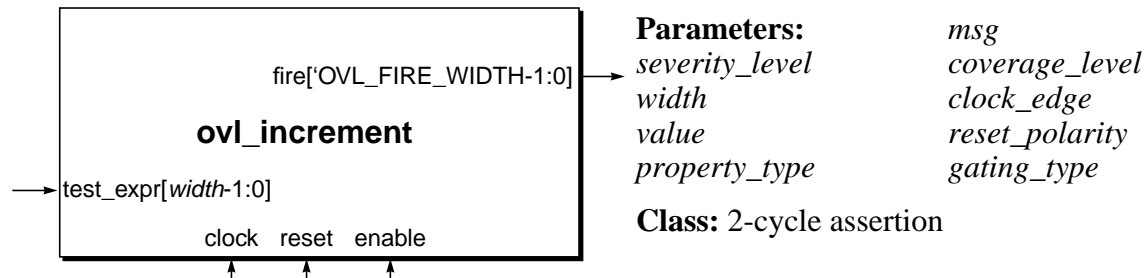
Syntax

```
ovl_increment
      [#(severity_level, width, value, property_type, msg, coverage_level,
          clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should increment by *value* whenever its value changes from the active edge of *clock* to the next active edge of *clock*.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_increment assertion checker checks the expression *test_expr* at each active edge of *clock* to determine if its value has changed from its value at the previous active edge of *clock*. 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 ovl_delta checker.

Assertion Checks

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 *clock* after *reset* deasserts.

See also

ovl_decrement   ovl_no_overflow
ovl_delta

Example

```
ovl_increment #(

    'OVL_ERROR,                              // severity_level
    4,                                       // width
    1,                                       // value
    'OVL_ASSERT,                             // property_type
    "Error: invalid binary increment",       // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_count (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        count,                               // test_expr
        fire_valid_count );                  // fire
```

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.

INCREMENT Error: invalid binary increment
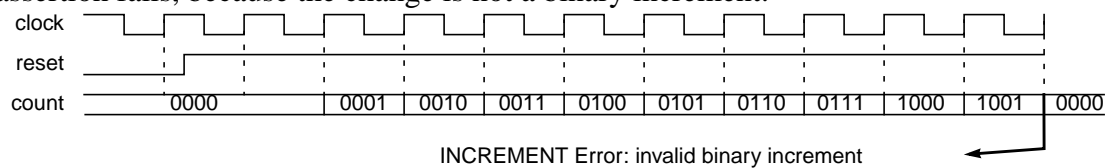
ovl_memory_async

Ensures the integrity of accesses to an asynchronous memory.

```
          ┌─────────────────────────────────────────┐
 ───▶ ren         fire ['OVL_FIRE_WIDTH-1:0] ───▶
 ───▶ raddr[addr_width-1:0]
 ───▶ rdata[data_width-1:0]
 ───▶ wen        ovl_memory_async
 ───▶ waddr[addr_width-1:0]
 ───▶ wdata[data_width-1:0]
 ───▶ start_addr[addr_width-1:0]
 ───▶ end_addr[addr_width-1:0]
                  reset   enable
          └─────────────────────────────────────────┘
                    ↑       ↑
```

Parameters:

severity_level	*value_check*
data_width	*property_type*
addr_width	*msg*
mem_size	*coverage_level*
addr_check	*ren_edge*
init_check	*wen_edge*
one_read_check	*reset_polarity*
one_write_check	*gating_type*

Class: event-bounded assertion

Syntax

```
ovl_memory_async
      [#(severity_level, data_width, addr_width, mem_size, addr_check,
         init_check, one_read_check, one_write_check, value_check,
         property_type, msg, coverage_level, ren_edge, wen_edge,
         reset_polarity, gating_type)]
   instance_name (reset, enable, start_addr, end_addr, ren, raddr, rdata,
      wen, waddr, wdata, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
data_width	Number of bits in a data item. Default: 1
addr_width	Number of bits in an address. Default: 1
mem_size	Number of data items in the memory. Default: 2
addr_check	Whether or not to perform address checks.

	addr_check = 0
	    Turns off the address check.
	addr_check = 1 (Default)
	    Turns on the address check.

init_check	Whether or not to perform initialization checks.

	init_check = 0
	    Turns off the initialization check.
	init_check = 1 (Default)
	    Turns on the initialization check.

one_read_check	Whether or not to perform one_read checks.
	one_read_check = 0 (Default)
	Turns off the one_read check
	one_read_check = 1
	Turns on the one_read check.

one_read_check Whether or not to perform one_read checks.
one_read_check = 0 (Default)
    Turns off the one_read check
one_read_check = 1
    Turns on the one_read check.

one_write_check Whether or not to perform one_write checks.
one_write_check = 0 (Default)
    Turns off the one_write check.
one_write_check = 1
    Turns on the one_write check.

value_check Whether or not to perform value checks.
value_check = 0 (Default)
    Turns off the value check.
value_check = 1
    Turns on the value check.

property_type Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).

msg Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").

coverage_level Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).

ren_edge Active edge of the *ren* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

wen_edge Active edge of the *wen* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).

gating_type Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

reset Synchronous reset signal indicating completed initialization.

enable Enable signal for *ren* and *wen,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.

start_addr First address of the memory.

end_addr Last address of the memory.

ren	Read enable input, whose active edge initiates a read operation from the memory location specified by *raddr*.
raddr	Read address input.
rdata	Read data input that holds the data item read from memory.
wen	Write enable input, whose active edge initiates a write operation of the data item in *wdata* to the memory location specified by *waddr*.
waddr	Write address input.
wdata	Write data input.
fire [`OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_memory_async checker checks the two memory access enable signals *wen* and *ren* combinationally. The active edges of these signals are specified the *wen_edge* and *ren_edge* parameters (and by *enable* if gating_type is 'OVL_GATE_CLOCK). At the active edge of *wen*, the values of *waddr*, *start_addr* and *end_addr* are checked. If *waddr* is not in the range [*start_addr*:*end_addr*], an address check violation occurs. Otherwise, a write operation to the location specified by *waddr* is assumed. Similarly, at the active edge of *ren*, the values of *raddr*, *start_addr* and *end_addr* are checked. If *raddr* is not in the range [*start_addr*:*end_addr*], an address check violation occurs. Otherwise, a read operation from the location specified by *raddr* is assumed. Also, if *raddr* is uninitialized (i.e., has not been written to previously or at the current time), then an initialization check violation occurs.

By default, the address and init checks are on, but can be turned off by setting the *addr_check* and *init_check* parameters to 0. Note that other checks are valid only if the addresses are valid, so it is recommended that *addr_check* be left at 1. The checker can be configured to perform the following additional checks:

- *one_write_check* = 1

  At the active edge of *wen*, if the previous access to the data at the address specified by *waddr* was a write or a simultaneous read/write to that address, a one_write check violation occurs, unless the current operation is a simultaneous read/write to that location.

- *one_read_check* = 1

  At the active edge of *ren*, if the previous access to the data at the address specified by *raddr* was a read (but not a simultaneous read/write to that address), a one_read check violation occurs.

- *value_check* = 1

At the active edge of *wen*, the current value of *wdata* is the value assumed to be written to the memory location specified by *waddr*. At the active edge of *ren*, if the value of *rdata* does not match the expected value last written to the address specified by *raddr*, a value check violation occurs.

Note that when active edges of *wen* and *ren* occur together, a simultaneous read/write operation is assumed. Here, the read is performed first (for example, if *raddr* = *waddr*).

Assertion Checks

ADDRESS	Write address was out of range. At an active edge of *wen, waddr* < *start_addr* or *waddr* > *end_addr*. Read address was out of range. At an active edge of *ren, raddr* < *start_addr* or *raddr* > *end_addr*.
INITIALIZATION	Read location was not initialized. At an active edge of *ren,* the memory location pointed to by *raddr* had not had data written to it since the last reset.
ONE_READ	Memory location had two read accesses without an intervening write access. *one_read_check* = 1 At an active edge of *ren,* the previous access to the memory location pointed to by *raddr* was another read.
ONE_WRITE	Memory location had two write accesses without an intervening read access. *one_read_check* = 1 At an active edge of *wen,* the previous access to the memory location pointed to by *waddr* was another write (and the current memory access is not a simultaneous read/write to that location).
VALUE	Data item read from a location did not match the data last written to that location. *value_check* = 1 At an active edge of *ren,* the value of *rdata* did not equal the expected value, which was the value of *wdata* when a write access to the memory location pointed to by the current value of *raddr* last occurred.

Implicit X/Z Checks

`start_addr contains X or Z`	Start address contained X or Z bits.
`end_addr contains X or Z`	End address contained X or Z bits.
`raddr contains X or Z`	Read address contained X or Z bits.
`rdata contains X or Z`	Read data contained X or Z bits.
`waddr contains X or Z`	Write address contained X or Z bits.
`wdata contains X or Z`	Write data contained X or Z bits.

Cover Points

`cover_reads`	SANITY — Number of read accesses.
`cover_writes`	SANITY — Number of write accesses.
`cover_write_then_read_ from_same_addr`	BASIC — Number of times a write access was followed by a read from the same address.
`cover_read_addr`	STATISTIC — Reports which addresses were read at least once.
`cover_write_addr`	STATISTIC — Reports which addresses were written at least once.
`cover_two_writes_ without_read`	STATISTIC — Number of times a memory location had two write accesses but no read access of the data item stored by the first write.
`cover_two_reads_ without_write`	STATISTIC — Number of times a memory location had two read accesses but no write access overwriting the data item read by the first read.
`cover_read_from_start_ addr`	CORNER — Number of read accesses to the location specified by *start_addr*.
`cover_write_to_start_ addr`	CORNER — Number of write accesses to the location specified by *start_addr*.
`cover_read_from_end_ addr`	CORNER — Number of read accesses to the location specified by *end_addr*.
`cover_write_to_end_ addr`	CORNER — Number of write accesses to the location specified by *end_addr*.
`cover_write_then_read_ from_start_addr`	CORNER — Number of times a write access to *start_addr* was followed by a read from *start_addr*.
`cover_write_then_read_ from_end_addr`	CORNER — Number of times a write access to *end_addr* was followed by a read from *end_addr*.

ovl_memory_sync

Ensures the integrity of accesses to a synchronous memory.

```
                   r_clock          w_clock
       ren                    fire ['OVL_FIRE_WIDTH-1:0]
       raddr[addr_width-1:0]
       rdata[data_width-1:0]
       wen          ovl_memory_sync
       waddr[addr_width-1:0]
       wdata[data_width-1:0]
       start_addr[addr_width-1:0]
       end_addr[addr_width-1:0]
                   reset    enable
```
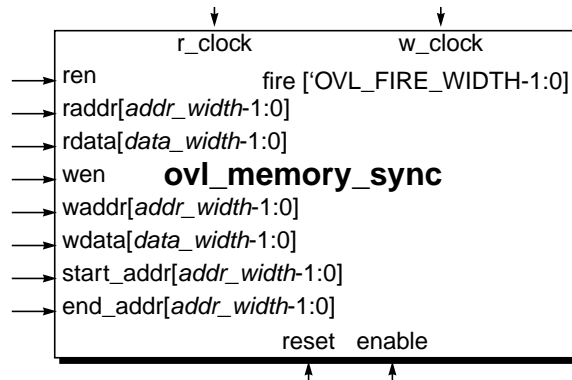
Parameters:
severity_level
data_width
addr_width
mem_size
pass_thru
addr_check
init_check
conflict_check
one_read_check

one_write_check
value_check
property_type
msg
coverage_level
w_clock_edge
r_clock_edge
reset_polarity
gating_type

Class: event-bounded assertion

Syntax

```
ovl_memory_sync
      [#(severity_level, data_width, addr_width, mem_size, pass_thru,
         addr_check, init_check, conflict_check, one_read_check,
         one_write_check, value_check, property_type, msg,
         coverage_level, r_clock_edge, w_clock_edge, reset_polarity,
         gating_type)]
   instance_name (r_clock, w_clock, reset, enable, start_addr, end_addr,
      ren, raddr, rdata, wen, waddr, wdata, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
data_width	Number of bits in a data item. Default: 1
addr_width	Number of bits in an address. Default: 1
mem_size	Number of data items in the memory. Default: 2
pass_thru	How the memory handles a simultaneous read and write access to the same address. This parameter applies to the initialization and value checks.

pass_thru = 0 (Default)

No pass-through mode (i.e., read before write). Simultaneous read/write access to the same location should return the current data item as the read data.

pass_thru = 1

Pass-through mode (i.e., write before read). Simultaneous read/write access to the same location should return the new data item as the read data. Only specify pass-through mode if *r_clock* === *w_clock* and *conflict_check* = 0.

addr_check	Whether or not to perform address checks.
	addr_check = 0
	Turns off the address check.
	addr_check = 1 (Default)
	Turns on the address check.
init_check	Whether or not to perform initialization checks.
	init_check = 0
	Turns off the initialization check.
	init_check = 1 (Default)
	Turns on the initialization check.
conflict_check	Whether or not to perform conflict checks.
	conflict_check = 0 (Default)
	Turns off the conflict check.
	conflict_check = 1
	Turns on the conflict check. Only select the conflict check if *r_clock* === *w_clock*.
one_read_check	Whether or not to perform one_read checks.
	one_read_check = 0 (Default)
	Turns off the one_read check.
	one_read_check = 1
	Turns on the one_read check.
one_write_check	Whether or not to perform one_write checks.
	one_write_check = 0 (Default)
	Turns off the one_write check.
	one_write_check = 1
	Turns on the one_write check.
value_check	Whether or not to perform value checks.
	value_check = 0 (Default)
	Turns off the value check.
	value_check = 1
	Turns on the value check.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
r_clock_edge	Active edge of the *r_clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
w_clock_edge	Active edge of the *w_clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).

Ports

r_clock	Clock event for read operations.
w_clock	Clock event for write operations.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *r_clock* and *w_clock,* if *gating_type =* 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type =* 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
start_addr	First address of the memory.
end_addr	Last address of the memory.
ren	Read enable input that initiates a read operation from the memory location specified by *raddr*.
raddr	Read address input.
rdata	Read data input that holds the data item read from memory.
wen	Write enable input that initiates a write operation of the data item in *wdata* to the memory location specified by *waddr*.
waddr	Write address input.
wdata	Write data input.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_memory_async checker checks *wen* at the active edge of *w_clock*. If *wen* is TRUE, the checker checks the values of *waddr*, *start_addr* and *end_addr*. If *waddr* is not in the range [*start_addr*:*end_addr*], an address check violation occurs. Otherwise, a write operation to the location specified by *waddr* is assumed. Similarly, the checker checks *ren* at the active edge of *r_clock*. If *ren* is TRUE, the checker checks the values of *raddr*, *start_addr* and *end_addr*. If *raddr* is not in the range [*start_addr*:*end_addr*], an address check violation occurs. Otherwise, a read operation from the location specified by *raddr* is assumed. Also, if *raddr* is uninitialized

(i.e., has not been written to previously or at the current time), then an initialization check violation occurs.

By default, the address and init checks are on, but can be turned off by setting the *addr_check* and *init_check* parameters to 0. Note that other checks are valid only if the addresses are valid, so it is recommended that *addr_check* be left at 1.

The checker can be configured to perform the following additional checks:

- `conflict_check = 1`

  At the active edges of *w_clock/r_clock*, if *wen* = *ren* = TRUE and *waddr* = *raddr*, then a conflict check violation occurs (*w_clock* and *r_clock* must be the same signal).

- `one_write_check = 1`

  `pass_thru = 0`

  At the active edge of *w_clock*, if *wen* is TRUE and the previous access to the data at the address specified by *waddr* was a write or a simultaneous read/write to that address, a one_write check violation occurs, unless the current operation is a simultaneous read/write to that location.

  `pass_thru = 1`

  At the active edge of *w_clock*, if *wen* is TRUE and the previous access to the data at the address specified by *waddr* was a write (but not a simultaneous read/write to that address), a one_write check violation occurs.

- `one_read_check = 1`

  `pass_thru = 0`

  At the active edge of *r_clock*, if *ren* is TRUE and the previous access to the data at the address specified by *raddr* was a read (but not a simultaneous read/write to that address), a one_read check violation occurs.

  `pass_thru = 1`

  At the active edge of *r_clock*, if *ren* is TRUE and the previous access to the data at the address specified by *raddr* was a read or a simultaneous read/write to that address, a one_read check violation occurs, unless the current operation is a simultaneous read/write to that location.

- `value_check = 1`

  At the active edge of *w_clock*, if *wen* is TRUE, the current value of *wdata* is the value assumed to be written to the memory location specified by *waddr*. At the active edge of *r_clock*, if *ren* is TRUE and the value of *rdata* does not match the expected value last written to the address specified by *raddr*, a value check violation occurs.

Assertion Checks

ADDRESS	Write address was out of range. At an active edge of *w_clock, wen* was TRUE but *waddr < start_addr* or *waddr > end_addr*. Read address was out of range. At an active edge of *r_clock, ren* was TRUE but *raddr < start_addr* or *raddr > end_addr*.
INITIALIZATION	Read location was not initialized. At an active edge of *r_clock, ren* was TRUE but the memory location pointed to by *raddr* had not had data written to it since the last reset.
CONFLICT	Simultaneous read/write accesses to same address. *conflict_check* = 1 At an active edge of *r_clock, ren* was TRUE but *wen* was also TRUE and *raddr = waddr*. This check assumes *r_clock* and *w_clock* are the same signal.
ONE_READ	Memory location had two read accesses without an intervening write access. *one_read_check* = 1 At an active edge of *r_clock, ren* was TRUE but the previous access to the memory location pointed to by *raddr* was another read.
ONE_WRITE	Memory location had two write accesses without an intervening read access. *one_read_check* = 1 At an active edge of *w_clock, wen* was TRUE but the previous access to the memory location pointed to by *waddr* was another write.
VALUE	Data item read from a location did not match the data last written to that location. *value_check* = 1 At an active edge of *r_clock, ren* was TRUE but the value of *rdata* did not equal the expected value, which was the value of *wdata* when a write access to the memory location pointed to by the current value of *raddr* last occurred.

Implicit X/Z Checks

`start_addr contains X or Z`	Start address contained X or Z bits.
`end_addr contains X or Z`	End address contained X or Z bits.
`ren contains X or Z`	Read enable was X or Z.
`raddr contains X or Z`	Read address contained X or Z bits.
`rdata contains X or Z`	Read data contained X or Z bits.
`wen contains X or Z`	Write enable was X or Z.
`waddr contains X or Z`	Write address contained X or Z bits.
`wdata contains X or Z`	Write data contained X or Z bits.

Cover Points

`cover_reads`	SANITY — Number of read accesses.
`cover_writes`	SANITY — Number of write accesses.
`cover_write_then_read_from_same_addr`	BASIC — Number of times a write access was followed by a read from the same address.
`cover_same_addr_simultaneous_read_write`	CORNER — Number of times a simultaneous read/write access to the same address occurred. Not meaningful unless *pass_thru* is 1.
`cover_different_addr_simultaneous_read_write`	CORNER — Number of times a simultaneous read/write access to different addresses occurred. Not meaningful unless *pass_thru* is 1.
`cover_read_from_start_addr`	CORNER — Number of read accesses to the location specified by *start_addr*.
`cover_write_to_start_addr`	CORNER — Number of write accesses to the location specified by *start_addr*.
`cover_read_from_end_addr`	CORNER — Number of read accesses to the location specified by *end_addr*.
`cover_write_to_end_addr`	CORNER — Number of write accesses to the location specified by *end_addr*.
`cover_write_then_read_from_start_addr`	CORNER — Number of times a write access to *start_addr* was followed by a read from *start_addr*.

`cover_write_then_read_from_end_addr`	CORNER — Number of times a write access to *end_addr* was followed by a read from *end_addr*.
`cover_read_addr`	STATISTIC — Reports which addresses were read at least once.
`cover_write_addr`	STATISTIC — Reports which addresses were written at least once.
`cover_read_to_write_delays`	STATISTIC — Reports which delays (in numbers of active *w_clock* edges) from a read to the next write (to any address) occurred at least once.
`cover_write_to_read_delays`	STATISTIC — Reports which delays (in numbers of active *r_clock* edges) from a write to the next read (to any address) occurred at least once.
`cover_two_writes_without_read`	STATISTIC — Number of times a memory location had two write accesses but no read access of the data item stored by the first write.
`cover_two_reads_without_write`	STATISTIC — Number of times a memory location had two read accesses but no write access overwriting the data item read by the first read.

ovl_multiport_fifo

Ensures the data integrity of a FIFO with multiple enqueue and dequeue ports, and ensures that the FIFO does not overflow or underflow.

```
enq[enq_count-1:0]
deq[deq_count-1:0]
full                    fire ['OVL_FIRE_WIDTH-1:0]
empty
          ovl_multiport_fifo
enq_data[enq_count*width-1:0]
deq_data[deq_count*width-1:0]
preload[preload_count*width-1:0]*
          clock  reset  enable
```

*if *preload_count* = 0:
   preload is *width* bits wide

Parameters:

severity_level	*high_water_mark*
width	*full_check*
depth	*empty_check*
enq_count	*value_check*
deq_count	*property_type*
pass_thru	*msg*
registered	*coverage_level*
enq_latency	*clock_edge*
deq_latency	*reset_polarity*
preload_count	*gating_type*

Class: *n*-cycle assertion

Syntax

```
ovl_multiport_fifo
      [#(severity_level, width, depth, enq_count, deq_count, pass_thru,
         registered, enq_latency, deq_latency, preload_count,
         high_water_mark, full_check, empty_check, value_check,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
    instance_name (clock, reset, enable, enq, deq, enq_data, deq_data,
      full, empty, preload, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of a data item in the FIFO. Default: 1.
depth	FIFO depth. The *depth* must be > 0 . Default: 2.
enq_count	Number of FIFO enqueue ports. Must be $\leq$ *depth*. Default: 2.
deq_count	Number of FIFO dequeue ports. Must be $\leq$ *depth*. Default: 2.

pass_thru	How the FIFO handles dequeues and enqueues in the same cycle if the FIFO count is such that a dequeue violation might occur.
	pass_thru = 0 (Default)
	No pass-through mode means dequeue before enqueue. A dequeue violation occurs if the number of scheduled dequeues > the current FIFO count.
	pass = 1
	Pass-through mode means enqueue before dequeue. A dequeue violation occurs if the number of scheduled dequeues – the number of scheduled enqueues > the current FIFO count.
registered	How the FIFO handles dequeues and enqueues in the same cycle if the FIFO count is such that an enqueue violation might occur.
	registered = 0 (Default)
	No registered mode means enqueue before dequeue. An enqueue violation occurs if the current FIFO count + the number of scheduled enqueues > *depth*.
	registered = 1
	Registered mode means dequeue before enqueue. An enqueue violation occurs if the current FIFO count + the number of scheduled enqueues – the number scheduled dequeues > *depth*.
enq_latency	Latency for enqueue data.
	enq_latency = 0 (Default)
	Checks and coverage assume *enq_data* is valid and the enqueue operation is performed in the same cycle *enq* asserts.
	enq_latency > 0
	Checks and coverage assume *enq_data* is valid and the enqueue operation is performed *enq_latency* cycles after *enq* asserts.
deq_latency	Latency for dequeued data. It is used for the value check.
	deq_latency = 0 (Default)
	Checks and coverage assume *deq_data* is valid and the dequeue operation is performed in the same cycle *deq* asserts.
	deq_latency > 0
	Checks and coverage assume *deq_data* is valid and the dequeue operation is performed *deq_latency* cycles after *deq* asserts.
preload_count	Number of items to preload the FIFO on reset. The preload port is a concatenated list of items to be preloaded into the FIFO. Default: 0 (FIFO empty on reset).
high_water_mark	FIFO high-water mark. Must be < *depth*. A value of 0 disables the high_water_mark cover point. Default: 0.

full_check	Whether or not to perform full checks.
	full_check = 0 (Default)
	Turns off the full check.
	full_check = 1
	Turns on the full check.
empty_check	Whether or not to perform empty checks.
	empty_check = 0 (Default)
	Turns off the empty check.
	empty_check = 1
	Turns on the empty check.
value_check	Whether or not to perform value checks.
	value_check = 0 (Default)
	Turns off the value check.
	value_check = 1
	Turns on the value check.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
enq[enq_count-1:0]	Concatenation of FIFO enqueue inputs. When one or more *enq* bits are sampled TRUE, the FIFO performs an enqueue operation from the asserted bits' corresponding enqueue data ports (*enq_latency* cycles later). Data items are enqueued in order from the least to most-significant bits and the FIFO counter is incremented by the number of TRUE *enq* bits

deq[deq_count-1:0]	Concatenation of FIFO dequeue inputs. When one or more *deq* bits are sampled TRUE, the FIFO performs a dequeue operation from the asserted bits' corresponding dequeue data ports (*deq_latency* cycles later). Data items are dequeued in order from the least to most-significant bits and the FIFO counter is decremented by the number of TRUE *deq* bits
full	Output status flag from the FIFO.
	full = 0
	FIFO not full.
	full = 1
	FIFO full.
empty	Output status flag from the FIFO.
	empty = 0
	FIFO not empty.
	empty = 1
	FIFO empty.
*enq_data [enq_count*width*-1:0]	Concatenation of enqueue data inputs. If the value check is on, this port contains the data items to enqueue *enq_latency* cycles after the *enq* bits assert.
*deq_data [deq_count*width*-1:0]	Concatenation of dequeue data inputs. If the value check is on, this port contains the dequeued data items *deq_latency* cycles after the *deq* bits assert.
*preload [preload_count*width*-1 :0]	Concatenated preload data to enqueue on reset.
	preload_count = 0
	No preload of the FIFO is assumed. The width of preload should be *width*, however no values from *preload* are used. The FIFO is assumed to be empty on reset.
	preload_count > 0
	Checker assumes the value of *preload* is a concatenated list of items that were all enqueued on the FIFO on reset (or simulation start). The width of preload should be *preload_count * width* (preload items are the same width). Preload values are enqueued from the low order item to the high order item.
fire [`OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_multiport_fifo assertion checker ensures a multiport FIFO functions legally. A multiport FIFO is a memory structure that stores and retrieves data items based on a first-in first-out queueing protocol. The FIFO can have multiple enqueue data ports and multiple dequeue data ports (the number of each does need to match). Each enqueue data port has a corresponding enqueue signal that indicates the data port's value should be enqueued. Similarly, each dequeue data port has a corresponding dequeue signal that indicates a data item from the FIFO should be dequeued to that port.

A FIFO with multiple enqueue ports can signal an enqueue from any combination of the ports each enqueue clock cycle. Similarly, a FIFO with multiple dequeue ports can signal a dequeue to any combination of the ports each dequeue clock cycle. When multiple ports are enqueued (dequeued) in a cycle, the order their contents are enqueued (dequeued) is always the same. A FIFO can also have enqueue and dequeue latency constants. Enqueue latency is the number of clock cycles after an enqueue signal asserts that the corresponding enqueue data value is valid at the corresponding enqueue data port. Dequeue latency is the number of clock cycles it takes for a dequeue to produce a data value at its corresponding dequeue port.

To connect the ovl_multiport_fifo checker to the FIFO logic:

- Concatenate the enqueue signals—arranged in order from first-in (least-significant bit) to last-in (most-significant bit)—and connect to the *enq* port. Concatenate the dequeue signals—arranged in order from first-out (least-significant bit) to last-out (most-significant bit)—and connect to the *deq* port.

- If the checker will perform value checks, concatenate the enqueue data ports in the same order as the *enq* bits and connect to the *enq_data* port. Concatenate the dequeue data ports in the same order as the *deq* bits and connect to the *deq_data* port. Otherwise, connect *enq_data* and *deq_data* to 0.

- If the checker will perform full checks, connect the FIFO-full status flag to the *full* port. Otherwise, connect *full* to 1'b0. If the checker will perform empty checks, connect the FIFO-full status flag to the *empty* port. Otherwise, connect *empty* to 1'b0.

The checker checks *enq* and *deq* at the active edge of *clock*. If an *enq* bit is TRUE, an enqueue operation is scheduled for the corresponding enqueue data port *enq_latency* cycles later (or in the current cycle if *enq_latency* is 0). Similarly, if a *deq* bit is TRUE, a dequeue operation is scheduled to the corresponding dequeue data port *deq_latency* cycles later (or in the current cycle if *deq_latency* is 0).

At each active edge of *clock*, the checker does the following:

1. Updates its FIFO counter with the results of enqueues and dequeues from the previous cycle.

2. Checks the *full* flag if *full_check* is 1. If *full* is FALSE and the FIF0 count = *depth* or if *full* is TRUE and the FIFO count < *depth*, a full check violation occurs.

3. Checks the *empty* flag if *empty_check* is 1. If *empty* is FALSE and the FIF0 count = 0 or if *empty* is TRUE and the FIFO count > 0, an empty check violation occurs.

4. Checks for a potential overflow. If the number of enqueues scheduled for the current cycle exceeds the current number of unused FIFO locations, an enqueue check violation occurs. In this case, since the FIFO state is unknown, value checks are turned off until the next checker reset.

5. Checks for a potential underflow. If the number of dequeues scheduled for the current cycle exceeds the current number of FIFO entries, a dequeue check violation occurs. In this case, since the FIFO state is unknown, value checks are turned off until the next checker reset.

6. If *value_check* is 1 (and no enqueue or dequeue violations have occurred), the checker maintains an internal copy of what it expects the FIFO entries to be. The checker issues a value check violation for each internal dequeued data item that does not match the corresponding value of *deq_data*.

A corner-case situation occurs when both enqueues and dequeues are scheduled simultaneously in the same cycle. By default, the checker enforces the best-case (i.e., most restrictive) scenarios. For the enqueue check, enqueues are "performed" before dequeues. For the dequeue check, dequeues are "performed" before enqueues. However, the checker can be configured to allow worse-case (i.e., less restrictive) scenarios by setting the *registered* and *pass_thru* parameters:

- In registered mode, the enqueue check calculates the FIFO count by subtracting the number of dequeues before adding the number of enqueues, resulting in a less restrictive check.

- In pass-through mode, the dequeue check calculates the FIFO count by adding the number of enqueues before subtracting the number of dequeues, resulting in a less restrictive check.

By default, the FIFO is empty at the start of the first cycle after a reset (or the start of simulation). However, the checker can be configured to match a FIFO that contains data items at these initial points. To do this, the checker "preloads" these data items. The *preload_count* parameter specifies the number of data items to preload.

If *value_check* is 1, at the start of any cycle in which reset has transitioned from active to inactive, the checker reads the *preload* port. This is a port containing a concatenated value equal to *preload_count* data items. The checker enqueues these data items onto the internal FIFO in order from the low-order item to the high-order item.

Uses: FIFO, queue, buffer, ring buffer, elasticity buffer.

Assertion Checks

ENQUEUE	Enqueue occurred that would overflow the FIFO.

registered = 0

> One or more *enq* bits were TRUE, but *enq_latency* cycles later, FIFO count + number of enqueued items > *depth*.

registered = 1

> One or more *enq* bits were TRUE, but *enq_latency* cycles later, FIFO count + number of enqueued items – number of dequeued items.

DEQUEUE	Dequeue occurred that would underflow the FIFO.

pass_thru = 0

> One or more *deq* bits were TRUE, but *deq_latency* cycles later, FIFO count < number of dequeued items.

pass_thru = 1

> One or more *deq* bits were TRUE, but *deq_latency* cycles later, FIFO count < number of dequeued items – number of enqueued items.

FULL	The FIFO was not full when the full signal was asserted.

> *Full* was TRUE, but the FIFO contained fewer than *depth* items.

The full signal was not asserted when the FIFO was full.

> *Full* was FALSE, but the FIFO \contained *depth* items.

FULL	FIFO 'full' signal was asserted, but the FIFO was not full.

> FIFO contained fewer than *depth* items but *full* was TRUE.

FIFO 'full' signal was not asserted, but the FIFO was full.

> FIFO contained *depth* items and *full* was FALSE.

EMPTY	FIFO 'empty' signal was asserted, but the FIFO was not empty.

> FIFO contained one or more items but *empty* was TRUE.

FIFO 'empty' signal was not asserted, but the FIFO was empty.

> FIFO contained no items but *empty* was FALSE.

VALUE   Dequeued FIFO value did not equal the corresponding
                               enqueued value.
                               deq_latency = 0
                                   A *deq* bit was TRUE, but the corresponding data item in
                                   deq_data did not equal the item originally enqueued.
                               deq_latency > 0
                                   A *deq* bit was TRUE, but *deq_latency* cycles later the
                                   corresponding data item in *deq_data* did not equal the item
                                   originally enqueued.
                               This check automatically turns off if an enqueue or dequeue
                               check violation occurs since it is no longer possible to correspond
                               enqueued with dequeued values. The check turns back on when
                               the checker resets.

Implicit X/Z Checks

enq contains X or Z   Enqueue contained X or Z bits.

deq contains X or Z   Dequeue contained X or Z bits.

full contains X or Z   FIFO full signal was X or Z. Check is off if *full_check* is 0.

empty contains X or Z   FIFO empty signal was X or Z. Check is off if *empty_check* is 0.

enq_data contains X or   Enqueue data item in the *enq_data* expression contained X or Z
Z   bits when it was scheduled to be enqueued onto the FIFO.

deq_data contains X or   Dequeue data item in the *deq_data* expression contained X or Z
Z   bits when it was scheduled to be dequeued from the FIFO.

Cover Points

cover_enqueues   SANITY — Number of data items enqueued on the FIFO.

cover_dequeues   SANITY — Number of data items dequeued from the FIFO.

cover_simultaneous_   BASIC — Number of cycles both an enqueue and a dequeue
enq_deq   (to/from the same port??) were scheduled to occur.

cover_high_water_mark   CORNER — Number of times the FIFO count transitioned from
                              < *high_water_mark* to ≥ *high_water_mark*. Not reported if
                              high_water_mark is 0.

cover_simultaneous_   CORNER — Number of cycles the FIFO was enqueued and
deq_enq_when_empty   dequeued simultaneously when it was empty.

cover_simultaneous_   CORNER — Number of cycles the FIFO was enqueued and
deq_enq_when_full   dequeued simultaneously when it was full.

cover_fifo_empty   CORNER — Number of cycles FIFO was empty after processing
                              enqueues and dequeues for the cycle.

cover_fifo_full   CORNER — Number of cycles FIFO was full after processing
                              enqueues and dequeues for the cycle.

`cover_observed_counts`   STATISTIC — Reports the FIFO counts that occurred at least once.

ovl_mutex

Ensures that the bits of an expression are mutually exclusive.

```
fire ['OVL_FIRE_WIDTH-1:0]

test_expr[width-1:0]
            ovl_mutex

        clock   reset   enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
invert_mode   *reset_polarity*
property_type   *gating_type*
msg

Class: 1-cycle assertion
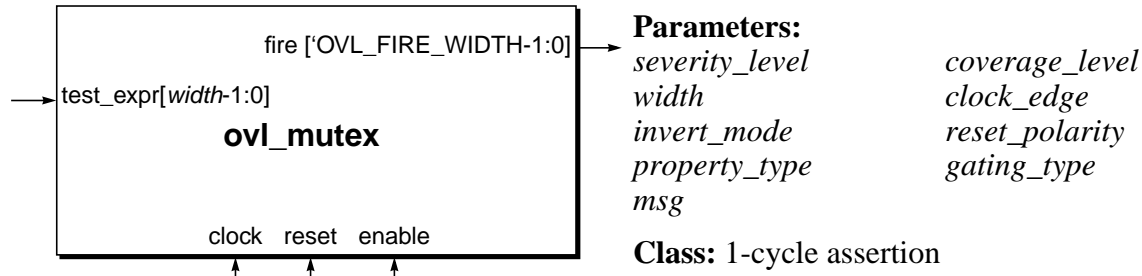
Syntax

```
ovl_mutex
    [#(severity_level, width, invert_mode, property_type, msg,
       coverage_level, clock_edge, reset_polarity, gating_type)]
    instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr*. Default: 2.

invert_mode	Sense of the active bits for the mutex check.
	 invert_mode = 0 (Default)
	    Expression value must not have more than one TRUE bit.
	 invert_mode = 1
	    Expression value must not have more than one FALSE bit.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Variable or expression to check.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_mutex assertion checker checks *test_expr* at each active edge of *clock.* By default, if more than one bit of *test_expr* is TRUE, a mutex violation occurs. Setting *invert_mode* to 1 reverses the sense of the bits. A mutex violation occurs if more than one bit of *test_expr* is FALSE.

Assertion Checks

MUTEX	Expression's bits are not mutually exclusive. *invert_mode* = 0 Expression had more than one TRUE bit. *invert_mode* = 1 Expression had more than one FALSE bit.

Implicit X/Z Checks

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

Cover Points

cover_values_checked	SANITY — Number of cycles *test_expr* loaded a new value.
cover_no_mutex_bits	CORNER — Number of cycles all bits in *test_expr* were TRUE and *invert_mode* = 0 or all bits in *test_expr* were FALSE and *invert_mode* = 1.
cover_mutex_bitmap	STATISTIC — Reports which of *test_expr* bits were TRUE (*invert_mode* = 0) or FALSE (*invert_mode* = 1) at least once.

ovl_never

Ensures that the value of an expression is not TRUE.

Parameters:
severity_level
property_type
msg

coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion
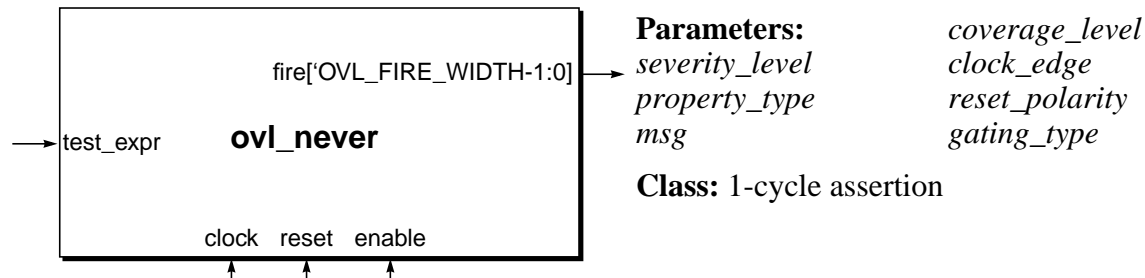
Syntax

```
ovl_never
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
         reset_polarity, gating_type )]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr	Expression that should not evaluate to TRUE on the active clock edge.

Description

The ovl_never assertion checker checks the single-bit expression *test_expr* at each active edge of *clock* to verify the expression does not evaluate to TRUE.

Assertion Checks

NEVER	Expression evaluated to TRUE.

Implicit X/Z Checks

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

Cover Points

Notes

1. By default, the ovl_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

ovl_always
ovl_always_on_edge

ovl_implication
ovl_proposition

Example

```
ovl_never #(

    `OVL_ERROR,                                // severity_level
    `OVL_ASSERT,                               // property_type
    "",                                        // msg
    `OVL_COVER_DEFAULT,                        // coverage_level
    `OVL_POSEDGE,                              // clock_edge
    `OVL_ACTIVE_LOW,                           // reset_polarity
    `OVL_GATE_CLOCK)                           // gating_type

    valid_count (

        clock,                                 // clock
        reset,                                 // reset
        enable,                                // enable
        reg_a < reg_b,                         // test_expr
        fire_valid_count );                    // fire
```

Ensures that (*reg_a* < *reg_b*) is FALSE at each rising edge of *clock.*
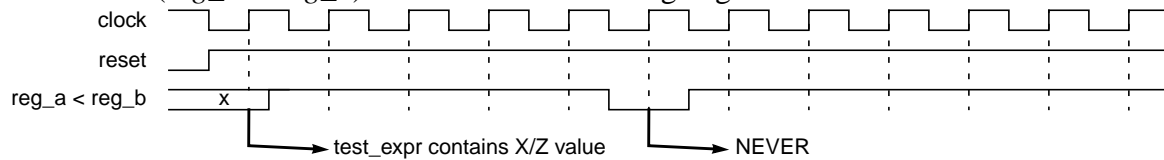
ovl_never_unknown

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

fire['OVL_FIRE_WIDTH-1:0]

qualifier

ovl_never_unknown

test_expr[*width*-1:0]

clock   reset   enable

Parameters:

severity_level   *coverage_level*
property_type   *clock_edge*
msg   *reset_polarity*
                               gating_type

Class: 1-cycle assertion
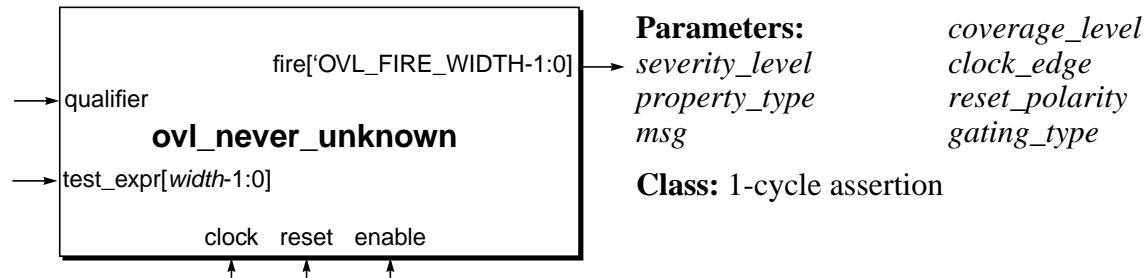
Syntax

```
ovl_never_unknown
    [#(severity_level, width, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type )]
  instance_name (clock, reset, enable, qualifier, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_never_unknown assertion checker checks the expression *qualifier* at each active edge of *clock* 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 ovl_never_unknown checkers are turned off.

See also

ovl_never   ovl_one_hot
ovl_never_unknown_async   ovl_zero_one_hot
ovl_one_cold

Example

```
ovl_never_unknown #(

  'OVL_ERROR,                              // severity_level
  8,                                       // width
  'OVL_ASSERT,                             // property_type
  "Error: data unknown or undriven",       // msg
  'OVL_COVER_DEFAULT,                      // coverage_level
  'OVL_POSEDGE,                            // clock_edge
  'OVL_ACTIVE_LOW,                         // reset_polarity
  'OVL_GATE_CLOCK)                         // gating_type

  valid_data (

    clock,                                 // clock
    reset,                                 // reset
    enable,                                // enable
    rd_data,                               // qualifier
    data,                                  // test_expr
    fire_valid_data );                     // fire
```

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

NEVER_UNKNOWN Error: data unknown or undriven
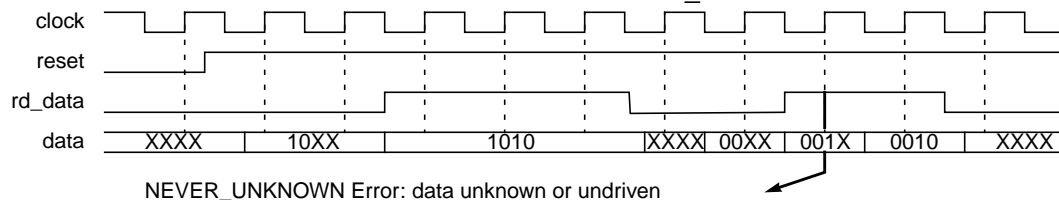
ovl_never_unknown_async

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

```
                                    fire['OVL_FIRE_WIDTH-1:0]
              ovl_never_unknown_async

     test_expr[width-1:0]
                   reset   enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: combinational assertion

Syntax

```
ovl_never_unknown_async
      [#(severity_level, width, property_type, msg, coverage_level,
         clock_edge, reset_polarity, gating_type)]
   instance_name (reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Ignored parameter.
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should contain only 0 or 1 bits when qualifier is TRUE.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_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 ovl_never_unknown_async checkers are turned off.

See also

ovl_never

Example

```
ovl_never_unknown_async #(

    'OVL_ERROR,                                 // severity_level
    8,                                          // width
    'OVL_ASSERT,                                // property_type
    "Error: data unknown or undriven",          // msg
    'OVL_COVER_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK)                            // gating_type

    valid_data (

        bus_gnt,                                // reset
        enable,                                 // enable
        data,                                   // test_expr
        fire_valid_data );                      // fire
```

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

NEVER_UNKNOWN_ASYNC Error: data unknown or undriven
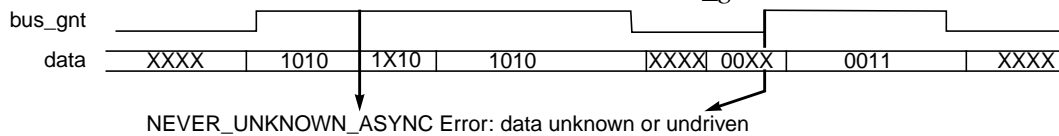
ovl_next

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

Parameters:

severity_level	*msg*
num_cks	*coverage_level*
check_overlapping	*clock_edge*
check_missing_start	*reset_polarity*
property_type	*gating_type*

Class: *n*-cycle assertion
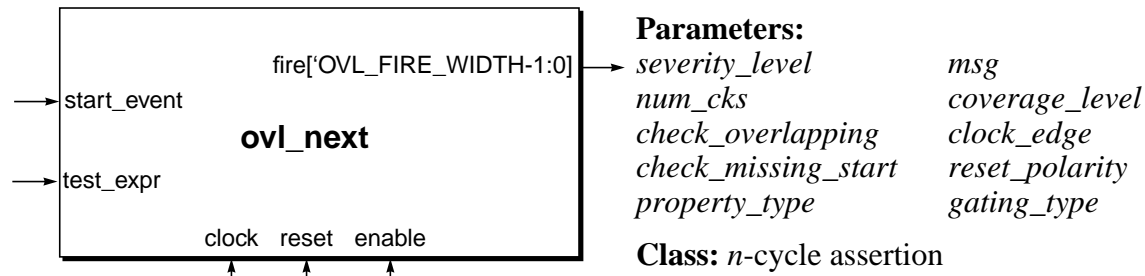
Syntax

```
ovl_next
    [#(severity_level, num_cks, check_overlapping, check_missing_start,
       property_type, msg, coverage_level, clock_edge, reset_polarity,
       gating_type)]
  instance_name (clock, reset, start_event, test_expr, fire);
```
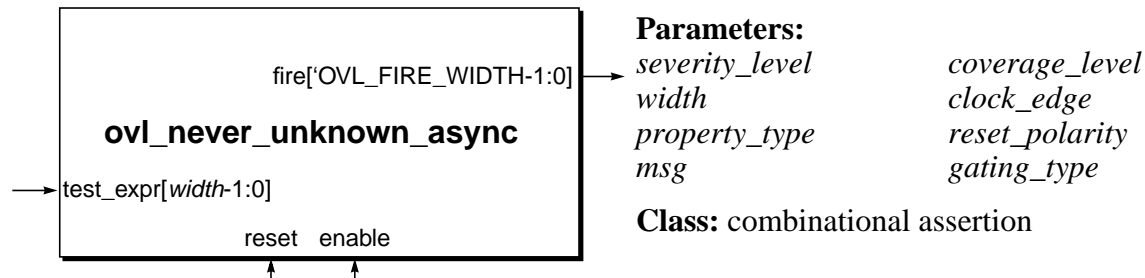
Parameters

`severity_level`	Severity of the failure. Default: 'OVL_SEVERITY_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 active edge of *clock* after *start_event* is sampled TRUE to the active edge of *clock* 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 an active edge of *clock*, 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 active edge of *clock*. If the value of *test_expr* is TRUE, then *num_cks* active edges of *clock* 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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_next assertion checker checks the expression *start_event* at each active edge of *clock*. If *start_event* is TRUE, a check is initiated. The check waits for *num_cks* cycles (i.e., for *num_cks* additional active edges of *clock*) 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 an active edge of *clock*, 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 active edge of *clock*, 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 active edge of *clock*, 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 an active edge of *clock*.
`cover_overlapping_ start_events`	CORNER — The *check_overlapping* parameter is TRUE and the value of *start_event* was TRUE on an active edge of *clock* while a check was pending.

See also

ovl_change

ovl_frame

ovl_time

ovl_unchange

Examples

```
ovl_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_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK)                         // gating_type

    valid_next_a_b (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        a,                                   // start_event
        b,                                   // test_expr
        fire_valid_next_a_b );               // fire
```

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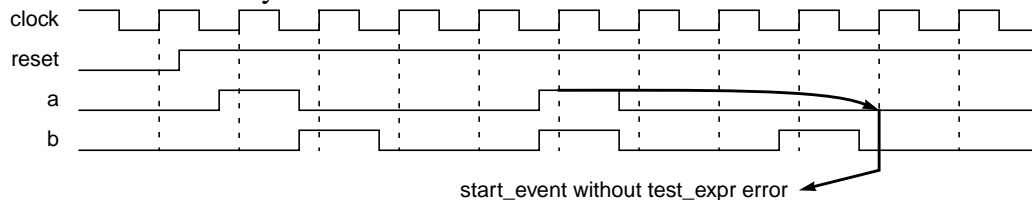

```
ovl_next #(

    `OVL_ERROR,                              // severity_level
    4,                                       // num_cks
    0,                                       // check_overlapping (on)
    0,                                       // check_missing_start (off)
    `OVL_ASSERT,                             // property_type
    "error:",                                // msg
    `OVL_COVER_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK)                         // gating_type

    valid_next_a_b (
```

```
   clock,                                       // clock
   reset,                                       // reset
   enable,                                      // enable
   a,                                           // start_event
   b,                                           // test_expr
   fire_valid_next_a_b );                       // fire
```

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

illegal overlapping condition detected error
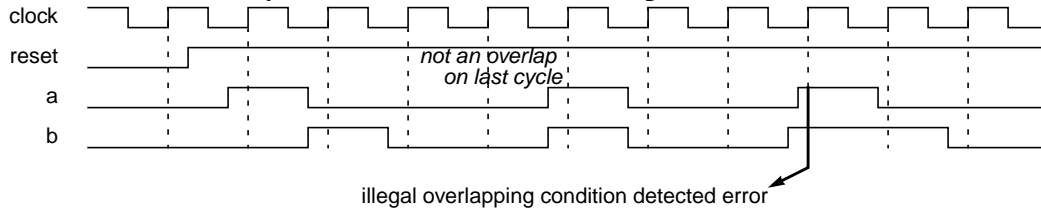
```
ovl_next #(

   'OVL_ERROR,                                  // severity_level
   4,                                           // num_cks
   1,                                           // check_overlapping (off)
   1,                                           // check_missing_start (on)
   'OVL_ASSERT,                                 // property_type
   "error:",                                    // msg
   'OVL_COVER_DEFAULT,                          // coverage_level
   'OVL_POSEDGE,                                // clock_edge
   'OVL_ACTIVE_LOW,                             // reset_polarity
   'OVL_GATE_CLOCK)                             // gating_type

   valid_next_a_b (

      clock,                                    // clock
      reset,                                    // reset
      enable,                                   // enable
      a,                                        // start_event
      b,                                        // test_expr
      fire_valid_next_a_b );                    // fire
```

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

missing-start check indeterminate for 3 cycles after reset

test_expr without start_event error
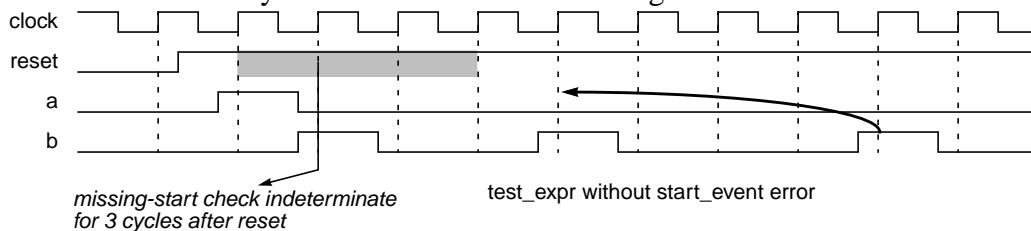
ovl_next_state

Ensures that an expression transitions only to specified values.

fire ['OVL_FIRE_WIDTH-1:0]

test_expr[*width*-1:0]

curr_state[*width*-1:0]

ovl_next_state

next_state[*next_count**width-1:0]

clock   reset   enable

Parameters:

severity_level   *property_type*
width   *msg*
next_count   *coverage_level*
min_hold   *clock_edge*
max_hold   *reset_polarity*
disallow   *gating_type*

Class: event-bounded assertion
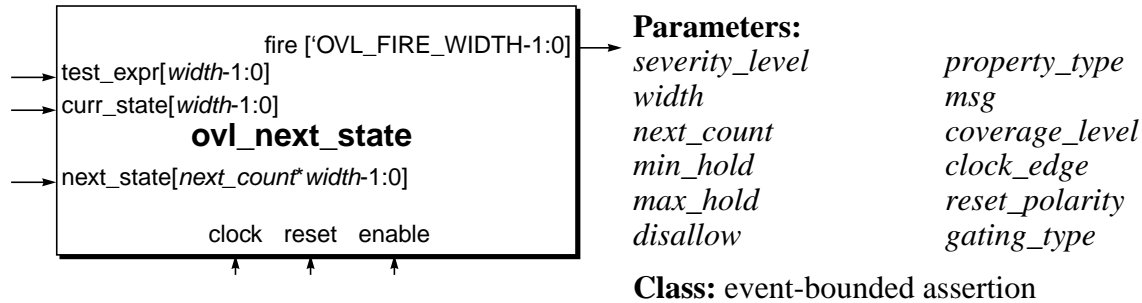
Syntax

```
ovl_next_state
    [#(severity_level, width, next_count, min_hold, max_hold, disallow,
        property_type, msg,coverage_level, clock_edge, reset_polarity,
        gating_type)]
  instance_name (clock, reset, enable, test_expr, curr_state, next_state,
    fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr*. Default: 1
next_count	Number of next state values. The *next_state* port is a concatenated list of next state values. Default: 1.
min_hold	Minimum number of cycles *test_expr* must not change value when it matches the value of *curr_state*. Must be > 0. Default: 1
max_hold	Maximum number of cycles *test_expr* can remain unchanged when it matches the value of *curr_state*. A value of 0 turns off checking for a maximum hold time. Must be 0 or > *min_hold*. Default: 1
disallow	Sense of the comparison of *test_expr* with *next_state*. *disallow* = 0 (Default) Next value of *test_expr* should match one of the values in *next_state*. *disallow* = 1 Next value of *test_expr* should not match one of the values in *next_state*.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).

msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock*, if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[*width*-1:0]	State variable or expression to check.
curr_state[*width*-1:0]	Value to compare with *test_expr*. If no event window is open and the value of *test_expr* matches the value *curr_state*, an event window opens.
next_state [*next_count*width*-1:0]	Concatenated list of next values. *disallow* = 0 Next values are valid values for *test_expr* when an event window closes. *disallow* = 1 Next values are not valid values for *test_expr* when an event window closes.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_next_state assertion checker evaluates *test_expr* and *curr_state* at each active edge of *clock*. If the value of *test_expr* matches the value of *curr_state*, the checker verifies that the value of *test_expr* behaves as follows:

- If *min_hold* > 0 and *test_expr* changes value before *min_hold* cycles (including the match cycle) transpire, a next_state violation occurs.

- Otherwise, when *test_expr* transitions, the checker evaluates *next_state*. If the new value of *test_expr* is not a value in *next_state*, a next_state violation occurs.

- However, if *max_hold* > 0 and *test_expr* does not change value before *max_hold* cycles (including the match cycle) transpire, a next_state violation occurs.

A next_state check is initiated each cycle *test_expr* and *curr_state* match.

Setting the *disallow* parameter to 1, changes the sense of the matching of *test_expr* and *next_state* values. A next_state violation occurs if *test_expr* transitions to a value *in* next_state.

Uses: FSM, state machine, controller, coverage, line coverage, path coverage, branch coverage, state coverage, arc coverage.

Assertion Checks

NEXT_STATE

Match occurred but expression value was not a next value, or expression changed too soon.
> *disallow* = 0 and *max_hold* = 0
> After matching *curr_state*, *test_expr* changed value before *min_hold* cycles (including the match cycle) or transitioned to a value not in *next_state* when it transitioned.

Match occurred but expression value was not a next value, or expression did not change in event window.
> *disallow* = 0 and *max_hold* > 0
> After matching *curr_state*, *test_expr* changed value before *min_hold* cycles (including the match cycle), transitioned to a value not in *next_state* when it transitioned, or did not change value for *max_hold* cycles (including the match cycle).

Match occurred but expression value was a next value, or expression changed too soon.
> *disallow* = 1 and *max_hold* = 0
> After matching *curr_state*, *test_expr* changed value before *min_hold* cycles (including the match cycle) or transitioned to a value in *next_state* when it transitioned.

Match occurred but expression value was a next value, or expression did not change in event window.
> *disallow* = 1 and *max_hold* > 0
> After matching *curr_state*, *test_expr* changed value before *min_hold* cycles (including the match cycle), transitioned to a value in *next_state* when it transitioned, or did not change value for *max_hold* cycles (including the match cycle).

Implicit X/Z Checks

`test_expr contains X or Z`	Expression contained X or Z bits.
`curr_state contains X or Z`	Current state expression contained X or Z bits.
`next_state contains X or Z`	Next state expression contained X or Z bits.

Cover Points

`cover_next_state_ transitions`	SANITY — Number of times *test_expr* matched *curr_state* and then transitioned correctly to a value in *next_state* (*disallow*=0) or not in *next_state* (*disallow*=1).
`cover_all_transitions`	CORNER — Non-zero if *test_expr* transitioned to every next value found in the sampled *next_state*. Not meaningful if *disallow* is 1.
`cover_cycles_checked`	STATISTIC — Number of cycles *test_expr* matched *curr_state*.
`observed_transition`	STATISTIC — Reports which values in *next_state* that *test_expr* transitioned to at least once. Not meaningful if *disallow* is 1.

ovl_no_contention

Ensures that a bus is driven according to specified contention rules.

```
                    fire ['OVL_FIRE_WIDTH-1:0]
   ──────▶  test_expr[width-1:0]

            ovl_no_contention

   ──────▶  driver_enables[num_drivers-1:0]

            clock   reset   enable
              ▲       ▲       ▲
```

Parameters:

severity_level	*msg*
width	*coverage_level*
num_drivers	*clock_edge*
min_quiet	*reset_polarity*
max_quiet	*gating_type*
property_type	

Class: event-bounded assertion
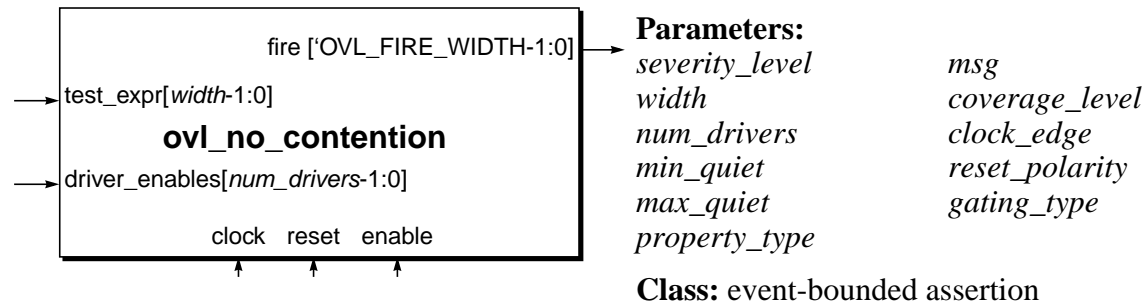
Syntax

```
ovl_no_contention
      [#(severity_level, width, num_drivers, min_quiet, max_quiet,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, test_expr, driver_enables, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr*. Default: 2.
num_drivers	Width of *driver_enables*. Default: 2.
min_quiet	Minimum number of cycles the bus must be quiet (i.e., when all *driver_enables* bits are 0) between transactions. Default: 0 (quiet periods between transactions are not necessary).
max_quiet	Maximum number of cycles the bus can be quiet (i.e., when all *driver_enables* bits are 0). The *min_quiet* parameter must be ≤ *max_quiet*. Default: 0 (quiet periods between transactions should not occur).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[*width*-1:0]	Bus to be checked.
driver_enables [*num_drivers*-1:0]	Enable bits for the drivers of *test_expr*.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_no_contention assertion checker checks the bus (*test_expr*) and the driver enable signals (*driver_enables*) at each active edge of *clock*. An implicit X/Z check violation occurs if any *driver_enables* bit is X or Z.. Otherwise:

- Number of TRUE *driver_enables* bits is > 1:

  A single_driver violation occurs and if *test_expr* contains an X or Z bit, a no_xz violation occurs.

- Number of TRUE *driver_enables* bits is 1:

  If *test_expr* contains an X or Z bit, a no_xz violation occurs.

In addition, the checker performs quiet-time checks. Quiet times consecutive cycles or bus inactivity where no bus transactions are occurring (i.e., *driver_enables* = 0). The checker verifies the specified configuration as follows:

- 0 = *min_quiet* = *max_quiet* (default)

  A quiet violation occurs each cycle *driver_enables* = 0.

- 0 = *min_quiet* < *max_quiet*

  A quiet violation occurs if *driver_enables* = 0 for *max_quiet*+1 consecutive cycles.

- $0 < min\_quiet \leq max\_quiet$

    A quiet violation occurs if either of the following occur:

    - The *driver_enables* expression transitions to 0 and then transitions from 0 less than *min_quiet* cycles later.

    - The *driver_enables* expression = 0 for *max_quiet*+1 cycles.

- $0 = max\_quiet < min\_quiet$

    A quiet violation occurs if *driver_enables* transitions to 0 and then transitions from 0 less than *min_quiet* cycles later.

Assertion Checks

SINGLE_DRIVER
        Bus has multiple drivers.
              Number of TRUE bits in *driver_enables* is > 1.

NO_XZ
        Bus is driven, but has X or Z bits.
              Number of TRUE bits in *driver_enables* is > 0, but *test_expr* has one or more X or Z bits.

QUIET
        Bus was quiet.
              $0 = min\_quiet = max\_quiet$
              Driver_enables was 0.

        Bus was quiet for too many cycles.
              $0 = min\_quiet < max\_quiet$
              Driver_enables was 0 for more than *max_quiet* consecutive cycles.

        Bus was quiet for too few or too many cycles.
              $0 < min\_quiet \leq max\_quiet$
              Driver_enables was not held 0 for at least *min_quiet* consecutive cycles or was 0 for more than *max_quiet* cycles.

        Bus was quiet for too few cycles.
              $0 = max\_quiet < min\_quiet$
              Driver_enables was not held 0 for at least *min_quiet* consecutive cycles.

Implicit X/Z Checks

driver_enables
contains X or Z
        Drivers enabled expression contained X or Z bits.

Cover Points

`cover_driver_bitmap`	BASIC — Bit map of the *driver_enables* signals that have been TRUE at least once.
`cover_quiet_equals_ min_quiet`	CORNER — Number of quiet periods that were exactly *min_quiet* cycles long (*min_quiet* > 0) or number of times bus control transferred from one driver to another (*min_quiet* = 0).
`cover_quiet_equals_ max_quiet`	CORNER — Number of quiet periods that were exactly *max_quiet* cycles long. Not meaningful if *max_quiet* = 0.
`observed_quiet_cycles`	STATISTIC — Reports the quiet periods (in cycles) that have occurred at least once.

ovl_no_overflow

Ensures that the value of an expression does not overflow.

fire['OVL_FIRE_WIDTH-1:0] →

ovl_no_overflow

→ test_expr[*width*-1:0]

clock   reset   enable

Parameters:

severity_level	*msg*
width	*coverage_level*
min	*clock_edge*
max	*reset_polarity*
property_type	*gating_type*

Class: *n*-cycle assertion
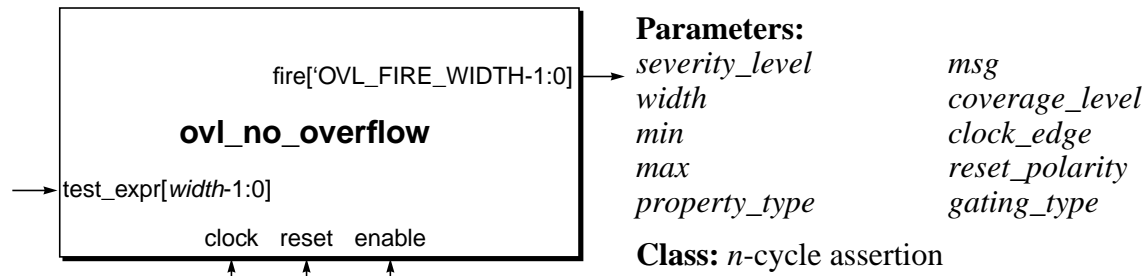
Syntax

```
ovl_no_overflow
    [#(severity_level, width, min, max, property_type, msg,
       coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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*.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_no_overflow assertion checker checks the expression *test_expr* at each active edge of *clock* to determine if its value has changed from a value (at the previous active edge of *clock*) 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 ovl_delta or ovl_fifo_index.

Assertion Checks

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 *clock* after *reset* deasserts.

See also

ovl_delta   ovl_increment
ovl_fifo_index   ovl_no_overflow

Example

```
ovl_no_overflow #(

    `OVL_ERROR,                              // severity_level
    3,                                       // width
    0,                                       // min
    4,                                       // max
    `OVL_ASSERT,                             // property_type
    "Error: addr overflow",                  // msg
    `OVL_COVER_DEFAULT,                      // coverage_level
    `OVL_POSEDGE,                            // clock_edge
    `OVL_ACTIVE_LOW,                         // reset_polarity
    `OVL_GATE_CLOCK)                         // gating_type

    addr_with_overflow (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        addr,                                // test_expr
        fire_addr_with_overflow );           // fire
```

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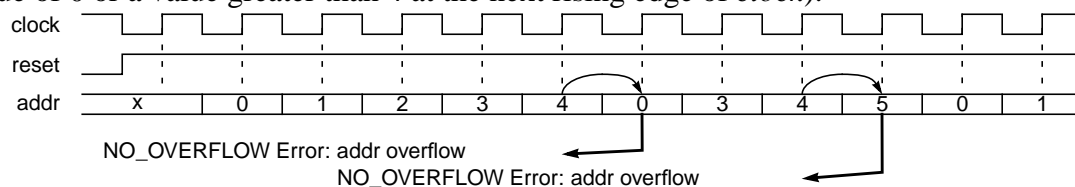
ovl_no_transition

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

fire ['OVL_FIRE_WIDTH-1:0]

test_expr[*width*-1:0]

ovl_no_transition

start_state[*width*-1:0]

next_state[*width*-1:0]

clock   reset   enable

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
property_type	*reset_polarity*
msg	*gating_type*

Class: 2-cycle assertion

Syntax

```
ovl_no_transition
    [#(severity_level, width, property_type, msg, coverage_level,
       clock_edge, reset_polarity, gating_type )]
  instance_name (clock, reset, enable, test_expr, start_state,next_state,
     fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`test_expr[width-1:0]`	Expression that should not transition to *next_state* on the active edge of *clock* if its value at the previous active edge of *clock* 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 active edge of *clock*, 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 active edge of *clock*, then the value of *test_expr* should not equal *next_state* on the current active edge of *clock*.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_no_transition assertion checker checks the expression *test_expr* and *start_state* at each active edge of *clock* to see if they are the same. If so, the checker evaluates and stores the current value of *next_state*. At the next active edge of *clock*, 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

`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 *clock* after *reset* deasserts.

See also

ovl_transition

Example

```
ovl_no_transition #(

    'OVL_ERROR,                             // severity_level
    3,                                      // width
    'OVL_ASSERT,                            // property_type
    "Error: bad state transition",          // msg
    'OVL_COVER_DEFAULT,                     // coverage_level
    'OVL_POSEDGE,                           // clock_edge
    'OVL_ACTIVE_LOW,                        // reset_polarity
    'OVL_GATE_CLOCK)                        // gating_type

    valid_transition (

        clock,                              // clock
        reset,                              // reset
        enable,                             // enable
        current_state,                      // test_expr
        requests > 2 ? 'FULL : 'ONE_IN_Q,   // start_state
        'EMPTY,                             // next_state
        fire_valid_transition);             // fire
```
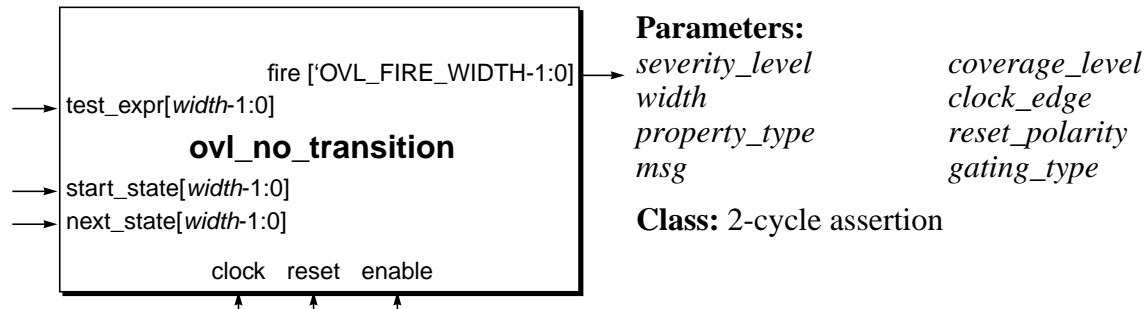
Ensures that *current_state* does not transition to 'EMPTY improperly. If *requests* is greater than 2 and the current_state is 'FULL, *current_state* should not transition to 'EMPTY in the next

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.

clock													
reset													
current_state	'IDLE	'ONE_IN_Q	'EMPTY	'FULL	'EMPTY	'ONE_IN_Q
requests	0	2	1	3	1	2	1

NO_TRANSITION Error: bad state transition

ovl_no_underflow

Ensures that the value of an expression does not underflow.

```
                                        Parameters:
      fire['OVL_FIRE_WIDTH-1:0]         severity_level        msg
                                        width                 coverage_level
      ovl_no_underflow                  min                   clock_edge
                                        max                   reset_polarity
   test_expr[width-1:0]                 property_type         gating_type

      clock   reset   enable            Class: 2-cycle assertion
```
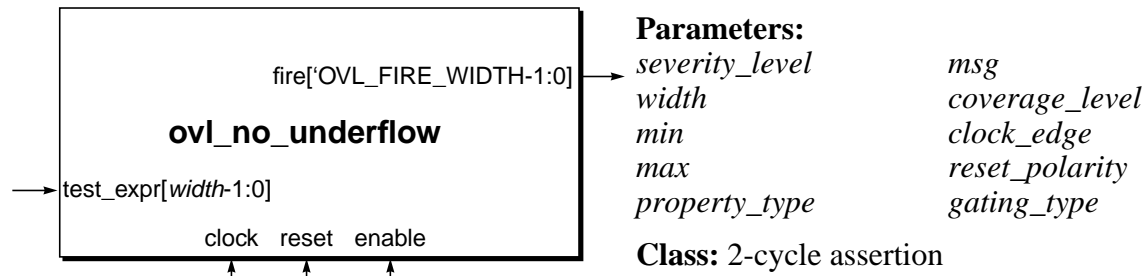
Syntax

```
ovl_no_underflow
      [#(severity_level, width, min, max, property_type, msg,
         coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`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*.
`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_no_underflow assertion checker checks the expression *test_expr* at each active edge of *clock* to determine if its value has changed from a value (at the previous active edge of *clock*) 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 ovl_delta or ovl_fifo_index.

Assertion Checks

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 *clock* after *reset* deasserts.
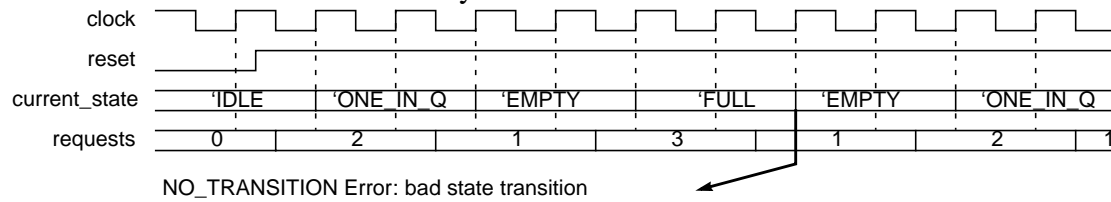
See also

ovl_delta   ovl_fifo_index
ovl_decrement   ovl_no_overflow

Example

```
ovl_no_underflow #(

    'OVL_ERROR,                                 // severity_level
    3,                                          // width
    3,                                          // min
    7,                                          // max
    'OVL_ASSERT,                                // property_type
    "Error: addr underflow",                    // msg
    'OVL_COVER_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK)                            // gating_type

    addr_with_underflow (

       clock,                                   // clock
       reset,                                   // reset
       enable,                                  // enable
       addr,                                    // test_expr
       fire_addr_with_underflow );              // fire
```

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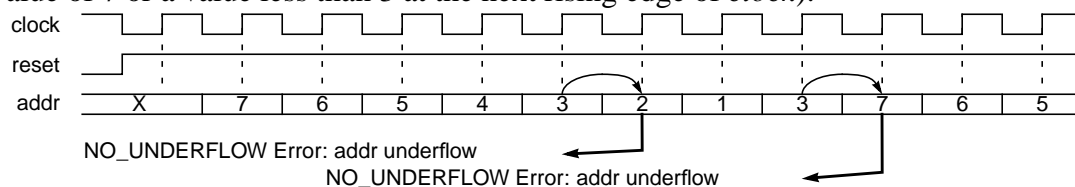
ovl_odd_parity

Ensures that the value of an expression has odd parity.

```
                    fire['OVL_FIRE_WIDTH-1:0]

              ovl_odd_parity

test_expr[width-1:0]

              clock   reset   enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: 1-cycle assertion

Syntax

```
ovl_odd_parity
      [#(severity_level, width, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type)]
    instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.

`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`test_expr[`*width*`-1:0]`	Expression that should evaluate to a value with odd parity on the rising clock edge.
`fire [`'OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_odd_parity assertion checker checks the expression *test_expr* at each active edge of *clock* 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

`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

ovl_even_parity

Example

```
ovl_odd_parity #(

    'OVL_ERROR,                              // severity_level
    8,                                       // width
    'OVL_ASSERT,                             // property_type
    "Error: data has even parity",          // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_data_odd_parity (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        data,                                // test_expr
        fire_valid_data_odd_parity );        // fire
```
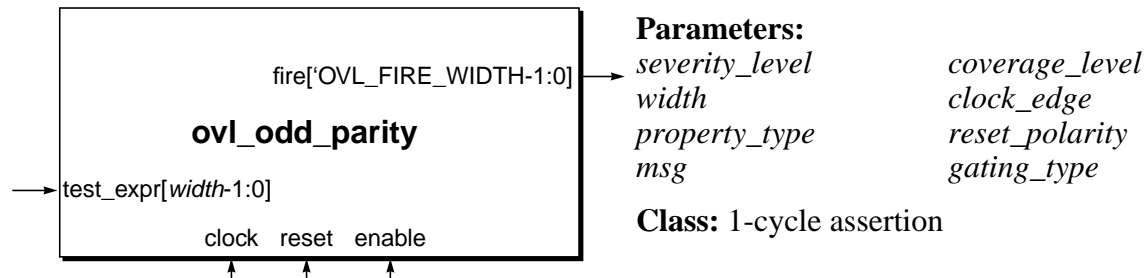
Ensures that *data* has odd parity at each rising edge of *clock*.

ODD_PARITY
Error: data has even parity
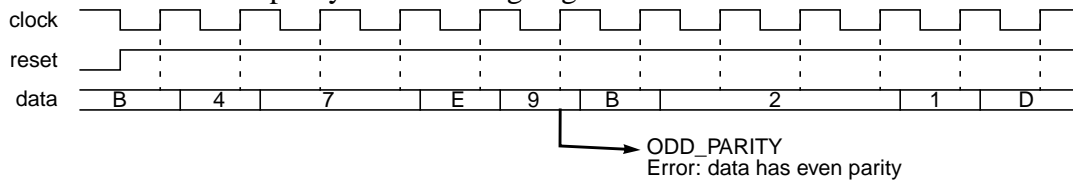
ovl_one_cold

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

fire['OVL_FIRE_WIDTH-1:0]

ovl_one_cold

test_expr[*width*-1:0]

clock   reset   enable

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
inactive	*reset_polarity*
property_type	*gating_type*

msg

Class: 1-cycle assertion
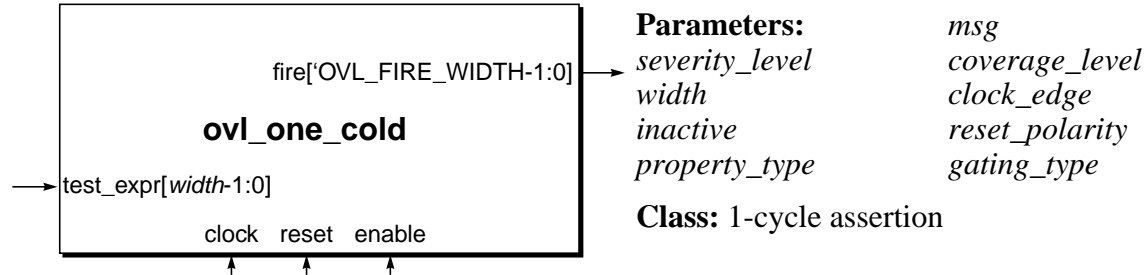
Syntax

```
ovl_one_cold
      [#(severity_level, width, inactive, property_type, msg,
          coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

`severity_level`	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
`msg`	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
`coverage_level`	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
`clock_edge`	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
`reset_polarity`	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
`gating_type`	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Expression that should evaluate to a one-cold or inactive value on the rising clock edge.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_one_cold assertion checker checks the expression *test_expr* at each active edge of *clock* 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

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 ovl_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

ovl_one_hot   ovl_zero_one_hot

Examples

```
ovl_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_DEFAULT,                   // coverage_level
    'OVL_POSEDGE,                         // clock_edge
    'OVL_ACTIVE_LOW,                      // reset_polarity
    'OVL_GATE_CLOCK)                      // gating_type

    valid_sel_n_one_cold (

        clock,                            // clock
        reset,                            // reset
        enable,                           // enable
        sel_n,                            // test_expr
        fire_valid_sel_n_one_cold );      // fire
```

Ensures that *sel_n* is one-cold at each rising edge of *clock*.

test_expr contains X/Z value

ONE_COLD Error: sel_n not one-cold
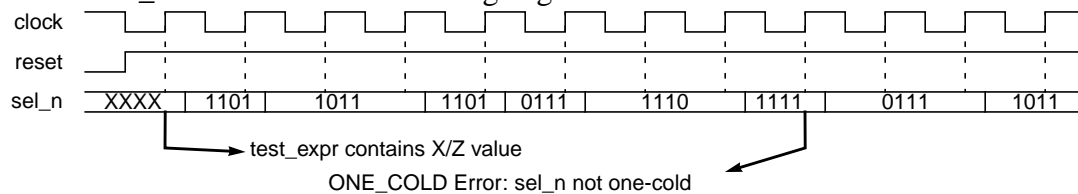
```
ovl_one_cold #(

    '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_DEFAULT,                            // coverage_level
    'OVL_POSEDGE,                                  // clock_edge
    'OVL_ACTIVE_LOW,                               // reset_polarity
    'OVL_GATE_CLOCK)                               // gating_type

    valid_sel_n_one_cold (

        clock,                                     // clock
        reset,                                     // reset
        enable,                                    // enable
        sel_n,                                     // test_expr
        fire_valid_sel_n_one_cold );               // fire
```

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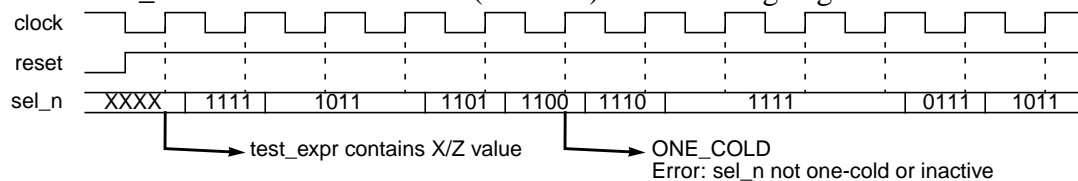

```
ovl_one_cold #(

    'OVL_ERROR,                                    // severity_level
    4,                                             // width
    'OVL_ALL_ZEROS,                                // inactive
    'OVL_ASSERT,                                   // property_type
    "Error: sel_n not one-cold",                   // msg
    'OVL_COVER_DEFAULT,                            // coverage_level
    'OVL_POSEDGE,                                  // clock_edge
    'OVL_ACTIVE_LOW,                               // reset_polarity
    'OVL_GATE_CLOCK)                               // gating_type

    valid_sel_n_one_cold (

        clock,                                     // clock
        reset,                                     // reset
        enable,                                    // enable
        sel_n,                                     // test_expr
        fire_valid_sel_n_one_cold );               // fire
```

Ensures that *sel_n* is one-cold or inactive (4'b0000) at each rising edge of *clock*.
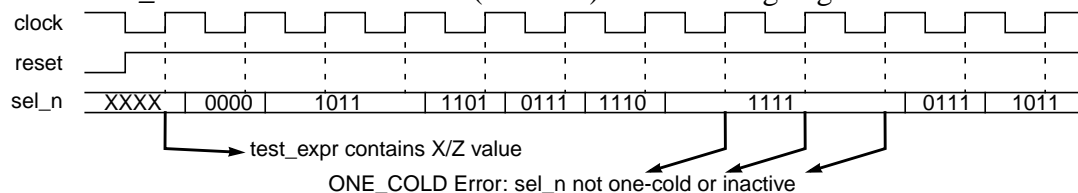
ovl_one_hot

Ensures that the value of an expression is one-hot.

```
                  fire['OVL_FIRE_WIDTH-1:0]
          ovl_one_hot

test_expr[width-1:0]

          clock  reset  enable
```

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
property_type	*reset_polarity*
msg	*gating_type*

Class: 1-cycle assertion

Syntax

```
ovl_one_hot
    [#(severity_level, width, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 32.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.

`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`test_expr[width-1:0]`	Expression that should evaluate to a one-hot value on the rising clock edge.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_one_hot assertion checker checks the expression *test_expr* at each active edge of *clock* 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

`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 ovl_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 ONE_HOT assertion fails if and only if *test_expr* is zero or has multiple bits that are 1.

See also

ovl_one_cold   ovl_zero_one_hot

Example

```
ovl_one_hot #(

    'OVL_ERROR,                              // severity_level
    4,                                       // width
    'OVL_ASSERT,                             // property_type
    "Error: sel not one-hot",                // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_sel_one_hot (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        sel,                                 // test_expr
        fire_valid_sel_one_hot );            // fire
```
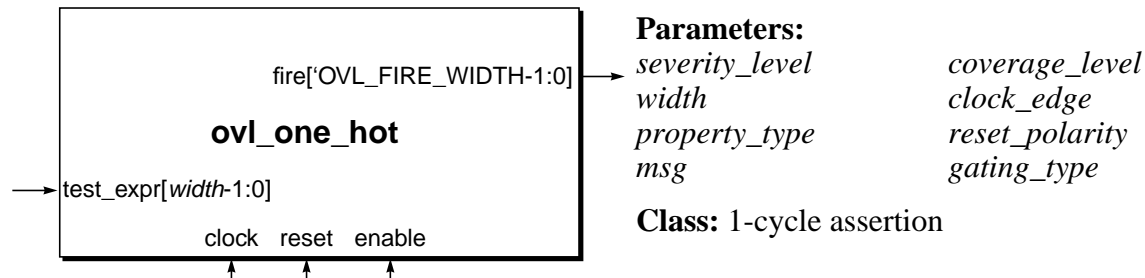
Ensures that *sel* is one-hot at each rising edge of *clock*.
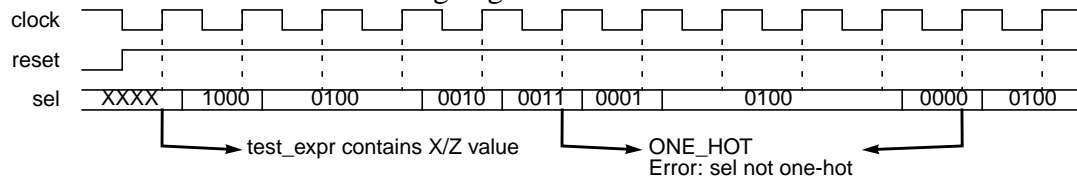
ovl_proposition

Ensures that the value of an expression is always combinationally TRUE.

```
                              fire['OVL_FIRE_WIDTH-1:0]

                    ovl_proposition


        test_expr


              reset    enable
```

Parameters: *coverage_level*
severity_level   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: combinational assertion
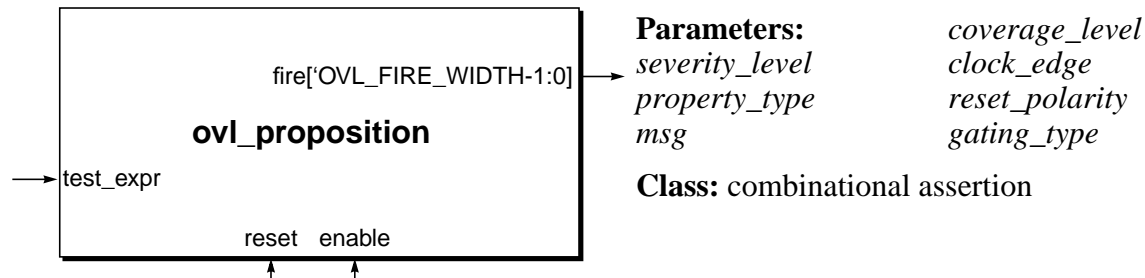
Syntax

```
ovl_proposition
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
         reset_polarity, gating_type)]
    instance_name (reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Ignored parameter.
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr	Expression that should always evaluate to TRUE.

fire [`OVL_FIRE_WIDTH-1:0`]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

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

Assertion Checks

PROPOSITION	Expression evaluated to FALSE.

Implicit X/Z Checks

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

Cover Points

Notes

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

See also

ovl_always   ovl_implication
ovl_always_on_edge   ovl_never

Example

```
ovl_proposition #(

    `OVL_ERROR,                             // severity_level
    `OVL_ASSERT,                            // property_type
    "Error: current_addr changed while bus  // msg
granted",                                   // coverage_level
    `OVL_COVER_DEFAULT,                     // clock_edge
    `OVL_POSEDGE,                           // reset_polarity
    `OVL_ACTIVE_LOW,                        // gating_type
    `OVL_GATE_CLOCK)

    valid_current_addr (

        bus_gnt,                            // reset
        enable,                             // enable
        current_addr == addr,               // test_expr
        fire_valid_current_addr );          // fire
```

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

bus_gnt		
addr	FFFF	AA00
current_addr	FFFF	AA00	AAF0

PROPOSITION Error: current_addr changed while bus granted
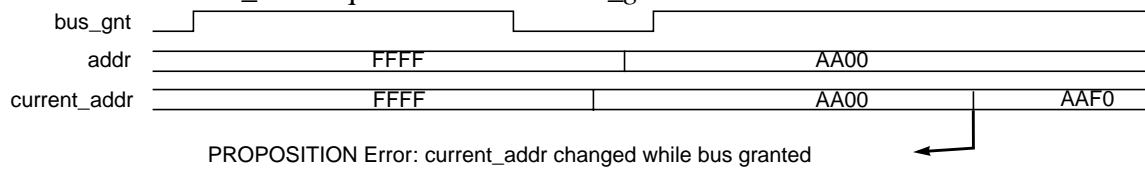
ovl_quiescent_state

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

```
                              fire ['OVL_FIRE_WIDTH-1:0]
          sample_event
                ovl_quiescent_state

          state_expr[width-1:0]
          check_value[width-1:0]

                clock  reset  enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: 2-cycle assertion
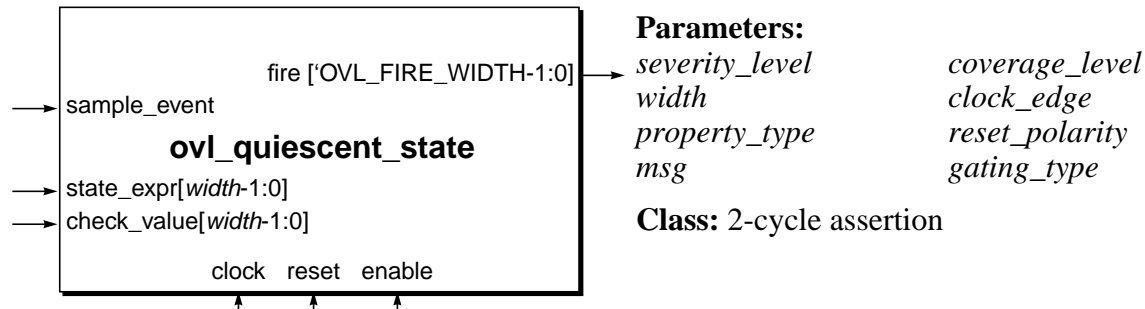
Syntax

```
ovl_quiescent_state
    [#(severity_level, width, property_type, msg, coverage_level,
       clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, state_expr, check_value,
     sample_event, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *state_expr* and *check_value* arguments. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`state_expr[width-1:0]`	Expression that should have the same value as *check_value* on the rising edge of *clock* if *sample_event* has just transitioned to TRUE (rising edge).
`check_value[width-1:0]`	Expression that indicates the value *state_expr* should have on the active edge of *clock* if *sample_event* has just transitioned to TRUE (rising edge).
`sample_event`	Expression that initiates the quiescent state check when its value transitions to TRUE.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_quiescent_state assertion checker checks the expression *sample_event* at each active edge of *clock* to see if its value has transitioned to TRUE (i.e., its current value is TRUE and its value on the previous active edge of *clock* 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

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

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 *clock* after *reset* 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

ovl_no_transition   ovl_transition

Example

```
ovl_quiescent_state #(

    'OVL_ERROR,                                // severity_level
    4,                                         // width
    'OVL_ASSERT,                               // property_type
    "Error: illegal end of transaction",       // msg
    'OVL_COVER_DEFAULT,                        // coverage_level
    'OVL_POSEDGE,                              // clock_edge
    'OVL_ACTIVE_LOW,                           // reset_polarity
    'OVL_GATE_CLOCK)                           // gating_type

    valid_end_of_transaction_state (

        clock,                                 // clock
        reset,                                 // reset
        enable,                                // enable
        transaction_state,                     // state_expr
        prev_tr == 'TR_READ ? 'TR_IDLE :'TR_WAIT   // check_value
        end_of_transaction,                    // sample_event
        fire_valid_end_of_transaction_state ); // fire
```

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).

QUIESCENT_STATE Error: illegal end of transaction
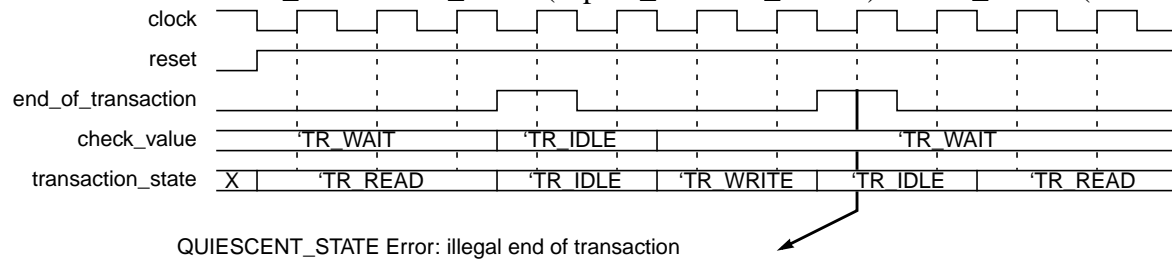
ovl_range

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

```
                                           Parameters:
        fire['OVL_FIRE_WIDTH-1:0]          severity_level        msg
                                           width                 coverage_level
            ovl_range                      min                   clock_edge
                                           max                   reset_polarity
   test_expr[width-1:0]                    property_type         gating_type

        clock  reset  enable               Class: 1-cycle assertion
```
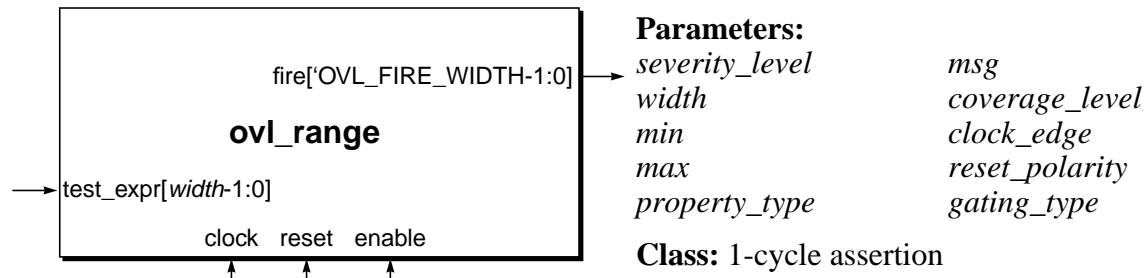
Syntax

```
ovl_range
    [#(severity_level, width, min, max, property_type, msg,
        coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

`severity_level`	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
`msg`	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
`coverage_level`	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
`clock_edge`	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
`reset_polarity`	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
`gating_type`	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_range assertion checker checks the expression *test_expr* at each active edge of *clock* 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

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

ovl_always   ovl_never
ovl_implication   ovl_proposition

Example

```
ovl_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_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK)                            // gating_type

    valid_sel (

        clock,                                  // clock
        reset,                                  // reset
        enable,                                 // enable
        sel_high - sel_low,                     // test_expr
        fire_valid_sel );                       // fire
```

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

RANGE Error: sel_high - sel_low not within 2 to 5
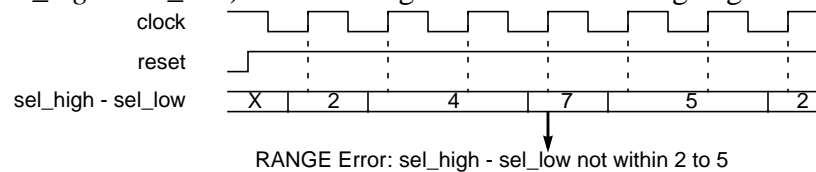
ovl_reg_loaded

Ensures that a register is loaded with source data within a specified time window.

```
                              ┌──────────────────────────────┐
      start_event  ───────►   │             fire ['OVL_FIRE_WIDTH-1:0]  ──►
      end_event    ───────►   │         ovl_reg_loaded       │
      src_expr[width-1:0] ─►  │                              │
      dest_expr[width-1:0] ─► │                              │
                              │    clock   reset   enable    │
                              └──────▲───────▲───────▲───────┘
```

Parameters: *msg*
severity_level *coverage_level*
width *clock_edge*
start_count *reset_polarity*
end_count *gating_type*
property_type

Class: event-bounded assertion
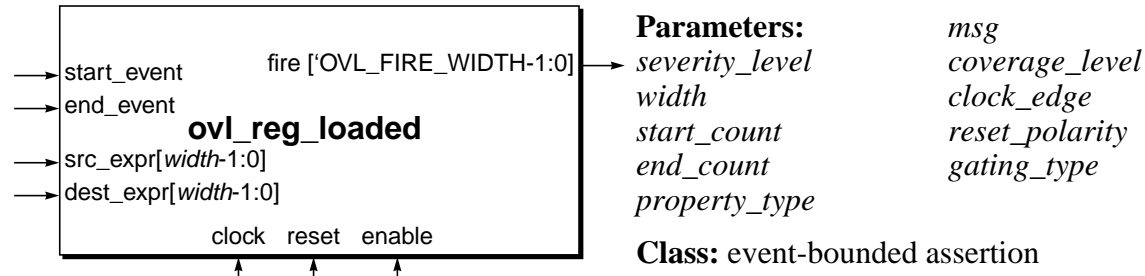
Syntax

```
ovl_reg_loaded
      [#(severity_level, width, start_count, end_count, property_type,
        msg, coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, start_event, end_event, src_expr,
      dest_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *src_expr* and *dest_expr* registers. Default: 4.
start_count	Number of cycles after *start_event* asserts that the time window opens. Default: 1.
end_count	Number of cycles after *start_event* asserts that the time window closes (if it is still open). If *end_count* is 0, only the *end_event* signal is used to define the time windows. Default: 10.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`start_event`	Start event signal for the reg_loaded check. If the time window is closed (or closing), the rising edge of *start_event* initiates a new check. The time window opens *start_count* cycles later.
`end_event`	End event signal for the reg_loaded check. If the time window is open (or opening), the rising edge of *end_event* terminates the current check, closes the window and issues a reg_loaded violation (if *dest_expr* loaded the value of *src_expr* in that cycle, the time window would be closing).
`src_expr[`*width*`-1:0]`	Source register containing the values that load the *dest_expr* register. For each reg_loaded check, the source value in *src_expr* is sampled in the same cycle that *start_event* asserts.
`dest_expr[`*width*`-1:0]`	Destination register for the values in *src_expr*.
`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_reg_loaded assertion checker checks *start_event* at each active edge of *clock.* If *start_event* has just transitioned to TRUE, the checker evaluates the source register (*src_expr*) and initiates a reg_loaded check to verify that this value gets loaded into the destination register (*dest_expr*) in the specified time window.

If *start_count* is 0, the time window opens immediately. Otherwise, the time window opens *start_count* cycles after the current cycle. The values of *dest_expr* in the cycles between the start of the reg_loaded check and the time window opening are not relevant. When the time window opens, the checker evaluates *dest_expr* and re-evaluates *dest_expr* each subsequent cycle. Once the value of *dest_expr* equals the captured value of *src_expr*, the current reg_loaded check terminates successfully. The time window closes when one of the following occur:

- The current cycle is *end_count* cycles after *start_event* asserted (*end_count* > 0).

- The *end_event* signal is TRUE.

If *dest_expr* has not loaded the *src_expr* value by the cycle the time window closes, a reg_loaded violation occurs.

Assertion Checks

REG_LOADED
Destination register did not equal the value of the source register in the specified time window.

end_count > 0

Either *end_event* became TRUE or *end_count* cycles passed after the rising edge of *start_event* and *dest_expr* was still not equal to the captured value of *src_expr* (ignoring values of *dest_expr* in the *start_count* cycles after *start_event* asserted).

Destination expression did not equal the value of the source expression in the time window that ended when 'end_event' asserted.

end_count = 0

End_event became TRUE after the rising edge of *start_event* and *dest_expr* was still not equal to the captured value of *src_expr* (ignoring values of *dest_expr* in the *start_count* cycles after *start_event* asserted).

Implicit X/Z Checks

start_event contains X or Z
Start event signal was X or Z.

end_event contains X or Z
End event signal was X or Z.

src_expr contains X or Z
Source expression contained X or Z bits.

dest_expr contains X or Z
Destination expression contained X or Z bits.

Cover Points

cover_values_checked
SANITY — Number of times a reg_loaded check was initiated (i.e., number of cycles *start_event* transitioned to TRUE).

cover_reg_loaded
BASIC — Number of times a reg_loaded check was terminated successfully (i.e, *dest_expr* was loaded with *src_expr* in the time window).

cover_end_in_window
BASIC — Number of time windows in which *end_event* asserted (whether or not *dest_expr* loaded *src_expr* in the window). Not meaningful if *end_count* = 0.

cover_no_end_in_window
BASIC — Number of time windows in which *end_event* did not assert (whether or not *dest_expr* loaded *src_expr* in the window). Not meaningful if *end_count* = 0.

cover_load_at_start_count
CORNER — Number of times *dest_expr* loaded *src_expr* exactly *start_count* cycles after *start_event* asserted.

`cover_load_at_stop_ count`	CORNER — Number of times *dest_expr* loaded *src_expr* exactly *end_count* cycles after *start_event* asserted. Not meaningful if *end_count* = 0.
`observed_load_times`	STATISTIC — Reports the load times (in cycles from asserting *start_event* to loading src_expr into *dest_expr*) that occurred at least once.

ovl_req_ack_unique

Ensures every request receives a corresponding acknowledge in a specified time window.

```
                    fire ['OVL_FIRE_WIDTH-1:0]
  ──▶ req
              ovl_req_ack_unique
  ──▶ ack


           clock  reset  enable
             ▲      ▲       ▲
```

Parameters:
severity_level
min_cks
max_cks
method
property_type

msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion
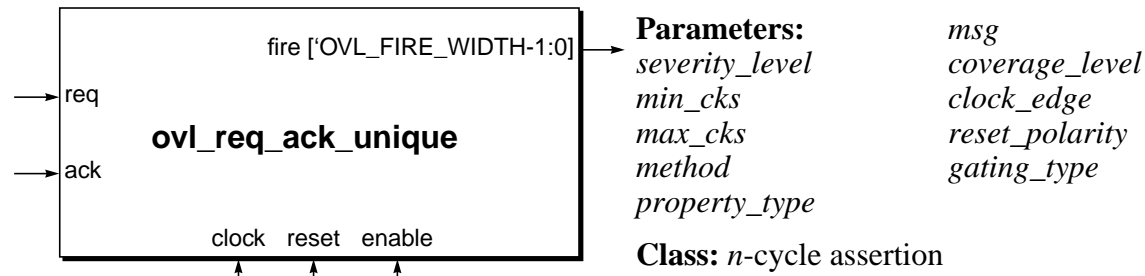
Syntax

```
ovl_req_ack_unique
      [#(severity_level, min_cks, max_cks, method, property_type, msg,
         coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, req, ack, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
min_cks	Minimum number of clock cycles after *req* asserts that its corresponding acknowledge can occur. Default: 1
max_cks	Maximum number of clock cycles after *req* asserts that its corresponding acknowledge can occur. Default: 15.
method	Method used to track and correlate request/acknowledge pairs.

method = 0 (Default)

    Method suitable for a short time window (*max_cks* ≤ 15). Uses internal IDs for requests. For each request, generates *max_cks* properties.

method = 1

    Method suitable for a long time window (*max_cks* > 15). Uses time stamps (computed mod 2 *max_cks*) to identify requests. To process an acknowledge, the time stamp for the request at the front of the queue is used to verify that the acknowledge meets timing requirements.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).

clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
req	Request signal.
ack	Acknowledgment signal.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_req_ack_unique assertion checker checks *req* and *ack* at each active edge of *clock*. If *req* is TRUE, a request becomes outstanding immediately. The checker tracks outstanding requests on a first-in first-out basis to verify the specified request/acknowledge handshake protocol is obeyed.

The protocol ensures each request has an acknowledgement that occurs in the time window that opens *min_cks* after the request (i.e., when the request becomes outstanding) and closes *max_cks* after the request. When *ack* is TRUE, the oldest outstanding request is checked. If this request has not been outstanding for at least *min_cks* cycles, the *ack* is ignored. Otherwise, the request is removed from the outstanding requests FIFO and "matched" with the current acknowledge. The checker detects the following violations:

- If *ack* is TRUE and no requests are outstanding, a no_extraneous_ack violation occurs.

- If a request is not acknowledged in its time window, an ack_timeout violation occurs.

- If max_cks requests are outstanding, additional requests cannot become outstanding. If a request occurs (without a simultaneous acknowledge), a max_outstanding_req violation occurs and the request is ignored.

To help collect coverage data, the checker tracks individual requests and their acknowledgements (up to the maximum outstanding requests limit, which is *max_cks* requests).

But the larger *max_cks* is, the greater the decrease in performance. To resolve this problem, the checker can be configured to a second method of tracking request/acknowledge pairs by setting the *method* parameter to 1. However with this method, the checker does not collect some coverage data.

Assertion Checks

NO_EXTRANEOUS_ACK	Acknowledge received when no requests were outstanding. No requests were outstanding and *ack* was TRUE (and if *min_cks* = 0, *req* was FALSE).
ACK_TIMEOUT	Acknowledge not received in time window. A request was pending for *max_cks* cycles and did not receive its acknowledge in the last cycle of its time window.
MAX_OUTSTANDING_REQ	Maximum number of requests were outstanding when an additional request was issued. *Req* was TRUE and *ack* was FALSE, but *max_cks* requests were outstanding.

Implicit X/Z Checks

req contains X or Z	Request signal was X or Z.
ack contains X or Z	Acknowledge signal was X or Z.

Cover Points

cover_requests	SANITY — Number of cycles *req* asserted.
cover_acknowledgements	SANITY — Number of cycles *ack* asserted.
cover_ack_at_min_cks	CORNER — Number of times acknowledge was received *min_cks* cycles after its request was issued. Not meaningful if *method* = 1.
cover_ack_at_max_cks	CORNER — Number of times acknowledge was received *max_cks* cycles after its request was issued. Not meaningful if *method* = 1.
observed_ack_times	STATISTIC — Reports the request-to-acknowledge times (in cycles) that occurred at least once. Not meaningful if *method* = 1.
observed_outstanding_ requests	STATISTIC — Reports the number of cycles in which exactly *index* requests become outstanding, for each *index* in the range [0: *max_cks*] (except for index = 0, which counts all cycles that no request was outstanding). Not meaningful if *method* = 1.

ovl_req_requires

Ensures that every request event initiates a valid request-response event sequence that finishes within a specified time window.

```
                                                    Parameters:
  ──→  req_trigger      fire ['OVL_FIRE_WIDTH-1:0] ──→  severity_level       coverage_level
  ──→  req_follower                                    min_cks              clock_edge
            ovl_req_requires                           max_cks              reset_polarity
  ──→  resp_leader                                     property_type        gating_type
  ──→  resp_trigger                                    msg

            clock  reset  enable                    Class: n-cycle assertion
             ↑      ↑      ↑
```
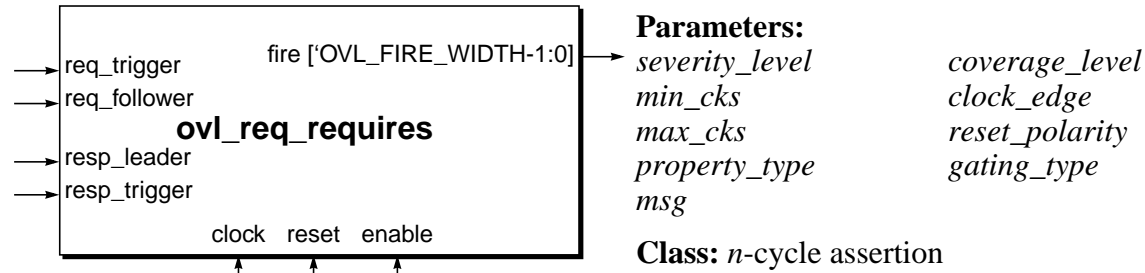
Syntax

```
ovl_req_requires
      [#(severity_level, min_cks, max_cks, property_type, msg,
         coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, req_trigger, req_follower,
      resp_leader, resp_trigger, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
min_cks	Minimum number of clock cycles after *req_trigger* is TRUE that the event sequence can finish. Value of *min_cks* must be > 0. Default: 1.
max_cks	Maximum number of clock cycles after *req_trigger* is TRUE that the event sequence should finish. The special value 0 selects no upper bound. If *max_cks* ≠ 0, then *max_cks* must be *min_cks*. Default: 0.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).

gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
req_trigger	Request trigger signal. If *req_trigger* is TRUE, the checker initiates a new check and its corresponding time window opens *min_cks* cycles later.
req_follower	Request follower signal. A request event finishes at the first rising edge of *req_follower* after *req_trigger* was TRUE.
resp_leader	Response leader signal. The first rising edge of *resp_leader* in a cycle after the request event initiates the response event.
resp_trigger	Response trigger signal. The response event finishes at the first rising edge of *resp_trigger* in the same or subsequent cycle as the rising edge of *resp_leader*. This event must be in the time window from *min_cks* to *max_cks* cycles after *req_trigger* was TRUE.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_req_requires assertion checker checks *req_trigger* at each active edge of *clock*. If *req_trigger* is TRUE, a req_requires check is initiated. The checker verifies that a semaphore request-response event sequence transpires with the last event occurring within the time window specified by [*max_cks*:*min_cks*]. The event sequence must have the following characteristics:

- When *req_trigger* is TRUE: *req_follower*, *resp_leader*, *resp_trigger* are TRUE in sequence.

- Each event happens at the active clock edge at which the first occurrence of its signal is TRUE following the previous event in the sequence.

- The sequence has the following timing relations:

$$t_{req\_trigger} \leq t_{req\_follower} < t_{resp\_leader} \leq t_{resp\_trigger}$$

That is, the *req_trigger* and *req_follower* events can occur in the same cycle and the *resp_leader* and *resp_trigger* events can occur in the same cycle, but the *resp_leader* event must be after the *req_follower* event.

A req_requires check violation occurs if one of the following cases arises:

- The semaphore event sequence finishes before the [*min_cks*:*max_cks*] time window opens.

- A cycle is reached at which the checker determines the semaphore event sequence cannot finish within the [*min_cks*:*max_cks*] time window.

- The [*min_cks*:*max_cks*] time window closes, but the semaphore event sequence did not finish.

The default value of *max_cks* is 0, which sets no upper bound for the time windows. In this case, a req_requires violation occurs only when a sequence finishes before *min_cks* cycles after the *req_trigger* event. The default value of *min_cks* is 1, so if both *min_cks* and *max_cks* are left set to their defaults, the req_requires check cannot be violated.

Assertion Checks

REQ_REQUIRES	A request-response event sequence started, but did not finish when the specified time window was open.
	max_cks > 0
	Req_trigger was TRUE, so a request-response event sequence started. But, either the sequence finished before *min_cks* cycles, or it could not finish by *max_cks* cycles.
	A request-response event sequence started, but it finished before the specified time window opened.
	max_cks = 0
	Req_trigger was TRUE, so a request-response event sequence started, but the sequence finished before *min_cks* cycles.

Implicit X/Z Checks

req_trigger contains X or Z	Request trigger was X or Z.
req_follower contains X or Z	Request follower was X or Z.
resp_leader contains X or Z	Response leader was X or Z.
resp_trigger contains X or Z	Response trigger was X or Z.

Cover Points

If overlapping request-response sequences are triggered, the coverage data might be inaccurate because the cover group vectors do not reflect which responses belong to which requests.

`cover_requests`	SANITY — Number of cycles *req_trigger* was TRUE.
`cover_request_ followers`	BASIC — Number of times *req_trigger* was TRUE and *req_follower* was TRUE in the same or subsequent cycle.
`cover_response_leaders`	BASIC — Number of times *req_trigger* was TRUE; *req_follower* was TRUE in the same or subsequent cycle; and then *resp_leader* was TRUE in a subsequent cycle.
`cover_req_requires`	BASIC — Number of valid request-response event sequences.
`cover_resp_trigger_at_ min_cks`	CORNER — Number of valid request-response event sequences that finished in *min_cks* cycles.
`cover_resp_trigger_at_ max_cks`	CORNER — Number of valid request-response event sequences that finished in *max_cks* cycles.
`cover_req_trigger_to_ resp_trigger`	STATISTIC — Reports the request-trigger to response-trigger times (in cycles) that occurred at least once.
`cover_req_trigger_to_ req_follower`	STATISTIC — Reports the request-trigger to request-follower times (in cycles) that occurred at least once.
`cover_req_follower_to_ resp_leader`	STATISTIC — Reports the request-follower to response-leader times (in cycles) that occurred at least once.
`cover_resp_leader_to_ resp_trigger`	STATISTIC — Reports the response-leader to response-trigger times (in cycles) that occurred at least once.

ovl_stack

Ensures the data integrity of a stack and ensures that the stack does not overflow or underflow.

```
                ┌──────────────────────────────────┐
  ──→  push     │                                  │
  ──→  pop      │                                  │
  ──→  full     │  fire ['OVL_FIRE_WIDTH-1:0]  ──→  │
  ──→  empty    │          ovl_stack               │
  ──→  push_data[width-1:0]                         │
  ──→  pop_data[width-1:0]                          │
                │    clock   reset   enable        │
                └──────────────────────────────────┘
                        ↑       ↑       ↑
```

Parameters:

severity_level	*property_type*
width	*msg*
depth	*coverage_level*
push_latency	*clock_edge*
pop_latency	*reset_polarity*
high_water_mark	*gating_type*

Class: *n*-cycle assertion
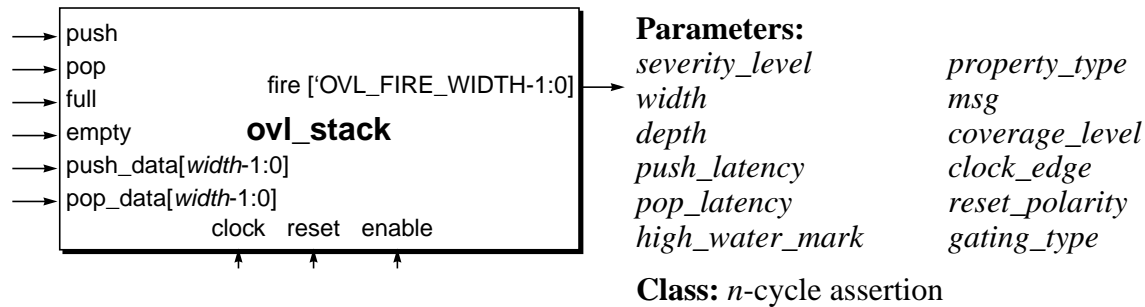
Syntax

```
ovl_stack
    [#(severity_level, width, depth, push_latency, pop_latency,
        high_water_mark, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
  instance_name (clock, reset, enable, push, pop, full, empty, push_data,
    pop_data, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of a data item. Default: 1.
depth	Stack depth. The *depth* must be > 0. Default: 2.
push_latency	Latency for push operation. *push_latency* = 0 (Default) Value of *push_data* is valid and the push operation is performed in the same cycle *push* asserts. *push_latency* > 0 Value of *push_data* is valid and the push operation is performed *push_latency* cycles after *push* asserts.
pop_latency	Latency for pop operation. *pop_latency* = 0 (Default) Value of *pop_data* is valid and the pop operation is performed in the same cycle *pop* asserts. *pop_latency* > 0 Value of *pop_data* is valid and the pop operation is performed *pop_latency* cycles after *pop* asserts.
high_water_mark	Stack high-water mark. Must be < *depth*. A value of 0 disables the cover_high_water_mark cover point. Default: 0.

property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
push	Stack push input. When *push* asserts, the stack performs a push operation. A data item is pushed onto the stack and the stack counter increments by 1. If *push_latency* is 0, the push is performed in the same cycle *push* asserts. Otherwise *push_latency* cycles later, *push_data* is latched, the push operation occurs, and the stack counter increments.
pop	Stack pop input. When *pop* asserts, the stack performs a pop operation. A data item is popped from the stack and the stack counter decrements by 1. If *deq_latency* is 0, the pop is performed in the same cycle *pop* asserts. Otherwise *enq_latency* cycles later, the pop operation occurs, the stack counter decrements, and *pop_data* is valid.

full	Output status flag from the stack.
	full = 0
	    Stack not full.
	full = 1
	    Stack full.

empty	Output status flag from the stack.
	empty = 0
	    Stack not empty.
	empty = 1
	    Stack empty.

push_data[width-1:0]	Push data input to the stack. Contains the data item to push onto the stack.
pop_data[width-1:0]	Pop data output from the stack. Contains the data item popped from the stack.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_stack checker checks *push* and *pop* at the active edge of *clock*. If *push* is TRUE, the checker assumes a push operation occurs *push_latency* cycles later (or in the same cycle if *push_latency* is 0). *In that cycle*, the checker does the following:

- If a pop operation is scheduled for this cycle, a simultaneous_push_pop check violation occurs.

- Otherwise, if the stack is already full, an overflow check violation occurs. The checker assumes the data item in *push_data* was latched in the current cycle and replaced the top entry.

- Otherwise, the checker assumes the data item in *push_data* was latched in the current cycle and pushed on the top of the stack. The checker increments the stack counter by 1 in the next cycle.

Similarly, if *pop* is TRUE, the checker assumes a pop operation occurs *pop_latency* cycles later (or in the same cycle if *pop_latency* is 0). *In that cycle*, unless a simultaneous_push_pop violation has occurred, the checker does the following:

- If the stack is already empty, an underflow check violation occurs.

- Otherwise, the checker assumes the data item on the top of the stack was popped and compares the value of *pop_data* with the expected value of the popped data item. If they do not match, a value check violation occurs. The checker decrements the stack counter by 1 in the next cycle.

The ovl_stack checker also checks *full* and *empty* at the active edge of *clock*. After the stack pointer is adjusted to reflect a push or pop performed in the previous cycle:

- If the stack is full and *full* is FALSE or if the stack is not full and *full* is TRUE, a full check violation occurs.

- If the stack is empty and *empty* is FALSE or if the stack is not empty and *empty* is TRUE, an empty check violation occurs.

Assertion Checks

OVERFLOW
Data pushed onto stack when the stack was full.
Stack had depth data items push_latency cycles after push was sampled TRUE.

UNDERFLOW
Data popped from stack when the stack was empty.
Stack was empty pop_latency cycles after pop was sampled TRUE.

SIMULTANEOUS_PUSH_POP
Push and pop operations occurred together.
A push operation and a pop operation were both scheduled for the same cycle.

VALUE
Data value popped from the stack did not match the corresponding data value pushed onto the stack.
Pop was sampled TRUE, but pop_latency cycles later the value of pop_data did not equal the expected value pushed onto the stack in a previous cycle.

FULL
Stack was empty, but 'empty' was deasserted.
Empty was sampled FALSE when the stack was empty.

Stack was not empty, but 'empty' was asserted.
Empty was sampled TRUE when the stack was not empty.

EMPTY
Stack was full, but 'full' was deasserted.
Full was sampled FALSE when the stack was full.

Stack was not full, but 'full' was asserted.
Full was sampled TRUE when the stack was not full.

Implicit X/Z Checks

push contains X or Z
Push signal was X or Z.

pop contains X or Z
Pop signal was X or Z.

push_data contains X or Z
Push data contained X or Z bits.

pop_data contains X or Z
Pop data contained X or Z bits.

full contains X or Z
Full signal was X or Z.

empty contains X or Z
Empty signal was X or Z.

Cover Points

`cover_pushes`	SANITY — Number of cycles *push* was asserted.
`cover_pops`	SANITY — Number of cycles *pop* was asserted.
`cover_max_entries`	BASIC — Number of cycles for which the number of data items in the stack was the same as the maximum number of data items the stack had held up to and including that cycle.
`cover_push_then_pop`	BASIC — Number of times a *push* was followed by a *pop* without an intervening *push* (or *pop*).
`cover_full`	CORNER — Number of times a push incremented the stack pointer to *depth* data items.
`cover_empty`	CORNER — Number of times a pop decremented the stack pointer to 0 data items.
`cover_high_water_mark`	CORNER — Number of times the stack had more data items than the specified *high_water_mark*. Not meaningful if *high_water_mark* is 0.

ovl_time

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

Parameters: *msg*
severity_level   *coverage_level*
num_cks   *clock_edge*
action_on_new_start   *reset_polarity*
property_type   *gating_type*

Class: *n*-cycle assertion
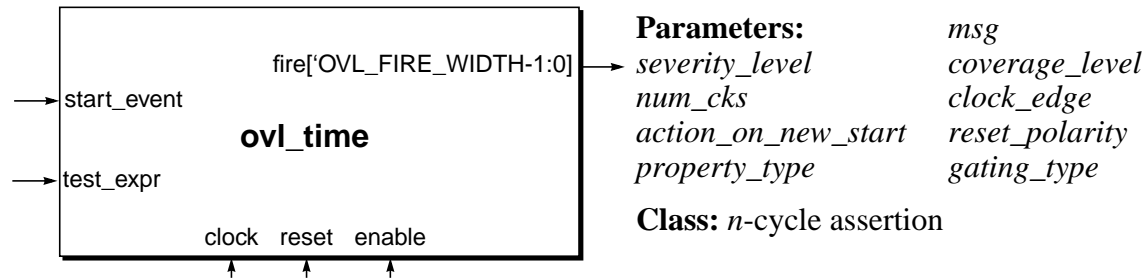
Syntax

```
ovl_time
    [#(severity_level, num_cks, action_on_new_start, property_type,
        msg, coverage_level, clock_edge, reset_polarity, gating_type)]
    instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).

`gating_type`	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`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.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_time assertion checker checks the expression *start_event* at each active edge of *clock* to determine whether or not to initiate a check. Once initiated, the check evaluates *test_expr* each active edge of *clock* 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

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

ovl_change
ovl_next
ovl_frame
ovl_unchange

ovl_win_change
ovl_win_unchange
ovl_window

Examples

```
ovl_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_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_transaction (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        req == 1,                            // start_event
        ptr >= 1 && ptr <= 3,                // test_expr
        fire_valid_transaction );            // fire
```

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 *clock*. 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).

TIME Error: invalid transaction

```
ovl_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_DEFAULT,                        // coverage_level
    'OVL_POSEDGE,                              // clock_edge
    'OVL_ACTIVE_LOW,                           // reset_polarity
    'OVL_GATE_CLOCK)                           // gating_type

    valid_transaction (

        clock,                                 // clock
        reset,                                 // reset
        enable,                                // enable
        req == 1,                              // start_event
        ptr >= 1 && ptr <= 3,                  // test_expr
        fire_valid_transaction );              // fire
```

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 *clock*. 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).

```
ovl_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_DEFAULT,                            // coverage_level
    'OVL_POSEDGE,                                  // clock_edge
    'OVL_ACTIVE_LOW,                               // reset_polarity
    'OVL_GATE_CLOCK)                               // gating_type

    valid_transaction (

        clock,                                     // clock
        reset,                                     // reset
        enable,                                    // enable
        req == 1,                                  // start_event
        ptr >= 1 && ptr <= 3,                      // test_expr
        fire_valid_transaction );                  // fire
```
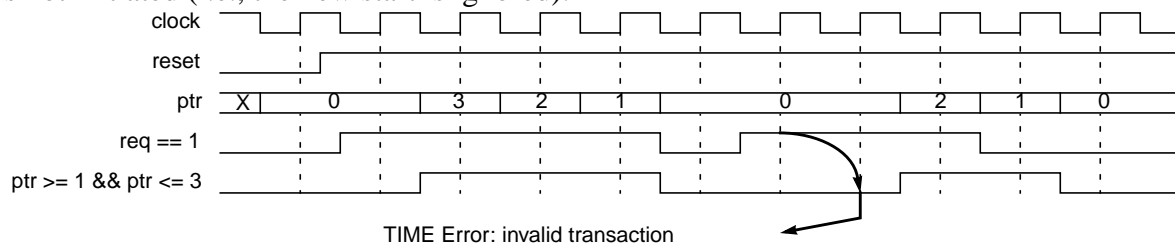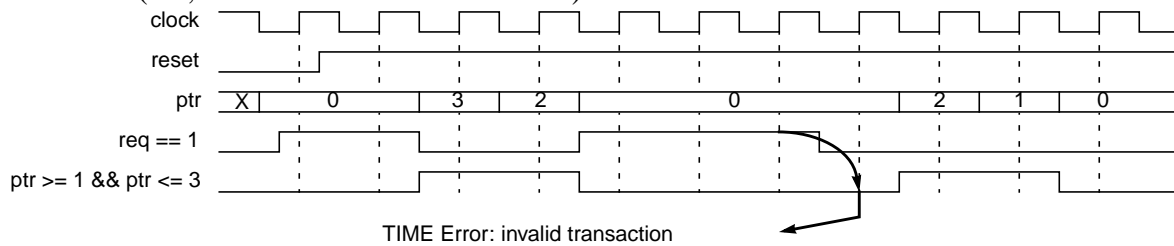
Ensures that *ptr* is sampled in the range 1 to 3 for three cycles after *req* is sampled equal to 1 at the rising edge of *clock*. If *req* is sampled equal to 1 when the checker samples *ptr*, the checker issues an *illegal start event* violation and does not start a new check.
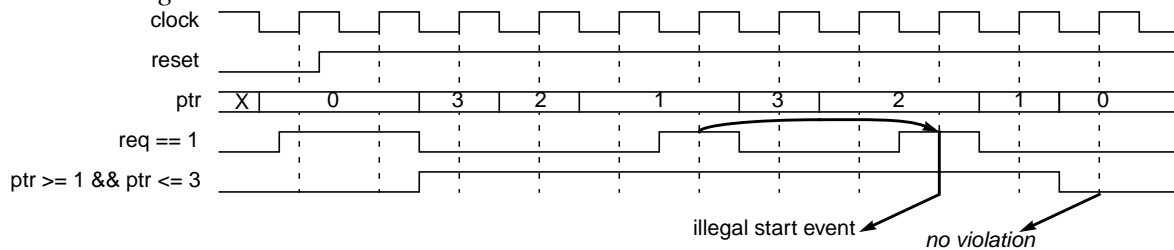
ovl_transition

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

```
                    fire ['OVL_FIRE_WIDTH-1:0]
  ── test_expr[width-1:0]
                ovl_transition
  ── start_state[width-1:0]
  ── next_state[width-1:0]
                clock  reset  enable
```

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
property_type	*reset_polarity*
msg	*gating_type*

Class: 2-cycle assertion
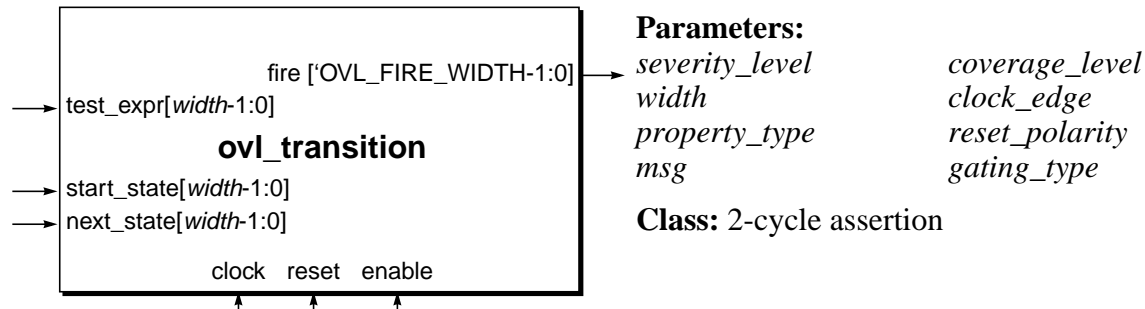
Syntax

```
ovl_transition
    [#(severity_level, width, property_type, msg, coverage_level,
       clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, start_state,
    ovl_transitionnext_state, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`test_expr[width-1:0]`	Expression that should transition to *next_state* on the active edge of *clock* if its value at the previous active edge of *clock* 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 active edge of *clock*, 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 active edge of *clock*, then the value of *test_expr* should equal *next_state* on the current active edge of *clock*.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_transition assertion checker checks the expression *test_expr* and *start_state* at each active edge of *clock* to see if they are the same. If so, the checker evaluates and stores the current value of *next_state*. At the next active edge of *clock*, 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

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 *clock* after *reset* deasserts.

See also

ovl_no_transition

Example

```
ovl_transition #(

    `OVL_ERROR,                             // severity_level
    3,                                      // width
    `OVL_ASSERT,                            // property_type
    "Error: bad count transition",          // msg
    `OVL_COVER_DEFAULT,                     // coverage_level
    `OVL_POSEDGE,                           // clock_edge
    `OVL_ACTIVE_LOW,                        // reset_polarity
    `OVL_GATE_CLOCK)                        // gating_type

    valid_count (

        clock,                              // clock
        reset,                              // reset
        enable,                             // enable
        count,                              // test_expr
        3'd3,                               // start_state
        (sel_8 == 1'b0) ? 3'd0 : 3'd4,      // next_state
        fire_valid_count );                 // fire
```

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.

TRANSITION Error: bad count transition
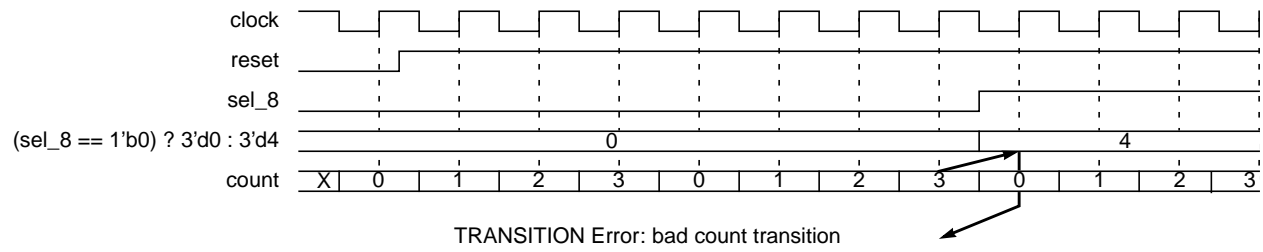
ovl_unchange

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

```
                    fire['OVL_FIRE_WIDTH-1:0]
    start_event
              ovl_unchange

    test_expr[width-1:0]


         clock  reset  enable
```

Parameters:

severity_level   *msg*
width   *coverage_level*
num_cks   *clock_edge*
action_on_new_start   *reset_polarity*
property_type   *gating_type*

Class: *n*-cycle assertion
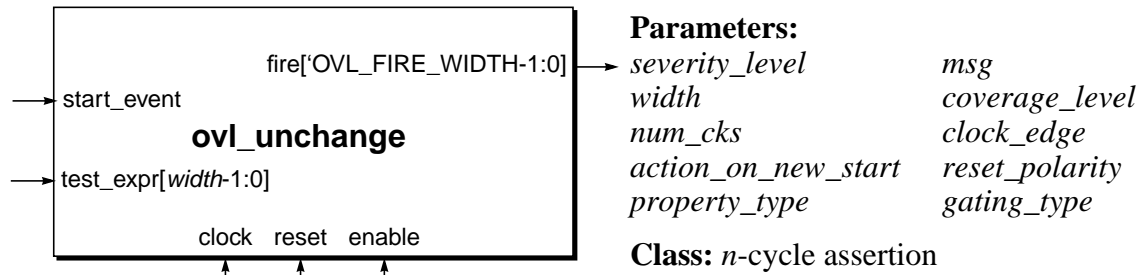
Syntax

```
ovl_unchange
      [#(severity_level, width, num_cks, action_on_new_start,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

`reset_polarity`	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
`gating_type`	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

`clock`	Clock event for the assertion.
`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`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.
`fire ['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_unchange assertion checker checks the expression *start_event* at each active edge of *clock* 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* active edges of *clock*. 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

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

ovl_change   ovl_win_unchange
ovl_time   ovl_window
ovl_win_change

Examples

```
ovl_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_DEFAULT,                      // coverage_level
   `OVL_POSEDGE,                            // clock_edge
   `OVL_ACTIVE_LOW,                         // reset_polarity
   `OVL_GATE_CLOCK)                         // gating_type

   valid_div_unchange_a (

      clock,                               // clock
      reset,                               // reset
      enable,                              // enable
      start == 1,                          // start_event
      a,                                   // test_expr
      fire_ );                             // fire
```

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

UNCHANGE Error: a changed during divide

```
ovl_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_DEFAULT,                      // coverage_level
   `OVL_POSEDGE,                            // clock_edge
   `OVL_ACTIVE_LOW,                         // reset_polarity
   `OVL_GATE_CLOCK)                         // gating_type

   valid_div_unchange_a (

      clock,                               // clock
      reset,                               // reset
      enable,                              // enable
      start == 1,                          // start_event
      a,                                   // test_expr
      fire_valid_div_unchange_a );         // fire
```

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

UNCHANGE Error: a changed during divide
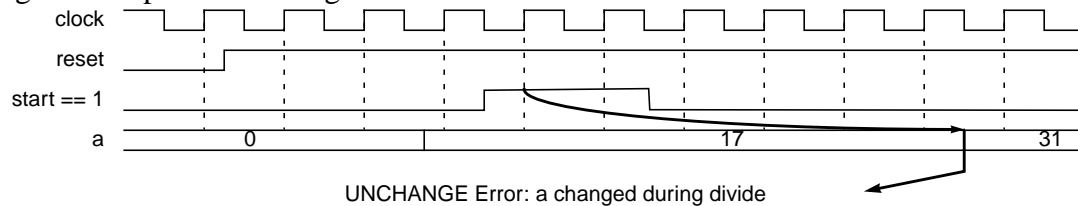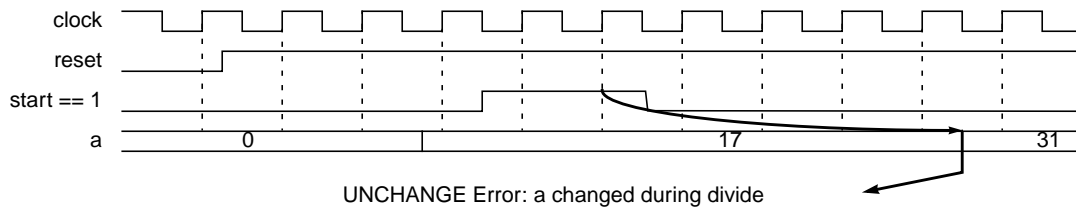
```
ovl_unchange #(

  '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_DEFAULT,                         // coverage_level
  'OVL_POSEDGE,                               // clock_edge
  'OVL_ACTIVE_LOW,                            // reset_polarity
  'OVL_GATE_CLOCK)                            // gating_type

  valid_div_unchange_a (

    clock,                                    // clock
    reset,                                    // reset
    enable,                                   // enable
    start == 1,                               // start_event
    a,                                        // test_expr
    fire_valid_div_unchange_a );              // fire
```
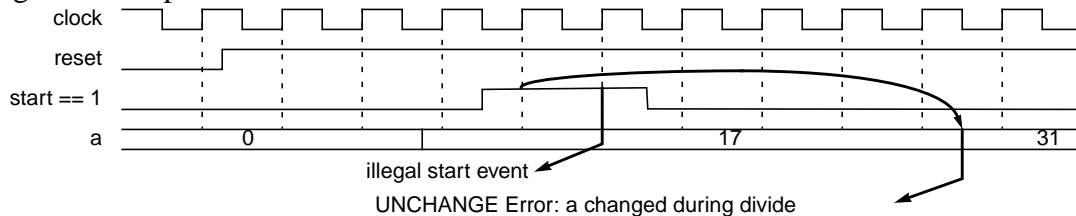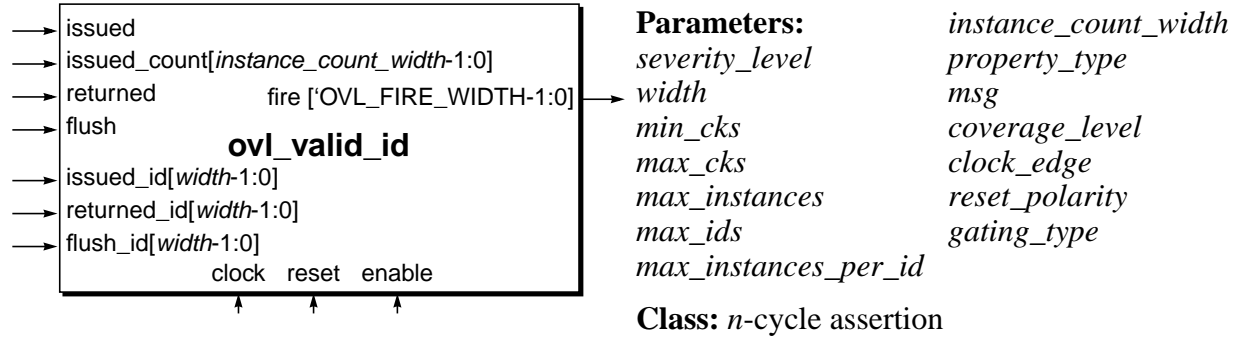
Ensures that *a* remains unchanged while a divide operation is performed (8 cycles). A restart during a divide operation is a violation.

illegal start event

UNCHANGE Error: a changed during divide

ovl_valid_id

Ensures that each issued ID is returned within a specified time window; that returned IDs match issued IDs; and that the issued and outstanding IDs do not exceed specified limits.

```
→ issued
→ issued_count[instance_count_width-1:0]
→ returned          fire ['OVL_FIRE_WIDTH-1:0] →
→ flush
              ovl_valid_id
→ issued_id[width-1:0]
→ returned_id[width-1:0]
→ flush_id[width-1:0]
         clock   reset   enable
           ↑       ↑        ↑
```

Parameters: *instance_count_width*
severity_level   *property_type*
width   *msg*
min_cks   *coverage_level*
max_cks   *clock_edge*
max_instances   *reset_polarity*
max_ids   *gating_type*
max_instances_per_id

Class: *n*-cycle assertion

Syntax

```
ovl_valid_id
    [#(severity_level, width, min_cks, max_cks, max_instances, max_ids,
       max_instances_per_id, instance_count_width, property_type, msg,
       coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, issued, issued_count, returned,
     flush, issued_id, returned_id, flush_id, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *issued_id, returned_id* and flush_id. Default: 2.
min_cks	Minimum number of clock cycles an ID instance must be outstanding. Must be > 0. Default: 1
max_cks	Maximum number of clock cycles an ID instance can be outstanding. Must be ≥ *min_cks*. Default: 1.
max_instances	Maximum number of ID instances that can be outstanding at any time. Default: 2.
max_ids	Maximum number of different IDs that can be outstanding at any time. Default: 1.
max_instances_per_id	Maximum number of instances of a single ID that can be outstanding at any time. Default: 1.
instance_count_width	Width of *issued_count*. Default: 2.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).

msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock*, if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
issued	Issued IDs signal indicating the ID in *issued_id* is added to the outstanding IDs list. The *issued_count* port specifies the number of instances of the ID to make outstanding.
issued_count [*instance_count_width*-1:0]	Number of instances of the issued ID to make outstanding when *issued* asserts.
returned	Returned ID signal indicating an instance of the ID in *returned_id* is removed from the outstanding IDs list.
flush	Flush ID signal indicating all instances of the ID in *flush_id* are removed from the outstanding IDs list.
issued_id[*width*-1:0]	Expression or variable containing the ID to add to the outstanding IDs list if *issued* is TRUE.
returned_id[*width*-1:0]	Expression or variable containing the ID of an instance returned and removed from the outstanding IDs list if *returned* is TRUE.
flush_id[*width*-1:0]	Expression or variable containing the ID to flush if *flush* is TRUE. All instances of the ID are removed from the outstanding IDs list.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_valid_id assertion checker checks *flush, returned* and *issued* at each active edge of *clock* and performs the following sequence of operations using an internal scratch pad of outstanding IDs:

1. If *flush* is TRUE, the ID specified in *flush_id* is compared to the outstanding IDs. All instances (if any) of the flush ID are removed from the list of outstanding IDs. If *returned* is TRUE and *flush_id = returned_id*, the returned instance is ignored (even if it was not previously outstanding or was outstanding longer that *max_cks*). If *issued* is TRUE and *flush_id = issued_id*, the issued ID instances are flushed as well (even if one of the outstanding IDs, instances or instances-per-ID limits for the issued ID instance were reached).

2. If *returned* is TRUE and the ID in *returned_ID* is not being flushed:

    a. If an instance of the returned ID is outstanding, the longest-outstanding instance of the returned ID is removed from the list of outstanding ID instances. If that ID instance was outstanding for fewer than *min_cks* cycles, a min_cks violation occurs.

    b. If no instance of the returned ID is outstanding, a returned_id violation occurs. Even if an instance of the returned ID were issued in the same cycle, all ID instances must be outstanding for *min_cks* cycles (and *min_cks* must be 1). In particular, the same ID instance cannot be issued and returned in the same cycle.

3. If *issued* is TRUE and *issued_count* is 0, an issued_count violation occurs.

4. If *issued* is TRUE and *issued_count* > 0, then:

    a. If the current number of unique outstanding IDs is *max_ids* and issued_id is not one of them, a max_instances violation occurs.

    b. If the current number of outstanding ID instances plus *issued_count* exceeds *max_instances*, a max_ids violation occurs.

    c. If the current number of outstanding instances of the issued ID plus *issued_count* exceeds *max_instances_per_id*, a max_instances_per_id violation occurs.

    d. If the none of these violations occur, *issued_count* instances of the ID in *issued_id* are added to the list of outstanding ID instances.

5. After flushing and returning IDs, if any IDs have been outstanding for *max_cks* cycles, a max_cks violation occurs in the next cycle.

Assertion Checks

RETURNED_ID   Returned ID not outstanding.
                                 Returned is TRUE, but the list of outstanding ID instances
                                 does not contain an instance of *returned_ID*.

MAX_CKS	ID instance outstanding for too many cycles. An ID instance was outstanding longer than *max_cks* cycles.
MIN_CKS	ID instance returned in too few cycles. *Returned* is TRUE and an instance of the ID in *returned_id* is outstanding, but the longest-outstanding instance of the ID has been outstanding for fewer than *min_cks* cycles.
MAX_IDS	Maximum number of outstanding IDs or ID instances exceeded. *Issued* is TRUE, but the number of outstanding instances plus *issued_count* (minus 1 if an instance of *issued_id* is returned without error) exceeds *max_instances* or the number of unique outstanding IDs plus *issued_count* (minus 1 if an instance of *issued_id* is returned without error) exceeds *max_ids*.
MAX_INSTANCES_PER_ID	Maximum number of outstanding ID instances for the issued ID exceeded. *Issued* is TRUE, but the number of outstanding instances of *issued_id* plus *issued_count* (minus 1 if an instance of *issued_id* is returned without error) exceeds *max_instances_per_id*.
ISSUED_COUNT	ID issued with count 0. *Issued* is TRUE, but *issued_count* is 0.

Implicit X/Z Checks

issued contains X or Z	Issued signal was X or Z.
returned contains X or Z	Returned signal was X or Z.
flush contains X or Z	Flush signal was X or Z.
issued_id contains X or Z when issued is asserted	Issued ID contained X or Z bits.
ret_id contains X or Z when returned is asserted	Returned ID contained X or Z bits.
flush_id contains X or Z when flush is asserted	Flush ID contained X or Z bits.

Cover Points

`cover_issued_asserted`	SANITY — Number of cycles *issued* was TRUE.
`cover_returned_asserted`	SANITY — Number of cycles *returned* was TRUE.
`cover_flush_asserted`	SANITY — Number of cycles *flush* was TRUE.
`turnaround_times`	BASIC — Reports the turnaround times (i.e., number of cycles after an ID instance is issued that the instance is returned) that occurred at least once.
`outstanding_ids`	BASIC — Reports the numbers of outstanding ID instances that occurred at least once.
`cover_returned_at_min_cks`	CORNER — Number of times the returned ID instance was outstanding for *min_cks* cycles.
`cover_returned_at_max_cks`	CORNER — Number of times the returned ID instance was outstanding for *max_cks* cycles.
`cover_max_ids`	CORNER — Number of cycles the outstanding IDs reached the *max_ids* limit or the *max_instances* limit.
`cover_max_instances_per_id`	CORNER — Number of cycles the outstanding instances of an ID reached the *max_instances_per_id* limit.

ovl_value

Ensures that the value of an expression either matches a value in a specified list or does not match any value in the list (as determined by a mode signal).

```
                    fire ['OVL_FIRE_WIDTH-1:0]

  test_expr[width-1:0]
  disallow
                    ovl_value

  vals[num_values*width-1:0]


       clock   reset   enable
```

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
num_values	*reset_polarity*
property_type	*gating_type*
msg	

Class: 1-cycle assertion
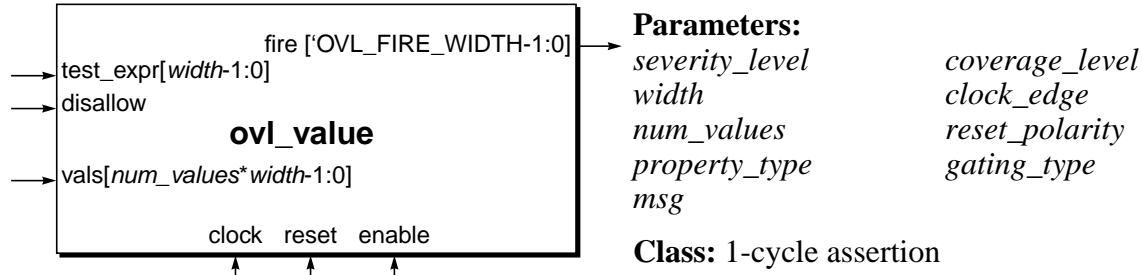
Syntax

```
ovl_value
    [#(severity_level, width, num_values, property_type, msg,
        coverage_level, clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, vals, disallow, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of *test_expr*. Default: 1.
num_values	Number of values in *vals*. Must be ≥ 1. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
test_expr[width-1:0]	Variable or expression to check.
vals [num_values*width-1:0]	Concatenated list of values for *test_expr*.

disallow	Sense of the comparison of *test_expr* with *vals*.
	 disallow = 0
	    Value of *test_expr* should match one of the values in *vals*.
	 disallow = 1
	    Value of *test_expr* should not match one of the values in *vals*.

fire ['OVL_FIRE_WIDTH*-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_value assertion checker checks *test_expr, vals* and *disallow* at each active edge of *clock* (except for the first cycle after a checker reset). The value of *test_expr* is compared with the list of values in *vals*. If *disallow* is FALSE and the value of *test_expr* is not a value in *vals*, a value check violation occurs. Similarly, if *disallow* is TRUE and the value of *test_expr* is one of the values in *vals*, an is_not check violation occurs. The check occurs at the active clock edge, .

Assertion Checks

VALUE	Expression value did not equal one of the specified values. Value of the *test_expr* did not match a value in *vals*, but *disallow* was FALSE.
IS_NOT	Expression value was equal to one of the specified values. Value of the *test_expr* matched one of the values in *vals*, but *disallow* was TRUE.

Implicit X/Z Checks

`test_expr contains X or Z`	Expression contained X or Z bits.
`vals contains X or Z`	Values contained X or Z bits.
`disallow contains X or Z`	Disallow signal was X or Z.

Cover Points

`cover_values_checked`	SANITY — Number of cycles *test_expr* loaded a new value.
`cover_in_vals`	BASIC — Number of cycles *disallow* was FALSE and the value of *test_expr* matched a value in *vals*.
`cover_not_in_vals`	BASIC — Number of cycles *disallow* was TRUE and the value of *test_expr* did not match a value in *vals*.
`cover_values_covered`	BASIC — Reports the values in *vals* that were covered at least once. Not applicable for cycles where *disallow* = 1.

ovl_width

Ensures that when value of an 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.

```
                              fire['OVL_FIRE_WIDTH-1:0]

        test_expr         ovl_width

                      clock  reset  enable
```

Parameters: *msg*
severity_level   *coverage_level*
min_cks   *clock_edge*
max_cks   *reset_polarity*
property_type   *gating_type*

Class: *n*-cycle assertion
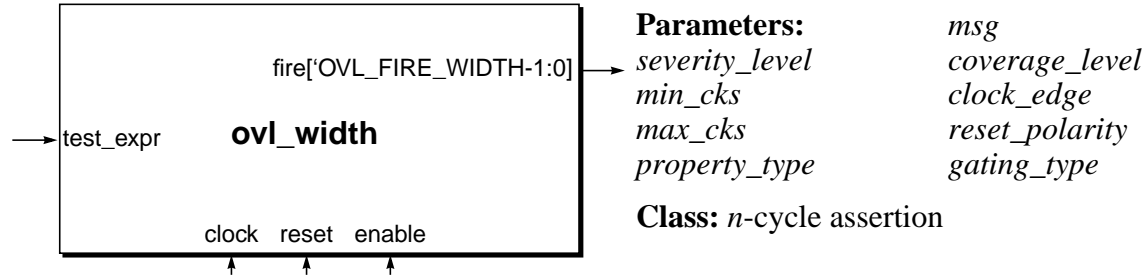
Syntax

```
ovl_width
     [#(severity_level, min_cks, max_cks, property_type, msg,
        coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_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_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).

reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_width assertion checker checks the single-bit expression *test_expr* at each active edge of *clock*. 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 active edge of *clock*. 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 active edge of *clock*. 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

ovl_change   ovl_unchange
ovl_time

Example

```
ovl_width #(

    'OVL_ERROR,                              // severity_level
    2,                                       // min_cks
    3,                                       // max_cks
    'OVL_ASSERT,                             // property_type
    "Error: invalid request",                // msg
    'OVL_COVER_DEFAULT,                      // coverage_level
    'OVL_POSEDGE,                            // clock_edge
    'OVL_ACTIVE_LOW,                         // reset_polarity
    'OVL_GATE_CLOCK)                         // gating_type

    valid_request (

        clock,                               // clock
        reset,                               // reset
        enable,                              // enable
        req == 1,                            // test_expr
        fire_valid_request );                // fire
```

Ensures *req* asserts for 2 or 3 cycles.
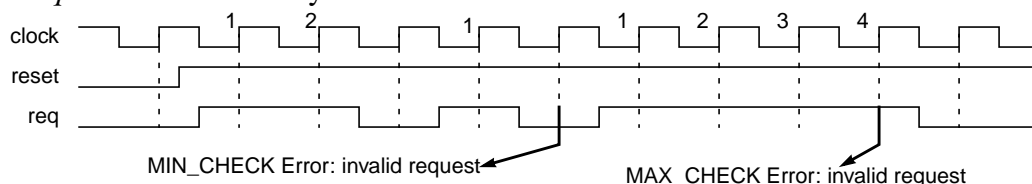
ovl_win_change

Ensures that the value of an expression changes in a specified window between a start event and an end event.
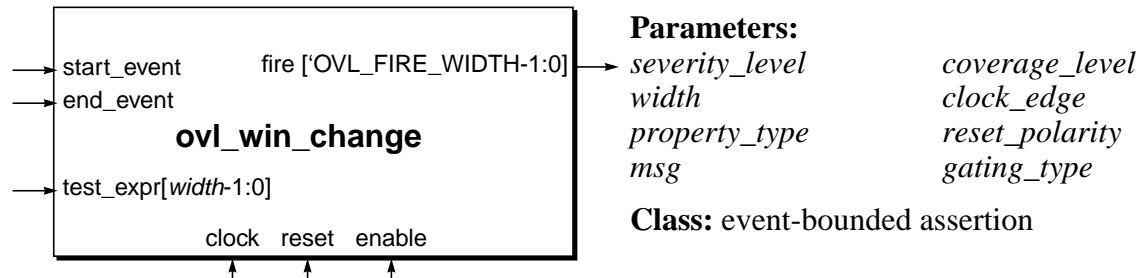
```
          start_event              fire ['OVL_FIRE_WIDTH-1:0]
          end_event
                     ovl_win_change

          test_expr[width-1:0]

                  clock   reset   enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: event-bounded assertion

Syntax

```
ovl_win_change
      [#(severity_level, width, property_type, msg, coverage_level,
         clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, start_event, test_expr, end_event,
      fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_win_change assertion checker checks the expression *start_event* at each active edge of *clock* 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 active edge of *clock*, 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 active edge of *clock* 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

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

ovl_change   ovl_win_unchange
ovl_time   ovl_window
ovl_unchange

Example

```
ovl_win_change #(

    `OVL_ERROR,                                 // severity_level
    32,                                         // width
    `OVL_ASSERT,                                // property_type
    "Error: read not synchronized",             // msg
    `OVL_COVER_DEFAULT,                         // coverage_level
    `OVL_POSEDGE,                               // clock_edge
    `OVL_ACTIVE_LOW,                            // reset_polarity
    `OVL_GATE_CLOCK)                            // gating_type

    valid_sync_data_bus_rd (

        clock,                                  // clock
        reset,                                  // reset
        enable,                                 // enable
        rd,                                     // start_event
        data,                                   // test_expr
        rd_ack,                                 // end_event
        fire_valid_sync_data_bus_rd );          // fire
```
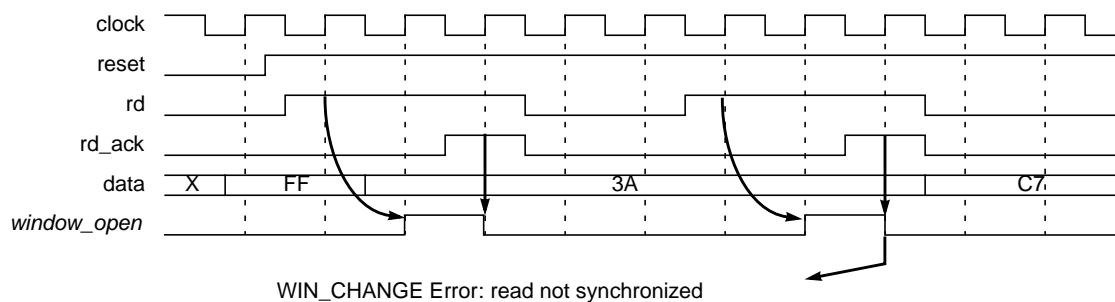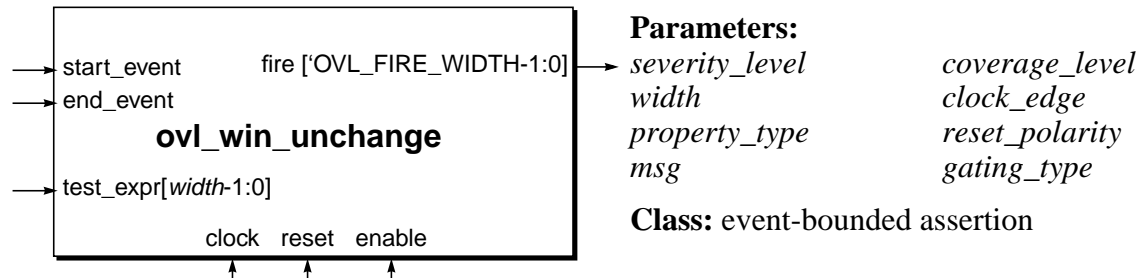
Ensures that *data* changes value in every data read window.

WIN_CHANGE Error: read not synchronized

ovl_win_unchange

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

Parameters:

severity_level	*coverage_level*
width	*clock_edge*
property_type	*reset_polarity*
msg	*gating_type*

Class: event-bounded assertion

Syntax

```
ovl_win_unchange
    [#(severity_level, width, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type)]
  instance_name (clock, reset, enable, start_event, test_expr, end_event,
      fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
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.
fire ['OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_win_unchange assertion checker checks the expression *start_event* at each active edge of *clock* 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 active edge of *clock*, 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 active edge of *clock*.

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

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

ovl_change   ovl_win_change
ovl_time   ovl_window
ovl_unchange

## Example

```
ovl_win_unchange #(

    'OVL_ERROR,                               // severity_level
    8,                                        // width
    'OVL_ASSERT,                              // property_type
    "Error: a changed during divide",         // msg
    'OVL_COVER_DEFAULT,                       // coverage_level
    'OVL_POSEDGE,                             // clock_edge
    'OVL_ACTIVE_LOW,                          // reset_polarity
    'OVL_GATE_CLOCK)                          // gating_type

    valid_div_win_unchange_a (

        clock,                                // clock
        reset,                                // reset
        enable,                               // enable
        start,                                // start_event
        a,                                    // test_expr
        done,                                 // end_event
        fire_valid_div_win_unchange_a );      // fire
```
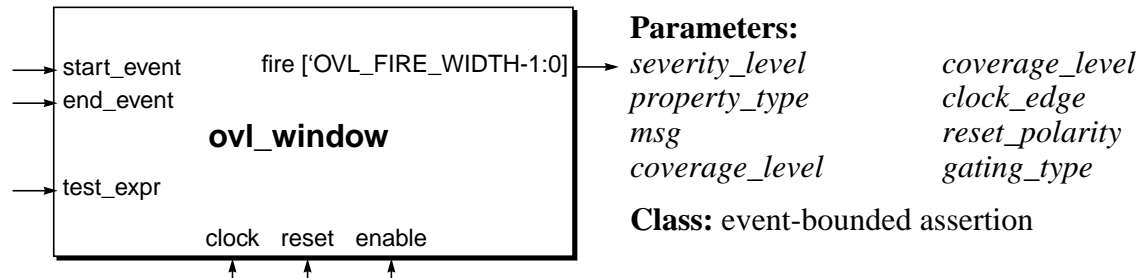
Ensures that the *a* input to the divider remains unchanged while a divide operation is performed (i.e., in the window from *start* to *done*).

WIN_UNCHANGE Error: a changed during divide
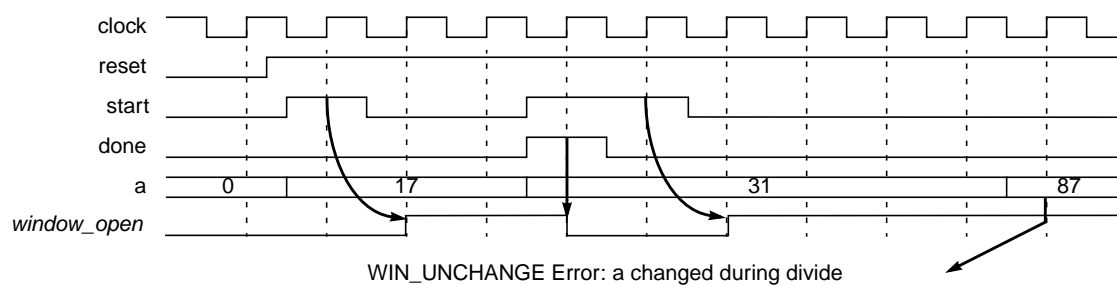
ovl_window

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

```
    ┌──────────────────────────────────┐
──→ │ start_event      fire ['OVL_FIRE_WIDTH-1:0] ──→
──→ │ end_event                        │
    │          ovl_window              │
    │                                  │
──→ │ test_expr                        │
    │                                  │
    │     clock   reset   enable       │
    └──────────────────────────────────┘
          ↑        ↑        ↑
```

Parameters:

severity_level	*coverage_level*
property_type	*clock_edge*
msg	*reset_polarity*
coverage_level	*gating_type*

Class: event-bounded assertion

Syntax

```
ovl_window
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
          reset_polarity, gating_type)]
    instance_name (clock, reset, enable, start_event, test_expr, end_event,
        fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.

`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`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.
`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_window assertion checker checks the expression *start_event* at each active edge of *clock* 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 active edge of *clock*, 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 active edge of *clock*.

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

`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

ovl_change

ovl_time

ovl_unchange

ovl_win_change

ovl_win_unchange

Example

```
ovl_window #(

    'OVL_ERROR,                             // severity_level
    'OVL_ASSERT,                            // property_type
    "Error: write without grant",          // msg
    'OVL_COVER_DEFAULT,                     // coverage_level
    'OVL_POSEDGE,                           // clock_edge
    'OVL_ACTIVE_LOW,                        // reset_polarity
    'OVL_GATE_CLOCK)                        // gating_type

    valid_sync_data_bus_write (

        clock,                              // clock
        reset,                              // reset
        enable,                             // enable
        write,                              // start_event
        bus_gnt,                            // test_expr
        write_ack,                          // end_event
        fire_valid_sync_data_bus_write );   // fire
```
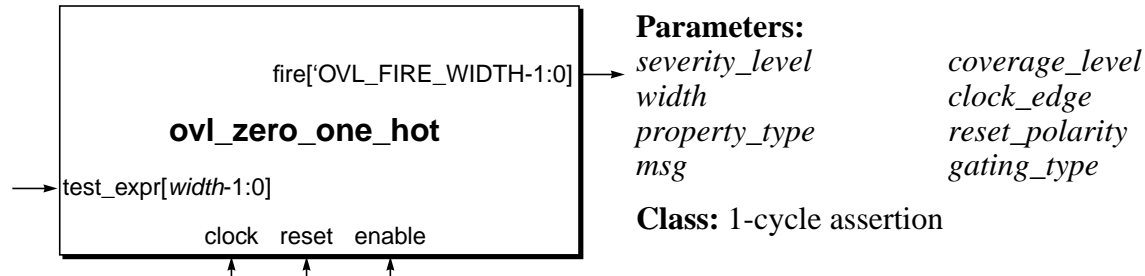
Ensures that the bus grant is not deasserted during a write cycle.

WINDOW Error: write without grant
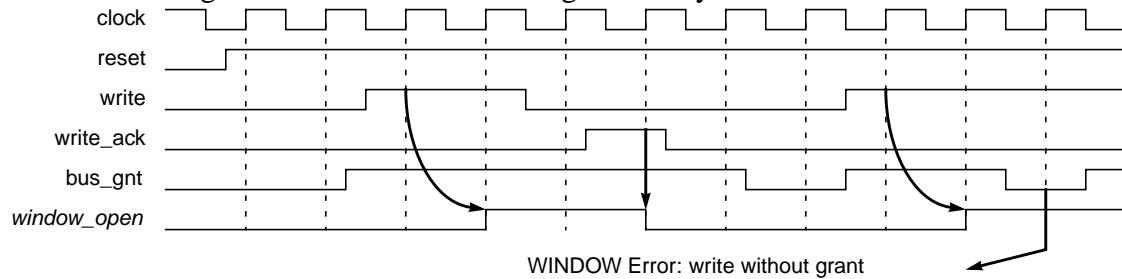
ovl_zero_one_hot

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

```
                    fire['OVL_FIRE_WIDTH-1:0]

              ovl_zero_one_hot


test_expr[width-1:0]

              clock   reset   enable
```

Parameters:

severity_level   *coverage_level*
width   *clock_edge*
property_type   *reset_polarity*
msg   *gating_type*

Class: 1-cycle assertion

Syntax

```
ovl_zero_one_hot
     [#(severity_level, width, property_type, msg, coverage_level,
        clock_edge, reset_polarity, gating_type, clock_edge,
        reset_polarity, gating_type)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters

severity_level	Severity of the failure. Default: 'OVL_SEVERITY_DEFAULT ('OVL_ERROR).
width	Width of the *test_expr* argument. Default: 32.
property_type	Property type. Default: 'OVL_PROPERTY_DEFAULT ('OVL_ASSERT).
msg	Error message printed when assertion fails. Default: 'OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: 'OVL_COVER_DEFAULT ('OVL_BASIC).
clock_edge	Active edge of the *clock* input. Default: 'OVL_CLOCK_EDGE_DEFAULT ('OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: 'OVL_RESET_POLARITY_DEFAULT ('OVL_ACTIVE_LOW).
gating_type	Gating behavior of the checker when *enable* is FALSE. Default: 'OVL_GATING_TYPE_DEFAULT ('OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.

`enable`	Enable signal for *clock,* if *gating_type* = 'OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = 'OVL_GATE_RESET). Ignored if *gating_type* is 'OVL_NONE.
`test_expr[width`-1:0]	Expression that should evaluate to either 0 or a one-hot value on the rising clock edge.
`fire` `['OVL_FIRE_WIDTH-1:0]`	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_zero_one_hot assertion checker checks the expression *test_expr* at each active edge of *clock* 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

`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 ovl_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

ovl_one_cold   ovl_one_hot

Example

```
ovl_zero_one_hot #(

    'OVL_ERROR,                                 // severity_level
    4,                                          // width
    'OVL_ASSERT,                                // property_type
    "Error: sel not zero or one-hot",           // msg
    'OVL_COVER_DEFAULT,                         // coverage_level
    'OVL_POSEDGE,                               // clock_edge
    'OVL_ACTIVE_LOW,                            // reset_polarity
    'OVL_GATE_CLOCK)                            // gating_type

    valid_sel_zero_one_hot (

        clock,                                  // clock
        reset,                                  // reset
        enable,                                 // enable
        sel                                     // test_expr
        fire_valid_sel_zero_one_hot);           // fire
```
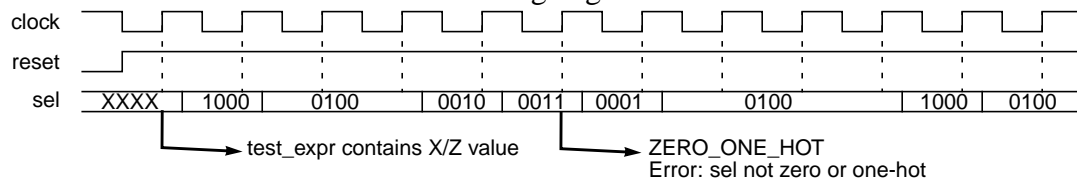
Ensures that *sel* is zero or one-hot at each rising edge of *clock*.

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`	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.
Default Parameter Values	`OVL_SEVERITY_DEFAULT`	Value of *severity_level* to use when the parameter is unspecified. Default: ‘OVL_ERROR.
	`OVL_PROPERTY_DEFAULT`	Value of *property_type* to use when the parameter is unspecified. Default: ‘OVL_ASSERT.
	`OVL_MSG_DEFAULT`	Value of *msg* to use when the parameter is unspecified. Default: “VIOLATION”.
	`OVL_COVER_DEFAULT`	Value of *coverage_level* to use when the parameter is unspecified. Default: ‘OVL_COVER_BASIC.
	`OVL_CLOCK_EDGE_ DEFAULT`	Value of *clock_edge* to use when the parameter is unspecified. Default: ‘OVL_POSEDGE.

Type	DEFINE	Description
	`'OVL_RESET_POLARITY_DEFAULT`	Value of *reset_polarity* to use when the parameter is unspecified. Default: 'OVL_ACTIVE_LOW.
	`'OVL_GATING_TYPE_DEFAULT`	Value of *gating_type* to use when the parameter is unspecified. Default: 'OVL_GATE_CLOCK.
Clock/Reset Gating	`'OVL_GATING_OFF`	Removes all gating logic and creates checkers with *gating_type* 'OVL_GATE_NONE. Default: each checker gated according to its *gating_type* parameter value..
Global Reset	`'OVL_GLOBAL_RESET=`*reset_signal*	Overrides the *reset* port assignments of all assertion checkers with the specified active low global reset signal. Default: each checker's reset is specified by the *reset* 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.
	`'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.

Type	DEFINE	Description
	`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_FIRE_WIDTH
`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`	Runtime error.
	`OVL_WARNING`	Runtime Warning.
	`OVL_INFO`	Assertion failure has no specific severity.
property_type	`OVL_ASSERT`	Assertion checks and X/Z checks are asserts.
	`OVL_ASSUME`	Assertion checks and X/Z checks are assumes.
	`OVL_ASSERT_2STATE`	Assertion checks are asserts. X/Z checks are disabled.
	`OVL_ASSUME_2STATE`	Assertion checks are assumes. X/Z checks are disabled.
	`OVL_IGNORE`	Assertion checks and X/Z checks are disabled.
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.

Parameter	DEFINE	Description
	`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, `OVL_COVER_CORNER and `OVL_COVER_STATISTIC.
	`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, `OVL_COVER_CORNER and `OVL_COVER_STATISTIC.
	`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, `OVL_COVER_BASIC and `OVL_COVER_STATISTIC.
	`OVL_COVER_STATISTIC	Includes coverage logic for the checker's STATISTIC cover points if `OVL_COVER_ON is defined. Can be bitwise-ORed with `OVL_COVER_SANITY, `OVL_COVER_BASIC and `OVL_COVER_CORNER.
clock_edge	`OVL_POSEDGE	Rising edges are active clock edges.
	`OVL_NEGEDGE	Falling edges are active clock edges.
reset_ polarity	`OVL_ACTIVE_LOW	*Reset* is active when FALSE.
	`OVL_ACTIVE_HIGH	*Reset* is active when TRUE.
gating_type	`OVL_GATE_NONE	Checker ignores the *enable* input.
	`OVL_GATE_CLOCK	Checker pauses when *enable* is FALSE. The checker treats the current cycle as a NOP. Checks, counters and internal values remain unchanged.
	`OVL_GATE_RESET	Checker resets (as if the *reset* input became active) when *enable* is FALSE.

Defines for Specific Assertions

Parameter	Checkers	DEFINE	Description
action_on_ new_start	ovl_change ovl_frame ovl_time ovl_unchange	`OVL_IGNORE_NEW_START	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.
		`OVL_ACTION_ON_NEW_ START_DEFAULT	Value of *action_on_new_ start* to use when the parameter is unspecified. Default: 'OVL_ IGNORE_NEW_START.
edge_type	ovl_always_ on_edge	`OVL_NOEDGE	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.
		`OVL_EDGE_TYPE_DEFAULT	Value of *edge_type* to use when the parameter is unspecified. Default: 'OVL_NOEDGE.
necessary_ condition	ovl_cycle_ sequence	`OVL_TRIGGER_ON_MOST_ PIPE	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.

Parameter	Checkers	DEFINE	Description
		`'OVL_NECESSARY_ CONDITION_DEFAULT`	Value of *necessary_condition* to use when the parameter is unspecified. Default: 'OVL_TRIGGER_ON_ MOST_PIPE.
inactive	ovl_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.
		`'OVL_INACTIVE_DEFAULT`	Value of *inactive* to use when the parameter is unspecified. Default: 'OVL_ONE_COLD.