Accellera Standard OVL V2

Library Reference Manual

Software Version 2.7

January 2013

License and Statement of Use of Accellera System Initiative's Open Verification Library

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

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

Accellera Systems Initiative Inc.
1370 Trancas Street #163, Napa, CA 94558, USA
E-mail: info@accellera.org

Note

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

Overview of this standard

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

Intent and scope of this document

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

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

ACKNOWLEDGEMENTS

These Accellera Systems Initiative OVL Libraries and Library Reference Manual (LRM) were specified and developed by experts from many different fields, including design and verification engineers, Electronic Design Automation companies and members of the OVL VSVA technical committee. The following contributors were involved in the creation of previous versions of the OVL: Bryan Bullis, Ben Cohen, Himanshu Goel, Vijay Gupta, Brent Hayhoe, Richard Ho, Dmitry Korchemny, Narayanan Krishnamurthy, David Lacey, Jim Lewis, Andrew MacCormack, Erich Marschner, Paul Menchini, Torkil Oelgaard, Uma Polisetti, Joseph Richards, Erik Seligman, Vinaya Singh, Sean Smith, Andy Tsay, Mike Turpin, Bipul Talukdar, and others.

The OVL technical committee and chair reports to Accellera TC Chairman:

TC Chairman Karen Pieper

The following individuals contributed to the creation, editing and review of the Accellera Systems Initiative OVL Libraries and LRM: Alan Becker/ARM, Shalom Bresticker/Intel, Eduard Cerny/Synopsys, Harry Foster/Mentor Graphics, Vijay Shanker Gottimukkula/Synchronicity, Jerry Kaczinsky/Aldec, David Lacey/Hewlett Packard, Kenneth Elmkjær Larsen/Mentor Graphics (OVL Chair), Ramesh Sathianathan/Mentor Graphics, Chris Shaw/Mentor Graphics, Manoj Kumar Thottasseri/Synopsys .

Major version 2.0 released June 2007
Minor version 2.1 released September 2007
Minor version 2.2 released January 2008
Minor version 2.3 released June 2008
Minor version 2.4 released March 2009
Minor version 2.5 released July 2010
Minor version 2.6 released November 2011
Minor version 2.7 released January 2013

SYSTEMS INITIATIVE

Table of Contents

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 unifying methodology for dynamic and formal verification.

OVL V2 is a superset of OVL V1 that includes all V1 checkers. 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, code 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

OVL V2 checker types are named ovl_*checker*. OVL V2 checkers are instantiated in Verilog and VHDL modules/entities with specified parameters/generics and connections to checker ports. Each checker type's data sheet shows a model of its checker's instance statement in a language-neutral mnemonic syntax statement. A checker type has parameters/generics common to all checkers and parameters/generics specific to its own type. The parameter/generic identifiers in a checker type's syntax statement are shown in this order:

```
severity_level, [checker specific parameter/generic identifiers],
property_type, msg, coverage_level, clock_edge, reset_polarity,
gating_type
```

A checker type has port identifiers common to all checkers and ports specific to its own type. The port identifiers in a checker type's syntax statement are declared in this order:

```
clock*, reset, enable, [checker specific ports], fire
```

except (*) that asynchronous checker types have no *clock* port and multiclock checker types have multiple clock ports.

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.
- Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A., *Verification Methodology Manual for SystemVerilog*, Springer, 2005, ISBN 978-0-387-25538-5.
- Foster, H., Krolnik, A., Lacey, D. *Assertion-Based Design*, Kluwer Academic Publishers, 2003.

Chapter 2
OVL Basics

The OVL is composed of a set of assertion checkers that verify specific properties of a design. These assertion checkers are instantiated in the design establishing a unifying methodology for dynamic and formal verification.

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 indicates whether the error captured by the assertion library is a major or minor problem.
- Coverage indicates whether or not specific corner-case events occur and counts the occurrences of specific 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 reuse the same assertions for different methodologies, typically simulation and formal verification.

OVL Assertion Checkers

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 corner-case behavior or interface protocols).
- 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.
- Some IP providers embed assertions with their designs, so they can be turned on for integration checking.

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 verification checks. It is also possible to combine an OVL assertion with additional HDL logic to check for the desired behavior. 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 achieve the desired level of checking in 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.

HDL Implementations

Designers instantiate OVL assertion checkers as logic components in design code. Two variations are available, corresponding to the two “base” HDL language families: Verilog and VHDL. Checker assertion and coverage logic can be instantiated in several different standard implementations. The current implementations are in five IEEE languages:

- Verilog Family
  - Verilog 1995 (IEEE 1364),
  - SVA 2005 (IEEE 1800),
  - PSL 2005 (IEEE 1850).

- VHDL
  - VHDL 1993 (IEEE 1076),
  - PSL 2005 (IEEE 1850).

Not all checker types have been implemented in all HDLs. Table 2-1 shows the currently implemented checker types with √ marks. The table shows the checker types that have full *fire* output ports implemented with ⇒ marks. *Fire* outputs of the other types of checkers are currently tied low. Green ( ) indicates the checker type is implemented in all languages; red ( ) indicates the checker type is implemented only in SVA; and wheat ( ) indicates the checker type is implemented in some other combination.

Checker implementations that are synthesizable are indicated with *synth*. You must specify OVL_SYNTHESIS (see "Generating Synthesizable Logic" on page 27) to disable unsynthesizable logic for these checkers. "Synthesizing the VHDL OVL Library" on page 51 shows how to instantiate synthesizable VHDL checker logic.

Table 2-1. OVL Library

	Verilog			VHDL	
checker type	**Verilog-95**	**SVA-05**	**PSL-05**	**VHDL-93**	**PSL-05**
ovl_always	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_always_on_edge	√	√	√		√
ovl_arbiter		√			
ovl_bits		√			
ovl_crc		√			
ovl_change	√	√	√		√
ovl_code_distance		√			
ovl_coverage		√			
ovl_cycle_sequence	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_decrement	√	√	√		√
ovl_delta	√	√	√		√
ovl_even_parity	√	√	√		√
ovl_fifo		√			
ovl_fifo_index	√	√	√		√
ovl_frame	√	√	√		√
ovl_handshake	√	√	√		√
ovl_hold_value		√			

Table 2-1. OVL Library

	Verilog			VHDL	
checker type	**Verilog-95**	**SVA-05**	**PSL-05**	**VHDL-93**	**PSL-05**
ovl_implication	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_increment	√	√	√		√
ovl_memory_async		√			
ovl_memory_sync		√			
ovl_multiport_fifo		√			
ovl_mutex		√			
ovl_never	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_never_unknown	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_never_unknown_async	√	√	√	√ ⇒ *synth*	√
ovl_next	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_next_state		√			
ovl_no_contention		√			
ovl_no_overflow	√	√	√		√
ovl_no_transition	√	√	√		√
ovl_no_underflow	√	√	√		√
ovl_odd_parity	√	√	√		√
ovl_one_cold	√	√	√		√
ovl_one_hot	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_proposition	√	√	√		√
ovl_quiescent_state	√	√	√		√
ovl_range	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√
ovl_reg_loaded		√			
ovl_req_ack_unique		√			
ovl_req_requires		√			
ovl_stack		√			
ovl_time	√	√	√		√
ovl_transition	√	√	√		√
ovl_unchange	√	√	√		√
ovl_valid_id		√			

Table 2-1. OVL Library

	Verilog			VHDL	
checker type	**Verilog-95**	**SVA-05**	**PSL-05**	**VHDL-93**	**PSL-05**
ovl_value		√			
ovl_value_coverage		√			
ovl_width	√	√	√		√
ovl_win_change	√	√	√		√
ovl_win_unchange	√ ⇒ *synth*	√ ⇒	√		√
ovl_window	√	√	√		√
ovl_xproduct_bit_coverage		√			
ovl_xproduct_value_coverage		√			
ovl_zero_one_hot	√ ⇒ *synth*	√ ⇒	√	√ ⇒ *synth*	√

OVL V1-Style Checkers

For backward-compatibility with designs that use OVL V1 checkers, the OVL V2 library includes copies of the checkers from the V1 library (updated with code fixes, but having the same "footprints" as the V1 library checkers). These checker types are recognized by their "assert_" prefixes. Table 2-2 shows the V1-style OVL library's checker types' implementations. None of these checker types have *fire* outputs because the *fire* ports were new on the ovl_* checkers. The V1-style checkers have no outputs, so their logic is optimized out by synthesis tools (i.e., no V1-style checkers are synthesizable).

Table 2-2. OVL V1-Style Checkers

	Verilog		
checker type	**Verilog-95**	**SVA-05**	**PSL-05**
assert_always	√	√	√
assert_always_on_edge	√	√	√
assert_change	√	√	√
assert_cycle_sequence	√	√	√
assert_decrement	√	√	√
assert_delta	√	√	√
assert_even_parity	√	√	√
assert_fifo_index	√	√	√
assert_frame	√	√	√

Table 2-2. OVL V1-Style Checkers

	Verilog		
checker type	**Verilog-95**	**SVA-05**	**PSL-05**
assert_handshake	√	√	√
assert_implication	√	√	√
assert_increment	√	√	√
assert_never	√	√	√
assert_never_unknown	√	√	√
assert_never_unknown_async	√	√	√
assert_next	√	√	√
assert_no_overflow	√	√	√
assert_no_transition	√	√	√
assert_no_underflow	√	√	√
assert_odd_parity	√	√	√
assert_one_cold	√	√	√
assert_one_hot	√	√	√
assert_proposition	√	√	√
assert_quiescent_state	√	√	√
assert_range	√	√	√
assert_time	√	√	√
assert_transition	√	√	√
assert_unchange	√	√	√
assert_width	√	√	√
assert_win_change	√	√	√
assert_win_unchange	√	√	√
assert_window	√	√	√
assert_zero_one_hot	√	√	√

OVL 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/Generics

Each OVL assertion checker has its own set of parameters as described in its corresponding data sheet (page First). 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 Verilog macros (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 28). VHDL OVL_CTRL_DEFAULTS are set in the ovl_ctrl_record record (see "ovl_ctrl_record Record" on page 45).

The checker parameters/generics can be assigned instance-specific values using the appropriate Verilog macros or VHDL constants defined in the *std_ovl_defines.h* and *std_ovl.vhd* files respectively. The macro and constant identifier names are the same in both HDLs.

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 (simulation stops).
`OVL_ERROR`	Runtime error.
`OVL_WARNING`	Runtime warning (e.g., software warning).
`OVL_INFO`	Information only (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 stimulus). 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. Used to turn off checking while maintaining coverage collection. To switch off sets of assertions, define macros for the property types, for example: *'define MY_OVL_CHECKS_OFF 'OVL_IGNORE*.

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 bitwise-OR of the defined cover point type values ("Cover Points" on page 24 and "Monitoring Coverage" on page 28):

`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_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 18): 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 18) 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 18):

- OVL_GATE_NONE (no effect),
- OVL_GATE_CLOCK (gate *clock*, see "Gating clock" on page 19) or
- OVL_GATE_RESET (gate *reset*, see "Gating reset" on page 20).

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 V2.7 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.

For most checkers, each *fire* output bit is implemented in a clocked process. A *fire* bit is TRUE for the cycle following the cycle in which a violation occurs and stays TRUE until the property passes. In particular, a *fire* bit can be TRUE for consecutive cycles because:

- A checker's checks are pipelined, so multiple violations can occur in adjacent clock cycles.
- A multi-cycle checker (for example, ovl_next) can have a single violation that takes multiple cycles to return to a passing state. Note that the number of cycles in which a *fire* bit is TRUE is not the same as the number of violations for the checker.

For the asynchronous checkers (ovl_memory_async and ovl_never_unknown_async). *fire* outputs are driven directly by combinatorial logic and so are only TRUE during the failing condition. 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).

The following macros are defined for accessing individual *fire* bits:

```
`define OVL_FIRE_2STATE 0
`define OVL_FIRE_XCHECK 1
`define OVL_FIRE_COVER 2
```

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 “Checking X and Z Values” on page 29). (Note that OVL does not differentiate between X and Z values.)

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 specific checker ports have known and driven values when the checker samples the ports. For example, the ovl_frame checker type has 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. For many checkers, implicit X/Z-check violations are not triggered for every occurrence of a sampled X/Z value for the associated checker port. For example, consider the ovl_implication checker, which has X/Z checks for *antecedent_expr* and *consequent_expr*:

	antecedent_expr	***consequent_expr***	**Assertion fails?**
a	True	X/Z	if *consequent_expr* is False
b	False	X/Z	no
c	X/Z	True	no
d	X/Z	False	if *antecedent_expr* is True
e	X/Z	X/Z	if *antecedent_expr* is True and *consequent_expr* is False

Cases *b* and *c* are not reported as X/Z-check violations, because in both cases the assertion is not violated—regardless of which 0/1 value the X/Z-valued expression takes in 2-state semantics. Such intelligent handling of X/Z checks eliminates many “false” violations that would be reported when a pessimistic view of X/Z values is assumed.

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_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`	(Default) Event that indicates that the logic monitored by the assertion checker assumed a state where assertion checking can 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.

Cover Groups

Some assertion types have one or more defined cover groups. Each cover group consists of one or more bin registers that accumulate coverage counts for corresponding coverage points. Some bin registers are two-dimensional, where the bin indexes represent the various cover cases being tracked and the bin values represent the associated coverage counts. For example, the ovl_valid_id assertion type has the two following cover groups:

`observed_latency`	Number of returned IDs with the specified turnaround time. Bins are: • *observed_latency_good*[*min_cks*:*max_cks*] — bin index is the observed turnaround time in clock cycles. • *observed_latency_bad* — default.
outstanding_ids	Number of cycles with the specified number of outstanding ids. Bins are: • *observed_outstanding_ids*[0:*max_instances*] — bin index is the instance ID.

Verilog OVL

The Verilog HDL Family OVL library has the following characteristics:

- All Verilog assertion checkers conform to Verilog IEEE Standard 1364-1995. Top-level files are either called `assert_`*`checker`*`.vlib` or `ovl_`*`checker`*`.v` (new in V2), and include the relevant logic (Verilog, SVA or PSL).
- All System Verilog assertion checkers conform to Accellera SVA 2005 (IEEE 1800).
- 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`.
- Parameter settings are assigned with macros 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`.
- OVL V2 is backward compatible in behavior with existing OVL V1 libraries, because OVL V2 includes the assert_*checker* modules.

Library Directory Structure

The Accellera OVL standard Verilog 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 2005 (IEEE 1364).
`$STD_OVL_DIR/sva05`	Directory with assertion logic described in SVA 2005 (IEEE 1800).
`$STD_OVL_DIR/psl05`	Directory with assertion logic described in PSL 2005 (IEEE 1850).
`$STD_OVL_DIR/psl05/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/psl05:
std_ovl/psl05/assert_always_logic.vlib
std_ovl/psl05/assert_always_on_edge_logic.vlib
.   .   .
std_ovl/psl05/vunits:
std_ovl/psl05/vunits/assert_always.psl
std_ovl/psl05/vunits/assert_always_on_edge.psl
.   .   .
std_ovl/sva05:
std_ovl/sva05/assert_always_logic.vlib
std_ovl/sva05/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
.   .   .
```

Use Model

An Accellera Standard OVL Verilog 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 Verilog Implementation Language

The Accellera Standard OVL is implemented in the following Verilog HDL languages: Verilog 1995(IEEE 1364), SVA 2005 (IEEE 1800) and PSL 2005 (IEEE 1850). The following Verilog macros select the implementation language:

`OVL_VERILOG`	(default) Creates assertion checkers defined in Verilog-95.
`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 Verilog-flavor 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 removes initialization logic from OVL assertions:

`OVL_SYNTHESIS`	Removes initialization logic from the OVL assertion logic. Default: logic inside the else branch of *ifdef OVL_SYNTHESIS* blocks is enabled.

Setting OVL_SYNTHESIS removes the unsynthesizable logic from Verilog-95 checkers, making them synthesizable.

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.

By default, coverage logic (activated with OVL_COVER_ON) monitors cover points and cover groups. To exclude logic that monitors cover groups define the following standard global variable:

`OVL_COVERGROUP_OFF`	Excludes cover group logic from the coverage logic if OVL_COVER_ON is defined. Default: not defined.

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.

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 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: V2.7: 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.

Header Files

std_ovl_defines.h

```
// Accellera Standard V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. All rights reserved.

`ifdef OVL_STD_DEFINES_H
// do nothing
`else
`define OVL_STD_DEFINES_H
`define OVL_VERSION "V2.7"

`ifdef OVL_ASSERT_ON
  `ifdef OVL_PSL
     `ifdef OVL_VERILOG
        `undef OVL_PSL
     `endif
     `ifdef OVL_SVA
        `ifdef OVL_PSL
          `undef OVL_PSL
        `endif
     `endif
  `else
    `ifdef OVL_VERILOG
    `else
      `define OVL_VERILOG
    `endif
    `ifdef OVL_SVA
       `undef OVL_VERILOG
    `endif
  `endif
`endif
```

```
`ifdef OVL_COVER_ON
  `ifdef OVL_PSL
     `ifdef OVL_VERILOG
        `undef OVL_PSL
     `endif
     `ifdef OVL_SVA
        `ifdef OVL_PSL
          `undef OVL_PSL
        `endif
     `endif
  `else
    `ifdef OVL_VERILOG
    `else
      `define OVL_VERILOG
    `endif
    `ifdef OVL_SVA
       `undef OVL_VERILOG
    `endif
  `endif
`endif

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

// specifying interface for System Verilog
`ifdef OVL_SVA_INTERFACE
  `define module interface
  `define endmodule endinterface
`else
  `define module module
  `define endmodule endmodule
`endif

// Selecting global reset or local reset for the checker reset signal
`ifdef OVL_GLOBAL_RESET
  `define OVL_RESET_SIGNAL `OVL_GLOBAL_RESET
`else
  `define OVL_RESET_SIGNAL reset_n
`endif

// active edges
`define OVL_NOEDGE  0
`define OVL_POSEDGE 1
`define OVL_NEGEDGE 2
`define OVL_ANYEDGE 3
```

```
// 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 2.4 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 V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. 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 V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. All rights reserved.
wire clk;
`ifdef OVL_SHARED_CODE
  wire gclk;
  `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;
`else
  assign clk = clock;
`endif // OVL_SHARED_CODE
```

std_ovl_reset.h

```
// Accellera Standard V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. All rights reserved.
wire reset_n;
`ifdef OVL_SHARED_CODE
  wire greset;
  `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;
`else
  assign reset_n = reset;
`endif // OVL_SHARED_CODE
```

std_ovl_count.h

```
// Accellera Standard V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. 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 V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. 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 V2.7 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2012. 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

  `ifdef OVL_MAX_REPORT_ERROR
    if (error_count < `OVL_MAX_REPORT_ERROR)
  `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_FINISH_OFF
  `else
     if (severity_level == `OVL_FATAL) begin
      case (property_type)
        `OVL_ASSERT,
        `OVL_ASSUME         : begin ovl_finish_t; end
        `OVL_ASSERT_2STATE,
        `OVL_ASSUME_2STATE  : begin
           if (xcheck == `OVL_FIRE_2STATE) begin; ovl_finish_t; end end
        `OVL_IGNORE         : begin end
        default             : begin end
      endcase
    end
`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_SVA
`else
  // FUNCTION THAT CALCULATES THE LOG BASE 2 OF A NUMBER
  // ========
  // NOTE: only used in sva05
  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_SVA
```

```
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
```

VHDL OVL

The OVL library includes VHDL implementations of OVL checkers. The pure-VHDL implementation of OVL contains only 10 checkers and the VHDL-flavor PSL implementation contains 33 checkers. The pure-VHDL OVL checkers are the ovl_*checker_type* versions of the components (which include the *enable* and *fire* ports). VHDL wrappers are provided for the missing checkers that allow the Verilog checkers to be instantiated from VHDL.

The VHDL OVL components are compatible with the Verilog OVL versions, except the VHDL components include an additional generic called *controls* that provides global configuration of the library. The VHDL implementation has the following additional characteristics:

- VHDL OVL is synthesizable (see "Synthesizing the VHDL OVL Library" on page 53).
- VHDL OVL components support both *std_logic* and *std_ulogic* port types.
- VHDL OVL implementation contains constants that are equivalent to (have the same name and values) the corresponding Verilog macro defines. However some macros are not present in the VHDL implementation because they are implemented by an *ovl_ctrl_record* constant (see "ovl_ctrl_record Record" on page 45) or are not needed.

Library Directory Structure

In the OVL installation, the following files are used for the VHDL implementation.

`std_ovl/`

ovl_*`checker_type`*.vhd	Checker entity declarations.
std_ovl.vhd	Type/constant declarations package.
std_ovl_procs.vhd	Procedures package.
std_ovl_components.vhd	*std_ovl_components* package containing checker component declarations for the checkers in pure-VHDL OVL.
std_ovl_vhdl_components.vhd	Checker component declarations for all PSL VHDL-flavor checkers.
std_ovl_u_components.vhd	*std_ovl_u_components* package and *std_ulogic* wrapper components.
std_ovl_components_vlog.vhd	Alternative *std_ovl_components* package containing wrappers to allow Verilog checkers to be used for checkers that are missing from the pure-VHDL implementation.

std_ovl_u_components_vlog.vhd	Alternative *std_ovl_u_components* package containing *std_ulogic* wrappers to allow Verilog checkers to be used for checkers that are missing from the pure-VHDL implementation
std_ovl_clock_gating.vhd	Internal clock gating component.
std_ovl_reset_gating.vhd	Internal reset gating component.
std_ovl/vhdl93/	
ovl_*checker_type*_rtl.vhd	Checker architecture bodies.
std_ovl/vhdl93/syn_src	
std_ovl_procs_syn.vhd	Synthesizable version of *std_ovl_procs.vhd*.
ovl_*checker_type*_rtl.vhd	Synthesizable versions of architecture bodies.
std_ovl/vhdl93/legacy/	
std_ovl.vhd	Component declarations to allow V1 assert_*checker* Verilog checkers to be used in VHDL.
std_ovl/psl05/	
assert_*_psl_logic.vhd	Entity declarations for PSL assertions and architecture definitions for ovl_*checker*.vhd
std_ovl/psl05/vunits_vhdl/	
assert_*.psl	Declarations and definitions of all properties in PSL files

Use Model

Compiling the VHDL OVL

All the VHDL files (except *std_ovl_u_components.vhd and std_ovl_vhdl_components.vhd)* should be compiled into the logical library name *accellera_ovl_vhdl* (standardized for portability) for implementation of the 10 pure-VHDL checkers. The *accellera_ovl_vhdl* library can be compiled into a central location that can be shared by designers. The library is configured using a project-specific *ovl_ctrl_record* record as shown in "Configuring the Library" on page 45, so modifying the default configuration values in the *std_ovl* package is not necessary. The library must be compiled using the EDA tools' VHDL-93 option.

The pure-VHDL OVL implementation does not contain all of the OVL checkers. Therefore, wrapper components are provided that allow Verilog implementations of the missing checkers to be used in VHDL. These wrapper components are found in the *std_ovl_componets_vlog.vhd* file (which also contains a *std_ovl_components* package). This package name is the same as the package in the *std_ovl_components.vhd* file, but it includes component declarations for the missing checkers. The same package name is used in both files, so only one *std_ovl_components* file should be compiled into the library.

For the VHDL-flavor PSL implementation of OVL, the VHDL components can be directly used wherever required. *std_ovl_vhdl_components.vhd* is an updated file that contains a package *std_ovl_vhdl_components* that needs to be compiled. This package has component declarations of all the checkers for VHDL-flavor PSL and has to be included for the checker implementation.

The following section shows how to compile the pure-VHDL OVL checkers and the VHDL OVL with PSL checkers.

Note

std_ovl_vhdl_components is a new package in addition to *std_ovl_components*. This needs to be compiled for VHDL-flavor PSL implementation of checkers. If it is not required to use the PSL checkers, *std_ovl_components* package is sufficient.

OVL Compile Order for pure-VHDL checkers

The *accellera_ovl_vhdl* library's compile order is as follows:

1. `std_ovl/std_ovl.vhd`
2. `std_ovl/std_ovl_components.vhd`
3. `std_ovl/std_ovl_procs.vhd`
4. `std_ovl/std_ovl_clock_gating.vhd`
5. `std_ovl/std_ovl_reset_gating.vhd`
6. `std_ovl/ovl_name.vhd`
7. `std_ovl/vhdl93/ovl_*_rtl.vhd`

ovl_*name*.vhd refers to the 10 pure-VHDL OVL checkers (see the list in "OVL Library" on page 12).

OVL Compile order for VHDL-flavor PSL

The *accellera_ovl_vhdl* library's compile order is as follows:

1. `std_ovl/std_ovl.vhd`
2. `std_ovl/std_ovl_procs.vhd`
3. `std_ovl/std_ovl_clock_gating.vhd`
4. `std_ovl/std_ovl_reset_gating.vhd`

5. `std_ovl/ovl_*.vhd`
6. `std_ovl/std_ovl_vhdl_components.vhd`
7. `std_ovl/psl05/ovl_*_psl_logic.vhd`
8. std_ovl/psl05/vunits_vhdl/ovl_*.psl

Compilation of the PSL files might require a tool-specific switch/command. For pure-VHDL checkers, if *std_ulogic*-based ports are required, then you must compile the *std_ovl_u_components.vhd* file into a separate *accellera_ovl_vhdl_u* library after the *accellera_ovl_vhdl* library files are compiled.

Configuring the Library

VHDL OVL has all the global library configuration features of the Verilog implementation (which are provided by the Verilog macro defines). For example: globally enabling/disabling X/Z-checking on all checker instances.

An *ovl_ctrl_record* constant controls global library configuration. This record is declared in *std_ovl.vhd* and is assigned to the *controls* generic on every checker instance. It should be defined in a design-specific work library package for use on all checker instances. With this implementation, the configuration of the checkers is controlled from one place.

In particular, changing constants in the central *std_ovl.vhd* file is not necessary. In fact, the VHDL OVL files are read-only and modifying any of them is not recommended. Apart from the *ovl_control_record*, each OVL assertion checker has its own set of parameters as described in its corresponding data sheet (see page 71).

ovl_ctrl_record Record

The *ovl_ctrl_record* record is divided into three groups:

- Elements that are of the *ovl_ctrl* type and can be assigned OVL_ON or OVL_OFF values. These elements mainly control the generate statements used in the checkers.
- User-configurable values that control the message printing and how long the simulation should continue after a fatal assertion occurs.
- Default values of the generics that are common to all checkers.

Table 2-3 shows the *ovl_ctrl_record* record elements and how they map to the Verilog macro values that configure the Verilog implementation of the OVL.

Table 2-3. ovl_ctrl_record Elements

ovl_ctrl_record	Description	Verilog Macro	VHDL Value
`xcheck_ctrl`	Enables/disables all X/Z checking code.	`OVL_XCHECK_OFF`	`OVL_OFF`

Table 2-3. ovl_ctrl_record Elements (cont.)

ovl_ctrl_record	Description	Verilog Macro	VHDL Value
`implicit_xcheck_ctrl`	Enables/disables implicit X/Z checks.	`OVL_IMPLICIT_XCHECK_OFF`	`OVL_OFF`
`init_msg_ctrl`	Enables/disables code that prints checker initialization messages or a count of the number of checkers initialized.	`OVL_INIT_MSG`	`OVL_OFF`
`init_count_ctrl`	Enables/disables counting of number of checkers initialized when init_msg_ctrl is set to OVL_ON.	`OVL_INIT_COUNT`	`OVL_OFF`
`assert_ctrl`	Enables/disables all 2-state and X/Z check assertions.	`OVL_ASSERT_ON`	`OVL_ON`
`cover_ctrl`	Enables/disables converge code.	`OVL_COVER_ON`	`OVL_ON`
`global_reset_ctrl`	Enables/disables the use of a global reset signal.	`OVL_GLOBAL_RESET`	`OVL_ON`
`finish_ctrl`	Enables/disables halting of simulation when a fatal assertion is detected.	`OVL_FINISH_OFF`	`OVL_OFF`
`gating_ctrl`	Enables/disables clock or reset gating.	`OVL_GATING_OFF`	`OVL_OFF`
`max_report_error`	Maximum number of assertion error messages that a checker should report.	`OVL_MAX_REPORT_ERROR`	`15`
`max_report_cover_point`	Maximum number of coverage messages that a checker should report.	`OVL_REPORT_COVER_POINT`	`15`
`runtime_after_fatal`	Time after a fatal assertion is detected that the simulation should be halted.	`OVL_RUNIME_AFTER_FATAL`	`100 ns`
`severity_level_default`	*severity_level* generic default value.	`OVL_SEVERITY_DEFAULT`	`OVL_ERROR`
`property_type_default`	*property_type* generic default value.	`OVL_PROPERTY_DEFAULT`	`OVL_ASSERT`
`msg_default`	*msg* generic default value.	`OVL_MSG_DEFAULT`	`“VIOLATION”`

Table 2-3. ovl_ctrl_record Elements (cont.)

ovl_ctrl_record	Description	Verilog Macro	VHDL Value
coverage_level_ default	*coverage_level* generic default value.	OVL_COVER_ DEFAULT	OVL_COVER_ BASIC
clock_edge_default	*clock_edge* generic default value.	OVL_CLOCK_ EDGE_DEFAULT	OVL_POSEDGE
reset_polarity_ default	*reset_polarity* generic default value.	OVL_RESET_ POLARITY_DEFAULT	OVL_ACTIVE_ LOW
gating_type_default	*gating_type* generic default value.	OVL_GATING_ TYPE_DEFAULT	OVL_GATE_ CLOCK

The following example shows how to declare and use an *ovl_ctrl_record* record constant:

```
library accellera_ovl_vhdl;
use accellera_ovl_vhdl.std_ovl.all;

package proj_pkg is
  -- OVL configuration
  constant ovl_proj_controls : ovl_ctrl_record := (
    -- generate statement controls
    xcheck_ctrl                 => OVL_ON,
    implicit_xcheck_ctrl        => OVL_ON,
    init_msg_ctrl               => OVL_ON,
    init_count_ctrl             => OVL_OFF,
    assert_ctrl                 => OVL_ON,
    cover_ctrl                  => OVL_ON,
    global_reset_ctrl           => OVL_OFF,
    finish_ctrl                 => OVL_ON,
    gating_ctrl                 => OVL_ON,

    -- user configurable library constants
    max_report_error            => 4,
    max_report_cover_point      => 15,
    runtime_after_fatal         => "150 ns    ",

    -- default values for common generics
    severity_level_default      => OVL_SEVERITY_DEFAULT,
    property_type_default       => OVL_PROPERTY_DEFAULT,
    --msg_default               => OVL_MSG_DEFAULT,
    msg_default                 => ovl_set_msg("YOUR DEFAULT MESSAGE"),
    coverage_level_default      => OVL_COVER_DEFAULT,
    clock_edge_default          => OVL_CLOCK_EDGE_DEFAULT,
    reset_polarity_default      => OVL_RESET_POLARITY_DEFAULT,
    gating_type_default         => OVL_GATING_TYPE_DEFAULT
  );
end package proj_pkg;

library accellera_ovl_vhdl;
use accellera_ovl_vhdl.std_ovl.all;
use accellera_ovl_vhdl.std_ovl_components.all; -- optional - not needed if
                                                -- using direct instantiation
use work.proj_pkg.all;
```

```
architecture rtl of design is
begin

   ---rtl code---
  ovl_gen : if (ovl_proj_controls.assert_ctrl = OVL_ON) generate

      ----user ovl signal conditioning code---

    ovl_u1 : ovl_next
      generic map (
         msg                  => "Check 1",
         num_cks              => 1,
         check_overlapping    => OVL_CHK_OVERLAP_OFF,
         check_missing_start => OVL_OFF,
         coverage_level       => OVL_COVER_CORNER,
         controls             => ovl_proj_controls
      )
      port map (
         clock                => clk,
         reset                => reset_n,
         enable               => enable_1,
         start_event ,        => start_event_1
         test_expr            => test_1,
         fire                 => fire_1
      );

    ovl_u2 : ovl_next
      generic map (
         msg                  => "Check 2",
         num_cks              => 2,
         check_overlapping    => OVL_CHK_OVERLAP_ON,
         check_missing_start => OVL_ON,
         coverage_level       => OVL_COVER_ALL,
         severity_level       => OVL_FATAL,
         controls             => ovl_proj_controls
      )
      port map (
         clock                => clk,
         reset                => reset_n,
         enable               => enable_2,
         start_event          => start_event_2,
         test_expr            => test_2,
         fire                 => fire_2
      );
  end generate ovl_gen;
end architecture rtl;
```

The *ovl_ctrl_record* is typically configured for various projects. For example, to enable assertion checks but no coverage, set *assert_ctrl* to OVL_ON and *cover_ctrl* to OVL_OFF where OVL_ON and OVL_OFF are of subtype *ovl_ctrl* declared in *std_ovl.vhd*.

Checker example with PSL-VHDL flavor

The following example shows the implementation of a PSL-VHDL checker *ovl_even_parity*.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_ovl.all;
use work.std_ovl_vhdl_components.all;

entity test is
   port(test_expr : in std_logic_vector(3 downto 0));
end test;

architecture test_architecture of test is

   signal clk: std_logic := '0';
   signal reset_n : std_logic := '0';
   signal temp : std_logic_vector (3 downto 0);
   signal en: std_logic := '1';

constant controls_param  : ovl_ctrl_record     :=
  (-- generate statement controls
    xcheck_ctrl                  => OVL_ON,
    implicit_xcheck_ctrl         => OVL_ON,
    init_msg_ctrl                => OVL_OFF,
    init_count_ctrl              => OVL_OFF,
    assert_ctr                   => OVL_ON,
    cover_ctr                    => OVL_ON,
    global_reset_ctrl            => OVL_OFF,
    finish_ctrl                  => OVL_ON,
    gating_ctrl                  => OVL_ON,

    -- user configurable library constants
    max_report_error               => 15,
    max_report_cover_point         => 15,
    runtime_after_fatal            => "200 ns    ",

    -- default values for common generics
    severity_level_default         => OVL_SEVERITY_DEFAULT,
    property_type_default          => OVL_PROPERTY_DEFAULT,
    msg_default                    => OVL_MSG_DEFAULT,
    coverage_level_default         => OVL_COVER_DEFAULT,
    clock_edge_default             => OVL_CLOCK_EDGE_DEFAULT,
    reset_polarity_default         => OVL_RESET_POLARITY_DEFAULT,
    gating_type_default            => OVL_GATING_TYPE_DEFAULT
  );

begin

   process
   begin
      wait for 5 ns;
      clk <= not clk;
   end process;
```

```
    process
    begin
       wait for 25 ns;
       reset_n <= '1';
    end process;

    temp<= test_expr xor "0101"; --sample operation

    one_ep1: ovl_even_parity
    generic map(
       property_type => OVL_ASSERT,
       width => 4,
       controls => controls_param)
    port map(
       clock => clk,
       reset =>reset_n,
       enable =>en,
       test_expr =>temp);

 end architecture test_architecture;
```

This example shows you must include *work.std_ovl_vhdl_components.all*, which has a package of declarations for all components. More than one checker can be included if needed.

Note

Each checker requires its corresponding PSL code and architecture. So, include **.psl* and **_psl_logic.vhd* files when compiling and simulating each checker. In addition, compilers typically have a tool-specific switch for PSL files.

std_ulogic Wrappers

The *std_ovl_u_components.vhd* file contains the *std_ovl_u_components* package and ovl_*checker_type* components that have *std_ulogic/std_ulogic_vector* ports. These components are wrappers for the ovl_*checker* components in the *accellera_ovl_vhdl* library. As these *std_ulogic* wrappers have the same entity names as the checkers in the *accellera_ovl_vhdl* library, the *std_ovl_u_components.vhd* file should be compiled into the *accellera_ovl_vhdl_u* library.To use these components, add the following declarations to the instantiating code:

```
library accellera_ovl_vhdl;
use accellera_ovl_vhdl.std_ovl.all;
library accellera_ovl_vhdl_u;
-- optional - not needed if using direct instantiation
use accellera_ovl_vhdl_u.std_ovl_u_components.all;
```

Number of Checkers in a Simulation

To print the number of OVL checkers initialized in a simulation set *init_msg_ctrl* and *init_count_ctrl* items to OVL_ON and include the following code:

```
library accellera_ovl_vhdl;
use accellera_ovl_vhdl.std_ovl.all;
use accellera_ovl_vhdl.std_ovl_procs.all;
use work.proj_pkg.all;
entity tb is
end entity tb;

architecture tb of tb is
...
begin
...
  ovl_print_init_count_p : process
  begin
     wait for 0 ns;
     ovl_print_init_count_proc(ovl_proj_controls);
     wait; -- forever
  end process ovl_print_init_count_p;
end architecture tb;
```

“2-state” and “X/Z-check” Assertions in VHDL

The OVL checker components contain separate sections of code that implement the “2-state” and “X/Z-check” assertion checks. These terms are derived from the use of the Verilog family of HDLs. However, the VHDL OVL implementation uses 9-state *std_logic* values so 2-state assertion checks and X/Z checks have a slightly different meaning for the VHDL OVL checkers. Note that the VHDL implementation is fully compatible with the Verilog implementation.

Verilog OVL checkers’ assertion checks are mapped to VHDL as follows:

- 2-state assertion checks:
  - Verilog 0 => VHDL ‘0’/‘L’
  - Verilog 1 => VHDL ‘1’/‘H’
- X/Z-checks:
  - Verilog X or Z => VHDL ‘X’, ‘Z’, ‘W’, ‘U’ or ‘-’.

Synthesizing the VHDL OVL Library

All code in the pure- VHDL implementation is synthesizable—apart from the *path_name* attribute in the architectures and the *std_ovl_procs.vhd* file. Until all the synthesis tool vendors support the use of the *path_name* attribute, a synthesizable version of the architectures is provided in the *std_ovl/vhdl93/syn_src* directory. The order of analysis for the synthesis version of the library is as follows (ensure that the files are compiled into the *accellera_ovl_vhdl* library):

1. `std_ovl/std_ovl.vhd`
2. `std_ovl/std_ovl_components.vhd`
3. `std_ovl/vhdl93/syn_src/std_ovl_procs_syn.vhd`
4. `std_ovl/std_ovl_clock_gating.vhd`
5. `std_ovl/std_ovl_reset_gating.vhd`
6. `std_ovl/ovl_*.vhd`
7. `std_ovl/vhdl93/syn_src/ovl_*_rtl.vhd`

Primary VHDL Packages

std_ovl.vhd

```
-- Accellera Standard V2.7 Open Verification Library (OVL).
-- Accellera Copyright (c) 2009 - 2012. All rights reserved.

library ieee;
use ieee.std_logic_1164.all;
package std_ovl is

  -- subtypes for common generics
  subtype ovl_severity_level         is integer            range -1 to 3;
  subtype ovl_severity_level_natural is ovl_severity_level range  0 to
                                                 ovl_severity_level'high;
  subtype ovl_property_type          is integer            range -1 to 4;
  subtype ovl_property_type_natural  is ovl_property_type  range  0 to
                                                  ovl_property_type'high;
  subtype ovl_coverage_level         is integer           range -1 to 15;
  subtype ovl_coverage_level_natural is ovl_coverage_level range  0 to
                                                 ovl_coverage_level'high;
  subtype ovl_active_edges           is integer            range -1 to 3;
  subtype ovl_active_edges_natural   is ovl_active_edges   range  0 to
                                                   ovl_active_edges'high;
  subtype ovl_reset_polarity         is integer            range -1 to 1;
  subtype ovl_reset_polarity_natural is ovl_reset_polarity range  0 to
                                                 ovl_reset_polarity'high;
  subtype ovl_gating_type            is integer            range -1 to 2;
  subtype ovl_gating_type_natural    is ovl_gating_type    range  0 to
                                                    ovl_gating_type'high;
```

```
-- subtypes for checker specific generics
subtype ovl_necessary_condition    is integer           range  0 to 2;
subtype ovl_action_on_new_start    is integer           range  0 to 2;
subtype ovl_inactive               is integer           range  0 to 2;
subtype ovl_positive_2             is integer           range  2 to
                                                         integer'high;
subtype ovl_chk_overlap            is integer           range  0 to 1;

-- subtypes for control constants
subtype ovl_ctrl                   is integer           range  0 to 1;
subtype ovl_msg_default_type       is string(1 to 50);

-- user modifiable library control items
type ovl_ctrl_record is record
  -- generate statement controls
  xcheck_ctrl                 : ovl_ctrl;
  implicit_xcheck_ctrl        : ovl_ctrl;
  init_msg_ctrl               : ovl_ctrl;
  init_count_ctrl             : ovl_ctrl;
  assert_ctrl                 : ovl_ctrl;
  cover_ctrl                  : ovl_ctrl;
  global_reset_ctrl           : ovl_ctrl;
  finish_ctrl                 : ovl_ctrl;
  gating_ctrl                 : ovl_ctrl;

  -- user configurable library constants
  max_report_error            : natural;
  max_report_cover_point      : natural;
  runtime_after_fatal         : string(1 to 10);

  -- default values for common generics
  severity_level_default      : ovl_severity_level_natural;
  property_type_default       : ovl_property_type_natural;
  msg_default                 : ovl_msg_default_type;
  coverage_level_default      : ovl_coverage_level_natural;
  clock_edge_default          : ovl_active_edges_natural;
  reset_polarity_default      : ovl_reset_polarity_natural;
  gating_type_default         : ovl_gating_type_natural;

end record ovl_ctrl_record;

-- global signals
signal ovl_global_reset_signal           : std_logic;
signal ovl_end_of_simulation_signal      : std_logic := '0';

-- global variable
shared variable ovl_init_count           : natural := 0;
```

```
----------------------------------------------------------------------
----------------------------------------------------------------------
-- Hard-coded library constants
-- NOTE:  These constants must not be changed by users. Users can
-- configure the library using the ovl_ctrl_record. Please see
-- "ovl_ctrl_record Record" on page 45.
----------------------------------------------------------------------
----------------------------------------------------------------------
constant OVL_VERSION                    : string := "V2.7";

-- This constant may be changed in future releases of the library or
-- by EDA vendors.
constant OVL_FIRE_WIDTH                 : natural := 3;

constant OVL_NOT_SET                    : integer := -1;

-- generate statement control constants
constant OVL_ON                         : ovl_ctrl := 1;
constant OVL_OFF                        : ovl_ctrl := 0;

-- fire bit selection constants
constant OVL_FIRE_2STATE                : integer := 0;
constant OVL_FIRE_XCHECK                : integer := 1;
constant OVL_FIRE_COVER                 : integer := 2;

-- severity level
constant OVL_SEVERITY_LEVEL_NOT_SET     : ovl_severity_level
                                          := OVL_NOT_SET;
constant OVL_FATAL                      : ovl_severity_level := 0;
constant OVL_ERROR                      : ovl_severity_level := 1;
constant OVL_WARNING                    : ovl_severity_level := 2;
constant OVL_INFO                       : ovl_severity_level := 3;

-- coverage levels
constant OVL_COVERAGE_LEVEL_NOT_SET     : ovl_coverage_level
                                          := OVL_NOT_SET;
constant OVL_COVER_NONE                 : ovl_coverage_level := 0;
constant OVL_COVER_SANITY               : ovl_coverage_level := 1;
constant OVL_COVER_BASIC                : ovl_coverage_level := 2;
constant OVL_COVER_CORNER               : ovl_coverage_level := 4;
constant OVL_COVER_STATISTIC            : ovl_coverage_level := 8;
constant OVL_COVER_ALL                  : ovl_coverage_level := 15;

-- property type
constant OVL_PROPERTY_TYPE_NOT_SET      : ovl_property_type
                                          := OVL_NOT_SET;
constant OVL_ASSERT                     : ovl_property_type := 0;
constant OVL_ASSUME                     : ovl_property_type := 1;
constant OVL_IGNORE                     : ovl_property_type := 2;
constant OVL_ASSERT_2STATE              : ovl_property_type := 3;
constant OVL_ASSUME_2STATE              : ovl_property_type := 4;

-- active edges
constant OVL_ACTIVE_EDGES_NOT_SET       : ovl_active_edges
                                          := OVL_NOT_SET;
constant OVL_NOEDGE                     : ovl_active_edges := 0;
constant OVL_POSEDGE                    : ovl_active_edges := 1;
constant OVL_NEGEDGE                    : ovl_active_edges := 2;
```

```
constant OVL_ANYEDGE                    : ovl_active_edges := 3;

-- necessary condition
constant OVL_TRIGGER_ON_MOST_PIPE       : ovl_necessary_condition := 0;
constant OVL_TRIGGER_ON_FIRST_PIPE      : ovl_necessary_condition := 1;
constant OVL_TRIGGER_ON_FIRST_NOPIPE    : ovl_necessary_condition := 2;

-- action on new start
constant OVL_IGNORE_NEW_START           : ovl_action_on_new_start := 0;
constant OVL_RESET_ON_NEW_START         : ovl_action_on_new_start := 1;
constant OVL_ERROR_ON_NEW_START         : ovl_action_on_new_start := 2;

-- inactive levels
constant OVL_ALL_ZEROS                  : ovl_inactive := 0;
constant OVL_ALL_ONES                   : ovl_inactive := 1;
constant OVL_ONE_COLD                   : ovl_inactive := 2;

-- reset polarity
constant OVL_RESET_POLARITY_NOT_SET     : ovl_reset_polarity
                                          := OVL_NOT_SET;
constant OVL_ACTIVE_LOW                 : ovl_reset_polarity := 0;
constant OVL_ACTIVE_HIGH                : ovl_reset_polarity := 1;

-- gating type
constant OVL_GATEING_TYPE_NOT_SET       : ovl_gating_type
                                          := OVL_NOT_SET;
constant OVL_GATE_NONE                  : ovl_gating_type := 0;
constant OVL_GATE_CLOCK                 : ovl_gating_type := 1;
constant OVL_GATE_RESET                 : ovl_gating_type := 2;

-- ovl_next check_overlapping values
constant OVL_CHK_OVERLAP_OFF            : ovl_chk_overlap := 1;
constant OVL_CHK_OVERLAP_ON             : ovl_chk_overlap := 0;

-- checker xcheck type
constant OVL_IMPLICIT_XCHECK            : boolean := false;
constant OVL_EXPLICIT_XCHECK            : boolean := true;

-- default values
constant OVL_SEVERITY_DEFAULT      : ovl_severity_level
                                     := OVL_ERROR;
constant OVL_PROPERTY_DEFAULT      : ovl_property_type
                                     := OVL_ASSERT;
constant OVL_MSG_NUL           : string(10 to ovl_msg_default_type'high)
                                     := (others => NUL);
constant OVL_MSG_DEFAULT           : ovl_msg_default_type
                                     := "VIOLATION" & OVL_MSG_NUL;
constant OVL_MSG_NOT_SET           : string
                                     := "";
constant OVL_COVER_DEFAULT         : ovl_coverage_level
                                     := OVL_COVER_BASIC;
constant OVL_CLOCK_EDGE_DEFAULT    : ovl_active_edges
                                     := OVL_POSEDGE;
constant OVL_RESET_POLARITY_DEFAULT : ovl_reset_polarity
                                     := OVL_ACTIVE_LOW;
constant OVL_GATING_TYPE_DEFAULT    : ovl_gating_type
                                     := OVL_GATE_CLOCK;
```

```
  constant OVL_CTRL_DEFAULTS          : ovl_ctrl_record := (
    -- generate statement controls
    xcheck_ctrl                  => OVL_ON,
    implicit_xcheck_ctrl         => OVL_ON,
    init_msg_ctrl                => OVL_OFF,
    init_count_ctrl              => OVL_OFF,
    assert_ctrl                  => OVL_ON,
    cover_ctrl                   => OVL_OFF,
    global_reset_ctrl            => OVL_OFF,
    finish_ctrl                  => OVL_ON,
    gating_ctrl                  => OVL_ON,

    -- user configurable library constants
    max_report_error             => 15,
    max_report_cover_point       => 15,
    runtime_after_fatal          => "100 ns    ",

    -- default values for common generics
    severity_level_default       => OVL_SEVERITY_DEFAULT,
    property_type_default        => OVL_PROPERTY_DEFAULT,
    msg_default                  => OVL_MSG_DEFAULT,
    coverage_level_default       => OVL_COVER_DEFAULT,
    clock_edge_default           => OVL_CLOCK_EDGE_DEFAULT,
    reset_polarity_default       => OVL_RESET_POLARITY_DEFAULT,
    gating_type_default          => OVL_GATING_TYPE_DEFAULT

  );
end package std_ovl;
```

std_ovl_procs.vhd

```
-- Accellera Standard V2.7 Open Verification Library (OVL).
-- Accellera Copyright (c) 2009 - 2012. All rights reserved.

-- NOTE : This file not suitable for use with synthesis tools, use
--        std_ovl_procs_syn.vhd instead.

library ieee;
use ieee.std_logic_1164.all;
use work.std_ovl.all;
use std.textio.all;

package std_ovl_procs is

  ----------------------------------------------------------------------
  -- Users must only use the ovl_set_msg and ovl_print_init_count_proc
  -- subprograms. All other subprograms are for internal use only.
  ----------------------------------------------------------------------

  ---------------------------------------------------------------------
  -- ovl_set_msg
  --
  -- This allows the default message string to be set for a
  -- ovl_ctrl_record.msg_default constant.
  ----------------------------------------------------------------------
  function ovl_set_msg (
    constant default             : in    string
```

```
) return string;

------------------------------------------------------------------------
-- ovl_print_init_count_proc
--
-- This is used to print a message stating the number of checkers
-- that have been initialized.
------------------------------------------------------------------------
procedure ovl_print_init_count_proc (
  constant controls            : in    ovl_ctrl_record
);

------------------------------------------------------------------------
------------------------------------------------------------------------

------------------------------------------------------------------------
-- ovl_error_proc
------------------------------------------------------------------------
procedure ovl_error_proc (
  constant err_msg             : in    string;
  constant severity_level      : in    ovl_severity_level;
  constant property_type       : in    ovl_property_type;
  constant assert_name         : in    string;
  constant msg                 : in    string;
  constant path                : in    string;
  constant controls            : in    ovl_ctrl_record;
  signal   fatal_sig           : out   std_logic;
  variable error_count         : inout natural
);

------------------------------------------------------------------------
-- ovl_init_msg_proc
------------------------------------------------------------------------
procedure ovl_init_msg_proc (
  constant severity_level      : in    ovl_severity_level;
  constant property_type       : in    ovl_property_type;
  constant assert_name         : in    string;
  constant msg                 : in    string;
  constant path                : in    string;
  constant controls            : in    ovl_ctrl_record
);

------------------------------------------------------------------------
-- ovl_cover_proc
------------------------------------------------------------------------
procedure ovl_cover_proc (
  constant cvr_msg             : in    string;
  constant assert_name         : in    string;
  constant path                : in    string;
  constant controls            : in    ovl_ctrl_record;
  variable cover_count         : inout natural
);
```

```
------------------------------------------------------------------------
-- ovl_finish_proc
------------------------------------------------------------------------
procedure ovl_finish_proc (
  constant assert_name         : in    string;
  constant path                : in    string;
  constant runtime_after_fatal : in    string;
  signal   fatal_sig           : in    std_logic
);

------------------------------------------------------------------------
-- ovl_2state_is_on
------------------------------------------------------------------------
function ovl_2state_is_on (
  constant controls            : in    ovl_ctrl_record;
  constant property_type       : in    ovl_property_type
) return boolean;

------------------------------------------------------------------------
-- ovl_xcheck_is_on
------------------------------------------------------------------------
function ovl_xcheck_is_on (
  constant controls            : in    ovl_ctrl_record;
  constant property_type       : in    ovl_property_type;
  constant explicit_x_check    : in    boolean
) return boolean;

------------------------------------------------------------------------
-- ovl_get_ctrl_val
------------------------------------------------------------------------
function ovl_get_ctrl_val (
  constant instance_val        : in    integer;
  constant default_ctrl_val    : in    natural
) return natural;

------------------------------------------------------------------------
-- ovl_get_ctrl_val
------------------------------------------------------------------------
function ovl_get_ctrl_val (
  constant instance_val        : in    string;
  constant default_ctrl_val    : in    string
) return string;

------------------------------------------------------------------------
-- cover_item_set
------------------------------------------------------------------------
function cover_item_set (
  constant level               : in    ovl_coverage_level;
  constant item                : in    ovl_coverage_level
) return boolean;
```

```
------------------------------------------------------------------------
-- ovl_is_x
------------------------------------------------------------------------
function ovl_is_x (
  s                               : in    std_logic
) return boolean;

------------------------------------------------------------------------
-- ovl_is_x
------------------------------------------------------------------------
function ovl_is_x (
  s                               : in    std_logic_vector
) return boolean;

------------------------------------------------------------------------
-- or_reduce
------------------------------------------------------------------------
function or_reduce (
  v                               : in    std_logic_vector
) return std_logic;

------------------------------------------------------------------------
-- and_reduce
------------------------------------------------------------------------
function and_reduce (
  v                               : in    std_logic_vector
) return std_logic;

------------------------------------------------------------------------
-- xor_reduce
------------------------------------------------------------------------
function xor_reduce (
  v                               : in    std_logic_vector
) return std_logic;

------------------------------------------------------------------------
-- "sll"
------------------------------------------------------------------------
function "sll" (
  l                               : in    std_logic_vector;
  r                               : in    integer
) return std_logic_vector;

------------------------------------------------------------------------
-- "srl"
------------------------------------------------------------------------
function "srl" (
  l                               : in    std_logic_vector;
  r                               : in    integer
) return std_logic_vector;

------------------------------------------------------------------------
------------------------------------------------------------------------
-- unsigned comparison functions
-- Note: the width of l must be > 0.
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
  ----------------------------------------------------------------------
  -- ">"
  ----------------------------------------------------------------------
  function ">" (
    l                             : in    std_logic_vector;
    r                             : in    natural
  ) return boolean;

  ----------------------------------------------------------------------
  -- "<"
  ----------------------------------------------------------------------
  function "<" (
    l                             : in    std_logic_vector;
    r                             : in    natural
  ) return boolean;

  ----------------------------------------------------------------------
  ----------------------------------------------------------------------

  type err_array is array (ovl_severity_level_natural) of string
                          (1 to 16);

  constant err_typ : err_array := (OVL_FATAL   => "       OVL_FATAL",
                                   OVL_ERROR   => "       OVL_ERROR",
                                   OVL_WARNING => "       OVL_WARNING",
                                   OVL_INFO    => "       OVL_INFO");

end package std_ovl_procs;


package body std_ovl_procs is

  ----------------------------------------------------------------------
  -- Users must only use the ovl_set_msg and ovl_print_init_count_proc
  -- subprograms. All other subprograms are for internal use only.
  ----------------------------------------------------------------------

  ----------------------------------------------------------------------
  -- ovl_set_msg
  --
  -- This allows the default message string to be set for a
  -- ovl_ctrl_record.msg_default constant.
  ----------------------------------------------------------------------
  function ovl_set_msg (
    constant default        : in    string
  ) return string is
    variable new_default : ovl_msg_default_type := (others => NUL);
  begin
     new_default(1 to default'high) := default;
     return new_default;
  end function ovl_set_msg;
```

```
------------------------------------------------------------------------
-- ovl_print_init_count_proc
--
-- This is used to print a message stating the number of checkers that
-- have been initialized.
------------------------------------------------------------------------
procedure ovl_print_init_count_proc (
  constant controls            : in    ovl_ctrl_record
) is
  variable ln : line;
begin
  if ((controls.init_msg_ctrl = OVL_ON) and
            (controls.init_count_ctrl = OVL_ON)) then
    writeline(output, ln);
    write(ln, "OVL_METRICS:
     " & integer'image(ovl_init_count) & " OVL assertions initialized");
    writeline(output, ln);
    writeline(output, ln);
  end if;
end procedure ovl_print_init_count_proc;

------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
-- ovl_error_proc
------------------------------------------------------------------------
procedure ovl_error_proc (
  constant err_msg             : in    string;
  constant severity_level      : in    ovl_severity_level;
  constant property_type       : in    ovl_property_type;
  constant assert_name         : in    string;
  constant msg                 : in    string;
  constant path                : in    string;
  constant controls            : in    ovl_ctrl_record;
  signal   fatal_sig           : out   std_logic;
  variable error_count         : inout natural
) is
  variable ln : line;
  constant severity_level_ctrl : ovl_severity_level_natural :=
    ovl_get_ctrl_val(severity_level, controls.severity_level_default);
  constant property_type_ctrl  : ovl_property_type_natural  :=
    ovl_get_ctrl_val(property_type, controls.property_type_default);
  constant msg_ctrl            : string                     :=
    ovl_get_ctrl_val(msg, controls.msg_default);
begin
  error_count := error_count + 1;

  if (error_count <= controls.max_report_error) then
    case (property_type_ctrl) is
      when OVL_ASSERT | OVL_ASSUME | OVL_ASSERT_2STATE
                      | OVL_ASSUME_2STATE =>
        write(ln, err_typ(severity_level_ctrl) & " : "
                  & assert_name & " : "
                  & msg_ctrl & " : "
                  & err_msg
                  & " : severity " &
                        ovl_severity_level'image(severity_level_ctrl)
                  & " : time " & time'image(now)
```

```
                  & " " & path);
          writeline(output, ln);
        when OVL_IGNORE => null;
      end case;
    end if;

    if ((severity_level_ctrl = OVL_FATAL) and
                (controls.finish_ctrl = OVL_ON)) then
      fatal_sig <= '1';
    end if;
  end procedure ovl_error_proc;

  ----------------------------------------------------------------------
  -- ovl_init_msg_proc
  ----------------------------------------------------------------------
  procedure ovl_init_msg_proc (
    constant severity_level      : in    ovl_severity_level;
    constant property_type       : in    ovl_property_type;
    constant assert_name         : in    string;
    constant msg                 : in    string;
    constant path                : in    string;
    constant controls            : in    ovl_ctrl_record
  ) is
    variable ln : line;
    constant severity_level_ctrl : ovl_severity_level_natural :=
      ovl_get_ctrl_val(severity_level, controls.severity_level_default);
    constant property_type_ctrl  : ovl_property_type_natural  :=
      ovl_get_ctrl_val(property_type, controls.property_type_default);
    constant msg_ctrl            : string                     :=
      ovl_get_ctrl_val(msg, controls.msg_default);
  begin
    if (controls.init_count_ctrl = OVL_ON) then
      ovl_init_count := ovl_init_count + 1;
    else
      case (property_type_ctrl) is
        when OVL_ASSERT | OVL_ASSUME | OVL_ASSERT_2STATE
                        | OVL_ASSUME_2STATE =>
          write(ln, "OVL_NOTE: " & OVL_VERSION & ": "
                    & assert_name
                    & " initialized @ " & path
                    & " Severity: " &
                          ovl_severity_level'image(severity_level_ctrl)
                    & ", Message: " & msg_ctrl);
          writeline(output, ln);
        when OVL_IGNORE => NULL;
      end case;
    end if;
  end procedure ovl_init_msg_proc;
```

```
----------------------------------------------------------------------
-- ovl_cover_proc
----------------------------------------------------------------------
procedure ovl_cover_proc (
  constant cvr_msg            : in    string;
  constant assert_name        : in    string;
  constant path               : in    string;
  constant controls           : in    ovl_ctrl_record;
  variable cover_count        : inout natural
) is
  variable ln : line;
begin
  cover_count := cover_count + 1;

  if (cover_count <= controls.max_report_cover_point) then
     write(ln, "OVL_COVER_POINT : "
            & assert_name & " : "
            & cvr_msg & " : "
            & "time " & time'image(now)
            & " " & path);
    writeline(output, ln);
  end if;
end procedure ovl_cover_proc;

----------------------------------------------------------------------
-- ovl_finish_proc
----------------------------------------------------------------------
procedure ovl_finish_proc (
  constant assert_name        : in    string;
  constant path               : in    string;
 constant runtime_after_fatal : in    string;
  signal   fatal_sig          : in    std_logic
) is
  variable ln : line;
  variable runtime_after_fatal_time : time;
begin
  if (fatal_sig = '1') then
    -- convert string to time
    write(ln, runtime_after_fatal);
    read(ln, runtime_after_fatal_time);

    wait for runtime_after_fatal_time;
    report "       OVL : Simulation stopped due to a fatal error : " &
                             assert_name & " : " & "time " &
            time'image(now) & " " & path severity failure;
  end if;
end procedure ovl_finish_proc;
```

```
----------------------------------------------------------------------
-- ovl_2state_is_on
----------------------------------------------------------------------
function ovl_2state_is_on (
  constant controls            : in    ovl_ctrl_record;
  constant property_type       : in    ovl_property_type
) return boolean is
  constant property_type_ctrl  : ovl_property_type_natural  :=
    ovl_get_ctrl_val(property_type, controls.property_type_default);
begin
  return (controls.assert_ctrl = OVL_ON) and
         (property_type_ctrl  /= OVL_IGNORE);
end function ovl_2state_is_on;

----------------------------------------------------------------------
-- ovl_xcheck_is_on
----------------------------------------------------------------------
function ovl_xcheck_is_on (
  constant controls            : in    ovl_ctrl_record;
  constant property_type       : in    ovl_property_type;
  constant explicit_x_check    : in    boolean
) return boolean is
  constant property_type_ctrl  : ovl_property_type_natural  :=
    ovl_get_ctrl_val(property_type, controls.property_type_default);
begin
  return (controls.assert_ctrl           = OVL_ON)            and
         (property_type_ctrl            /= OVL_IGNORE)        and
         (property_type_ctrl            /= OVL_ASSERT_2STATE) and
         (property_type_ctrl            /= OVL_ASSUME_2STATE) and
         (controls.xcheck_ctrl           = OVL_ON)            and
        ((controls.implicit_xcheck_ctrl = OVL_ON) or explicit_x_check);
end function ovl_xcheck_is_on;

----------------------------------------------------------------------
-- ovl_get_ctrl_val
----------------------------------------------------------------------
function ovl_get_ctrl_val (
  constant instance_val        : in    integer;
  constant default_ctrl_val    : in    natural
) return natural is
begin
  if (instance_val = OVL_NOT_SET) then
    return default_ctrl_val;
  else
    return instance_val;
  end if;
end function ovl_get_ctrl_val;
```

```
------------------------------------------------------------------------
-- ovl_get_ctrl_val
------------------------------------------------------------------------
function ovl_get_ctrl_val (
  constant instance_val        : in    string;
  constant default_ctrl_val    : in    string
) return string is
  variable msg_default_width : integer := ovl_msg_default_type'high;
begin
  if (instance_val = OVL_MSG_NOT_SET) then
    -- get width of msg_default value
    for i in 1 to ovl_msg_default_type'high loop
      if (default_ctrl_val(i) = NUL) then
        msg_default_width := i - 1;
        exit;
      end if;
    end loop;

    return default_ctrl_val(1 to msg_default_width);
  else
    return instance_val;
  end if;
end function ovl_get_ctrl_val;

------------------------------------------------------------------------
-- cover_item_set
-- determines if a bit in the level integer is set or not.
------------------------------------------------------------------------
function cover_item_set (
  constant level               : in    ovl_coverage_level;
  constant item                : in    ovl_coverage_level
) return boolean is
begin
  return ((level mod (item * 2)) >= item);
end function cover_item_set;

------------------------------------------------------------------------
-- ovl_is_x
------------------------------------------------------------------------
function ovl_is_x (
  s                            : in    std_logic
) return boolean is
begin
  return is_x(s);
end function ovl_is_x;

------------------------------------------------------------------------
-- ovl_is_x
------------------------------------------------------------------------
function ovl_is_x (
  s                            : in    std_logic_vector
) return boolean is
begin
  return is_x(s);
end function ovl_is_x;
```

```
-----------------------------------------------------------------------
-- or_reduce
-----------------------------------------------------------------------
function or_reduce (
  v                             : in    std_logic_vector
) return std_logic is
  variable result : std_logic;
begin
  for i in v'range loop
    if i = v'left then
      result := v(i);
    else
      result := result or v(i);
    end if;
    exit when result = '1';
  end loop;
  return result;
end function or_reduce;

-----------------------------------------------------------------------
-- and_reduce
-----------------------------------------------------------------------
function and_reduce (
  v                             : in    std_logic_vector
) return std_logic is
  variable result : std_logic;
begin
  for i in v'range loop
    if i = v'left then
      result := v(i);
    else
      result := result and v(i);
    end if;
    exit when result = '0';
  end loop;
  return result;
end function and_reduce;

-----------------------------------------------------------------------
-- xor_reduce
-----------------------------------------------------------------------
function xor_reduce (
  v                             : in    std_logic_vector
) return std_logic is
  variable result : std_logic;
begin
  for i in v'range loop
    if i = v'left then
      result := v(i);
    else
      result := result xor v(i);
    end if;
  end loop;
  return result;
end function xor_reduce;
```

```
----------------------------------------------------------------------
-- “sll”
----------------------------------------------------------------------
function “sll” (
  l                              : in    std_logic_vector;
  r                              : in    integer
) return std_logic_vector is
begin
  return to_stdlogicvector(to_bitvector(l) sll r);
end function “sll”;

----------------------------------------------------------------------
-- “srl”
----------------------------------------------------------------------
function “srl” (
  l                              : in    std_logic_vector;
  r                              : in    integer
) return std_logic_vector is
begin
  return to_stdlogicvector(to_bitvector(l) srl r);
end function “srl”;

----------------------------------------------------------------------
----------------------------------------------------------------------
-- private functions used by “<“ and “>” functions
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
-- unsigned_num_bits
----------------------------------------------------------------------
function unsigned_num_bits (arg: natural) return natural is
  variable nbits: natural;
  variable n: natural;
begin
  n := arg;
  nbits := 1;
  while n > 1 loop
    nbits := nbits+1;
    n := n / 2;
  end loop;
  return nbits;
end unsigned_num_bits;
----------------------------------------------------------------------
-- to_unsigned
----------------------------------------------------------------------
function to_unsigned (arg, size: natural) return std_logic_vector is
  variable result: std_logic_vector(size-1 downto 0);
  variable i_val: natural := arg;
begin
  for i in 0 to result’left loop
    if (i_val mod 2) = 0 then
      result(i) := ‘0’;
    else result(i) := ‘1’;
    end if;
    i_val := i_val/2;
  end loop;
  return result;
end to_unsigned;
```

```
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    -- unsigned comparison functions
    -- Note: the width of l must be > 0.
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------

    ------------------------------------------------------------------------
    --">"
    ------------------------------------------------------------------------
    function ">" (
      l                              : in    std_logic_vector;
      r                              : in    natural
    ) return boolean is
    begin
      if is_x(l) then return false; end if;
      if unsigned_num_bits(r) > l'length then return false; end if;
      return not (l <= to_unsigned(r, l'length));
    end function ">";

    ------------------------------------------------------------------------
    -- "<"
    ------------------------------------------------------------------------
    function "<" (
      l                              : in    std_logic_vector;
      r                              : in    natural
    ) return boolean is
    begin
      if is_x(l) then return false; end if;
      if unsigned_num_bits(r) > l'length then return 0 < r; end if;
      return (l < to_unsigned(r, l'length));
    end function "<";

    ------------------------------------------------------------------------
    ------------------------------------------------------------------------

  end package body std_ovl_procs;
```

Chapter 3
OVL Checkers

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/Generics — parameters/generics 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 point messages with descriptions.
  - Cover Groups — cover group 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.

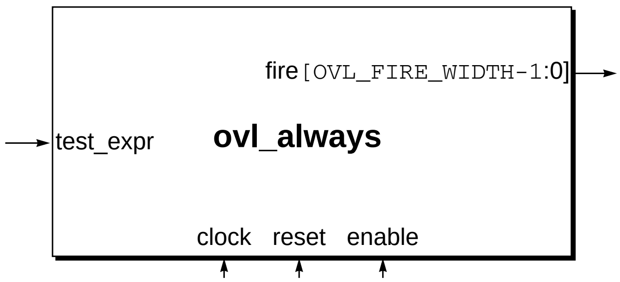
- **Examples**

  Examples of directives and checker applications.

* not applicable to all checker types.

ovl_always

Checks that the value of an expression is TRUE.



Parameters/Generics:
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/Generics

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_COVER_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 active 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

none

Cover Groups

none

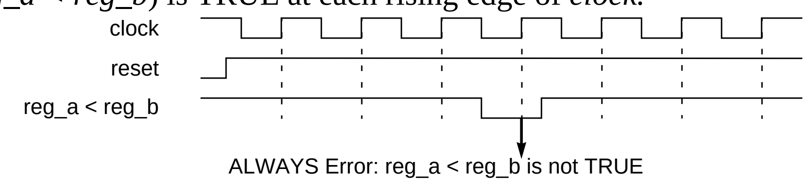
See also

ovl_always_on_edge
ovl_implication
ovl_never
ovl_proposition

Example

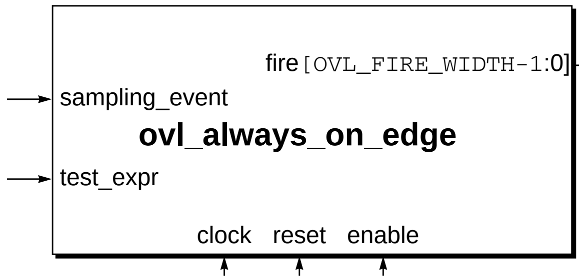
```
ovl_always #(
   ‘OVL_ERROR,                                 // severity_level
   ‘OVL_ASSERT,                                // property_type
   “Error: reg_a < reg_b is not TRUE”,         // msg
   ‘OVL_COVER_NONE,                            // coverage_level
   ‘OVL_POSEDGE,                               // 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
```

Checks that (*reg_a* < *reg_b*) is TRUE at each rising edge of *clock.*



ovl_always_on_edge

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



Parameters/Generics:

severity_level
edge_type
property_type
msg
coverage_level
clock_edge
reset_polarity
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, sampling_event, test_expr, fire);
```

Parameters/Generics

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_COVER_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 active 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 or Z	Expression value was X or Z.
sampling_event contains X or Z	Sampling event value was X or Z.

Cover Points

none

Cover Groups

none

See also

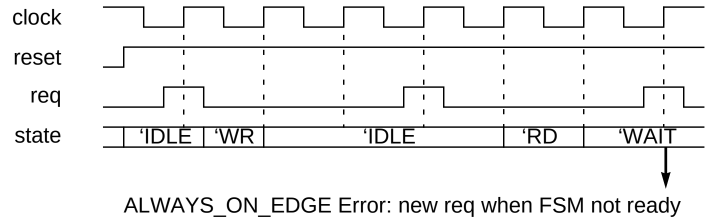
ovl_always
ovl_implication
ovl_never
ovl_proposition

Examples

Example 1

```
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
```

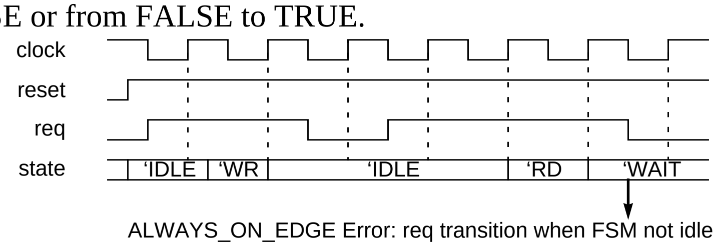
Checks that (*state* == ‘IDLE) is TRUE at each rising edge of *clock* when *req* transitions from FALSE to TRUE.



Example 2

```
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
```

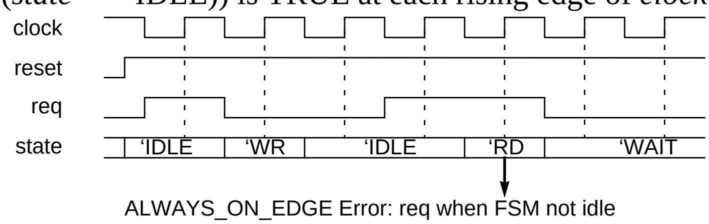
Checks that (*state* == *‘IDLE*) is TRUE at each rising edge of *clock* when *req* transitions from TRUE to FALSE or from FALSE to TRUE.



Example 3

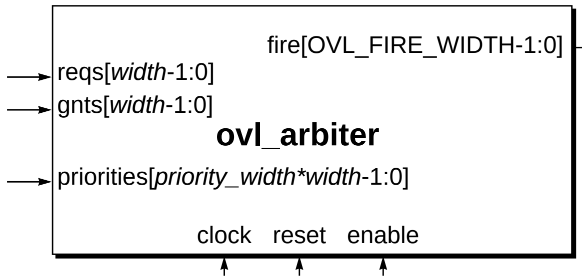
```
ovl_always_on_edge #(
   'OVL_ERROR,                                // severity_level
   'OVL_NOEDGE,                               // edge_type
   'OVL_ASSERT,                               // property_type
   "Error: req when FSM not idle",            // msg
   'OVL_COVER_NONE,                           // 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
```

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



ovl_arbiter

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



Parameters/Generics:

severity_level	*one_cycle_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,
         one_cycle_gnt_check, priority_check, arbitration_rule,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, reqs, priorities, gnts, fire);
```

Parameters/Generics

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

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.
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_COVER_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.
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).

gnts[*width*-1:0]	Concatenation of grant signals from the arbiter. Each bit in the vector is a grant to the corresponding channel.
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 checks that 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.` *`one_cycle_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. Because this value is held internally, the checker cannot operate correctly until reset.
grants contains X or Z	Grants contained X or Z bits. Because this value is held internally, the checker cannot operate correctly until reset.
priorities contains X or Z	Priorities contained X or Z bits.

Cover Points

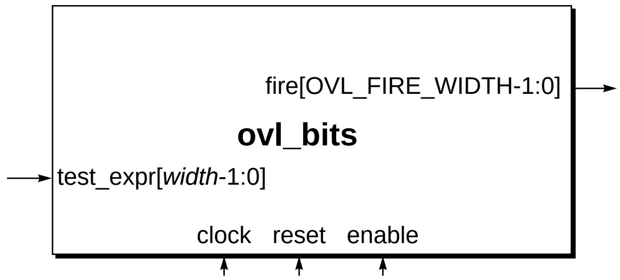
`cover_req_granted`	BASIC — Number of granted requests for each channel.
`cover_req_aborted`	BASIC — Number of aborted requests for each channel.
`cover_req_granted_at_min_cks`	CORNER — Number of times grant was issued *min_cks* cycles after its request was asserted.
`cover_req_granted_at_max_cks`	CORNER — Number of times grant was issued *max_cks* cycles 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.

Cover Groups

`time_to_grant`	Number of grants with the specified request-to-grant latency. Bins are: • *time_to_grant_good*[*min_cks*:*max_cks*] — bin index is the observed latency in clock cycles. • *time_to_grant_bad* — default.
`concurrent_requests`	Number of cycles with the specified number of concurrent requests. Bins are: • *observed_reqs_good*[1:*width*] — bin index is the number of concurrent requests.

ovl_bits

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



Parameters/Generics:
severity_level
width
asserted
min
max
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion

Syntax

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

Parameters/Generics

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	Maximum number of bits in *test_expr* that should be asserted (or deasserted). *Max* must be ≥ *min*. 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_COVER_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 *asserted* is 1) or FALSE bits (if *asserted* is 0). If the count is < *min* a min violation occurs and if the count is > *max*, a max violation occurs. X and Z bits are not included in the bit count.

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. *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. *asserted* = 0 The number of FALSE bits in the value of *test_expr* was more than the maximum specified by *max*.
`Illegal parameter values set where min > max`	Max is not 0, but *max* < *min*.

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*.

Cover Groups

none

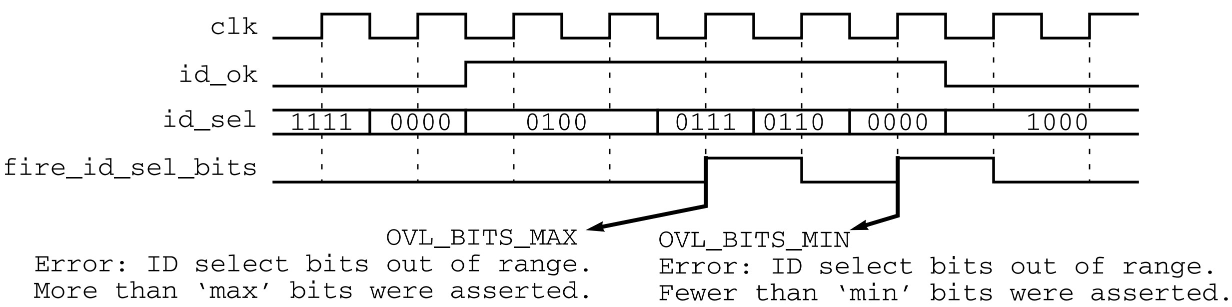
See also

ovl_mutex
ovl_one_cold
ovl_one_hot

Examples

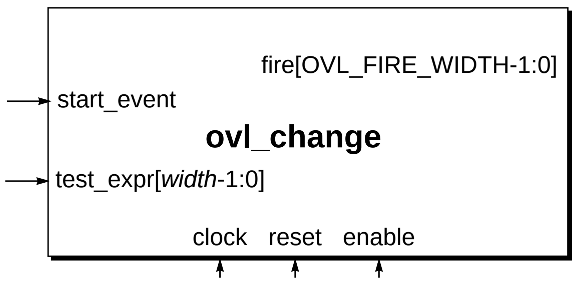
```
ovl_bits #(
    `OVL_ERROR,                                     // severity_level
    4,                                              // width
    1,                                              // asserted
    1,                                              // min
    2,                                              // max
    `OVL_ASSERT,                                    // property_type
    "Error: ID select bits out of range.",          // msg
    `OVL_COVER_NONE,                                // coverage_level
    `OVL_POSEDGE,                                   // clock_edge
    `OVL_ACTIVE_LOW,                                // reset_polarity
    `OVL_GATE_CLOCK )                               // gating_type
    ovl_id_sel_bits_in_range (
        clk,                                        // clock
        reset,                                      // reset
        id_ok,                                      // enable
        id_sel,                                     // test_expr
        fire_id_sel_bits);                          // fire
```

Checks that *id_sel* has exactly 1 or 2 TRUE bits each *clk* cycle *id_ok* is TRUE.



ovl_change

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



Parameters/Generics:

severity_level
width
num_cks
action_on_new_start
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-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/Generics

`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_COVER_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 (even if *test_expr* changed).

- 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 pending check (no violation occurs even if the current cycle is *num_cks* cycles after the start event and *test_expr* has not changed) and initiates a new check with the current value of *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 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.

Cover Groups

none

See also

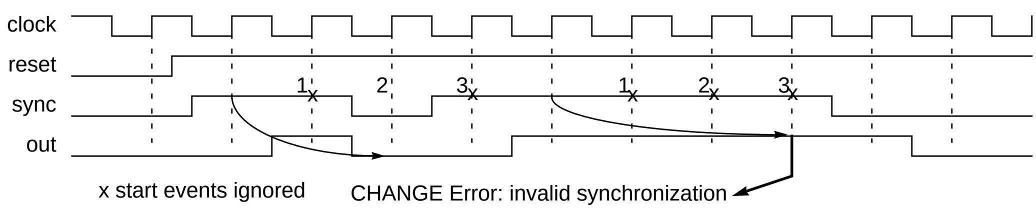
ovl_time
ovl_unchange
ovl_win_change
ovl_win_unchange
ovl_window

Examples

Example 1

```
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
```

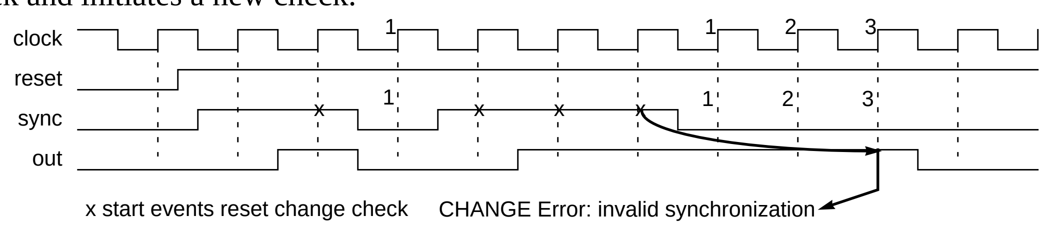
Checks that *out* changes within 3 cycles after *sync* asserts. New starts are ignored.



Example 2

```
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
```

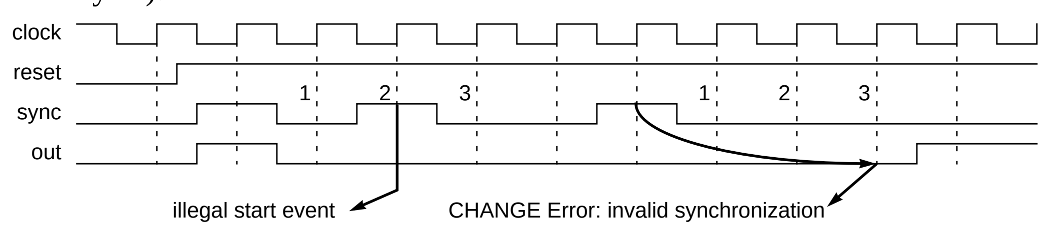
Checks that *out* changes within 3 cycles after *sync* asserts. A new start terminates the pending check and initiates a new check.



Example 3

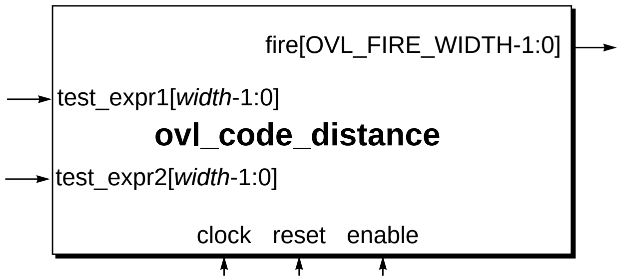
```
ovl_change #(
   `OVL_ERROR,                                    // severity_level
   1,                                             // width
   3,                                             // num_cks
   `OVL_ERROR_ON_NEW_START,                       // action_on_new_start
   `OVL_ASSERT,                                   // 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
```

Checks 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).



ovl_code_distance

Checks 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.



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

Class: 1-cycle assertion

Syntax

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

Parameters/Generics

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_COVER_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_expr1* 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*.

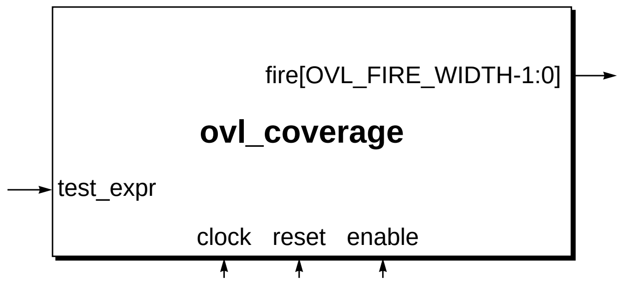
`cover_code_distance_at_max`	CORNER — Number of cycles *test_expr1* changed to a value whose code distance from *test_expr2* was *max*.

Cover Groups

`observed_code_distance`	Number of cycles *test_expr1* changed to a value having the specified code distance from *test_expr2*. Bins are: • *observed_code_distance_good*[*min*:*max*] — bin index is the code distance from *test_expr2*. • *observed_code_distance_bad* — default.

ovl_coverage

Ensures that an HDL statement is covered during simulation.



Parameters/Generics:
severity_level
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion

Syntax

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

Parameters/Generics

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_COVER_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 checker. The checker samples on the rising edge of the clock.
reset	Synchronous reset signal indicating completed initialization.
enable	Expression that indicates whether or not to check *test_expr*.

`test_expr`	Signal 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 *test_expr* must not be 1 when the checker is enabled. The checker checks the single-bit expression *test_expr* at each rising edge of *clock* whenever *enable* is TRUE. If *test_expr* is 1, the assertion fails and *msg* is printed.

This checker is used to determine coverage of the *test_expr* and to gather coverpoint data. As such, the sense of the assertion is reversed. Unlike other OVL checkers (which verify assertions that are not expected to fail), *ovl_coverage* checkers' assertions are intended to fail. You can set *property_type* to `OVL_IGNORE to disable the OVL_COVERED assertion check, but retain the collection of cover point data.

Assertion Checks

`COVERAGE`	`The HDL statement was covered.`
	Expression evaluated to 1.

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_computations_checked`	STATISTIC — Number of times *test_expr* was 1 when *enable* was TRUE.

Cover Groups

None

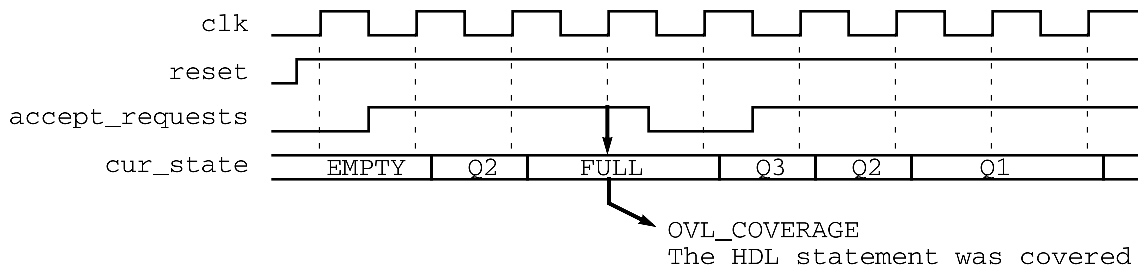
See also

ovl_value_coverage

Examples

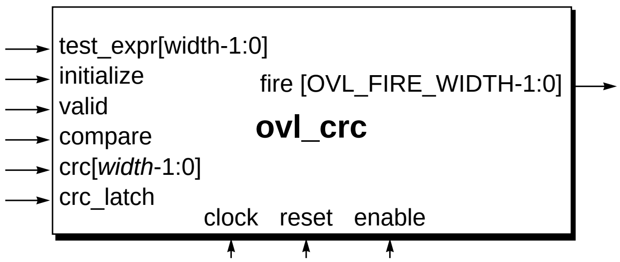
```
ovl_coverage #(
   .severity_level(`OVL_INFO),
   .property_type(`OVL_ASSERT),
   .msg("OVL_COVERAGE: queue full"),
   .coverage_level(`OVL_COVER_ALL))
   ovl_cover_queue_state_full(
      .clock(clock),
      .reset(reset),
      .enable(accept_requests),
      .test_expr(cur_state == FULL),
.fire(fire));
```

Issues a coverage message when *accept_requests* is TRUE and *cur_state* is FULL at the rising edge of *clock*.



ovl_crc

Ensures that the CRC checksum values for a specified expression are calculated properly.



Parameters/Generics:

severity_level	*big_endian*
width	*reverse_endian*
data_width	*invert*
crc_width	*combinational*
crc_enable	*property_type*
crc_latch_enable	*msg*
polynomial	*coverage_level*
standard_polynomial	*clock_edge*
initial_value	*reset_polarity*
lsb_first	*gating_type*

Class: event-bounded assertion

Syntax

```
ovl_crc
     [#(severity_level, width, data_width, crc_width, crc_enable,
        crc_latch_enable, polynomial, standard_polynomial,
        initial_value, lsb_first, big_endian, reverse_endian, invert,
        combinational, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, initialize, valid,
      compare, crc, crc_latch, fire);
```

Parameters/Generics

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

`data_width`	Width of a data item in the message stream.
	`data_width = 0`
	Data item width is *width* bits (i.e., *test_expr* holds a complete data item).
	`data_width =` $n \times width$ `(`n` > 0)`
	Data item width is *n* times the width of *test_expr*. Each data item is the concatenation of the values of *test_expr* collected over *n* valid cycles. For example, if *test_expr* has the values 2’b11, 2’b10, 2’b01 and 2’b10 over 4 consecutive valid cycles, then the corresponding data item is 8’b11100110.

`crc_width`	Degree of the CRC generator polynomial, width of the CRC checksum and width of the *crc* port (if *crc_enable* is 1). Default: 5.

crc_enable	Which data port contains the input CRC value. *crc_enable* = 0 (Default) *Test_expr* contains the input CRC value. *Crc_width* cannot be < *width*, or a CRC check violation occurs each compare cycle. The *crc* port is ignored. *crc_enable* = 1 The *crc* port contains the complete input CRC value.
crc_latch_enable	Whether or not to latch the internal CRC register value. *crc_latch_enable* = 0 (Default) The current value of the CRC register is compared with the input CRC value when *compare* asserts. The *crc_latch* port is ignored. *crc_latch_enable* = 1 The current value of the CRC register is latched if *crc_latch* is TRUE. The latched CRC value is compared with the input CRC value when *compare* asserts.
polynomial	Normal representation of the CRC generator polynomial. Equal to the concatenation of the polynomial coefficients in descending order, skipping the high-order coefficient. For example, the *polynomial* value representing: $x^{16} + x^{12} + x^{5} + 1$ is 4h'1021 (16'b0001 0000 0010 0001). Default: 5'b00101 ($x^5 + x^2 + 1$)
standard_polynomial	Polynomial to use if *polynomial* is 0: 1 — CRC-5-USB (2'h05) 2 — CRC-7 (2'h09) 3 — CRC-16-CCITT (4'h1021) 4 — CRC-32-IEEE802.3 (8'h04C11DB7) 5 — CRC-64-ISO (16'h000000000000001B)
initial_value	Initial value of the internal CRC register. *initial_value* = 0 (Default) All 0’s, for example: 8'h00000000. *initial_value* = 1 All 1’s, for example: 8'b11111111. *initial_value* = 2 Alternating 10’s, for example: 8'b10101010. *initial_value* = 3 Alternating 01’s, for example: 8'b01010101.
lsb_first	Bit order in the CRC register. *lsb_first* = 0 (Default) MSB first bit order. *lsb_first* = 1 LSB first bit order (i.e., reflected).

big_endian	Byte order of a message data item. *big_endian* = 0 (Default) Little-endian byte order. *big_endian* = 1 Big-endian byte order.
reverse_endian	Byte order in the CRC value. *reverse_endian* = 0 (Default) Byte order is the same as the byte order of a message data item (i.e., same as the *big_endian* parameter). *reverse_endian* = 1 Byte order is the opposite of the byte order of a message data item (i.e., inverse of *big_endian* parameter).
invert	Sense of the input CRC value. *invert* = 0 (Default) Input CRC value is the CRC checksum. *invert* = 1 Input CRC value is the inverted CRC checksum.
combinational	Type of logic used to calculate CRC values. *combinational* = 0 (Default) CRC is calculated sequentially. The input CRC value is the CRC checksum for the previous cycle. *combinational* = 1 CRC is calculated combinationally. The input CRC value is the CRC checksum for the current 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_COVER_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 checker. The checker samples inputs on the rising edge of the clock.

`reset`	Synchronous reset signal indicating completed initialization.
`enable`	Expression that indicates whether or not to check the inputs.
`test_expr[width-1:0]`	Variable or expression containing the input data.
`initialize`	Initialization signal. If TRUE, the checker loads its internal CRC register with the initial value specified by the *initial_value* parameter (before reading *test_expr*).
`valid`	Data valid signal. If TRUE, the checker loads the next group of bits from the message stream (or the input CRC value if *compare* is TRUE and the *crc_enable* parameter is 0) from *test_expr*.
`compare`	CRC check signal. If TRUE, the checker initiates a crc assertion check in the current cycle.
`crc[crc_width-1:0]`	Variable or expression containing the input CRC value if the *crc_enable* parameter is 1. If *crc_enable* is 0, this port is ignored.
`crc_latch`	Internal CRC register latch signal. If TRUE, the checker loads and processes the *test_expr* value (if valid) and latches the value of the internal CRC register for comparison with an input CRC value (the next cycle *compare* asserts). This input is ignored unless *crc_latch_enable* is 1.
`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_crc* checker ensures CRC checksums are calculated properly. The checker evaluates the *initialize* signal at each rising edge of *clock* whenever *enable* is TRUE. If *initialize* is TRUE, the checker restarts its CRC calculation algorithm, which initializes the internal CRC register to the initial value specified by the *initial_value* parameter. After that, in the current cycle and in each subsequent cycle, the checker checks the *valid* signal. If *valid* is TRUE and *compare* is FALSE, the value of *test_expr* is taken as the next group of bits in the message stream. By default, this group is shifted into the internal CRC register, displacing the group at the opposite end and the internal CRC register is then updated with the CRC register value XORed with a value from a lookup table. This internal CRC value is the calculated CRC checksum for the message stream read from *test_expr* since initialization.

After initialization, the checker also checks the *compare* signal each cycle. By default:

- *width* Š *crc_width*

  If *compare* and *valid* are both TRUE, the checker compares the value of *test_expr* with the internal CRC value. If they do not match, a CRC check violation occurs.

- *width* < *crc_width*

  If *compare* and *valid* are both TRUE, the checker compares the value of *test_expr* with the first *width* bits of the internal CRC value. If they do not match, a CRC check violation occurs. Then, each successive cycle in which *compare* and *valid* are both TRUE, the checker compares the value of *test_expr* with the corresponding bits of the internal CRC value. If they do not match, a CRC check violation occurs.

Because applications for CRC checking are so diverse, the ovl_crc checker contains a generic CRC calculator adaptable to virtually any CRC scheme and implementation. The following information is required to configure the calculator properly:

- Data stream handling

  The algorithm shifts data into the CRC register and generates the internal CRC value one data item at a time. By default, the *test_expr* port contains an entire data item. However, the checker can support serial input and systems where data items are loaded in multibit pieces. In these cases, specify the width of a data item with the *data_width* parameter. The checker will accumulate the data item from *test_expr* over consecutive valid cycles and on the last cycle (i.e., when the data item is complete) shift the data item onto the CRC register.

- Algorithm controls

  The standard variations on CRC computation are configured with checker parameters. The CRC generator polynomial is specified by setting the *polynomial* parameter to its normal representation. LSB first and big-endian data representation conventions are selected by setting the *lsb_first* and *big_endian* parameters respectively to 1.

- CRC comparison

  By default, the input CRC values are embedded in the data stream seen at the *test_expr* port. Setting the *crc_enable* parameter to 1 configures the checker to take the input CRC value from the *crc* port instead, so message data load and CRC compare operations can overlap.

  Input CRC transformations that invert the sense and flip the endian nature of CRC values are controlled with the *invert* and *reverse_endian* parameters respectively.

- CRC computation timing

  CRC comparison can be adjusted to handle the different time requirements for various implementations.

  By default, the current internal CRC register value is used when comparing input and expected CRC values. Setting the *crc_latch_enable* parameter to 1 configures the checker to latch the current internal CRC register value each cycle *crc_latch* is TRUE (and then initialize the register). In the next cycle *compare* is TRUE, the input CRC value is compared with the latched value (even as a new message is being accumulated and a new CRC is being calculated).

By default, the checker assumes the input CRC is calculated sequentially, so the input CRC value reflects the message accumulated up to the previous clock cycle. Setting the *combinational* parameter to 1 configures the checker to assume the computation is combinational. The input CRC value reflects the message accumulated up to the current clock cycle.

Standard CRC polynomials:

Name	***crc_width***	**Generator Polynomial**	**polynomial**
CRC-5-USB	5	$x^5 + x^2 + 1$	2'h05
CRC-7	7	$x^7 + x^3 + 1$	2'h09
CRC-16-CCITT	16	$x^{16} + x^{12} + x^5 + 1$	4'h1021
CRC-32-IEEE802.3	32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11}$ $x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	8'h04C11DB7
CRC-64-ISO	64	$x^{64} + x^4 + x^3 + x + 1$	16'h00000000 000001B

Assertion Checks

`CRC` — `Input CRC value did not match the expected CRC value.`

`crc_enable` = 0
: *Compare* was TRUE, but the value of *test_expr* (or inverted value if *invert* is 1) does not match the internal CRC value calculated for the associated message stream.

`crc_enable` = 1
: *Compare* was TRUE, but the value of *crc* (or inverted value if *invert* is 1) does not match the internal CRC value calculated for the associated message stream.

Implicit X/Z Checks

test_expr contains X or Z	Expression contained X or Z bits.
valid contains X or Z	Expression contained X or Z bits.
initialize contains X or Z	Expression contained X or Z bits.
crc contains X or Z	Expression contained X or Z bits.
crc_latch contains X or Z	Expression contained X or Z bits.
compare contains X or Z	Expression contained X or Z bits.

Cover Points

cover_values_checked	SANITY — Number of cycles test_expr changed value.
cover_crc_ computations_checked	STATISTIC — Number of cycles the internal CRC register was updated.
cover_cycles_checked	CORNER — Number of cycles CRC checksum comparisons were performed.

Cover Groups

None

See also

none

Examples

Example 1

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(8),
   .crc_width(4),
   .crc_enable(1),
   .polynomial(4’b0101),
   .initial_value(0),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "))
   .coverage_level(‘OVL_COVER_NONE),

   CRC1(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(start_crc),
      .valid(1’b1),
      .compare(1’b1),
      .crc(crc_out),
      .crc_latch(1’b0),
.fire(fire));
```

Checks that CRC checksums are calculated properly on all active edges of the clock. The CRC generator polynomial is $x^4 + x^2 + 1$.

Example 2

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(8),
   .crc_width(4),
   .crc_enable(1),
   .crc_latch_enable(1),
   .polynomial(4’b0101),
   .initial_value(0),
.property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC2(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(start_crc),
      .valid(1’b1),
      .compare(!sel_data),
      .crc(crc_out),
      .crc_latch(data_block_rdy),
.fire(fire));
```

Checks that CRC checksums (latched when *data_block_rdy* asserts) are equal to the input CRC checksums on *crc_out* when *sel_data* deasserts. The CRC generator polynomial is $x^4 + x^2 + 1$.

Example 3

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(32),
   .crc_width(32),
   .polynomial(8’h04C11DB7),
   .initial_value(1)
   .reverse_endian(1),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC3(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(start_crc),
      .valid(data_in_valid),
      .compare(crc_valid),
      .crc(32’b0),
      .crc_latch(1’b0),
.fire(fire));
```

Checks that reverse-endian transformations of the CRC checksums equal the values on *data_in* when *data_in_valid* and *crc_valid* both assert. The CRC generator polynomial is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Example 4

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(7),
   .crc_width(7),
   .crc_latch_enable(1),
   .polynomial(7’b0001001),
   .initial_value(1),
   .big_endian(1),
   .reverse_endian(1),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC4(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(start_crc),
      .valid(data_in_valid),
      .compare(sel_crc),
      .crc(7’b0),
      .crc_latch(data_block_rdy),
.fire(fire));
```

Checks that CRC checksums (latched when *data_block_rdy* asserts) are equal to the input CRC checksums on *data_in* when *sel_crc* asserts. Data values of *data_in* are big endian and CRC values of *data_in* are little endian. The CRC generator polynomial is $x^7 + x^3 + 1$.

Example 5

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(4),
   .data_width(16),
   .crc_width(16),
   .polynomial(16’h1021),
   .initial_value(1),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC5(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(start_crc),
      .valid(data_in_valid),
      .compare(compare),
      .crc(16’b0),
      .crc_latch(1’b0),
.fire(fire));
```

Checks that the associated bits of CRC checksums equal the values on *data_in* when *data_in_valid* and *compare* both assert. Each 16-bit data item is composed of 4-bit groups accumulated over 4 consecutive valid data cycles. Each cycle a data item is complete, its value is shifted onto the CRC register and the register is updated with the internal CRC value. The input CRC value is also accumulated from *data_in* in consecutive valid data cycles (i.e., when *data_in_valid* is TRUE) if *compare* is TRUE. However, since the internal CRC value is known, a CRC check violation occurs each cycle the current group of *data_in* bits does not match the corresponding bits in the internal CRC value. The CRC generator polynomial is $x^{16} + x^{12} + x^5 + 1 \cdot$

Example 6

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(112),
   .crc_width(16),
   .crc_enable(1),
   .polynomial(16’h1021),
   .initial_value(3),
   .combinational(1),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC5(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in[127:16),
      .initialize(valid),
      .valid(valid),
      .compare(valid),
      .crc(data_in[15:0]),
      .crc_latch(1’b0),
.fire(fire));
```

Checks that every cycle *valid* is TRUE, *data_in*[15:0] equals the CRC checksum for the current value of *data_in*[127:16] with an initial value of 4’h5555. The CRC generator polynomial is $x^{16} + x^{12} + x^5 + 1 \cdot$

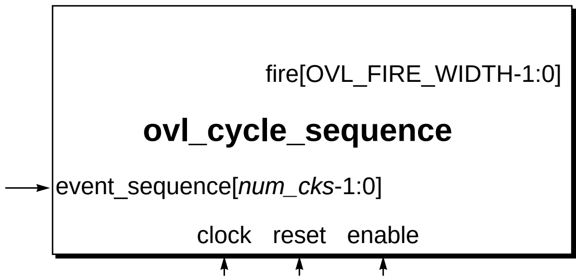
Example 7

```
ovl_crc #(
   .severity_level(‘OVL_ERROR),
   .width(128),
   .crc_width(16),
   .crc_enable(1),
   .polynomial(16’h1021),
   .property_type(‘OVL_ASSERT),
   .msg(“OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))
   CRC5(
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr(data_in),
      .initialize(1’b1),
      .valid(1’b1),
      .compare(1’b1),
      .crc(crc),
      .crc_latch(1’b0),
.fire(fire));
```

Checks that every active clock cycle, the value of *crc* equals the CRC checksum of the value of *data_in* sampled in the previous cycle. The CRC generator polynomial is $x^{16} + x^{12} + x^{5} + 1$.

ovl_cycle_sequence

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



Parameters/Generics:
severity_level
num_cks
necessary_condition
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

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/Generics

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_COVER_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:

```
1bbb –> b1bb –> bb1b –> bbb1
```

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:

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

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.

Cover Groups

none

See also

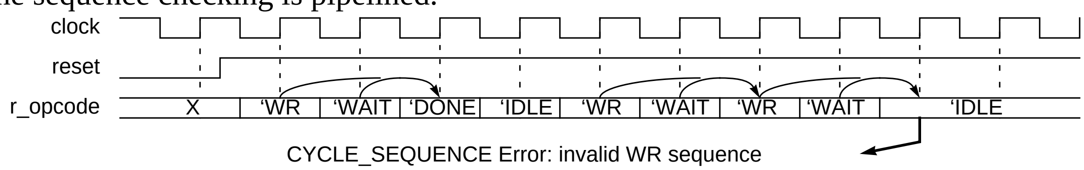
ovl_change ovl_unchange

Examples

Example 1

```
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
```

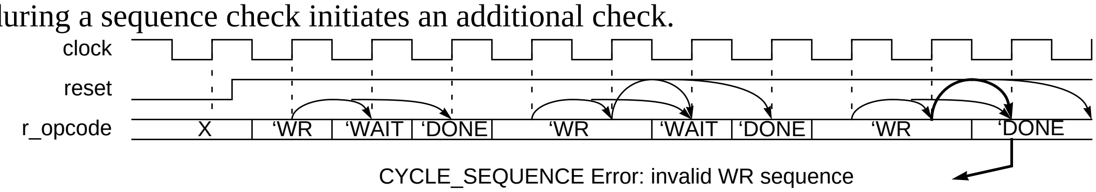
Checks that a ‘WR, ‘WAIT sequence in consecutive cycles is followed by a ‘DONE or ‘WR. The sequence checking is pipelined.



Example 2

```
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
```

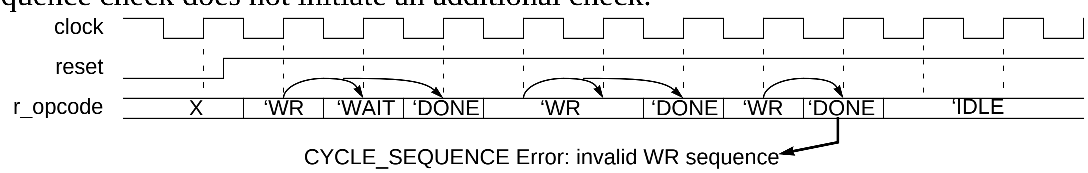
Checks that a ‘WR is followed by a ‘WAIT or another ‘WR, which is then followed by a ‘WAIT or a ‘DONE (in consecutive cycles). The sequence checking is pipelined: a new ‘WR during a sequence check initiates an additional check.



Example 3

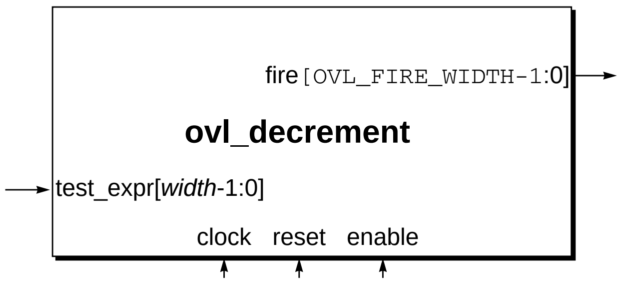
```
ovl_cycle_sequence #(
   `OVL_ERROR,                              // severity_level
   3,                                       // num_cks
   `OVL_TRIGGER_ON_FIRST_NOPIPE,            // necessary_condition
   `OVL_ASSERT,                             // property_type
   "Error: invalid WR sequence",            // 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
```

Checks 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.



ovl_decrement

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



Parameters/Generics:	*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/Generics

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_COVER_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.

Cover Groups

none

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 edge of *clock* after *reset* deasserts.

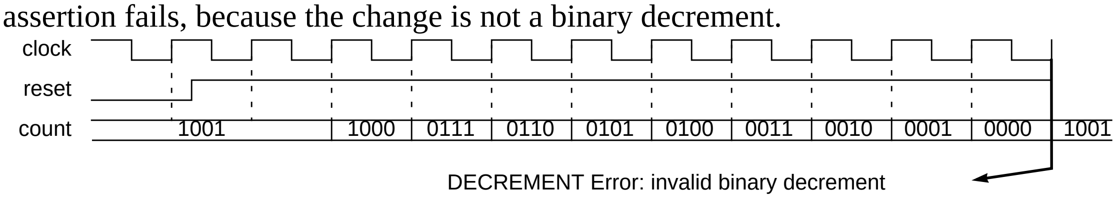
See also

ovl_delta
ovl_increment
ovl_no_underflow

Examples

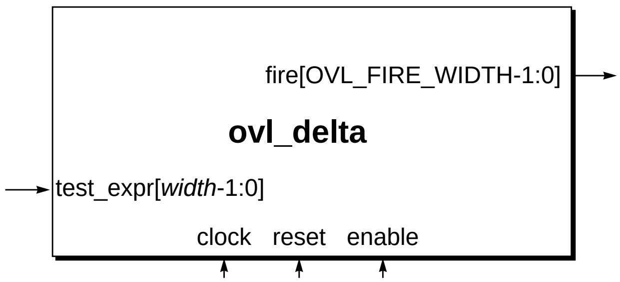
```
ovl_decrement #(
    `OVL_ERROR,                              // severity_level
    4,                                       // width
    1,                                       // value
    `OVL_ASSERT,                             // property_type
    "Error: invalid binary decrement",       // 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
```

Checks 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.



ovl_delta

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



Parameters/Generics:

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/Generics

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_COVER_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*.

Cover Groups

none

Errors

The parameters/generics *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 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.

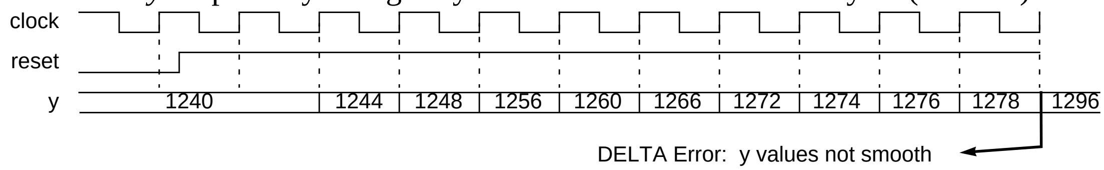
See also

ovl_decrement
ovl_increment
ovl_no_overflow
ovl_no_underflow
ovl_range

Examples

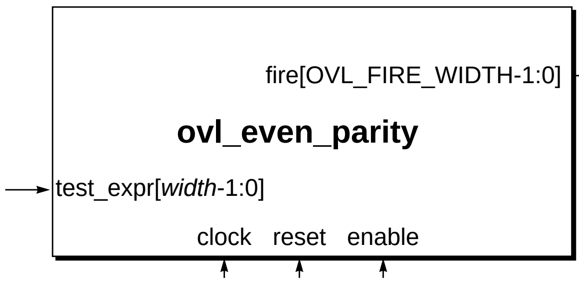
```
ovl_delta #(
    `OVL_ERROR,                           // severity_level
    16,                                   // width
    0,                                    // min
    8,                                    // max
    `OVL_ASSERT,                          // property_type
    "Error: y values 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
```

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



ovl_even_parity

Checks that the value of an expression has even parity.



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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 active 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.

Cover Groups

none

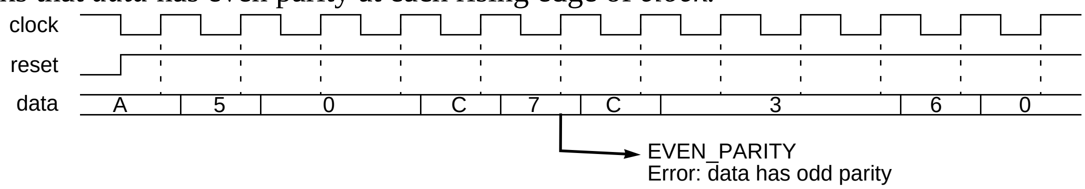
See also

ovl_odd_parity

Examples

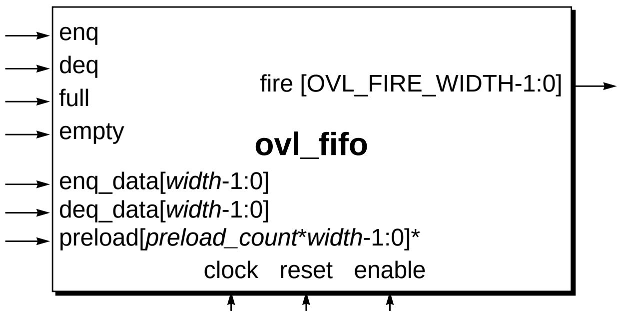
```
ovl_even_parity #(
    `OVL_ERROR,                                  // severity_level
    8,                                           // width
    `OVL_ASSERT,                                 // property_type
    "Error: data has odd parity",                // msg
    `OVL_COVER_DEFAULT,                          // 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
```

Checks that *data* has even parity at each rising edge of *clock*.



ovl_fifo

Checks the data integrity of a FIFO and checks that the FIFO does not overflow or underflow.



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

Parameters/Generics:
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

Class: event-bounded assertion

Syntax

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

Parameters/Generics

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_COVER_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.
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	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.
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).
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.

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 checks that 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/generics 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 checks that 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 checks 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 ≥ *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.

Cover Groups

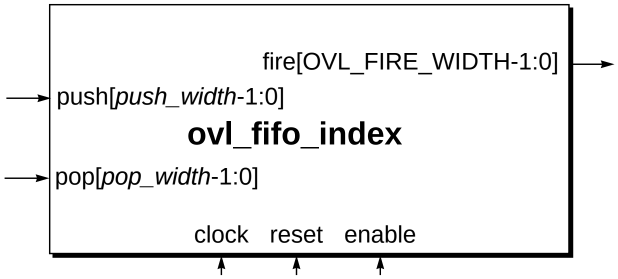
`observed_contents`	Number of cycles the number of entries in the FIFO changed to the specified value. Bins are: • observed_fifo_contents[0:*depth*] — bin index is the number of entries in the FIFO.

See also

ovl_fifo_index
ovl_no_overflow
ovl_no_underflow

ovl_fifo_index

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



Parameters/Generics:
severity_level
depth
push_width
pop_width
simultaneous_push_pop
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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 checks that 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.

Cover Groups

none

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 checks 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

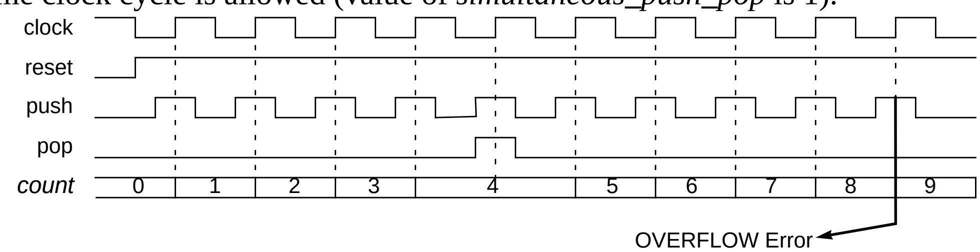
ovl_fifo
ovl_no_overflow
ovl_no_underflow

Examples

Example 1

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

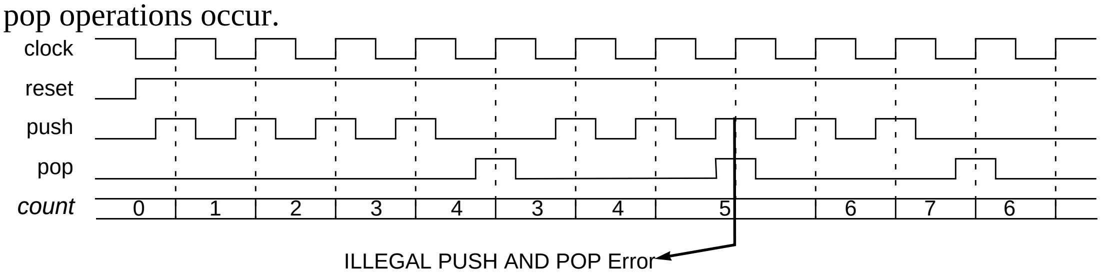
Checks that an 8-element FIFO never overflows or underflows. Only single pushes and pops can occur in a clock cycle (*push_width* and *pop_width* values are 1). A push and pop operation in the same clock cycle is allowed (value of *simultaneous_push_pop* is 1).



Example 2

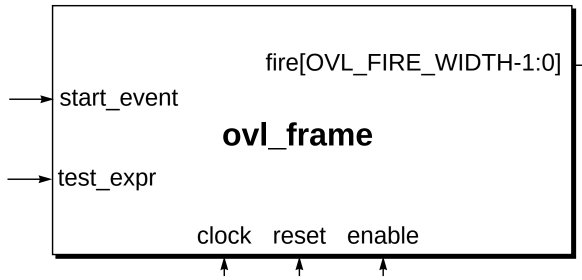
```
ovl_fifo_index #(
   `OVL_ERROR,                                  // severity_level
   8,                                           // depth
   1,                                           // push_width
   1,                                           // pop_width
   0,                                           // 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
```

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



ovl_frame

Checks 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.



Parameters/Generics:

severity_level
min_cks
max_cks
action_on_new_start
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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 cycle following 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_COVER_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* is a rising signal (i.e., has transitioned from FALSE to TRUE, either at the clock edge or in the previous cycle). A start event also occurs if *start_event* is TRUE at the active clock edge after a checker reset.

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

1. A frame violation occurs if *test_expr* is not TRUE at the start event.
2. 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.

3. 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*.
4. 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 rising, then:

  - If *test_expr* is TRUE, a frame violation occurs and all pending checks are terminated.
  - If *test_expr* is not TRUE, pending checks are terminated (no violation occurs even if the current cycle is the last cycle of a *max_cks* check or a cycle with a pending min_cks check). If *min_cks* and *max_cks* are not both 0, new frame checks are initiated.

- 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 active clock edge.

Assertion Checks

`FRAME_MIN`	Value of *test_expr* was TRUE at a rising *start_event* or before *min_cks* cycles after a rising *start_event.*
`FRAME_MAX`	Value of *test_expr* was not TRUE at a cycle starting *min_cks* cycles after a rising *start_event* and ending *max_cks* after the rising edge of *start_event*.
`FRAME_MIN0_MAX_0`	Both *min_cks* and *max_cks* are 0, but the value of *test_expr* was not TRUE at the rising edge of *start_event*.

`illegal start event`	The *action_on_new_start* parameter is set to OVL_ERROR_ON_NEW_START and a rising *start_event* occurred while a check was pending .
`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*.

Cover Groups

none

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

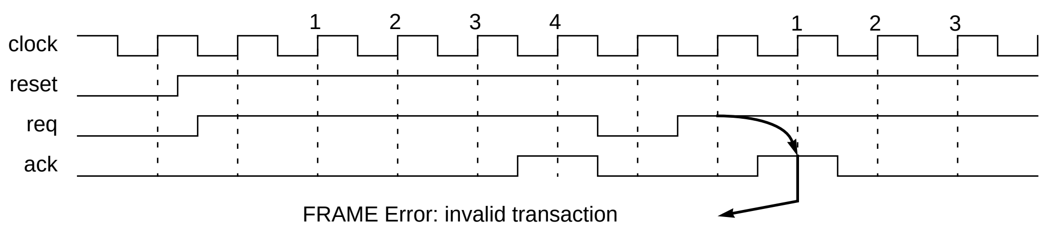
ovl_change
ovl_next
ovl_time
ovl_unchange
ovl_width

Examples

Example 1

```
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
```

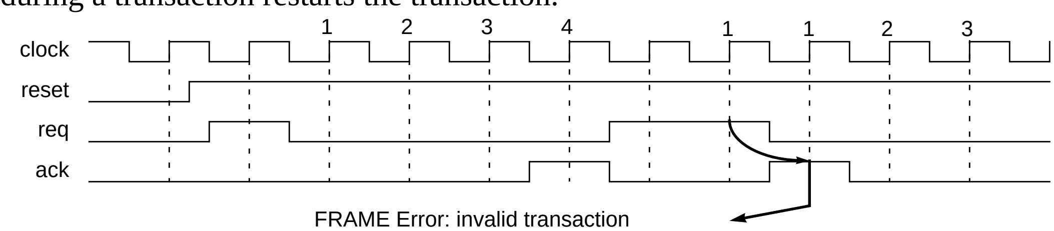
Checks that after a rising edge of *req*, *ack* goes high between 2 and 4 cycles later. New start events during transactions are not considered to be new transactions and are ignored.



Example 2

```
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
```

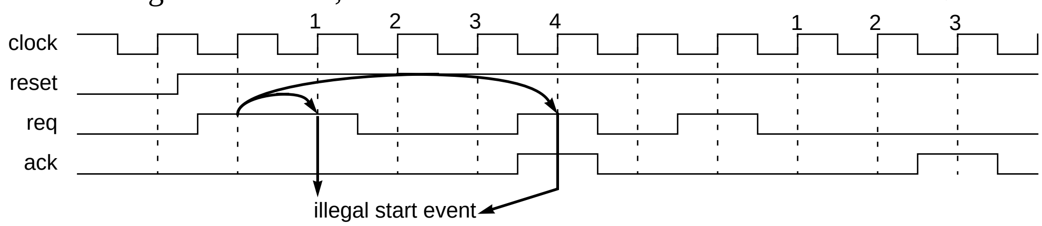
Checks that after a rising edge of *req*, *ack* goes high between 2 and 4 cycles later. A new start event during a transaction restarts the transaction.



Example 3

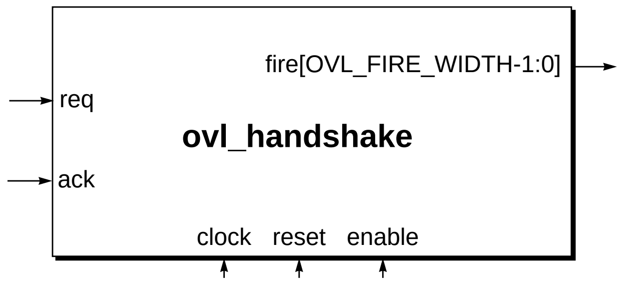
```
ovl_frame #(
   `OVL_ERROR,                                   // severity_level
   2,                                            // min_cks
   4,                                            // max_cks
   `OVL_ERROR_ON_NEW_START,                      // action_on_new_start
   `OVL_ASSERT,                                  // 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
```

Checks that after a rising edge of *req*, *ack* goes high between 2 and 4 cycles later. Also checks 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

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



Parameters/Generics:
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

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/Generics

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_COVER_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/generics. 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/generics.

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.

Cover Groups

none

See also

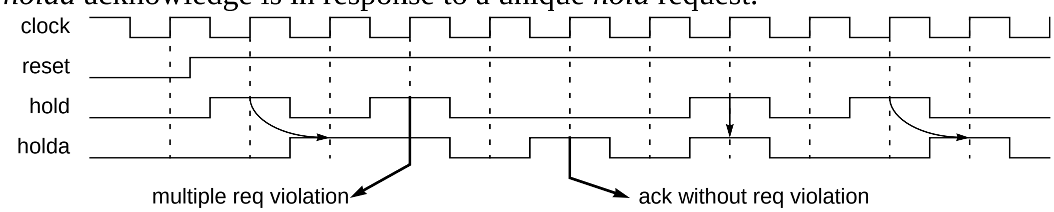
ovl_win_change
ovl_win_unchange
ovl_window

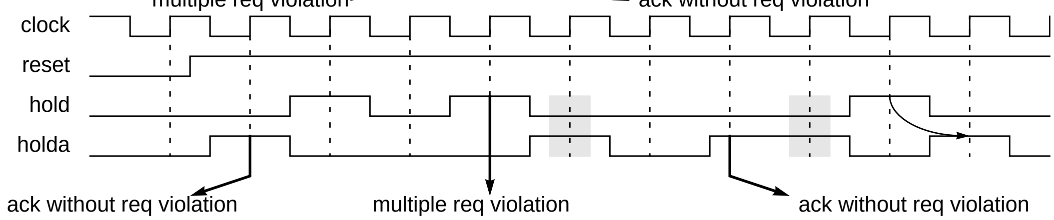
Examples

Example 1

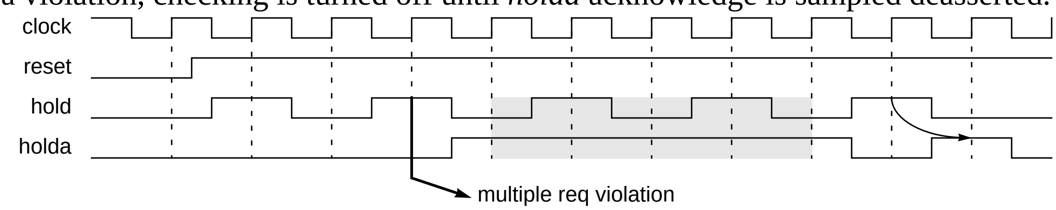
```
ovl_handshake #(
   `OVL_ERROR,                              // severity_level
   0,                                       // min_ack_cycle
   0,                                       // max_ack_cycle
   0,                                       // req_drop
   0,                                       // 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
```

Checks 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.





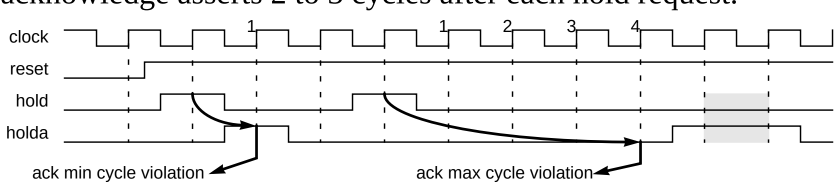
After a violation, checking is turned off until *holda* acknowledge is sampled deasserted.



Example 2

```
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
```

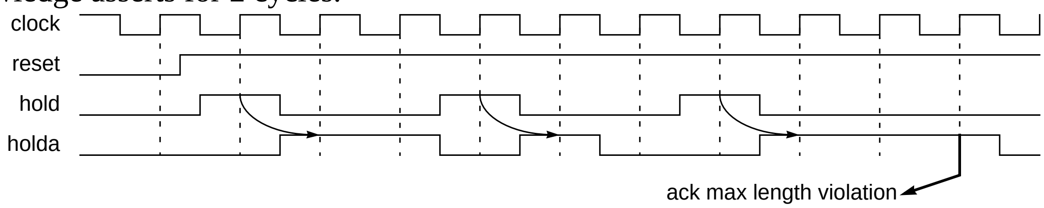
Checks that multiple *hold* requests are not made while waiting for a *holda* acknowledge and that every *holda* acknowledge is in response to a unique *hold* request. Checks that *holda* acknowledge asserts 2 to 3 cycles after each hold request.



Example 3

```
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
```

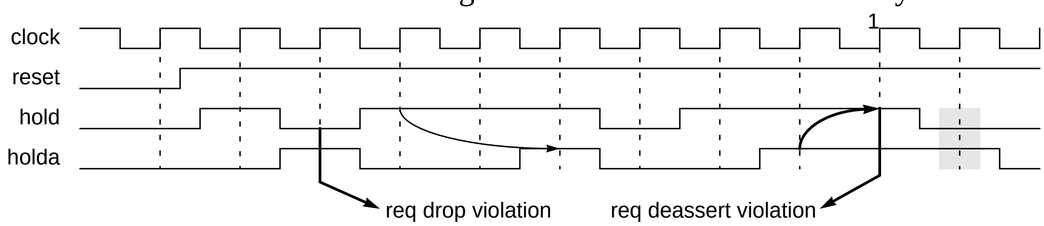
Checks that multiple *hold* requests are not made while waiting for a *holda* acknowledge and that every *holda* acknowledge is in response to a unique *hold* request. Checks that *holda* acknowledge asserts for 2 cycles.



Example 4

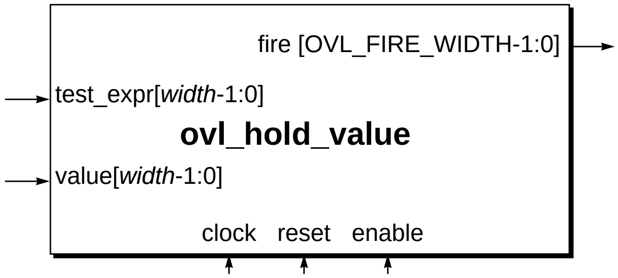
```
ovl_handshake #(
   'OVL_ERROR,                                    // severity_level
   0,                                             // min_ack_cycle
   0,                                             // max_ack_cycle
   1,                                             // req_drop
   1,                                             // 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
```

Checks 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. Checks that *hold* request remains asserted until its *holda* acknowledge and then deasserts in the next cycle.



ovl_hold_value

Checks 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.



Parameters/Generics:

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

Class: *n*-cycle assertion

Syntax

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

Parameters/Generics

`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_COVER_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.

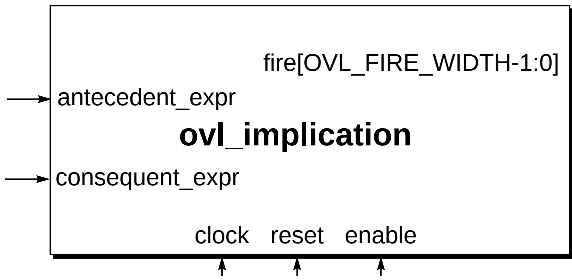
`cover_hold_value_for_max_cks`	CORNER — Indicates that the *test_expr* was held exactly equal to *value* for specified *max* clocks. Not reported if *max* = 0 and *min* > 0.
`observed_hold_time`	STATISTIC — Reports the hold times (in cycles) that occurred at least once.

Cover Groups

`observed_hold_time`	Number of times the *test_expr* value was held for the specified number of hold cycles. Bins are:
	• *observed_hold_time_good*[*min*+1:*maximum*] — bin index is the observed hold time in clock cycles. The value of *maximum* is:
	  • 1 (if *min* = *max* = 0),
	  • *min* + 4095 (if *min* > *max* = 0), or
	  • *max* + 1 (if *max* > 0).
	• *observed_hold_time_bad* — default.

ovl_implication

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



Parameters/Generics: *severity_level* *property_type* *msg* *coverage_level* *clock_edge* *reset_polarity* *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/Generics

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_COVER_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.

Cover Groups

none

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);
```

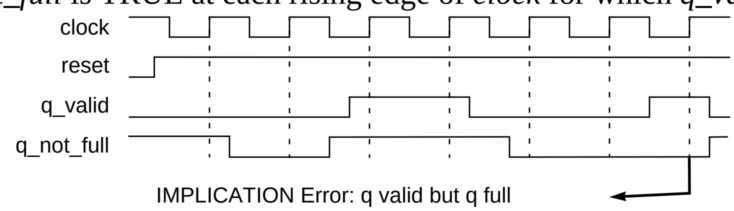
See also

ovl_always
ovl_always_on_edge
ovl_never
ovl_proposition

Examples

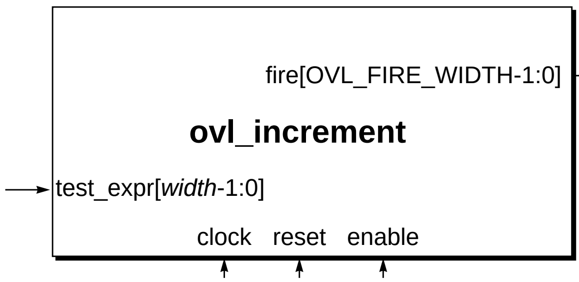
```
ovl_implication #(
    `OVL_ERROR,                                 // severity_level
    `OVL_ASSERT,                                // property_type
    "Error: q valid but q full",                // msg
    `OVL_COVER_DEFAULT,                         // coverage_level
    `OVL_POSEDGE,                               // 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
```

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



ovl_increment

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



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

Class: 2-cycle assertion

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/Generics

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_COVER_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.

Cover Groups

none

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 edge of *clock* after *reset* deasserts.

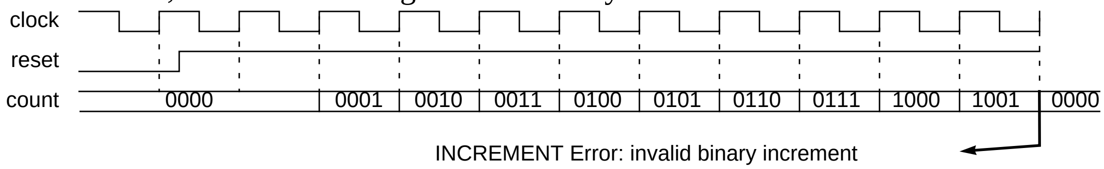
See also

ovl_decrement
ovl_delta
ovl_no_overflow

Examples

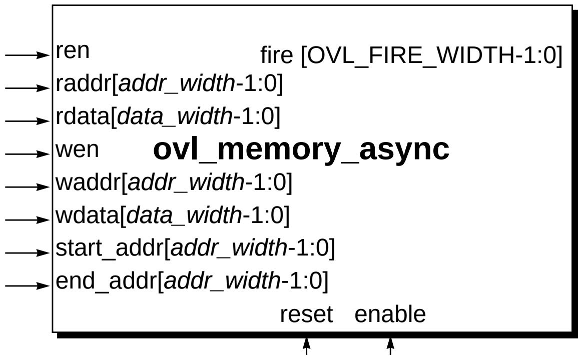
```
ovl_increment #(
   `OVL_ERROR,                              // severity_level
   4,                                       // width
   1,                                       // value
   `OVL_ASSERT,                             // property_type
   "Error: invalid binary increment",       // msg
   `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
```

Checks 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.



ovl_memory_async

Checks the integrity of accesses to an asynchronous memory.



Parameters/Generics:

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

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, wen_edge, ren_edge,
         reset_polarity, gating_type)]
   instance_name (reset, enable, start_addr, end_addr, ren, raddr, rdata,
      wen, waddr, wdata, fire);
```

Parameters/Generics

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_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_COVER_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/generics (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/generics 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

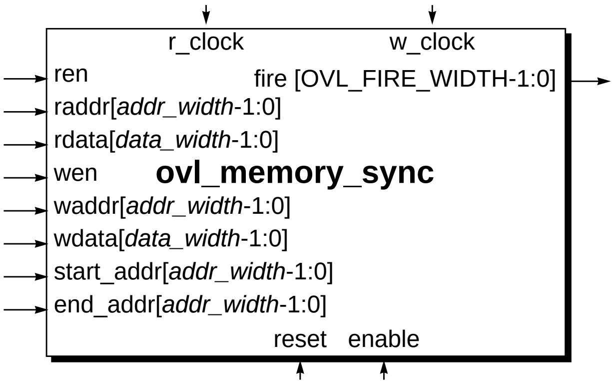
`cover_reads`	SANITY — Number of read accesses.
`cover_writes`	SANITY — Number of write accesses.
`cover_write_then_read_from_same_addr`	BASIC — Number of times a write access was followed by a read from the same address.
`cover_read_addr`	STATISTIC — Reports which addresses were read at least once.
`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*.

Cover Groups

`observed_read_addr`	Number of read operations made from the specified address. Bins are: • observed_read_addr[0:*addr_width* - 1] — bin index is the memory address.
`observed_write_addr`	Number of write operations made to the specified address. Bins are: • observed_write_addr[0:*addr_width* - 1] — bin index is the memory address.

ovl_memory_sync

Checks the integrity of accesses to a synchronous memory.



Parameters/Generics: *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*, *wen_edge*, *ren_edge*, *reset_polarity*, *gating_type*

Class: event-bounded assertion

Syntax

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

Parameters/Generics

`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_COVER_BASIC).
ren_edge	Active edge of the *r_clock* input. Default: OVL_CLOCK_EDGE_DEFAULT (OVL_POSEDGE).
wen_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).

Ports

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.
r_clock	Clock event for read operations.
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.
w_clock	Clock event for write operations.
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/generics 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.

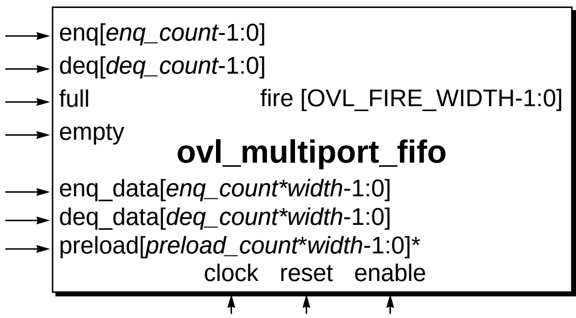
`cover_read_to_write_delays`	STATISTIC — Reports which delays (in numbers of active *w_clock* edges) from a read to the next write (to any address) occurred at least once.
`cover_write_to_read_delays`	STATISTIC — Reports which delays (in numbers of active *r_clock* edges) from a write to the next read (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.

Cover Groups

`observed_read_addr`	Number of read operations made from the specified address. Bins are: • *observed_read_addr*[0:*addr_width* - 1] — bin index is the memory address.
`observed_write_addr`	Number of write operations made to the specified address. Bins are: • *observed_write_addr*[0:*addr_width* - 1] — bin index is the memory address.
observed_delay_from_read_to_write	Number of times the delay (in cycles) between a read from a memory location and a write to that location matched the specified latency value. Bins are: • *observed_delay_from_read_to_write*[0:31] — bin index is the observed latency.
observed_delay_from_write_to_read	Number of times the delay (in cycles) between a write to a memory location and a read from that location matched the specified latency value. Bins are: • *observed_delay_from_write_to_read*[0:31] — bin index is the observed latency.

ovl_multiport_fifo

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



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

Parameters/Generics:

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

Class: *n*-cycle assertion

Syntax

```
ovl_multiport_fifo
      [#(severity_level, width, depth, enq_count, deq_count,
         preload_count, pass_thru, registered, high_water_mark,
         enq_latency, deq_latency, value_check, full_check, empty_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/Generics

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 ≤ *depth*. Default: 2.
deq_count	Number of FIFO dequeue ports. Must be ≤ *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_COVER_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 checks that 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/generics:

- 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 Z	Enqueue data item in the *enq_data* expression contained X or Z bits when it was scheduled to be enqueued onto the FIFO.
deq_data contains X or Z	Dequeue data item in the *deq_data* expression contained X or 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_enq_deq`	BASIC — Number of cycles both an enqueue and a dequeue (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_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.

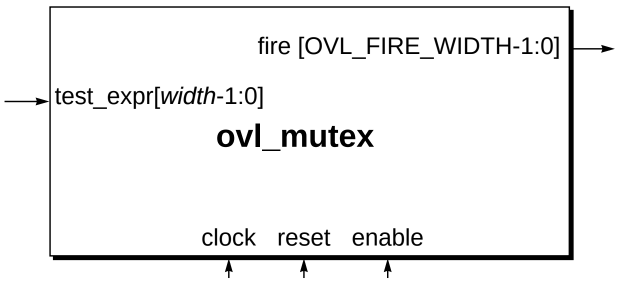
`cover_observed_counts`	STATISTIC — Reports the FIFO counts that occurred at least once.

Cover Groups

`multiport_fifo_corner`	Number of cycles the number of entries in the FIFO changed to a value with the specified characteristic. Bins are: • *cov_fifo_full_count* — FIFO is full. • *cov_fifo_empty_count* — FIFO is empty. • *cov_fifo_full_count* — number of entries is ≥ *high_water_mark*.
`multiport_fifo_statistic`	Current number of entries in the FIFO. Bin is: • *cov_observed_fifo_contents*

ovl_mutex

Checks that the bits of an expression are mutually exclusive.



Parameters/Generics:
severity_level
width
invert_mode
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

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/Generics

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_COVER_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

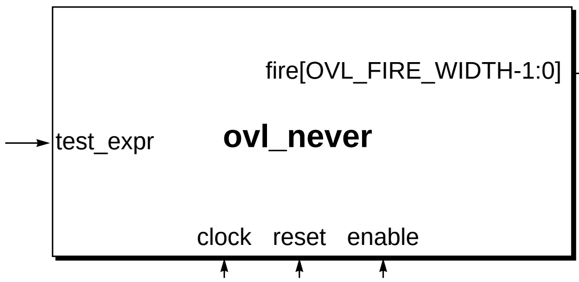
cover_values_checked	SANITY — Number of cycles *test_expr* loaded a new value.
cover_no_mutex_bits	CORNER — Number of cycles all bits in *test_expr* were TRUE and *invert_mode* = 0 or all bits in *test_expr* were FALSE and *invert_mode* = 1.
cover_all_mutexes_ covered	CORNER — Whether or not all mutex bits were covered.
cover_mutex_bitmap	STATISTIC — Number of cycles a new mutex bit was covered legally. The TRUE bits of the *mutex_bitmap* variable indicate the covered mutex bits.

Cover Groups

none

ovl_never

Checks that the value of an expression is not TRUE.



Parameters/Generics:
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/Generics

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_COVER_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 not evaluate to TRUE on the active 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_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

none

Cover Groups

none

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.

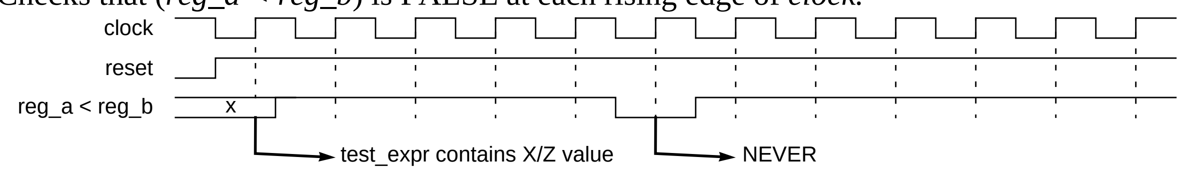
See also

ovl_always
ovl_always_on_edge
ovl_implication
ovl_proposition

Examples

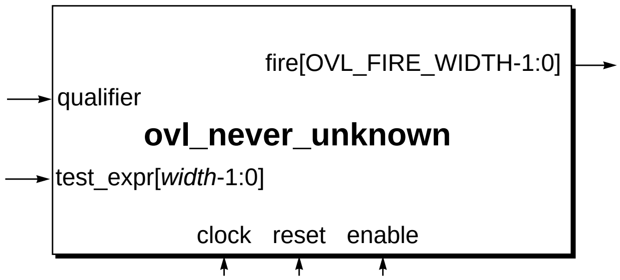
```
ovl_never #(
   `OVL_ERROR,                          // severity_level
   `OVL_ASSERT,                         // property_type
   "",                                  // msg
   `OVL_COVER_DEFAULT,                  // coverage_level
   `OVL_POSEDGE,                        // clock_edge
   `OVL_ACTIVE_LOW,                     // 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
```

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



ovl_never_unknown

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



Parameters/Generics:
severity_level
property_type
msg
coverage_level
clock_edge
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/Generics

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_COVER_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.

Cover Groups

none

Notes

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

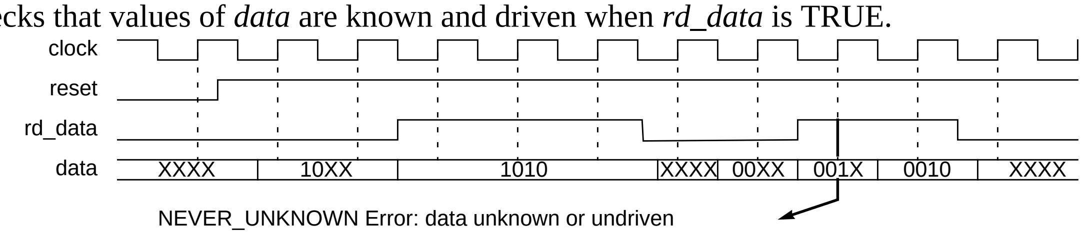
See also

ovl_never
ovl_never_unknown_async
ovl_one_cold
ovl_one_hot
ovl_zero_one_hot

Examples

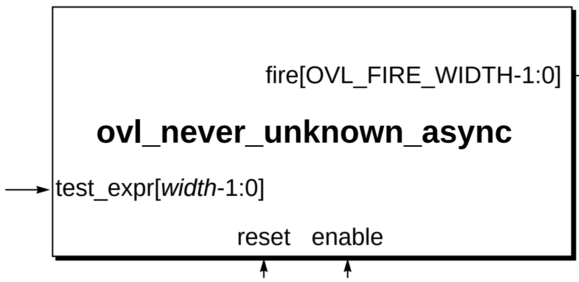
```
ovl_never_unknown #(
  `OVL_ERROR,                            // severity_level
  8,                                     // width
  `OVL_ASSERT,                           // property_type
  "Error: data unknown or undriven",     // msg
  `OVL_COVER_DEFAULT,                    // coverage_level
  `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
```

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



ovl_never_unknown_async

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



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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. Cannot be OVL_ASSUME for SVA and PSL implementations. 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_COVER_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

none

Cover Groups

none

Notes

1. If OVL_XCHECK_OFF is set, all ovl_never_unknown_async checkers are turned off.
2. The Verilog-95 version of this asynchronous checker handles ‘OVL_ASSERT, ‘OVL_ASSUME and ‘OVL_IGNORE. The SVA and PSL versions of this checker do not implement *property_type* ‘OVL_ASSUME. The SVA version uses immediate assertions and in IEEE 1800-2005 SystemVerilog immediate assertions cannot be assumptions. Assume is only available in a concurrent (clocked) form of an assertion statement. The SVA version treats ‘OVL_ASSUME as an ‘OVL_ASSERT. The PSL version generates an error if *property_type* is ‘OVL_ASSUME.

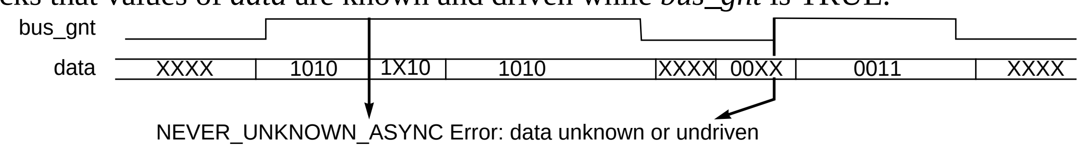
See also

ovl_never

Examples

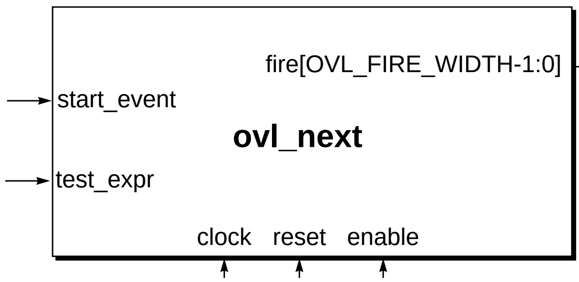
```
ovl_never_unknown_async #(
  `OVL_ERROR,                                    // severity_level
  8,                                             // width
  `OVL_ASSERT,                                   // property_type
  "Error: data unknown or undriven",             // msg
  `OVL_COVER_DEFAULT,                            // 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
```

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



ovl_next

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



Parameters/Generics:

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, enable, start_event, test_expr, fire);
```

Parameters/Generics

`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).
- If set to 1, overlap checking is not performed.

`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_COVER_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. These checks are pipelined, that is, a check is initiated each cycle start_event is TRUE (even if overlap checking is on and even if an overlap violation occurs).

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 <= 0`	The *num_cks* parameter is less than 1.
`num_cks == 1 and check_overlapping == 0`	The *num_cks* parameter is 1 and check_overlapping is 0, which turns on overlap checking even though overlaps are not relevant.

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.

Cover Groups

none

See also

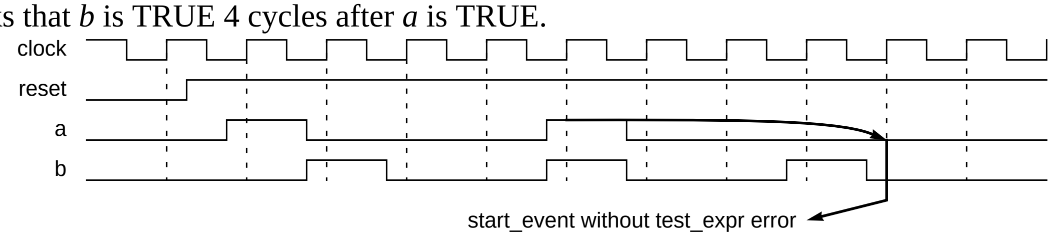
ovl_change
ovl_frame
ovl_time
ovl_unchange

Examples

Example 1

```
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
```

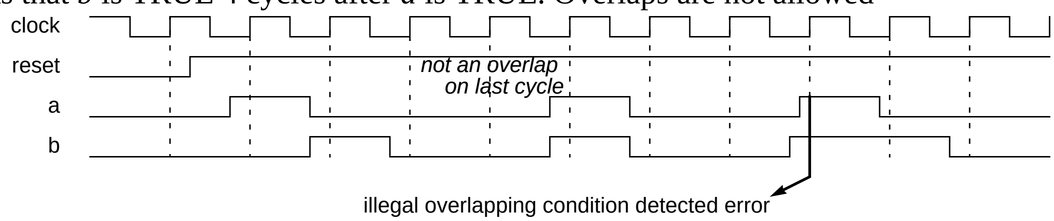
Checks that *b* is TRUE 4 cycles after *a* is TRUE.



Example 2

```
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
```

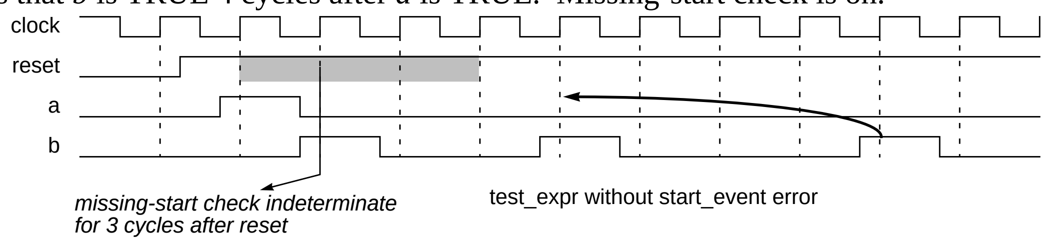
Checks that *b* is TRUE 4 cycles after *a* is TRUE. Overlaps are not allowed



Example 3

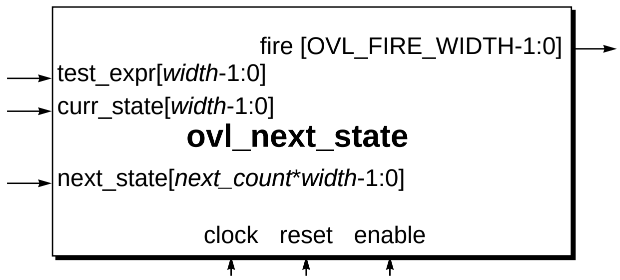
```
ovl_next #(
   ‘OVL_ERROR,                                    // severity_level
   4,                                             // num_cks
   1,                                             // check_overlapping (off)
   1,                                             // check_missing_start (on)
   ‘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
```

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



ovl_next_state

Checks that an expression transitions only to specified values.



Parameters/Generics:
severity_level
width
next_count
min_hold
max_hold
disallow
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: event-bounded assertion

Syntax

```
ovl_next_state
     [#(severity_level, next_count, width, 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/Generics

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_COVER_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

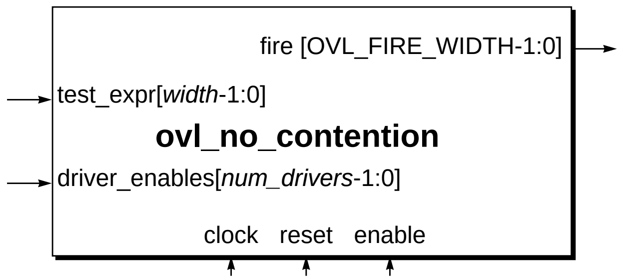
`cover_next_state_transitions`	SANITY — Number of times *test_expr* matched *curr_state* and then transitioned correctly to a value in *next_state* (*disallow*=0) or not in *next_state* (*disallow*=1).
`cover_all_transitions`	CORNER — Non-zero if *test_expr* transitioned to every next value found in 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.

Cover Groups

`next_state_corner`	Whether or not the specified corner case occurred. Bin is: • *all_transitions_covered* — The *test_expr* has transitioned to every next value found in the sampled *next_state*. Not meaningful if *disallow* is 1.
`next_state_statistic`	Coverage statistics. Bins are: • *number_of_transitions_covered* — number of transitions made. • *cycles_checked* — number of cycles *test_expr* and *curr_state* matched.

ovl_no_contention

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



Parameters/Generics:
severity_level
width
num_drivers
min_quiet
max_quiet
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: event-bounded assertion

Syntax

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

Parameters/Generics

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_COVER_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. A quiet time consists of 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`* `≤` *`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`* `≤` *`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.

Cover Groups

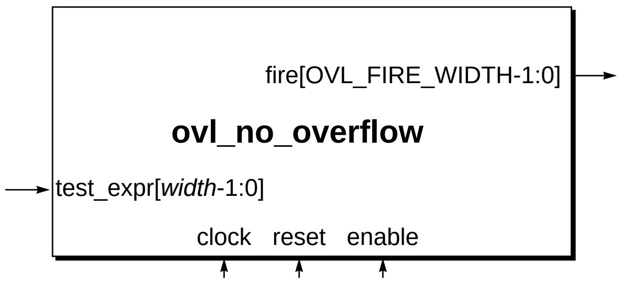
observed_quiet_cycles

Number of times the bus (*test_expr*) was quiet (driver_enables = 0) for the specified number of quiet cycles. Bins are:

- *observed_quiet_cycles_good*[*min_quiet*+1:*maximum*] — bin index is the observed quiet time in clock cycles. The value of *maximum* is:
  - 0 (if *min_quiet* = *max_quiet* = 0),
  - *min_quiet* + 4095 (if *min_quiet* > *max_quiet* = 0), or
  - *max_quiet* (if *max_quiet* > 0).
- *observed_hold_time_bad* — default.

ovl_no_overflow

Checks that the value of an expression does not overflow.



Parameters/Generics:
severity_level
width
min
max
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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*.

Cover Groups

none

Errors

The parameters/generics *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 edge of *clock* after *reset* deasserts.

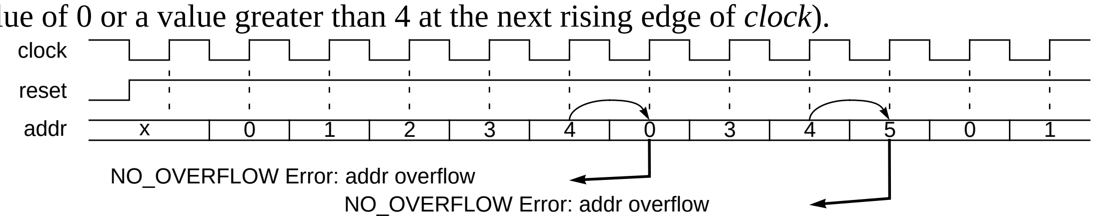
See also

ovl_delta
ovl_fifo_index
ovl_increment
ovl_no_overflow

Examples

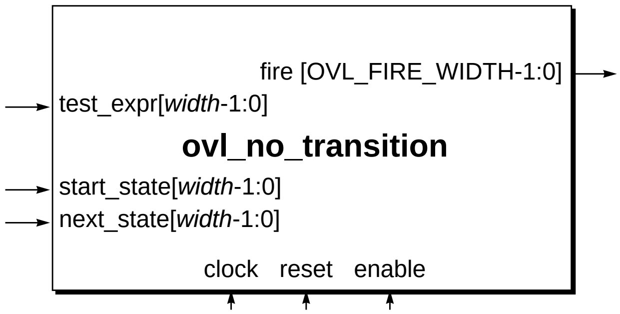
```
ovl_no_overflow #(
    ‘OVL_ERROR,                        // severity_level
    3,                                 // width
    0,                                 // min
    4,                                 // max
    ‘OVL_ASSERT,                       // property_type
    “Error: addr overflow”,            // 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
```

Checks 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

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



Parameters/Generics:

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/Generics

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_COVER_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.

Cover Groups

none

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 edge of *clock* after *reset* deasserts.

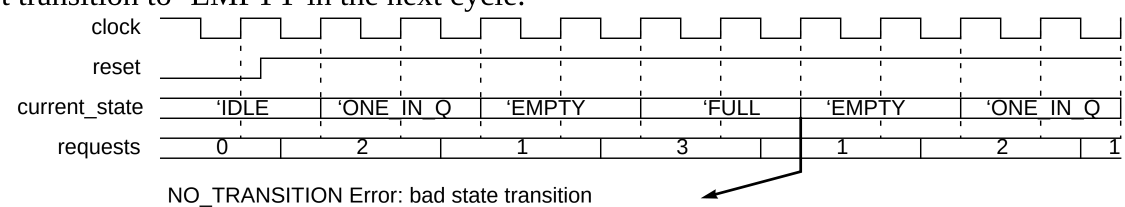
See also

ovl_transition

Examples

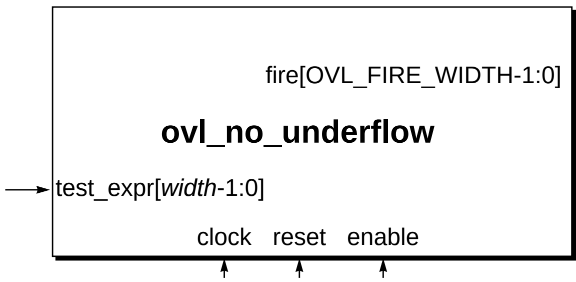
```
ovl_no_transition #(
    `OVL_ERROR,                             // severity_level
    3,                                      // width
    `OVL_ASSERT,                            // property_type
    "Error: bad state transition",          // msg
    `OVL_COVER_DEFAULT,                     // coverage_level
    `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
```

Checks 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.



ovl_no_underflow

Checks that the value of an expression does not underflow.



Parameters/Generics:

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

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/Generics

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_COVER_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*.

Cover Groups

none

Errors

The parameters/generics *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 edge of *clock* after *reset* deasserts.

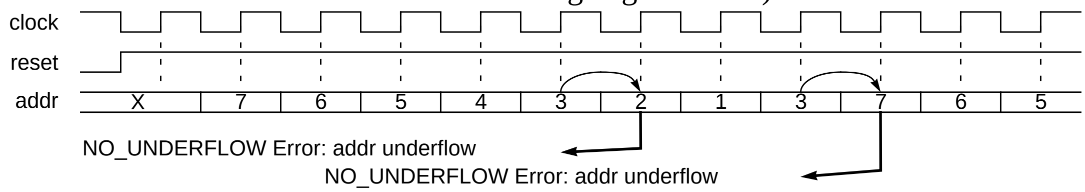
See also

ovl_delta
ovl_decrement
ovl_fifo_index
ovl_no_overflow

Examples

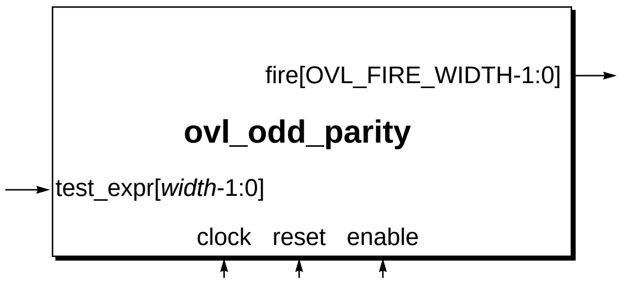
```
ovl_no_underflow #(
   `OVL_ERROR,                              // severity_level
   3,                                       // width
   3,                                       // min
   7,                                       // max
   `OVL_ASSERT,                             // property_type
   "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
```

Checks 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

Checks that the value of an expression has odd parity.



Parameters/Generics:

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/Generics

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_COVER_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 active 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.

Cover Groups

none

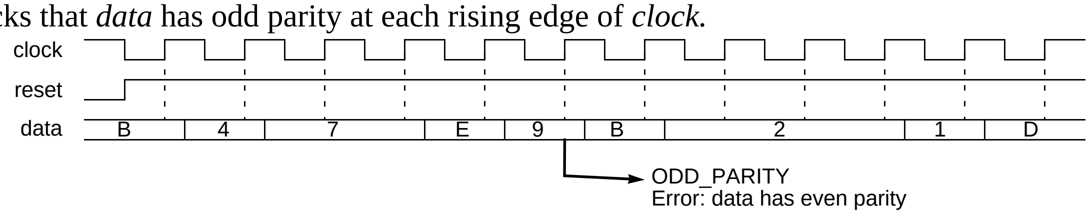
See also

ovl_even_parity

Examples

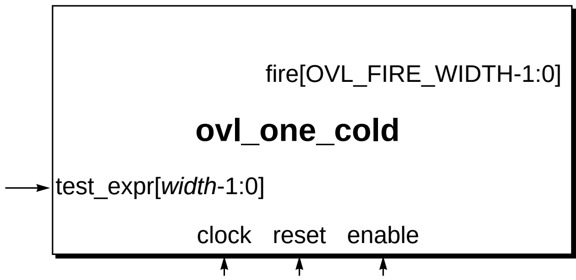
```
ovl_odd_parity #(
    `OVL_ERROR,                              // severity_level
    8,                                       // width
    `OVL_ASSERT,                             // property_type
    “Error: data has even parity”,           // msg
    `OVL_COVER_DEFAULT,                      // 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
```

Checks that *data* has odd parity at each rising edge of *clock*.



ovl_one_cold

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



Parameters/Generics:
severity_level
width
inactive
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

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/Generics

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_COVER_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 active 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.

Cover Groups

none

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

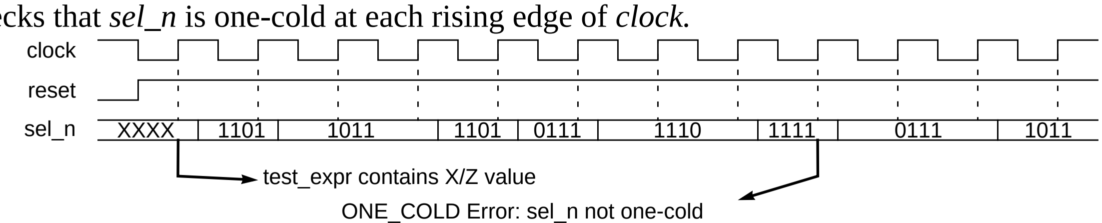
ovl_one_hot   ovl_zero_one_hot

Examples

Example 1

```
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
```

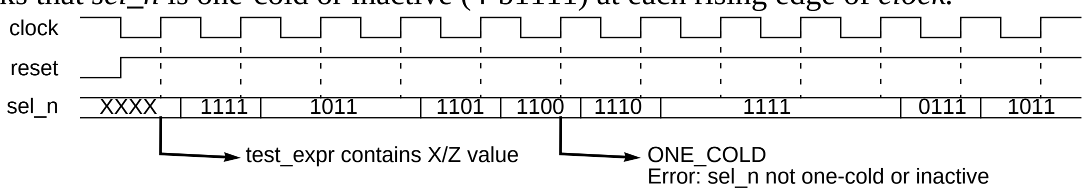
Checks that *sel_n* is one-cold at each rising edge of *clock*.



Example 2

```
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
```

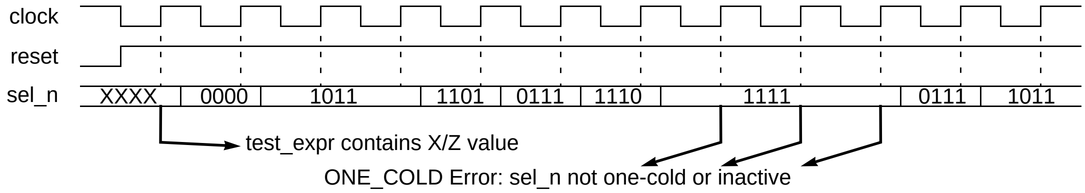
Checks that *sel_n* is one-cold or inactive (4’b1111) at each rising edge of *clock*.



Example 3

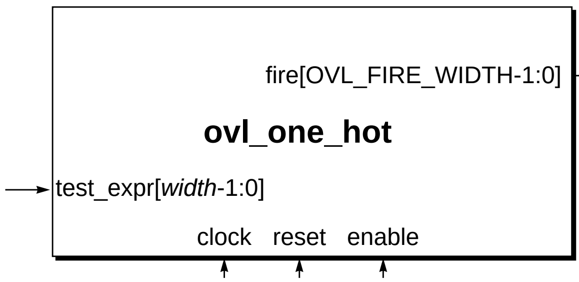
```
ovl_one_cold #(
    `OVL_ERROR,                                    // severity_level
    4,                                             // width
    `OVL_ALL_ZEROS,                                // inactive
    `OVL_ASSERT,                                   // property_type
    "Error: sel_n not 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
```

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



ovl_one_hot

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



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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 active 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.

Cover Groups

none

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.

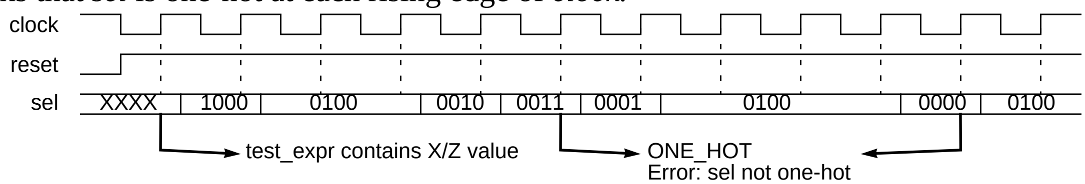
See also

ovl_one_cold   ovl_zero_one_hot

Examples

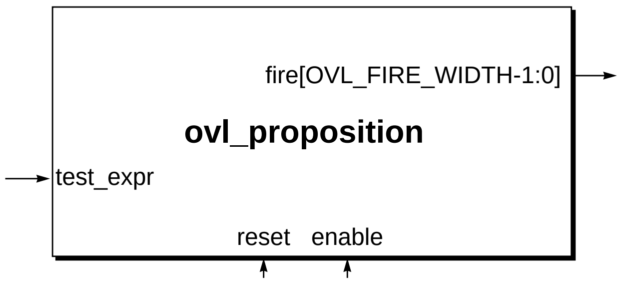
```
ovl_one_hot #(
   `OVL_ERROR,                                  // severity_level
   4,                                           // width
   `OVL_ASSERT,                                 // property_type
   "Error: sel not one-hot",                    // msg
   `OVL_COVER_DEFAULT,                          // 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
```

Checks that *sel* is one-hot at each rising edge of *clock*.



ovl_proposition

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



Parameters/Generics: *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/Generics

severity_level	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
property_type	Property type. Cannot be OVL_ASSUME for SVA and PSL implementations. 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_COVER_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

none

Cover Groups

none

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.
2. The Verilog-95 version of this asynchronous checker handles ‘OVL_ASSERT, ‘OVL_ASSUME and ‘OVL_IGNORE. The SVA and PSL versions of this checker do not implement *property_type* ‘OVL_ASSUME. The SVA version uses immediate assertions and in IEEE 1800-2005 SystemVerilog immediate assertions cannot be assumptions. Assume is only available in a concurrent (clocked) form of an assertion statement. The SVA version treats ‘OVL_ASSUME as an ‘OVL_ASSERT. The PSL version generates an error if *property_type* is ‘OVL_ASSUME.

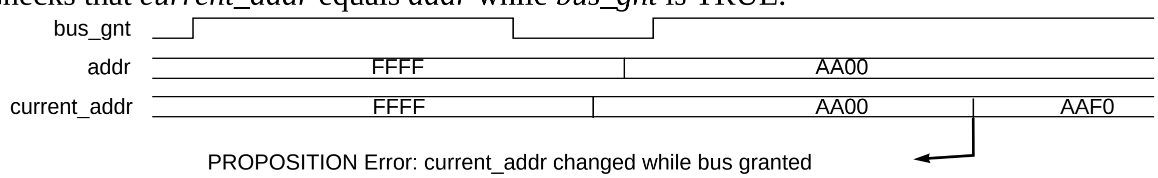
See also

ovl_always
ovl_always_on_edge
ovl_implication
ovl_never

Examples

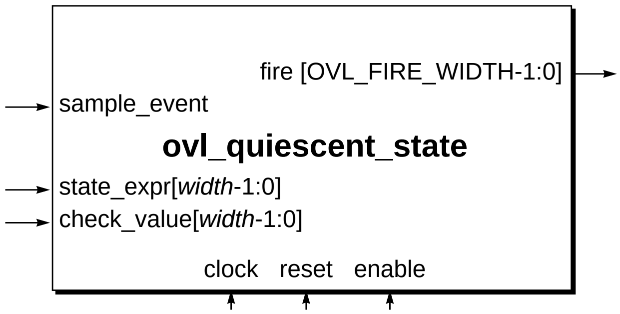
```
ovl_proposition #(
    `OVL_ERROR,                                        // severity_level
    `OVL_ASSERT,                                       // property_type
    "Error: current_addr changed while bus             // msg
    granted",                                          // coverage_level
    `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
```

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



ovl_quiescent_state

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



Parameters/Generics:

severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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

none

Cover Groups

none

Notes

1. The assertion check compares the current value of *sample_event* with its previous value. Therefore, checking does not start until the second rising edge of *clock* after *reset* deasserts.
2. 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.

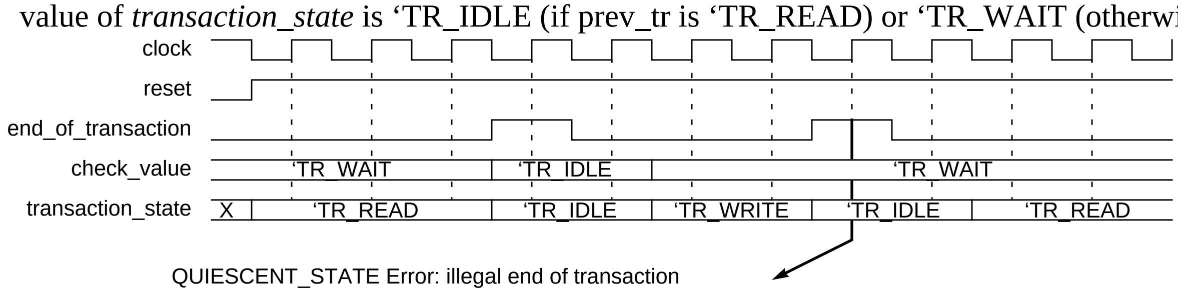
See also

ovl_no_transition ovl_transition

Examples

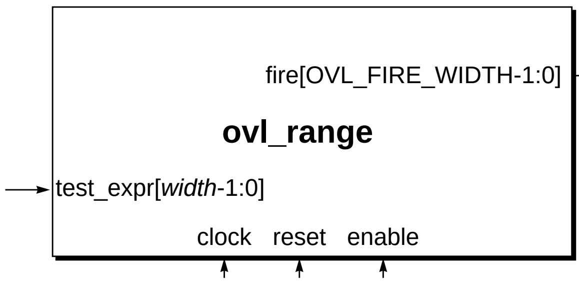
```
ovl_quiescent_state #(
   `OVL_ERROR,                                      // severity_level
   4,                                               // width
   `OVL_ASSERT,                                     // property_type
   "Error: illegal end of transaction",             // msg
   `OVL_COVER_DEFAULT,                              // 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
```

Checks 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).



ovl_range

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



Parameters/Generics:

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

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/Generics

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_COVER_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 active 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_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*.

Cover Groups

none

Errors

The parameters/generics *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.

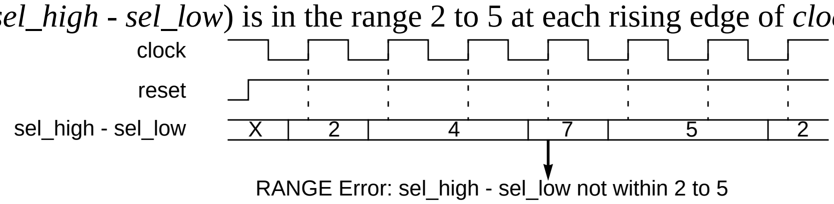
See also

ovl_always
ovl_implication
ovl_never
ovl_proposition

Examples

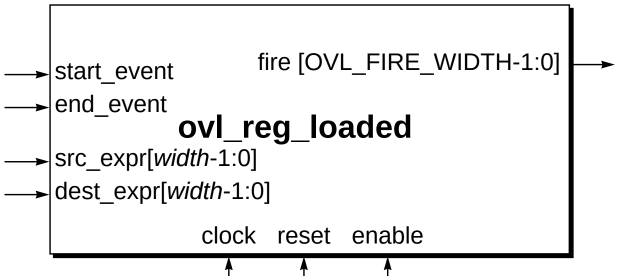
```
ovl_range #(
   `OVL_ERROR,                                          // severity_level
   3,                                                   // width
   2,                                                   // min
   5,                                                   // max
   `OVL_ASSERT,                                         // 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
```

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



ovl_reg_loaded

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



Parameters/Generics: *severity_level*, *width*, *start_count*, *end_count*, *property_type*, *msg*, *coverage_level*, *clock_edge*, *reset_polarity*, *gating_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/Generics

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_COVER_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`	`Test expression 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).

	`Test 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	Test 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_event_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_event_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_end_count`	CORNER — Number of times *dest_expr* loaded the *src_expr* value exactly *end_count* cycles after *start_event* asserted. Not meaningful if *end_count* = 0.

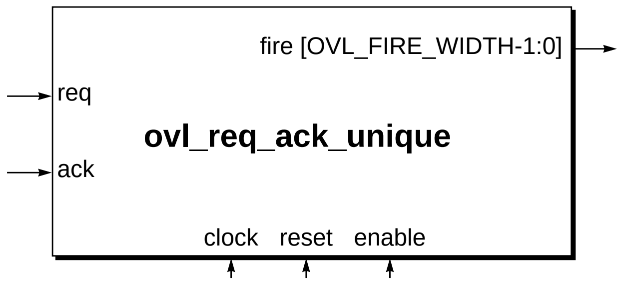
`cover_load_times`	STATISTIC — Reports the load times (in cycles from asserting *start_event* to loading *src_expr* into *dest_expr*) that occurred at least once.

Cover Groups

observed_dest_expr_reg_load_time	Number of times *dest_expr* was loaded in the specified number of cycles. Bins are:
	• *observed_load_time_good*[*start_count*+1:*maximum*] — bin index is the observed load time in clock cycles. The value of *maximum* is:
	  • *start_count* + 4095 (if *end_count* = 0) or
	  • *end_count* (if *end_count* > 0).
	• *observed_load_time_bad* — default.

ovl_req_ack_unique

Checks that every request receives a corresponding acknowledge in a specified time window.



Parameters/Generics:
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/Generics

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_COVER_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

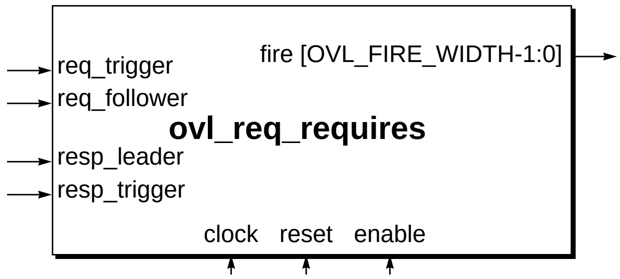
`cover_requests`	SANITY — Number of cycles *req* asserted.
`cover_acknowledgements`	SANITY — Number of cycles *ack* asserted.
`cover_ack_at_min_cks`	CORNER — Number of times acknowledge was received *min_cks* cycles after its request was issued. Not meaningful if *method* = 1.
`cover_ack_at_max_cks`	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.

Cover Groups

observed_latency	Number of acknowledgements with the specified req-to-ack latency. Bins are: • *observed_latency_good*[*min_cks*:*max_cks*] — bin index is the observed latency in clock cycles. • *observed_latency_bad* — default.
observed_outstanding_ requests	Number of cycles with the specified number of outstanding requests. Bins are: • *observed_outstanding_requests*[1:*max_cks*] — bin index is the number of outstanding requests.

ovl_req_requires

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



Parameters/Generics:
severity_level
min_cks
max_cks
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

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/Generics

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 Š *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_COVER_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* in the same or subsequent cycle as the rising edge of *req_trigger*.
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:

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

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.

Cover Groups

observed_latency_btw_req_trigger_and_resp_trigger

Number of requests with the specified request-trigger to response-trigger latency. Bins are:

- *observed_req_trigger_resp_trigger_latency_good* [*min_cks*:*maximum*] — bin index is the observed latency in clock cycles from the request trigger to the response trigger. The value of *maximum* is:
  - 4095 (if *max_cks* = 0) or
  - *max_cks* (if *max_cks* > 0).
- *observed_req_trigger_resp_trigger_latency_bad* — default.

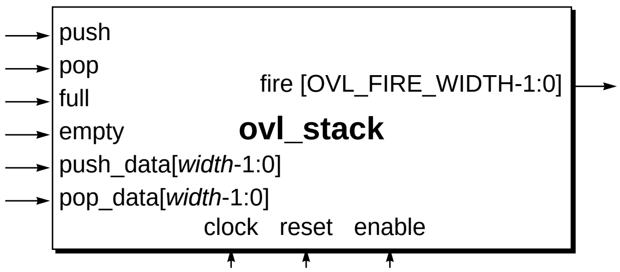
observed_latency_btw_ req_trigger_and_ resp_follower	Number of requests with the specified request-trigger to response-follower latency. Bins are:
	• *observed_req_trigger_resp_follower_latency_good* [0:*maximum*] — bin index is the observed latency in clock cycles from the request trigger to the response follower. The value of *maximum* is:
	  • 4095 (if *max_cks* = 0) or
	  • *max_cks* (if *max_cks* > 0).
	• *observed_req_trigger_resp_follower_latency_bad* — default.

observed_latency_btw_ req_follower_and_ resp_leader	Number of requests with the specified request-follower to response-leader latency. Bins are:
	• *observed_req_follower_resp_leader_latency_good* [1:*maximum*] — bin index is the observed latency in clock cycles from the request follower to the response leader. The value of *maximum* is:
	  • 4095 (if *max_cks* = 0) or
	  • *max_cks* (if *max_cks* > 0).
	• *observed_req_follower_resp_leader_latency_bad* — default.

observed_latency_btw_ resp_leader_and_ resp_trigger	Number of requests with the specified response-leader to response-trigger latency. Bins are:
	• *observed_resp_leader_resp_trigger_latency_good* [0:*maximum*] — bin index is the observed latency in clock cycles from the response leader to the response trigger. The value of *maximum* is:
	  • 4095 (if *max_cks* = 0) or
	  • *max_cks* (if *max_cks* > 0).
	• *observed_resp_leader_resp_trigger_latency_bad* — default.

ovl_stack

Checks the data integrity of a stack and checks that the stack does not overflow or underflow.



Parameters/Generics:

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

Parameters/Generics

`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_COVER_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.
push_data[*width*-1:0]	Push data input to the stack. Contains the data item to push onto the stack.
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.
pop_data[*width*-1:0]	Pop data output from the stack. Contains the data item popped from the stack.

`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.

`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.

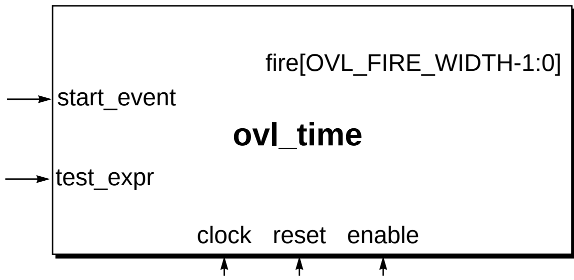
`cover_max_entries`	BASIC — Number of cycles for which the number of data items in the stack was the same as the maximum number of data items the stack had held up to and including that cycle.
`cover_push_then_pop`	BASIC — Number of times a *push* was followed by a *pop* without an intervening *push* (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.

Cover Groups

none

ovl_time

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



Parameters/Generics:
severity_level
num_cks
action_on_new_start
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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 subsequent active edge of *clock* for *num_cks* cycles to verify that the value of *test_expr* is TRUE. During that time, the assertion fails the first 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 (no violation occurs even if *test_expr* has changed to FALSE) and initiates a new check starting in the next cycle.

- 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*.

Cover Groups

none

See also

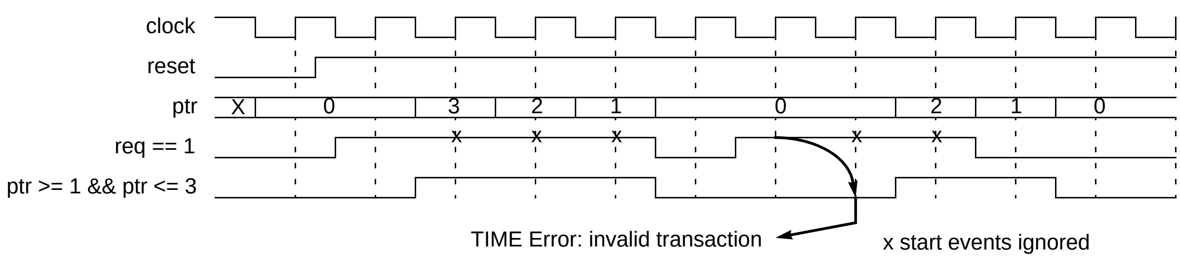
ovl_change
ovl_next
ovl_frame
ovl_unchange
ovl_win_change
ovl_win_unchange
ovl_window

Examples

Example 1

```
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
```

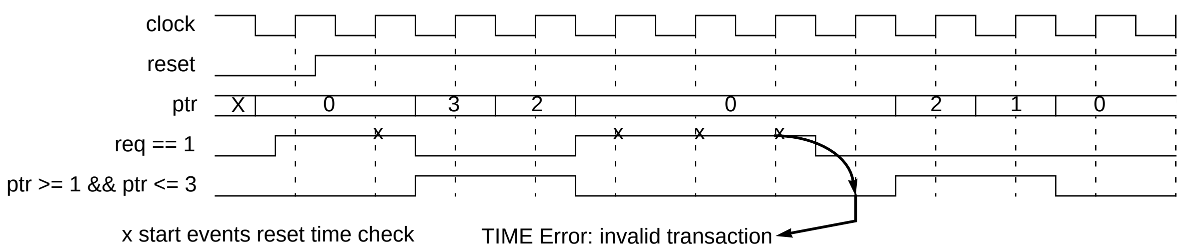
Checks that *ptr* is sampled in the range 1 to 3 for three cycles after *req* is sampled equal to 1 at the rising edge of *clock*. If *req* is sampled equal to 1 when the checker samples *ptr*, a new check is not initiated (i.e., the new start is ignored).



Example 2

```
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
```

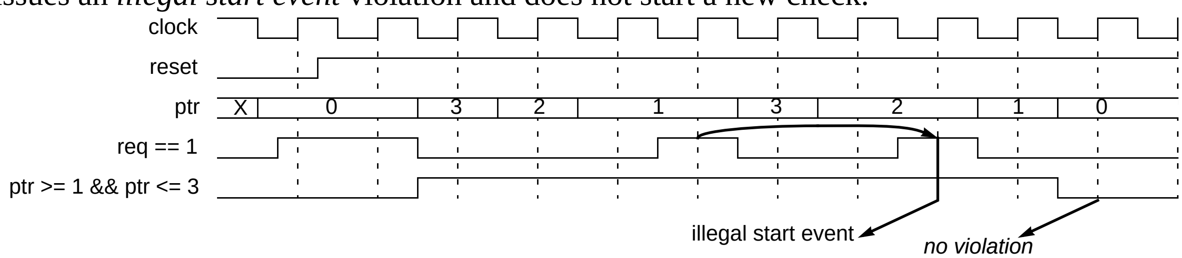
Checks that *ptr* is sampled in the range 1 to 3 for three cycles after *req* is sampled equal to 1 at the rising edge of *clock*. If *req* is sampled equal to 1 when the checker samples *ptr*, a new check is initiated (i.e., the new start restarts a check).



Example 3

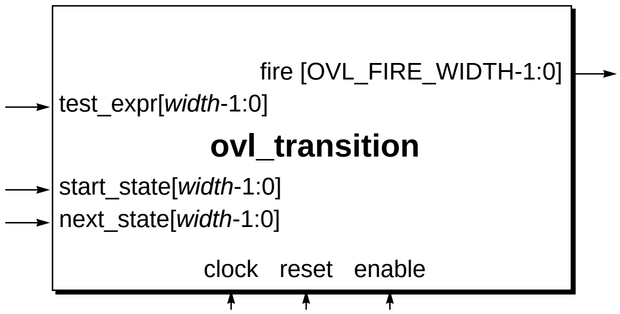
```
ovl_time #(
   'OVL_ERROR,                                 // severity_level
   3,                                          // num_cks
   'OVL_ERROR_ON_NEW_START,                    // action_on_new_start
   'OVL_ASSERT,                                // property_type
   "Error: invalid transaction",               // 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
```

Checks 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

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



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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,
        next_state, fire);
```

Parameters/Generics

`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_COVER_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.

Cover Groups

none

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 edge of *clock* after *reset* deasserts.

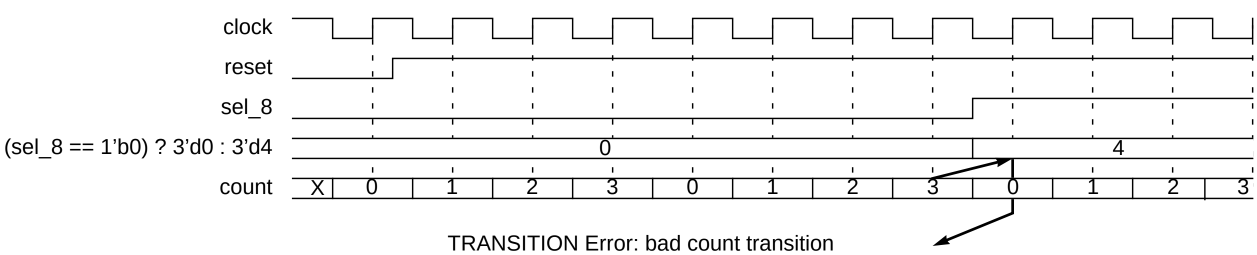
See also

ovl_no_transition

Examples

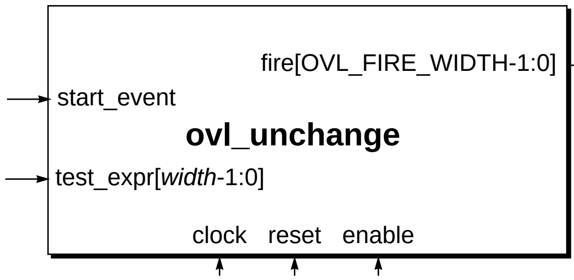
```
ovl_transition #(
   ‘OVL_ERROR,                                  // severity_level
   3,                                           // width
   ‘OVL_ASSERT,                                 // property_type
   “Error: bad count transition”,               // msg
   ‘OVL_COVER_DEFAULT,                          // 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
```

Checks 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.



ovl_unchange

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



Parameters/Generics:

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/Generics

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_COVER_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 pending check (no violation occurs even if *test_expr* has changed in the current cycle) and initiates a new check with the current value of *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 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.

Cover Groups

none

See also

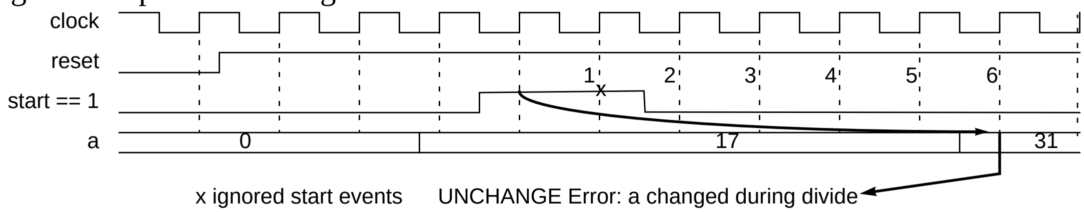
ovl_change
ovl_time
ovl_win_change
ovl_win_unchange
ovl_window

Examples

Example 1

```
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_valid_div_unchange_a );              // fire
```

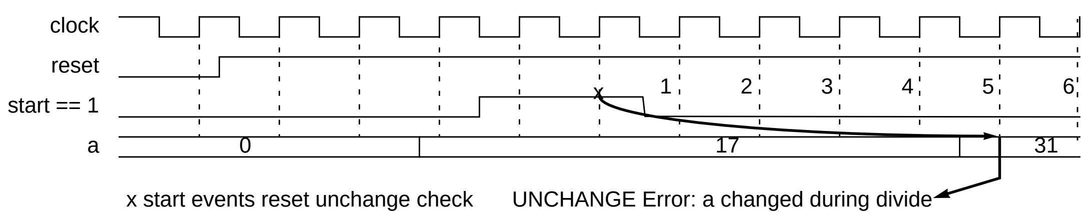
Checks that *a* remains unchanged while a divide operation is performed (8 cycles). Restarts during divide operations are ignored.



Example 2

```
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
```

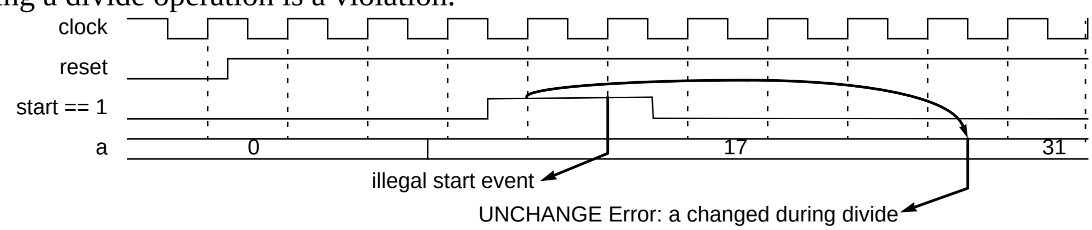
Checks that *a* remains unchanged while a divide operation is performed (8 cycles). A restart during a divide operation starts the check over.



Example 3

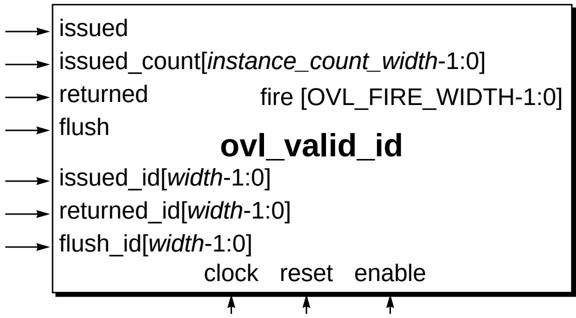
```
ovl_unchange #(
  `OVL_ERROR,                                  // severity_level
  8,                                           // width
  8,                                           // num_cks
  `OVL_ERROR_ON_NEW_START,                     // action_on_new_start
  `OVL_ASSERT,                                 // 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
```

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



ovl_valid_id

Checks 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.



Parameters/Generics:

severity_level
width
min_cks
max_cks
max_id_instances
max_ids
max_instances_per_id
instance_count_width
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

Syntax

```
ovl_valid_id
      [#(severity_level, min_cks, max_cks, width, max_id_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_id, returned,
      returned_id, flush, flush_id, issued_count, fire);
```

Parameters/Generics

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_id_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_COVER_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_id[*width*-1:0]	Expression or variable containing the ID to add to the outstanding IDs list if *issued* is TRUE.
returned	Returned ID signal indicating an instance of the ID in *returned_id* is removed from the outstanding IDs list.
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	Flush ID signal indicating all instances of the ID in *flush_id* are removed from the outstanding IDs list.
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.
issued_count [*instance_count_width*-1:0]	Number of instances of the issued ID to make outstanding when *issued* asserts.
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 Š 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_id_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_id_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.

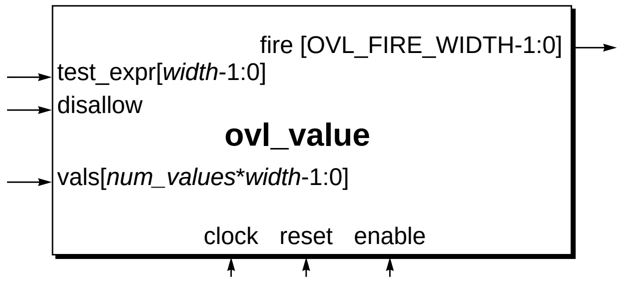
`turnaround_times`	BASIC — Reports the turnaround times (i.e., number of cycles after an ID instance is issued that the instance is returned) that occurred at least once.
`outstanding_ids`	BASIC — Reports the numbers of outstanding ID instances that occurred at least once.
`cover_returned_at_min_cks`	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_id_instances* limit.
`cover_max_instances_per_id`	CORNER — Number of cycles the outstanding instances of an ID reached the *max_instances_per_id* limit.

Cover Groups

observed_latency	Number of returned IDs with the specified turnaround time. Bins are: • *observed_latency_good*[*min_cks*:*max_cks*] — bin index is the observed turnaround time in clock cycles. • *observed_latency_bad* — default.
outstanding_ids	Number of cycles with the specified number of outstanding ids. Bins are: • *observed_outstanding_ids*[0:*max_id_instances*] — bin index is the instance ID.

ovl_value

Checks 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).



Parameters/Generics:
severity_level
num_values
width
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion

Syntax

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

Parameters/Generics

severity_level	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
num_values	Number of values in *vals*. Must be ≥ 1. Default: 1.
width	Width of *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_COVER_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

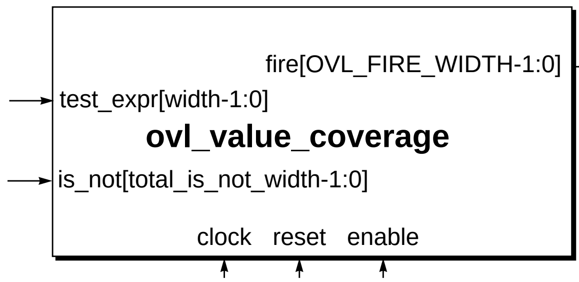
`cover_values_checked`	SANITY — Number of cycles *test_expr* loaded a new value.
`cover_in_vals`	BASIC — Number of cycles *disallow* was FALSE and the value of *test_expr* matched a value in *vals*.
`cover_not_in_vals`	BASIC — Number of cycles *disallow* was TRUE and the value of *test_expr* did not 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.

Cover Groups

none

ovl_value_coverage

Ensures that values of a specified expression are covered during simulation.



Parameters/Generics:
severity_level
width
is_not_width
is_not_count
value_coverage
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 2-cycle assertion

total_is_not_width = (*is_not_count***is_not_width*) ? *is_not_count***is_not_width* : 1

Syntax

```
ovl_value_coverage
      [#(severity_level, width, is_not_width, is_not_count,
         value_coverage, property_type, msg, coverage_level, clock_edge,
         reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, is_not, fire);
```

Parameters/Generics

`severity_level`	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
`width`	Width of *test_expr*. Default: 1.
`is_not_width`	Maximum width of an *is_not* value. Default: 1.
`is_not_count`	Number of *is_not* values. Default: 0.
`value_coverage`	Whether or not to perform value_coverage checks. `value_coverage = 0` (Default) Turns off the value_coverage check. `value_coverage = 1` Turns on the value_coverage 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_COVER_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 checker. The checker samples on the rising edge of the clock.
reset	Synchronous reset signal indicating completed initialization.
enable	Expression that indicates whether or not to check the inputs.
test_expr[*width-1:0*]	Variable or expression to check.
is_not [*total_is_not_width* - 1:0]	Concatenated list of *is_not_count* variables containing 'is-not' values for *test_expr*. The variables' values are latched at reset and are then used as values of *test_expr* to exclude from cover point data. If *is_not* = 1'b0 and both *is_not_width* and *is_not_count* are undefined, then is-not values are not used. The *test_expr* variable is covered when all possible values have been covered.
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_coverage* checker ensures the value of *test_expr* does not change when the checker is active. The checker checks the multiple-bit expression *test_expr* at each rising edge of *clock* whenever *enable* is TRUE. If *test_expr* has changed value, the assertion fails and *msg* is printed. This checker is used to determine coverage of *test_expr* and to gather coverpoint data. As such, the sense of the assertion is reversed. Unlike most other OVL checkers (which verify assertions that are not expected to fail), ovl_coverage checkers' assertion is intended to fail, therefore the value_coverage check typically is turned off (*value_coverage* = 0).

Assertion Checks

VALUE_COVERAGE	The value of the variable was covered. *property_type* = 'OVL_ASSERT The value of *test_expr* should not change. This check occurs at every active clock edge and fires if the value of *test_expr* has changed from the value at the previous active clock edge.

Implicit X/Z Checks

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

Cover Points

`cover_values_checked`	SANITY — Number of cycles *test_expr* changed value.
`cover_computations_checked`	STATISTIC — Number of times the cover value was checked.
`cover_values_covered`	STATISTIC — Number of values (including is-not values) that *test_expr* has covered
`cover_values_uncovered`	STATISTIC — Number of values (except is-not values) that *test_expr* has not covered.
`cover_all_values_covered`	CORNER — Non-zero if all values of *test_expr* (except is_not values) have been covered. Otherwise it is set to 0.

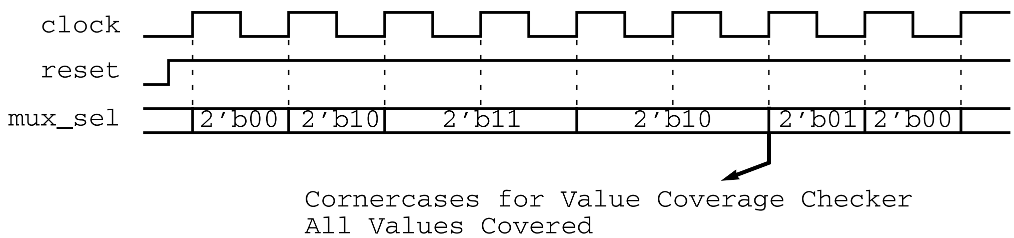
See also

ovl_coverage

Examples

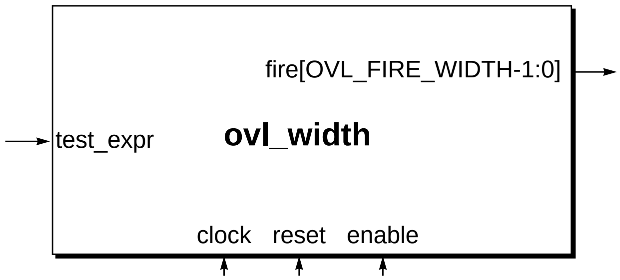
```
ovl_value_coverage #(
   .severity_level(`OVL_ERROR),
   .width(2),
   .property_type(`OVL_ASSERT),
   .coverage_level(`OVL_COVER_ALL))
   ovl_coverage_mux_select(
      .clock(clock),
      .reset(reset),
      .enable(1'b1),
      .test_expr(mux_sel),
      .is_not(1'b0),
.fire(fire));
```

All Values Covered corner case asserts when *mux_sel* has covered all encodings. *Is_not_count* by default is 0; *is_not_width* by default is 1 and the *is_not* port is tied to 1'b0, so no is-not values are included.



ovl_width

Checks 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.



Parameters/Generics:
severity_level
min_cks
max_cks
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

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_COVER_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).

Cover Groups

none

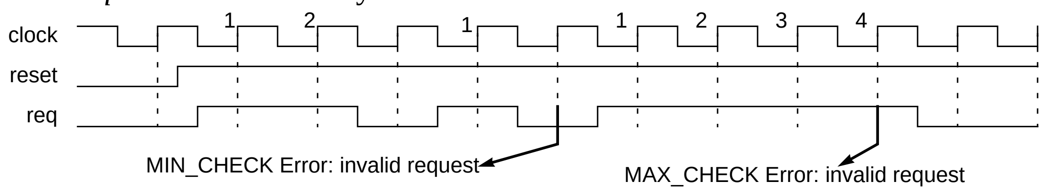
See also

ovl_change
ovl_time
ovl_unchange

Examples

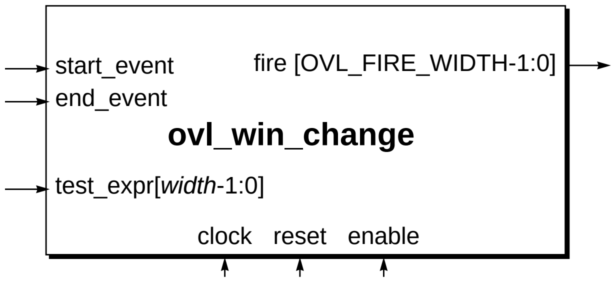
```
ovl_width #(
    `OVL_ERROR,                                // severity_level
    2,                                         // min_cks
    3,                                         // max_cks
    `OVL_ASSERT,                               // property_type
    "Error: invalid request",                  // 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
```

Checks that *req* asserts for 2 or 3 cycles.



ovl_win_change

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



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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/Generics

`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_COVER_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).

Cover Groups

none

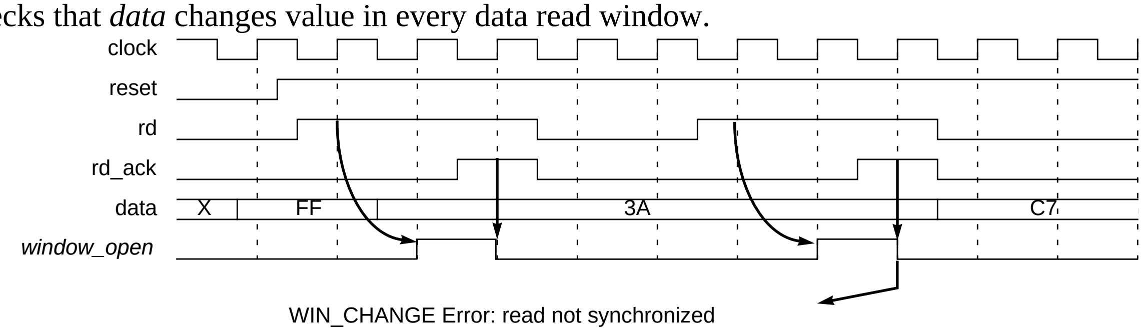
See also

ovl_change
ovl_time
ovl_unchange
ovl_win_unchange
ovl_window

Examples

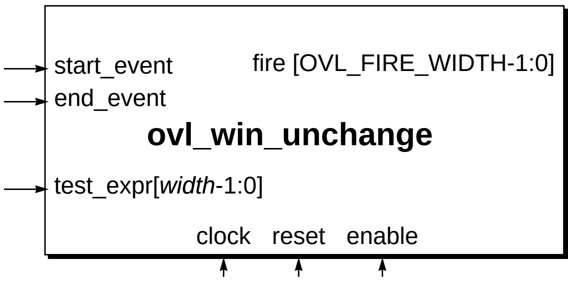
```
ovl_win_change #(

   `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
```

Checks that *data* changes value in every data read window.



ovl_win_unchange

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



Parameters/Generics:

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/Generics

`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_COVER_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 (after reporting a violation if *test_expr* has changed) 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).

Cover Groups

none

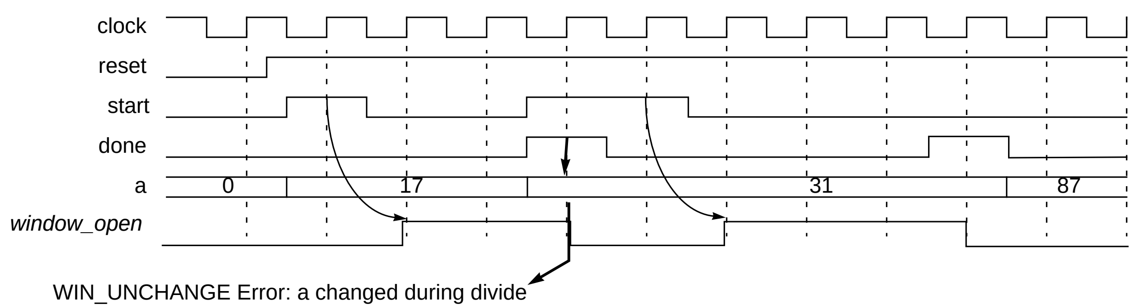
See also

ovl_change
ovl_time
ovl_unchange
ovl_win_change
ovl_window

Examples

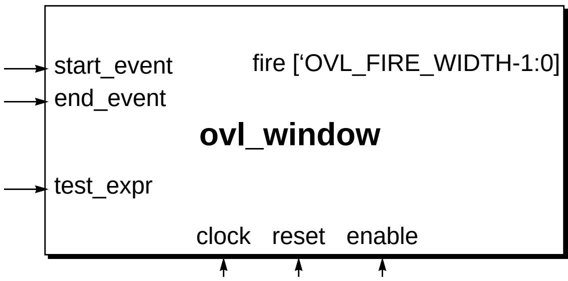
```
ovl_win_unchange #(
   `OVL_ERROR,                              // severity_level
   8,                                       // width
   `OVL_ASSERT,                             // property_type
   "Error: a changed during divide",        // msg
   `OVL_COVER_DEFAULT,                      // coverage_level
   `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
```

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



ovl_window

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



Parameters/Generics:

severity_level
property_type
msg
coverage_level
coverage_level
clock_edge
reset_polarity
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/Generics

`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_COVER_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.

Cover Groups

none

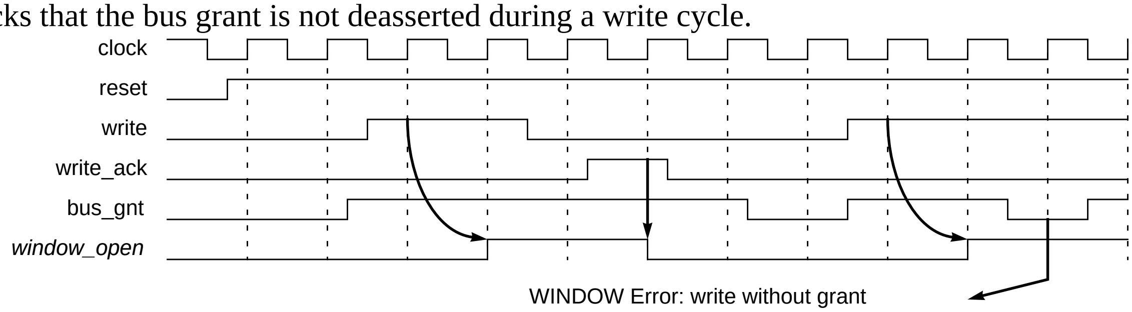
See also

ovl_change
ovl_time
ovl_unchange
ovl_win_change
ovl_win_unchange

Examples

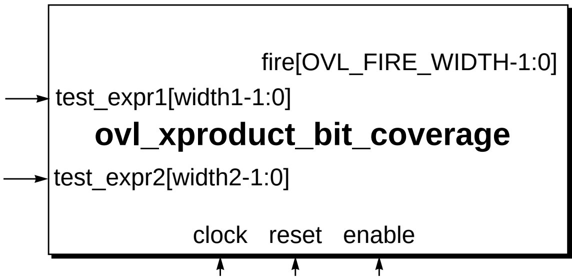
```
ovl_window #(
   `OVL_ERROR,                                   // severity_level
   `OVL_ASSERT,                                  // property_type
   "Error: write without grant",                 // msg
   `OVL_COVER_DEFAULT,                           // coverage_level
   `OVL_POSEDGE,                                 // 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
```

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



ovl_xproduct_bit_coverage

Ensures functional cross product bit coverage of two vectors.



Parameters/Generics: severity_level, width1, width2, test_expr2_enable, coverage_check, *property_type*, *msg*, *coverage_level*, *clock_edge*, *reset_polarity*, *gating_type*

Class: event-bounded assertion

Syntax

```
ovl_xproduct_bit_coverage
      [#(severity_level, width1, width2, test_expr2_enable,
         coverage_check, property_type, msg, coverage_level, clock_edge,
         reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr1, test_expr2, fire);
```

Parameters/Generics

severity_level	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
width1	Width of the *test_expr1*. Default: 1.
width2	Width of the *test_expr2*. Default: 1.
test_expr2_enable	Whether or not to use *test_expr2* as the second vector. `test_expr2_enable = 0` (Default) Use *test_expr1* as the second vector (`test_expr2` is ignored). `test_expr2_enable = 1` Use *test_expr2* as the second vector.
coverage_check	Whether or not to perform coverage checks. `coverage_check = 0` (Default) Turns off the coverage check. `coverage_check = 1` Turns on the coverage 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_COVER_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 checker. The checker samples on the rising edge of the clock.
reset	Synchronous reset signal indicating completed initialization.
enable	Expression that indicates whether or not to check the inputs.
test_expr1[*width1*-1:0]	First vector, specified as a signal, vector or concatenation of signals.
test_expr2[*width2*-1:0]	Second vector (if *test_expr2_enable* is 1), specified as a signal, vector or concatenation of signals (or 1'b0).
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_xproduct_bit_coverage* checker determines cross-product coverage of the bits of one or two variables and gathers coverpoint data. By default, the checker performs no assertion checks. If *test_expr2_enable* is 1, the checker checks the expressions *test_expr1* and *test_expr2* at each rising edge of *clk* whenever *enable* is TRUE. If *test_expr1*or *test_expr2* has changed value, the checker updates its cross-product coverage matrix based on the values of *test_expr1* and *test_expr2*.

The checker's cross-product coverage matrix is a bit matrix whose rows correspond to the descending bits of *test_expr1* and whose columns correspond to the descending bits of *test_expr2*. Elements in the matrix are the corresponding bits of *test_expr1*and *test_expr2* ANDed together. For example, if:

```
test_expr1 is a[9:6]
```

and

```
test_expr2 is b[5:3]
```

then the cross-product coverage matrix is:

`a[9] & b[5]`	`a[9] & b[4]`	`a[9] & b[3]`
`a[8] & b[5]`	`a[8] & b[4]`	`a[8] & b[3]`
`a[7] & b[5]`	`a[7] & b[4]`	`a[7] & b[3]`
`a[6] & b[5]`	`a[6] & b[4]`	`a[6] & b[3]`

At reset, the matrix is initialized to all 0's. Each cycle *test_expr1* or *test_expr2* changes, the checker calculates a temporary matrix for the current values of *test_expr1* and *test_expr2*. Then, the cross-coverage matrix is updated by setting all elements to 1 whose corresponding elements in the temporary matrix are 1. That is, the bits of the cross-product coverage matrix are "sticky": once set to 1, they remain set to 1. The matrix is considered covered when all bits are 1.

To help analyze partial coverage, the Coverage Matrix Bitmap statistic coverpoint is a concatenated list of the bits of the cross-product coverage matrix arranged by rows.

By default, the value of *test_expr2_enable* is 0, which disables the *test_expr2* port. This is the special case where the checker maintains a cross-product coverage matrix for a vector with itself. However, the Coverage Matrix Bitmap value reported is not the same as one for a matrix where *test_expr2* = *test_expr1*. In this special case, diagonal elements are extraneous (for example, a[3]==1 && a[3]==1) and the elements of the lower-half matrix are redundant. So, the matrix reported by the Coverage Matrix Bitmap is formed by removing the diagonal elements and setting all lower-half matrix elements to 1. For example, if:

```
test_expr2_enable is 0
test_expr1 is a[9:6]
test_expr2 is 1'b0
```

then the cross-product coverage matrix reported by Coverage Matrix Bitmap is:

`a[9] & a[8]`	`a[9] & a[7]`	`a[9] & a[6]`
`1`	`a[8] & a[7]`	`a[8] & a[6]`
`1`	`1`	`a[7] & a[6]`

Assertion Checks

`COVERAGE`	`All bits of the coverage matrix were covered.` Every bit of the cross product coverage matrix is 1.

Implicit X/Z Checks

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

Cover Points

`cover_test_expr1_checked`	SANITY — Number of cycles *test_expr1* changed value.
`cover_test_expr2_checked`	SANITY — Number of cycles *test_expr2* changed value if parameter *test_expr2_enable* is set to 1
`cover_value_checked`	STATISTIC — Number of times the cover value was checked.
`cover_matrix_covered`	CORNER — Number of times all bits of the matrix is 1.

Cover Groups

None

See also

ovl_coverage
ovl_value_coverage
ovl_xproduct_value_coverage

Examples

Example 1

```
ovl_xproduct_bit_coverage #(
    .severity_level(‘OVL_ERROR),
.width1(5),
    .property_type(‘OVL_ASSERT),
    .msg(‘OVL_VIOLATION : "),
    .coverage_level(‘OVL_COVER_NONE))
    XPD1 (
       .clock(clock),
       .reset(1’b1),
       .enable(1’b1),
       .test_expr1(a[4:0]),
       .test_expr2(1’b0))
.fire(fire));
```

Maintains the following bit coverage matrix:

a[4] & a[3]	a[4] & a[2]	a[4] & a[1]	a[4] & a[0]
1	a[3] & a[2]	a[3] & a[1]	a[3] & a[0]
1	1	a[2] & a[1]	a[2] & a[0]

1	1	1	a[1] & a[0]

Example 2

```
ovl_xproduct_bit_coverage #(
   .severity_level(‘OVL_ERROR),
   .width1(4),
   .coverage_check(1’b1),
   .property_type(‘OVL_ASSERT),
   .msg(‘OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))

   XPD2 (
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr1({sig3, sig2, sig1, sig0}))
.fire(fire));
```

Maintains the following bit coverage matrix:

sig3 & sig2	sig3 & sig1	sig3 & sig0
1	sig2 & sig1	sig2 & sig0
1	1	sig1 & sig0

Example 3

```
ovl_xproduct_bit_coverage #(
   .severity_level(‘OVL_ERROR),
   .width1(5),
   .width2(4),
   .test_expr2_enable(1),
   .coverage_check(1’b1),
   .property_type(‘OVL_ASSERT),
   .msg(‘OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))

   XPD3 (
      .clock(clock),
      .reset(1’b1),
      .enable(1’b1),
      .test_expr1(a[4:0]),
      .test_expr2(b[3:0]),
.fire(fire));
```

Maintains the following bit coverage matrix:

a[4] & b[3]	a[4] & b[2]	a[4] & b[1]	a[4] & b[0]
a[3] & b[3]	a[3] & b[2]	a[3] & b[1]	a[3] & b[0]
a[2] & b[3]	a[2] & b[2]	a[2] & b[1]	a[2] & b[0]
a[1] & b[3]	a[1] & b[2]	a[1] & b[1]	a[1] & b[0]
a[0] & b[3]	a[0] & b[2]	a[0] & b[1]	a[0] & b[0]

Example 4

```
ovl_xproduct_bit_coverage #(
   .severity_level(‘OVL_ERROR),
   .width1(4),
   .width2(1),
   .test_expr2_enable(1),
   .property_type(‘OVL_ASSERT),
   .msg(‘OVL_VIOLATION : "),
   .coverage_level(‘OVL_COVER_NONE))

   XPD4 (
      .clock(clock),
      .reset(1’b1),
      .active(1’b1),
      .test_expr1(a[3:0]),
      .test_expr2(sig));
```

Maintains the following bit coverage matrix:

a[3] & sig

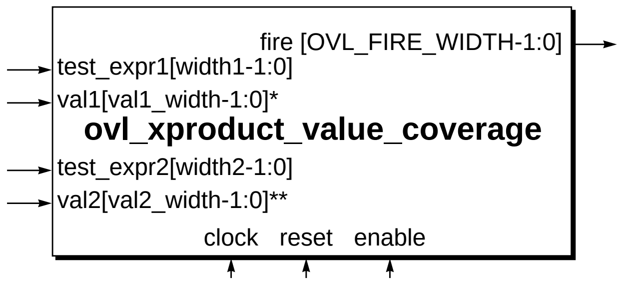
a[2] & sig

a[1] & sig

a[0] & sig

ovl_xproduct_value_coverage

Ensures functional cross product value coverage of two variables.



Parameters/Generics:
severity_level
width1
width2
val1_width
val2_width
val1_count
val2_count
min1
min2
max1
max2
coverage_check
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: event-bounded assertion

```
*val1_width = val1_count > 0 ? val1_count * val1_width : 1
**val2_width = val2_count > 0 ? val2_count * val2_width : 1
```

Syntax

```
ovl_xproduct_value_coverage
    [#(severity_level, width1,width2, val1_width, val2_width,
       val1_count, val2_count, min1, min2, max1, max2, coverage_check,
       property_type, msg, coverage_level, clock_edge, reset_polarity,
       gating_type)]
  instance_name (clock, reset, enable, test_expr1, test_expr2, val1,
    val2);
```

Parameters/Generics

severity_level	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
width1	Width of the *test_expr1*. Default: 1.
width2	Width of the *test_expr2*. Default: 1.
val1_width	Width of each item in *val1*. Default: 1.
val2_width	Width of each item in *val2*. Default: 1.
val1_count	Number of items in *val1*. Default: 0.
val2_count	Number of items in *val2*. Default: 0.
min1	Minimum value of the range of *test_expr1*. Ignored unless *val1_count* = 0. Default : 0
min2	Minimum value of the range of *test_expr2*. Ignored unless *val2_count* = 0. Default : 0

max1	Maximum value of the range of *test_expr1*. Ignored unless *val1_count* = 0. `max1 = 0` (Default) Maximum value is the largest possible value of *test_expr1*. `max1 > 0` Maximum value is *max1*.
max2	Maximum value of the range of *test_expr2*. Ignored unless *val2_count* = 0. `max2 = 0` (Default) Maximum value is the largest possible value of *test_expr2*. `max2 > 0` Maximum value is *max2*.
coverage_check	Whether or not to perform coverage checks. `coverage_check = 0` (Default) Turns off the coverage check. `coverage_check = 1` Turns on the coverage 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_COVER_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 checker. The checker samples on the rising edge of the clock.
reset	Synchronous reset signal indicating completed initialization.
enable	Expression that indicates whether or not to check the inputs.
test_expr1[*width1*-1:0]	First variable or expression.
test_expr2[*width2*-1:0]	Second variable or expression.

`val1[val1_width-1:0]`	`val1_count = 0`
	Connect to 1‘b0.
	`val1_count > 0`
	Concatenated list of *val1_count* elements that define the range of *test_expr1*. Each element is a *val1_width* wide variable or expression.

`val2[val2_width-1:0]`	`val2_count = 0`
	Connect to 1‘b0.
	`val2_count > 0`
	Concatenated list of *val2_count* elements that define the range of *test_expr2*. Each element is a *val2_width* wide variable or expression.

`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_xproduct_value_coverage* checker determines cross-product coverage of the ranges of two variables and gathers coverpoint data. By default, the checker performs no assertion checks. The checker checks the expressions *test_expr1* and *test_expr2* at each rising edge of *clock* whenever *enable* is TRUE. If *test_expr1*or *test_expr2* has changed value, the checker updates its cross-product coverage matrix based on the values of *test_expr1* and *test_expr2*.

The checker’s cross-product coverage matrix is a bit matrix whose rows correspond to the range of values of *test_expr1* and whose columns correspond to the range of values of *test_expr2*. At reset, the matrix is initialized to all 0’s. In a cycle in which both *test_expr1* and *test_expr2* have values in their respective ranges, the matrix element corresponding to that event is set to 1. The bits of the cross-product coverage matrix are “sticky”: once set to 1, they remain set to 1. The matrix is considered covered when all bits are 1. To help analyze partial coverage, the Coverage Matrix Bitmap statistic coverpoint is a concatenated list of the bits of the cross-product coverage matrix arranged by rows.

The ranges of *test_expr1* and *test_expr2* can be specified in two ways: as contiguous value ranges and as discrete value ranges.

Contiguous Value Range

By default, the ranges of *test_expr1* and *test_expr2* are from 0 to their largest possible value. Setting *min1* and *max1* restricts the range of *test_expr1* to *min1*, *min1*+1, ... , *max1*. Similarly, setting *min2* and *max2* restricts the range of *test_expr2* to *min2*, *min2*+1, ... , *max2*. The default value of *min1* and *min2* is 0. The default value of *max1* and *max2* is 0, which sets the top range values to the maximum values of *test_expr1* and *test_expr2*.

For example, if:

```
test_expr1 is a
min1 = 6 and max1 = 9
```

and

```
test_expr2 is b
min2 = 3 and max2 = 5
```

then the cross-product coverage matrix is:

```
(a==9)&&(b==5)  (a==9)&&(b==4)  (a==9)&&(b==3)
(a==8)&&(b==5)  (a==8)&&(b==4)  (a==8)&&(b==3)
(a==7)&&(b==5)  (a==7)&&(b==4)  (a==7)&&(b==3)
(a==6)&&(b==5)  (a==6)&&(b==4)  (a==6)&&(b==3)
```

Discrete Value Range

Setting *val1_count* > 1 enables discrete values for the range of *test_expr1*. The *val1* port contains these values as a concatenated list of *val1_count* values, each value having width *val1_width*. The values of *min1* and *max1* are ignored. Similarly, setting *val2_count* > 1 enables discrete values for the range of *test_expr2*. The *val2* port contains these values as a concatenated list of *val2_count* values, each value having width *val2_width*. The values of *min2* and *max2* are ignored.

For example, if:

```
test_expr1 is a
val1_count = 4, val1_width = 16 and val2 = {1'h9, 1‘hB, 1'hF, 1'h4}
```

and

```
test_expr2 is b
val1_count = 3, val1_width = 12 and val1 = {1'h3, 1'h8, 1'h7}
```

then the cross-product coverage matrix is:

```
(a==4)&&(b==7)  (a==4)&&(b==8)  (a==4)&&(b==3)
(a==F)&&(b==7)  (a==F)&&(b==8)  (a==F)&&(b==3)
(a==B)&&(b==7)  (a==B)&&(b==8)  (a==B)&&(b==3)
(a==9)&&(b==7)  (a==9)&&(b==8)  (a==9)&&(b==3)
```

Discrete value ranges have the following characteristics:

- One test expression can have a contiguous range while the other test expression has a discrete range.
- Discrete ranges can be dynamic. Typically, the *val1* and *val2* ports should remain constant, so the coverage matrix makes sense. However, the checker does not check this restriction. If the value of *val1* or *val2* has changed, a new set of range values are used for the current cycle. The same cross-product coverage matrix is updated, but the updated elements correspond to the new ranges.
- Discrete ranges can have duplicate values. Although this is not a typical usage, if a value with duplicates is covered, all corresponding matrix bits are set.

Assertion Checks

`COVERAGE`	`All bits of the coverage matrix were covered.`
	Every bit of the cross-product coverage matrix is 1.

Implicit X/Z Checks

test_expr1 contains X or Z	Expression contained X or Z bits.
test_expr2 contains X or Z	Expression contained X or Z bits.
val1 contains X or Z	Expression contained X or Z bits.
val2 contains X or Z	Expression contained X or Z bits.

Cover Points

`cover_test_expr1_checked`	SANITY — Number of cycles test_expr1 changed value.
`cover_test_expr2_checked`	SANITY — Number of cycles test_expr2 changed value.
`cover_value_checked`	STATISTIC — Number of cycles in which *test_expr1* or *test_expr2* loaded a value.
`cover_matrix_covered`	CORNER — If non-zero, all bits of the cross-product coverage matrix are covered.

Cover Groups

None

See also

ovl_coverage
ovl_xproduct_bit_coverage
ovl_value_coverage

Examples

Example 1

```
ovl_xproduct_value_coverage #(
   .severity_level(`OVL_ERROR),
   .width1(3),
   .width2(2),
   .coverage_check(1'b0),
   .property_type(`OVL_ASSERT),
   .msg("OVL_VIOLATION : "),
   .coverage_level(`OVL_COVER_NONE))
   XVC1 (
      .clock(clock),
      .reset(1'b1),
      .enable(1'b1),
      .test_expr1(a),
      .test_expr2(b),
      .val1(1'b0),
      .val2(1'b0),
.fire(fire));
```

Maintains the following cross-product coverage matrix:

(a==7)&&(b==3)	(a==7)&&(b==2)	(a==7)&&(b==1)	(a==7)&&(b==0)
(a==6)&&(b==3)	(a==6)&&(b==2)	(a==6)&&(b==1)	(a==6)&&(b==0)
(a==5)&&(b==3)	(a==5)&&(b==2)	(a==5)&&(b==1)	(a==5)&&(b==0)
(a==4)&&(b==3)	(a==4)&&(b==2)	(a==4)&&(b==1)	(a==4)&&(b==0)
(a==3)&&(b==3)	(a==3)&&(b==2)	(a==3)&&(b==1)	(a==3)&&(b==0)
(a==2)&&(b==3)	(a==2)&&(b==2)	(a==2)&&(b==1)	(a==2)&&(b==0)
(a==1)&&(b==3)	(a==1)&&(b==2)	(a==1)&&(b==1)	(a==1)&&(b==0)
(a==0)&&(b==3)	(a==0)&&(b==2)	(a==0)&&(b==1)	(a==0)&&(b==0)

Example 2

```
ovl_xproduct_value_coverage #(
   .severity_level(`OVL_ERROR),
   .width1(3),
   .width2(2),
   .min1(3),
   .min2(1),
   .max1(4),
   .coverage_check(1'b1),
   .property_type(`OVL_ASSERT),
   .msg("OVL_VIOLATION : "),
   .coverage_level(`OVL_COVER_NONE))

   XVC2 (
      .clock(clock),
```

```
        .reset(1’b1),
        .enable(1’b1),
        .test_expr1(a),
        .test_expr2(b),
        .val1(1’b0),
        .val2(1’b0),
  .fire(fire));
```

Maintains the following cross-product coverage matrix:

```
(a==4)&&(b==3)  (a==4)&&(b==2)  (a==4)&&(b==1)

(a==3)&&(b==3)  (a==3)&&(b==2)  (a==3)&&(b==1)
```

If the Coverage Matrix Bitmap is 111100, the cross-product coverage matrix is:

```
1           1           1

1           0           0
```

Here, all combinations were covered except (a==3)&&(b==2) and (a==3)&&(b==1).

Example 3

```
  ovl_xproduct_value_coverage #(
     .severity_level(‘OVL_ERROR),
     .width1(8),
     .width2(4),
     .val1_width(8),
     .val1_count(3),
     .val2_width(4),
     .val2_count(4),
     .coverage_check(1’b1),
     .property_type(‘OVL_ASSERT),
     .msg(“OVL_VIOLATION : "),
     .coverage_level(‘OVL_COVER_NONE))

     XVC3 (
        .clock(clock),
        .reset(1’b1),
        .enable(1’b1),
        .test_expr1(a),
        .test_expr2(b),
        .val1(24’b111111110111111100000001),
        .val2(16’b0111000001010010),
  .fire(fire));
```

Maintains the following coverage matrix:

```
(a==225)&&(b==7)  (a==225)&&(b==0)  (a==225)&&(b==5)  (a==225)&&(b==2)

(a==127)&&(b==7)  (a==127)&&(b==0)  (a==127)&&(b==5)  (a==127)&&(b==2)

(a==1)&&(b==7)    (a==1)&&(b==0)    (a==1)&&(b==5)    (a==1)&&(b==2)
```

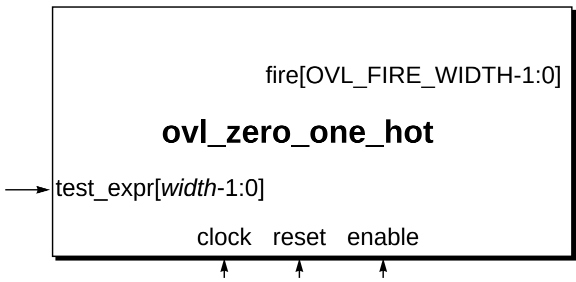
If the Coverage Matrix Bitmap is 101111111110, the cross-product coverage matrix is:

1	0	1	1
1	1	1	1
1	1	1	0

Here, all combinations were covered except (a==225)&&(b==0) and (a==1)&&(b==2).

ovl_zero_one_hot

Checks that the value of an expression is zero or one-hot.



Parameters/Generics:
severity_level
width
property_type
msg
coverage_level
clock_edge
reset_polarity
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)]
  instance_name (clock, reset, enable, test_expr, fire);
```

Parameters/Generics

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_COVER_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 active 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.

Cover Groups

none

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.

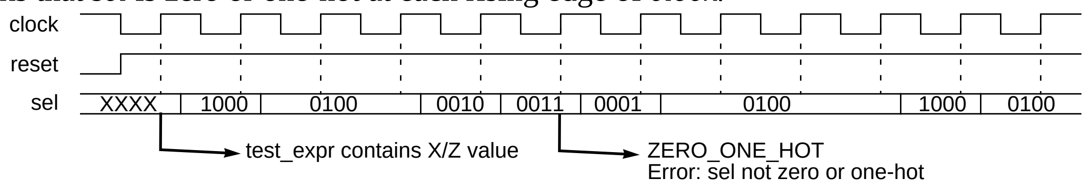
See also

ovl_one_cold   ovl_one_hot

Examples

```
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
```

Checks that *sel* is zero or one-hot at each rising edge of *clock*.



Appendix A
OVL Macros

Global Macros

Type	Macro	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	Removes initialization logic from checkers. Default: not defined.
Function	OVL_ASSERT_ON	Activates assertion logic. Default: not defined.
	OVL_COVER_ON	Activates coverage logic. Default: not defined.
	OVL_COVERGROUP_OFF	Excludes cover group monitoring logic from 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.

Type	Macro	Description
	`OVL_CLOCK_EDGE_ DEFAULT`	Value of *clock_edge* to use when the parameter is unspecified. Default: OVL_POSEDGE.
	`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.

Type	Macro	Description
X/Z Values	OVL_IMPLICIT_XCHECK_OFF	Turns off implicit X/Z checks. Default: not defined.
	OVL_XCHECK_OFF	Turns off all X/Z checks. Default: not defined.

Internal Global Macros

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
```

Macros Common to All Assertions

Parameter	Macro	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`	Includes coverage logic for all of the checker's cover points if OVL_COVER_ON is defined.
	`OVL_COVER_NONE`	Excludes coverage logic for all of the checker's cover points.
	`OVL_COVER_SANITY`	Includes coverage logic for the checker's SANITY cover points if OVL_COVER_ON is defined. Can be bitwise-ORed with OVL_COVER_BASIC, OVL_COVER_CORNER and OVL_COVER_STATISTIC.
	`OVL_COVER_BASIC`	(default) 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.

Parameter	Macro	Description
	`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.

Macros for Specific Assertions

Parameter	Checkers	Macro	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	Macro	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.

Appendix B
OVL Backward Compatibility

V2.3

The V2.3 version of OVL is compatible with the V1.8 version. That is, EDA tools that analyzed designs with V1.8 checkers will work seamlessly with the V2.3 OVL implementation. These checkers are identified by the prefix *assert_* (see Table B-1).

Table B-1. *assert_ Checker Types**

assert_always	`assert_increment`	assert_proposition
assert_always_on_edge	`assert_never`	assert_quiescent_state
assert_change	`assert_never_unknown`	assert_range
assert_cycle_sequence	`assert_never_unknown_async`	assert_time
assert_decrement	`assert_next`	assert_transition
assert_delta	`assert_no_overflow`	assert_unchange
assert_even_parity	`assert_no_transition`	assert_width
assert_fifo_index	`assert_no_underflow`	assert_win_change
assert_frame	`assert_odd_parity`	assert_win_unchange
assert_handshake	`assert_one_cold`	assert_window
assert_implication	`assert_one_hot`	assert_zero_one_hot

The *assert_** checkers have the same parameters and ports as the V1.*x* versions of the checkers, so their instance specifications have not changed. However, these checkers do not have the extended parameters (*clock_edge*, *reset_polarity* and *gating_type*) and ports (*enable* and *fire*) added to the new V2 OVL implementations. For this reason, they are deprecated.

The new V2 OVL checkers are identified by the prefix *ovl_* (see Table B-2).

Table B-2. *ovl_ Checker Types**

ovl_always	ovl_memory_async	ovl_quiescent_state
ovl_always_on_edge	ovl_memory_sync	ovl_range
ovl_arbiter	ovl_multiport_fifo	ovl_reg_loaded
ovl_bits	ovl_mutex	ovl_req_ack_unique
ovl_change	ovl_never	ovl_req_requires
ovl_code_distance	ovl_never_unknown	ovl_stack
ovl_cycle_sequence	ovl_never_unknown_async	ovl_time
ovl_decrement	ovl_next	ovl_transition
ovl_delta	ovl_next_state	ovl_unchange
ovl_even_parity	ovl_no_contention	ovl_valid_id
ovl_fifo	ovl_no_overflow	ovl_value
ovl_fifo_index	ovl_no_transition	ovl_width
ovl_frame	ovl_no_underflow	ovl_win_change
ovl_handshake	ovl_odd_parity	ovl_win_unchange
ovl_hold_value	ovl_one_cold	ovl_window
ovl_implication	ovl_one_hot	ovl_zero_one_hot
ovl_increment	ovl_proposition	

These include 33 checkers that are extended versions of their *assert_** counterparts. Plus completely new checkers.

assert_fifo_index and *ovl_fifo_index*

The V1 assert_fifo_index checker is compatible with the V2 implementation. But the ovl_fifo_index implementation has a change in the parameter order. The *simultaneous_push_pop* parameter was moved to before the *property_type* parameter. So, the assert_fifo_index checker has the following syntax:

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

Whereas the ovl_fifo_index checker has the following 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);
```