OpenVera Checker Library Reference Manual

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

Contents

1

Checker Library

The OpenVera Assertions Checker Library is a collection of temporal expressions and assertions intended for a variety of commonly needed tests. These checkers will enable you to significantly increase the speed of coding your own temporal assertions.

This chapter covers the following topics:

- Conventions — Explains the basic syntax components of the checkers covered in this manual.
- Coverage Properties
- Shared Syntax — Describes syntax elements shared among most of the checkers
- Functional Groups — Lists and organizes all OVA checkers within functional groups.

- Checker Descriptions — Provides syntax, descriptions, and examples for all standard OVA checkers.

Conventions

All OVA checkers described in this chapter are available in both unit and template form. The two forms have identical functions, however, their usage differs as follows:

- Use the unit form to bind checkers to a design
- Use the template form to build more complex checkers inside your own units

The unit form of a checker is distinguished from the template form by an "`ova_`" prefix. For example, "`ova_bit`" is the unit form and "`bit`" is the template form of the same checker. To simplify the descriptions, the checkers are referred to by their unit names unless only the template form is meant.

Units are the preferred way to inline assertions in your Verilog.

The following is an example of a one-hot check of signal `a` inlined within Verilog:

```
module m(a, b, clk)
input a, b, clk;
//ova bind ova_one_hot(1'b1, top.clk, a);
//unit instantiation. XMRs okay
endmodule
```

Coverage Properties

The checkers contain OVA assertions used in verification of the intended behavior (the original purpose of the checkers), and a large number of them also contain coverage properties that can be used to detect the occurence of events related to the behavior, in particular such as triggering conditions and corner cases.

The coverage can be controlled globally by a macro symbol `OVA_COVER_ON` and locally on a per-instance basis using a parameter `coverage_level` (See “Shared Syntax” on page1- 5.).

Note that you can also use $ova tasks such as $ova_stop, $ova_start, $ova_assertion_start, and $ova_assertion_stop also to control cover properties locally by inserting them in your code.

The following checkers contain coverage statements:

Standard OVA Checkers

`ova_arbiter`	`ova_asserted`
`ova_bits`	`ova_code_distance`
`ova_deasserted`	`ova_driven`
`ova_dual_clk_fifo`	`ova_even_parity`
`ova_fifo`	`ova_follows`
`ova_hold`	`ova_hold_value`
`ova_memory`	`ova_mutex`
`ova_next_state`	`ova_no_contention`
`ova_one_hot`	`ova_overflow`
`ova_range`	`ova_req_ack_unique`
`ova_req_requires`	`ova_stack`
`ova_timeout`	`ova_value`
`ova_window`	

OVL-Equivalent Checkers

`assert_always`	`assert_always_on_edge`
`assert_change`	`assert_even_parity`
`assert_odd_parity`	`assert_one_cold`
`assert_one_hot`	`assert_unchange`
`assert_zero_one_hot`	

Shared Syntax

Many checkers share the same syntax elements. These elements are described in this section in order to avoid repeating their descriptions throughout this document.

Macro Symbols

The way checkers are used, whether for checking by assertions or coverage gathering or both, can be selected using two global ‘define symbols:

`OVA_ASSERT_OFF`

When the symbol is **defined**, all assertions are **removed** from the checker. That is, when left undefined the behavior is backward compatible with earlier versions of the library.

`OVA_COVER_ON`

When the symbol is **defined**, the cover statements in the checkers are **included**.

Parameters and Ports

The following are shared ports and parameters of all the checkers.

`en`

Used as a scaler guard expression (port). This expression enables the start of a check. Default = 1`b1 (if *`en`* is not specified, it defaults to true).

Once the check starts, *`en`* can change without affecting the checker. The *`en`* scaler expression can be used to specify a not-a-reset situation or some other guard expression (potentially in combination with the reset expression in a complex boolean expression).

`edge_expr`

Specifies the active edge for the clock signal (*`clk`*) in unit syntax. Use the following parameter values to specify the edge type:

- posedge: 0 (the default)
- negedge: 1
- edge: 2

If *`edge_expr`* is not specified, it defaults to posedge.

`clk`

Specifies the clock signal (port) on which inputs are sampled and the checks are performed. In unit syntax, give just the clock signal. The active edge is specified with *`edge_expr`*. In template syntax, use the full edge expression, as in `posedge m_clk`.

`msg`

Specifies a quoted text string (parameter) that is printed to output when the checker's assertion fails. The default message is "`assertion triggered`". If the default message is used in a checker instantiation, the empty actual parameter can be omitted.

`severity`

Specifies the severity level (parameter) of the assertion, default is 0. This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/disable individual or groups of assertions.

`category`

Specifies the category of the assertion, default is 0. This parameter can be used to group assertions used for a similar purpose, and provide a selection/filtering mechanism to enable/ disable individual or groups of assertions.

Note also the following:

- The descriptions also give the names of the assertions used in the checkers. Use these names to identify the results of the checkers in the OVA reports. If a checker has more than one assertion, this information also identifies more exactly what succeeded or failed.
- In the unit form, all parameters are integers except for *`msg`*, which is a string. All ports are the logic type. Port widths are 1 bit unless otherwise indicated.
- All standard OVA checkers described in this chapter have *`severity`*, *`category`*, and `coverage_level` (if present at all) parameters as the the last items on the unit parameter list and the template argument list. Note that coverage properties are not available in the template form of the checkers.

coverage_level

Specifies which coverage levels should be enabled (provided that the symbol `COVER_ON` is defined.) The following levels are supported (default is 2):

Level 1: Basic coverage, implemented using cover statements. Used by simulation and Magellan.

Level 1 is enabled by setting bit 0 of `coverage_level` to 1.

Example: The number of Enqueues and Dequeues in a FIFO.

L*evel 2:* This level is intended mainly for data coverage using cover groups in System Verilog. Since OVA does not support such constructs this level is absent in all but a few checkers where it is implemented using cover (property) statements.

Level 2 is enabled setting bit 1 of `coverage_level` to 1. **This is the default level selected by this parameter (for compatibility with SVA checkers.)**

Example: Inidividual bits in a vector asserted at-least once.

Level 3: Mostly cover statements for specific corner points as specified by parameters of the checker. Used primarily by formal tools as goals, but can be enabled in simulation too. These coverage items ensure that the corner case condition of the RTL/design block are verified during testing.

Level 3 is enabled setting bit 2 of `coverage_level` to 1.

Examples: The number of times FIFO reached HIGH water mark. The number of times ACK was received at the next clock after REQ was issued. The number of times the specified Min latency value was reached.

Functional Groups

This section lists all checkers according to functional groups. The following groups are included:

- Value Integrity
- State Integrity

- Temporal Sequence
- Protocol

Value Integrity

Value Integrity checkers check and verify that the values of objects (signals, etc.) are within the constraints and limitations specified. The checks are performed specifically to a single object and not with respect to other objects. The following checkers are included in this category:

`ova_arith_overflow`	`ova_asserted`
`ova_bits`	`ova_check_bool`
`ova_const`	`ova_deasserted`
`ova_dec`	`ova_delta`
`ova_even_parity`	`ova_forbid_bool`
`ova_inc`	`ova_mutex`
`ova_no_contention`	`ova_odd_parity`
`ova_overflow`	`ova_range`
`ova_underflow`	`ova_value`

State Integrity

State Integrity checkers ensure correct state transitions and correct value encodings of state variables. The following checkers are included in this category:

`ova_code_distance`	`ova_driven`
`ova_multiport_fifo`	`ova_next_state`
`ova_one_cold`	`ova_one_hot`
`ova_quiescent_state`	`ova_tri_state`

Temporal Sequence

Temporal Sequence checkers ensure the correct timing of events. The following checkers are included in this category:

`ova_hold`	`ova_hold_value`
`ova_reg_loaded`	`ova_sequence`
`ova_timeout`	`ova_window`

Protocol

Protocol checkers ensure the correct ordering of events and values over time. These events and values normally involve two or more signals from the design. The following checkers are included in this category:

`ova_arbiter`	`ova_data_used`
`ova_dual_clk_fifo`	`ova_fifo`
`ova_follows`	`ova_memory`
`ova_memory_async`	`ova_multiport_fifo`
`ova_no_contention`	ova_req_ack_unique
`ova_req_requires`	`ova_req_resp`
`ova_stack`	`ova_valid_id`

Checker Descriptions

This section provides syntax, descriptions, and examples for all standard OVA checkers.

Note: The severity and category parameters for these checkers can take defalt values as specified by the `set_severity` and `set_category` OVA commands.

ova_arbiter

Ensures that a resource arbiter provides grants to corresponding requests between the specified minimum and maximum number of clock cycles between a request and a grant.

Unit syntax:

```
ova_arbiter
#(no_chnl, bw_prio, grant_one_chk, fairness_chk,
  priority_chk, fifo_chk, min_lat, max_lat, edge_expr,
  msg, severity, category, coverage_level)
instance_name (en, clk, requests, priority, grants);
```

Template syntax:

```
arbiter(en, clk, requests, priority, bw_prio, grants,
no_chnl, grant_one_chk, fairness_chk, priority_chk,
fifo_chk, min_lat, max_lat, msg, severity, category);
```

Arguments

`no_chnl`
: The number of channels (bits) in requests and grants. Default = 2.

`bw_prio`
: The number of bits that are used to encode *`req_priority`*. Default = 1.

`grant_one_chk`
: If true, then the unit will check to ensure that only one grant is issued per clock cycle. Default = 1.

`fairness_chk``, `*`req_priority_chk`*`, and `*`fifo_chk`*
: Indicate which arbitration scheme is to be verified. These must be compile-time constants. If no arbitration scheme is asserted, no checks will be performed to verify the arbitration scheme.

`fairness_chk`

If true, then this unit will ensure that no channel will be issued more than one grant while other channels have requests pending except if this is the only request at the highest *req_priority* when *req_priority_chk* is asserted. Default = 0.

`req_priority_chk`

If true, then this checker will ensure that grants are issued according to the priority indicated in the *req_priority* vector.

If 0 (disabled), then req_priority is not taken into account in any of the other checks. However, the argument *priority* must still have the correct dimension even though the actual values do not matter (e.g., pass vector of 0’s). The *req_priority* vector may be a design vector (i.e., not a constant array). However, while a request is being processed the *req_priority* should not change, otherwise certain checks may produce incorrect results (success or failure). Default = 0.

`fifo_chk`

If true, then this unit will ensure that grants are issued according to the order that their requests were received unless *req_priority_chk* is asserted which means that the fifo check is performed only on requests of the current highest req_priority. Default = 0.

`min_lat`

The minimum global grant latency. Default = 1.

If 0, then the grant is expected starting the same cycle as the request (i.e., combinational arbiter is possible with *max_lat* = 0). If priority arbitration check is enabled, then *min_lat* should be 0 or 1 only.

`max_lat`

If *max_lat* > 0, it specifies the maximum global grant latency regardless of the selection criterion. That is, a persistent request must be granted within *max_lat* clock cycles. The check is useful in systems where a request must be granted within a certain latency even in the presence of other requests.

If *max_lat* = 0, the global latency check is disabled. Default = 0.

requests

Requests signals as vectors.

Vector of size [no_chnl-1:0] where the bits correspond to the corresponding channels in reqs . req is assumed to be 1 when active.

priority

A [bw_prio*no_chnl-1 : 0] bitvector of bw_prio*no_chnl bits formed by concatenating non-negative integer req_priority values corresponding to the request lines. The right-most bw_prio bits in req_priority corresponds to channel 0, etc. The req_priority value 0 is the lowest *req_priority*.

For the assertions in the checker to operate correctly, the *priority* assignments to the requests should remain constant over time. Otherwise, the assertions may report unwanted failures.

grants

Grants signals as vectors.Vector of size [no_chnl-1:0] where the bits correspond to the corresponding channels in *reqs* . Assumed to be 1 when active.

Note the following:

- If *`min_lat`* is 0 then the grant is expected starting the same cycle as the request (that is, a combinational arbiter is possible with *`min_lat`* = 0).
- If *`min_lat`* > 0 and *`max_lat`* = 0 then the grant can arrive any time at or after the next clock cycle, that is, in [*`min_lat`* ..]. Usually this is only possible when a priority arbiter is employed.
- The *`fairness_chk`*, *`priority_chk`*, and *`fifo_chk`* arguments indicate which arbitration scheme is to be verified. These must be compile-time constants. If no arbitration scheme is asserted, no checks are performed to verify the arbitration scheme.

Assumptions:

- FIFO and fairness checks should not normally be enabled at the same time because they may contradict each other.
- When *`req_priority_chk = 0`* it is assumed that the req_priority vector is set to all 0 values.
- A grant is expected to be one clock cycle wide, unless the grant is for more than one consecutive request.

  It is assumed that a request holds asserted until granted. It is assumed that a request is removed on the clock tick the grant is sampled, unless another request is immediately asserted. Its removal before a grant causes the grant to be cancelled, and this does cancel the check for a grant.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_one_cycle_gnt[i]` checks that, in position `[i]`, the grant signal is deasserted on the next cycle.

- Assertion `ova_c_req_granted[i]` checks that, in position `[i]`, the grant signal arrives within the specified latency.
- Assertion `ova_c_granted_only_if_req[i]` checks that, in position `[i]`, there is no grant signal when there is no req signal.
- Assertion `ova_c_highest_grant[i]` checks that, for line `[i]`, the granted request is that of the highest priority. This assertion is active only if *`priority_chk`* is 1.
- Assertion `ova_c_fifo_chk[i][j]` checks if, on line `[j]`, the grant is not asserted before the grant on line `[i]`, while the requests came in the opposite order. This assertion is active only if *`fifo_chk`* is 1.
- Assertion `ova_f_two_active[i][j]` checks that, although there are continuous requests on lines `[i]` and `[j]`, the grant on line `[i]` does not arrive twice without a grant on line `[j]`. This assertion is active only if *`fairness_chk`* is 1.
- Assertion `ova_c_single_grant` checks that there is no more than one grant at the same time. This assertion is active only if *`grant_one_chk`* is 1.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_arbiter_req_granted` counts the number of granted requests, for each channel

Cover `cover_abandoned_req` counts the number of abandoned request, for each channel

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_req_granted_exactly_after_min_lat` indicates how many times the req-to-grant latency was exactly equal to the specified `min_lat` value.

Cover `cover_req_granted_exactly_after_max_lat` indicates how many times the req-to-grant latency was exactly equal to the specified `max_lat` value.

Examples

- Assume that there are two request and grant lines enabled all the time.
- Priority 0 and 1 encoded over one bit, for channel 0 and 1, respectively.
- Implement *`grant_one_chk`*, *`fairness_chk`*, and *`priority_chk.`*
- Require that a grant be issued within 2 to 5 clock cycles after the request.
- Coverage Level 1 enabled by default in the unit instance.

Unit:

```
bind module dut : ova_arbiter
  #(2, , , 1, 1, , 2, 5, 0, "Arbiter failed.")
arbiter_inst(1'b1, clk, 2'b1_0, requests, grants);
```

Template:

```
arbiter ( , posedge clk, requests, , grants, , , 1,
     1, , 2, 5, "Arbiter failed.");
```

ova_arith_overflow

Checks that the value of a signal does not overflow the range of a specified target signal. An overflow occurs when the bits of exp that do not fit in the target signal have a non-zero value.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_arith_overflow
#(target_bw, exp_bw, edge_expr, msg, severity, category)
instance_name (en, clk, target, exp);
```

Template syntax:

```
arith_overflow(en, clk, target, target_bw, exp, exp_bw, msg,
severity, category);
```

Arguments

`target_bw`
: Number of bits in the specifed target signal (*`target`*). Default = 1.

`exp_bw`
: Number of bits in the specified signal of interest (*`exp`*). Default = 1.

`target`
: Signal that receives the signal of interest (*`exp`*).

  The target port is optional. It is not currently used but might be in a future version. You can include target for readability, so that it is clear what the target signal is.

`exp`

Signal of interest whose value is compared against the target signal (*`target`*).

Assumptions:

- Unsigned values only.
- The target and exp ports are vectors of descending range where the msb is the left-most bit.
- For the checker to have meaning, *`target_bw`* must have a value less than exp_bw and both must be greater than zero.
- The exp is assumed to have a size of [exp_bw - 1 : 0] and target a size of [target_bw - 1 : 0].

Report Message

In report messages, the assertion name is `ova_c_arith_oflow`.

Examples

Note that the following examples:

- Ensure that the value on sigA, a 16-bit vector, fits on sigB, which has only eight bits.
- A message is returned when an overflow occurs.

Unit:

```
bind module dut : ova_arith_overflow sigAtoB
  #(8, 16, , "sigA overflowed sigB.")
oflo_inst (rst_n, clk, sigB, sigA);
```

Template:

```
arith_overflow sigAtoB (rst_n, posedge clk, sigB, 8,
     sigA, 16, "sigA overflowed sigB.");
```

ova_asserted

Once the specified start expression (*`start`*) evaluates as true, this checker makes sure that the signal under test (*`exp`*) is asserted (1 or true) until the stop expression (*`stop`*) evaluates true. Once the stop expression is true, the signal under test (*`exp`*) can be de-asserted. The check is performed on the active clock (*`clk`*) specification. The delay specification is the number of cycles after the start expression evaluates true before the signal under test (`exp`) will be asserted. The default value for *`delay`* is zero, meaning that the evaluation starts on the same clock tick that the start expression becomes true. The *`delay`* must be a non-negative integer.

Unit syntax:

```
ova_asserted
#(delay, edge_expr, msg, severity, category, coverage_level)
 instance_name (en, clk, exp, start, stop);
```

Template syntax:

```
asserted(en, clk, exp, start, stop, delay, msg, severity,
category);
```

Arguments

`delay`
: The number of clock cycles after the start signal (*`start`*) is true and before the signal under test (*`exp`*) is asserted. Default = 0.

`exp`
: Signal being tested.

`start`
: Signal that marks the start of the window.

`stop`
: Signal that marks the end of the window.

Report Message

In report messages, the assertion name is `ova_c_asserted`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_num_of_start_events` indicates how many times `start` occurred.

Cover `cover_num_of_matches` indicates how many times `exp` remained true within the required interval from `start` plus `delay` to `stop`.

Example

If in `burst` mode, ensure that the burst control signal remains asserted starting 1 clock after burst mode is entered and ending after leaving burst mode. The default message is output on a failure. Coverage Level 1 is enabled in the unit instance.

Unit:

```
bind module dut : ova_asserted
  #(1, , , , , 1)
asserted_inst (rst_n, clk, burst, start, stop);
```

Template:

```
bool start : (mode == 2'b01);
asserted burst_asserted(rst_n, posedge clk, burst,
     start, ~start, 1, );
```

ova_bits

Checks that the value of the signal being tested (`exp)` falls between the specified minimum (*`min`*) and maximum (*`max`*) number of bits (inclusive) that are asserted or deasserted as indicated by the *`deasserted`* flag. To specify a single number and not a range, ensure that *`min`* == *`max`*. Make sure that *`min`* and *`max`* are non-negative integers and *`min`* ≤ *`max`*.

Syntax

Unit syntax:

```
ova_bits
#(min, max, deasserted, exp_bw, edge_expr, msg, severity,
   category, coverage_level)
instance_name (en, clk, exp);
```

Template syntax:

```
bits(en, clk, exp, min, max, deasserted, msg, severity,
category);
```

Arguments

`min`
  Minimum number of bits asserted or deasserted. Default = 1.

`max`
  Maximum number of bits asserted or deasserted. Default = 1.

`deasserted`
  If 1, checks for deasserted (0) bits. If 0, checks for asserted (1) bits. Default = 0.

`exp_bw`
  The number of bits in <exp>. Default = 2.

`exp`

Signal being tested (it must be more than one bit wide.)

Report Message

In report messages, the assertion name is `ova_c_bits`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_bits_exp_change` indicates how many times `exp` changed value.

Cover `cover_bits` indicates how many times the required behavior was matched on a change of `exp` value.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_max_bits_asserted` indicates how many times the `max` number of bits was (de)asserted on a change of `exp` value.

Cover `cover_min_bits_asserted` indicates how many times the `min` number of bits was (de)asserted on a change of `exp` value.

Covers `cover_bits_asserted[i]` indicate how many times bit `exp[i]` was (de)asserted on a change of `exp` value.

Example

The following examples ensure that state is one cold by counting the bits that are deasserted. Coverage Levels 1 and 3 are enabled in the unit instance.

Unit:

```
bind module dut : ova_bits one_cold
  #(1, 1, 1, 8, 1, "Too many cold bits.", , ,5)
 cold_inst_name (rst_n, clk, state);
```

Template:

```
bits one_cold(rst_n, negedge clk, state, 1, 1, 1,
     "Too many cold bits.");
```

ova_check_bool

Verifies that the specified expression is always true.

Syntax

Unit syntax:

```
ova_check_bool
#(edge_expr, msg, severity, category)
instance_name (expr, clk);
```

Template syntax:

```
check_bool(expr, msg, clk, severity, category);
```

Arguments

`expr`
: Signal being tested.

Report Message

In report messages, the assertion name is `cb`.

Example

The following examples ensure that signal is always true.

Unit:

```
bind module dut : ova_check_bool
  #( , "sig is false.")
bool_inst (sig, sysclk);
```

Template:

```
check_bool (sig, "sig is false.", posedge sysclk);
```

ova_code_distance

Checks that when the specified signal (*exp*) changes, the number of bits that are different compared to *exp2* fall within the specified minimum (*min*) and maximum (*max*) number of bits.

Syntax

Unit syntax:

```
ova_code_distance
#(min, max, bw, edge_expr, msg, severity, category,
  coverage_level)
instance_name (en, clk, exp, exp2);
```

Template syntax:

```
code_distance(en, clk, exp, exp2, min, max, msg, severity,
category);
```

Arguments

`min`
: The minimum number of bits that are different. Default = 1.

`max`
: The maximum number of bits that are different. Default = 1.

`bw`
: The number of bits in *exp* and *exp2*. Default = 2.

`exp`
: Signal being tested.

`exp2`
: Signal that that signal under test (*exp*) is compared to.

Report Message

In report messages, the assertion name is `ova_c_code_distance`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_exp_change` indicates how many times `exp` value changed.

Cover `cover_code_distance_match` indicates how many time the code distance matched the requirements.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_code_distance_eq_to_min` indicates how many times the code distance was exactly `min` bits.

Cover `cover_code_distance_eq_to_max` indicates how many times the code distance was exactly `max` bits.

Examples

Raw data response is checked against the selected likely response chosen via Hamming code correction to ensure that it is within the difference of 0 to 3 bits permitted by the code. The default message is output on a failure. Coverage Level 3 is enabled in the unit instance.

Unit:

```
bind module dut : ova_code_distance
  #(0, 3, 16, , , , ,4)
 check_hamming (rst_n, clk, raw_data, selected_match);
```

Template:

```
code_distance check_hamming_result(rst_n, posedge clk,
     raw_data, selected_match, 0, 3, );
```

ova_const

Checks that the value of the signal being tested (*exp*) is always constant.

The check is performed only on the active clock (*clk*) edge (the signal being tested can glitch in between the specified clock edges without causing this check to fail).

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_const
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template syntax:

```
const(en, clk, exp, msg, severity, category);
```

Arguments

`bw`
    The number of bits in the signal being tested (*exp*). Default = 1.

`exp`
    Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_const`.

Examples

The following examples ensure that the local sync reset is not active when there is no global asynchronous reset. The default message is sent on a failure.

Unit:

```
bind module dut : ova_const no_running_rst
  (rst_n, clk, sync_rst_n);
```

Template:

```
const no_running_rst (rst_n, posedge clk,
  sync_rst_n, );
```

ova_cover_bool

Collects information about when the mandatory argument *`expr`* is high, low, posedge or negedge as selected by the parameter *`cover_kind`* and sampled by the clock.

Syntax

Unit syntax:

```
ova_cover_bool
#(edge_expr, msg, cover_kind, severity, category)
 instance_name (expr, clk);
```

Template syntax:

```
cover_bool(expr, msg, clk, cover_kind, severity, category);
```

Arguments

`expr`

Signal being tested.

`cover_kind`

Specifies value at which to test *`expr`*.

`cover_kind`	**Expression covered**
`0`	`expr == 1'b0`
`1`	`expr == 1'b1  (default)`
`2`	`posedge expr`
`3`	`negedge expr`

ova_data_used

Checks that data from the source signal (`src[sleft:sright]`) appears in the desitination signal (`dest[dleft:dright])` within the specified window. The window is specified as the number (*`start)`* of cycles from the specified time the trigger signal (*`trigger`*) is asserted until the specified number of cycles (*`finish`*) after the trigger signal is asserted.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_data_used
#(sleft, sright, dleft, dright, start, finish, edge_expr,
msg, severity, category)
instance_name (en, clk, trigger, src, dest);
```

Template syntax:

```
data_used(en, clk, trigger, src, sleft, sright, dest, dleft,
dright, start, finish, msg, severity, category);
```

Arguments

`sleft`

The most significant bit of the source signal's (*`src`*) bit slice. Default = 1.

`sright`

The least significant bit of the source signal's (*`src`*) bit slice. Default = 0.

`dleft`

The most significant bit of the destination signal's (*`dest`*) bit slice. Default = 1.

`dright`
: The least significant bit of the destination signal's (*`dest`*) bit slice. Default = 0.

`start`
: The number of cycles after the trigger signal (*`trigger`*) asserts to start the window. Default = 1. Note that the specified number must be greater than zero.

`finish`
: The number of cycles after the trigger signal (*`trigger`*) asserts to stop the window. Default = 1.

  For an open-ended interval [*`start`* ..], set *`finish`* equal to zero.

`trigger`
: Signal that is part of starting the window.

`src`
: Source signal.

`dest`
: Destination signal.

Report Message

In report messages, the assertion name is `ova_c_data_used`. When the assertion fails, the "offending" expression in the message might include "`ova_v_src_value`". This is an OVA variable that stores the original source signal (*`src`*) value.

Examples

The following examples ensure that the lower 18 bits of `busAddr` is the address value communicated to the memory chip, and that the `memAddr` transitions two cycles after `adrStrb` signals a good address value on `busAddr` and remains valid 5 cycles after `adrStrb`.

Unit:

```
bind module dut : ova_data_used addr_decode
  #(17, 0, 17, 0, 2, 5)
  (rst_n, clk, adrStrb, busAddr, memAddr);
```

Template:

```
data_used addr_decode(rst_n, posedge clk, adrStrb,
     busAddr, 17, 0, memAddr, 17, 0, 2, 5, );
```

ova_deasserted

Once the start expression (*`start`*) evaluates true, this checker makes sure that the signal being tested (*`exp`*) is deasserted (0 or false) until the stop expression (*`stop`*) evaluates true (excluding the clock tick when *`stop`* is true). The specified delay (*`delay`*) is the number of cycles after the start expression (*`start`*) evaluates true before the signal being test (*`exp`*) is deasserted. The specified delay must be a non-negative integer (it defaults to zero, meaning that the check starts on the same clock tick as when start expression (*`start`*) becomes true.

Syntax

Unit syntax:

```
ova_deasserted
#(delay, edge_expr, msg, severity,
  category, coverage_level)
instance_name (en, clk, exp, start, stop);
```

Template syntax:

```
deasserted(en, clk, exp, start, stop, delay, msg, severity,
category);
```

Arguments

`delay`
: The number of clock cycles after the start signal (*`start`*) goes true before the signal being tested (*`exp`*) is deasserted. Default = 0.

`exp`
: Signal being tested.

`start`
: Signal that marks the start of the window.

`stop`

Signal that marks the end of the window.

Report Message

In report messages, the assertion name is `ova_c_deasserted`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_num_of_start_events` indicates how many times `start` occurred.

Cover `cover_num_of_matches` indicates how many times `exp` remained deasserted within the required interval from `start` plus `delay` to `stop`.

Examples

The following example check that the `wen` line remains deasserted during a read. Coverage Level 1 is enabled in the unit instance.

Unit:

```
bind module dut : ova_deasserted
  #(0, , , , ,1)
no_wen_during_read (rst_n, clk, wen, start, stop);
```

Template:

```
`define READ_OP 4'b0100
bool start : (op === `READ_OP);
deasserted no_wen_during_read(rst_n, posedge clk,
     wen, start, ~start, 0);
```

ova_dec

Checks that when the signal being tested (*exp*) changes value, the new value is always between the specifed minimum (*min*) and maximum (*max*) less than the previous value.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_dec
#(min, max, bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template syntax:

```
dec(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`
: The minimum change in value. Default = 1.

`max`
: The maximum change in value. Default = 1.

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_dec`.

Examples

The following examples ensure that the countdown timer is always decremented by 1.

Unit:

```
bind module dut : ova_dec countdown_check
  #(1, 1, 16, , "Countdown timer error.")
dec_inst(rst_n, clk, count);
```

Template:

```
dec countdown_check(rst_n, posedge clk, count, 1, 1,
     "Countdown timer error.");
```

ova_delta

Checks that when the signal being tested (*exp*) changes value, the new value is ± the specified minimum (*min*) to maximum (*max*) change of the previous value. Both the specified minimum and maximum values must be positive integers.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_delta
#(min, max, bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template syntax:

```
delta(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`
: The minimum change in value. Default = 1.

`max`
: The maximum change in value. Default = 1.

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_delta`.

Examples

The following examples ensure that number of elements in the FIFO changes by only 1 from one clock to the next.

Unit:

```
bind module dut : ova_delta check_state_transition
  #(1, 1, 16, , "FIFO size changed wrong.")
  (rst_n, clk, fifo.elems);
```

Template:

```
delta check_state_transition(rst_n, posedge clk,
     fifo.elems, 1, 1, "FIFO size changed wrong.");
```

ova_driven

Checks that all bits are driven (none are floating Z or X).

Syntax

Unit syntax:

```
ova_driven
#(bw, edge_expr, msg, severity, category, coverage_level)
instance_name (en, clk, exp);
```

Template syntax:

```
driven(en, clk, exp, msg, severity, category);
```

Arguments

`bw`
: The number of bits in the signal being tested (*`exp`*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_driven`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover `cover_exp_not_x_or_z` indicates the number of times `exp` was neither `x` nor `z`. Not usable with Magellan.

Examples

The following examples check that the on-chip bus is always driven. Coverage Level 1 is enabled.

Unit:

```
bind module dut : ova_driven
  #(8, 2, "On-chip bus has X or Z bits.", , ,1)
driven_inst (rst_n, clk, cbus);
```

Template:

```
driven onchip_bus (rst_n, edge clk, cbus,
     "On-chip bus has X or Z bits.");
```

ova_dual_clk_fifo

Implements a checker for a dual-clock, single in- and single out-port queue.

Syntax

Unit syntax:

```
ova_dual_clk_fifo
#(depth, elem_sz, hi_water_mark, enq_lat, deq_lat,
  oflow_chk, uflow_chk, value_chk, enq_edge_expr,
  deq_edge_expr, msg, severity, category, coverage_level)
instance_name
 (reset, enq_clk, enq, enq_data, deq_clk, deq, deq_data);
```

Template syntax:

```
dual_clk_fifo(reset, depth, hi_water_mark, elem_sz,
enq_clk, enq, enq_lat, enq_data, deq_clk, deq, deq_lat,
deq_data, oflow_chk, uflow_chk, value_chk, msg, severity,
category);
```

Arguments

`depth`

The maximum number of elements in the queue. Default = 2. The specified *`depth`* can be at most 2^{16}.

`elem_sz`

The size of queue elements in bits. Default = 1.

`hi_water_mark`

If positive, then the depth of the queue after enqueue will be checked to see if *`hi_water_mark`* is reached. Default = 0.

`enq_lat`

The number of specified cycles (*`enq_clk`*) between *enq* being asserted 1 and *`enq_data`* being valid. Default = 0.

`deq_lat`

The number of *`deq_clk`* cycles between *`deq`* being asserted 1 and *`deq_data`* being valid. Default = 0.

`oflow_chk`

If 1, checks that queue does not overflow the maximum size given by the *`depth`* specification. Default = 1.

`uflow_chk`

If 1, checks that queue is not empty before dequeuing data. Default = 1.

`value_chk`

If 1, checks that *`deq_data`* matches the data at the head of the queue. Default = 1.

`enq_edge_expr`

The active clock edge of *`enq_clk`*. Default = 0.

`deq_edge_expr`

The active clock edge of *`deq_clk`*. Default = 0.

`reset`

Initializes the queue to empty when set to 1. It is assumed that the specified value (*`reset`*) is synchronous and spans at least one clock period of both clocks.

`enq_clk`

Clock signal for enqueue side.

`enq`

Set to 1 when data is being enqueued.

`enq_data`
: Data being enqueued.

`deq_clk`
: Clock signal for dequeue side.

`deq`
: Set to 1 when data is being dequeued.

`deq_data`
: Data being dequeued.

When *`enq`* is asserted 1: If *`oflow_chk`* evaluates true, ensures that queue does not overflow the maximum size given in `depth`. The specified *`depth`* can be at most 2**16.

The *`enq_lat`* value is a compile-time non-negative integer constant that indicates the number of *`enq_clk`* cycles between *`enq`* being asserted 1 and *`enq_data`* being valid. At that point all enqueue data is dropped and further checking is disabled until dequeue occurs. If an enqueue and dequeue happen simultaneously then no overflow is reported.

If the specified *`hi_water_mark`* is a positive value, then the depth of the queue after enqueue will be checked to see if the *`hi_water_mark`* is reached. Once high water has been exceeded once, this check is disabled until the FIFO size falls below the mark again.

When the specified *`deq`* is asserted 1: If *`uflow_chk`* evaluates true, ensures that queue was not empty (underflow). If a dequeue on empty is detected then the check is disabled until the next enqueue operation.

If *`value_chk`* evaluates true, ensures that *`deq_data`* is the same as that at the head of the queue. The *`deq_lat`* specification is a compile-time non-negative integer constant that indicates the number of *`deq_clk`* cycles between *`deq`* being asserted and *`deq_data`* being valid.

If an enqueue and dequeue operations happen simultaneously on an empty queue, it is assumed that enqueue happens after dequeue. This means that if a queue is empty at the time of a simultaneous enqueue and dequeue, it will flag it as error if *`uflow_chk`* is enabled.

If an enqueue and dequeue operations happen simultaneously on a full queue then the enqueue detects a full queue and reports an error if *`oflow_chk`* is enabled.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_f_fifo_dcq_overflow` checks that enq is not issued while the FIFO is full.
- Assertion `ova_f_fifo_dcq_underflow` checks that the deq command is not issued while the FIFO is empty.
- Assertion `ova_c_fifo_dcq_value_chk` checks the integrity of values coming off the queue. This assertion is active only if value_chk is 1.
- Assertion `ova_f_fifo_dcq_hi_water_chk` checks that the FIFO is not filled above the high-water mark.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_number_of_enqs` indicates the number an enqueues.

Cover `cover_number_of_deqs` indicates the number of dequeues.

Cover `cover_enq_followed_eventually_by_deq` indicates the number of times an enqueue was followed later by a dequeue.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_fifo_hi_water_chk` indicates how many times the high water mark was reached on an enqueue.

Cover `cover_number_of_empty` indicates how many times empty was reached on dequeue.

Cover `cover_number_of_full` indicates how many times full was reached on enqueue.

Examples

The examples below specify the following:

- Will reset the queue when reset is 1 (must cover at least one posedge eclk and one negedge dclk)
- The number of elements in the FIFO is 10
- High-water mark is disabled by default, the data size is 16 bits
- All checks (overflow, underflow, and value) are enabled
- There is only the default message
- Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_dual_clk_fifo
  #(10, 16, , , , , , , 0, 1, , , ,5)
fifo_inst
   (reset, eclk, enqueue, data_in, dclk, dequeue, data_out);
```

Template:

```
dual_clk_fifo(reset, 10, , 16, posedge eclk, enqueue,
  , data_in, negedge dclk, dequeue, , data_out);
```

ova_even_parity

Checks that the value of the signal being tested (*exp*) always has an even number of bits set to 1. Usually the specified signal (*exp*) is formed by concatenating the data and parity bits.

Syntax

Unit syntax:

```
ova_even_parity
#(bw, edge_expr, msg, severity, category)
instance_name (en, clk, exp);
```

Template syntax:

```
even_parity(en, clk, exp, msg, severity, category);
```

Arguments

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_even_parity`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover `cover_exp_change` indicates how many times `exp` changed value.

Examples

The following examples ensure data read from memory has even parity. Note that `mdata` includes both the data and the parity bits.

Unit:

```
bind module dut : ova_even_parity
  #(9, , "Parity error on mdata.")
parity_inst (rst_n, clk, mdata);
```

Template:

```
even_parity memory_data(rst_n, posedge clk, mdata,
  "Parity error on mdata.");
```

ova_fifo

Implements a checker for a single-clock, single in- and single out-port queue. The width of the fifo elements is set using the parameter *`elem_sz`*, and the width of the head and tail pointers is set using `ptr_width`.

Syntax

Unit syntax:

```
ova_fifo
 #(depth, elem_sz, hi_water_mark, enq_lat, deq_lat,
   oflow_chk, uflow_chk, value_chk, pass_thru, edge_expr,
   msg, severity, category, coverage_level. ptr_width)
instance_name (reset, clk, enq, enq_data, deq, deq_data);
```

Template syntax:

```
fifo(reset, clk, depth, hi_water_mark, elem_sz, enq, enq_lat,
enq_data, deq, deq_lat, deq_data, oflow_chk, uflow_chk,
value_chk, pass_thru, msg, severity, category, ptr_width);
```

Arguments

`depth`

The maximum size of the queue. Default = 2. The specified depth can be at most 2^{16}.

`elem_sz`

The size of queue elements in bits. Default = 1.

`hi_water_mark`

If positive, then the depth of the queue after enqueue will be checked to see if *`hi_water_mark`* is reached. Default = 0.

`enq_lat`

The number of *`enq_clk`* cycles between *`enq`* being asserted 1 and *`enq_data`* being valid. Default = 0.

`deq_lat`

The number of *`deq_clk`* cycles between *`deq`* being asserted 1 and *`deq_data`* being valid. Default = 0.

`oflow_chk`

If 1, checks that queue does not overflow the maximum size given in *`depth`*. Default = 1.

`uflow_chk`

If 1, checks that queue is not empty before dequeuing data. Default = 1.

`value_chk`

If 1, checks that *`deq_data`* matches the data at the head of the queue. Default = 1.

`pass_thru`

Specifies behavior when enqueue and dequeue operations happen at the same time with an empty queue. If 0, dequeue happens first, triggering an underflow. If 1, enqueue happens first and the data is passed through. Default = 0.

`reset`

Initializes the queue to empty when set to 1. It is assumed that `reset` is synchronous and spans at least one clock period of both clocks.

`enq`

Set to 1 when data is being enqueued.

`enq_data`

Data being enqueued.

`deq`

Set to 1 when data is being dequeued.

`deq_data`

Data being dequeued.

`ptr_width`

Width of the pointer. Default = 16.

`reset` asserted 1 initializes the queue to empty. All operations are synchronous to the clock (*`clk`*) ticks, including reset.

When *`enq`* is asserted 1: If *`oflow_chk`* evaluates true, ensures that queue does not overflow the max size given in *`depth`*. The specified *`depth`* can be at most 2**16.

The *`enq_lat`* is a compile-time non-negative integer constant that indicates the number of cycles between *`enq`* being asserted 1 and *`enq_data`* being valid. At that point all enqueue data is dropped and further checking is disabled until dequeue occurs. If an enqueue and dequeue happen simultaneously then no overflow is reported.

If *`hi_water_mark`* is a positive value, then the depth of the queue after enqueue will be checked to see if *`hi_water_mark`* is reached. Once high water has been exceeded once, this check is disabled until the FIFO size falls below the mark again. If *`hi_water_mark`* = 0 then the high-water mark check is disabled and only overflow is checked when the depth of the queue is exceeded (provided that *`oflow_chk`* = 1).

When *`deq`* is asserted 1: If *`uflow_chk`* evaluates true, ensures that queue was not empty (underflow). If a dequeue on empty is detected then the check is disabled until the next enqueue operation.

If *value_chk* evaluates true, ensures *deq_data* is the same as that at the head of the queue. The specified *deq_lat* is a compile-time non-negative integer constant that indicates the number of cycles between *deq* being asserted and *deq_data* being valid.

If an enqueue and dequeue operation happens simultaneously on an empty queue, then the behavior depends on the *pass_thru* argument to the checker instance (it must be a compile-time constant):

If *pass_thru* = 0 then the dequeue happens before enqueue, hence the empty condition is detected and reported and an underflow (provided that *uflow_chk* = 1). If *value_chk* = 1 then the value check fails.

If *pass_thru* = 1 then it is assumed that enqueue happens first and the data is immediately dequeued and compared with *deq_data* if *value_chk* i s enabled. Also, there is no underflow error reported.

If the enqueue and dequeue operations happen simultaneously on a full queue then no overflow is reported and the new element is enqueued while the element at the head of the queue is dequeued without changing the size of the queue.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_f_fifo_overflow` checks that *enq* is not issued while the FIFO is full.
- Assertion `ova_f_fifo_underflow` checks that the deq command is not issued while the FIFO is empty (and no simultaneous *enq* with *pass_thru* enabled).

- Assertion `ova_c_fifo_value_chk` checks the integrity of values coming off the queue. This assertion is active only if value_chk is 1.
- Assertion `ova_f_fifo_hi_water_chk` checks that the FIFO is not filled above the high-water mark.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_number_of_enqs` indicates the number of enqueue operations.

Cover `cover_number_of_deqs` indicates the number of dequeue operations.

Cover `cover_simultaneous_enq_deq` indicates the number of simultaneous enqueue and dequeue operations.

Cover `cover_enq_followed_eventually_by_deq` matches whenever there is an enqueue followed eventually by a dequeue.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_fifo_hi_water_chk` indicates how many time the high water mark was reached on an enqueue.

Cover `cover_simultaneous_enq_deq_when_empty` indicates how many times there were simultaneous enqueue and dequeue operations on an empty queue.

Cover `cover_simultaneous_enq_deq_when_full` indicates how many times there were simultaneous enqueue and dequeue operations on a full queue.

Cover `cover_number_of_empty` indicates how many times empty is reached on dequeue.

Cover `cover_number_of_full` indicates how many times empty is reached on enqueue.

Examples

The following examples specify the following:

- The FIFO is initialized anytime reset is 1 (synchronously with posedge clk)
- There are up to 10 elements in the fifo
- The *`high_water_mark`* is by default 0 (disabled)
- The size of the data is 16 bits, data_in is enqueued when enqueue is 1 with no latency
- The *`data_out`* specification must be equal to that at the head of the fifo when dequeue is 1 with latency
- The overflow, underflow and value checks are enabled with pass through when empty allowed
- Coverage Levels 1 and 3 are enabled (*`coverage_level`* = 5).
- The default pointer width of 16 is selected.

Unit:

```
bind module dut : ova_fifo
  #(10, 16, , , , , , , 1, , , , , 5)
fifo_inst(reset, clk, enqueue, data_in, dequeue, data_out);
```

Template:

```
fifo(reset, posedge clk, 10, ,16 , enqueue, ,
     data_in, dequeue, , data_out, , , , 1);
```

ova_follows

Checks that the follower expression (*follower*) evaluates true within the specified minimum *min_lat* and maximum *max_lat* latency period once the leader expression (*leader*) evaluates true. That is, the check for *follower* begins in the same clock as *leader* is true. Both the *leader* and *follower* trigger on their positive edges. If *min_lat* and *max_lat* == 0 (the defaults), then leader and follower are expected to be true in the same cycle.

Syntax

Unit syntax:

```
ova_follows
#(min_lat, max_lat, edge_expr, msg, severity,
  category, coverage_level)
instance_name (en, clk, leader, follower);
```

Template syntax:

```
follows(en, clk, leader, follower, min_lat, max_lat, msg,
severity, category);
```

Arguments

`min_lat`

Number of clock cycles between the leader signal (*leader*) going true and the beginning of the latency period. Default = 0.

`max_lat`

Number of clock cycles between leader signal (*leader*) going true and the end of the latency period. Default = 0.

`leader`

Signal that precedes the follower signal (*follower*).

`follower`

Signal that follows the leader signal (*`leader`*).

Report Message

In report messages, the assertion name is `ova_c_follows`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_num_of_leader_triggers` indicates the number of times the leader triggered the evaluation.

Cover `cover_num_of_matches` indicates the number of matches of follower after the leader.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_num_of_matches_exactly_on_min_lat` indicates the number of times follower occurred exactly at `min_lat` clock cycles after leader.

Cover `cover_num_of_matches_exactly_on_max_lat` indicates the number of times follower occurred exactly at `max_lat` clock cycles after leader.

Examples

The following examples ensure a req is always followed by a response from an arbiter arbitrating 2 users. Response must follow req within 1 to 6 clock ticks. Coverage Level 1 is enabled (*`coverage_level`* = 1).

Unit:

```
bind module dut : ova_follows
  #(1, 6, , "Arbiter response 0 is late.", , ,1)
req0_resp0 (rst_n, clk, req[0], resp[0]);
```

```
bind module dut : ova_follows
  #(1, 6, , "Arbiter response 1 is late.", , ,1)
req1_resp1 (rst_n, clk, req[1], resp[1]);
```

Template:

```
follows req0_resp0(rst_n, posedge clk,
     posedge req[0], posedge resp[0], 1, 6,
     "Arbiter response 0 is late.");
follows req1_resp1(rst_n, posedge clk,
     posedge req[1], posedge resp[1], 1, 6,
     "Arbiter response 1 is late.");
```

ova_forbid_bool

Checks that the expression is never true.

Syntax

Unit syntax:

```
ova_forbid_bool
#(edge_expr, msg, severity, category)
instance_name(expr, clk);
```

Template syntax:

```
forbid_bool(expr, msg, clk, severity, category);
```

Arguments

`expr`
    Signal being tested.

Report Message

In report messages, the assertion name is `cb`.

Examples

The following examples ensure that `sig` is never true.

Unit:

```
bind module dut : ova_forbid_bool
  #( , "sig is true.")
  (sig, sysclk);
```

Template:

```
forbid_bool(sig, "sig is true.", posedge sysclk);
```

ova_hold

Checks that the value of the signal being tested (*exp*) remains constant for the minimum (*min* +1) to maximum (max) number of cycles. Both the minimum and maximum specifications default to zero, meaning that the signal being tested (*exp*) is to change on every clock cycle. A new check begins every time the signal being tested (*exp*) changes. If the maximum number of cycles (*max*) is specified, then the signal being tested (*exp*) must change within *min* +1 to *max* +1 cycles.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_hold
#(min, max, bw, edge_expr, msg, severity,
 category, coverage_level)
instance_name (en, clk, exp);
```

Template syntax:

```
hold(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`

The minimum number of clock cycles (minus one) to hold the value. Default = 0.

`max`

The maximum number of clock cycles (minus one) to hold the value. Default = 0. For an open-ended interval [min ..], set *max* equal to zero and *min* greater than zero.

`bw`

The number of bits in the signal being tested (*`exp`*). Default = 1.

`clk`

The clock signal on which inputs are sampled and the checks are performed. In unit syntax, give just the clock signal. The active edge is specified with *`edge_expr`*. In template syntax, use the full edge expression, as in `posedge m_clk`.

`exp`

Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_hold`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_num_of_exp_changes` indicates the number of times `exp` changed value.

Cover `cover_num_of_matches` indicates the number of matches of `exp` changing value within the specified interval.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_num_of_matches_exactly_on_min` indicates the number of times `exp` changed exactly `min` clock cycles.

Cover `cover_num_of_matches_exactly_on_max` indicates the number of times `exp` changed exactly `max` clock cycles.

Examples

In the examples below, `wen` must be held for at least 3 cycles and is dropped after 7. Coverage Levels 1 and 3 are enabled (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_hold
  #(3, 7, , , "wen hold time error.", , ,5)
 hold_wen (en, clk, wen);
```

Template:

```
`define WRITE_OP 4'b1000
bool en : (rst_n && (memop === `WRITE_OP));
hold hold_wen(rst_n, posedge clk, wen, 3, 7,
     "wen hold time error.");
```

ova_hold_value

Checks that the signal being tested (*exp*) remains the value to hold (value) from the specified minimum (*min*) to maximum (*max*) number of cycles. That is, it must stay at *value* for *min* cycles, then it may change. After *max* cycles it must change to some other value.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_hold_value
#(min, max, bw, edge_expr, msg, severity,
  category, coverage_level)
instance_name (en, clk, exp, value);
```

Template syntax:

```
hold_value(en, clk, exp, value, min, max, msg, severity,
category);
```

Arguments

`min`

Minimum number of clock cycles to hold the value. Default = 0.

`max`

Maximum number of clock cycles to hold the value. Default = 0.

For an open-ended interval [min ..], set max equal to zero and *min* to greater than zero.

`bw`

The number of bits in the signal being tested (*exp*). Default = 2.

`exp`

Signal being tested.

`value`

Value to hold.

The check is triggered by a value change of the signal being tested (*`exp`*) to *`value`*. Both the minimum (*`min`*) and maximum (*`max`*) default to zero, meaning that *`exp`* is to equal *`value`* for one clock cycle.

Report Message

In report messages, the assertion name is `ova_c_hold_value`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_exp_change` indicates the number of times `exp` changed to `value`.

Cover `cover_num_of_matches` indicates the number of matches of `exp` holding `value` within the specified interval.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_hold_value_exactly_for_min` indicates the number of times `exp` held `value` exactly for `min` clock cycles.

Cover `cover_hold_value_exactly_for_min` indicates the number of times `exp` held value exactly for `max` clock cycles.

Examples

The following examples specify that the address must be held for at least 3 cycles, but not more than 5 cycles after it has the value 4'hff00. Coverage Level 1 is enabled (`coverage_level` = 1).

Unit:

```
bind module dut : ova_hold_value
  #(3, 5, 16, , , , ,1)
hold_address (en, clk, bus_addr[15:0], 4'hff00);
```

Template:

```
`define WRITE_OP 4'b1000
bool en : (rst_n && (memop === `WRITE_OP));
hold_value hold_address(en, posedge clk, addr,
     bus_addr[16:0], 3, 5);
```

ova_inc

Checks that when the signal being tested (*exp*) changes value, the new value is always between the specified minimum (*min*) and maximum (*max*) more than the previous value.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_inc
#(min, max, bw, edge_expr, msg, severity, category)
(en, clk, exp);
```

Template syntax:

```
inc(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`
: The minimum change in value. Default = 1.

`max`
: The maximum change in value. Default = 1.

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_inc`.

Examples

The following examples ensure `counter` is always incremented by 1 or 2.

Unit:

```
bind module dut : ova_inc check_counter
  #(1, 2, 8, , "Counter incremented wrong.")
  (en, clk, count);
```

Template:

```
inc check_counter(en, posedge clk, count, 1, 2,
     "Counter incremented wrong.");
```

ova_memory

Checks the integrity of synchronous memory contents and accesses.

Syntax

Unit syntax:

```
ova_memory
#(data_bits, addr_bits, mem_sz, addr_chk, init_chk,
  conflict_chk, pass_thru, read1_chk, write1_chk,
  value_chk, w_edge_expr, r_edge_expr, msg,
  severity, category, coverage_level)
 instance_name (start_addr, end_addr, ren, raddr, rclk,
                rdata, wen, waddr, wclk, wdata);
```

Template syntax:

```
memory(data_bits, addr_bits, start_addr, end_addr, mem_sz,
ren, raddr, rclk, rdata, wen, waddr, wclk, wdata, addr_chk,
init_chk, conflict_chk, pass_thru, read1_chk, write1_chk,
value_chk, msg, severity, category);
```

Arguments

`data_bits`
: Number of bits in the data. Default = 1.

`addr_bits`
: Number of bits in the addresses. Default = 1.

`mem_sz`
: The number of words in the memory. Default = 2. The *`end_addr`* - *`start_addr`* + 1 must be less than or equal to *`mem_sz`*.

`addr_chk`
: If 1, checks that address is valid. Default = 1.

`init_chk`

If 1, checks that addresses read have been previously written. Default = 1.

`conflict_chk`

If 1, checks that simultaneous reading and writing of the same address does not occur. Default = 0.

This check should only be enabled when *`rclk`* == *`wclk`*. When the two clocks are different the conflict check does not make much sense.

`pass_thru`

Specifies behavior when read and write happen at the same time on the same address. If 0, read gets the old data before the write. If 1, read gets the new data after the write. Default = 0.

`read1_chk`

If 1, checks that an address has at most one read between writes. Default = 0.

`write1_chk`

If 1, checks that an address is read at least once before it is over-written. Default = 0.

`value_chk`

If 1, checks that the value read from an address is the value that was written to that address. Default = 0.

`w_edge_expr`

The active clock edge of *`wclk`*. Default = 0.

`r_edge_expr`

The active clock edge of *`rclk`*. Default = 0.

`start_addr`

Starting address of the memory.

`end_addr`
Ending address of the memory.

`ren`
Read enable.

`raddr`
Read address.

`rclk`
Read clock.

`rdata`
Read data.

`wen`
Write enable.

`waddr`
Write address.

`wclk`
Write clock.

`wdata`
Write data.

When *`addr_chk`* evaluates true, ensures that *`start_addr`* ≤ *`raddr`* ≤ *`end_addr`* when `ren` is true, and that *`start_addr`* ≤ *`waddr`* ≤ *`end_addr`* when *`wen`* is true. All other checks apply only if the address is valid. Therefore, we recommend that *`addr_chk`* be enabled.

When *init_chk* evaluates true, ensures that addresses read have been previously written.

When *value_chk* evaluates true, ensures that the value read from an address is the value that was written to that address.

A read/write conflict occurs when a read operation occurs simultaneous with a write operation on the same address. When a conflict occurs, a read is assumed to happen before a write in the same cycle.

When *conflict_chk* evaluates true, ensures that simultaneous reading and writing of the same address does not occur. This check should only be enabled when *rclk* == *wclk*. When the two clocks are different the conflict check does not make much sense.

The *pass_thru* specification defines the behavior of the memory when a read and write occur simultaneously on the same address. When *pass_thru* = 0 then the read operation obtains the old data (before the write takes place). If *pass_thru* = 1, then the read operation gets the new value written in memory. Note that this option has effect only when *value_chk* or *init_chk* are enabled. Furthermore, *pass_thru* should *only* be enabled when *rclk* = *wclk* and *conflict_chk* = 0.

When <read1_chk> evaluates true, ensures that an address has at most one read in between writes.

When *write1_chk* evaluates true, ensures that an address is read at least once before it is over-written by different data.

Separate read and write port RAMs are naturally supported. For single-port R/W RAMs, simply associate the same actuals with the appropriate parameters. For single clock synchronous RAMs, provide the same clock edge parameter for both *`rclk`* and *`wclk`*.

The parameters *`data_bits`* = number of bits in the data, *`addr_bits`* = number of bits in the addresses, *`start_addr`* = first address, *`end_addr`* = last address, and *`mem_sz`* = number of words, are compile-time constant values which describe the layout of the memory. Note that *`end_addr`* - *`start_addr`* + 1 must be less than or equal to *`mem_sz`*.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_mem_init` checks that the memory address is initialized before reading or that there is a simultaneous read and write with pass-through allowed. When the assertion fails, the "offending" expression in the message might include "`ova_v_mem_addr_init [temp_raddr]`". This is an OVA variable holding a record of which addresses have been written.
- Assertion `ova_c_mem_waddr_chk` checks that the write address is within the limits when *`wen`* is asserted.
- Assertion `ova_c_mem_raddr_chk` checks that the read address is within the limits when *`ren`* is asserted.
- Assertion `ova_c_mem_conflict_chk` checks that, when *`conflict_check`* is asserted and there is a simultaneous read and write, the addresses are not the same.

- Assertion `ova_c_mem_read1_chk` checks that when *read1_chk* is asserted, the address is read only once before being overwritten. When the assertion fails, the “offending” expression in the message might include “`ova_v_mem_read1_flags[temp_raddr]`” and “`ova_v_mem_write1_flags[temp_raddr]`”. These are OVA variables holding a record of which addresses have been read and written.

- Assertion `ova_c_mem_write1_chk` checks that, when *`write1_chk`* is asserted, the address is read at least once before being overwritten. When the assertion fails, the “offending” expression in the message might include “`Ova_v_mem_addr_init[temp_waddr]`”, “`ova_v_mem_read1_flags[temp_waddr]`”, and “`ova_v_mem_write1_flags[temp_waddr]`”. These are OVA variables holding a record of which addresses have been read and written.

- Assertion `ova_c_mem_val_chk` checks the behavior of the memory when a read and write occur simultaneously on the same address. The check is according to the value of *`pass_thru`*. When the assertion fails and *`pass_thru`* is 0, the “offending” expression in the message might include “`ova_v_mem_mirror[temp_raddr]`”. This is an OVA variable holding a mirror of what is written in the memory (the old data).

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_number_of_reads` indicates the number of read operations to any address.

Cover `cover_number_of_writes` indicates the number of write operations to any address.

Cover `write_followed_by_read` indicates how many times a write was followed by a read to the same address.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_two_or_more_writes_without_intervening_read` indicates how many times two writes occurred to the same (any) address without an intervening read operation to that address.

Cover `cover_two_or_more_reads_without_intervening_write` indicates how many times two reads occurred to the same (any) address without an intervening write operation to that address.

Cover `simultaneous_read_and_write_to_same_addr` indicates how many times (quasi)simultaneous read and write operations occurred to the same (any) address as seen by the read clock `rclk`.

Cover `simultaneous_read_and_write_to_different_addr` how many times (quasi)simultaneous read and write operations occurred to the different addresses as seen by the read clock `rclk`.

Cover `read_to_start_addr` indicates how many read operations occurred to the address `start_addr`.

Cover `write_to_start_addr` indicates how many write operations occurred to the address `start_addr`.

Cover *`read_to_end_addr`* indicates how many read operations occurred to the address `end_addr`.

Cover `write_to_end_addr` indicates how many write operations occurred to the address `end_addr`.

Cover `write_followed_by_read_to_start_addr` indicates how many write operations were followed by a read to the address `start_addr`.

Cover `write_followed_by_read_to_end_addr` indicates how many write operations were followed by a read to the address `end_addr`.

Examples

The following examples do the following:

- Memory accesses are checked with data and address width of 16 bits
- The low address bound is 0
- The high address bound is 16’h0fff
- The memory size is 2**12 = 16’h1000
- The same clock is used for read and write,
- The *`addr_chk`* and *`init_chk`* specifications are enabled by default
- The *`conflict_chk`* and *`pass_thru`* specifications are enabled
- The *`read1_chk`* and *`write1_chk`* specifications are disabled by default
- The *`value_chk`* is enabled.

- Coverage Levels 1 and 3 are enabled in the unit instance (*coverage_level* = 5).

Unit:

```
bind module dut : ova_memory
  #(16, 16, 16’h1000, , , 1, 1, , , 1, , , , , , 5)
 mem_inst(16’h0000, 16’h0fff, ren, addr, clk, rdata,
           wen, addr, clk, wdata);
```

Template:

```
memory(16, 16, 16’h0000, 16’h0fff, 16’h1000, ren,
  addr, posedge clk, rdata, wen, addr, posedge clk,
  wdata, , , 1, 1, , , 1);
```

ova_memory_async

Checks the integrity of asynchronous memory contents and accesses.

Syntax

Unit syntax:

```
ova_memory_async
#(data_bits, addr_bits, mem_sz, addr_chk, init_chk,
read1_chk, write1_chk, value_chk, msg, severity, category)
(start_addr, end_addr, ren, raddr, rdata, wen, waddr, wdata);
```

Template syntax:

```
memory_async(data_bits, addr_bits, start_addr, end_addr,
mem_sz, ren, raddr, rdata, wen, waddr, wdata, addr_chk,
init_chk, read1_chk, write1_chk, value_chk, msg, severity,
category);
```

Arguments

`data_bits`
: Number of bits in the data. Default = 1.

`addr_bits`
: Number of bits in the addresses. Default = 1.

`mem_sz`
: The number of words in the memory. Default = 2. Note that *`end_addr`* - *`start_addr`* + 1 must be less than or equal to *`mem_sz`*.

`addr_chk`
: If 1, checks that address is valid. Default = 1.

`init_chk`

If 1, checks that addresses read have been previously written. Default = 1.

`read1_chk`
: If 1, checks that an address has at most one read between writes. Default = 0.

`write1_chk`
: If 1, checks that an address is read at least once before it is over-written. Default = 0.

`value_chk`
: If 1, checks that the value read from an address is the value that was written to that address. Default = 0.

`start_addr`
: Starting address of the memory.

`end_addr`
: Ending address of the memory.

`ren`
: Read enable.

`raddr`
: Read address.

`rdata`
: Read data.

`wen`
: Write enable.

`waddr`
: Write address.

`wdata`
Write data.

When *`addr_chk`* evaluates true, ensures that *`start_addr`* ≤ *`raddr`* ≤ *`end_addr`* as sampled by the negedge of *`ren`*, and that *`start_addr`* ≤ *`waddr`* ≤ *`end_addr`* as sampled by the negedge of *`wen`*. There is thus no clock other than the *`ren`* and *`wen`* signals that indicate when each operation is to take place by their falling edges. If negative *`ren`* and *`wen`* assert is desired, just pass a complement of the design signals as the actual arguments.

Note:
All sampling is done “before” the *`ren`* and *`wen`* falling edges at t-1.

All other checks apply only if the address is valid. Therefore, we recommend that *`addr_chk`* be enabled.

When *`init_chk`* evaluates true, ensures that addresses read have been previously written.

When *`value_chk`* evaluates true, ensures that the value read from an address is the value that was written to that address.

A read/write conflict occurs when a read operation occurs simultaneously with a write operation on the same address. When a conflict occurs, a read is assumed to happen before a write in the same cycle. There is however no conflict check, because in asynchronous read and write this condition is bound to happen and the check is meaningless. If the exclusion is actually required because *`ren`* and *`wen`* are derived from some synchronous signals, the mutual exclusion over these signals can be verified separately using the ova_mutex checker, for instance.

When *`read1_chk`* evaluates true, ensures that an address has at most one read in between writes.

When *write1_chk* evaluates true, ensures that an address is read at least once before it is over-written by different data. A write is considered to have occurred even when the value did not change.

Separate read and write port RAMs are naturally supported. For single port R/W RAMs, simply associated the same actuals with the appropriate parameters.

The parameters *data_bits* = number of bits in the data, *addr_bits* = number of bits in the addresses, *start_addr* = first address, *end_addr* = last address, and *mem_sz* = number of words, are compile-time constant values which describe the layout of the memory. Note that *end_addr* - *start_addr* +1 must be less or equal to *mem_sz*.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_mem_async_init` checks that the memory address is initialized before reading. When the assertion fails, the "offending" expression in the message might include "`ova_v_mem_async_addr_init[temp_raddr]`". This is an OVA variable holding a record of which addresses have been written. This assertion is active only if *init_chk* is 1.
- Assertion `ova_c_mem_async_waddr_chk` checks that the write address is within the limits just before negedge of *wen*. This assertion is active only if *addr_chk* is 1.
- Assertion `ova_c_mem_async_raddr_chk` checks that the read address is within the limits just before negedge of *ren*. This assertion is active only if *addr_chk* is 1.

- Assertion `ova_c_mem_read1_chk` checks that the address is read no more than once before being overwritten. When the assertion fails, the “offending” expression in the message might include “`ova_v_mem_async_read1_flags[temp_raddr]`” and “`ova_v_mem_async_write1_flags[temp_raddr]`”. These are OVA variables holding a record of which addresses have been read and written. This assertion is active only if *`read1_chk`* is 1.
- Assertion `ova_c_mem_async_write1_chk` checks that the address is read at least once before being overwritten. When the assertion fails, the “offending” expression in the message might include “`ova_mem_async_addr_init[temp_waddr]`”, “`ova_v_mem_async_read1_flags[temp_waddr]`”, and “`ova_v_mem_async_write1_flags[temp_waddr]`”. These are OVA variables holding a record of which addresses have been read and written. This assertion is active only if *`write1_chk`* is 1.
- Assertion `ova_c_mem_async_val_chk` checks that the value read from an address is the value that was written to that address. When the assertion fails, the “offending” expression in the message might include “`ova_v_mem_async_mirror[temp_raddr]`”. This is an OVA variable holding a mirror of what is written in the memory. This assertion is active only if *`value_chk`* is 1.

Examples

The examples below specify the following:

- Memory accesses are to be checked with data and address width of 16 bits
- The low address bound is 0
- The high address bound is 16’h0fff

- The memory size is 2**12 = 16’h1000
- *addr_chk* and *init_chk* are enabled by default
- *read1_chk* and *write1_chk* are enabled
- *value_chk* is enabled.

Unit:

```
bind module dut : ova_memory_async
  #(16, 16, , , 1, 1, 1)
  (16’h0000, 16’h0fff, 16’h1000, ren, addr, rdata,
    wen, addr, wdata);
```

Template:

```
memory_async(16, 16, 16’h0000, 16’h0fff, 16’h1000,
     ren, addr, rdata, wen, addr, wdata, , , 1, 1, 1);
```

ova_multiport_fifo

Implements a checker for a single-clock, multi-port in- and multi-port out queue.

Unit Syntax:

```
ova_multiport_fifo
#(depth, elem_sz, no_ports, hi_water_mark, enq_lat, deq_lat,
oflow_chk, uflow_chk, value_chk, pass_thru, edge_expr, msg,
severity, category)
(reset, clk, enq, enq_data, deq, deq_data);
```

Template Syntax:

```
multiport_fifo(reset, clk, depth, hi_water_mark, elem_sz,
no_ports, enq, enq_lat, enq_data, deq, deq_lat, deq_data,
oflow_chk, uflow_chk, value_chk, pass_thru, msg, severity,
category);
```

`reset`

Asserted 1 initializes the queue to empty. All operations are synchronous to *`clk`* ticks, including reset.

`enq and deq`

Bit vectors of equal size *`no_ports`*. Each pair of corresponding bits in these vectors defines the enqueue and dequeue signals for a port. Their priority is such that bit 0 is the lowest priority and the highest order bit, *`no_ports`*-1, is the highest priority. That is, the enqueue port and the dequeue port of the highest priority are processed at every *`clk`* tick.

`enq_data`

A 2-D array of data. It is assumed that it is dimensioned as `[elem_size-1:0] enq_data [0:no_ports-1]`. Any time a bit in *`enq`* is asserted 1, the corresponding data element in *`enq_dat`* must be valid after *`enq_lat`* clock cycles. Only the highest priority data is actually enqueued.

`enq_lat`

A compile-time, non-negative integer constant that indicates the number of cycles between *`enq`* being asserted 1 and *`enq_data`* being valid in the corresponding position.

`oflow_chk`

When a *`enq`* is asserted 1: If *`oflow_chk`* evaluates true, ensures that queue does not overflow the maximum size given in *`depth`*. The *`depth`* can be at most 2**16.

`hi_water_mark`

If *`hi_water_mark`* is a positive value, then the depth of the queue after enqueue will be checked to see if *`hi_water_mark`* is reached. Once high water has been exceeded once, this check is disabled until the FIFO size falls below the mark again. If *`hi_water_mark`* = 0 then the high-water mark check is disabled and only overflow is checked when the depth of the queue is exceeded (provided that *`oflow_chk`* = 1).

`deq_data`

A 2-D array of data. It is assumed that it is dimensioned as `[elem_size-1:0] deq_data [0:no_ports-1]`. Whenever a bit in deq is asserted 1, the corresponding data value must be available on *`deq_data`* in the corresponding position delayed *`deq_lat`* clock cycles. The number of elements in *`deq_data`* a nd *`enq_data`* must be the same as the number of bits in *`enq`* and *`deq`*.

`deq_lat`

A compile-time non-negative integer constant that indicates the number of cycles between when *deq* is asserted and *deq_data* is valid.

`uflow_chk`

If this evaluates true, it ensures that the queue is not empty (underflow) when a *deq* bit is asserted. If a dequeue on empty is detected then the check is disabled until the next enqueue operation.

`value_chk`

If this evaluates true, it ensures *deq_data* as selected by the same position as the highest priority The *deq* bit is the same as that at the head of the queue.

`pass_thru`

If an enqueue and dequeue operation happens simultaneously on an empty queue, then the behavior depends on the *pass_thru* argument to the checker instance (it must be a compile-time constant).

If *pass_thru* = 0 then the dequeue happens before enqueue, hence the empty condition is detected and reported, and an underflow (provided that *uflow_chk* = 1). If *value_chk* = 1 then the value check fails.

If *pass_thru* = 1 then it is assumed that enqueue happens first and the data is immediately dequeued and compared with *deq_data* if *value_chk* is enabled. Also, there is no underflow error reported.

If an enqueue and dequeue operation happens simultaneously on a full queue then no overflow is reported and the new element is enqueued while the element at the head of the queue is dequeued without changing the size of the queue.

Defaults

`depth  = 2`

Maximum number of elements in the queue is 2.

`elem_sz  = 1, no_ports  = 2`

Number of channels or ports.

`hi_water_mark  = 0`

High-water mark check disabled.

`oflow_chk  = 1`

Overflow check enabled.

`uflow_chk  = 1`

Underflow check enabled.

`value_chk  = 1`

Checking of dequeued data enabled.

`pass_thru  = 0`

Simultaneous enqueue and dequeue on empty reports underflow.

`enq_lat  = 0`

Enqueue data at the same time as <enq> asserted.

`deq_lat  = 0`

Dequeue data checked at the same time as `deq` asserted.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_f_fifo_overflow` checks that `enq` is not issued while the FIFO is full.

- Assertion `ova_f_fifo_underflow` checks that when the deq command is issued, the FIFO is not empty.
- Assertion `ova_c_fifo_value_chk` checks that the integrity of values coming off the queue. This assertion is active only if *`value_chk`* is 1.
- Assertion `ova_f_fifo_hi_water_chk` checks that the FIFO is not filled above the high-water mark.

Example

The following examples check that the FIFO is initialized anytime reset is 1 (synchronously with posedge `clk`), there are up to 10 elements in the FIFO, *`high_water_mark`* is by default 0 (disabled), the size of the data is 16 bits, there are two channels (enqueue and dequeue are 2 bits wide, *`data_in`* and *`data_out`* are arrays of two 16-bit words), `data_in[i]` is enqueued when `enqueue[i]` is 1 with no latency, `data_out[i]` must be equal to that at the head of the fifo when `dequeue[i]` is 1 with no latency, and overflow, underflow and value checks are enabled with pass through when empty allowed.

Unit:

```
bind module dut : ova_multiport_fifo
  #(10, 16, , , , , , , , 1)
  (reset, clk, enqueue, data_in, dequeue, data_out);
```

Template:

```
multiport_fifo(reset, posedge clk, 10, ,16 , ,
  enqueue, , data_in, dequeue, , data_out, , , , 1 );
```

ova_mutex

Checks that the specified signals (`a`) and (`b`) never evaluate true at the same time.

Syntax

Unit syntax:

```
ova_mutex
#(edge_expr, msg, severity, category, coverage_level)
 instance_name(en, clk, a, b);
```

Template syntax:

```
mutex(en, clk, a, b, msg, severity, category);
```

Arguments

`a`

First signal being tested.

`b`

Second signal being tested.

Report Message

In report messages, the assertion name is `ova_c_mutex`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_changes_on_a` indicates how many times `a` changed value.

Cover `cover_changes_on_b` indicates how many times `b` changed value.

Examples

The following examples ensure that reading and writing are not enabled at the same time. The check is always enabled by default. Coverage Level 1 is enabled in the unit instance (*coverage_level* = 1).

Unit:

```
bind module dut : ova_mutex read_write
  #(, "Read and write enables asserted at same time.", , , 1)
 mutex_inst (1'b1, clk, ren, wen);
```

Template:

```
mutex read_write( , posedge clk, ren, wen,
     "Read and write enables asserted at same time.");
```

ova_next_state

Checks that when the signal being tested (*exp*) is in the specified current state (*cs*) it will transition to one of the specified legal next states.

Syntax

Unit syntax:

```
ova_next_state
#(no_ns, width, min_hold, max_hold, disallow, edge_expr,
  msg, severity, category, coverage_level)
 instance_name (en, clk, exp, cs, ns);
```

Template syntax:

```
next_state(en, clk, exp, cs, no_ns, ns, min_hold, max_hold,
disallow, msg, severity, category);
```

Arguments

`no_ns`
: The number of legal next states possible from the specified current state (*cs*). Default = 1.

`width`
: The number of bits in the signal being tested (*exp*), the current state (*cs*), and each element of a bitvector of the concatenated legal state values (*ns*). The vector is `ns`[`width` * `no_ns`- 1:0]. Default = 1.

`min_hold`
: The minimum number of clock ticks the signal being tested (*exp*) must hold at the current state (*cs*) value. Default = 1.

`max_hold`

The maximum number of clock ticks the signal being tested (*exp*) can hold at the the current state (*cs*) value. Default = 0.

`disallow`

If 1, checks that the signal being tested (*exp*) does *not* transition to any of the values in a specified array of legal states (*ns*). Default = 0. If 0, the checker makes sure that a transition to one of the states occurs.

`exp`

Signal being tested.

`cs`

The current state. The check starts when *exp* = *cs* (and *en* = 1).

`ns`

A bitvecor of concatenated legal states that *exp* can transition to from *cs*.

Note the following:

- *max_hold* indicates the maximum number of clock ticks the signal being tested (*exp*) may hold at the current state (*cs*) value.
- *min_hold* = *max_hold* = 1 indicates that the signal being tested (*exp*) changes to the next value on the next clock tick.
- *min_hold* = m, *max_hold* = 0 indicates that the signal being tested (*exp*) must hold the current state value (*cs*) for at least m clock ticks, no upper bound on when it must change.
- *min_hold* = m, *max_hold* = n indicates that the signal being tested (*exp*) must hold the current state value (*cs*) for at least m clock ticks and must advance to the next value within n ticks.

Report Message

This checker uses two assertions to cover different modes:

- Assertion `ova_c_next_state` checks for a transition to an allowed state during the hold time. This assertion is active only if *`disallow`* is 0.
- Assertion `ova_f_next_state` checks for no transition to a disallowed state during the hold time. This assertion is active only if *`disallow`* is 1.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

When `disallow` == 0:

Cover `cover_exp_state_transitions` indicates how many times a valid transition to a state in `ns` occurred.

When `disallow` == 1:

Cover `cover_exp_state_transitions` indicates how many times a transition to a state other than those in `ns` occurred.

`Level_3 (bit 2 set in coverage_level)`
`Exists only when disallow == 0`

Cover `cover_exp_changes_to_ns[i]` indicates how many times there was a transition from state `cs` to state `ns[i]`.

Examples

The following examples verify that when *`state_var`* takes on value 0, then on the next posedge clk, *`state_var`* takes on one of the values 0, 2, or 4. Coverage levels 1 and 3 are enabled in the unit (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_next_state
  #(3, 4, 1, 1, 0, , , , , 5)
 next_inst (!reset, clk, state_var, 0,
             {12'b0100_0010_0000});
```

Template:

```
next_state(!reset, posedge clk, state_var, 0, 3,
     {12'b0100_0010_0000}, 1, 1, 0, );
```

ova_no_contention

Checks that bus signal being tested (*bus*) always has a single active driver and that there is no X or Z on the bus when driven. That is, that *en_vector* is not zero. The total number of *en_vector* bits that are asserted can be at most 1. Specify 0 for no minimum bus quiet time between bus transactions.

Syntax

Unit syntax:

```
ova_no_contention
#(min_quiet, max_quiet, bw_en, bw_bus, edge_expr, msg,
  severity, category, coverage_level)
 instance_name (en, clk, en_vector, bus);
```

Template syntax:

```
no_contention(en, en_vector, clk, bus, min_quiet, max_quiet,
msg, severity, category);
```

Arguments

`min_quiet`
: The minimum number of clock cycles between bus transactions. Default = 0.

`max_quiet`
: The maximum number of clock cycles between bus transactions. Default = 0.

`bw_en`
: The number of bits in *en_vector*. Default = 2.

`bw_bus`
: The number of bits in the bus signal being tested (*bus*). Default = 2.

`en_vector`
: Enable signals for bus drivers as a vector.

`bus`
: Bus signal being tested.

Report Message

This checker uses three assertions to cover different aspects:

- Assertion `ova_c_no_xs` checks that the bus does not have an x or z on any of its bits while at least one driver is enabled.
- Assertion `ova_c_1_driver` checks that there is no more than one driver enabled.
- Assertion `ova_c_quiet_time` checks that all drivers are disabled for the specified quiet time followed by only one driver being enabled.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover `cover_driver_enable` indicates how many times bit `en_vector[i]` was set to 1 (enabled).

`Level_3 (bit 2 set in coverage_level)`
: Cover `cover_no_contention_quiet_time_equal_to_min_quiet` indicates how many times the observed quiet time is exactly equal to the specified `min` value.

  Cover `cover_no_contention_quiet_time_equal_to_max_quiet` indicates how many times the observed quiet time is exactly equal to the specified `max` value.

Examples

The following examples ensure that the bus is not multiply driven; it must have 2 cycles of quiet time in between transactions, but no more than 100 cycles of uninterrupted quiet time. Coverage Level 1 is enabled in the unit instance (`coverage_level` = 1).

Unit:

```
bind module dut : ova_no_contention
  #(2, 100, 8, 8, , , , , 1)
 chip_bus_check
   (1'b1, clk, {chp0_oe, chp1_oe, chp2_oe, chp3_oe},
    data_bus);
```

Template:

```
no_contention chip_bus_check(
     {chp0_oe, chp1_oe, chp2_oe, chp3_oe},
     posedge clk, data_bus, 2, 100);
```

ova_odd_parity

Checks that the value of the signal being tested (*exp*) always has an odd number of bits set to 1. Usually the signal (*exp*) is formed by concatenating the data and parity bits.

Syntax

Unit syntax:

```
ova_odd_parity
#(bw, edge_expr, msg, severity, category)
(en, clk, exp);
```

Template syntax:

```
odd_parity(en, clk, exp, msg, severity, category);
```

Arguments

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_odd_parity`.

Examples

The following examples ensure data read from memory has odd parity. `mdata` contains the data bits and the parity bit.

Unit:

```
bind module dut : ova_odd_parity memory_data
```

```
  #(9, , "Parity error on mdata.")
  (rst_n, clk, mdata);
```

Template:

```
odd_parity memory_data(rst_n, posedge clk, mdata,
     "Parity error on mdata.");
```

ova_one_cold

Checks that only one bit is set to zero or, optionally, that all bits are set to 1 in the state value.

Syntax

Unit syntax:

```
ova_one_cold
#(strict, bw, edge_expr, msg, severity, category)
(en, clk, state);
```

Template syntax:

```
one_cold(en, clk, state, strict, msg, severity, category);
```

Arguments

`strict`

If 1, checks for a strict one-cold state encoding. Default = 0.

The check fails if all bits are set to one.

`bw`

The number of bits in the signal being tested (*`state`*). Default = 2.

`state`

Signal being tested. The signal must be more than one bit wide.

Report Message

In report messages, the assertion name is `ova_c_one_cold`.

Examples

The following examples ensure state is a strict one cold (always 1 bit that is set to zero).

Unit:

```
bind module dut : ova_one_cold assert_one_cold
  #(1, 8, , "state does not have a one-cold value.")
  (rst_n, clk, state);
```

Template:

```
one_cold assert_one_cold(rst_n, posedge clk, state,
    1, "state does not have a one-cold value.");
```

ova_one_hot

Checks that only one bit is set to one or, optionally, that all bits are set to zero in the state value.

Syntax

Unit syntax:

```
ova_one_hot
#(strict, bw, edge_expr, msg, severity,
  category, coverage_level)
 instance_name (en, clk, state);
```

Template syntax:

```
one_hot(en, clk, state, strict, msg, severity, category);
```

Arguments

`strict`
: If 1, checks for a strict one-hot state encoding. Default = 0.

  The check fails if all bits are set to zero.

`bw`
: The number of bits in the signal being tested (*`state`*). Default = 2.

`state`
: Signal being tested. The signal must be more than one bit wide.

Report Message

In report messages, the assertion name is `ova_c_one_hot`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_state_change` indicates how many times `test_expr` changed value.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_state_bit_is_1[i]` indicates how many times bit `i` was 1 after a change of value of `state[i]`.

Examples

The following examples ensure state is one hot (can have a state with no bits asserted). Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 1).

Unit:

```
bind module dut : ova_one_hot
  #(0, 8, , "state is not a one-hot or all-0 value.", , , 5)
 hot_inst (rst_n, clk, state);
```

Template:

```
one_hot hot_inst(rst_n, posedge clk, state, 0,
     "state does not have a one-hot value.");
```

ova_overflow

Checks that the signal being tested (*exp*) does not transition from ≥ *max* to ≤ *min*. Works only with unsigned vector values.

Syntax

Unit syntax:

```
ova_overflow
#(min, max, bw, edge_expr, msg, severity,
  category, coverage_level)
instance_name (en, clk, exp);
```

Template syntax:

```
overflow(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`
: The minimum value allowed. Default = 0.

`max`
: The maximum value allowed. Default = 1.

`bw`
: The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_overflow`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_exp_change` indicates how many times `exp` changed value.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_exp_reached_min` indicates how many times exp reached the `min` value.

Cover `cover_exp_reached_max` indicates how many times exp reached the `max` value.

Examples

The following examples check that the value of an 8-bit counter does not go from FF to 0 unless there is a reset. Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 1).

Unit:

```
bind module dut : ova_overflow
  #(0, 8'hff, 8, , , , , 5)
 counter_oflo_inst (rst_n, clk, counter);
```

Template:

```
overflow assert_cntr_no_overflow(rst_n, posedge clk,
  counter, 0, 8'hff);
```

ova_quiescent_state

Checks that when *eos_exp* evaluates true, *exp* has value of *fstate*.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_quiescent_state
#(bw, edge_expr, msg, severity, category)
(en, clk, exp, fstate, eos_exp);
```

Template syntax:

```
quiescent_state(en, clk, exp, fstate, eos_exp, msg);
```

Arguments

`bw`
: The number of bits in the signal being tested (*exp*) and the state to match (*fstate*). Default = 2.

`exp`
: Signal being tested.

`fstate`
: State to match.

`eos_exp`
: When true, signals that the state to match (*exp*) is in the state.

Report Message

In report messages, the assertion name is `ova_c_quiescent`.

Examples

The following examples specify that at the end of a long computation, an *`output_ready`* flag is asserted. At this time, we want to ensure that the chip's input ready flag is asserted.

Unit:

```
bind module dut : ova_quiescent_state computation_done
  #(1, , "Output ready but ready flag not set.")
  (rst_n, clk, ready, 1'b1, output_ready);
```

Template:

```
quiescent_state computation_done(rst_n,
     posedge clk, ready, 1'b1, output_ready,
     "Output ready but ready flag not set.");
```

ova_range

Checks that the signal being tested greater than or equal to the specified minimum value (*min*), and less than or equal to the specified maximum value (*max*). Works with signed or unsigned vector values, depending on the Verilog data type.

Syntax

Unit syntax:

```
ova_range
 #(bw, edge_expr, msg, severity, category, coverage_level)
 instance_name (en, clk, exp, min, max);
```

Template syntax:

```
range(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`bw`
: The number of bits in the signal being tested (*exp*), the minimum value allowed (*min*), and the maximum value allowed (*max*). Default = 1.

`exp`
: Signal being tested.

`min`
: The minimum value allowed.

`max`
: The maximum value allowed.

Report Message

In report messages, the assertion name is `ova_c_range`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_exp_change` indicates how many times `exp` changed value.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_exp_reached_min` indicates how many times exp reached the `min` value.

Cover `cover_exp_reached_max` indicates how many times exp reached the `max` value.

Examples

The following example ensure that `coeff` is between 2 and 6. Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 1).

Unit:

```
bind module dut : ova_range
  #(4, , "coeff out of range.", , , 5)
 filter_coeff_check (rst_n, clk, coeff, 4'd2, 4'd6);
```

Template:

```
range filter_coeff_check(rst_n, posedge clk, coeff,
     2, 6, "coeff out of range.");
```

ova_reg_loaded

Checks that the register being tested (*reg*) is loaded with source data (*src*). The value of the register (*reg*) is checked against a stored source value of (*src*) starting with a specified number of delay (*delay*) cycles (minimum of one, which is the default) after the *trigger* condition evaluates true and within the specified *end_cycle* cycles (defaults to one) after the *trigger* evaluates true or when the stop signal (*stop*) evaluates true (whichever occurs first). The default of the stop signal (*stop*) is zero. If used to control the end time, *stop* should become true at least one clock cycle after the register being tested (*reg*) is loaded, which means a minimum of two cycles after the *trigger* condition. The source data value (*src*) is “captured” when the *trigger* evaluates true. The check is made by comparing this saved source value (*src*) against the register contents (*reg*) during the load window. Once the register has loaded the value during the window, the check terminates with success.

If the window is terminated only by the stop signal (*stop*) (that is, there is no timeout and *end_cycle* does not apply), then set *end_cycle* to 0.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_reg_loaded
#(delay, end_cycle, bw, edge_expr, msg, severity, category)
(en, clk, trigger, src, dst_reg, stop);
```

Template syntax:

```
reg_loaded(en, clk, trigger, src, reg, delay, end_cycle,
stop, bw, msg, severity, category);
```

Arguments

`delay`

The number of cycles after the trigger signal (*`trigger`*) goes true to start the window. Default = 1.

`end_cycle`

The number cycles after the trigger signal (*`trigger`*) goes true to end the window. Default = 1.

`bw`

The number of bits in the source date (*`src`*) and the register under test (*`reg`*). Default = 2.

The width (*`bw`*) of *`src`* and *`reg`* should match their actual widths. If *`bw`* is less than the actual width, the check might fail if the bits above *`bw`* are not zero. If *`bw`* is more than the actual width, the additional bits are taken to be zero. A *`bw`* of zero is illegal.

`trigger`

Signal that is part of starting the window.

`src`

Data loaded into the register.

`reg`

Register being tested.

`stop`

Signal that stops the check.

Report Message

In report messages, the assertion name is `ova_c_reg_loaded`. When the assertion fails, the “offending” expression in the message might include “`ova_v_reg_src`”. This is an OVA variable that stores the value of the source data (*`src`*) when the triger signal (*`trigger`*) is true.

Examples

The examples below specity that the adder output must be loaded into the multiplier register within 2 cycles of the adder's output being ready.

Unit:

```
bind module dut : ova_reg_loaded sum_to_multiplier
  #(1, 2, 16)
  (rst_n, clk, adder_oe, sum, multiplier, 1'b0);
```

Template:

```
reg_loaded sum_becomes_multiplier(rst_n, posedge clk,
  adder_oe, sum, multiplier, 0, 2, 1'b0, 16);
```

ova_req_ack_unique

Verifies that each `req` receives an `ack` within the specified interval `min_time` and `max_time` clock `clk` ticks. Note that acks are attributed to reqs in a fifo manner.

Syntax

(Unit syntax only)

```
ova_req_ack_unique
#(min_time, max_time, max_time_log_2, edge_expr, 
msg,
  version, severity, category, coverage_level)
 instance_name (reset, clk, req, ack)
```

Arguments

`min_time`

Defines the minimum time separation between a req and an ack (default is 1).

`max_time`

Defines the maximum time separation between a req and an ack (default is 15).

`max_time_log_2`

Specifies the superior integer of log2 of *`max_time`*, used to dimension the data structures. The default is 4 (= sup(log2(15))).

`version`

This parameter specifies two versions of the checker:

- `==0` — Selects a version that is suitable for *`max_time`* <= 15. It uses IDs to identify requests and then generates as many assertions as the *`max_time`* clock ticks.
- `==1` — Selects a version that is suitable for *`max_time`* > 15. It uses a time stamp computed mod 2 *`max_time`* to mark the requests, the time stamp is enqueued. When an ack arrives it verifies that the time stamp at the head of the queue satisfies the timing requirements.

`reset`

Synchronous reset, active high (1), initializes all request history to nill.

`req` and *`ack`*

The signals of interest.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_number_of_req` indicates how many times `req` was asserted.

Cover `cover_number_of_ack` indicates how many times `ack` was asserted.

NOTE: Coverage at Level 3 is only available with `version` = 1.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_ack_with_exact_min_lat` indicates how many time the observed latency was exactly equal to the specified `min` value.

Cover `cover_ack_with_exact_max_lat` indicates how many time the observed latency was exactly equal to the specified `max` value.

Example

```
bind module mod_name: ova_req_ack_unique
         #( 20, 100, 7, 0, "req_ack failed", 1, , , 5)
     req_grant_1_1 (!reset_n, clk, request, grant);
```

In the example, the instance `req_grant_1_1` of `ova_req_ack_unique` verifies that for each request there is a grant received within 20 and 100 posedge edges of `clk`. The checker is reset on `reset_n` low. Version 1 is used, i.e., using time stamps. Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 1).

ova_req_requires

Checks that if the first expression in a sequence (*trig_req*) evaluates true, then the second (*follow_req*) and third (*follow_resp*) expressions in the sequence evaluate true before the last expression (*trig_resp*) evaluates true. The delay between the first expression in a sequence (*trig_req*) and the last expression *(trig_resp)* in the sequence is specified as a latency window via the *min_lat* and *max_lat* parameters.

Syntax

Unit syntax:

```
ova_req_requires
  #(min_lat, max_lat, edge_expr, msg, severity,
    category, coverage_level)
 instance_name
   (en, clk, trig_req, follow_req, follow_resp, trig_resp);
```

Template syntax:

```
req_requires(en, clk, trig_req, follow_req, follow_resp,
trig_resp, min_lat, max_lat, msg, severity, category);
```

Arguments

`min_lat`
: Minimum number of clock cycles between the first expression in a sequence (*trig_req*) going true and the last expression in the same sequence (*trig_resp*) going true. Default = 1.

`max_lat`
: Maximum number of clock cycles between the first expression in a sequence (*trig_req*) going true and the last expression in the same sequence (*trig_resp*) going true. Default = 0. For an open-ended interval [*min_lat* ..], set the *max_lat* equal to zero.

`trig_req`
: First signal in the sequence.

`follow_req`
: Second signal in the sequence.

`follow_resp`
: Third signal in the sequence.

`trig_resp`
: Last signal in the sequence.

Note:
: The defaults are *`min_lat`* = 1 and *`max_lat`* = 0, meaning that *`trig_resp`* must come at least one clock tick after *`trig_req`*.

Report Message

In report messages, the assertion name is `ova_c_req_requires`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover `cover_no_of_trig_reqs` indicates the number of times `trig_req` was asserted.

  Cover `cover_cover_req_requires` indicates how many times the specified sequence occurred.

`Level_2 (bit 1 set in coverage_level)`
: Cover `cover_trig_req_follow_req` indicates how many times there was a a `follow_req` after a `trig_req`.

  Cover `cover_trig_req_follow_req_follow_resp` indicates how many times there was a `follow_req` after a `trig_req` and then followed by `follow_resp`.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_trig_resp_exactly_on_min_lat` indicates how many times the observed latency between `trig_req` and `trig_resp` was exactly equal to the `min` value.

Cover `cover_trig_resp_exactly_on_max_lat` indicates how many times the observed latency between `trig_req` and `trig_resp` was exactly equal to the `max` value.

Note: Coverage is collected correctly only when the transactions delimited by `trig_req` and `trig_resp` asserted do not overlap, i.e., there is no new assertion of `trig_req` while such a transaction is in progress.

Examples

The following examples specify a request to transmit a packet can only be satisfied if the transmitter can be granted access to the bus. The packet must be sent within 10 cycles. Levels 1, 2 and 3 are enabled in the unit instance (`coverage_level` = 7).

Unit:

```
bind module dut : ova_req_requires
   #(1, 10, , , , , 7)
 xmit_packet (rst_n, clk, send_pkt[0], bus_req[1],
               bus_grant[1], pkt_ack[0]);
```

Template:

```
req_requires xmit_packet(rst_n, posedge clk,
  send_pkt[0], bus_req[1], bus_grant[1], pkt_ack[0],
     1, 10);
```

ova_req_resp

Checks that the rising edge of a bit in the vector of a request signal (*req*) is followed by a single rising edge of the corresponding bit of the vector of a response signal (*resp*) within the latency response window specified by the minimum (*min_lat*) and maximum number of clock cycles (*max_lat*). It is assumed that no new request is issued until after a response is received for the current request.

Syntax

Unit syntax:

```
ova_req_resp
#(no_chnl, min_lat, max_lat, resp_cycles, no_req4resp,
req_till_resp, req_drop_after_resp, edge_expr, msg,
severity, category)
(en, clk, req, resp);
```

Template syntax:

```
req_resp(en, clk, req, resp, no_chnl, min_lat, max_lat,
resp_cycles, no_req4resp, req_till_resp,
req_drop_after_resp, msg, severity, category);
```

Arguments

`no_chnl`

The number of bits in the vector of the request signal (*req)* and the vector of the response signal (*resp*.) Default = 1.

`min_lat`

Minimum number of clock cycles between a bit in the specified vector of a request signal (*req*) going true and a bit in the vector of a response signal (*resp*) going true. Default = 1.

`max_lat`

Maximum number of clock cycles between a bit in the specified vector of a request signal (*req*) going true and a bit in the vector of a response signal (*resp*) going true. Default = 1.

For an open-ended interval [*min_lat* ..], set the *max_lat* equal to zero.

`resp_cycles`

Number of clock cycles that the vector of a response signal (*resp*) must stay asserted. Default = 0. If zero or less, the duration is not checked.

`no_req4resp`

If 1, checks that each response has a corresponding request. Default = 0.

`req_till_resp`

If 1, checks that the request remains asserted until the response is received. Default = 0.

`req_drop_after_resp`

Number of cycles after the response is deasserted that the request must be deasserted. Default = 0.

`req`

Vector of 1-bit request signals.

`resp`

Vector of 1-bit response signals.

The specified *req* and *resp* are vectors where the bit position indicates the corresponding *req* and *resp*. The *no_chnl* specification is the number of *req* and *resp* bits. That is, the size of the *req* and *resp* vectors is `[no_chnl - 1 : 0]` or `[0 : no_chnl - 1]`.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_req_gets_resp[i]` checks that the response on channel `[i]` arrives within the expected latency interval.
- Assertion `ova_c_req_until_resp[i]` checks that *`req[i]`* remains asserted until a rising edge on *`resp[i]`*.
- Assertion `ova_c_resp_cycles[i]` checks that `resp[i]` stays asserted for *`resp_cycles`* clock ticks.
- Assertion `ova_c_drop_after_resp[i]` checks that *`req[i]`* stays asserted for exactly *`req_drop_after_resp`* clock cycles after the falling edge of *`resp[i]`*.
- Assertion `ova_f_no_req4resp[i]` checks that *`resp[i]`* is not asserted while there is no new *`req[i]`* asserted. This assertion is active only if *`no_req4resp`* is 1.

Examples

The following examples check that the response comes 1 to 3 cycles after a rising edge on request.

Unit:

```
bind module dut : ova_req_resp
  #(2, 1, 3, 0, 1, 1, 0, 1)
  (1'b1, clk, request, response);
```

Template:

```
req_resp( , negedge clk, request, response, 2, 1,
     3, 0, 1, 1, 0);
```

ova_sequence

Ensure that *exp* takes on values in the order implied by their sequence in the *vals* bitvector. *bw* is the number of bits in *exp* and in each of the required values.

Syntax

Unit syntax:

```
ova_sequence
# (no_vals, min_hold, max_hold, bw, disallow, edge, exp,
msg, severity, category);
(en, clk, exp, vals);
```

Template syntax:

```
sequence(en, clk, exp, no_vals, vals, min_hold, max_hold,
disallow, bw, msg, severity, category)
```

Arguments

`no_vals`

An integer value indicating the number of values in the sequence. For example, if *bw = 3, no_vals = 2*, and the values are 3'b000 and 3'b110 (to be reached in that order) then the value bound to the *vals* port is 6'b110_000. Default = 2.

`min_hold`

The minimum number of clock ticks ticks *exp* must hold at a specific value. Default = 1.

`max_hold`

The maximum number of clock ticks *exp* must hold at a specific value. Default = 1.

`bw`

Number of bits in *exp* and in each of the required values.

disallow

If 1, then the sequence is forbidden. Default = 0, sequence is required.

If 0, ensures that the sequence progression occurs.

exp

The assertion will check the sequence whenever exp takes on the first value in the sequence and en evaluates true.

vals

The constant bitvector is formed by concatenating the bitvectors of each of the required values in the sequence, such as., *vals* has *bw * no_vals* bits.

Note the following:

- *min_hold* = *max_hold* = 1 indicates that *exp* progresses to the next value on the next clock tick (i.e., held for one cycle).
- *min_hold* = *max_hold* = 0 indicates that *exp* must hold the current value for at least m clock ticks, no upper bound.
- *min_hold* = m, *max_hold* = 0 indicates that *exp* must hold the current value for at least m clock ticks, no upper bound.
- *min_hold* = m, *max_hold* = n indicates that *exp* must hold the current value for at least m clock ticks and must advance to the next value within n ticks.

Report Message

This checker uses two assertions to cover different modes:

- Assertion `ova_c_sequence` checks that the sequence happens correctly. When *`disallow`* = 0, the assertion ova_c_sequence will report as the failing expression

  ```
  (past(exp) == exp)
  ```

  when exp is not stable for *`min_hold`* clock ticks after entering one of the first *`no_vals`* -1 values in *`vals`*, and

  ```
     (exp == val_i)
  ```

  When *`exp`* did not take on the expected value val_i in the constant array *`vals`* within the expected hold time interval.

- Assertion `ova_f_sequence` checks that the sequence does not happen. When *`disallow`* = 1, the failure of the assertion ova_f_sequence means that the whole sequence and the hold intervals were satisfied which was disallowed to happen.

Examples

```
ova_sequence #(5, 1, 1, 4, 0, 1)
             (1, clk, counter, 20'h0_2_3_1_0);
```

It is required that the 4-bit variable counter, once it takes on the value 0, it must forever follow the Grey-code cycle 0-1-3-2-0-..., advancing to the next value on every tick of negedge clk. (*`disallow`* = 0, meaning that the sequence is required). Note that the sequence in *`vals`* is read from right to left.

ova_stack

Checks operations on a stack.

Syntax

Unit syntax:

```
ova_stack
  #(depth, elem_sz, hi_water_mark, push_lat, pop_lat,
    value_chk, push_pop_chk, edge_expr, msg,
    severity, category, coverage_level)
 instance_name (reset, clk, push, push_data, pop, pop_data);
```

Template syntax:

```
stack(reset, clk, depth, elem_sz, hi_water_mark, push,
push_lat, push_data, pop, pop_lat, pop_data, value_chk,
push_pop_chk, msg, severity, category);
```

Arguments

`depth`
: The maximum size of the stack. Default = 2. The specified *`depth`* can be at most 2^{16}.

`elem_sz`
: The size of data elements in bits. Default = 1.

`hi_water_mark`
: If positive, checks that the depth of the queue is not greater than specified *`hi_water_mark`*. Default = 0.

`push_lat`
: The number of *`enq_clk`* cycles between *`push`* being asserted 1 and *`push_data`* being valid. Default = 0.

`pop_lat`

The number of *`deq_clk`* cycles between *`pop`* being asserted 1 and *`pop_data`* being valid. Default = 1.

`value_chk`
: If 1, checks that *`pop_data`* matches the data at the top of the stack. Default = 1.

`push_pop_chk`
: If 1, checks that push and pop operations do not occur simultaneously. Default = 1.

`reset`
: Initializes the stack to empty when set to 1.

`push`
: Set to 1 when data is being pushed.

`push_data`
: Data being pushed.

`pop`
: Set to 1 when data is being popped.

`pop_data`
: Data being popped.

Note the following:

- All operations including *`reset`* are synchronous to a tick of *`clk`*.
- When *`push`* is asserted 1: Ensures no stack overflow. *`push_lat`* specifies the number of ticks of *`clk`* between the asserting of *`push`* and when *`push_data`* is valid. It must be a compile-time non-negative integer constant (not an interval).

- If *hi_water_mark* is a positive value, then the depth of the stack after the push will be checked to see if *hi_water_mark* is exceeded (>=). Once high water has been exceeded once, this check is disabled until the stack falls below the mark again (<). *hi_water_mark* can be a constant or a design variable.
- If the stack depth is exceeded, a failure is reported and all further checks are disabled until the stack is reset.
- When *pop* is enabled: Ensures the stack is not empty when popped. If a pop is performed on an empty stack, all checking of pop operations is disabled until *reset* is applied or a push occurs.
- If *value_chk* evaluates to 1, it ensures *pop_data* is what was on the *top* of the stack. *pop_lat* specifies the number of cycles of *clk* between the asserting of *pop* and when *pop_data* is valid. It must be a compile-time non-negative integer constant (not an interval).
- If *push_pop_chk* evaluates 1, ensures that push and pop operations do not occur simultaneously. If push and pop do occur simultaneously, the effect is the same as if push were done first followed by a pop (that is, the stack is not changed). If *value_chk* = 1 then *pop_data* is compared with *push_data* in that case.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_stack_overflow` checks that when *push* is issued, the stack is not full. When the assertion fails, the reported time of failure is on the clock tick following the failed deferred push operation.

- Assertion `ova_c_stack_underflow` checks that when *`pop`* is issued, the stack is not empty (and no simultaneous *`push`*). When the assertion fails, the reported time of failure is on the clock tick following the failed deferred push operation.
- Assertion `ova_c_stack_value_chk` checks that the *`pop_data`* value matches the expected top-of-stack value. If the stack is empty and there is a simultaneous *`push`*, the assertion checks that the value pushed on the stack matches the value popped off the stack. This assertion is active only if *`value_chk`* is 1.
- Assertion `ova_c_stack_hi_water_chk` checks that the stack is not filled above the high-water mark.
- Assertion `ova_c_push_pop_fail` checks that *`push`* and *`pop`* are not enabled simultaneously (and *`push_pop_chk`* is 1).

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_number_of_pushes` indicates how many times there was a push operation.

Cover `cover_number_of_pops` indicates how many times there was a pop operation.

Cover `cover_push_followed_eventually_by_pop` indicates how many times a push was followed eventually by a pop without an intervening push.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_simultaneous_push_pop` indicates how many times there were simultaneous push and pop operations.

Cover `cover_simultaneous_push_pop_when_empty` indicates how many times there were simultaneous push and pop operations while the stack was empty.

Cover `cover_simultaneous_push_pop_when_full` indicates how many times there were simultaneous push and pop operations while the stack was full.

Cover `cover_stack_hi_water_chk` indicates how many times the high water mark was reached.

Cover `cover_number_of_full` indicates how many times the stack became full after a push.

Cover `cover_number_of_empty` indicates how many times the stack became empty after a pop.

Examples

The examples below specify the following:

- Checks that the stack is initialized when *`sys_reset`* is asserted.
- The stack is 10 elements deep and 16 bits wide.
- The water mark is set at 8, which is point at which the water-mark check is enabled.
- The push and pop latencies are both 0, which means that *`data_in`* must be present at the same time that push is asserted and *`data_out`* must be present at the same time that pop is asserted.
- The value check is enabled meaning that *`data_out`* will be checked against the data expected on the top of the stack.

- Levels 1 and 3 are enabled in the unit instance (`coverage_level` = 6).

Unit:

```
bind module dut : ova_stack
  #(10, 16, 8, 0, 0, 1, 0, , , , , 6)
 stack_inst (sys_reset, clk, push, data_in, pop, data_out);
```

Template:

```
stack(sys_reset, posedge clk, 10, 16, 8, push, 0,
  data_in, pop, 0, data_out, 1, 0);
```

ova_timeout

Checks that the value of the signal being tested (*exp*), a bit vector, changes within the specified number of cycles (*period*). That is, the value of the signal (*exp*) cannot remain constant longer than the specified maximum number of clock cycles (*period*). A new check begins every time the signal (*exp*) changes.

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_timeout
  #(period, bw, edge_expr, msg, severity, category,
    coverage_level)
 instance_name (en, clk, exp);
```

Template syntax:

```
timeout(en, clk, exp, period, msg, severity, category);
```

Arguments

`period`
: The maximum number of clock cycles before the specified signal (*exp*) changes. Default = 1. The specified *period* must be greater than zero. The default is one, meaning that the specified signal (*exp*) must change every clock cycle.

`bw`
: The number of bits in the specified signal (*exp*). Default = 1.

`exp`
: Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_timeout`.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover property `exp_change` indicates how many times `exp` changed value.

Cover property `cover_exp_changes_within_period` indicates how many times the change occurred within the required period.

`Level_3 (bit 2 set in coverage_level)`

Cover property `cover_exp_changes_exactly_at_period_clks` indicates how many times exp changed exactly at `period` clock cycles.

Examples

The following examples ensure that a PLL output changes at least once every 10 clocks. Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_timeout
  #(10, 1, , "PLL output lasted too long.", , , 5)
 pll_pulse (rst_n, clk, pll_out0);
```

Template:

```
timeout pll_pulse(rst_n, posedge clk, pll_out0, 10,
     "PLL output lasted too long.");
```

ova_tri_state

Checks that the tri-states of the specified input and output signals are equal (*`inp`* == *`outp`*) at the start of the assertion (*`en`* is 1).

Note: You can use a four-state version of this checker. See Appendix A, Four-State OVA Checkers.

Syntax

Unit syntax:

```
ova_tri_state
#(bw, edge_expr, msg, severity, category)
(en, clk, inp, outp);
```

Template syntax:

```
tri_state(en, clk, inp, outp, msg, severity, category);
```

Arguments

`bw`
: The number of bits in input (*`inp`*) and output (*`outp`*) signals. Default = 1.

`inp`
: Input signal.

`outp`
: Output signal.

Report Message

In report messages, the assertion name is `ova_c_tri_state`.

Examples

Unit:

```
bind module dut : ova_tri_state assert_tri_state
  #(,, "Asserted oen, data does not match data_read.")
  (oen, clk, data_read, data);
```

Template:

```
tri_state assert_tri_state(oen, posedge clk,
     data_read, data, "Asserted oen, but data does not
     match data_read.");
```

ova_underflow

Checks that the signal being tested (*exp*) does not transition between the specified minimum (*min*) and maximum (*max*) values. Works only with unsigned vector values.

Syntax

Unit syntax:

```
ova_underflow
#(min, max, bw, edge_expr, msg, severity, category)
(en, clk, exp);
```

Template syntax:

```
underflow(en, clk, exp, min, max, msg, severity, category);
```

Arguments

`min`
    The minimum value allowed. Default = 0.

`max`
    The maximum value allowed. Default = 1.

`bw`
    The number of bits in the signal being tested (*exp*). Default = 2.

`exp`
    Signal being tested.

Report Message

In report messages, the assertion name is `ova_c_underflow`.

Examples

The following examples check that the countdown timer does not flip from 0 to FF unless it is reset.

Unit:

```
bind module dut : ova_underflow counter_underflow
  #(0, 8'hff, 8, , "Countdown timer underflowed.")
  (rst_n, clk, counter);
```

Template:

```
underflow assert_cntr_no_underflow(rst_n, posedge clk,
  counter, 0, 8'hff, "Countdown timer underflowed.");
```

ova_valid_id

Checks that IDS are issued and returned.

Syntax

Unit syntax:

```
ova_valid_id
#(id_bw, max_ids, max_out_ids, max_out_per_id, min_lat,
max_lat, edge_expr, msg, severity, category)
(en, clk, issued_sig, issued_id, ret_sig, ret_id, reset_sig,
reset_id);
```

Template syntax:

```
valid_id(en, clk, id_bw, max_ids, max_out_ids,
max_out_per_id, issued_sig, issued_id, ret_sig, ret_id,
reset_sig, reset_id, min_lat, max_lat, msg, severity,
category);
```

Arguments

`id_bw`

The number of bits in *`issued_id`*, *`ret_id`*, and *`reset_id`*. Default = 2. That is, *`id_bw`* is the maximum number of bits used to encode an ID.

`max_ids`

The maximum number of IDs. Default = 4.

The maximum value is 2^{16}.

`max_out_ids`

The maximum number of IDs that can be outstanding. Default = 1.

`max_out_per_id`

The maximum number of issues outstanding per ID. Default = 1.

`min_lat`

Minimum number of clock cycles between an ID being issued and returned. Default = 1.

`max_lat`

Maximum number of clock cycles between an ID being issued and returned. Default = 0.

`eissued_sig`

If 1, *`issued_id`* has a valid value.

`issued_id`

The ID being issued.

`ret_sig`

If 1, *`ret_id`* has a valid value.

`ret_id`

The ID being returned.

`reset_sig`

If 1, *`reset_id`* has a valid value.

`reset_id`

The ID whose outstanding count is being reset.

Note the following:

- The signal *`issued_sig`* asserted 1 validates a request identified by the value in *`issued_id`*. This request is expected to be acknowledged by *`ret_id`* and validated by *`ret_sig`* asserted 1 within [*`min_lat`* .. *`max_lat`*] delay.

- When *`issued_sig`* is asserted 1, it ensures that *`max_out_ids`* is not exceeded and that *`issued_id`* is at most *`max_out_per_id`* outstanding (that is, not returned). When *`max_out_per_id`* > 1 then the returns for an ID cannot be distinguished. But for any issue, there must be a return of the ID within the latency interval.
- The assumption is that *`issued_id`* and *`ret_id`* will always be values in the range of 0:(*`max_ids`* -1).
- When *`reset_sig`* is asserted 1 then the outstanding count of *`reset_id`* ID is reset to 0. The specified *`issued_sig`* must not be asserted at the same time as *`reset_sig`*. The specified *`ret_id`* must not be asserted at the same time as *`reset_id`*.
- If an *`issued_id`* exceeds the number of outstanding IDs, *`max_out_ids`*, an error is reported. Necessarily *`max_ids`* >= *`max_out_ids`* must hold. Only those IDs are counted that have *`en`* asserted at issuance.
- When *`ret_sig`* is asserted 1, *`ret_id`* must match an outstanding ID. A returned *`ret_id`* is counted only against issued IDs in the previous clock cycles. It also ensures that *`issued_id`* is outstanding for at least *`min_lat`* cycles but no longer than *`max_lat`* cycles after issuance.
- The *`issued_sig`* and *`ret_sig`* control signals are active for only one clock or they are edge expressions (posedge *`ret_sig`* or negedge *`ret_sig`*, for example).
- To trigger a check for an issued ID, *`en`* must be asserted 1 at the time *`issued_sig`* is asserted and also at the time *`ret_sig`* is asserted. If an ID is returned and *`en`* is asserted while at issuance of this ID, *`en`* is deasserted, this return is flagged as an error.
- *`min_lat`* = 1 and *`max_lat`* = 1 means that the ID must be returned on the next clock cycle after issuance.

- *min_lat* = m and *max_lat* = 0 means that the ID must be returned any time after m clock ticks.
- *min_lat* = m and *max_lat* = n means that the ID must be returned within the interval m..n clock ticks after issuance.

Report Message

This checker uses several assertions to cover different aspects and modes:

- Assertion `ova_c_valid_id[i]` checks that when an ID of value `i` is issued, that ID is returned within the required latency interval [*min_lat* .. *max_lat*] clock cycles.
- Assertion `ova_c_issued_id_ok` checks that when an ID is issued, it is not in circulation with a count exceeding *max_out_per_id*.
- Assertion `ova_c_ret_id_ok` checks that when an ID is returned, there is such an ID value in circulation (that is, this ID has been issued).
- Assertion `ova_c_max_issued_ids_ok` checks that the total of different IDs in circulation does not exceed the *max_out_ids* value.

Examples

The following examples specify:

- The checker is enabled when out of reset. All signals are sampled on posedge `clk`.
- There are at most 8 IDs (0 to 7) of which at most 6 can be in circulation at any time and only one copy of each.

- The count of the copies of an ID is reset when clear is asserted with the corresponding ID value on *clear_id*.
- An issued ID is valid when *issued_valid* is asserted.
- A returned ID is valid when *ret_valid* is asserted.
- An issued ID must be returned in 1 to 10 clock cycles.

Unit:

```
bind module dut : ova_valid_id
  #(8, 8, 6, 6, 1, 1, 10, ,
    "latency 1 to 10, 6 out of 8 IDs, 1 copy each")
  (!reset, clk, issued_valid, issued_id, ret_valid,
    ret_id, clear, clear_id);
```

Template:

```
valid_id(!reset, posedge clk, 8, 8, 6, 1,
  issued_valid, issued_id, ret_valid, ret_id,
  clear, clear_id, 1, 10,
  "latency 1 to 10, 6 out of 8 IDs, 1 copy each");
```

ova_value

Checks that the signal being tested (*exp*) is only one of the specified values.

Syntax

Unit syntax:

```
ova_value
  #(no_vals, disallow, bw, edge_expr, msg, severity,
    category, coverage_level)
 instance_name(en, clk, exp, vals);
```

Template syntax:

```
value(en, clk, exp, no_vals, vals, disallow, bw, msg,
severity, category);
```

Arguments

`no_vals`
: The number of entries in the *vals* specification. Default = 1.

`disallow`
: If 1, checks that the signal being tested (*exp*) does not match any of the values in the specified array of values (*val*). Default = 0.

  If 0, checks that the signal being tested (*exp*) does match one of the values in the specified array of values (*val*).

`bw`
: Number of bits in the signal being tested (*exp*) and each element of the specified array of values (*val*). Default = 2.

`exp`
: Signal being tested.

`vals`

A bitvector of concatenated values that the signal being tested (*`exp`*) must evaluate to (`logic [bw*no_vals-1:0] vals`).

Report Message

This checker uses two assertions to cover different modes:

- Assertion `ova_c_value` checks that *`exp`* is one of the specified values. This assertion is active only if *`disallow`* is 0.
- Assertion `ova_f_value` checks that *`exp`* is not one of the specified values. This assertion is active only if *`disallow`* is 1.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_exp_change` indicates how many times exp changed value.

`When disallow == 0:`

Cover `cover_value` indicates how many times a valid value from `vals` occurred.

`When disallow == 1:`

Cover `cover_exp_state_transitions` indicates how many times a value other than those in `vals` occurred.

`Level_3 (bit 2 set in coverage_level)`
`Exists only when disallow == 0`

Cover `cover_exp_changes_to_value[i]` indicates how many times `exp` was equal to the value `vals[i]`.

Examples

The following examples check that when load is 1 and reset is 0, `control_reg` has one of the four values 0, 1, 3, or 5. Coverage Levels 1 and 3 are enabled in the unit instance (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_value
  #(4, 0, 8, , , , , 5)
 value_inst (load && !reset, clk, control_reg,
              {{0}, {1}, {3}, {5}});
```

Template:

```
value(load && !reset, posedge clk, control_reg, 4,
     {{0}, {1}, {3}, {5}}, 0, );
```

ova_window

Checks that individual bits in the bitvector signal (*assert_vector*) are asserted or deasserted either within or outside the window. The window is defined at the beginning by *start_sig* evaluating true plus *delay* cycles and ending with *stop_sig* evaluating true or *win_time* number of cycles after posedge *start_sig* plus *delay* cycles evaluates true.

Syntax

Unit syntax:

```
ova_window
   #(check_type, delay, win_time, bw, edge_expr, msg,
     severity, category, coverage_level)
instance_name (en, clk, start_sig, stop_sig, assert_vector);
```

Template syntax:

```
window(en, clk, start_sig, delay, stop_sig, win_time,
assert_vector, bw, check_type, msg, severity, category);
```

Arguments

`check_type`

The type of check. Default = 0.

0: Each bit must be asserted at some time during the window, but not necessarily at the same time.

1: Each bit must be asserted at least once outside the window.

2: Each bit must be de-asserted during the entire length of the window.

3: Each bit must be de-asserted on every *clk* outside the window.

`delay`

The number of cycles after the start signal (*`start_sig`*) evaluates true to the beginning of the window. Default = 0.

`win_time`

The maximum length of the window in clock cycles. Default = 1.

The cycle that the window opens on is outside the window. The cycle that closes the window is inside the window. If there is no window timeout, then *`win_time`* should be 0.

`bw`

The number of bits in the *`assert_vector`* specification. Default = 1.

A signal value is checked in or out of the window only if *`en`* is asserted at that clock tick. Whenever there is no window active, it is considered as outside the window and *en* can be used to control when checking outside the window is performed (types 1 and 3).

`start_sig`

Signal that is part of starting the window.

`stop_sig`

Signal that marks the end of the window. If there is no *`stop_sig`* (only a specific number of cycles is to be considered), then *`stop_sig`* should be 0 (the default value).

`assert_vector`

Signal being tested.

Report Message

This checker uses several assertions to cover different modes:

- Assertion `ova_c_asserted_in` checks that all bits of *`asserted_in`* are set during the window. When the assertion fails, the “offending” expression in the message might include “`var_asserted_in`”. This is an OVA variable that accumulates the bits asserted during the window. This assertion is active only if *`check_type`* is 0.
- Assertion `ova_c_asserted_out` checks that all bits of *`asserted_out`* are set outside the window. When the assertion fails, the “offending” expression in the message might include “`var_asserted_out`”. This is an OVA variable that accumulates the bits asserted outside the window. This assertion is active only if *`check_type`* is 1.
- Assertion `ova_c_deasserted_in` checks that all bits of *`deasserted_in`* remain 0 during the window. When the assertion fails, the “offending” expression in the message might include “`var_deasserted_in`”. This is an OVA variable that accumulates the bits asserted during the window. This assertion is active only if *`check_type`* is 2.
- Assertion `ova_c_deasserted_out` checks that all bits of *`deasserted_out`* remain 0 outside the window. When the assertion fails, the “offending” expression in the message might include “`var_deasserted_out`”. This is an OVA variable that accumulates the bits asserted outside the window. This assertion is active only if *`check_type`* is 3.

Coverage modes

```
Check_type = 0 (asserted inside)
Level_1 (bit 0 set in coverage_level)
```

Cover `cover_asserted_in` indicates how many times there was a match within the window.

```
Level_3 (bit 2 set in coverage_level)
```

Cover
`cover_num_of_times_bit_asserted_just_after_start_sig_plus_delay[i]` indicates how many times bit `assert_vector[i]` was set to 1 exactly delay cycles after `start_sig` was asserted.

Cover
`cover_num_of_times_bit_asserted_just_at_stop_sig[i]` indicates how many times bit `assert_vector[i]` as set to 1 exactly when `stop_sig` occurred.

Cover
`cover_num_of_times_bit_asserted_just_at_win_time_expires[i]` indicates how many times bit `assert_vector[i]` as set to 1 exactly when `win_time` expired.

Cover
`cover_num_of_times_all_bits_asserted_just_after_start_sig_plus_delay` indicates how many times all bits were asserted at the same time `delay` cycles after `start_sig` was asserted.

Cover
`cover_num_of_times_all_bits_asserted_just_at_stop_sig` indicates how many times all bits were asserted at the same time when `stop_sig` was asserted.

Cover
`cover_num_of_times_all_bits_asserted_just_at_win_time_expires` indicates how many times all bits were asserted at the same time when `win_time` expired.

```
Check_type = 1 (asserted outside)
```

`Level_1 (bit 0 set in coverage_level)`

Cover `cover_asserted_out` indicates how many times there was a match outside the window.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_num_of_times_bit_asserted_just_at_start_sig_plus_delay[i]` indicates how many times bit `assert_vector[i]` was set to 1 exactly delay cycles after `start_sig` was asserted.

Cover `cover_num_of_times_bit_asserted_just_after_stop_sig[i]` indicates how many times bit `assert_vector[i]` as set to 1 just after `stop_sig` occurred.

Cover `cover_num_of_times_bit_asserted_just_after_win_time_expires[i]` indicates how many times bit `assert_vector[i]` as set to 1 just after `win_time` expired.

Cover `cover_num_of_times_all_bits_asserted_just_at_start_sig_plus_delay` indicates how many times all bits were asserted at the same time at `delay` cycles after `start_sig` was asserted.

Cover `cover_num_of_times_all_bits_asserted_just_after_stop_sig` indicates how many times all bits were asserted at the same time just after `stop_sig` was asserted.

Cover `cover_num_of_times_all_bits_asserted_just_after_win_time_expires` indicates how many times all bits were asserted at the same time just after `win_time` expired.

`Check_type = 2 (deasserted inside)`
`Level_1 (bit 0 set in coverage_level)`

Cover `cover_deasserted_in` indicates how many times there was a match within the window.

`Level_3 (bit 2 set in coverage_level)`

Cover `cover_num_of_times_bit_deasserted_just_after_start_sig_plus_delay[i]` indicates how many times bit `assert_vector[i]` as set to 1 exactly delay cycles after `start_sig` was asserted.

Cover `cover_num_of_times_bit_reasserted_just_after_stop_sig[i]` indicates how many times bit `assert_vector[i]` rose right after `stop_sig` occurred.

Cover `cover_num_of_times_bit_reasserted_just_after_win_time_expires[i]` indicates how many times bit `assert_vector[i]` rose right after `win_time` expired.

Cover `cover_num_of_times_all_bits_deasserted_just_after_start_sig_plus_delay` indicates how many times all bits were deasserted at the same time just after `delay` cycles after `start_sig` was asserted.

Cover
`cover_num_of_times_all_bits_reasserted_just_after_stop_sig` indicates how many times all bits were reasserted at the same time just after `stop_sig` was asserted.

Cover
`cover_num_of_times_all_bits_reasserted_just_after_win_time_expires` indicates how many times all bits were reasserted at the same time just after `win_time` expired.

```
Check_type = 3 (deasserted outside)
Level_1 (bit 0 set in coverage_level)
```

Cover cover_deasserted_out indicates how many times there was a match outside the window.

```
Level_3 (bit 2 set in coverage_level)
```

Cover
`cover_num_of_times_bit_reasserted_just_after_start_sig_plus_delay[i]` indicates how many times bit `assert_vector[i]` rose just after delay cycles after start_sig was asserted.

Cover
`cover_num_of_times_bit_deasserted_just_after_stop_sig[i]` indicates how many times bit `assert_vector[i]` fell just after `stop_sig` occurred.

Cover
`cover_num_of_times_bit_deasserted_just_after_win_time_expires[i]` indicates how many times bit `assert_vector[i]` fell just after `win_time` expired.

Cover `cover_num_of_times_all_bits_reasserted_just_after_start_sig_plus_delay` indicates how many times all bits were reasserted at the same time just after `delay` cycles after `start_sig` was asserted.

Cover `cover_num_of_times_all_bits_deasserted_just_after_stop_sig` indicates how many times all bits were deasserted at the same time just after `stop_sig` was asserted.

Cover `rvmLIMIT3cover_num_of_times_all_bits_deasserted_just_after_win_time_expires` indicates how many times all bits were deasserted at the same time just after `win_time` expired.

Examples

The following examples specify that the window starts 1 clock tick after start transitions to 1 and lasts for two clock cycles. There are two bits in flags that should be asserted during the window. Coverage Levels 1 and 3 are selected (*`coverage_level`* = 5).

Unit:

```
bind module dut : ova_window
   #(0, 1, 2, 2, , , , , 5)
 win_inst (reset_n, clk, start, 1'b0, flags);
```

Template:

```
window (reset_n, posedge clk, start, 1, 0, 2, flags,
  2, 0);
```

2

OVL-Equivalent Checkers

This chapter describes OVA checkers that verify the same behavior as checkers available in Accellera’s proposed "Open Verification Library", Version 02.09.24.

This chapter covers the following topics:

- Converting from a Verilog OVL Library to OVA
- Descriptions of OVL-Equivalent OVA Checkers

Converting from a Verilog OVL Library to OVA

There are several methods you can use to convert OVL Verilog checker instances in a Verilog model to equivalent inlined OVA checkers:

- Single Line Replacement
- Multiple Line Replacement
- Inlining OVA Units in a Verilog Wrapper Module
- Combining OVA and OVL Checkers in the Same Design

This section describes each of these methods.

Single Line Replacement

If the OVL module instance extends over only one line of code, you only need to place the prefix "`//ova bind `" in front of the original OVL instance.

For example, suppose you want to replace the following OVL module instance:

```
assert_always my_inst (clk, reset_n, expression);
```

The equivalent OVA is as follows:

```
//ova bind assert_always my_inst (clk, reset_n, expression);
```

Multiple Line Replacement

dIf the OVL module instance extends over multiple lines, you can use either of the following two multi-line specification techniques:

Technique #1:

1. Insert a line "`/* ova bind`" before the instance.
2. insert a line with "`*/`" after the instance.

Technique #2:

1. Insert a line with "`//ova_begin`" before the instance.
2. Prefix the OVL instance with "bind".
3. Insert a line with "`//ova_end`" after the instance.

For example, suppose you want to replace the following OVL instance with an equivalent OVA checker.

```
assert_always #(1, 0, "my_message")
     my_always_instance (clk, reset_n, expression);
```

The following example shows a valid forms of specifying OVA inlined checkers:

```
/* ova bind
assert_always #(1, 0, "my_message")
     my_always_instance (clk, reset_n, expression);
*/
```

or

```
ova_begin bind
assert_always #(1, 0, "my_message")
     my_always_instance (clk, reset_n, expression);
ova_end
```

Inlining OVA Units in a Verilog Wrapper Module

Many of the OVA units described in this chapter can be inlined in a Verilog wrapper module, which then can be instantiated in the design (Note: Of the 30 available checkers, 20 can be used for inlining; exceptions are described in the "Restrictions" section below). The compile command must include the "`-ova_inline`" option to indicate that the OVA checkers should be processed.

To change from an original OVL checker to a OVA-based checker, you must do the following:

1. Remove any reference to the original OVL library.
2. Include a reference to the directory where the OVA Verilog wrapper modules are located.
3. Compile with the `-ova_inline` option.

Combining OVA and OVL Checkers in the Same Design

You can use original OVL checkers and new OVA-based checkers in the same design. Using the `define macro, you can select which checker to use in which situation (e.g., use OVA to get functional coverage information).

For example:

```
 `ifdef OVA
//ova bind assert_always #(,,"message-OVA") my_AG_instance (clk, rst_n, expr);
`elsif
assert_always #(,,"message-OVL") my_AG_instance (clk, rst_n, expr);
`endif
```

Restrictions

Note the following restrictions when using OVL-equivalent checkers:

- The OVL checker `assert_proposition` is not available in OVA because it is an asynchronous checker that does not require variable sampling.

- Unit parameters that control the extent of synchronous delays (number of clock ticks) and assertion variants in a checker must be compile-time constants — they must not be specified using design parameters. This restriction concerns the following checkers and parameters:

Checker	Parameters
assert_always_on_edge	edge_type
`assert_change`	`num_cks, flag`
`assert_cycle_sequence`	`necessary_condition, num_cks`
`assert_frame`	`min_cks, max_cks, flag`
`assert_handshake`	`min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length`
`assert_next`	`num_cks, check_overlapping, only_if`
`assert_one_cold`	`inactive`
`assert_time`	`num_cks, flag`
`assert_unchange`	`num_cks, flag`
`assert_width`	`min_cks, max_cks`

Macro symbol ASSERT_OFF

The `assert` directives in the OVL-like checkers that include `cover` directives as listed in Chapter 1 can be globally disabled by defining the symbol ASSERT_OFF.

Inlining OVA Units in a Verilog Wrapper Module

Many of the OVA units described in this chapter can be inlined in a Verilog wrapper module, which then can be instantiated in the design (Note: Of the 30 available checkers, 20 can be used for inlining; exceptions are described in the "Restrictions" section below). The compile command must include the "`-ova_inline`" option to indicate that the OVA checkers should be processed.

To change from an original OVL checker to a OVA-based checker, you must do the following:

1. Remove any reference to the original OVL library.
2. Include a reference to the directory where the OVA Verilog wrapper modules are located.
3. Compile with the `-ova_inline` option.

Note: Coverage using cover statements in the select checkers listed in “Coverage Properties” on page 1-3 is not available through OVL Verilog wrappers at this time.

Using OVL-Equivalent Checkers with VHDL Designs

OVA assertions and checkers cannot yet be inlined in VHDL designs. Therefore, the only way to add the checkers to such designs is by creating an external OVA file that contains the appropriate OVA bind statements.

Descriptions of OVL-Equivalent OVA Checkers

This section provides descriptions of the following OVL-equivalent checkers:

```
assert_always_on_edge
assert_change
assert_cycle_sequence
assert_decrement
assert_delta
assert_even_parity
assert_fifo_index
assert_frame
assert_handshake
assert_implication
assert_increment
assert_never
assert_next
assert_no_overflow
assert_no_transition
assert_no_underflow
assert_odd_parity
assert_one_cold
```

assert_one_hot
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

Note: The severity and category parameters for these checkers can take defalt values as specified by the `set_severity` and `set_category` OVA commands.

---

assert_always

This checker continuously monitors *`test_expr`* at every positive edge of clock, *`clk`*. It verifies that *`test_expr`* will always evaluate TRUE. If *`test_expr`* evaluates to FALSE, the assertion will fire.

Syntax

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

Arguments

`severity_level`
Severity of the failure (default is 0).

`options`
Currently, the only supported option is options=1, which defines the assertion as an assumption for formal tools. The default is 0 (no options specified).

`msg`
Error message printed when the checker fires.

`category`
Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`test_expr`

Expression being verified at the posedge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover property `cover_always` indicates the number of times test_expr was asserted when enabled by `reset_n`.

Example

```
'define ASSERT_ON
'define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg [3:0] count;

initial $monitor("count = %b \n", count);
```

```
always @(posedge clk) begin
    if (reset_n == 0 || count >= 9)
        count <= 1'b0;
    else
        count <= count + 1;
end

/*
ova bind assert_always //coverage_level = 4 (Level 3)
  #(0, 0, "ERROR: count not within 0 and 9", 0, 4)
    valid_count
  (clk, reset_n, (count >= 4'b0000) && (count <= 4'b1001));
*/
endmodule
```

assert_always_on_edge

This checker continuously monitors the *test_expr* at every specified edge of the *sampling_event* that coincides with the positive edge of clock, *clk*. The *test_expr* should always evaluate TRUE at the *sampling_event*. If *test_expr* evaluates to FALSE, the assertion will fire.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0.)

`edge_type`

Selects the transition for *sampling_event*:

`0` — no edge (default)

`1` — positive edge

`2` — negative edge

`3` — any edge

`options`
Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
The error message that will be printed if the assertion fires.

`category`
Checker type (default is 0).

`clk`
Triggering or clocking event that monitors the assertion.

`reset_n`
Signal indicating completed initialization.

`sampling_event`
Expression defines when to evaluate *`test_expr`*. Transition of sampling_event are selected by *`edge_type`*.

`test_expr`
Expression being verified at the positive edge of *`clk`*, AND if *`sampling_event`* matches transition selected by *`edge_type`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
Cover property `cover_always_on_edge` indicates the number of times `test_expr` was asserted on the specified edge of `sampling_event`.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk, sig;

child CH (reset_n, clk, sig);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    sig = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;
always #40 sig = ~sig;

endmodule


module child(reset_n, clk, sig);
input reset_n, clk, sig;
reg [3:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk) begin
    if (reset_n == 0 || count >= 9)
        count <= 1'b0;
    else
        count <= count + 1;
end

/* ova
bind assert_always_on_edge // coverage_level = 1
  #(0, 1, 0, "ERROR: count not within 0 and 9", , 1)
    valid_always_on_edge
 (clk, reset_n, sig, (count >= 4'b0000) &&
 (count <=  4'b1001));
```

```
*/
endmodule
```

assert_change

This checker continuously monitors the *start_event* at every positive edge of the clock. When *start_event* is TRUE, the checker ensures that the expression, *test_expr*, changes values on a clock edge at some point within the next *num_cks* number of clocks. This assertion will fire upon a violation.

Syntax

```
assert_change
    [#(severity_level, width, num_cks, flag, options, msg,
       category, coverage_level)]
  instance_name (clk, reset_n, start_event, test_expr);
```

Arguments

`severity_level`

Severity of the failure (default is 0).

`Width`

Width of the expression, *test_expr* (default is 1.

`Num_cks`

Number of clocks for *test_expr* to change its value before an error is triggered after *start_event* is asserted (default is 1).

`flag`

0 — Ignore any start_event assertion after the first one has been detected.

1 — Restart the monitoring test_expr, if *start_event* is asserted in any subsequent clock while monitoring *test_expr*.

2 — Issue an error if an asserted *start_event* occurs in any clock cycle while monitoring *test_expr*.

options

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

msg

Error message that will be printed when the assertion fires.

category

Checker type (default is 0).

clk

Sampling clock of the checker.

reset_n

Signal that indicates a completed initialization.

start_event

Starting event that triggers monitoring of the *test_expr*.

test_expr

Expression or variable being verified at the positive edge of *clk*.

Coverage modes

Level_1 (bit 0 set in coverage_level)

Cover property cover_change indicates the number of times exp changed within num_cks.

Cover property cover_start_event indicates the number of times start_event occurred.

`Level_3 (bit 2 set in coverage_level)`

Cover property `cover_overlapping_start_events` indicates how many times `start_event` occured while there was another evaluation attempt in progress.

Cover property `cover_change_after_1_clk` indicates the number of times `test_expr` changed value at the next clock tick after `start_event`.

Cover property `cover_change_after_num_cks` indicates the number of times `test_expr` changed value at `num_clks` clock ticks after `start_event`.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #1000 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg start;
reg [3:0] expr;
```

```
integer count;

initial $monitor ("count = %d  start = %b  expr = %b \n",
count, start, expr);

always @(posedge clk)
begin
     if (reset_n == 0)
     begin
         start <= 0;
         expr  <= 4'b0000;
         count <= 0;
     end
     else
         count <= count + 1;

     if (count == 3 || count == 4 || count == 5 || count ==
6 || count == 7 || count == 8 || count == 9)
         expr <= 4'b0110;
     else
         expr <= expr + 1;

     if (count == 4)
         start <= 1;
     else
         start <= 0;
end

/* ova bind
assert_change #(0, 4, 4, 0, 0,
"ERROR: expr did not change in num_clk cycles after start",
                      , 7) // all Coverage Levels
valid_change (clk, reset_n, start, expr);
*/

endmodule
```

assert_cycle_sequence

This checker verifies the following conditions:

- When *necessary_condition* = 0, if all num_cks-1 first events of a sequence (event_sequence[num_cks-1:1]) are TRUE, the last sequence (event_sequence[0]) should follow.
- When *necessary_condition* = 1, if the first event of a sequence (*event_sequence*[*num_cks*-1]) is TRUE, then all the remaining *event_sequence*[*num_cks*-2:0] events should follow.

Syntax

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

Arguments

severity_level
    Severity of the failure (default is 0).

num_cks
    The length of the *event_sequence* (number of clock cycles of the *event_sequence*) that must be valid. Otherwise, the checker will fire.

necessary_condition
    Either 1 or 0 (default 0).

options
    Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

msg
    Error message that will be printed when the assertion fires.

category
    Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`event_sequence`

A Verilog concatenation expression, where each bit represents an event.

---

assert_decrement

This checker continuously monitors the *`test_expr`* at every positive edge of the clock signal, *`clk`*. It checks that the *`test_expr`* will never decrease by anything other than the value specified by value.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default 0).

`width`

Width of *`test_expr`* (default is 1).

`value`

Maximum decrement value allowed for *`test_expr`* (default is 1).

`options`

Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
Error message that will be printed when the assertion fires.

`category`
Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`test_expr`
Expression being verified at the positive edge of *`clk`*.

---

assert_delta

This checker continuously monitors the *`test_expr`* at every positive edge of clock signal, *`clk`*. It verifies that *`test_expr`* will never change value by anything less than "min" and anything more than "max" value.

Syntax

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

Arguments

`severity_level`
Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 1).

`min`
: Minimum changed value allowed for *`test_expr`* in two consecutive clocks of *`clk`* (default is 1).

`max`
: Maximum changed value allowed for *`test_expr`* in two consecutive clocks of *`clk`* (default is 1).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

assert_even_parity

This checker continuously monitors the *`test_expr`* at every positive edge of the clock signal, *`clk`*. It verifies that *`test_expr`* will always have an even number of bits asserted.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 1).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
: Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover property `cover_test_expr_change` indicates how many times `test_expr` changed value.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg [7:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
        count <= 8'b11111111;
    else
        count <= count << 2;
end
/* ova bind
assert_even_parity // Default coverage level
#(0, 8, 0, "ERROR: count has odd number of bits asserted")
valid_count_even (clk, reset_n, (count));
*/
endmodule
```

assert_fifo_index

This checker ensures that the FIFO element:

- Never overflows and underflows
- Allows/disallows simultaneous push and pop.

Syntax

```
assert_fifo_index
  [#(severity_level, depth, push_width, pop_width, options,
msg, category)]
  instance_name (clk, reset_n, push, pop);
```

Arguments

`severity_level`
  Severity of the failure (default is 0).

`depth`
  Depth of the FIFO (default is 1). It should never be set to 0, otherwise an assertion will fire.

`push_width`
  Width of the PUSH signal (default is 1).

`pop_width`
  Width of the POP signal (default is 1).

`options`
  Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
  Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`push`
FIFO PUSH/enqueue signal.

`pop`
FIFO POP/dequeue signal.

---

assert_frame

This checker validates proper cycle timing relationships between two events in the design. When a *`start_event`* evaluates TRUE, then the *`test_expr`* must evaluate TRUE within a minimum and maximum number of clock cycles.

Syntax

```
assert_frame
   [#(severity_level, min_cks, max_cks, flag, options, msg,
category)]
   instance_name (clk, reset_n, start_event, test_expr);
```

Arguments

`severity_level`
Severity of the failure (default is 0).

`min_cks`

Minimum number of clock cycles, within which the *`test_expr`* should not become TRUE. When *`min_cks`* is 0, then *`test_expr`* can occur at the same time as *`start_event`* or after, as controlled by *`max_cks`*. Default is 0.

`max_cks`

Maximum number of clock cycles, before which *`test_expr`* must become TRUE. This check will be disabled when *`max_cks`* is not specified. If both *`min_cks`* and *`max_cks`* are 0 then *`test_expr`* must occuer at the same time as there is a 0 to 1 transition on *`start_event`*. The default is 0.

`flag`

0 — Ignores any asserted start_event after the first one has been detected (default).

1 — Restart monitoring *`test_expr`* if *`start_event`* is asserted in any subsequent clock while monitoring *`test_expr`*.

2 — Issue an error if an asserted *`start_event`* occurs in any clock cycle while monitoring test_expr.

`options`

Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0)

`clk`

Triggering or clocking sampling event for assertion.

`reset_n`

Signal indicating completed initialization.

start_event

Starting event that triggers monitoring of the *`test_expr`*. The *`start_event`* is a cycle transition from 0 to 1.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

---

assert_handshake

This checker continuously monitors the *`req`* and *`ack`* signals at every positive edge of the clock *`clk`*. Note that both *`req`* and *`ack`* must go inactive prior to starting a new cycle.

To activate one or more checks in the checker, the following parameters should be specified with a non-zero value:

`min_ack_cycle`

When this parameter is greater than 0, the assertion will ensure that an *`ack`* does not occur before *`min_ack_cycle`* clock ticks.

`max_ack_cycle`

When this parameter is greater than 0, the assertion will ensure that an *`ack`* does not occur after *`max_ack_cycle`* clock ticks.

`req_drop`

When this parameter is greater than 0, the assertion will ensure that *`req`* remains active until an *`ack`* occurs.

`deassert_count`

When this parameter is greater than 0, the assertion will ensure that req becomes inactive (0) within *deassert_count* clock ticks after an *ack*.

max_ack_length

When this parameter is greater than 0, the assertion will ensure that *ack* is not asserted for greater than *max_ack_length* clock cycles and does not become inactive (0) within *deassert_count* clocks after *ack* is asserted (that is, check for *ack* stuck active).

Note that if you do not specifiy a parameter with a non-zero value, the corresponding check will not be active.

Syntax

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

Arguments

severity_level

Severity of the failure (default is 0).

min_ack_cycle

Activate *min_ack_cycle* check if greater than 0.

max_ack_cycle

Activate *max_ack_cycle* check if greater than 0.

req_drop

Activate *req_drop* check if greater than 0.

deassert_count

Activate *deassert_count* if greater than 0.

`max_ack_length`
: Activate max_ack_length check if greater than 0.

`options`
: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0)

`clk`
: Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

---

assert_implication

This checker continuously monitors *antecedent_expr*. If it evaluates to TRUE, then this checker will verify that *consequent_expr* is TRUE.

When *antecedent_expr* is evaluated to FALSE, then *consequent_expr* expression will not be checked at all and the implication is satisfied.

Syntax

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

Arguments

`severity_level`
: Severity of the failure, default 0.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type, default 0.

`clk`
Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`antecedent_expr`
: Expression verified at the positive edge of the clock, *clk*.

`consequent_expr`
: Expression verified at the positive edge of the clock, *clk*.

assert_increment

This checker continuously monitors *test_expr* at every positive edge of the clock, *clk*. It verifies that *test_expr* will never increase by anything other than the value specified by value. The *test_expr* can be any valid Verilog expression. The check will not start until the first clock after the *reset_n* is asserted.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 1).

`value`
: Maximum increment value allowed for *`test_expr`* (default is 1).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
: Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

---

assert_never

This checker continuously monitors *`test_expr`* at every positive edge of clock, *`clk`*. It verifies that *`test_expr`* will never evaluate TRUE. The *`test_expr`* can be any valid Verilog expression. When *`test_expr`* evaluates TRUE, this checker will fail.

Syntax

```
assert_never
   [#(severity_level, options, msg, category)]
   instance_name (clk, reset_n, test_expr);
```

Arguments

`severity_level`
: Severity of the failure (default is 0).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type, default 0.

`clk`
: Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

assert_next

This checker verifies the proper cycle timing relationship between two events in the design at every posedge of the clock, *`clk`*. When a *`start_event`* evaluates TRUE, then *`test_expr`* must evaluate TRUE exactly *`num_cks`* number of clock cycles later.

This checker supports overlapping sequences. For example, if you assert that *test_expr* will evaluate TRUE exactly four cycles after *start_event*, it is not necessary to wait until the sequence finishes before another sequence can begin.

Syntax

```
assert_next [#(severity_level, num_cks, check_overlapping,
only_if, options, msg, category)]
   instance_name (clk, reset_n, start_event, test_expr);
```

Arguments

`severity_level`
: Severity of the failure (default is 0).

`num_cks`
: Number of clocks for the test_expr to become TRUE after *start_event* is asserted (default is 0).

`check_overlapping`
: If set to 1, permits overlapping sequences. In other words, a new *start_event* can occur (starting a new sequence in parallel) while the previous sequence continues. (Default is 1.)

`only_if`
: If set to 1, a *test_expr* can only evaluate TRUE if preceded *num_cks* earlier by a *start_event*. If *test_expr* occurs without a *start_event*, then an error is reported. Default 0.

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
: Sampling clock of the assertion on posedge *`clk`*.

`reset_n`
: Signal indicating completed initialization.

`start_event`
: Starting event that triggers monitoring of the *`test_expr`*.

`test_expr`
: Expression or variable being verified at the positive edge of *`clk`*.

---

assert_no_overflow

This checker ensures that the *`expr`*, from 'max' value, never goes to a value that is less than or equal to 'min' and greater than 'max', at every posedge of the clock, *`clk`*.

Syntax

```
assert_no_overflow
   [#(severity_level, width, min, max, options, msg,
category)]
   instance_name (clk, reset_n, expr);
```

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of the monitored expression, expr (default is 1).

`min`

Minimum value limit for the expr at clock tick t+1 when expr == max at clock tick 't' (This value is excluded from the acceptable range). Default is 0.

`max`

Maximum value limit for the expr at clock tick t+1 when expr == max at clock tick 't' (This value is included in the acceptable range). Default is 1.

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type, default 0.

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`expr`

Expression being verified at the positive edge of *`clk`*.

assert_no_transition

This checker ensures that, when the state variable *`test_expr`* reaches a value specified by *`start_state`*, it does not transit to a state/value specified by *`next_state`*. All variables are sampled at posedge of the clock, *`clk`*.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0.)

`width`

Width of *`test_expr`*, *`start_state`*, and *`next_state`* signals (default is 1).

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

`start_state`

State value at the start. When *`test_expr`* equals this value, the evaluation starts.

`next_state`

Next state value. Once test_expr matches with *`start_state`*, *`test_expr`* should not transit to this value at the next clock tick.

---

assert_no_underflow

This checker ensures that *`test_expr`* never canges from 'min' value to a value that is less than 'min' and greater than or equal to 'max'.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0).

`width`

Width of the monitored expression, *`test_expr`*. Currently, this value is limited to 32 bits due to a Verilog limitation on the number of bits in a parameter. Default 1.

`min`

Minimum value limit for the test_expr at clock tick t+1 when test_expr == max at clock tick 't' (this value is included in the acceptable range).

`max`

Maximum value limit for the test_expr at clock tick t+1 when test_expr == max at clock tick 't' (This value is excluded from the acceptable range).

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

---

assert_odd_parity

This checker monitors for odd number of '1's in *`test_expr`* at every positive edge of the clock, *`clk`* .

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0).

`width`

Width of *`test_expr`* (default is 1).

`options`
Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
Error message that will be printed when the assertion fires.

`category`
Checker type (default is 0).

`clk`
The sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`test_expr`
Expression being verified at every positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
Cover property `cover_test_expr_change` indicates how many times `test_expr` changed.

Example

```
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg [6:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        count <= 7'b1111111;
    end
    else
        count <= count << 1;
end
/*ova bind
assert_odd_parity #(0, 7, 0, //
  "ERROR: count does not have an odd number of bits asserted",
  0, 1)
valid_count_odd (clk, reset_n, count);// Coverage Level 1
*/
endmodule
```

assert_one_cold

This checker ensures that the variable, *`test_expr`*, has only one bit low at any positive clock edge when the checker is configured for no inactive states.

The checker can also be configured to accept all bits equal to either 0 or 1 as the inactive level.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 32).

`inactive`
: Specifies the inactive state of *`test_expr`*:

inactive = 0 allows the inactive state of *`test_expr`* to be all zeros.

inactive = 1 allows the inactive state of *`test_expr`* to be all ones.

inactive = 2 (default) specifies that no inactive state is allowed.

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
The sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`test_expr`
Expression to be verified for "one cold" at the positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
Cover property `cover_test_expr_change` indicates how many times `test_expr` changed value.

`Level_2 (bit 1 set in coverage_level)`
Cover property `cover_test_expr_with_all_1` indicates how many times `test_expr` was all 1s. Enabled when `inactive == 1`.

Cover property `cover_test_expr_with_all_0` indicates how many times `test_expr` was all 0s. Enabled when `inactive == 0`.

`Level_3 (bit 2 set in coverage_level)`
Cover property `cover_test_expr_bit_is_0[i]` indicates how many times bit `i` of `test_expr` was 0 when `test_expr` changed value.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);
```

```
initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg [7:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
        count <= 8'b11111110;
    else
        count <= ((count << 1) | {7'b0000000, count[7]});
end
/* ova bind
assert_one_cold #(0, 8, 0, 0, "ERROR: count is not one-cold",
                  0 ,2) // Level 2 coverage
valid_one_cold (clk, reset_n, count);
*/
endmodule
```

assert_one_hot

This checker ensures that the variable, `test_expr`, has only one bit high at any positive clock edge.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 32).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
: The sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression to be verified for "one hot" at the positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`
: Cover property `cover_test_expr_change` indicates how many times `test_expr` changed value.

`Level_3 (bit 2 set in coverage_level)`

Cover property `cover_test_expr_bit_is_1[i]` inidcates how many times bit i of `test_expr` was 1 when `test_expr` changes value.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg [7:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
        count <= 8'b00000001;
    else
        count <= ((count << 1) | {7'b0000000, count[7]});
end
/*ova bind
```

```
assert_one_hot #(0, 8, 0, "ERROR: count is not one-hot",
           0, 5) // coverage_level = 5: Level 1 and 3 coverage
invalid_one_hot (clk, reset_n, count);
*/
endmodule
```

assert_quiescent_state

This checker verifies that the value in the variable *`state_expr`*, is equal to the value specified by *`check_value`* when a sampled positive edge is detected on *`sample_event`*.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`state_expr`* and *`check_value`* signals (default is 1).

`options`
: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`state_expr`
: Variable to be checked at every posedge of *`clk`*.

`check_value`
: Signal that holds the value to be compared with *`state_expr`* when *`sample_event`* is asserted.

`sample_event`
: Sampling trigger signal.

---

assert_range

This checker ensures that the value of *`test_expr`* will always be within the 'min' and 'max' value range.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`width`
: Width of *`test_expr`* (default is 1).

`min`
: Minimum value allowed for range check (default is 0).

`max`
: Maximum value allowed for range check. (default is 1).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`
: Sampling clock of the checker.

`reset_n`
: Signal indicating completed initialization.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

assert_time

This checker continuously monitors the *`start_event`* at every positive edge of the clock, *`clk`*. When *`start_event`* is TRUE, the checker ensures that the expression, *`test_expr`*, is TRUE up to *`num_cks`* number of clock ticks.

Syntax

```
assert_time
  [#(severity_level, num_cks, flag, options, msg, category)]
  instance_name (clk, reset_n, start_event, test_expr);
```

Arguments

`severity_level`

Severity of the failure (default is 0).

`num_cks`

Number of clock ticks for *`test_expr`* to remain TRUE after *`start_event`* is asserted.

`Flag`

0 - Ignores any asserted start_event after the first one has been detected.

1 - Restart monitoring *`test_expr`*, if start_event is asserted in any subsequent clock cycle while monitoring *`test_expr`*.

2 - Issue an error if an asserted *`start_event`* occurs in any clock cycle while monitoring *`test_expr`*.

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the assertion.

`reset_n`

Signal indicating completed initialization.

`start_event`

Starting event that triggers monitoring of *`test_expr`*.

`test_expr`

One-bit variable verified at the positive edge of *`clk`*.

---

assert_transition

This checker ensures that, when the state variable *`test_expr`* reaches the value specified by *`start_state`*, it does transit to a state/value specified by *`next_state`*.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0).

`width`

Width of *`test_expr`*, *`start_state`*, and *`next_state`* signals (default is 1).

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`test_expr`

Expression verified at the positive edge of *`clk`*.

`start_state`

Start value of *`test_expr`*. When *`test_expr`* equals this value, the veriication begins.

`next_state`

Next value. Once *`test_expr`* matches with *`start_state`*, *`test_expr`* should transit to this next value (or hold at *`start_state`*).

---

assert_unchange

This checker monitors the *`start_event`* at every positive edge of the clock, *`clk`*. When *`start_event`* is TRUE, the checker ensures that the expression, *`test_expr`* does not change its value within *`num_cks`* clocks.

Syntax

```
assert_unchange
   [#(severity_level, width, num_cks, flag, options, msg,
      category, coverage_level)]
   instance_name (clk, reset_n, start_event, test_expr);
```

Arguments

`severity_level`

Severity of the failure (default is 0).

`Width`

Width of *`test_expr`* (default is 1).

`Num_cks`

Number of clock ticks for *`test_expr`* to remain unchanged after *`start_event`* is asserted.

`Flag`

0 - Ignores any asserted *`start_event`* after the first one has been detected.

1 - Re-start monitoring *`test_expr`* if start_event is asserted in any subsequent clock while monitoring *`test_expr`*.

2 - Issue an error if an asserted *`start_event`* occurs in any clock cycles while monitoring test_expr.

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`start_event`

Starting event that triggers monitoring of *`test_expr`*.

`test_expr`

Expression verified at the positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover property `cover_start_event` indicates how many times `start_event` was asserted.

Cover property `cover_unchange` indicates how many times `test_expr` remained stable the required time interval.

`Level_3 (bit 2 set in coverage_level)`

Cover property `cover_overlapping_start_events` indicates how many times a `start_event` occurred while a previously triggered evaluation attempt was still in progress.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #1000 $finish;
end

always #20 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
```

```
reg start;
reg [3:0] expr;
integer count;

initial $monitor ("count = %d  start = %b  expr = %b \n",
count, start, expr);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        start <= 0;
        expr  <= 4'b0000;
        count <= 0;
    end
    else
        count <= count + 1;

    if (count == 3)
        start  <= 1;
    else
        start  <= 0;

    if (count == 3 || count == 4 || count == 5 || count ==
6 || count == 7 || count == 8 || count == 9)
        expr  <= 4'b0110;
    else
        expr  <= expr + 1;
end

/*ova bind
assert_unchange #(0, 4, 4, 1, 0,
"ERROR: expr not stable in the num_clk cycles after start",
   0, 3) // coverage_level = 3: Levels 1 and 2
valid_unchange (clk, reset_n, start, expr);
*/
endmodule
```

assert_width

This checker ensures that, when `test_expr` becomes TRUE it should remain TRUE at least for 'min' number of clock cycles and at most 'max' number of clock cycles. It should never remain TRUE beyond that limit.

Syntax

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

Arguments

`severity_level`
: Severity of the failure (default is 0).

`min_cks`
: *`test_expr`* should be held TRUE at least for *`min_cks`* n umber of clocks (default is 1).

`max_cks`
: *`test_expr`* should not be held TRUE for more than *`max_cks`* number of clocks (default is 1).

`options`
: Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`
: Error message that will be printed when the assertion fires.

`category`
: Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`test_expr`

Expression verified at every positive edge of *`clk`*.

assert_win_change

This checker ensures that *`test_expr`* changes its value at least once between the assertions of *`start_event`* and *`end_event`*.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0).

`width`

Width of the monitored expression, *`test_expr`*. (default is 1).

`options`

Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type, (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`start_event`

Start of the window.

`test_expr`

Expression verified at the positive edge of *`clk`*.

`end_event`

End of the window.

---

assert_win_unchange

This checker ensures that the *`test_expr`* never changes its value between the assertions of *`start_event`* and *`end_event`*.

Syntax

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

Arguments

`severity_level`

Severity of the failure (default is 0).

`width`

Width of the monitored expression, test_expr (default is 1).

`options`

Currently, the only supported option is options=1, which defines that the assertion is an assumption for formal tools.

`msg`

Error message that will be printed when the assertion fires.

`category`

Checker type (default is 0).

`clk`

Sampling clock of the checker.

`reset_n`

Signal indicating completed initialization.

`start_event`

Start of the window.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

`end_event`

End of the window.

---

assert_window

This checker ensures that *`test_expr`* is asserted 1 as long as the window is open. Window open and close events are signaled by *`start_event`* and *`end_event`* expressions. The verification starts on the next clock tick following *`start_event`*.

Syntax

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

Arguments

`severity_level`
Severity of the failure (default is 0).

`options`
Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
Error message that will be printed when the assertion fires.

`category`
Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`start_event`
Start of the window.

`test_expr`
Signal being verified at the positive edge of *`clk`*.

`end_event`
End of the window.

assert_zero_one_hot

This checker ensures that the variable, *`test_expr`*, has only one bit 1 or all bits 0 at any positive edge of the clock, *`clk`*.

Syntax

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

Arguments

`severity_level`
Severity of the failure (default is 0).

`width`
Width of *`test_expr`* (default is 32).

`options`
Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

`msg`
Error message that will be printed when the assertion fires.

`category`
Checker type (default is 0).

`clk`
Sampling clock of the checker.

`reset_n`
Signal indicating completed initialization.

`test_expr`

Expression to be verified for "one hot or all bits 0" at the positive edge of *`clk`*.

Coverage modes

`Level_1 (bit 0 set in coverage_level)`

Cover property `cover_test_expr_change` indicates how many times `test_expr` changed value.

`Level_2 (bit 1 set in coverage_level)`

Cover property `cover_test_expr_with_all_0` indicates how many times the all 0 value occurred when `test_expr` changed value.

`Level_3 (bit 2 set in coverage_level)`

Cover property `cover_test_expr_bit_is_1[i]` indicates how many times bit `test_expr[i]` was 1 after a change of value.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

child CH (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
    #40 reset_n = 1;
    #980 $finish;
end

always #20 clk = ~clk;

endmodule
```

```
module child(reset_n, clk);
input reset_n, clk;
reg [7:0] count;

initial begin
    $monitor("count = %b \n", count);
end

always @(posedge clk)
begin
    if (reset_n == 0 || count == 8'b00000000)
        count <= 8'b00000001;
    else
        count <= ((count << 1) | {7'b0, count[7]});

    if (count == 8'b10000000)
        count <= 8'b00000000;
end

/*ova bind
assert_zero_one_hot #(0, 8, 0, "ERROR: count is not one-hot",
      0, 5) // coverage_level = 5: Levels 1 and 3
valid_zero_one_hot (clk, reset_n, count);
*/
endmodule
```

3

Four-State OVA Checkers

Checkers in the OVA Checker Library, except ova_driven, ova_forbid_bool, and ova_no_contention, use boolean equality (==, !=) in their underlying assertions. While the checkers may be used in four-state simulation, they do not detect equality or inequality on x and z (a check of x == y will be false if any operand has an x or z, even if it is in the same bit position).

The following come in two versions. The default one uses boolean equality while another version, also available in the library under a different name, supports case equality (===). The latter checkers in both unit and template forms are located in files having the postfix .4state in the OVA Checker Library. Note that most formal tools, supports only synthesizable assertions and should use only the default checkers should be used (case equality (===) as in all .4state checkers is not synthesizable, whereas boolean equality is). See your documentation for information.

ova_arith_overflow	ova_hold_value
ova_const	ova_inc
ova_data_used	ova_quiescent_state
ova_dec	ova_reg_loaded
ova_delta	ova_tri-state
ova_hold	ova_timeout

To use the 4-state checkers, you define a macro of the same name as the checker in the OVA file, and then `include the checker file.

For example, to use four-state version of the std unit ova_inc, add the following two lines at the beginning of the OVA file:

```
`define inc
`include "$VCS_HOME/etc/ova/inc_u.ova.4state"
```

If you need to only the inc template, the two lines becomes:

```
`define inc
`include "$VCS_HOME/etc/ova/inc.ova.4state"
```

Index