

SystemVerilog Assertions Checker Library Quick Reference

Version C-2009.06, June 2009

Comments?

E-mail your comments about Synopsys documentation to
vcs_support@synopsys.com

or

pioneer_support@synopsys.com.



Copyright Notice and Proprietary Information

Copyright © 2008 Synopsys, Inc. All rights reserved. This software and documentation are owned by Synopsys, Inc., and furnished under a license agreement. The software and documentation may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc. for the exclusive use of _____ and its employees. This is copy number _____.”

Destination Control Statement

All technical data contained in this publication is subject to the export laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and the Synopsys logo are registered trademarks of Synopsys, Inc.

OpenVera is a trademark of Synopsys, Inc.

All other trademarks are the exclusive property of their respective holders and should be treated as such.

Printed in the U.S.A.

Document Order Number:38057-000 ZA

SystemVerilog Assertions Checker Library Quick Reference

Contents

Section 1: The SystemVerilog Assertions (SVA) Checker Library	1
Overview	1
Global Controls	2
Checker Triggering	2
Custom Reporting:	3
Shared Syntax	3
Use Mode in SystemVerilog	5
The Entire Design Targets SystemVerilog	5
Some Design Files Are Restricted to Verilog2001	5
Use Mode in VHDL in VCS MX	5

Section 2: SVA Basic Checkers	9
Checker Descriptions	9
assert_always	9
assert_always_on_edge	9
assert_change	10
assert_cycle_sequence	10
assert_decrement	10
assert_delta	10
assert_even_parity	11
assert_fifo_index	11
assert_frame	11
assert_handshake	11
assert_implication	12
assert_increment	12
assert_never	12
assert_next	13
assert_no_overflow	13
assert_no_transition	13
assert_no_underflow	13
assert_odd_parity	14
assert_one_cold	14
assert_one_hot	14
assert_proposition	14
assert_quiescent_state	15
assert_range	15
assert_time	15
assert_transition	15
assert_unchange	16
assert_width	16
assert_win_change	16
assert_win_unchange	16

assert_window	16
assert_zero_one_hot	17

Section 3: SVA Advanced Checkers 19

Checker Descriptions	19
Shared Syntax	19
assert_arbiter	19
assert_bits	19
assert_code_distance	20
assert_data_used	20
assert_driven	20
assert_dual_clk_fifo	20
assert_fifo	21
assert_hold_value	21
assert_memory_async	21
assert_memory_sync	22
assert_multiport_fifo	22
assert_mutex	23
assert_next_state	23
assert_no_contention	23
assert_packet_flow	24
assert_rate	24
assert_reg_loaded	24
assert_req_ack_unique	25
assert_req_requires	25
assert_stack	25
assert_valid_id	25
assert_value	26

Section 1: The SystemVerilog Assertions (SVA) Checker Library

The SystemVerilog Assertions (SVA) Checker Library consists of two parts:

- SVL (described in Chapter 2) consists of checkers that have the same behavior and controls as those in the Open Verification Library (OVL) - see the *Accellera Open Verification Library Reference Manual* available at <http://www.verificationlib.org/>. These checkers have additional features such as coverage in the SVA Library.
- SVA Advanced Checkers (described in Chapter 3) contains checkers that generally verify more complex behaviors. These have similar controls to the SVL checkers, but, in addition, they allow selecting the sampling clock edge(s), posedge or negedge. Coverage is also provided.

Note that the header of each checker file also contains a description of the behavior and parameters.

Overview

The checker library resides in a release in the directory

`$VCS_HOME/packages/sva/`

or

`$PIONEER_HOME/packages/sva/`

The library consists of 53 checker files having the name `assert_checker_name.v`. There is also a symbolic link to that file with the name `assert_checker_name.sv` that can be used in situations where parts of the Verilog design follow Verilog2001 syntax and should be compiled with that in mind to avoid keyword clashes with SystemVerilog keywords. In that way, the extension `.sv` identifies the checkers to be compiled with SystemVerilog, while the files with the `.v` extension will compile with Verilog2001 syntax and keywords.

The checkers have dual functions:

1. As a verification object using `assert` property statements. This form is employed for verifying the behavior as specified for the checker.
2. As a coverage objects using `cover` property and cover points implemented using the equivalent of `covergroup` statements. This form is employed to obtain information about the observed presence of certain behavioral patterns in the design - coverage of such patterns.

The two forms can coexist or be enabled independently as described later - See “Global Controls” on page 2. and “Shared Syntax” on page 3.

Note also that there is header file `sva_std_task.h` in the directory. It contains user-modifiable reporting tasks and is included in each checker using `include`.

Global Controls

The following symbols are macro's defined using ``define` and apply to all instances of the checkers.

`ASSERT_ON`

When this macro is defined, the assertions and related variables in checkers are included. I.e., the checking functionality is enabled.

`COVER_ON`

When this macro is defined, the coverage items in the checkers are included. I.e., the coverage functionality is enabled. Note that additional per-instance control is provided using the checker parameters `coverage_level_1`, `coverage_level_2`, and `coverage_level_3`.

`ASSERT_INIT_MSG`

When this macro is defined, the all checker instances will output a message to stdout to indicate their hierarchical name.

`ASSERT_GLOBAL_RESET`

When this macro is defined its defining expression is taken as the reset signal for all the checkers. Otherwise, the `reset_n` port signal is used as reset.

`ASSERT_MAX_REPORT_ERROR`

When this global macro variable is defined, the assertion instance stops reporting messages if the number of errors for that instance is greater than the value defined by the macro. Using this macro as described requires modifying the reporting tasks in `sva_std_task.h`.

`SVA_CHECKER_INTERFACE`

If not defined, the interface is a SystemVerilog "module". If defined then the checker interface is "interface". The parameters and ports remain the same in both cases.

`SVA_CHECKER_NO_MESSAGE`

Causes the std tasks in `sva_std_task.h` file that is included in every checker will skip over any user generated messages. So to disable all message, use the `-assert quiet` option and define the above symbol.

Checker Triggering

In the SVL library (Chapter 2) all but one checker, `assert_proposition`, is triggered at the positive edge of a triggering signal or expression `clk`.

In the Advanced Checker Library (Chapter 3), all the checkers can select the sampling edge(s) using a parameter. It can be either the positive or the negative edge of the clock port expression.

The values of all actual signals or expressions in the ports of the checkers are sampled just before the sampling edge of `clk`. Therefore, any pulses happening on the signals / expressions between

consecutive sampling edges of clk are not observed by the checkers.

Moreover, whenever an edge of a signal / expression on a port of a checker other than the clock is used in the checker, the value is the sampled form of the edge as detected by looking at two consecutive samples of the signal.

The checker `assert_proposition` monitors an expression at all times and triggers by a change of its value to 0.

Custom Reporting:

As in OVL, if you wish to have more elaborate error reporting, counting, etc., you must edit the `sva_std_task.h` file that is distributed with the library and also resides in the directory

`$VCS_HOME/packages/sva/`

or

`$PIONEER_HOME/packages/sva/`

This file is automatically `#include`-ed in each checker. If the location of the header file is changed, the `#incdir` directive must also be modified accordingly.

Shared Syntax

`severity_level`

Severity of the failure with default value of 0. Currently SystemVerilog does not provide support of severity levels through a parameter or a variable, hence this parameter is provided only for compatibility with older versions of the checkers. However by modifying the error tasks in the `sva_std_task.h` file appropriately, severity level can be taken into account. For example, the simulation can be terminated when a failure of an assertion occurs and the checker severity level is set to some specific value.

`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).¹

`msg`

Error message that will be printed when the assertion fires using the default definition of the `sva_checker_error` task that is included in the `sva_std_task.h` file. Its output can also be controlled by the macro `SVA_CHECKER_NO_MESSAGE`.

`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.

1. Note that Magellan does not use options to determine the role (assume or assert) of the assertions.

`coverage_level_i`

Specifies which coverage levels should be enabled (provided that the symbol `COVER_ON` is defined) by specifying parameters. There are three coverage levels controlled by parameters `coverage_level_1`, `coverage_level_2`, and `coverage_level_3`. The bits in each of the parameters when set to 1 enable some specific coverage point, one per bit. The number of bits used thus depends on the number of cover points in the checker at each level and varies from checker to checker. Generally, the following levels are supported (default is 1):

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

Level 1 coverages are enabled by setting bits in coverage levels to 1.

Level 1 coverages are enabled by setting bits in `coverage_level_1` to 1.

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

Level 2: Usually includes data coverage on ranges of values. These are coverage items which show certain interesting activities of the RTL behavior in the form of statistics. Used in simulation only but may also include cover properties on some particular sequences that can be of interest as search goals for formal tools.

Level 2 coverages are enabled by setting bits in `coverage_level_2` to 1.

Example: Range of latency values classified on at-least-once basis.

Level 3: Mostly cover property coverage of specific corner points as specified. Used primarily by formal tools as goals. These cover properties can be enabled in simulation too, however, the same information can be obtained from Level 2 range coverages in the extreme bins. These coverage items ensure that the corner case condition of the RTL/design block are verified during testing.

Level 3 coverages are enabled by setting bits in `coverage_level_3` 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.

`inst_name`

Instance name of assertion monitor.

`clk`

Sampling clock of the assertion.

`reset_n`

Signal indicating completed initialization when set to 1. Note that `reset_n` is used as an asynchronous reset so that glitches on the sequence or expression may cause the checker to reset.

Use Mode in SystemVerilog

The Entire Design Targets SystemVerilog

To use the SVA Checker Library in VCS with a design written in SystemVerilog (which thus respects its enlarged set of keywords), you instantiate or bind any number of the checkers in the design and indicate the placement of the library using the Verilog library mechanism as follows:

```
vcs -sverilog +define+ASSERT_ON <design files and options> \
-y $VCS_HOME/packages/sva +libext+.v \
+incdir+$VCS_HOME/packages/sva
```

Note that `+define+ASSERT_ON` on the VCS command line is necessary to turn on the assertions in all checker instances. Alternately or in addition, by defining `COVER_ON` the coverage functionality can be included (it can be controlled at the instance level by the checker parameter `coverage_level_i`, where $i = 1, 2$, or 3).

To gather coverage information using the coverage functionality of the checkers, do not include the `-cm assert` option. This is because coverage on `cover` `property` and `covergroup` statements is automatically turned on in VCS.

Some Design Files Are Restricted to Verilog2001

In this case, the Verilog2001 files have names with the `.v` extensions, while all SystemVerilog files, including the checkers, have a name with the `.sv` extension. In the case of the checkers, it is the symbolic link with the `.sv` extension that is used. The vcs compilation command line will then have the following form:

```
vcs +define+ASSERT_ON <design files and other options> \
-y $VCS_HOME/packages/sva +libext+.sv -sverilog \
+verilog2001ext+.v+incdir+$VCS_HOME/packages/sva
```

The compiler uses Verilog2001 syntax for all `.v` files, and SystemVerilog syntax for all other files including those with the `.sv` extension.

Use Mode in VHDL in VCS MX

The SVA Checker Library `.v` files can be found in the `$VCS_HOME/packages/sva/` directory.

There is also a VHDL package called `sva_lib` in this directory, containing the component definitions for all the checkers in the library. The name of the file is `component.sva_v.vhd`.

To use the SVA library in VHDL, the user should proceed as follows:

1. Compile the component package in a library. For example, suppose that the default `WORK` library used. The command to compile the package is then:

```
vhdlan $VCS_HOME/packages/sva/component.sva_v.vhd
```

2. Compile the selected SVA checkers using `vlogan` and deposit in a VHDL library. For example, suppose again that the default `WORK` library is used and that we wish to compile all the checkers in the library (it takes a fraction of a second to complete), then the command is:

```
vlogan +vc -sverilog $VCS_HOME/packages/sva/*.v \
      +incdir+$VCS_HOME/packages/sva \
      +define+ASSERT_ON
```

Other macro definitions can be supplied with `+define`.

Note that if you choose to compile only some specific checkers that are described in Chapter 2, you also must compile the file `sva_std_edge_select.v`. This is because these checkers allow to specify the sampling clock edge.

3. To use the compiled checkers in the VHDL design, it must include commands to use the following libraries:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library WORK;
use WORK.sva_lib.all;
```

and instantiate the checker component(s) as any other VHDL entity.

Note that the signals on the checker ports are of the IEEE `std_ulogic` and unsigned types.

Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library WORK;
use WORK.sva_lib.all;

entity top is
  generic (P1 : integer := 7;
          P2 : integer := 31);
end top;

architecture top_arch of top is

  signal a,b,c : std_ulogic_vector(P1 downto 0);
  signal clk : std_ulogic;
  component child is
    generic (p : integer := 10);
    port (in1, in2 : std_ulogic_vector(p downto 0);
          clk : std_logic;
          out1 : out std_ulogic_vector(p downto 0));
  end component;
```

```

begin

-- Some design entity instance
i1 : child generic map(P1) port map(a,b,clk,c);

-- sva assert_always checker
always_inst : assert_always port map(clk, '1', a(1));

-- sva assert_cycle_sequence checker
seq_inst : assert_cycle_sequence
    generic map (0, P1+1, 0, 0, "a3")
    port map(clk, '1', a);

process
begin
    clk <= '0';
    wait for 3 ns;
    clk <= '1';
    wait for 3 ns;
end process;
end top_arch;

configuration cfg_top of top is
    for top_arch
        end for;
end;

```

Section 2: SVA Basic Checkers

This chapter contains 31 OVL-like checkers. All checkers, except `assert_proposition`, are triggered at the positive edge of a triggering signal or expression `clk`.

The values of all actual signals or expressions in the ports of the checkers are sampled just before the positive edge of `clk`. Therefore, any pulses happening on the signals / expressions between consecutive positive edges of `clk` are not observed by the checkers.

Moreover, whenever an edge of a signal / expression on a port of a checker other than the clock is used in the checker, it is the sampled form of the edge as detected by looking at two consecutive samples of the signal.

The checker `assert_proposition` monitors an expression at all times and triggers at the falling edge of the `test_expr`.

Checker Descriptions

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

assert_always

This checker continuously monitors `test_expr` at every positive edge of clock, `clk`. It verifies that `test_expr` will always evaluate to *true*. If `test_expr` evaluates to *false*, the assertion will fire. The `test_expr` can be any valid expression.

Syntax

```
assert_always [#(severity_level, options, msg,  
category, coverage_level_1, coverage_level_2,  
coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_always_on_edge

This checker continuously monitors `test_expr` at every positive edge of clock, `clk`. It verifies that `test_expr` will always evaluate to *true*. If `test_expr` evaluates to *false*, the assertion will fire. The `test_expr` can be any valid expression.

Note that the transition on the sampling event is as determined by sampling `sampling_event` at two consecutive clock ticks.

Syntax

```
assert_always_on_edge [#(severity_level, edge_type,  
options, msg, category, coverage_level_1,  
coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, sampling_event, test_expr);
```

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.

Syntax

```
assert_change [#(severity_level, width, num_cks, flag,  
              options, msg, category, coverage_level_1,  
              coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr);
```

assert_cycle_sequence

This checker verifies the following conditions:

- When *necessary_condition* = 0, if all *num_cks*-1 first bits of a vector of boolean events (*event_sequence[num_cks-1:1]*) are *true* (1) in consecutive clock cycles, the last boolean (*event_sequence[0]*) should be *true* in the next clock cycle.
- When *necessary_condition* = 1, if the first bit of a vector of (*event_sequence[num_cks-1]*) is *true*, then all the remaining *event_sequence[num_cks-2:0]* bits should become true in the subsequent *num_cks*-1 clock cycles.

Syntax

```
assert_cycle_sequence [#(severity_level, num_clks,  
                        necessary_condition,  
                        options, msg, category,  
                        coverage_level_1, coverage_level_2,  
                        coverage_level_3)]  
inst_name (clk, reset_n, event_sequence);
```

assert_decrement

This checker continuously monitors *test_expr* at every positive edge of clock signal, *clk*. It checks that the *test_expr* will never decrease 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 tick after *reset_n* is asserted.

Syntax

```
assert_decrement [#(severity_level, width, value,  
                   options, msg, category, coverage_level_1,  
                   coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_delta

This checker continuously monitors *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. The *test_expr* can be any valid expression. The check will not start until the first clock after *reset_n* is asserted.

Syntax

```
assert_delta [#(severity_level, width, min, max,  
              options, msg, category, coverage_level_1,  
              coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_even_parity

This checker continuously monitors *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_1,  
                     coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_fifo_index

This checker ensures that a FIFO element a) never overflows and underflows b) allows/disallows simultaneous push and pop.

Syntax

```
assert_fifo_index [#(severity_level, depth,  
                    push_width, pop_width, options, msg,  
                    category, coverage_level_1,  
                    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, push, pop);
```

assert_frame

This checker validates proper cycle timing relationships between two events in the design. When a *start_event* (a bit) evaluates *true*, then *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, coverage_level_1,  
               coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr);
```

assert_handshake

This checker continuously monitors the *req* and *ack* signals at every positive edge of the clock *clk*. It ensures that *ack* occurs after *req* within a specified minimum and maximum number of clock cycles. Both *req* and *ack* must go inactive prior to starting a new cycle. Verifying that *req* is persistent until *ack* arrives and that it remains active for some cycle after *ack* is controlled by checker parameters.

Syntax

```
assert_handshake [#(severity_level, min_ack_cycle,
    max_ack_cycle, req_drop, deassert_count,
    max_ack_length, options, msg, category,
    coverage_level_1, coverage_level_2,
    coverage_level_3)]
instance_name (clk, reset_n, req, ack);
```

assert_implication

This checker continuously monitors *antecedent_expr*. If it evaluates to *true*, then this checker will verify that the *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.

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, coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, antecedent_expr,
    consequent_expr);
```

assert_increment

This checker continuously monitors *test_expr* at every positive edge of the triggering event, *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 expression. The check will not start until the first clock after *reset_n* is asserted.

Syntax

```
assert_increment [#(severity_level, width, value,
    options, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_never

This checker continuously monitors *test_expr* at every positive edge of the triggering event or clock *clk*. It verifies that *test_expr* will never evaluate *true*. The *test_expr* can be any valid expression. When *test_expr* evaluates *true*, this checker will fire.

Syntax

```
assert_never [#(severity_level, options, msg,
    category)]
inst_name (clk, reset_n, test_expr);
```

assert_next

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* exactly *num_cks* number of clock cycles later.

This checker supports overlapping sequences. For example, if you assert that *test_expr* should 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,  
    coverage_level_1, coverage_level_2,  
    coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr);
```

assert_no_overflow

This checker continuously monitors *test_expr* at every positive edge of the triggering event or clock *clk*. This assertion verifies that a specified *test_expr* will never:

- Change value from a *max* value (default is $(2^{width}) - 1$) to a value greater than *max*, or
- Change value from a *max* value (default is $(2^{width}) - 1$) to a value less than or equal to a *min* value (default is 0).

Syntax

```
assert_no_overflow [#(severity_level, width, min, max,  
    options, msg, category, coverage_level_1,  
    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_no_transition

This checker continuously monitors *test_expr* at every positive edge of the triggering event, positive *clk* edge. When this variable evaluates to the value of *start_state*, the monitor ensures that *test_expr* will never transition to the value of *next_state*. The *width* parameter defines the number of bits in *test_expr*.

Syntax

```
assert_no_transition [#(severity_level, width,  
    options, msg, category, coverage_level_1,  
    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr, start_state,  
    next_state);
```

assert_no_underflow

This checker continuously monitors *test_expr* at every positive edge of the triggering event or clock *clk*. This checker verifies that *test_expr* will never:

- Change value from a *min* value (default is 0) to a value less than *min*, or

- Change to a value greater than or equal to *max* (default is $(2 \times \text{width}) - 1$).

Syntax

```
assert_no_underflow [#(severity_level, width, min,
    max, options, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_odd_parity

This checker monitors for odd number of 1's in *test_expr* at every positive edge of the clock, *clk*. Syntax

Syntax

```
assert_odd_parity [#(severity_level, width, options,
    msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_one_cold

This checker ensures that the variable, *test_expr*, has only one bit set to 0 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_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_one_hot

This checker ensures that the variable, *test_expr*, has only one bit set to 1 at any positive clock edge, *clk*.

Syntax

```
assert_one_hot [#(severity_level, width, options,
    msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_proposition

This checker continuously monitors *test_expr*. Hence, this assertion is unlike *assert_always*; that is, *test_expr* is not being sampled by a clock. It verifies that *test_expr* will always evaluate *true*. If *test_expr* transits from *true* to *false* while *reset_n* is 1, it will fire.

Syntax

```
assert_proposition [#(severity_level, options, msg,
    category)]
inst_name (reset_n, test_expr);
```

assert_quiescent_state

This checker verifies that the value in the variable, *state_expr*, is equal to the value specified by *check_value* and optionally at the end of simulation. Verification occurs when a sampled positive edge is detected on *sample_event*.

Syntax

```
assert_quiescent_state [#(severity_level, width,
    options, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, state_expr, check_value,
    sample_event);
```

assert_range

This checker continuously monitors *test_expr* at every positive edge of the triggering event, *clk*. The checker ensures that the value of *test_expr* will always be within the *min* and *max* value range. The *min* and *max* should be a valid parameter and *min* must be less than or equal to *max*.

Syntax

```
assert_range [#(severity_level, width, min, max,
    options, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

assert_time

This checker continuously monitors *start_expr*. When this signal (or expression) evaluates *true*, the checker ensures that *test_expr* evaluates to *true* for the next *num_cks* number of clock cycles.

Syntax

```
assert_time [#(severity_level, num_cks, flag, options,
    msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, start_event, test_expr);
```

assert_transition

This checker continuously monitors *test_expr* at every positive edge of the triggering event, *clk*. When *test_expr* evaluates to the value of *start_state*, the assertion monitor ensures that *test_expr* will always change to the value of *next_state*. The *width* parameter defines the number of bits in *test_expr*.

Syntax

```
assert_transition [#(severity_level, width, options,
    msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr, start_state,
    next_state);
```

assert_unchange

This checker continuously monitors *start_event* at every positive edge of the triggering event, *clk*. When this signal (or expression) evaluates *true*, the checker ensures that *test_expr* will not change value within the next *num_cks* number of clock cycles.

Syntax

```
assert_unchange [#(severity_level, width, num_cks,  
    flag, options, msg, category, coverage_level_1,  
    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr);
```

assert_width

This checker continuously monitors *test_expr*. When this signal (or expression) evaluates *true*, it ensures that *test_expr* evaluates to *true* for a specified *minimum* number of clock cycles and does not exceed a *maximum* number of clock cycles.

Syntax

```
assert_width [#(severity_level, min_cks, max_cks,  
    options, msg, category, coverage_level_1,  
    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

assert_win_change

This checker continuously monitors *start_event* at every positive edge of the triggering event, *clk*. When this signal (or expression) evaluates *true*, it ensures that *test_expr* changes values prior to and including the occurrence of *end_event*.

Syntax

```
assert_win_change [#(severity_level, width, options,  
    msg, category, coverage_level_1,  
    coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr,  
    end_event);
```

assert_win_unchange

This checker continuously monitors *start_event* at every positive edge of the triggering event, *clk*. When this signal (or expression) evaluates *true*, it ensures that *test_expr* will not change in value up to and including *end_event* becoming true.

Syntax

```
assert_win_unchange [#(severity_level, width,  
    options, msg, category)]  
inst_name (clk, reset_n, start_event, test_expr,  
    end_event);
```

assert_window

This checker continuously monitors *start_event* at every positive clock edge *clk*. When *start_event* evaluates *true*, it ensures that

the *test_expr* evaluates *true* at every successive positive clock edge of *clk* up to and including the *end_event* expression becoming *true*.

Note: This assertion does not evaluate *test_expr* on *start_event*, it begins evaluating at the next positive clock edge of *clk*.

Syntax

```
assert_window [#(severity_level, options, msg,  
               category, coverage_level_1,  
               coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, start_event, test_expr,  
          end_event);
```

assert_zero_one_hot

This checker continuously monitors *test_expr* at every positive edge of the triggering event, *clk*. It verifies that *test_expr* has exactly one bit asserted or no bit asserted.

Syntax

```
assert_zero_one_hot [#(severity_level, width,  
                      options, msg, category, coverage_level_1,  
                      coverage_level_2, coverage_level_3)]  
inst_name (clk, reset_n, test_expr);
```

Section 3: SVA Advanced Checkers

This section describes 22 checkers. These advanced checkers use the same controls as the OVL-equivalent checkers in the previous section. In addition, they have a clock edge selection parameter, *edge_expr*, that allows the user to select *posedge* or *negedge* clock edge selection for sampling in the assertions and cover statements.

Checker Descriptions

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

Shared Syntax

Many checkers share the same syntax elements. In addition to using *severity_level*, *reset_n*, *clk*, and *msg* described in Chapter 1, the checkers in this chapter can select the active edge the clock:

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

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

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

assert_arbiter

This checker ensures that a resource arbiter provides grants to corresponding requests within *min_lat* and *max_lat* cycles. *reqs* and *grants* are vectors of size *[no_chnl-1:0]* where the bits correspond to the individual channels. They are assumed to be 1 when active. The checker can verify a priority arbitration scheme alone or in conjunction with (as a secondary criterion) round robin, fifo or LRU selection algorithms. The checks are not enabled unless *reset_n* evaluates true.

Syntax

```
assert_arbiter [#(severity_level, no_chnl, bw_prio,
    grant_one_chk, req_priority_chk,
    arbitration_rule, min_lat, max_lat,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, reqs, req_priority, grants);
```

assert_bits

This checker ensures that the value of *exp* has between *min* and *max* number of bits that are asserted or deasserted as indicated by the *deasserted* flag. The check is not enabled unless *reset_n* evaluates true.

Syntax

```
assert_bits [#(severity_level, min, max, deasserted,
    exp_bw, edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, exp);
```

assert_code_distance

This checker ensures that when *exp* changes, the number of bits that are different compared to *exp2* are at least *min* but no more than *max* in number (that is, it verifies that the Hamming distance is within that interval). The check is not enabled unless *reset_n* evaluates true.

Syntax

```
assert_code_distance [#(severity_level, min, max, bw,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name ( clk, reset_n, exp ,exp2);
```

assert_data_used

This checker ensures that data from *src* [*sleft*:*sright*] appears in *dest* [*dleft*:*dright*] within the window specified as *start* cycles from after the time *trigger* is asserted until *finish* number of cycles after *trigger* is asserted.

Syntax

```
assert_data_used [#(severity_level, sleft, sright,
    dleft, dright, start, finish, edge_expr, msg,
    category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, trigger, src, dest);
```

assert_driven

This checker ensure that all bits of *exp* are driven (none are 'Z' or 'X'). The check is not enabled unless *reset_n* evaluates true.

Syntax

```
assert_driven [#(severity_level, bw, edge_expr, msg,
    category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_en, exp)
```

assert_dual_clk_fifo

This is a checker for a dual-clock, single in- and single out-port FIFO. It assume that enqueue is enabled when *enq* is asserted at the active clock edge of *enq_clk* and effectively occurs *enq_lat* cycles later. Dequeue is enabled when *deq* is asserted at the active edge of *deq_clk* and effectively occurs *deq_lat* cycles later. It can verify that neither overflow or underflow of the queue occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue.

Syntax

```
assert_dual_clk_fifo [#(severity_level, depth, elem_sz,
    hi_water_mark, enq_lat, deq_lat, oflow_chk,
    uflow_chk, value_chk, pass_thru, enq_edge_expr,
    deq_edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (reset_n, enq_clk, enq, enq_data, deq_clk,
    deq, deq_data);
```

assert_fifo

This is a checker for a single-clock, single in- and single out-port FIFO. All signals are sampled at the active edge of the clock, *clk*. It assume that enqueue is enabled when *enq* is asserted and effectively occurs *enq_lat* cycles later. Dequeue is enabled when *deq* is asserted and effectively occurs *deq_lat* cycles later. It can verify that neither overflow or underflow of the queue occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue. Also, if *pass_thru* is 1 it allows simultaneous enqueue and dequeue of data on empty or full queue. Otherwise a dequeue on an empty queue will report an underflow.

Syntax

```
assert_fifo [#(severity_level, depth, elem_sz,
    hi_water_mark, enq_lat, deq_lat, oflow_chk,
    uflow_chk, value_chk, pass_thru, edge_expr, msg,
    category], coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, enq, enq_data, deq, deq_data);
```

assert_hold_value

This checker ensure that *exp* of width *bw* remains at *value* for *min* to *max* number of cycles. That is, it must stay at *value* for *min* cycles, then it may change and after *max* cycles it must change to some other value. The check is not enabled unless *reset_n* evaluates *true*.

Syntax

```
ova_hold_value [#(severity_level, min, max, bw,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, exp, value);
```

assert_memory_async

This checker ensures the integrity of an asynchronous memory contents and accesses. When *addr_chk* evaluates *true*, it 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*. All other checks apply only if the address is valid. There is no clock other than the *ren* and *wen* expressions that indicate when each operation is to take place by their falling edges. Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or

similarly that there is at least one write between two consecutive reads to an address. The checker can also verify that the value written last to a memory location is the one being read out later.

Syntax

```
assert_memory_async [#(severity_level, data_bits,
    addr_bits, mem_sz, addr_chk, init_chk, readl_chk,
    writel_chk, value_chk, w_edge_expr, r_edge_expr,
    msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (reset_n, start_addr, end_addr, ren, raddr,
    rdata_wen, waddr, wdata);
```

assert_memory_sync

This checker ensures the integrity of a synchronous memory contents and accesses. When *addr_chk* evaluates true, it ensures that *start_addr* <= *raddr* <= *end_addr* when *ren* is true as sampled by the active edge of *rclk*, and that *start_addr* <= *waddr* <= *end_addr* when *wen* is true at the active edge of *wclk*. All other checks apply only if the address is valid. Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or similarly that there is at least one write between two consecutive reads to an address. The occurrence of simultaneous read and write operation when *rclk* is the same as *wclk* can be verified. The checker can also verify that the value written last to a memory location is the one being read out later or at the same time if *pass_thru* is enabled.

Syntax

```
assert_memory_sync [#(severity_level, data_bits,
    addr_bits, mem_sz, addr_chk, init_chk, conflict_chk,
    pass_thru, readl_chk, writel_chk, value_chk, w_edge_expr,
    r_edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (start_addr, end_addr, ren, raddr,
    rclk rdata, wen, waddr, wclk, wdata);
```

assert_multiport_fifo

This checker implements a checker for a single-clock, multi-port in and multi-port out queue. *enq* and *deq* are bit vectors of equal size *no_ports*. Each pair of corresponding bits in these vectors defines the enqueue and dequeue enable signals for a fifo port. Their priority is such the bit 0 is the lowest priority, the highest order bit *no_ports-1* is the highest priority. The enqueue port and the dequeue port of the highest priority are processed at every active *clk* edge. *enq_data* is a concatenation of the data from the different ports, dimensioned as [*no_ports*elem_size-1:0*], with data vectors appearing in the same order as the *enq* requests. Whenever a bit in *enq* is asserted 1, the corresponding data part in *enq_data* must be valid after *enq_lat* clock cycles. Only the highest priority data is actually enqueued. *deq_data* is a concatenation of the data from the different ports. It is assumed that it is dimensioned the same way as *enq_data*, with data vectors appearing in the same order as

the *deq* requests. Whenever a bit in *deq* is asserted 1, the corresponding data part in *deq_data* must be valid after *deq_lat* clock cycles. Only the data of the highest priority dequeue request is compared with the reference data when *value_chk* is 1. Overflow, underflow, watermark, value and pass-thru checks can be enabled as in the *assert_fifo* checker.

Syntax

```
assert_multiport_fifo [#(severity_level, depth,
    elem_sz, no_ports, hi_water_mark, enq_lat,
    deq_lat, oflow_chk, uflow_chk, value_chk,
    pass_thru, edge_expr, msg, category,
    coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, enq, enq_data, deq, deq_data);
```

assert_mutex

This checker ensures that *a* and *b* never evaluate *true* at the same time. The check is not enabled unless *reset_n* evaluates *true*.

Syntax

```
assert_mutex [#(severtiy_level, edge_expr, msg,
    category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, a, b);
```

assert_next_state

This checker ensures that when *exp* is in current state *cs* that *exp* will transition to one of the specified legal next states in *ns*. *no_ns* specifies the number of legal next states. *ns* is a bitvector of the concatenated legal state values which *exp* can transition to from *cs*.

Syntax

```
assert_next_state [#(severtiy_level, no_ns, width,
    min_hold, max_hold, disallow, edge_expr, msg,
    category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, exp, cs, ns);
```

assert_no_contention

This checker ensures that *bus* always has a single active driver and that there is no 'X' or 'Z' on the bus when driven (*en_vector* != 0). The total number of *en_vector* bits that are asserted can be at most 1. *min_quiet* and *max_quiet* define an interval in the number of clock cycles within when the bus may remain quiet, i.e., no driver enabled.

Syntax

```
assert_no_contention [#(severtiy_level, min_quiet,
    max_quiet, bw_en, bw_bus, edge_expr, msg, category,
    coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, en_vector, bus1);
```

assert_packet_flow

The `assert_packet_flow` checker continuously monitors `sop` and `eop` at every positive edge of clock, `clk`. It does the following checks on the sequencing of `sop` and `eop`.

`assert_no_eop_till_sop`

After `eop` is issued or just after reset is deasserted, no `eop` should be asserted till `sop` is asserted.

`assert_no_sop_til_eop`

After `sop` is issued, no `sop` should be asserted till `eop` is asserted.

Syntax

```
assert_packet_flow [#(severity_level, msg, category,
    coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, sop, eop);
```

assert_rate

The checker continuously monitors `test_expr` and `trigger` at every positive edge of the clock. Once `trigger` is sampled asserted, the checker will make sure that `test_expr` is remain high intermittently or continuously for `[min:max]` clock cycles within `period` clock cycles.

Syntax

```
assert_rate [#(severity_level, min, max, period, msg,
    category, coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, trigger, test_expr);
```

assert_reg_loaded

This checker ensures that the register `dst_reg` is loaded with `src` data. The check for `dst_reg` holding the memorized value of `src` starts with `delay` cycles (minimum 1 which is default) after the `trigger` condition evaluates `true` and within `end_cycle` cycles after the `trigger` evaluates `true` or when `stop` becomes `true` (whichever occurs first).

Syntax

```
assert_reg_loaded [#(severtiy_level, delay,
    end_cycle, bw, edge_expr, msg, category,
    coverage_level_1, coverage_level_2,
    coverage_level_3)]
inst_name (clk, reset_n, trigger, src, dst_reg, stop);
```

assert_req_ack_unique

This checker verifies that each *req* receives an *ack* within the specified interval *min_time* and *max_time* active clock edges of *clk*. The arriving *ack*'s are attributed to *req*'s in a fifo manner.

Syntax

```
assert_req_ack_unique [#(severtiy_level, min_time,
    max_time, max_time_log_2, version,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, req, ack)
```

assert_req_requires

This checker ensures that if the *trig_req* expression evaluates true then the *follow_re*> expression and *follow_resp* expression will evaluate true (in sequence) before the *trig_resp* expression evaluates true. The check is not enabled unless *reset_n* evaluates true.

```
assert_req_requires [#(severity_level, min_lat,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, trig_req, follow_req,
    follow_resp, trig_resp);
```

assert_stack

This checker verifies operations on a stack. When *push* is asserted 1 it ensures that there is no stack overflow. *push_lat* specifies the number of clock cycles between the assertion of *push* and when *push_data* is valid. Similarly, when *pop* is asserted 1 it ensures that the operation is not on an empty stack. *pop_lat* specifies the number of clock cycles between the assertion of *pop* and when *pop_data* must be valid. Data value, stack empty, full, watermark and pass-thru checks can be selectively enabled.

Syntax

```
ova_stack [#(severtiy_level, depth, elem_sz,
    hi_water_mark, push_lat, pop_lat, value_chk,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, push, push_data, pop,
    pop_data);
```

assert_valid_id

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* validated by *ret_sig* asserted 1 within [*min_lat*:*max_lat*] latency. A *reset_sig* asserted true with *reset_id* value of one of the currently issued and still outstanding id's resets that outstanding id to empty, i.e., a *ret_sig* asserted for the id is then considered as invalid until newly issued. The bit width *id_bw* of the id's can be any value supported by the tool, however, the maximum number of outstanding id's at any time is limited by the

value of the parameter *max_ids*. For a given id, there can be at most *max_out_per_id* outstanding issues. The arriving returns of that id are matched in a fifo manner to the requests when verifying the latency of the return (similarly as in the *assert_req_ack_unique* checker).

Syntax

```
assert_valid_id [ (#severity_level, id_bw, max_ids,
    max_out_ids, max_out_per_id, min_lat,
    max_lat, edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, issued_sig, issued_id,
    ret_sig, ret_id, reset_sig, reset_id);
```

assert_value

This checker ensures that *exp* can only be one of the specified values in a set. *no_vals* indicates the number of values in the set which is defined by a bitvector *vals* of width $[bw * no_vals - 1 : 0]$ of the concatenated values of *bw* bits each that *exp* must evaluate to.

Syntax

```
assert_value [ #(severity_level, no_vals, disallow, bw,
    edge_expr, msg, category, coverage_level_1,
    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, exp, vals);
```