SystemVerilog Assertions Checker Library Reference Manual

Version C-2009.06
June 2009

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

SYNOPSYS®

Copyright Notice and Proprietary Information

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 control 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.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

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 compiler 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 -the sections “Global Controls” and “Shared Syntax” .

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, then 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`
: When defined it eliminates reporting the user message when a assertion fails. It doesn not control the default failure message from the simulator.

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.

Viewing Coverage Results

See the *VCS User Guide* or the *Pioneer User Guide* for information on viewing coverage results.

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.

`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_CHCKER_NO_MESSAGE`.

`category`

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

Specifies the category of the assertion (Default = 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.

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 = Coverage Level 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_level_1` to 1.

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

L*evel 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: 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.

Using Coverage Level to Report Checker Results

As described in the previous section, you use *coverage_level_i* to generate coverage reports of checker results. This section describes how to generate and view those reports.

Generating a Report

To generate coverage level reports, you set the bits for the desired levels of coverage to 1 when specifying a checker. The checker descriptions in chapters 2 and 3 describe the coverage levels available for each checker.

To generate a report:

1. Instantiate or bind the checker in your design as described in the sections "Use Mode in SystemVerilog" and "Use Mode in VHDL (VCS MX)" .
2. Set the bits for desired coverage to 1 when specifying the checker.

   For example, in the checker `assert_fifo_index` below, `coverage_level_1` and `coverage_level_2` are specified to have the two `coverage_level_1` properties and the three `coverage_level_2` properties reported.

   ```
   assert_fifo_index #(0, 4, 1, 1, 1, "ERROR: Overflow or
   Underflow in fifo", 0, 3, 7, 31)
   invalid_fifo (clk, reset_n, push_sig, pop_sig)
   ```

   Note the following:

- The default for `coverage_level_1` is on, so it is not necessary to specify `coverage_level_1` to generate results. However, since `coverage_level_1` has two bits, specifying 3 sets both bits to 1.
- Similarly, `coverage_level_2` has three bits, so specifying 7 sets all three bits to 1.
- `coverage_level_3` has four bits, so specifying 31 sets all four bits to 1..

3. Compile the design as described in the sections "Use Mode in SystemVerilog" and "Use Mode in VHDL (VCS MX)" .
4. Run the simulation.
5. Generate the report by entering:

   ```
   assertCovReport -cm_assert_dir coverage_directory_name
   ```

   Your results are generated.

Viewing Results

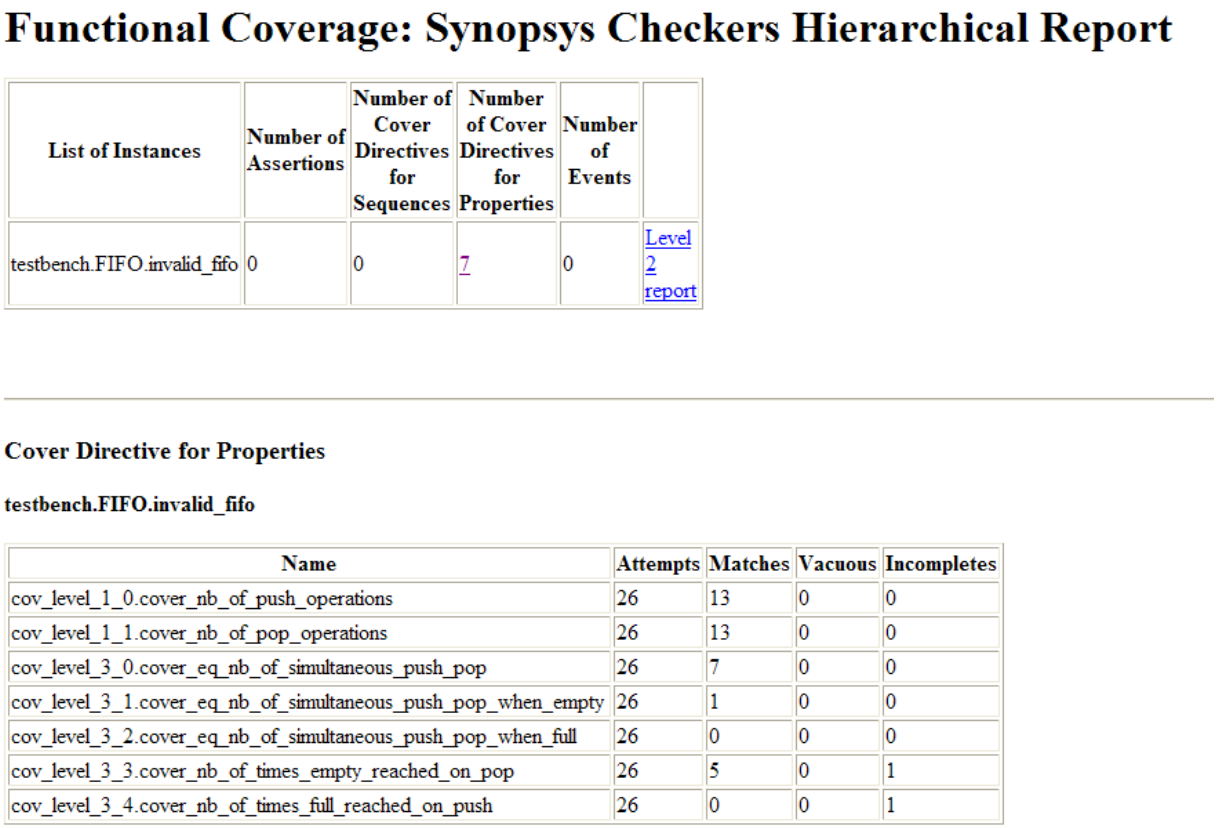
Results are generated as html files in the subdirectory report.fcov in the coverage .vdb directory you specified in the previous section. You can access the results from the file report.index.html in the .vdb directory or navigate to the subdirectory and view the files directly. The following files are generated:

- category.html displays results by category.
- hier.html displays the hierarchical results
- *instance_name*__vmm_details.html shows level 2 coverage results.

- tests.html shows the tests results.
- vmm_hier.html is the hierarchical report.

Figure 1-1 is an example of a hierarchical report displaying a summary of instance coverage results and a summary of cover directives generaed by `coverage_level_1` and `coverage_level_3`.

Figure 1-1 Results for coverage levels 1 and 3



Functional Coverage: Synopsys Checkers Hierarchical Report

List of Instances	Number of Assertions	Number of Cover Directives for Sequences	Number of Cover Directives for Properties	Number of Events	
testbench.FIFO.invalid_fifo	0	0	7	0	Level 2 report

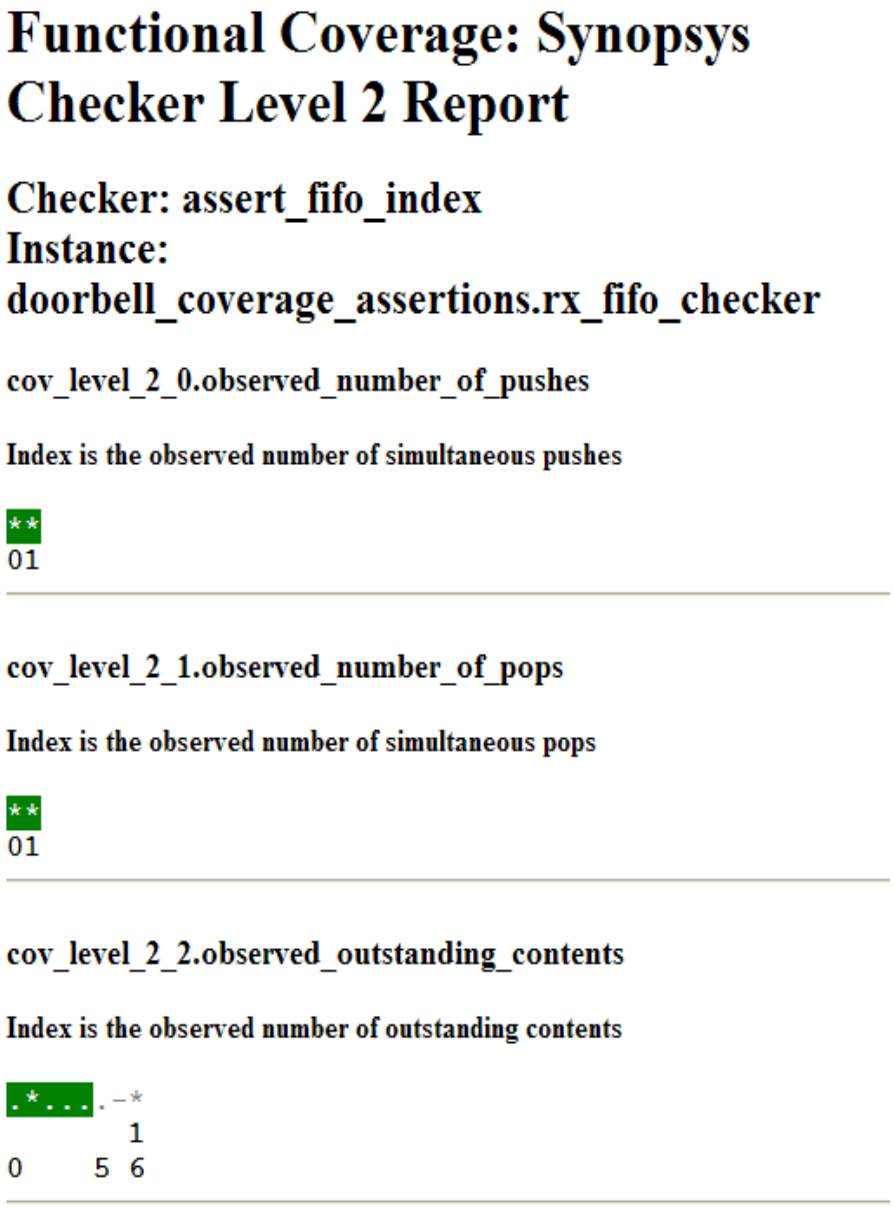
Cover Directive for Properties

testbench.FIFO.invalid_fifo

Name	Attempts	Matches	Vacuous	Incompletes
cov_level_1_0.cover_nb_of_push_operations	26	13	0	0
cov_level_1_1.cover_nb_of_pop_operations	26	13	0	0
cov_level_3_0.cover_eq_nb_of_simultaneous_push_pop	26	7	0	0
cov_level_3_1.cover_eq_nb_of_simultaneous_push_pop_when_empty	26	1	0	0
cov_level_3_2.cover_eq_nb_of_simultaneous_push_pop_when_full	26	0	0	0
cov_level_3_3.cover_nb_of_times_empty_reached_on_pop	26	5	0	1
cov_level_3_4.cover_nb_of_times_full_reached_on_push	26	0	0	1

Clicking the link to the level 2 report in the instance table displays the report as shown in Figure 1-2.

Figure 1-2 `coverage_level_2` *results*



Functional Coverage: Synopsys Checker Level 2 Report

Checker: assert_fifo_index
Instance:
doorbell_coverage_assertions.rx_fifo_checker

cov_level_2_0.observed_number_of_pushes

Index is the observed number of simultaneous pushes

```
**
01
```

---

cov_level_2_1.observed_number_of_pops

Index is the observed number of simultaneous pops

```
**
01
```

---

cov_level_2_2.observed_outstanding_contents

Index is the observed number of outstanding contents

```
.*...._*
      1
0     5 6
```

In the figure:

- The green zones indicate what was covered.
- * delimits the specified range observed,

- . is a point in the range
- - is an interval.

The first two graphs show the range observed was [0:1] and was covered. The third one has an observed range [1:16] with 1, 2, 3 and 4 covered and 5-16 not covered. Note that the 16 is displayed vertically.

Use Mode in SystemVerilog

The Entire Design Targets SystemVerilog

In VCS, to use the SVA Checker Library 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+COVER_ON \
<design files and other 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 further 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 -sverilog+define+ASSERT_ON+COVER_ON  \
<design files and other options> -y $VCS_HOME/packages/sva \
+libext+.sv  +verilog2001ext+.v +incdir+$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.

Note: To use the .sv checkers,

```
-y $VCS_HOME/packages/sva_cg +libext+.sv
```

must be placed ahead of any other option.

Use Mode in VHDL (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, do the following:

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

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

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

   ```
   vlogan -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 specification of the sampling clock edge.

3. To use the compiled checkers in the VHDL design, you 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;
```

2

SVA Basic Checkers

This chapter contains 31checkers. 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. Note that each checker example is followed by an example of the checker written in the alternative name-based format.

assert_always

The `assert_always` 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 (that is, an error condition will be detected in the code.) The *`test_expr`* can be any valid Verilog 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);
```

Arguments

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property `cover_always` indicates that the *`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

assert_always //coverage_level_1 = 7 level 1
#(0, 0, "ERROR: count not within 0 and 9", 0, 15, 0, 0)
valid_count
(clk, reset_n, (count >= 4'b0000) && (count <= 4'b1001));

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_always #(.severity_level(0),.options(0),
 .msg("ERROR: count not within 0 and 9"),.category(0),
 .coverage_level_1(15),.coverage_level_2(0),
 .coverage_level_3(0))
valid_count (.clk(clk),.reset_n(reset_n),
 .test_expr((count >= 4'b0000) && (count <= 4'b1001));
```

assert_always_on_edge

The checker continuously monitors the *test_expr* at every specified edge of the sampling_event. *test_expr* should always evaluate true at the *sampling_event*. If *test_expr* evaluates to false the assertion will fire.

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);
```

Arguments

edge_type

Selects the transition for *sampling_event*.

```
0 - no edge (default)
1 - positive edge
2 - negative edge
3 - any edge
```

Note that the transition is as determined by samplingsampling_event at two consecutive clock ticks.

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

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `cover_always_on_edge` indicates that the *`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

assert_always_on_edge
#(0, 1, 0, "ERROR: count not within 0 and 9", 0, 15, 0, 0)

   valid_always_on_edge
     (clk, reset_n, sig,
     (count >= 4'b0000) && (count <= 4'b1001));

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_always_on_edge ##(.severity_level(0),
 .options(0),.msg("ERROR: count not within 0 and 9"),
 .category(0),.coverage_level_1(15),.coverage_level_2(0),
 .coverage_level_3(0))
valid_always_on_edge (.clk(clk),.reset_n(reset_n),
 .test_expr((count >= 4'b0000) && (count <= 4'b1001));
```

assert_change

Change `assert_change` 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_1,coverage_level_2,
                coverage_level_3)]
inst_name (clk, reset_n, start_event, test_expr);
```

Arguments

`width`
Width of the expression, *`test_expr`*. Default = 1.

`num_cks`
Number of clocks for *`test_expr`* to change its value before an error is triggered after *`start_event`* is asserted. Default = 1.

`flag`
0 - Ignore any *`start_event`* assertion after the first one has been detected. Default = 0.

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`*.

`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

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_change` indicates that the *`exp`* changed within `num_cks`.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_start_event` indicates that the *`start_event`* occurred.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_delay` indicates which delays within from *`start_event`* were observed at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_overlapping_start_events` indicates that the *`start_event`* occured while there was another evaluation attempt in progress.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_change_after_1_clk` indicates that the *`test_expr`* changed value at `the` next clock tick after *`start_event`*.

`cov_level_3_2 (bit 2 set in coverage_level_3)`
Cover property `cover_change_after_num_cks` indicates that the *`test_expr`* changed value at *`num_clk`* 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

//Coverage levels 1 and 2 enabled
assert_change #(0, 4, 4, 0, 0, "ERROR: expr value did not
change in the specified number of clock cycles after
start_event", 0, 15, 15, 0)
valid_change (clk, reset_n, start, expr);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_change #(.severity_level(0),.width(4),.num_cks(4),
 .flag(0),.options(0),.msg("ERROR: expr value did not change
 in the specified  number of clock cycles after
 start_event"),.category(0),.coverage_level_1(15),
 .coverage_level_2(15),.coverage_level_3(0))
valid_change(.clk(clk),.reset_n(reset_n),
 .start_event(start_event),.test_expr(test_expr));
```

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_clks,
        necessary_condition, options, msg,
        category, coverage_level_1, coverage_level_2,
        coverage_level_3)]
inst_name (clk, reset_n, event_sequence);
```

Arguments

num_clks
: The width of the event_sequence (number of clock cycles of the *event_sequence*) that must be valid. Otherwise, an assertion will fire. Should be *num_cks* > 1.

necessary_condition
: Either 1 or 0. The default is 0.

event_sequence
: A verilog concatenation expression, where each bit represents an event.

Coverage Modes:

cov_level_1_0 (bit 0 set in coverage_level_1) :

Cover property, `cover_cycle_sequence_antecedent_match`, indicates that the antecedent part of the sequence matched (i.e., the checker was triggered).

`cov_level_1_1 (bit 1 set in coverage_level_1) :`

Cover property, `cover_cycle_sequence`, indicates that the complete sequence occurred.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`

Cover property, `cover_sequence_slice[i,]` indicates that the sequence matched at least till the `'i'`-th boolean expression.

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;
    #100 reset_n = 1;
    #980 $finish;
end

always #50 clk = ~clk;

endmodule


module child(reset_n, clk);
input reset_n, clk;
reg e1, e2, e3, e4, e5;
reg [3:0] count;

initial $monitor ("count = %d  e1 = %b e2 = %b e3 = %b e4 =
```

```
%b  e5 = %b \n", count, e1, e2, e3, e4, e5);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        count <= 4'b0000;
        e1 <= 0;
        e2 <= 0;
        e3 <= 0;
        e4 <= 0;
        e5 <= 0;
    end
    else
        count <= count + 1;

    case(count)
        4'b0001: e1 <= 1;
        4'b0010: e2 <= 1;
        4'b0011: e3 <= 1;
        4'b0100: e4 <= 1;
        4'b0101: e5 <= 1;
    endcase
end

//Coverage level 1 enabled by default
assert_cycle_sequence
#(0, 5, 1, 0, "ERROR: event sequence did not complete")
valid_seq (clk, reset_n, {e1,e2,e3,e4,e5});

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_cycle_sequence #(.severity_level(0),.num_cks(5),
  necessary_condition(1),.options(0),
  .msg("ERROR: event sequence did not complete"))
valid_seq (.clk(clk),.reset_n(reset_n),
```

```
.event_sequence{e1,e2,e3,e4,e5});
```

assert_decrement

The `assert_decrement` 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`*. The *`test_expr`* can be any valid Verilog. 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);
```

Arguments

`width`
: Width of *`test_expr`* with default value of 1.

`value`
: Maximum decrement value allowed for *`test_expr`* with default value of 1.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
: Cover property, `cover_of_test_expr_change`, indicates that *`test_expr`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
: Cover point, `cover_observed_decrement` indicates all the observed decrement values of the *`test_expr`* when it changes.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_decrement_eq_to_value`, indicates that the reduction in *`test_expr`* was exactly equal to the value parameter.

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 [3:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
        count <= 4'b1111;
    else
        count <= count - 2;
end
```

```
//Coverage level 1 enabled
assert_decrement
#(0, 4, 2, 0, "ERROR: count has decreased beyond allowable
limit ")
valid_count_decrease (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_decrement #(.severity_level(0),.width(4),.value(4),
 .options(0),.msg("ERROR: count has decreased
  beyond allowable limit "))
valid_count_decrease(.clk(clk),.reset_n(reset_n),
 .start_event(start_event),.count(count));
```

assert_delta

The `assert_delta` 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. The *`test_expr`* can be any valid Verilog expression. The check will not start until the first clock after *`reset_n`* is asserted.

Syntax

```
assert_delta [#(severity_level, width, min, max,
                op coverage_level_2,
                coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

Arguments

`width`
: Width of *`test_expr`* with default value of 1. However, with the value of 1 the assertion cannot fail and a warning is issued at time 0. The default is set to 1 for compatibility with OVL

`min`
: Minimum changed value allowed for *`test_expr`* in two consecutive clocks of clk. Default = 1.

`max`
: Maximum changed value allowed for *`test_expr`* in two consecutive clocks of clk. Default = 1.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property, `cover_test_expr_change`, indicates that the *`test_expr`* has changed value.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property, `cover_delta`, indicates that *`test_expr`* changed value within the *`min:max`* range.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point, `observed_delta`, reports the observed delta values that occured at least once on a value change in the *`test_expr`*.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property, `cover_delta_eq_to_min`, indicates that the delta value was exactly equal to the specified *`min`* value when the *`test_expr`* changed value.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_delta_eq_to_max`, indicates that the delta value was exactly equal to the specified *`max`* value when the *`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

//Coverage level 3 enabled
assert_delta
#(0, 8, 2, 4, 0, "ERROR: count has decreased by greater than
the max (4) or less than the min (2) limit", 0, 0, 0, 15)
valid_delta (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_delta
#(.severity_level(0),.width(8),.min(2),.max(4),
 .options(0),.msg("ERROR: count has decreased by greater
 than the max (4) or less than the min (2) limit"),
 .category(0), .coverage_level_1(0), .coverage_level_2(0),
 .coverage_level_3(15))
valid_delta(.clk(clk),.reset_n(reset_n),.count(count));
```

assert_even_parity

The `assert_even_parity` 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_1, coverage_level_2,
                      coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

Arguments

`width`
: Width of the monitored expression *`test_expr`*.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property `cover_test_expr_change` indicates that the *`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

//Coverage level 1 enabled by default
assert_even_parity // no cover selected
#(0, 8, 0, "ERROR: count has odd number of bits asserted")
valid_count_even (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_even_parity #(.severity_level(0),.width(8),
 .opotions(0),.msg("ERROR: ccount has odd number of bits
asserted"))
valid_count_even(.clk(clk),.reset_n(reset_n),
 .count(count));
```

assert_fifo_index

The assert_fifo_index checker ensures that the 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);
```

Arguments

depth
: Depth of the FIFO, default 1. It should never be set to 0, otherwise an assertion will fire.

push_width
: Width of the *push* signal, default 1.

pop_width
: Width of the *pop* signal, default 1.

options
: Currently, the only supported option is options=1, which defines the assertion is an assumption for formal tools.

  If bit 1 of options is set (i.e., option &2 != 0) then simultaneous push and pop operations are disallowed in the same clock cycle. (Note that multiple pushes or pops are allowed always, provided that *push_width*, resp. *pop_width*, is greater than 1.)

push

The value of push indicates the number of writes that are occurring on that particular clock cycle. The *push_width* defines the width of the *push* expression. By default, only a single write can be performed on a particular clock cycle.

pop
: The value of pop indicates the number of reads that are occurring on that particular clock cycle. The *pop_width* defines the width of the *pop* expression. By default, only a single read can be performed on a particular clock cycle.

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1) :
: Cover property, cover_nb_of_push_operations, indicates that *push* was greater than 0.

cov_level_1_1 (bit 1 set in coverage_level_1) :
: Cover property, cover_nb_of_pop_operations, indicates that *pop* was greater than 0.

cov_level_2_0 (bit 0 set in coverage_level_2) :
: Cover point, observed_number_of_pushes, reports the observed value of simultaneous pushes (i.e., the value of the *push* expression) that occurred at least once.

cov_level_2_1 (bit 1 set in coverage_level_2) :
: Cover point, observed_number_of_pops, reports the observed value of simultaneous pops (i.e., the value of the expression *push*) that occurred at least once.

cov_level_2_2 (bit 2 set in coverage_level_2) :
: Cover point, observed_outstanding_contents, reports the observed value of outstanding contents in the FIFO that occured at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_eq_nb_of_simultaneous_push_pop`, indicates that *`push == pop`* occurred simultaneously, when *`push`* and *`pop`* were not 0.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_eq_nb_of_simultaneous_push_pop_when_empty`, indicates that *`push == pop`* occurred simultaneously, when both push and pop were not 0 and the FIFO was empty.

`cov_level_3_2 (bit 2 set in coverage_level_3) :`
Cover property, `cover_eq_nb_of_simultaneous_push_pop_when_full`, indicates that *`push == pop`* occurred simultaneously, when both *`push`* and *`pop`* were not 0 and the FIFO was full.

`cov_level_3_3 (bit 3 set in coverage_level_3) :`
Cover property, `cover_nb_of_times_empty_reached_on_pop`, indicates that the FIFO reached empty on a *`pop`*.

`cov_level_3_4 (bit 4 set in coverage_level_3) :`
Cover property, `cover_nb_of_times_full_reached_on_push`, indicates that the FIFO reached full on a *`push`*.

Example

```
‘define ASSERT_ON
‘define COVER_ON
module testbench;
reg reset_n, clk;

fifo FIFO (reset_n, clk);

initial begin
    $vcdpluson;
    clk = 0;
    reset_n = 0;
```

```
    #40 reset_n = 1;
    #1000 $finish;
end

always #20 clk = ~clk;

endmodule


module fifo(reset_n, clk);
input reset_n, clk;
reg [3:0] fifo;
reg push_sig, pop_sig;

always @(posedge clk) #5 push_sig = ~push_sig;
always @(negedge clk) #45 pop_sig = ~pop_sig;

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        push_sig <= 0;
        pop_sig <= 1'b0;
        fifo <= 4'b0000;
    end

    if (push_sig)
    begin
        fifo <= fifo << 1;
        fifo <= fifo + 1;
        $display("time = %d fifo = %b push_sig = %b \n",
$time, fifo, push_sig);
    end

    if (pop_sig)
    begin
        fifo <= fifo >> 1;
        $display("time = %d fifo = %b pop_sig = %b \n",
$time, fifo, pop_sig);
    end
end
```

```
//Coverage level 1 enabled by default
assert_fifo_index
#(0, 4, 1, 1, 1, "ERROR: Overflow or Underflow in fifo")
invalid_fifo (clk, reset_n, push_sig, pop_sig);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_fifo_index
#(.severity_level(0),.depth(4),.push_width(1),.
 pop_width(1), .options(1),.msg("ERROR: Overflow
 or Underflow in fifo"))
invalid_fifo(.clk(clk),.reset_n(reset_n),.count(count));
```

assert_frame

The `assert_frame` 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,
               coverage_level_1, coverage_level_2,
               coverage_level_3)]
inst_name (clk, reset_n, start_event, test_expr);
```

Arguments

`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 = 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, i.e., when *`min_cks > 0`* and *`max_cks == 0`*. If both *`min_cks`* and *`max_cks`* are 0 then *`test_expr`* must occur at the same time as there is a 0 to 1 transition on *`start_event`*. Default = 0.

`flag`

0 - Ignores any asserted *`start_event`* after the first one has been detected (default);

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 cycle while monitoring *test_expr*.

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*.

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1) :
: Cover property, cover_start_event, indicates that

the *start_event* occured.

cov_level_1_1 (bit 1 set in coverage_level_1) :
: Cover property, cover_frame, indicates that the *test_expr* became true within a minimum and maximum number of clock cycles after the occurence of the *start_event*.

cov_level_2_0 (bit 0 set in coverage_level_2) :
: Cover point, observed_delay, reports the observed delay from a rising *start_event* to the subsequent rising *test_expr*.

cov_level_3_0 (bit 0 set in coverage_level_3) :
: Cover property, cover_overlapping_start_events, indicates that the *start_event* occured while there was already an evaluation in progress.

cov_level_3_1 (bit 1 set in coverage_level_3) :
: Cover property, cover_frame_exactly_at_min_cks, indicates that the test_expr became true exactly at min_cks after *start_event* occurred.

`cov_level_3_2 (bit 2 set in coverage_level_3) :`

Cover property, `cover_frame_exactly_at_max_cks`, indicates that the *`test_expr`* became true exactly at *`min_cks`* after *`start_event`* occurred.

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, expr;
integer count;

initial $monitor ("count = %d  start = %b  expr = %b \n",
count, start, expr);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        count <= 0;
        start <= 0;
```

```
        expr  <= 0;
    end
    else
        count <= count + 1;

    if (count == 3)
        start <= 1;

    if (count == 7)
        expr <= 1;
end

//Coverage level 2 enabled
assert_frame
#(0, 2, 4, 0, 1, "ERROR: expr is not true in the intended
time window after start_event", 0, 0, 1, 0)
invalid_frame (clk, reset_n, start, expr);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_frame
#(.severity_level(0),.min_cks(2),.max_cks(4),.flag(0),
 .options(1),.msg("ERROR: expr is not true in the
 intended time window after start_event"),
 .category(0), .coverage_level_1(0), .coverage_level_2(1),
 .coverage_level_3(0))
invalid_frame(.clk(clk),.reset_n(reset_n),
  .start_event(start),.test_expr(expr));
```

assert_handshake

The `assert_handshake` 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 (0) prior to starting a new cycle.

There are no defaults for this assertion; therefore, if you do not specify parameters, the corresponding assertion is not included. To activate one or more checks in the checker, the corresponding parameters should be specified with a non-zero value.

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);
```

Arguments

`min_ack_cycle`

Activate *`min_ack_cycle`* check if greater than default value of 0.

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`

Activate *`max_ack_cycle`* check if greater than default value of 0.

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`

Activate *`req_drop`* check if greater than default value of 0.

When this parameter is greater than 0, the assertion will ensure that *req* remains active until an *ack* is asserted.

`deassert_count`
Activate *deassert_count* if greater than default value of 0. When this parameter is greater than 0, the assertion will ensure that *req* becomes inactive (0) within *deassert_count* clock ticks after *ack* is asserted.

`max_ack_length`
Activate *max_ack_length* check if greater than default value of 0.

When this parameter is greater than 0, the assertion will ensure that *ack* is not asserted for greater than *max_ack_length* clock cycles (that is, check for *ack* stuck active) and does not become inactive (0) within *deassert_count* clocks after *ack* is asserted.

`req`
Signal that starts the transaction.

`ack`
Signal that terminates the transaction.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
Cover property, `cover_nb_of_reqs`, indicates that *req* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_nb_of_acks`, indicates that *ack* was asserted.

`cov_level_1_2 (bit 2 set in coverage_level_1)`
Cover property, `cover_req_eventually_followed_by_ack`, indicates that there was a *req* followed eventually by an *ack*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point, `observed_delay_pos_req_pos_ack`, records the delay between the arrival of a *`req`* and the subsequent arrival of an *`ack`*.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
Cover point, `observed_delay_pos_ack_neg_req`, records the delay between arrival of an *`ack`* and the deassertion of the *`req`*.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property, `cover_ack_exactly_after_min_ack_cycles`, indicates that the observed latency between *`req`* and *`ack`* was equal to the specified *`ack_cycle`* value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property, `cover_ack_exactly_after_max_ack_cycles`, indicates that the observed latency between *`req`* and *`ack`* was equal to the specified *`ack_cycle`* 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;
    #1000 $finish;
end

always #20 clk = ~clk;

endmodule
```

```
module child(reset_n, clk);
input reset_n, clk;
reg req, ack;
integer count;

initial $monitor ("count = %d  req = %b  ack = %b \n", count,
req, ack);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        req <= 0;
        ack <= 0;
        count <= 0;
    end
    else
        count <= count + 1;

    if (count == 4)
        req <= 1;

    if (count == 6)
        ack <= 1;

    if (count == 9)
        ack <= 0;
end

//Default coverage 1 enabled
assert_handshake #(0, 3, 5, 0, 0, 3, 0,
"ERROR: handshake incorrect")
invalid_handshake (clk, reset_n, req, ack);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_handshake
#(.severity_level(0),.min_ack_cycle(3),.max_ack_cycle(5),
 .req_drop(0),.deassert_count(0),.max_ack_length(3),
 .options(0),.msg("ERROR: handshake incorrect"))
invalid_handshake(.clk(clk),.reset_n(reset_n),
  .req(req),.ack(ack));
```

assert_implication

The `assert_implication` checker continuously monitors the *`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.

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);
```

Arguments

`antecedent_expr`
: Expression verified at the positive edge of *`clk`*.

`consequent_expr`
: Expression verified if *`antecedent_expr`* is true.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
: Cover property `cover_nb_of_antecedent_expr` indicates that the *`antecedent_expr`* was asserted when enabled by *`reset_n`*

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
: Cover property `cover_implication` indicates that the implication was non-vacuously covered 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
    $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 expr1, expr2;
integer count;

initial $monitor ("count = %d  expr1 = %b  expr2 = %b \n",
count, expr1, expr2);

always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        expr1 <= 0;
        expr2 <= 0;
        count <= 0;
    end
    else
        count <= count + 1;

    if (count == 4)
    begin
        expr1 <= 1;
        expr2 <= 1;
    end
```

```
end
//Coverage levels all disabled
assert_implication #(0, 0, "ERROR: implication violated",
0, 0)
invalid_handshake (clk, reset_n, expr1, expr2);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_implication #(.severity_level(0),.options(0),
 .msg("ERROR: implication violated"),.category(0),
 .coverage_level_1(0))
invalid_handshake (.clk(clk),.reset_n(reset_n),
 .antecedent_expr(expr1),.consequent_expr(expr2));
```

assert_increment

The `assert_increment` checker continuously monitors the *`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 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, coverage_level_1,
                    coverage_level_2, coverage_level_3)]
inst_name (clk, reset_n, test_expr);
```

Arguments

`width`
: Width of test_expr with default value of 1. However, with the value of 1 the assertion cannot fail and a warning is issued at time 0. The default is set to 1 for compatibility with OVL.

`value`
: Maximum increment value allowed for *`test_expr`* with default value of 1.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property, `cover_of_test_expr_change`, indicates that the *`test_expr`* changed when enabled by *`reset_n`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `observed_increment`, records the increment value of `test_expr` whenever the *`test_expr`* changes.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_increment_eq_to_value`, indicates that the increment value of *`test_expr`* was exactly equal to the specified 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 [3:0] count;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
    if (reset_n == 0)
        count <= 4'b0000;
```

```
    else
        count <= count + 4;
end

//Coverage level 3 enabled
assert_increment #(0, 4, 4, 0,
"ERROR: count has increased beyond allowable limit", 0, 0,
0, 1)

invalid_count_increase (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_increment #(.severity_level(0),.width(4),.value(4),
  .options(0), .msg("ERROR: count has increased beyond allowable
  limit"),.category(0),.coverage_level_1(0),
 .coverage_level_2(0),.coverage_level_3(1))
invalid_count_increase (.clk(clk),.reset_n(reset_n),
  .test_expr(count));
```

assert_never

The `assert_never` checker continuously monitors the *`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 Verilog expression. When *`test_expr`* evaluates true, this checker will fail.

Syntax

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

Arguments

`test_expr`
Expression being verified at the positive edge of *`clk`*.

Notes

- If the test_expr contains any x or z values then the checkers is effectively disabled and the assertion `assert_never_contains_x_z_value` will fire.
- There are no cover properties or cover groups in thie checker. Therefore the coverage-level_i parameters are not provided.

Example

```
‘define ASSERT_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 <= 4'b1011;
    else
        count <= count + 1;
end

assert_never #(0, 0, "ERROR: count within 0 and 9")
valid_expr (clk, reset_n,
  (count >= 4'b0000) && (count <= 4'b1001));

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_never #(.severity_level(0),.options(0),
 .msg("ERROR: count within 0 and 9"),
 valid_expr (.clk(clk),.reset_n(reset_n),
  .test_expr((count >= 4'b0000) && (count <= 4'b1001));
```

assert_next

The `assert_next` 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`* 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,
           coverage_level_1, coverage_level_2,
           coverage_level_3)]
inst_name (clk, reset_n, start_event, test_expr);
```

Arguments

`num_cks`
: The number of clocks *`test_expr`* must evaluate to true after *`start_event`* is asserted.

`check_overlapping`
: If true, permits overlapping sequences. In other words, a new *`start_event`* can occur (starting a new sequence in parallel) while the previous sequence continues.

`only_if`
: If true, 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 flagged.

`start_event`

Starting event that triggers monitoring of the *test_expr*.

`test_expr`
Expression being verified at the positive edge of *clk*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
Cover property, `cover_start_event`, indicates that the *start_event* occurred.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_next`, indicates that the test_expr became true exactly after *num_cks* from the *start_event*.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_overlapping_start_events`, indicates that a *start_event* occured while there was another evaluation 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, 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  <= 0;
        count <= 0;
    end
    else
        count <= count + 1;

    if (count == 4)
        start <= 1;
    else
        start <= 0;

    if (count == 8)
        expr <= 1;
    else
        expr <= 0;
end

//Cover property, cover_start_event at level 1 enabled
assert_next #(0, 4, 0, 1, 0, "ERROR: expr is not true within
the specified number of clock cycles after start_event", 0,
1)
invalid_next (clk, reset_n, start, expr);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_next #(.severity_level(0),num_cks(4),
 check_overlapping(0), only_if(1),.options(0),
 .msg("ERROR: expr is not true within the specified number of
 clock cycles after start_event"),.category(0),
 .coverage_level_1(1))
invalid_next (.clk(clk),.reset_n(reset_n),
 .start_event(start),  .test_expr(expr));
```

assert_no_overflow

The assert_no_overflow checker continuously monitors the *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 (2**width)-1) to a value greater than *max*

  or

- Change value from a *max* value (default (2**width)-1) to a value less than or equal to a *min* value (default 0).

  The *test_expr* can be any valid Verilog expression.

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);
```

Arguments

width
: Width of the monitored expression *test_expr*.

min
: Minimum value sampled at clock (t+1) of *clk* with default value of 0.

max
: Maximum value sampled at clock ($_t$) of *clk* with default value of (2**width-1).

test_expr

Expression being verified at the positive edge of *clk*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property, `cover_nb_of_test_expr_changes`, indicates that the *test_expr* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `cover_observed_value`, reports all the values taken by *test_expr*

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property, `cover_test_expr_reached_min_value`, indicates that the *test_expr* reached the *min* parameter value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`

Cover property, `cover_test_expr_reached_max_value`, indicates that the *test_expr* reached the *max* parameter value.

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;
    #20 reset_n = 1;
    #980 $finish;
end

always #10 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 <= 4'b0011;

    if (count >= 4'b0010 && count <= 4'b1001)
    begin
        case(count)
            4'b0010: count <= 4'b1010;
            4'b0011: count <= 4'b1011;
            4'b0100: count <= 4'b1100;
            4'b0101: count <= 4'b1101;
            4'b0110: count <= 4'b1110;
            4'b0111: count <= 4'b1111;
            4'b1000: count <= 4'b0000;
            4'b1001: count <= 4'b0010;
       endcase
    end
    else
    begin
        case(count)
            4'b1010: count <= 4'b0011;
            4'b1011: count <= 4'b0100;
            4'b1100: count <= 4'b0101;
            4'b1101: count <= 4'b0110;
            4'b1110: count <= 4'b0111;
            4'b1111: count <= 4'b1000;
            4'b0000: count <= 4'b1001;
            4'b0001: count <= 4'b0010;
       endcase
    end
end

//Coverage level 1 enabled by default
```

```
assert_no_overflow #(0, 4, 2, 9, 0, "ERROR: count changed
from max (9) value to a value either > max (9) or <= min (2)")
value_overflow (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_no_overflow #(.severity_level(0),.width(4),
 .min(2),.max(9),.options(0),.msg("ERROR:count
 changed from max (9) value to a value either > max (9)
 or <= min (2)"))
value_overflow(.clk(clk),.reset_n(reset_n),
 .test_expr(count));
```

assert_no_transition

The assert_no_transition checker continuously monitors the *test_expr* variable 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 size (that is, number of bits) of the *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);
```

Arguments

width
: Width of the monitored expression *test_expr*.

test_expr
: Expression being verified at the positive edge of *clk*.

start_state
: Triggering state of *test_expr*.

next_state
: Invalid state for the machine represented by *test_expr* when traversed from state *start_state*. The value of *next_state* at the time when *test_expr* becomes equal to *start_state* is that used for comparison with *test_expr* even when test_expression changes to a new value in subsequent cycles.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property, `cover_nb_of_times_start_state_occured`, indicates that *`test_expr`* was set to the value specified in *`start_state`*.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`

Cover point, `observed_hold_time`, records the number of *`clk`* cycles the *`test_expr`* remained in *`start_state`* before making a transition to another state other than *`next_state`*.

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, start, next;

initial $monitor("count = %b \n", count);

always @(posedge clk)
begin
```

```
    if (reset_n == 0)
    begin
        count <= 4'b0000;
        start <= 4'b0011;
        next  <= 4'b1001;
    end
    else
       count <= count + 1;

    if (count == 4'b0011)
        count <= 4'b1001;
end

//Coverage level 2 enabled
assert_no_transition #(0, 4, 0, "ERROR: count has reached
the next state of 9 from the start state of 3", 0, 0, 1)
transition_stop (clk, reset_n, count, start, next);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_no_transition
#(.severity_level(0),.width(4),.options(0),
 .msg("ERROR: count has reached the next state of 9 from the
 start state of 3"), .category(0), .coverage_level_1(0),
 .coverage_level_2(1),
transition_stop (.clk(clk),.reset_n(reset_n),
.test_expr((count),.start_state(start),.next_state(next));
```

assert_no_underflow

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

- Change value from a *`min`* value (default = 0) to a value less than *`min`*,

  or

- Change to a value greater than or equal to *`max`* (default (`2**width`)-1).

  The *`test_expr`* can be any valid Verilog expression.

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);
```

Arguments

`width`
: Width of the monitored expression *`test_expr`*.

`min`
: Minimum value sampled at clock (t) of *`clk`* with default value of 0.

`max`
: Maximum value sampled at clock (t+1) of *`clk`* with default value of (`2**width - 1`).

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property, `cover_nb_of_test_expr_changes`, indicates that the *`test_expr`* changed when enabled by *`reset_n`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point, `observed_value`, records the values taken by *`test_expr`*

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property, `cover_test_expr_reached_min_value`, indicates that the *`test_expr`* changed to value equal to *`min`*.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property, `cover_test_expr_reached_max_value`, indicates that the *`test_expr`* changed to value equal to *`max`*.

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 <= 4'b0011;

    if (count >= 4'b0010 && count <= 4'b1001)
    begin
        case(count)
            4'b0010: count <= 4'b0001;
            4'b0011: count <= 4'b1011;
            4'b0100: count <= 4'b1100;
            4'b0101: count <= 4'b1101;
            4'b0110: count <= 4'b1110;
            4'b0111: count <= 4'b1111;
            4'b1000: count <= 4'b0000;
            4'b1001: count <= 4'b0010;
       endcase
    end
    else
    begin
        case(count)
            4'b1010: count <= 4'b0011;
            4'b1011: count <= 4'b0100;
            4'b1100: count <= 4'b0101;
            4'b1101: count <= 4'b0110;
            4'b1110: count <= 4'b0111;
            4'b1111: count <= 4'b1000;
            4'b0000: count <= 4'b1001;
            4'b0001: count <= 4'b0010;
       endcase
    end
end

//Coverage level 3 enabled
assert_no_underflow #(0, 4, 2, 9, 0, "ERROR: count changed
from min (2) to a value either < min (2) or >= max (9)", 0,
0, 0, 0)
value_underflow (clk, reset_n, count);
```

```
endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_no_underflow #(.severity_level(0),.width(4),
 .min(2),.max(9),.options(0),.msg("ERROR:count changed
 from min (2) to a value either < min (2) or >= max (9) "),
 .category(0), .coverage_level_1(0),
 .coverage_level_2(0),.coverage_level_3(0))
value_underflow(.clk(clk),.reset_n(reset_n),
 .test_expr(count));
```

assert_odd_parity

The `assert_odd_parity` checker monitors for odd number of ’1’s in *`test_expr`* at every positive edge of the clock, *`clk`*. The *`test_expr`* can be any valid Verilog expression.

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);
```

Arguments

`width`
Width of the monitored expression *`test_expr`*.

`test_expr`
Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_test_expr_change` indicates that the `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 [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

//There are 7 bits in test_expr
//By default Level 1 //coverage is enabled
assert_odd_parity #(0, 7, 0,
  "ERROR: count does not have an odd number of bits asserted")
valid_count_odd (clk, reset_n, count);
endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_odd_parity #(.severity_level(0),.width(7),
.options(0),.msg("ERROR:count does not have an odd number
of bits asserted"))
valid_count_odd(.clk(clk),.reset_n(reset_n),
```

```
.test_expr(count));
```

assert_one_cold

The `assert_one_cold` 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.

The *`test_expr`* can be any valid Verilog expression.

Syntax

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

Arguments

`width`
Width of the monitored expression *`test_expr`*.

`inactive`
Specifies the inactive state of *`test_expr`*.

If *`inactive`* is set to 0, then this checker acts like a `zero_one_cold` checker (that is, the inactive state of *`test_expr`* is all zeroes).

If *`inactive`* is set to 1, then this checker acts like a `one_one_cold` checker (that is, the inactive state is all ones).

The default value (that is, 2) specifies that there is no inactive state (that is, always `one_cold`).

`test_expr`
Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_test_expr_change` indicates that the *`test_expr`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `bit_is_0_after_a_change` reports which bit in *`test_expr`* was 0 at least once after a change of *`test_expr`* value.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
Cover property `cover_test_expr_with_all_1` indicates that the *`test_expr`* was all 1s. Enabled when *`inactive`* `== 1`.

`cov_level_2_2 (bit 2 set in coverage_level_2)`
Cover property `cover_test_expr_with_all_0` indicates that the *`test_expr`* was all 0s. Enabled when *`inactive`* `== 0`.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cov_level_3[i].cover_test_expr_bit_is_0` indicates that 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

//The check is done on count having 8 bits, inactive is 0
//Level 1 coverage is enabled by default.

assert_one_cold #(0, 8, 0, 0, "ERROR: count is not one-cold")
  invalid_one_cold (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_one_cold #(.severity_level(0),.width(8),
 .inactive(0),.options(0),.msg("ERROR: count is not one
 cold"))
invalid_one_cold(.clk(clk),.reset_n(reset_n),
 .test_expr(count));
```

assert_one_hot

The `assert_one_hot` checker ensures that the variable, *`test_expr`*, has only one bit set to 1 at any positive clock *`clk`* edge. The *`test_expr`* can be any valid Verilog Expression.

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);
```

Arguments

`width`
Width of the monitored expression *`test_expr`*.

`test_expr`
Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_test_expr_change` indicates that the *`test_expr`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `cover_test_expr_bit_is_1` indicates which bit in *`test_expr`* was 1 at least once after a change of *`test_expr`* value.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `test_expr_bit_is_1[i].cover_test_expr_bit_is_1` indicates that bit i of *`test_expr`* was 1 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'b00000001;
    else
        count <= ((count << 1) | {7'b0000000, count[7]});
end

//The width of count is 8 bits
//Coverage Level 1 is enabled by default.
assert_one_hot #(0, 8, 0, "ERROR: count is not one-hot")
invalid_one_hot (clk, reset_n, count);
endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_one_hot #(.severity_level(0),.width(8),
 .inactive(0),.options(0),.msg("ERROR:count is not
 one-hot"))
invalid_one_hot(.clk(clk),.reset_n(reset_n),
 .test_expr(count));
```

assert_proposition

The `assert_proposition` checker continuously monitors the *`test_expr`*. Hence, this assertion is unlike `assert_always`; that is, *`test_expr`* is not being sampled by a clock. This `assert_proposition` assertion verifies that *`test_expr`* will always evaluate true. If *`test_expr`* evaluates to false while *`reset_n`* is 1, an assertion will fire (that is, an error condition will be detected in the code). The *`test_expr`* can be any valid Verilog expression.

Syntax

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

Arguments

`test_expr`
: Expression being verified.

Note

There is no cover property or cover group in this checker. Therefore the `coverage_level_i` parameters are not provided.

Example

```
‘define ASSERT_ON
module testbench;
reg reset_n, clk, sig;

child CH (reset_n, sig);

initial begin
    clk = 0;
    reset_n = 0;
    sig = 0;
    #80 reset_n = 1;
```

```
    #980 $finish;
end

always #20 clk = ~clk;
always #40 sig = ~sig;

endmodule


module child(reset_n, sig);
input reset_n, sig;
reg [3:0] count;

initial $monitor("time = %d  count = %b \n", $time, count);

always @(sig)
begin
    if (reset_n == 0 || count >= 9)
        count <= 4'b0000;
    else
        count <= count + 1;
end

assert_proposition
#(0, 0, "ERROR: count not within 3 and 9, both inclusive")

valid_proposition
(reset_n, (count >= 4'b0011) && (count <= 4'b1001));

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_proposition #(.severity_level(0),.property_type(0),
 .msg("ERROR:count not within 3 and 9, both inclusive"),
 valid_proposition(.reset_n(reset_n),
 .test_expr(count >= 4'b0011) && (count <= 4'b1001));
```

assert_quiescent_state

The `assert_quiescent_state` 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 using the macro *`ASSERT_END_OF_SIMULATION*. Verification occurs when a sampled rising edge (as sampled by *`clk`*) 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);
```

Arguments

`width`
: Width of the monitored expression *`state_expr`*.

`state_expr`
: Expression being verified at the positive edge of *`clk`*.

`check_value`
: Specifies value for `state_expr` when quiescent. (i.e., a signal that holds the value to be compared with *`state_expr`* when *`sample_event`* is asserted).

`sample_event`
: Triggers the quiescent state check when a sampled rising edge is detected (as sampled by *`clk)`*.

`ASSERT_END_OF_SIMULATION`
: This macro can be used to quiescent the state expression at the end of simulation. For example:

`+define+ASSERT_END_OF_SIMULATION=top.sim_end`
If this macro is associated with a signal it will trigger when this signal raises (sampled transition by the clock).

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_quiescent_state` indicates that the quiescent state was reached

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;
    #80 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, value;

initial $monitor(" sig = %b  count = %b \n", sig, count);
```

```
always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        count <= 4'b0000;
        value <= 4'b0100;
    end
    else
        count <= count + 1;

    if (count >= 9)
        count <= 1'b0;
end

//Coverage level 1 enabled by default
assert_quiescent_state #(0, 4, 0, "ERROR: count not equal
to 4 when sampled on posedge of event sig")
valid_q_state (clk, reset_n, count, value, sig);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_quiescent_state #(.severity_level(0),.width(4),
 .property_type(0),.msg("ERROR:count not equal to 4 when
 sampled on posedge of event sig"))
valid_q_state(.clk(clk),.reset_n(reset_n),
 .state_expr(count),.check_value(value),
 .sample_event(sig));
```

assert_range

The `assert_range` checker continuously monitors the *`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`*. The *`test_expr`* can be any valid Verilog expression.

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);
```

Arguments

`width`
: Width of the monitored expression *`test_expr`*.

`min`
: Minimum value allowed for range check. Default = 0.

`max`
: Maximum value allowed for range check. Default = (`2**width - 1`).

`est_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property, `cover_nb_of_test_expr_changes`, indicates that the *`test_expr`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point, `cover_observed_value`, reports all the values taken by *`test_expr`*

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property, `cover_test_expr_reached_min_value`, indicates that the *`test_expr`* reached the *`min`* parameter value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property, `cover_test_expr_reached_max_value`, indicates that the *`test_expr`* reached the *`max`* parameter value.

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)
```

```
    begin
        count <= 4'b0010;
    end
    if (count >= 4'b0010 && count <= 4'b1001)
    begin
        case(count)
            4'b0010: count <= 4'b1010;
            4'b0011: count <= 4'b1011;
            4'b0100: count <= 4'b1100;
            4'b0101: count <= 4'b1101;
            4'b0110: count <= 4'b1110;
            4'b0111: count <= 4'b1111;
            4'b1000: count <= 4'b0000;
            4'b1001: count <= 4'b0001;
       endcase
    end
    else
    begin
        case(count)
            4'b1010: count <= 4'b0011;
            4'b1011: count <= 4'b0100;
            4'b1100: count <= 4'b0101;
            4'b1101: count <= 4'b0110;
            4'b1110: count <= 4'b0111;
            4'b1111: count <= 4'b1000;
            4'b0000: count <= 4'b1001;
            4'b0001: count <= 4'b0010;
       endcase
    end
end
//Coverage level 2 enabled
assert_range
#(0, 4, 2, 9, 0, "ERROR: count not within 2 and 9", 0, 0, 1, 0)
valid_range (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_range
#(.severity_level(0),.width(4),.min(2),.max(9),
 .options(0),.msg("ERROR: count not within 2 and 9"),
 .category(0), .coverage_level_1(0),
 .coverage_level_2(10), .coverage_level_3(0))
valid_range(.clk(clk),.reset_n(reset_n),.test_expr(count))
;
```

assert_time

The `assert_time` checker continuously monitors the *`start_expr`*. When this signal (or expression) evaluates true, the assertion monitor ensures that the *`test_expr`* evaluates to true for the next *`num_cks`* number of clocks.

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);
```

Arguments

`num_cks`

The number of clocks *`test_expr`* must evaluate to true after *`start_event`* is asserted.

`flag`

0 - Ignores any asserted *`start_event`* after the first one has been detected (default);

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 cycle while monitoring *`test_expr`*.

`start_event`

Starting event that triggers monitoring of the *`test_expr`*.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `cover_start_event` indicates that there was a *`start_event`* asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property `cover_time` indicates that there was a match.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property `cover_overlapping_start_events` indicates that there were overlapping start events.

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, 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  <= 0;
        count <= 0;
    end
    else
        count <= count + 1;

    if (count == 4)
        start <= 1;
    else
        start <= 0;

     if (count == 4 || count == 5 || count == 6 ||
         count == 7 || count == 8 || count == 9)
        expr <= 1;
    else
        expr <= 0;
end

//The time interval is 4 clock cycles
//Restart checking on start event
//Coverage levels 1 and 3 are enabled

assert_time #(0, 4, 1, 0,
    "ERROR: expr not true within 4 cycles after start",
     0, 15, 0, 15)
valid_time (clk, reset_n, start, expr);
/*
The time interval is 4 clock cycles, restart checking on
start event and coverage levels 1 and 3 are enabled
*/

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_time #(.severity_level(0),.num_cks(4),
 .flag(1),.options(0),.msg("ERROR: expr not true
  within4 cycles after start"),.category(0),
 .coverage_level_1(15),.coverage_level_2(0),
 .coverage_level_3(15))
valid_time(.clk(clk),.reset_n(reset_n),
 .start_event(start),.test_expr(expr));
```

assert_transition

The `assert_transition` checker continuously monitors the *`test_expr`* variable at every positive edge of the triggering event, *`clk`*. When *`test_expr`* evaluates to the value of *`start_state`*, the checker ensures that *`test_expr`* will always change to the value of *`next_state`*. The *`width`* parameter defines the size (that is, number of bits) of the *`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);
```

Arguments

`width`
: Width of *`test_expr`* with default value of 1.

`test_expr`
: State variable representing finite-state machine (FSM) being checked at the positive edge of *clk*.

`start_state`
: Triggering state of *`test_expr`*. The *`start_state`* value should be different from the *`next_state`* value.

`next_state`
: Next valid state for machine represented by *`test_expr`* when traversed from state *`start_state`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`

Cover property, `cover_nb_of_times_start_state_occured`, indicates that the *`test_expr`* entered the *`start_state`*.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property, `cover_nb_of_transitions`, indicates that the *`test_expr`* entered the *`start_state`* and made a transtion to the *`next_state`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `observed_hold_time`, reports the number of *`clk`* cycles the *`test_expr`* remained in the *`start_state`* before making a transition to the *`next_state`*.

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, start, next;

initial $monitor("count = %b \n", count);
```

```
always @(posedge clk)
begin
    if (reset_n == 0)
    begin
        count <= 4'b0000;
        start <= 4'b0011;
        next  <= 4'b1001;
    end
    else
        count <= count + 4'b0001;

    if (count == 4'b0011)
        count <= 4'b0111;
end

//Cover point cov_level_1_1 enabled at level 1
assert_transition #(0, 4, 0, "ERROR: count did not reach the
next state of 9 from the start state of 3", 0, 2, 0, 0)
valid_transition (clk, reset_n, count, start, next);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_transition #(.severity_level(0),width(4),
 .property_type(0), .msg("ERROR: count did not reach the
 next state of 9 from the start state of 3"),.category(0),
 .coverage_level_1(2), .coverage_level_2(0),
 .coverage_level_3(0))
valid_transition (.clk(clk),.reset_n(reset_n),
  .test_expr(count),.start_state(start),
  .next_state(next));
```

assert_unchange

The `assert_unchange` checker continuously monitors the *`start_event`* at every positive edge of the triggering event, *`clk`*. When this signal (or expression) evaluates true, the checker ensures that the *`test_expr`* will not change values 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);
```

Arguments

`width`
: Width of the expression, *`test_expr`*. Default = 1.

`num_cks`
: Number of clock cycles for *`test_expr`* to change its value before an error is triggered after *`start_event`* is asserted. Default = 1.

`flag`
: 0 - Ignore any *`start_event`* assertion after the first one has been detected. Default = 0.

  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`*.

`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

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_start_event` indicates that the *`start_event`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_unchange` indicates that the *`test_expr`* remained stable the required time interval.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_overlapping_start_events` indicates that 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)
          expr <= 4'b0110;
     else
          expr <= expr + 1;
     end
// Coverage Level 2 selected by default
assert_unchange #(0, 4, 4, 1, 0, "ERROR:test_expr changed
within 4 clock cycles")
valid_unchange (clk, reset_n, start, expr);

endmodule
```

The value of `expr` should not change within 4 clock cycles after start. expr has 4 bits and `flag = 1`, i.e., RESET ON START is requested. Coverage Level 1 is enabled by default. Since `expr` changes changes after 3 clock cycles the assertion will fire.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_unchange #(.severity_level(0),width(4),num_cks(4),
 flag(1),.options(0), .msg("ERROR: test_expr changed
 within 4 clock cycles"))
valid_unchange (.clk(clk),.reset_n(reset_n),
 .start_event(start),.test_expr(expr));
```

assert_width

The `assert_width` checker continuously monitors the *`test_expr`*. When this signal (or expression) evaluates true, it ensures that the *`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);
```

Arguments

`min_cks`

Specifies the minimum number of clock cycles where *`test_expr`* should be true. If the *`test_expr`* goes false before *`min_cks`*, then an error occurs. The default value is 1.

`max_cks`

Specifies the maximum number of clock cycles where *`test_expr`* can be true. If set to 0, then there is no maximum check (any value is valid). If the *`test_expr`* is true for more than *`max_cks`*, then an error occurs. The default value is 1.

`test_expr`

Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`

Cover property, `cover_nb_of_test_expr_posedges`, indicates that a postive edge was observed on the *`test_expr`*.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`

Cover property, `cover_width`, indicates that the *`test_expr`* was asserted for a duration defined by the *`min_cks`* and *`max_cks`* parameters.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`

Cover point, `cover_observed_hold_time`, reports the observed numbers of clock cycles for which the *`test_expr`* was asserted.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`

Cover property, `cover_test_exp_change_exactly_after_min_cks`, indicates that the *`test_expr`* was sampled deasserted after being asserted for exactly *`min_cks`* clock ticks.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`

Cover property, `cover_test_exp_change_exactly_after_max_cks`, indicates that the *`test_expr`* was sampled deasserted after being asserted for exactly *`max_cks`* clock ticks.

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 expr;
integer count;

initial $monitor ("time = %d expr = %b, count = %d \n",
$time, expr, count);

always @(posedge clk)
begin
     if (reset_n == 0)
     begin
         expr <= 0;
         count <= 0;
     end
     else
         count <= count + 1;

    if (count == 4 || count == 5 || count == 6 || count == 7)
         expr <= 1;
     else
         expr <= 0;
end

//Coverage is disabled on this instance
assert_width #(0, 0, 4, 0, "ERROR: expr is not true for the
             specified number of clock cycles", 0, 0, 0, 0)
valid_width (clk, reset_n, expr);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_width #(.severity_level(0),min_cks(0),max_cks(4),
```

```
  .options(0), .msg("ERROR: expr is not true for the
  specified number of clock cycles"),.category(0),
  .coverage_level_1(0), .coverage_level_2(0),
  .coverage_level_3(0))
valid_width (.clk(clk),.reset_n(reset_n),
  .test_expr(expr));
```

assert_win_change

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

Arguments

`width`
Width of *`test_expr`* with default value of 1.

`start_event`
Starting event that triggers monitoring of the *`test_expr`*

`test_expr`
Expression being verified at the positive edge of *`clk`*.

`end_event`
Event that terminates monitoring of a *`start_event`*.

Coverage Modes

`cov_level_1_0 (bit 0 is set in coverage_level_1)`
Cover property, `cover_nb_of_start_events`, indicates that the *`start_event`* was sampled high.

`cov_level_1_1 (bit 1 is set in coverage_level_1)`
Cover property, `cover_win_change`, indicates that the *`test_expr`* changed value before and including the *`end_event`*.

`cov_level_2_0 (bit 0 is set in coverage_level_2)`

Cover point, `cover_observed_hold_time`, reports the numbers of clock cycles for which the *`test_expr`* remained unchanged after the *`start_event`* occurred.

`cov_level_3_0 (bit 0 is set in coverage_level_3)`

Cover property, `cover_test_exp_change_exactly_at_end_event`, indicates that a change on the *`test_expr`* was sampled at the same time as the *`end_event`* occurred.

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, expr, end_event;
integer count;

initial $monitor ("expr = %b, start = %b end_event = %b count
= %d \n", expr, start, end_event, count);
```

```
always @(posedge clk)
begin
     if (reset_n == 0)
     begin
         start <= 0;
         expr <= 0;
         end_event <= 0;
         count <= 0;
     end
     else
         count <= count + 1;

     if (count == 4)
         start <= 1;
     else
         start <= 0;

     if (count == 7)
         expr = 1;

     if (count == 8)
         end_event <= 1;
     else
         end_event <= 0;
end

//Coverage level 1 enabled by default
assert_win_change #(0, 1, 0, "ERROR: expr did not change
                   even once between start and end events")
valid_win_change (clk, reset_n, start, expr, end_event);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_win_change #(.severity_level(0),.width(1),
  .options(0), .msg("ERROR: expr did not change even
  once between start and end events"))
```

```
valid_win_change (.clk(clk),.reset_n(reset_n),
 .start_event(start),.test_expr(expr));
```

assert_win_unchange

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

Syntax

```
assert_win_unchange [#(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);
```

Arguments

`width`
: Width of *`test_expr`* with default value of 1.

`start_event`
: Starting event that triggers monitoring of the *`test_expr`*

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

`end_event`
: Event that terminates monitoring of a *`start_event`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property, `cover_nb_of_start_events`, indicates that *`start_event`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property, `cover_win_unchange`, indicates that *`test_expr`* remained stable for the duration between a *`start_event`* and an *`end_event`* including the arrival of *`end_event`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `observed_end_event_time`, records the number of clock cycles between a *`start_event`* and the subsequent *`end_event`*.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property, `cover_test_expr_changes_exactly_after_end_event`, indicates that the *`test_expr`* changed at the clock tick just after an *`end_event`*.

assert_window

The `assert_window` checker continuously monitors the *`start_event`* at every positive clock edge *`clk`*. When the *`start_event`* evaluates true, it ensures that the *`test_expr`* evaluates true at every successive positive clock edge *`clk`* up to and including the *`end_event`* expression.

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

Syntax

```
assert_window [#(severity_level, options,
                 msg, category, coverage_level_1,
                 coverage_level_2,overage_level_3)]
inst_name (clk, reset_n, start_event, test_expr, end_event);
```

Arguments

`start_event`
: Starting event that triggers monitoring of the *`test_expr`*

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

`end_event`
: Event that terminates monitoring of a *`start_event`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property, `cover_nb_of_start_events`, indicates that *`start_event`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property, `cover_window`, indicates that *`test_expr`* remained asserted for the duration between a *`start_event`* and an *`end_event`* including the arrival of *`end_event`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `observed_end_event_time`, records the number of clock cycles between a *`start_event`* and the subsequent *`end_event`*.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property, `cover_window_terminates_exactly_after_end_event`, indicates that the *`test_expr`* was deasserted at the clock tick just after an *`end_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, expr, end_event;
integer count;

initial $monitor ("expr = %b, start = %b end_event = %b count
= %d \n", expr, start, end_event, count);

always @(posedge clk)
begin
     if (reset_n == 0)
     begin
         start <= 0;
         expr <= 0;
         end_event <= 0;
         count <= 0;
     end
     else
         count <= count + 1;

     if (count == 4)
         start <= 1;
     else
         start <= 0;

     if ((count >= 5 && count <= 8))
         expr = 1;
     else
         expr = 0;

     if (count == 8)
         end_event <= 1;
     else
         end_event <= 0;
end

//Coverage level 3 enabled
assert_window #(0, 1, "ERROR: expr did not remain true at
every clock edge between start and end events", 0, 0, 0, 1)
valid_window (clk, reset_n, start, expr, end_event);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_window #(.severity_level(0),.options(1),
 .msg("ERROR: expr did not remain true at every clock edge
 between start and end events"),.category(0),
 .coverage_level_1(0), .coverage_level_2(0),
 .coverage_level_3(1))
valid_window (.clk(clk),.reset_n(reset_n),
 .start_event(start),.test_expr(expr),
 .end_event(end_event));
```

assert_zero_one_hot

The `assert_zero_one_hot` checker continuously monitors the test_expr at every positive edge of the triggering event, *`clk`*. It verifies that *`test_expr`* will have exactly one bit asserted or no bit asserted, otherwise, an assertion will fire (that is, an error condition will be detected in the code). The *`test_expr`* can be any valid Verilog expression.

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);
```

Arguments

`width`
: Width of the monitored expression *`test_expr`*.

`test_expr`
: Expression being verified at the positive edge of *`clk`*.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
: Cover property `cover_test_expr_change` indicates that the *`test_expr`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
: Cover property `cover_test_expr_with_all_0` indicates that the all 0 value occurred when *`test_expr`* changed value.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
: Cover point `bit_is_1_after_a_change` reports which bits in *`test_expr`* were set to 1 at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property `test_expr_bit_is_1[i].cover_test_expr_bit_is_1` indicates that bit `i` was 1 after 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

//The tested expression count is 8 bits wide.
//Coverage level 1 enabled by default
assert_zero_one_hot #(0, 8, 0, "ERROR: count is not one-hot")
valid_zero_one_hot (clk, reset_n, count);

endmodule
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_zero_one_hot #(.severity_level(0),.width(8),
 .options(0),.msg("ERROR: count is not one-hot"))
valid_zero_one_hot (.clk(clk),.reset_n(reset_n),
 .test_expr(count));
```

3

SVA Advanced Checkers

The 2nd part of the checker library contains 22 checkers plus an `sva_std_edge_select.v` file that contains an interface definition that is instantiated in all these new checkers and computes the appropriate clock edge.

The new checkers use the same controls as the SVL checkers in the previous chapter, but in addition they have a clock edge selection parameter, *`edge_expr`*.

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, msg* and *coverage_level_i*, where *i* = 1, 2, or 3, described in Chapter 1, the checkers in this chapter can select the active edge of 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

  If edge_expr is not specified, it defaults to posedge.

assert_arbiter

Ensures that a resource arbiter provides grants to corresponding requests within *min_lat* and *max_lat* cycles.

The checks are not enabled unless reset_n evaluates true at the time of the request. The checks are performed on the active *clk* edge specification.

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);
```

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.

`req_priority_chk`

If true, then this checker will ensure that grants are issued according to the priority indicated in the *`req_priority`* vector. *`req_priority_chk. req_priority_chk`* functions on top of any other scheme using *`arbitration_rule`*. That is, if priority and, say, FIFO is selected then first priority arbitration is applied and if multiple requests exist at the highest priority level, the final selection must satisfy the FIFO rule.

If *`req_priority_chk`* is 0 (disabled), then *`req_priority`* is not taken into account in any of the other checks. However, the argument *`req_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.

`arbitration_rule`

Selects which arbitration algorithm is verified among fairness, FIFO or least-recently used (LRU). It must be a compile-time constant.

arbitration_rule = 0 - no rule checked

arbitration_rule = 1 - fairness (round-robin)

arbitration_rule = 2 - FIFO

arbitration_rule = 3 - LRU

If fairness is selected, then this checker 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.

If FIFO is selected, then this checker 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*.

If LRU is selected, then this checker will ensure that grants are issued to the least recently granted request. If *req_priority_chk* is also asserted than LRU is applied to (simultaneous) highest priority requests. Note that initially, all requests are assumed to have happened equally in the past, so that the selection between them is arbitrary until granted at least once.

`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.

reqs
: 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.

req_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*.

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

Assumptions

When *req_priority_chk* = 0 it is assumed that the *req_priority* vector is set to all 0 values.

It is preferrable that *min_lat* > 0 when *req_priority_chk* = 0, i.e., combinational response is possible only in *req_priority* - based arbitration.

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.

Failure modes

Upon failure assertion `assert_arbiter_one_cycle_gnt[i]` will report `!grants[i]` as the failing expression meaning that in position `i` the grant signal was not deasserted on the next cycle.

Assertion `assert_arbiter_req_granted[i]` will report failure when in position `i` the grant signal did not arrive within the specified latency.

Assertion `assert_arbiter_granted_only_if_req[i]` will report failure when in position `i` the grant signal occurred without a request being asserted at the same time.

Assertion `assert_arbiter_highest_grant[i]` will report failure when line `i` the granted request was not that of the highest req_priority.

Assertion `assert_arbiter_fifo_chk[i][j]` will report failure if on line `j` the grant was asserted before the grant on line `i` while the requests came in the opposite order.

`Assertion assert_arbiter_two_active[i][j]`, a failure means that although there were continuous requests on lines `i` and `j`, the grant on line `i` arrived twice without a grant on line `j`.

`Assertion assert_arbiter_single_grant` will report failure there were more than one grant at the same time.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `cover_arbiter_req_granted` indicates there were granted requests for each channel

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover property `loop_A[i].cover_abandoned_req` indicates there were abandoned requests, for each channel

`cov_level_2_1 (bit 1 set in coverage_level_2)`

Cover point `observed_latency` indicates which request-to-grant latency within the range *`min-max`* occurred at least once.

`cov_level_2_2 (bit 2 set in coverage_level_2) :`

Cover point, `concurrent_requests`, reports which multiplicity of concurrent requests occurred. It is sampled when a grant to one of the requests is issued.

`cov_level_3_0 (bit 1 set in coverage_level_3)`

Cover property Cover property `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.

`cov_level_3_1 (bit 2 set in coverage_level_3)`

Cover property
`loop_A[i].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.

Example

The instance

```
assert_arbiter #(0, 2, 1, 1, 1, 0, 1, 5, 0, "arbiter failed",
                 0, 15, 15, 0)
arbiter_inst ( clk, 1, requests, 2'b1_0, grants);
```

There are 2 request and 2 grant lines, enabled all the time, *`req_priority`* is 0 for channel 0 and 1 for channel 1, encoded over one bit; *`grant_one_chk`*=1, *`req_priority_chk`*=1, *`arbitration_rule`*=0, *`min_lat`*=1, *`max_lat`*=5. Signals are sampled at posedge *`clk`*. The (optional) message is "arbiter failed" and it is output in addition to the standard simulator message. Coverage levels 1 and 2 are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_arbiter #(.severity_level(0),.no_chnl(2),
 .bw_prio(1),.grant_one_chk(1),.req_priority_chk(1),
 .arbitration_rule(0),.min_lat(1),.max_lat(5),
 .edge_expr(0),.msg("arbiter failed"),.category(0),
 .coverage_level_1(15),.coverage_level_2(15),
 .coverage_level_3(0))
arbiter_inst (.clk(clk),.reset_n(1),.reqs(requests),
 .req_priority(2'b1_0),.grants(grants));
```

assert_bits

Ensures that the value of *exp* has between *min* and *max* number of bits that are asserted or deasserted as indicated by the *asserted* flag. To specify a single number and not a range, use *min == max*.

Syntax

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

Arguments

`min`

Minimum number of bits asserted or deasserted. Valid values of constant parameters: *min, max* must be non-negative integers. Default = 1.

`max`

Maximum number of bits asserted or deasserted. Valid values of constant parameters: *min, max* must be non-negative integers. Default = 1.

`asserted`

If deasserted evaluates true, then the check is to ensure that the specified number or range of bits in *exp* is 1. Otherwise, the check is to ensure the specified number or range of bits in *exp* is 0. Default = 1, meaning that the test is for bits set to 1.

`exp_bw`

The number of bits in *exp*.

Failure Modes

Upon failure, if *asserted* = 1, then failure indicates that the number of bits set to 1 in exp was not in the interval [*min, max*], and if *deasserted* = 1, then failure indicates that the number of bits set to 0 was not in the specified interval.

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1)

Cover property cover_bits_exp_change indicates that the exp changed.

cov_level_1_1 (bit 1 set in coverage_level_1)

Cover property cover_bits indicates that the number of bits asserted was within the *min : max* range.

cov_level_3_0 (bit 0 set in coverage_level_3)

Cover property cover_min_bits_asserted indicates that the number of bits set was exactly equal to the specified *min* value.

cov_level_3_1 (bit 1 set in coverage_level_3)

Cover property cover_max_bits_asserted indicates the number of times the number of bits set was exactly equal to the specified *max* value.

Example

This example verifies that exp of width 4 has between 1 and 3 bits set to 0. The checker is always enabled because *reset_n* is 1 and exp is sampled at posedge *clk*. Coverage levels 1 and 3 are enabled.

```
assert_bits #(0, 1, 3, 0, 4, 5 , "bits failed", 0, 15, 0, 15)
     bits_inst ( clk, 1, exp);
```

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_bits #(.severity_level(0),.min12),
 .max(3),.asserted(0),.exp_bw(4),.edge_expr(5),
 .msg("bits failed"),.category(0),.coverage_level_1(15),
 .coverage_level_2(0),.coverage_level_3(15),
bits_inst (.clk(clk),.reset_n(1),.exp(exp)) ;
```

assert_code_distance

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. The check is not enabled unless *reset_n* evaluates true. The check is performed on the active *clk* edge specification.

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);
```

Arguments

min
: The minimum number of bits that are different (must be non-negative integer). *min <= max*. 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 the signal under test (*exp*) is compared to.

Failure Modes

The assertion assert_code_distance will report failure when the expressions differ in more bits than the value of the *max* parameter or less than the *min* parameter.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_exp_change` indicates that the `exp` changed value.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_code_distance_match` indicates that the code distance was within the specified interval when `exp` changed.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_code_distance` repoorts which code distances occured at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_code_distance_eq_to_min` indicates that the observed code distance was exactly equal to the specified `min` value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_code_distance_eq_to_max` indicates that the observed code distance was exactly equal to the specified `max` value.

Example

```
assert_code_distance #(0, 1, 3, 4, 0,"bad code",0,15,15,15)
cd_inst ( clk, 1, exp1, exp2);
```

Reports failure when the number of different bits is not within the interval 1 to 3 at positive edge of *`clk`*. *`exp1`* and *`exp2`* are 4 bits wide. Coverage levels 1, 2 and 3 are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_code_distance #(.severity_level(0),.min(1),
 .max(3),.bw(4),.edge_expr(0),.msg("bad code"),
 .category(0), .coverage_level_1(15),
 .coverage_level_2(15),.coverage_level_3(15))
cd_inst (.clk(clk),.reset_n(1),.exp(exp1),.exp2(exp2)) ;
```

assert_data_used

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. The checks are not enabled unless *reset_n* evaluates true. The checks are performed on the active *clk* edge specification.

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);
```

Arguments

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

  sleft - *sright* should be equal to *dleft* - *dright* otherwise some bits would be lost or compared with 0

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.

Note: *sleft* - *sright* should be equal to *dleft* - *dright;* otherwise some bits would be lost or compared with 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 or equal to 1.

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

  Note that the specified number must be greater than or equal to 0.

  `start = m, m>0`, and `finish = 0` means that the open-ended interval `[m:$]` is to be used. Note that if this is the case, the assertion cannot be falsified.

trigger
: Signal that triggers the start of the window.

src
: Source signal.

dest
: Destination signal.

Failure Modes

Upon failure the reported failing expression by the assertion `sva_data_used` is `#[`*start*`:`*finish*`](`*dest*`[`*dleft*`:`*dright*`] == sva_v_src_value)` where `sva_v_src_value` is a variable that stores the original *src* value needed for the comparison.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property, `cover_nb_of_times_trigger_asserted`, indicates that *`trigger`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property, `cover_data_used`, indicates that the data from *`src`*[*`sleft`*:*`sright`*] appeared in *`dest`*`[`*`dleft`*`:`*`dright`*`]` within the specified window.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover property, `observed_delay`, records the delay observed between a *`trigger`* and the subsequent arrival of the data at *`dest`*`[`*`dleft`*`:`*`dright`*`]`.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property, `cover_data_used_exactly_after_start_clk_cycles`, indicates that data arrived at *`dest`*`[`*`dleft`*`:`*`dright`*`]` start number of clock cycle after a *`trigger`* was asserted.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property, `cover_data_used_exactly_after_finish_clk_cycles`, indicates that data arrived at *`dest`*`[`*`dleft`*`:`*`dright`*`]` finish number of clock cycle after a *`trigger`* was asserted.

Example

```
assert_data_used #( 0, 15, 3, 12, 0, 1, 3, 0,
                    "data not used", 0, 15, 15, 15)
data_used_inst ( clk, 1, load, data, out_reg);
```

The transfer is triggered by the signal `load`, from `data[15:3]` to *`out_reg`*`[12:0]`, and it has to happen within 1 to 3 clock cycles. All 3 coverage levels are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_data_used #(.severity_level(0),.sleft(15),
 .sright(3),.dleft(12),.dright(0),
 .start(1),.finish(3),.edge_expr(0),
 .msg("data not used"),.category(0),
 .coverage_level_1(15),.coverage_level_2(15),
 .coverage_level_3(15)
 data_used_inst (.clk(clk),.reset_n(1),.trigger(load),
 .src(data),.dest(out_reg));
```

assert_driven

Ensure that all bits of *exp* are driven (none are floating z or x). The check is not enabled unless *reset_n* evaluates true (1). The check is performed on the active *clk* edge specification.

Syntax

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

Arguments

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

exp
: Signal being tested.

Coverage Mode

cov_level_2_0 (bit 0 set in coverage_level_2)
: Cover point, cover_observed_driven_X_or_Z, indicates which bits were either x or z at any point of time during the simulation.

  NOTE: Since the default coverage is always level 1, this coverage is not enabled by default! There is no coverage at level 1 and at level 3.

Example

```
assert_driven #(0, 16, 1, "bad X or Z", 0, 1, 0)
driven_inst (clk, 1 , sig);
```

It will report a failure when at least one bit in the 16-bit bitvector *exp* was X or Z. Sampling is on negedge of clk due to edge_expr = 1. Coverage statistics for the level 2 will be generated by default.

NOTE: When this checker is used with SV Compiler for formal tools, the assertion is removed because the tool uses only 2-state signal values (case equality === with 1'bx is not synthesizable).

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_driven #(.severity_level(0),.bw(16),
 .edge_expr(1),.msg("bad X or Z"),.category(0),
 .coverage_level_1(1),.coverage_level_2(0),
 .coverage_level_3(0))
driven_inst (.clk(clk),.reset_n(1),.exp(sig));
```

assert_dual_clk_fifo

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

Syntax

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

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

`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. 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. Default = 0.

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

oflow_chk
: When an *enq* bit is asserted 1: If *oflow_chk* evaluates true, ensures that queue does not overflow the max size given in *depth*. *depth* (maximum size = t 2**16). Default = 1.

uflow_chk
: When a *deq* bit 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. Default = 1.

value_chk
: If *value_chk* evaluates true, ensures *deq_data* as selected by the same position as the highest priority *deq* bit is the same as that 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.

enq_clk
: Clock signal that synchronizes the enqueue side of the queue.

`enq`
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.

`enq_data`
Data being enqueued.

`deq_clk`
Clock signal that synchronizes dequeue side of the queue.

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

`deq_data`
Data being dequeued compared with the value at the head of the queue.

Failure Modes

Assertion `assert_fifo_dcq_overflow` will report a failure when enq is issued while the FIFO is full.

Assertion `assert_fifo_dcq_underflow` will report a failure when the dequeue command is issued and the FIFO is empty at that time.

Assertion `assert_fifo_dcq_value_chk` will report failure when there is a dequeue, the FIFO is not empty and the dequeued data does not match that on *`deq_data`* port..

Assertion `assert_fifo_dcq_hi_water_chk` will report a failure if the FIFO is filled above the high-water mark.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_number_of_enqs` indicates that an enqueue occurred.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_number_of_deqs`, indicates that a dequeue occurred.

`cov_level_1_2 (bit 2 set in coverage_level_1)`
Cover property `cover_enq_followed_eventually_by_deq` indicates that an enqueue was followed later by a dequeue.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_outstanding_contents` indicates the fill levels reached at least once, sampled whenever the FIFO occupancy changed.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_fifo_hi_water_chk` that the high water mark was reached on an enqueue.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_number_of_empty` indicates that empty was reached on dequeue.

`cov_level_3_2 (bit 2 set in coverage_level_3)`
Cover property `cover_number_of_full` indicates that full was reached on enqueue.

Example

```
assert_dual_clk_fifo #(.depth(10), .elem_sz(16), .deq_edge_expr(1),
              .coverage_level_1(15), .coverage_level_2(15),
              .coverage_level_3(15))
   dcq_inst (reset_n, eclk, enqueue, data_in,
             dclk, dequeue, data_out);
```

It will reset the queue when `reset_n` is 0 (must cover at least one `posedge` (by default) `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. All 3 coverage levels are enabled.

assert_fifo

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

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);
```

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

`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. 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. Default = 0.

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

oflow_chk
: When an *enq* bit is asserted 1: If *oflow_chk* evaluates true, ensures that queue does not overflow the max size given in *depth*. *depth* (maximum size = t 2**16). Default = 1.

uflow_chk
: When a *deq* bit 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. Default = 1.

value_chk
: If *value_chk* evaluates true, ensures *deq_data* as selected by the same position as the highest priority *deq* bit is the same as that at the head of the queue. Default = 1.

pass_thru
: If an enqueue and dequeue operation happens simultaneously on an empty queue, then the behavior depends on the *pass_thru* parameter (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. Default = 0.

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 a dequeue operation 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.

enq
: 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.

enq_data
: Data being enqueued.

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

deq_data
: Data being dequeued.

Failure Modes.

Assertion `assert_fifo_overflow` will report a failure when *`enq`* is issued while the FIFO is full.

Assertion `assert_fifo_underflow` will report a failure when the *`deq`* command is issued delayed by *`deq_lat`* and the FIFO is empty at that time (and no simultaneous *`enq`* with *`pass_thru`* is enabled).

Assertion `assert_fifo_value_chk` will report (`sva_v_q[sva_v_head_ptr] == deq_data)`) as the failing expression when there is a dequeue and the FIFO is not empty and the data does not correspond to the expected value. It will report (`enq_data == deq_data`) as the failing expression if there is a dequeue and `pass_thru` is enabled (=1), otherwise if there is a dequeue on an empty FIFO it is a failure.

Assertion `assert_fifo_hi_water_chk` will report a failure if the FIFO fill crosses the high-water mark.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `cover_number_of_enqs` indicates that there was an enqueue operation.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property `cover_number_of_deqs` indicates that there was a dequeue operation.

`cov_level_1_2 (bit 2 set in coverage_level_1)`

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

`cov_level_1_3 (bit 3 set in coverage_level_1)`

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

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_outstanding_contents` reports which FIFO fill levels have been reached at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_fifo_hi_water_chk` indicates that the high water mark was reached on an enqueue.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_simultaneous_enq_deq_when_empty` indicates that there were simultaneous enqueue and dequeue operations on an empty queue.

`cov_level_3_2 (bit 2 set in coverage_level_3)`
Cover poperty `cover_simultaneous_enq_deq_when_full` indicates that there were simultaneous enqueue and dequeue operations on a full queue.

`cov_level_3_3 (bit 3 set in coverage_level_3)`
Cover property `cover_number_of_empty` indicates that empty was reached on dequeue.

`cov_level_3_4 (bit 4 set in coverage_level_3)`
Cover property `cover_number_of_full` indicates that full was reached on enqueue.

Example:

```
assert_fifo
         #(.depth(10), .elem_sz(16),.coverage_level_1(15),
         .coverage_level_2(1), .coverage_level_3(31)  )
SVA_FIFO_inst (clk, !reset, enqueue, data_in, dequeue, data_out);
```

A FIFO is initialized anytime *reset* is 1 (synchronously with posedge clk). There are up to 10 elements in the FIFO, the size of the data is 16 bits, *high_water* mark is by default 0 (disabled), *data_in* is enqueued when *enqueue* is 1 with no latency, *data_out* must be equal to that at the head of the FIFO when *dequeue* is 1 with latency, and overflow, underflow and value checks are enabled with pass through when empty is allowed. All 3 coverage levels are enabled.

assert_hold_value

Ensure that *exp* 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. *bw* is the bit width of *exp*.

The check is not enabled unless *reset_n* evaluates true (1). The check is performed on the active *clk* edge specification and is triggered by a value change of *exp* to *value*.

`min = max = 0` means that *exp* will change every cycle (i.e., keeps the value for one cycle).

`min = max = 1` means that *exp* will keep *value* for two cycles.

A value `min > 0` and `max = 0` is interpreted as the open-ended interval `[min:$]`.

Syntax

```
assert_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);
```

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`
Expression or signal being tested.

`value`
Value to hold by *`exp`*.

Failure Modes

The assertion assert_hold_value will report failure if *`exp`* does not hold the value for at least `min+1` clock ticks or if it does not change from *`value`* at `max+1` clock ticks at the latest.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
Cover property, `cover_nb_of_exp_changes`, indicates that the *`exp`* changed value.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
Cover point, `observed_hold_time`, reports the observed number of clock cycles during which *`exp`* was held at value triggered whenever there was a transition on *`exp`* to value.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_hold_value_for_min_clks`, indicates that the *`exp`* was held exactly equal to value for the specified *`min_clks`*.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_hold_value_for_max_clks`, indicates that the *`exp`* was held exactly equal to value for the specified *`max_clks`*.

Example

```
assert_hold_value #(0, 3, 0, 4, 1,
"hold_value_error",0,15,15,0)
```

```
hold_value_inst ( clk, 1, sig, 4'h5);
```

A failure is reported if sig did not stay at 5 for at least 4 clock ticks. Since `max = 0` there cannot be a failure on sig not changing from the value of 5 because the interval `[3:0]` is translated into the open-ended interval `[3:$]`.

All Level_1 and Level_2 coverages are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_hold_value #(.severity_level(0),.min(3),
 .max(0),.bw(4),.edge_expr(1),.msg("hold_value_error"),
 .category(0), .coverage_level_1(15),
 .coverage_level_2(15),.coverage_level_3(0))
 hold_value_inst (.clk(clk),.reset_n(1),.exp(sig),
 .value(4'h5));
```

assert_memory_async

Ensures the integrity of an asynchronous memory contents and accesses.

Syntax

```
assert_memory_async [#(severity_level, data_bits,
                   addr_bits, mem_sz, addr_chk, init_chk,
                   read1_chk,write1_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);
```

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, operation is indicated by negative edge.

`r_edge_expr`
The active clock edge of `rclk`. Default = 0, operation is indicated by negative edge.

`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.

Note the following:

- 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 (i.e., it is assumed that they are positive pulses). This is the behavior when *`r_edge_expr`* (resp. *`w_edge_expr`*) is 0. When *`r_edge_expr`* is 1 (resp. *`w_edge_expr`*), the *`ren`* (resp. wen) signal is complemented and the positive edge of the (negative) pulse is used.
- Whenever the port *`reset_n`* is 0, it disables reading and writing to the memory.

- Al 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 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 sampling is by negedge of ren (wen) if r_edge_expr (w_edge_expr) is 0 and by posedge of ren (wen) if r_edge_expr (w_edge_expr) is 1...

Failure Modes

Assertion `assert_memory_async_init` will report failure when the location was not initialized before reading as determined by negedge of *ren* if *r_edge_expr* is 0 and by posedge of *ren* if *r_edge_expr* is 1..

Assertion `assert_memory_async_waddr_chk` will report failure when the write address was not within the limits just before negedge of *wen*.

if *w_edge_expr* is 0 and by posedge of *wen* if *w_edge_expr* is 1.

if *r_edge_expr* (w_edge_expr) is 0 and by posedge of *ren* (wen) if *r_edge_expr* (w_edge_expr) is 1.

Assertion `assert_memory_async_raddr_chk` will report failure when the write address was not within the limits just before negedge of *ren* if *r_edge_expr* is 0 and by posedge of *ren* if *r_edge_expr* is 1.

Assertion `assert_memory_read1_chk` will report failure when the variables recording reads and writes indicated the same state as sampled by negedge of *`ren`*. Hence, there was more than one read between writes.

Assertion `assert_memory_async_write1_chk` will report failure if after the memory location was initialized the location was not read before the next write as sampled by negedg *`wen`* if *`w_edge_expr`* is 0 and by posedge of *`wen`* if *`w_edge_expr`* is 1.

Assertion `assert_memory_async_val_chk` will report failure if there is a disagreement between the value in the design *`rdata`* and that stored in the memory mirror as sampled by negedge of *`ren`* if *`r_edge_expr`* is 0 and by posedge of *`ren`* if *`r_edge_expr`* is 1.

The *`ren`* and *`wen`* signals are qualified by *`reset_n`*, i.e., no read or write takes place if *`reset_n`* is asserted low (0).

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
Cover property, `cover_number_of_reads`, indicates that there was a read operation to any address.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_number_of_writes`, indicates that there was a write operations to any address.

`cov_level_1_2 (bit 2 set in coverage_level_1) :`
Cover property, `write_followed_by_read`, indicates that there was a write followed by a read to the same address.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
Cover point, `read`, reports which addresses within the range [`start_addr : end_addr`] were read at least once.

`cov_level_2_1 (bit 1 set in coverage_level_2) :`
Cover point, `write`, reports which addresses within the range [`start_addr : end_addr`] were written at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_two_or_more_writes_without_intervening_read`, indicates that two writes occurred to the same (any) address without an intervening read operation to that address.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_two_or_more_reads_without_intervening_write`, indicates that two reads occurred to the same (any) address without an intervening write operation to that address.

`cov_level_3_2 (bit 2 set in coverage_level_3) :`
Cover property, `read_to_start_addr`, indicates that a read operation occurred to the address *`start_addr`*.

`cov_level_3_3 (bit 3 set in coverage_level_3)`
Cover property, `write_to_start_addr`, indicates that a write operation occurred to the address *`start_addr`*.

`cov_level_3_4 (bit 4 set in coverage_level_3) :`
Cover property, `read_to_end_addr`, indicates that a read operation occurred to the address *`end_addr`*.

`cov_level_3_5 (bit 5 set in coverage_level_3) :`
Cover property, `write_to_end_addr`, indicates that a write operation occurred to the address *`end_addr`*.

`cov_level_3_6 (bit 6 set in coverage_level_3) :`
Cover property, `write_followed_by_read_to_start_addr`, that there was a write operation followed by a read to the address *`start_addr`*.

`cov_level_3_7 (bit 7 set in coverage_level_3) :`
Cover property, `write_followed_by_read_to_end_addr`, indicates that there was a write operation followed by a read to the address *`end_addr`*.

Example

```
assert_memory_async #(0, 16, 16, 16'h1000, 1, 1, 1, 1, 1)
         mem_inst (1'b1, 16'h0000, 16'h0fff,
              ren, addr, rdata,
              wen, addr, wdata);
```

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, init_chk, read1_chk, write1_chk and value_chk are enabled. The default message is output upon failure. reset_n is 1, hence always enabled. Category is 0 by default. Since both w_edge_expr and r_edge_expr parameters are 0 by default, positive ren and wen pulses are assumed. Coverage Level 1 is enabled by default (all cover properties).

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_memory_async #(.severity_level(0),.data_bits(16),
 .addr_bits(16),.mem_sz(16'h1000),.addr_chk(1),
 .init_chk10),.read1_chk(1),.write1_chk(1),
 .value_chk(1))
mem_inst (.reset_n(1'b1),.start_addr(16'h0000),
 .end_addr(16'h0fff),.ren(ren),.raddr(addr),
 .rdata(rdata),.wen(wen),.waddr(addr),.wdata(wdata));
```

assert_memory_sync

Ensures the integrity of a synchronous memory contents and accesses.

Syntax

```
assert_memory_sync [#(severity_level, 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,
                  category, coverage_level_1,
                  coverage_level_2, coverage_level_3)]
inst_name (start_addr, end_addr, ren, raddr, rclk,
            rdata, wen, waddr, wclk, wdata);
```

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`*.

Failure Modes

Assertion `assert_mem_init` will report failure when the memory word was not initialized before reading and there was no simultaneous read and write with pass through allowed.

Assertion `assert_mem_waddr_chk` will report failure when the write address was not within the limits when *`wen`* was asserted.

Assertion `assert_mem_raddr_chk` will report failure when the read address was not within the limits when *`ren`* was asserted.

Assertion `assert_mem_conflict_chk` will report failure when *`wen`* and *`ren`* were asserted simultaneously and the write and read addresses were the same sampled by the write clock *`wclk`*.

Assertion `assert_mem_read1_chk` will report failure when at the time of failure the sva variables recording reads and writes indicated the same state. Hence there was more than one read between writes.

Assertion `assert_mem_write1_chk` will report failure when after a memory location was initialized, neither the location was read before the next write nor there was a simultaneous read and write (as sampled by the write clock *`wclk`*) with *`pass_thru`* disabled.

Assertion `assert_mem_val_chk` will report failure when pass through is allowed and read and write happen at the same time (as sampled by the read clock *`rclk`*) to the same address and the data are not equal.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_number_of_reads` indicates that a read operation occurred to any address.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_number_of_writes` indicates that a write operation occurred to any address.

`cov_level_1_2 (bit 2 set in coverage_level_1)`
Cover property `write_followed_by_read` indicates that a write was followed by a read to the same address.

`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `read` indicates which addresses within the range `start_addr` : `end_addr` have been read at least once.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
Cover point `write` indicates which addresses within the range `start_addr` : `end_addr` have been written at least once.

`cov_level_2_2 (bit 2 set in coverage_level_2)`
Cover point `delay_from_read_to_write` reports which delays were observed at least once from the last read in a sequence of reads to the nearest subsequent write.

`cov_level_2_3 (bit 3 set in coverage_level_2)`
Cover point `delay_from_write_to_read` reports which delays were observed at least once from the last write in a sequence of writes to the nearest subsequent read.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

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

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_two_or_more_reads_without_intervening_write` indicates that two reads occurred to the same (any) address without an intervening write operation to that address.

`cov_level_3_2 (bit 2 set in coverage_level_3)`
Cover property `simultaneous_read_and_write_to_same_addr` indicates that there were (quasi)simultaneous read and write operations to the same (any) address as seen by the read clock `rclk`.

`cov_level_3_3 (bit 3 set in coverage_level_3)`
Cover property `simultaneous_read_and_write_to_different_addr` indicates that there were (quasi)simultaneous read and write operations to different addresses as seen by the read clock `rclk`.

`cov_level_3_4 (bit 4 set in coverage_level_3)`
Cover property `read_to_start_addr` indicates that read operations occurred to the address `start_addr`.

`cov_level_3_5 (bit 5 set in coverage_level_3)`
Cover property `write_to_start_addr` indicates that operations occurred to the address `start_addr`.

`cov_level_3_6 (bit 6 set in coverage_level_3)`
Cover property `read_to_end_addr` indicates that read operations occurred to the address `end_addr`.

`cov_level_3_7 (bit 7 set in coverage_level_3)`
Cover property `write_to_end_addr` indicates that write operations occurred to the address `end_addr`.

`cov_level_3_8 (bit 8 set in coverage_level_3)`
Cover property `write_followed_by_read_to_start_addr` indicates that a write operation was followed by a read to the address start_addr.

`cov_level_3_9 (bit 9 set in coverage_level_3)`
Cover property `write_followed_by_read_to_end_addr` indicates that a write operation was followed by a read to the address end_addr.

Example

```
assert_memory_sync #(1, 16, 16, 16'h1000,1 ,1 , 1, 1, 0, 0,  1)
     mem_inst (16'h0000, 16'h0fff,
            ren, addr, clk, rdata,
            wen, addr, clk, wdata );
```

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, the same clock is used for read and write, sampling by posedge in both cases (by default), *`addr_chk`* and *`init_chk`* are enabled by default, *`conflict_chk`* and *`pass_thru`* are enabled, *`read1_chk`* and *`write1_chk`* are disabled by default, and *`value_chk`* is enabled. The default message is output upon failure. By default coverage Level 1 is enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_memory_sync #(.severity_level(1), .data_bits(16),
```

```
    .addr_bits(16), .mem_sz(16'h1000),.addr_chk(1),
    .init_chk(1) , .conflict_chk(1),.pass_thru(1),
    .read1_chk(0), .write1_chk(0), .value_chk(1))
  mem_inst (.start_addr(16'h0000), .end_addr(16'h0fff),
   .ren(ren), .raddr(addr), .rclk(clk), .rdata(rdata),
   .wen(wen), .waddr(addr), .wclk(clk), .wdata(wdata) );
```

assert_multiport_fifo

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

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);
```

Arguments

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

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

`no_ports`
: Number of ports. Default = 2.

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

`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. 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. Default = 0.

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

oflow_chk
: When an *enq* bit is asserted 1: If *oflow_chk* evaluates true, ensures that queue does not overflow the max size given in *depth*. *depth* (maximum size = t 2**16). Default = 1.

uflow_chk
: When a *deq* bit 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. Default = 1.

value_chk
: If *value_chk* evaluates true, ensures *deq_data* as selected by the same position as the highest priority *deq* bit is the same as that at the head of the queue. Default = 1.

pass_thru
: If an enqueue and dequeue operation happens simultaneously on an empty queue, then the behavior depends on the *pass_thru* parameter (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. Default = 0.

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 a dequeue operation 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.

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 the bit 0 is the lowest priority, 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 concatenation of the data from the different ports. It is assumed that it is 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_dat* must be valid after *enq_lat* clock cycles. Only the highest priority data is actually enqueued.

deq_data

A concatenation of the data from the different ports. It is assumed that it is dimensioned as [no_ports*elem_size-1:0], 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_dat* must be valid after *deq_lat* clock cycles. The highest priority data is compared with the data at the head of the queue if *value_chk* evaluates true.

Failure Modes

Assertion assert_multiport_fifo_overflow will report a failure when *enq* is issued while the FIFO is full.

Assertion assert_multiport_fifo_underflow will report a failure when the *deq* command is issued and the FIFO is empty.

Assertion assert_multiport_fifo_value_chk will report failure when there is a dequeue and the FIFO is not empty and the dequeued value does not correspond to that on deq_data.

Assertion assert_fifo_hi_water_chk will report a failure if the FIFO is filled above the high-water mark.

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1) :

Cover property, cover_number_of_enqs, indicates that an enqueue operation occurred. The cover is incremented whenever enq > 0.

cov_level_1_1 (bit 1 set in coverage_level_1) :

Cover property, cover_number_of_deqs, indicates that a dequeue operation occurred. The cover is incremented whenever deq > 0.

cov_level_1_2 (bit 2 set in coverage_level_1) :

Cover property, `cover_port_enqued`, indicates for each port separately that that an enqueue operation occurred.

`cov_level_1_3 (bit 3 set in coverage_level_1) :`
Cover property, `cover_port_dequed`, indicates for each port separately that that a dequeue operation occurred.

`cov_level_1_4 (bit 4 set in coverage_level_1) :`
Cover property, `cover_enq_followed_eventually_by_deq_per_port`, indicates for each port that a successful *`enq`* on a port was followed eventually by a successful *`deq`*.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
Cover point, `observed_outstanding_contents`, reports on the FIFO fill levels that occurred at least once.

`cov_level_2_1 (bit 1 set in coverage_level_2) :`
Cover property, `cover_ignored_enque`, indicates for each port whether an enqueue was ignored due to other higher priority requests.

`cov_level_2_2 (bit 2 set in coverage_level_2) :`
Cover property, `cover_ignored_deque`, indicates for each port whether a dequeue was ignored due to other higher priority requests.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_fifo_hi_water_chk`, indicates that the high water mark was reached on an enqueue.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_simultaneous_enq_deq_on_same_port`, indicates for each port that successful enqueue and dequeue happened at the same time.

`cov_level_3_2 (bit 2 set in coverage_level_3) :`
Cover property, `cover_simultaneous_enq_deq_on_different_port`, indicates that successful enqueue and dequeue happened simultaneously on different ports.

`cov_level_3_3 (bit 3 set in coverage_level_3) :`
Cover property, `cover_simultaneous_enq_deq_when_empty`, indicates that there were successful simultaneous enqueue and dequeue operations on an empty queue.

`cov_level_3_4 (bit 4 set in coverage_level_3) :`
Cover poperty, `cover_simultaneous_enq_deq_when_full`, indicates that there were successful simultaneous enqueue and dequeue operations on a full queue.

`cov_level_3_5 (bit 5 set in coverage_level_3) :`
Cover property, `cover_number_of_empty`, indicates that empty queue was reached on a dequeue.

`cov_level_3_6 (bit 6 set in coverage_level_3) :`
Cover property, `cover_number_of_full`, indicates that full queue was reached on enqueue.

Example:

```
assert_multiport_fifo #(.severity_level(0), .depth(10),
                        .elem_sz(16) ,.pass_thru(1),
                        .coverage_level_3(255))
             mpfifo_inst (clk, reset_n,
                          enqueue, data_in,
                          dequeue, data_out );
```

In this example, a FIFO is initialized anytime reset is 0 (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 (by default, i.e., enqueue, dequeue are 2-bit wide, `data_in` and `data_out` are bitvectors of

2 * 16 = 32 bits); `data_in[16*i+:16]` is enqueued when *`enqueue[i]`* is 1 with no latency; `data_out[16*i+:16]` 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 by default with pass through when empty is allowed. All level 3 coverages are enabled.

assert_mutex

Ensures that *a* and *b* never evaluate true at the same time. The check is not enabled unless *reset_n* evaluates true (1). The checks is performed on the active *clk* edge specification.

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);
```

Arguments

a
: First signal being tested.

b
: Second signal being tested.

Failure Modes

The assertion assert_mutex will report failure when *a* and *b* are both sampled 1 at the positive edge of *clk*.

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1)
: Cover property cover_changes_on_a indicates that a changed value.

cov_level_1_1 (bit 1 set in coverage_level_1)
: Cover property cover_changes_on_b indicates that b changed value.

Example

```
assert_mutex mutex_inst( clk, 1, gnt[0], gnt[1]);
```

In this example, `assert_mutex` verifies that `gnt[0]` and `gnt[1]` are never 1 at the same time as sampled by posedge of clk. The checker is always enabled because *`reset_n`* is constant 1. By default Level 1 coverages will be enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_mutex mutex_inst
(.clk(clk),.reset_n(1),.a(gnt[0]),.b(gnt[1]) ;
```

assert_next_state

Ensures that when *exp* is in current state *cs* that *exp* will transition to one of the specified legal next states in *ns*.

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);
```

Arguments

`no_ns`
Specifies the number of legal next states possible from current state (`cs`). Default = 1.

`width`
The number of bits in the signal being tested (`exp`), the current state (`cs`), and each element of the bitvector *ns* of the concatenated legal state values (`ns`). 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 equal to 1, then a transition to none of the values in *ns* may occur. Otherwise, if `disallow` is equal to 0 then the check ensures a transition to one of the states occurs.

`exp`
Signal being tested.

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

`ns`
A bitvector of concatenated legal states that `exp` can transition to from `cs`.

The state transitions are checked at the active *`clk`* edge specification. For example, if *`width`* = 3, *`ns`* = 2, and the values are 3'b000 and 3'b110 then the value bound to the *`ns`* port should be 6'b000_110.

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 is imposed on when it must change.

  `min_hold = m`, `max_hold = n, n => m` 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.

Assumptions

`no_ns > 0` (and the *`ns`* array should have the correct number of elements. *`min_hold`* and *`max_hold`* both > 0, except when *`min_hold`* > 0 *`and max_hold`* = 0*.*

Failure Modes

When *`disallow`* = 0, the assertion `assert_next_state` will report failure when *`exp`* is not one of the expected next states.

When *`disallow`* = 1, the failure of the assertion `assert_next_state` means that the next state values and the hold intervals were satisfied, indicating that the design reached a disallowed state within the correct expected timing.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property, *`cover_nb_of_next_state_transitions`*, indicates that the *`expr`* changed its state from `cs` to a state in *`ns`* (`disallow = 0`) or from the curent state to a state outside *`ns`* (`disallow = 1`), while meeting the hold times in the current state as defined by the `min_hold` and `max_hold` parameters.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point, `cover_observed_hold_time`, reports on the number of clock cycles the test_expr remained in the current state `cs` before making a transition to one of the *`next_states`* (`disallow = 0`) or one of the states other than the *`next_states`* (`disallow = 1`).

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property, `cover_nb_of_transitions_from_cs_to_each_ns[i]`, indicates that the *`test_expr`* made a transtion from the current state *`cs`* to the 'i'th state in the *`ns`*. This coverage is only generated when `disallow = 0`.

Example

The example will verify that when *`state_var`* takes on the value 0, then on the next posedge *`clk`* it will take on one of the values 0 or 2 (`disallow = 1`). The assertion is enabled when reset is 0 .

```
assert_next_state
     #(0, 2, 4, 1, 1, 0, 0, "nxts_inst failed", 0, 1, 1, 1)
   nxt_st_inst (clk, !reset, state_var, 4'h0, {4'h1, 4'h2});
```

All three coverage statistics will be generated for this checker.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_hold_value #(.severity_level(0),.no_ns(2),
 .width(4),.min_hold(1),.max_hold(1),.disallow(0),
 .edge_expr(0),.msg("nxts_inst failed"),
 .category(0), .coverage_level_1(1),
 .coverage_level_2(1),.coverage_level_3(1))
hold_value_inst (.clk(clk),.reset_n(!reset),
 .exp(state_var),.cs(4'h0), .ns({4'h1, 4'h2}));
```

assert_no_contention

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. Specify 0 for no minimum bus quiet time between bus transactions.

Syntax

```
assert_no_contention [#(severity_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);
```

Arguments

min_quiet
: The minimum number of cycles between bus activity during which all bits in *en_vector* must be 0. Default = 0.

max_quiet
: The maximum number of cycles that the bus can remain quiet (without an active driver). 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 (bus1). Default = 2.

en_vector
: Enable signals for bus drivers as a vector of cache bits.

bus1

Bus signal being tested.

Failure Modes

The assertion `assert_no_contention_no_xs` will report failure when the bus had an x or z on one of its bits when at least one driver was enabled (i.e., when *`en_vector`* was not all 0).

NOTE: The assertion `assert_no_contention_no_xs` is eliminated when the template or unit is used with Magellan. This is because the tool uses only 2-state signal values.

The assertion `assert_no_contention_1_driver` will report failure when there was more than one driver enabled.

The assertion `assert_no_contention_quiet_time` will report failure when the bus was not quiet for *`min_quiet`* clock cycles or that it remained quiet after *`max_quiet`* cycles. (Or that there was more than one driver after *`max_quiet`* cycles, however, this can be checked by observing the assertions `assert_no_contention_1_driver`.)

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `driver[i].cover_driver_enable` indicates indicates that each bit `[i]` in *`en_vector`* was set to 1 (enabled).

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point `observed_quiet_cycles` reports the numbers of quiet cycles that occurred at least once. The report will also indicate the specified *`min_quiet`* and *`max_quiet`* values.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property

`cover_no_contention_quiet_time_equal_to_min_quiet`

indicates that the observed quiet time was exactly equal to the specified *`min_quiet`* value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`

Cover property

`cover_no_contention_quiet_time_equal_to_max_quiet`

indicates that the observed quiet time was exactly equal to the specified *`max_quiet`* value.

Example

```
assert_no_contention #(0, 0, 0, 2, 32)
        no_contention_inst (clk , 1, bus_enables, bus);
```

In this example, `assert_no_contention` verifies that signal `bus` that is 32 bits wide is driven at every clock cycle because *`min_quiet`* `=` *`max_quiet`* `= 0,` and there are two drivers because *`bw_en`* takes on the default value of 2. *`reset_n`* is a constant 1. Hence the checker is enabled all the time. By default, Level 1 coverage is enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_no_contention #(.severity_level(0),.min_quiet(0),
 .max_quiet(0),.bw_en(22),.bw_bus(32))
no_contention_inst (.clk(clk),.reset_n(1),
 .en_vector(bus_enables),.bus1(bus));
```

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);
```

Arguments

`sop`
: Signal indicating start of packet.

`eop`
: Signal indicating end of packet.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
: Cover property, `cover_sop`, indicates the number of sop.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
: Cover point `observed_sop_eop_delay` provides a profile of the observed separation in clock cycles between `sop` and `eop`. The report shows which such delays were observed at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_simultaneous_sop_eop`, indicates the number of times eop and sop occured together.

Example

```
assert_packet_flow  #( .coverage_level_1(-1),
                       .coverage_level_2(-1),
                       .coverage_level_3(-1))
packet_1 (clk, reset_n, sop, eop);
```

In this example, `assert_packet_flow` will report failure whenever `eop` happens without a preceding `sop` and whenver `sop` is followed by another `sop`. All the three coverage levels are enabled.

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 continousely 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);
```

Arguments

`min`
: Minimum number of clock cycles for which `test_expr` need to be true within period number of clock cycles.

`max`
: Maximum number of clock cycles for which `test_expr` need to be true within period number of clock cycles.

`period`
: Number of clock cycles during which `test_expr` must be true [`min:max`] number of times.

`trigger`
: Signal indicating the start of evaluation

`test_expr`
: Logical expression that is being verified at the positive edge of clk.

Coverage Modes

```
cov_level_1_0 (bit 0 set in coverage_level_1) :
```

Cover property, `cover_rate`, indicates the number of times the sequence matched within the limits specified.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_trigger`, indicates the number of times trigger got asserted.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
Cover point `observed_nb_of_test_expr_high_within_period` provides a profile of the number of clock cycles `test_expr` was high within period number of clock cycles. The report shows if any of these occurrences were observed at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_rate_min`, indicates the number of times the sequence matched exactly at `min` limit.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_rate_max`, indicates the number of times the sequence matched exactly at `max` limit.

Example

```
assert_rate  #(.min(3), .max(5), .period(6),
               .coverage_level_1( -1),
               .coverage_level_2( -1),
               .coverage_level_3( -1))
rate_1 (clk, reset_n, trigger, test_xpr);
```

In this example, `assert_rate` will report failure if `test_expr` does NOT remain high intermittently or continousely for minimum of 3 clock cycles to a maximum of 5 clock cycles within a continous period of 6 clock cycles once `trigger` is sampled asserted, where `trigger` is a pulse indicating the start of the sequence evaluation. All three coverage levels are enabled.

assert_reg_loaded

Ensure 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* evaluates true (whichever occurs first).

If used to control the end time *stop* should become true at least one clock cycle after *dst_reg* is loaded which means minimum 2 cycles after *trigger*.

The *src* value is "captured" when *trigger* evaluates true.

The check is made by comparing the *src* value against the *dst_reg* contents during the "load window". Once the register has loaded the value during the window, the check terminates with success.

The check is not enabled unless *reset_n* evaluates true. The check is performed on the active *clk* edge specification.

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);
```

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 = 0.

bw
: The number of bits in the source data (src) and the register under test (*dst_reg*). Default = 2.

  The width (bw) of src and *dst_reg* should correspond to the actual width of these operands.. 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.

Assumptions

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

The width *bw* of the *src* and *dst_reg* should correspond to the actual width of these operands. If *bw* is less than the actual width, then the check may fail if the bits above the *bw* are not 0. If *bw* is more than the actual width, then the additional bits are taken to be 0.

Failure Modes

The assertion `assert_reg_loaded` will report failure when either *`stop`* became true or maximum delay *`end_cycle`* was reached while *`dst_reg`* was not yet equal to the stored value.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1) :`
Cover property, `cover_nb_of_triggers`, indicates that the trigger rose.

`cov_level_1_1 (bit 1 set in coverage_level_1) :`
Cover property, `cover_reg_loaded`, indicates that the *`dst_reg`* was successfully loaded with *`src`* within the specified time limits.

`cov_level_2_0 (bit 0 set in coverage_level_2) :`
Cover point, `observed_dst_reg_load_time`, reports on the number clock cycles taken to load the *`dst_reg`*. The clock cycles are counted from the first assertion of *`trigger`*.

`cov_level_3_0 (bit 0 set in coverage_level_3) :`
Cover property, `cover_nb_of_times_stop_happened_at_or_prior_to_end_cycle`, indicates that a *`stop`* happened at or prior to *`end_cycle`* after the occurrence of a *`trigger`*.

`cov_level_3_1 (bit 1 set in coverage_level_3) :`
Cover property, `cover_nb_of_times_no_stop_happened_till_end_cycle`, indicates that no *`stop`* happened till the *`end_cycle`* expired after the occurrence of a *`trigger`*.

`cov_level_3_2 (bit 2 set in coverage_level_3) :`
Cover property, `cover_data_transfer_exactly_at_end_cycles`, indicates that the *`dst_reg`* was loaded wih *`src`* exactly at *`end_cycle`*.

`cov_level_3_3 (bit 3 set in coverage_level_3) :`
Cover property, `cover_data_transfer_exactly_at_delay_cycles`, indicates that the *`dst_reg`* was loaded wih *`src`* exactly at *`delay`*.

Example

```
assert_reg_loaded #(0, 1, 3, 8)
          rg_ld_inst ( clk, 1, load, data, reg_data, 0);
```

In this example *`reg_data`* (of width 8 bits) is to be loaded by the value of *`data`* at posedge *`clk`* within 1 to 3 clock cycles from *`load`* evaluating true. By default Level 1 coverages are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_reg_loaded #(.severity_level(0),.delay(1),
 .end_cycle(3),.bw(32),.edge_expr(8)
rg_ld_inst (.clk(clk),.reset_n(1),.trigger(load),
 .src(data),.dst_reg(reg_data),.stop(0));
```

assert_req_ack_unique

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

Syntax

(Unit syntax only)

```
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)
```

Arguments

`min_time`

Defines the minimum time separation between a req and an ack. Default = 1.

`max_time`

Defines the maximum time separation between a req and an ack Default = 1.

`max_time_log_2`

Specifies the superior integer of log2 of `max_time`, used to dimension the data structures. Default = 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. Default = 0.
- `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.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_number_of_req` indicates that *`req`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_number_of_ack` indicates that *`ack`* was asserted.

`Level 2 and 3 are available only with version = 1 selection`
`cov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_latency` reports the *`req`*-to-*`ack`* latencies observed at least once.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
Cover point `outstanding_requests` reports the numbers of outstanding requests that occurred at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_ack_with_exact_min_lat` indicates that the observed latency was exactly equal to the specified *`min_time`* value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`

Cover property `cover_ack_with_exact_max_lat` indicates that the observed latency was exactly equal to the specified *`max_time`* value.

Example

```
assert_req_ack_unique #(1, 20, 100, 7, 1, 0,
                        "req_ack failed", 0, 3, 0, 3)
            req_grant_1_1 (clk, reset_n, request, grant);
```

The instance `req_grant_1_1` of `assert_req_ack_unique` verifies that for each request there is a grant received within 20 and 100 posedge edges of `clk` (`max_time_log2` is thus set to 7). The checker is reset on `reset_n` low. Version 1 is used, i.e., using time stamps. Coverage Levels 1 and 3 are selected (coverage_level_1 = 3 and coverage_level_3 = 3).

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_req_ack_unique #(.severity_level(1),.min_time(20),
 .max_time(100),.max_time_log_2(7),.version(1),
 .edge_expr(0),.msg("req_ack failed"),.category(0),
 .coverage_level_1(3),.coverage_level_2(0),
 .coverage_level_3(3))
req_grant_1_1 (.clk(1'b1),.reset_n(reset_n),
 .req(request),.ack(grant));
```

assert_req_requires

Ensures that if the `trig_req` expression evaluates true then the `follow_req` 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.

Syntax

```
assert_req_requires #(severity_level, min_lat,
                      max_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);
```

Arguments

`min_lat`

Defines the minimum latency from `trig_req` to `trig_resp`. Default = 1.

`max_lat`

Defines the maximum latency from `trig_req` to `trig_resp`. `max_lat == 0` means that there is no upper bound. Default = 0.

Assumptions

Time separations (in clock ticks) are `t(trig_req) <= t(follow_req) < t(follow_resp) <= t(trig_resp)`, i.e., `trig_req` and `follow_req`, and `follow_resp` and `trig_resp` may coincide.

Note: If multiple overlapping evaluations are triggered, the latency coverage information may not be accurate since the vector does not distinguish which responses belong to which requests.

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`
Cover property `cover_no_of_trig_reqs` indicates that the *`trig_req`* was asserted.

`cov_level_1_1 (bit 1 set in coverage_level_1)`
Cover property `cover_cover_req_requires` indicates that the specified sequence occurred.

`ov_level_2_0 (bit 0 set in coverage_level_2)`
Cover point `observed_latency_btw_trig_req_and_trig_resp` reports which values of latency between *`trig_req`* and *`trig_resp`* were observed at least once.

`cov_level_2_1 (bit 1 set in coverage_level_2)`
Cover point `observed_latency_btw_trig_req_and_follow_req` reports which latencies between *`trig_req`* and *`follow_req`* were observed at least once.

`cov_level_2_2 (bit 2 set in coverage_level_2)`
Cover point `observed_latency_btw_follow_req_and_follow_resp` reports which latencies between *`follow_req`* and *`follow_resp`* were observed at least once.

`cov_level_2_3 (bit 3 set in coverage_level_2)`
Cover point `observed_latency_btw_follow_resp_and_trig_resp` reports which latencies between *`follow_resp`* and *`trig_resp`* were observed at least once.

`cov_level_2_4 (bit 4 set in coverage_level_2)`
Cover property `cover_trig_req_follow_req` indicates that there was a a `follow_req` after a *`trig_req`* within *`max_lat`* cycles.

`cov_level_2_5 (bit 5 set in coverage_level_2)`
Cover property `cover_trig_req_follow_req_follow_resp` indicates that there was a *`follow_req`* after a *`trig_req`* and then followed by *`follow_resp`* within *`max_lat`* cycles..

`cov_level_3_0 (bit 0 set in coverage_level_3)`
Cover property `cover_trig_resp_exactly_on_min_lat` indicates that the observed latency between *`trig_req`* and *`trig_resp`* was exactly equal to the *`min_lat`* value.

`cov_level_3_1 (bit 1 set in coverage_level_3)`
Cover property `cover_trig_resp_exactly_on_max_lat` indicates that the observed latency between *`trig_req`* and *`trig_resp`* was exactly equal to the *`max_lat`* 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.

Example:

```
assert_req_requires #(0, 1, 10, 0, "req_requires
fails", 0,
                      0, 63, 15)
    req_resp_inst(clk, 1, req, 1'b1, 1'b1, resp);
```

meaning that `min_lat` == 1, `max_lat` == 10, and coverage Levels 2 and 3 are selected.The assertion verifies that req is followed by resp within 10 clock cycles. All other parameters take default values (`posedge clk` sampling).

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_req_requires #(.severity_level(0),.min_lat(1),
 .max_lat(10),.edge_expr(0),.msg("req_requires fails"),
 .category(0), .coverage_level_1(15),
 .coverage_level_2(63), .coverage_level_3(3))
req_resp_inst (.clk(clk),.reset_n(1),
 .trig_req(req),.follow_req(1'b1),
 .follow_resp(1'b1),.trig_resp(resp));
```

assert_stack

Checks operations on a stack.

Syntax

```
assert_stack [#(severtiy_level, depth, elem_sz, hi_water_mark,
              push_lat, pop_lat, value_chk, push_pop_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);
```

Arguments

`depth`
: The maximum size of the stack, a compile-time integer constant < 2**16. Default = 2.

`elem_sz`
: The width of data elements, a compile-time constant. Default = 1.

`hi_water_mark`
: If `hi_water_mark` is a positive value, then the level of the fill of the queue after *`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 size decreases again to or below the mark. Default = 0.

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

`pop_lat`
: The number of *`clk`* cycles between `pop` being asserted 1 and `pop_data` being valid. Default = 0.

`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.

`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`*.
- *`reset_n`* asserted 0 initializes the stack to empty.
- 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 fill 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*.

Failure Modes

Assertion assert_stack_overflow will report a failure when *push* is issued and the stack is full. The reported time of failure is on the clock tick following the failed deferred push operation.

Assertion assert_stack_underflow will report a failure when *pop* is issued and the stack is empty (and no simultaneous *push*). The reported time of failure is on the clock tick following the failed deferred push operation.

Assertion assert_stack_value_chk will report failure when *pop_data* value does not match the expected top-of-stack value, provided the stack is not empty or when the stack is empty and there is a simultaneous *push* and the value pushed on the stack does not match the value popped off the stack.

Assertion assert_stack_hi_water_chk will report a failure if the stack is filled above the high-water mark.

Assertion `assert_stack_push_pop_fail` will report a failure if *`push`* and *`pop`* are enabled simultaneously (and *`push_pop_chk`* is 1).

Coverage Modes

`cov_level_1_0 (bit 0 set in coverage_level_1)`

Cover property `cover_number_of_pushes` indicates that there was a *`push`* operation.

`cov_level_1_1 (bit 1 set in coverage_level_1)`

Cover property `cover_number_of_pops` indicates that there was a *`pop`* operation.

`cov_level_1_2 (bit 2 set in coverage_level_1)`

Cover property `cover_push_followed_eventually_by_pop` indicates that a push was followed eventually by a *`pop`* without an intervening *`push`*.

`cov_level_2_0 (bit 0 set in coverage_level_2)`

Cover point `observed_outstanding_contents` reports which observed fill levels were observed at least once.

`cov_level_3_0 (bit 0 set in coverage_level_3)`

Cover property `cover_stack_hi_water_chk` indicates that the high water mark was reached.

`cov_level_3_1 (bit 1 set in coverage_level_3)`

Cover property `cover_simultaneous_push_pop` indicates that there were simultaneous *`push`* and *`pop`* operations.

`cov_level_3_2 (bit 2 set in coverage_level_3)`

Cover property `cover_simultaneous_push_pop_when_empty` indicates that there were simultaneous *`push`* and *`pop`* operations while the stack was empty.

`cov_level_3_3 (bit 3 set in coverage_level_3)`
Cover property `cover_simultaneous_push_pop_when_full` indicates that there were simultaneous *`push`* and *`pop`* operations while the stack was full.

`ccov_level_3_4 (bit 4 set in coverage_level_3)`
Cover property `cover_number_of_empty` indicates that the stack became empty after a *`pop`*.

`ov_level_3_5 (bit 5 set in coverage_level_3)`
Cover property `cover_number_of_full` indicates that the stack became full after a *`push`*.

Example:

```
assert_stack #(0, 10, 16, 8, 0, 0,  1, 0)
        stack_inst (clk, !sys_reset,
                    push, data_in,
                     pop, data_out);
```

When *`sys_reset`* is asserted 1 the stack is initialized. It is 10 elements deep and 16 bits wide. The water mark is set at 8 which means that 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 as push is asserted and *`data_out`* must be present at the same time as 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. Push-Pop check is disabled. Coverage Level 1 is selected by default.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_stack #(.severity_level(0),.depth(10),
 .elem_sz(16),.hi_water_mark(8),.push_lat(0),
 .pop_lat(0),.value_chk(1),.push_pop_chk(0))
```

```
stack_inst (.clk(clk),.reset_n(!sys_reset),.push(push),
.push_data(data_in),.pop(pop),.pop_data(data_out));
```

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*] delay. Both *issued_id* and *ret_id* are qualified by *reset_n* being 1. If *ret_sig* does not arrive at or before *max_lat* clock cycles after an *issued_sig*, an assertion will fire at max_lat+1 cycles. This is the only failure on *max_lat* reported until a *ret_sig* with that id arrives which will re-enable the check for that id. max_lat >= min_lat >= 1 must hold

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, nb_ret_per_issued);
```

Arguments

id_bw
: The max. number of bits to encode an id. Default = 2.There are max_ids id's at most at any time.

max_ids
: The maximum number of IDs. Default = 2.

  The maximum value is 2^{16}.

max_out_ids
: Indicates how many issued id's out of *max_ids* can have outstanding ret_sig's at any time. By setting max_out_ids > max_ids then effectively max_ids becomes the limit. 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 = 1.

Note the following:

`issued_sig`
Ensures that `max_out_ids` and *`max_ids`* are 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.

`issued_id`, *`ret_id`*, and *`reset_id`*
Values encoded using *`id_bw`* number of bits.

`reset_sig`
Resets the outstanding count of *`reset_id`* id is reset to 0, if that id has an outstanding return.

`nb_ret_per_issued`
Indicates how many *`ret_sig`* are required for each *`issued_sig`*. The port is included by defining the macro SVA_NEW_PORTS.

Important:

When the macro SVA_NEW_PORTS is defined, there is a additional port, nb_ret_per_issued, It is assumed valid when issued_sig is asserted and it indicates how many returns are required for that assertion of issued_sig and the associated id. For example, if nb_ret_per_issued == 1, then for each issued_sig, only one ret_sig with that id is allowed. If nb_ret_per_issued == 2, then the checker expects ret_sig to be asserted 2 times for that issued_sig. All ret_sig returns must satisfy the [min_lat : max_lat] latency with respect to the current issued_sig and the same id. That is, this interval of latencies applies to all nb_ret_per_issued cases. The width of this port is determined by the parameter max_ret_bw (default 2);

Note the following:

- If an *issued_id* exceeds the maximum number of outstanding *max_ids* or *max_out_ids,* an error is reported. Only those id's are counted that have *reset_n* == 1 at issuance.
- When *reset_sig* is asserted 1, *ret_id* must match an outstanding id. A returned *ret_id* is counted only against issued id's 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 period.
- To trigger a check for an issued ID, *reset_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 *reset_n* is asserted while at issuance of this ID `reset_n` was deasserted, this return is flagged as an error if there is no other outstanding request for that id.
- All checks are performed on the active *clk* edge specification.

- *min_lat* = 1, *max_lat* = 1 means that the ID must be returned on the next clock cycle after issuance.
- *min_lat* = m, *max_lat* = n, m <= n, means that the ID must be returned within the interval [m:n] clock ticks after issuance. *max_lat* must be less than or equal to 2**15.

Failure Modes

assertion assert_valid_id_max_lat will report failure when an id has been issued, but that id has not been returned at or before max_lat clock cycles. This will happen exactly at max_lat+1 cycles after the issue.

assertion assert_valid_id_min_lat will report failure when an id has been issued, but that id is returned before min_lat clock cycles.

Assertion assert_valid_issued_id_ok will report failure when an id is issued and the same id is in circulation with the count exceeding *max_out_per_id*.

Assertion assert_valid_ret_id_ok will report failure if an id is returned while there is no such id value still in circulation (i.e., this id has not been issued).

Assertion assert_valid_max_issued_ids_ok will report failure when an id is issued and the total number of different id's in circulation is already equal to the *max_ids* or the *max_out_ids* value.

assertion assert_nb_ret_per_issued_0 will report failure when nb_ret_per_issued is not greater than 0 at the time issued_sig is asserted.

Coverage modes (Only available with version = 1 selection)

`cov_level_1_0( Bit 0 set in coverage_level_1)`
Cover property, `cover_number_of_issued_sig`, indicates that *`issued_sig`* was asserted regardless the value of *`issued_id`*.

`cov_level_1_1( Bit 1 set in coverage_level_1)`
Cover property, `cover_number_of_ret_sig`, indicates that *`ret_sig`* was asserted regardless the value of *`ret_id`*.

`cov_level_1_2( Bit 2 set in coverage_level_1)`
Cover property, `cover_number_of_reset_sig`, indicates that *`reset_sig`* was asserted regardless the value of *`reset_id`*. z

`cov_level_2_0( Bit 0 set in coverage_level_2)`
Cover point, `observed_latency`, reports the *`issued_sig`* to *`ret_sig`* latencies that were observed at least once.

`cov_level_2_1( Bit 1 set in coverage_level_2)`
Cover point, `outstanding_ids`, reports the numbers of outstanding ids that occurred at least once.

`cov_level_3_0( Bit 0 set in coverage_level_3)`
Cover property, `cover_ret_sig_with_exact_min_lat`, indicates that the observed latency from *`issued_sig`* to *`ret_sig`* was exactly equal to the specified *`min_lat`* value, regardless the id value.

`cov_level_3_1( Bit 1 set in coverage_level_3)`
Cover property, `cover_ret_sig_with_exact_max_lat`, indicates that the observed latency from *`issued_sig`* to *`ret_sig`* was exactly equal to the specified *`max_lat`* value, regardless of the id value.

`cov_level_3_2( Bit 2 set in coverage_level_3)`
Cover property, `cover_issued_ids_reached_max_ids`, indicates that the number of outstanding id's was exactly equal to the specified min of *`max_ids`* and *`max_out_ids`* values.

`cov_level_3_3( Bit 3 set in coverage_level_3)`
Cover property, `cover_issued_ids_reached_max_out_per_id`, indicates that the number of outstanding *`issued_id's`* was exactly equal to the specified *`max_out_per_id`* value.

Example

```
assert_valid_id #(0, 3, 6, 6, 1, 1, 10, 0,
                  "latency 1 to 10, 6 out of 6 id's, "1 copy each")
      val_id_inst (clk, !reset,
                   issued_valid, issued_id,
                    ret_valid, ret_id,
                    clear, clear_id);
```

The checker is enabled when out of reset. All signals are sampled on posedge *`clk`*. There are 3 bits to encode id's. At most 6 id's can be in circulation at any time and that 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, similarly, a returned id is valid when *`ret_valid`* is asserted. An issued id must be returned in 1 to 10 clock cycles. A custom message is appended to the standard failure message. The coverage Level 1 is enabled by default.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_valid_id #(.severity_level(0),.id_bw(3),
 .max_ids(6),.max_out_ids(6),.max_out_per_id(1),
 .min_lat(1),.max_lat(10),.edge_expr(0),
 ,.msg("1 copy each"))
val_id_inst (.clk(clk),.reset_n(!reset),
 .trig_req(issued_valid),.follow_req(ret_id),
 .follow_resp(clear),.trig_resp(clear_id));
```

assert_value

Ensure that *exp* can only be one of the specified values in *vals*.

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);
```

Arguments

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

disallow
: If *disallow* evaluates true, then *exp* is forbidden to take on any of the specified value in *vals*. Otherwise, the check ensures *exp* takes on only the specified value in *vals*. Default = 0.

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

exp
: Signal or expression being tested.

vals
: A bitvector of concatenated values that the signal being tested (exp) must evaluate to.

  For example, if *bw* = 3, *no_vals* = 2, and the values are 3'b000 and 3'b110 then the value bound to the *vals* port should be 6'b000_110 .

Assumptions

no_vals > 0 (and the *vals* bitvector should have the correct size.)

Failure Modes

When *disallow* = 0, the assertion assert_value will report failure when *exp* was not one of the expected values in *vals* at the time when reset_n = 1.

When *disallow* = 1, the failure of the assertion assert_value means that at the time when *reset_n* = 1, the value of *exp* was one of the values in *vals* (which was disallowed).

Coverage Modes

cov_level_1_0 (bit 0 set in coverage_level_1)

Cover property, value[i].cover_nb_of_exp_changes, indicates that the *exp* changed value, when enabled by *reset_n*.

For disallow = 0 :

cov_level_1_1 (bit 1 set in coverage_level_1)

Cover property, cover_value, indicates that the *exp* changed to one of the values specified in *vals*.

cov_level_3_0 (bit 0 set in coverage_level_3)

Cover property, value[i].cover_nb_of_transitions_to_each_vals, indicates that the *exp* changed to the i (the value specified).

For disallow = 1 :

cov_level_1_1 (bit 1 set in coverage_level_1) :

Cover property, `cover_value,` indicates that the *`exp`* changed to any value different from the values specified in *`vals`*. There is no Level 3 coverage in this case.

Example

```
assert_value #(0, 4, 0, 8, 0, "value error", 0, 1, 0, 1)
       value_instance (clk, load, control_reg, 16h0_1_3_5);
```

When *`load`* is 1, then at posedge *`clk`* (default) the 4-bit variable *`control_reg`* must have one of the four values 0, 1, 3 or 5 (since *`disallow`* is 0). Coverage Levels 1 and 3 are enabled.

Name-based Example

For greater clarity, the checker in the example can be written in the name-based format as follows:

```
assert_value #(.severity_level(0),.no_vals(4),
 .disallow(0),.bw(8),.edge_expr(0),.msg("value error"),
 .category(0), .coverage_level_1(1),
 .coverage_level_2(0), .coverage_level_3(1))
value_instance (.clk(clk),.reset_n(load),
 .exp(control_reg),.vals(16h0_1_3_5));
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z