accellera

Open Verification Library

Assertion Monitor Reference Manual

v 02.10.16

This page intentionally left blank.

Contents

CHAPTER 1

Introduction

The verification community has recently seen the emergence of a plethora of proprietary languages and interfaces to design verification tools. Most of these languages and interfaces are not natural to the designer's verification flows, requiring designers to master a large number of verification techniques and interfaces. In addition, these multiple interfaces make it harder for designers to evaluate new verification techniques and tools.

The Accellera Open Verification Library (OVL) provides designers, integrators, and verification engineers with a single and vendor-independent interface for design validation using simulation, semi-formal verification, and formal verification techniques. By using a single well-defined interface, the OVL bridges the gap between the different types of verification, making more advanced verification tools and techniques available for non-expert users.

The OVL is composed of a set of assertion monitors that verify specific properties of a design. These assertion monitors are instantiated in the design establishing a single interface for design validation. Vendors are expected to provide tool-specific libraries compliant with the terms of the OVL standard.

The OVL is based on the principles stated in the book *Principles of Verifiable RTL Design* by Lionel Bening and Harry Foster, published by Kluwer Academic Publishers.

This document provides the reader with the Verilog and VHDL definitions of the version of the assertion library published by the OVL, as well as examples (to embed these assertion monitors into a design), syntax, and usage patterns.

Who should read this document?

It is assumed the reader is familiar with hardware description languages and conventional simulation environments.

This document targets designers, integrators, and verification engineers who intend to use the assertion library in their verification flow, and to tool developers interested in integrating the assertion library in their products.

What are assertion monitors?

Assertion monitors are instances of modules whose purpose in the design is to guarantee that some conditions hold true. Assertion monitors are modeled after VHDL *assertions*: they are composed of an *event*, *message,* and *severity*.

- *Event* is a property that is being verified by an assertion.

  An *event* can be classified as a temporal or static property. (A static property is a property that must be valid at all times, whereas a temporal property is a property that is valid during certain times.)
- *Message* is the string that is displayed in the case of an assertion failure.
- A *severity* represents whether the error captured by the assertion library is a major or minor problem.

Benefits of using assertion monitors

The benefits of using assertion libraries have been explained in several papers listed at the end of this chapter, and can be found in the book *Principles of Verifiable RTL Design*. Notably, assertion monitors benefit users by:

- testing internal points of the design, thus increasing observability of the design
- simplifying the diagnosis and detection of bugs by constraining the occurrence of a bug to the assertion monitor being checked
- allowing designers to use the same assertions for both simulation and formal verification.

Implementing assertion monitors

Assertion monitors address design verification concerns and can be used as follows to increase design confidence:

- Combine assertion monitors to increase the coverage of the design (for example, in interface circuits and corner cases).
- Include assertion monitors when a module has an external interface. In this case, assumptions on the correct input and output behavior should be guarded and verified.
- Include assertion monitors when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores), or may not completely understand the module. In these cases, guarding the module with assertion monitors may prevent incorrect use of the module.

Usually there is a specific assertion monitor suited to cover a potential problem. In other cases, even though a specific assertion monitor may not exist, a combination of two or three assertion monitors can provide the desired coverage.

The number of actual assertions that must be added to a specific design may vary from a few to hundreds, depending on the complexity of the design and the complexity of the properties that must be checked.

Writing assertion monitors for a given design requires careful analysis and planning for maximum efficiency. While writing too few assertions may not increase the coverage on a design, writing too many assertions may increase verification time, sometimes without increasing the coverage. In most cases, however, the runtime penalty incurred by adding assertion monitors is relatively small.

Triggering assertion monitors

An assertion monitor is triggered when an error condition occurs—usually, in the following cycle. However, when the test_expr is not synchronized with the assertion clock clk, either a non-deterministic delay or false assertion triggering may occur. To avoid this consequence, always line up test_expr with the assertion monitor sampling clock clk. Non-deterministic triggering delay refers to the delay between the time the error condition occurs and the time it is detected. False triggering can occur with more complex assertions if the test_expr and assertion clock clk are not synchronized.

Conventions used in this document

Throughout this document, the following conventions are used:

Bold case	indicates the valid syntax of an assertion monitor.
`[ ]`	indicates optional arguments to the assertion monitor.
Italics	indicates variable names in the assertion monitor definition.
`Courier`	indicates Verilog® or VHDL code formatted for this manual. Note that in some cases minor editing may be required for an assertion monitor, especially for the messages.

References

The following is a list of resources related to design verification and assertion monitors.

1. Bening, L. and Foster, H., “Principles of Verifiable RTL Design, a Functional Coding Style Supporting Verification Processes in Verilog,” 2nd ed., Kluwer Academic Publishers, 2001.
2. Bergeron, J., “Writing Testbenches: Functional Verification of HDL Models,” Kluwer Academic Publishers, 2000.
3. Foster, H. and Coelho, C., “Assertions Targeting a Diverse Set of Tools,” International HDL Conference, 2001.
4. Taylor, S., Quinn, M., Brown, D., Dohm, N., Hildebrandt, J., Huggins, J., Ramey, C., Ramey, J., “Functional Verification of a Multiple-issue, Out-of-order, Superscalar Alpha Processor - The DEC Alpha 21264 Microprocessor,” 35th. Design Automation Conference, 1998.
5. Kantrowitz, M. and Noack, L. M., “I’m Done Simulating: Now What? Verification Coverage Analysis and Correctness of the DECchip 21164 Alpha microprocessor,” 33rd. Design Automation Conference, 1996.

This page intentionally left blank.

CHAPTER 2

Verifying a Design Using Assertion Monitors

This chapter, provides a simple example of a Verilog design for which assertion monitors were added to enable better detection and diagnosis of design bugs.

Traffic Light Controller

This simple design is adapted from the Mead and Conway traffic light controller example. The original file can be found in the VIS distribution from the University of California, Berkeley. It was edited by Tom Shipple.

The design consists of a traffic light controller between two roads, a farm road and a multi-lane highway. The highway should have right of way, and as a result, the controller must maximize the time the green light remains on.   This controller has a timer with two outputs, *short* and *long* signals. The green light must be at the green state for at least *long* time units. At the end of this period, if there is a car waiting at the farm intersection, the light at the highway turns yellow for a *short* period and then red. The light remains red for the highway until all cars have crossed the highway or until a *long* timer has expired. The description for this design in Verilog is presented below.

```
`define YES      1
`define NO       0

`define START    0
`define SHORT    1
`define LONG     2

`define GREEN    2'd0
`define YELLOW   2'd1
`define RED      2'd2

`define TIMER_WIDTH     5

module main(clk, reset_n, car_present, long_timer_value,
   short_timer_value, farm_light, hwy_light);

input clk, reset_n, car_present;
input [`TIMER_WIDTH-1:0] long_timer_value,
   short_timer_value;
output [1:0] farm_light, hwy_light;

wire start_timer, short_timer, long_timer;
wire enable_farm, farm_start_timer, enable_hwy,
   hwy_start_timer;

assign start_timer = farm_start_timer || hwy_start_timer;

timer timer(clk, reset_n, start_timer, short_timer,
   long_timer, long_timer_value, short_timer_value);

farm_control farm_control(clk, reset_n, car_present,
   enable_farm, short_timer, long_timer, farm_light,
   farm_start_timer, enable_hwy);

hwy_control hwy_control (clk, reset_n, car_present,
   enable_hwy, short_timer, long_timer, hwy_light,
   hwy_start_timer, enable_farm);

endmodule

/*
 * From the START state, the timer produces the signal
 * "short" after a non-deterministic amount of time. The signal
 * "short" remains asserted until the timer is reset (via the
 * signal "start"). From the SHORT state, the timer produces
 * the signal "long" after a non-deterministic amount of time.
 * The signal "long" remains asserted until the timer is reset
 * (via the signal "start").
 */

module timer(clk, reset_n, start, short, long, long_timer_value,
short_timer_value);
input clk, reset_n, start;
output short, long;
input [`TIMER_WIDTH-1:0] long_timer_value, short_timer_value;

reg [1:0] state;
reg [`TIMER_WIDTH-1:0] timer;

initial state = `START;

assign short = ((state == `SHORT) || (state == `LONG));
assign long  = (state == `LONG);

always @(posedge clk)
begin
  if (start || reset_n == 1'b0) begin
    timer <= 0;
    state <= `START;
  end
  else begin
    case (state)
    `START:
      begin
        timer = timer + 1;
        if (timer >= short_timer_value) state <= `SHORT;
      end
    `SHORT:
      begin
        timer = timer + 1;
        if (timer >= long_timer_value) state <= `LONG;
      end
    endcase
  end
end
endmodule

/*
 * Farm light stays RED until it is enabled by the highway
 * control. At this point, it resets the timer, and moves to
 * GREEN. It stays in GREEN until there are no cars, or the
 * long timer expires. At this point, it moves to YELLOW and
 * resets the timer. It stays in YELLOW until the short
 * timer expires. At this point, it moves to RED and enables
 * the highway controller.
 */

module farm_control(clk, reset_n, car_present, enable_farm,
     short_timer, long_timer, farm_light, farm_start_timer,
     enable_hwy);
input clk, reset_n, car_present, enable_farm, short_timer,
      long_timer;
output farm_light, farm_start_timer, enable_hwy;

reg [1:0] farm_light;

initial farm_light = `RED;

assign farm_start_timer = (((farm_light == `GREEN) &&
  ((car_present == `NO) || long_timer)) ||
   (farm_light == `RED) && enable_farm);

assign enable_hwy = ((farm_light == `YELLOW) && short_timer);

always @(posedge clk) begin
  if (reset_n == 1'b0) begin
    farm_light <= `RED;
  end
  else begin
    case (farm_light)
    `GREEN:
      if ((car_present == `NO) || long_timer)
        farm_light <= `YELLOW;
    `YELLOW:
      if (short_timer) farm_light <= `RED;
    `RED:
      if (enable_farm) farm_light <= `GREEN;
    endcase
  end
end
endmodule

/*
```

```
 * Highway light stays RED until it is enabled by the farm
 * control. At this point, it resets the timer, and moves to
 * GREEN. It stays in GREEN until there are cars and the
 * long timer expires. At this point, it moves to YELLOW and
 * resets the timer. It stays in YELLOW until the short
 * timer expires. At this point, it moves to RED and enables
 * the farm controller.
 */
module hwy_control(clk, reset_n, car_present, enable_hwy,
  short_timer, long_timer, hwy_light, hwy_start_timer,
  enable_farm);
input clk, reset_n, car_present, enable_hwy, short_timer,
      long_timer;
output hwy_light, hwy_start_timer, enable_farm;

reg [1:0] hwy_light;

initial hwy_light = `GREEN;

assign hwy_start_timer =
  (((hwy_light == `GREEN) && ((car_present  == `YES) &&
     long_timer)) || (hwy_light == `RED) && enable_hwy);

assign enable_farm = ((hwy_light == `YELLOW) &&
                      short_timer);

always @(posedge clk) begin
  if (reset_n == 1'b0) begin
    hwy_light <= `GREEN;
  end
  else begin
    case (hwy_light)
    `GREEN:
      if ((car_present == `YES) && long_timer)
        hwy_light <= `YELLOW;
    `YELLOW:
      if (short_timer) hwy_light <= `RED;
    `RED:
      if (enable_hwy) hwy_light <= `GREEN;
    endcase
  end
end
endmodule
```

To verify this circuit, first the designer must define the properties that should be verified. The following conditions must be satisfied for this circuit to work correctly:

- at any time, the traffic light on both sides should never be green
- the short_timer_value should be less than the long_timer_value
- if there are no more cars on the farm road and the traffic light is green for the farm road, then the traffic light should switch to yellow, then red
- it is not possible for a car on the farm road to wait forever

There are probably several other conditions that could be extracted from this design, to increase the designer's confidence in the design's correctness. For the sake of this example, however, this set should suffice.

The next step is to create an assertion that captures the conditions elaborated above.

The traffic light on both sides should never be green at the same time

The variables representing the light on both sides are *hwy_light* and *farm_light*. By looking at the list of assertions from the assertion library, it can be seen that this condition is best encapsulated by the assertion *assert_never*. The resulting assertion can be seen below.

```
assert_never both_are_green (clk, reset_n,
  (farm_light == ‘GREEN && hwy_light == ‘GREEN));
```

The short timer value should be smaller than the longer timer value

This assertion makes an assumption about the conditions for this module’s use. It ensures that in every instantiation of this module, this condition between two inputs should be observed.

```
assert_always short_is_smaller_longer (clk, reset_n,
  (short_timer_value < long_timer_value));
```

If there are no more cars on the farm road when the farm light is green, the light should switch to yellow

This condition asserts that the traffic light controller should always maximize the green time for the highway.

```
assert_time #(0,1) change_from_green_if_no_car
  (clk, reset_n, (farm_light == ‘GREEN && ~car_present),
   (farm_light == ‘YELLOW));
```

It is not possible for a car on the farm road to wait forever

In this assertion, the internal timer counts up to 32, allowing enough time for cars on the major road to pass, thus, *long_timer* is asserted. At this point, the farm road is checked for the presence of a car. The maximum length of the farm road’s red light is then 32 cycles. The assertion is presented below.

```
assert_change #(0,2,32) car_should_not_wait_forever
  (clk, reset_n,
   long_timer && hwy_light == ‘GREEN && car_present,
   hwy_light);
```

This chapter briefly presented specifying assertion monitors to a design to capture the design’s intended behavior and assumptions. Assertion monitors are defined in the next chapter.

This page intentionally left blank.

CHAPTER 3

Structure and Use of Verilog and VHDL Assertion Monitors

This chapter describes the differences in functionality between the Verilog version of the OVL and the VHDL version. Furthermore, it describes the formats of the two versions and highlights version-specific features. The chapter closes with a list of the supported assertions and parameters, which are available in both the Verilog and VHDL versions.

Functionality: Verilog vs. VHDL

TBD

While the OVL is available in both Verilog and VHDL versions, functionality differs for these two versions as a result of inherent capabilities. For example, the Verilog version includes preprocessor text macros that have no counterpart in the VHDL version, since VHDL does not support preprocessing.

Verilog Version

Verilog Format

All Verilog assertion monitors defined by the Open Verification Library initiative observe the following BNF format, defined in compliance with Verilog Module instantiation of the IEEE Std 1364-1995 "Verilog Hardware Description Language".

```
assertion_instantiation ::= assert_identifier
  [parameter_value_assignment] module_instance ;

parameter_value_assignment ::= #(severity_level {,other parameter
  expressions}, options, msg)

module_instance ::= name_of_instance ([list_of_module_connections])

name_of_instance ::= module_instance_identifier

list_of_module_connections ::=
  ordered_port_connection {, ordered_port_connection}
  | named_port_connection {, named_port_connection}

ordered_port_connection ::= [expression]

named_port_connection ::= .port_identifier ([expression])
assert_identifier ::= assert_[type_identifier]

type_identifier ::= identifier
```

Verilog Macro Global Variables

If you are using the Verilog version of the Assertion Monitor Library, please note that it currently includes four Verilog Macro Global Variables:

- ***‘ASSERT_GLOBAL_RESET***
- ***‘ASSERT_MAX_REPORT_ERROR***
- ***‘ASSERT_ON***
- ***‘ASSERT_INIT_MSG***

These four variables are described briefly in the table below and in greater detail in the following paragraphs.

Variable	**Definition**
ASSERT_GLOBAL_RESET	Overrides individual *reset_n* signals
ASSERT_MAX_REPORT_ERROR	Defines the number of errors required to trigger a report
ASSERT_ON	Enables assertion monitors during verification
ASSERT_INIT_MSG	Prints a report that lists the assertions present in a given simulation environment.

The `list_of_module_connections` has a required parameter, ***reset_n***. The signal *reset_n* is an active low signal that indicates to the assertion monitor when the initialization of the circuit being monitored is complete. During the time when *reset_n* is low, the assertion monitor is disabled and initialized. Alternatively, to specifying a reset_n signal or condition for each assertion monitor, you may specify the global macro variable ***‘ASSERT_GLOBAL_RESET***. If this variable is defined, all instantiated monitors will disregard their respective *reset_n* signals. Instead, they will be initialized whenever *‘ASSERT_GLOBAL_RESET* is low.

Every assertion monitor maintains an internal register *error_count* that stores the number of times the assertion monitor instance fires. This internal register can be accessed by the testbench to signal when a given testbench should be aborted. When the global macro variable ***‘ASSERT_MAX_REPORT_ERROR*** is defined, the assertion instance stops reporting messages if the number of errors for that instance is greater than the value defined by the ***‘ASSERT_MAX_REPORT_ERROR*** macro.

To enable the assertion monitors during verification, you must define the macro ***‘ASSERT_ON*** (for example, *+define+ASSERT_ON*). During synthesis, the *ASSERT_ON* would not be defined. In addition, *//synthesis translate_off* meta-comments are contained within the body of each monitor to prevent accidental synthesis of the monitor logic.

The current release of the library includes the (*options*) parameter for each assertion monitor. This parameter allows future added features to the library without the need to modify the interface. For example, an assertion monitor can now be defined as a constraint to formal tools by setting the *options* parameter to a value of "1".

When you define the ***‘ASSERT_INIT_MSG*** macro, an “initial” block calls a task to report the instantiation of the assertion. This macro is useful for identifying each of the assertions present in a given simulation environment.

Please note: Most assertions are triggered at the positive edge of a triggering signal or expression *clk.* The assertion *assert_proposition* is an exception, it monitors an expression at all times.

Verilog Assertion Monitor Messages

The OVL library includes a file named *ovl task.h* that contains a set of tasks that allow you to customize the following:

- simulation startup identification of assertions
- error message reporting mechanism
- actions associated with assertion firing (for example, $finish)

You may use this file to name your own PLI user-defined task upon triggering an assertion, as an alternative to the default $display reporting mechanism currently built into the OVL. To take advantage of this feature, edit the *ovl task.h* file (shown below) to reflect your preferences.

```
task ovl_error;
    input [8*63:0] err_msg;
  begin
    error_count = error_count + 1;
    `ifdef ASSERT_MAX_REPORT_ERROR
      if (error_count <= `ASSERT_MAX_REPORT_ERROR)
    `endif
        $display(
          "OVL_ERROR : %s : %s : %0s : severity %0d : time %0t : %m",
          assert_name, msg, err_msg, severity_level, $time);
      if (severity_level == 0) ovl_finish;
    end
endtask

task ovl_finish;
  begin
    #100 $finish;
  end
endtask

task ovl_init_msg;
  begin
    $display(
       "OVL_NOTE: %s initialized @ %m Severity: %0d, Message: %s",
               assert_name,severity_level, msg);
  end
endtask
```

VHDL Version

VHDL Format

All VHDL assertion monitors defined by the Open Verification Library are based on open public OVL Verilog simulation models and defined in compliance with IEEE-1076. The VHDL version of OVL is implemented as two VHDL files: assert.vhd and assert_pkg.vhd.

assert.vhd

The first of the two VHDL files used to implement the OVL contains ENTITY/ARCHITECTURE declarations. Each assertion model is implemented as an ENTITY/ARCHITECTURE pair.

assert_pkg. vhd

The second VHDL file contains one package header and one corresponding package body: `ovl_assertlib`. The following items are declared in this file:

Component declarations: Each assertion model includes one COMPONENT declaration.

Global signals: Each assertion model includes one global reset signal, one global reset enable signal, and one end of simulation signal.

- The global reset signal is as follows:

```
SIGNAL ovl_reset_n: std_ulogic := ’1’;
```

- The global resent enable signal is as follows:

```
SIGNAL ovl_reset_n_enable: std_ulogic := ’0’;
```

- The end of simulation signal is as follows:

```
SIGNAL ovl_END_OF_SIMULATION_SIGNAL: std_ulogic := ’0’;
```

Supporting functions: Each assertion model includes two supporting functions.

- The first is a reduction XOR function:

```
FUNCTION xorr ( V: unsigned) return std_ulogic:
```

- The second function converts TRUE to "1" and FALSE to "0":

```
FUNCTION to_std ( V: boolean) return std_ulogic;
```

WORK VHDL Library

To use OVL assertions in VHDL designs, compile the `assert.vhd` and `assert_pkg.vhd` VHDL files to the WORK VHDL library as follows:

1. Add the `USE WORK.ovl_assertlib.all;` statement before instantiating OVL assertion components in the VHDL design.
2. Instantiate OVL assertion components.

VHDL Example

The following is a complete OVL VHDL example. Abbreviated VHDL examples for each OVL assertion appear in Chapter 4, “Assertion Monitor Library” on page 17.

```
--
-- file test.vhd
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE WORK.ovl_assertlib.all; -- so we can use OVL components
--
ENTITY test IS PORT (
   reset : IN std_logic;
   clk : IN std_logic);
END ENTITY;
```

```
--
ARCHITECTURE behavioral OF test IS
  SIGNAL count : UNSIGNED (3 downto 0) := "0000";
  SIGNAL count_le_8, count_gt_9 : std_logic;
BEGIN
  count_le_8 <= ’1’ when (count <= 8) else ’0’;
  count_gt_9 <= ’1’ when (count >  9) else ’0’;
  PROCESS (clk)
  BEGIN
    IF (rising_edge (clk)) THEN
       IF (reset = ’0’ OR (count >= 9)) THEN
           count <= "0000";
       ELSE
           count <= count + 1;
       END IF;
    END IF;
  END PROCESS;

  er1: assert_never
         GENERIC MAP (1, 0, "ASSERT : Sorry, my fault!")
         PORT MAP (clk, ’1’, count_le_8);
  ok1: assert_never
         GENERIC MAP (1, 0, "ASSERT : Sorry, your fault!")
         PORT MAP (clk, ’1’, count_gt_9);
END ARCHITECTURE;
```

Supported Assertions and Parameters

The OVL library supports the following assertions. Please visit the Open Verification Library web page (located at http://www.verificationlib.org/) for the most up-to-date assertions.

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

The following three parameters are present in every assertion library definition: ***severity_level***, ***options***, and ***message***.

- ***severity_level*** is an optional parameter that is used to describe the severity of a failure. By default, this parameter is set to 0, the highest severity. The way a specific tool deals with this parameter is tool dependent. In simulation, if an error is encountered with a severity_level of 0, the simulation should halt. Higher numbered severity levels are displayed as warnings, however, simulation will continue to run.
- ***options*** is a 32 bit integer optional parameter that is used to describe characteristics of the assertion monitor to various EDA tools. Currently, the only option supported is options=1, which is used to identify the assertion monitor as a constraint to formal verification tools. The default option is 0, or no option specified.
- ***msg*** is an optional parameter that is used to describe the error message that should be printed when the assertion fires.

CHAPTER 4

Assertion Monitor Library

This chapter includes an entry for each assertion currently supported by the assertion monitor library. Each entry consists of the following:

- Overview - A general introduction to the assertion, including an overview of its attributes.
- Syntax - The valid syntax, including optional and required arguments. (Refer to Chapter 1: Introduction, "Conventions used in this document" on page 3.) Each of the options and variables is defined in a table, which follows the syntax.
- Usage - Fundamental information that will help you determine when to use the assertion.
- Verilog Example and VHDL Example - Verilog and VHDL modules that contain assertion monitors. Many of the examples are self-explanatory. However, in some cases, additional explanation is offered.

Please see Appendix A, "Assertion Library Definitions for Verilog" on page 83 and Appendix B, "Assertion Library Definitions for VHDL" on page 147 for assertion-specific library definitions.

assert_always

Overview The **assert_always** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will always evaluate TRUE. If *test_expr* evaluates to FALSE, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_always [#(severity_level, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_always** assertion belongs to the most general class of assertions, which includes “assert_implication” (see page 40), “assert_never” (see page 44) and “assert_proposition” (see page 60). Assert_always does not contain any complex sequential check other than sampling *test_expr* at every positive edge of *clk*. It should be used whenever you want to verify a propositional property that should always hold TRUE at clock boundaries or at the positive edge of *clk*.

Verilog Example

```
module counter_0_to_9(reset_n,clk);
input reset_n, clk;

reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count >= 9) count = 1’b0;
  else count = count + 1;
end

assert_always #(0, 0, "error: count not within 0 and 9")
  valid_count (clk, reset_n, (count >= 4’b0000) &&
                (count <= 4’b1001));
endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY counter_0_to_9 IS
. . .
end counter_0_to_9;

ARCHITECTURE ovl1 OF counter_0_to_9 IS
```

```
  . . .
BEGIN
  . . .
  ok1: assert_always GENERIC MAP (1) PORT MAP
   (clk => clk,
   reset_n => resetn,
   test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_always_on_edge

Overview The **assert_always_on_edge** assertion continuously monitors the *test_expr* at every specified edge of the *sampling_event* and positive edge of clock *clk*. It contends that a specified *test_expr* will always evaluate TRUE on the edge of a *sampling_event*. If *test_expr* evaluates to FALSE, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_always_on_edge [#(severity_level, edge_type, options, msg)] inst_name (clk, reset_n, sampling_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
edge_type	Selects the transition for *sampling_event.* 0 - no edge (default) 1 - positive edge 2 - negative edge 3 - any edge
options	Vendor options. Currently, the only supported option is *option*s=1, which defines the assertion as a constraint on formal tools.
msg	Error message that will be printed if the assertion fires.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
sampling_event	Expression monitored defining when to evaluate *test_expr.* Transition of *sampling_event* must match transition selected by *edge_type.*
test_expr	Expression being verified at the positive edge of *cl*k AND *sampling_event* matches transition selected by *edge_type*.

Usage The **assert_always_on_edge** assertion is a variant of the **assert_always** assertion, where the ability to qualify the assertion with a transition of the *sampling_event* is provided. This assertion is useful when events are identified by their transition and not just the logical state.

Verilog Example

```
assert_always_on_edge #(0, 0, 1, “Error: New request when FSM is not ready”) i1
                                        (clk, reset_n, request, (state == IDLE)) ;
```

In the example above, the assert_always_on_edge assertion insures that for every new request (identified by the rising edge of *request*, that the FSM that handles the request is ready for the request, (that is, in the IDLE state). If a request arrives when the FSM is not ready, the assertion will fire.

Below is an example verilog module that uses the assert_always_on_edge assertion to insure that the rate of new requests is not beyond the rate that the FSM can handle. The example stimulus has the first two requests handled correctly. The third request occurs prior to the FSMs completion of the 2nd request and the ovl assertion then fires.

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY counter_0_to_9 IS
   . . .
end counter_0_to_9;

ARCHITECTURE ovl1 OF counter_0_to_9 IS
  . . .
BEGIN

. . .

  ok1: assert_always_on_edge GENERIC MAP (1, 1) PORT MAP
    (clk => clk,
     reset_n => resetn,
     sampling_event => sample,
     test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;

   END PROCESS;

END ovl1;
```

assert_change

Overview The **assert_change** assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When the *start_event* evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values on a clock edge at some point within the next *num_cks* number of clocks. As soon as the *test_expr* evaluates TRUE, this assertion is satisfied; and checking is discontinued for the remaining *num_cks*.

Syntax ***assert_change [#(severity_level, width, num_cks, flag, options, msg)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
num_cks	The number of clocks for *test_expr* to change its value before an error is triggered 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*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
start_event	Starting event that triggers monitoring of the *test_expr*.
test_expr	Expression or variable being verified at the positive edge of *clk*.

Usage The **assert_change** assertion should be used in circuits to ensure that after a specified initial event; a particular variable or expression will change. Common uses for **assert_change** include:

- verification that synchronization circuits respond after a specified initial stimuli. For example, in protocol verification, this assertion may be used to check that after a *request* an *acknowledge* will occur within a specified number of cycles.
- verification that finite-state machines (FSM) change state, or will go to a specific state, after a specified initial stimuli.

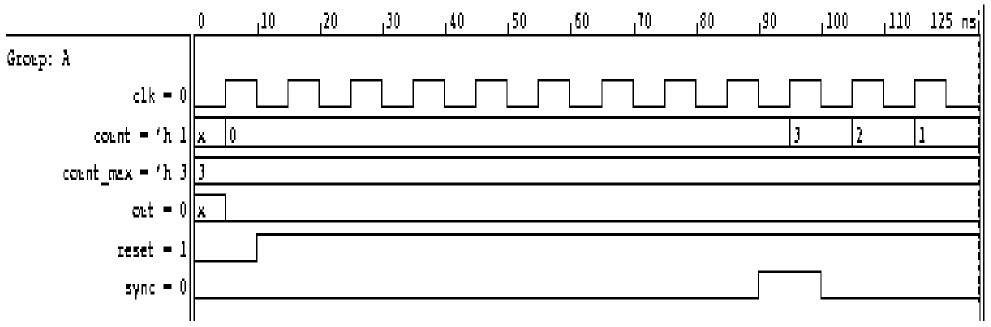
Verilog Example In this example, the module *synchronizer_with_bug* is designed to respond by asserting *out* after *count_max* cycles *sync* was asserted. Note in the accompanying figure, however, that *out* is not asserted until after the trigger of *clk*. In this figure, the waveform is shown until the error is triggered by the assertion library. At this point, the simulation is aborted.

```
module synchronizer_with_bug (clk, reset_n, sync,
  count_max, out);

  input clk, reset_n, sync;
  input [3:0] count_max;
  output out;
  reg out;
  reg [3:0] count;

  always @(posedge clk) begin
    if (reset_n == 0) begin
      out <= 0;
      count <= 0;
    end
    else if (count != 0) begin
      count <= count - 1;
      if (count == 1) out <= 1;
    end
    else if (sync == 1) count <= count_max;
    else if (out == 1) out <= 0;
  end

  assert_change #(0,1,3,0) synch_test (clk,reset_n,
      (sync == 1), out);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY synchronizer IS
   . . .
end synchronizer;

ARCHITECTURE ovl1 OF synchronizer IS
  . . .
BEGIN
  test_expr <= to_unsigned(count_out);
  ok1: assert_change GENERIC MAP (1,1,3,0) PORT MAP
                 (clk => clk,
                  reset_n => resetn,
                  start_event => sync,
                  test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' AND count < 9 THEN
           count <= 0 ;
        ELSIF (count /= 0) THEN
            count <= count - 1;
        ELSIF (count = 1) THEN
             count_out <= '1';
        ELSIF (sync = '1') THEN
           count <= to_int(count_max);
        ELSIF (count_out = '1') THEN
           count_out <= '0';
        END IF;
   END PROCESS;
END ovl1;
```


0
10
20
30
40
50
60
70
80
90
100
110
125 ns
Group: A
clk = 0
count = 'h 1
count_max = 'h 3
out = 0
reset = 1
sync = 0
x
0
3
2
1
3
x

assert_cycle_sequence

Overview The **assert_cycle_sequence** assertion checks the following:

- If *necessary_condition* is 0:
  This assertion checks to ensure that if all *num_cks-1* first events of *event_sequence* are true, then the last one (`event_sequence[0]`) must occur.
- If *necessary_condition* is 1:
  This assertion checks to ensure that once the first event (`event_sequence[num_cks-1]`) occurs, all the remaining events occur.

Syntax ***assert_cycle_sequence [#(severity_level, num_cks, necessary_condition, options, msg)] inst_name (clk, reset_n, event_sequence);***

severity_level	Severity of the failure with default value of 0.
num_cks	The width of the *event_sequence* (length of number of clock cycles in the sequence) that must be valid. Otherwise, the assertion will fire; that is, an error occurs.
necessary_condition	Either 1 or 0. The default is 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
event_sequence	A Verilog or VHDL concatenation expression, where each bit represents an event.

Usage The **assert_cycle_sequence** assertion should be used in circuits to ensure a proper sequence of events. An event is a Verilog or VHDL expression (depending on the library you are using) which evaluates TRUE. Common uses for **assert_cycle_sequence** are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification of single cycle operations to operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Verilog Example The following examples asserts that when write cycle starts, followed by one wait statement, then the next opcode will have the value `‘DONE`.

```
assert_cycle_sequence #(0,3)  init_test (clk, reset_n,
        {r_opcode == ‘WRITE, r_opcode == ‘WAIT,
         r_opcode == ‘DONE});
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY cycle_seq IS
   . . .
```

```
end cycle_seq;

ARCHITECTURE ovl1 OF cycle_seq IS
  . . .
BEGIN
  test_expr <= to_unsigned((count = 2#0010#) & (count =   2#0011#) & (count =
2#0100#));
  ok1: assert_cycle_sequence GENERIC MAP (0, 3) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    event_sequence => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_decrement

Overview

The **assert_decrement** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will never decrease by anything other than *value*. The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). The check will not start until the first clock after *reset_n* is asserted.

Note: It is also permissible for *test_expr* to remain the same, but if it does decrease, it will never decrease by anything other than *value*.

This checker is targeted for circular queue structures with an address counter that is permitted to wrap. That is, this checker permits the *test_expr* expression to wrap, as long as it decreases by the specified *value*. In the following example, the first clock is `4'b0000` and the second clock is `4'b1111`. We can see that this *test_expr* has decreased by a value of one (that is, the *test_expr* is wrapped). Hence, if the *value* parameter was set to 1, this is valid.

Example:

`4'b0000 → 4'b1111`

Special consideration: Do not use this assertion for general counters that can both increment and decrement.

Syntax

assert_decrement [#(severity_level, width, value, options, msg)] inst_name (clk, reset_n, test_expr);

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
value	The decrement value allowed for *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage

The **assert_decrement** assertion should be used in circuits to ensure the proper change of values in structures such as counters and finite-state machines (FSM). However, if the variable or expression being checked is allowed to increment and decrement, then “assert_delta” (see page 29) should be used instead of assert_decrement.

Verilog Example

```
module counter_0_to_9(reset_n,clk,dec);
  input reset_n, clk;
  input [1:0] dec;

  reg [3:0] count;

  always @(posedge clk)
  begin
    if (reset_n == 0) count <= 4'd0;
    else if (count == 0) count <= 4'd9;
    else count <= count - dec;
  end

  assert_decrement #(0,4,1) valid_count (clk, reset_n,
     count);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY counter_0_to_9 IS
   . . .
end counter_0_to_9;

ARCHITECTURE ovl1 OF counter_0_to_9 IS
  . . .
BEGIN
  test_expr <= to_unsigned(count);
  ok1: assert_decrement GENERIC MAP (1,4,1) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' THEN
           count <= 0;
        ELSIF count = 0 THEN
           count <= 9;
        ELSE
           count <= count - 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_delta

Overview The **assert_delta** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will never change values by anything less than *min* value or anything more than *max* value. The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). The check will not start until the first clock after *reset_n* is asserted.

Syntax ***assert_delta [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
min	Minimum changed value allowed for *test_expr* in two consecutive clocks of *clk*. Default value is set to 1.
max	Maximum changed value allowed for *test_expr* in two consecutive clocks of *clk*. Default value is set to 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_delta** assertion should be used in circuits to ensure the proper change of values in control structures such as up/down counters. For these types of circuits, **assert_delta** can be used to check for overflow and underflow.

In datapath circuits, **assert_delta** should be used to check whether there is a “smooth” change of value in a variable. This is especially useful in cases where the variable controls a physical variable that cannot perceive any drastic change from a previous value.

Verilog Example This example shows how **assert_delta** can be used in arithmetic circuits, especially when these circuits must generate a smooth output. In this case, we check that *y* generates a smooth output. In the following example, the resulting consecutive outputs are increasing by a maximum of 8 units.

```
module smooth_test (clk,reset_n,a,b,x,y);
input clk, reset_n;
input [15:0] a,b,x;
output [15:0] y;

reg [15:0] y, xo;

always @(posedge clk)
begin
  if (reset_n == 0) begin
    y <= b;
    xo <= 0;
  end
  else begin
    y  <= y + a * (x - xo);
    xo <= x;
  end
end

assert_delta #(0,16,0,8) valid_smooth (clk, reset_n, y);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY smooth_test IS
   . . .
end smooth_test;

ARCHITECTURE ovl1 OF smooth_test IS
  . . .
BEGIN
  test_expr <= to_unsigned(y);
  ok1: assert_delta GENERIC MAP (1) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    test_expr => test_expr);
  PROCESS
  VARIABLE int_temp2 : INTEGER;
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' THEN
           y <= b;
           x0 <= (others => '0') ;
        ELSE
           y <= to_int(y) + to_int(a) * (to_int(x) - to_int(x0));
           x0 <= x ;
        END IF;
   END PROCESS;
END ovl1;
```

assert_even_parity

Overview The **assert_even_parity** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will always have an even number of bits 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 or VHDL expression (depending on the library you are using).

Syntax ***assert_even_parity [#(severity_level, width, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_even_parity** assertion is most useful for control and datapath circuits. This assertion ensures that a variable or expression has an even number of bits asserted. Some example uses of **assert_even_parity** are:

- address or data busses with error checking based on parity
- finite-state machines (FSM) with error detection mechanisms

Verilog Example

```
module counter_0_to_9_with_parity (reset_n,clk,count);
input reset_n, clk;
output [4:0] count;
reg [4:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count >= 9) count <= 5'b10000;
  else begin
    count[3:0] = count[3:0] + 1;
    count[4] = ^count[3:0];
  end
end

assert_even_parity #(0,5) parity_sanity_check (clk, reset_n,
   count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY even_parity IS
   . . .
end even_parity;

ARCHITECTURE ovl1 OF even_parity IS
 . . .
BEGIN
  ok1: assert_even_parity GENERIC MAP (0, 4) PORT MAP
    (clk => clk,
     reset_n => resetn,
     test_expr => stdv_to_unsigned(addr));

p1: process (RAM)
    begin
      addr <= "0011";
      case RAM is
        when bank0 =>
          addr <= "0011";
          next_bank <= bank1;
        when bank1 =>
          addr <= "0101";
          next_bank <= bank2;
        when bank2 =>
          addr <= "1001";
          next_bank <= bank3;
        when bank3 =>
          addr <= "1010" ;
          next_bank <= bank0;
    end case;
end process;

p2: process (clk, resetn)
    begin

      if resetn = ’0’ then
         RAM  <= bank0 ;
           elsif clk’event and clk = ’1’ and shift = ’1’ then
         RAM <= next_bank;
      end if;
    end process;

      adat <= addr ;

END ovl1;
```

assert_fifo_index

Overview The **assert_fifo_index** assertion ensures that a FIFO-type structure will never overflow nor underflow. This monitor can be configured to support multiple pushes (writes) into and pops (reads) from a fifo within a given cycle.

Note 1: This assertion is in a beta release status. It is being introduced in this release. Your feedback and input will help shape the final version of this assertion. Send your comments via email to: info@verificationlib.org.

Note 2: A VHDL version of this assertion is currently under construction.

Syntax ***assert_fifo_index [#(severity_level, depth, push_width, pop_width, options, msg)] inst_name (clk, reset_n, push, pop);***

severity_level	Severity of the failure.
depth	Specifies the maximum number of elements in the queue or fifo structure.
push_width	Defines the width of the push argument (that is, the maximum number of pushes for a given cycle). By default, the width is 1.
pop_width	Defines the width of the pop argument (that is, the maximum number of pops for a given cycle). By default, the width is 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified. If bit 2 is set to 1 (that is, `options&2`), then *simultaneous* pushes and pops are not allowed within any given cycle. Multiple pushes *or* pops (as specified by *push* and *pop*) within a cycle are allowed, provided that *both* pushes and pops never occur within the *same* given cycle.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking sampling event for assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
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.

Usage The **assert_fifio_index** assertion keeps track of the total number of writes and reads that have occurred for a FIFO or queue memory structure. This assertion *does* permit simultaneous pushes and pops into the queue within the same clock cycle. It ensures that the FIFO never overflows (too many writes without enough reads), and that it never underflows (too many reads without enough writes). This assertion is more powerful than the **assert_no_overflow** and **assert_no_overflow** assertions, which check only the boundary condition.

Verilog Example This example shows how **assert_fifo_index** can be used to verify that a FIFO will never overflow nor underflow. This example uses the default parameter settings for the push_width and pop_width (that is, only one push and pop can occur on a given cycle). Note: It is possible for a push *and* a pop to occur on the same cycle.

```
assert_fifo_index #(0,16) no_over_underflow (clk, reset_n, push, pop);
```

VHDL Example

TBD

assert_frame

Overview The **assert_frame** assertion 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. If the *test_expr* does not occur within these width boundaries, an assertion will fire (that is, an error condition will be detected in the code). Note: The *start_event* is a cycle transition from 0 to 1.

The intent of the minimum and maximum range is to identify legal boundaries in which *test_expr* can occur at or after *start_event*. When you specify both the minimum (equal to or greater than 0) and maximum (greater than 0) ranges, a *test_expr* must occur within the specified frame. The frame is from the time of *start_event* through *max_cks*. Additionally, *max_cks* must be greater than *min_cks*.

Special consideration one: If you do not specify a maximum range, the checker ensures that the *test_expr* does not occur until *min_cks* or later. That is, the *test_expr* must not occur before *min_cks*. However, *test_expr* is not required to occur after *min_cks*, we are simply asserting that *test_expr* does not occur before *min_cks*.

Special consideration two: If you specify that *min_cks* is equal to 0, the checker ensures that the *test_expr* occurs prior to *max_cks.* That is, *test_expr* must occur at some point in any cycle from *start_event* through *max_cks*).

Special consideration three: If you specify that both *min_cks* and *max_cks* equal 0, *test_expr* must be true when there is a 0 to 1 transition for *start_event*. That is, *start_event* implies *test_expr* (*start_event* → *test_expr*).

Syntax ***assert_frame [#(severity_level, min_cks, max_cks, flag, options, msg)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure.
min_cks	The *test_expr* cannot occur prior to (but not including) the specified minimum number of clock cycles. That is, if *test_expr* occurs at or after *start_event* but before *min_cks*, then an error occurs. **The exception is:** When *min_cks* is set to 0, then there is no minimum check (that is, *test_expr* may occur at start event).
max_cks	The *test_expr* must occur at or prior to the specified number of clock cycles. That is, if the *test_expr* does not occur at or prior to *max_cks*, then an error occurs. **The exception is:** When *max_cks* is set to 0, then there is no maximum check (any value is valid).
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.*
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking sampling event for assertion.

reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
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*.

Usage

The **assert_frame** assertion should be used in control circuits to ensure proper synchronization of events. Common uses of **assert_frame** are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification of single cycle operations to operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Verilog Example

This example shows how **assert_frame** can be used to verify cycle timing relationships between two events. This assertion claims that after the rising edge of `req` is detected, then an `ack` signal must go high within 2 to 4 clocks.

```
assert_frame #(0,2,4) check_req_ack (clk, reset_n, req, ack);
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY arbiter IS
   . . .
end arbiter;

ARCHITECTURE ovl1 OF arbiter IS
  . . .
BEGIN
  ack <= to_std(count = 2#0101#);
  req <= to_std((count = 2#0010#) OR (count = 2#0100#));
  ok1: assert_frame GENERIC MAP (1,2,4) PORT MAP
                 (clk => clk,
                  reset_n => resetn,
                  start_event => req,
                  test_expr => ack);
  PROCESS
    BEGIN
      WAIT UNTIL (clk'EVENT AND clk = '1');
        IF resetn = '0' AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_handshake

Overview The **assert_handshake** assertion continuously monitors the *req* and *ack* signals at every positive edge of the triggering event or clock *clk*. There are no defaults for this assertion; therefore, if you do not specify parameters, the assertion is invalid.

Note that both *req* and *ack* must go inactive (0) prior to starting a new handshake validation sequence. Optional checks can be performed as follows:

- *min_ack_cycle*: When this parameter is greater than zero, the check is activated. It checks whether an *ack* occurs at or after *min_ack_cycle* clocks.
- *max_ack_cycle*: When this parameter is greater than zero, the check is activated. It checks whether an *ack* occurs at or before *max_ack_cycle* clocks.
- *req_drop*: When this parameter is greater than zero, the check is activated. It checks whether *req* remains active for the entire cycle until an *ack* occurs.
- *deassert_count*: When this parameter is greater than zero, the check is activated. It checks whether *req* becomes inactive (0) within *deassert_count* clocks after an *ack* (that is, check for *req* stuck active).
- *max_ack_length*: When this parameter is greater than zero, the check is activated. It describes the maximum pulse length in terms of cycles. It checks
(a) whether *ack* is less than or equal to *max_ack_length* and
(b) whether *req* remains active (0) within *deassert_count* clocks after an *ack* (that is, check for *ack* stuck active).

Syntax ***assert_handshake [#(severity_level, min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length, options, msg)] inst_name (clk, reset_n, req, ack);***

severity_level	Severity of the failure with default value of 0.
min_ack_cycle	Activate *min_ack_cycle* check if greater than default value of 0.
max_ack_cycle	Activate *max_ack_cycle* check if greater than default value of 0.
req_drop	Activate *req_drop* check if greater than default value of 0.
deassert_count	Activate *deassert_count* if greater than default value of 0.
max_ack_length	Activate *max_ack_length* check if greater than default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion module.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
req	Signal that starts the transaction.
ack	Signal that terminates the transaction.

Usage The **assert_handshake** assertion is a general and powerful transaction check that should be used in bus transaction monitoring or protocol checking. You must include parameters with this assertion. There are no defaults.

Verilog Example

This example shows the check that a bus *hold* signal must be asserted until an acknowledgement issues (*hlda*). When the acknowledgement is asserted, the *hold* signal must be de-asserted within the next cycle.

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
module processor (CLK, RESET, HOLD, HLDA, . . .);
  input CLK;
  input RESET;
  input HOLD;
  output HDLA;

  . . .

  assert_handshake #(0,0,0,1,1)
    PROC_HOLD_HDLA (CLK, RESETIN, HOLD, HDLA);
endmodule
```

VHDL Example

```
ENTITY arbiter IS
   . . .
end arbiter;

ARCHITECTURE ovl1 OF arbiter IS
 . . .
BEGIN
       ok1: assert_handshake GENERIC MAP (0, 0, 0, 1, 1) PORT MAP
                    (clk => clk,
                     reset_n => resetn,
                     req => req,
                     ack => ack_i);
p1: process (present_state)
        begin
                if resetn = '1' then
                        ack_i <= '0';
                else
                case present_state is
                        when s0 =>
                                IF req = '1' THEN
                                  next_state <= s1;
                                ELSE
                                  next_state <= s0;
                                END IF;
                        when s1 =>
                                next_state <= s2;
                        when s2 =>
                                next_state <= s3;
                        when s3 =>
                                ack_i <= '1';
                                next_state <= s0;
                end case;
                end if;
        end process;
p2: process (clk, resetn)
        begin
                if resetn = '1' then
                        present_state <= s0;
                elsif clk'event and clk = '1' then
```

```
                              present_state <= next_state;
                        end if;
                  end process;

            END ovl1;
```

assert_implication

Overview The **assert_implication** assertion continuously monitors the *antecedent_expr*. If it evaluates to TRUE, then this checker will verify that the *consequent_expr* is TRUE.

Note: if the *antecedent_expr* evaluates FALSE, then the logic implication is valid. This is equivalent to:

```
assert_always imply (clk, A ? C : 1'b1);
```

In this example, if `A` evaluates TRUE, then `C` should evaluate TRUE. If `A` evaluates FALSE, then the assertion is still valid.

Syntax ***assert_implication [#(severity_level, options, msg)] inst_name (clk, reset_n, antecedent_expr, consequent_expr);***

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only option supported is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
antecedent_expr	Expression verified at the positive edge of *clk*.
consequent_expr	Expression verified if *antecedent_expr* is TRUE.

Usage The **assert_implication** assertion belongs to the most general class of assertions, which includes “assert_never” (see page 44) and “assert_proposition” (see page 60). Assert_implication does not contain any complex sequential check other than sampling the *antecedent_expr* and *consequent_expr* at every positive edge of *clk*. Use this assertion when you want to verify a propositional property that should always hold TRUE at clock boundaries or at the positive edge of *clk*.

Verilog Example The following example illustrates a conditional check using the assert_implication

```
assert_implication not_full (clk, reset_n, q_valid, q_not_full);
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity implication_chk is
  . . .
end implication_chk;

architecture behavior of implication_chk is

begin
. . .

assert_implication_chk: assert_implication
```

```
generic map(severity_level=>0,msg=>"error: wrong implication")

port
map(clk=>clk,reset_n=>reset_n,antecedent=>q_valid,consequence=>q_not_full);

end behavior;
```

assert_increment

Overview The **assert_increment** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will never *increase* by anything other than *value*. The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). The check will not start until the first clock after *reset_n* is asserted.

Note: It is also permissible for *test_expr* to remain the same, but if it does increase, it will never increase by anything other than *value*.

This checker is targeted for circular queue structures with an address counter that is permitted to wrap. That is, this checker permits the *test_expr* expression to wrap, as long as it increases by the specified *value*. In the following example, the first clock is `4'b1111` and the second clock is `4'b0000`. We can see that this *test_expr* has increased by a value of one (that is, the *test_expr* is wrapped). Hence, if the *value* parameter was set to 1, this is valid.

Example:

```
4'b1111 → 4'b0000
```

Special consideration: Do not use this assertion for general counters that can both increment and decrement.

Syntax ***assert_increment [#(severity_level, width, value, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
value	Maximum increment value allowed for *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_increment** assertion should be used in circuits to ensure the proper change of values in structures such as counters and finite-state machines (FSM). However, if the variable or expression being checked is allowed to increment and decrement, then “assert_delta” (see page 29) should be used instead of assert_increment.

Verilog Example The following example shows a programmable counter from 0-9 triggering an assertion monitor in the transition from 9 to 0, since this does not characterize a binary increment.

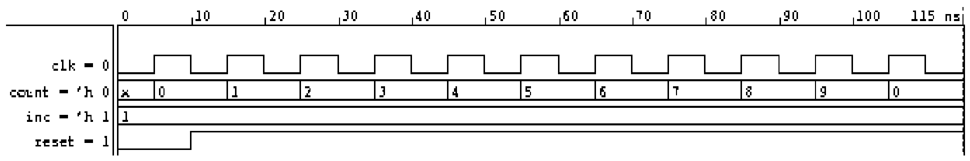
```
module programmable_counter_0_to_9 (reset_n, clk, inc);
input reset_n, clk;
input [1:0] inc;

reg [3:0] count;
```

```
always @(posedge clk)
begin
  if (reset_n == 0) count <= 4'd0;
  else if (count == 9) count <= 4'd0;
  else count <= count + inc;
end

assert_increment #(0, 4, 1, 0, “invalid binary increment”)
  invalid_count (clk, reset_n, count);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY incrementer IS
   . . .
end incrementer;

ARCHITECTURE ovl1 OF incrementer IS
   . . .
BEGIN
  test_expr <= to_unsigned(count);
  ok1: assert_increment GENERIC MAP (0, 4, 1) PORT MAP
                 (clk => clk,
                  reset_n => resetn,
                  test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk’EVENT AND clk = ’1’);
        IF resetn = ’0’ AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
   END PROCESS;
END ovl1;
```



assert_never

Overview The **assert_never** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will never evaluate TRUE. The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). When *test_expr* evaluates to TRUE, an assertion will fire (that is, an error condition will be detected in the code).

Syntax ***assert_never [#(severity_level, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_never** assertion, like “assert_always” (see page 18), “assert_implication” (see page 40), and “assert_proposition” (see page 60) is a general assertion. It does not contain any complex sequential check, other than sampling *test_expr* at every positive edge of *clk*. It should be used whenever you want to verify a propositional property that should *never* hold TRUE at clock boundaries or at the positive edge of *clk*.

Verilog Example The interface for this example assumes the fifo never overflows, nor underflows. As a result, we are safer if we guard every instance of this fifo with checks for overflow and underflow.

```
module guarded_fifo (clk, reset_n, read, write, data_in,
  data_out);

  input clk, reset_n, read, write;
  input [15:0] data_in;
  output [15:0] data_out;

  wire fifo_full, fifo_empty;

  fifo fifo (clk, reset_n, read, write, data_in, data_out,
             fifo_full, fifo_empty);

  assert_never #(0, 0, “Fifo overflow”) fifo_overflow
    (clk, reset_n, fifo_full && write);

  assert_never #(0, 0, “Fifo underflow”) fifo_underflow
    (clk, reset_n, fifo_empty && read);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY guarded_fifo IS
  . . .
end guarded_fifo;

ARCHITECTURE ovl1 OF guarded_fifo IS
  . . .

Begin
    test_expr <= fifo_full AND wr ;

ok1: assert_never GENERIC MAP (1, 0) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    test_expr => test_expr);

FIFO1 : fifo  PORT MAP
                   ( clk => clk,
     rst_ => resetn,
                     read => rd,
                     write => wr,
                     data_in => datain,
                     data_out => dataout,
                     full = fifo_full,
                     empty => fifo_empty);
END ovl1;
```

assert_next

Overview The **assert_next** assertion 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 assertion 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)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
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.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
start_event	Starting event that triggers monitoring of the *test_expr*.
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_next** assertion should be used in circuits to ensure a proper sequence of events. Common uses for **assert_next** are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification that single-cycle operations operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require that data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Verilog Example The following example includes two overlapping sequences that are being verified. The assertion claims that when A occurs, B will occur exactly 4 cycles later. Notice how a new A (or *starting event*) occurs prior to the completion of the first sequence (that is, *test_expr* B). The assertion would be coded as follows:

assert_next #(0, 4, 1) AB_check (clk, reset_n, A, B);

The overlapping sequences of length four could occur as follows:

```
A . . B
. . A . . B
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY next_count IS
   Port (
       . . .
end next_count;

ARCHITECTURE ovl1 OF next_count IS
  . . .
BEGIN
  test_expr <= to_std(count = "0100" OR count = "0101");
  start <= to_std(count = "0000");
  ok1: assert_next GENERIC MAP (1, 4, 1) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    start_event => start_event,
                    test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk’EVENT AND clk = ’1’);
        IF resetn = ’0’ AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_no_overflow

Overview The **assert_no_overflow** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. This assertion contends that a specified expression will never change values from a *max* value (default (2***width*)-1) to a value greater than *max* or less than a *min* value.

The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_no_overflow [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*. The current implementation assumes that *width* is less than or equal to 32 bits due to a limitation of "<<".
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).
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_no_overflow** assertion is most useful in counters, where it can be used to check whether there is no wrap around from the highest value to the lowest value in a range. For example, it can be used to check that pointers to memory structures will never wrap around. For a more general test of overflow, the designer should use “assert_delta” (see page 29). For fifo structures, see the more general “assert_fifo_index” (see page 33).

Verilog Example

```
module overflow_counter (reset_n, clk, count);
  input reset_n, clk;
  output [3:0] count;

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

  assert_no_overflow #(0,4) count_with_overflow (clk,
    reset_n, count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity overflow_counter is
  . . .
end overflow_counter;

architecture behavior of overflow_counter is
  . . .

begin

Op: process (clk)
  . . .
  begin
  . . .
  end if;
end process Op;

count<= cnt;
assert_no_overflow_chk: assert_no_overflow
--generic map(severity_level=>0,width=>4,max=>9)
generic map(severity_level=>0,width=>4)

port map(clk=>clk,reset_n=>reset_n,test_expr=>cnt);

end behavior;
```

assert_no_transition

Overview The **assert_no_transition** assertion continuously monitors the *test_expr* variable at every positive edge of the triggering event or clock *clk*. 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)] inst_name (clk, reset_n, test_expr, start_state, next_state);***

severity_level	Severity of the failure with default value of 0.
width	Width of state variable *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	State variable representing finite-state machine (FSM) being checked 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*.

Usage The **assert_no_transition** assertion should be used in control circuits, especially finite-state machines (FSM), to ensure that invalid transitions are never triggered.

Please note: *start_state* and *next_state* include any valid Verilog or VHDL expression (depending on the library you are using). As a result, multiple transitions can be specified by encoding the transitions in these variables. Please refer to the following example.

Verilog Example

```
module counter_09_or_0F (reset_n,clk,count,sel_09);
input reset_n, clk, sel_09;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count == 4’d9 && sel_09 == 1’b1)
    count <= 4’d0;
  else
    count <= count + 1;
end

assert_no_transition #(0,4) valid_count
  (clk,reset_n,count,4’d9, (sel_09 == 1)?4’d10:4’d0);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY counter_0_to_9 IS
   . . .
end counter_0_to_9;

ARCHITECTURE ovl1 OF counter_0_to_9 IS
  . . .
BEGIN
  test_expr <= to_unsigned(texpr);
  ok1: assert_no_transition GENERIC MAP (1) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    start_state => to_unsigned(count),
                    next_state => to_unsigned(2#1001#),
                    test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk’EVENT AND clk = ’1’);
        IF resetn = ’0’ AND count = 9 AND sel_09 = ’1’ THEN
           count <= 0;
        ELSE
           count <= count + 1;
        END IF;
        IF sel_09 = ’1’ THEN
          texpr <= "1010";
        ELSE
          texpr <= "1010";
        END IF;
   END PROCESS;
END ovl1;
```

assert_no_underflow

Overview The **assert_no_underflow** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. This assertion contends that a specified *test_expr* will never change values from a *min* value (default 0) to a value less than *min,* nor will it change to a value greater than or equal to a *max* value (default (2***width*)-1). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_no_underflow [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr.* The current implementation assumes that *width* is less than or equal to 32 bits due to a limitation of "<<".
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).
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk.*

Usage The **assert_no_underflow** assertion is most useful in counter circuits, where it can be used to check whether there is no wrap around from the lowest value to the highest value in a range. For example, it can be used to check that pointers to memory structures will never wrap around. For a more general test of underflow, the designer should use “assert_delta” (see page 29). For fifo structures, see the more general “assert_fifo_index” (see page 33).

Verilog Example

```
module underflow_counter (reset_n,clk,count);
input reset_n, clk;
output [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0) count <= 4’b0000;
  else count <= count - 1;
end

assert_no_underflow #(0,4) count_with_underflow (clk, reset_n,
   count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY underflow IS
   . . .
end underflow;

ARCHITECTURE ovl1 OF underflow IS
  . . .
BEGIN
  test_expr <= to_unsigned(count);
  ok1: assert_no_underflow GENERIC MAP (1) PORT MAP
                   (clk => clk,
                    reset_n => resetn,
                    test_expr => test_expr);
  PROCESS
    BEGIN
      WAIT UNTIL (clk’EVENT AND clk = ’1’);
        IF resetn = ’0’ AND count < 9 THEN
           count <= 0;
        ELSE
           count <= count - 1;
        END IF;
   END PROCESS;
END ovl1;
```

assert_odd_parity

Overview The **assert_odd_parity** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will always have an odd number of bits 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 or VHDL expression (depending on the library you are using).

Syntax ***assert_odd_parity [#(severity_level, width, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_odd_parity** assertion is most useful in control and datapath circuits. It ensures that a variable or expression has an odd number of bits asserted. Some example uses of **assert_odd_parity** are:

- address or data busses with error checking based on parity
- finite-state machines (FSM) with error detection mechanisms

Verilog Example

```
module counter_0_to_9_with_parity (reset_n,clk,count);
input reset_n, clk;
output [4:0] count;
reg [4:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count >= 9) count <= 5'b10000;
  else begin
    count[3:0] = count[3:0] + 1;
    count[4] = ~(^count[3:0]);
  end
end

assert_odd_parity #(0,5) parity_sanity_check (clk, reset_n,
   count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY odd_parity IS
  . . .
end odd_parity;

ARCHITECTURE ovl1 OF odd_parity IS
 . . .
BEGIN
  ok1: assert_odd_parity GENERIC MAP (0, 4) PORT MAP
    (clk => clk,
     reset_n => resetn,
     test_expr => stdv_to_unsigned(addr));

p1: process (RAM)
    begin
      addr <= "0001";
      case RAM is
        when bank0 =>
          addr <= "0001";
          next_bank <= bank1;
        when bank1 =>
          addr <= "0111";
          next_bank <= bank2;
        when bank2 =>
          addr <= "1101";
          next_bank <= bank3;
        when bank3 =>
          addr <= "1110" ;
          next_bank <= bank0;
      end case;
end process;

p2: process (clk, resetn)
    begin

      if resetn = '0' then
         RAM  <= bank0 ;
      elsif clk'event and clk = '1' and shift = '1' then
         RAM <= next_bank;
    end if;
end process;

  adat <= addr ;

END ovl1;
```

assert_one_cold

Overview The **assert_one_cold** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will always be one_cold or will have the appropriate *inactive* state. Otherwise, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_one_cold [#(severity_level, width, inactive, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
inactive	Specifies the inactive state of *test_expr*. If inactive is set to 0, then the expression can be one_cold or all zeroes. If inactive is set to 1, then the expression can be one_cold or all ones. Otherwise, the default value is 2, which specifies that *text_expr* can only be one_cold.
options	Vendor options. Currently, the only option supported is *options*=1, which defines the assertion as a constraint to formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_one_cold** assertion is most useful in control circuits. It ensures that the state variable of a finite-state machine (FSM) implemented with one-hot encoding will maintain proper behavior (that is, exactly one bit is asserted low). In datapath circuits, **assert_one_cold** can be used to ensure that the enabling signals of bus-based designs will not generate bus contention.

Verilog Example

```
module one_cold_example (reset_n,clk,count);
input reset_n, clk;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0) count <= 4’b0001;
  else count <= (count << 1) | count[3];
end

assert_one_cold #(0,4) count_one_cold_check (clk, reset_n,
   ~(count));

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY one_cold IS
   . . .
end one_cold;

ARCHITECTURE ovl1 OF one_cold IS
 . . .
BEGIN
  ok1: assert_one_cold GENERIC MAP (2, 4) PORT MAP
      (clk => clk,
       reset_n => resetn,
       test_expr => stdv_to_unsigned(set));

p1: process (present_state)
    begin
      set <= "1110";
      case present_state is
        when s0 =>
          set <= "1110";
          next_state <= s1;
        when s1 =>
          set <= "1101";
          next_state <= s2;
        when s2 =>
          set <= "1011";
          next_state <= s3;
        when s3 =>
          set <= "0111" ;
          next_state <= s0;
    end case;
end process;

p2: process (clk, resetn)
    begin
      if resetn = ’1’ then
         present_state <= s0;
      elsif clk’event and clk = ’1’ and shift = ’1’ then
        present_state <= next_state;
    end if;
end process;

  a <= set ;

END ovl1;
```

assert_one_hot

Overview The **assert_one_hot** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will have exactly one 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 or VHDL expression (depending on the library you are using).

Syntax ***assert_one_hot [#(severity_level, width, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_one_hot** assertion is most useful in control circuits. It ensures that the state variable of a finite-state machine (FSM) implemented with one-hot encoding will maintain proper behavior (that is, exactly one bit is asserted). In datapath circuits, **assert_one_hot** can be used to ensure that the enabling signals of bus-based designs will not generate bus contention.

Verilog Example

```
module one_hot_example (reset_n,clk,count);
input reset_n, clk;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0) count <= 4’b0001;
  else count <= (count << 1) | count[3];
end

assert_one_hot #(0,4) count_one_hot_check (clk, reset_n,
   count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
ENTITY one_hot IS
 . . .
end one_hot;
```

```
ARCHITECTURE ovl1 OF one_hot IS
 . . .

  ok1: assert_one_hot GENERIC MAP (2, 4) PORT MAP
      (clk => clk,
       reset_n => resetn,
       test_expr => stdv_to_unsigned(set));

p1: process (present_state)
    begin
      set <= "0001";
      case present_state is
        when s0 =>
          set <= "0001";
          next_state <= s1;
        when s1 =>
          set <= "0010";
          next_state <= s2;
        when s2 =>
          set <= "0100";
          next_state <= s3;
        when s3 =>
          set <= "1000" ;
           next_state <= s0;
      end case;
end process;

p2: process (clk, resetn)
    begin
      if resetn = '1' then
        present_state <= s0;
      elsif clk'event and clk = '1' and shift = '1' then
        present_state <= next_state;
      end if;
end process;

        a <= set ;

END ovl1;
```

assert_proposition

Overview The **assert_proposition** assertion *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 contends that a specified *test_expr* will always evaluate TRUE. If *test_expr* evaluates to FALSE, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using).

Syntax ***assert_proposition [#(severity_level, options, msg)] inst_name (reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified.

Usage The **assert_proposition** assertion belongs to the most general class of assertions, which includes “assert_always” (see page 18) and “assert_never” (see page 44). It does not contain any implied sequential behavior. This assertion should be used whenever you want to verify a propositional property that should hold TRUE at all times.

The monitor **assert_proposition** differs from **assert_always** in that the former monitors *test_expr* at all times, the latter monitors *test_expr* at clock boundaries. Thus, if the correct behavior of *test_expr* involves changes in value outside clock boundaries, the designer should use *“assert_always” (see page 18)* rather than assert_proposition.

Verilog Example

```
module counter_0_to_9(reset_n,clk);
input reset_n, clk;

reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count >= 9) count <= 1’b0;
  else count <= count + 1;
end

assert_proposition #(0, 0, “error: count not within 0 and 9”)
   valid_count (reset_n, (count >= 4’b0000) &&
     (count <= 4’b1001));

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity counter_proposition is
. . .
end counter_proposition;

architecture behavior of counter_proposition is
. . .
begin

Op: process (clk)
. . .
  begin
. . .
end if;
end process Op;

Op2: process (cnt)
begin
. . .
end if;
end process Op2;
count<= cnt;
assert_proposition_chk: assert_proposition
generic map(severity_level=>0,options=>0,msg=>"error:count is not within 0 and 
9")
port map(reset_n=>reset_n,test_expr=>test_xpr);

end behavior;
```

assert_quiescent_state

Overview The **assert_quiescent_state** assertion continuously monitors the *sample_event* at every positive edge of clock *clk*. It contends that the state machine (that is, *state_expr*) is equal to a specified *check_value* constant value at the rising edge of *sample_event*, and optionally, at the end of simulation.

Syntax ***assert_quiescent_state [#(severity_level, width, options, msg)] inst_name (clk, reset_n, state_expr, check_value, sample_event);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *state_expr*.
options	Vendor options. Currently, the only option supported is *options*=1, which defines the assertion as a constraint to formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
state_expr	Expression being verified at the positive edge of *clk* when quiescent.
check_value	Specified value for *state_expr* when quiescent.
sample_event	When TRUE, it will trigger the quiescent state check.

Usage This assertion is useful for validating state machines after a transaction completes (using the *sample_event* to quiescent the state). In addition, if you are using the Verilog library, you can define the Verilog macro ***'ASSERT_END_OF_SIMULATION,*** which can be used to quiescent the state expression at the end of simulation. For example:

```
+define+ASSERT_END_OF_SIMULATION=top.simulation_done.
```

A rising edge on ASSERT_END_OF_SIMULATION will trigger a quiescent check.

Verilog Example

```
assert_quiescent_state valid_state #(0,4) (clk, reset_n,  transaction_state,
4'b000, end_of_transaction);
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity assert_quiescent_state_example is
  . . .
end assert_quiescent_state_example;

architecture behavior of assert_quiescent_state_example is
  . . .

begin
  . . .
```

```
assert_quiescent_state_chk: assert_quiescent_state
generic map(severity_level=>0,width=>4)
port map(clk=>clk, reset_n=>reset_n, state_expr=>transaction_state,
check_value=>chk_val, sample_event=>end_of_transaction);

end behavior;
```

assert_range

Overview The **assert_range** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *test_expr* will always have a value within a legal *min*/*max* range, otherwise, an assertion will fire (that is, an error condition will be detected in the code). The *test_expr* can be any valid Verilog or VHDL expression (depending on the library you are using). The *min* and *max* should be a valid parameter and *min* must be less than or equal to *max*.

Syntax ***assert_range [#(severity_level, width, min, max, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
min	Minimum value allowed for range check. Default to 0.
max	Maximum value allowed for range check. Default to (2***width* - 1).
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_range** assertion should be used in circuits to ensure the proper range of values in control structures, such as counters and finite-state machines (FSM). In datapath circuits, this assertion can be used to check whether the variable or expression is evaluated within the allowed range.

Verilog Example

```
module counter (reset_n,clk,count);
input reset_n, clk;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || counter == 4’d9) count <= 4’b0000;
  else count <= count + 1;
end

assert_range #(0,4,0,9) count_range_check (clk, reset_n,
   count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity counter is
  . . .
end counter;

architecture behavior of counter is
  . . .
begin

Op: process (clk)
  . . .
  begin
  . . .
  end if;
end process Op;

count<= cnt;
assert_range_chk: assert_range
generic map(severity_level=>0,width=>4,min=>0,max=>9)
port map(clk=>clk,reset_n=>reset_n,test_expr=>test_xpr);

end behavior;
```

assert_time

Overview The **assert_time** assertion 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)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure.
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*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking sampling event for assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
start_event	Starting event that triggers monitoring of the *test_expr*.
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_time** assertion should be used in control circuits to ensure proper synchronization of events. Common uses of **assert_time** are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification of single cycle operations to operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Verilog Example

```
module transaction (clk, reset_n, req, ack, data);
input reset_n, clk, req;
input [3:0] data;
output ack;
reg ack;

parameter MAX_COUNT = 3;

reg [3:0] count;
reg [3:0] data_in;

always @(posedge clk) begin
  if (reset_n == 0) begin
    ack <= 0;
    count <= 0;
  end
  else if (count != 0) begin
    count = count - 1;
    if (count == 1) begin
      ack <= 1;
      data_in <= data;
    end
  end
  else if (req == 1) begin
    count <= MAX_COUNT;
  end else if (ack == 1) begin
    ack <= 0;
  end
end

assert_time #(0,3,0) req_ack_test (clk, reset_n, req == 1,
  (ack == 0) || (data >= 1 && data <= 3));

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity transaction is
  . . .
end transaction;

architecture behavior of transaction is
  . . .
  begin
Op: process (clk)
  . . .
  begin
  . . .
  end if;
  . . .
end process Op;

Op2: process (ackn, data)
  begin
  . . .
  end if;
end process Op2;
ack <= ackn;
assert_time_chk: assert_time
generic map(severity_level=>0,num_cks=>3,flag=>0)
port map(clk=>clk,reset_n=>reset_n,start_event=>req,test_expr=>test_xpr);

end behavior;
```

assert_transition

Overview The **assert_transition** assertion continuously monitors the *test_expr* variable at every positive edge of the triggering event or clock *clk*. When *test_expr* evaluates to the value of *start_state*, the assertion monitor ensures that if *test_expr* changes value, then it will 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)] inst_name (clk, reset_n, test_expr, start_state, next_state);***

severity_level	Severity of the failure with default value of 0.
width	Width of state variable *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	State variable representing finite-state machine (FSM) being checked at the positive edge of *clk*.
start_state	Triggering state of *test_expr*.
next_state	Next valid state for machine represented by *test_expr* when traversed from state *start_state*.

Usage The **assert_transition** assertion should be used in control circuits, especially finite-state machines (FSM) to ensure that required transitions are triggered.

Please note *start_state* and *next_state* are verification events that can be represented by any valid Verilog or VHDL expression. As a result, multiple transitions can be specified by encoding the transitions in these variables.

Verilog Example

```
module counter_09_or_0F (reset_n,clk,count,sel_09);
input reset_n, clk, sel_09;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 || count == 4’d9 && sel_09 == 1’b1)
    count <= 4’d0;
  else
    count <= count + 1;
end

assert_transition #(0,4) valid_count (clk, reset_n, count,
  4’d9, (sel_09 == 1’b0) ? 4’d10 : 4’d0);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity counter is
  . . .
end counter;
architecture behavior of counter is
  . . .
begin

start <= "1001";

Op1: process (clk)
  . . .

  begin
    . . .
  end if;
end process Op1;

Op2: Next_st <= "0000" when sel_09 = '1' and cnt = start else cnt;

  . . .

assert_transition_example:assert_transition
generic map (severity_level=>0,width=>4)
port map
(clk=>clk,reset_n=>reset_n,test_expr=>cnt,start_state=>start,next_state=>Next
_st);
end behavior;
```

assert_unchange

Overview The **assert_unchange** assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When this signal (or expression) evaluates TRUE, the assertion monitor ensures that the *test_expr* will not change values within the next *num_cks* number of clocks.

Syntax ***assert_unchange [#(severity_level, width, num_cks, flag, options, msg)] inst_name (clk, reset_n, start_event, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
num_cks	For this number of clocks after *start_event* is asserted, *test_expr* must remain unchanged. Otherwise, an error is triggered.
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*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
start_event	Starting event that triggers monitoring of the *test_expr*
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_unchange** assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will remain unchanged until another event occurs. Common uses for **assert_unchange** include:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification of single cycle operations will operate correctly with data loaded at different cycles
- verification of synchronizing conditions that require data is stable after a specified initial triggering event

Verilog Example In this example, we assume that for the multi-cycle divider operation to work properly, the value of signal *a* must remain unchanged for the duration of the operation that, in this example, is 16 cycles.

```
module division_with_check (clk,reset_n,a,b,start,done);
  input clk, reset_n;
  input [15:0] a,b;
  input start;
  output done;

  wire [15:0] q,r;

  div16 #(16) div01 (clk, reset_n, start, a, b, q, r, done);
  assert_unchange #(0,16,16) div_unchange_a (clk, reset_n,
     start == 1, a);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity division_with_check is
   . . .

end division_with_check;

architecture behavior of disision_with_check is
  . . .
begin

  . . .

assert_unchange_chk: assert_unchange
generic map(severity_level=>0, width=>16, num_clks=>16,options=>0,
msg=>"Division op error")
port map(clk=>clk,reset_n=>reset_n,start_event=>start,test_expr=>a);

end behavior;
```

assert_width

Overview The **assert_width** assertion continuously monitors the *test_expr*. When this signal (or expression) evaluates TRUE, the assertion monitor 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)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
min_cks	The *test_expr* cannot occur prior to (but not including) the specified minimum number of clock cycles. That is, if *test_expr* occurs at or after *start_event* but before *min_cks*, then an error occurs. **The exception is:** When *min_cks* is set to 0, then there is no minimum check (that is, *test_expr* may occur at start event).
max_cks	The *test_expr* must occur at or prior to the specified number of clock cycles. That is, if the *test_expr* does not occur at or prior to *max_cks*, then an error occurs. **The exception is:** When *max_cks* is set to 0, then there is no maximum check (any value is valid).
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_width** assertion, a superset of assert_time, should be used in control circuits to ensure proper synchronization of events. Common uses for **assert_width** are as follows:

- verification that multicycle operations with enabling conditions will always work with the same data
- verification that single cycle operations perform correctly with data loaded at different cycles
- verification of synchronizing conditions that require data is stable after a specified initial triggering event (such as in an asynchronous transaction requiring req/ack signals)

Verilog Example

```
module transaction (clk, reset_n, req, ack, data);
input reset_n, clk, req;
input [3:0] data;
output ack;
reg ack;

parameter MAX_COUNT = 3;

reg [3:0] count;
reg [3:0] data_in;

always @(posedge clk) begin
  if (reset_n == 0) begin
```

```
      ack <= 0;
      count <= 0;
    end
    else if (count != 0) begin
      count = count - 1;
      if (count == 1) begin
        ack <= 1;
        data_in <= data;
      end
    end
    else if (req == 1) begin
      count <= MAX_COUNT;
    end else if (ack == 1) begin
      ack <= 0;
    end
  end

  //Verify that the ACK will occur within 2 or 3 clocks

  assert_width #(0,2,3) req_ack_test (clk, reset_n, ack == 1);

  endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity transaction is
  . . .
end transaction;
architecture behavior of transaction is

  . . .
begin
  . . .

begin
  if rising_edge(clk) then
  . . .
  end if;
end process Op;
assert_width_chk: assert_width
generic map (severity_level=>0, min_cks=>3,max_cks=>5,options=>0,msg=>"ACK
fails")
port map (clk=>clk, reset_n=>reset_n, test_expr=> ackn);
end behavior;
```

assert_win_change

Overview The **assert_win_change** assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When this signal (or expression) evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values prior to the *end_event*.

When the *start_event* evaluates TRUE, the assertion monitor ensures that the *test_expr* changes values on a clock edge at some point up to and including the next *end_event*. Once the *test_expr* evaluates TRUE, it is not necessary for it to remain TRUE throughout the remainder of the test (up to and including *end_event*). Hence, the *test_expr*, does not have to be TRUE at the *end_event*, provided it was true at some point during the test (up to and including *end_event*).

Syntax ***assert_win_change [#(severity_level, width, options, msg)] inst_name (clk, reset_n, start_event, test_expr, end_event);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
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*.

Usage The **assert_win_change** assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will change after the *start_event* and before the *end_event*. Common uses for **assert_win_change** include:

- verification that synchronization circuits respond after a specified initial stimuli. For example, that a bus transaction will occur without any bus interrupts. Another example is that a memory write command will not occur while we are in a memory read cycle.
- verification that finite-state machines (FSM) change state or will go to a specific state after a specified initial stimuli and before another specified event occurs

Verilog Example In this example, an assertion is used to check if *data_bus* is asserted before an asynchronous read operation is finished.

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
module processor (clk, reset_n, ..., rd, rd_ack, ...);
  input clk;
  input reset_n;

  output rd
  input rd_ack;

  inout [31:0] data_bus;

  ...

  assert_win_change #(0,32) sync_data_bus_with_rd
    (clk, reset_n, rd, data_bus, rd_ack);
endmodule
```

VHDL Example

```
entity processor is

  . . .


--end entity processor;--end entity only allows in VHDL 93;
end processor;

architecture behavior of processor is

begin

  . . .
assert_win_change_chk: assert_win_change
generic map(severity_level=>0, width => 32,options=>0, msg=>"Memory read op
error")
port
map(clk=>clk,reset_n=>reset_n,start_event=>rd,test_expr=>data_bus,end_event=>
rd_ack);

end behavior;
```

assert_win_unchange

Overview The **assert_win_unchange** assertion continuously monitors the *start_event* at every positive edge of the triggering event or clock *clk*. When this signal (or expression) evaluates TRUE, the assertion monitor 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)] inst_name (clk, reset_n, start_event, test_expr, end_event);***

severity_level	Severity of the failure with default value of 0.
width	Width of *test_expr* with default value of 1.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
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 the *start_event.*

Usage The **assert_win_unchange** assertion should be used in circuits to ensure that after a specified initial event, a particular variable or expression will remain unchanged after the *start_event* and before the *end_event*. Common uses for **assert_win_unchange** include:

- verification that non-deterministic multicycle operations with enabling conditions will always work with the same data
- verification of synchronizing conditions requiring that data is stable after a specified initial triggering event, and before an ending condition takes place.
  For example: a bus transaction will occur without any bus interrupts
  Another example: a memory write command will not occur if we are in a memory read cycle

Verilog Example In this example, assume that for the multi-cycle divider operation to work properly, the value of signal *a* must remain unchanged for the duration of the operation. Completion is signaled by the signal *done*.

```
module division_with_check (clk,reset_n,a,b,start,done);
  input clk, reset_n;
  input [15:0] a,b;
  input start;
  output done;

  wire [15:0] q,r;

  div16 div01 (clk, reset_n, start, a, b, q, r, done);

  assert_win_unchange #(0,16)
    div_win_unchange_a (clk, reset_n, start, a, done);

endmodule
```

VHDL Example

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
entity division_with_check is
  . . .

--end entity division_with_check;--end entity only allows in VHDL 93;
end division_with_check;

architecture behavior of disision_with_check is
  . . .
begin

  . . .

assert_win_unchange_chk: assert_win_unchange
generic map(severity_level=>0, options=>0, msg=>"Division op error")
port
map(clk=>clk,reset_n=>reset_n,start_event=>start,test_expr=>a,end_event=>done
);

end behavior;
```

assert_window

Overview The **assert_window** assertion continuously monitors the *start_event* at every positive clock edge *clk*. When the *start_event* evaluates TRUE, the assertion monitor 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)] inst_name (clk, reset_n, start_event, test_expr, end_event);***

severity_level	Severity of the failure with default value of 0.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
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*.

Usage The **assert_window** assertion should be used in control circuits to ensure properly synchronized events. Common uses of **assert_window** are the following:

- verification that multicycle operations with enabling conditions always work with the same data;
- verification that single cycle operations operate correctly with data loaded at different cycles;
- verification of synchronizing conditions that require data is stable after a specified initial triggering event.

Verilog Example In this example we test that an asynchronous operation will change the value on a data bus between a starting and an ending event.

Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
module asynchronous_processor (clk, reset_n, ...);

...
assert_window #(0, 0, “bus op error”) bus_operation (clk, reset_n, bus_req,
  data_bus, bus_ack);
...

endmodule
```

VHDL Example

```
entity assert_window_example is
· · ·

--end entity assert_window_example;--end entity only allows in VHDL 93;
end assert_window_example;
```

```
architecture behavior of assert_window_example is
. . .
  begin

    OP: process (clk)
      begin
        if rising_edge (clk) then
        ...

        end if;
    end process OP;
  assert_window_chk: assert_window
  generic map(severity_level=>0, options=>0, msg=>"bus op error")
  port
  map(clk=>clk,reset_n=>reset_n,start_event=>T_RDY,test_expr=>te  st_xpr,end_
event=>T_DNE);
  end behavior;
```

assert_zero_one_hot

Overview The **assert_zero_one_hot** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. It contends that a specified *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 or VHDL expression (depending on the library you are using).

Syntax ***assert_zero_one_hot [#(severity_level, width, options, msg)] inst_name (clk, reset_n, test_expr);***

severity_level	Severity of the failure with default value of 0.
width	Width of the monitored expression *test_expr*.
options	Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.
msg	Error message that will be printed if the assertion fires.
inst_name	Instance name of assertion monitor.
clk	Triggering or clocking event that monitors the assertion.
reset_n	Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*).
test_expr	Expression being verified at the positive edge of *clk*.

Usage The **assert_zero_one_hot** assertion is most useful in control circuits. It ensures that the state variable of a finite-state machine (FSM) implemented with zero-one-hot encoding will maintain proper behavior. In datapath circuits, **assert_zero_one_hot** can be used to ensure that the enabling signals of bus-based designs will not generate bus contention. Examples of uses for **assert_zero_one_hot** include controllers, circuit enabling logic, and arbitration logic.

Verilog Example

```
module zero_one_hot_example (reset_n, clk, count, load,
  load_count);
input reset_n, clk, load;
input [3:0] load_count;
output [3:0] count;
reg [3:0] count;

always @(posedge clk)
begin
  if (reset_n == 0 | count == 4’d0) count <= 4’b0001;
  else if (load) count <= load_count;
  else count <= count << 1;
end

assert_zero_one_hot #(0,4) count_zero_one_hot_check (clk,
  reset_n, count);

endmodule
```

VHDL Example Note: Ellipses (. . .) are used to reduce the size of the example, they denote text that is not shown. The following example demonstrates usage, it is not intended for implementation.

```
    ENTITY zero_one_hot IS
    . . .
    end zero_one_hot;

    ARCHITECTURE ovl1 OF zero_one_hot IS
     . . .
    BEGIN
      ok1: assert_zero_one_hot GENERIC MAP (0, 4) PORT MAP
          (clk => clk,
           reset_n => resetn,
           test_expr => stdv_to_unsigned(set));

    p1: process (present_state)
        begin
          set <= "0000";
          case present_state is
            when s0 =>
              set <= "0000";
              next_state <= s1;
            when s1 =>
              set <= "0001";
              next_state <= s2;
            when s2 =>
              set <= "0010";
              next_state <= s3;
            when s3 =>
              set <= "0100" ;
              next_state <= s0;
        end case;
    end process;

    p2: process (clk, resetn)
        begin
          if resetn = ’0’ then
             present_state <= s0;
          elsif clk’event and clk = ’1’ and shift = ’1’ then
             present_state <= next_state;
        end if;
    end process;

      a <= set ;

END ovl1;
```

This page intentionally left blank.

APPENDIX A

Assertion Library Definitions for Verilog

The following pages of this appendix include an entry for each of the Verilog assertion-specific library definitions. Please find the VHDL definitions in Appendix B, “Assertion Library Definitions for VHDL” on page 147.

assert_always

```
module assert_always (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n, test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_ALWAYS";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        if (test_expr  != 1’b1) begin
          ovl_error("");
        end
      end
  end
‘endif
//synopsys translate_on

endmodule
```

assert_always_on_edge

```
module assert_always_on_edge (clk, reset_n, sampling_event, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter edge_type= 0;
  parameter options=0;
  parameter msg="VIOLATION";
  input clk, reset_n, sampling_event, test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT ALWAYS ON EDGE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_header.h"
  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  reg sampling_event_fired;
  reg sampling_event_prev;

  initial sampling_event_prev <= 0;

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin // active low reset
    ‘endif
          // Capture Sampling Event @Clock for rising edge detections
        sampling_event_prev <= sampling_event;
        if ((edge_type == ‘OVL_NOEDGE) && (!test_expr))
          ovl_error("");
        else if ((edge_type == ‘OVL_POSEDGE) && (!sampling_event_prev) &&
                        (sampling_event) && (!test_expr))
          ovl_error("");
        else if ((edge_type == ‘OVL_NEGEDGE) && (sampling_event_prev) &&
                        (!sampling_event) && (!test_expr))
          ovl_error("");
          else if ((edge_type == ‘OVL_ANYEDGE) &&
                        (sampling_event_prev != sampling_event) &&
(!test_expr))
            ovl_error("");
          end
      end
```

```
‘endif
//synopsys translate_on
endmodule
```

assert_change

```
module assert_change (clk, reset_n, start_event, test_expr);
// synopsys template
  parameter severity_level=0;
  parameter width=1;
  parameter num_cks=1;
  parameter flag=0;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input start_event;
  input [width-1:0] test_expr;


//synopsys translate_off
‘ifdef ASSERT_ON

  parameter CHANGE_START = 1’b0;
  parameter CHANGE_CHECK = 1’b1;
  parameter FLAG_IGNORE_NEW_START = 2’b00;
  parameter FLAG_RESET_ON_START   = 2’b01;
  parameter FLAG_ERR_ON_START     = 2’b10;
  parameter assert_name = "ASSERT_CHANGE";

  reg r_change;
  reg [width-1:0] r_test_expr;
  reg r_state;
  integer i;

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  initial begin
    if (~((flag == FLAG_IGNORE_NEW_START) ||
          (flag == FLAG_RESET_ON_START) ||
          (flag == FLAG_ERR_ON_START))) begin
      ovl_error("illegal flag parameter");
    end
    r_state=CHANGE_START;
    r_change=1’b0;
  end
```

```
    always @(posedge clk) begin
      ‘ifdef ASSERT_GLOBAL_RESET
        if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
      ‘else
        if (reset_n != 0) begin  // active low reset
      ‘endif
          case (r_state)
            CHANGE_START:
              if (start_event == 1’b1) begin
                r_change <= 1’b0;
                r_state <= CHANGE_CHECK;
                r_test_expr <= test_expr;
                i <= num_cks;
              end
            CHANGE_CHECK:
              begin
                // Count clock ticks
                if (start_event == 1’b1) begin
                  if (flag == FLAG_IGNORE_NEW_START && i > 0)
                    i <= i-1;
                  else if (flag == FLAG_RESET_ON_START)
                    i <= num_cks;
                  else if (flag == FLAG_ERR_ON_START) begin
                    ovl_error("illegal start event");
                  end
                end
                else if (i > 0)
                  i <= i-1;

                if (r_test_expr != test_expr) begin
                  r_change <= 1’b1;
                  // finish matching when change has occurred
                  r_state  <= CHANGE_START;
                end

                // go to start state on last check
                if (i == 1 && !(start_event == 1’b1 &&
                                flag == FLAG_RESET_ON_START)) begin
                  r_state <= CHANGE_START;

                  // Check that the property is true
                  if ((r_change != 1’b1) && (r_test_expr == test_expr))
  begin
                    ovl_error("");
                  end
                end
                r_test_expr <= test_expr;
              end
          endcase
        end
        else begin
           r_state <= CHANGE_START;
           r_change <= 1’b0;
        end
    end // always
  ‘endif
  //synopsys translate_on

  endmodule
```

assert_cycle_sequence

```
module assert_cycle_sequence (clk, reset_n, event_sequence);
// synopsys template
  parameter severity_level = 0;
  parameter num_cks=1;// number of clocks for the sequence
  parameter necessary_condition = 0;
  parameter options = 0;
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [num_cks-1:0] event_sequence;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_CYCLE_SEQUENCE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  // internal parameters
  parameter width     = (num_cks>0)?num_cks:1;
  parameter num_cks_1 = width - 1;
  parameter num_buf_1 = (num_cks_1+1)*num_cks_1/2;

  reg [num_buf_1:1] pipe_regs;
  initial pipe_regs = 0;

  integer e_idx;
  initial e_idx = num_cks_1;

  initial begin
    if (num_cks <= 0) begin // assert invalid num_cks
      $display("WARNING: %s : parameter num_cks (%d <= 0) is invalid :
time %0t : %m",
                     assert_name, num_cks, $time);
      $display("          Use default value 1 for parameter num_cks.");
    end
  end

  ‘ifdef ASSERT_GLOBAL_RESET
    buf (rst_n, ‘ASSERT_GLOBAL_RESET);
  ‘else
    buf (rst_n, reset_n);
  ‘endif
```

```
    integer ii, jj, nn, pp, lim, start;

    reg and_res;
    always @(posedge clk) begin
      if (rst_n == 0) begin // active low reset
        pipe_regs = 0;      // fill ZEROs to pipes
        e_idx <= num_cks_1;
      end
      else begin
        if (width == 1) begin
          if (!event_sequence[0]) begin // check error
                ovl_error("");
          end
        end
        else if (necessary_condition != 0) begin // non-pipelined
          if (pipe_regs[1] == 1’b0) begin        // INIT
            if (event_sequence[num_cks_1]) begin
              pipe_regs[1] <= 1’b1;              // start CHECK
              e_idx <= num_cks_1 - 1’b1;
            end
          end
          else begin                       // CHECK
            if (event_sequence[e_idx] != 1’b1) begin
              pipe_regs[1] <= 1’b0;
              ovl_error("");
            end
            if (e_idx > 0) begin
              e_idx <= e_idx - 1’b1;
            end
            else begin
              pipe_regs[1] <= 1’b0;       // done CHECK, go to INIT
            end
          end
        end
        else begin         // (necessary_condition==0), pipelined
          lim = num_cks_1;
          nn = 0; jj = 0; start = 0;
          for (pp = num_cks_1; pp >= lim; pp = pp-1) begin
            nn = nn + 1;
            start = start + nn;
            jj = start;
            and_res = 1’b1;               // get preconditon with and_res
            for (ii = nn; ii < (pp+nn); ii = ii+1) begin
              and_res = and_res && pipe_regs[jj];
              jj = jj + ii;
            end
            if (and_res && !event_sequence[num_cks_1 - pp]) begin // check
  error
              ovl_error("");
            end
          end
          for (ii = 1; ii < num_buf_1; ii = ii+1) begin // update pipes
            pipe_regs[ii] <= pipe_regs[ii+1];
          end
          jj = 1;
          for (ii = 1; ii <= num_cks_1; ii = ii+1) begin
            pipe_regs[jj] <= event_sequence[ii];
            jj = jj + ii + 1;
          end
        end
      end
    end // always
  ‘endif
  //synopsys translate_on

  endmodule
```

assert_decrement

```
module assert_decrement (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter value=1;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg [width-1:0] last_test_expr;
  reg [width:0] temp_expr;
  reg r_reset_n, r_r_reset_n;
  initial r_reset_n = 0;
  initial r_r_reset_n = 0;

  parameter assert_name = "ASSERT_DECREMENT";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin

    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
        r_reset_n <= ‘ASSERT_GLOBAL_RESET;
    ‘else
      if (reset_n != 1’b0) begin
        r_reset_n <= reset_n;
    ‘endif
        r_r_reset_n <= r_reset_n;
        last_test_expr <= test_expr;

      // check second clock afer reset
        if (r_reset_n && r_r_reset_n &&
            (last_test_expr != test_expr)) begin
          temp_expr = {1’b0,last_test_expr} - {1’b0,test_expr};
        // 2’s complement result
```

```
          if (temp_expr[width-1:0] != value) begin
            ovl_error("");
          end
        end
      end
      else begin
        r_reset_n <= 0;
        r_r_reset_n <= 0;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_delta

```
module assert_delta (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter min=1;
  parameter max=1;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg [width-1:0] last_test_expr;
  reg [width:0] temp_expr1;
  reg [width:0] temp_expr2;
  reg r_reset_n, r_r_reset_n;
  initial r_reset_n = 0;
  initial r_r_reset_n = 0;

  parameter assert_name = "ASSERT_DELTA";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
        r_reset_n <= ‘ASSERT_GLOBAL_RESET;
    ‘else
      if (reset_n != 1’b0) begin
        r_reset_n <= reset_n;
    ‘endif
        r_r_reset_n <= r_reset_n;
        last_test_expr <= test_expr;

     // check second clock afer reset
        if (r_reset_n && r_r_reset_n &&
            (last_test_expr != test_expr)) begin
          temp_expr1 = {1’b0,last_test_expr} - {1’b0,test_expr};
          temp_expr2 = {1’b0,test_expr} - {1’b0,last_test_expr};
        // 2’s complement result
          if (!((temp_expr1[width-1:0]>=min && temp_expr1[width-
1:0]<=max) ||
                (temp_expr2[width-1:0]>=min && temp_expr2[width-
1:0]<=max)))
          begin
            ovl_error("");
          end
        end
      end
      else begin
        r_reset_n <= 0;
        r_r_reset_n <= 0;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_even_parity

```
module assert_even_parity (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1  // Previous version of the library
‘else                 // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_EVEN_PARITY";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if ((^(test_expr)) == 1’b1) begin
          ovl_error("");
        end
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_fifo_index

```
module assert_fifo_index (clk, reset_n, push, pop);
// synopsys template
  parameter severity_level = 0;
  parameter depth=1;
  parameter push_width = 1;
  parameter pop_width = 1;
  parameter options=0;
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [push_width-1:0] push;
  input [pop_width-1:0] pop;

//synopsys translate_off
‘ifdef ASSERT_ON
  // local parameters
  parameter no_push_pop = ((options & 2) != 0);
  parameter assert_name = "ASSERT_FIFO_INDEX";

  integer error_count;
  integer cnt;
  initial begin
    cnt=0;
    error_count = 0;
    if (depth==0) ovl_error("Depth parameter value must be > 0");
  end

  ‘include "ovl_header.h"
  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin // active low reset
    ‘endif
        if ({push!=0,pop!=0} == 2’b10) begin // push
          if ((cnt + push) > depth) begin
             ovl_error("OVERFLOW");
          end
          else begin
            cnt <= cnt + push;
          end
        end
        else if ({push!=0,pop!=0} == 2’b01) begin // pop
          if (cnt < pop) begin
             ovl_error("UNDERFLOW");
          end
          else begin
            cnt <= cnt - pop;
          end
        end
        else if ({push!=0,pop!=0} == 2’b11) begin // push & pop
          if (no_push_pop) begin
             ovl_error("ILLEGAL PUSH AND POP");
          end
          else begin
            if ((cnt + push - pop) > depth) begin
              ovl_error("OVERFLOW");
            end
            if ((cnt + push) < pop) begin
              ovl_error("UNDERFLOW");
            end
            else begin
              cnt <= cnt + push - pop;
            end
          end
        end
      end
      else begin
        cnt <= 0;
      end
  end
‘endif
//synopsys translate_on
endmodule
```

assert_frame

```
module assert_frame (clk, reset_n, start_event, test_expr);
  // synopsys template
  parameter severity_level=0;
  parameter min_cks=0;
  parameter max_cks=0;
  parameter flag=0;
  parameter options = 0;
  parameter msg="VIOLATION";

  input clk;
  input reset_n;
  input start_event;
  input test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON
  // local parameters
  parameter assert_name = "ASSERT_FRAME";
  parameter num_cks = (max_cks>min_cks)?max_cks:min_cks;
  parameter FRAME_START = 1’b0;
  parameter FRAME_CHECK = 1’b1;
  parameter FLAG_IGNORE_NEW_START = 2’b00;
  parameter FLAG_RESET_ON_START   = 2’b01;
  parameter FLAG_ERR_ON_START     = 2’b10;
  parameter START_flag=flag & 2’b11;
  parameter EDGE_flag=flag & 3’b100;

  reg r_state;
  reg r_start_event;

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  initial begin
    // *** NEW STATIC CHECK FOR MIN/MAX and flag ***
    if (~((START_flag == FLAG_IGNORE_NEW_START) ||
          (START_flag == FLAG_RESET_ON_START) ||
          (START_flag == FLAG_ERR_ON_START))) begin
       ovl_error("illegal flag parameter");
    end
    if (max_cks && (max_cks < min_cks)) begin
       ovl_error("min_cks > max_cks");
    end
    r_state=FRAME_START;
```

```
      r_start_event = 1’b0;
    end

    integer ii;

    reg r_test_expr;
    initial r_test_expr = 1’b0;

    always @(posedge clk) begin
      r_start_event <= start_event;
      if (EDGE_flag) r_test_expr <= test_expr;
    end

    always @(posedge clk) begin
      ‘ifdef ASSERT_GLOBAL_RESET
        if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
      ‘else
        if (reset_n != 0) begin  // active low reset
      ‘endif
          case (r_state)
            FRAME_START:
              // assert_frame() behaves like assert_implication()
              //     when min_cks==0 and max_cks==0
              if ((min_cks==0) && (max_cks==0)) begin
                if ((start_event==1’b1) && (test_expr==1’b0)) begin
                  // FAIL, it does not behave like assert_implication()
                  ovl_error("");
                end
              end
              // wait for start_event (0->1)
              else if ((r_start_event == 1’b0) && (start_event == 1’b1))
  begin
                r_state <= FRAME_CHECK;
                ii <= 1;
              end
            FRAME_CHECK:
              // start_event (0->1) has occurred
              // start checking
              begin
                // Count clock ticks
                if ((r_start_event == 1’b0) && (start_event == 1’b1)) begin
                  // start_event (0->1) happens again -- re-started!!!
                  if (START_flag == FLAG_IGNORE_NEW_START) begin
                    if (max_cks) ii <= ii + 1;
                    else if (ii < min_cks) ii <= ii + 1;
                  end
                  else if (START_flag == FLAG_RESET_ON_START)
                    ii <= 1;
                  else if (START_flag == FLAG_ERR_ON_START) begin
                    ovl_error("illegal start event");
                    r_state <= FRAME_START;
                  end
                end
                else begin
                    if (max_cks) ii <= ii + 1;
                    else if (ii < min_cks) ii <= ii + 1;
                end

                // Check for (0,0), (0,M), (m,0), (m,M) conditions
                if (min_cks == 0) begin
                  if (max_cks == 0) begin
                    // (0,0): (min_cks==0, max_cks==0)
                    // This condition is UN-REACHABLE!!!
                    ovl_error("");
                    r_state <= FRAME_START;
                  end
                  else begin // max_cks > 0
                    // (0,M): (min_cks==0, max_cks>0)
                    if ((r_test_expr==1’b0) && (test_expr == 1’b1)) begin
                      // OK, ckeck is done. Go to FRAME_START state for
  next check.
                      r_state <= FRAME_START;
                    end
                    else begin
                      if (ii == max_cks) begin
                        // FAIL, test_expr does not happen at/before
  max_cks
                        ovl_error("");
                        r_state <= FRAME_START;
                      end
                    end
                  end
                end
                else begin // min_cks > 0
                  if (max_cks == 0) begin
                    // (m,0): (min_cks>0, max_cks==0)
                    if ((r_test_expr==1’b0) && (test_expr == 1’b1)) begin
                      // FAIL, test_expr should not happen before min_cks
                      ovl_error("");
                      r_state <= FRAME_START;
                    end
                    else begin
                      if (ii == min_cks) begin
                        // OK, test_expr does not happen before min_cks
                        r_state <= FRAME_START;
                      end
                    end
                  end
                  else begin // max_cks > 0
                    // (m,M): (min_cks>0, max_cks>0)
                    if ((r_test_expr==1’b0) && (test_expr == 1’b1)) begin
                      r_state <= FRAME_START;
                      if (ii < min_cks) begin
                        // FAIL, test_expr should not happen before min_cks
                        ovl_error("");
                      end
                      // else OK, we are done!!!
                    end
```

```
                  else begin
                    if (ii == max_cks) begin
                      // FAIL, test_expr does not happen at/before
max_cks
                      ovl_error("");
                      r_state <= FRAME_START;
                    end
                  end
                end
              end
            end
        endcase
      end
      else begin
         r_state <= FRAME_START;
      end
  end // always
‘endif
endmodule
```

assert_handshake

```
module assert_handshake (clk, reset_n, req, ack);
// synopsys template
  parameter severity_level=0;
  parameter min_ack_cycle=0;      // default don’t check
  parameter max_ack_cycle=0;      // default don’t check
  parameter req_drop=0;           // default don’t check
  parameter deassert_count=0;     // default don’t check
  parameter max_ack_length=0;     // default don’t check
‘ifdef ASSERT_V1_0_1          // Previous version of the library
‘else                         // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input req;
  input ack;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter REQ_ACK_START    = 2’b00;
  parameter REQ_ACK_WAIT     = 2’b01;
  parameter REQ_ACK_ERR      = 2’b10;
  parameter REQ_ACK_DEASSERT = 2’b11;

  reg [1:0] r_state;
  reg [1:0] r_r_state;
  reg r_req;
  reg r_ack;
  integer i;
  integer j;

  parameter assert_name = "ASSERT_EVENT_HANDSHAKE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  initial begin
    r_state=REQ_ACK_START;
    r_r_state=REQ_ACK_START;
    r_req=0;
    r_ack=0;
    i = 0;
```

```
      j = 0;
    end

    always @(posedge clk) begin
      ‘ifdef ASSERT_GLOBAL_RESET
        if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
      ‘else
        if (reset_n != 0) begin  // active low reset
      ‘endif
          case (r_state)
            REQ_ACK_START:
              begin
                if ((max_ack_length != 0) && ack == 1’b1 && r_ack == 1’b1)
  begin
                  j <= j+1;
                  if (j >= max_ack_length) begin
                    r_state <= REQ_ACK_ERR;
                    ovl_error("ack max length violation");
                  end
                end
                if (req == 1’b1) begin

                  if (r_ack == 1’b1 && ack == 1’b1 && r_req == 1’b0) begin
                    r_state <= REQ_ACK_ERR;
                    ovl_error("multiple req violation");
                  end
                  else if (deassert_count != 0 && r_req == 1’b1 && req ==
  1’b1 &&
                      ack == 1’b0) begin
                    r_state <= REQ_ACK_DEASSERT;
                    i <= deassert_count;
                  end
                  else if ((min_ack_cycle != 0) && ack && r_ack == 1’b0)
  begin
                    ovl_error("ack min cycle violation");
                  end
                  else if (ack == 1’b0) begin
                    r_state <= REQ_ACK_WAIT;
                    i <= 1;
                    j <= 0;
                  end
                end
                else begin
                  if (ack == 1’b1 && r_ack == 1’b0) begin
                    r_state <= REQ_ACK_ERR;
                    ovl_error("ack without req violation");
                  end
                end
              end
            REQ_ACK_WAIT:
              begin
                i <= i + 1;
                if (ack) begin
                  r_state <= REQ_ACK_START;
                  j <= 1;
                end

                if ((min_ack_cycle != 0) && (i < min_ack_cycle) &&
                                      ack == 1'b1) begin
                  r_state <= REQ_ACK_ERR;
                  ovl_error("ack min cycle violation");
                end
                else if ((!ack) && (max_ack_cycle != 0) && i >=
max_ack_cycle) begin
                  r_state <= REQ_ACK_ERR;
                  ovl_error("ack max cycle violation");
                end
                else if (req_drop == 1'b1 && req == 1'b0) begin
                  r_state <= REQ_ACK_ERR;
                  ovl_error("req drop violation");
                end
                else if (req == 1'b1 && r_req == 1'b0) begin
                  r_state <= REQ_ACK_ERR;
                  ovl_error("multiple req violation");
                end
              end
            REQ_ACK_ERR:
              begin
                if (req == 1'b1 && ack == 1'b0 && r_req == 1'b0) begin
                  r_state <= REQ_ACK_WAIT;
                  i <= 0;
                  j <= 0;
                end
                else if (ack == 1'b0 && r_ack == 1'b1) begin
                  r_state <= REQ_ACK_START;
                  i <= 0;
                  j <= 0;
                end
              end
            REQ_ACK_DEASSERT:
              begin
                i <= i-1;
                if (i == 1) begin
                  if (req == 1'b1) begin
                    r_state <= REQ_ACK_ERR;
                    ovl_error("req deassert violation");
                  end
                  else
                    r_state <= REQ_ACK_START;
                end
              end
          endcase
          r_r_state <= r_state;
          r_ack <= ack;
          r_req <= req;
        end
        else begin
           r_state <= REQ_ACK_START;
```

```
         r_r_state <= REQ_ACK_START;
         r_ack <= 0;
         r_req <= 0;

         i <= 0;
         j <= 0;
      end
  end // always
‘endif
//synopsys translate_on

endmodule
```

assert_implication

```
module assert_implication (clk, reset_n, antecendent_expr,
consequent_expr);
// synopsys template
  input clk, reset_n, antecendent_expr, consequent_expr;
  parameter severity_level = 0;
  parameter options = 0;
  parameter msg="VIOLATION";

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_IMPLICATION";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        if (antecendent_expr  == 1’b1 && consequent_expr  == 1’b0) begin
          ovl_error("");
        end
      end
  end
‘endif
//synopsys translate_on

endmodule
```

assert_increment

```
module assert_increment (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter value=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg [width-1:0] last_test_expr;
  reg [width:0] temp_expr;
  reg r_reset_n, r_r_reset_n;
  initial r_reset_n = 0;
  initial r_r_reset_n = 0;

  parameter assert_name = "ASSERT_INCREMENT";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
        r_reset_n <= ‘ASSERT_GLOBAL_RESET;
    ‘else
      if (reset_n != 1’b0) begin
        r_reset_n <= reset_n;
    ‘endif
        r_r_reset_n <= r_reset_n;
        last_test_expr <= test_expr;

      // check second clock afer reset
        if (r_reset_n && r_r_reset_n &&
          (last_test_expr != test_expr)) begin
          temp_expr = {1’b0,test_expr} - {1’b0,last_test_expr};
        // 2’s complement result
          if (temp_expr[width-1:0] != value) begin
```

```
            ovl_error("");
          end
        end
      end
      else begin
        r_reset_n <= 0;
        r_r_reset_n <= 0;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_never

```
module assert_never (clk, reset_n, test_expr);
// synopsys template
  input clk, reset_n, test_expr;
  parameter severity_level = 0;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_NEVER";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if (test_expr  != 1’b0) begin
          ovl_error("");
        end
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_next

```
module assert_next (clk, reset_n, start_event, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter num_cks=1;
  parameter check_overlapping=1;
  parameter only_if=0;  // if 1, test_expr can only appear if a
corresponding
                        // start_event occurs
  parameter options = 0;
  parameter msg="VIOLATION";
  input clk, reset_n, start_event, test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  initial begin
    if (num_cks <= 0) begin
      ovl_error("num_cks parameter<=0");
    end
  end

  parameter assert_name = "ASSERT_NEXT";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  reg [((num_cks>0)?num_cks-1:0):0] monitor;

  wire [((num_cks>0)?num_cks-1:0):0] monitor_1 = (monitor << 1);

  initial monitor = 0;

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
       if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
       if (reset_n != 0) begin  // active low reset
    ‘endif
         monitor <= (monitor_1 | start_event);
         if ((check_overlapping == 0) && (monitor_1 != 0) && start_event)
begin
           ovl_error("illegal overlapping condition detected");
         end
         else if ((only_if == 1) && !monitor[num_cks-1] && test_expr)
begin
           ovl_error("test_expr without start_event");
         end
         else if (monitor[num_cks-1] && ~test_expr) begin
           ovl_error("start_event without test_expr");
         end
       end
       else begin
         monitor <= 0;
       end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_no_overflow

```
module assert_no_overflow (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter min=0;
  parameter max= ((1<<width)-1);
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  // local paramaters used as defines
  parameter OVERFLOW_START = 1’b0;
  parameter OVERFLOW_CHECK = 1’b1;

  reg r_state;
  initial r_state=OVERFLOW_START;

  parameter assert_name = "ASSERT_NO_OVERFLOW";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
          OVERFLOW_START:
            if (test_expr == max) begin
               r_state <= OVERFLOW_CHECK;
            end
          OVERFLOW_CHECK:
            if (test_expr != max) begin
              r_state <= OVERFLOW_START;
              if (test_expr <= min || test_expr > max) begin
```

```
                ovl_error("");
              end
            end
        endcase
      end
      else begin
        r_state <= OVERFLOW_START;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_no_transition

```
module assert_no_transition (clk, reset_n, test_expr, start_state,
next_state);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr, start_state, next_state;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg [width-1:0] r_next_state, r_start_state;

  reg assert_state;
  initial assert_state = 1’b0;

  parameter assert_name = "ASSERT_NO_TRANSITION";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if (assert_state == 1’b0) begin // INIT_STATE
          if (test_expr == start_state) begin
            assert_state  <= 1’b1;      // CHECK_STATE
            r_start_state <= start_state;
            r_next_state  <= next_state;
          end
        end
        else begin                      // CHECK_STATE
          if (test_expr == r_next_state) begin
            ovl_error("");    // test_expr moves to unexpected state
            assert_state <= 1’b0;
          end
          else if (test_expr != r_start_state) begin
            assert_state <= 1’b0; // done ok.
          end
        end
      end
      else begin
        assert_state <= 1’b0;
      end
  end // always
‘endif
//synopsys translate_on

endmodule
```

assert_no_underflow

```
module assert_no_underflow (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter min=0;
  parameter max= ((1<<width)-1);
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

// local paramaters used as defines
  parameter UNDERFLOW_START = 1’b0;
  parameter UNDERFLOW_CHECK = 1’b1;

  reg r_state;
  initial r_state=UNDERFLOW_START;

  parameter assert_name = "ASSERT_NO_OVERFLOW";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
          UNDERFLOW_START:
            if (test_expr == min) begin
              r_state <= UNDERFLOW_CHECK;
            end
          UNDERFLOW_CHECK:
            begin
              if (test_expr != min) begin
                r_state <= UNDERFLOW_START;
                if (test_expr >= max || test_expr < min) begin
                  ovl_error("");
                end
              end
            end
        endcase
      end
      else begin
        r_state <= UNDERFLOW_START;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_odd_parity

```
module assert_odd_parity (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_ODD_PARITY";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if ((^(test_expr)) != 1’b1) begin
          ovl_error("");
        end
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_one_cold

```
module assert_one_cold (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=32;
  parameter inactive=2;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                               // New version to allow for future
options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON
  wire [width-1:0] test_expr_i = ~test_expr;
  wire [width-1:0] test_expr_i_1 = test_expr_i - {{width-1{1’b0}},1’b1};
  wire inactive_val=(inactive==1)?1’b1:1’b0;

  parameter assert_name = "ASSERT_ONE_COLD";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if ((test_expr ^ test_expr)==0) begin
          // OK, test_expr contains no X/Z.
          if ((inactive>1) || (test_expr!={width{inactive_val}})) begin
            if (( test_expr_i == {width{1’b0}}) ||
                ((test_expr_i & test_expr_i_1) != {width{1’b0}})) begin
              ovl_error("");
            end
          end
        end
        else begin
          ovl_error("Error: test_expr contains X/Z value");
        end
      end
```

```
   end // always

‘endif
//synopsys translate_on

endmodule
```

assert_one_hot

```
module assert_one_hot (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=32;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                               // New version to allow for future
options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON
  wire [width-1:0] test_expr_1 = (test_expr - {{width-1{1’b0}},1’b1}) &&
            (&test_expr!=1’bx) && (^test_expr!=1’bx);

  parameter assert_name = "ASSERT_ONE_HOT";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if ((test_expr ^ test_expr)==0) begin
          // OK, test_expr contains no X/Z.
          if ((test_expr == {width{1’b0}}) ||
              (test_expr & test_expr_1) != {width{1’b0}}) begin
            ovl_error("");
          end
        end
        else begin
          ovl_error("Error: test_expr contains X/Z value");
        end
      end
   end // always

‘endif
//synopsys translate_on
```

```
endmodule
```

assert_proposition

```
module assert_proposition (reset_n, test_expr);
// synopsys template
  input reset_n, test_expr;
  parameter severity_level = 0;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_PROPOSITION";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        if (test_expr == 1’b0) begin
          ovl_error("");
        end
      end
    @(reset_n or test_expr);
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_quiescent_state

```
module assert_quiescent_state (clk, reset_n, state_expr,
                               check_value, sample_event);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter options = 0;
  parameter msg="VIOLATION";
  input clk, reset_n, sample_event;
  input [width-1:0] state_expr, check_value;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_QUIESCENT_STATE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  reg r_sample_event;
  initial r_sample_event=1’b0;

  always @(posedge clk) r_sample_event <= sample_event;

 ‘ifdef ASSERT_END_OF_SIMULATION
    reg r_EOS;
    initial r_EOS=1’b0;
    always @(posedge clk) r_EOS <= ‘ASSERT_END_OF_SIMULATION;
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        ‘ifdef ASSERT_END_OF_SIMULATION
          if ((r_EOS == 1’b0 && ‘ASSERT_END_OF_SIMULATION ==1’b1) ||
                        (r_sample_event == 1’b0 && sample_event == 1’b1))
&&
                        (state_expr  != check_value)) begin
        ‘else
          if ((r_sample_event == 1’b0 && sample_event == 1’b1) &&
                        (state_expr  != check_value)) begin
        ‘endif
```

```
            ovl_error("");
          end
      end
  end
‘endif
//synopsys translate_on

endmodule
```

assert_range

```
module assert_range (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
  parameter min=0;
  parameter max= ((1<<width)-1);
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter assert_name = "ASSERT_RANGE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        if (((test_expr)<min) || ((test_expr)>max)) begin
          ovl_error("");
        end
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_time

```
module assert_time (clk, reset_n, start_event, test_expr);
// synopsys template
  input clk, reset_n, start_event, test_expr;
  parameter severity_level = 0;
  parameter num_cks = 1;
  parameter flag = 2’b00; //ignore_new_start
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";

//synopsys translate_off
‘ifdef ASSERT_ON

// local paramaters used as defines
  parameter TIME_START = 1’b0;
  parameter TIME_CHECK = 1’b1;
  parameter FLAG_IGNORE_NEW_START = 2’b00;
  parameter FLAG_RESET_ON_START   = 2’b01;
  parameter FLAG_ERR_ON_START     = 2’b10;

  reg [31:0] i;

  reg r_state;
  initial begin
    if (~((flag == FLAG_IGNORE_NEW_START) ||
          (flag == FLAG_RESET_ON_START) ||
          (flag == FLAG_ERR_ON_START))) begin
      ovl_error("illegal flag parameter");
    end
    r_state=TIME_START;
  end

  parameter assert_name = "ASSERT_TIME";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
          TIME_START:
            if (start_event == 1’b1) begin
              r_state <= TIME_CHECK;
              i <= num_cks;
            end
          TIME_CHECK:
            begin
              // Count clock ticks
              if (start_event == 1’b1) begin
                if (flag == FLAG_IGNORE_NEW_START)
                  i <= i-1;
                else if (flag == FLAG_RESET_ON_START)
                  i <= num_cks;
                else if (flag == FLAG_ERR_ON_START) begin
                  ovl_error("illegal start event");
                end
              end
              else
                i <= i-1;

              // Check that the property is true
              if (test_expr != 1’b1) begin
                ovl_error("");
              end

              // go to start state on last time check
              // NOTE: i == 0 at end of current simulation
              // timeframe due to non-blocking assignment!
              // Hence, check  i == 1.
              if (i == 1 && !(start_event == 1’b1 &&
                              flag == FLAG_RESET_ON_START))
                r_state <= TIME_START;
            end
        endcase
      end
      else begin
        r_state <= TIME_START;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_transition

```
module assert_transition (clk, reset_n, test_expr, start_state,
next_state);
// synopsys template
  parameter severity_level = 0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr, start_state, next_state;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg [width-1:0] r_start_state, r_next_state;

  reg assert_state;
  initial assert_state = 1’b0;

  parameter assert_name = "ASSERT_TRANSITION";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        if (assert_state == 1’b0) begin // INIT_STATE
          if (test_expr == start_state) begin
            assert_state  <= 1’b1; // CHECK_STATE
            r_start_state <= start_state;
            r_next_state  <= next_state;
          end
        end
        else begin                      // CHECK_STATE
          if (test_expr == r_next_state) begin
            assert_state <= 1’b0; // done ok.
          end
          if (test_expr != r_start_state) begin
```

```
            ovl_error("");    // test_expr moves to unexpected state
            assert_state <= 1’b0; // done error.
          end
        end
      end
      else begin
        assert_state <= 1’b0;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_unchange

```
module assert_unchange (clk, reset_n, start_event, test_expr);
// synopsys template
  parameter severity_level=0;
  parameter width=1;
  parameter num_cks=1;
  parameter flag=0;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input start_event;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON

  parameter UNCHANGE_START = 1’b0;
  parameter UNCHANGE_CHECK = 1’b1;
  parameter FLAG_IGNORE_NEW_START = 2’b00;
  parameter FLAG_RESET_ON_START   = 2’b01;
  parameter FLAG_ERR_ON_START     = 2’b10;

  reg r_change;
  reg [width-1:0] r_test_expr;
  reg r_state;
  integer i;

  initial begin
    if (~((flag == FLAG_IGNORE_NEW_START) ||
          (flag == FLAG_RESET_ON_START) ||
          (flag == FLAG_ERR_ON_START))) begin
      ovl_error("illegal flag parameter");
    end
    r_state=UNCHANGE_START;
    r_change=1’b0;
  end

  parameter assert_name = "ASSERT_UNCHANGE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif
```

```
    always @(posedge clk) begin
      ‘ifdef ASSERT_GLOBAL_RESET
        if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
      ‘else
        if (reset_n != 0) begin  // active low reset
      ‘endif
          case (r_state)
            UNCHANGE_START:
              if (start_event == 1’b1) begin
                r_change <= 1’b0;
                r_state <= UNCHANGE_CHECK;
                r_test_expr <= test_expr;
                i <= num_cks;
              end
            UNCHANGE_CHECK:
              begin
                // Count clock ticks
                if (start_event == 1’b1) begin
                  if (flag == FLAG_IGNORE_NEW_START && i > 0)
                    i <= i-1;
                  else if (flag == FLAG_RESET_ON_START)
                    i <= num_cks;
                  else if (flag == FLAG_ERR_ON_START) begin
                    ovl_error("illegal start event");
                  end
                end
                else if (i > 0) begin
                  i <= i-1;
                end

                if (r_test_expr != test_expr) begin
                  r_change <= 1’b1;
                end
                // go to start state on last check
                if (i == 1 && !(start_event == 1’b1 &&
                                flag == FLAG_RESET_ON_START)) begin
                  r_state <= UNCHANGE_START;
                end
                // Check that the property is true
                if ((r_change == 1’b1) ||
                    (r_test_expr != test_expr)) begin
                  ovl_error("");
                end
                  r_test_expr <= test_expr;
              end
          endcase
        end
        else begin
          r_state<=UNCHANGE_START;
          r_change<=1’b0;
        end
    end // always
  ‘endif
```

```
//synopsys translate_on

endmodule
```

assert_width

```
module assert_width (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level=0;
  parameter min_cks=1;
  parameter max_cks=1;
  parameter options=0;
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input test_expr;


//synopsys translate_off
‘ifdef ASSERT_ON

‘ifdef ASSERT_REPORT
‘else
‘define ASSERT_REPORT $display
‘endif

  parameter WIDTH_START = 2’b00;
  parameter WIDTH_CKMIN = 2’b01;
  parameter WIDTH_CKMAX = 2’b10;
  parameter WIDTH_IDLE  = 2’b11;

  reg r_test_expr;
  reg [1:0] r_state;
  integer num_cks;

  initial begin
    r_state=WIDTH_START;
    r_test_expr = 1’b0;
    num_cks = 0;
  end


  parameter assert_name = "ASSERT_WIDTH";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  initial begin
    if ((min_cks > 0) && (max_cks > 0))
      if (min_cks > max_cks) ovl_error("min_cks > max_cks");
```

```
end

always @(posedge clk) begin
  r_test_expr <= test_expr;
  ‘ifdef ASSERT_GLOBAL_RESET
    if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
  ‘else
    if (reset_n != 0) begin  // active low reset
  ‘endif
      case (r_state)
        WIDTH_START:
          if ((r_test_expr == 1’b0) && (test_expr == 1’b1)) begin
            num_cks <= 1;
            if      (min_cks > 0) r_state <= WIDTH_CKMIN;
            else if (max_cks > 0) r_state <= WIDTH_CKMAX;
          end
        WIDTH_CKMIN:
          if (test_expr == 1’b1) begin
            num_cks <= num_cks + 1;
            if (num_cks >= min_cks) begin
              if (max_cks > 0) r_state <= WIDTH_CKMAX;
              else             r_state <= WIDTH_IDLE;
            end
          end
          else begin
            if (num_cks < min_cks) begin
              ovl_error("MIN_CHECK");
            end
            r_state <= WIDTH_START;
          end
        WIDTH_CKMAX:
          if (test_expr == 1’b1) begin
            num_cks <= num_cks + 1;
            if (num_cks > max_cks) begin
              ovl_error("MAX_CHECK");
              r_state <= WIDTH_IDLE;
            end
          end
          else begin
            if (num_cks > max_cks) begin
              ovl_error("MAX_CHECK");
            end
            r_state <= WIDTH_START;
          end
        WIDTH_IDLE:
          if (test_expr == 1’b0) begin
            r_state <= WIDTH_START;
          end
      endcase
    end
    else begin
       r_state <= WIDTH_START;
       r_test_expr <= 1’b0;
    end
  end // always
‘endif
//synopsys translate_on

endmodule
```

assert_win_change

```
module assert_win_change (clk, reset_n, start_event, test_expr,
end_event);
// synopsys template
  parameter severity_level=0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input start_event;
  input [width-1:0] test_expr;
  input end_event;


//synopsys translate_off
‘ifdef ASSERT_ON

  reg r_change;
  reg [width-1:0] r_test_expr;
  reg r_state;

  parameter WIN_CHANGE_START = 1’b0;
  parameter WIN_CHANGE_CHECK = 1’b1;

  initial begin
    r_state=WIN_CHANGE_START;
    r_change=1’b0;
  end

  parameter assert_name = "ASSERT_WIN_CHANGE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
```

```
          WIN_CHANGE_START:
            if (start_event == 1’b1) begin
              r_change <= 1’b0;
              r_state <= WIN_CHANGE_CHECK;
              r_test_expr <= test_expr;
            end
          WIN_CHANGE_CHECK:
            begin
              if (r_test_expr != test_expr) begin
                r_change <= 1’b1;
//              r_state<=WIN_CHANGE_START;
              end
              // go to start state on last check
              if (end_event == 1’b1) begin
                r_state <= WIN_CHANGE_START;
                // Check that the property is true
                if ((r_change != 1’b1) &&
                    (r_test_expr == test_expr)) begin
                  ovl_error("");
                end
              end
              r_test_expr <= test_expr;
            end
        endcase
      end
     else begin
       r_state<=WIN_CHANGE_START;
       r_change<=1’b0;
     end
  end // always
‘endif
//synopsys translate_on

endmodule
```

assert_win_unchange

```
module assert_win_unchange (clk, reset_n, start_event, test_expr,
end_event);
// synopsys template
  parameter severity_level=0;
  parameter width=1;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk;
  input reset_n;
  input start_event;
  input [width-1:0] test_expr;
  input end_event;

//synopsys translate_off
‘ifdef ASSERT_ON

  reg r_change;
  reg [width-1:0] r_test_expr;
  reg r_state;

  parameter WIN_UNCHANGE_START = 1’b0;
  parameter WIN_UNCHANGE_CHECK = 1’b1;

  initial begin
    r_state=WIN_UNCHANGE_START;
    r_change=1’b0;
  end

  parameter assert_name = "ASSERT_WIN_UNCHANGE";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
          WIN_UNCHANGE_START:
            if (start_event == 1’b1) begin
              r_change <= 1’b0;
              r_state <= WIN_UNCHANGE_CHECK;
              r_test_expr <= test_expr;
            end
          WIN_UNCHANGE_CHECK:
            begin
              if (r_test_expr != test_expr) begin
                r_change <= 1’b1;
              end
              // go to start state on last check
              if (end_event == 1’b1) begin
                r_state <= WIN_UNCHANGE_START;
              end
              // Check that the property is true
              if ((r_change == 1’b1) ||
                  (r_test_expr != test_expr)) begin
                ovl_error("");
              end
              r_test_expr <= test_expr;
            end
        endcase
      end
      else  begin
        r_state<=WIN_UNCHANGE_START;
        r_change<=1’b0;
      end
  end // always
‘endif
//synopsys translate_on

endmodule
```

assert_window

```
module assert_window (clk, reset_n, start_event, test_expr, end_event);
// synopsys template
  input clk, reset_n, start_event, test_expr, end_event;
  parameter severity_level = 0;
‘ifdef ASSERT_V1_0_1   // Previous version of the library
‘else                  // New version to allow for future options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";

//synopsys translate_off
‘ifdef ASSERT_ON

// local paramaters used as defines
  parameter WINDOW_START = 1’b0;
  parameter WINDOW_CHECK = 1’b1;

  reg r_state;
  initial r_state=WINDOW_START;

  parameter assert_name = "ASSERT_WINDOW";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0) begin
    ‘else
      if (reset_n != 0) begin  // active low reset
    ‘endif
        case (r_state)
          WINDOW_START:
            if (start_event == 1’b1) begin
              r_state <= WINDOW_CHECK;
            end
          WINDOW_CHECK:
            begin
              if (end_event == 1’b1) begin
                r_state <= WINDOW_START;
              end
              if (test_expr != 1’b1) begin
                ovl_error("");
              end
            end
        endcase
      end
      else begin
        r_state <= WINDOW_START;
      end
  end // always

‘endif
//synopsys translate_on

endmodule
```

assert_zero_one_hot

```
module assert_zero_one_hot (clk, reset_n, test_expr);
// synopsys template
  parameter severity_level = 0;
  parameter width=32;
‘ifdef ASSERT_V1_0_1 // Previous version of the library
‘else                               // New version to allow for future
options
  parameter options = 0;
‘endif
  parameter msg="VIOLATION";
  input clk, reset_n;
  input [width-1:0] test_expr;

//synopsys translate_off
‘ifdef ASSERT_ON
  wire [width-1:0] test_expr_1 = test_expr - {{width-1{1’b0}},1’b1};

  parameter assert_name = "ASSERT_ZERO_ONE_HOT";

  integer error_count;
  initial error_count = 0;

  ‘include "ovl_task.h"

  ‘ifdef ASSERT_INIT_MSG
    initial
      ovl_init_msg; // Call the User Defined Init Message Routine
  ‘endif

  always @(posedge clk) begin
    ‘ifdef ASSERT_GLOBAL_RESET
      if (‘ASSERT_GLOBAL_RESET != 1’b0 && test_expr != {width{1’b0}})
begin
    ‘else
      if (reset_n != 1’b0) begin
    ‘endif
        if ((test_expr ^ test_expr)==0) begin
          // OK, test_expr contains no X/Z.
          if ((test_expr & test_expr_1) != {width{1’b0}}) begin
            ovl_error("");
          end
        end
        else begin
          ovl_error("Error: test_expr contains X/Z value");
        end
      end
    end
  ‘endif
  //synopsys translate_on
endmodule
```

This page intentionally left blank.

APPENDIX B

Assertion Library Definitions for VHDL

The following pages of this appendix include an entry for each of the VHDL assertion-specific library definitions. Additionally, the “VHDL Package” begins on page 208.

Please find the *Verilog* definitions in Appendix A, “Assertion Library Definitions for Verilog” on page 83.

assert_always

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_always IS
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ALWAYS VIOLATION");
  PORT (clk, reset_n, test_expr: IN std_ulogic);
END assert_always;

ARCHITECTURE ovl OF assert_always IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (test_expr = ’0’) THEN
        valid <= ’0’;
      ELSE
        valid <= ’1’;
      END IF;
    ELSE
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
--
-- /* {
```

assert_always_on_edge

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_always_on_edge IS
  GENERIC (severity_level: INTEGER := 0;
           edge_type: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ALWAYS ON EDGE VIOLATION");
  PORT (clk, reset_n, sampling_event, test_expr: IN std_ulogic);
END assert_always_on_edge;

ARCHITECTURE ovl OF assert_always_on_edge IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL sampling_event_fired : std_ulogic;
  SIGNAL sampling_event_prev  : std_ulogic := ’0’;
  CONSTANT OVL_NOEDGE  : INTEGER := 0;
  CONSTANT OVL_POSEDGE : INTEGER := 1;
  CONSTANT OVL_NEGEDGE : INTEGER := 2;
  CONSTANT OVL_ANYEDGE : INTEGER := 3;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      -- Capture Sampling Event @Clock for rising edge detections
      sampling_event_prev <= sampling_event;
      IF (test_expr /= ’1’) THEN
        IF (edge_type = OVL_NOEDGE) THEN
          valid <= ’0’;
        ELSIF ((edge_type = OVL_POSEDGE) AND
               (sampling_event_prev = ’0’) AND (sampling_event = ’1’))
THEN
          valid <= ’0’;
        ELSIF ((edge_type = OVL_NEGEDGE) AND
               (sampling_event_prev = ’1’) AND (sampling_event = ’0’))
THEN
          valid <= ’0’;
```

```
            ELSIF ((edge_type = OVL_ANYEDGE) AND
               (sampling_event_prev /= sampling_event)) THEN
          valid <= ’0’;
            END IF;
      END IF;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_change

```
-- } */-
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_change IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           num_cks: INTEGER := 1;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT CHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_change;

ARCHITECTURE ovl OF assert_change IS
  TYPE stateT IS (CHANGE_START, CHANGE_CHECK);
  CONSTANT FLAG_IGNORE_NEW_START: INTEGER := 0;
  CONSTANT FLAG_RESET_ON_START  : INTEGER := 1;
  CONSTANT FLAG_ERR_ON_START    : INTEGER := 2;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL flag_error: std_ulogic := ’0’;
  SIGNAL flag_para_error: std_ulogic := ’0’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL ii: INTEGER;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
  ASSERT flag_error = ’0’ REPORT msg & " : illegal start event"
                               SEVERITY ovlSevTab(severity_level);
  ASSERT flag_para_error = ’0’ REPORT msg & " : illegal flag parameter"
                             SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := CHANGE_START;
    VARIABLE r_change: std_ulogic := ’0’;
    VARIABLE r_test_expr : UNSIGNED((width-1) DOWNTO 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
```

```
        IF (rst_n = '1') THEN
          CASE (r_state) IS
            WHEN CHANGE_START =>
              IF (start_event = '1') THEN
                r_change := '0';
                r_state := CHANGE_CHECK;
                r_test_expr := test_expr;
                ii <= num_cks;
              END IF;
            WHEN CHANGE_CHECK =>
              -- Count clock ticks
              IF (start_event = '1') THEN
                CASE (flag) IS
                  WHEN FLAG_IGNORE_NEW_START =>
                    IF (ii > 0) THEN
                      ii <= ii-1;
                    END IF ;
                  WHEN FLAG_RESET_ON_START =>
                    ii <= num_cks;
                  WHEN FLAG_ERR_ON_START =>
                    flag_error <= '1';
                  WHEN OTHERS =>
                    flag_para_error <= '1';
                END CASE;
              ELSIF (ii > 0) THEN
                ii <= ii-1;
              END IF;
--      // Check that the property is true
              IF (r_test_expr /= test_expr) THEN
                r_change := '1';
                r_state  := CHANGE_START;
              END IF;
--
              -- go to start state on last check
              IF ((ii = 1) AND NOT((start_event = '1') AND
                                   (flag = FLAG_RESET_ON_START))) THEN
                r_state := CHANGE_START;
              -- Check that the property is true
                IF ((r_change = '0') AND (r_test_expr = test_expr)) THEN
                  valid <= '0';
                END IF;
              END IF;
--
              r_test_expr := test_expr;
          END CASE;
        ELSE
          r_state  := CHANGE_START;
          r_change := '0';
          valid <= '1';
          flag_error <= '0';
          flag_para_error <= '0';
          r_test_expr := test_expr;
        END IF;
```

```
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_cycle_sequence

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_cycle_sequence IS
  GENERIC (severity_level: INTEGER := 0;
           num_cks: INTEGER := 1;
           necessary_condition: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT CYCLE SEQUENCE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        event_sequence: IN UNSIGNED((num_cks-1) DOWNTO 0));
END assert_cycle_sequence;
--
ARCHITECTURE ovl OF assert_cycle_sequence IS
  CONSTANT num_cks_1: INTEGER := num_cks - 1;
  CONSTANT num_buf_1: INTEGER := (num_cks_1+1)*num_cks_1/2;

  CONSTANT nec_cond:  INTEGER := necessary_condition mod 2;
  CONSTANT non_pipe:  INTEGER := (necessary_condition-nec_cond) mod 4;

  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    CONSTANT ZERO:      UNSIGNED(num_buf_1 DOWNTO 0) := (OTHERS => ’0’);
    VARIABLE pipe_regs: UNSIGNED(num_buf_1 DOWNTO 0) := (OTHERS => ’0’);
    VARIABLE jj, nn, lim, start: INTEGER;
    VARIABLE and_res: std_ulogic;
    VARIABLE e_idx: INTEGER;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (num_cks = 1) THEN               -- same as assert_always()
        IF (event_sequence(0) = ’0’) THEN -- check error
          valid <= ’0’;
        END IF;
      ELSE
        IF (nec_cond /= 0) THEN
          IF (non_pipe /= 0) THEN -- non-pipelined
```

```
            IF (pipe_regs(1) = '0') THEN   -- INIT
              IF (event_sequence(num_cks_1) = '1') THEN
                pipe_regs(1) := '1';       -- start CHECK
                e_idx := num_cks_1 - 1;
              END IF;
            ELSE                           -- CHECK
              IF (event_sequence(e_idx) /= '1') THEN
                -- check error
                pipe_regs(1) := '0';
                valid <= '0';
              END IF;
              IF (e_idx > 0) THEN
                e_idx := e_idx - 1;
              ELSE
                pipe_regs(1) := '0';          -- done CHECK, go to INIT
              END IF;
            END IF;
          ELSE -- pipelined
            FOR ii IN num_cks_1-1 DOWNTO 1 LOOP
              IF ((pipe_regs(ii+1)='1') AND (event_sequence(num_cks_1-ii-
1) = '0')) THEN
                valid <= '0';
              END IF;
              pipe_regs(ii+1) := (pipe_regs(ii) AND
event_sequence(num_cks_1-ii));
            END LOOP;
            pipe_regs(1) := event_sequence(num_cks_1);
          END IF;
        ELSE               -- (necessary_condition==0), pipelined
          lim := num_cks_1;
          nn := 0; jj := 0; start := 0;
          FOR pp IN num_cks_1 DOWNTO lim LOOP
            nn := nn + 1;
            start := start + nn;
            jj := start;
            and_res := '1';               -- get preconditon with and_res
            FOR ii IN nn TO (pp+nn-1) LOOP
              and_res := and_res AND pipe_regs(jj);
              jj := jj + ii;
            END LOOP;
            IF ((and_res = '1') AND (event_sequence(num_cks_1 - pp) =
'0')) THEN
              -- check error
              valid <= '0';
            END IF;
          END LOOP;
          FOR ii IN 0 TO (num_buf_1 - 1) LOOP -- update pipes
            pipe_regs(ii) := pipe_regs(ii+1);
          END LOOP;
          jj := 1;
          FOR ii IN 1 TO num_cks_1 LOOP
            pipe_regs(jj) := event_sequence(ii);
            jj := jj + ii + 1;
          END LOOP;
        END IF;
      END IF;
    ELSE
      pipe_regs := ZERO;
      e_idx := num_cks_1;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_decrement

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_decrement IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           value: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT DECREMENT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_decrement;

ARCHITECTURE ovl OF assert_decrement IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL r_reset_n: std_ulogic := ’0’;
  SIGNAL r_r_reset_n: std_ulogic := ’0’;
  SIGNAL last_test_expr: UNSIGNED((width-1) DOWNTO 0);
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE temp_expr : UNSIGNED(width downto 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      r_reset_n <= ’1’;
      r_r_reset_n <= r_reset_n;
      last_test_expr <= test_expr;
-- check second clock after reset
      IF (((r_reset_n=’1’) AND (r_r_reset_n=’1’)) AND
          (last_test_expr /= test_expr)) THEN
        temp_expr := (’0’ & last_test_expr) - (’0’ & test_expr);
        IF (UNSIGNED(temp_expr(width-1 downto 0)) /= value) THEN
          valid <= ’0’;
        END IF;
      END IF;
    ELSE
      valid <= ’1’;
      r_reset_n <= ’0’;
      r_r_reset_n <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
--
-- /* {
```

assert_delta

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_delta IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 1;
           max: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT DELTA VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_delta;

ARCHITECTURE ovl OF assert_delta IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL r_reset_n: std_ulogic := ’0’;
  SIGNAL r_r_reset_n: std_ulogic := ’0’;
  SIGNAL last_test_expr: UNSIGNED((width-1) DOWNTO 0);
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--   ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE temp_expr1 : UNSIGNED(width downto 0);
    VARIABLE temp_expr2 : UNSIGNED(width downto 0);
    VARIABLE int_temp1 : INTEGER;
    VARIABLE int_temp2 : INTEGER;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      r_reset_n <= ’1’;
      r_r_reset_n <= r_reset_n;
      last_test_expr <= test_expr;
-- check second clock afer reset
      IF (((r_reset_n=’1’) AND (r_r_reset_n=’1’)) AND
          (last_test_expr /= test_expr)) THEN
        temp_expr1 := (’0’ & last_test_expr) - (’0’ & test_expr);
        temp_expr2 := (’0’ & test_expr) - (’0’ & last_test_expr);
            int_temp1 := TO_INTEGER(temp_expr1(width-1 downto 0));
        int_temp2 := TO_INTEGER(temp_expr2(width-1 downto 0));
        IF (NOT(((int_temp1>=min) AND (int_temp1<=max)) OR
                ((int_temp2>=min) AND (int_temp2<=max)))) THEN
          ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
          valid <= ’0’;
        END IF;
      END IF;
    ELSE
      valid <= ’1’;
      r_reset_n <= ’0’;
      r_r_reset_n <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
--
-- /* {
```

assert_even_parity

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_even_parity IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT EVEN PARITY VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_even_parity;

ARCHITECTURE ovl OF assert_even_parity IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      valid <= not (xorr(test_expr));
    ELSE
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
--
-- /* {
```

assert_frame

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_frame IS
  GENERIC (severity_level: INTEGER := 0;
           min_cks: INTEGER := 0;
           max_cks: INTEGER := 0;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT FRAME VIOLATION");
  PORT (clk, reset_n, start_event: IN std_ulogic;
        test_expr: IN std_ulogic);
END assert_frame;
--
ARCHITECTURE ovl OF assert_frame IS
  TYPE stateT IS (FRAME_START, FRAME_CHECK);
--
  CONSTANT FLAG_IGNORE_NEW_START: INTEGER := 0;
  CONSTANT FLAG_RESET_ON_START:   INTEGER := 1;
  CONSTANT FLAG_ERR_ON_START:     INTEGER := 2;
  CONSTANT msg_st: STRING := msg & " : illegal start event";
  CONSTANT msg_bp: STRING := msg & " : bad parameter max_cks < min_cks";
  CONSTANT START_flag:            INTEGER := flag mod 4;
  CONSTANT EDGE_flag:             INTEGER := (flag-START_flag) mod 8;
--
  SIGNAL r_test_expr:   std_ulogic := ’0’;
  SIGNAL r_state:       stateT     := FRAME_START;
  SIGNAL r_start_event: std_ulogic := ’0’;
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL start_ok  : std_ulogic := ’1’;
--
  SIGNAL ii: INTEGER := 0;
  SIGNAL rst_n: std_ulogic;
--
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT (max_cks = 0) OR (max_cks >= min_cks);
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
  ASSERT start_ok = ’1’ REPORT msg_st SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE numClks: INTEGER := 0;
  BEGIN
```

```
        WAIT UNTIL clk'EVENT AND clk = '1';
        r_start_event <= start_event;
        IF (EDGE_flag /= 0) THEN
          r_test_expr <= test_expr;
        END IF;
    --
        valid <= '1';
        start_ok <= '1';
    --
        IF (rst_n = '1') THEN
          IF (r_state = FRAME_START) THEN
            -- assert_frame() behaves like assert_implication()
            --     when min_cks==0 and max_cks==0
            IF ((min_cks=0) AND (max_cks=0)) THEN
              IF ((start_event='1') AND (test_expr='0')) THEN
                -- FAIL, it does not behave like assert_implication()
                valid <= '0';
              END IF;
            ELSIF ((r_start_event = '0') AND (start_event = '1')) THEN
              -- start_event (0->1) happened
              r_state <= FRAME_CHECK;
              ii <= 1;
            END IF;
          ELSIF (r_state = FRAME_CHECK) THEN
            -- start_event (0->1) has occurred
            -- start checking
            -- Count clock ticks
            IF ((r_start_event = '0') AND (start_event = '1')) THEN
              IF (flag = FLAG_IGNORE_NEW_START) THEN
                IF ((max_cks >0) OR (ii < min_cks)) THEN
                  ii <= ii + 1;
                END IF;
              ELSIF (flag = FLAG_RESET_ON_START) THEN
                ii <= 1;
              ELSIF (flag = FLAG_ERR_ON_START) THEN
                r_state <= FRAME_START;
                start_ok <= '0';
              END IF;
            ELSIF ((max_cks >0) OR (ii < min_cks)) THEN
              ii <= ii + 1;
            END IF;

            -- Check for (0,0), (0,M), (m,0), (m,M) conditions
            IF (min_cks = 0) THEN
              IF (max_cks = 0) THEN
                -- (0,0): (min_cks==0, max_cks==0)
                -- This condition is UN-REACHABLE!!!
                valid <= '0';
                r_state <= FRAME_START;
              ELSE -- max_cks > 0
                -- (0,M): (min_cks==0, max_cks>0)
                IF ((r_test_expr='0') AND (test_expr = '1')) THEN
                  -- OK, ckeck is done. Go to FRAME_START state for next
    check.
```

```
              r_state <= FRAME_START;
            ELSIF (ii = max_cks) THEN
              -- FAIL, test_expr does not happen at/before max_cks
              valid <= ’0’;
              r_state <= FRAME_START;
            END IF;
          END IF;
        ELSE -- min_cks>0
          IF (max_cks = 0) THEN
            -- (m,0): (min_cks>0, max_cks==0)
            IF ((r_test_expr=’0’) AND (test_expr = ’1’)) THEN
              -- FAIL, test_expr should not happen before min_cks
              valid <= ’0’;
              r_state <= FRAME_START;
            ELSIF (ii = min_cks) THEN
              -- OK, test_expr does not happen before min_cks
              r_state <= FRAME_START;
            END IF;
          ELSE -- max_cks > 0
            -- (m,M): (min_cks>0, max_cks>0)
            IF ((r_test_expr=’0’) AND (test_expr = ’1’)) THEN
              r_state <= FRAME_START;
              IF (ii < min_cks) THEN
                -- FAIL, test_expr should not happen before min_cks
                valid <= ’0’;
              END IF;-- else OK, we are done!!!
            ELSIF (ii = max_cks) THEN
              -- FAIL, test_expr does not happen at/before max_cks
              valid <= ’0’;
              r_state <= FRAME_START;
            END IF;
          END IF;-- max_cks
        END IF;-- min_cks
      END IF;-- FRAME_CHECK
    ELSE
      r_state <= FRAME_START;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_handshake

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_handshake IS
  GENERIC (severity_level: INTEGER := 0;
           min_ack_cycle: INTEGER := 0;
           max_ack_cycle: INTEGER := 0;
           req_drop: INTEGER := 0;
           deassert_count: INTEGER := 0;
           max_ack_length: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT HANDSHAKE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        req, ack: IN std_ulogic);
END assert_handshake;

--   // synopsys template
--   // ovl assertion_library

ARCHITECTURE ovl OF assert_handshake IS
  TYPE stateT IS (REQ_ACK_START, REQ_ACK_WAIT, REQ_ACK_ERR,
REQ_ACK_DEASSERT);
--
  SIGNAL error_max_ack_length:  std_ulogic := ’0’;
  SIGNAL error_multiple_req:    std_ulogic := ’0’;
  SIGNAL error_min_ack_cycle:   std_ulogic := ’0’;
  SIGNAL error_max_ack_cycle:   std_ulogic := ’0’;
  SIGNAL error_ack_without_req: std_ulogic := ’0’;
  SIGNAL error_req_drop:        std_ulogic := ’0’;
  SIGNAL error_req_deassert:    std_ulogic := ’0’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL ii, jj: INTEGER := 0;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT (error_max_ack_length = ’0’)
    REPORT msg & " : ack max length violation"
      SEVERITY ovlSevTab(severity_level);
  ASSERT (error_multiple_req = ’0’)
    REPORT msg & " : multiple req violation"
      SEVERITY ovlSevTab(severity_level);
  ASSERT (error_min_ack_cycle = ’0’)
    REPORT msg & " : ack min cycle violation"
      SEVERITY ovlSevTab(severity_level);
  ASSERT (error_max_ack_cycle = ’0’)
    REPORT msg & " : ack max cycle violation"
```

```
        SEVERITY ovlSevTab(severity_level);
    ASSERT (error_ack_without_req = ’0’)
      REPORT msg & " : ack without req violation"
        SEVERITY ovlSevTab(severity_level);
    ASSERT (error_req_drop = ’0’)
      REPORT msg & " : req drop violation"
        SEVERITY ovlSevTab(severity_level);
    ASSERT (error_req_deassert = ’0’)
      REPORT msg & " : req deassert violation"
        SEVERITY ovlSevTab(severity_level);
  --
    rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
             reset_n;
  --
    PROCESS
      VARIABLE r_state:    stateT  := REQ_ACK_START;
      VARIABLE r_req:      std_ulogic := ’0’;
      VARIABLE r_ack:      std_ulogic := ’0’;
    BEGIN
      WAIT UNTIL clk’EVENT AND clk = ’1’;
      IF (rst_n = ’1’) THEN
        CASE (r_state) IS
          WHEN REQ_ACK_START =>
              IF ((max_ack_length /= 0) AND (ack = ’1’) AND (r_ack = ’1’))
  THEN
                jj <= jj+1;
                IF (jj >= max_ack_length) THEN
                  r_state := REQ_ACK_ERR;
                  error_max_ack_length <= ’1’;
                END IF;
              END IF;
              IF (req = ’1’) THEN
                IF ((r_ack = ’1’) AND (ack = ’1’) AND (r_req = ’0’)) THEN
                  r_state := REQ_ACK_ERR;
                  error_multiple_req <= ’1’;
                ELSIF ((deassert_count /= 0) AND (r_req = ’1’) AND
                       (req = ’1’) AND (ack = ’0’)) THEN
                  r_state := REQ_ACK_DEASSERT;
                  ii <= deassert_count;
                ELSIF ((min_ack_cycle /= 0) AND (ack = ’1’) AND (r_ack =
  ’0’)) THEN
                  error_min_ack_cycle <= ’1’;
                ELSIF (ack = ’0’) THEN
                  r_state := REQ_ACK_WAIT;
                  ii <= 1;
                  jj <= 0;
                END IF;
              ELSIF ((ack = ’1’) AND (r_ack = ’0’)) THEN
                r_state := REQ_ACK_ERR;
                error_ack_without_req <= ’1’;
              END IF;
          WHEN REQ_ACK_WAIT =>
            ii <= ii + 1;
            IF (ack = ’1’) THEN
              r_state := REQ_ACK_START;
              jj <= 1;
            END IF;
  --
            IF ((min_ack_cycle /= 0) AND (ii < min_ack_cycle) AND (ack =
  '1')) THEN
              r_state := REQ_ACK_ERR;
              error_min_ack_cycle <= '1';
            ELSIF ((ack /= '1') AND (max_ack_cycle /= 0) AND (ii >=
  max_ack_cycle)) THEN
              r_state := REQ_ACK_ERR;
              error_max_ack_cycle <= '1';
            ELSIF ((req_drop = 1) AND (req = '0')) THEN
              r_state := REQ_ACK_ERR;
              error_req_drop <= '1';
            ELSIF ((req = '1') AND (r_req = '0')) THEN
              r_state := REQ_ACK_ERR;
              error_multiple_req <= '1';
            END IF;
          WHEN REQ_ACK_ERR =>
            IF ((req = '1') AND (ack = '0') AND (r_req = '0')) THEN
              r_state := REQ_ACK_WAIT;
              ii <= 0;
              jj <= 0;
            ELSIF ((ack = '0') AND (r_ack = '1')) THEN
              r_state := REQ_ACK_START;
              ii <= 0;
              jj <= 0;
            END IF;
          WHEN REQ_ACK_DEASSERT =>
            ii <= ii-1;
            IF (ii = 1) THEN
              IF (req = '1') THEN
                r_state := REQ_ACK_ERR;
                error_req_deassert <= '1';
              ELSE
                r_state := REQ_ACK_START;
              END IF;
            END IF;
        END CASE;
        r_ack := ack;
        r_req := req;
      ELSE
        r_state := REQ_ACK_START;
        r_ack := '0';
        r_req := '0';
  --
        ii <= 0;
        jj <= 0;
  --
        error_max_ack_length  <= '0';
        error_multiple_req    <= '0';
        error_min_ack_cycle   <= '0';
        error_max_ack_cycle   <= '0';
        error_ack_without_req <= ’0’;
        error_req_drop        <= ’0’;
        error_req_deassert    <= ’0’;
      END IF;
    END PROCESS;
  -- synopsys translate_on
  END ovl;
```

assert_implication

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_implication IS
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT IMPLICATION VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        antecendent_expr, consequent_expr: IN std_ulogic);
END assert_implication;
--
ARCHITECTURE ovl OF assert_implication IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF NOT ((antecendent_expr = ’0’) OR (consequent_expr = ’1’)) THEN
        valid <= ’0’;
      END IF;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_increment

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_increment IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           value: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT INCREMENT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_increment;

ARCHITECTURE ovl OF assert_increment IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL r_reset_n: std_ulogic := ’0’;
  SIGNAL r_r_reset_n: std_ulogic := ’0’;
  SIGNAL last_test_expr: UNSIGNED((width-1) DOWNTO 0);
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE temp_expr : UNSIGNED(width downto 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      r_reset_n <= ’1’;
      r_r_reset_n <= r_reset_n;
      last_test_expr <= test_expr;
-- check second clock afer reset
      IF (((r_reset_n AND r_r_reset_n) = ’1’) AND
          (last_test_expr /= test_expr)) THEN
        temp_expr := (’0’ & test_expr) - (’0’ & last_test_expr);
        IF (UNSIGNED(temp_expr(width-1 downto 0)) /= value) THEN
          valid <= ’0’;
        END IF;
      END IF;
    ELSE
      valid <= ’1’;
      r_reset_n <= ’0’;
      r_r_reset_n <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_never

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_never IS
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NEVER VIOLATION");
  PORT (clk, reset_n, test_expr: IN std_ulogic);
END assert_never;

ARCHITECTURE ovl OF assert_never IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (test_expr = ’1’) THEN
        valid <= ’0’;
      END IF;
    ELSE
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_next

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_next IS
  GENERIC (severity_level: INTEGER := 0;
           num_cks: INTEGER := 1;
           check_overlapping: std_ulogic := ’1’;
           only_if: std_ulogic := ’0’;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NEXT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN std_ulogic);
END assert_next;

ARCHITECTURE ovl OF assert_next IS
  CONSTANT msg_ov: STRING := msg & " : illegal overlapping condition
detected";
  CONSTANT msg_st: STRING := msg & " : test_expr without start_event";
  CONSTANT msg_te: STRING := msg & " : start_event without test_expr";
--
  CONSTANT ZERO : UNSIGNED((num_cks-1) DOWNTO 0) := (OTHERS => ’0’);
  CONSTANT ONE :  UNSIGNED((num_cks-1) DOWNTO 0) := ZERO + 1;
--
  SIGNAL monitor : UNSIGNED((num_cks-1) DOWNTO 0) := (OTHERS => ’0’);
  SIGNAL monitor_1 : UNSIGNED((num_cks-1) DOWNTO 0) := (OTHERS => ’0’);
  SIGNAL overlap_err, start_error, test_error : std_ulogic := ’0’;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT overlap_err = ’0’ REPORT msg_ov SEVERITY
ovlSevTab(severity_level);
  ASSERT start_error = ’0’ REPORT msg_st SEVERITY
ovlSevTab(severity_level);
  ASSERT test_error  = ’0’ REPORT msg_te SEVERITY
ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
--  parameter only_if=0;  -- if 1, test_expr can only appear if a
corresponding
--                        -- start_event occurs
--
```

```
  monitor_1 <= SHIFT_LEFT(monitor, 1);
--
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    valid <= '1';

    overlap_err <= '0';
    start_error <= '0';
    test_error  <= '0';

    IF (rst_n = '1') THEN
      if (start_event = '1') THEN
        monitor <= UNSIGNED(std_logic_vector(monitor_1) OR
std_logic_vector(ONE));
      ELSE
        monitor <= monitor_1;
      END IF;
      IF ((check_overlapping = '0') AND (monitor_1 /= ZERO)
                              AND (start_event = '1')) THEN
        overlap_err <= '1';
      ELSIF (only_if = '1') THEN
        IF ((monitor(num_cks-1) = '0') AND (test_expr = '1')) THEN
          start_error <= '1';
        END IF;
      ELSIF ((monitor(num_cks-1) = '1') AND (test_expr = '0')) THEN
        test_error <= '1';
      END IF;
    ELSE
      monitor <= ZERO;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_no_overflow

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;
use std.textio.all;
use ieee.std_logic_textio.all;


ENTITY assert_no_overflow IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 0;
           max: INTEGER := -1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NO OVERFLOW VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_no_overflow;

ARCHITECTURE ovl OF assert_no_overflow IS
  TYPE stateT IS (OVERFLOW_START, OVERFLOW_CHECK);
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL pmax : INTEGER;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  pmax <= max when (max > 0) else
              (2**width -1);

  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := OVERFLOW_START;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (r_state = OVERFLOW_START) THEN
        IF (test_expr = pmax) THEN
          r_state := OVERFLOW_CHECK;
        END IF;
      ELSIF (r_state = OVERFLOW_CHECK) THEN
        IF (test_expr /= pmax) THEN
          r_state := OVERFLOW_START;
          IF ((test_expr <= min) OR (test_expr > pmax)) THEN
```

```
            valid <= ’0’;
          END IF;
        END IF;
      END IF;
    ELSE
      r_state := OVERFLOW_START;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_no_transition

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_no_transition IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NO TRANSITION VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        start_state: IN UNSIGNED((width-1) DOWNTO 0);
        next_state: IN UNSIGNED((width-1) DOWNTO 0));

END assert_no_transition;

ARCHITECTURE ovl OF assert_no_transition IS
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL assert_state  : std_ulogic := ’0’;
  SIGNAL r_next_state  : UNSIGNED((width-1) DOWNTO 0);
  SIGNAL r_start_state : UNSIGNED((width-1) DOWNTO 0);
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      IF (assert_state = ’0’) THEN -- START
        IF (test_expr = start_state) THEN
          assert_state  <= ’1’;
          r_start_state <= start_state;
          r_next_state  <= next_state;
        END IF;
      ELSE -- (assert_state = ’1’) -- CHECK
        IF (test_expr = r_next_state) THEN
           valid <= ’0’;
           assert_state  <= ’0’; -- done error.
        ELSIF (test_expr /= r_start_state) THEN
           assert_state <= ’0’; -- done ok.
        END IF;
      END IF;
    ELSE
      assert_state  <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
--
-- /* {
```

assert_no_underflow

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_no_underflow IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 0;
           max: INTEGER := -1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NO UNDERFLOW VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_no_underflow;

ARCHITECTURE ovl OF assert_no_underflow IS
  TYPE stateT IS (UNDERFLOW_START, UNDERFLOW_CHECK);
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL pmax : INTEGER;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
  pmax <= max when (max > 0) else
            (2**width -1);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := UNDERFLOW_START;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (r_state = UNDERFLOW_START) THEN
        IF (test_expr = min) THEN
          r_state := UNDERFLOW_CHECK;
        END IF;
      ELSIF (r_state = UNDERFLOW_CHECK) THEN
        IF (test_expr /= min) THEN
          r_state := UNDERFLOW_START;
          IF ((test_expr >= pmax) OR (test_expr < min)) THEN
            valid <= ’0’;
          END IF;
        END IF;
      END IF;
    ELSE
      r_state := UNDERFLOW_START;
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;

-- } */
--
-- /* {
```

assert_odd_parity

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_odd_parity IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ODD PARITY VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_odd_parity;

ARCHITECTURE ovl OF assert_odd_parity IS
  SIGNAL valid: std_ulogic := '1';
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = '1' REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = '1') else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    valid <= '1';
    IF (rst_n = '1') THEN
      valid <= xorr(test_expr);
    ELSE
      valid <= '1';
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
--
-- /* {
```

assert_one_cold

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_one_cold IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           inactive: INTEGER := 2;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ONE COLD VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_one_cold;
--
ARCHITECTURE ovl OF assert_one_cold IS
  CONSTANT msg_hasx: STRING := msg & " : Error: test_expr contains X/Z
value";
  CONSTANT ZERO : UNSIGNED((width-1) DOWNTO 0) := (OTHERS => ’0’);
  CONSTANT ONES : UNSIGNED((width-1) DOWNTO 0) := (OTHERS => ’1’);
  CONSTANT ONE :  UNSIGNED((width-1) DOWNTO 0) := ZERO + 1;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
--
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
--  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE test_expr_i: UNSIGNED((width-1) DOWNTO 0);
    VARIABLE test_expr_i_1: UNSIGNED((width-1) DOWNTO 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;

    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (has_x(test_expr)) THEN
        ASSERT false REPORT msg_hasx SEVERITY ovlSevTab(severity_level);
      ELSE
--
        test_expr_i   := UNSIGNED(NOT std_logic_vector(test_expr));
        test_expr_i_1 := test_expr_i - ONE;
--
        IF (test_expr = ZERO) THEN
```

```
          IF (inactive /= 0) THEN
            valid <= ’0’;
            ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
          END IF;
        ELSIF (test_expr = ONES) THEN
          IF (inactive /= 1) THEN
            valid <= ’0’;
            ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
          END IF;
        ELSIF (UNSIGNED(std_logic_vector(test_expr_i) AND
               std_logic_vector(test_expr_i_1)) /= ZERO) THEN
          valid <= ’0’;
          ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
        END IF;
      END IF;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_one_hot

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_one_hot IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ONE HOT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_one_hot;

ARCHITECTURE ovl OF assert_one_hot IS
  CONSTANT msg_hasx: STRING := msg & " : Error: test_expr contains X/Z
value";
  CONSTANT ZERO : UNSIGNED((width-1) DOWNTO 0) := (OTHERS => '0');
--
  SIGNAL valid: std_ulogic := '1';
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
-- ASSERT valid = '1' REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = '1') else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';

    valid <= '1';
    IF (rst_n = '1') THEN
      IF (has_x(test_expr)) THEN
        ASSERT false REPORT msg_hasx SEVERITY ovlSevTab(severity_level);
      ELSIF (test_expr = ZERO) THEN
        valid <= '0';
        ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
      ELSIF (((test_expr) AND (test_expr - 1)) /= (ZERO)) THEN
        valid <= '0';
        ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
      END IF;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
```

```
--
-- /* {
```

assert_proposition

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_proposition IS
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT PROPOSITION VIOLATION");
  PORT (reset_n, test_expr: IN std_ulogic);
END assert_proposition;

ARCHITECTURE ovl OF assert_proposition IS
  SIGNAL proposition: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT proposition = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS (test_expr)
  BEGIN
    IF (rst_n = ’1’) THEN
      IF (test_expr = ’1’) THEN
        proposition <= ’1’;
      ELSE
        proposition <= ’0’;
      END IF;
    ELSE
      proposition <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
-- } */
--
-- /* {
```

assert_quiescent_state

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_quiescent_state IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT QUIESCENT_STATE VIOLATION";
           ASSERT_END_OF_SIMULATION: std_ulogic := ’0’);
  PORT (clk, reset_n: IN std_ulogic;
        state_expr, check_value: IN UNSIGNED((width-1) DOWNTO 0);
        sample_event: IN std_ulogic);
END assert_quiescent_state;

ARCHITECTURE ovl OF assert_quiescent_state IS
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
--
  SIGNAL r_sample_event: std_ulogic := ’0’;
  SIGNAL r_EOS         : std_ulogic := ’0’;
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    r_sample_event <= sample_event;
    IF (ASSERT_END_OF_SIMULATION = ’1’) THEN
      r_EOS <= ovl_END_OF_SIMULATION_SIGNAL;
    END IF;
    IF (rst_n = ’1’) THEN
      IF ((r_sample_event = ’0’) AND (sample_event = ’1’) AND
                               (state_expr /= check_value)) THEN
        valid <= ’0’;
      ELSIF ((ASSERT_END_OF_SIMULATION = ’1’) AND (r_EOS = ’0’) AND
                            (ovl_END_OF_SIMULATION_SIGNAL = ’1’))
THEN
        valid <= ’0’;
      END IF;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_range

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_range IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 1;
           max: INTEGER := -1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT RANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_range;

ARCHITECTURE ovl OF assert_range IS
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL pmax : INTEGER;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  pmax <= max when (max > 0) else
              (2**width -1);
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    IF (rst_n = ’1’) THEN
      IF (test_expr < min) THEN
        valid <= ’0’;
        ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
      ELSIF (test_expr > pmax) THEN
        valid <= ’0’;
        ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
      ELSE
        valid <= ’1’;
      END IF;
    ELSE
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
```

```
END ovl;
-- } */
--
-- /* {
```

assert_time

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_time IS
  GENERIC (severity_level: INTEGER := 0;
           num_cks: INTEGER := 1;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT TIME VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN std_ulogic);
END assert_time;

ARCHITECTURE ovl OF assert_time IS
  TYPE stateT IS (TIME_START, TIME_CHECK);
  CONSTANT FLAG_IGNORE_NEW_START: INTEGER := 0;
  CONSTANT FLAG_RESET_ON_START  : INTEGER := 1;
  CONSTANT FLAG_ERR_ON_START    : INTEGER := 2;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL flag_error: std_ulogic := ’0’;
  SIGNAL flag_para_error: std_ulogic := ’0’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL ii: INTEGER;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--   ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
  ASSERT flag_error = ’0’ REPORT msg & " : illegal start event"
                               SEVERITY ovlSevTab(severity_level);
  ASSERT flag_para_error = ’0’ REPORT msg & " : illegal flag parameter"
                             SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := TIME_START;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      CASE (r_state) IS
        WHEN TIME_START =>
```

```
          IF (start_event = ’1’) THEN
            r_state := TIME_CHECK;
            ii <= num_cks;
          END IF;
        WHEN TIME_CHECK =>
          -- Count clock ticks
          IF (start_event = ’1’) THEN
            CASE (flag) IS
              WHEN FLAG_IGNORE_NEW_START =>
                IF (ii > 0) THEN
                  ii <= ii-1;
                END IF ;
              WHEN FLAG_RESET_ON_START =>
                ii <= num_cks;
              WHEN FLAG_ERR_ON_START =>
                flag_error <= ’1’;
              WHEN OTHERS =>
                flag_para_error <= ’1’;
            END CASE;
          ELSIF (ii > 0) THEN
            ii <= ii-1;
          END IF;
--  // Check that the property is true
          IF (test_expr /= ’1’) THEN
            valid <= ’0’;
            ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
          END IF;
--
          -- go to start state on last check
          IF ((ii = 1) AND NOT((start_event = ’1’) AND
                               (flag = FLAG_RESET_ON_START))) THEN
            r_state := TIME_START;
          END IF;
--
      END CASE;
    ELSE
      r_state  := TIME_START;
      valid <= ’1’;
      flag_error <= ’0’;
      flag_para_error <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_transition

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_transition IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT TRANSITION VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        start_state: IN UNSIGNED((width-1) DOWNTO 0);
        next_state: IN UNSIGNED((width-1) DOWNTO 0));

END assert_transition;

ARCHITECTURE ovl OF assert_transition IS
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL assert_state  : std_ulogic := ’0’;
  SIGNAL r_start_state : UNSIGNED((width-1) DOWNTO 0);
  SIGNAL r_next_state  : UNSIGNED((width-1) DOWNTO 0);
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      IF (assert_state = ’0’) THEN -- START
        IF (test_expr = start_state) THEN
          assert_state  <= ’1’;
          r_start_state <= start_state;
          r_next_state  <= next_state;
        END IF;
      ELSE -- (assert_state = ’1’) -- CHECK
          IF (test_expr = r_next_state) THEN
            assert_state <= ’0’; -- done OK.
          ELSIF (test_expr /= r_start_state) THEN
```

```
            valid <= ’0’;
            assert_state <= ’0’; -- done ERROR.
          END IF;
      END IF;
    ELSE
      assert_state <= ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
--
-- /* {
```

assert_unchange

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;
use std.textio.all;
use ieee.std_logic_textio.all;

ENTITY assert_unchange IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           num_cks: INTEGER := 1;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT UNCHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_unchange;

ARCHITECTURE ovl OF assert_unchange IS
  TYPE stateT IS (UNCHANGE_START, UNCHANGE_CHECK);
  CONSTANT FLAG_IGNORE_NEW_START: INTEGER := 0;
  CONSTANT FLAG_RESET_ON_START  : INTEGER := 1;
  CONSTANT FLAG_ERR_ON_START    : INTEGER := 2;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL flag_error: std_ulogic := ’0’;
  SIGNAL flag_para_error: std_ulogic := ’0’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL ii: INTEGER;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
  ASSERT flag_error = ’0’ REPORT msg & " : illegal start event"
                               SEVERITY ovlSevTab(severity_level);
  ASSERT flag_para_error = ’0’ REPORT msg & " : illegal flag parameter"
                             SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := UNCHANGE_START;
    VARIABLE r_change: std_ulogic := ’0’;
    VARIABLE r_test_expr : UNSIGNED((width-1) DOWNTO 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
--
    IF (rst_n = ’1’) THEN
      CASE (r_state) IS
        WHEN UNCHANGE_START =>
          IF (start_event = ’1’) THEN
            r_change := ’0’;
            r_state := UNCHANGE_CHECK;
            r_test_expr := test_expr;
            ii <= num_cks;
          END IF;
        WHEN UNCHANGE_CHECK =>
          -- Count clock ticks
          IF (start_event = ’1’) THEN
            CASE (flag) IS
              WHEN FLAG_IGNORE_NEW_START =>
                IF (ii > 0) THEN
                  ii <= ii-1;
                END IF ;
              WHEN FLAG_RESET_ON_START =>
                ii <= num_cks;
              WHEN FLAG_ERR_ON_START =>
                flag_error <= ’1’;
              WHEN OTHERS =>
                flag_para_error <= ’1’;
            END CASE;
          ELSIF (ii > 0) THEN
            ii <= ii-1;
          END IF;
--  // Check that the property is true
          IF (r_test_expr /= test_expr) THEN
            r_change := ’1’;
          END IF;
--
          -- go to start state on last check
          IF ((ii = 1) AND NOT((start_event = ’1’) AND
                               (flag = FLAG_RESET_ON_START))) THEN
            r_state := UNCHANGE_START;
          END IF;
            -- Check that the property is true
            IF ((r_change = ’1’) OR (r_test_expr /= test_expr)) THEN
              valid <= ’0’;
              ASSERT false  REPORT msg SEVERITY
ovlSevTab(severity_level);
            END IF;
--
          r_test_expr := test_expr;
      END CASE;
    ELSE
      r_state  := UNCHANGE_START;
      r_change := ’0’;
      valid <= ’1’;
      flag_error <= ’0’;
      flag_para_error <= ’0’;
      r_test_expr := test_expr;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_width

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_width IS
  GENERIC (severity_level: INTEGER := 0;
           min_cks, max_cks: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIDTH VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN std_ulogic);
END assert_width;
--
ARCHITECTURE ovl OF assert_width IS
  TYPE stateT IS (WIDTH_START, WIDTH_CKMIN, WIDTH_CKMAX, WIDTH_IDLE);
--
  SIGNAL r_test_expr: std_ulogic := ’0’;
  SIGNAL r_state: stateT := WIDTH_START;
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
  SIGNAL num_cks : INTEGER := 0;
  SIGNAL ii: INTEGER;
BEGIN
-- synopsys translate_off
--  // synopsys template
--  // verplex assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE min: INTEGER := min_cks;
    VARIABLE max: INTEGER := max_cks;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;
    r_test_expr <= test_expr;
    IF (rst_n = ’1’) THEN
      IF (r_state = WIDTH_START) THEN
        IF ((r_test_expr = ’0’) AND (test_expr = ’1’)) THEN
          num_cks <= 1;
          IF (min_cks > 0) THEN -- any max range legal
             r_state <= WIDTH_CKMIN;
          ELSIF (max_cks >0) THEN
             r_state <= WIDTH_CKMAX;
          END IF;
        END IF;
      ELSIF (r_state = WIDTH_CKMIN) THEN
        IF (test_expr = ’1’) THEN
          num_cks <= num_cks + 1;
            IF (num_cks >= min_cks) THEN
              IF (max_cks > 0) THEN
                r_state <= WIDTH_CKMAX;
              ELSE
                r_state <= WIDTH_IDLE;
              END IF;
            END IF;
        ELSE
          IF (num_cks < min_cks) THEN
            valid <= ’0’;  --error
          END IF;
          r_state <= WIDTH_START;
        END IF;
      ELSIF (r_state = WIDTH_CKMAX) THEN
        IF (test_expr = ’1’) THEN
          num_cks <= num_cks + 1;
          IF (num_cks >= max_cks) THEN
            valid <= ’0’;
            r_state <= WIDTH_IDLE;
          END IF;
        ELSE
          IF (num_cks > max_cks) THEN
            valid <= ’0’;
          END IF;
          r_state <= WIDTH_START;
        END IF;
      ELSIF (r_state = WIDTH_IDLE) THEN
        IF (test_expr = ’0’) THEN
          r_state <= WIDTH_START;
        END IF;
      END IF;
    ELSE
      r_state <= WIDTH_START;
      r_test_expr <= ’0’;
      num_cks <= 0;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_win_change

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_win_change IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIN CHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        end_event: IN std_ulogic);
END assert_win_change;

ARCHITECTURE ovl OF assert_win_change IS
  TYPE stateT IS (WIN_CHANGE_START, WIN_CHANGE_CHECK);
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := WIN_CHANGE_START;
    VARIABLE r_change: std_ulogic := ’0’;
    VARIABLE r_test_expr : UNSIGNED((width-1) DOWNTO 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      CASE (r_state) IS
        WHEN WIN_CHANGE_START =>
          IF (start_event = ’1’) THEN
            r_change := ’0’;
            r_state := WIN_CHANGE_CHECK;
            r_test_expr := test_expr;
          END IF;
        WHEN WIN_CHANGE_CHECK =>
          -- Count clock ticks
          IF (r_test_expr /= test_expr) THEN
```

```
            r_change := ’1’;
--          r_state := WIN_CHANGE_START;
          END IF;

          IF (end_event = ’1’) THEN
            r_state := WIN_CHANGE_START;
              -- Check that the property is true
            IF ((r_change = ’0’) AND (r_test_expr = test_expr)) THEN
              valid <= ’0’;
            END IF;
          END IF;
          r_test_expr := test_expr;
      END CASE;
    ELSE
      r_state  := WIN_CHANGE_START;
      r_change := ’0’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
```

assert_win_unchange

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_win_unchange IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIN UNCHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        end_event: IN std_ulogic);
END assert_win_unchange;

ARCHITECTURE ovl OF assert_win_unchange IS
  TYPE stateT IS (WIN_UNCHANGE_START, WIN_UNCHANGE_CHECK);
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := WIN_UNCHANGE_START;
    VARIABLE r_unchange: std_ulogic := ’1’;
    VARIABLE r_test_expr : UNSIGNED((width-1) DOWNTO 0);
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      CASE (r_state) IS
        WHEN WIN_UNCHANGE_START =>
          IF (start_event = ’1’) THEN
            r_unchange := ’1’;
            r_state := WIN_UNCHANGE_CHECK;
            r_test_expr := test_expr;
          END IF;
        WHEN WIN_UNCHANGE_CHECK =>
          -- Count clock ticks
          IF (r_test_expr /= test_expr) THEN
```

```
              r_unchange := ’0’;
            END IF;

            IF (end_event = ’1’) THEN
              r_state := WIN_UNCHANGE_START;
            END IF;
                -- Check that the property is true
            IF ((r_unchange = ’0’) OR (r_test_expr /= test_expr)) THEN
              valid <= ’0’;
              ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
            END IF;
            r_test_expr := test_expr;
        END CASE;
      ELSE
        r_state  := WIN_UNCHANGE_START;
        r_unchange := ’1’;
        r_test_expr := test_expr;
      END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
--
-- /* {
```

assert_window

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_window IS
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WINDOW VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN std_ulogic;
        end_event: IN std_ulogic);
END assert_window;

ARCHITECTURE ovl OF assert_window IS
  TYPE stateT IS (WINDOW_START, WINDOW_CHECK);
--
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
--
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
--  ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
    VARIABLE r_state : stateT    := WINDOW_START;
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;
    valid <= ’1’;

    IF (rst_n = ’1’) THEN
      CASE (r_state) IS
        WHEN WINDOW_START =>
          IF (start_event = ’1’) THEN
            r_state := WINDOW_CHECK;
          END IF;
        WHEN WINDOW_CHECK =>
          IF (end_event = ’1’) THEN
            r_state := WINDOW_START;
          END IF;
          IF (test_expr /= ’1’) THEN
            valid <= ’0’;
            ASSERT false  REPORT msg SEVERITY ovlSevTab(severity_level);
          END IF;
      END CASE;
    ELSE
      r_state  := WINDOW_START;
      valid <= ’1’;
    END IF;
  END PROCESS;
-- synopsys translate_on
END ovl;
--
-- /* {
```

assert_zero_one_hot

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;
USE work.ovl_assertlib.ALL;

ENTITY assert_zero_one_hot IS
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ZERO ONE HOT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END assert_zero_one_hot;

ARCHITECTURE ovl OF assert_zero_one_hot IS
  CONSTANT msg_hasx: STRING := msg & " : Error: test_expr contains X/Z
value";
  SIGNAL valid: std_ulogic := ’1’;
  SIGNAL rst_n: std_ulogic;
BEGIN
-- synopsys translate_off
--   // synopsys template
--   // ovl assertion_library
--
-- ASSERT valid = ’1’ REPORT msg SEVERITY ovlSevTab(severity_level);
--
  rst_n <= ovl_reset_n when (ovl_reset_n_enable = ’1’) else
           reset_n;
--
  PROCESS
  BEGIN
    WAIT UNTIL clk’EVENT AND clk = ’1’;

    IF (rst_n = ’1’) THEN
      IF (has_x(test_expr)) THEN
        ASSERT false REPORT msg_hasx SEVERITY ovlSevTab(severity_level);
      ELSE
        IF (((test_expr) = TO_UNSIGNED(0,width)) OR
           (((test_expr) AND (test_expr - 1)) = TO_UNSIGNED(0,width)))
THEN
          valid <= ’1’;
        ELSE
          valid <= ’0’;
          ASSERT false REPORT msg SEVERITY ovlSevTab(severity_level);
        END IF;
      END IF;
    ELSE
      valid <= ’1’;
    END IF;

  END PROCESS;
```

```
-- synopsys translate_on
END ovl;
```

VHDL Package

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;

--
-- Make a OVL package for global reset signal
--
PACKAGE ovl_assertlib IS
--
  FUNCTION ovlSevTab ( idx: INTEGER) return severity_level;
--
  SIGNAL ovl_reset_n: std_ulogic := ’1’;
  SIGNAL ovl_reset_n_enable: std_ulogic := ’0’;
  SIGNAL ovl_end_of_simulation_signal: std_ulogic := ’0’;
--
  FUNCTION xorr ( V: unsigned) return std_ulogic;
  FUNCTION to_std ( V: boolean) return std_ulogic;
  FUNCTION stdv_to_unsigned ( V: std_logic_vector) return unsigned;
  FUNCTION unsigned_to_stdv ( V: unsigned) return std_logic_vector;
--
  FUNCTION has_x (A : unsigned) RETURN boolean;
--
COMPONENT assert_always
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ALWAYS VIOLATION");
  PORT (clk, reset_n, test_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_change
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           num_cks: INTEGER := 1;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT CHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_next
  GENERIC (severity_level: INTEGER := 0;
           num_cks: INTEGER := 1;
           check_overlapping: INTEGER := 1;
           only_if: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NEXT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN std_ulogic);
```

```
END COMPONENT;
--
COMPONENT assert_decrement
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           value: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT DECREMENT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_delta
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 1;
           max: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT DELTA VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_even_parity
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT EVEN PARITY VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_frame
  GENERIC (severity_level: INTEGER := 0;
           min_cks: INTEGER := 0;
           max_cks: INTEGER := 0;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT FRAME VIOLATION");
  PORT (clk, reset_n, start_event: IN std_ulogic;
        test_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_handshake
  GENERIC (severity_level: INTEGER := 0;
           min_ack_cycle: INTEGER := 0;
           max_ack_cycle: INTEGER := 0;
           req_drop: INTEGER := 0;
           deassert_count: INTEGER := 0;
           max_ack_length: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT HANDSHAKE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        req, ack: IN std_ulogic);
```

```
END COMPONENT;
--
COMPONENT assert_implication
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT IMPLICATION VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        antecendent_expr, consequent_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_increment
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           value: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT INCREMENT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_never
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NEVER VIOLATION");
  PORT (clk, reset_n, test_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_no_overflow
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 0;
           max: INTEGER := -1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NO OVERFLOW VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_no_transition
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT NO TRANSITION VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        start_state: IN UNSIGNED((width-1) DOWNTO 0);
        next_state: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_no_underflow
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           min: INTEGER := 0;
           max: INTEGER := -1;
```

```
              options: INTEGER := 0;
              msg: STRING := "ASSERT NO UNDERFLOW VIOLATION");
    PORT (clk, reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_odd_parity
    GENERIC (severity_level: INTEGER := 0;
              width: INTEGER := 1;
              options: INTEGER := 0;
              msg: STRING := "ASSERT ODD PARITY VIOLATION");
    PORT (clk, reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_proposition
    GENERIC (severity_level: INTEGER := 0;
              options: INTEGER := 0;
              msg: STRING := "ASSERT PROPOSITION VIOLATION");
    PORT (reset_n, test_expr: IN std_ulogic);
  END COMPONENT;
  --
  COMPONENT assert_quiescent_state
    GENERIC (severity_level: INTEGER := 0;
              width: INTEGER := 1;
              options: INTEGER := 0;
              msg: STRING := "ASSERT QUIESCENT_STATE VIOLATION";
              ASSERT_END_OF_SIMULATION: std_ulogic := ’0’);
    PORT (clk, reset_n: IN std_ulogic;
          state_expr, check_value: IN UNSIGNED((width-1) DOWNTO 0);
          sample_event: IN std_ulogic);
  END COMPONENT;
  --
  COMPONENT assert_prop_one_hot
    GENERIC (severity_level: INTEGER := 0;
              width: INTEGER := 32;
              inactive: INTEGER := 2;
              options: INTEGER := 0;
              msg: STRING := "ASSERT PROPOSITIONAL ONE HOT VIOLATION");
    PORT (reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_prop_one_cold
    GENERIC (severity_level: INTEGER := 0;
              width: INTEGER := 32;
              inactive: INTEGER := 2;
              options: INTEGER := 0;
              msg: STRING := "ASSERT PROP ONE COLD VIOLATION");
    PORT (reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_prop_one_one_cold
```

```
    GENERIC (severity_level: INTEGER := 0;
             width: INTEGER := 32;
             inactive: INTEGER := 2;
             options: INTEGER := 0;
             msg: STRING := "ASSERT PROPOSITIONAL ONE ONE COLD VIOLATION");
    PORT (reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_prop_zero_one_hot
    GENERIC (severity_level: INTEGER := 0;
             width: INTEGER := 32;
             options: INTEGER := 0;
             msg: STRING := "ASSERT PROPOSITIONAL ZERO ONE HOT VIOLATION");
    PORT (reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_range
    GENERIC (severity_level: INTEGER := 0;
             width: INTEGER := 1;
             min: INTEGER := 1;
             max: INTEGER := -1;
             options: INTEGER := 0;
             msg: STRING := "ASSERT RANGE VIOLATION");
    PORT (clk, reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_time
    GENERIC (severity_level: INTEGER := 0;
             num_cks: INTEGER := 1;
             flag: INTEGER := 0;
             options: INTEGER := 0;
             msg: STRING := "ASSERT TIME VIOLATION");
    PORT (clk, reset_n: IN std_ulogic;
          start_event: IN std_ulogic;
          test_expr: IN std_ulogic);
  END COMPONENT;
  --
  COMPONENT assert_transition
    GENERIC (severity_level: INTEGER := 0;
             width: INTEGER := 1;
             options: INTEGER := 0;
             msg: STRING := "ASSERT TRANSITION VIOLATION");
    PORT (clk, reset_n: IN std_ulogic;
          test_expr: IN UNSIGNED((width-1) DOWNTO 0);
          start_state: IN UNSIGNED((width-1) DOWNTO 0);
          next_state: IN UNSIGNED((width-1) DOWNTO 0));
  END COMPONENT;
  --
  COMPONENT assert_unchange
    GENERIC (severity_level: INTEGER := 0;
             width: INTEGER := 1;
```

```
           num_cks: INTEGER := 1;
           flag: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT UNCHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_win_change
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIN CHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        end_event: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_win_unchange
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIN UNCHANGE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0);
        end_event: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_window
  GENERIC (severity_level: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WINDOW VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        start_event: IN std_ulogic;
        test_expr: IN std_ulogic;
        end_event: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_one_cold
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           inactive: INTEGER := 2;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ONE COLD VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_one_hot
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ONE HOT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_zero_one_hot
  GENERIC (severity_level: INTEGER := 0;
           width: INTEGER := 32;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ZERO ONE HOT VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN UNSIGNED((width-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_cycle_sequence
  GENERIC (severity_level: INTEGER := 0;
           num_cks: INTEGER := 1;
           necessary_condition: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT CYCLE SEQUENCE VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        event_sequence: IN UNSIGNED((num_cks-1) DOWNTO 0));
END COMPONENT;
--
COMPONENT assert_width
  GENERIC (severity_level: INTEGER := 0;
           min_cks, max_cks: INTEGER := 1;
           options: INTEGER := 0;
           msg: STRING := "ASSERT WIDTH VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        test_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_always_on_edge
  GENERIC (severity_level: INTEGER := 0;
           edge_type: INTEGER := 0;
           options: INTEGER := 0;
           msg: STRING := "ASSERT ASSERT ALWAYS ON EDGE VIOLATION");
  PORT (clk, reset_n, sampling_event, test_expr: IN std_ulogic);
END COMPONENT;
--
COMPONENT assert_fifo_index
  GENERIC (severity_level: INTEGER := 0;
           depth         : INTEGER := 1;
           push_width    : INTEGER := 1;
           pop_width     : INTEGER := 1;
           options       : INTEGER := 0;
           msg           : STRING := "ASSERT FIFO INDEX VIOLATION");
  PORT (clk, reset_n: IN std_ulogic;
        push        : IN UNSIGNED((push_width-1) DOWNTO 0);
        pop         : IN UNSIGNED((pop_width-1)  DOWNTO 0));
END COMPONENT;
--
```

```
END ovl_assertlib ;


PACKAGE body ovl_assertlib IS

  FUNCTION ovlSevTab ( idx: INTEGER) return severity_level IS
  BEGIN
    CASE (idx) IS
      WHEN 0 => RETURN failure;
      WHEN 1 => RETURN error;
      WHEN 2 => RETURN warning;
      WHEN 3 => RETURN note;
      WHEN OTHERS =>
               ASSERT false REPORT "In ovlSevTab(idx): idx not in 0..3"
SEVERITY error;
               RETURN failure;
    END CASE;
  END ovlSevTab;

  FUNCTION to_std ( V: boolean) return std_ulogic IS
  BEGIN
    IF (V) THEN
      RETURN ’1’;
    ELSE
      RETURN ’0’;
    END IF;
  END to_std;

  FUNCTION xorr ( V: unsigned) return std_ulogic IS
    variable reduce: std_ulogic;
  BEGIN
    FOR i in V’range LOOP
      IF i = V’left THEN
        reduce := V(i);
      ELSE
        reduce := reduce XOR V(i);
      END IF;
    END LOOP;
    RETURN reduce;
  END xorr;

  FUNCTION stdv_to_unsigned ( V: std_logic_vector) return unsigned IS
    variable result: unsigned(V’range);
  BEGIN
    FOR i in V’range LOOP
      result(i) := V(i);
    END LOOP;
    RETURN result;
  END stdv_to_unsigned;

  FUNCTION unsigned_to_stdv ( V: unsigned) return std_logic_vector IS
    variable result: std_logic_vector(V’range);
  BEGIN
```

```
    FOR i in V’range LOOP
      result(i) := V(i);
    END LOOP;
    RETURN result;
  END unsigned_to_stdv;

---
  type tbl_mvl9_boolean is array (STD_ULOGIC) of boolean;
  constant is_x : tbl_mvl9_boolean :=
        (true, true, false, false, true, true, false, false, true);

  FUNCTION has_x (A : unsigned) RETURN boolean IS
  BEGIN
    FOR ii IN A’range LOOP
      IF (is_x(A(ii))) THEN
        RETURN( true ); -- has X
      END IF;
    END LOOP;
    RETURN( false );    -- does not have X
  END has_x;
---

END ovl_assertlib ;
```