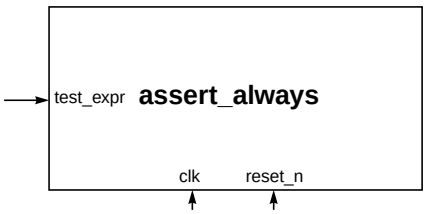
assert_always

Ensures that the value of a specified expression is TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_always
    [#(severity_level, property_type, msg, coverage_level)]
    instance_name (clk, reset_n, test_expr);
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should evaluate to TRUE on the rising clock edge.

Description

The assert_always assertion checker checks the single-bit expression *test_expr* at each rising edge of *clk* to verify the expression does not evaluate to FALSE.

The checker does not contain any complex sequential check other than evaluating *test_expr* at each rising edge of *clk*. It is used to verify a propositional property is not FALSE at clock boundaries or at the positive edge of *clk*.

Assertion Check

ASSERT_ALWAYS	Expression evaluated to FALSE.

Cover Points

none

See also

assert_implication, assert_never, assert_proposition

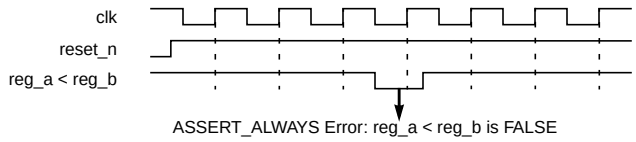
Example

```
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 #(
      'OVL_ERROR,                                          // severity_level
      'OVL_ASSERT,                                         // property_type
      "Error: count not within 0 to 9",                    // msg
      'OVL_COVER_ALL)                                      // coverage_level
      valid_count (
            clk,                                           // clock
            reset_n,                                       // reset
            (count >= 4'b0000) && (count <= 4'b1001 ) );   // test_expr
endmodule
```

Ensures that `count` is in the range [4'b0000 : 4'b1001] at each rising edge of `clk`.

```
assert_always #(
      'OVL_ERROR,                                          // severity_level
      'OVL_ASSERT,                                         // property_type
      "Error: reg_a < reg_b is FALSE",                     // msg
      'OVL_COVER_ALL)                                      // coverage_level
      reg_a_lt_reg_b (
            clk,                                           // clock
            reset_n,                                       // reset
            reg_a < reg_b );                               // test_expr
endmodule
```

Ensures that (`reg_a < reg_b`) is not FALSE at each rising edge of `clk`.



assert_always_on_edge

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

Parameters:
severity_level
edge_type
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_always_on_edge
    [#(severity_level, edge_type, property_type, msg, coverage_level)]
    instance_name (clk, reset_n, sampling_event, test_expr);
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
edge_type	Transition type for sampling event: ‘OVL_NOEDGE, ‘OVL_POSEDGE, ‘OVL_NEGEDGE, ‘OVL_ANYEDGE. Default: ‘OVL_NOEDGE.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
sampling_event	Expression that (along with *edge_type*) identifies when to evaluate and test *test_expr*.
test_expr	Expression that should evaluate to TRUE on the rising clock edge.

Description

The assert_always_on_edge assertion checker checks the single-bit expression *sampling_event* for a particular type of transition. If a matching transition of the sampling event occurs, the single-bit expression *test_expr* is evaluated at the rising edge of *clk* to verify the expression does not evaluate to FALSE.

The *edge_type* parameter determines which type of transition of *sampling_event* initiates the check:

- ‘OVL_POSEDGE initiates the check if *sampling_event* transitions to 1.
- ‘OVL_NEGEDGE initiates the check if *sampling_event* transitions to 0.
- ‘OVL_ANYEDGE initiates the check if *sampling_event* transitions to 0 or to 1.
- ‘OVL_NOEDGE always initiates the check. This is the default value of *edge_type*. In this case, *sampling_event* is never sampled and the checker has the same functionality as assert_always.

The checker is a variant of assert_always, with the added capability of qualifying the assertion with a sampling event transition. This checker is useful when events are identified by their transition in addition to their logical state.

Assertion Check

ASSERT_ALWAYS_ON_EDGE	Expression evaluated to FALSE when the sampling event transitioned as specified by *edge_type*.

Cover Points

none

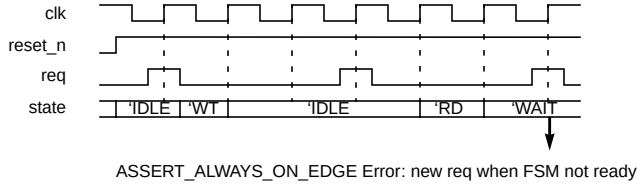
See also

assert_always, assert_implication, assert_never, assert_proposition

Example

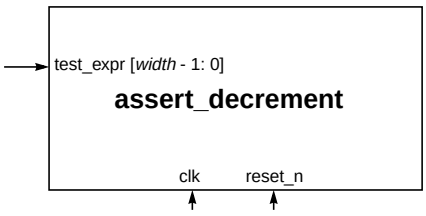
```
assert_always_on_edge #(
    ‘OVL_FATAL,                                  // severity_level
    ‘OVL_POSEDGE,                                // edge_type
    ‘OVL_ASSERT,                                 // property_type
    “Error: new req when FSM not ready”,         // msg
    ‘OVL_COVER_ALL)                              // coverage_level
    reg_a_lt_reg_b (
        clk,                                     // clock
        reset_n,                                 // reset
        req,                                     // sampling_event
        state == ‘IDLE);                         // test_expr
endmodule
```

Ensures that (state == ‘IDLE) is not FALSE at each rising edge of clk when req transitions to 1. The rising edge transition of req indicates initiation of a new request. This assertion ensures the FSM is ready to handle each new request.



assert_decrement

Ensures that the value of a specified expression changes only by the specified decrement value.



Parameters:
severity_level
width
value
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_decrement
    [ # ( severity_level, width, value, property_type, msg, coverage_level ) ]
    instance_name ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
value	Decrement value for *test_expr*. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should decrement by *value* whenever its value changes from the rising edge of clk to the next rising edge of clk.

Description

The assert_decrement assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the new value equals the previous value decremented by *value*. The checker allows the value of *test_expr* to wrap, if the total change equals the decrement *value*. For example, if width is 5 and value is 4, then the following change in *test_expr* is valid:

```
4'b00010 —> 4'b11110
```

The checker is useful for ensuring proper changes in structures such as counters and finite-state machines. For example, the checker is useful for circular queue structures with address counters that can wrap. Do not use this checker for variables or expressions that can increment. Instead consider using the assert_delta checker.

Assertion Check

ASSERT_DECREMENT	Expression evaluated to a value that is not its previous value decremented by *value*.

Cover Points

test_expr_change covered	Expression changed value.

Notes

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

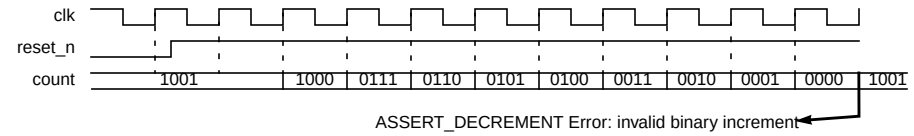
See also

assert_delta, assert_increment

Example

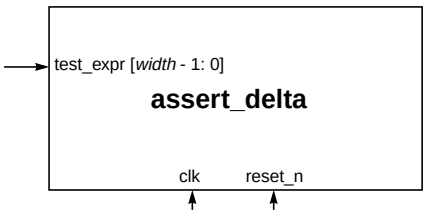
```
module programmable_counter_0_to_9 (reset_n, clk, dec);
  input reset_n, clk;
  input [1:0] dec;
   reg [3:0] count;
  always @(posedge clk)
  begin
     if (reset_n == 0) count <= 4'd9;
     elseif (count == 0) count <= 4'd9;
     else count <= count - dec;
  end
  assert_decrement #(
      `OVL_FATAL,                                  // severity_level
      4,                                           // width
      1,                                           // value
      `OVL_ASSERT,                                 // property_type
      "Error: invalid binary decrement",           // msg
      `OVL_COVER_ALL)                              // coverage_level
      valid_count (
            clk,                                   // clock
            reset_n,                               // reset
            count  );                              // test_expr
endmodule
```

Ensures that the programmable counter's count variable only decrements by 1. If count wraps, the assertion fails, because the change is not a binary decrement.



assert_delta

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



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_delta
    [ # ( severity_level, width, min, max, property_type, msg, coverage_level ) ]
    instance_name ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
min	Minimum delta value allowed for *test_expr*. Default: 1.
max	Maximum delta value allowed for *test_expr*. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should only change by a delta value in the range [min : max].

Description

The assert_delta assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the difference between the new value and the previous value (i.e., the delta value) is in the range from *min* to *max*, inclusive. If the delta value is less than *min* or greater than *max*, the assertion fails.

The checker is useful for ensuring proper changes in control structures such as up-down counters. For these structures, assert_delta can check for underflow and overflow. In datapath and arithmetic circuits, assert_delta can check for “smooth” transitions of the values of various variables (for example, for a variable that controls a physical variable that cannot detect a severe change from its previous value).

Assertion Check

ASSERT_DELTA	Expression changed value by a delta value not in the range [*min* : *max*].

Cover Points

test_expr_change covered	Expression changed value.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle.

Notes

1. The assertion check compares the current value of *test_expr* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.
2. The assertion check allows the value of *test_expr* to wrap. The overflow or underflow amount is included in the delta value calculation.

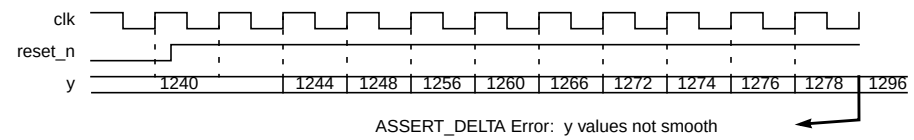
See also

assert_decrement, assert_increment

Example

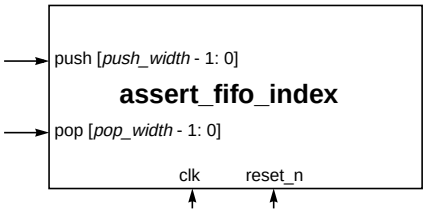
```
module smooth_test (reset_n, clk, a, b, x, y);
  input reset_n, clk;
  input [15:0] a, b, x;
  output [15:0] y;
   reg [15:0] y, xo;
  always @(posedge clk)
  begin
     if (reset_n == 0) begin
         y <= b;
         xo <= 0;
     end
     else begin
         y <= y + a * (x - xo);
         xo <= x;
     end
  end
  assert_delta #(
      'OVL_FATAL,                                  // severity_level
      16,                                          // width
      0,                                           // min
      8,                                           // max
      'OVL_ASSERT,                                 // property_type
      "Error: y values not smooth",                // msg
      'OVL_COVER_ALL)                              // coverage_level
      valid_smooth (
            clk,                                   // clock
            reset_n,                               // reset
            y  );                                  // test_expr
endmodule
```

Ensures that the smooth_test y output only changes by a maximum of 8 units each cycle (*min* is 0). This ensures the y output is "smooth".



assert_fifo_index

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



Parameters:
severity_level
depth
push_width
pop_width
property_type
msg
coverage_level
simultaneous_push_pop

Class:
2-cycle assertion

Syntax

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

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
depth	Maximum number of elements in the FIFO or queue structure. This parameter must be > 0. Default: 1.
push_width	Width of the *push* argument. Default: 1.
pop_width	Width of the *pop* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.
simultaneous_push_pop	Whether or not to allow simultaneous push/pop operations in the same clock cycle. When set to 0, if push and pop operations occur in the same cycle, the assertion fails. Default: 1 (simultaneous push/pop operations are allowed).

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
push	[*push_width* - 1: 0]	Expression that indicates the number of push operations that will occur during the current cycle.
pop	[*pop_width* - 1: 0]	Expression that indicates the number of pop operations that will occur during the current cycle.

Description

The assert_fifo_index assertion checker tracks the numbers of writes and reads that occur for a FIFO or queue memory structure. This checker does permit simultaneous pushes/pops on the queue within the same clock cycle. It ensures the FIFO never overflows (i.e., too many writes occur without enough reads) and never underflows (i.e., too many reads occur without enough writes). This checker is more complex than the assert_no_overflow and assert_no_underflow checkers, which check only the boundary conditions (overflow and underflow respectively).

Assertion Checks

OVERLOW	Push operation overflowed the FIFO.
UNDERFLOW	Pop operation underflowed the FIFO.
ILLEGAL PUSH AND POP	Push and pop operations performed in the same clock cycle, but the simultaneous_push_pop parameter is set to 0.

Cover Points

fifo_push covered	Push operation.
fifo_pop covered	Pop operation.
fifo_full covered	FIFO full.
fifo_empty covered	FIFO empty.
fifo_simultaneous_push_pop covered	Push and pop operations in the same clock cycle.

Errors

Depth parameter value must be > 0	Depth parameter is set to 0.

Notes

1. The checker checks the values of the *push* and *pop* expressions. By default, (i.e., simultaneous_push_pop is 1), “simultaneous” push/pop operations are allowed. In this case, the checker only ensures that the FIFO buffer index at the *end of the cycle* has not overflowed or underflowed. In particular, the checker assumes the design properly handles simultaneous push/pop operations. The assertion does not check the scenario where the FIFO overflows or underflows during a clock cycle during which both a push operation and a pop operation occur.

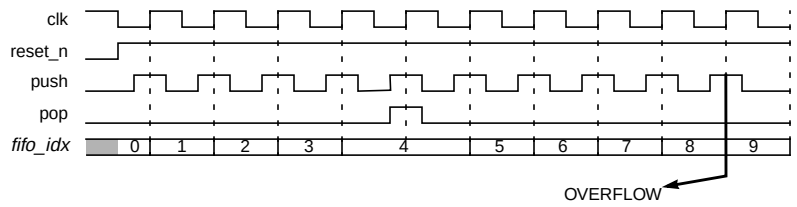
See also

assert_no_overflow, assert_no_underflow

Examples

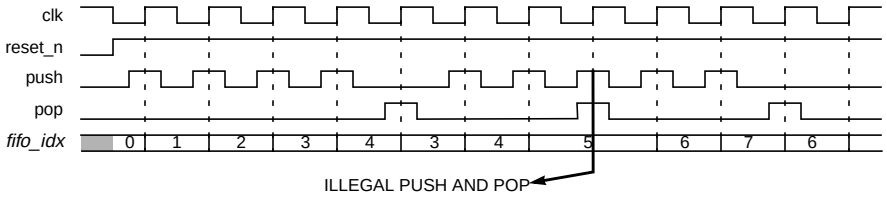
```
assert_fifo_index #(
    ‘OVL_FATAL,                    // severity_level
    8)                             // depth
    no_over_underflow (
        clk,                       // clock
        reset_n,                   // reset
        push,                      // push
        pop);                      // pop
```

Ensures that an 8-element FIFO never overflows or underflows. The severity of a violation is fatal (simulation terminates). The checker uses the default values for *push_width* and *pop_width* (1): Only single pushes and pops can occur in a clock cycle. The checker uses the default value of *simultaneous_push_pop* (i.e., 1): A push and pop operation in the same clock cycle is allowed.



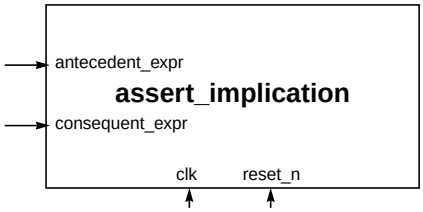
```
assert_fifo_index #(
    ‘OVL_ERROR,                 // severity_level
    8,                          // depth
    1,                          // push_width
    1,                          // pop_width
    ‘OVL_ASSERT,                // property_type
    “violation”,                // msg
    ‘OVL_COVER_ALL              // coverage_level
    1)                          // simultaneous_push_pop
    no_over_underflow (
        clk,                    // clock
        reset_n,                // reset
        push,                   // push
        pop);                   // pop
```

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



assert_implication

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



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_implication
        [#(severity_level, property_type, msg, coverage_level)]
        instance_name (clk, reset_n, antecedent_expr, consequent_expr);
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
antecedent_expr	Antecedent expression that is tested at the clock event.
consequent_expr	Consequent expression that should evaluate to TRUE if *antecedent_expr* evaluates to TRUE when tested.

Description

The assert_implication assertion checker checks the single-bit expression *antecedent_expr* at each rising edge of *clk*. If *antecedent_expr* is TRUE, then the checker verifies that the value of *consequent_expr* is also TRUE. If antecedent_expr not TRUE, then the assertion is valid regardless of the value of consequent_expr.

The checker does not contain any complex sequential check other than evaluating *antecedent_expr* and *consequent_expr* at each rising edge of *clk*. It is used to verify a propositional property always implies another propositional property at clock boundaries or at the positive edge of *clk*.

Assertion Check

ASSERT_IMPLICATION	Expression evaluated to FALSE.

Cover Points

cover_antecedent covered	The *antecedent_expr* evaluated to TRUE.

Notes

1. This assertion checker is equivalent to:

```
assert_always
    [#(severity_level, property_type, msg, coverage_level)]
    instance_name (clk, reset_n, (antecedent_expr ? consequent_expr : 1'b1 ));
```

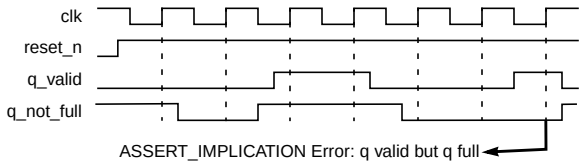
See also

`assert_always, assert_never, assert_proposition`

Example

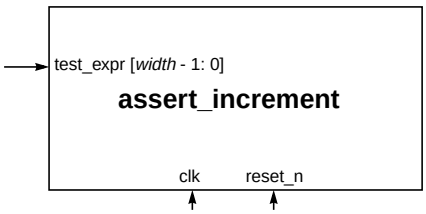
```
assert_implication #(
    ‘OVL_FATAL,                                  // severity_level
    ‘OVL_ASSERT,                                 // property_type
    “Error: q valid but q full”,                 // msg
    ‘OVL_COVER_ALL)                              // coverage_level
    not_full (
        clk,                                     // clock
        reset_n,                                 // reset
        q_valid,                                 // antecedent_expr
        q_not_full );                            // consequent_expr
endmodule
```

Ensures that `q_not_full` is TRUE at each rising edge of `clk` for which `q_valid` is TRUE.



assert_increment

Ensures that the value of a specified expression changes only by the specified increment value.



Parameters:
severity_level
width
value
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_increment
        [ # ( severity_level, width, value, property_type,  msg,  coverage_level ) ]
        instance_name  ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
value	Increment value for *test_expr.* Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should increment by *value* whenever its value changes from the rising edge of clk to the next rising edge of clk.

Description

The assert_increment assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from its value at the previous rising edge of *clk*. If so, the checker verifies that the new value equals the previous value incremented by *value*. The checker allows the value of *test_expr* to wrap, if the total change equals the increment *value*. For example, if *width* is 5 and *value* is 4, then the following change in *test_expr* is valid:

```
4'b11110 —> 4'b00010
```

The checker is useful for ensuring proper changes in structures such as counters and finite-state machines. For example, the checker is useful for circular queue structures with address counters that can wrap. Do not use this checker for variables or expressions that can decrement. Instead consider using the assert_delta checker.

Assertion Check

ASSERT_INCREMENT	Expression evaluated to a value that is not its previous value incremented by *value*.

Cover Points

test_expr_change covered	Expression changed value.

Notes

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

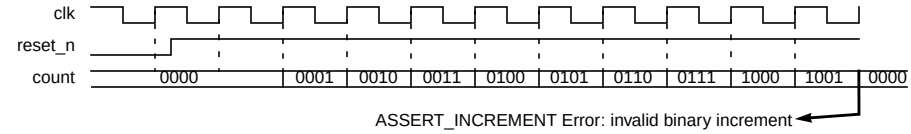
See also

assert_decrement, assert_delta

Example

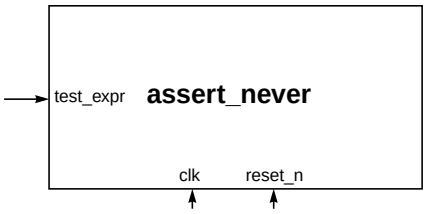
```
module programmable_counter_0_to_9 (reset_n, clk, inc);
  input reset_n, clk;
  input [1:0] inc;
   reg [3:0] count;
  always @(posedge clk)
  begin
     if (reset_n == 0) count <= 4'd0;
     elseif (count == 9) count <= 4'd0;
     else count <= count + inc;
  end
  assert_increment #(
      `OVL_FATAL,                                  // severity_level
      4,                                           // width
      1,                                           // value
      `OVL_ASSERT,                                 // property_type
      "Error: invalid binary increment",           // msg
      `OVL_COVER_ALL)                              // coverage_level
      valid_count (
            clk,                                   // clock
            reset_n,                               // reset
            count  );                              // test_expr
endmodule
```

Ensures that the programmable counter's count variable only increments by 1. If count wraps, the assertion fails, because the change is not a binary increment.



assert_never

Ensures that the value of a specified expression is not TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_never
        [ # ( severity_level, property_type,  msg,  coverage_level ) ]
        instance_name  ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk	Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should not evaluate to TRUE on the rising clock edge.

Description

The assert_never assertion checker checks the single-bit expression *test_expr* at each rising edge of *clk* to verify the expression does not evaluate to TRUE.

The checker does not contain any complex sequential check other than evaluating *test_expr* at each rising edge of *clk*. It is used to verify a propositional property is never TRUE at clock boundaries or at the positive edge of *clk*.

Assertion Check

ASSERT_NEVER	Expression evaluated to TRUE.
test_expr contains X/Z value	Expression evaluated to X or Z, and ‘OVL_XCHECK_OFF is not set.

Cover Points

none

Notes

1. By default, the assert_never assertion is pessimistic and the assertion fails if *test_expr* is not 0. However, if ‘OVL_XCHECK_OFF is set, the assertion fails if and only if *test_expr* is 1.

See also

assert_always, assert_implication, assert_proposition

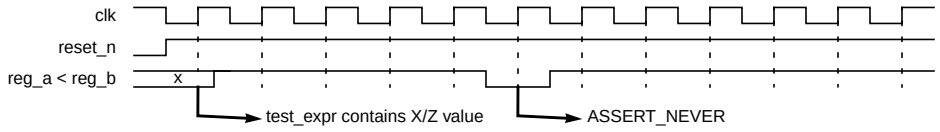
Example

```
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 #(
      `OVL_FATAL,                                   // severity_level
      `OVL_ASSERT,                                  // property_type
      "FIFO overflow",                              // msg
      `OVL_COVER_ALL)                               // coverage_level
      fifo_overflow (
            clk,                                    // clock
            reset_n,                                // reset
            fifo_full && write );                   // test_expr
  assert_never #(
      `OVL_FATAL,                                   // severity_level
      `OVL_ASSERT,                                  // property_type
      "FIFO underflow",                             // msg
      `OVL_COVER_ALL)                               // coverage_level
      fifo_underflow (
            clk,                                    // clock
            reset_n,                                // reset
            fifo_empty && read );                   // test_expr
endmodule
```

Ensures that `fifo` does not overflow (i.e., no write to a full `fifo`) or underflow (i.e., no read from an empty `fifo`).

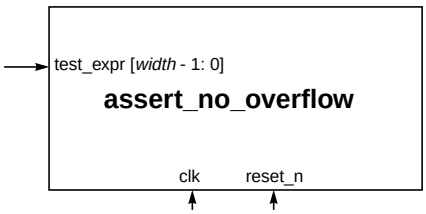
```
assert_never #(
      `OVL_ERROR,                                   // severity_level
      `OVL_ASSERT,                                  // property_type
      "",                                           // msg
      `OVL_COVER_ALL)                               // coverage_level
      valid_count (
            clk,                                    // clock
            reset_n,                                // reset
            reg_a < reg_b );                        // test_expr
endmodule
```

Ensures that (`reg_a < reg_b`) is FALSE at each rising edge of `clk`.



assert_no_overflow

Ensures that the value of a specified expression does not overflow.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_no_overflow
        [#( severity_level, width, min, max, property_type, msg, coverage_level )]
        instance_name ( clk, reset_n, test_expr );
```

Parameters

severity_level		Severity of the failure. Default: ‘OVL_ERROR.
width		Width of the *test_expr* argument. Width must be less than or equal to 32. Default: 1.
min		Minimum value in the test range of *test_expr*. Default: 0.
max		Maximum value in the test range of *test_expr*. Default: 2**_width_ - 1.
property_type		Property type. Default: ‘OVL_ASSERT.
msg		Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level		Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should not change from a value of *max* to a value out of the test range or to a value equal to *min*.

Description

The assert_no_overflow assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from a value (at the previous rising edge of *clk*) that was equal to *max*. If so, the checker verifies that the new value has not overflowed *max*. That is, it verifies the value of *test_expr* is not greater than *max* or less than or equal to *min* (in which case, the assertion fails).

The checker is useful for verifying counters, where it can ensure the counter does not wrap from the highest value to the lowest value in a specified range. For example, it can be used to check that memory structure pointers do not wrap around. For a more general test for overflow, use assert_delta or assert_fifo_index.

Assertion Check

ASSERT_NO_OVERFLOW	Expression changed value from *max* to a value not in the range [*min* + 1 : *max* - 1].

Cover Points

test_expr_change covered	Expression changed value.
test_expr_at_min covered	Expression evaluated to *min*.
test_expr_at_max covered	Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle for which *test_expr* changed from *max*.

Notes

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

See also

assert_delta, assert_fifo_index

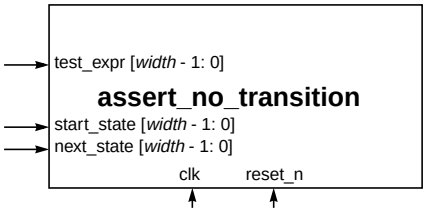
Example

```
module 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_range #(
      'OVL_FATAL,                          // severity_level
      4,                                   // width
      0,                                   // min
      15,                                  // max
      'OVL_ASSERT,                         // property_type
      "Error: count overflow",             // msg
      'OVL_COVER_ALL)                      // coverage_level
      counter_with_overflow (
            clk,                           // clock
            reset_n,                       // reset
            count  );                      // test_expr
endmodule
```

Ensures that count does not overflow (i.e., change from a value of 15 at the rising edge of clk to a value of 0 at the next rising edge of clk).

assert_no_transition

Ensures that the values of a specified expression do not transition from start states to the corresponding next states.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_no_transition
    [ # ( severity_level, width, property_type, msg, coverage_level ) ]
    instance_name ( clk, reset_n, test_expr, start_state, next_state ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should not transition to *next_state* on the rising edge of *clk* if its value at the previous rising edge of *clk* is the same as the current value of *start_state*.
start_state	[*width* - 1: 0]	Expression that indicates the start state for the assertion check. If the start state matches the value of *test_expr* on the previous rising edge of *clk*, the check is performed.
next_state	[*width* - 1: 0]	Expression that indicates the invalid next state for the assertion check. If the value of *test_expr* was *start_state* at the previous rising edge of *clk*, then the value of *test_expr* should not equal *next_state* on the current rising edge of *clk*.

Description

The assert_no_transition assertion checker checks the expression *test_expr* and *start_state* at each rising edge of *clk* to see if the value of *test_expr* at the previous rising edge of *clk* equals the current value of *start_state*. If so, the checker verifies that the current value of *test_expr* does not equal the current value of *next_state*. The assertion fails if *test_expr* is equal to *next_state*.

The *start_state* and *next_state* expressions are verification events that can change. In particular, the same assertion checker can be coded to verify multiple types of transitions of *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) do not transition to invalid values.

Assertion Check

ASSERT_no_transition	Expression transitioned from *start_state* to a value equal to *next_state*.

Cover Points

start_state covered	Expression assumed a start state value.

Notes

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

See also

assert_transition

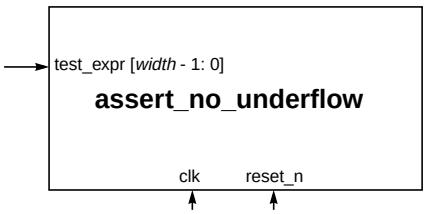
Example

```
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 #(
      `OVL_FATAL,                                          // severity_level
      4,                                                   // width
      `OVL_ASSERT,                                         // property_type
      "Error: bad count transition",                       // msg
      `OVL_COVER_ALL)                                      // coverage_level
      valid_count (
            clk,                                           // clock
            reset_n,                                       // reset
            count,                                         // test_expr
            4'd9,                                          // start_state
            (sel_09 == 1'b1) ? 4'd10 : 4'd0  );            // next_state
endmodule
```

Ensures that count transitions from 4'd9 properly. If sel_09 is 1, count should not have transitioned to 4'd10. Otherwise, count should not have transitioned to 4'd0.

assert_no_underflow

Ensures that the value of a specified expression does not underflow.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_no_underflow
        [ # ( severity_level, width, min, max, property_type, msg, coverage_level ) ]
        instance_name ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level		Severity of the failure. Default: ‘OVL_ERROR.
width		Width of the *test_expr* argument. Width must be less than or equal to 32. Default: 1.
min		Minimum value in the test range of *test_expr*. Default: 0.
max		Maximum value in the test range of *test_expr*. Default: 2**_width_ - 1.
property_type		Property type. Default: ‘OVL_ASSERT.
msg		Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level		Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should not change from a value of *min* to a value out of range or to a value equal to *max*.

Description

The assert_no_underflow assertion checker checks the expression *test_expr* at each rising edge of *clk* to determine if its value has changed from a value (at the previous rising edge of *clk*) that was equal to *min*. If so, the checker verifies that the new value has not underflowed *min*. That is, it verifies the value of *test_expr* is not less than *min* or greater than or equal to *max* (in which case, the assertion fails).

The checker is useful for verifying counters, where it can ensure the counter does not wrap from the lowest value to the highest value in a specified range. For example, it can be used to check that memory structure pointers do not wrap around. For a more general test for underflow, use assert_delta or assert_fifo_index.

Assertion Check

ASSERT_NO_UNDERFLOW	Expression changed value from *min* to a value not in the range [*min* + 1 : *max* - 1].

Cover Points

test_expr_change covered	Expression changed value.
test_expr_at_min covered	Expression evaluated to *min*.
test_expr_at_max covered	Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle for which *test_expr* changed from *max*.

Notes

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

See also

assert_delta, assert_fifo_index

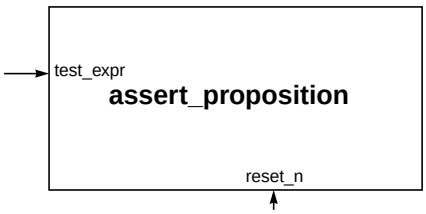
Example

```
module 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_range #(
      'OVL_FATAL,                                  // severity_level
      4,                                           // width
      0,                                           // min
      15,                                          // max
      'OVL_ASSERT,                                 // property_type
      "Error: count underflow",                    // msg
      'OVL_COVER_ALL)                              // coverage_level
      counter_with_underflow (
            clk,                                   // clock
            reset_n,                               // reset
            count  );                              // test_expr
endmodule
```

Ensures that count does not underflow (i.e., change from a value of 0 at the rising edge of clk to a value of 15 at the next rising edge of clk).

assert_proposition

Ensures that the value of a specified expression is always combinationally TRUE.



Parameters:
severity_level
property_type
msg
coverage_level

Class:
combinational assertion

Syntax

```
assert_proposition
        [#(severity_level, property_type, msg, coverage_level)]
        instance_name (reset_n, test_expr);
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

reset_n	Active low synchronous reset signal indicating completed initialization.
test_expr	Expression that should always evaluate to TRUE.

Description

The assert_proposition assertion checker checks the single-bit expression *test_expr* when it changes value to verify the expression evaluates to TRUE.

The checker does not contain any complex sequential check other than evaluating *test_expr* when it changes value. It is used to verify a propositional property is TRUE at all times.

Assertion Check

ASSERT_PROPOSITION	Expression evaluated to FALSE.

Cover Points

none

Notes

1. Formal verification tools and hardware emulation/acceleration systems ignore this checker. To verify propositional properties with these tools, consider using assert_always.

See also

assert_always, assert_implication, assert_never

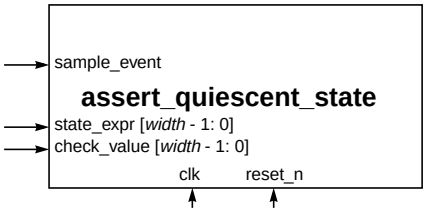
Example

```
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 #(
      'OVL_FATAL,                                          // severity_level
      'OVL_ASSERT,                                         // property_type
      "Error: count not within 0 to 9",                    // msg
      'OVL_COVER_ALL)                                      // coverage_level
      valid_count (
            clk,                                           // clock
            reset_n,                                       // reset
            (count >= 4'b0000) && (count <= 4'b1001 ) );   // test_expr
endmodule
```

Ensures that count is in the range [4'b0000 : 4'b1001] at each rising edge of clk.

assert_quiescent_state

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



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_quiescent_state
        [ # ( severity_level, width, property_type, msg, coverage_level ) ]
        instance_name ( clk, reset_n, state_expr, check_value, sample_event ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *state_expr* and *check_value* arguments. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
state_expr	[*width* - 1: 0]	Expression that should have the same value as *check_value* on the rising edge of *clk* if *sample_event* transitioned to TRUE in the previous clock cycle (or is currently transitioning to TRUE).
check_value	[*width* - 1: 0]	Expression that indicates the value *state_expr* should have on the rising edge of *clk* if *sample_event* transitioned to TRUE in the previous clock cycle (or is currently transitioning to TRUE).
sample_event		Expression that initiates the quiescent state check when its value transitions to TRUE.

Description

The assert_quiescent_state assertion checker checks the expression *sample_event* at each rising edge of *clk* to see if its value has transitioned to TRUE (i.e., its current value is TRUE and its value on the previous rising edge of *clk* is not TRUE). If so, the checker verifies that the current value of *state_expr* equals the current value of *check_value*. The assertion fails if *state_expr* is not equal to *check_value*.

The *state_expr* and *check_value* expressions are verification events that can change. In particular, the same assertion checker can be coded to compare different check values (if they are checked in different cycles).

The checker is useful for verifying the states of state machines when transactions complete.

Assertion Check

ASSERT_QUIESCENT_STATE	The *sample_event* expression transitioned to TRUE, but the values of *state_expr* and *check_value* were not the same.

Cover Points

none

Notes

1. The assertion check compares the current value of *sample_event* with its previous value. Therefore, checking does not start until the second rising clock edge of *clk* after *reset_n* deasserts.
2. The checker recognizes the Verilog macro ‘OVL_ASSERT_END_OF_SIMULATION. If set, the quiescent state check is also performed at the end of simulation (regardless of the value of *sample_event*).

See also

assert_no_transition, assert_transition

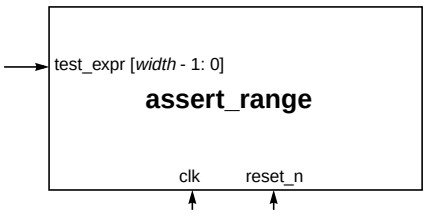
Example

```
  assert_quiescent_state #(
      ‘OVL_FATAL,                                  // severity_level
      4,                                           // width
      ‘OVL_ASSERT,                                 // property_type
      “Error: bad count transition”,               // msg
      ‘OVL_COVER_ALL)                              // coverage_level
      valid_end_of_transaction_state (
            clk,                                   // clock
            reset_n,                               // reset
            transaction_state,                     // state_expr
            ‘TR_FSM_IDLE,                          // check_value
            end_of_transaction;                    // sample_event
endmodule
```

Ensures that whenever end_of_transaction asserts at the completion of each transaction, the value of transaction_state is ‘TR_STATE_IDLE.

assert_range

Ensures that the value of a specified expression is in a specified range.



Parameters:
severity_level
width
min
max
property_type
msg
coverage_level

Class:
single-cycle assertion

Syntax

```
assert_range
        [ # ( severity_level, width, min, max, property_type, msg, coverage_level ) ]
        instance_name ( clk, reset_n, test_expr ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
min	Minimum value allowed for *test_expr*. Default: 0.
max	Maximum value allowed for *test_expr*. Default: 2**width - 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should evaluate to a value in the range from *min* to *max* (inclusive) on the rising clock edge.

Description

The assert_range assertion checker checks the expression *test_expr* at each rising edge of *clk* to verify the expression falls in the range from *min* to *max*, inclusive. The assertion fails if *test_expr* < *min* or *max* < *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) are within their proper ranges. The checker is also useful for ensuring datapath variables and expressions are in legal ranges.

Assertion Check

ASSERT_RANGE	Expression evaluated outside the range *min* to *max*.

Cover Points

cover_test_expr_change covered	Expression changed value.
cover_test_expr_at_min covered	Expression evaluated to *min*.
cover_test_expr_at_max covered	Expression evaluated to *max*.

Errors

The parameters *min* and *max* must be specified such that *min* is less than or equal to *max*. Otherwise, the assertion fails on each tested clock cycle.

See also

assert_always, assert_implication, assert_never, assert_proposition

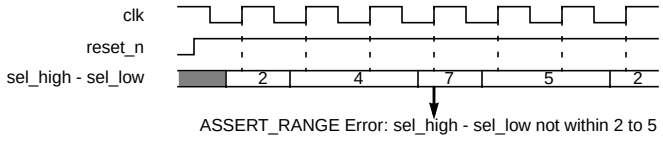
Example

```
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  || count == 4'd9) count <= 4'b0000;
    else count <= count + 1;
  end
  assert_range #(
      'OVL_FATAL,                                  // severity_level
      4,                                           // width
      0,                                           // min
      9,                                           // max
      'OVL_ASSERT,                                 // property_type
      "Error: count not within 0 to 9",            // msg
      'OVL_COVER_ALL)                              // coverage_level
      valid_count (
            clk,                                   // clock
            reset_n,                               // reset
            count  );                              // test_expr
endmodule
```

Ensures that count is in the range [4'b0000 : 4'b01001] at each rising edge of clk.

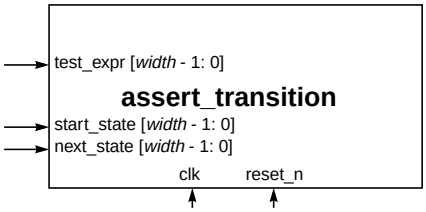
```
assert_range #(
      'OVL_ERROR,                                  // severity_level
      3,                                           // width
      2,                                           // min
      5,                                           // max
      'OVL_ASSERT,                                 // property_type
      "Error: sel_high - sel_low not within 2 to 5",  // msg
      'OVL_COVER_ALL)                              // coverage_level
      valid_sel (
            clk,                                   // clock
            reset_n,                               // reset
            sel_high - sel_low );                  // test_expr
endmodule
```

Ensures that (sel_high - sel_low) is in the range [2 : 5] at each rising edge of clk.



assert_transition

Ensures that the values of a specified expression transition properly from start states to the corresponding next states.



Parameters:
severity_level
width
property_type
msg
coverage_level

Class:
2-cycle assertion

Syntax

```
assert_transition
        [ # ( severity_level, width, property_type, msg, coverage_level ) ]
        instance_name ( clk, reset_n, test_expr, start_state, next_state ) ;
```

Parameters

severity_level	Severity of the failure. Default: ‘OVL_ERROR.
width	Width of the *test_expr* argument. Default: 1.
property_type	Property type. Default: ‘OVL_ASSERT.
msg	Error message printed when assertion fails. Default: “VIOLATION”.
coverage_level	Coverage level. Default: ‘OVL_COVER_ALL.

Ports

clk		Clock event for the assertion. The checker assumes the rising edge of the clock.
reset_n		Active low synchronous reset signal indicating completed initialization.
test_expr	[*width* - 1: 0]	Expression that should transition to *next_state* on the rising edge of *clk* if its value at the previous rising edge of *clk* is the same as the current value of *start_state*.
start_state	[*width* - 1: 0]	Expression that indicates the start state for the assertion check. If the start state matches the value of *test_expr* on the previous rising edge of *clk*, the check is performed.
next_state	[*width* - 1: 0]	Expression that indicates the only valid next state for the assertion check. If the value of *test_expr* was *start_state* at the previous rising edge of *clk*, then the value of *test_expr* should equal *next_state* on the current rising edge of *clk*.

Description

The assert_transition assertion checker checks the expression *test_expr* and *start_state* at each rising edge of *clk* to see if the value of *test_expr* at the previous rising edge of *clk* equals the current value of *start_state*. If so, the checker verifies that the current value of *test_expr* equals the current value of *next_state*. The assertion fails if *test_expr* is not equal to *next_state*.

The *start_state* and *next_state* expressions are verification events that can change. In particular, the same assertion checker can be coded to verify multiple types of transitions of *test_expr*.

The checker is useful for ensuring certain control structure values (such as counters and finite-state machine values) transition properly.

Assertion Check

ASSERT_TRANSITION	Expression transitioned from *start_state* to a value different from *next_state*.

Cover Points

start_state covered	Expression assumed a start state value.

Notes

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

See also

assert_no_transition

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 #(
      `OVL_FATAL,                                          // severity_level
      4,                                                   // width
      `OVL_ASSERT,                                         // property_type
      "Error: bad count transition",                       // msg
      `OVL_COVER_ALL)                                      // coverage_level
      valid_count (
            clk,                                           // clock
            reset_n,                                       // reset
            count,                                         // test_expr
            4'd9,                                          // start_state
            (sel_09 == 1'b0) ? 4'd10 : 4'd0  );            // next_state
endmodule
```

Ensures that count transitions from 4'd9 properly. If sel_09 is 0, count should have transitioned to 4'd10. Otherwise, count should have transitioned to 4'd0.

