COMS30026 Design Verification

Assertion-based Verification

Kerstin Eder

Trustworthy Systems Laboratory

https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/

University of Bristol

Department of COMPUTER SCIENCE

What is an assertion?

- An assertion is a statement that a particular property is required to be true.
  – A property is a Boolean-valued expression, e.g. in SystemVerilog.
- Assertions can be checked either during simulation or using a formal property checker.

What is an assertion?

- An assertion is a statement that a particular property is required to be true.
  - A property is a Boolean-valued expression, e.g. in SystemVerilog.
- Assertions can be checked either during simulation or using a formal property checker.
- Assertions have been used in SW development for a long time.
  - **`assert.h`** in standard library of C

    ```
    #include <assert.h>
    ```

  - C preprocessor macro **`assert()`**
  - Used to detect **`NULL`** pointers, out-of-range data, ensure loop invariants, pre- and post-conditions, etc.

Assertions in C code

```c
#include <stdio.h>
#include <assert.h>

int mysquare(int n) {
  int s = 0;
  int i = 0;
  int k = 0; /* assertion variable to count the number of times in the loop */

  assert (n >= 0); // Pre-condition to catch invalid input

  assert (s == k*n && i==k); // Invariant to catch mistaken variable initialisation, e.g. i != 0 or s != 0

  while (i < n) {
    s = s + n;
    i = i + 1;
    k = k + 1;
    assert ((s == k*n) && (i==k)); // Invariant to catch errors in the loop computation

  }

  assert (k == n); // Post-condition to catch a mistaken final state of the loop

  assert (s == k*n && i==k); // Invariant to catch errors in the loop computation

  assert (s == n * n); // Check desired post-condition

  return s;
}


int main() {
  int n = -4;
  int square = 0;

  printf("n = %d\n", n);
  square = mysquare(n);
  printf("n^2 = %d\n", square);

  return 0;
}
```

Assertions in C code

```c
1  #include <stdio.h>
2  #include <assert.h>
3
4  int mysquare(int n) {
5    int s = 0;
6    int i = 0;
7    int k = 0; /* assertion variable to count the number of times in the loop */
8
9    assert (n >= 0); // Pre-condition to catch invalid input
10
11   assert (s == k*n && i==k); // Invariant to catch mistaken variable initialisation, e.g. i != 0 or s != 0
12
13   while (i < n) {
14     s = s + n;
15     i = i + 1;
16     k = k + 1;
17     assert ((s == k*n) && (i==k)); // Invariant to catch errors in the loop computation
18
19   }
20
21   assert (k == n); // Post-condition to catch a mistaken final state of the loop
22
23   assert (s == k*n && i==k); // Invariant to catch errors in the loop computation
24
25   assert (s == n * n); // Check desired post-condition
26
27   return s;
28 }
29
30
31 int main() {
32   int n = -4;
33   int square = 0;
34
35   printf("n = %d\n", n);
36   square = mysquare(n);
37   printf("n^2 = %d\n", square);
38
39   return 0;
40 }
```

```
cskie@it000908:SLIDES$ gcc mysquare.c -o mysquare
cskie@it000908:SLIDES$ ./mysquare
n = 4
n^2 = 16
cskie@it000908:SLIDES$ gcc mysquare.c -o mysquare
cskie@it000908:SLIDES$ ./mysquare
n = -4
Assertion failed: (n >= 0), function mysquare, file mysquare.c, line 9.
Abort trap: 6
cskie@it000908:SLIDES$ _
```

Assertions in C code

```c
1  #include <stdio.h>
2  #include <assert.h>
3
4  int mysquare(int n) {
5    int s = 0;
6    int i = 0;
7    int k = 0; /* assertion variable to count the number of times in the loop */
8
9    assert (n >= 0); // Pre-condition to catch invalid input
10
11   assert (s == k*n && i==k); // Invariant to catch mistaken variable initialisation, e.g. i != 0 or s != 0
12
13   while (i < n) {
14     s = s + n;
15     i = i + 1;
16     k = k + 1;
17     assert ((s == k*n) && (i==k)); // Invariant to catch errors in the loop computation
18
19   }
20
21   assert (k == n); // Post-condition to catch a mistaken final state of the loop
22
23   assert (s == k*n && i==k); // Invariant to catch errors in the loop computation
24
25   assert (s == n * n); // Check desired post-condition
26
27   return s;
28 }
29
30
31 int main() {
32   int n = -4;
33   int square = 0;
34
35   printf("n = %d\n", n);
36   square = mysquare(n);
37   printf("n^2 = %d\n", square);
38
39   return 0;
40 }
```

```
[cskie@it000908:SLIDES$ gcc mysquare.c -o mysquare
[cskie@it000908:SLIDES$ ./mysquare
n = 4
n^2 = 16
[cskie@it000908:SLIDES$ gcc mysquare.c -o mysquare
[cskie@it000908:SLIDES$ ./mysquare
n = -4
Assertion failed: (n >= 0), function mysquare, file mysquare.c, line 9.
Abort trap: 6
[cskie@it000908:SLIDES$ _
```

HW Assertions

- **Combinatorial** (i.e. “zero-time”) **conditions**
  - ensure functional correctness
  - must be valid at all times
    - “The buffer never overflows.”
    - “The register always holds a single-digit value.”
    - “The state machine encoding is *one hot*.”

HW Assertions

- **Combinatorial** (i.e. "zero-time") **conditions**
  - ensure functional correctness
  - must be valid at all times
    - "The buffer never overflows."
    - "The register always holds a single-digit value."
    - "The state machine encoding is *one hot*."
- **Temporal conditions**
  - to verify sequential functional behaviour over a period of time
    - "The grant signal must be asserted for a single clock cycle."
    - "A request must always be followed by a grant or an abort within 5 clock cycles."
  - Temporal assertion languages facilitate specification of temporal properties.
    - System Verilog Assertions (SVA)
    - Property Specification Language (PSL)

The Open Verification Library

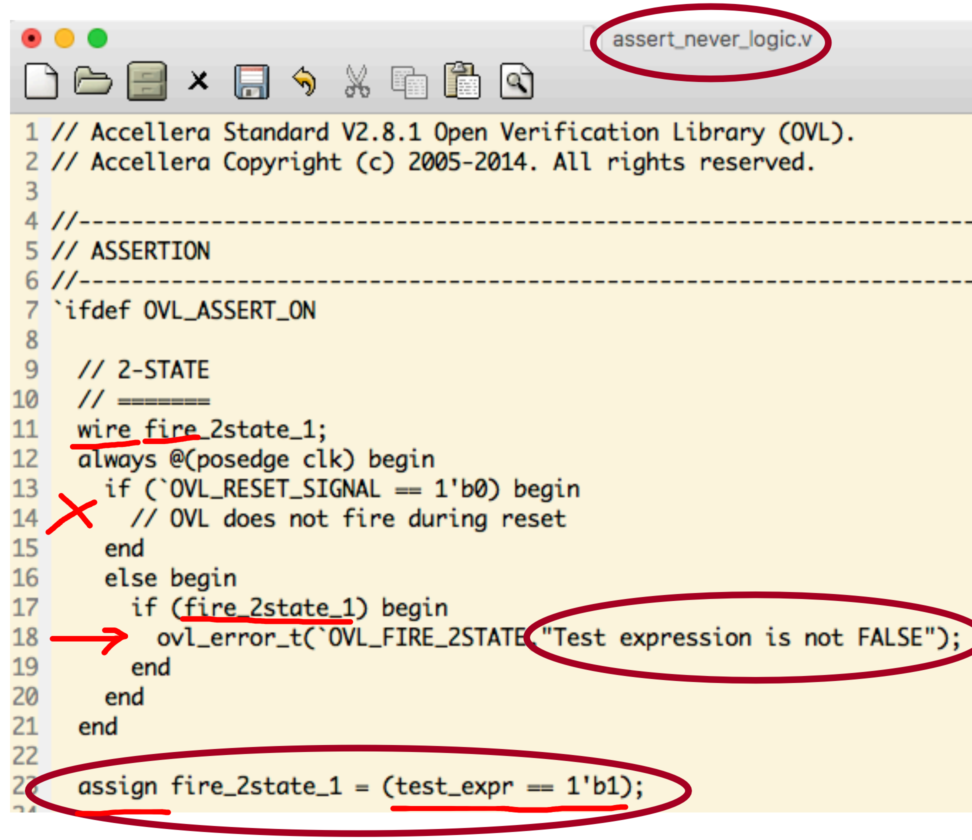
- Revolution through Foster & Bening's OVL for Verilog in early 2000
  - Clever way of encoding a re-usable assertion library originally in Verilog. ☺
  - 33 assertion checkers
  - OVL language support for: Verilog, VHDL, PSL, SVA

assert_never_logic.v

```
// Accellera Standard V2.8.1 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2014. All rights reserved.

//-----------------------------------------------------------------------
// ASSERTION
//-----------------------------------------------------------------------
`ifdef OVL_ASSERT_ON

  // 2-STATE
  // =======
  wire fire_2state_1;
  always @(posedge clk) begin
    if (`OVL_RESET_SIGNAL == 1'b0) begin
      // OVL does not fire during reset
    end
    else begin
      if (fire_2state_1) begin
        ovl_error_t(`OVL_FIRE_2STATE,"Test expression is not FALSE");
      end
    end
  end

  assign fire_2state_1 = (test_expr == 1'b1);
```

The Open Verification Library

- Revolution through Foster & Bening's OVL for Verilog in early 2000
  - Clever way of encoding a re-usable assertion library originally in Verilog. ☺
  - 33 assertion checkers
  - OVL language support for: Verilog, VHDL, PSL, SVA




```
// Accellera Standard V2.8.1 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2014. All rights reserved.

//------------------------------------------------------------------
// ASSERTION
//------------------------------------------------------------------
`ifdef OVL_ASSERT_ON

  // 2-STATE
  // =======
  wire fire_2state_1;
  always @(posedge clk) begin
    if (`OVL_RESET_SIGNAL == 1'b0) begin
      // OVL does not fire during reset
    end
    else begin
      if (fire_2state_1) begin
        ovl_error_t(`OVL_FIRE_2STATE,"Test expression is not FALSE");
      end
    end
  end

  assign fire_2state_1 = (test_expr == 1'b1);
```

The Open Verification Library

- Revolution through Foster & Bening's OVL for Verilog in early 2000
  - Clever way of encoding a re-usable assertion library originally in Verilog. ☺
  - 33 assertion checkers
  - OVL language support for: Verilog, VHDL, PSL, SVA
- Assertions have now become very popular for Verification, giving rise to **Assertion-Based Verification** (and also Assertion-Based Design).

assert_never_logic.v

```
// Accellera Standard V2.8.1 Open Verification Library (OVL).
// Accellera Copyright (c) 2005-2014. All rights reserved.

//------------------------------------------------------------------------
// ASSERTION
//------------------------------------------------------------------------
`ifdef OVL_ASSERT_ON

  // 2-STATE
  // =======
  wire fire_2state_1;
  always @(posedge clk) begin
    if (`OVL_RESET_SIGNAL == 1'b0) begin
      // OVL does not fire during reset
    end
    else begin
      if (fire_2state_1) begin
        ovl_error_t(`OVL_FIRE_2STATE,"Test expression is not FALSE");
      end
    end
  end

  assign fire_2state_1 = (test_expr == 1'b1);
```

OVL is an Accellera Standard

SYSTEMS INITIATIVE™

http://www.accellera.org/downloads/standards/ovl

SAFETY & LIVENESS

Safety Properties

- Safety: Something bad does not happen
  - The FIFO does not overflow.
  - The system does not allow more than one process at a time to modify the shared memory.
  - Requests are answered within 5 clock cycles.

Safety Properties

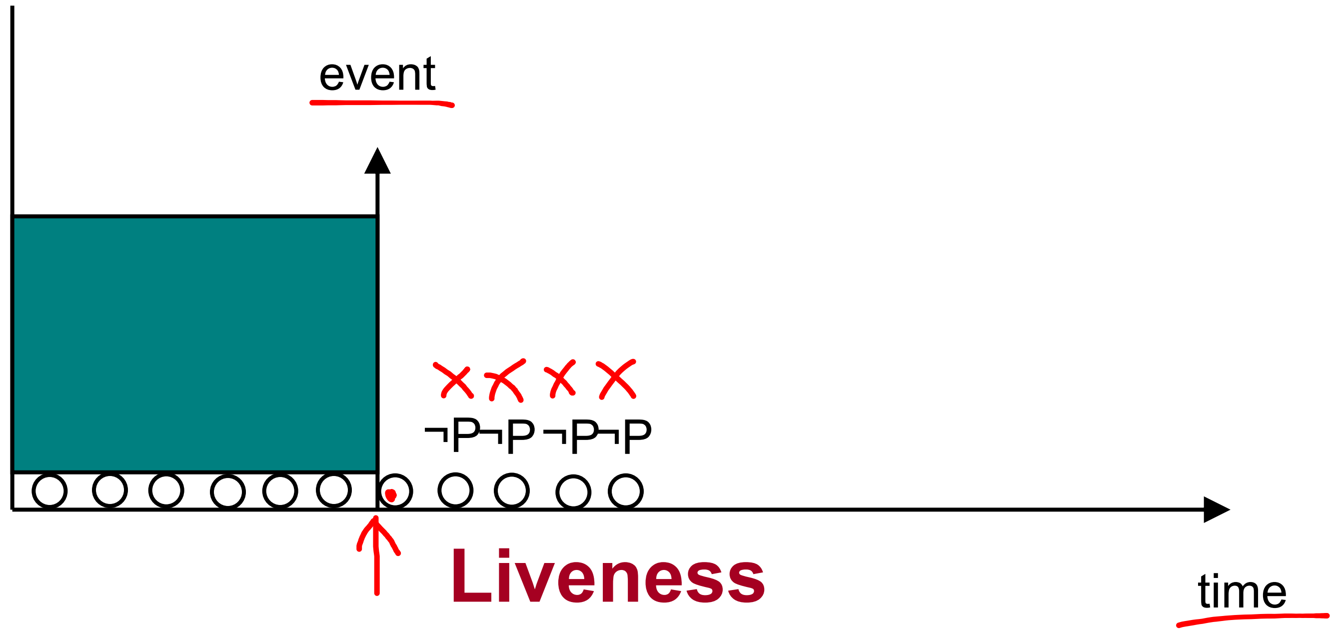
- Safety: Something bad does not happen
  – The FIFO does not overflow.
  – The system does not allow more than one process at a time to modify the shared memory.
  – Requests are answered within 5 clock cycles.
- More formally: *A safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property.* *[Accellera PSL-1.1 2004]*

Safety properties can be falsified by a finite simulation run.

Liveness Properties

- Liveness: Something good eventually happens
  - The decoding algorithm eventually terminates.
  - Every request is eventually acknowledged.
- More formally: *A liveness property is a property for which any finite path can be extended to a path satisfying the property.* *[Foster etal.: Assertion-Based Design. 2nd Edition, Kluwer, 2010.]*

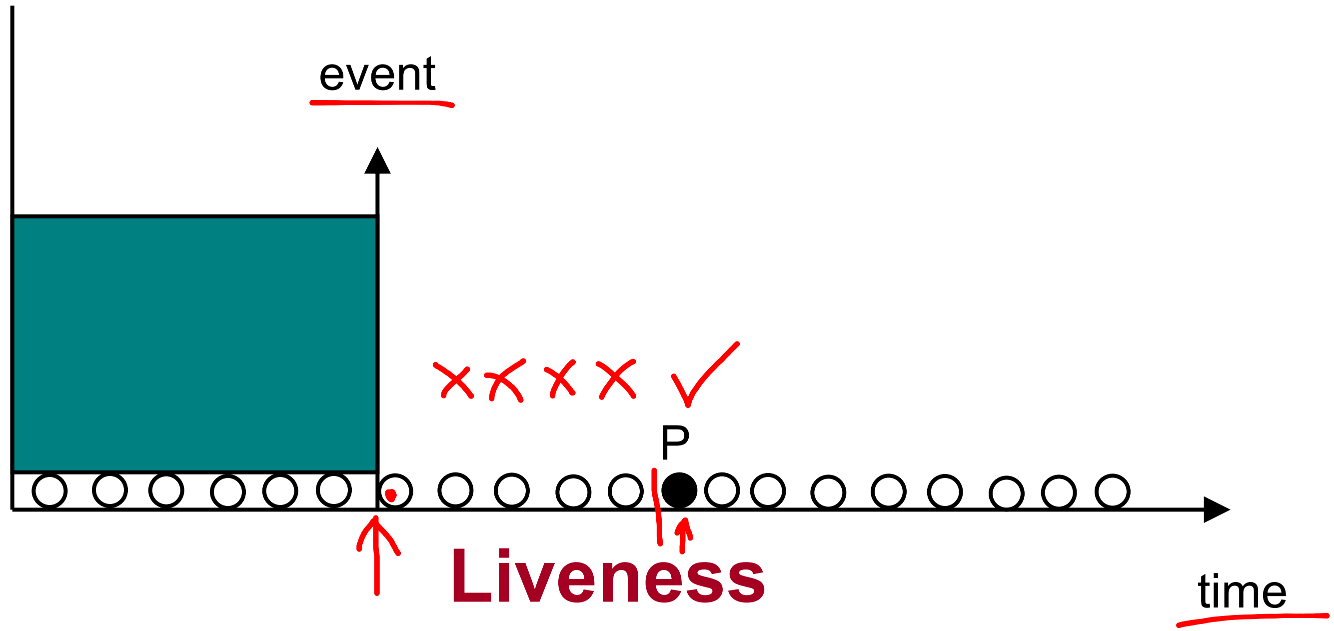
Liveness



- Assertion P must **eventually** be valid after the event occurs.

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

Liveness



- Assertion P must **eventually** be valid after the event occurs.

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

Von André Karwath aka Aka - Eigenes Werk, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=103762

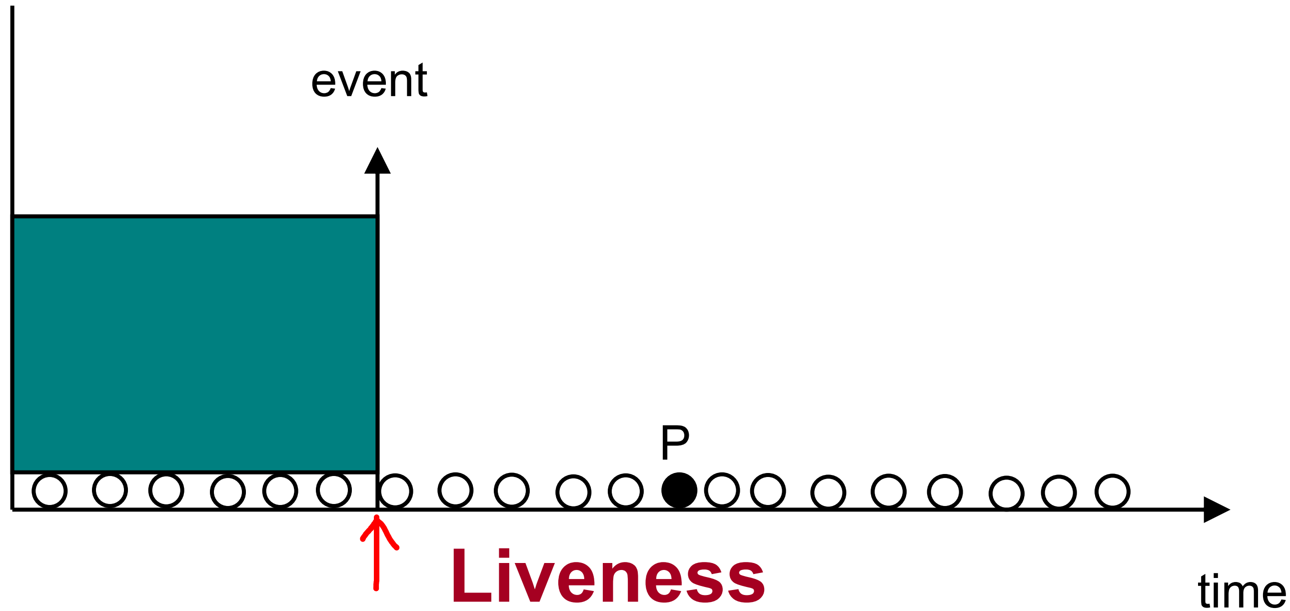
Von André Karwath aka Aka - Eigenes Werk, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=103762

Liveness Properties

- Liveness: Something good eventually happens
  - The decoding algorithm eventually terminates.
  - Every request is eventually acknowledged.
- More formally: *A liveness property is a property for which any finite path can be extended to a path satisfying the property.* *[Foster etal.: Assertion-Based Design. 2nd Edition, Kluwer, 2010.]*

  In theory, liveness properties can only be falsified by an infinite simulation run.
  - Practically, we often assume that the "graceful end-of-test" represents infinite time.
    - If the good thing did not happen after this period, we assume that it will never happen, and thus the property is falsified.

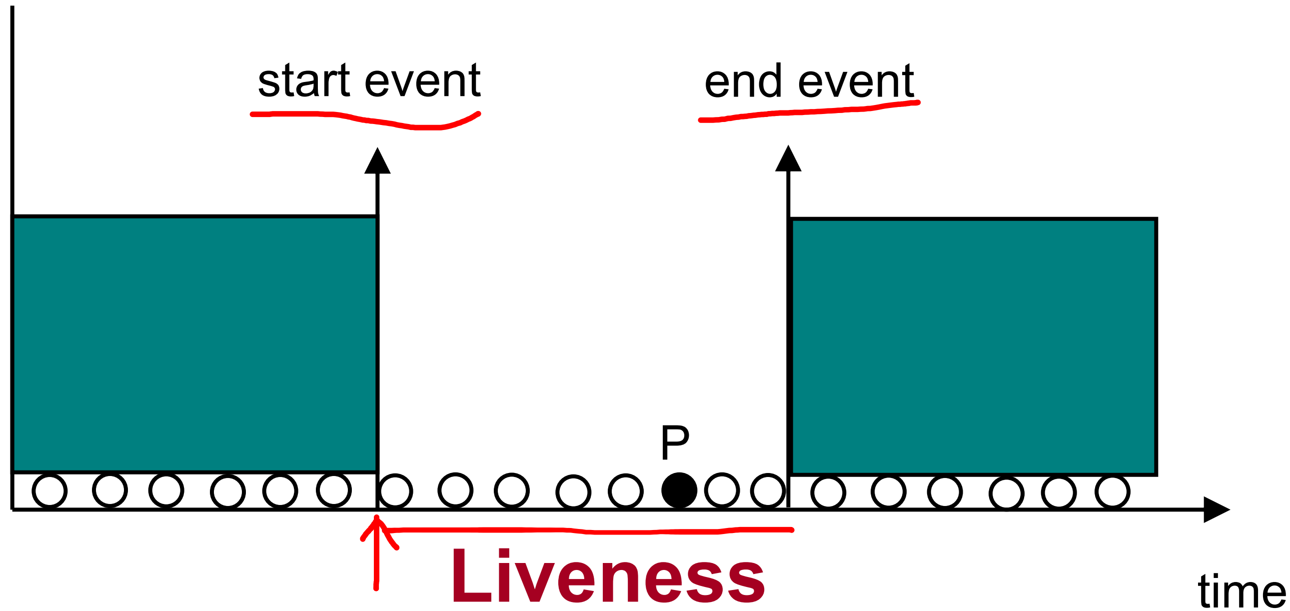
Bounded Liveness



- Assertion P must **eventually** be valid after the event occurs

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

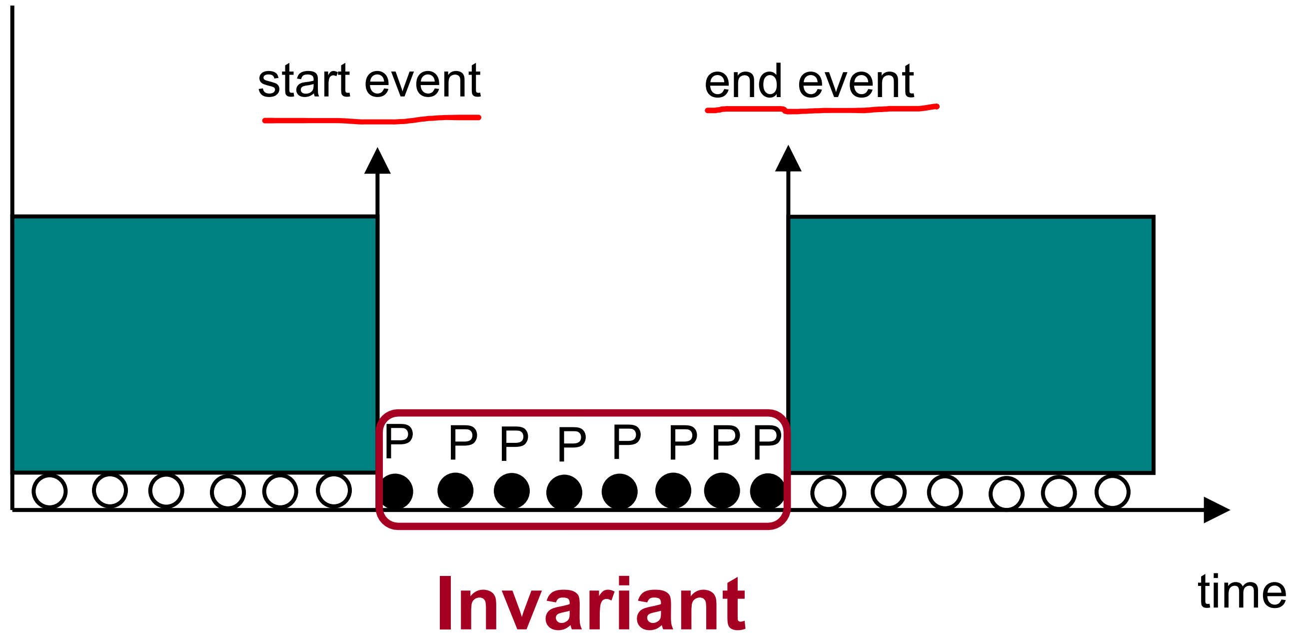
Bounded Liveness



- Assertion P must **eventually** be valid after the **start event trigger** occurs and **before the end event trigger** occurs.

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

Invariant



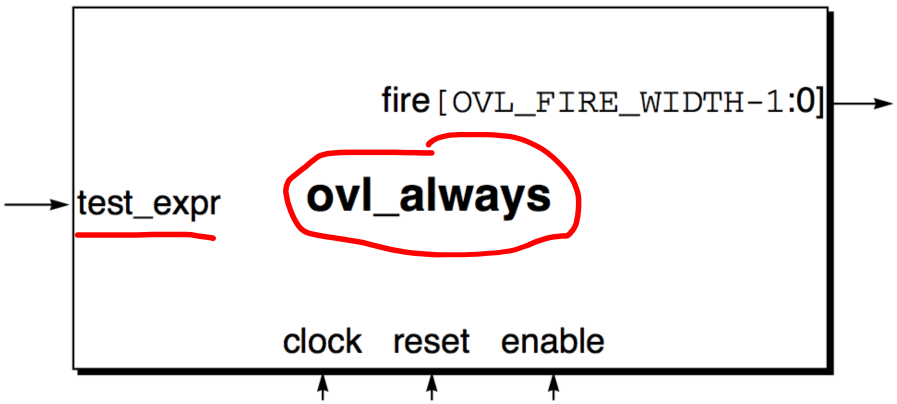
- **Invariant Assertion Window:**
  Assertion P is checked and expected to hold after the **start event** occurs and continues to be checked and is expected to hold until the **end event.**

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

EXAMPLE OVL CHECKERS

ovl_always

Checks that the value of an expression is TRUE.



Parameters/Generics:
severity_level
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

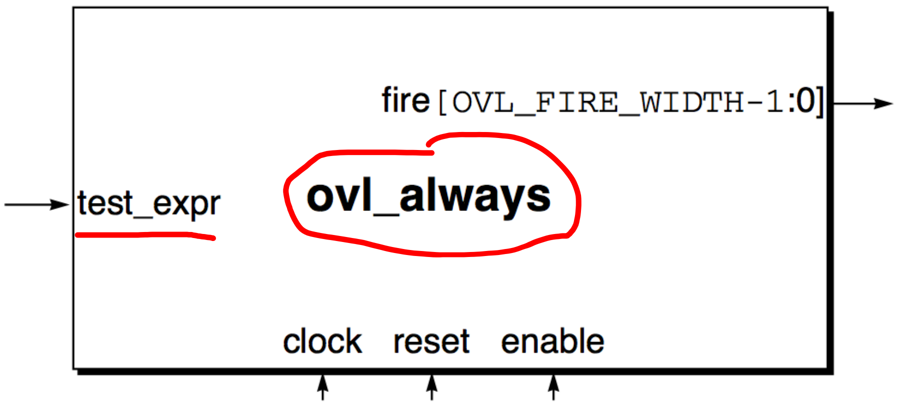
Class: 1-cycle assertion

Syntax

```
ovl_always
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

ovl_always

Checks that the value of an expression is TRUE.



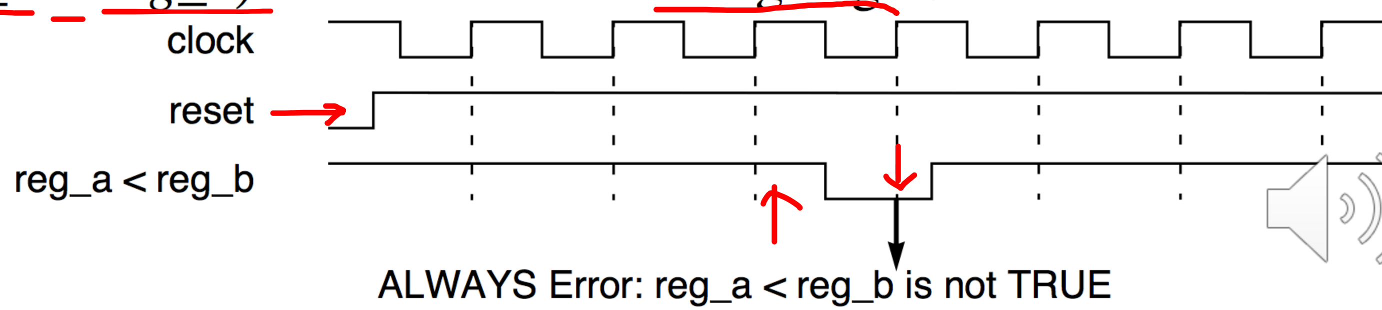
Parameters/Generics:
severity_level
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: 1-cycle assertion

Syntax

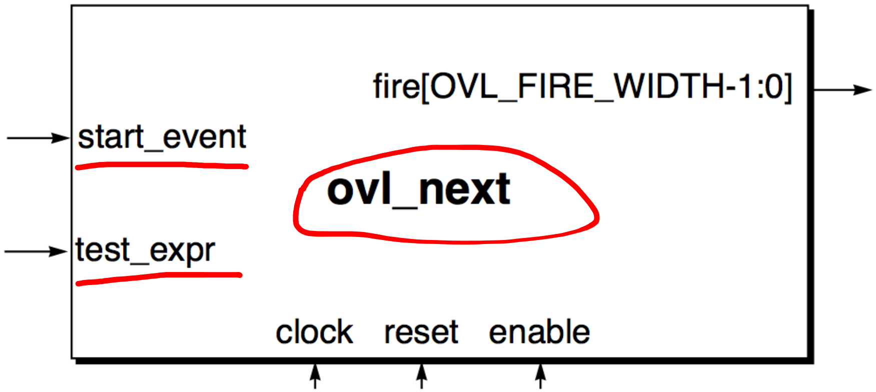
```
ovl_always
      [#(severity_level, property_type, msg, coverage_level, clock_edge,
        reset_polarity, gating_type)]
   instance_name (clock, reset, enable, test_expr, fire);
```

Checks that (*reg_a* < *reg_b*) is TRUE at each rising edge of *clock*.



ovl_next

Checks that the value of an expression is TRUE a specified number of cycles after a start event.



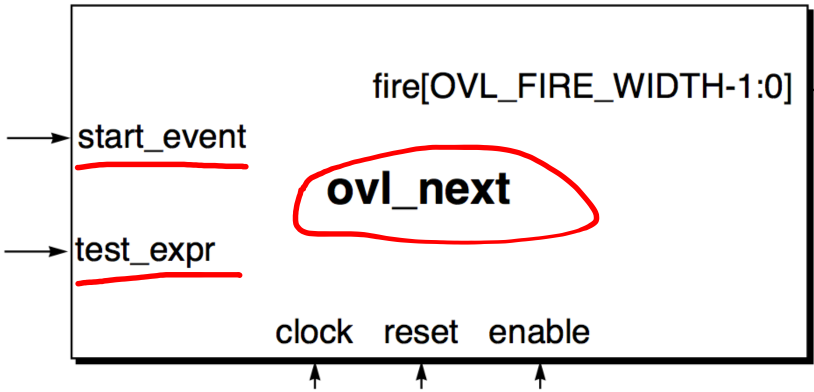
Parameters/Generics:

severity_level *msg*
num_cks *coverage_level*
check_overlapping *clock_edge*
check_missing_start *reset_polarity*
property_type *gating_type*

Class: *n*-cycle assertion

ovl_next

Checks that the value of an expression is TRUE a specified number of cycles after a start event.



Parameters/Generics:

severity_level
num_cks
check_overlapping
check_missing_start
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

Syntax

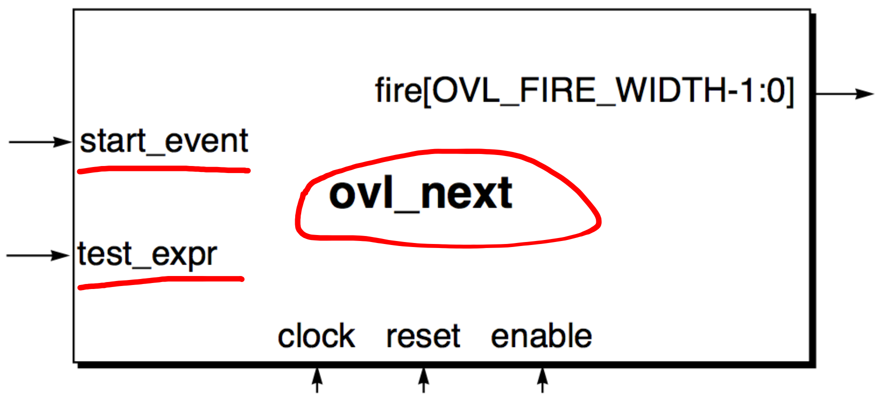
```
ovl_next
    [#(severity_level, num_cks, check_overlapping, check_missing_start,
      property_type, msg, coverage_level, clock_edge, reset_polarity,
      gating_type)]
  instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Number of cycles after start_event is TRUE to wait to check that the value of test_expr is TRUE. Default: 1.

28

ovl_next

Checks that the value of an expression is TRUE a specified number of cycles after a start event.



Parameters/Generics:

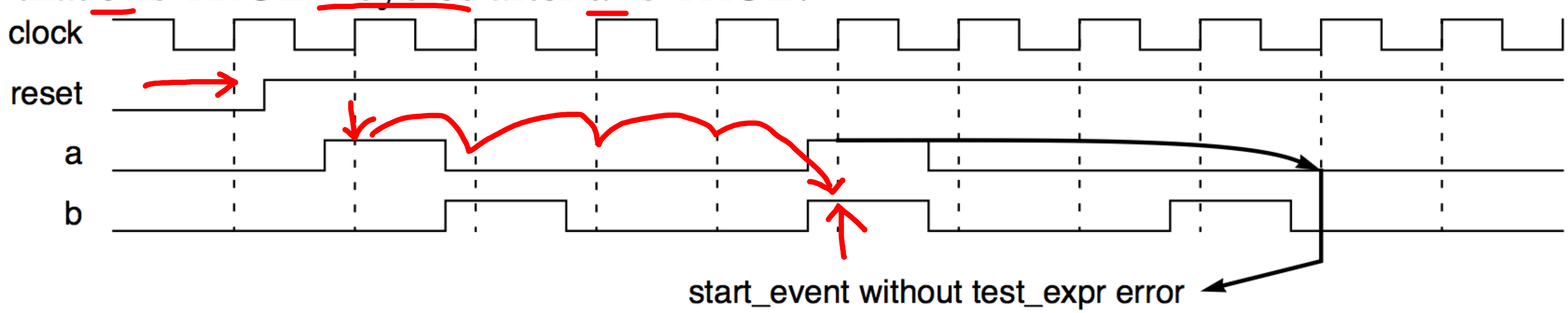
severity_level
num_cks
check_overlapping
check_missing_start
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

Syntax

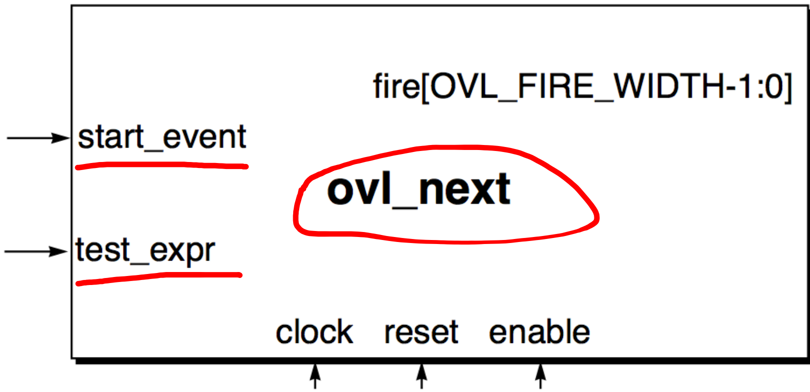
```
ovl_next
      [#(severity_level, num_cks, check_overlapping, check_missing_start,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Checks that *b* is TRUE 4 cycles after *a* is TRUE.



ovl_next

Checks that the value of an expression is TRUE a specified number of cycles after a start event.



Parameters/Generics:

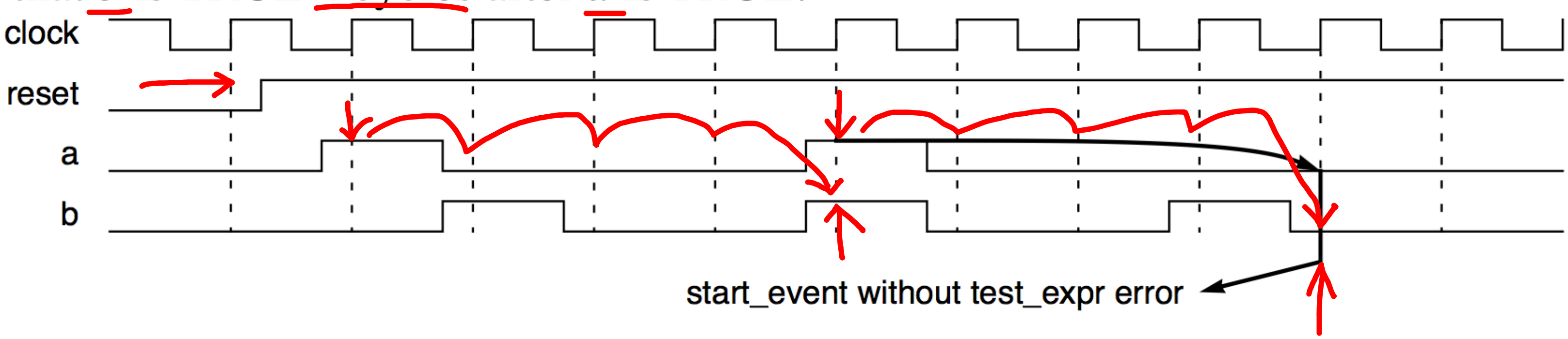
severity_level
num_cks
check_overlapping
check_missing_start
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

Syntax

```
ovl_next
      [#(severity_level, num_cks, check_overlapping, check_missing_start,
         property_type, msg, coverage_level, clock_edge, reset_polarity,
         gating_type)]
   instance_name (clock, reset, enable, start_event, test_expr, fire);
```

Checks that *b* is TRUE 4 cycles after *a* is TRUE.



OVL QUICK REFERENCE (www.eda.org/ovl) Last updated: 25th May 2007

TYPE	NAME	PARAMETERS	PORTS	DESCRIPTION
Single-Cycle	ovl_always	#(severity_level, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must always hold
Two Cycles	ovl_always_on_edge	#(severity_level, edge_type property_type, msg, coverage_level)	(clock, reset, enable, sampling_event, test_expr, fire)	test_expr is true immediately following the specified edge (edge_type: 0=no-edge, 1=pos, 2=neg, 3=any)
Event-bound	ovl_arbiter	#(severity_level, width, priority_width, min_cks, max_cks, arbitration_rule, priority_check, single_cnt_check, property_type, msg, coverage_level)	(clock, reset, enable, reqs, gnts, priorities, fire)	provides grants in response to requests, as per specified arbitration scheme and within a specified time window
Single-Cycle	ovl_bits	#(severity_level, width, asserted, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	checks number of asserted (or deasserted) bits is within a specified range
n-Cycles	ovl_change	#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, fire)	test_expr must change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
Single-Cycle	ovl_code_distance	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr1, test_expr2, fire)	checks hamming distance between two expressions
n-Cycles	ovl_cycle_sequence	#(severity_level, num_cks, necessary_condition, property_type, msg, coverage_level)	(clock, reset, enable, event_sequence, fire)	if the initial sequence holds, the final sequence must also hold (necessary_condition: 0=trigger-on-most, 1=trigger-on-first, 2=trigger-on-first-unpipelined)
Two Cycles	ovl_decrement	#(severity_level, width, value property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	if test_expr changes, it must decrement by the value parameter (modulo 2^width)
Two Cycles	ovl_delta	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	if test_expr changes, the delta must be >min and <=max
Single Cycle	ovl_even_parity	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must have an even parity, i.e. an even number of bits asserted
Event-bound	ovl_fifo	#(severity_level, width, depth, pass_thru, registered, enq_latency, deq_latency, preload_count, high_water_mark, value_check, property_type, msg, coverage_level)	(clock, reset, enable, enq, deq, full, empty, enq_data, deq_data, preload, fire)	checks data integrity of a FIFO and ensures that the FIFO does not overflow or underflow
Two Cycles	ovl_fifo_index	#(severity_level, depth, push_width, pop_width, property_type, msg, coverage_level, simultaneous_push_pop)	(clock, reset, enable, push, pop, fire)	FIFO pointers should never overflow or underflow
n-Cycles	ovl_frame	#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, fire)	test_expr must not hold before min_cks cycles, but must hold at least once by max_cks cycles (action_on_new_start: 0=ignore, 1=restart, 2=error)
n-Cycles	ovl_handshake	#(severity_level, min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length, property_type, msg, coverage_level)	(clock, reset, enable, req, ack, fire)	req and ack must follow the specified handshaking protocol
n-Cycles	ovl_hold_value	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, value, fire)	once test_expr matches value, test_expr doesn't change value until a specified event
Single-Cycle	ovl_implication	#(severity_level, property_type, msg, coverage_level)	(clock, reset, enable, antecedent_expr, consequent_expr, fire)	if antecedent_expr holds then consequent_expr must hold in the same cyle
Two Cycles	ovl_increment	#(severity_level, width, value property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	if test_expr changes, it must increment by the value parameter (modulo 2^width)
Event-bound	ovl_memory_async	#(severity_level, data_width, addr_width, mem_size, addr_check, init_check, one_read_check, one_write_check, value_check, property_type, msg, coverage_level)	(reset, enable, start_addr, end_addr, ren, raddr, rdata, wen, waddr, wdata, fire)	ensures the integrity of accesses to an asynchronous memory
Event-bound	ovl_memory_sync	#(severity_level, data_width, addr_width, mem_size, pass_thru, addr_check, init_check, conflict_check, one_read_check, one_write_check, value_check, property_type, msg, coverage_level)	(r_clock, w_clock, reset, enable, start_addr, end_addr, ren, raddr, rdata, wen, waddr, wdata, fire)	ensures the integrity of accesses to an synchronous memory
n-Cycles	ovl_multiport_fifo	#(severity_level, width, depth, enq_count, deq_count, pass_thru, registered, enq_latency, deq_latency, preload_count, high_water_mark, full_check, empty_check, value_check, property_type, msg, coverage_level)	(clock, reset, enable, enq, deq, enq_data, deq_data, full, empty, preload, fire)	ensures data integrity of a FIFO with multiple enqueue and deque ports, and checks underflow and overflow
Single-Cycle	ovl_mutex	#(severity_level, width, invert_mode, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	ensures that the bits of an expression are mutually exclusive
Single-Cycle	ovl_never	#(severity_level, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must never hold
Single-Cycle	ovl_never_unknown	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, qualifier, test_expr, fire)	test_expr must never be an unknown value, just boolean 0 or 1
Combinatorial	ovl_never_unknown_async	#(severity_level, width, property_type, msg, coverage_level)	(reset, enable, test_expr, fire)	test_expr must never go to an unknown value asynchronously, it must remain boolean 0 or 1
n-Cycles	ovl_next	#(severity_level, num_cks, check_overlapping, check_missing_start, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, fire)	test_expr must hold num_cks cycles after start_event holds
Event-bound	ovl_next_state	#(severity_level, width, next_count, min_hold, max_hold, disallow, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, curr_state, next_state, fire)	ensures expression transitions only to specified values
Event-bound	ovl_no_contention	#(severity_level, width, num_drivers, min_quiet, max_quiet, property_type, msg, coverage_level)	(reset, enable, test_expr, driver_enables, fire)	ensures that a bus is driven according to specified contention rules
Two Cycles	ovl_no_overflow	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	if test_expr is at max, in the next cycle test_expr must be >min and <=max
Two Cycles	ovl_no_transition	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, start_state, next_state, fire)	if test_expr==start_state, in the next cycle test_expr must not change to next_state
Two Cycles	ovl_no_underflow	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	if test_expr is at min, in the next cycle test_expr must be >=min and <max
Single-Cycle	ovl_odd_parity	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must have an odd parity, i.e. an odd number of bits asserted
Single-Cycle	ovl_one_cold	#(severity_level, width, inactive property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must be one-cold i.e. exactly one bit set low (inactive: 0=also-all-zero, 1=also-all-ones, 2=pure-one-cold)
Single-Cycle	ovl_one_hot	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must be one-hot i.e. exactly one bit set high
Combinatorial	ovl_proposition	#(severity_level, property_type, msg, coverage_level)	(reset_n, enable, test_expr, fire)	test_expr must hold asynchronously (not just at a clock edge)
Two Cycles	ovl_quiescent_state	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, state_expr, check_value, sample_event, fire)	state_expr must equal check_value on a rising edge of sample_event (also checked on rising edge of `OVL_END_OF_SIMULATION)
Single-Cycle	ovl_range	#(severity_level, width, min, max, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must be >=min and <=max
Event-bound	ovl_reg_loaded	#(severity_level, width, start_count, end_count, property_type, msg, coverage_level)	(clock, reset, enable, start_event, end_event, src_expr, dest_expr, fire)	ensures that a register is loaded with source data within a specified time window
n-Cycles	ovl_req_ack_unique	#(severity_level, min_cks, max_cks, method, property_type, msg, coverage_level)	(clock, reset, enable, req, ack, fire)	ensures every request receives a corresponding acknowledge in a specified time window
n-Cycles	ovl_req_requires	#(severity_level, min_cks, max_cks, property_type, msg, coverage_level)	(clock, reset, enable, req_trigger, req_follower, resp_leader, resp_trigger, fire)	ensures that every request event initiates a valid request-response event sequence that finishes within a specified time window
n-Cycles	ovl_stack	#(severity_level, width, depth, push_latency, pop_latency, high_water_mark, property_type, msg, coverage_level)	(clock, reset, enable, push, pop, full, empty, push_data, pop_data, fire)	ensures the data integrity of a stack and ensures that the stack does not overflow or underflow
n-Cycles	ovl_time	#(severity_level, num_cks, action_on_new_start, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, fire)	test_expr must hold for num_cks cycles after start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
Two Cycles	ovl_transition	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, start_state, next_state, fire)	if test_expr changes from start_state, then it can only change to next_state
n-Cycles	ovl_unchange	#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, fire)	test_expr must not change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
n-Cycles	ovl_valid_id	#(severity_level, width, min_cks, max_cks, max_instances, max_ids, max_instances_per_id, instance_count_width, property_type, msg, coverage_level)	(clock, reset, enable, issued, issued_count, returned, flush, issued_id, returned_id, flush_id, fire)	Ensures that each issued ID is returned within a specified time window; that returned IDs match issued IDs; and that the issued and outstanding IDs do not exceed specified limits.
Single-Cycle	ovl_value	#(severity_level, width, num_values, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, vals, disallow, fire)	ensures the value of an expression either matches a value in a specified list or does not match any value in the list
n-Cycles	ovl_width	#(severity_level, min_cks, max_cks, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must hold for between min_cks and max_cks cycles
Event-bound	ovl_win_change	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, end_event, fire)	test_expr must change between start_event and end_event
Event-bound	ovl_window	#(severity_level, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, end_event, fire)	test_expr must hold after the start_event and up to (and including) the end_event
Event-bound	ovl_win_unchange	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, start_event, test_expr, end_event, fire);	test_expr must not change between start_event and end_event
Single-Cycle	ovl_zero_one_hot	#(severity_level, width, property_type, msg, coverage_level)	(clock, reset, enable, test_expr, fire)	test_expr must be one-hot or zero, i.e. at most one bit set high

PARAMETERS

severity_level

`OVL_FATAL

`OVL_ERROR

`OVL_WARNING

`OVL_INFO

property_type

`OVL_ASSERT

`OVL_ASSUME

`OVL_IGNORE

msg descriptive string

USING OVL

+define+OVL_ASSERT_ON

+define+OVL_MAX_REPORT_ERROR=1

+define+OVL_INIT_MSG

+define+OVL_INIT_COUNT=<tbench>.ovl_init_count

+libext+.v+.vlib

-y <OVL_DIR>/std_ovl

+incdir+<OVL_DIR>/std_ovl

DESIGN ASSERTIONS

Monitors internal signals & Outputs

Examples

* One hot FSM
* Hit default case items
* FIFO / Stack
* Counters (overflow/increment)
* FSM transitions
* X checkers (ovl_never_unknown)

INPUT ASSUMPTIONS

Restricts environment

Examples

* One hot inputs
* Range limits e.g. cache sizes
* Stability e.g. cache sizes
* No back-to-back reqs
* Handshaking sequences
* Bus protocol

http://www.accellera.org/downloads/standards/ovl

OVL QUICK REFERENCE (www.eda.org/ovl) Last updated: 25th May 2007

TYPE	NAME
n-Cycles	ovl_hold_value
Single-Cycle	ovl_implication
Two Cycles	ovl_increment
Event-bound	ovl_memory_async
Event-bound	ovl_memory_sync
n-Cycles	ovl_multiport_fifo
Single-Cycle	ovl_mutex
Single-Cycle	ovl_never
Single-Cycle	ovl_never_unknown
Combinatorial	ovl_never_unknown_async
n-Cycles	ovl_next
Event-bound	ovl_next_state
Event-bound	ovl_no_contention
Two Cycles	ovl_no_overflow
Two Cycles	ovl_no_transition
Two Cycles	ovl_no_underflow
Single-Cycle	ovl_odd_parity
Single-Cycle	ovl_one_cold

PARAMETERS

severity_level

`OVL_FATAL

`OVL_ERROR

`OVL_WARNING

`OVL_INFO

property_type

`OVL_ASSERT

`OVL_ASSUME

`OVL_IGNORE

msg descriptive string

+incdir+<OVL_DIR>/std_ovl

* FSM transitions
* X checkers (ovl_never_unknown)

INPUT ASSUMPTIONS

Restricts environment

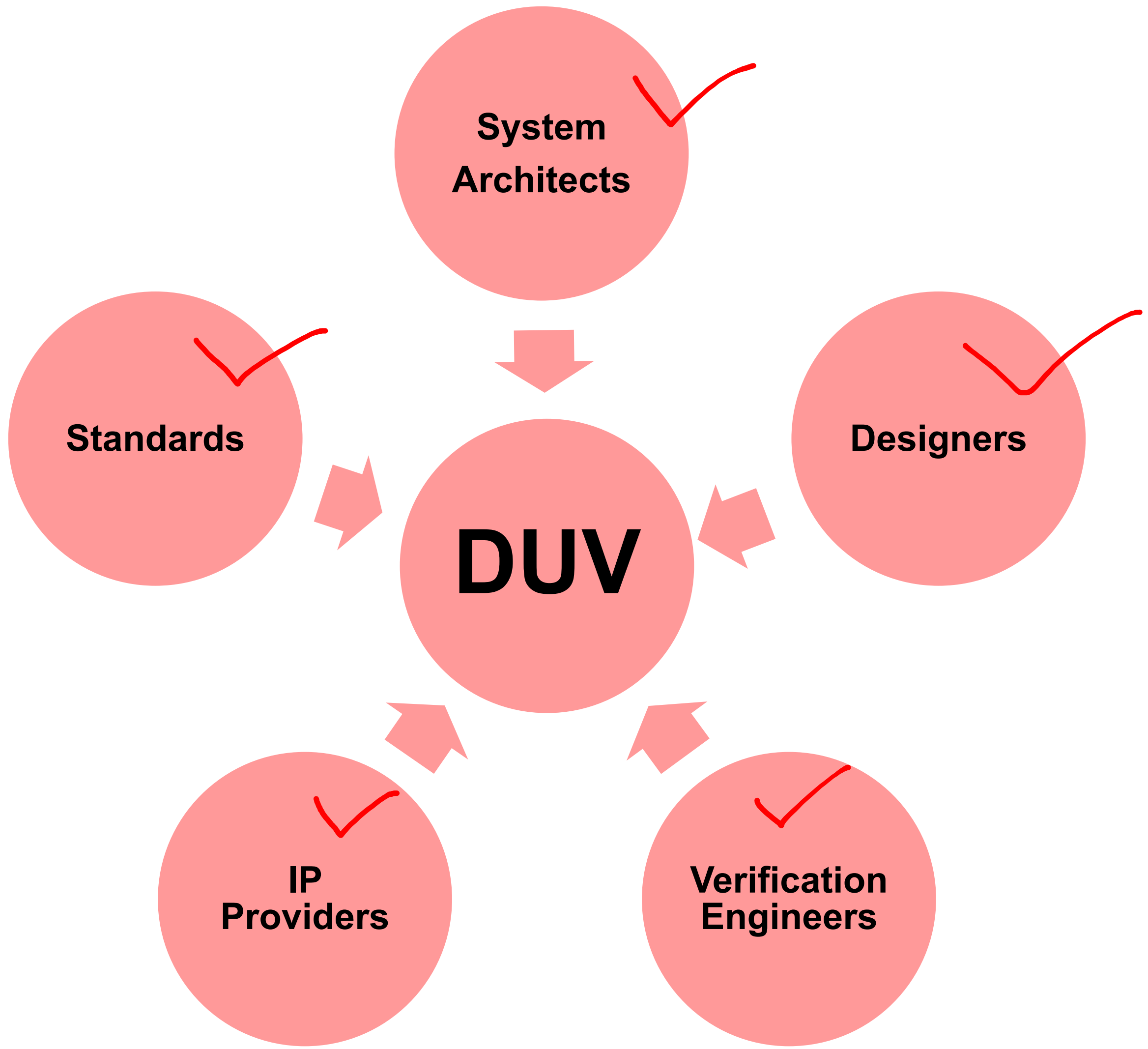
Examples

* One hot inputs
* Range limits e.g. cache sizes
* Stability e.g. cache sizes
* No back-to-back reqs
* Handshaking sequences
* Bus protocol

http://www.accellera.org/downloads/standards/ovl

WHERE DO ASSERTIONS COME FROM?

Who writes the assertions?



Implementation Assertions

- Also called "design" assertions.
  - Specified by the designer.
- Encode designer's assumptions.
  - Interface assertions:
    - Catch different interpretations between individual designers.
  - Conditions of design misuse or design faults:
    - detect buffer over/under flow
    - detect buffer read & write at the same time when only one is allowed
- Implementation assertions can detect discrepancies between design assumptions and implementation.
- But implementation assertions won't detect discrepancies between functional intent and design!

(Remember: Verification Independence!)

Specification Assertions

- Also called "intent" assertions
  - Often high-level properties.
- Specified by architects, verification engineers, IP providers, standards.
- Encode expectations of the design based on understanding of functional intent.
- Provide a "functional error detection" mechanism.
- Supplement error detection performed by self-checking testbenches.
  - Instead of using (implementing) a monitor and checker, in many cases writing a block-level assertion can be much simpler.

End of Part I

COMS30026 Design Verification

Assertion-based Verification

Kerstin Eder

Trustworthy Systems Laboratory

https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/