

COMS30026 Design Verification

Assertion-based Verification

Kerstin Eder

Trustworthy Systems Laboratory

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

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

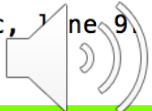
```
1 #include <stdio.h>
2 #include <assert.h>
3
4 int mysquare(int n) {
5     int s = 0;
6     int i = 0;
7     int k = 0; /* assertion variable to count the number of times in the loop */
8
9     assert (n >= 0); // Pre-condition to catch invalid input
10
11     assert (s == k*n && i==k); // Invariant 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     assert (k == n); // Post-condition to catch a mistaken final state of the loop
21
22     assert (s == k*n && i==k); // Invariant to catch errors in the loop computation
23
24     assert (s == n * n); // Check desired post-condition
25
26     return s;
27 }
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 }
```



Assertions in C code

```
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     assert (k == n); // Post-condition to catch a mistaken final state of the loop
21
22     assert (s == k*n && i==k); // Invariant to catch errors in the loop computation
23
24     assert (s == n * n); // Check desired post-condition
25
26     return s;
27 }
28
29
30 int main() {
31     int n = -4;
32     int square = 0;
33
34     printf("n = %d\n", n);
35     square = mysquare(n);
36     printf("n^2 = %d\n", square);
37
38     return 0;
39 }
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

```
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     assert (k == n); // Post-condition to catch a mistaken final state of the loop
21
22     assert (s == k*n && i==k); // Invariant to catch errors in the loop computation
23
24     assert (s == n * n); // Check desired post-condition
25
26     return s;
27 }
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
```

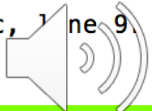
```
[cskie@it000908:SLIDES$ ./mysquare
```

```
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*.”



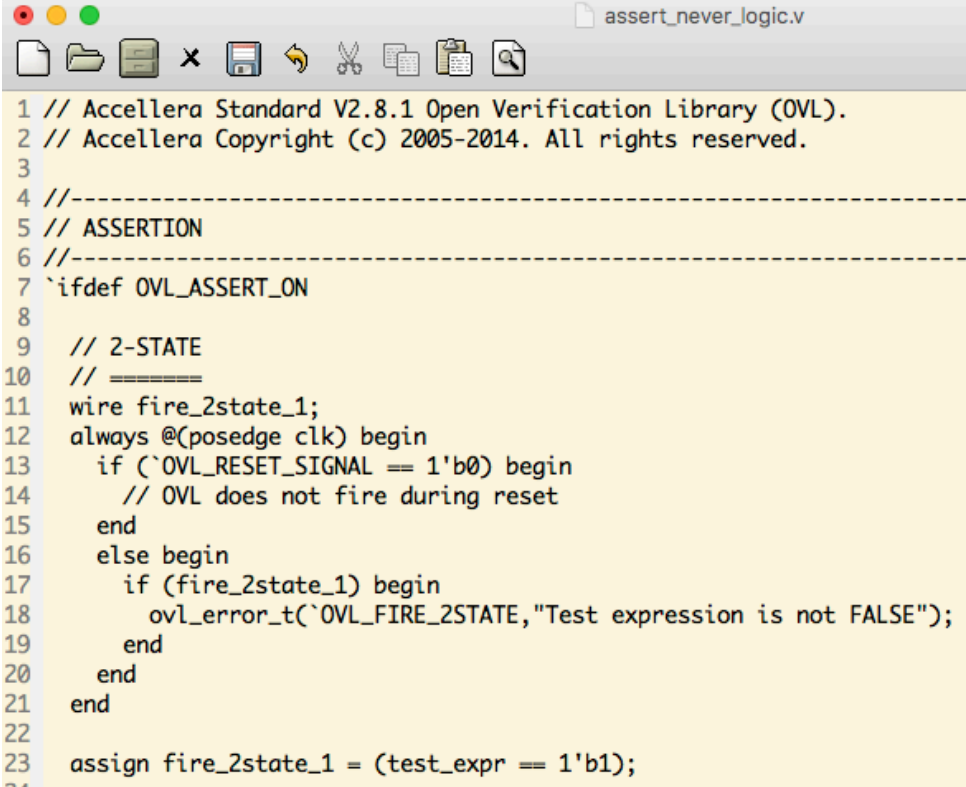
HW Assertions

- **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

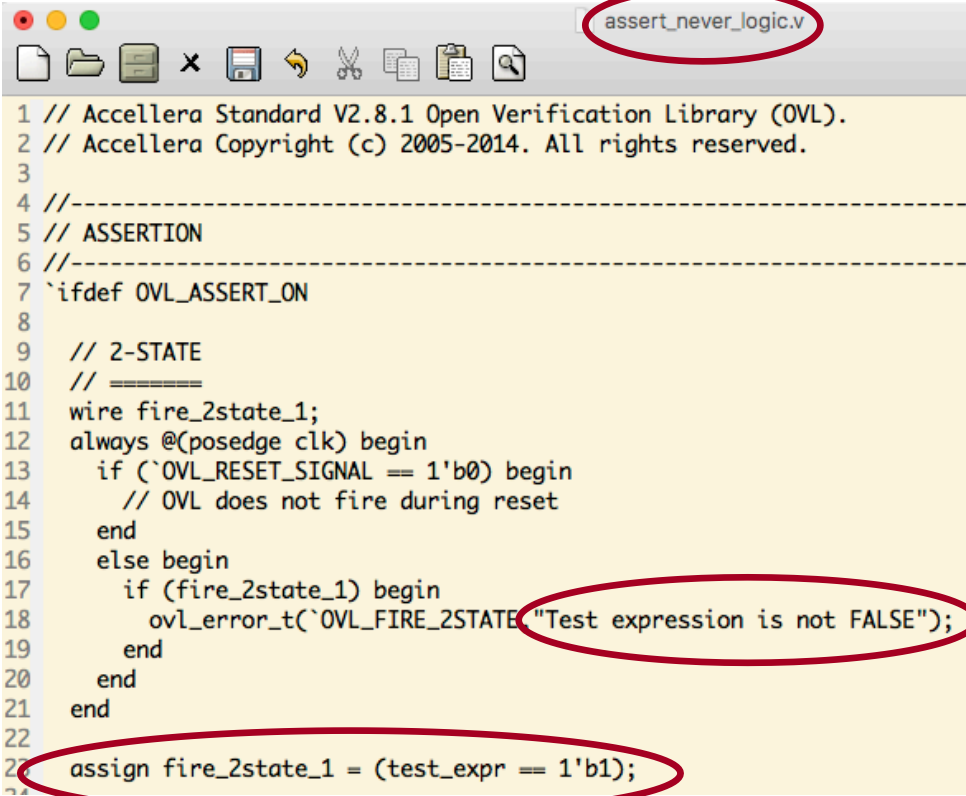


```
assert_never_logic.v
1 // Accellera Standard V2.8.1 Open Verification Library (OVL).
2 // Accellera Copyright (c) 2005-2014. All rights reserved.
3
4 //-----
5 // ASSERTION
6 //-----
7 `ifdef OVL_ASSERT_ON
8
9     // 2-STATE
10    // =====
11    wire fire_2state_1;
12    always @(posedge clk) begin
13        if (`OVL_RESET_SIGNAL == 1'b0) begin
14            // OVL does not fire during reset
15        end
16        else begin
17            if (fire_2state_1) begin
18                ovl_error_t(`OVL_FIRE_2STATE,"Test expression is not FALSE");
19            end
20        end
21    end
22
23    assign fire_2state_1 = (test_expr == 1'b1);
24
```



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

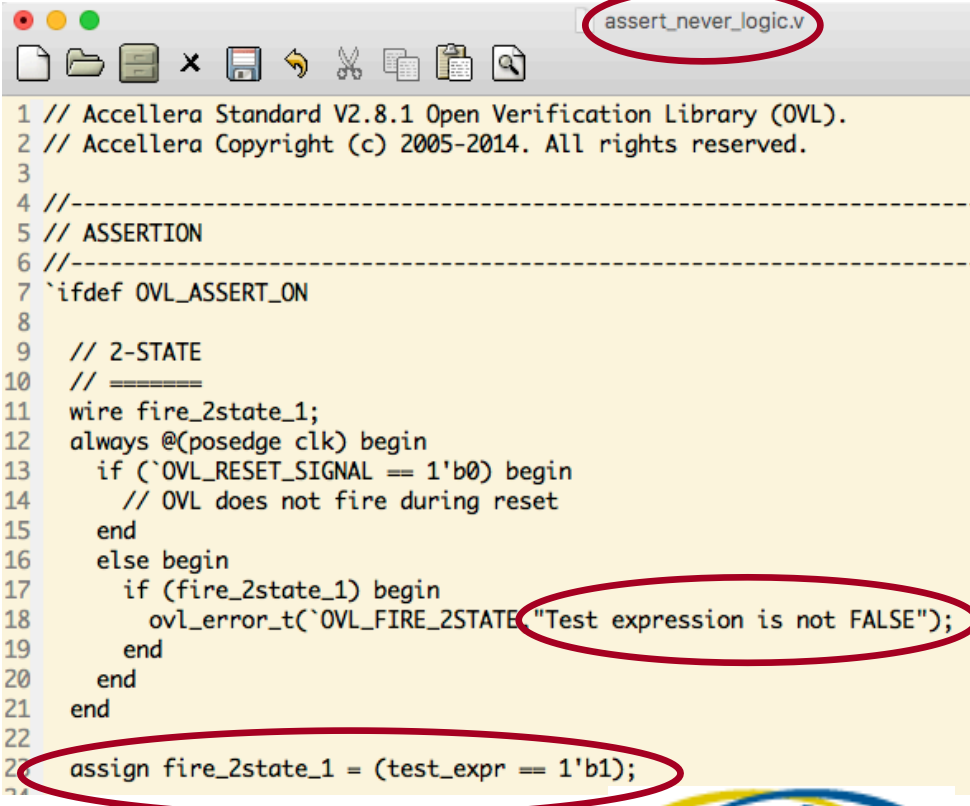


```
1 // Accellera Standard V2.8.1 Open Verification Library (OVL).
2 // Accellera Copyright (c) 2005-2014. All rights reserved.
3
4 //-----
5 // ASSERTION
6 //-----
7 `ifdef OVL_ASSERT_ON
8
9     // 2-STATE
10    // =====
11    wire fire_2state_1;
12    always @(posedge clk) begin
13        if (`OVL_RESET_SIGNAL == 1'b0) begin
14            // OVL does not fire during reset
15        end
16        else begin
17            if (fire_2state_1) begin
18                ovl_error_t(`OVL_FIRE_2STATE "Test expression is not FALSE");
19            end
20        end
21    end
22
23    assign fire_2state_1 = (test_expr == 1'b1);
24 `endif
```



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).



```
1 // Accellera Standard V2.8.1 Open Verification Library (OVL).
2 // Accellera Copyright (c) 2005-2014. All rights reserved.
3
4 //-----
5 // ASSERTION
6 //-----
7 `ifdef OVL_ASSERT_ON
8
9     // 2-STATE
10    // =====
11    wire fire_2state_1;
12    always @(posedge clk) begin
13        if (`OVL_RESET_SIGNAL == 1'b0) begin
14            // OVL does not fire during reset
15        end
16        else begin
17            if (fire_2state_1) begin
18                ovl_error_t(`OVL_FIRE_2STATE "Test expression is not FALSE");
19            end
20        end
21    end
22
23    assign fire_2state_1 = (test_expr == 1'b1);
```

OVL is an
Accellera Standard

<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.

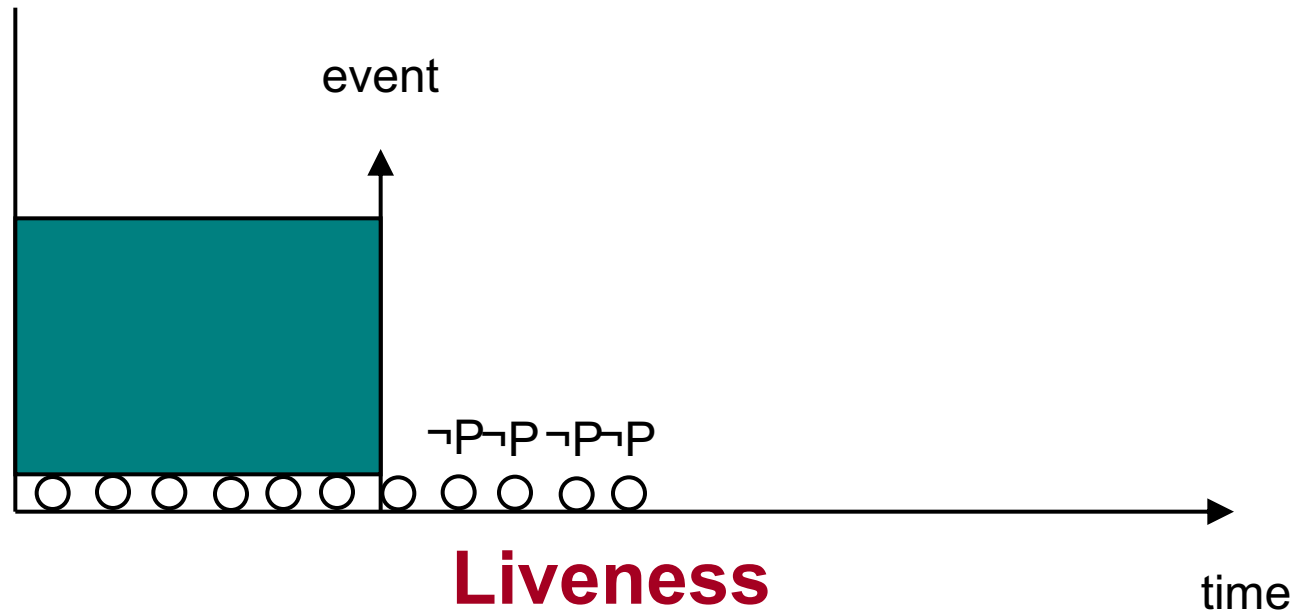


Liveness Properties

- **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 et al.: Assertion-Based Design. 2nd Edition, Kluwer, 2010.]



Liveness

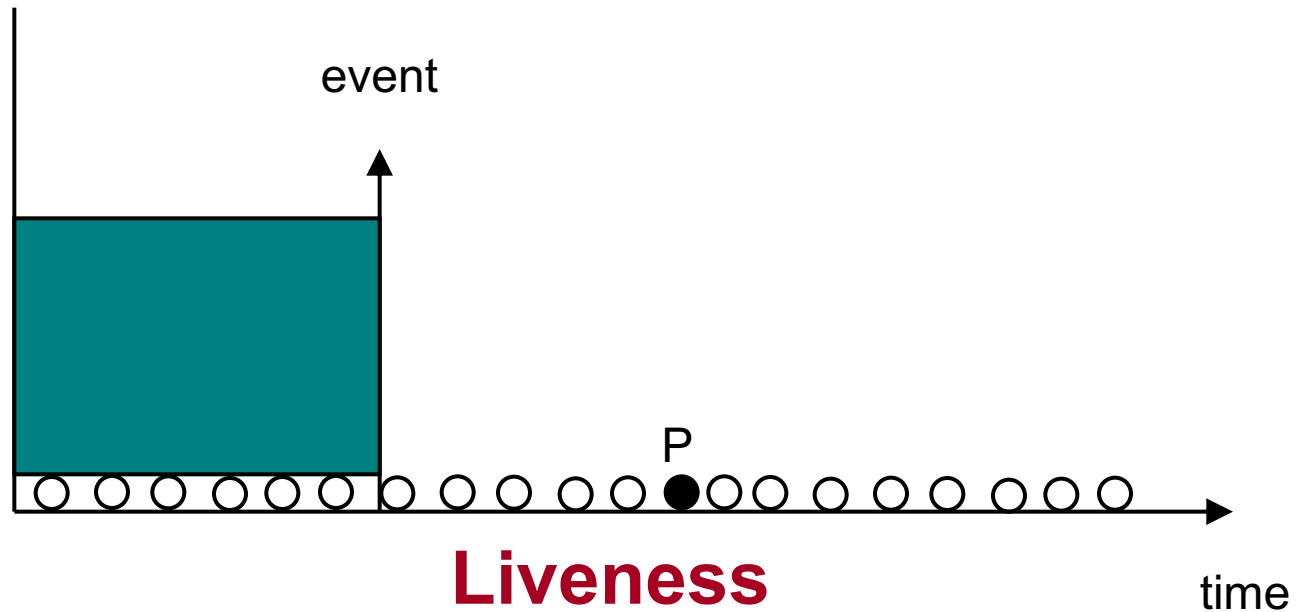


- Assertion P must **eventually** be valid after the event occurs.

[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](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](https://commons.wikimedia.org/w/index.php?curid=103762)





Liveness Properties



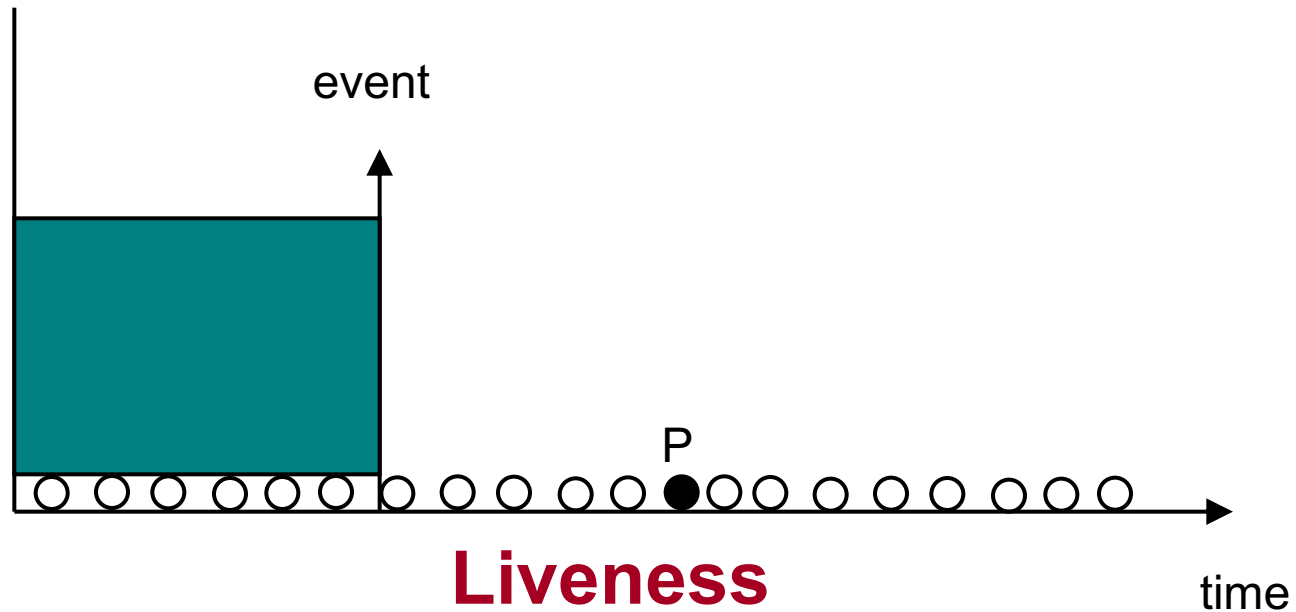
- **Liveness:** Something good eventually happens
 - The decoding algorithm **eventually** terminates.
 - Every request is **eventually** acknowledged.
- More formally: *A liveness property is a property for which any finite path can be extended to a path satisfying the property.* [Foster et al.: Assertion-Based 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

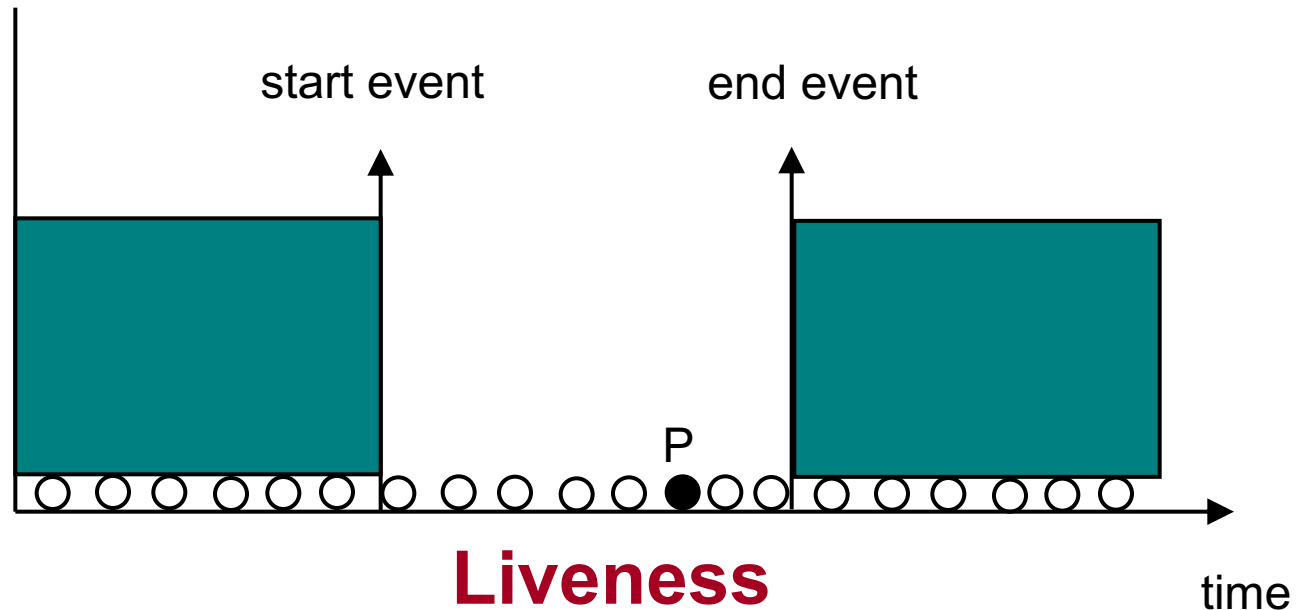


- Assertion P must **eventually** be valid after the event occurs

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



Bounded Liveness

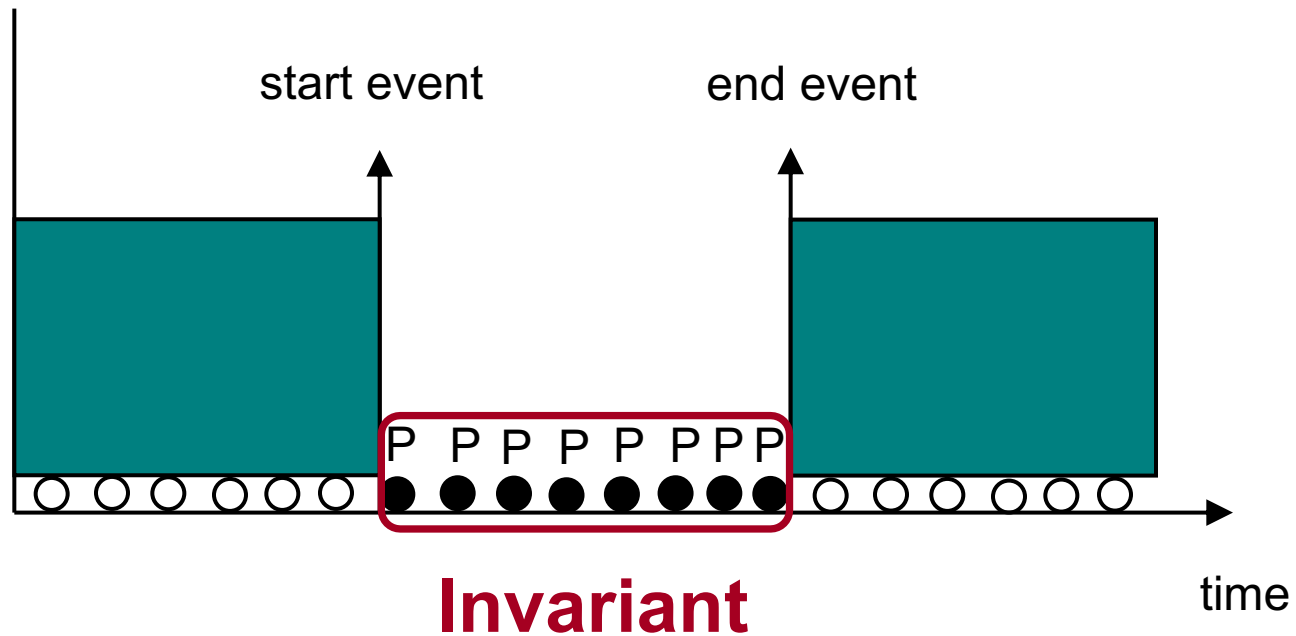


- Assertion P must **eventually** be valid after the **start event trigger** occurs and **before the end event trigger** occurs.

[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**.

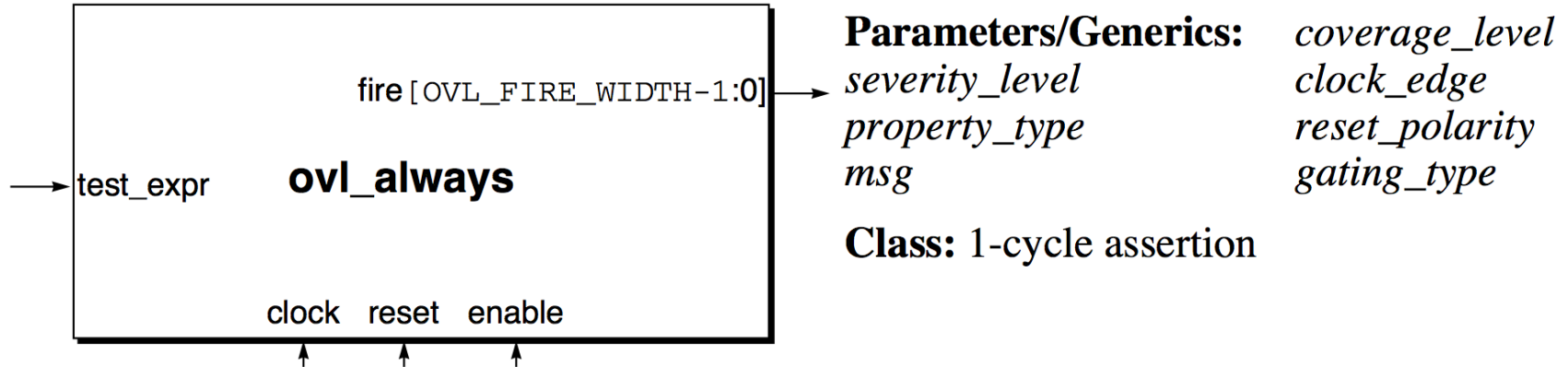


EXAMPLE OVL CHECKERS



ovl_always

Checks that the value of an expression is TRUE.



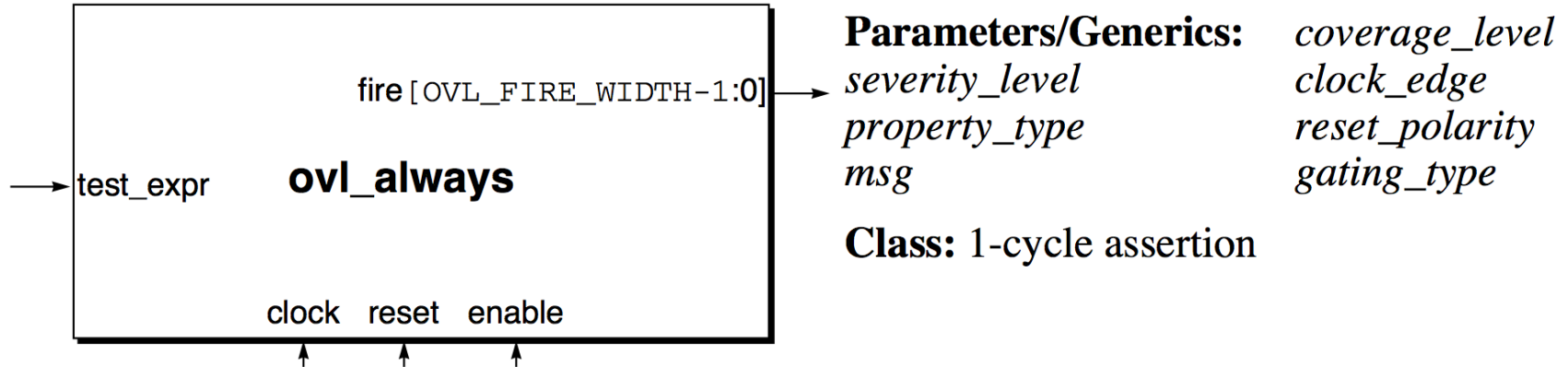
Syntax

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.

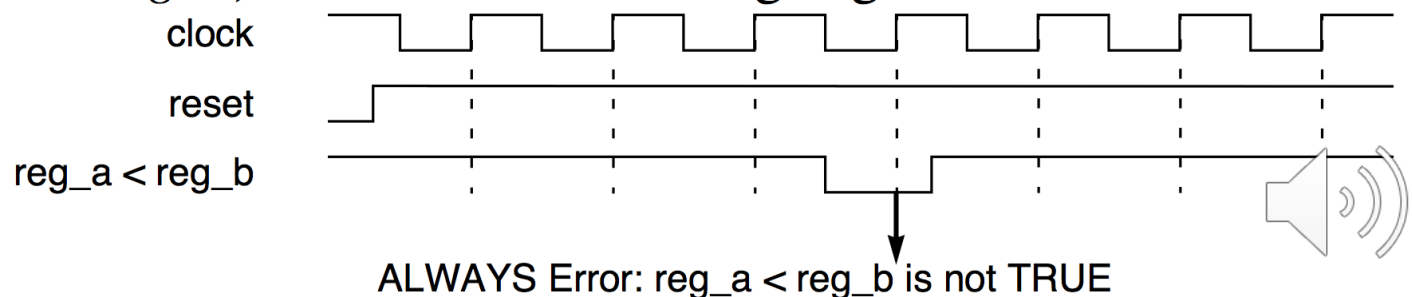


Syntax

ovl_always

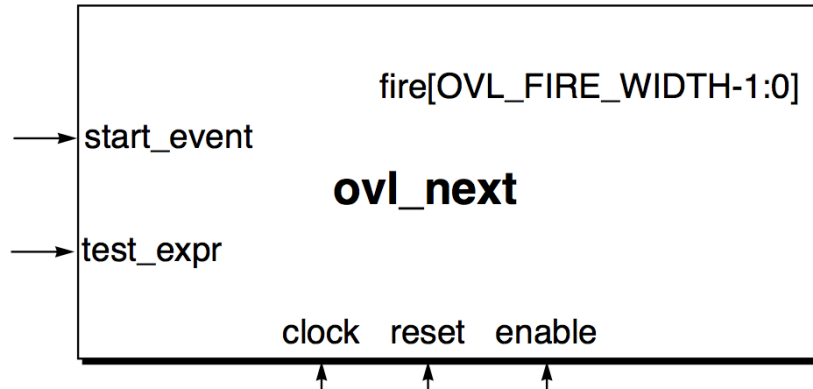
```
[#(severity_level, property_type, msg, coverage_level, clock_edge,  
    reset_polarity, gating_type)]  
instance_name (clock, reset, enable, test_expr, fire);
```

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

num_cks

check_overlapping

check_missing_start

property_type

msg

coverage_level

clock_edge

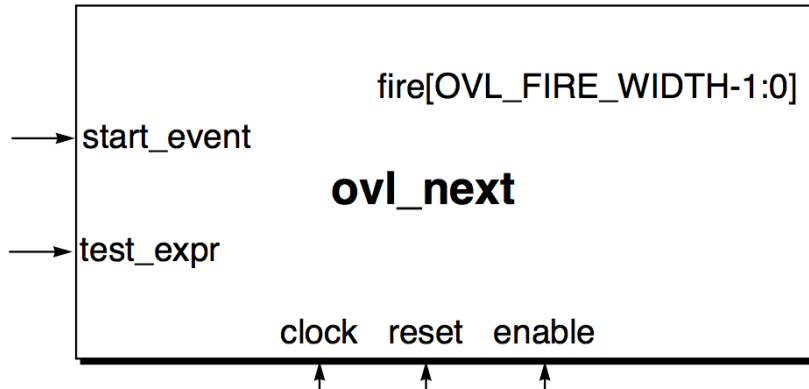
reset_polarity

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

msg

num_cks

coverage_level

check_overlapping

clock_edge

check_missing_start

reset_polarity

property_type

gating_type

Class: *n*-cycle assertion

Syntax

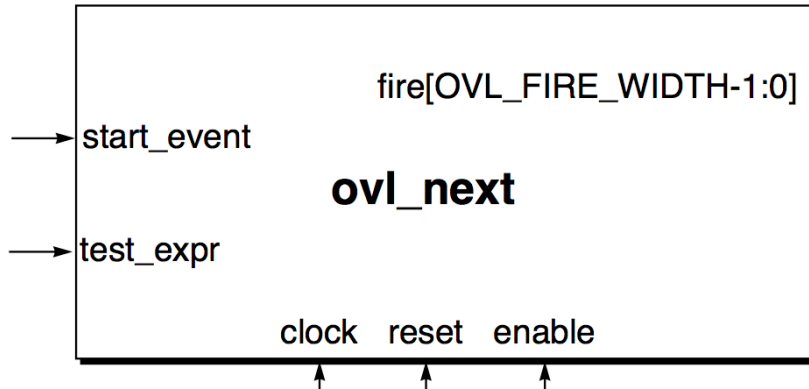
ovl_next

```
[#(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.

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

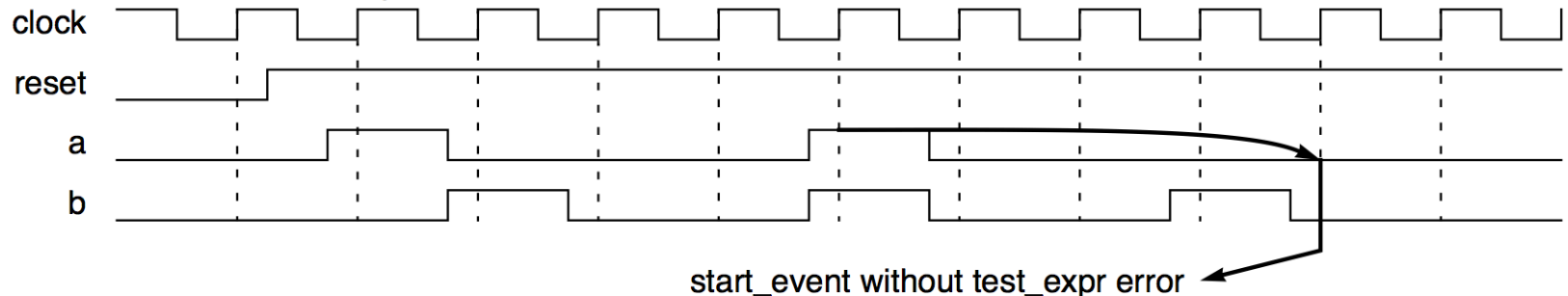
Class: *n*-cycle assertion

Syntax

ovl_next

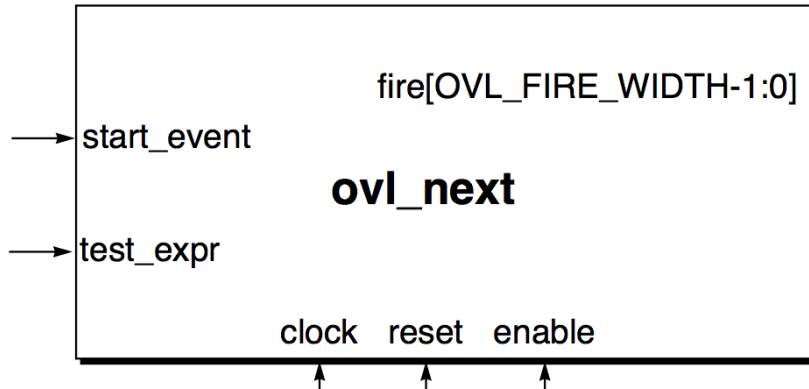
```
[#(severity_level, num_cks, check_overlapping, check_missing_start,  
   property_type, msg, coverage_level, clock_edge, reset_polarity,  
   gating_type)]  
instance_name (clock, reset, enable, start_event, test_expr, fire);
```

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

msg

num_cks

coverage_level

check_overlapping

clock_edge

check_missing_start

reset_polarity

property_type

gating_type

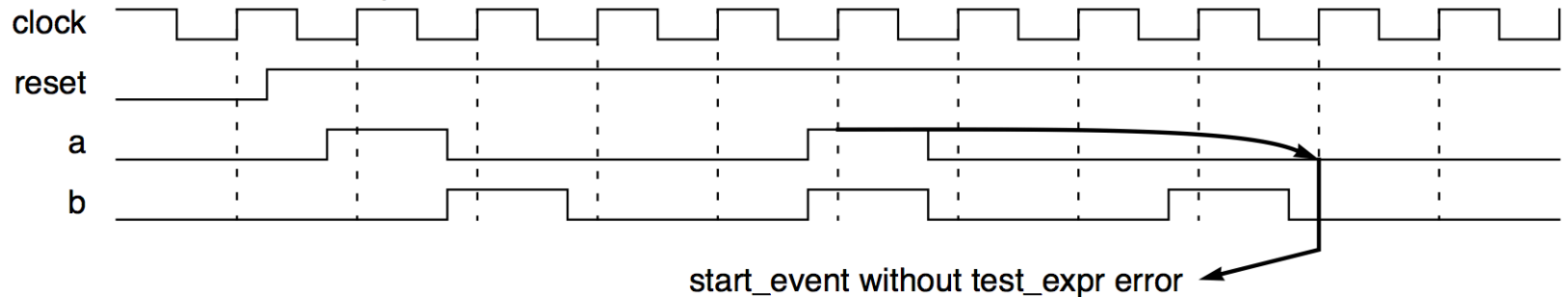
Class: *n*-cycle assertion

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.



TYPE	NAME	PARAMETERS	PORTS	DESCRIPTION
SingleCycle	<i>ovl_always</i>	<i>severity_level</i> (property_type, msg, coverage_level)	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must always hold
Two Cycles	<i>ovl_always_on_edge</i>	<i>severity_level</i> , <i>edge_type</i> (property_type, msg, coverage_level)	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>sampling</i> , <i>assert</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must immediately following the specified edge (e.g. type 0=on-edge, 1=tops, 2=eq, 3=any)
Event-bound	<i>ovl_atbit</i>	<i>severity_level</i> , <i>width</i> , <i>priority</i> , <i>width</i> , <i>min_cks</i> , <i>max_cks</i> , <i>abstraction</i> , <i>rule</i> , <i>priority</i> , <i>check</i> , <i>single_ckt</i> , <i>check</i> , <i>enforce</i> , <i>low</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>reqs</i> , <i>grts</i> , <i>priorities</i> , <i>fire</i>	provides grants in response to requests, as per specified arbitration scheme and within a specified time window
SingleCycle	<i>ovl_bits</i>	<i>severity_level</i> , <i>width</i> , <i>asserted</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	checks number of asserted (or deasserted) bits is within a specified range
nCycles	<i>ovl_change</i>	<i>severity_level</i> , <i>width</i> , <i>num_cks</i> , <i>action_on_new_start</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must change within num_cks of start_event (action_on_new_start: 0=ignore, 1=reset, 2=error)
SingleCycle	<i>ovl_cdb_distance</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr1</i> , <i>test_expr2</i> , <i>fire</i>	checks hamming distance between two expressions
nCycles	<i>ovl_cyle_sequence</i>	<i>severity_level</i> , <i>num_cks</i> , <i>necessary_condition</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>event_sequence</i> , <i>fire</i>	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-unless-did)
Two Cycles	<i>ovl_decrement</i>	<i>severity_level</i> , <i>width</i> , <i>value</i> (property_type, msg, coverage_level)	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	If <i>test_expr</i> changes, it must decrement by the value parameter (modulo 2 ^{width})
Two Cycles	<i>ovl_delta</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	If <i>test_expr</i> changes, the delta must be >min and <=max
SingleCycle	<i>ovl_even_parity</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must have an even parity, i.e. an even number of bits asserted
Event-bound	<i>ovl_fifo</i>	<i>severity_level</i> , <i>width</i> , <i>depth</i> , <i>pass_thru</i> , <i>registered</i> , <i>enq</i> , <i>latency</i> , <i>daq</i> , <i>latency</i> , <i>prod</i> , <i>count</i> , <i>high_water</i> , <i>mark</i> , <i>value</i> , <i>check</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>enq</i> , <i>daq</i> , <i>full</i> , <i>empty</i> , <i>enq_data</i> , <i>daq_data</i> , <i>prod</i> , <i>fire</i>	checks data integrity of a FIFO and ensures that the FIFO does not overflow or underflow
Two Cycles	<i>ovl_fifo_index</i>	<i>severity_level</i> , <i>depth</i> , <i>push</i> , <i>width</i> , <i>pop</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i> , <i>simultaneous</i> , <i>push</i> , <i>pop</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>push</i> , <i>pop</i> , <i>fire</i>	FIFO pointers should never overflow or underflow
nCycles	<i>ovl_frame</i>	<i>severity_level</i> , <i>min_cks</i> , <i>max_cks</i> , <i>action_on_new_start</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must not hold before min_cks cycles, but must hold at least once by max_cks cycles (action_on_new_start: 0=ignore, 1=reset, 2=error)
nCycles	<i>ovl_handshake</i>	<i>severity_level</i> , <i>min_ack</i> , <i>cycle_max_ack</i> , <i>cycle_req</i> , <i>req_ack_count</i> , <i>max_ack_length</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>req</i> , <i>ack</i> , <i>fire</i>	req and ack must follow the specified handshaking protocol
nCycles	<i>ovl_hold_value</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>value</i> , <i>fire</i>	once <i>test_expr</i> matches value, <i>test_expr</i> doesn't change value until a specified event
SingleCycle	<i>ovl_implication</i>	<i>severity_level</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>antecedent_expr</i> , <i>consequent_expr</i> , <i>fire</i>	if antecedent_expr holds then consequent_expr must hold in the same cycle
Two Cycles	<i>ovl_increment</i>	<i>severity_level</i> , <i>width</i> , <i>value</i> (property_type, msg, coverage_level)	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	If <i>test_expr</i> changes, it must increment by the value parameter (modulo 2 ^{width})
Event-bound	<i>ovl_memory_async</i>	<i>severity_level</i> , <i>data</i> , <i>width</i> , <i>add</i> , <i>width</i> , <i>mem_size</i> , <i>add_check</i> , <i>init</i> , <i>check</i> , <i>one_read</i> , <i>check</i> , <i>one_write</i> , <i>check</i> , <i>value</i> , <i>check</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>reset</i> , <i>enable</i> , <i>start</i> , <i>add</i> , <i>and</i> , <i>addr</i> , <i>ren</i> , <i>raddr</i> , <i>rdata</i> , <i>wen</i> , <i>waddr</i> , <i>wdata</i> , <i>fire</i>	ensures the integrity of accesses to an asynchronous memory
Event-bound	<i>ovl_memory_sync</i>	<i>severity_level</i> , <i>data</i> , <i>width</i> , <i>add</i> , <i>width</i> , <i>mem_size</i> , <i>pass_thru</i> , <i>addr</i> , <i>check</i> , <i>init</i> , <i>check</i> , <i>conflict</i> , <i>check</i> , <i>one_read</i> , <i>check</i> , <i>one_write</i> , <i>check</i> , <i>value</i> , <i>check</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>r</i> , <i>clock</i> , <i>w</i> , <i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start</i> , <i>addr</i> , <i>and</i> , <i>addr</i> , <i>ren</i> , <i>raddr</i> , <i>rdata</i> , <i>wen</i> , <i>waddr</i> , <i>wdata</i> , <i>fire</i>	ensures the integrity of accesses to a synchronous memory
nCycles	<i>ovl_multiport_fifo</i>	<i>severity_level</i> , <i>width</i> , <i>depth</i> , <i>enq</i> , <i>count</i> , <i>daq</i> , <i>count</i> , <i>pass_thru</i> , <i>registered</i> , <i>enq</i> , <i>latency</i> , <i>daq</i> , <i>latency</i> , <i>prod</i> , <i>count</i> , <i>high_water</i> , <i>mark</i> , <i>full</i> , <i>full</i> , <i>check</i> , <i>empty</i> , <i>check</i> , <i>value</i> , <i>check</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>enq</i> , <i>daq</i> , <i>enq_data</i> , <i>daq_data</i> , <i>full</i> , <i>empty</i> , <i>preload</i> , <i>fire</i>	ensures data integrity of a FIFO with multiple enqueue and dequeue ports, and checks underflow and overflow
SingleCycle	<i>ovl_mutex</i>	<i>severity_level</i> , <i>width</i> , <i>invert_mode</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	ensures that the bits of an expression are mutually exclusive
SingleCycle	<i>ovl_never</i>	<i>severity_level</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must never hold
SingleCycle	<i>ovl_never_unknown</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must never hold to an unknown value, just below 0 or 1
Combinational	<i>ovl_never_unknown_async</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must never go to an unknown value asynchronously, it must remain to be 0 or 1
nCycles	<i>ovl_read</i>	<i>severity_level</i> , <i>num_cks</i> , <i>check</i> , <i>overlapping</i> , <i>check</i> , <i>missing_start</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must hold num_cks cycles after start_event holds
Event-bound	<i>ovl_next_state</i>	<i>severity_level</i> , <i>width</i> , <i>next_count</i> , <i>min_hold</i> , <i>max_hold</i> , <i>disallow</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>curr_state</i> , <i>next_state</i> , <i>fire</i>	ensures expression transitions only to specified values
Event-bound	<i>ovl_no_contention</i>	<i>severity_level</i> , <i>width</i> , <i>num_drivers</i> , <i>min_quit</i> , <i>max_quit</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>driver</i> , <i>and</i> , <i>fire</i>	ensures that a bus is driven according to specified contention rules
Two Cycles	<i>ovl_no_overflow</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	If <i>test_expr</i> is at max, in the next cycle <i>test_expr</i> must be >min and <=max
Two Cycles	<i>ovl_no_transition</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>start_state</i> , <i>next_state</i> , <i>fire</i>	If <i>test_expr</i> is at start_state, in the next cycle <i>test_expr</i> must not change to next_state
Two Cycles	<i>ovl_no_underflow</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	If <i>test_expr</i> is at min, in the next cycle <i>test_expr</i> must be >=min and <=max
SingleCycle	<i>ovl_odd_parity</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must have an odd parity, i.e. an odd number of bits asserted
SingleCycle	<i>ovl_one_cold</i>	<i>severity_level</i> , <i>width</i> , <i>inactive</i> (property_type, msg, coverage_level)	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must be one-cold, i.e. exactly one bit set low (inactive: 0=all-zero, 1=all-one, 2=pure-one-cold)
SingleCycle	<i>ovl_one_hot</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must be one-hot, i.e. exactly one bit set high
Combinational	<i>ovl_proposition</i>	<i>severity_level</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must hold asynchronously (not just at a clock edge)
Two Cycles	<i>ovl_quiescent_state</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>state_expr</i> , <i>check</i> , <i>value</i> , <i>sample</i> , <i>event</i> , <i>fire</i>	<i>state_expr</i> must equal <i>check_value</i> on a rising edge of <i>sample_event</i> (also checked on rising edge of 'OVL_END_OF_SIMULATION')
SingleCycle	<i>ovl_range</i>	<i>severity_level</i> , <i>width</i> , <i>min_max</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must be >min and <=max
Event-bound	<i>ovl_reg_loaded</i>	<i>severity_level</i> , <i>width</i> , <i>start_count</i> , <i>end_count</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>end_event</i> , <i>src_expr</i> , <i>dst_expr</i> , <i>fire</i>	ensures that a register is loaded with source data within a specified time window
nCycles	<i>ovl_req_ack_unique</i>	<i>severity_level</i> , <i>min_cks</i> , <i>max_cks</i> , <i>method</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>req</i> , <i>ack</i> , <i>fire</i>	ensures every request receives a corresponding acknowledge in a specified time window
nCycles	<i>ovl_req_requires</i>	<i>severity_level</i> , <i>min_cks</i> , <i>max_cks</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>req</i> , <i>trigger</i> , <i>req</i> , <i>follower</i> , <i>resp</i> , <i>leader</i> , <i>resp</i> , <i>trigger</i> , <i>fire</i>	ensures that every request event in a valid request-response event sequence that finishes within a specified time window
nCycles	<i>ovl_stack</i>	<i>severity_level</i> , <i>width</i> , <i>depth</i> , <i>push</i> , <i>latency</i> , <i>pop</i> , <i>latency</i> , <i>high_water</i> , <i>mark</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>push</i> , <i>pop</i> , <i>full</i> , <i>empty</i> , <i>push_data</i> , <i>pop_data</i> , <i>fire</i>	ensures the data integrity of a stack and ensures that the stack does not overflow or underflow
nCycles	<i>ovl_time</i>	<i>severity_level</i> , <i>num_cks</i> , <i>action_on_new_start</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must hold for num_cks cycles after start_event (action_on_new_start: 0=ignore, 1=reset, 2=error)
Two Cycles	<i>ovl_transition</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>start_state</i> , <i>next_state</i> , <i>fire</i>	If <i>test_expr</i> changes from start_state, then it can only change to next_state
nCycles	<i>ovl_unchange</i>	<i>severity_level</i> , <i>width</i> , <i>num_cks</i> , <i>action_on_new_start</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must not change within num_cks of start_event (action_on_new_start: 0=ignore, 1=reset, 2=error)
nCycles	<i>ovl_valid_id</i>	<i>severity_level</i> , <i>width</i> , <i>min_cks</i> , <i>max_cks</i> , <i>max_instances</i> , <i>max_ids</i> , <i>max_instances</i> , <i>per_id</i> , <i>instances</i> , <i>count</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>issued</i> , <i>issued_count</i> , <i>returned</i> , <i>flush</i> , <i>issued_id</i> , <i>returned_id</i> , <i>flush_id</i> , <i>fire</i>	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
SingleCycle	<i>ovl_value</i>	<i>severity_level</i> , <i>width</i> , <i>num_values</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>value</i> , <i>disallow</i> , <i>fire</i>	ensures the value of an expression either matches a value in a specified list or does not match any value in the list
nCycles	<i>ovl_width</i>	<i>severity_level</i> , <i>min_cks</i> , <i>max_cks</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must hold for between min_cks and max_cks cycles
Event-bound	<i>ovl_within_change</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>and_event</i> , <i>fire</i>	<i>test_expr</i> must change between start_event and end_event
Event-bound	<i>ovl_within</i>	<i>severity_level</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>and_event</i> , <i>fire</i>	<i>test_expr</i> must hold after the start_event and upto (and including) the end_event
Event-bound	<i>ovl_within_unchange</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>start_event</i> , <i>test_expr</i> , <i>and_event</i> , <i>fire</i>	<i>test_expr</i> must not change between start_event and end_event
SingleCycle	<i>ovl_zero_one_hot</i>	<i>severity_level</i> , <i>width</i> , <i>property_type</i> , <i>msg</i> , <i>coverage_level</i>	<i>clock</i> , <i>reset</i> , <i>enable</i> , <i>test_expr</i> , <i>fire</i>	<i>test_expr</i> must be one-hot or zero, i.e. at most one bit set high

PARAMETERS

severity_level

```

+define+OVL_ASSERT_ON
'OVL_FATAL
'OVL_ERROR
'OVL_WARNING
'OVL_INFO

```

property_type

```

+libext+.v+lib
'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=<bench>+ovl_init_count
+libext+.v+lib
-y <OVL_DIR>/std_ovl
+inidir+<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>

TYPE	NAME		DESCRIPTION
Single-Cycle	<code>ovl_always</code>	<code>ovl_hold_value</code>	<code>test_expr</code> must always hold
Two Cycles	<code>ovl_always_on_start</code>		<code>test_expr</code> is true immediately following the specified edge (e.g. type 0=no-edge, 1=top, 2=eq, 3=any)
Event-bound	<code>ovl_arbitrar</code>		provides grants in response to requests, as per specified arbitration scheme and within a specified time window
Single-Cycle	<code>ovl_bits</code>	<code>ovl_implication</code>	checks number of asserted (or deasserted) bits is within a specified range
nCycles	<code>ovl_change</code>		<code>test_expr</code> must change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
Single-Cycle	<code>ovl_code_distance</code>		checks hamming distance between two expressions
nCycles	<code>ovl_cycle_sequence</code>		if the initial sequence holds, the final sequence must also hold (pass_start_condition: 0=trigger-on-start, 1=trigger-on-first-unload, 2=)
Two Cycles	<code>ovl_decrement</code>	<code>ovl_increment</code>	if <code>test_expr</code> changes, it must decrement by the value parameter (modulo 2 ^{width})
Two Cycles	<code>ovl_delta</code>		if <code>test_expr</code> changes, the delta must be >=min and <=max
Single-Cycle	<code>ovl_even_parity</code>		<code>test_expr</code> must have an even parity, i.e. an even number of bits asserted
Event-bound	<code>ovl_fifo</code>	<code>ovl_memory_async</code>	checks data integrity of a FIFO and ensures that the FIFO does not overflow or underflow
Two Cycles	<code>ovl_fifo_index</code>		FIFO pointers should never overflow or underflow
nCycles	<code>ovl_frame</code>		<code>test_expr</code> 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)
nCycles	<code>ovl_handshake</code>		req and ack must follow the specified handshaking protocol
nCycles	<code>ovl_hold_value</code>		once <code>test_expr</code> matches value, <code>test_expr</code> doesn't change value until a specified event
Single-Cycle	<code>ovl_implication</code>	<code>ovl_memory_sync</code>	if antecedent_expr holds then consequent_expr must hold in the same cycle
Two Cycles	<code>ovl_increment</code>		if <code>test_expr</code> changes, it must increment by the value parameter (modulo 2 ^{width})
Event-bound	<code>ovl_memory_async</code>		ensures the integrity of accesses to an asynchronous memory
Event-bound	<code>ovl_memory_sync</code>		ensures the integrity of accesses to a synchronous memory
nCycles	<code>ovl_multiport_fifo</code>	<code>ovl_multiport_fifo</code>	ensures data integrity of a FIFO with multiple enqueue and dequeue ports, and checks underflow and overflow
Single-Cycle	<code>ovl_mutex</code>		ensures that the bits of an expression are mutually exclusive
Single-Cycle	<code>ovl_never</code>	<code>ovl_mutex</code>	<code>test_expr</code> must never hold
Single-Cycle	<code>ovl_never_unknown</code>		<code>test_expr</code> must never take an unknown value, just boolean 0 or 1
Combinatorial	<code>ovl_never_unknown</code>		<code>test_expr</code> must never go to an unknown value asynchronously, it must remain to be an 0 or 1
nCycles	<code>ovl_next</code>		<code>test_expr</code> must hold num_cks cycles after start_event holds
Event-bound	<code>ovl_next_state</code>	<code>ovl_never</code>	ensures expression transitions only to specified values
Event-bound	<code>ovl_no_contention</code>		ensures that a bus is driven according to specified contention rules
Two Cycles	<code>ovl_no_overflow</code>	<code>ovl_never</code>	if <code>test_expr</code> is at max, in the next cycle <code>test_expr</code> must be >min and <max
Two Cycles	<code>ovl_no_transition</code>	<code>ovl_never_unknown</code>	if <code>test_expr</code> is at min, in the next cycle <code>test_expr</code> must be >min and <max
Two Cycles	<code>ovl_no_underflow</code>	<code>ovl_never_unknown</code>	<code>test_expr</code> must have an odd parity, i.e. an odd number of bits asserted
Single-Cycle	<code>ovl_odd_parity</code>	<code>ovl_never_unknown_async</code>	<code>test_expr</code> must be one-hot, i.e. exactly one bit set, low (inactive), 0=all-zero, 1=all-one, 2=pure-one-hot
Single-Cycle	<code>ovl_one_hot</code>		<code>test_expr</code> must be one-hot, i.e. exactly one bit set, high
Combinatorial	<code>ovl_proposition</code>		<code>test_expr</code> must hold asynchronously (not just at a clock edge)
Two Cycles	<code>ovl_quiescent_state</code>	<code>ovl_next</code>	<code>test_expr</code> must equal check_value on a rising edge of sample_event (also checked on rising edge of 'OVL_END_OF_SIMULATION')
Single-Cycle	<code>ovl_range</code>		<code>test_expr</code> must be >min and <max
Event-bound	<code>ovl_reg_loaded</code>		ensures that a register is loaded with source data within a specified time window
nCycles	<code>ovl_req_ack_timeout</code>		ensures every request receives a corresponding acknowledge in a specified time window
nCycles	<code>ovl_req_requires</code>		ensures that every request event in takes a valid request-response event sequence that finishes within a specified time window
nCycles	<code>ovl_stack</code>	<code>ovl_next_state</code>	ensures the data integrity of a stack and ensures that the stack does not overflow or underflow
nCycles	<code>ovl_time</code>		<code>test_expr</code> must hold for num_cks cycles after start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
Two Cycles	<code>ovl_transition</code>		if <code>test_expr</code> changes from start_state, then it can only change to next_state
nCycles	<code>ovl_unchange</code>	<code>ovl_no_contention</code>	<code>test_expr</code> must not change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)
nCycles	<code>ovl_valid_id</code>		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	<code>ovl_value</code>		ensures the value of an expression either matches a value in a specified list or does not match any value in the list
nCycles	<code>ovl_width</code>		<code>test_expr</code> must hold for between min_cks and max_cks cycles
Event-bound	<code>ovl_win_change</code>	<code>ovl_no_overflow</code>	<code>test_expr</code> must change before start_event and end_event
Event-bound	<code>ovl_window</code>	<code>ovl_no_transition</code>	<code>test_expr</code> must hold after the start_event and upto (and including) the end_event
Event-bound	<code>ovl_win_unchange</code>		<code>test_expr</code> must not change between start_event and end_event
Single-Cycle	<code>ovl_zero_one_hot</code>		<code>test_expr</code> must be one-hot or zero, i.e. at most one bit set, high
PARAMETERS		INPUT ASSUMPTIONS	
<code>severity_level</code>		Restricts environment	
<code>'OVL_FATAL</code>		Examples	
<code>'OVL_ERROR</code>		* One hot inputs	
<code>'OVL_WARNING</code>		* Range limits e.g. cache sizes	
<code>'OVL_INFO</code>		* Stability e.g. cache sizes	
<code>property_type</code>		* No back-to-back reqs	
<code>'OVL_ASSERT</code>		* Handshaking sequences	
<code>'OVL_ASSUME</code>		* Bus protocol	
<code>'OVL_IGNORE</code>			
<code>msg</code> descriptive string			

+includer<<OVL_DIR>/std_ovl

* FSM transitions

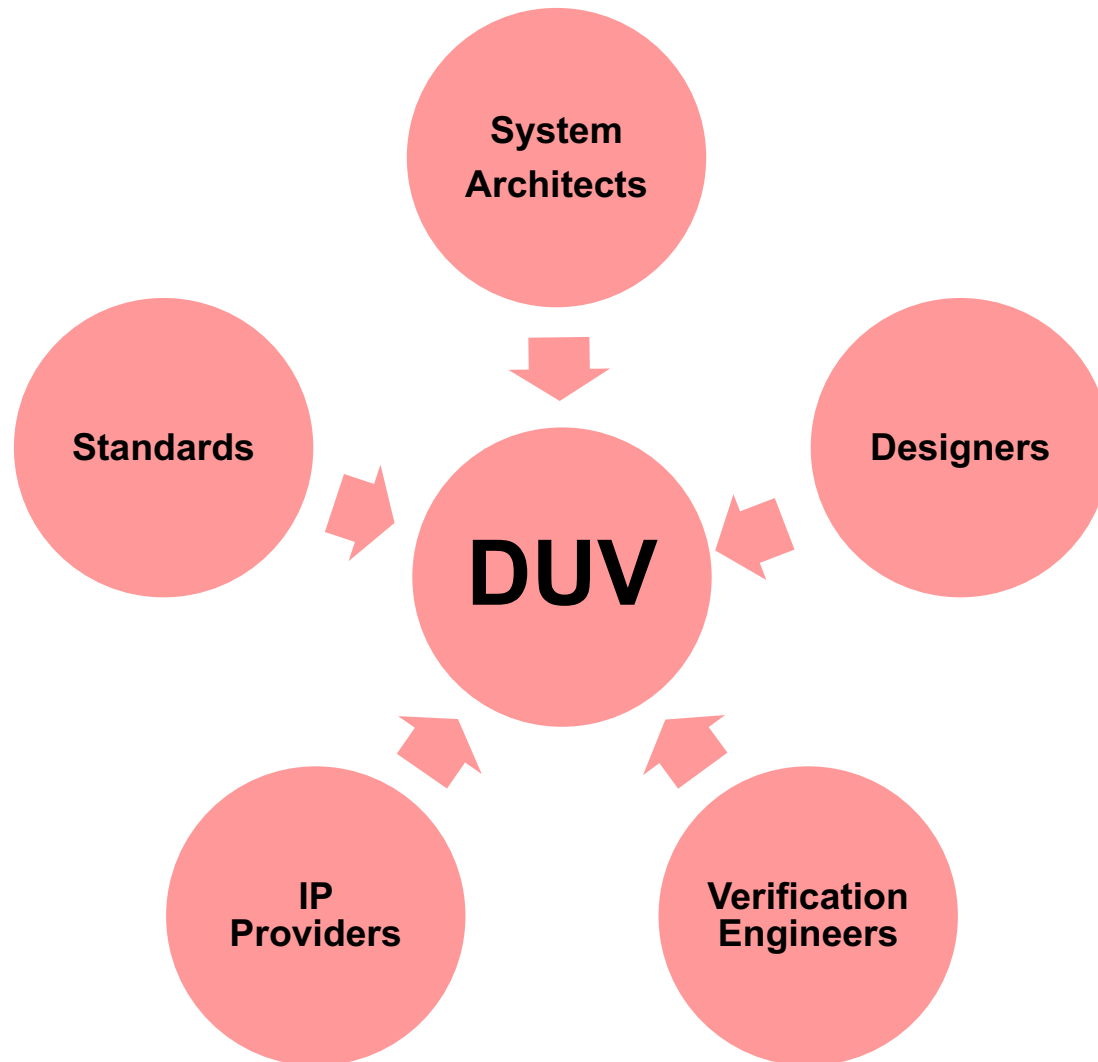
* X checkers (ovl_never_unknown)

<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/>



