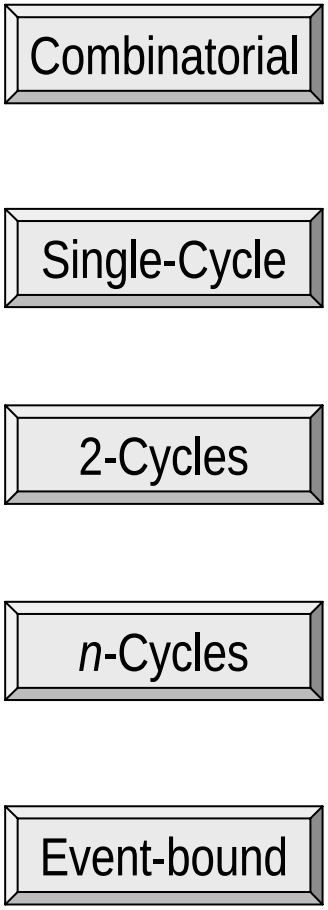
Timing Diagrams for Accellera Standard OVL V1.8

Mike Turpin / ARM

18th October 2006

Contents

Types of OVL Assertion



Combinatorial Assertions

- assert_proposition, assert_never_unknown_async

Single-cycle Assertions

- assert_always, assert_implication, assert_range, …

Sequential over 2 cycles

- assert_always_on_edge, assert_decrement, …

Sequential over num_cks cycles

- assert_change, assert_cycle_sequence, assert_next, …

Sequential between two events

- assert_win_change, assert_win_unchange, assert_window

accellera

OVL Release History and Major Changes

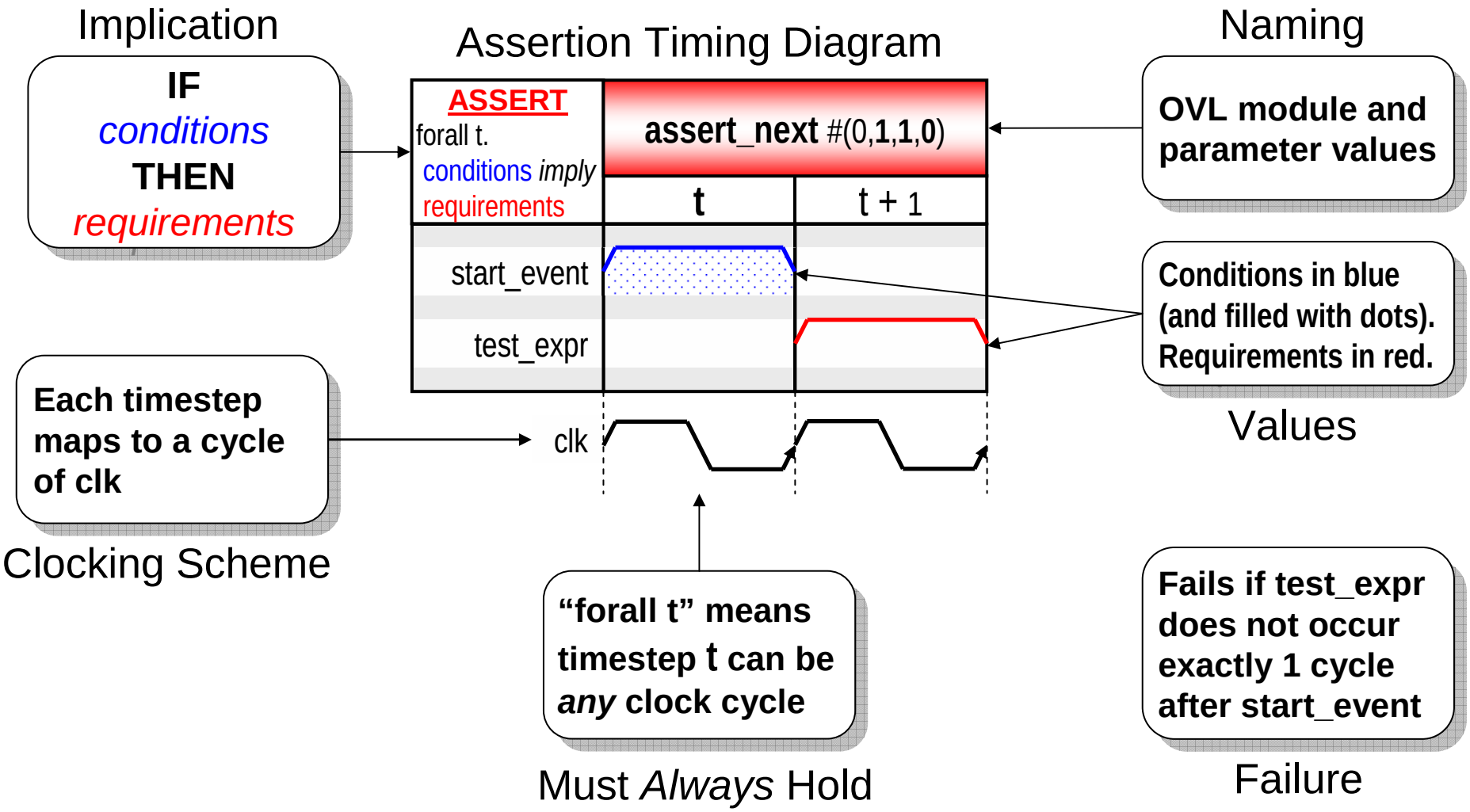
- pre-Accellera, April 2003
  - Verilog updated in April, but VHDL still October 2002
- v1.0, July 2005
  - Changed: assert_fifo_index (no longer uses property_type in functionality)
- v1.1, August 2005
  - New: assert_never_unknown
  - Changed:
    - assert_implication: anteceндent_expr typo fixed
    - assert_change: window length fixed to num_cks
- v1.1a, August 2005
  - Fixed: assert_width
- v1.1b, August 2005 (minor updates to doc)

OVL Release History and Major Changes

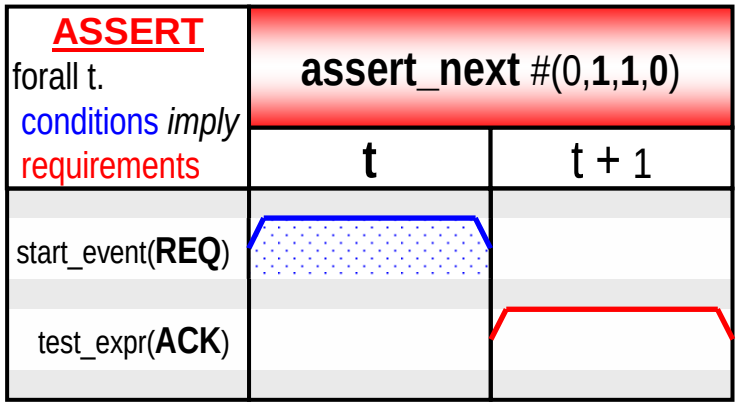
- v1.5, December 2005
  - New:
    - Preliminary PSL support
    - `OVL_IGNORE property_type
  - Fixed: assert_always_on_edge (startup delayed by 1 cycle)
- v1.6, March 2006
  - New: assert_never_unknown_async
- v1.7, July 2006
  - Consistent X Semantics & Coverage Levels
  - PSL support
- v1.8, Oct 2006
  - Bug fixes

Contents

Introduction: OVL Timing Diagram

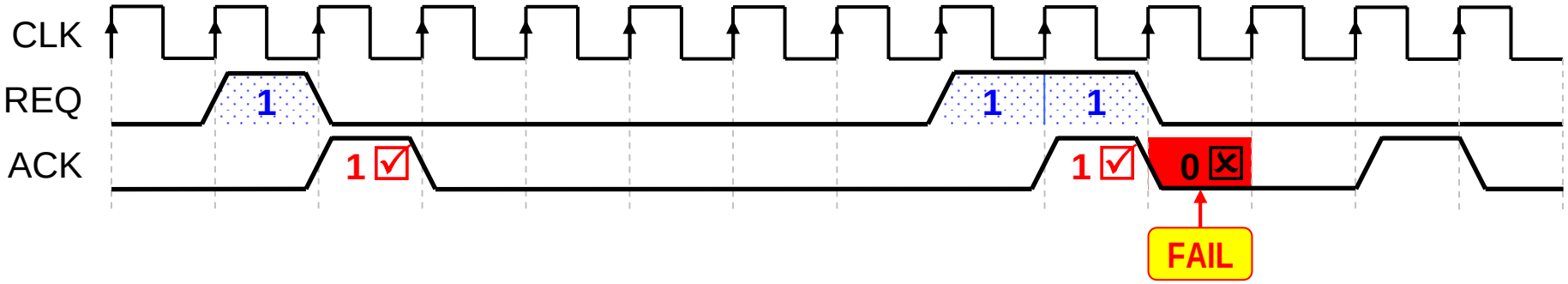


Introduction: Verification of Assertion



Imagine *sliding* the timing diagram, pipeline style, over each simulation cycle …

… *if* all conditions match, *then* all requirements must hold.

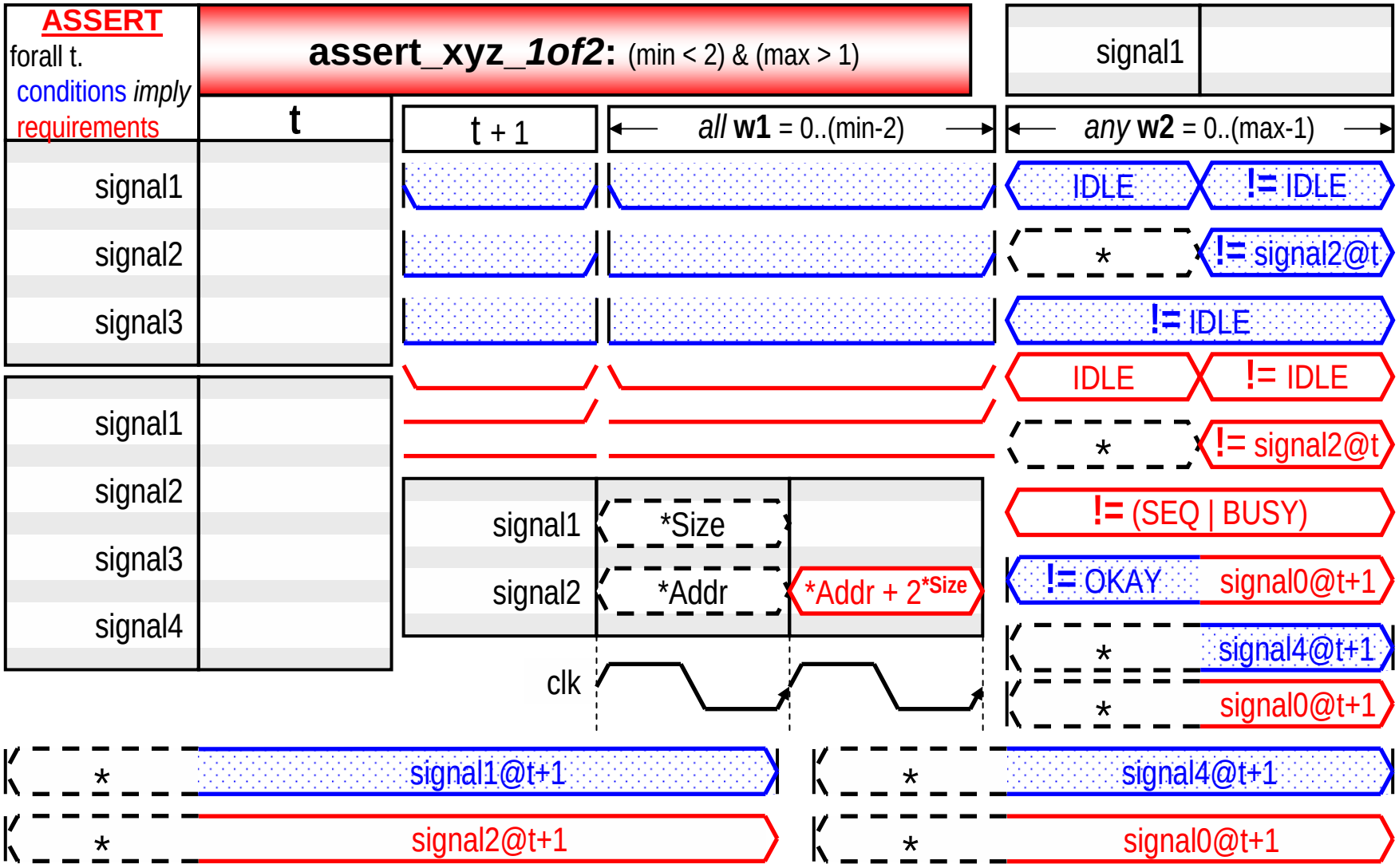


Simulation *might* show this failure, but only if stimulus covers back-to-back REQs.

Formal Verification would never pass this, and should show the failure with a short debug trace.

accellera

Template


ASSERT
forall t.
conditions imply requirements
assert_xyz_1of2: (min < 2) & (max > 1)
t
t + 1
all w1 = 0..(min-2)
any w2 = 0..(max-1)
signal1
signal2
signal3
signal1
signal2
signal3
signal4
signal1
signal2
signal3
signal4
clk
*Size
*Addr
*Addr + 2*Size
IDLE
!= IDLE
!= signal2@t
!= IDLE
IDLE
!= IDLE
!= signal2@t
!= (SEQ | BUSY)
!= OKAY
signal0@t+1
signal4@t+1
signal0@t+1
signal1@t+1
signal2@t+1
signal4@t+1
signal0@t+1

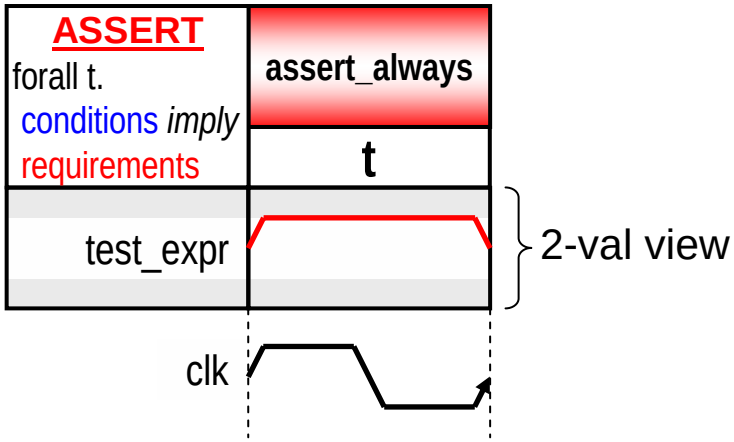
Contents

assert_always

```
#(severity_level, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must always hold

Single-Cycle



assert_always will also *pessimistically* fail if test_expr is X

Can disable failure on X/Z via:

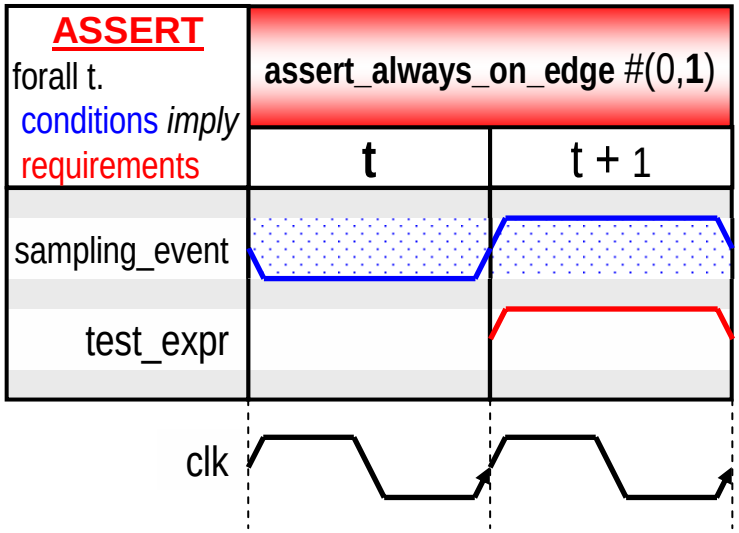
`define OVL_XCHECK_OFF

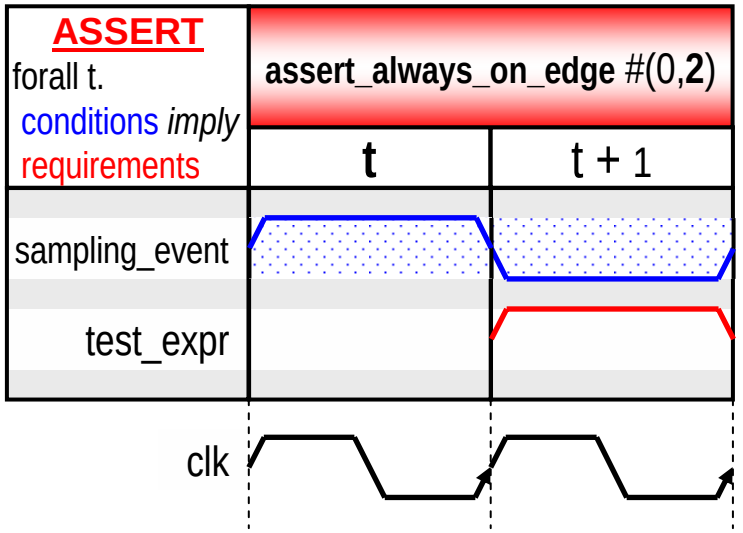
assert_always_on_edge

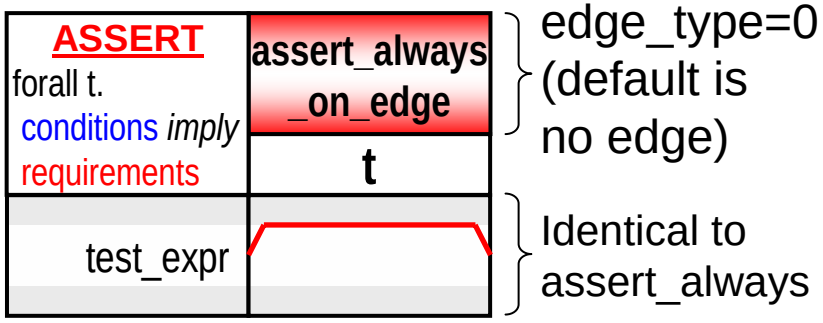
```
#(severity_level, edge_type, property_type, msg, coverage_level)
u1 (clk, reset_n, sampling_event, test_expr)
```

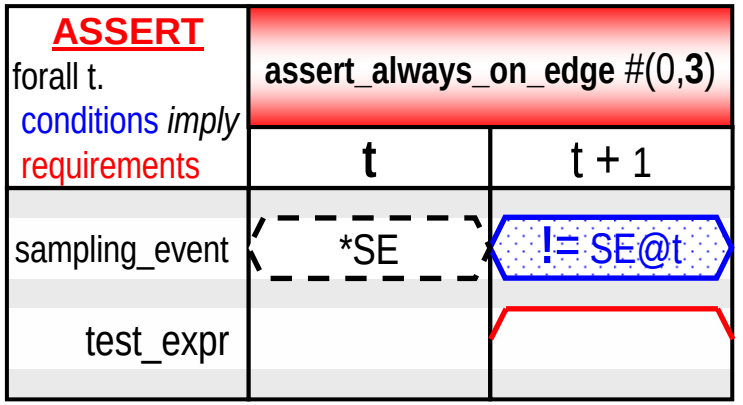
`test_expr` is true immediately following the edge specified by the `edge_type` parameter

2-Cycles







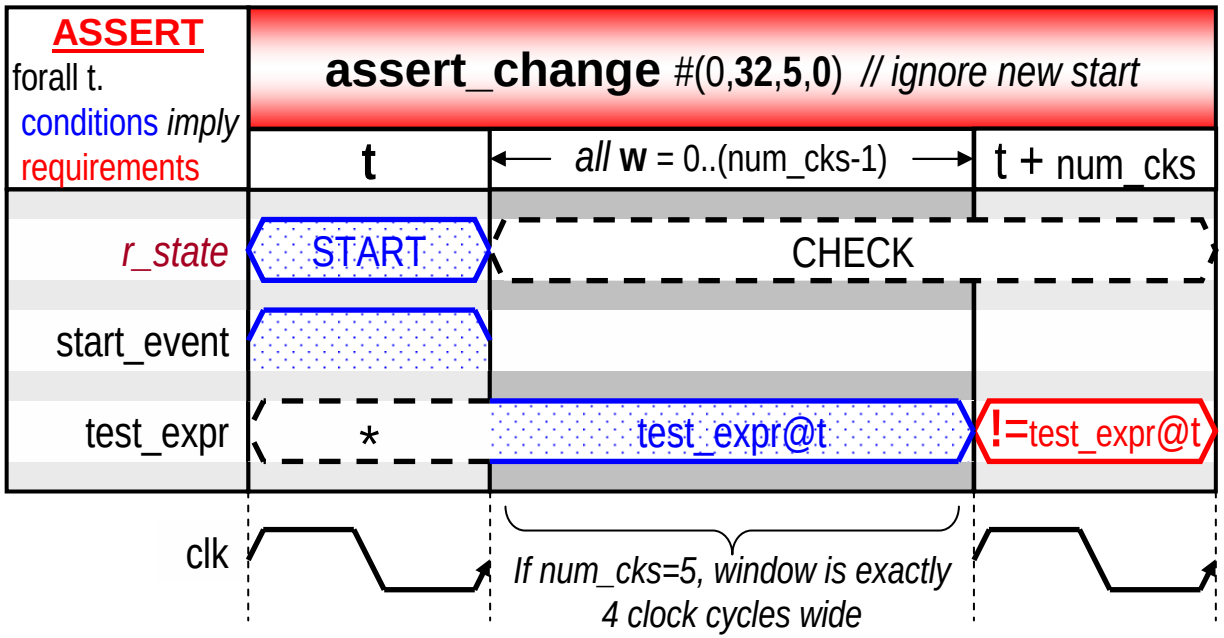


assert_change

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must change within `num_cks` cycles of `start_event`

n-Cycles

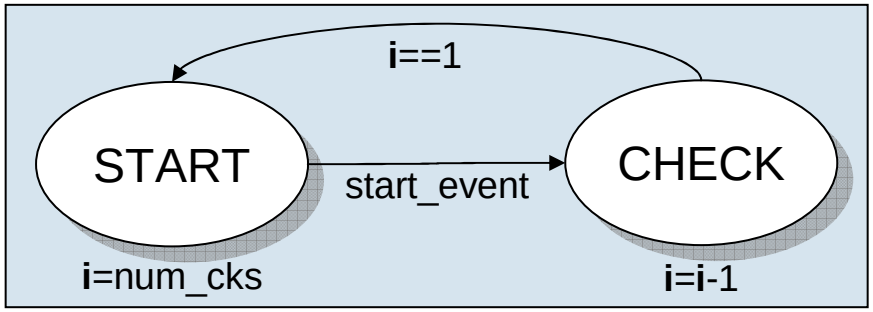


num_cks=5
action_on_new_start=0
(`OVL_IGNORE_NEW_START)

Will pass if test_expr changes at *any* cycle:
t+1, t+2, …, t+num_cks
Fails if test_expr is stable for all num_cks cycles.

Differs to April 2003
From OVL version 1.0 the check window spans the entire num_cks-1 cycles.

r_state (auxiliary logic)



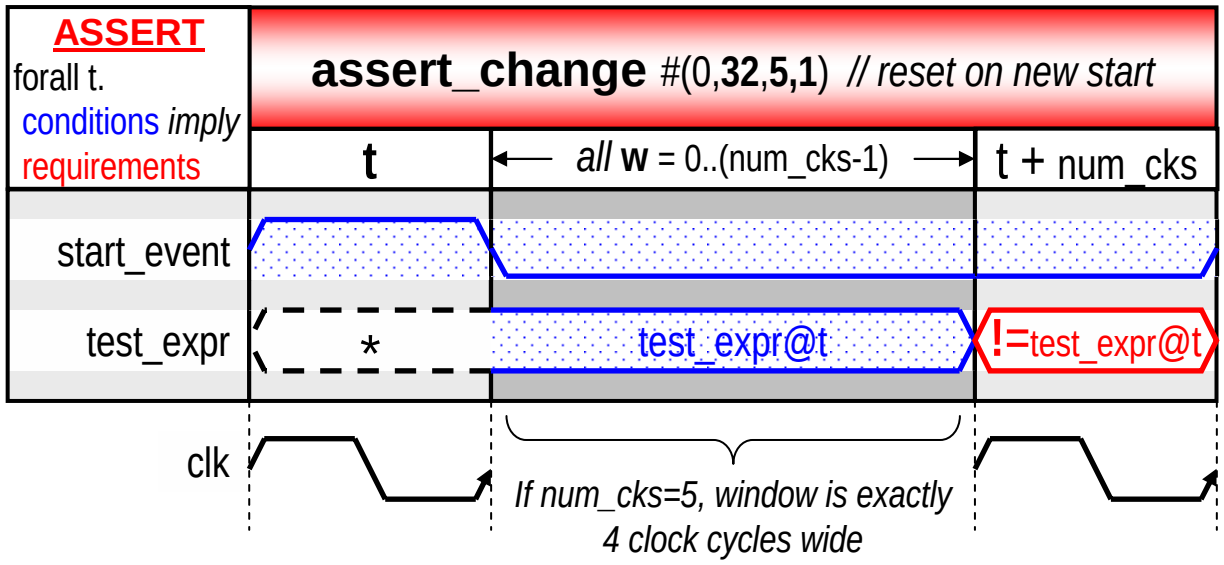
Auxiliary logic necessary, to *ignore* new start.
Checking only begins after start_event is true *and* r_state==START.

assert_change

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must change within `num_cks` cycles of `start_event`

n-Cycles



num_cks=5
action_on_new_start=1
(`OVL_RESET_ON_NEW_START)

extra condition on start_event

"reset on new start" timing diagram does not require auxiliary logic (see note 2).

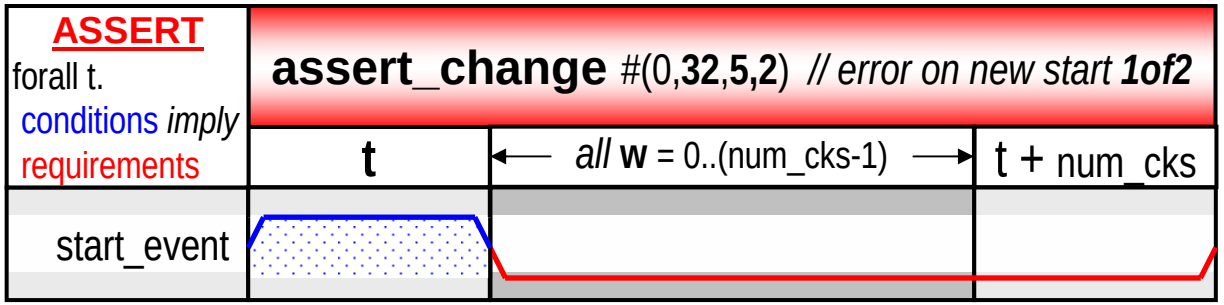
Differs to April 2003
From OVL version 1.0 the check window spans the entire num_cks-1 cycles.

assert_change

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must change within `num_cks` cycles of `start_event`

n-Cycles



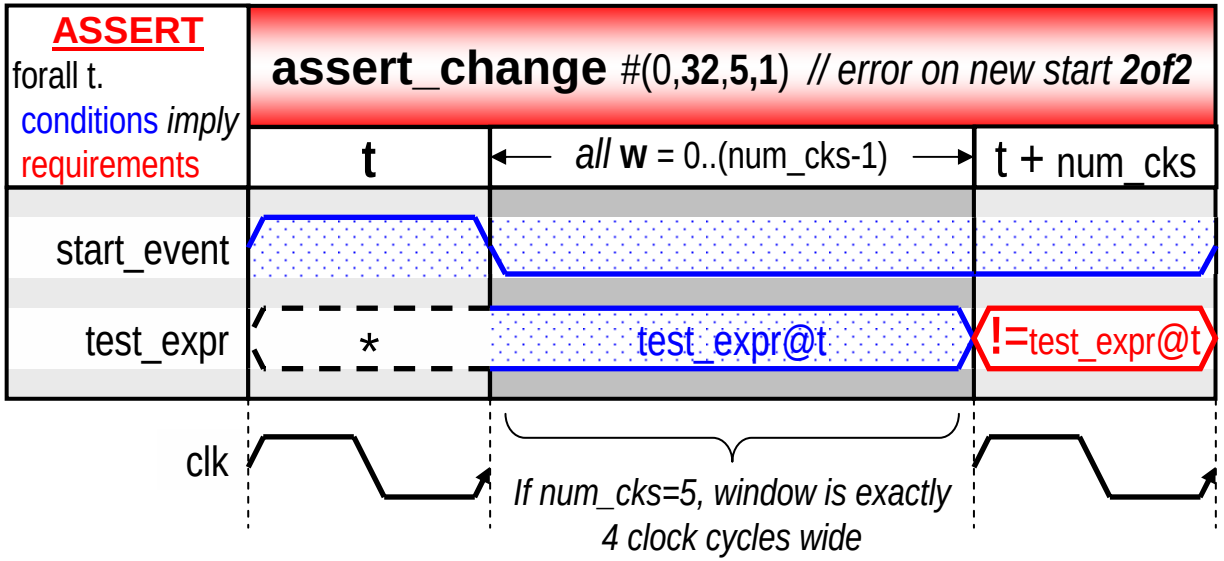
num_cks=5
action_on_new_start=2
(`OVL_ERROR_ON_NEW_START)

requirement on start_event

+

"error on new start" requires **two timing diagrams**, with 2nd being the same as "reset on new start"

Differs to April 2003
From OVL version 1.0 the check window spans the entire num_cks-1 cycles.



num_cks=5
action_on_new_start=2
(`OVL_ERROR_ON_NEW_START)

condition on start_event

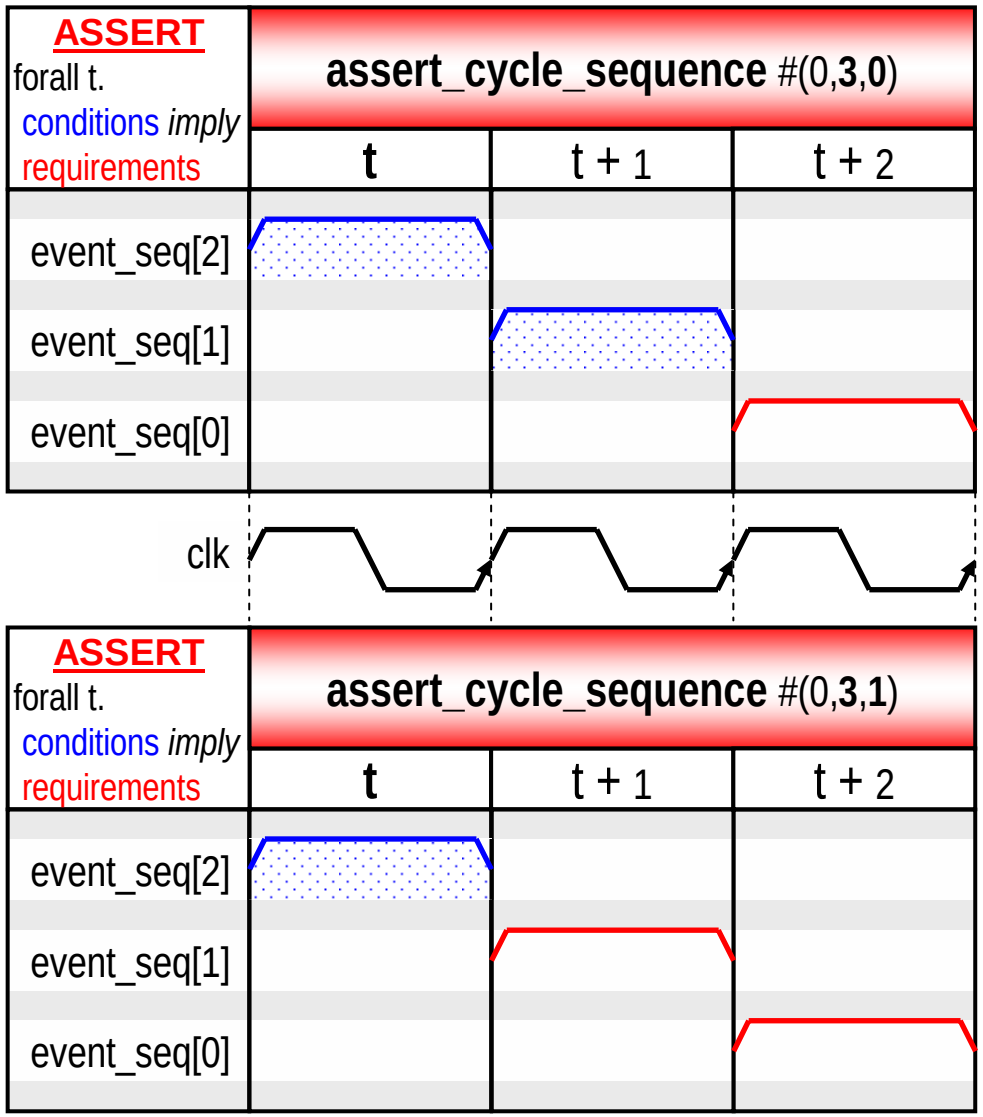
accellera

assert_cycle_sequence

```
#(severity_level, num_cks, necessary_condition, property_type, msg, coverage_level)
u1 (clk, reset_n, event_sequence)
```

If the initial sequence holds, the final sequence must also hold (final is 1-cycle or N-1 cycles)

n-Cycles



num_cks=3
necessary_condition=0
(`OVL_TRIGGER_ON_MOST_PIPE)

num_cks=3
necessary_condition=1
(`OVL_TRIGGER_ON_FIRST_PIPE)

Both timing diagrams are pipelined. They do not require any auxiliary logic.

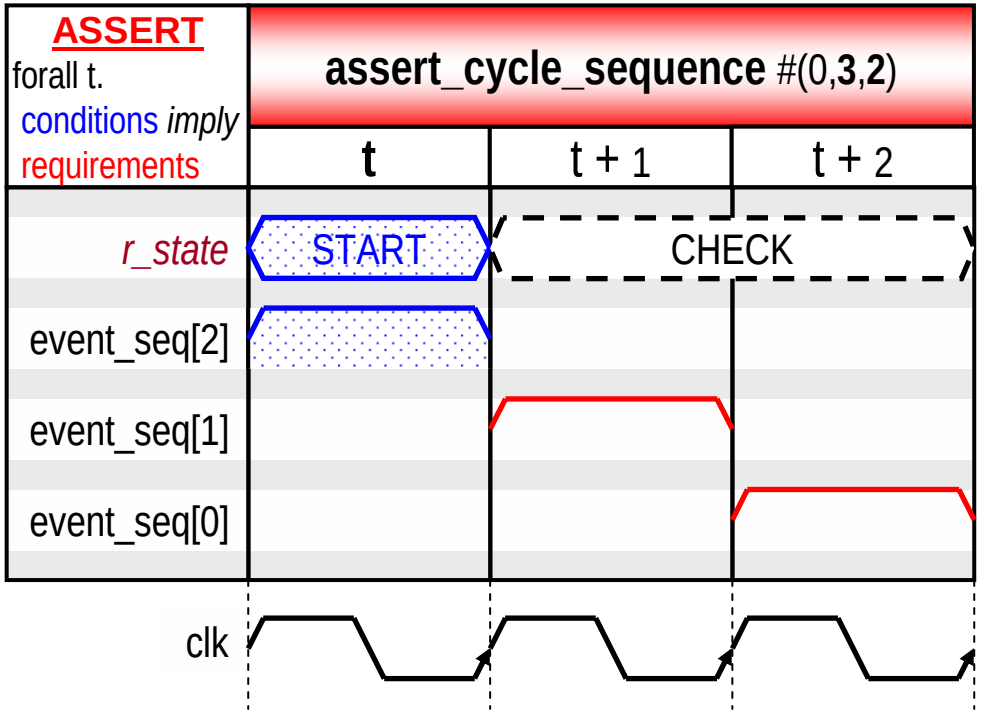
accellera

assert_cycle_sequence

```
#(severity_level, num_cks, necessary_condition, property_type, msg, coverage_level)
u1 (clk, reset_n, event_sequence)
```

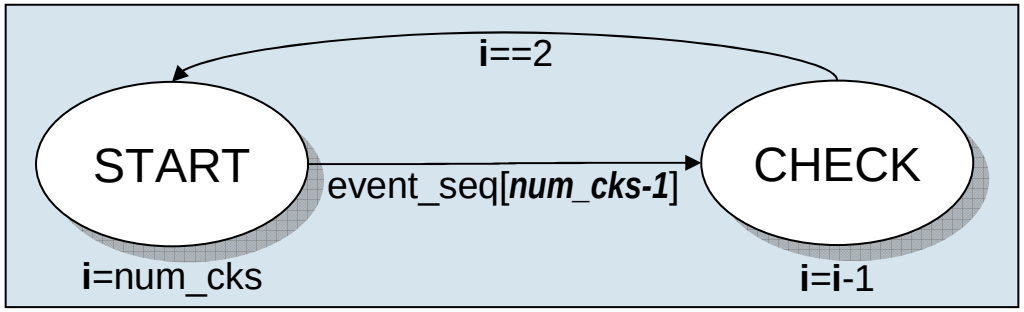
If the initial sequence holds, the final sequence must also hold (final is 1-cycle or N-1 cycles)

n-Cycles



num_cks=3
necessary_condition=2
(`OVL_TRIGGER_ON_FIRST_NONPIPE)

r_state (auxiliary logic)



Need auxiliary logic, to *ignore* subsequent **event_seq[*num_cks-1*]** when non-pipelined.

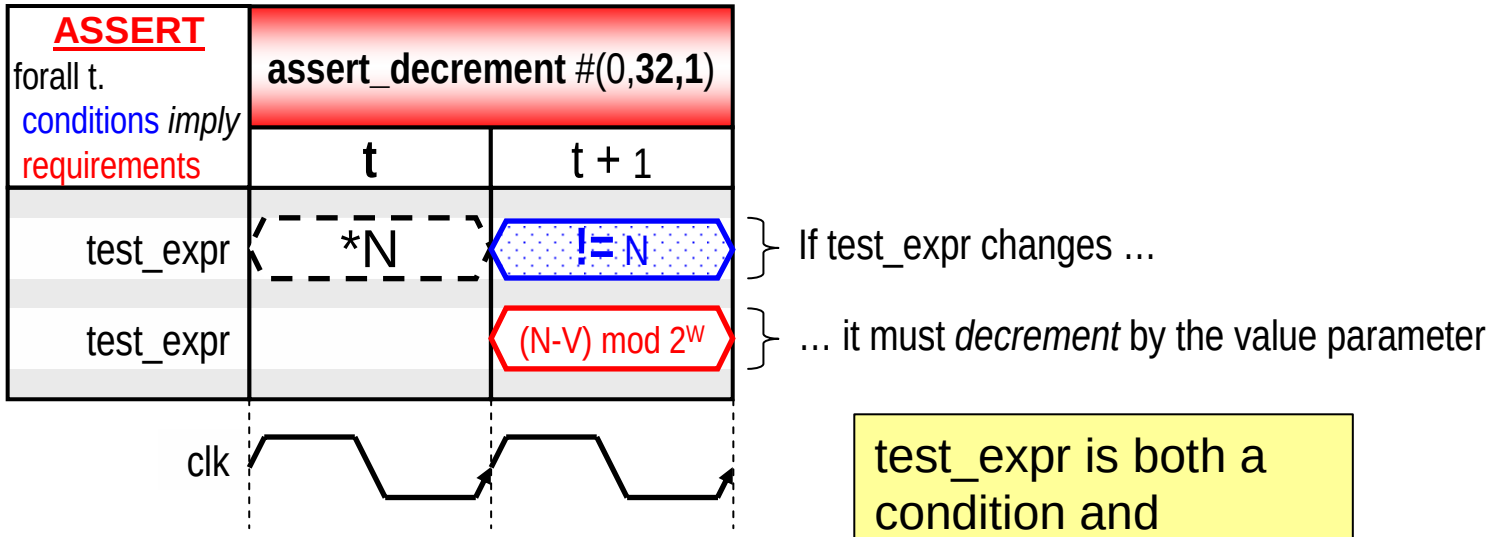
accellera

assert_decrement

```
#(severity_level, width, value, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

If `test_expr` changes, it must decrement by the `value` parameter (modulo 2^{width})

2-Cycles



Decrement is allowed to wrap, i.e. arithmetic is modulo 2^{width}.

test_expr is both a condition and requirement at t+1. Hence it appears on two rows.

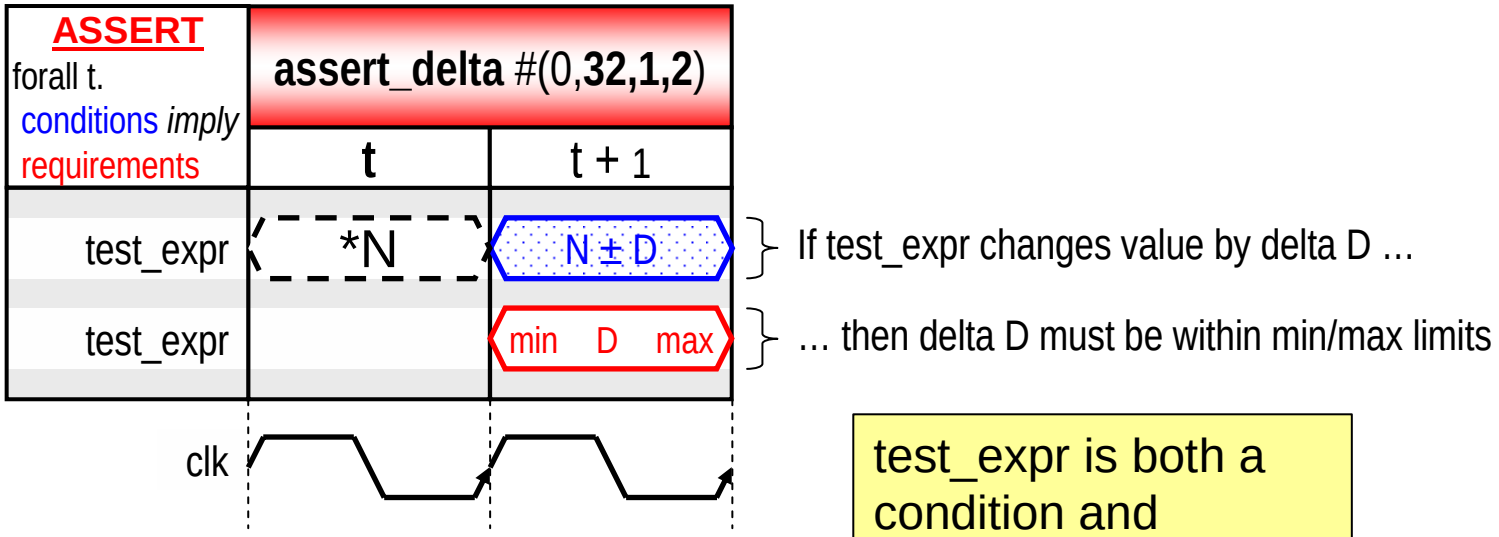
accellera

assert_delta

```
#(severity_level, width, min, max, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

If `test_expr` changes, the delta must be `min` and `max`

2-Cycles



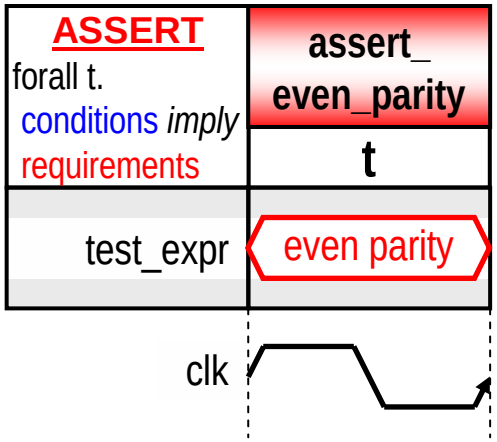
test_expr is both a condition and requirement at t+1. Hence it appears on two rows.

assert_even_parity

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must have an even parity, i.e. an even number of bits asserted.

Single-Cycle

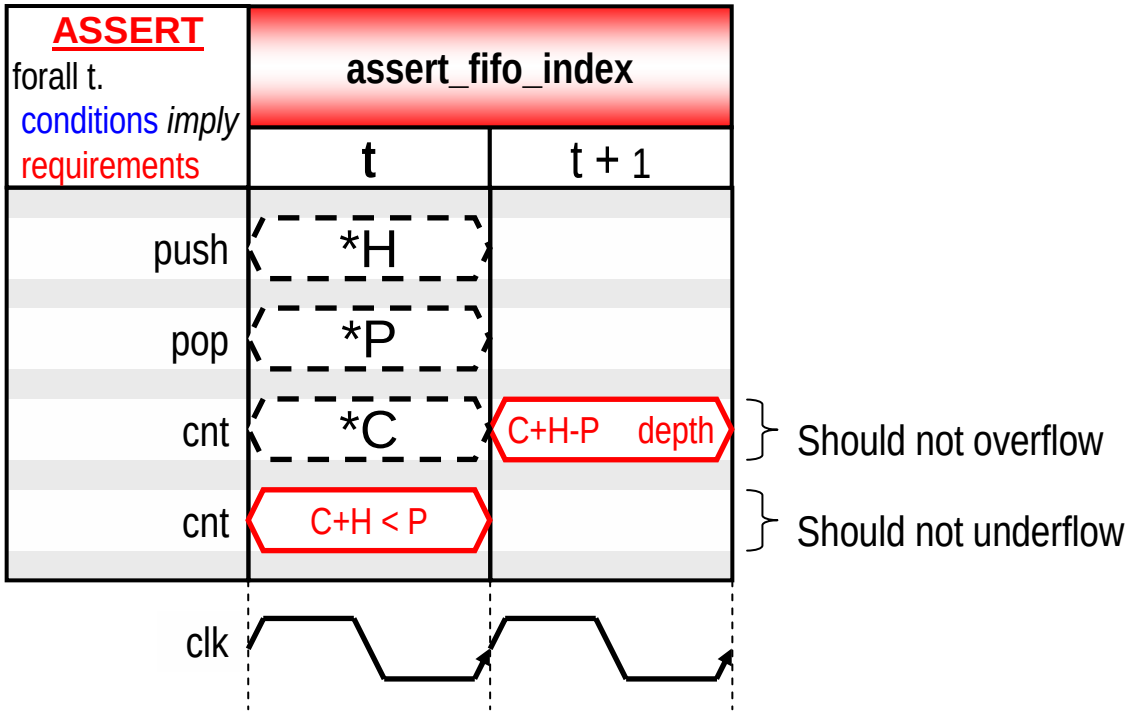


assert_fifo_index

```
#(severity_level, depth, push_width, pop_width, property_type, msg, coverage_level, simultaneous_push_pop)
u1 (clk, reset_n, push, pop)
```

FIFO pointers should never overflow or underflow.

2-Cycles



The counter “cnt” changes by a (push-pop) delta every cycle.

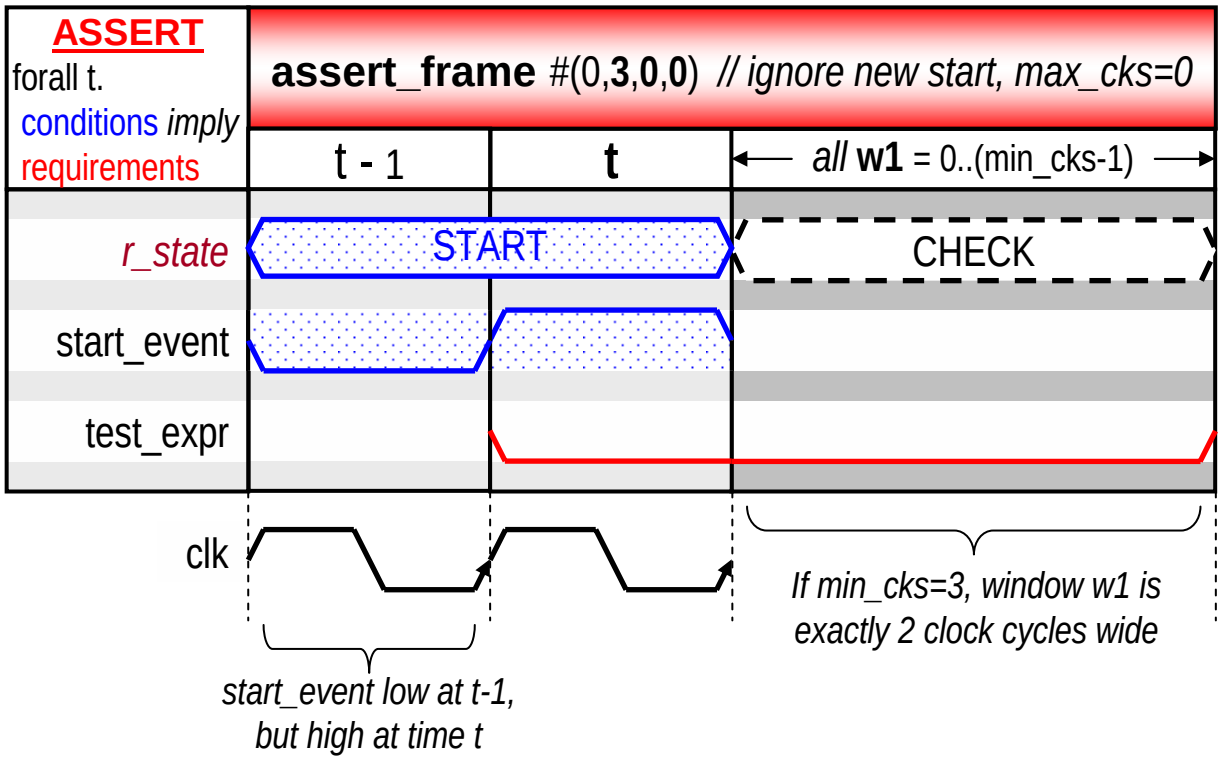
If simultaneous_push_pop is low, there is an additional check to ensure that push and pop are not both >1

assert_frame

```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not hold before `min_cks` cycles, but must hold at least once by `max_cks`.

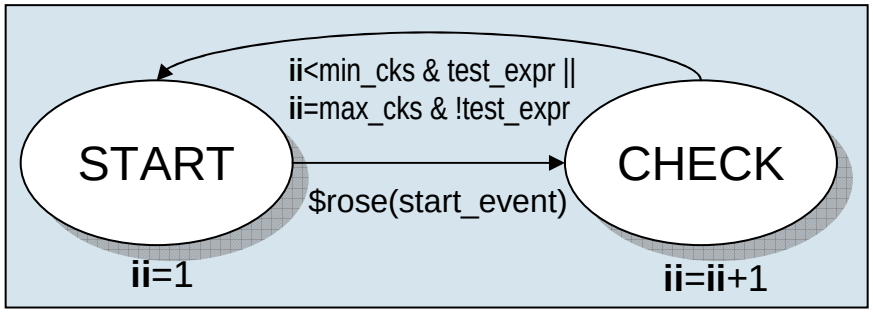
n-Cycles



min_cks>0, max_cks=0
action_on_new_start=0
(`OVL_IGNORE_NEW_START)

Shows min_cks>0 and max_cks=0 (no upper limit). Only checks that test_expr stays low up until t+(min_cks-1).

r_state (auxiliary logic)



Auxiliary logic necessary, to *ignore* new rising edge on start_event. The $rose syntax indicates high now but low in previous cycle.

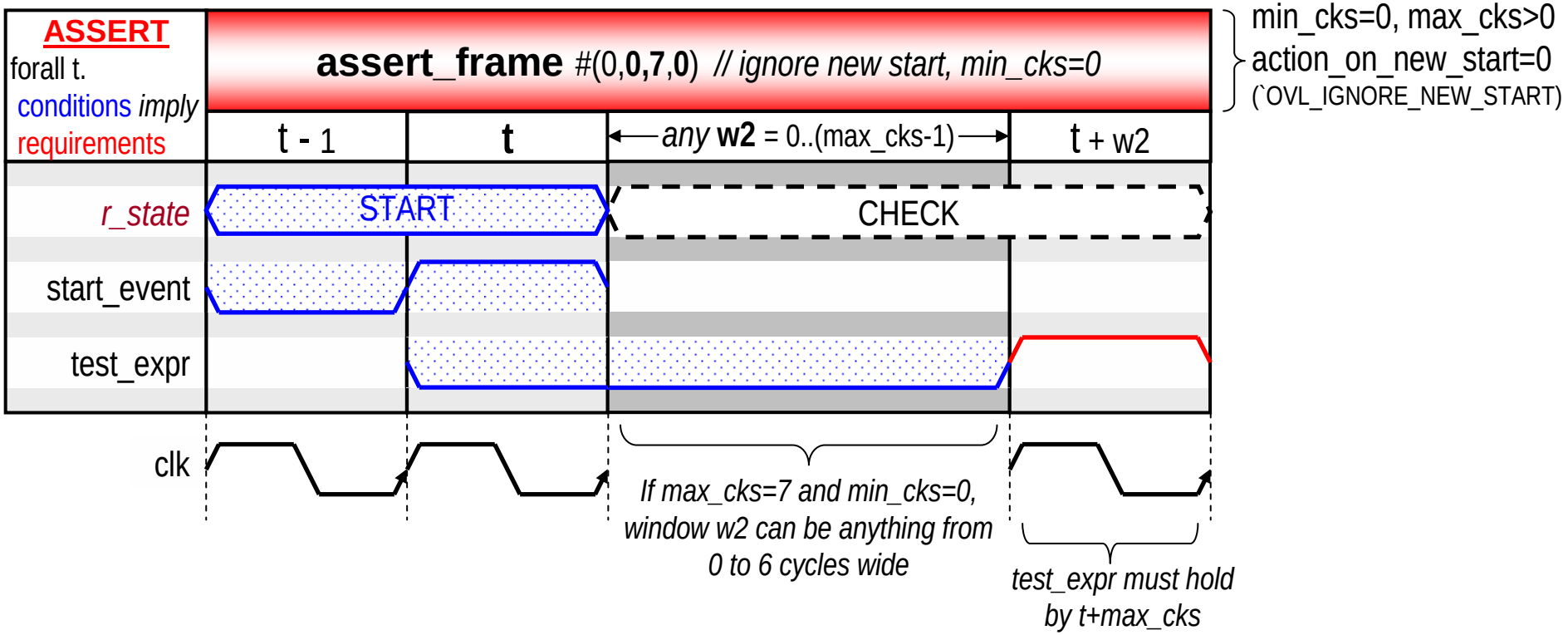
accellera

assert_frame

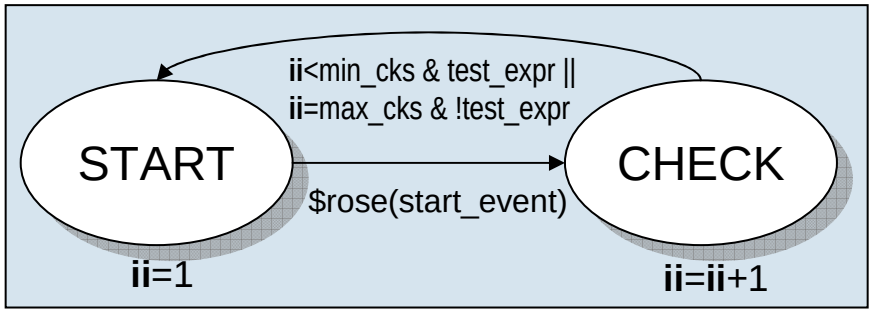
```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not hold before `min_cks` cycles, but must hold at least once by `max_cks`.

n-Cycles



r_state (auxiliary logic)



Important to have test_expr@t==1'b0 condition. Avoids extra checking if test_expr already holds at time t.

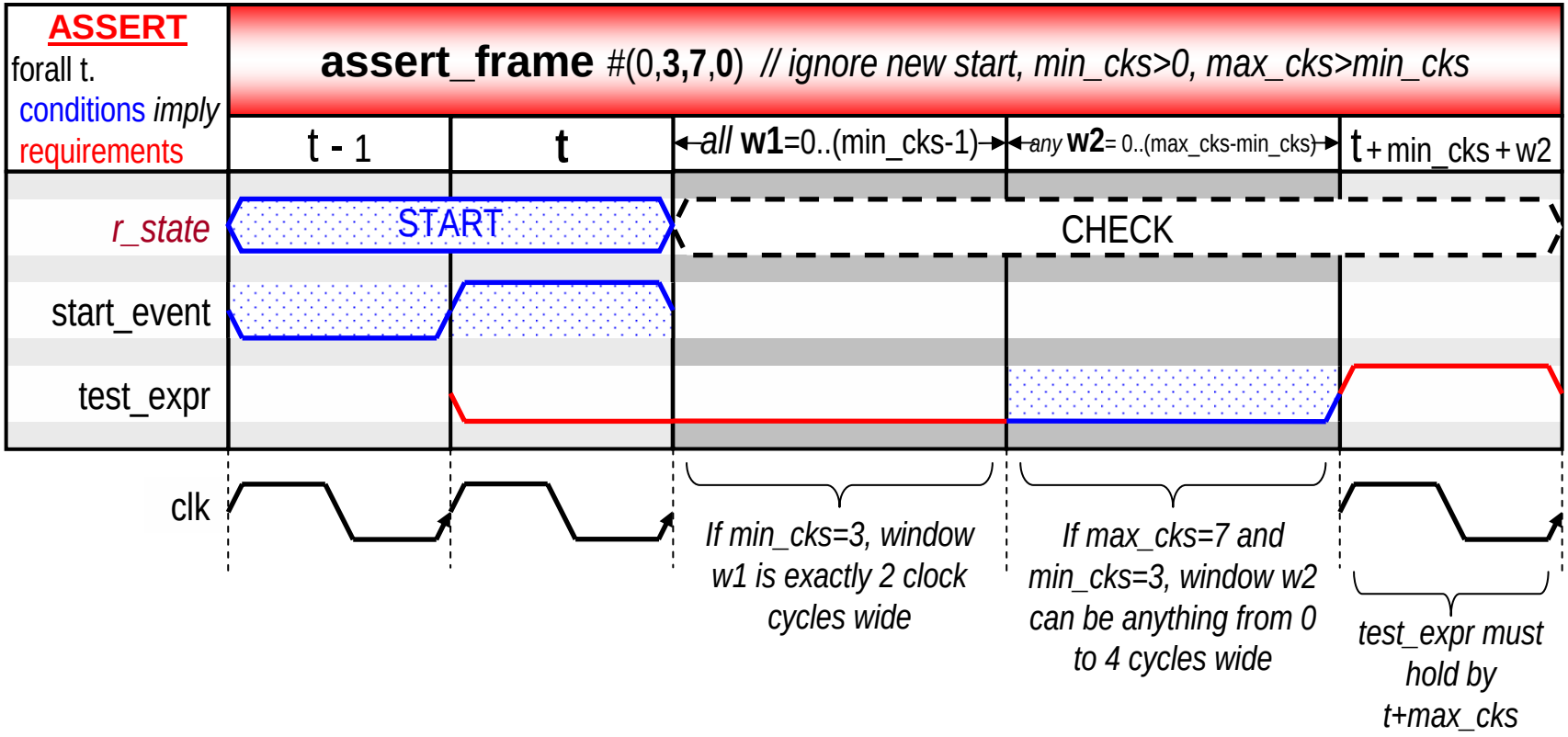
accellera

assert_frame

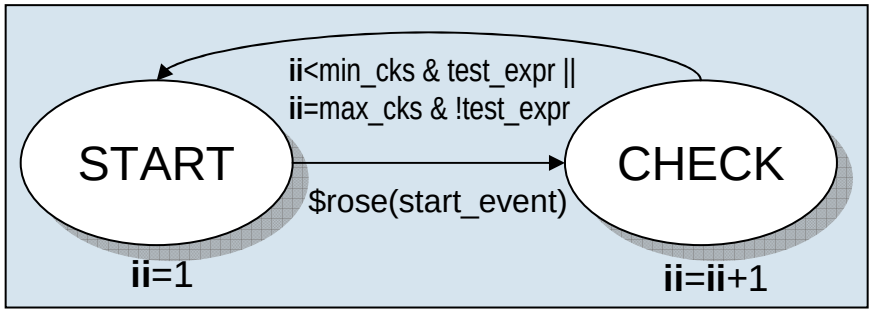
```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not hold before `min_cks` cycles, but must hold at least once by `max_cks`.

n-Cycles



r_state (auxiliary logic)

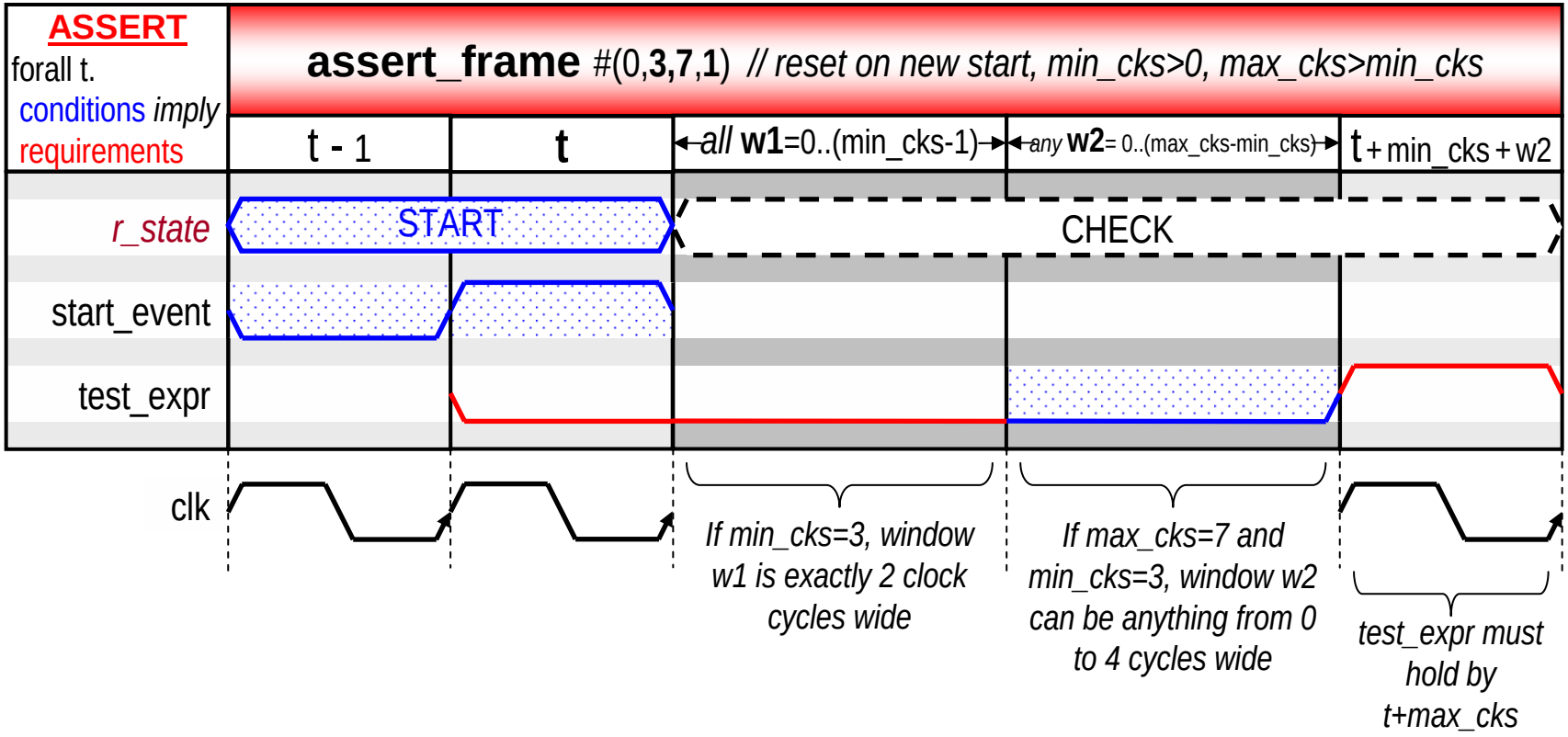


assert_frame

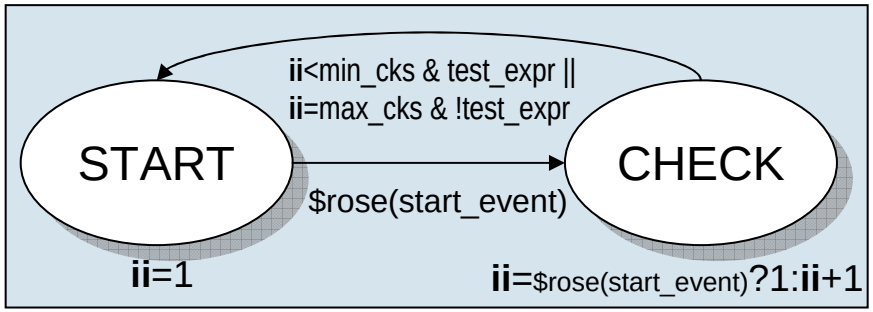
```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not hold before `min_cks` cycles, but must hold at least once by `max_cks`.

n-Cycles



r_state (auxiliary logic)



Auxiliary logic also necessary for “reset on new start”, but counter resets to 1 on new rising edge of start_event.

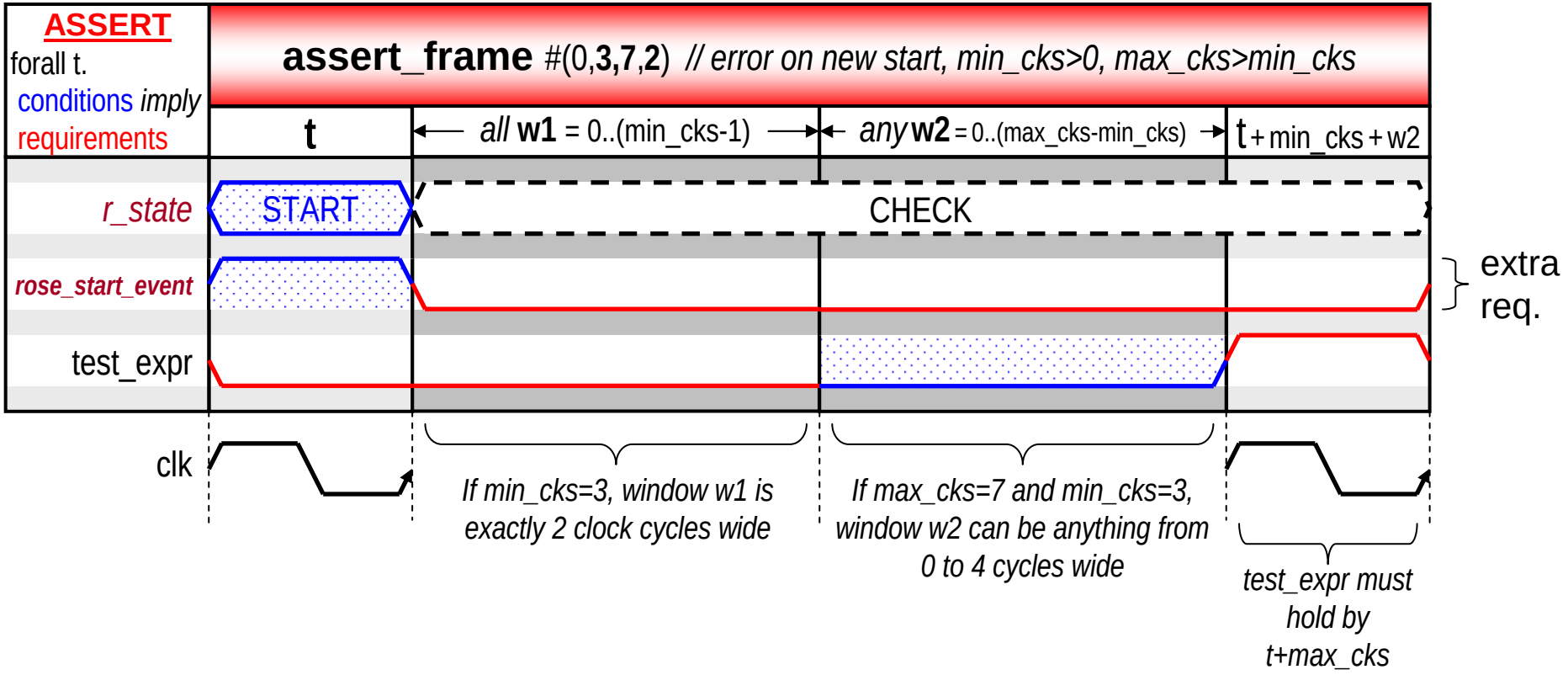
accellera

assert_frame

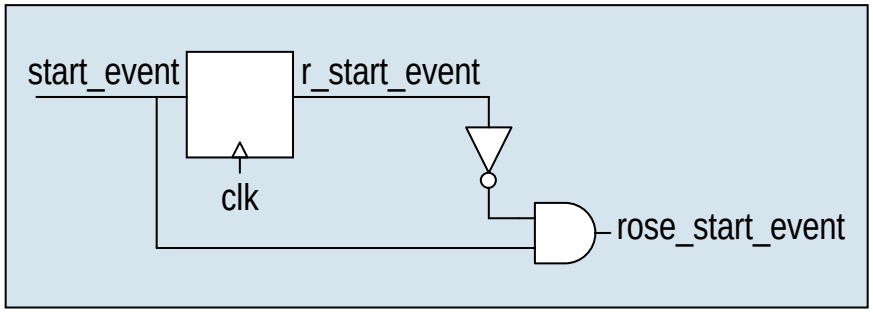
```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not hold before `min_cks` cycles, but must hold at least once by `max_cks`.

n-Cycles



rose_start_event (auxiliary logic)



"error on new start" has an additional requirement from t+1 (no new rising edge on start_event).

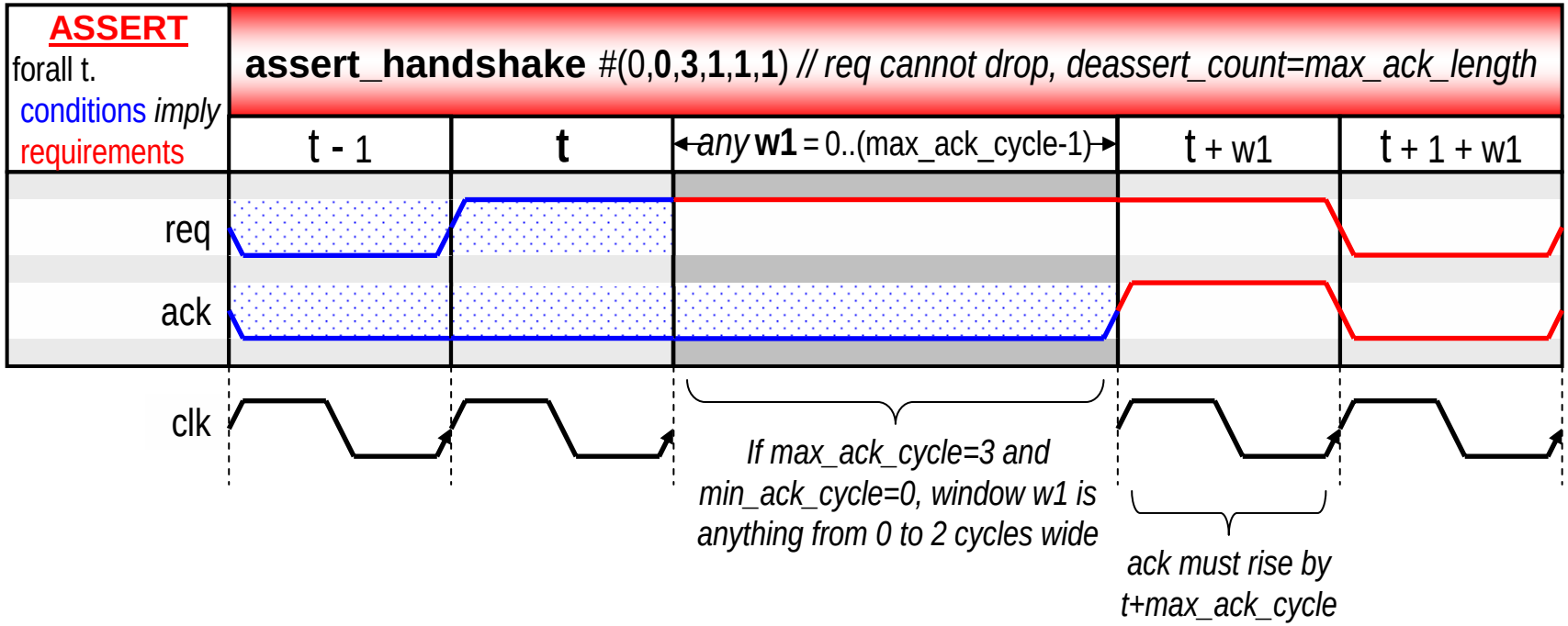
accellera

assert_handshake

```
#(severity_level, min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length,
  property_type, msg, coverage_level) u1 (clk, reset_n, req, ack)
```

`req` and `ack` must follow the specified handshaking protocol

n-Cycles



assert_handshake is highly configurable. This timing diagram shows the most common usage.

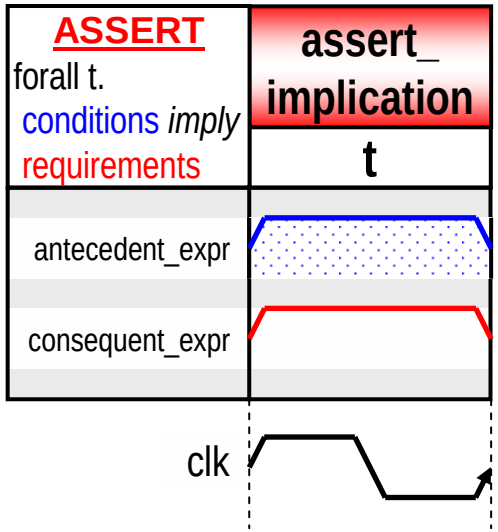
Consider splitting up more complex uses into multiple OVL (simplifies formal property checking).

assert_implication

```
#(severity_level, property_type, msg, coverage_level)
u1 (clk, reset_n, antecedent_expr, consequent_expr)
```

If `antecedent_expr` holds then `consequent_expr` must hold in the same cycle

Single-Cycle



Assertion will only fail if consequent_expr is low when antecedent_expr holds.

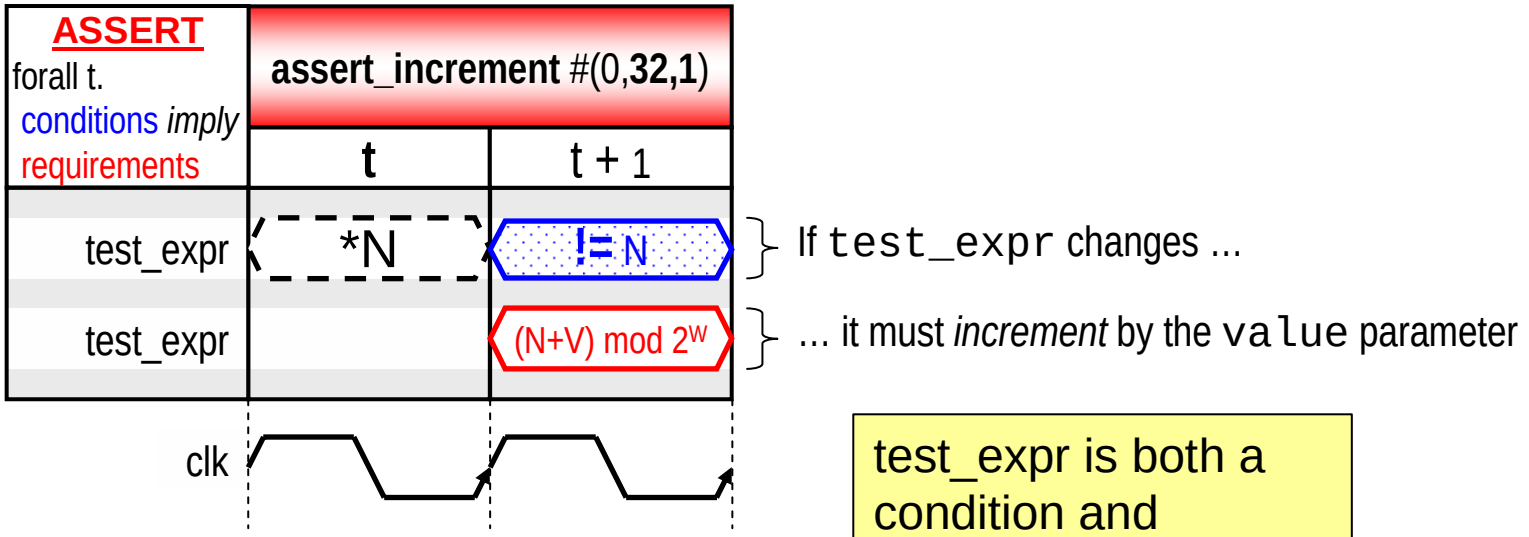
Assertion will trivially pass if conditions do not occur i.e. if antecedent_expr=0.

assert_increment

```
#(severity_level, width, value, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

If `test_expr` changes, it must increment by the `value` parameter (modulo 2^{width})

2-Cycles



Increment is allowed to wrap, i.e. arithmetic is modulo 2^{width}.

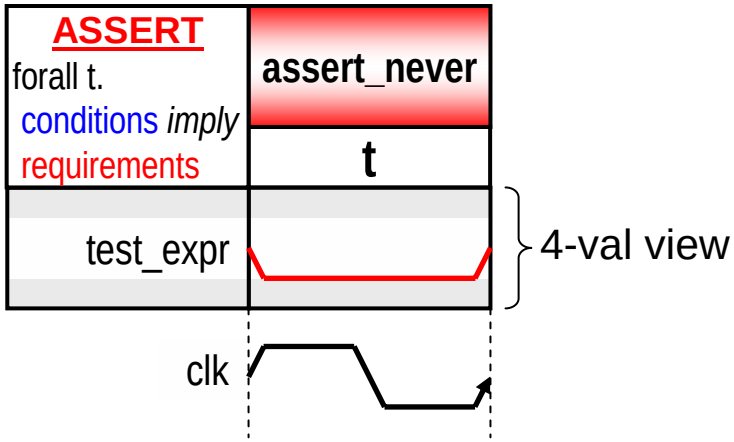
test_expr is both a condition and requirement at t+1. Hence it appears on two rows.

assert_never

```
#(severity_level, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must never hold

Single-Cycle



assert_never will also *pessimistically* fail if test_expr is X

Can disable failure on X/Z via:

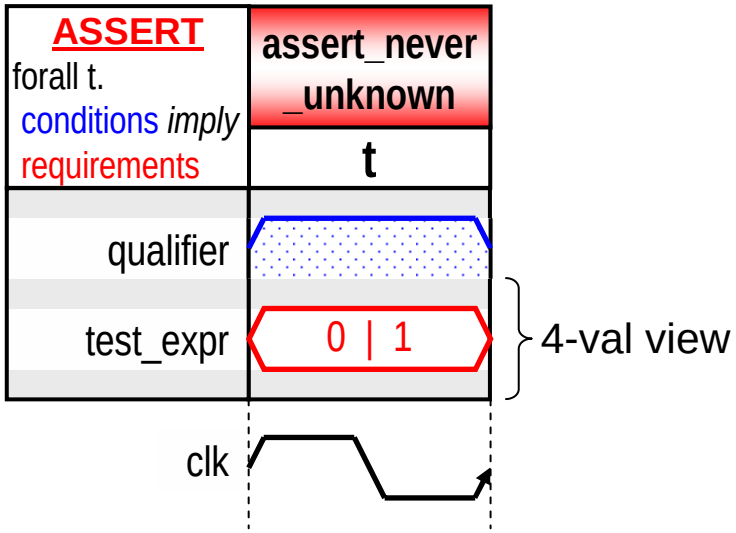
`define OVL_XCHECK_OFF

assert_never_unknown

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, qualifier, test_expr)
```

`test_expr` must never be at an unknown value, just boolean 0 or 1.

Single-Cycle

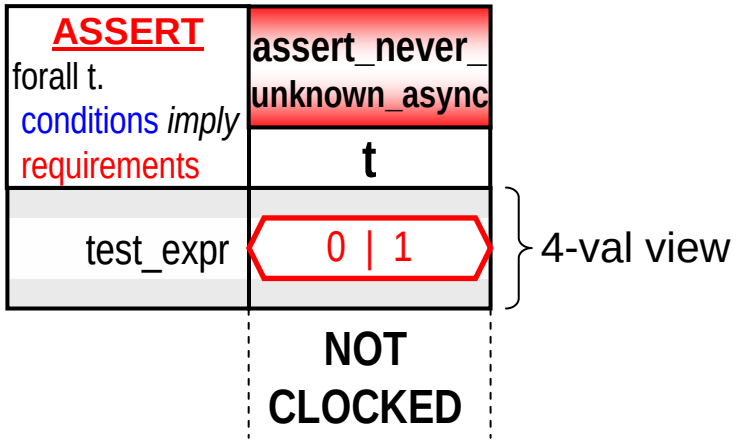


Often used as an explicit X-checking assertion

assert_never_unknown_async

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (reset_n, test_expr)
```

`test_expr` must never go to an unknown value asynchronously (must stay boolean 0 or 1). **Combinatorial**



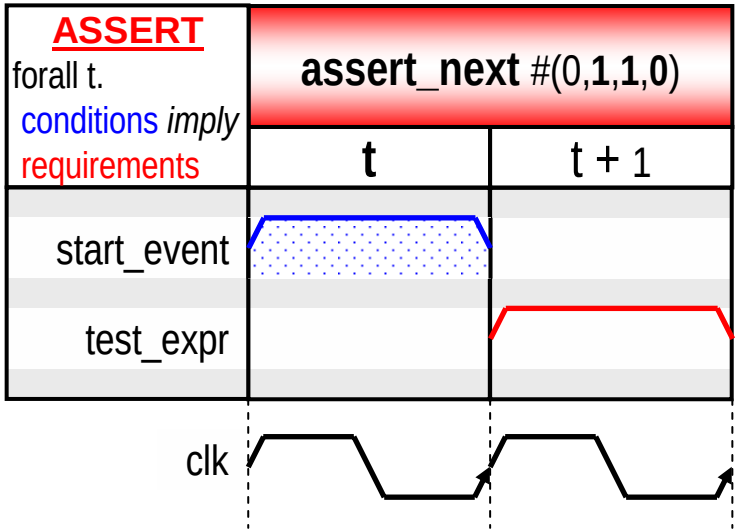
This is the asynchronous version of the clocked assert_never_unknown.

assert_next

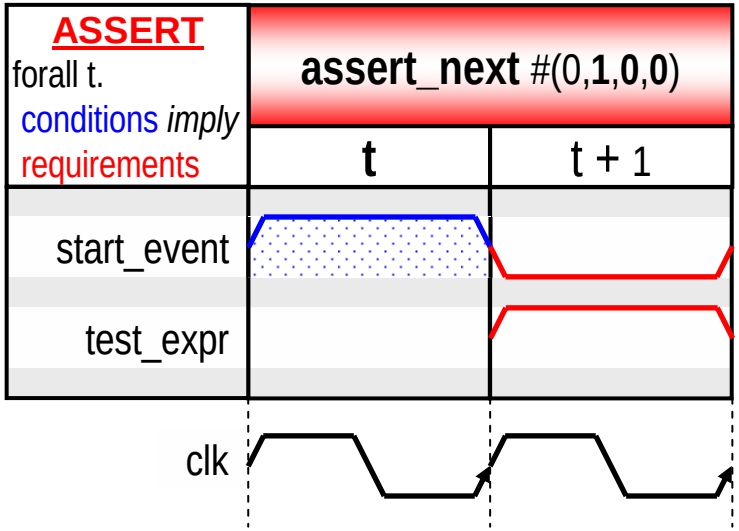

```
#(severity_level, num_cks, check_overlapping,check_missing_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must hold `num_cks` cycles after `start_event` holds

N Cycles



num_cks=1
check_overlapping=1
check_missing_start=0



num_cks=1
check_overlapping=0
check_missing_start=0

requirement on start_event

With check_overlapping at 0, assertion will error if there is a subsequent start_event .

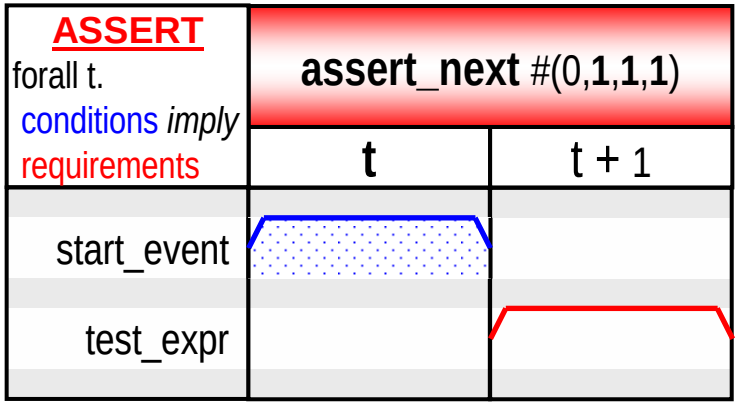
accellera

`assert_next`

```
#(severity_level, num_cks, check_overlapping,check_missing_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must hold `num_cks` cycles after `start_event` holds

N Cycles

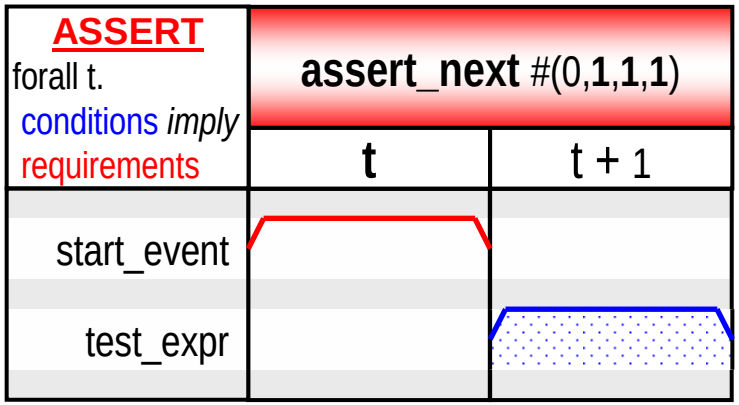


num_cks=1
check_overlapping=1
check_missing_start=1

condition on start_event

+

"check missing start" requires **two timing diagrams**, which together form an *if-and-only-if* check.



num_cks=1
check_overlapping=1
check_missing_start=1

condition on test_expr

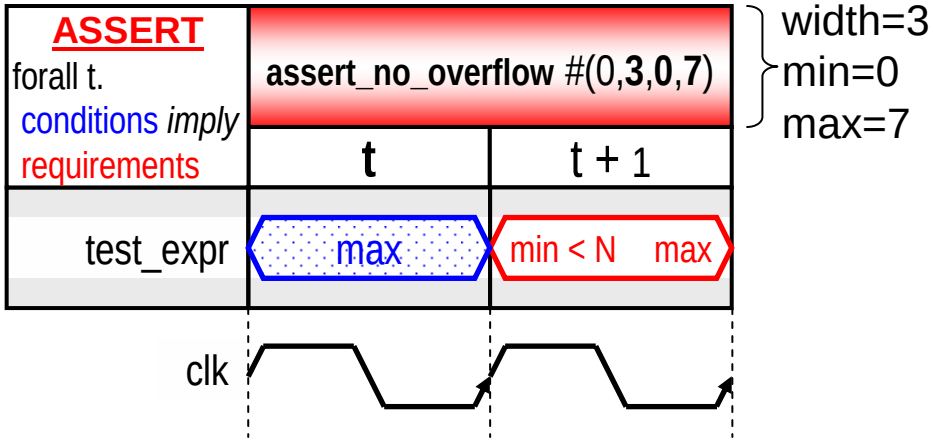
accellera

assert_no_overflow

```
#(severity_level, width, min, max, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

If `test_expr` is at `max`, in the next cycle `test_expr` must be > `min` and `max`

2-Cycles



Example can check that a 3-bit pointer cannot do a wrapping increment from 7 back to 0.

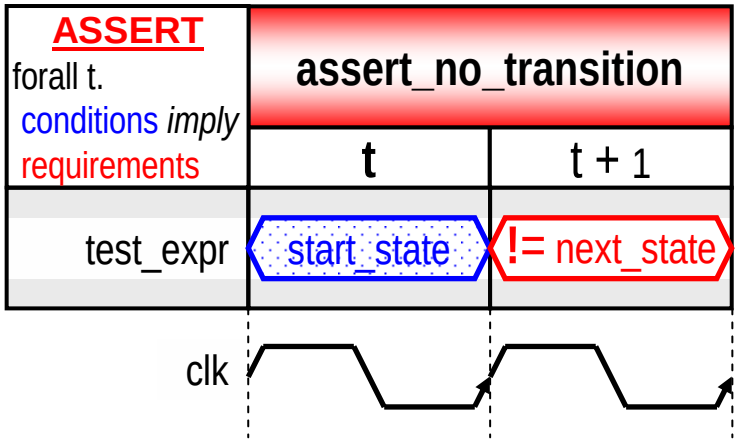
The min and max values do not need to span the full range of test_expr.

assert_no_transition

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr, start_state, next_state)
```

If `test_expr` equals `start_state`, then `test_expr` must not change to `next_state`

2-Cycles

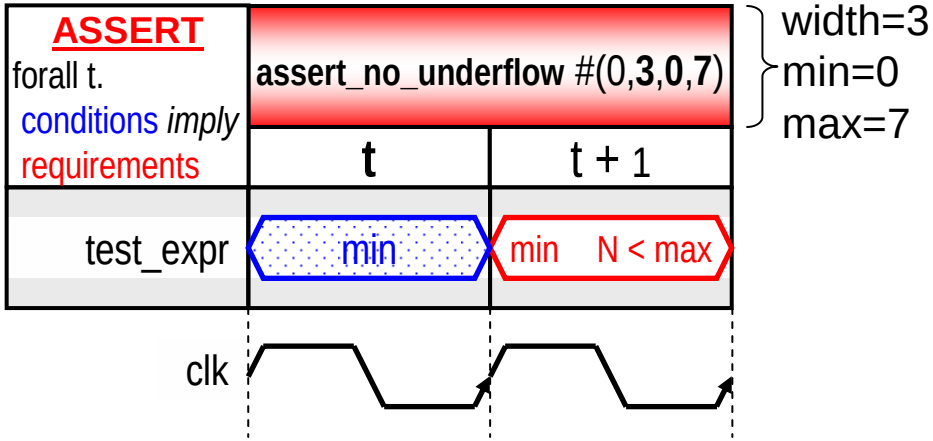


assert_no_underflow

```
#(severity_level, width, min, max,property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

If `test_expr` is at `min`, in the next cycle `test_expr` must be `min` and < `max`

2-Cycles



Example can check that a 3-bit pointer cannot do a wrapping decrement from 0 to 7.

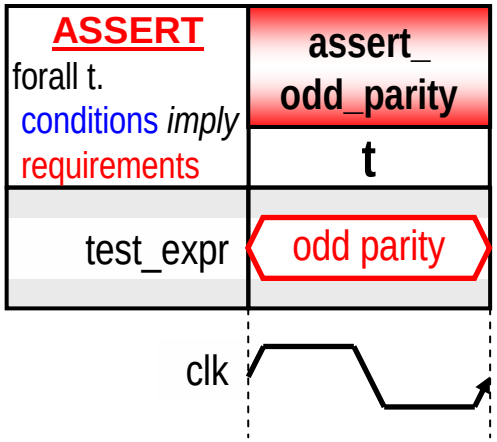
The min and max values do not need to span the full range of test_expr.

assert_odd_parity

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must have an odd parity, i.e. an odd number of bits asserted.

Single-Cycle

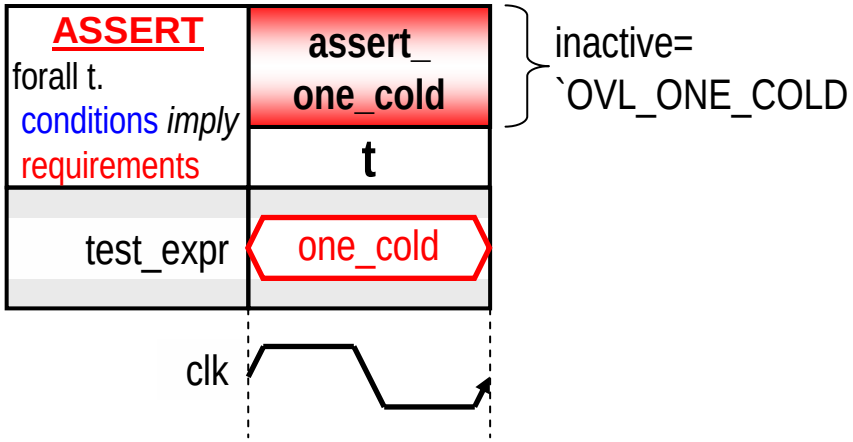


assert_one_cold

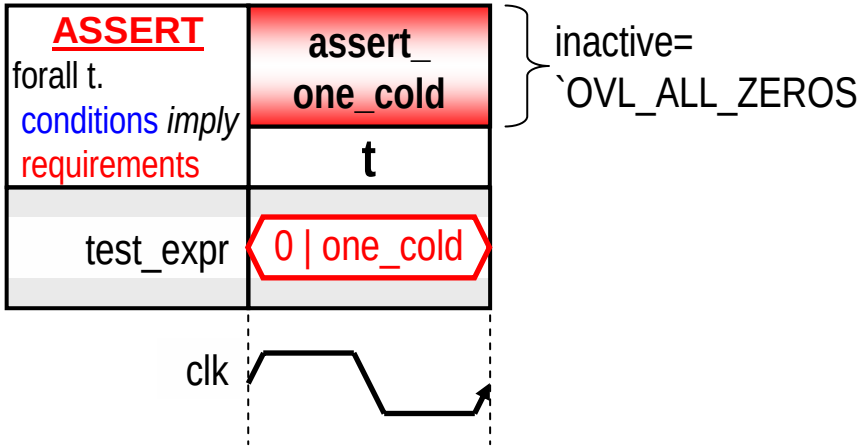
```
#(severity_level, width, inactive, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

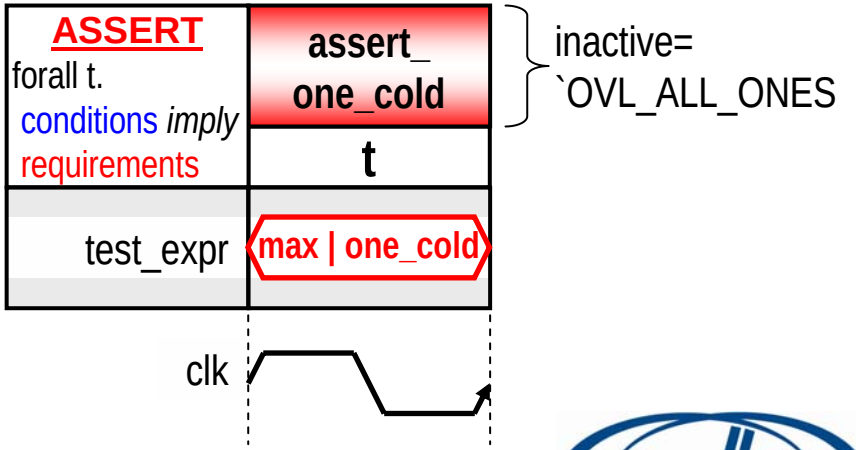
`test_expr` must be one-cold, i.e. exactly one bit set low

Single-Cycle



Unlike one_hot and zero_one_hot, just one configurable OVL is used for one_cold.





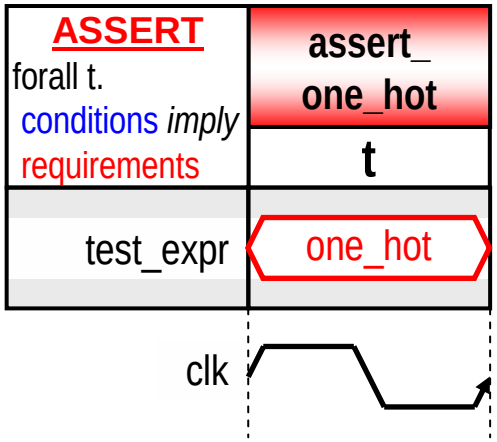
accellera

assert_one_hot

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must be one-hot, i.e. exactly one bit set high

Single-Cycle



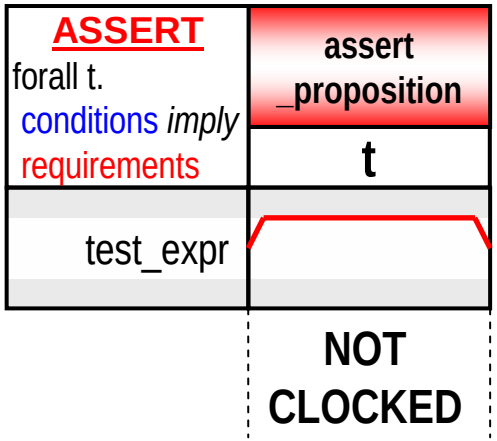
accellera

assert_proposition

```
#(severity_level, property_type, msg, coverage_level)
u1 (reset_n, test_expr)
```

`test_expr` must hold asynchronously (not just at a clock edge)

Combinatorial



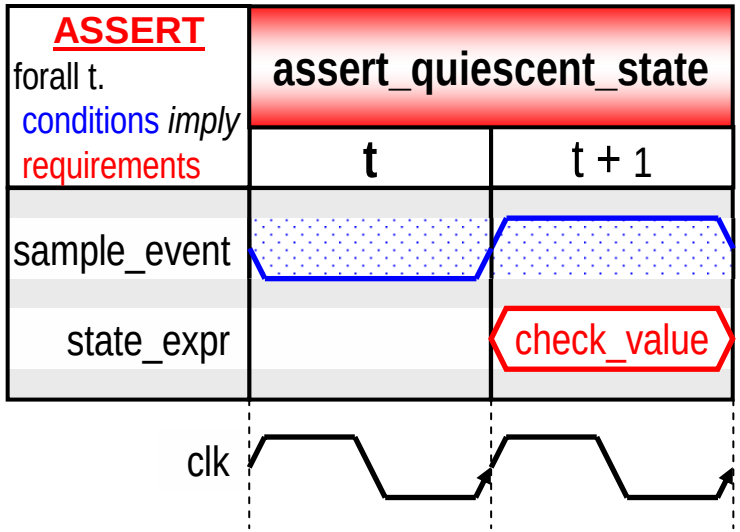
This is an asynchronous version of the clocked assert_always

assert_quiescent_state

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, state_expr, check_value, sample_event)
```

`state_expr` must equal `check_value` on a rising edge of `sample_event`

2-Cycles



Can also be checked on rising edge of:
`OVL_END_OF_SIMULATION
Used for extra check at simulation end.

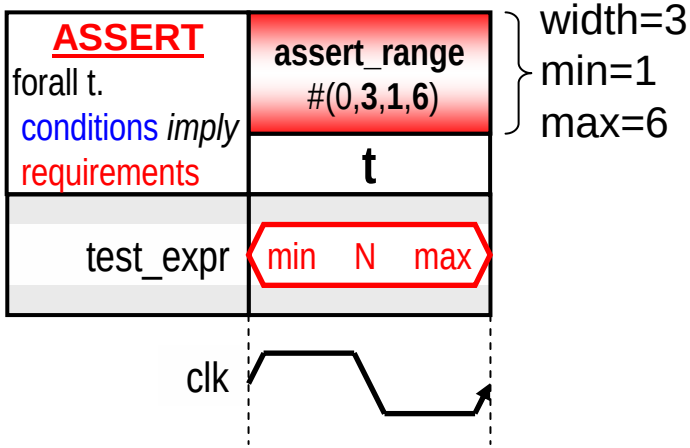
Can *just* trigger at end of simulation by setting sample_event to 1'b0 and defining:
`OVL_END_OF_SIMULATION

assert_range

```
#(severity_level, width, min, max, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must be `min` and `max`

Single-Cycle

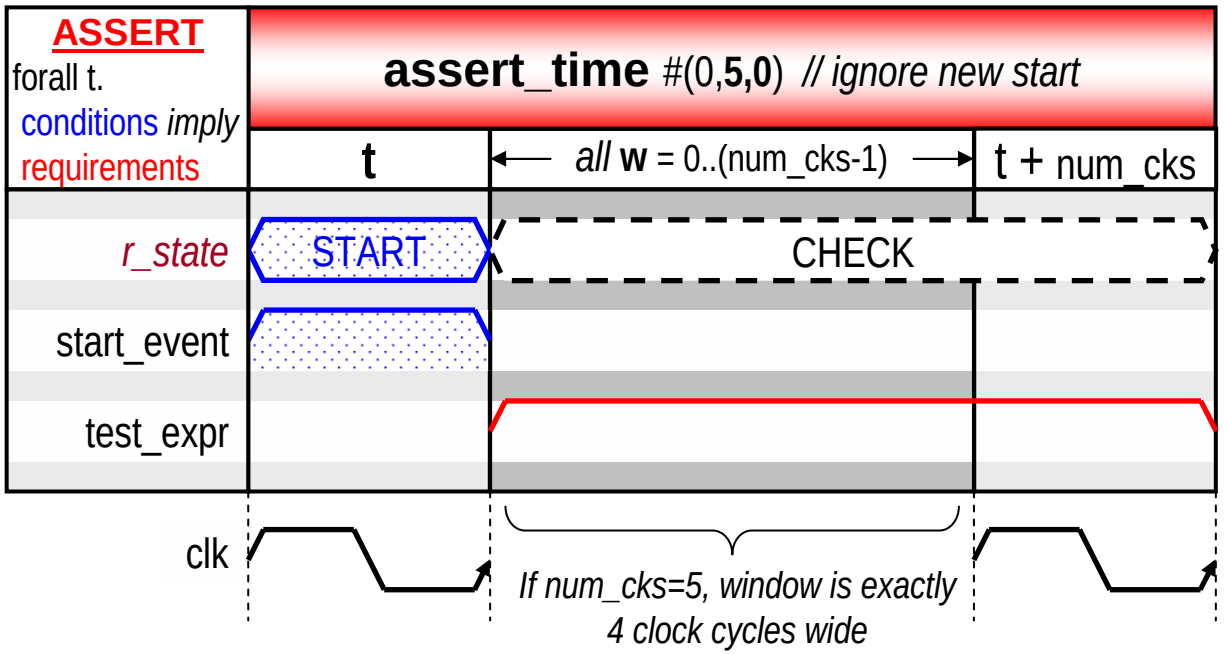


assert_time


```
#(severity_level, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must hold for `num_cks` cycles after `start_event`

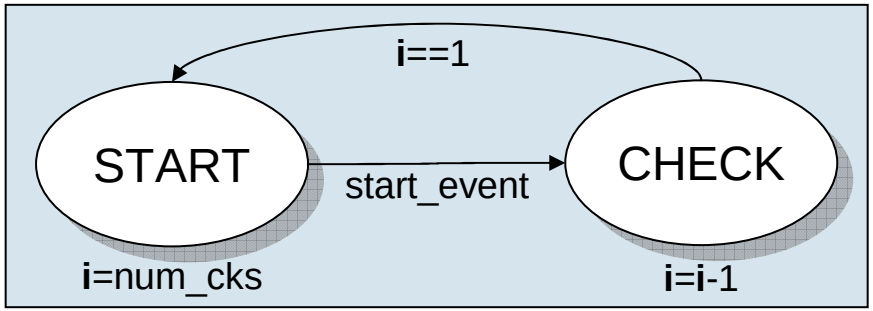
n-Cycles



num_cks=5
action_on_new_start=0
(`OVL_IGNORE_NEW_START)

Only passes if test_expr is high for *all* cycles:
t+1, t+2, …, t+num_cks
Fails if test_expr is low in any of these cycles.

r_state (auxiliary logic)



Auxiliary logic necessary, to *ignore* new start.
Checking only begins after start_event is true *and* r_state==START.

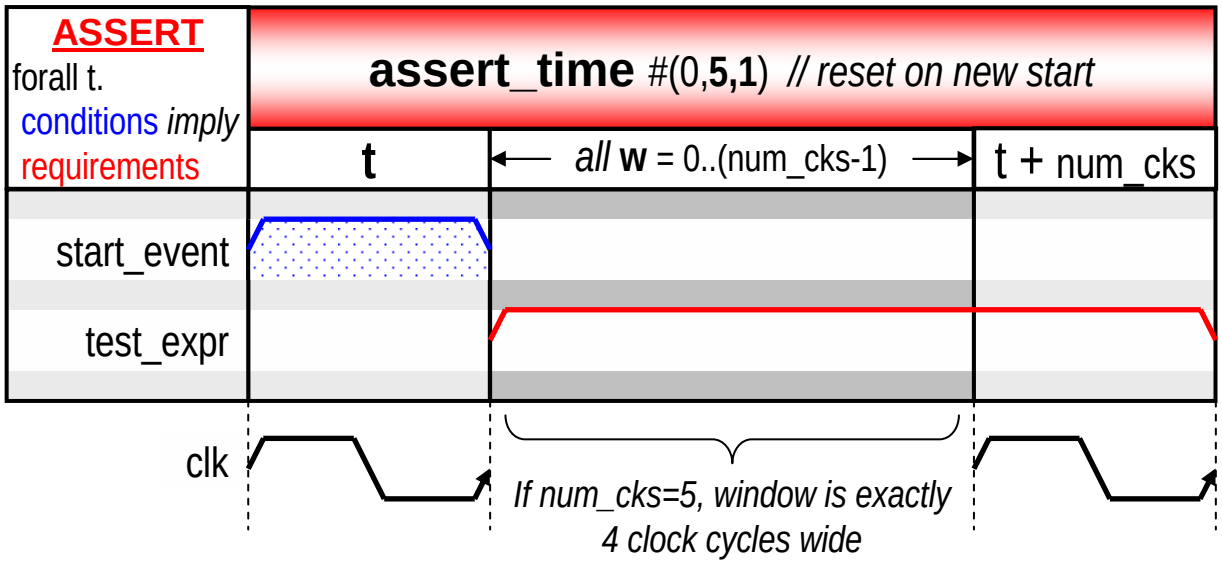
accellera

assert_time

```
#(severity_level, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

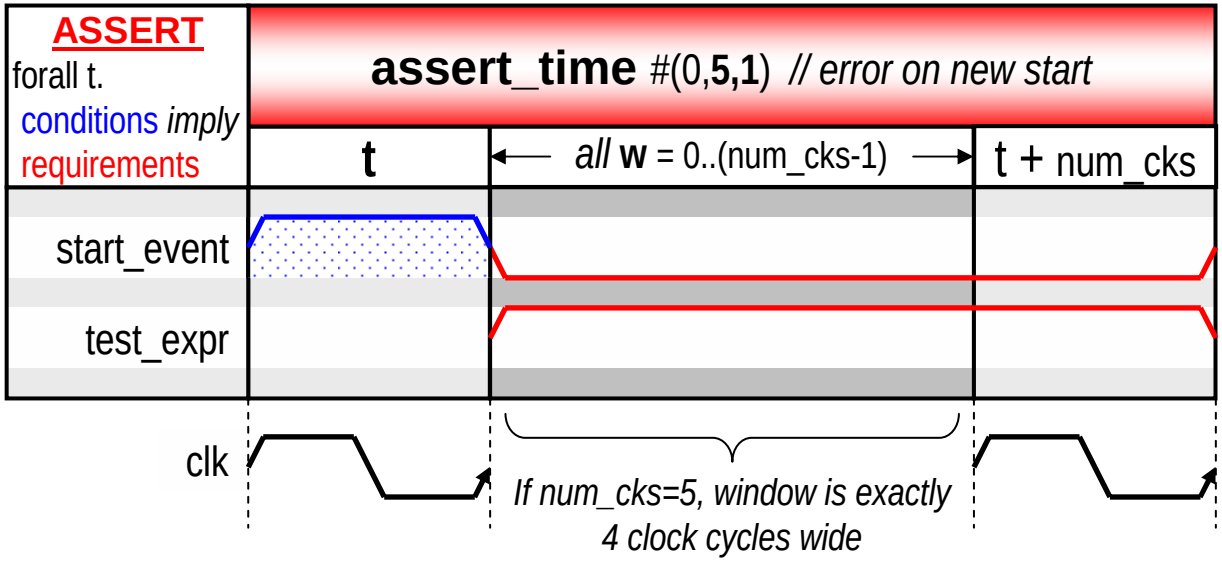
`test_expr` must hold for `num_cks` cycles after `start_event`

n-Cycles



num_cks=5
action_on_new_start=1
(`OVL_RESET_ON_NEW_START)

For assert_time, "reset on new start" effectively perfoms *pipelined* checking (see note 2).



num_cks=5
action_on_new_start=2
(`OVL_ERROR_ON_NEW_START)

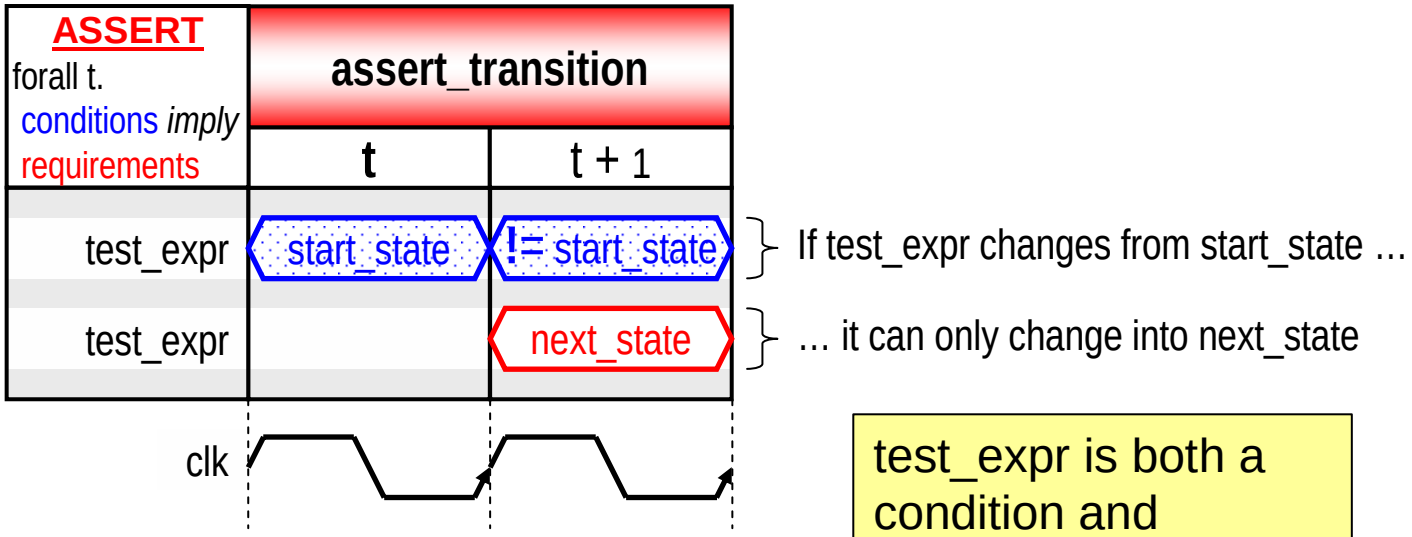
requirement on start_event

assert_transition

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr, start_state, next_state)
```

If `test_expr` changes from `start_state`, then it can only change to `next_state`

2-Cycles



test_expr can remain in start_state (in which case the condition at t+1 does not hold).

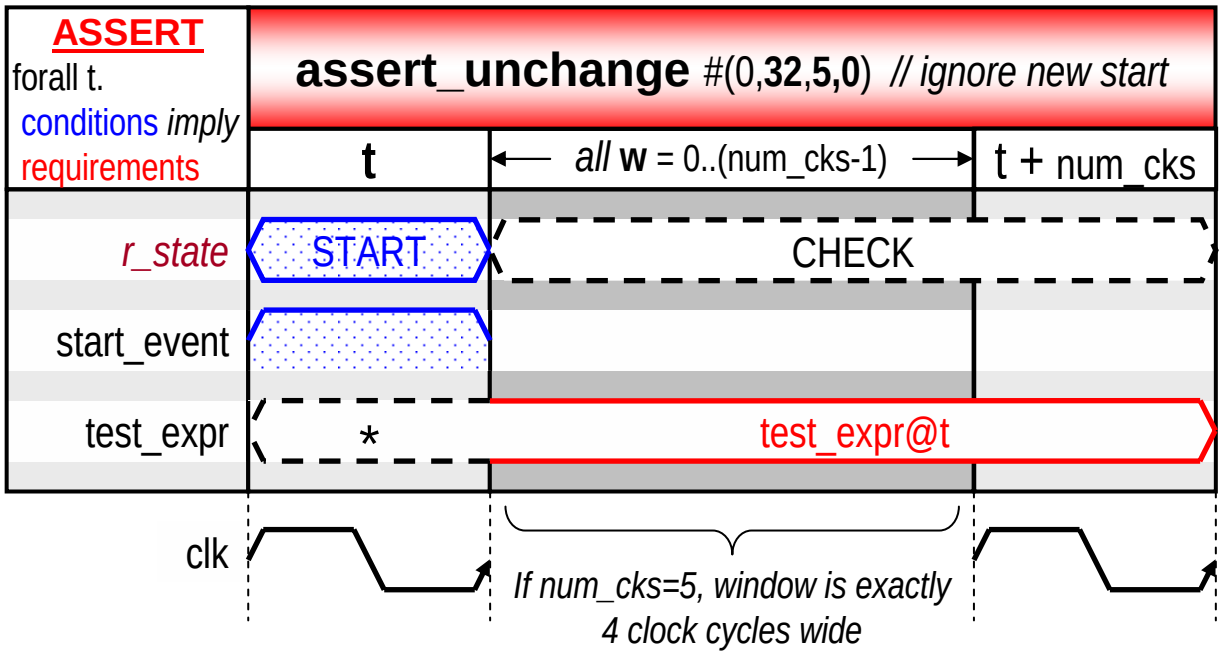
test_expr is both a condition and requirement at t+1. Hence it appears on two rows.

assert_unchange

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not change within `num_cks` cycles of `start_event`

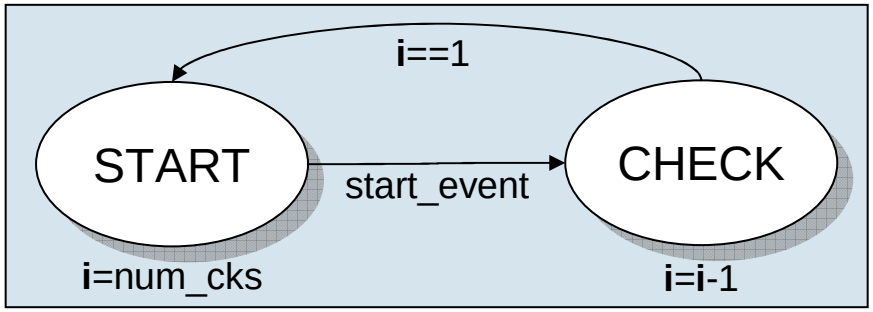
n-Cycles



num_cks=5
action_on_new_start=0
(`OVL_IGNORE_NEW_START)

Only passes if test_expr is stable for *all* cycles:
t+1, t+2, …, t+num_cks
Fails if test_expr changes in any of these cycles.

r_state (auxiliary logic)



Need auxiliary logic to be able to *ignore* new start. Checking only begins after start_event is true *and* r_state==START.

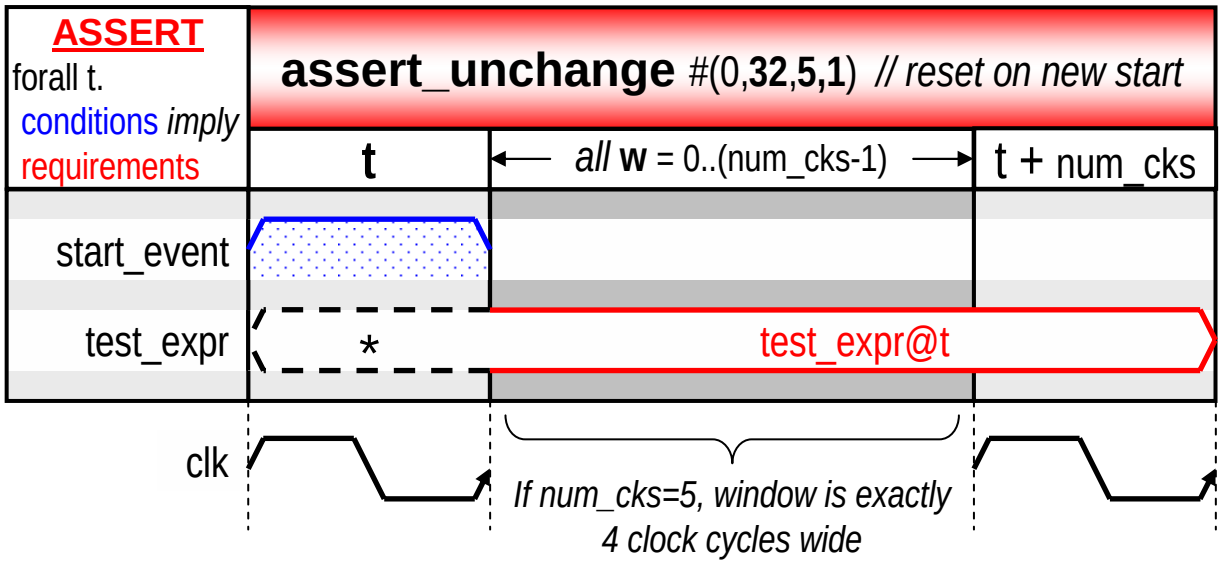
accellera

assert_unchange

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

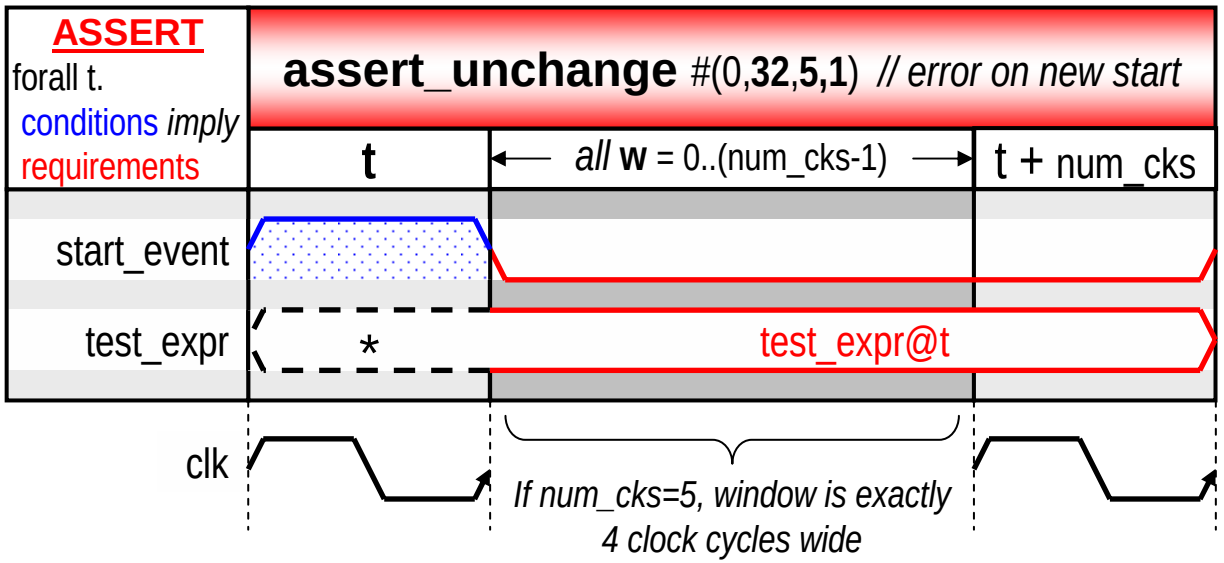
`test_expr` must not change within `num_cks` cycles of `start_event`

n-Cycles



num_cks=5
action_on_new_start=1
(`OVL_RESET_ON_NEW_START)

For assert_unchange "reset on new start" is *pipelined* checking. Don't need auxiliary logic to express this.



num_cks=5
action_on_new_start=2
(`OVL_ERROR_ON_NEW_START)

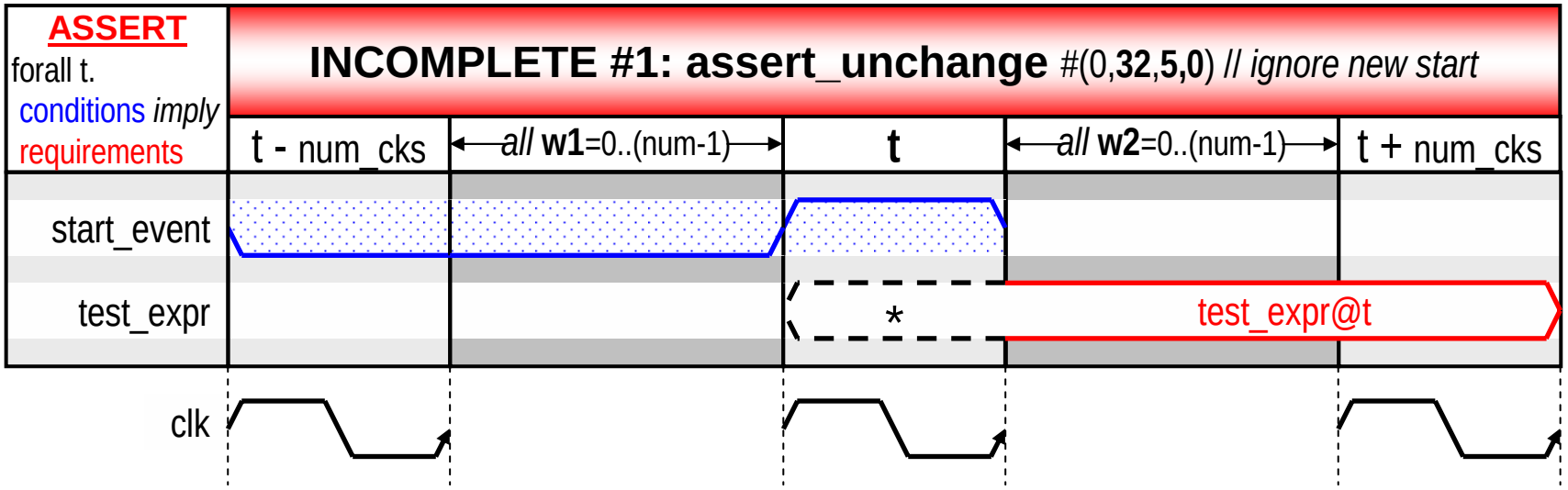
requirement on start_event

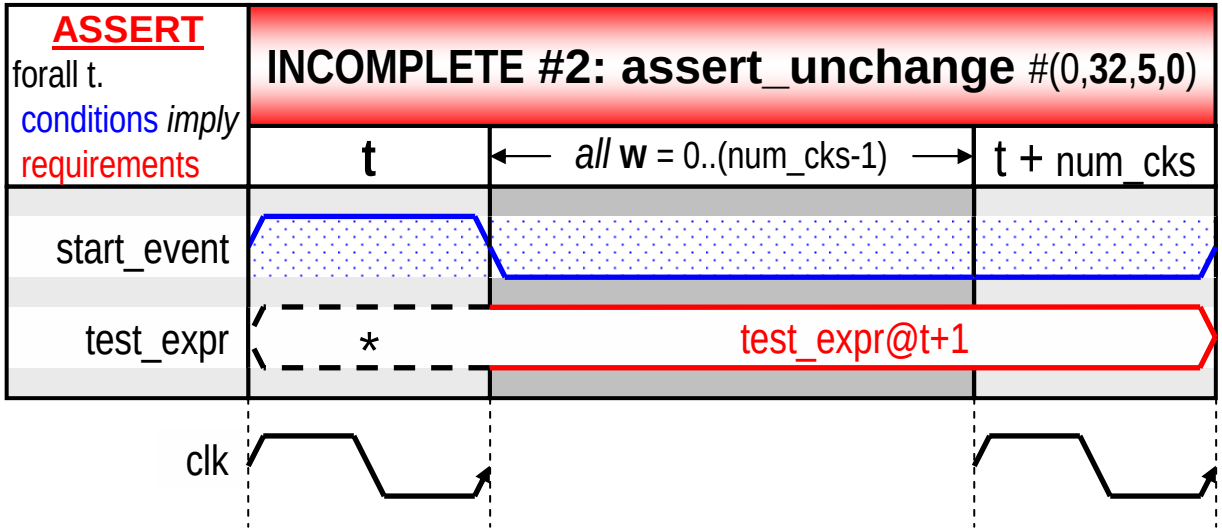
assert_unchange

```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr)
```

`test_expr` must not change within `num_cks` cycles of `start_event`

n-Cycles





Both timing diagrams are incomplete for "ignore new start", as start_event=0 will mask some errors!

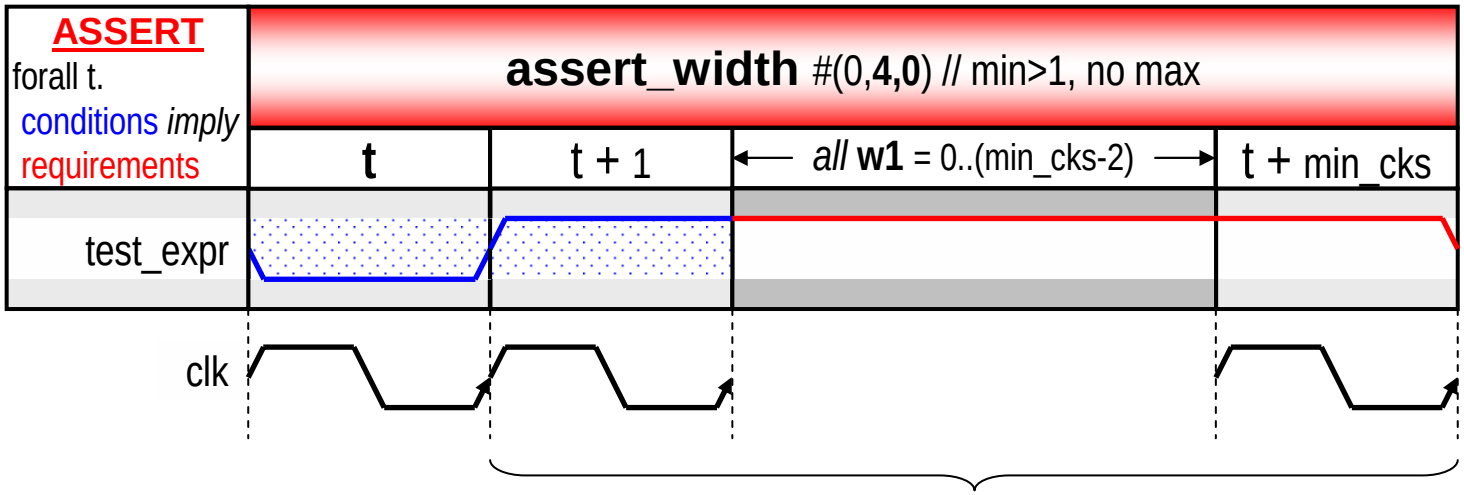
accellera

assert_width

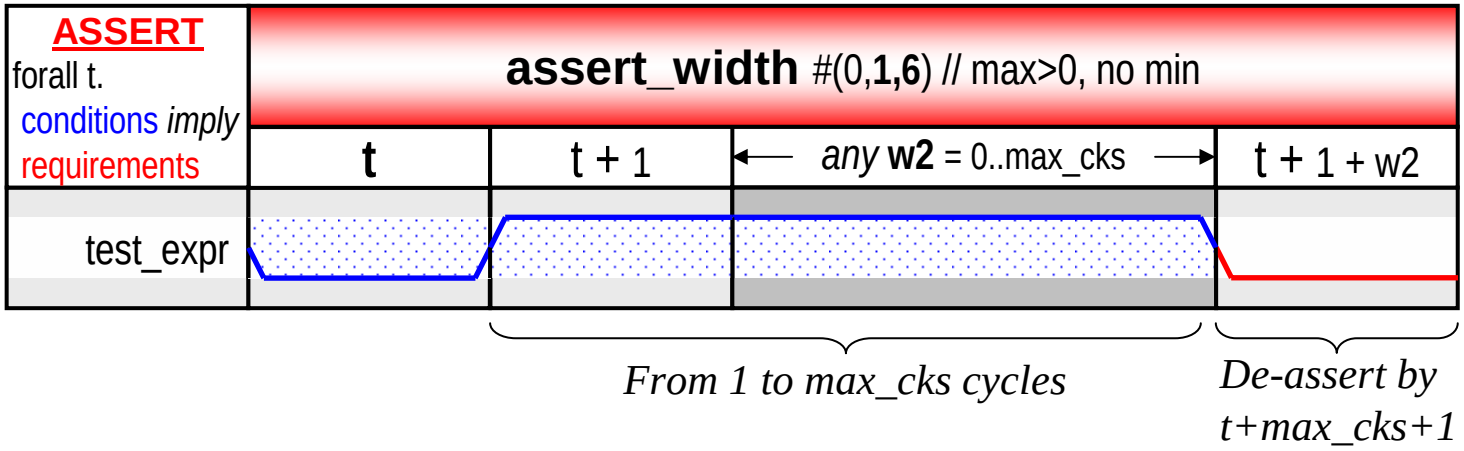
```
#(severity_level, min_cks, max_cks,property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must hold for between `min_cks` and `max_cks` cycles

n-Cycles



min_cks>1
max_cks=0
(just check min)



min_cks<2
max_cks>0
(just check max)

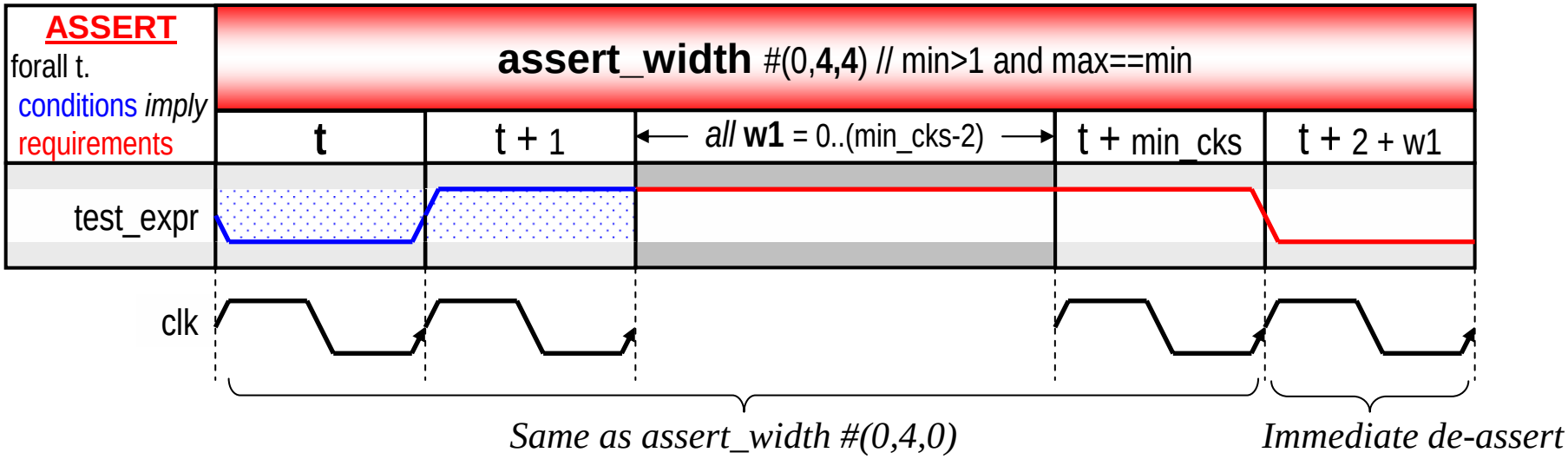
accellera

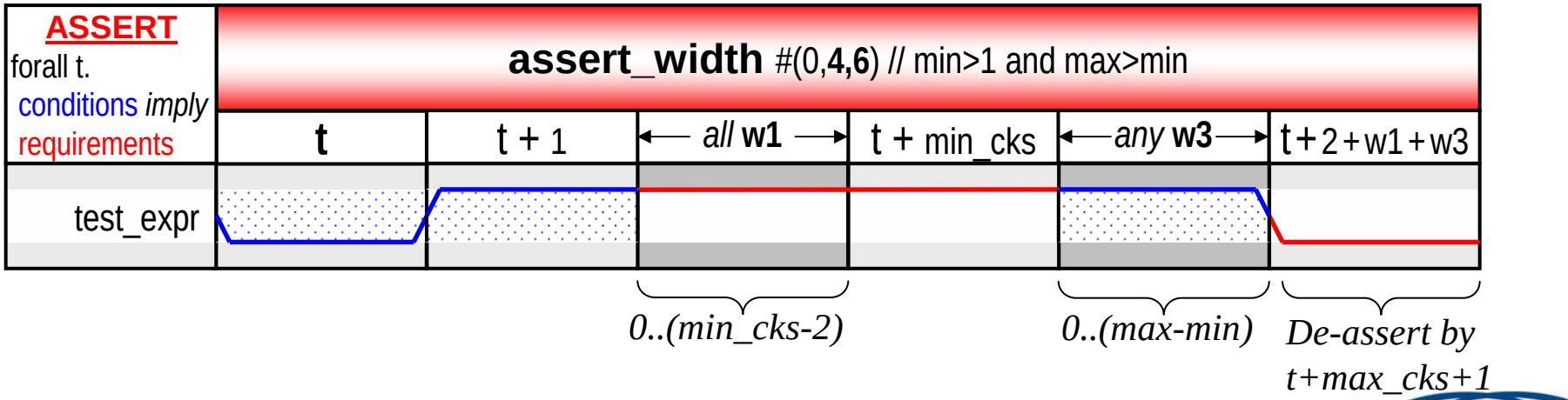
assert_width

```
#(severity_level, min_cks, max_cks,property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must hold for between `min_cks` and `max_cks` cycles

n-Cycles





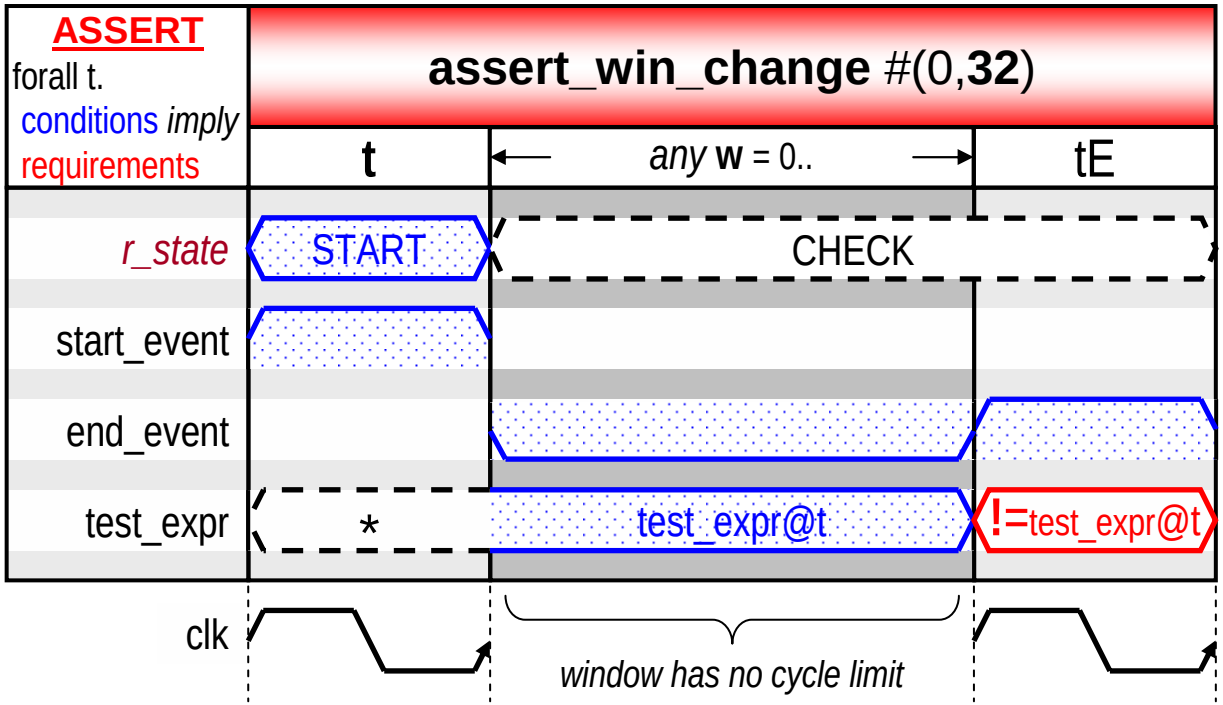
accellera

assert_win_change

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr, end_event)
```

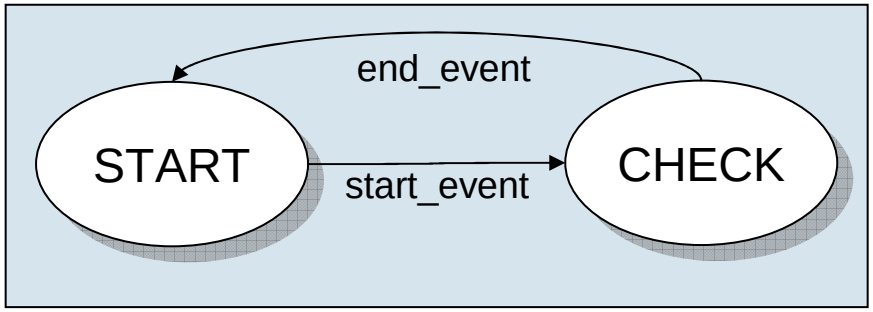
`test_expr` must change between `start_event` and `end_event`

Event-bound



Will pass if test_expr changes at *any* cycle during window: t+1, … Fails if test_expr is stable for all cycles after start.

r_state (auxiliary logic)



Auxiliary logic necessary, to *ignore* new start. Checking only begins after start_event is true *and* r_state==START.

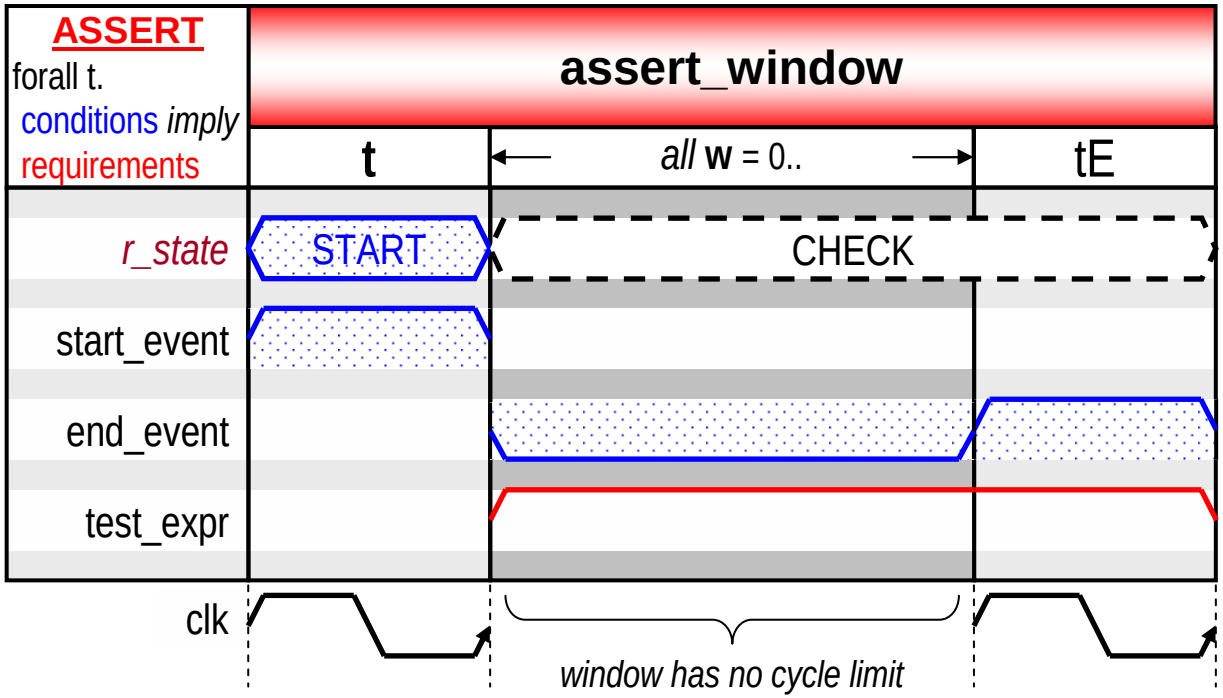
accellera

assert_window

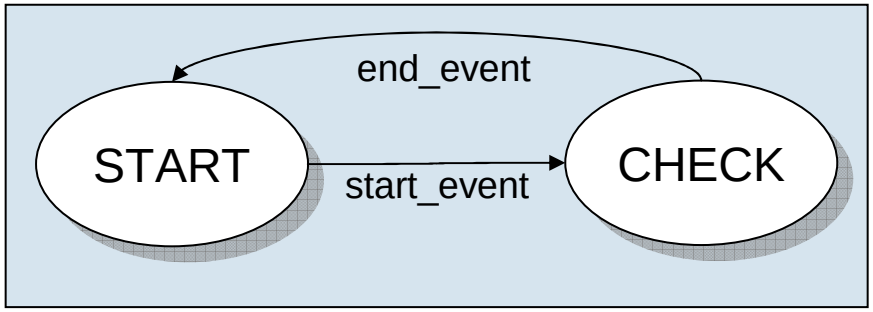
```
#(severity_level, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr, end_event)
```

`test_expr` must hold after the `start_event` and up to (and including) the `end_event`

Event-bound



r_state (auxiliary logic)



Auxiliary logic necessary, to *ignore* new start. Checking only begins after start_event is true *and* r_state==START.

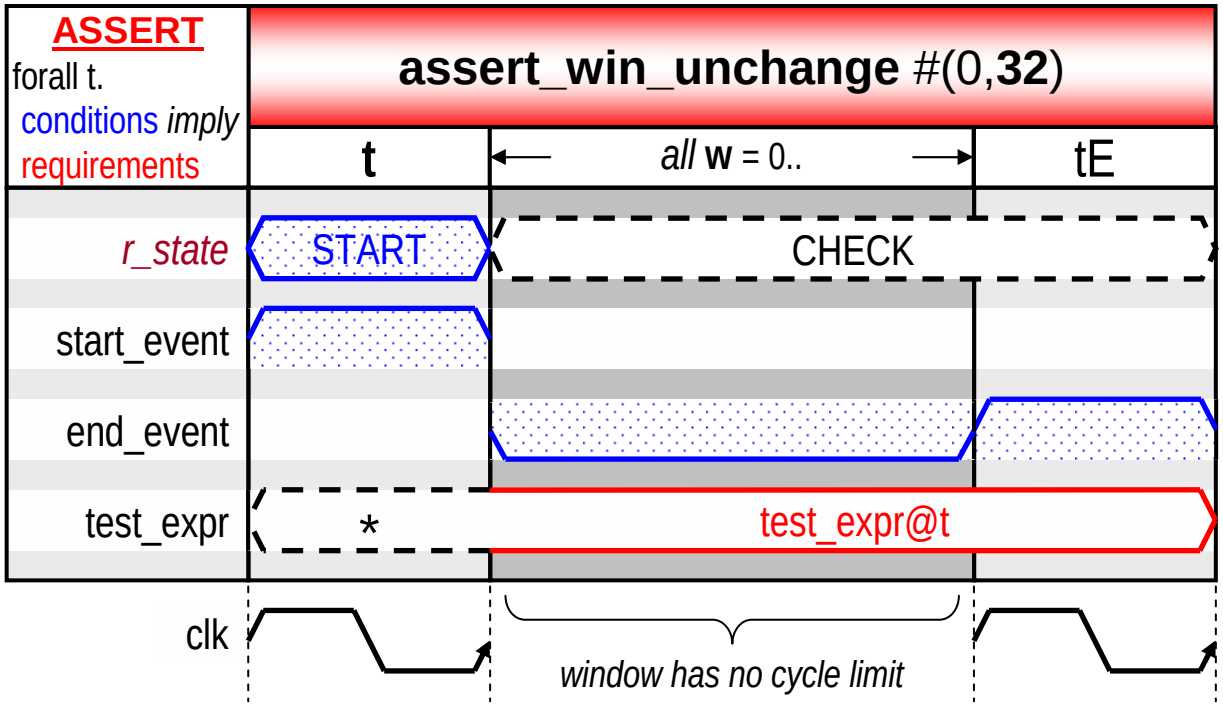
accellera

assert_win_unchange

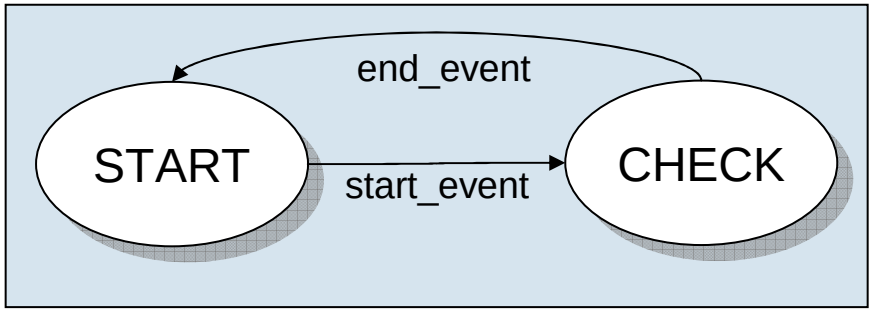
```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, start_event, test_expr, end_event)
```

`test_expr` must not change between `start_event` and `end_event`

Event-bound



r_state (auxiliary logic)



Auxiliary logic necessary, to *ignore* new start. Checking only begins after start_event is true *and* r_state==START.

accellera

assert_zero_one_hot

```
#(severity_level, width, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)
```

`test_expr` must be one-hot or zero, i.e. at most one bit set high

Single-Cycle