



## Timing Diagrams for Accellera Standard OVL V2.1

`assert_<checker> modules`

Mike Turpin / ARM  
15<sup>th</sup> September 2007

# Contents

- Introduction to OVL
  - Types of OVL
  - OVL Release History & Major Changes
    - pre-Accellera Apr 2003
    - v1.0 May 2005
    - v1.1, v1.1a, b Aug 2005
    - v1.5 Dec 2005
    - v1.6 Mar 2006
    - v1.7 July 2006
    - v1.8 Oct 2006
    - V2.0 Jun 2007 (Beta in April)
    - V2.1 Sep 2007
- Introduction to Timing Diagrams
  - Timing Diagram Syntax & Semantics
  - Timing Diagram Template
- Assert Timing Diagrams (alphabetical order)



# Types of OVL Assertion

Combinatorial

## Combinatorial Assertions

- `assert_proposition`, `assert_never_unknown_async`

Single-Cycle

## Single-cycle Assertions

- `assert_always`, `assert_implication`, `assert_range`, ...

2-Cycles

## Sequential over 2 cycles

- `assert_always_on_edge`, `assert_decrement`, ...

$n$ -Cycles

## Sequential over `num_cks` cycles

- `assert_change`, `assert_cycle_sequence`, `assert_next`, ...

Event-bound

## Sequential between two events

- `assert_win_change`, `assert_win_unchange`, `assert_window`



# OVL Release History and Major Changes

- pre-Accellera, April 2003
  - Verilog updated in April, but VHDL still October 2002
- v1.0, May 2005
  - Changed:
    - assert\_change (window can no longer finish before num\_cks-1 cycles)
    - assert\_fifo\_index (property\_type removed from functionality)
    - assert\_time/unchange (RESET\_ON\_NEW\_START corner case)
- v1.1, July 2005
  - New: assert\_never\_unknown
  - Changed:
    - assert\_implication: antecedent\_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



# OVL Release History and Major Changes

- v2.0-Beta, April 2007
  - ovl\_<checker> modules (not documented here)
    - enable input & fire output (tied low in beta)
    - clock\_edge, reset\_polarity & gating\_type parameters
  - 17 new ovl\_<checker> modules (in SVA)
  - Preliminary VHDL release (10 checkers)
  - Bug fixes
- v2.0, June 2007
  - fire output implemented in top-10 Verilog OVLs
    - Also implemented in VHDL checkers
    - Still tied low in SVA & PSL versions
  - `OVL\_ASSERT\_2STATE & `OVL\_ASSUME\_2STATE
    - property\_type values for local X-checking disable
  - Bug fixes



# OVL Release History and Major Changes

- v2.1, Sept 2007
  - Bug fixes

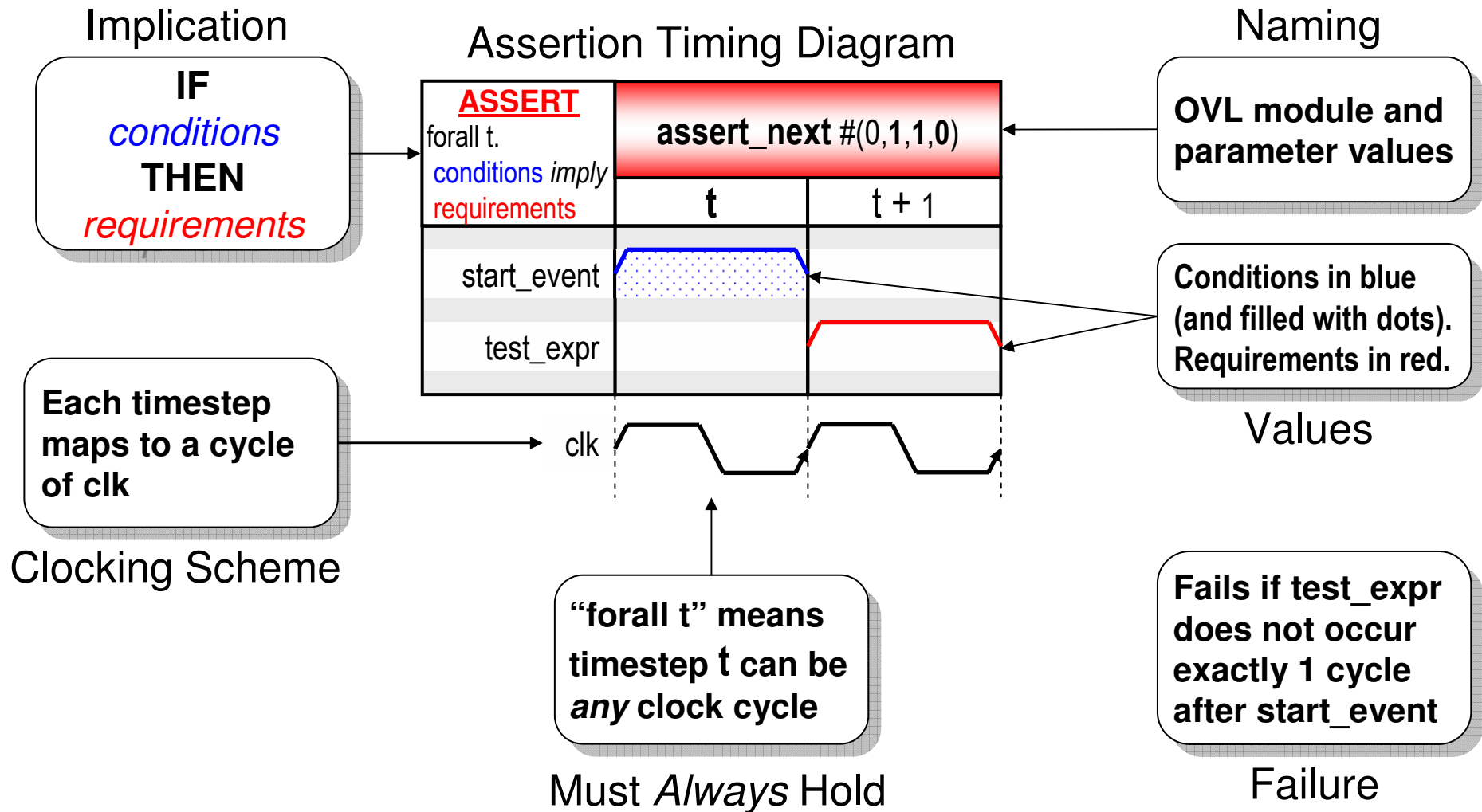




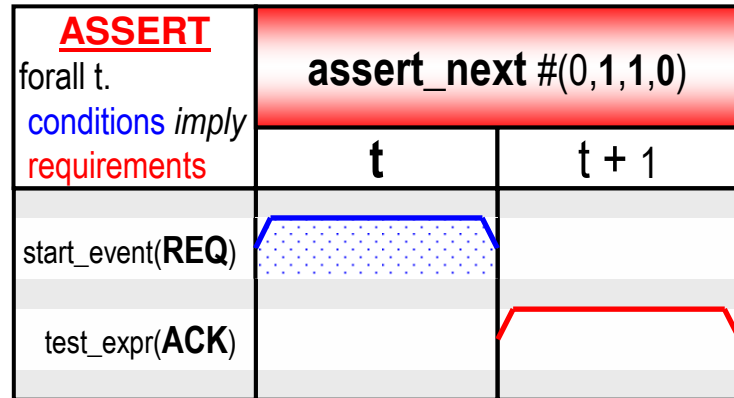
## Introduction to Timing Diagrams

- Timing Diagram Syntax & Semantics
- Timing Diagram Template

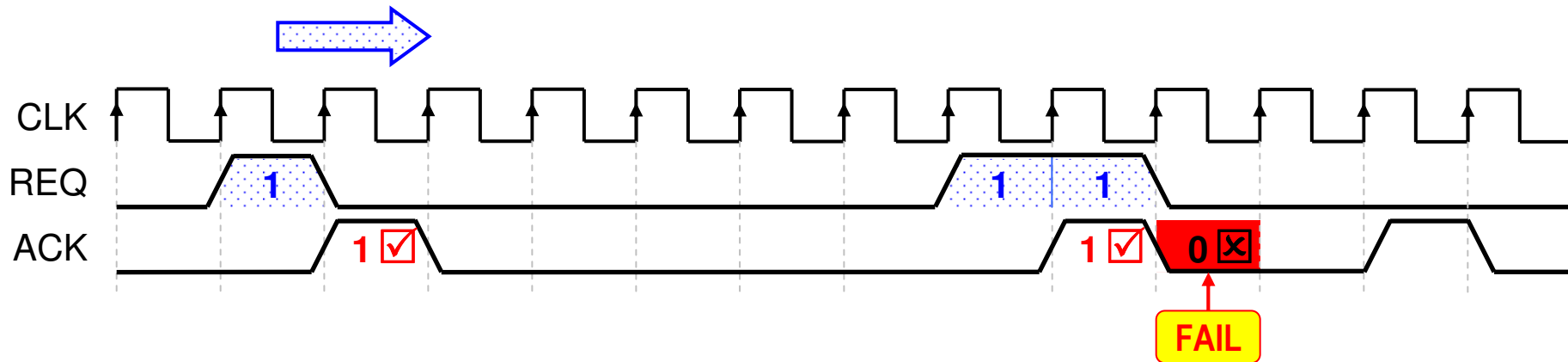
# Introduction: OVL Timing Diagram



# 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.







## Assert Timing Diagrams

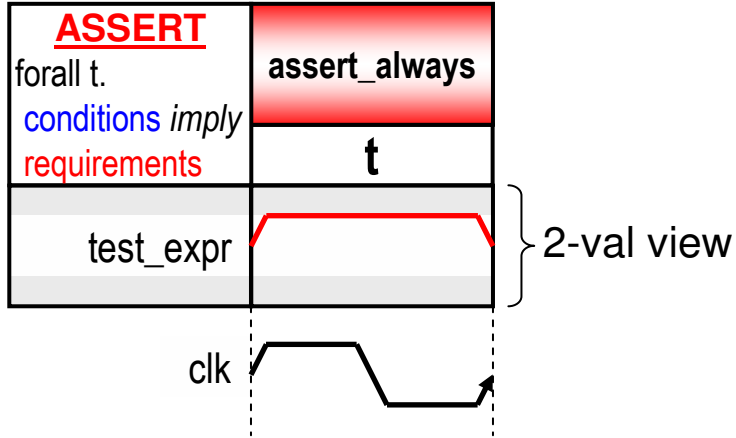
33 assert\_<checker> modules

*does not include ovl\_<checker> modules*

# assert\_always

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

test\_expr must always hold Single-Cycle



assert\_always will also *pessimistically* fail if test\_expr is X

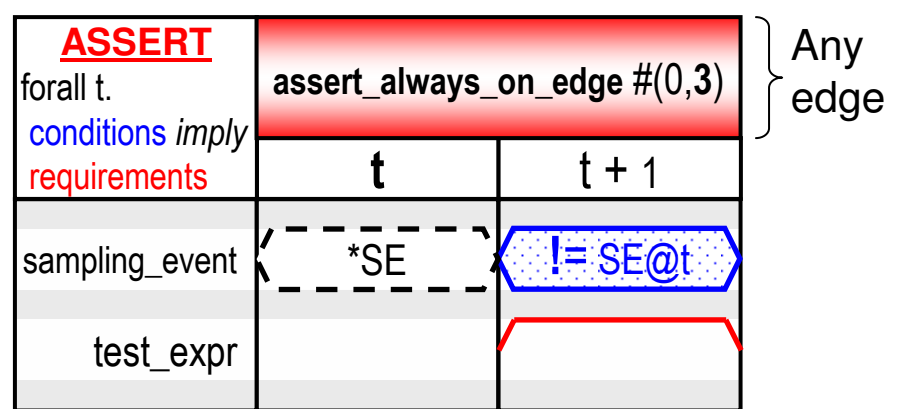
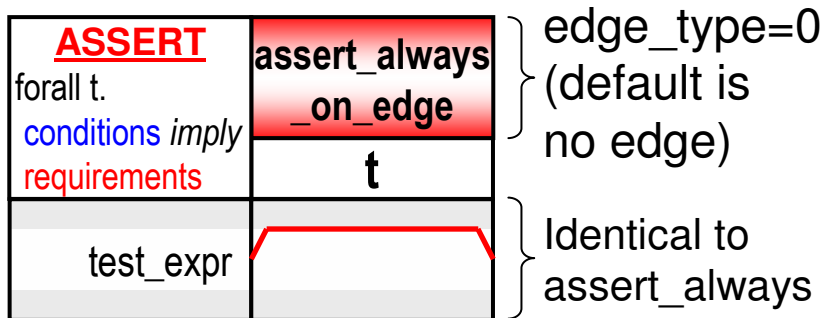
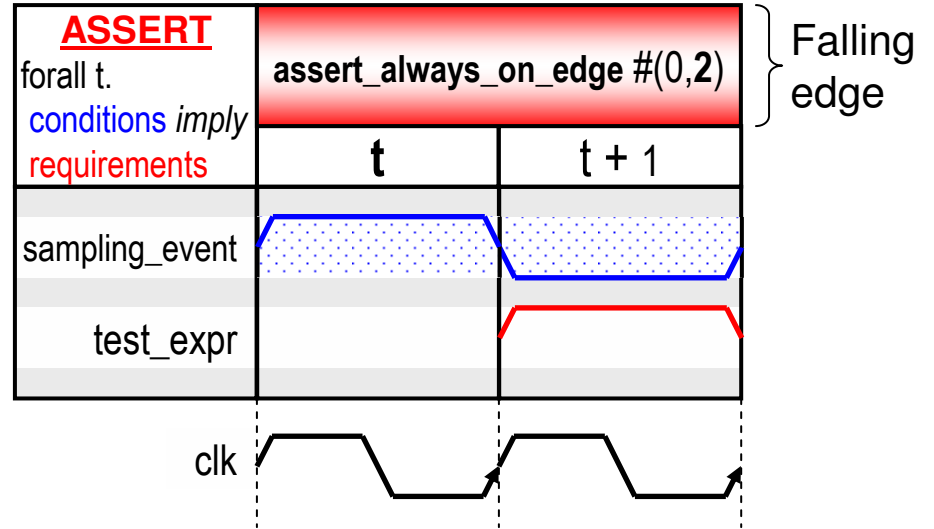
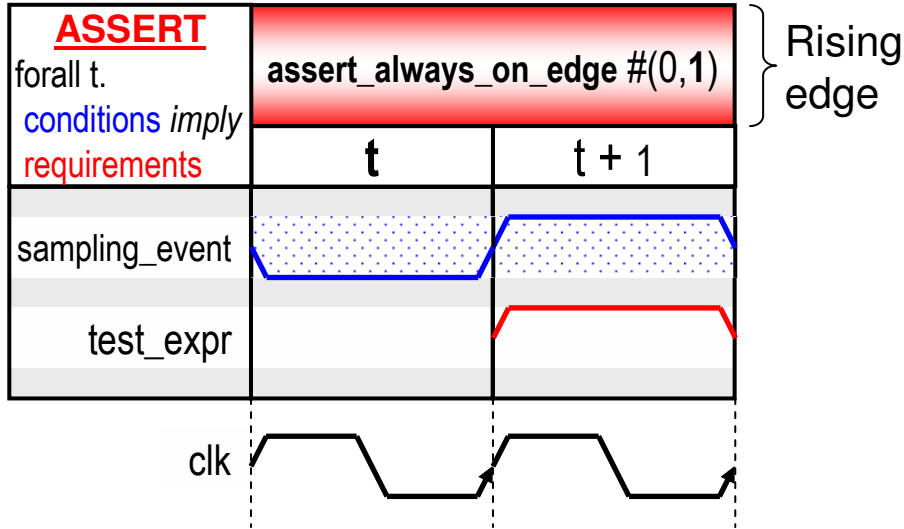
Can disable failure on X/Z via:  
  
global: OVL\_XCHECK\_OFF  
local: OVL\_ASSERT\_2STATE



# assert\_always\_on\_edge

```
#(severity_level, edge_type, property_type, msg, coverage_level)
ul (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

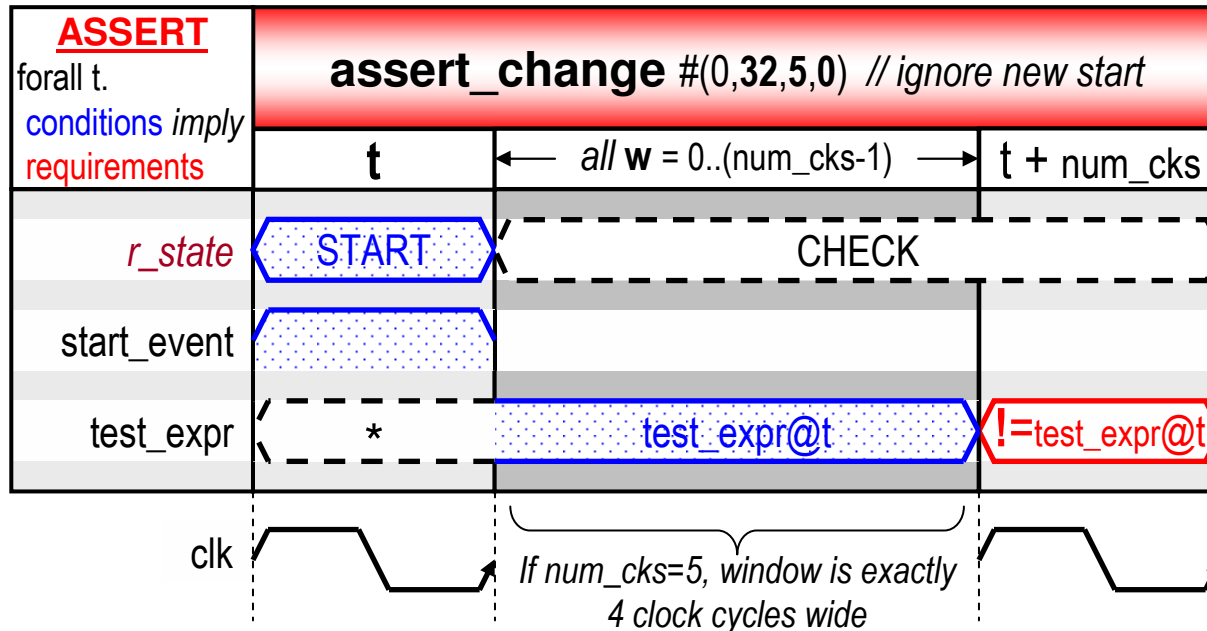
```

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

```

n-Cycles

test\_expr must change within num\_cks cycles of start\_event

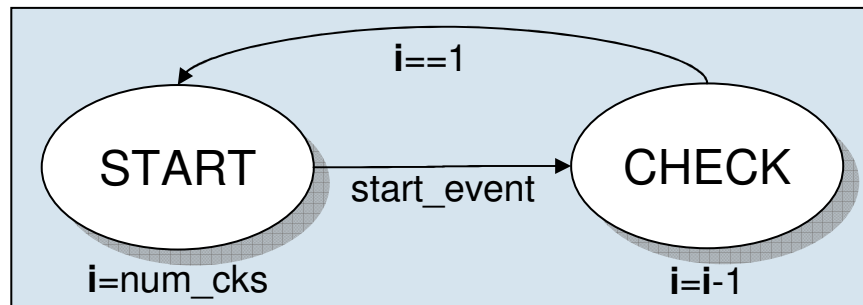


num\_cks=5  
 action\_on\_new\_start=0  
 ('OVL\_IGNORE\_NEW\_START)

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 (even if it finishes early).

### r\_state (auxiliary logic)



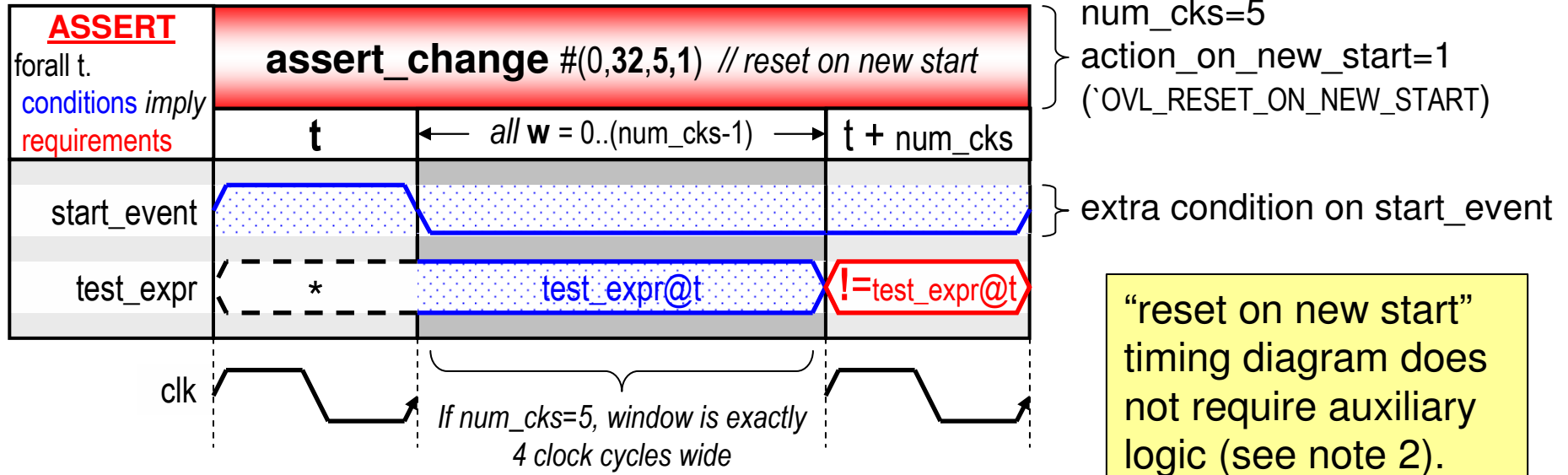
Auxiliary logic necessary, to ignore new start. Checking only begins after start\_event is true and r\_state==START.



# assert\_change

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

test\_expr must change within num\_cks cycles of start\_event n-Cycles



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

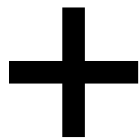
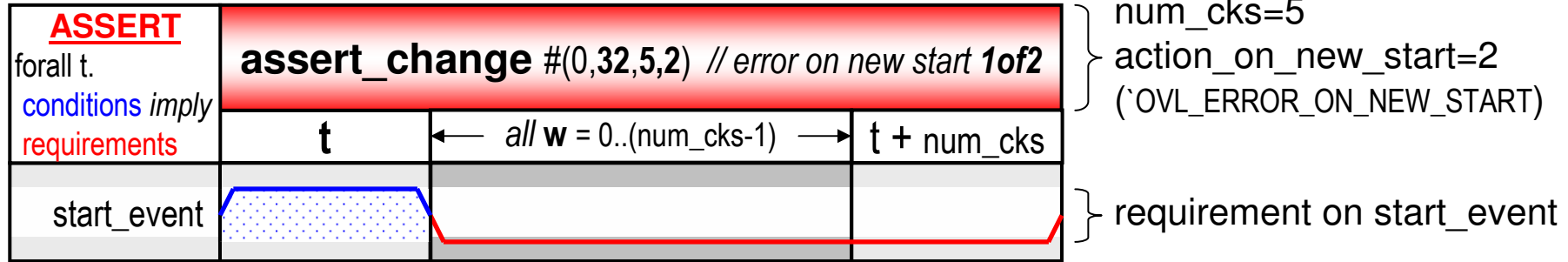
**Differs to April 2003**  
From OVL version 1.0 the check window spans the entire `num_cks-1` cycles.



### assert\_change

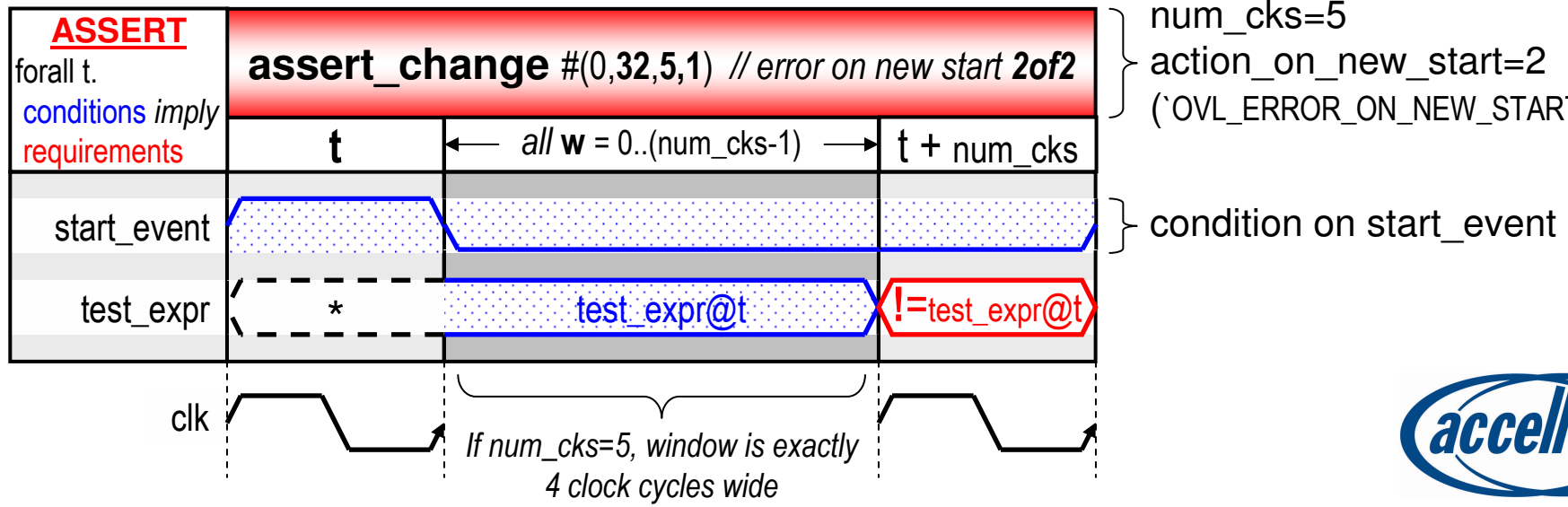
```
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
ul (clk, reset_n, start_event, test_expr)
```

test\_expr must change within num\_cks cycles of start\_event n-Cycles



“error on new start” requires **two timing diagrams**, with 2<sup>nd</sup> 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.

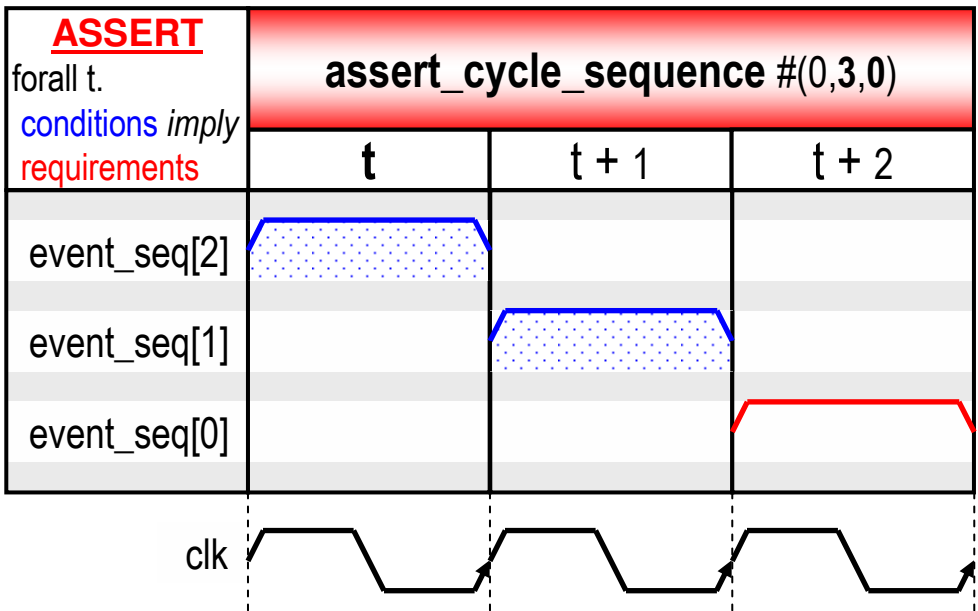


### assert\_cycle\_sequence

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

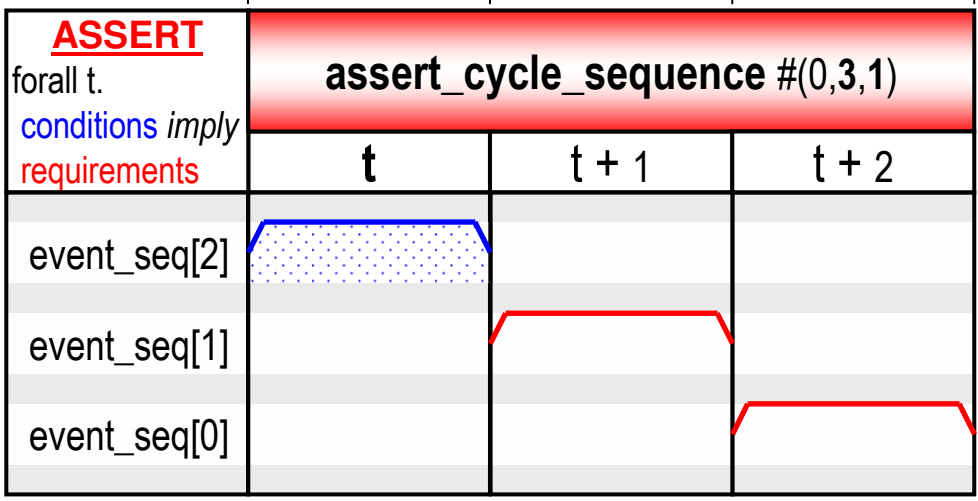
If the initial sequence holds, the final sequence must also hold (final is 1-cycle or N-1 cycles)

**n-Cycles**



num\_cks=3  
necessary\_condition=0  
(`OVL\_TRIGGER\_ON\_MOST\_PIPE)

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



num\_cks=3  
necessary\_condition=1  
(`OVL\_TRIGGER\_ON\_FIRST\_PIPE)



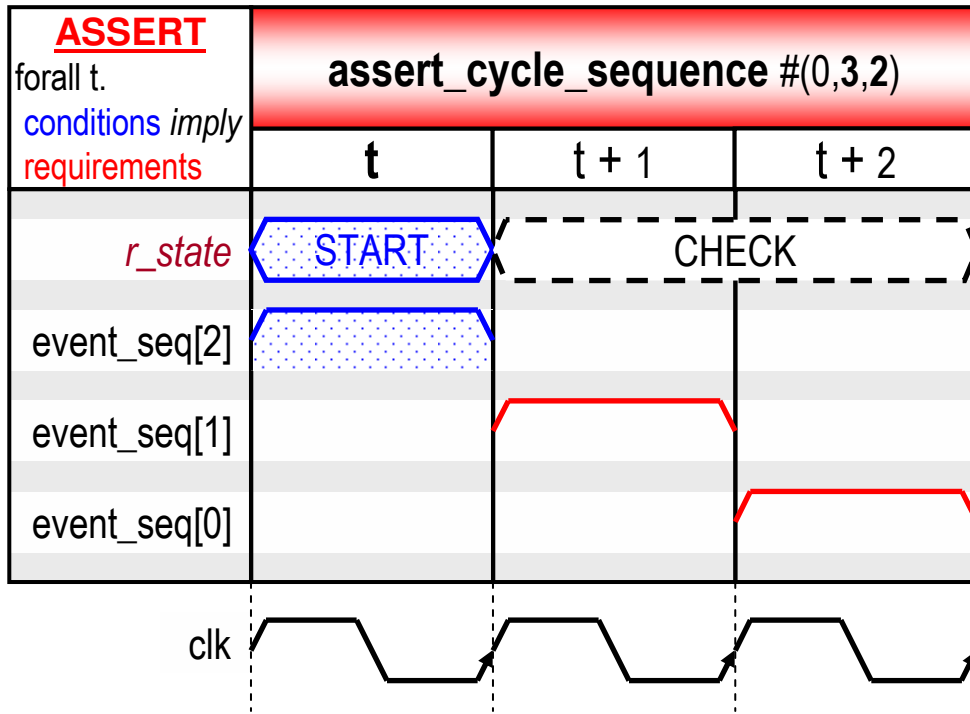
### assert\_cycle\_sequence

```

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

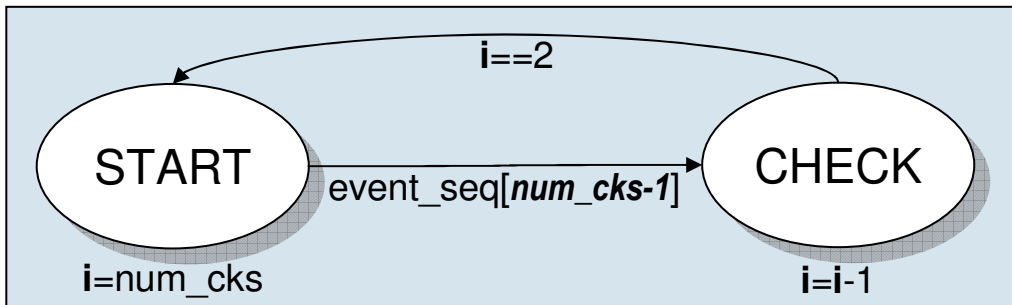
```

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.



## assert\_decrement

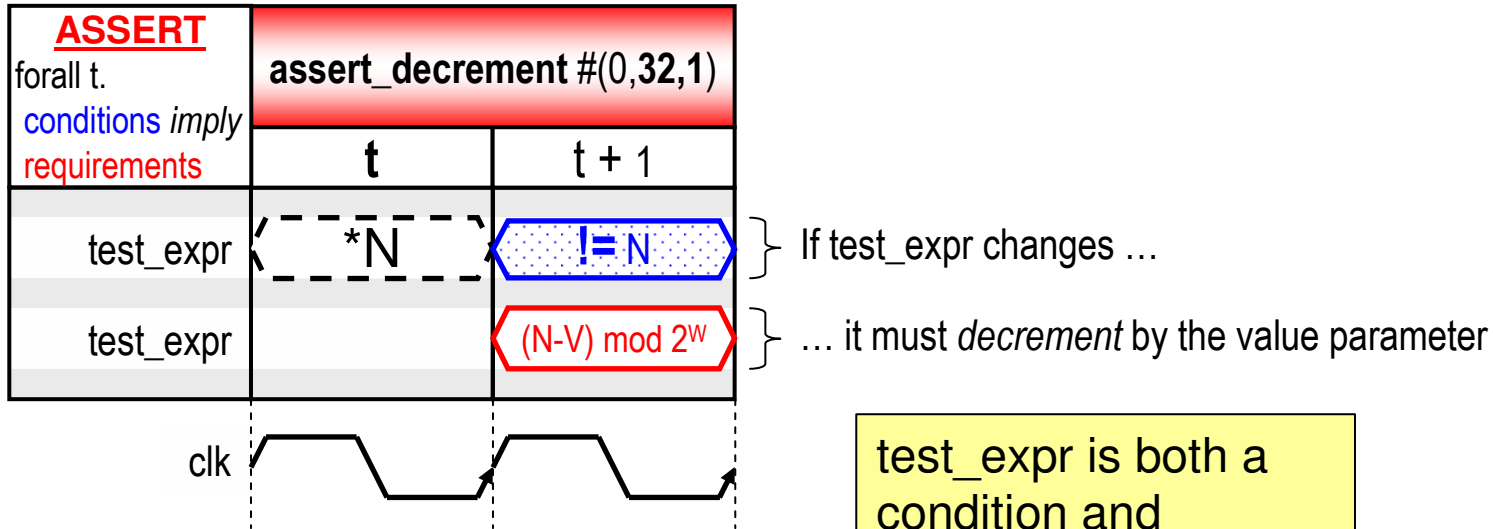
```

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

```

2-Cycles

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



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

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

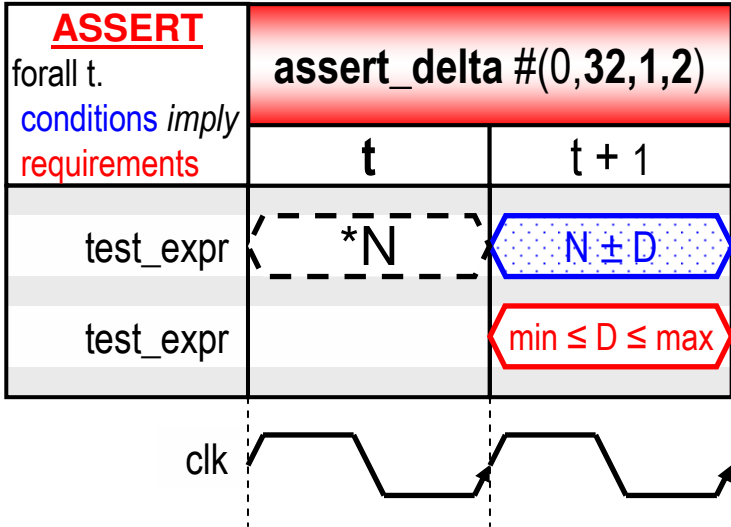


# assert\_delta

```

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

If test\_expr changes, the delta must be  $\geq \text{min}$  and  $\leq \text{max}$  2-Cycles



If test\_expr changes value by any delta D ...  
 ... then delta D must be within min/max limits

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

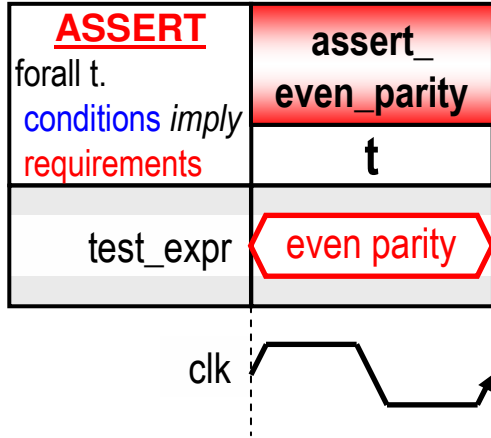


## assert\_even\_parity

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

test\_expr must have an even parity, i.e. an even number of bits asserted.

Single-Cycle

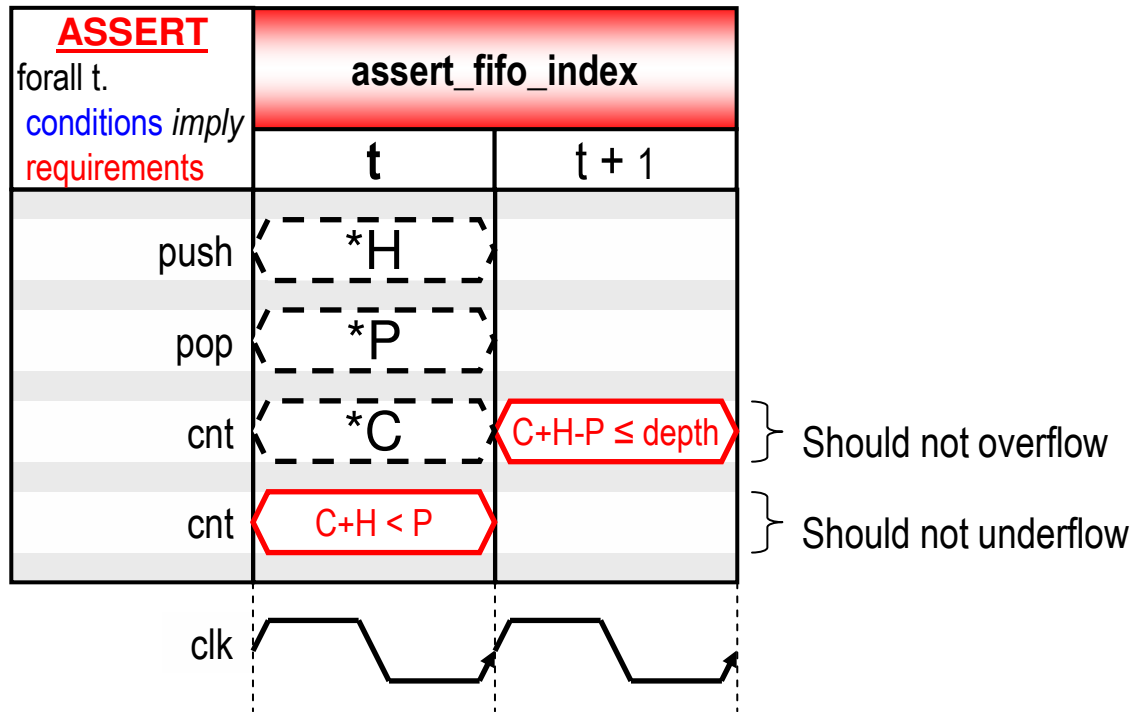


## assert\_fifo\_index

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

2-Cycles

FIFO pointers should never overflow or underflow.



### Differs to April 2003

From OVL version 1.0 the property\_type parameter does not affect the functionality.

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

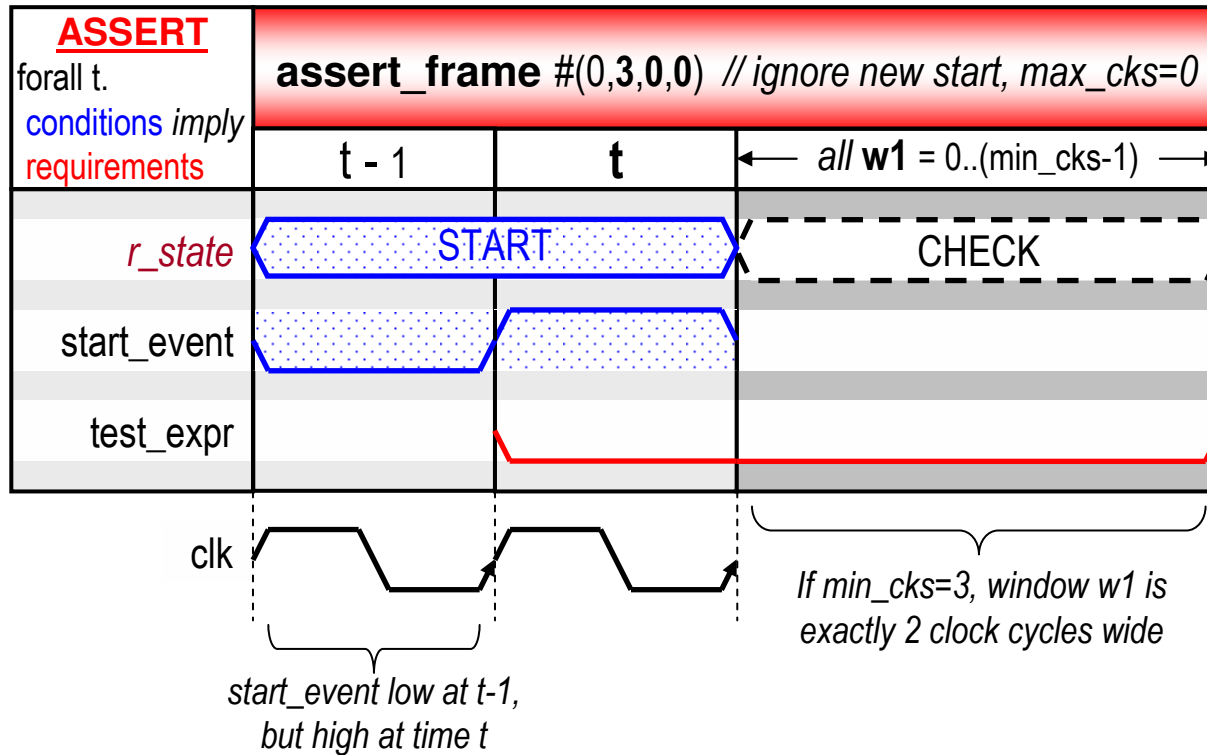
```

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

```

**n-Cycles**

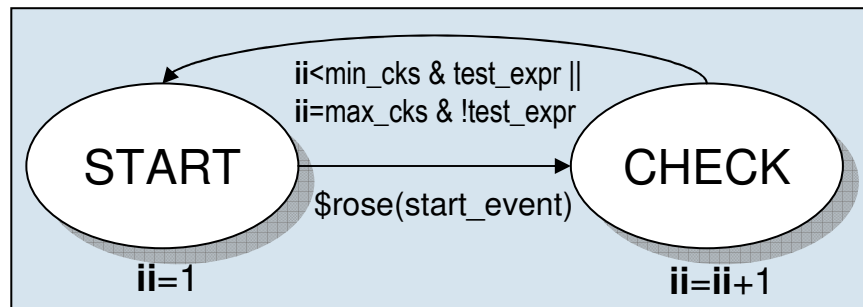
test\_expr must not hold before min\_cks cycles, but must hold at least once by max\_cks.



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.



### assert\_frame

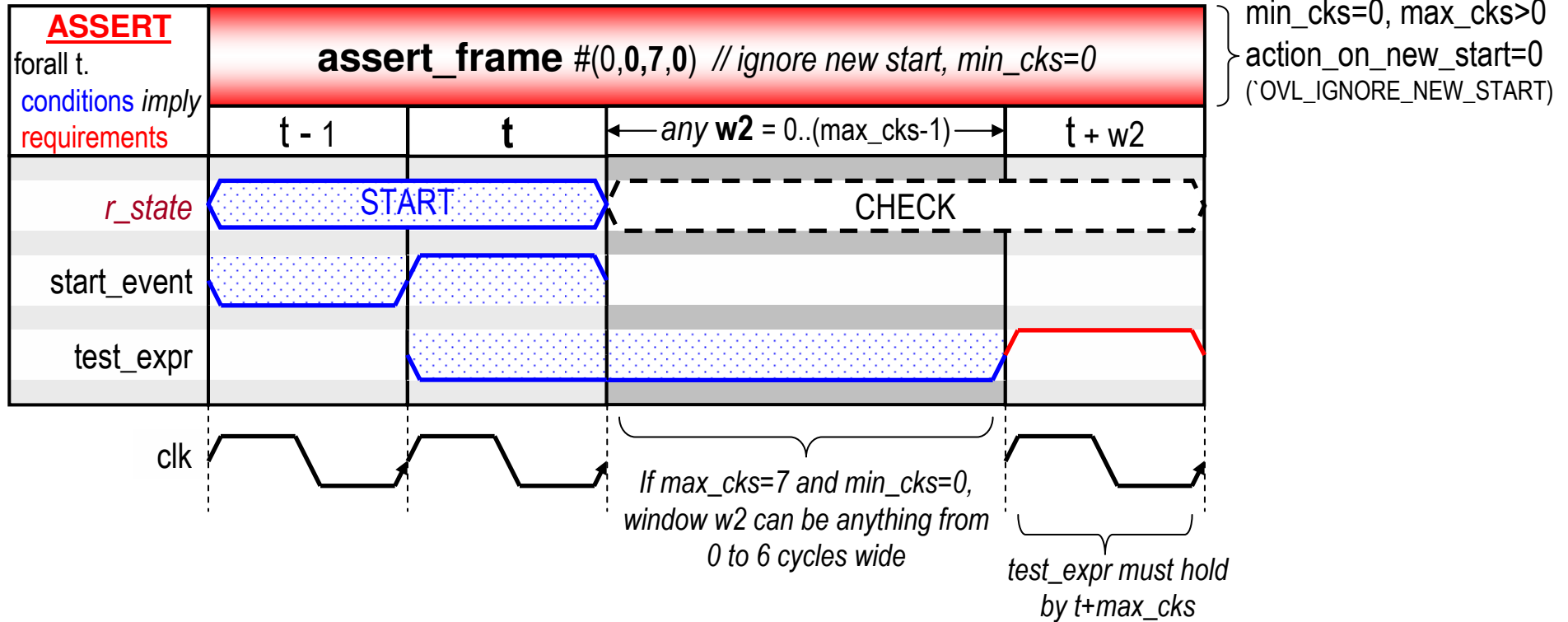
```

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

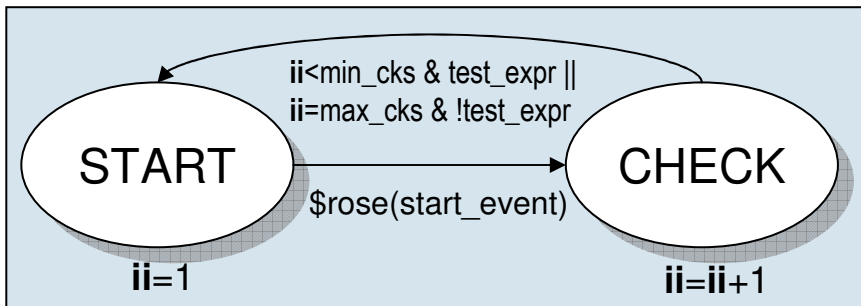
```

**n-Cycles**

test\_expr must not hold before min\_cks cycles, but must hold at least once by max\_cks.



### r\_state (auxiliary logic)



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



### assert\_frame

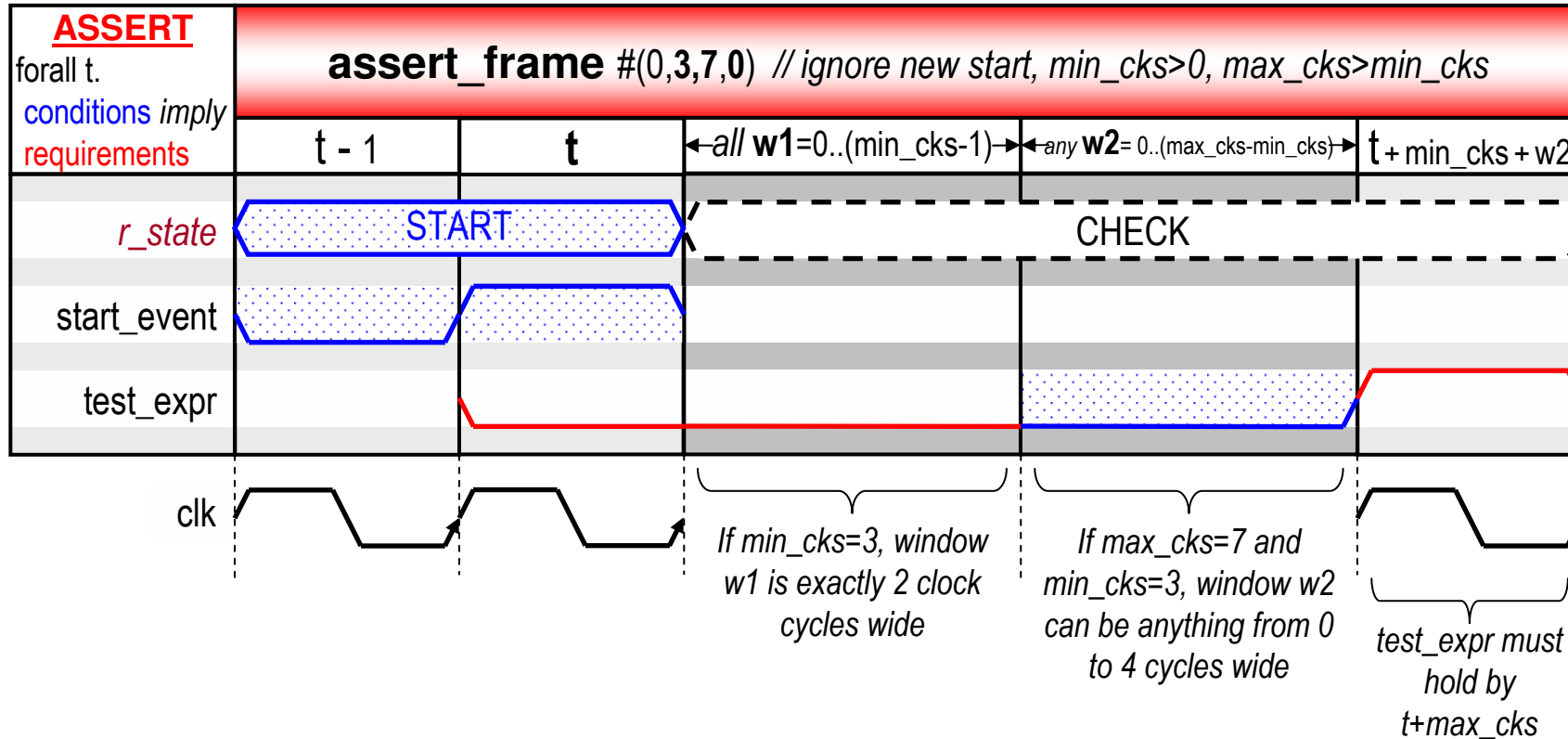
```

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

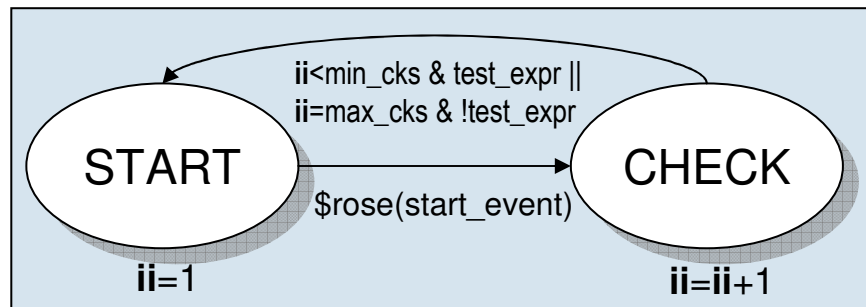
```

n-Cycles

test\_expr must not hold before min\_cks cycles, but must hold at least once by max\_cks.



### r\_state (auxiliary logic)



### assert\_frame

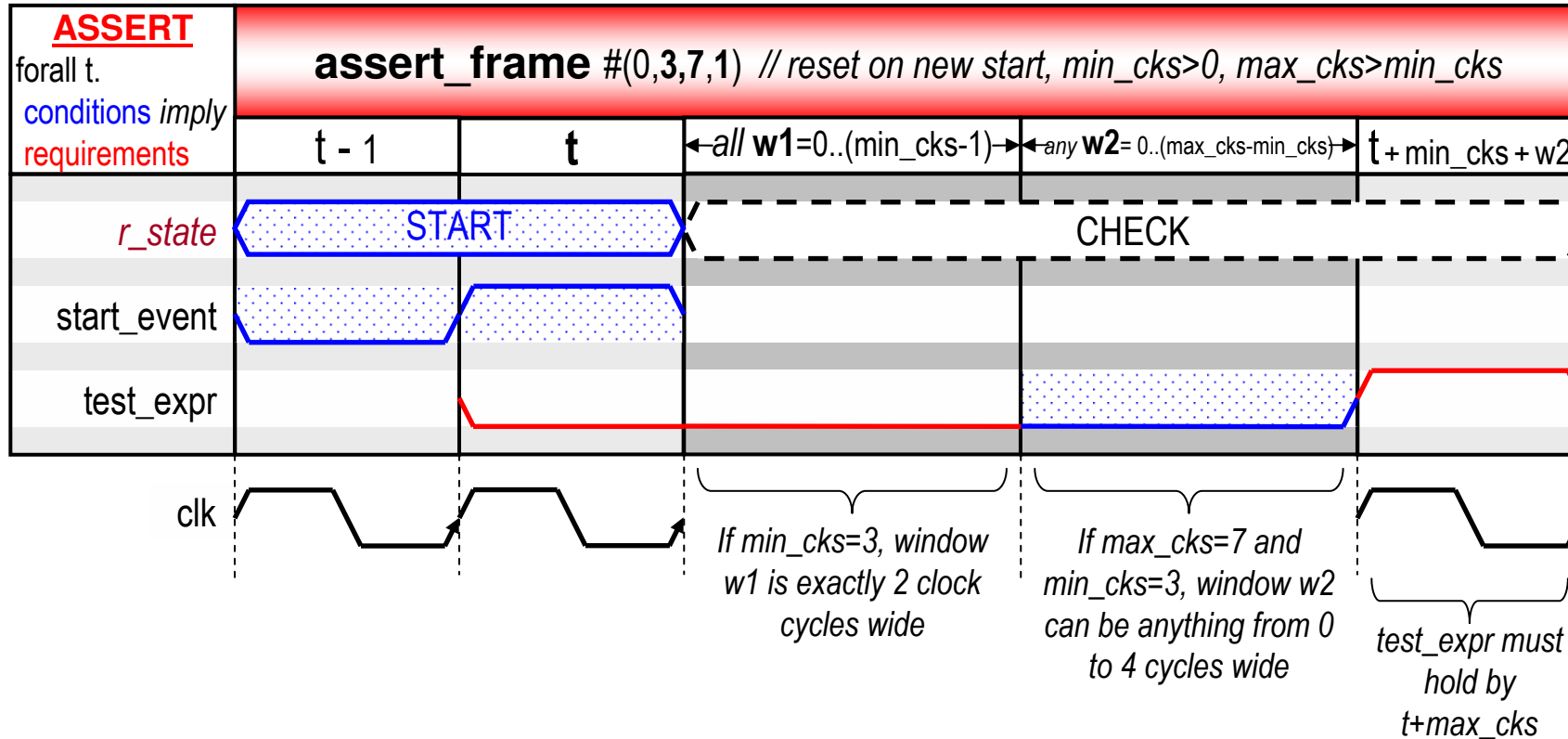
```

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

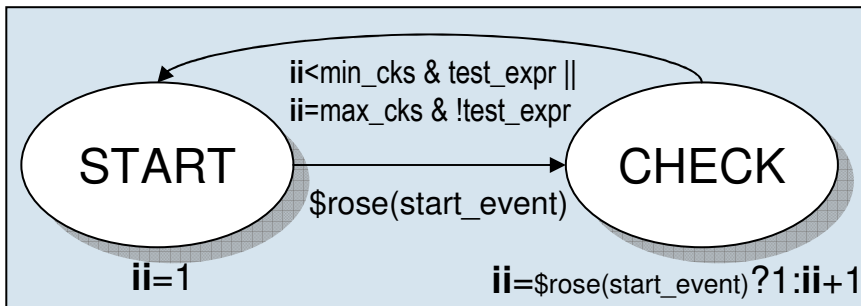
```

n-Cycles

test\_expr must not hold before min\_cks cycles, but must hold at least once by max\_cks.



### 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.

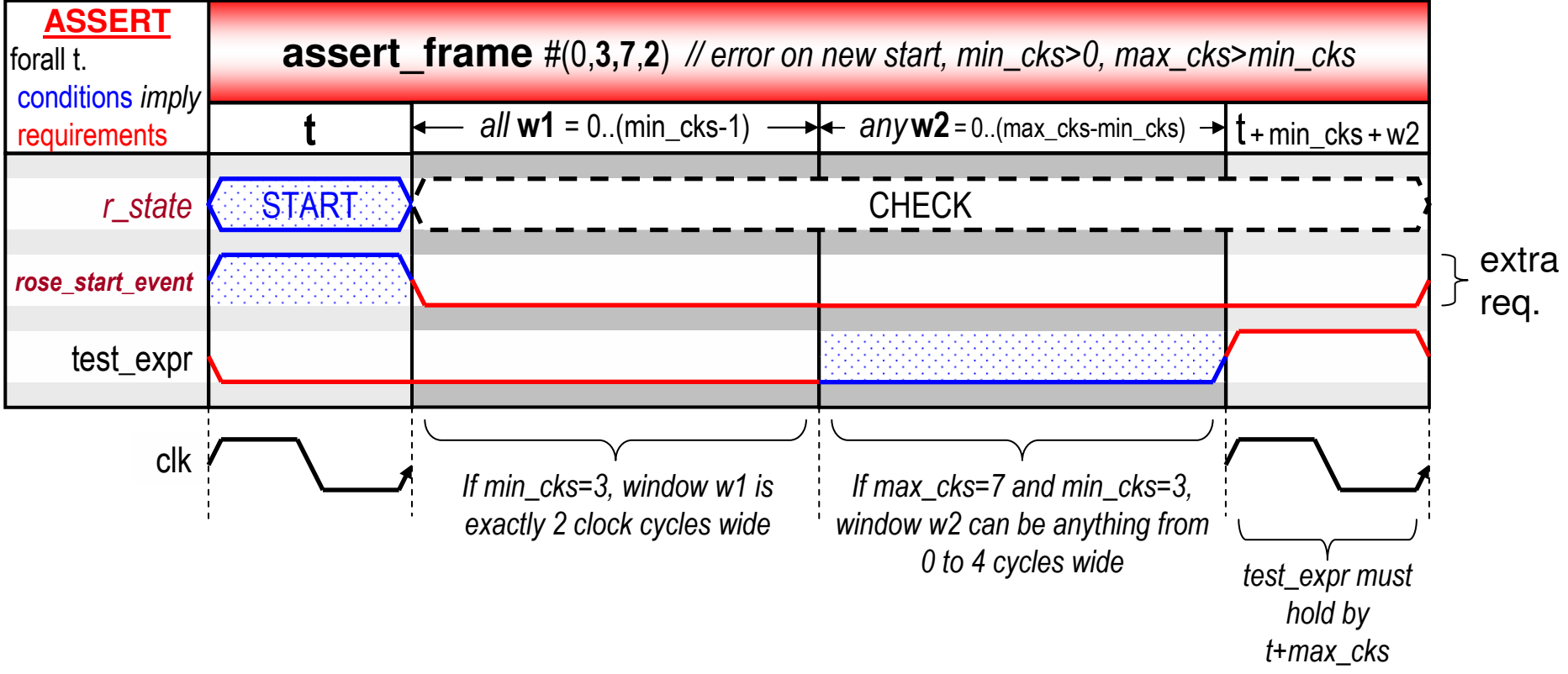


# assert\_frame

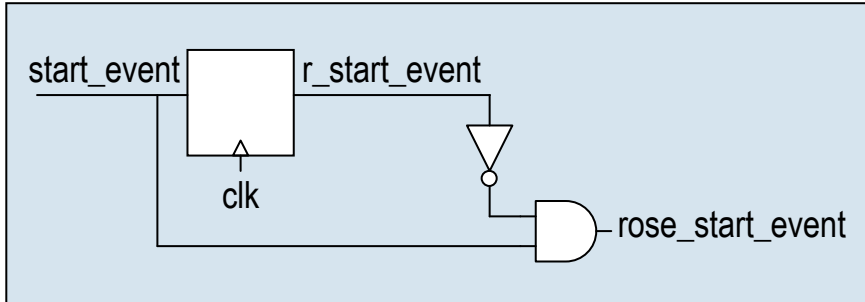
```
#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
ul (clk, reset_n, start_event, test_expr)
```

**n-Cycles**

test\_expr must not hold before min\_cks cycles, but must hold at least once by max\_cks.



## rose\_start\_event (auxiliary logic)



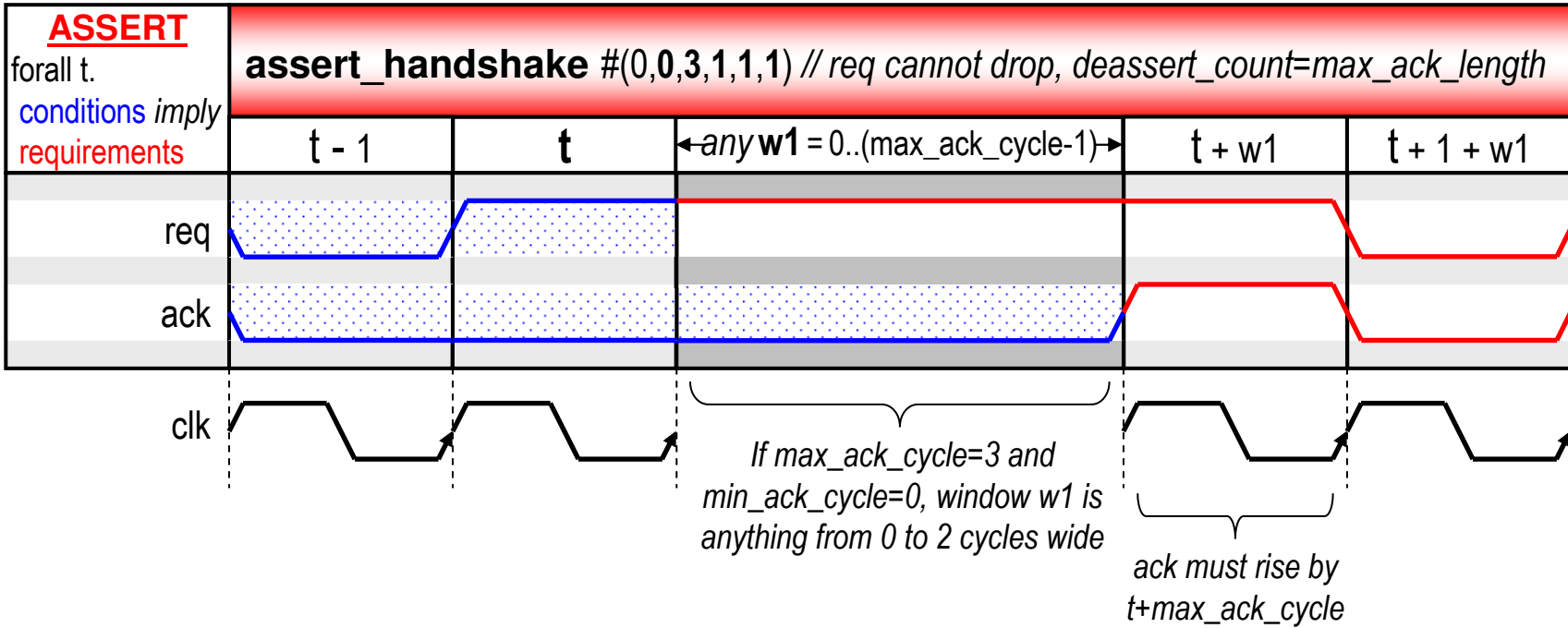
“error on new start” has an additional requirement from `t+1` (no new rising edge on `start_event`).



# assert\_handshake

```
#(severity_level, min_ack_cycle, max_ack_cycle, req_drop, deassert_count, max_ack_length,
property_type, msg, coverage_level) ul (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).

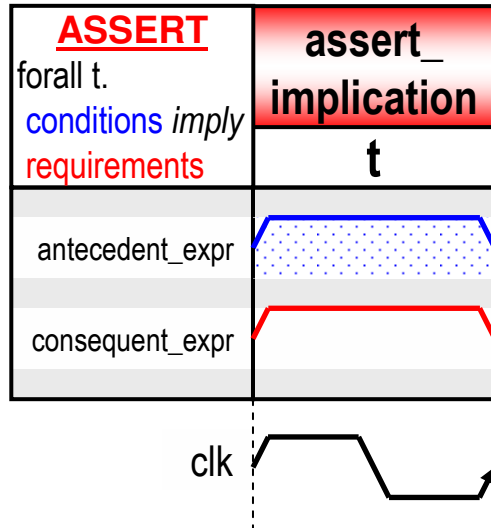


## assert\_implication

```
 #(severity_level, property_type, msg, coverage_level)  
 ul (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.

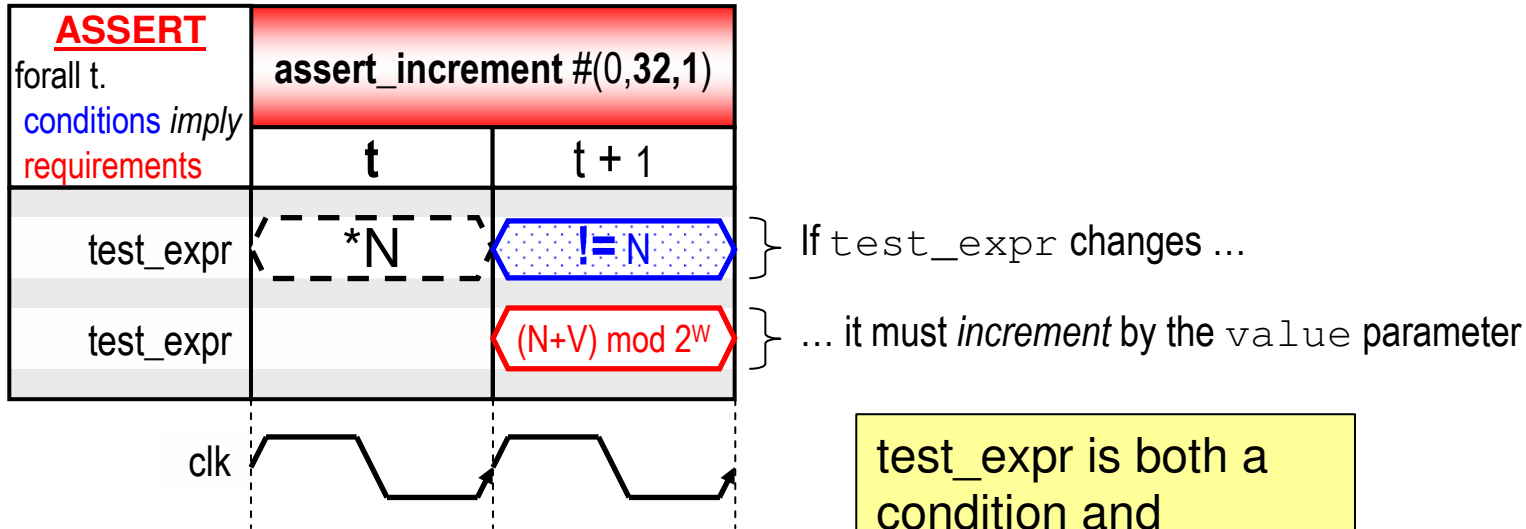


# assert\_increment

```

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

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



If `test_expr` changes ...  
 ... it must *increment* by the `value` parameter

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

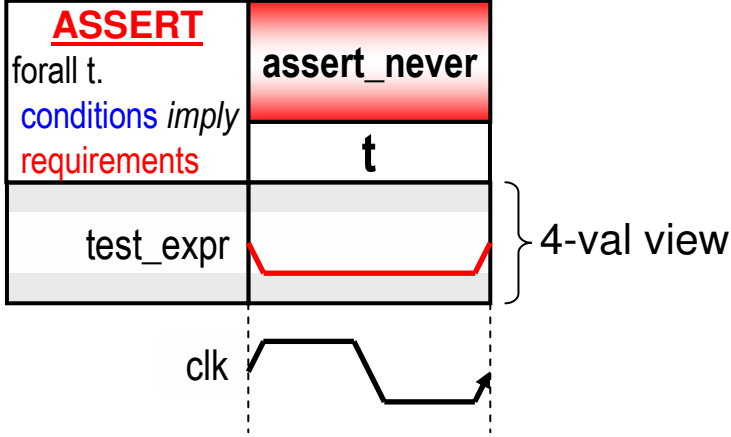
`test_expr` is both a condition and requirement at `t+1`. Hence it appears on two rows.



## assert\_never

```
 #(severity_level, property_type, msg, coverage_level)  
 ul (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:

global: OVL\_XCHECK\_OFF  
local: OVL\_ASSERT\_2STATE

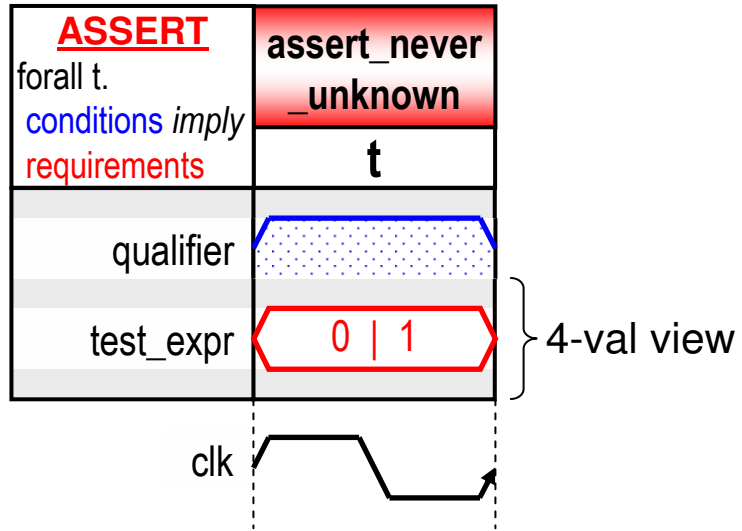


## assert\_never\_unknown

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

test\_expr must never be at an unknown value, just boolean 0 or 1.

Single-Cycle



This is an explicit X-checking assertion

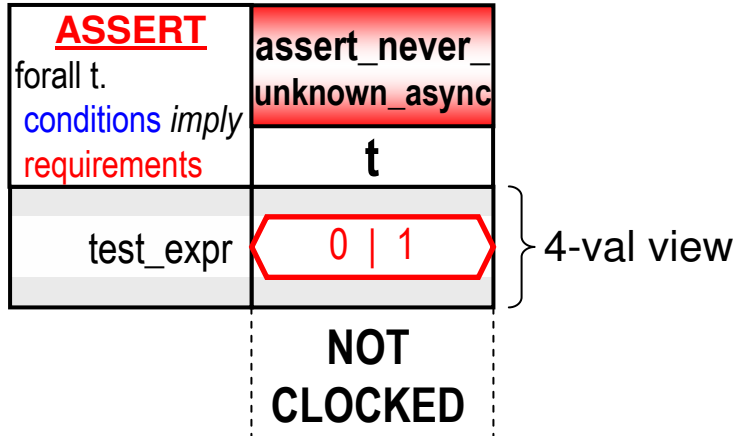


## assert\_never\_unknown\_async

```
 #(severity_level, width, property_type, msg, coverage_level)  
 ul (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.

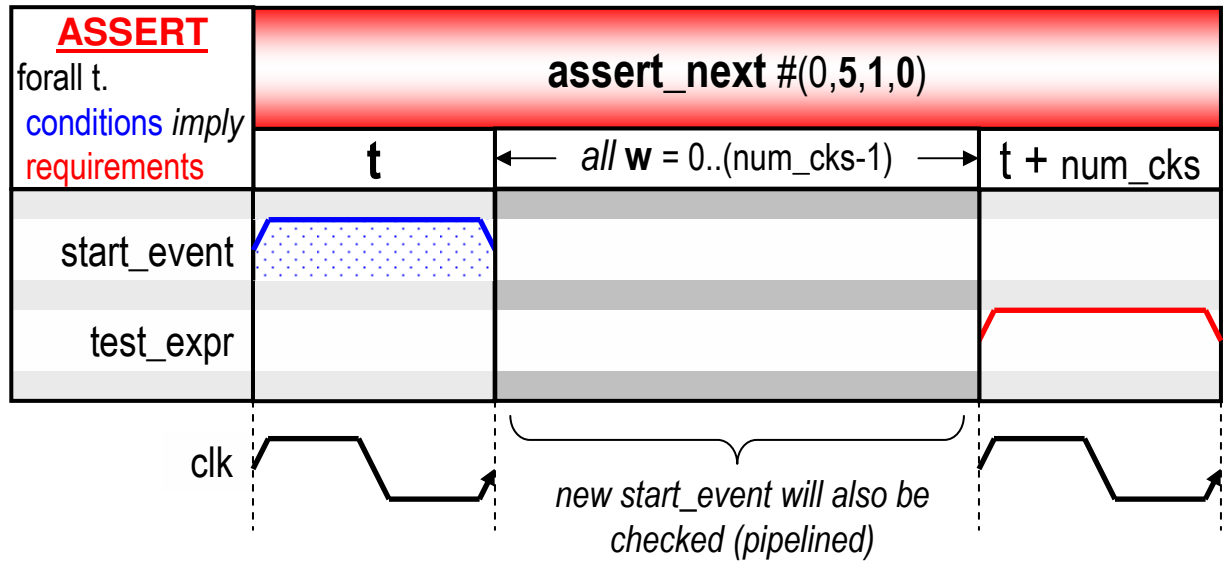
Does not have a *qualifier* input (unlike the synchronous version)



# assert\_next

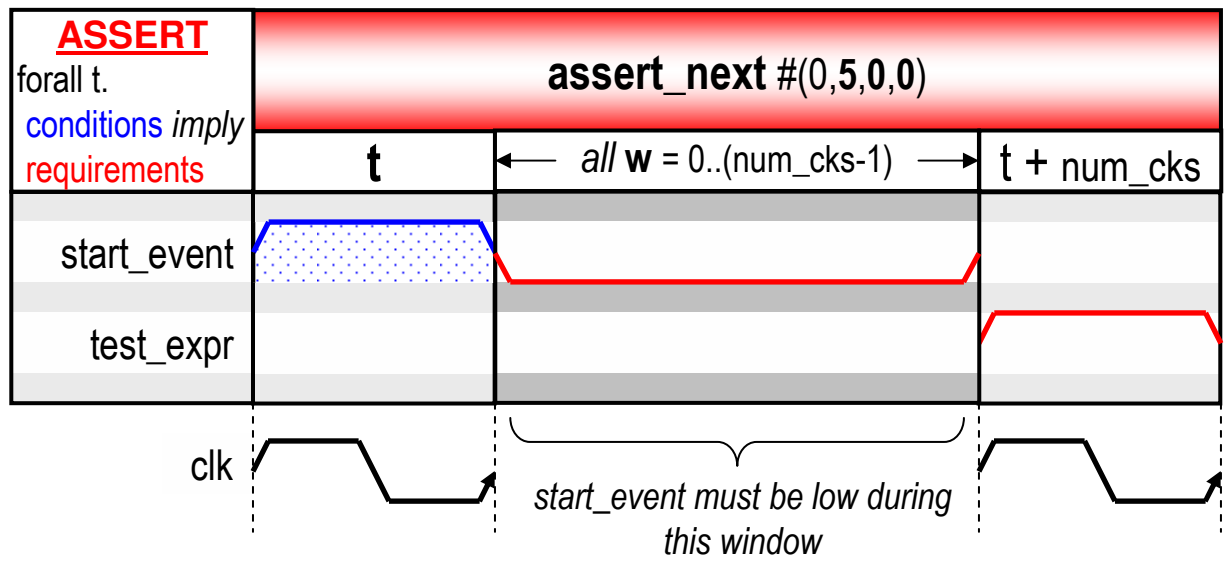
```
#(severity_level, num_cks, check_overlapping, check_missing_start, property_type, msg, coverage_level)
ul (clk, reset_n, start_event, test_expr)
```

test\_expr must hold num\_cks cycles after start\_event holds N Cycles



check\_overlapping=1  
check\_missing\_start=0

check\_overlapping=1 is a pipelined check, e.g. new start\_event@t+1 checks test\_expr@t+1+num\_cks



check\_overlapping=0  
check\_missing\_start=0

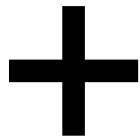
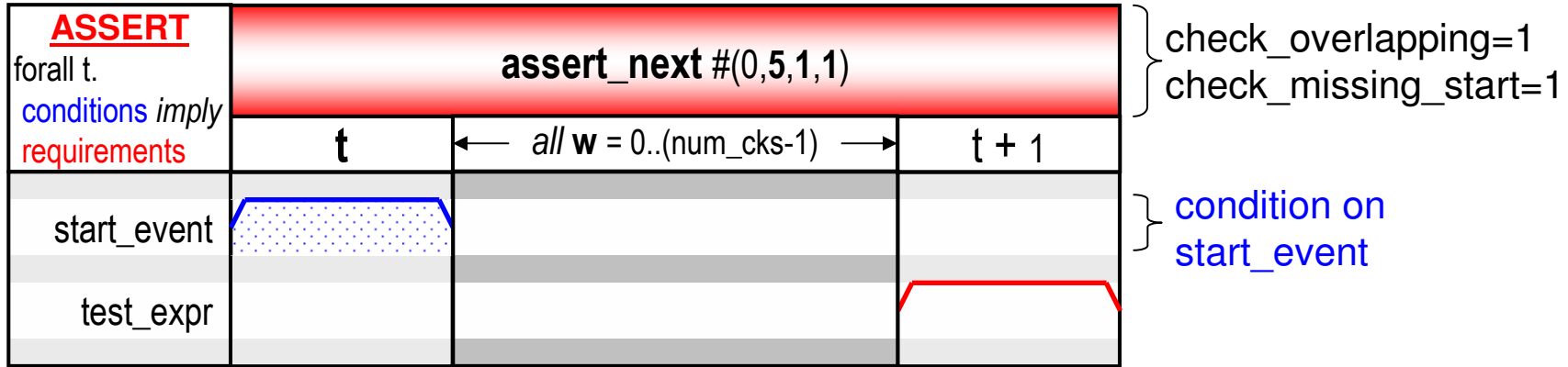
check\_overlapping=0 only allows start\_event every num\_cks cycles. When num\_cks=1, behaviour is same as default config



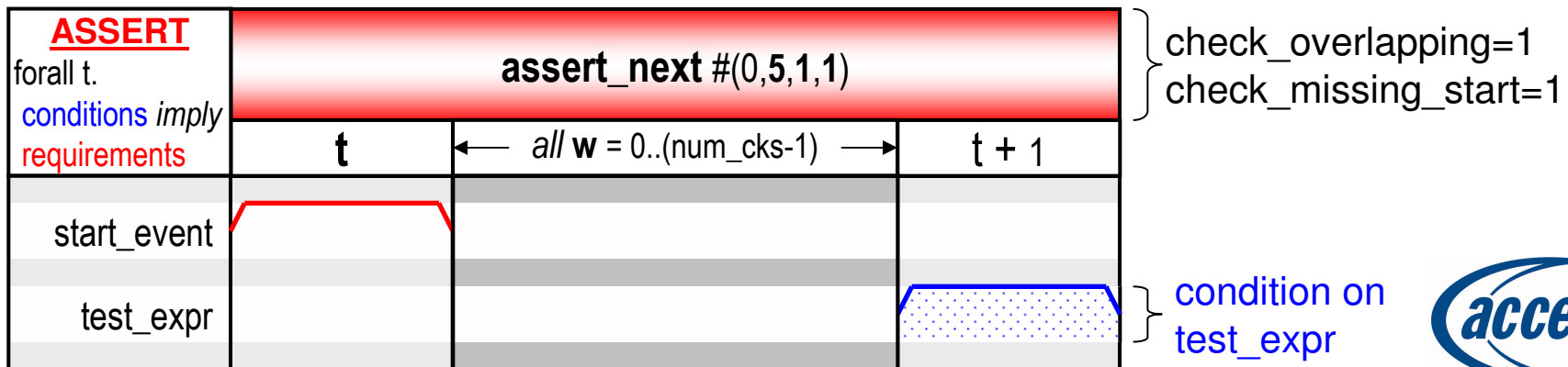
### assert\_next

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

test\_expr must hold num\_cks cycles after start\_event holds N Cycles



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

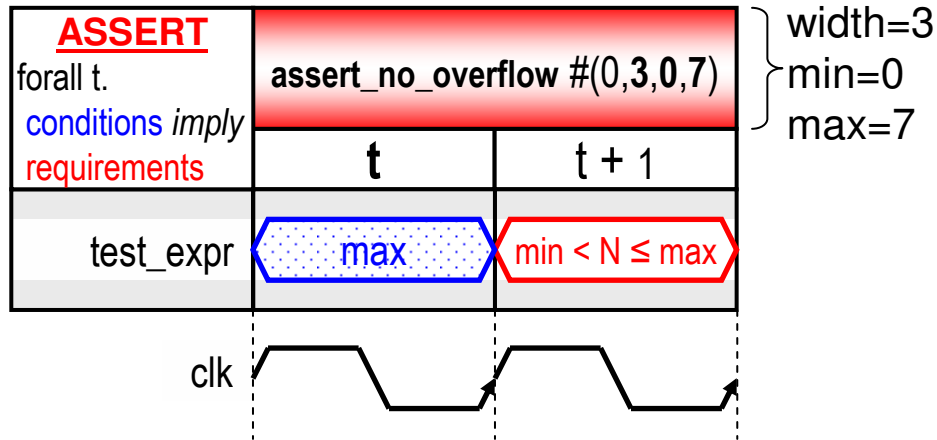


## assert\_no\_overflow

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

2-Cycles

If test\_expr is at max, in the next cycle test\_expr must be > min and ≤ max



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.

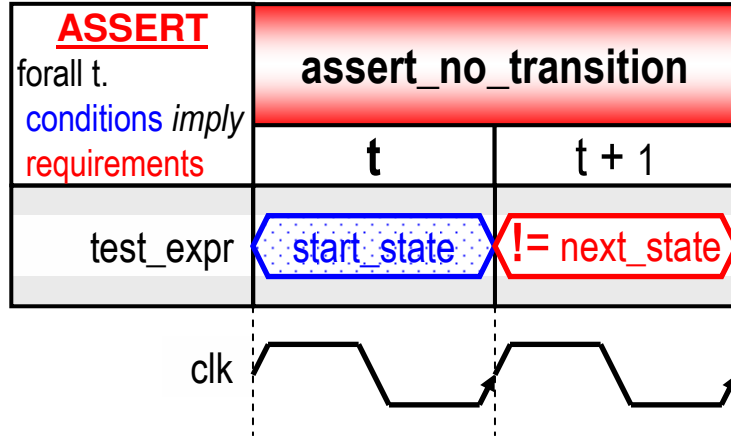


## assert\_no\_transition

```
 #(severity_level, width, property_type, msg, coverage_level)  
 ul (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

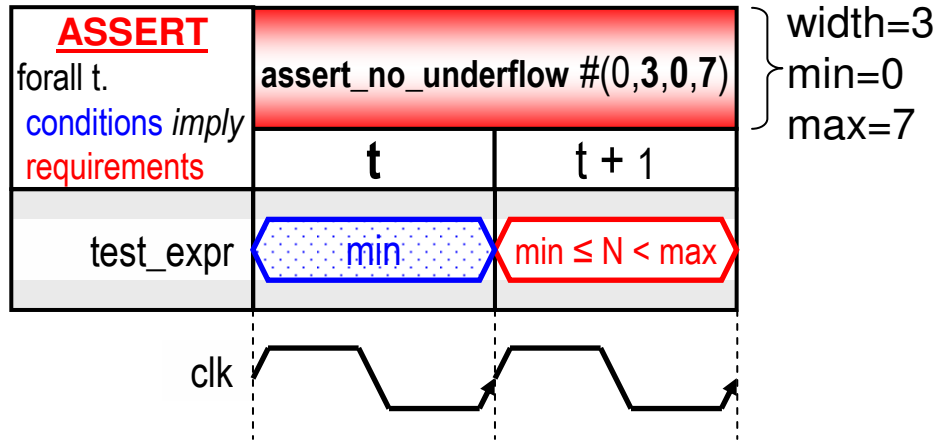


## assert\_no\_underflow

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

2-Cycles

If test\_expr is at min, in the next cycle test\_expr must be  $\geq$  min and  $<$  max



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

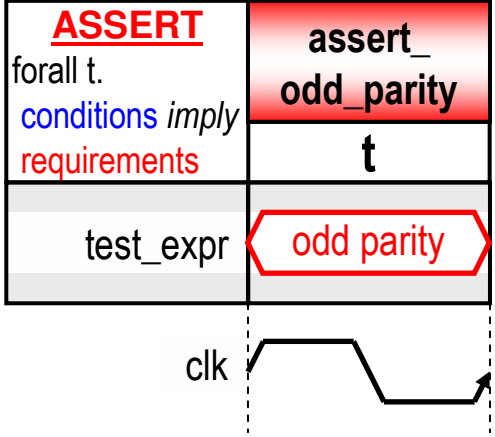
The min and max values do not need to span the full range of test\_expr.



## assert\_odd\_parity

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

test\_expr must have an odd parity, i.e. an odd number of bits asserted. Single-Cycle

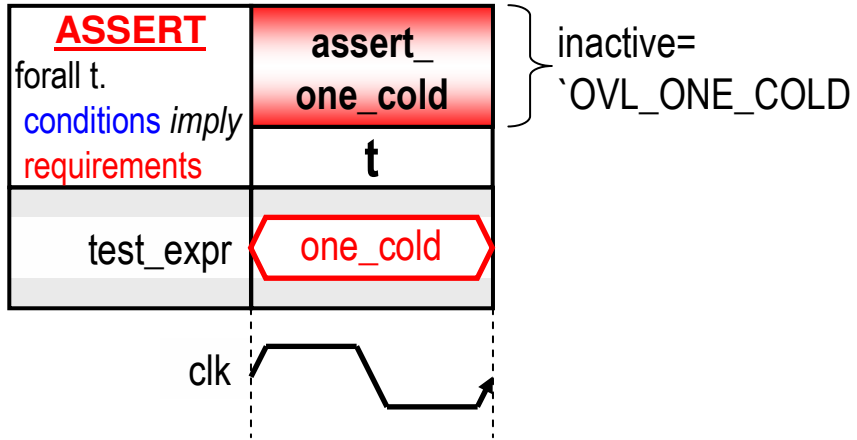


## assert\_one\_cold

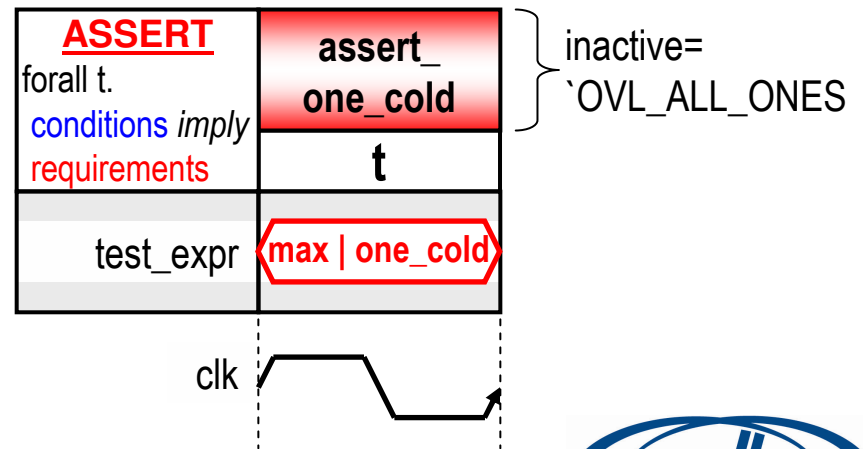
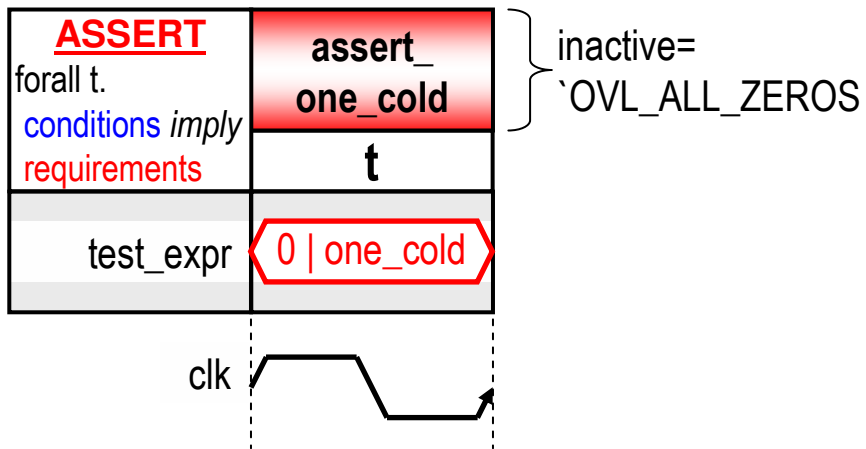
```

#(severity_level, width, inactive, property_type, msg, coverage_level)
ul (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.

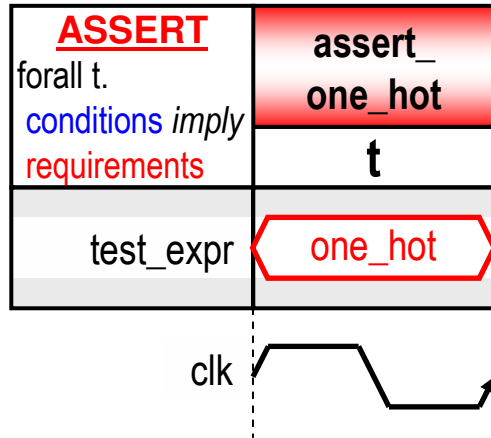


## assert\_one\_hot

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

test\_expr must be one-hot, i.e. exactly one bit set high

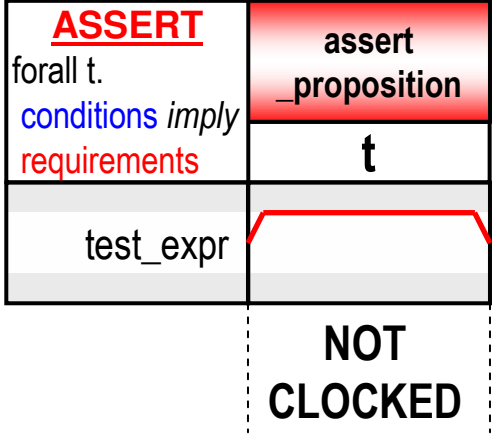
Single-Cycle



## assert\_proposition

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

test\_expr must hold asynchronously (not just at a clock edge) Combinatorial



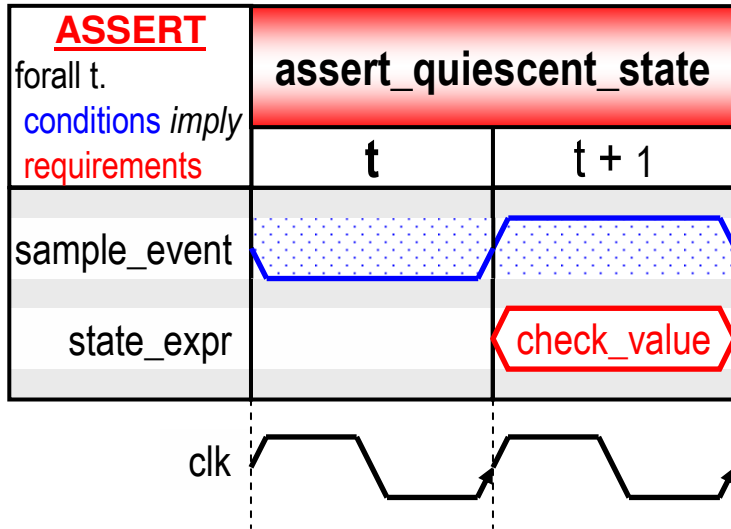
This is an asynchronous version of the clocked assert\_always



## assert\_quiescent\_state

```
#(severity_level, width, property_type, msg, coverage_level)
ul (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.

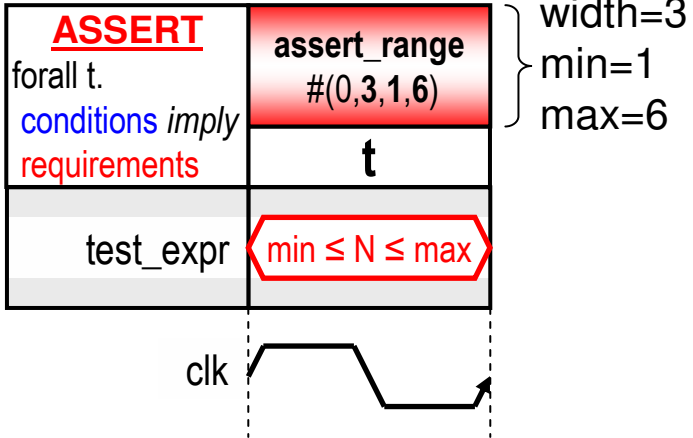
Can *just* trigger at end of simulation by setting sample\_event to 1'b0 and defining:  
`OVL\_END\_OF\_SIMULATION



## assert\_range

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

test\_expr must be  $\geq$  min and  $\leq$  max Single-Cycle

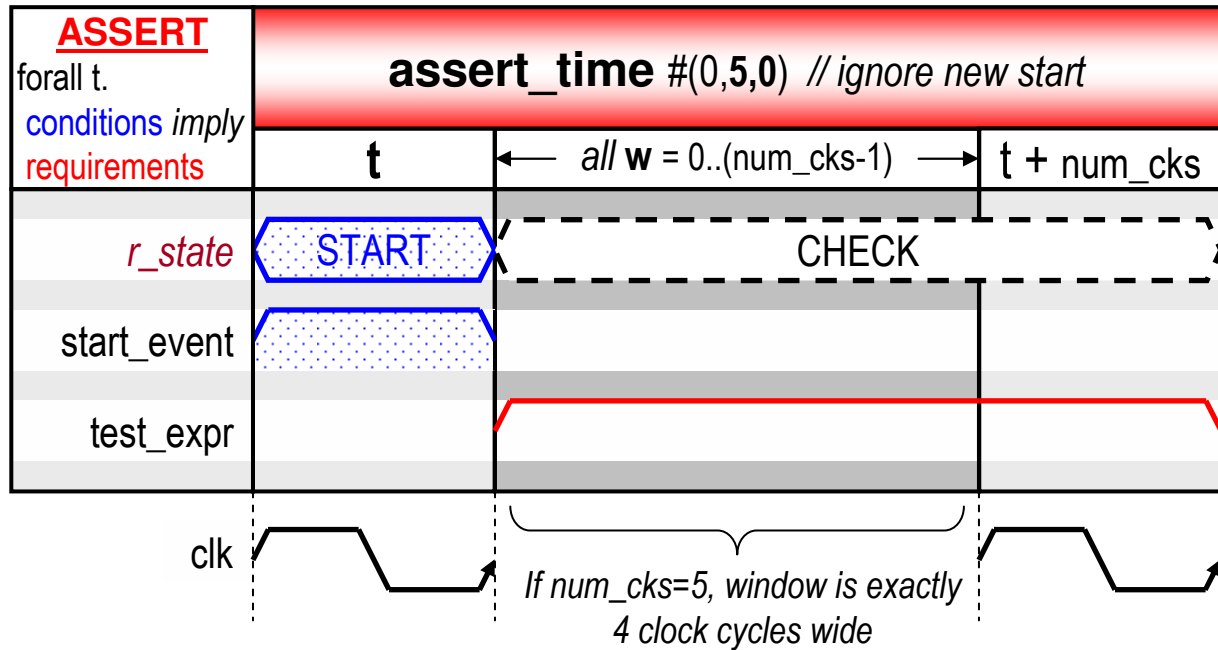


# assert\_time

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

**n-Cycles**

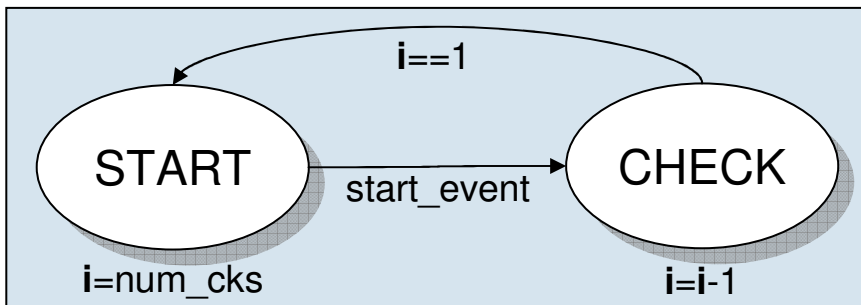
test\_expr must hold for num\_cks cycles after start\_event



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)



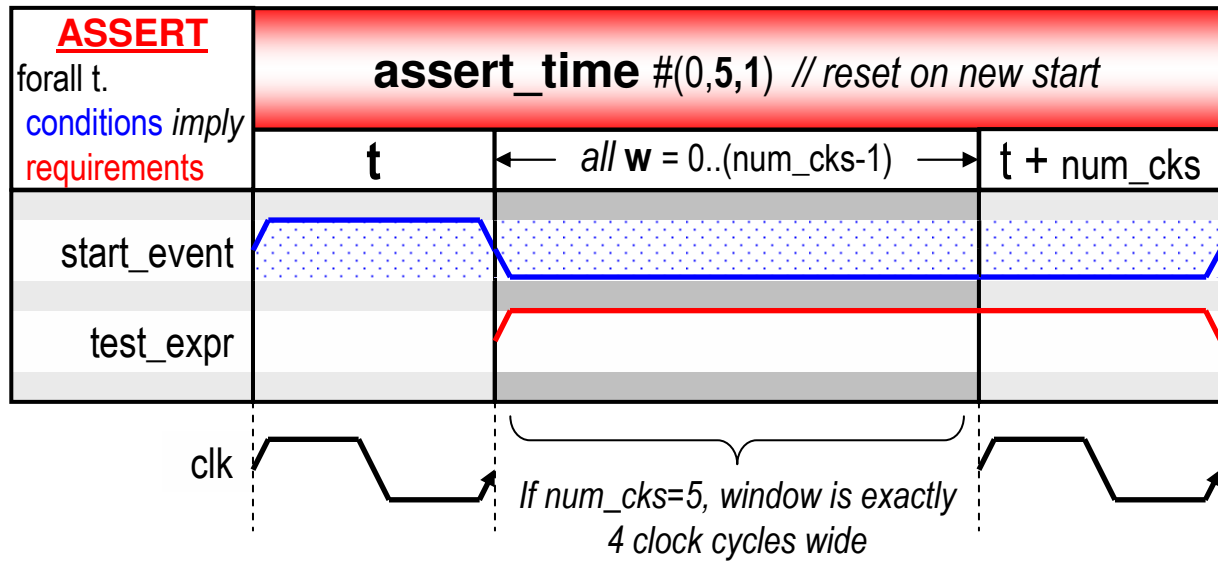
Auxiliary logic necessary, to ignore new start. Checking only begins after start\_event is true and r\_state==START.



# assert\_time

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

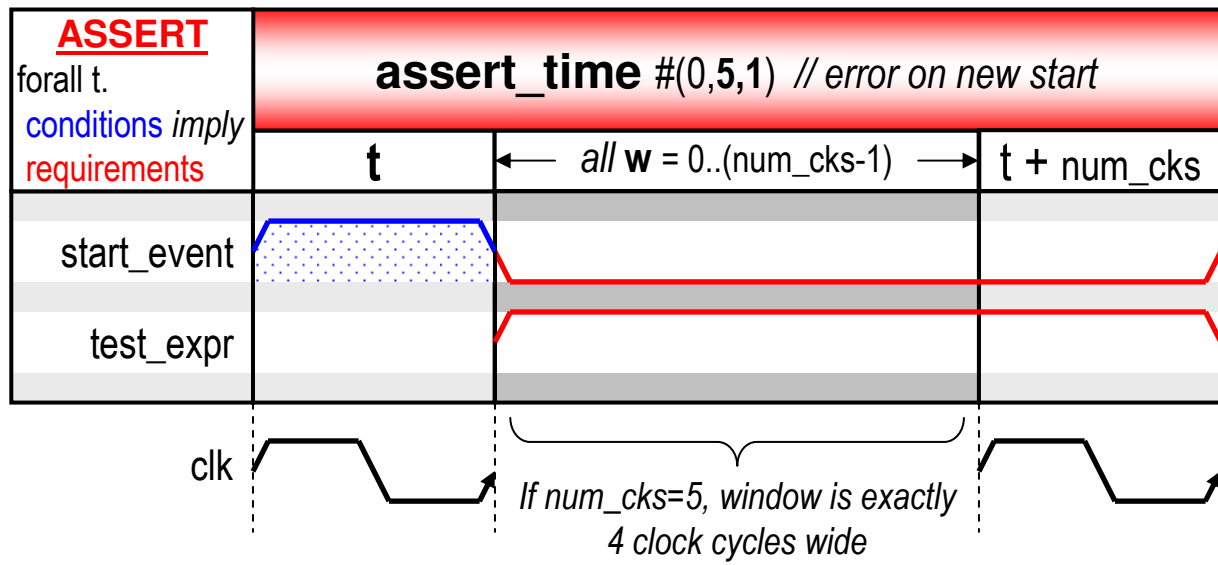
test\_expr must hold for num\_cks cycles after start\_event n-Cycles



num\_cks=5  
action\_on\_new\_start=1  
(OVL\_RESET\_ON\_NEW\_START)

extra condition on start\_event

**Differs to April 2003**  
From OVL version 1.0, RESET\_ON\_NEW\_START does not fire if test\_expr changes with new start



num\_cks=5  
action\_on\_new\_start=2  
(OVL\_ERROR\_ON\_NEW\_START)

requirement on start\_event



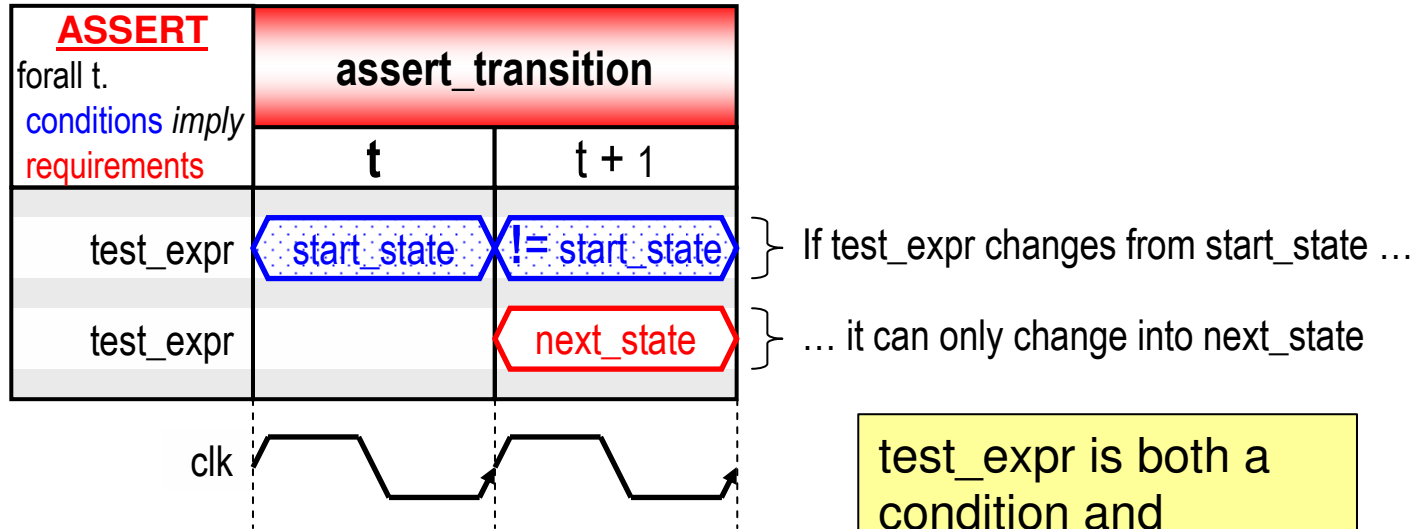
## assert\_transition

```

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

2-Cycles

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



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.

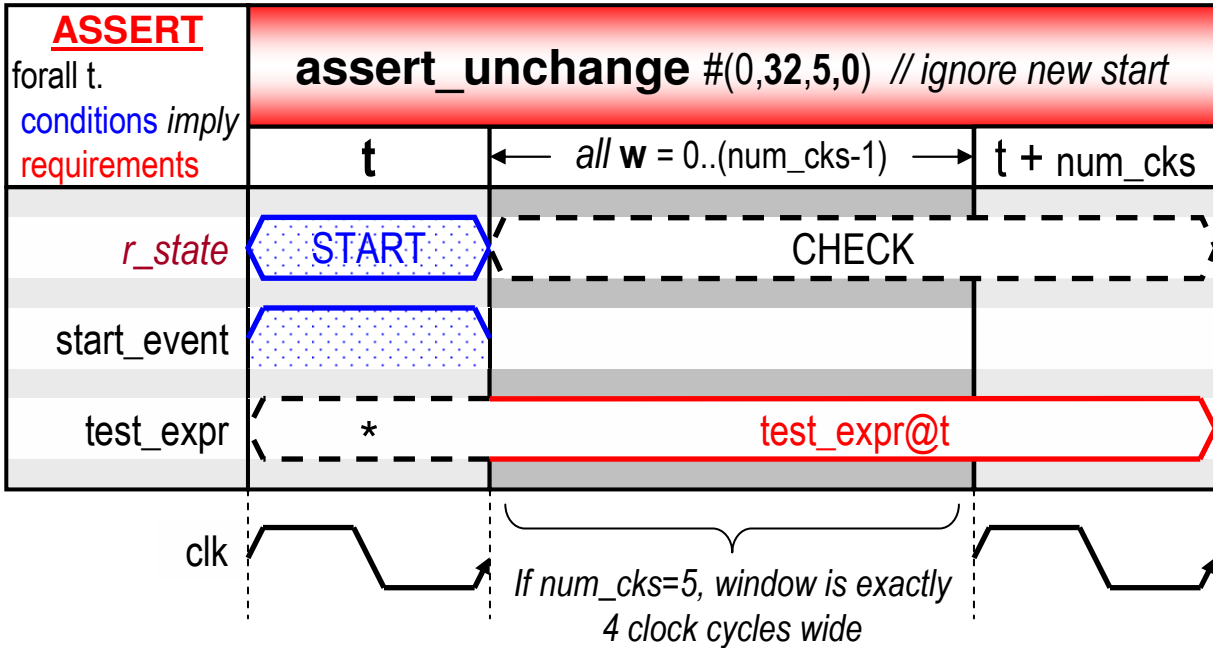


# assert\_unchange

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

**n-Cycles**

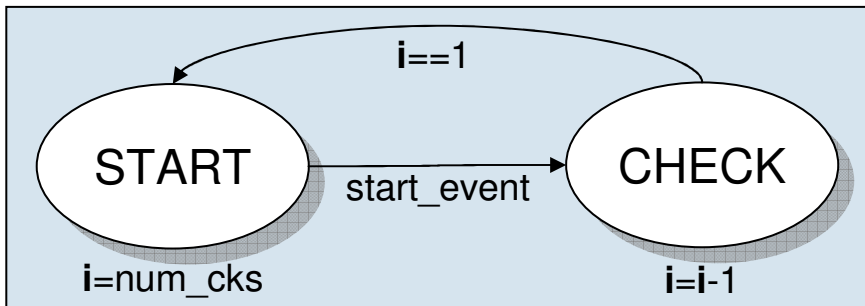
test\_expr must not change within num\_cks cycles of start\_event



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.

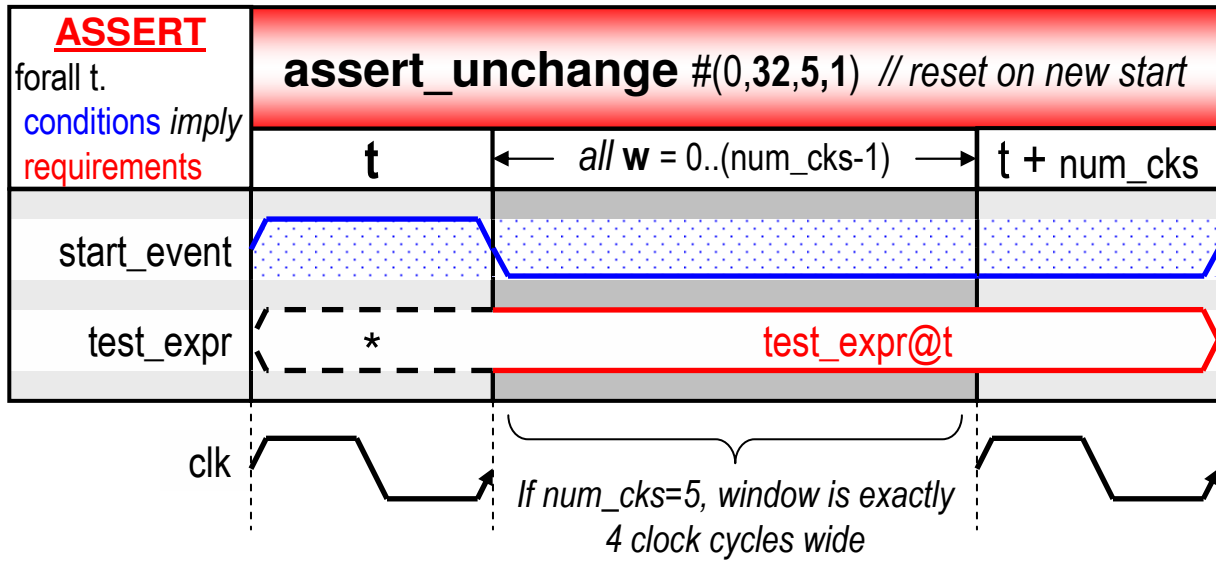


# assert\_unchange

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

**n-Cycles**

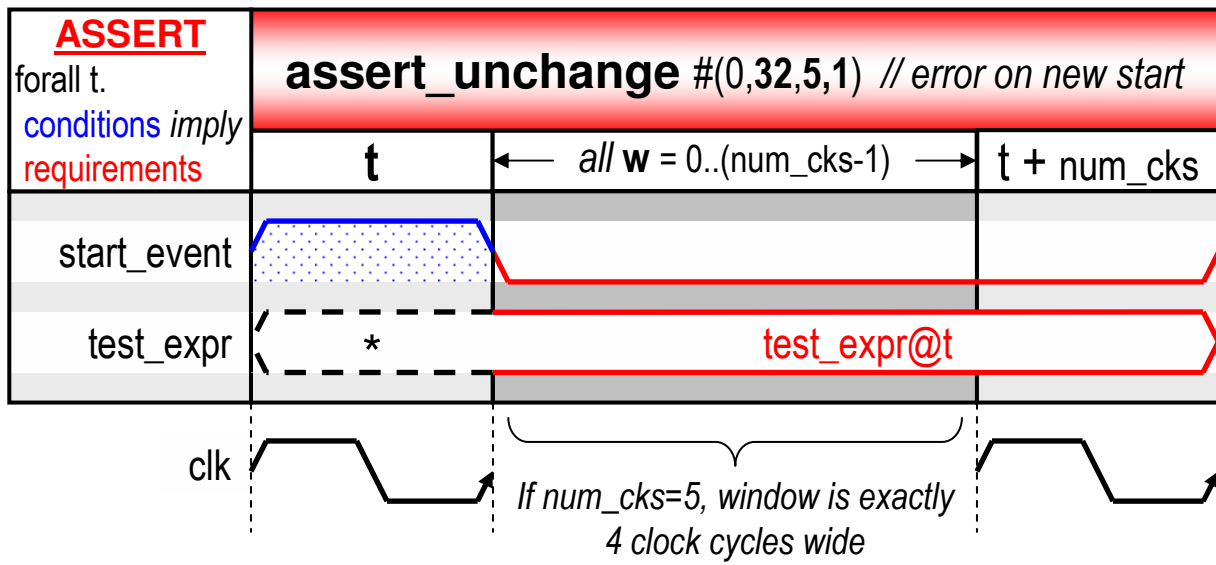
test\_expr must not change within num\_cks cycles of start\_event



num\_cks=5  
action\_on\_new\_start=1  
(`OVL\_RESET\_ON\_NEW\_START)

extra condition on start\_event

**Differs to April 2003**  
From OVL version 1.0, RESET\_ON\_NEW\_START does not fire if test\_expr changes with new start



num\_cks=5  
action\_on\_new\_start=2  
(`OVL\_ERROR\_ON\_NEW\_START)

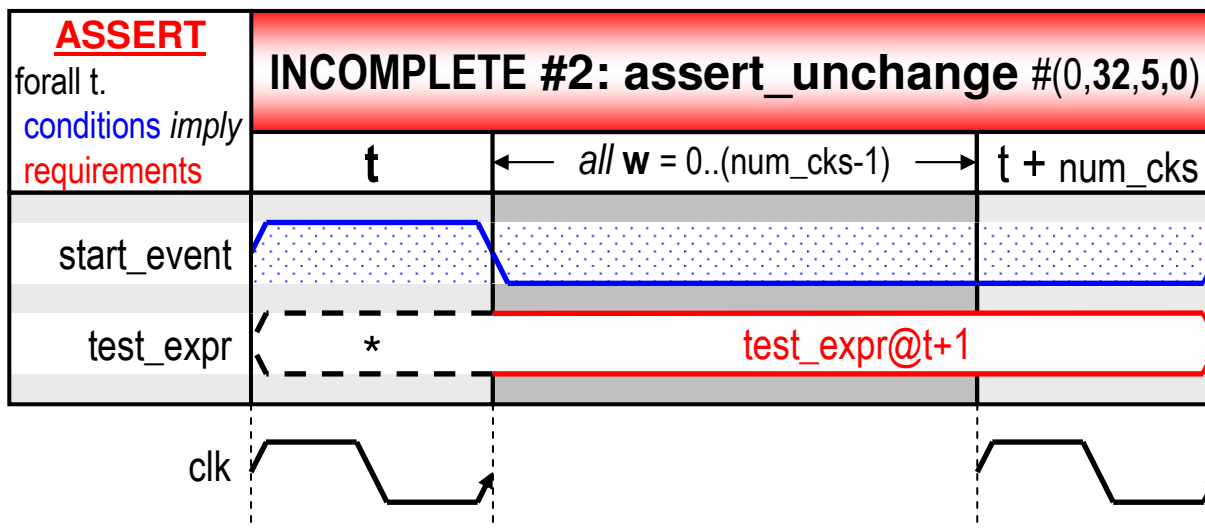
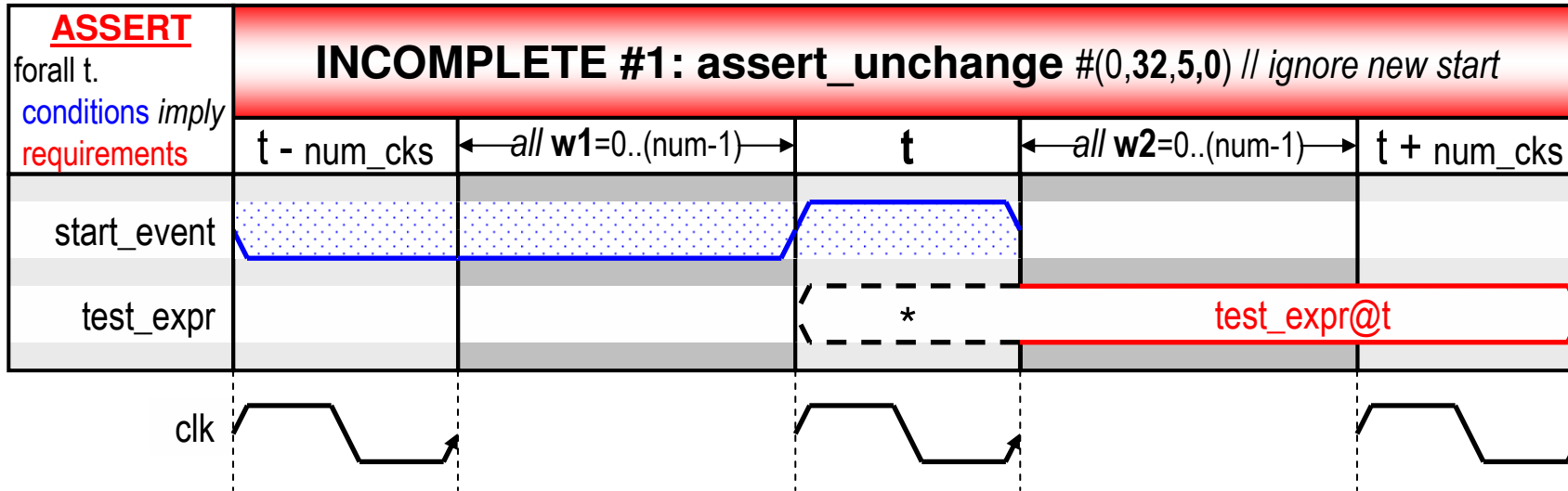
requirement on start\_event



# assert\_unchange

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

test\_expr must not change within num\_cks cycles of start\_event n-Cycles



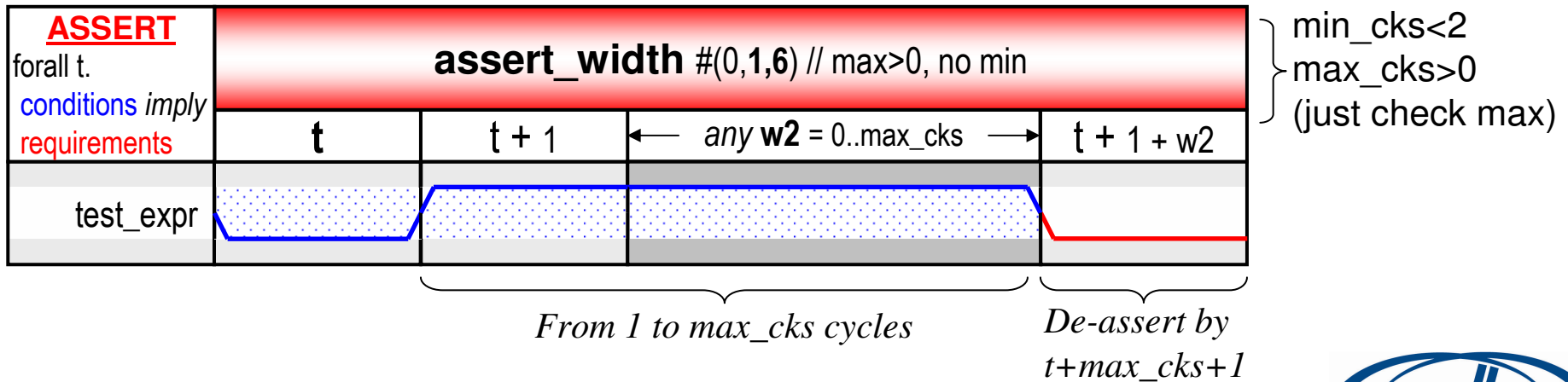
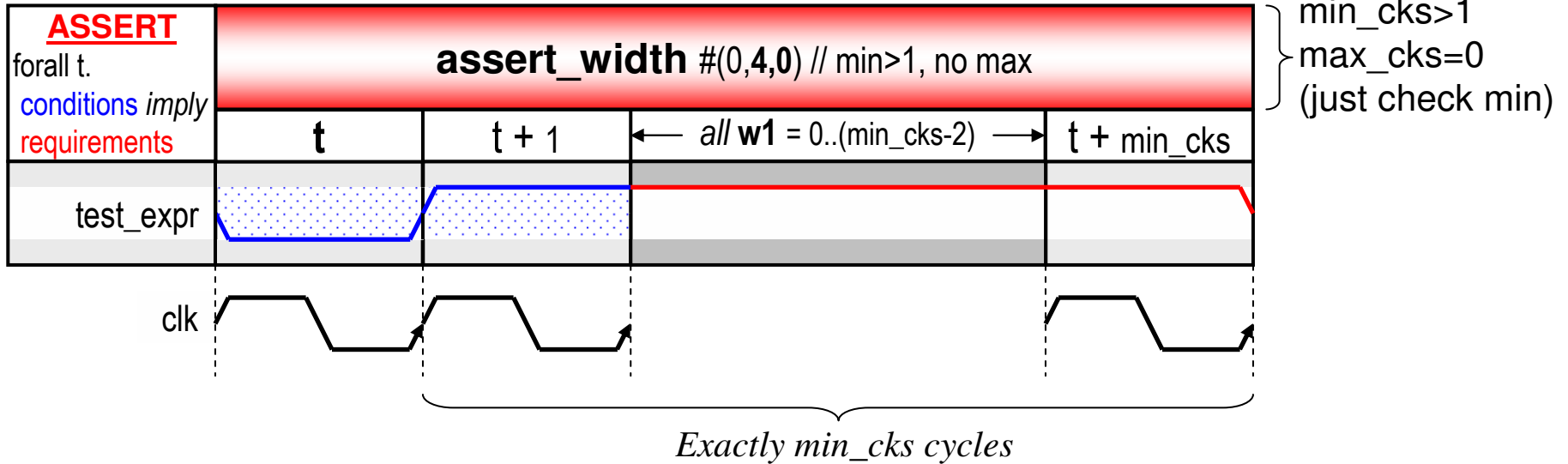
Both timing diagrams are incomplete for "ignore new start", as start\_event=0 will mask some errors!



### assert\_width

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

test\_expr must hold for between min\_cks and max\_cks cycles n-Cycles

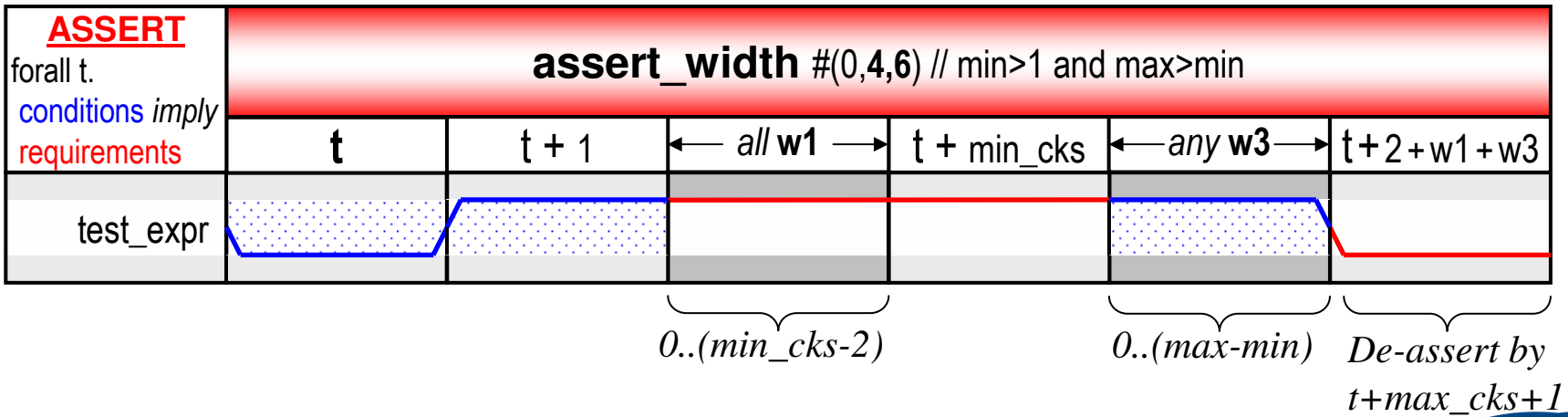
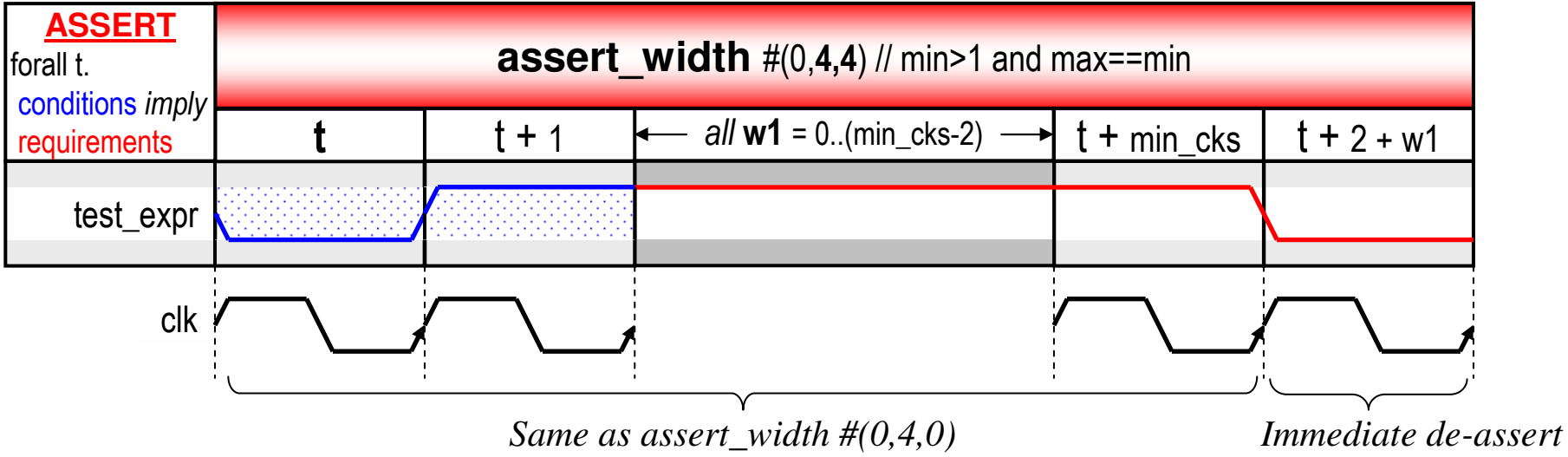


# assert\_width

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

n-Cycles

test\_expr must hold for between min\_cks and max\_cks cycles



## assert\_win\_change

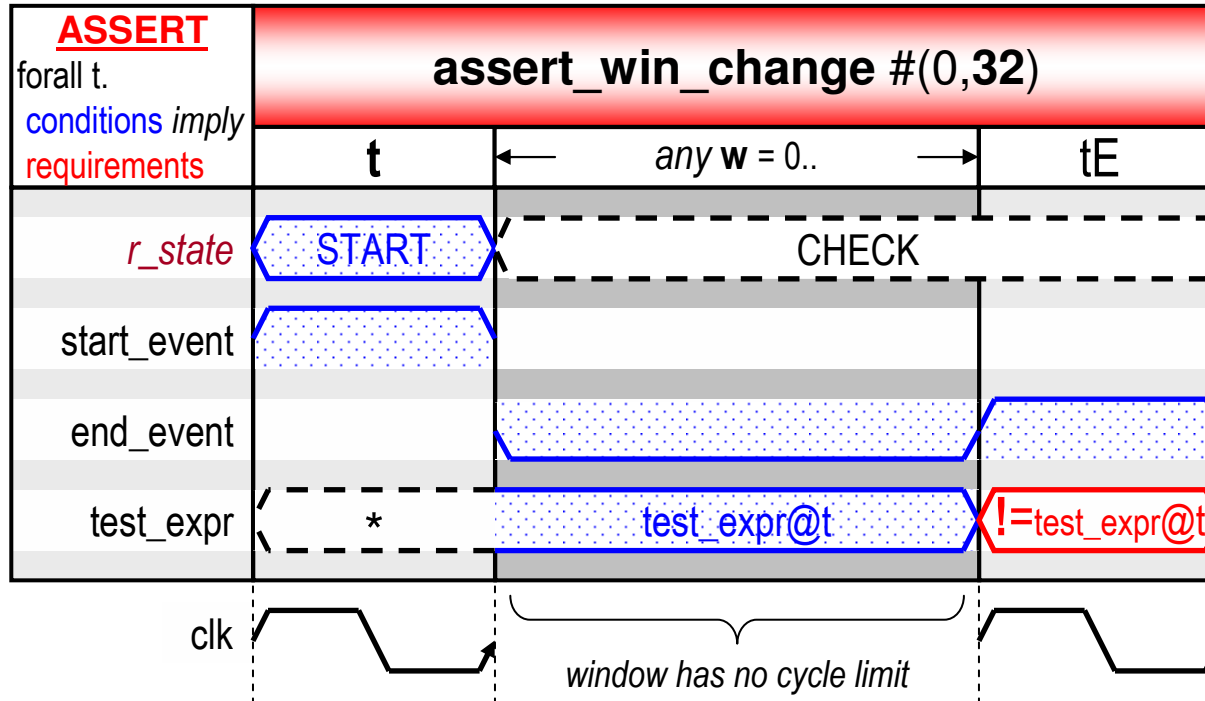
```

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

```

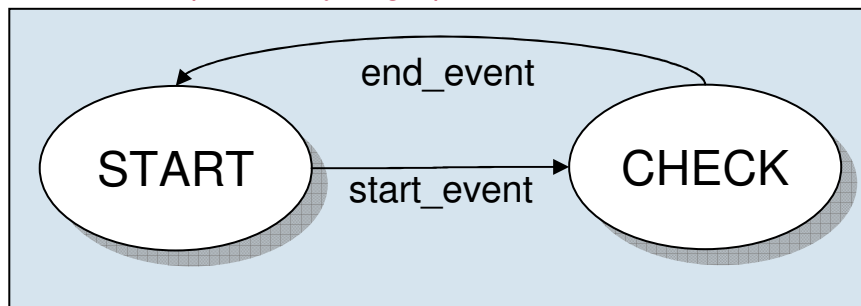
test\_expr must change between start\_event and end\_event

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.

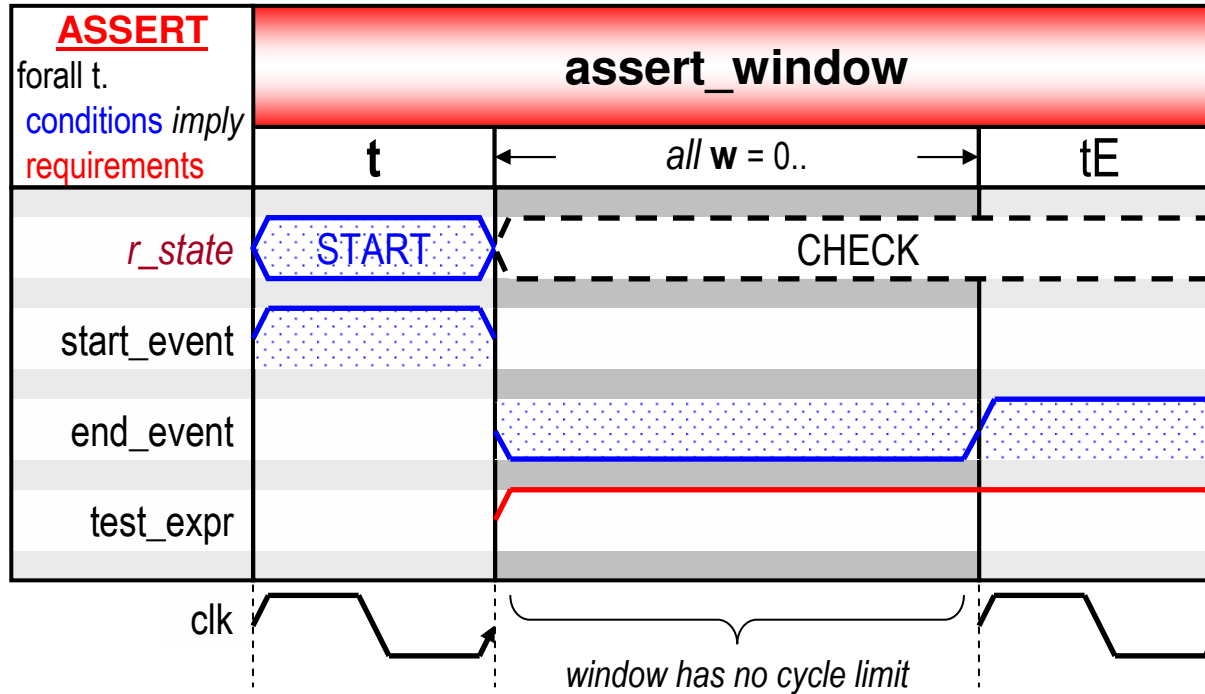


## assert\_window

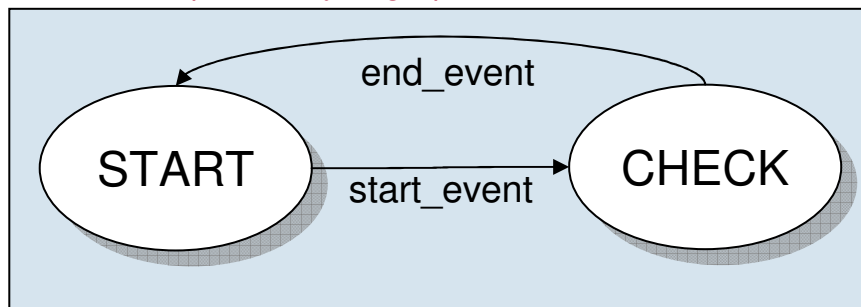
```
#(severity_level, property_type, msg, coverage_level)
ul (clk, reset_n, start_event, test_expr, end_event)
```

test\_expr must hold after the start\_event and up to (and including) the end\_event

Event-bound



### *r\_state* (auxiliary logic)



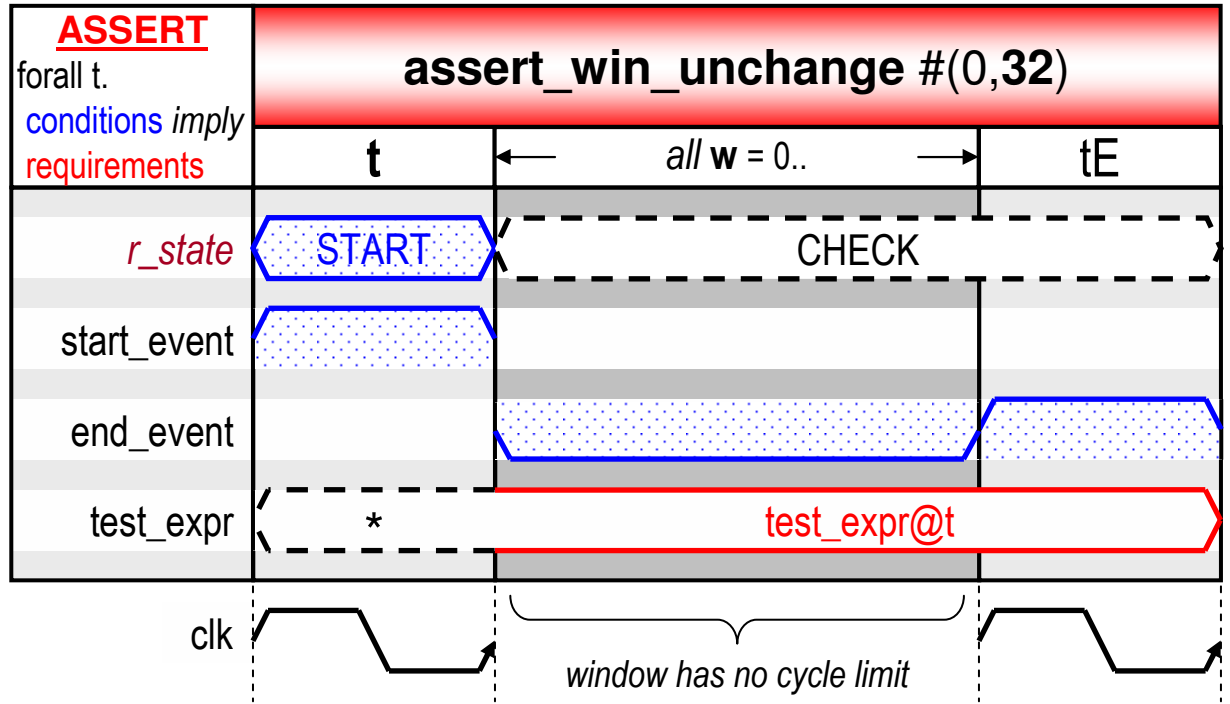
Auxiliary logic necessary, to ignore new start. Checking only begins after `start_event` is true and `r_state==START`.



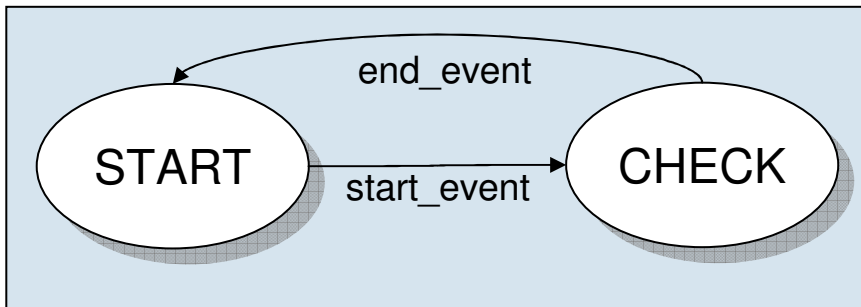
# assert\_win\_unchange

```
#(severity_level, width, property_type, msg, coverage_level)
ul (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.



## assert\_zero\_one\_hot

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

test\_expr must be one-hot or zero, i.e. at most one bit set high

Single-Cycle

