Timing Diagrams for Accellera Standard OVL V1.7

Mike Turpin / ARM
11th October 2006
Contents

§ Introduction to OVL
  § Types of OVL
  § OVL Release History & Major Changes
    § pre-Accellera  Apr 2003
    § v1.0        July 2005
    § v1.1, v1.1a, b Aug 2005
    § v1.5        Dec 2005
    § v1.6        Mar 2006
    § v1.7        July 2006

§ Introduction to Timing Diagrams
  § Timing Diagram Syntax & Semantics
  § Timing Diagram Template

§ OVL Timing Diagrams (alphabetical order)
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
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: 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
Contents

§ Introduction to OVL
  § Types of OVL
  § OVL Release History & Major Changes
    § pre-Accellera Apr 2003
    § v1.1 July 2005
    § v1.1a, b Aug 2005
    § v1.5 Dec 2005
    § v1.6 Mar 2006

§ Introduction to Timing Diagrams
  § Timing Diagram Syntax & Semantics
  § Timing Diagram Diagram Template

§ OVL Timing Diagrams (alphabetical order)
Introduction: OVL Timing Diagram

Implication

**IF**

conditions

**THEN**

requirements

Each timestep maps to a cycle of clk

Clocking Scheme

Assertion Timing Diagram

<table>
<thead>
<tr>
<th>assert_next #(0,1,1,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
</tr>
<tr>
<td>t + 1</td>
</tr>
</tbody>
</table>

start_event

test_expr

“forall t” means timestep t can be any clock cycle

Must Always Hold

Naming

OVL module and parameter values

Conditions in blue (and filled with dots). Requirements in red.

Values

Fails if test_expr does not occur exactly 1 cycle after start_event

Failure

accellera
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.
### Template

**assert_xyz_1of2**: \((\text{min} < 2) \& (\text{max} > 1)\)

<table>
<thead>
<tr>
<th>(t)</th>
<th>(t + 1)</th>
<th>(\text{all } w_1 = 0..(\text{min}-2))</th>
<th>(\text{any } w_2 = 0..(\text{max}-1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal1</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal2</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{signal2} @t)</td>
</tr>
<tr>
<td>signal3</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal1</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal2</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{signal2} @t)</td>
</tr>
<tr>
<td>signal3</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal1</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal2</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{signal2} @t)</td>
</tr>
<tr>
<td>signal3</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal1</td>
<td></td>
<td></td>
<td>(\text{IDLE} \neq \text{IDLE})</td>
</tr>
<tr>
<td>signal2</td>
<td></td>
<td></td>
<td>(\text{SEQ} | \text{BUSY})</td>
</tr>
<tr>
<td>signal3</td>
<td></td>
<td></td>
<td>(\text{OKAY})</td>
</tr>
</tbody>
</table>

**forall \(t\). conditions imply requirements**

- \(t + 1\) = \(\text{IDLE}\)
- \(\text{IDLE} \neq \text{IDLE}\)
- \(\text{IDLE} \neq \text{signal2} @t\)
- \(\text{IDLE} \neq \text{IDLE}\)
- \(\text{IDLE} \neq \text{IDLE}\)
- \(\text{SEQ} \| \text{BUSY}\)
- \(\text{OKAY}\)

**clk**

- \(\text{signal1} @t+1\)
- \(\text{signal4} @t+1\)
- \(\text{signal0} @t+1\)
- \(\text{signal2} @t+1\)
- \(\text{signal0} @t+1\)
Contents

§ Introduction to OVL
  § Types of OVL
  § OVL Release History & Major Changes
    § pre-Accellera Apr 2003
    § v1.1 July 2005
    § v1.1a, b Aug 2005
    § v1.5 Dec 2005
    § v1.6 Mar 2006

§ Introduction to Timing Diagrams
  § Timing Diagram Syntax & Semantics
  § Timing Diagram Template

§ OVL Timing Diagrams (alphabetical order)
**assert_always**

#(severity_level, property_type, msg, coverage_level)

ui (clk, reset_n, test_expr)

test_expr must always hold

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_always</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t. conditions imply requirements</td>
<td>t</td>
</tr>
<tr>
<td>test_expr</td>
<td></td>
</tr>
</tbody>
</table>

2-val view

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_always</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t. conditions imply requirements</td>
<td>t</td>
</tr>
<tr>
<td>test_expr</td>
<td>1/X/Z</td>
</tr>
</tbody>
</table>

4-val view

assert_always will **optimistically** pass if test_expr is X (assert_never will **pessimistically** fail).
assert_always_on_edge

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

ui (clk, reset_n, sampling_event, test_expr)

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

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_always_on_edge #$(0,1)</th>
<th>Rising edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td>conditions imply requirements</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_always_on_edge #$(0,2)</th>
<th>Falling edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td>conditions imply requirements</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_always_on_edge #$(0,3)</th>
<th>Any edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td>conditions imply requirements</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sampling_event</td>
<td>*SE</td>
<td></td>
</tr>
<tr>
<td>test_expr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-Cycles
assert_change

```verilog
#(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

<table>
<thead>
<tr>
<th>ASSERT</th>
<th><code>assert_change #(0, 32, 5, 0) // ignore new start</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t. conditions</td>
<td>imply requirements</td>
</tr>
<tr>
<td><strong>r_state</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>START</strong></td>
</tr>
<tr>
<td></td>
<td>CHECK</td>
</tr>
<tr>
<td><strong>start_event</strong></td>
<td></td>
</tr>
<tr>
<td><strong>test_expr</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>test_expr@t</code></td>
</tr>
<tr>
<td></td>
<td><code>!=test_expr@t</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>clk</th>
<th>If num_cks=5, window is exactly 4 clock cycles wide</th>
</tr>
</thead>
</table>

**r_state (auxiliary logic)**

```
START
start_event
i=num_cks

CHECK
i-1=1
```

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

```
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.
assert_change

\[(0, 32, 5, 1)\] // reset on new start

---

### Differs to April 2003

From OVL version 1.0 the check window spans the entire `num_cks-1` cycles.
test_expr must change within num_cks cycles of start_event

assert_change
#(0, 32, 5, 2) // error on new start 1of2

---

assert_change
#(0, 32, 5, 1) // error on new start 2of2

---

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

If num_cks=5, window is exactly 4 clock cycles wide
If the initial sequence holds, the final sequence must also hold (final is 1-cycle or N-1 cycles)

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

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

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

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

<table>
<thead>
<tr>
<th>( r_{state} )</th>
<th>( event_seq[2] )</th>
<th>( event_seq[1] )</th>
<th>( event_seq[0] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>( t+1 )</td>
<td>( t+2 )</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{assert_cycle_sequence} \#(0,3,2) \)

\( \text{r}_{\text{State}} \)

\( \text{START} \)

\( \text{CHECK} \)

\( \text{clk} \)

\( \text{num_cks}=3 \)

\( \text{necessary\_condition}=2 \)

\( ('OVL\_TRIGGER\_ON\_FIRST\_NONPIPE) \)

\( r_{\text{Cycles}} \)

\( \text{r}_{\text{State}} \) (auxiliary logic)

Need auxiliary logic, to \textit{ignore} subsequent \( \text{event\_seq[\text{num\_cks}-1]} \) when non-pipelined.
If `test_expr` changes, it must decrement by the `value` parameter (modulo `2^width`).

### `assert_decrement`

```verilog
assert_decrement #(severity_level, width, value, property_type, msg, coverage_level)
```

- **∀ t.**
  - Conditions imply requirements
  - `test_expr` is both a condition and requirement at `t + 1`.

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

---

**2-Cycles**

```verilog
ui (clk, reset_n, test_expr)
```

- `assert_decrement #(0, 32, 1)`
  - If `test_expr` changes ...
  - ... it must **decrement** by the value parameter

- `test_expr` is both a condition and requirement at `t + 1`. Hence it appears on two rows.
If `test_expr` changes, the delta must be **min** and **max**

<table>
<thead>
<tr>
<th>t</th>
<th>t + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_expr</td>
<td><em>N</em></td>
</tr>
<tr>
<td>test_expr</td>
<td><strong>N ± D</strong></td>
</tr>
<tr>
<td><strong>min D max</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{assert_even_parity}(\text{severity_level, width, property_type, msg, coverage_level})
\]

\[
\text{ui (clk, reset_n, test_expr)}
\]

<table>
<thead>
<tr>
<th><strong>ASSERT</strong></th>
<th><strong>assert_even_parity</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t. conditions imply requirements</td>
<td>t</td>
</tr>
</tbody>
</table>

**test_expr** must have an even parity, i.e. an even number of bits asserted.
FIFO pointers should never overflow or underflow.

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_fifo_index</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td></td>
</tr>
<tr>
<td>conditions imply</td>
<td></td>
</tr>
<tr>
<td>requirements</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
</tr>
</tbody>
</table>

- push: *H
- pop: *P
- cnt: *C

- C+H-P < P

Should not overflow

{ Should not underflow

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)
ui (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.

**assert_frame**

```markdown
forall t. conditions imply requirements
```

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_frame #(0,3,0) // ignore new start, max_cks=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t - 1</td>
<td>t</td>
</tr>
<tr>
<td>all w1 = 0..(min_cks-1)</td>
<td></td>
</tr>
</tbody>
</table>

If min_cks=3, window w1 is exactly 2 clock cycles wide.

**r_state** (auxiliary logic)

- `ii<min_cks & test_expr || ii=max_cks & !test_expr`
- `$roose(start_event)`

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

(assert_frame, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)

assert_frame #(0,0,7,0) // ignore new start, min_cks=0

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

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

(assert (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.

---

assert_frame #(0,3,7,0) // ignore new start, min_cks>0, max_cks>min_cks

---

r_state (auxiliary logic)

---

accellera
assert_frame

#(severity_level, min_cks, max_cks, action_on_new_start, property_type, msg, coverage_level)
ui (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.

---

### assert_frame

#(0,3,7,1) // reset on new start, min_cks>0, max_cks>min_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.
The document contains a diagram and text explaining the behavior of a VCS assertion frame.

### assert_frame

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

**assert_frame**

The assertion frame is defined using a guard expression that checks the conditions for a new start event.

### Conditions

1. **forall t.**
   - **conditions imply requirements**

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

2. **If min_ks=3, window w1 is exactly 2 clock cycles wide**

   ```verilog
   test_expr must hold by t+max_cks
   ```

3. **If max_ks=7 and min_ks=3, window w2 can be anything from 0 to 4 cycles wide**

   ```verilog
   test_expr must hold by t+max_cks
   ```

### Diagram

The diagram illustrates the timing and logic flow for the assertion frame, with annotations explaining the different states and transitions.

- **rose_start_event** (auxiliary logic)

  ```verilog
  start_event -> r_start_event
  clk -> rose_start_event
  ```

  **“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

---

**assert_handshake**

\#(0.0,3,1,1.1) // req cannot drop, deassert_count=max_ack_length

---

**ASSERT**

forall t.

conditions imply

requirements

<table>
<thead>
<tr>
<th></th>
<th>t - 1</th>
<th>t</th>
<th>any w1 = 0..(max_ack_cycle-1)</th>
<th>t + w1</th>
<th>t + 1 + w1</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**clk**

If max_ack_cycle=3 and min_ack_cycle=0, window w1 is anything from 0 to 2 cycles wide

ack must rise by t+max_ack_cycle

---

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).
If antecedent_expr holds then consequent_expr must hold in the same 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.
If `test_expr` changes, it must increment by the `value` parameter (modulo `2^width`).

**assert_increment**

#(severity_level, width, value, property_type, msg, coverage_level)

u1 (clk, reset_n, test_expr)

---

### ASSERT

FOR ALL `t`.

**CONDITIONS IMPLY REQUIREMENTS**

<table>
<thead>
<tr>
<th><code>test_expr</code></th>
<th><code>test_expr</code></th>
</tr>
</thead>
</table>

**assert_increment**

#(0, 32, 1)

<table>
<thead>
<tr>
<th><code>t</code></th>
<th><code>t + 1</code></th>
</tr>
</thead>
</table>

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

\[(N + V) \mod 2^w\]

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

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

---

2-Cycles
assert_never

#(severity_level, property_type, msg, coverage_level)
ui (clk, reset_n, test_expr)

test_expr must never hold

assert_never will **pessimistically** fail if test_expr is X (assert_always will **optimistically** pass).

Can disable failure on X/Z via:

```vhdl`
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.

Often used as an explicit X-checking assertion
assert_never_unknown_async

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

ui (reset_n, test_expr)

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

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)
ul (clk, reset_n, start_event, test_expr)

test_expr must hold num_cks cycles after start_event holds

### ASSERT
for all t.

**conditions imply requirements**

<table>
<thead>
<tr>
<th></th>
<th>assert_next #$(0,1,1,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t + 1</td>
</tr>
</tbody>
</table>

- num_cks=1
- check_overlapping=1
- check_missing_start=0

### ASSERT
for all t.

**conditions imply requirements**

<table>
<thead>
<tr>
<th></th>
<th>assert_next #$(0,1,0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t + 1</td>
</tr>
</tbody>
</table>

- num_cks=1
- check_overlapping=0
- check_missing_start=0

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

<table>
<thead>
<tr>
<th>Assert</th>
<th>assert_next #(0,1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>num_cks=1</td>
</tr>
<tr>
<td></td>
<td>check_overlapping=1</td>
</tr>
<tr>
<td></td>
<td>check_missing_start=1</td>
</tr>
</tbody>
</table>

forall t.
conditions imply requirements

<table>
<thead>
<tr>
<th>start_event</th>
<th>t</th>
<th>t + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>test_expr</td>
<td></td>
</tr>
</tbody>
</table>

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

```verilog
define assert_no_overflow
  #(severity_level, width, min, max, property_type, msg, coverage_level)
  u1 (clk, reset_n, 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

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_no_transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td></td>
</tr>
<tr>
<td>conditions</td>
<td>imply requirements</td>
</tr>
<tr>
<td>test_expr</td>
<td>start_state</td>
</tr>
<tr>
<td></td>
<td>!= next_state</td>
</tr>
<tr>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>

2-Cycles
If `test_expr` is at `min`, in the next cycle `test_expr` must be `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`.

The expression is:

```
assert_no_underflow #(0, 3, 0, 7) #
```

**Conditions:**
- `width=3`
- `min=0`
- `max=7`

**Example: **

```
assert_no_underflow (#(severity_level, width, min, max, property_type, msg, coverage_level))
```

```
ui (clk, reset_n, test_expr)
```

```
ASSERT
forall t.
conditions imply requirements
```

<table>
<thead>
<tr>
<th>Test</th>
<th>Expr</th>
<th>Min</th>
<th>MinN &lt; Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### assert_odd_parity

```verilog
assert_odd_parity
#(severity_level, width, property_type, msg, coverage_level)
{ui (clk, reset_n, test_expr)
```

**assert_odd_parity**

- **forall** `t`
- **conditions imply** `odd parity`
- **test_expr odd parity**

**clk**

---

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

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) u1 (clk, reset_n, test_expr)

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

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

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

<table>
<thead>
<tr>
<th>assert_proposition</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_expr</td>
<td></td>
</tr>
</tbody>
</table>

This is an asynchronous version of the clocked `assert_always`
**assert_quiescent_state**

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

- **state_expr** must equal **check_value** on a rising edge of **sample_event**

### ASSERT

<table>
<thead>
<tr>
<th>conditions imply requirements</th>
<th>state_expr</th>
<th>check_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td><code>t</code></td>
<td><code>t + 1</code></td>
</tr>
</tbody>
</table>

**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

**2-Cycles**
```plaintext
assert_range
#(severity_level, width, min, max, property_type, msg, coverage_level)
u1 (clk, reset_n, test_expr)

<table>
<thead>
<tr>
<th><strong>ASSERT</strong></th>
<th>assert_range</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t. conditions imply requirements</td>
<td>#(0,3,1,6)</td>
<td>t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test_expr</th>
<th>min N max</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>
```

**test_expr must be** min and max
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

assert_time #(0,5,0) // ignore new start

forall t.
conditions imply requirements

r_state

start_event

test_expr

clk

If num_cks=5, window is exactly 4 clock cycles wide

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)
ui (clk, reset_n, start_event, test_expr)

forall t. conditions imply requirements

assert_time #(0,5,1) // reset on new start

forall t. conditions imply requirements

assert_time #(0,5,1) // error on new start

For assert_time, “reset on new start” effectively performs pipelined checking (see note 2).

If num_cks=5, window is exactly 4 clock cycles wide

n-Cycles

num_cks=5
action_on_new_start=1
(‘OVL_RESET_ON_NEW_START)

requirement on start_event

num_cks=5
action_on_new_start=2
(‘OVL_ERROR_ON_NEW_START)

test_expr must hold for num_cks cycles after start_event
Assert transition

(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.

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td>conditions imply requirements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>t + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_expr</td>
<td>start_state</td>
</tr>
<tr>
<td>test_expr</td>
<td>next_state</td>
</tr>
</tbody>
</table>

If test_expr changes from start_state ...

... it can only change into next_state.

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

test_expr can remain in start_state (in which case the condition at t+1 does not hold).
assert_unchange
#(severity_level, width, num_cks, action_on_new_start, property_type, msg, coverage_level)
ui (clk, reset_n, start_event, test_expr)

test_expr must not change within num_cks cycles of start_event

assert_unchange #((0,32,5,0) // ignore new start

- Conditions
  - forall t.
  - conditions imply requirements
  - r_state
  - start_event
  - test_expr

- Requirements
  - t = all w = 0..(num_cks-1) → t + num_cks
  - If num_cks=5, window is exactly 4 clock cycles wide

- 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.
test_expr must not change within num_cks cycles of start_event

ASSERT
for all t. conditions imply requirements
assert_unchange #(0,32,5,1) // reset on new start

ASSERT
for all t. conditions imply requirements
assert_unchange #(0,32,5,1) // error on new start

For assert_unchange “reset on new start” is pipelined checking. Don’t need auxiliary logic to express this.

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

num_cks=5
action_on_new_start=1
('OVL_RESET_ON_NEW_START)

num_cks=5
action_on_new_start=2
('OVL_ERROR_ON_NEW_START)

requirement on start_event

num_cks=5
If num_cks=5, window is exactly 4 clock cycles wide

If num_cks=5, window is exactly 4 clock cycles wide
**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

---

<table>
<thead>
<tr>
<th>assert_unchange</th>
<th>INCOMPLETE #1: assert_unchange #(0,32,5,0) // ignore new start</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASSERT</strong></td>
<td>for all t. conditions imply requirements</td>
</tr>
<tr>
<td>t - num_cks</td>
<td>all ( w_1 = 0..(num-1) )</td>
</tr>
<tr>
<td>t</td>
<td>all ( w_2 = 0..(num-1) )</td>
</tr>
<tr>
<td>test_expr@t</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>assert_unchange</th>
<th>INCOMPLETE #2: assert_unchange #(0,32,5,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASSERT</strong></td>
<td>for all t. conditions imply requirements</td>
</tr>
<tr>
<td>t</td>
<td>all ( w = 0..(num_cks-1) )</td>
</tr>
<tr>
<td>start_event</td>
<td></td>
</tr>
<tr>
<td>test_expr@t+1</td>
<td></td>
</tr>
</tbody>
</table>

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)  
ui (clk, reset_n, test_expr)

test_expr must hold for between min_cks and max_cks cycles

### Exact min_cks cycles

```plaintext
assert_width #(0,4,0) // min>1, no max

---

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t+1</td>
<td><strong>all w1 = 0..(min_cks-2)</strong></td>
<td>t + min_cks</td>
</tr>
</tbody>
</table>
```

min_cks > 1  
max_cks = 0  
(just check min)

### From 1 to max_cks cycles

```plaintext
assert_width #(0,1,6) // max>0, no min
```

```plaintext
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t+1</td>
<td><strong>any w2 = 0..max_cks</strong></td>
<td>t + 1 + w2</td>
</tr>
</tbody>
</table>
```

min_cks < 2  
max_cks > 0  
(just check max)

---

**De-assert by**

\( t + \text{max} \_\text{cks} + 1 \)
**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

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>\textbf{assert_width} #(0,4,4) // min&gt;1 and max==min</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{forall t.}</td>
<td>conditions imply requirements</td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
</tr>
<tr>
<td>test_expr</td>
<td>Same as assert_width #(0,4,0)</td>
</tr>
</tbody>
</table>

| clk |

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>\textbf{assert_width} #(0,4,6) // min&gt;1 and max&gt;min</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{forall t.}</td>
<td>conditions imply requirements</td>
</tr>
<tr>
<td>t</td>
<td>t + 1</td>
</tr>
<tr>
<td>test_expr</td>
<td>0..(min_cks-2)</td>
</tr>
</tbody>
</table>
**assert_win_change**

```vhdl
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

**assert_win_change** #(0,32)

<table>
<thead>
<tr>
<th></th>
<th><code>t</code></th>
<th><code>any w = 0..</code></th>
<th><code>tE</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r_state</code></td>
<td>START</td>
<td></td>
<td>CHECK</td>
</tr>
<tr>
<td>start_event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end_event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_expr</td>
<td></td>
<td>test_expr@t</td>
<td>!test_expr@t</td>
</tr>
</tbody>
</table>

**clk**

Window has no cycle limit

**assert_win_change** 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**

\[ \text{#(severity\_level, property\_type, msg, coverage\_level)} \]

\[ u1 (\text{clk, reset\_n, start\_event, test\_expr, end\_event}) \]

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

---

**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)

ui (clk, reset_n, start_event, test_expr, end_event)

- **test_expr** must not change between **start_event** and **end_event**

---

**assert_win_unchange** #(0,32)

<table>
<thead>
<tr>
<th>Event-bound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASSERT</strong></td>
</tr>
<tr>
<td>for all t.</td>
</tr>
<tr>
<td>conditions</td>
</tr>
<tr>
<td>imply</td>
</tr>
<tr>
<td>requirements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>all w = 0..</th>
<th>tE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>r_state</strong></td>
<td>START</td>
<td>CHECK</td>
<td></td>
</tr>
<tr>
<td>start_event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end_event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_expr</td>
<td>*</td>
<td>test_expr@t</td>
<td></td>
</tr>
</tbody>
</table>

**clk**

- Window has no cycle limit

---

**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

<table>
<thead>
<tr>
<th>ASSERT</th>
<th>assert_zero_one_hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall t.</td>
<td>conditions imply requirements</td>
</tr>
<tr>
<td>t</td>
<td></td>
</tr>
<tr>
<td>test_expr</td>
<td>0</td>
</tr>
</tbody>
</table>

clk