Introduction to OVL (Open Verification Library)

Alexander Gnusin

What is OVL?

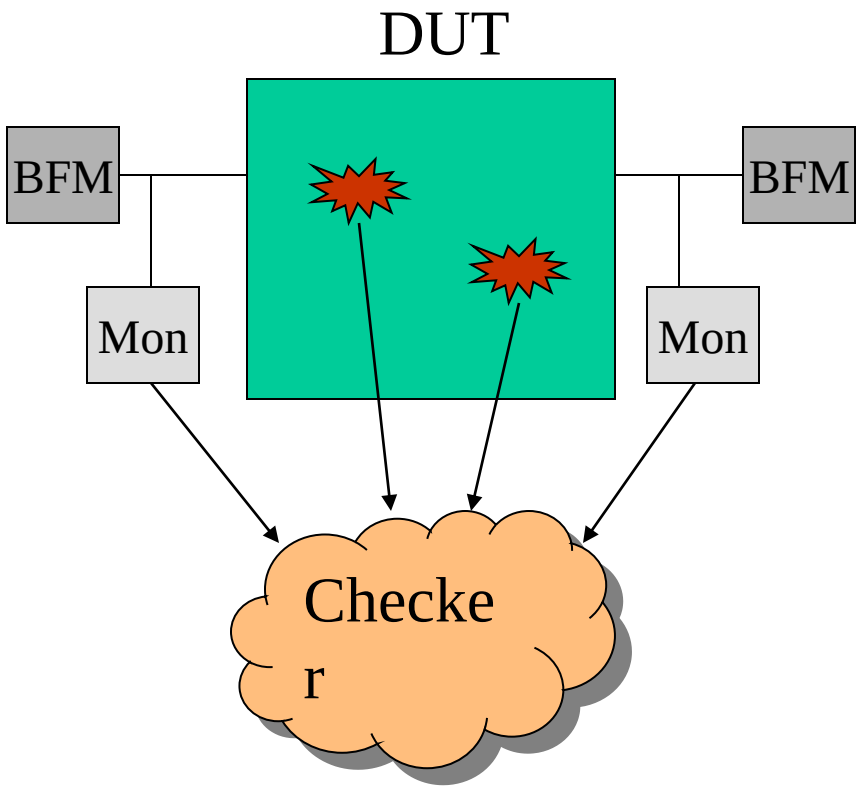
- It is open library of non-synthesisable modules – assertion monitors.
- Their purpose is to guarantee that some conditions hold true.
- If condition goes false, they “fire” with specified message
- They may be connected to any internal point of design.
- They are composed from an event, message and severity.

OVL Example

```
module counter_0_to_9(reset_n,clk);
input reset_n, clk;
reg [3:0] count;
always @(posedge clk)
begin
  if (reset_n == 0 || count >= 9) count = 1'b0;
  else count = count + 1;
end
assert_always #(0,0,0,"error: count not within 0 and 9")
valid_count (clk, reset_n, (count >= 4'b0000) &&
(count <= 4'b1001));
endmodule
```

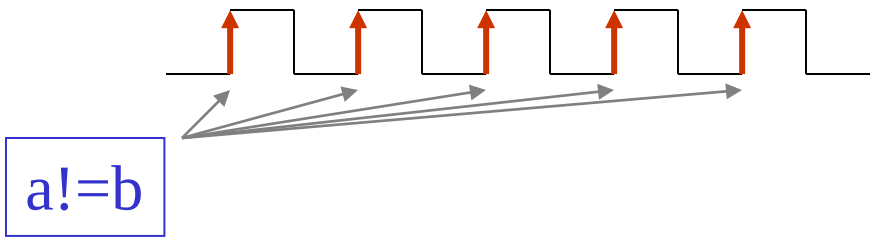
OVL in Test Environment

- **Black-Box Verification approach is not always effective**
- **Use Assertion monitors to track important internal design features**
- **Checker uses Assertion monitors information to make final decision**
- **Assertions "firing" statistics represents functional coverage of tests.**



assert_always

Test_expr must be true at any rising edge of the clock:

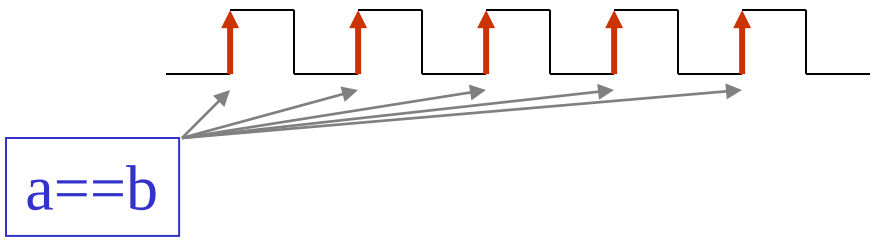


A always is nonequal B at any clock edge:
assert_always a_noteq_b (clk, rst_n, (a!=b));

*Note: Some module parameters are intentionally waived for to facilitate the understanding of basic concepts

assert_never

Test_expr must be false at any rising clock edge:



A never equals B:

```
assert_never a_eq_b (clk, rst_n, (a==b));
```

assert_proposition

Test_expr must be always true (not only on the clock edge!!!)

Useful for the signals on the asynchronous boundaries

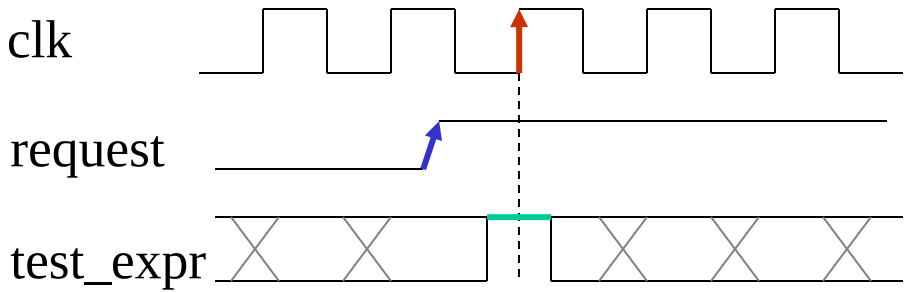
a!=b

A always is nonequal B:

```
assert_proposition a_noteq_b (clk, rst_n, (a!=b));
```

assert_always_on_edge

Test_expr is true at every specified edge of the sampling event and positive clk edge:

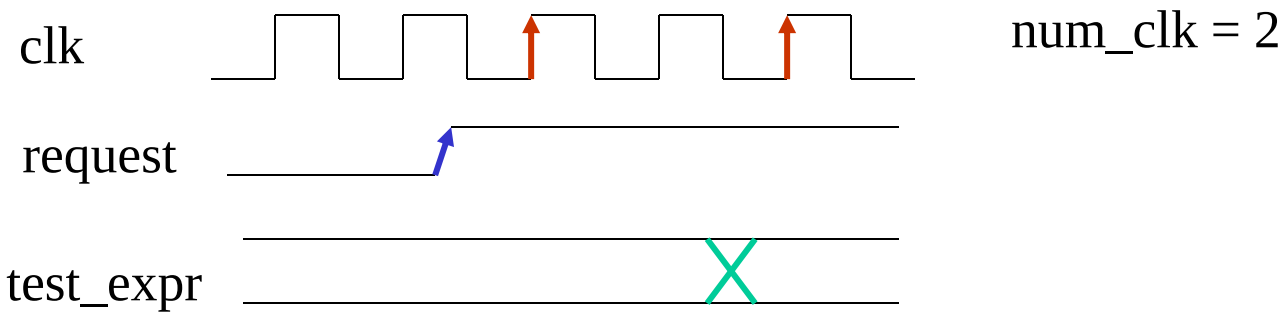


For every new request FSM is ready for it (it is in IDLE state):

assert_always_on_edge fsm_check (clk, rst_n, request, (state == IDLE));

assert_change

Test_expr changes values within the next num_clk cycles after specified event:

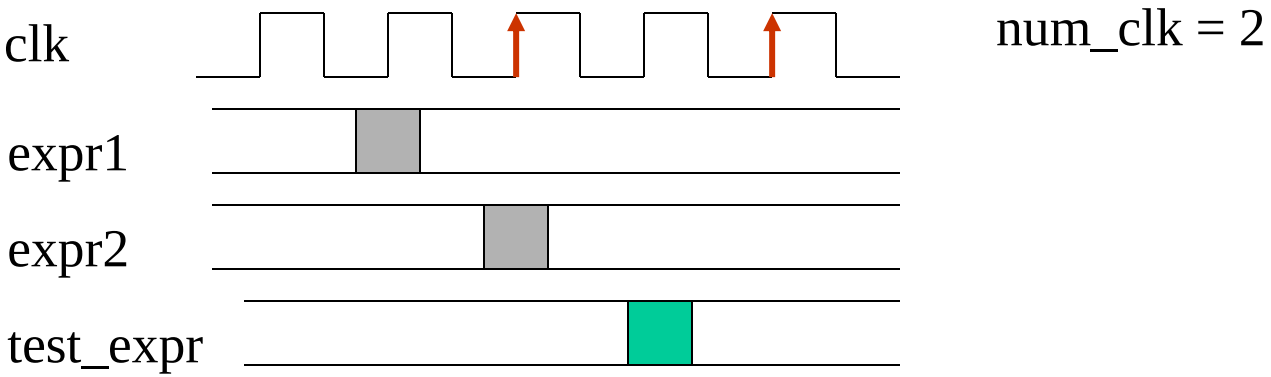


After every new request FSM exits IDLE state in less than 3 cycles:

assert_change #2 fsm_check (clk, rst_n, request, (state == IDLE));

assert_cycle_sequence

Test_expr is true within num_clk cycles after specified event sequence:

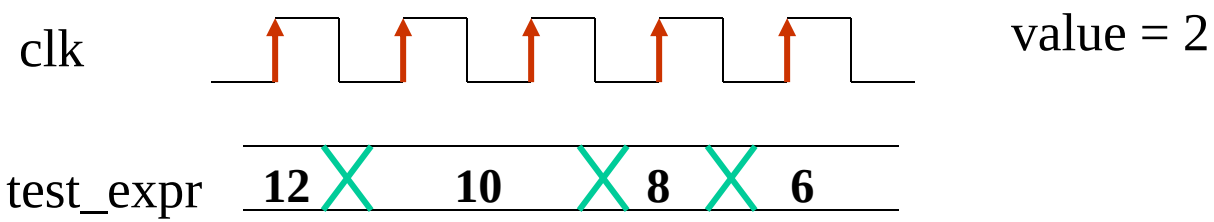


If FSM passes through State1 and State2, it should enter State3 within 2 clock cycles:

```
assert_cycle_sequence #2 fsm_check (clk, rst_n, request, (state == State1, state == State2, state==State3));
```

assert_decrement

Test_expr will never decrease by anything other than value:

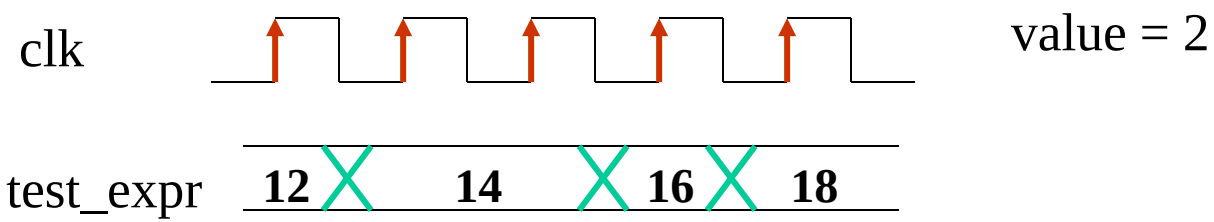


Counter value can be decremented by 2 only:

assert_decrement #2 cnt_check (clk, rst_n, count);

assert_increment

Test_expr will never increase by anything other than value:

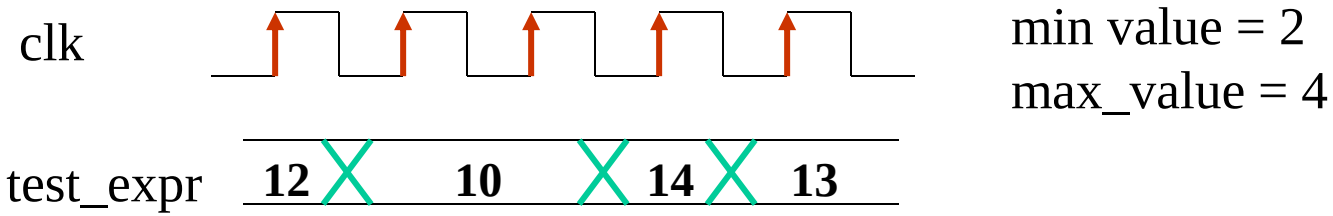


Counter value can be icremented by 2 only:

assert_increment #2 cnt_check (clk, rst_n, count);

assert_delta

Test_expr will never change values by anything less than min value or anything more than max value

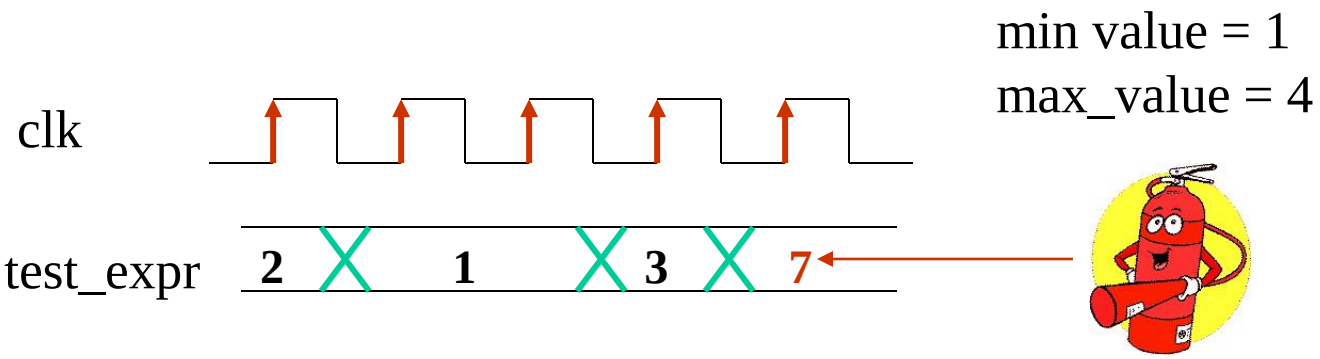


Counter value can be decremented less than by 2 or incremented more than by 4 only:

```
assert_delta #(2,4) cnt_check (clk, rst_n, count);
```

assert_range

Test_expr will always have value within specified min/max range:

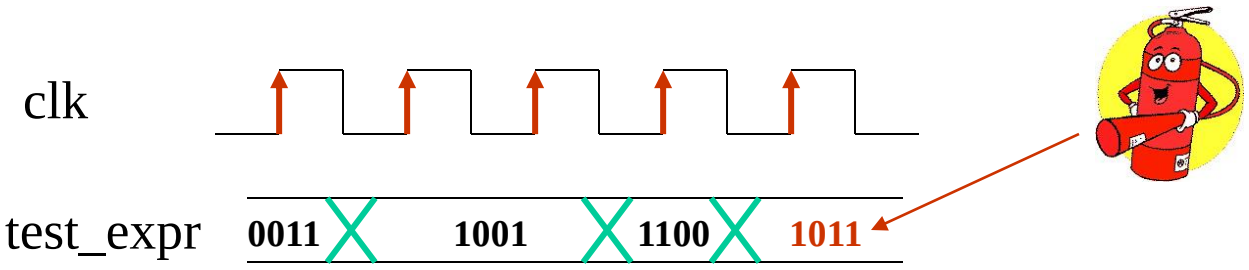


Counter value is always within [1… 4] range:

assert_range #(1,4) cnt_check (clk, rst_n, count);

assert_even_parity

Test_expr always has even number of bits asserted

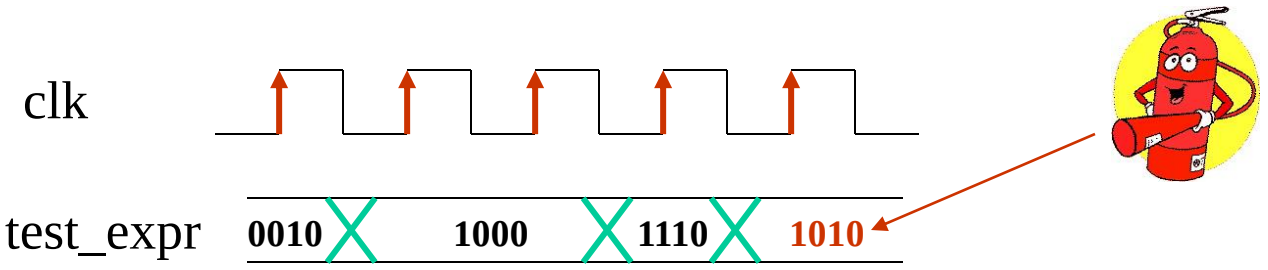


Counter value always hold even parity of bits:

assert_even_parity cnt_parity (clk, rst_n, count);

assert_odd_parity

Test_expr always has odd number of bits asserted

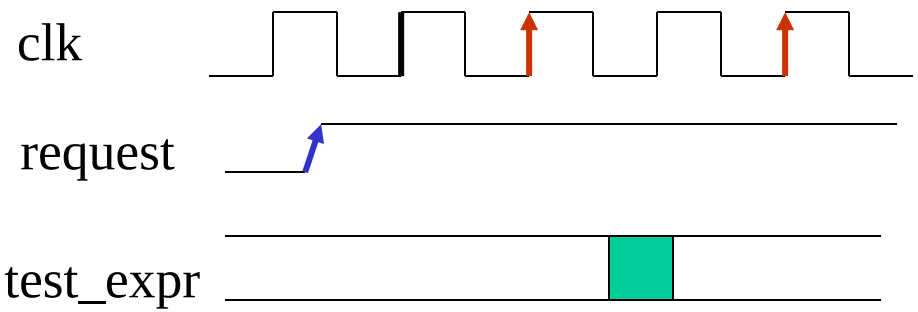


Counter value always hold odd parity of bits:

assert_odd_parity cnt_parity (clk, rst_n, count);

assert_frame

Test_expr must be true within min and max number of cycles after specified event:



min value = 1
max_value = 3

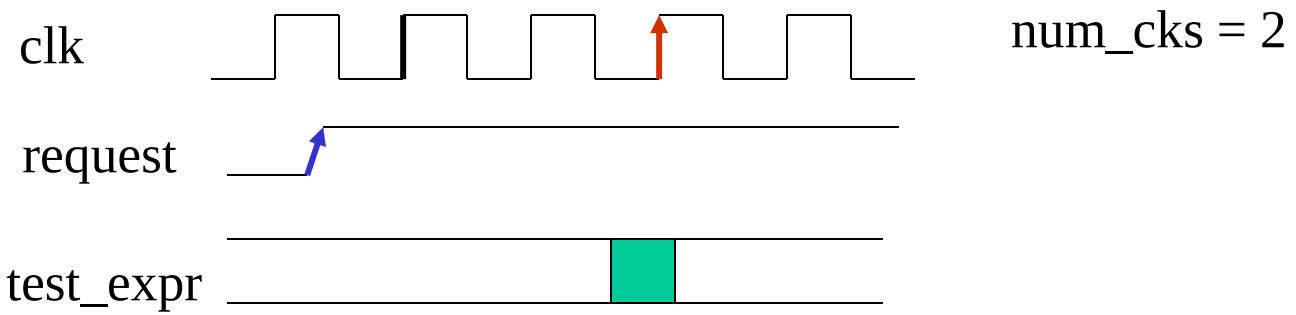
After every new request design issues acknowledgement no earlier than in one cycle and no later than in 3 cycles:

```
assert_frame #(1,3) fsm_check (clk, rst_n, request, ack);
```

assert_next

Test_expr must be true num_cks later after specified event.

Allows also event pairs overlap

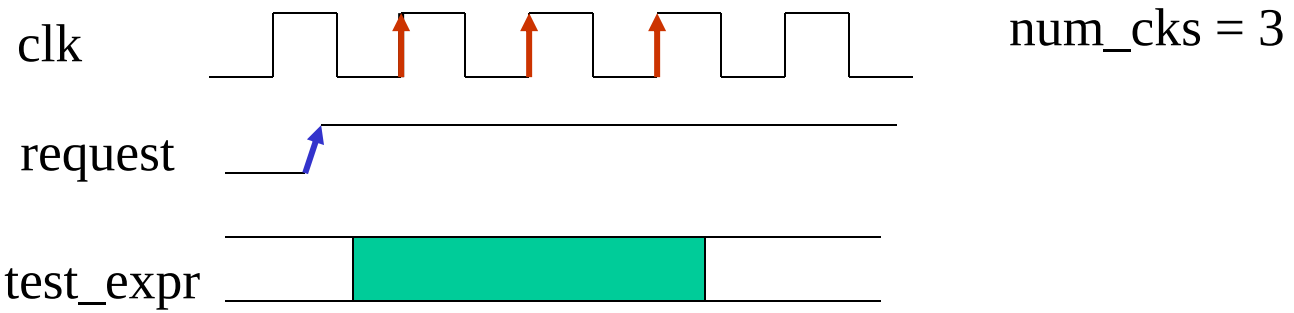


After every new request design issues acknowledgement exactly after 2 clock cycles:

```
assert_next #(2) ack_check (clk, rst_n, request, ack);
```

assert_time

Test_expr must be true during num_cks after specified event.

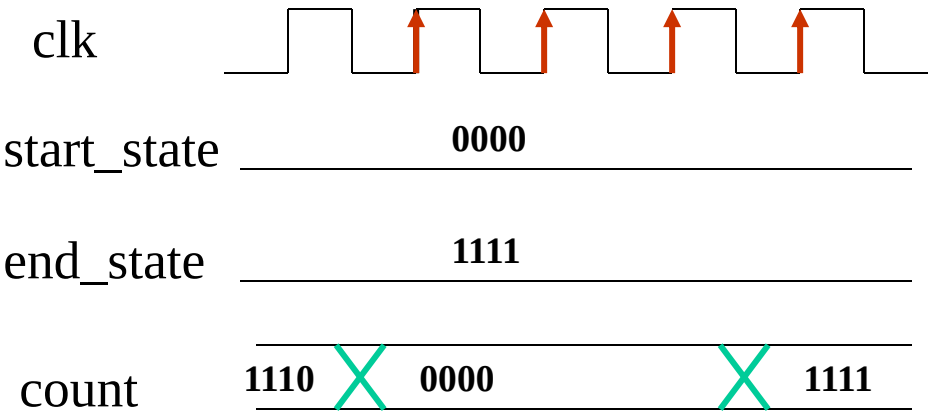


After every request ack goes high and stays high at least 3 cycles:

assert_time #(3) ack_check (clk, rst_n, request, ack);

assert_transition

If test_expr evaluates to the start_state, it must be stable and then eventually transit to the next_state

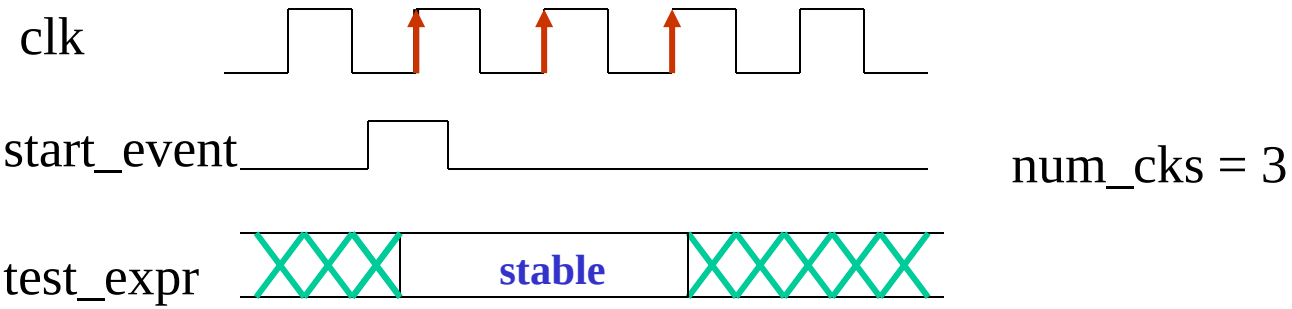


If count reaches “0000” state, it must eventually transit to “1111” state

assert_transition state_check (clk, rst_n, count, 4’b0000, 4’b1111);

assert_unchange

If start_event evaluates TRUE, test_expr will not change values within the next num_cks cycles.

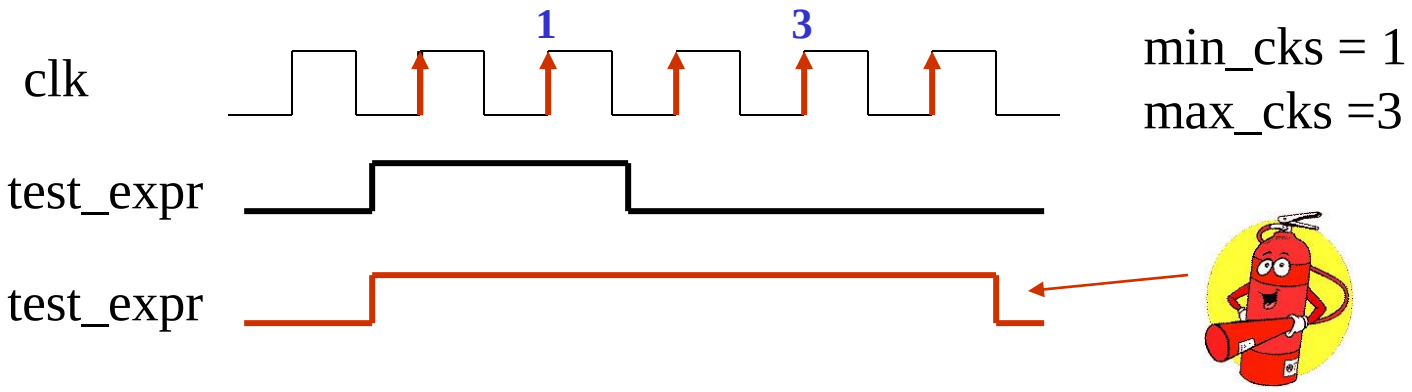


If req equals to 1, FSM don't change states during 3 clock cycles

assert_unchange #3 fsm_check (clk, rst_n, (req==1), state);

assert_width

If test_expr evaluates TRUE, it stays TRUE no less than min and no more than max number of clock cycles.



If frame goes high, it must stay high more than 4, but less than 10 cycles:

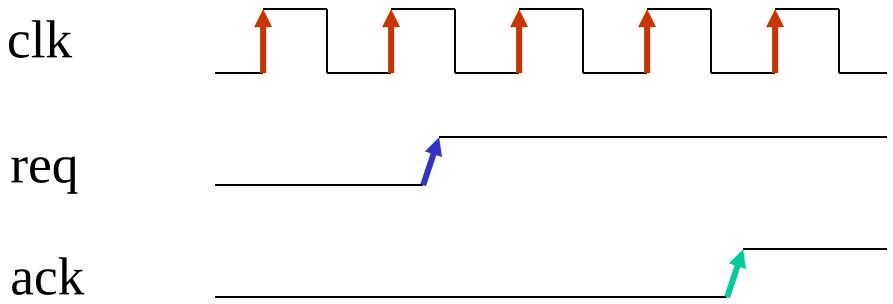
assert_width #(4,10) frame_check (clk, rst_n, (frame==1));

assert_handshake

Constantly monitors **req** and **ack** signals, signaling about:

- Multiple **req's** without no **ack**
- An **ack** without a **req**
- Multiple **ack's** for an active **req**

Timing relationship between req's and ack's can be also monitored.



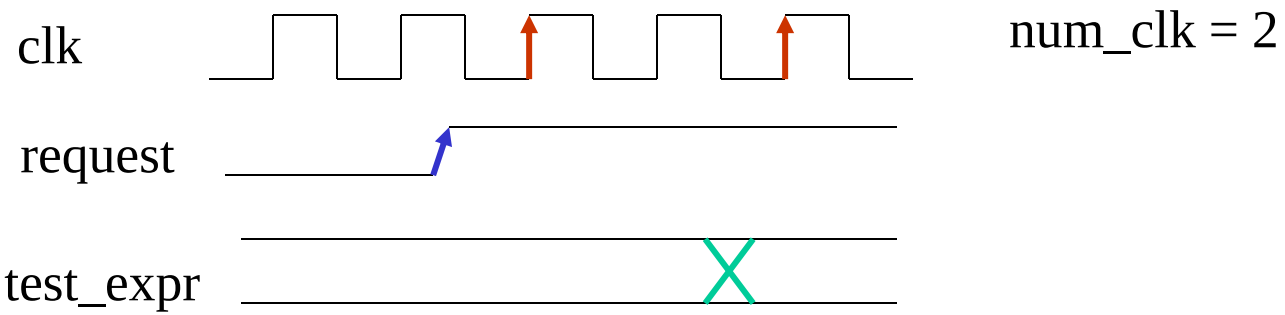
min_ack_cycle = 1
max_ack_cycle = 3

After every new request we have only one ack within 1..3 cycles after req:

assert_handshake #(1,3) handshake_check (clk, rst_n, req, ack);

assert_implication

Test_expr changes values within the next num_clk cycles after specified event:



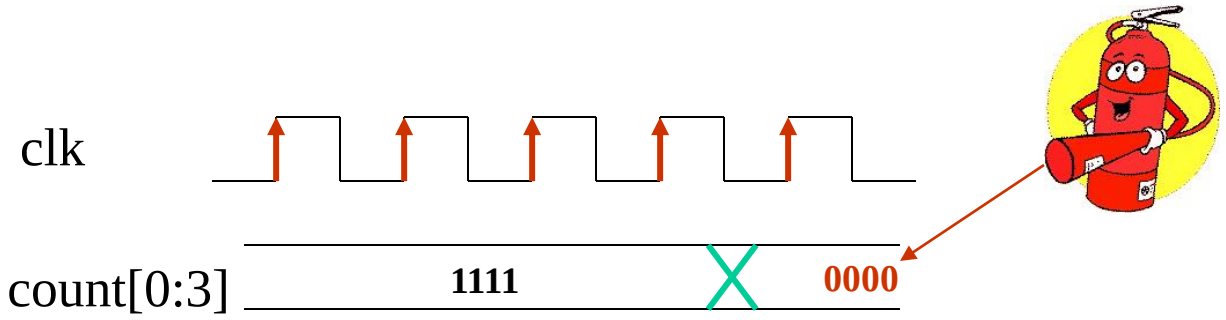
After every new request FSM exits IDLE state in less than 3 cycles:

assert_change #2 fsm_check (clk, rst_n, request, (state == IDLE));

assert_no_overflow

Constantly checks that test_expr will never:

- Changes values from max value (2^width –1) to a value greater than max
- Reach a value less than or equal to a min value (default 0)



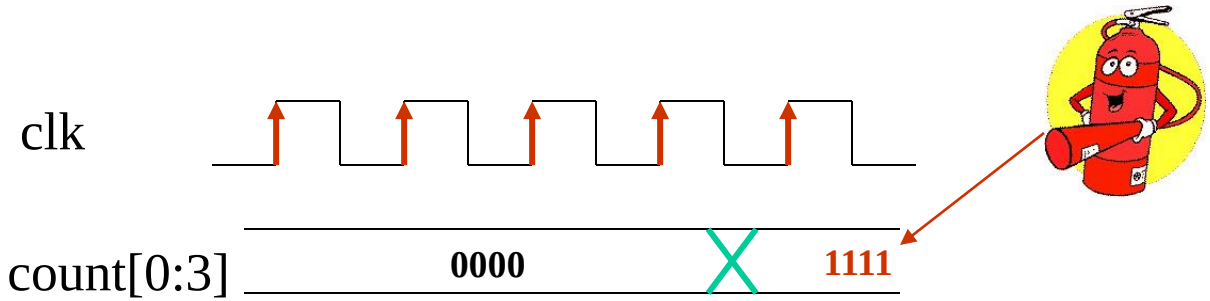
Checks that counter will never wrap around from the highest to lowest value in a range:

```
assert_no_overflow #4 cnt_check (clk, rst_n, count);
```

assert_no_underflow

Constantly checks that test_expr will never:

- Changes values from min value (default 0) to a value less than min or greater than max (default : 2^width –1)

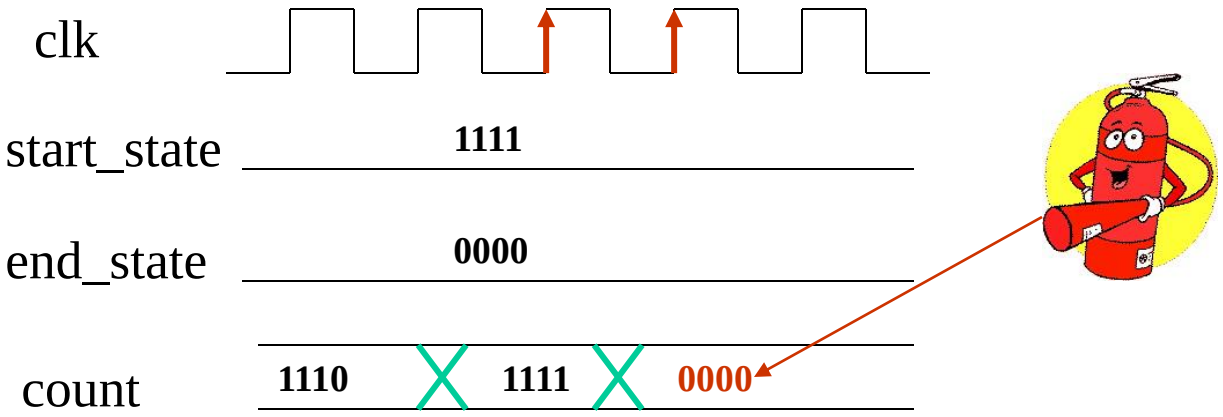


Checks that counter will never wrap around from the lowest to highest value in a range:

```
assert_no_underflow #4 cnt_check (clk, rst_n, count);
```

assert_no_transition

Whenever test_expr evaluates to the start_state, it cannot be equal to the end_state in the next cycle:

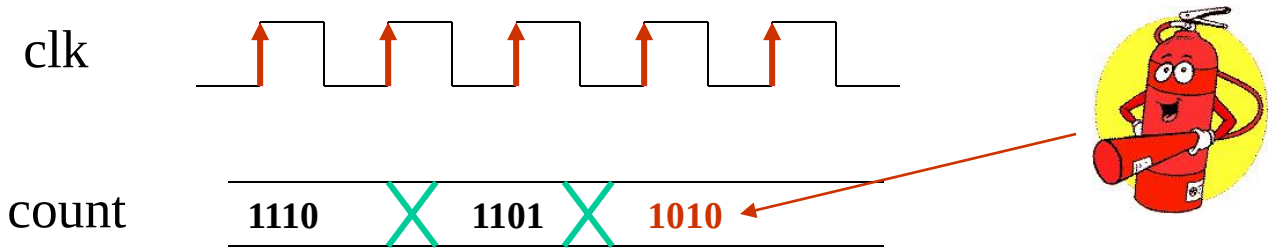


If count evaluates to 4'b1111, it cannot be 4'b0000 in the next clock cycle

assert_no_transition cnt_check (clk, rst_n, count, 4'b1111, 4'b0000);

assert_one_cold

Continuously monitors that test_expr always has exactly one bit asserted low:

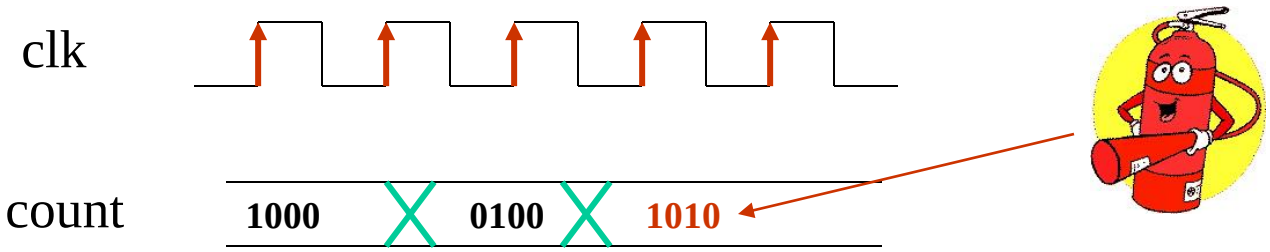


At any time, only one bit in count is asserted low:

assert_one_cold fsm_check (clk, rst_n, count);

assert_one_hot

Continuously monitors that test_expr always has exactly one bit asserted high:

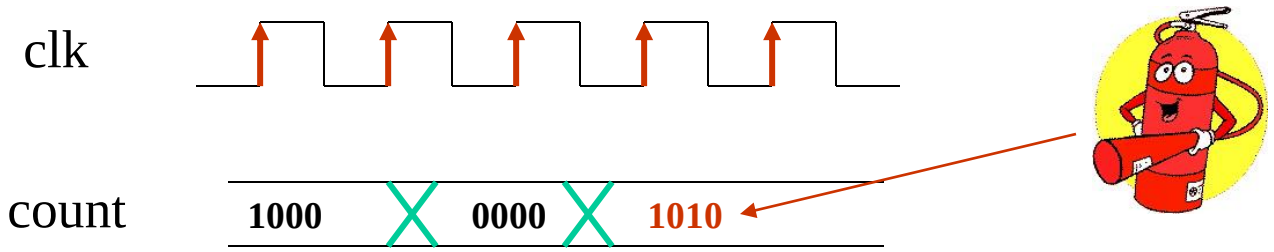


At any time, only one bit in count is asserted high:

assert_one_hot fsm_check (clk, rst_n, count);

assert_zero_one_hot

Continuously monitors that test_expr always has exactly one bit asserted high or no bits asserted:

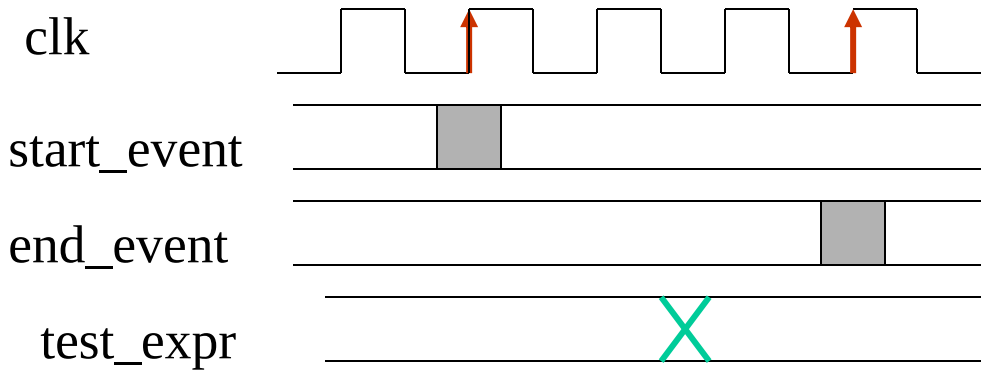


At any time, only one bit in count is asserted high:

assert_zero_one_hot fsm_check (clk, rst_n, count);

assert_win_change

Test_expr must change values between start_event and end_event:

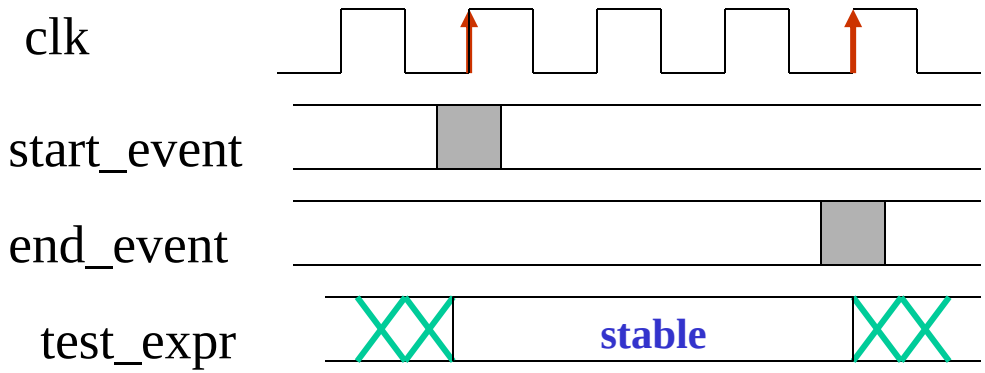


Between State1 and State2, count should change values:

assert_win_change cnt_check (clk, rst_n, (state==State1), count, (state==State2));

assert_win_unchange

Test_expr must not change values between start_event and end_event:

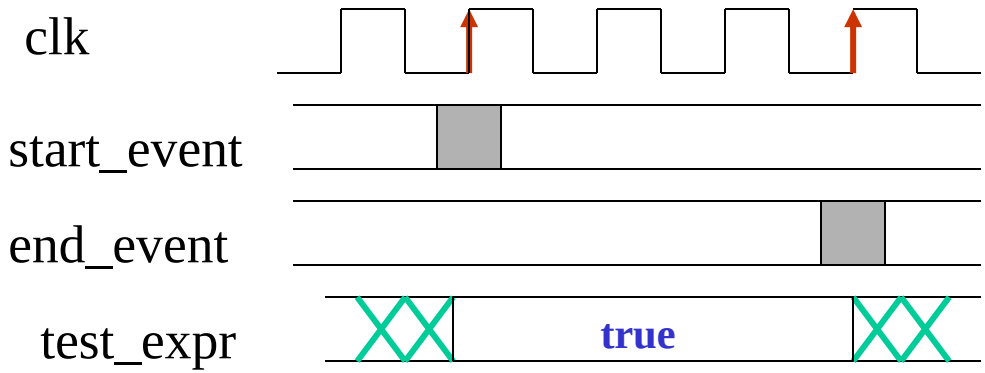


Between State1 and State2, count should NOT change values:

assert_win_unchange cnt_check (clk, rst_n, (state==State1), count, (state==State2));

assert_window

Test_expr must evaluate TRUE between start_event and end_event:



Between State1 and State2, count should be equal to 4'b0101:

```
assert_window cnt_check (clk, rst_n, (state==State1), (count == 4'b0101),
    (state==State2));
```