System Verilog Assertions Simplified

By Smit Patel, eInfochips

Abstract

Assertion is a very powerful feature of System Verilog HVL (Hardware Verification Language). Nowadays it is widely adopted and used in most of the design verification projects.

This article explains the concurrent assertions syntaxes, simple examples of their usage and details of passing and failing scenarios along with waveform snippets for the ease of understanding.

This article is helpful to anyone who is new to system verification and who wishes to learn System Verification (SV) assertions quickly with simple examples. It is also useful to an experienced person who would like to have a quick refresh before he/she starts implementing it after some break.

Introduction

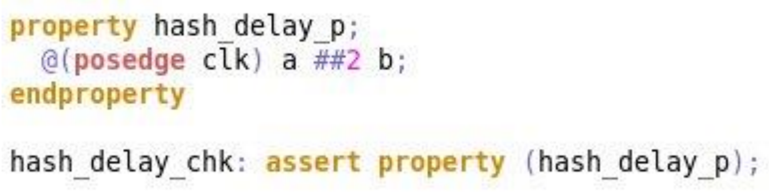
Irrespective of the verification methodology used in a project, System Verilog assertions help speed up the verification process. Identifying the right set of checkers in verification plan and implementing them using effective SV assertions helps to quickly catch the design bugs and ultimately helps in high-quality design.

Some of the key advantages of SV assertions are as mentioned below:

- Multiple lines of checker code can be represented in a few lines effectively using SVA code.
- SVA can be ignored by synthesis.
- Assertion takes lesser time to debug as they pin point the exact time of failure. Assertions can be turned on/off during simulations. They can have severity levels; failures can be non-fatal or fatal errors.
- Multi-Clock assertions are useful in writing checkers around Clock Domain Crossing (CDC) logic
- Assertions can be also used for formal verification.

Let us look at different types of examples of SV assertions.

1. Simple ## delay assertion:



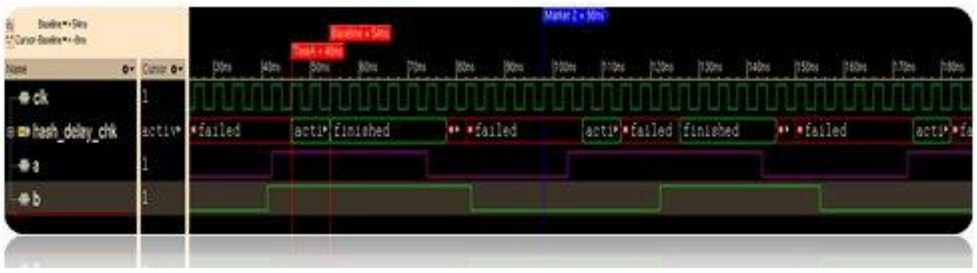
```
property hash_delay_p;
  @(posedge clk) a ##2 b;
endproperty

hash_delay_chk: assert property (hash_delay_p);
```

Property *hash_delay_p* checks for,

a) Signal “a” is asserted high on each clock cycle

b) If “a” is high in a cycle after two clock cycles, signal “b” has to be asserted high.

Snippet:

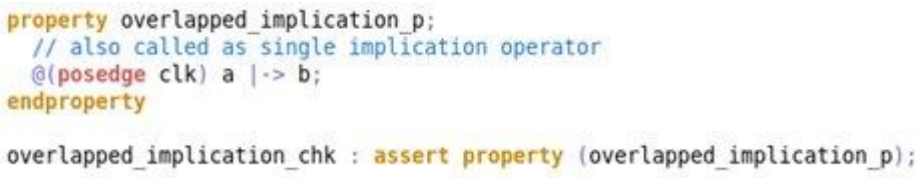


Assertion passes when signal “a” is high and after two clock cycles signal “b” is high.

There are two cases for which assertion fails,

- when signal “a” is not asserted high in any cycle.
- when signal “a” asserts high and after two clock signal “b” is not asserted high.

2. Overlapped Implication:

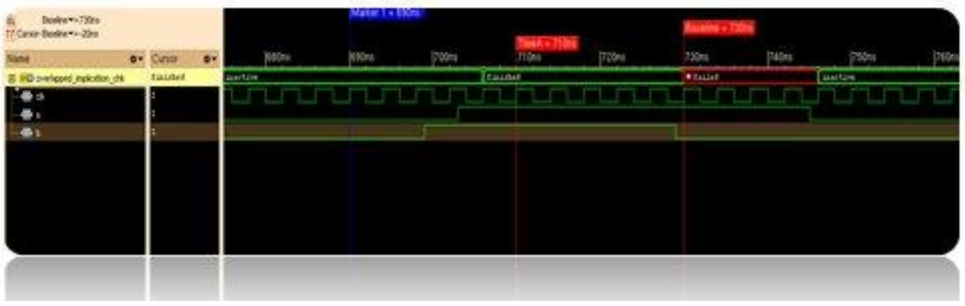


```
property overlapped_implication_p;
  // also called as single implication operator
  @(posedge clk) a |-> b;
endproperty

overlapped_implication_chk : assert property (overlapped_implication_p);
```

Property *overlapped_implication_p* checks,

If the left hand side condition (“a”==1) of the implication “|->” (called "antecedent") is true, check the right hand side (called "consequent") condition (“b” == 1) in the same cycle.

Snippet:



In the above snippet, Assertion passes when signal “a” is high and in the same clock cycles, signal “b” is high. Assertion remains in “Inactive” state when signal “a” is not asserted high.

Assertion fails only when signal “a” is asserted high and in the same clock cycle signal “b” is not asserted high.

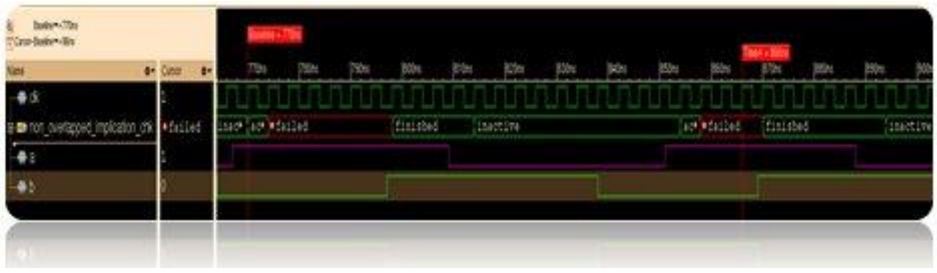
3. Non-Overlapped Implication:

```
property non_overlapped_implication_p;
  // also called as double implication operator
  @(posedge clk) a |=> b;
endproperty

non_overlapped_implication_chk : assert property (non_overlapped_implication_p);
```

Property *Non_overlapped_implication_p* checks,

If the left hand side condition (“a”==1) of the implication “|->” (called "antecedent") is true, check the right hand side (called "consequent") condition (“b” == 1) in the next cycle.

Snippet:



In above snippet, Assertion passes when signal “a” is high and in the next clock cycles, signal “b” is high. Assertion remains in “Inactive” state when signal “a” is not asserted high.

Assertion fails only when signal “a” is asserted high and **in the next clock cycle**, signal “b” is not asserted high.

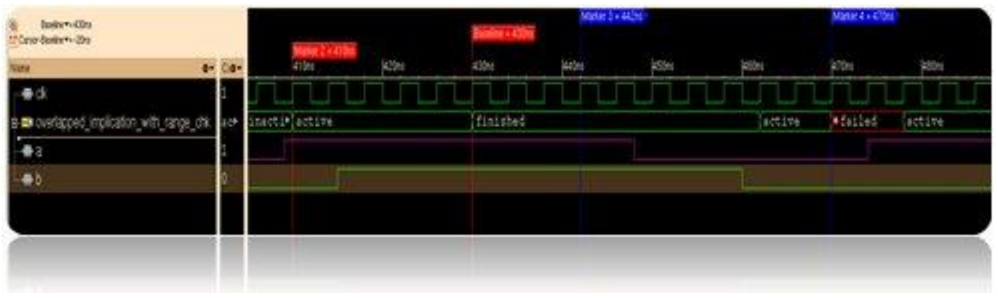
4. Assertion with delay ranges:

```
parameter MIN_DEALY_RANGE = 5;
parameter MAX_DEALY_RANGE = 7;

property overlapped_implication_with_range_p;
  @(posedge clk) a  |-> ##[MIN_DEALY_RANGE : MAX_DEALY_RANGE] b;
endproperty

overlapped_implication_with_range_chk : assert property (overlapped_implication_with_range_p);
```

Property *overlapped_implication_with_range_p* checks,

when signal “a” is asserted high, within given latency ranges it checks assertion of signal “b”.

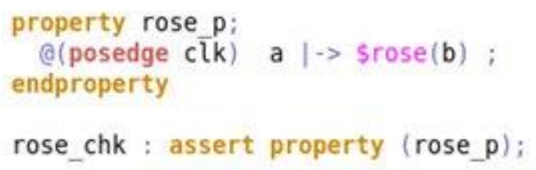
Snippet:



In the above snippet, assertion finishes when signal “a” is asserted high and within 5 to 7 (MIN_DELAY:MAX_DELAY) clock cycles, signal “b” asserts high.

Assertion fails when signal “a” is asserted high and within 5 to 7 clock cycles, signal “b” does **not assert high**.

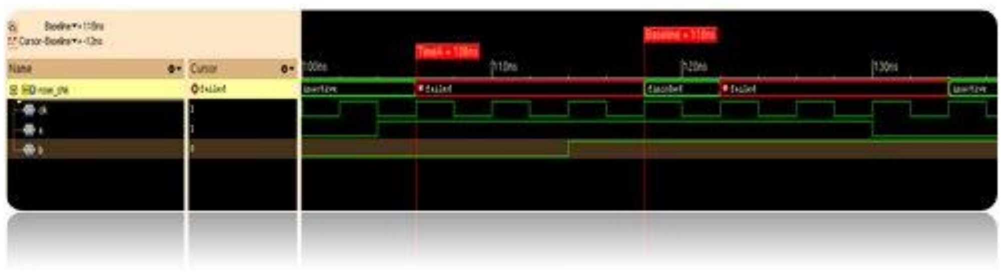
5. $rose:



```
property rose_p;
  @(posedge clk)  a |-> $rose(b) ;
endproperty

rose_chk : assert property (rose_p);
```

Property *rose_p* checks when signal “a” asserts high, in the same cycle it also detects the positive edge on signal “b”.

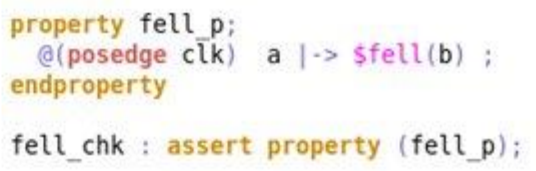
Snippet:



In the above snippet, assertion passes when signal “a” is asserted high and in the same clock cycle, positive edge on signal “b” is detected.

Assertion fails when signal “a” is asserted high, but in the same clock cycle, positive edge on signal “b” is not detected.

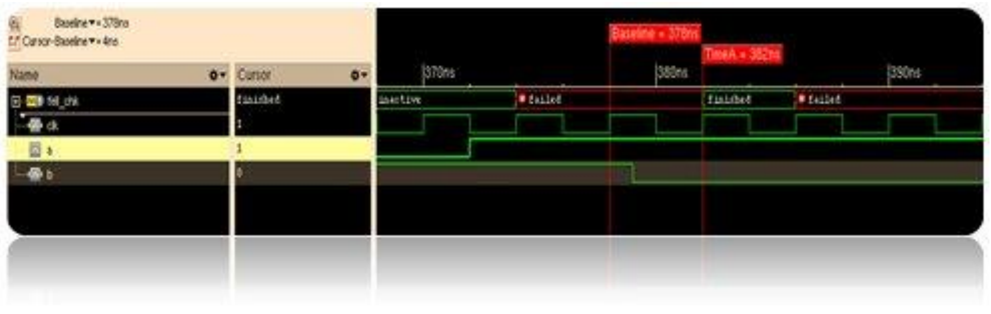
6. $fell:



```
property fell_p;
  @(posedge clk)  a |-> $fell(b) ;
endproperty

fell_chk : assert property (fell_p);
```

Property *fell_p* checks when signal “a” asserts high, in the same cycle, it also detects negative edge on signal “b”.

Snippet:



In the above snippet, assertion passes when signal “a” is asserted high and in the same clock cycle, negative edge on signal “b” is detected.

Assertion fails when signal “a” is asserted high and in the same clock cycle, negative edge on signal “b” is not detected.

7. $stable:

Property *stable_p* checks when signal “a” asserts high, it checks for no change in signal “b”. It means, signal “b” should stay as it was in the previous cycle.

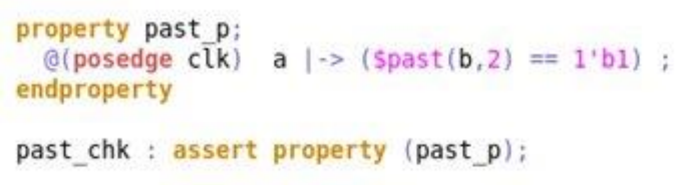
Snippet:

In the above snippet, assertion passes when signal “a” is asserted high and in the same clock cycle, signal “b” remains to the same value as in the previous clock cycle.

Assertion fails when signal “a” is asserted high and in the same clock cycle signal “b” is also changed from to 0 to 1 (not stable).

Similarly, the assertion would fail if signal “a” is asserted high and in the same clock cycle, signal “b” is also changed from to 1 to 0 (not stable).

8. $past:



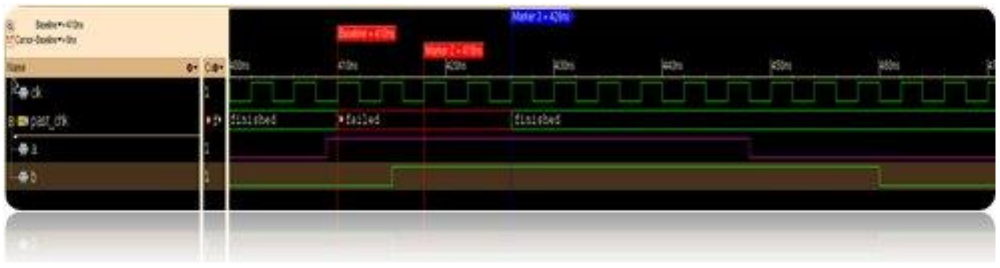
```
property past_p;
  @(posedge clk)  a |-> ($past(b,2) == 1'b1) ;
endproperty

past_chk : assert property (past_p);
```

Syntax:
$past (signal_name)
$past (signal_name, number of clock cycles)

Property *past_p* checks when signal “a” asserts high, it checks signal “b” was high before 2 clock cycles.

If the second argument is not specified, then by default, $past checks for the signal value in the immediate previous cycle.

Snippet:



In the above snippet, assertion passes when signal “a” is asserted high and signal “b” was asserted high before two clock cycles.

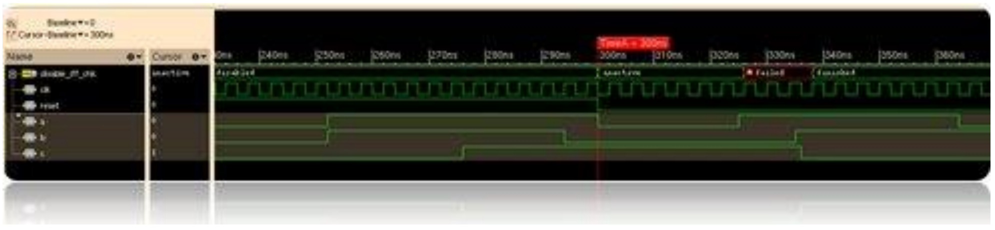
Assertion fails when signal “a” is asserted high and before two clock cycles, signal “b” was not asserted high.

9. Disable Iff :

```
property disable_iff_p;
  disable iff(reset)
  @(posedge clk) a |-> b;
endproperty

disable_iff_chk: assert property (disable_iff_p);
```

Property *disable_iff_p*, remains disabled if signal “reset” is asserted high. If reset is not asserted high, then it checks if signal “a” is asserted high, then in the same cycle, signal “b” should also be asserted high.

Snippet:



In the above snippet, assertion remains disabled as long as reset is asserted high.

When reset is not asserted and when signal “a” is asserted high, assertion passes OR fails according to the value of signal “b” in the same cycle.

10. Consecutive repetition operator:

```
property consecutive_repetition_p;
   @(posedge clk) $rose(a) |=> b[*6] ##1 c;
endproperty

consecutive_repetition_chk : assert property (consecutive_repetition_p);
```

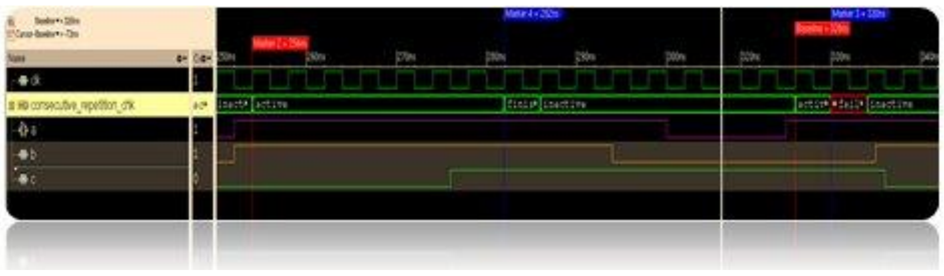
Syntax:

Signal_name [*n]

It specifies that a signal or a sequence will match continuously for the number of clocks specified.

Property consecutive_repeatation_p checks when the positive edge is detected on signal “a”, check from next clock onwards, signal “b” is asserted high continuously for 6 clock cycles, followed by a high value on signal “c” in the next cycle.

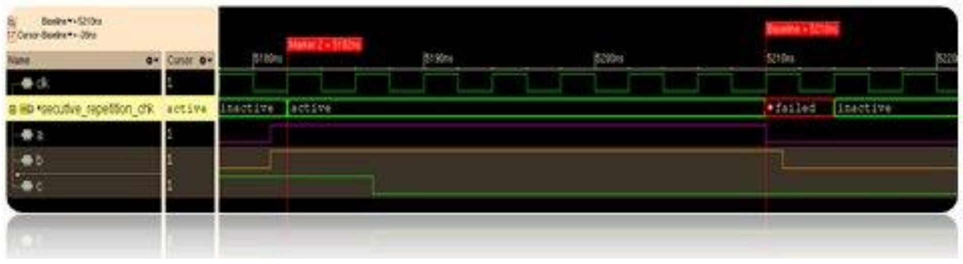
Snippet-1:



In the above snippet, at timestamp 254ns, assertion triggers when positive edge of signal “a” is detected, starting next clock cycle, signal “b” is continually high for 6 clock cycles and in the following clock cycle, signal “c” is asserted high so assertion finishes with pass status at 282ns.

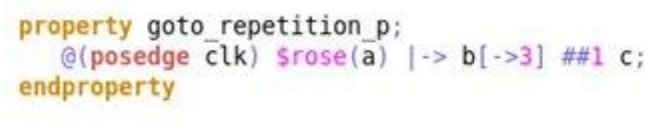
Assertion fails at the time stamp 330ns as signal “b” is asserted low in the cycle following the positive edge on signal “a”.

Snippet-2:



In the above snippet, assertion fails when positive edge of signal “a” is detected, followed by signal “b” asserted high continuously for 6 clock cycles and in the next clock cycle, signal “c” is not asserted high.

11. Go to repetition operator:



```
property goto_repetition_p;
   @(posedge clk) $rose(a) |-> b[->3] ##1 c;
endproperty
```

Syntax:

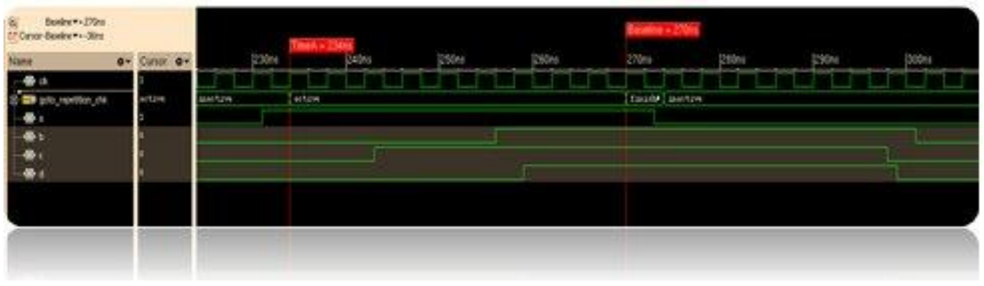
Signal_name1 [->n] ##1 signal_name2

It specifies that an expression matches the number of times specified, not necessarily in the consecutive clock cycles. The matches can be intermittent.

The key requirement of a "go to" repeat is that the last match of the expression (which is checked for repetition) should happen in the clock cycle before the end of the entire sequence match.

Property goto_repeatation_p checks, when positive edge on signal “a” is detected, checks for signal “b” to be high continuously or intermittently for 3 clock cycles and in the following cycle (after 3rd match of “b” high), signal “c “ is asserted high.

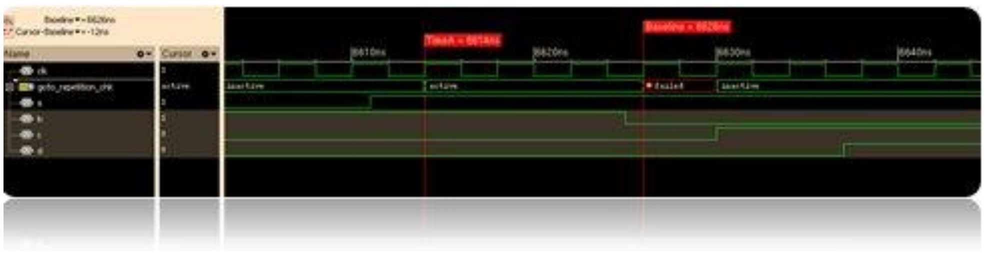
Snippet- 1: Passing case:



In the above snippet, at time stamp 234ns, assertion becomes active when the positive edge of signal “a” is detected. It waits for signal “b” to be high for 3 clock cycles (not necessarily consecutive cycles, in the snippet, it is 3 cycles before 270ns), followed by signal “c” asserted high in the immediate next cycle at 270ns, where assertion passes.

Note that if consecutive repetition operator had been used here, then the assertion would have failed at timestamp 234ns due to signal “b” asserted low, as it requires a consecutive match of signal “b” to be high upon posedge of signal “a”.

Snippet - 2: Failing case:



In the above snippet, at the time stamp 6614ns, assertion triggers upon a positive edge of signal “a”, starting same cycle signal “b” continually or intermediately high for 3 clock cycles, followed by signal “c”, which is not asserted high at 6626ns and hence, assertion fails.

Consider a sequence example which would also pass:

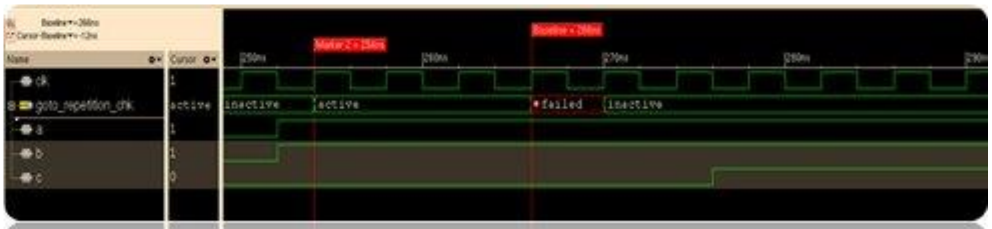
!a a !b !b b !b !b b !b !b !b b c

In this sequence, out of three repetitions of signal “b” to be high, 2 repetitions of signal “b” can be the intermediate, but the 3rd repetition of signal “b” must be followed by signal “c” (b c) for assertion to pass.

In short, with go to repetition operator, sequence “b c” must be followed for the last repetition of signal “b”.

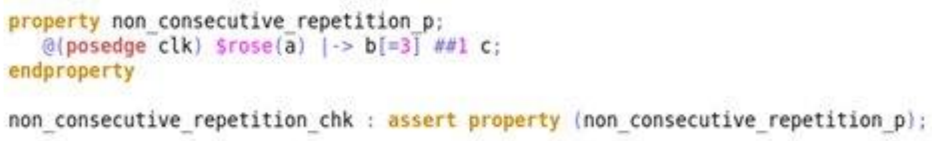
It fails if we have!b c (signal “b” is not followed by signal “c”) for the last repetition.

Snippet - 3 Failing case:



In the above snippet, when the positive edge of signal “a” is detected, signal “b” is continually (or can be intermediate) high for 3 clocks, but as signal “c” is not asserted high, following the last occurence of “b”, the assertion fails.

12. Non Consecutive repetition operator



```
property non_consecutive_repetition_p;
   @(posedge clk) $rose(a) |-> b[=3] ##1 c;
endproperty

non_consecutive_repetition_chk : assert property (non_consecutive_repetition_p);
```

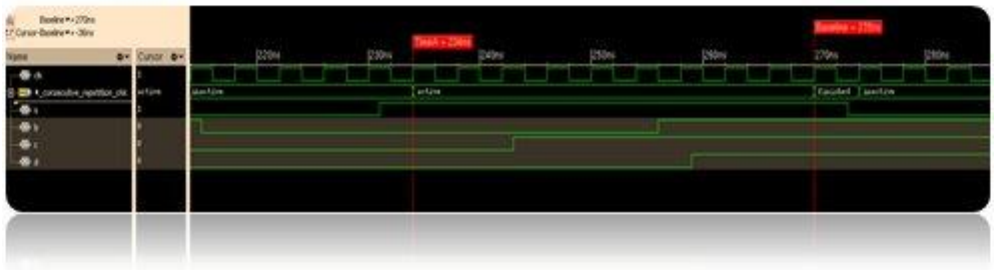
This is very similar to "go to" repetition except that it does not require the last match on the signal repetition to happen in the clock cycle before the end the entire sequence match.

Syntax:

Signal_name1 [=n] ##1 signal_name2

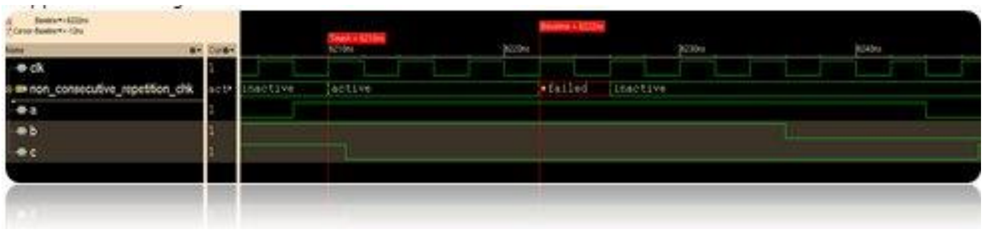
Property non_consecutive_repeatation_p check, when the positive edge of signal “a” is detected, check for signal “b” to be high continuously or intermittently for 3 clock cycles, followed by signal “c “ to be high in any cycle after that while signal “b” remains low.

Snippet - 1: Passing case:



In the above snippet, the assertion triggers when the positive edge of signal "a" is detected. It waits for signal "b" to be high for 3 clock cycles, followed by signal "c" asserted high in the next cycle. This behavior is exactly the same as "Go to repetition".

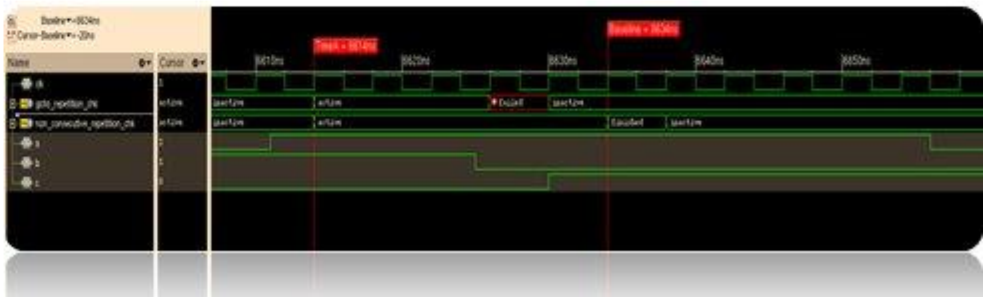
Snippet - 2: Failing case:



In the above snippet, the assertion triggers when the positive edge of signal "a" is detected. It waits for signal "b" to be high for 3 clock cycles, followed by signal "c" not asserted high and hence, the assertion fails. This behavior is the same as "Go to repetition".

Difference Between "Go to repetition" and "non-consecutive repetition":

Snippet:



Go to repetition:

In the above snippet, at the time stamp 6614ns, assertion becomes active as positive edge of signal "a" is detected. From the time stamp 6614ns onwards, signal "b" is asserted high for 3 clock cycles, but in the following cycle, as signal "c" is not asserted high, goto_repetition_chk fails.

This is because out of 3 repetitions of signal "b", the last repetition of signal "b" must be followed by signal "c".

Nonconsecutive repetition:

In the above snippet, Nonconsecutive repetition passes at the time stamp 6634ns as after 3rd repetition of signal “b”, signal “c” is asserted high and after 2 clock cycle, signal “b” is asserted low (!b c).

Similarly, below sequence would also pass for Nonconsecutive repetition.

!a a !b !b b !b !b b !b !b !b b !b !b !b c

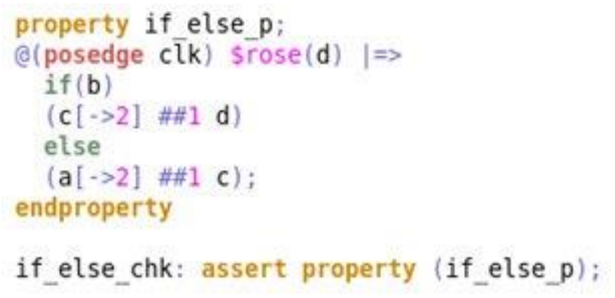
This is because after the (last) 3rd occurrence of “b”, occurrence of “c” happens without additional occurrences of “b”.

The below sequence would fail.

!a a !b !b b !b !b b !b !b !b b !b b !b c

This is because the last occurrence of “b” is followed by another occurrence of “b” before the occurrence of “c”.

13. If else inside property:

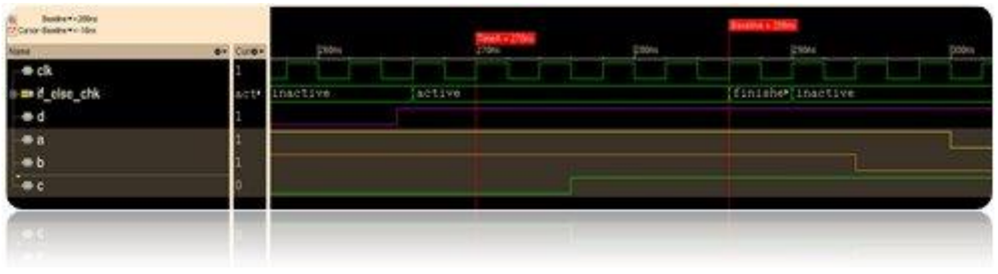


```
property if_else_p;
@(posedge clk) $rose(d) |=>
  if(b)
  (c[->2] ##1 d)
  else
  (a[->2] ##1 c);
endproperty

if_else_chk: assert property (if_else_p);
```

Property *if_else_p* checks,

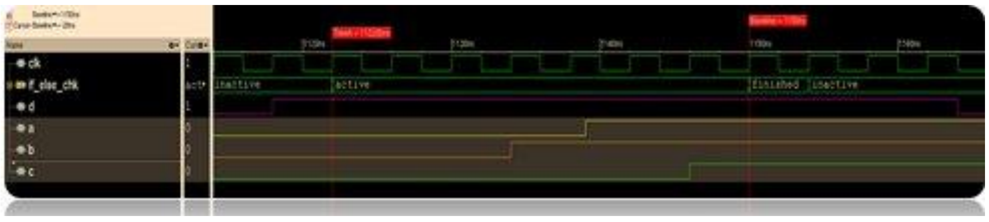
1. When signal “d” changes to 1, on next cycle, if signal “b” is true, then signal “c” should be high continuously or intermittently for 2 clock cycles, followed by high on signal “d“ in the next cycle.
2. When signal “d” changes to 1, on next cycle, if signal “b” is low, then signal “a” should be high continuously or intermittently for 2 clock cycles, followed by high on signal “c“ in the next cycle.

Snippet-1:



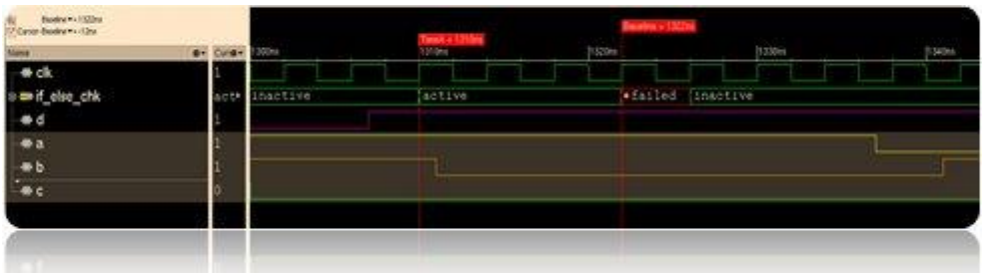
In the above snippet, assertion triggers when the positive edge of signal “d” is detected, in the next clock cycle, signal “b” is high, so assertion waits for signal “c” to be asserted high continuously or intermittently for 2 clock cycles, followed by high on signal “d“ in the next cycle. Assertion passes at the time stamp 286ns.

Snippet - 2:



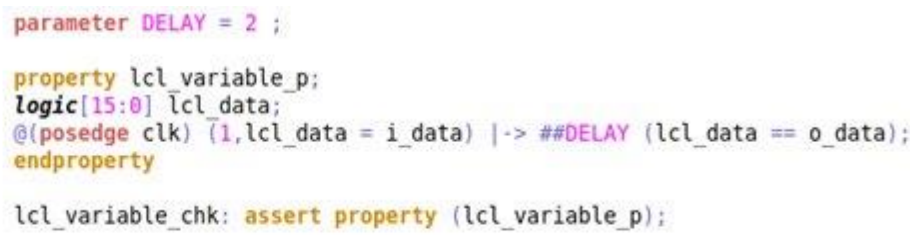
In the above snippet, assertion triggers when positive edge of signal “d” is detected, in the next clock cycle, signal “b” is low, so assertion waits for signal “a” to be asserted high continuously or intermittently for 2 clock cycles, followed by high on signal “c“ in the next cycle. Assertion passes at the time stamp 1150ns.

Snippet - 3:



In the above snippet, the assertion triggers when positive edge of signal “d” is detected. In the next clock cycle, signal “b” is low, so it waits for signal “a” to be asserted high continuously or intermittently for 2 clock cycles. It expects signal “c“ to be high in the following cycle, but as it is not asserted high, the assertion fails at the time stamp 1322ns.

14. Local variable inside property:

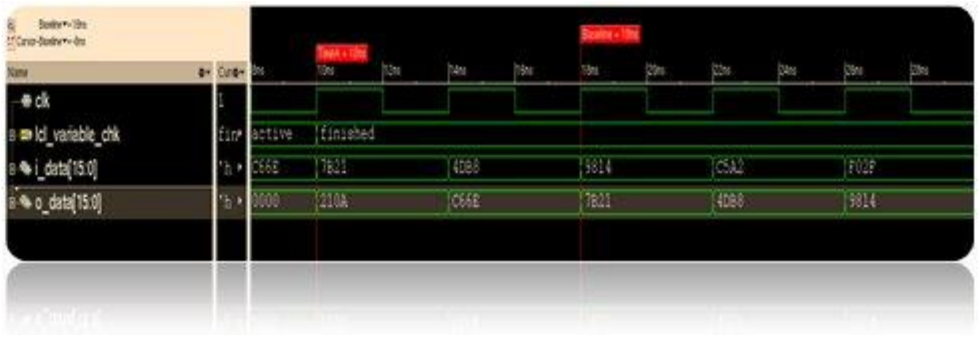


```
parameter DELAY = 2 ;

property lcl_variable_p;
logic[15:0] lcl_data;
@(posedge clk) (1,lcl_data = i_data) |-> ##DELAY (lcl_data == o_data);
endproperty

lcl_variable_chk: assert property (lcl_variable_p);
```

Property *lcl_variable_p* checks data on *i_data* to be reflected on o_data after two clock cycles.

Local variables can be used inside the property.

Snippet- 1:



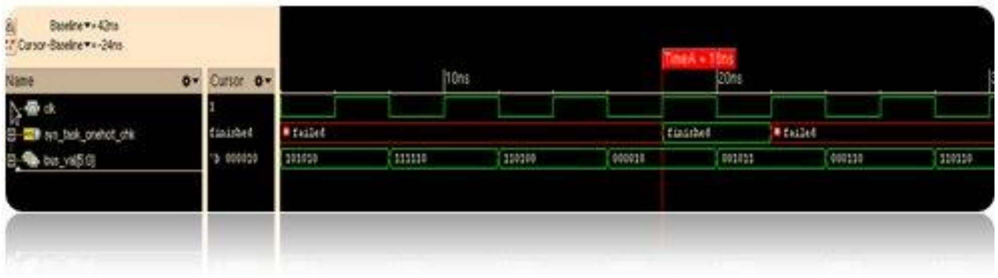
In the above snippet, the assertion passes on every clock cycle as data on i_data is reflected to o_data signal after 2 clock cycles.

15. $onehot property:

```
property sys_task_onehot_p;
  // number of ones must be one in given expression
  @(posedge clk) $onehot(bus_val);
endproperty
```

Property *one_hot_p* checks the number of ones must be one in “bus_val” for each cycle.

Snippet:



In the above snippet, at the time stamp 18ns, where “bus_val” has the number of ones = 1, assertion passes.

Assertion fails in each cycle where the number of ones is not equal to 1.

Similarly, $onehot(~bus_val) can be used to detect “WALK0” (5’b11110, 5’b10111, 5’b1111 etc) pattern on signal “bus_val”.

16. Throughout:

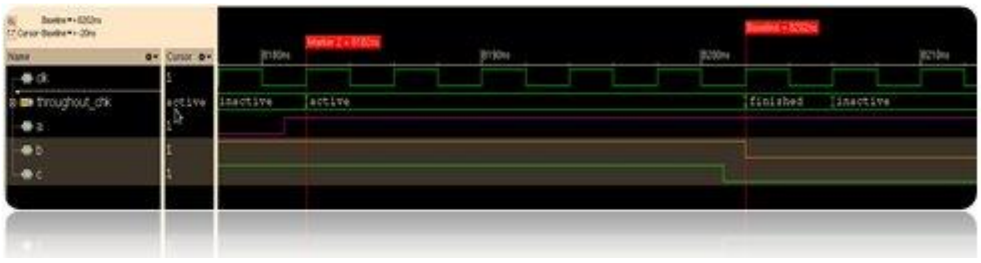
```
property throughout_p;
  @(posedge clk)  $rose(a) |=> (b throughout (!c[->1]));
endproperty

throughout_chk : assert property (throughout_p);
```

To check whether a certain condition holds true during the evaluation of the entire sequence, "throughout" operator is used.

Property throughout_p checks,

1. When the positive edge of signal “a” is detected, check signal “b” has to be high continuously until signal “c” goes low.
2. !c[->1] checks for the occurrence of c[*] ##1 !c, so assertion would stop when the signal “c” goes low.

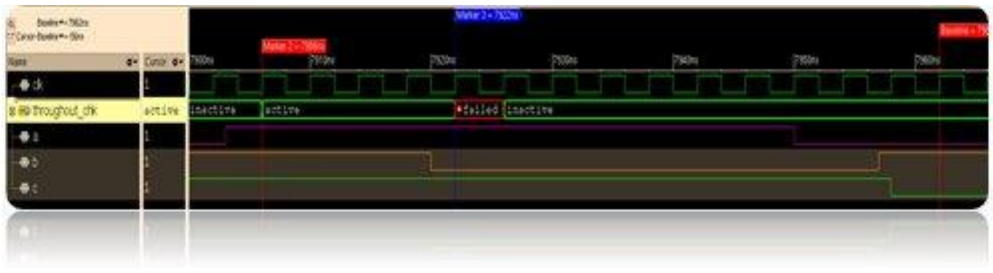
Snippet - 1:



In the above snippet, at time stamp 8182ns, assertion triggers as signal “a” changes to 1. From the next clock cycle onwards, it checks for signal “b” to be high till the time stamp 8202ns where signal “c” goes low.

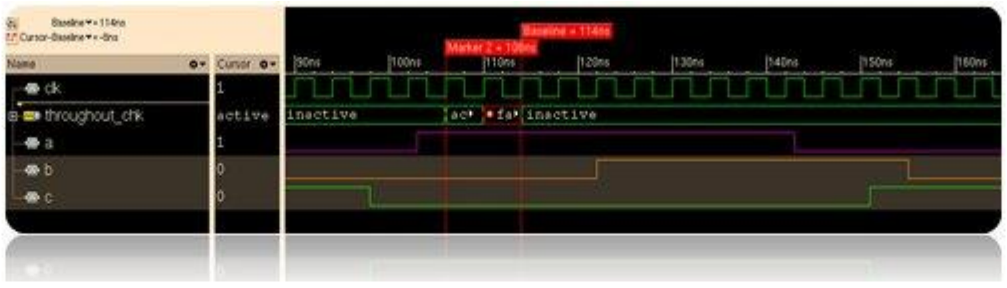
Note that once the assertion triggers, signal “b” is continuously asserted high, including the last cycle where signal “c” goes low and hence, assertion passes at the time stamp 8202ns.

Snippet - 2:



In the above snippet, assertion triggers when the positive edge of signal “a” is detected, signal “b” goes low before signal “c ” goes low and hence, the assertion fails at the time stamp 7922ns.

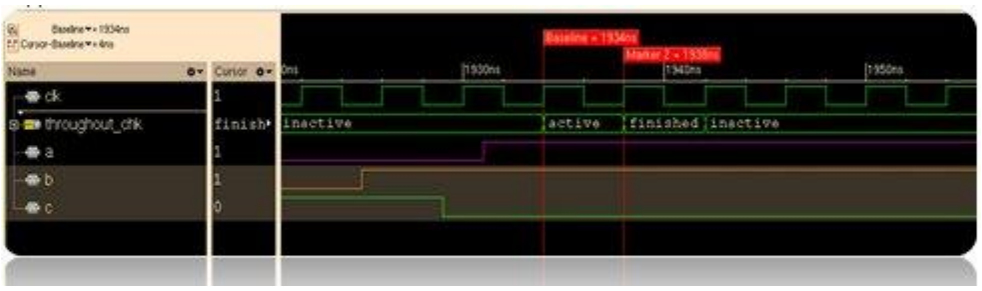
Snippet - 3:



Once throughout assertion triggers, it fails if checking condition is not true in the very first checking cycle.

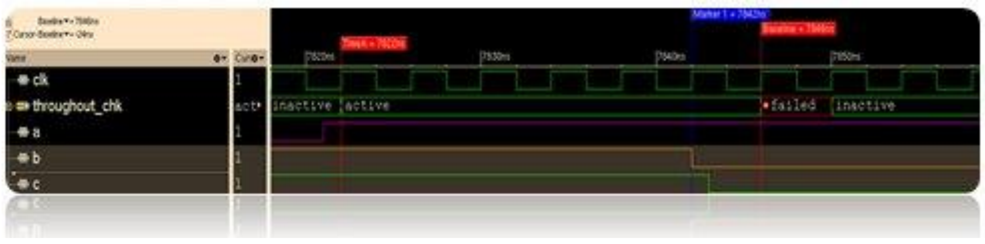
In the above snippet assertion triggers, once signal “a” changes to “1”, in the next clock cycle signal “b” is not asserted high and hence, the assertion fails even if signal “c” is not asserted high.

Snippet - 4:



In the above snippet, the assertion triggers once signal “a” changes to “1”.In the next clock cycle, signal “b” is asserted high and hence assertion passes even if signal “c” is not asserted high. This is because the first cycle checking condition is the last cycle of checking.

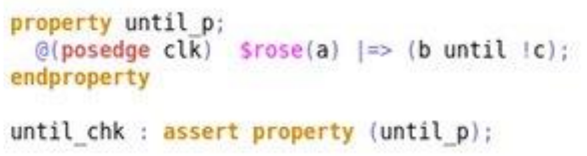
Snippet - 5 throughout assertion failure in last cycle:



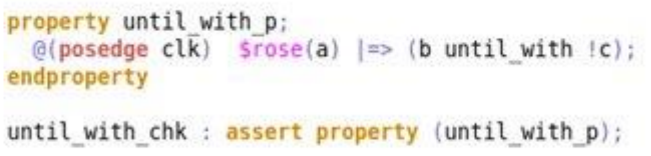
In the above snippet, assertion triggers when positive edge of signal “a” is detected, from the next clock cycle onwards, it checks for signal “b” to be high till the time stamp 7842ns, which is the second last cycle. In the following clock cycle, both signal “b” and signal “c” goes low and throughout the assertion fails even if signal “c” is not high. This is because the assertion expects signal “b” to be high in the last cycle where signal “c” goes low. Here, the assertion would have passed if signal “b” remained asserted for one more cycle.

To conclude, the “throughout” assertion fails if checking condition is not true in any cycle, including the last cycle (which is the end of checking boundary). Here it’s including the cycle where signal “c” goes low.

17. Until & Until_with:



```
property until_p;
  @(posedge clk)  $rose(a) |=> (b until !c);
endproperty

until_chk : assert property (until_p);
```



```
property until_with_p;
  @(posedge clk)  $rose(a) |=> (b until_with !c);
endproperty

until_with_chk : assert property (until_with_p);
```

Until construct is similar to throughout construct. Until construct further have overlap and non-overlap flavor.

Until construct is non overlapping form.

Until_with is overlapping form.

Non-overlapping form evaluates to true if property_expr1 evaluates to true at every clock tick until one clock tick before property_expr2 evaluates to true.

Overlapping form evaluates to true if property_expr1 evaluates to true at every clock tick until property_expr2 evaluates to true.

Syntax:
property_expr1 until property_expr2
property_expr1 until_with property_expr2

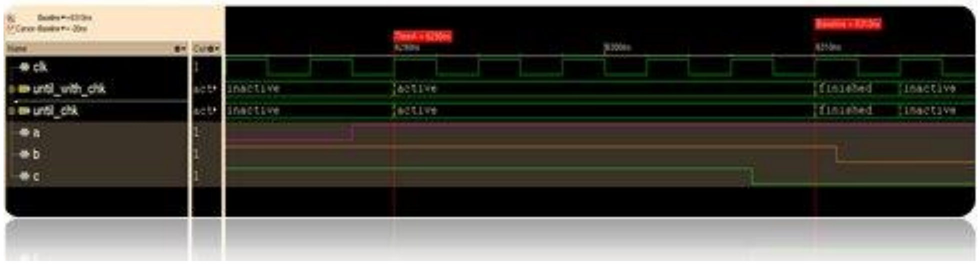
Property until_with checks for:

When the positive edge of signal "a" is detected, assertion checks for signal "b" to be high continuously until signal "c" goes low.

Property until checks,

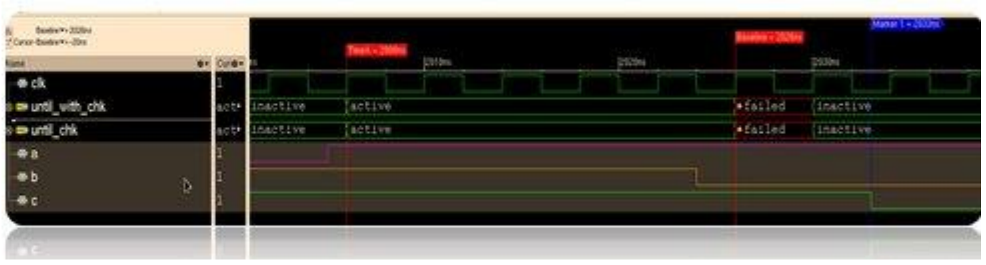
When the positive edge of signal "a" is detected, assertion checks for signal "b" to be high continuously until one cycle before signal "c" goes low.

Snippet-1:



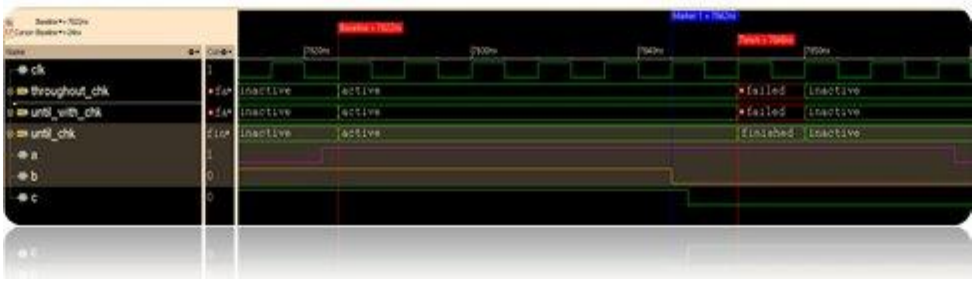
In the above snippet, at the time stamp 6290ns, assertion triggers due to signal “a” changes to 1. From the next clock cycle onwards, it checks for signal “b” to be high till the time stamp 8310ns, where signal “c” goes low.

Snippet-2:



In the above snippet, the assertion triggers when the positive edge of signal “a” is detected, signal “b” goes low before signal “c ” goes low and, the assertion fails at the time stamp 2026ns.

Difference between throughout/until/until_with:

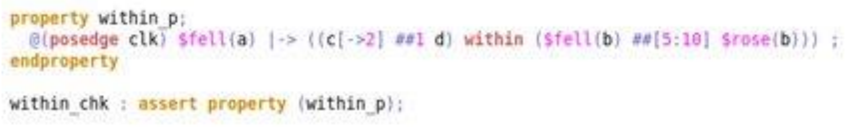


Until_with assertion works same as throughout assertion.

In the above snippet, assertion triggers when positive edge of signal “a” is detected. From the next clock cycle onwards, it checks for signal “b” to be high till the time stamp 7842ns, which is the second last cycle. In following clock cycle, both signal “b” and signal “c” goes low and throughout assertion fails even if signal “c” is not high. This is because the assertion expects signal “b” to be high in the last cycle where signal “c” goes low. Here throughout and until_with assertions would have passed if signal “b” remained asserted for one more cycle.

Until assertion passes when throughout and until_with assertion fails because until assertion is non overlapping form and it checks condition till one cycle before signal “c” goes low.

18. Within:



```
property within_p;
  @(posedge clk) $fell(a) |-> ((c[->2] ##1 d) within ($fell(b) ##[5:10] $rose(b))) ;
endproperty

within_chk : assert property (within_p);
```

Syntax:

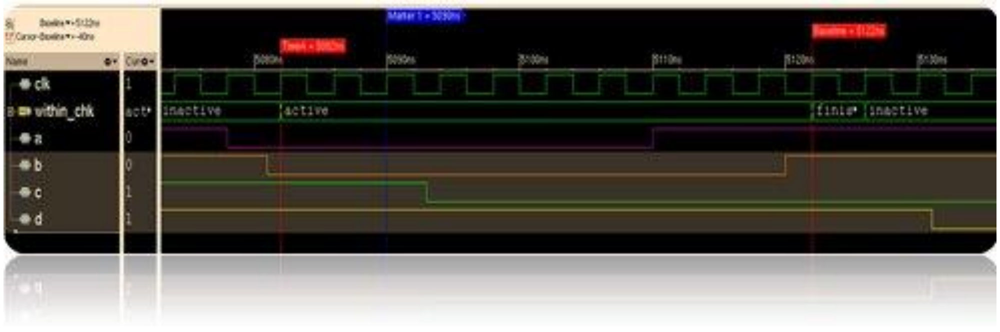
sequence1 within sequence 2;

This means that seq1 happens within the start and completion of seq2. The starting matching point of seq2 must happen before the starting matching point of seq1. The ending matching point of seq1 must happen before the ending matching point of seq2.

Property *within_p checks* for

sequence 1 happens within the start and completion of sequence 2.

Snippet Pass Case:



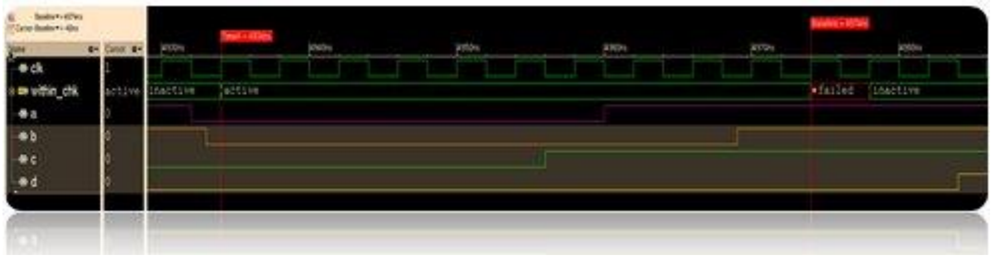
Sequence 1: c [->2] ##1 d, it checks for the intermediate assertion of signal “c” for the two clock cycles, in the following clock cycle signal “d” to be asserted high.

Sequence 2: $fell(b) ##[5:10] $rose(b), it checks negedge to posedge of signal “b” within latency of 5 to 10 clock cycles.

In the above snippet, the assertion triggers when the negative edge of signal “a” is detected, and remains active due to signal “b” changes to “0”. Sequence -1 gets completed at the time stamp 5090ns and sequence -2 gets completed at the time stamp 5122ns.

Assertion is finished at the time stamp 5122ns due to sequence 1 happened within the start and completion of sequence 2.

Snippet Fail case:



In the above snippet, assertion triggers when negative edge of signal “a” is detected and remains active due to signal “b” changes to “0”, assertion failed at the time stamp 4974ns as sequence 1 is not completed (signal “d” is not asserted high) within the start and completion of sequence 2.

Summary

By using appropriate SVA syntaxes explained in this paper, Design Verification engineers can easily implement any complex checker in any SV-based design verification project. This is irrespective of the design protocol, complexity, and verification methodology adopted for the project. Such assertions really add value to catch the design bugs early in time without waiting for 100% development of other DV components.