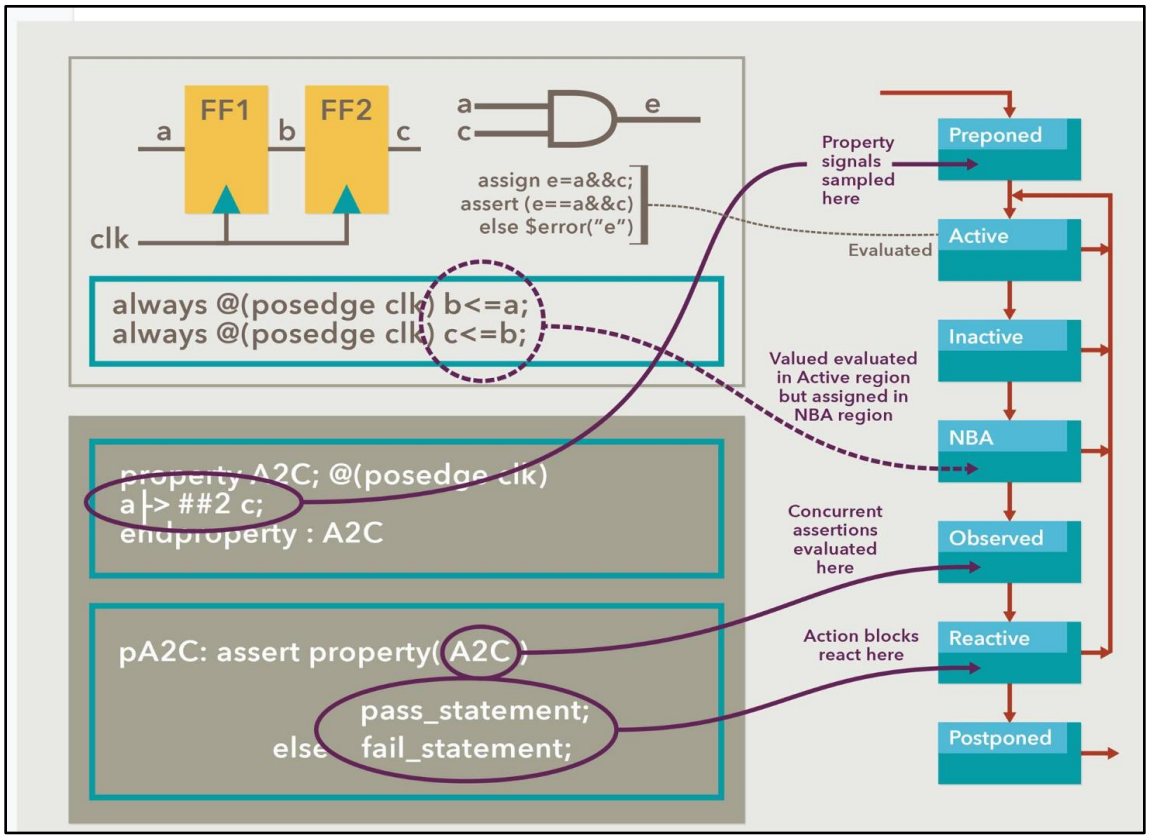
Understanding Assertion Processing Within a Time Step

1.0 INTRODUCTION

An interesting issue came up in the VerificationAcademy forum where an assertion leading clocking (LC) is one of those signals modified in the NBA or Reactive regions instead of the traditional *posedge clk*. For example, what is the difference between (*@(sig) b|-> c;*) and (*@(posedge clk) b|->c;*) when *sig* is a signal modified in the NBA or Observed, or Reactive region of the (*posedge clk*) time step? Another question related to regions: how is the **disable iff** handled when the disabling condition is updated in any of the SystemVerilog regions? Referring to the SystemVerilog evaluation regions, I must admit that using a signal modified in the Observed or Reactive region as the LC of an assertion or a disabling condition is not the common or recommended usage of assertions.



Evaluation regions within a time step.

This paper goes into detail about how these regions should be handled by a simulator as described in the SystemVerilog LRM; this should give you a better understanding of how assertions work. The paper makes two important points**: 1) Clocking events occurring in the same time step in different regions (*e.g. Active, NBA, Observed, and Reactive*) are indistinguishable due to the sampling of signals. 2) Upon entry or reentry into the Observed region, a scheduled assertion will be disabled with a disable iff(signal) if that signal was true before it entered that evaluation region.** Recommendations for handling this type of coding style are also provided.

2.0 Discussion

2.1 Assertion in a time step

In SystemVerilog, a time step is considered an activity at a time unit when an event or action that requires some processing occurs. For example, with a statement like (**always** #5 clk=!clk), at every 5 time units there is an action that the simulator must handle. *1800'2017: 4.5 SystemVerilog simulation reference algorithm* addresses the execution simulation, time slot, and regions where processing occurs within a time step.

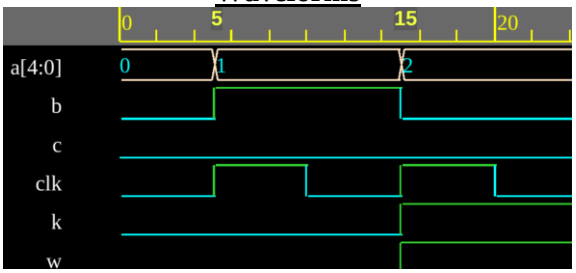
Consider the following code where we have three assertions each with a different leading clock occurring in the same time step (full code: https://www.edaplayground.com/x/qt9d).

```
bit [4:0] a;
bit clk, b=0, c, k, w;
initial forever #5 clk = !clk;
always @(posedge clk) begin  a <= a + 1; b<=!b; c <= 0;  end  // a, b, c modified in the NBA region

ap_1: assert property (@(posedge clk) b |-> c) //  Leading clock (LC) in Active region
   else begin
    k=!k;   // k is modified in the Reactive region
    $display("%t ap_1 @posedge clk FAIL sampled b= %b sampled a=%d Reactive b=%b Reactive a=%b Reactive
k=%b",   $realtime, $sampled(b), $sampled(a), b, a, k);
   end

ap_2: assert property (@(a) b |-> c) // LC is in NBA region of same time step.
   else $display("%t ap_2 @aFAIL sampled b= %b a=%d $sampled(a)=%d", $realtime, $sampled(b),a,  $sampled(a));

ap_3: assert property (@(posedge k) a < 4 ##0 b |-> c) // LC in Reactive region  of same time step
   else begin
    w=1;
    $display("%t ap_3 @posedge clk FAIL sampled(k)= %b k=%b sampled(a)=%d   a=%d",
    $realtime, $sampled(k), k, $sampled(a), a);
   end
```

Simulation results for assertions at 15 ns

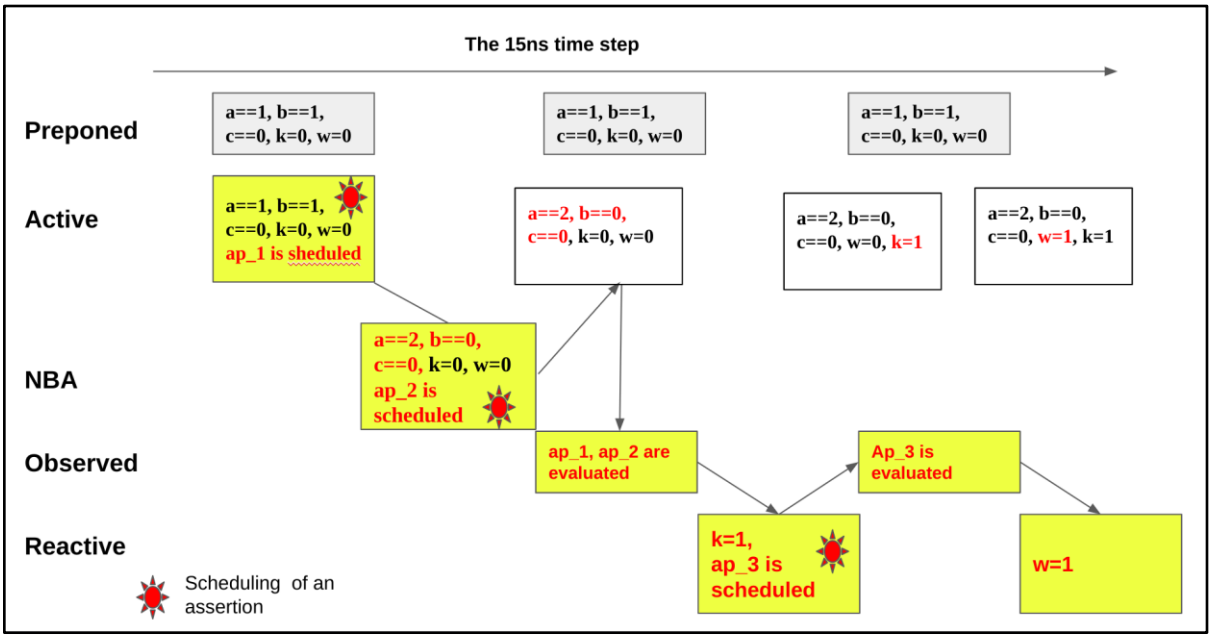
```
# 15 ap_1 @posedge clk FAIL sampled b= 1
      sampled a= 1 Reactive b=0 Reactive a=00010
      Reactive k=1
# 15 ap_2 @aFAIL sampled b= 1 a= 2 $sampled(a)= 1
# 15 ap_3 @posedge clk FAIL sampled(k)= 0
      k=1 sampled(a)= 1   a= 2
```

Waveforms



Let's analyze the processing of these assertions within the 15ns time step. The code is displayed without the ***$display*** statements for brevity.

```
// test code
 int a=0; bit b=0, c=0, k=0, w=0;
always @(posedge clk) begin  a <= a + 1; b<=!b; c <= 0;  end
ap_1: assert property (@(posedge clk) b |-> c) else k=!k;  // Leading clock (LC) in Active region
ap_2: assert property (@(a) b |-> c); // LC is in NBA region of same time step.
ap_3: assert property (@(posedge k) a < 4 |-> c) else w=1; // LC in Reactive region  of same time step
```



Evaluation regions of one time step as it traverses those regions

1) **ACTIVE Region:**

   always @(**posedge** clk) **begin** a <= a + 1; b<=!b; c <= 0; **end**

   * The *clk* change starts the evaluations for that time step. If this is a posedge of clk, the Preponed values of the variables used in the assertions are saved. In that case, that would be the "*a, b, c, k, w*".
   * The simulator schedules the future values of "*a, b, c*" to be updated in the NBA region.
   * The simulator detects the *ap_1* assertion because the (*posedge clk*) is a leading clock and schedules *ap_1* to be processed in the Observed region.

2) **NBA Region**

   * Values of *a, b, c* are updated to *a==2, b==0, c==0.*
   * Since "*a*" has a change in value, then the simulator detects the *@(a)* which is the LC of *ap_2*, and it schedules *ap_2* to be processed in the Observed region.

3) **Observed Region**

   * *ap_1 and ap_2* are evaluated because they were previously scheduled. The assertions use the Preponed values of the variables (i.e., *a==1, b==1, c==0).*
   * *ap_1* has an action block, but its evaluation is delayed to the Reactive region.

4) **Reactive Region**
   * *ap_1* action block is evaluated and *k* is changed to 1.
   $display in the action block:
   # 15 ap_1 @posedge clk FAIL sampled b= 1 sampled a= 1 Reactive b=0 Reactive a=00010 Reactive k=1
   # 15 ap_2 @aFAIL sampled b= 1 a= 2 $sampled(a)= 1

   * Since *k* has changed in value, the *@(k)* is the LC of a*p_3* and a*p_3* is scheduled to be processed in the Observed region in the loopback.

5) **Observed Region, loopback**
   * *ap_3* is evaluated.
   * *ap_3* has an action block, but its evaluation is scheduled for the Reactive region.

6) **Reactive Region, loopback**
   * *ap_3* action block is evaluated and *w* is changed to 1.
   $display in the action block:
   # 15 ap_3 @posedge clk FAIL sampled(k)= 0 k=1 sampled(a)= 1 a= 2

2.2 Asynchronous disable iff

Upon entry or reentry into the Observed region, a scheduled assertion will be disabled with a disable iff(signal) if that signal was true before it entered that Observed evaluation region. If the disabling condition of an assertion is not already set when it enters the Observed region, then a scheduled assertion cannot disable itself. A function call or action block setting the disable condition will not affect an ongoing assertion evaluation. However, once the disable signal is set and another assertion is evaluated in the loopback then that other assertion will be disabled. The following examines different cases all occurring in the same time step.

Source of the disabling condition	Assertion enabling/disabling
Disabling condition is set in the Active of NBA regions before entering the Observed region	Assertion **will be** disabled in that time step
Disable signal is set in the Observed region	Scheduled assertions **will NOT** be disabled https://www.edaplayground.com/x/Vyfr ap_1 is **NOT** disabled at that time step
Disable signal is set in the action block region	Scheduled assertions were already executed https://www.edaplayground.com/x/LmqC ap_1 and ap_3 **are NOT** disabled at that time step
The disable signal is set before a reentry into the Observed region because of loopback.	Scheduled assertions **will be** disabled https://www.edaplayground.com/x/Vyfr ap_3 **is** disabled at that time step

3.0 Conclusions and recommendations

SystemVerilog is a very powerful and flexible design and verification language, thus allowing users to come up with very creative approaches to applying the language; sometimes way too creative! These undisciplined applications of SVG are minefield guarantees to result in mistakes that are tricky to debug and may most definitely not synthesize. This paper covered examples of non-conventional applications such as using signals updated in regions outside the Active or NBA as assertion leading clocks and resets. This paper explained how assertions are processed within a time step so that users appreciate that good design disciplines are mandatory. Below is a summary of the processing of assertions within a time step and guidelines necessary for a better discipline in applying SVG with SVA.

3.1 Time step processing summary

SytemVerilog uses regions within a time step to correctly simulate concurrency because the simulation program is sequential. **Clocking events occurring in the same time step in different regions (e.g., Active, NBA, Observed, Reactive) are indistinguishable in concurrent assertions due to the sampling of signals**. In other words, if *@(a)* and *@(**posedge** clk)* occur in any order in the same time step, then ***assert property** (@(a) x ##0 @(**posedge** clk) y)* and ***assert property(@(posedge** clk) x ##0 @(a) y)* are indistinguishable. However, this is not to be relied on because the design would have to guarantee that both events do always happen in the same time step; someone could change the design without respecting this assumption.

Upon entry or reentry into the Observed region, a scheduled assertion will be disabled with a disable iff(signal) if that signal was true before it entered that Observed evaluation region. Be aware of disabling conditions set in the Observed or Reactive regions as they will not take effect in the same time step.

3.2 Recommendations

3.2.1 Signals to use as leading clock of assertions

Avoid using signals assigned outside the Active or NBA regions as leading clocks or resets

This can be misleading and often may not meet the requirements For example, consider this variation of *ap_1* and *ap_2* assertions listed above:

*ap_1b: **assert property** (@(**posedge** clk) b |-> ##1 c); // Preferred approach for LC*

*ap_2b: **assert property** (@(a) b |-> ##1 c); // May not be synthesizable*

Here, *ap_1b* is not the same as *ap_2b* because *ap_2b* is equivalent to:

*ap_2b_equiv: **assert property** (@(a) b |-> @(a) ##1 c); //c is evaluated at changes in "a"*

If *@(a)* does not change at every *posedge clk*, then these two assertions behave differently. The intent may be that "*c*" is evaluated at the *posedge of clk*.

If you do choose a signal for the LC of an assertion, then within the property adjust the clocking events to the one(s) used to modify the variables of that property; For example:

for ap_2b, write instead:

*ap_2b_better: **assert prop**erty(@(a) 1 ##0 @(**posedge** clk) b |-> ##1 c); // multiclocking*

*T*he two assertions *ap2b* and *ap_2b_better* are different; make sure you meet the requirements.

3.2.3 Signals to use for disabling assertions

3.2.3.1 Using signals

For synchronous disabling of assertions, use signals updated in the NBA *(or loopback from the NBA)* region; the loopback is into the Active region. For example:

```
always @(posedge clk) if(sig) reset<=1'b1; else reset <= 1'b0;  // Do this for the reset
ap_do: assert property( @(posedge clk) disable iff(reset) prop);
```

A corollary to this, do not use signals updated in assertions or sequences. Two reasons for this recommendation:
1) Because a reset issued from an assertion is not synthesizable.
2) Because a reset issued from an assertion will NOT take effect in the same time step; this may be unexpected.