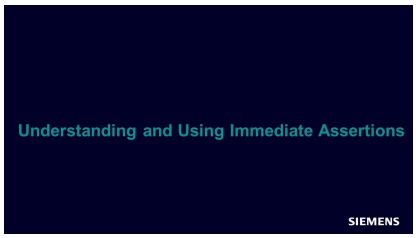
Understanding and Using Immediate Assertions



Immediate assertions are typically used to verify that expressions are within their required bounds, such as no overreach of the value of a counter or an illegal condition such a write without an enable. The action block is typically used for debug to display more information as to the cause of the error. However, immediate assertions can also be used to modify testbench variables for use in monitors or in other assertions, or to change the course of a testbench flow.

Assertions

Ben Cohen Last Updated Dec 2022

Assertions, Block Level, SystemVerilog,
December 2022 - Volume 18 Issue 3, Verification Horizons, Intermediate

Mark as completed

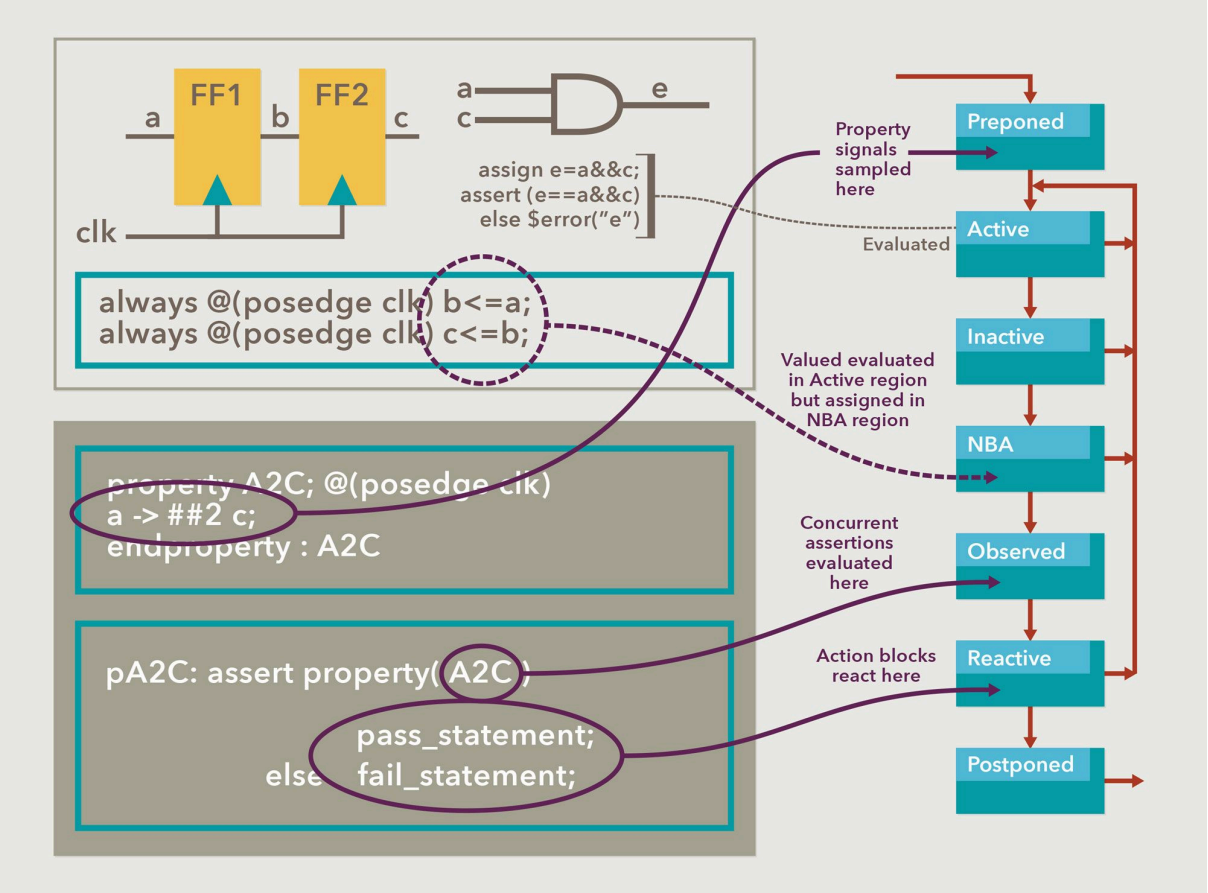
INTRODUCTION

Immediate assertions are typically used to verify that expressions are within their required bounds, such as no overreach of the value of a counter or an illegal condition such a write without an enable. The action block is typically used for debug to display more information as to the cause of the error. However, immediate assertions can also be used to modify testbench variables for use in monitors or in other assertions, or to change the course of a testbench flow. This article explains the immediate assertion modes, their restrictions, and it provides guidelines into their uses.

IMMEDIATE ASSERTIONS AND SCHEDULED REGIONS

1800:4.4 Stratified event scheduler defines the regions within a time slot. The figure below is a graphical view of the relationship of constructs and the regions in which region they are processed. Note that these regions do iterate from the Active region to the Reactive region on an as-needed basis.

Figure 1:



Immediate assertions can be inserted in the following constructs: **always/always_ff, always_comb, function, task,** and as a *module_common_item*[1] within a **module**, a **checker**, and a **program**. All immediate assertions execute in the location where and when they are seen. The action block execution region depends upon the immediate assertion mode. These modes are discussed in the following subsections.

SIMPLE IMMEDIATE

A simple immediate assertion *(without the **#0** or the **final**)* executes immediately after the evaluation of the assertion expression in the region it is called, thus being efficient. Constructs where the processing region is iterated once are ideal candidates for simple immediate assertions because the immediate assertion produces results only once in that iteration. If there were multiple iterations of that construct where the variables can change values at each iteration, then each iteration where the immediate assertions are processed may produce different results. Consider this example:

Figure 2:

```
always_ff @(posedge clk_tb) begin a_tb <= a1; b_tb <= b1; end
//a_tb, b_tb change in NBA
always_comb A=a_tb ^ b_tb; // A evaluated once in the Active region
//  in the iteration from NBA to Active upon a change of a_tb, b_tb
always_ff @(posedge clk_rtl) r_rtl <= c1;
// r_rtl changes in the NBA region
// clk_rtl and clk_tb are synchronous in the same time step
always_comb am_B: assert (A ^ r_rtl)
        $display("%t  line 44 B PASS    A=%b r_rtl=%b", $realtime, A, r_rtl);
  else  $display("%t  line 45 B FAIL    A=%b r_rtl=%b", $realtime, A, r_rtl);
```

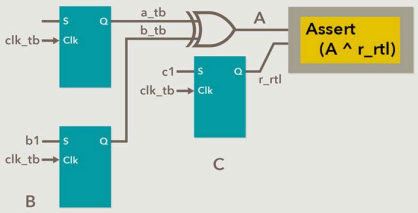
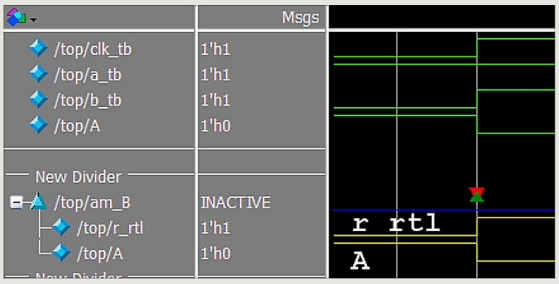


Figure 3:

```
Immediate assertion with (A ^ r_rtl) can be
evaluated twice in the Active region
* Once because of a change in the current value of r_rtl
  in the NBA
* Once more because of a change in the current value
  of A evaluated in the Active region
// Simulation results
 4320  line 45 B FAIL   A=1 r_rtl=1
 4320  line 44 B PASS   A=0 r_rtl=1
```



Simple immediate assertions recommendation:

Use simple immediate assertions in the following constructs:

- **always_ff** or **always** sensitive to a clocking event.
- **task** initiated from a clocked statement.
- **function**

Examples:

Figure 4:

```
// function with NO side effects
function void f_xy(bit x, y);
    am_xy: assert(x && y);
endfunction: f_xy

// Called from always_ff
task automatic t();
   a1 <= d;  b1 <= e;
   am_ab_OK: assert(a1 & !b1);
   f_xy(a1, !b1);
endtask
```

```
always_ff @(posedge clk) begin
      fork t(); join_none
      a <= d;  b <= e;
      am_ab_OK: assert(a & !b);
      f_xy(a, !b);
end
```

DEFERRED ASSERTIONS

As demonstrated in section 1.2, an assertion in an **always_comb** can experience multiple iterations as the simulator navigates through the event scheduling regions. To overcome the false reporting of immediate assertions during these event scheduling iterations, the reporting of the deferred immediate assertion action block (*implicit to report an error*) or explicit (*user-defined reporting*) is delayed until there are no more

allowed iterations in the allowed evaluation regions. This provides some level of protection against unintended multiple executions on transient or "glitch" values. There are two different kinds of deferred assertions: *observed deferred immediate assertions* using the **assert #0**, and *final deferred immediate assertions* using the **assert final**. A deferred assertion (#0, **final**) may be used as a *module_common_item*.

Observed deferred immediate assertions (assert #0)

Observed deferred immediate assertions address glitches caused by combinational logic (e.g., evaluations in the Active region using variables assigned in the other regions (Active, NBA, Reactive), thus causing multiple evaluation iterations. In a simplified view, results of each iteration of an *observed deferred immediate assertion* are put into a queue; old results are flush. This process repeats until there are no more iterations from the Active region to the post-Observed region, i.e., the combination logic finally settled. The action block is scheduled in the Reactive region.

Action block Restrictions: Action blocks may only contain a single subroutine call and may not make any direct assignment to variables. However, the subroutine may do the assignment.

For example:

```
function void f2(bit w); m2=w;  endfunction // variable m2 is assigned a value
always_comb  am_ACFbk3: assert #0 (b && k) f2(1); else f2(0);  // OK
always_comb  am_BAD: assert #0 (b && k) m=1; m=0;  // ERROR
```

Observed deferred assertions recommendation: Write observed deferred immediate assertions in **always_comb** constructs or in *module_common_items* (which are equivalent to **always_comb**) when all of the following conditions are met:

1. The immediate assertion uses variables that emulate RTL combinational logic
2. When variables used in the immediate assertion are not driven by any action block
3. When variables used in the immediate assertion are not driven in a **program** block

For example:

```
always_ff @(posedge clk) begin a <= w; b<= z; end
always_comb  a_f=!a;
always_comb  am_ba_f: assert#0 (b && a_f);
```

Final deferred immediate assertions, assert final:

There are two situations where variables can be updated in the Reactive region. These include:

1. Statements and constructs within a **program** block that are sensitive to changes (e.g., update events).
2. Variables assigned in the action block of an observed immediate differed or a concurrent assertion.

For example:

```
ap_gh: assert property(@(posedge clk) g |-> ##1 h) k=1;  else k=0;
```

1800-2017: 16.4.1 **Deferred assertion reporting:** *"Note that if code in the Reactive region modifies signals and causes another pass to the Active region to occur, this still may create glitching behavior in observed deferred assertions, as the new passage in the Active region may re-execute some of the deferred assertions with different reported results. In general, observed deferred assertions prevent glitches due to order of procedural execution, but do not prevent glitches caused by execution loops between regions that the assignments from the Reactive region may cause".*

To avoid the vulnerability to the potential glitch behavior described for observed deferred assertions the final deferred immediate assertion schedules the action block subroutine in the Postponed region.

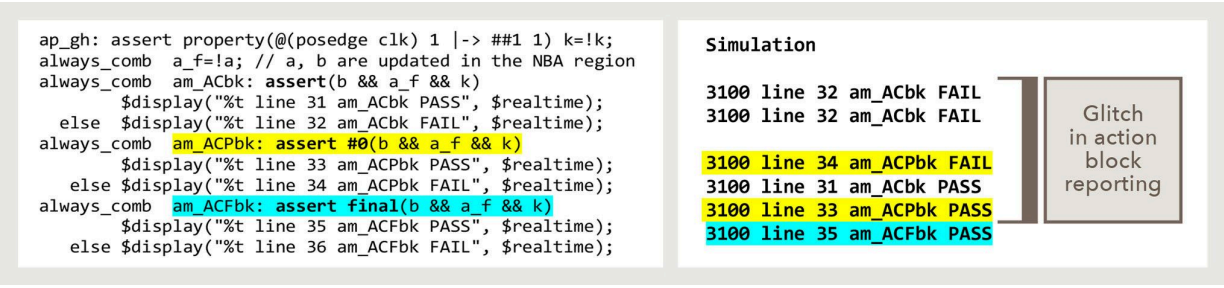
Action block Restrictions: Action blocks of deferred assertions (observed or final) may only contain a single subroutine call. In addition, in deferred final assertions it is illegal to write values to any net or variable or to schedule an event in any previous region within the current time slot.

For example:

```
function void f2(bit w); m2=w;  endfunction
always_comb  am_BAD:  assert final (a && r) f2(1); else f2(0);  // ERROR
always_comb  am_BAD2: assert final (a && r) m2=1; else m2=0;  // ERROR
always_comb  am_OK: assert final (a && r) else $display("k=%t, k=%t", b, k);
```

Example: www.edaplayground.com/x/nbA9

Figure 5:



Final deferred immediate assertions recommendation: Use *final deferred immediate assertions* when variables used in the immediate assertion are either driven in any action block or driven in a **program** block.

COMMENTS

- The use of the **program** block has been discouraged by many experts
- The discussion addressed on the immediate **assert** statement also applies to the **assume** and **cover** statements. *1800: 16.3 Immediate assertions* provides the syntax for the immediate_assertion_statement. Note that the cover statement does not have an action block, but supports a simple statement when the expression is true.
- Assertions and action blocks can be used very creatively, and the more creative they are the more difficult the understanding of the verification model.
- If action blocks are used for purposes other than debug, try to limit the usage to a monitoring action.
- If you must have variables driven by the immediate assertion action blocks, avoid using those variables into other immediate assertions; though legal, it can become a nightmare to debug.

END NOTE

1. A *module_common_item* is a statement or construct that can be used in the context of a module definition.

Back to Top

December 2022 Articles

IoT, and military/aerospace, while also maintaining backward compatibility with all previous generations of PCIe.

Data Integrity through TLP Encryption in PCI Express®

Verificati... Article

In today's modern era, there has been an increase in the relentless push for higher-speed data transfers along with the security of the data to be transferred. The need for protecting the confidentiality of data is increasing day by day. The need to process higher data bandwidth is catered with the increased speed of PCIe®.

Missing Your Buttons and Menus? Create Your Own

Debug Article

Increasing efficiency while debugging involves having access to the right tools, permissions, and the ability to use the debug tool to its fullest. We know how engineers spend their time debugging rather than verifying. Provide them with all the features of a debugging tool, and they would not have exited the tool while enjoying debugging.

Application of AI/ML to Optimize Fault Simulation Coverage

Machine ... Article

Fault simulation replicates the typical sources of failures and analyzes them using the leading EDA tool. Testing is structured using fault simulator techniques to gain insight into fault effects and ensure defect coverage and quality testing are achieved. This article is based on standard fault Simulation and standard fault simulation techniques used in industry and applies intelligence and learning algorithms to reduce simulation cycles.

Understanding and Using Immediate Assertions

Assertions Article

Immediate assertions are typically used to verify that expressions are within their required bounds, such as no overreach of the value of a counter or an illegal condition such a write without an enable. The action block is typically used for debug to display more information as to the cause of the error. However, immediate assertions can also be used to modify testbench variables for use in monitors or in other assertions, or to change the course of a testbench flow.

View all articles ›

Worldwide offices