A common incorrect usage of immediate assertions is with randomization of variables; this is because an assert control that turns off all assertions will prevent the randomization (see 4.2.4). For example:

```
import uvm_pkg::*; `include "uvm_macros.svh"
class class_seq;
  rand int addr;
  task main();
    a_BAD_STYLE: assert (randomize(addr)) // 👎👎
      else $fatal(1, "Randomization of var failed: conflicting constraints");
    a_GOOD_STYLE: if(!randomize(addr)) // ✓✓
          `uvm_error("run_phase", "seq randomization failure");
  endtask : main
endclass : class_seq
```

Will not randomize if **$assertoff** is used on the design.

Could also use instead
`$error("seq randomization failure");`

4.6.2 Deferred assertions

Simple assertions work well when the signals used in the assertions represent registers (i.e., assigned in clocked blocks However, immediate assertions may involve signals that can be evaluated multiple times in the same time stamp because they are defined in **always_comb** combinational blocks or **assign** statements. This may produce unnecessary noise in simulation reports because of possible multiple revisits of the assertions in the same time unit. This is because the order of the assignments is not guaranteed. Thus, the assertion may evaluate to false in one evaluation loop, but evaluates to true in another evaluation loop.

To prevent race conditions between the assertion and the value being tested, IEEE 1800-2009 introduced the concept of deferred assertions, which are a kind of immediate assertion. *1] They can be used to suppress false reports that occur due to glitching activities on combinational inputs. Since deferred assertions are a subset of immediate assertions, the term deferred assertion (often used for brevity) is equivalent to the term deferred immediate assertion. The term simple immediate assertion refers to an immediate assertion that is not deferred. [1] A deferred assertion is similar to a simple immediate assertion, but with the following key differences:*

- *Syntax: Deferred assertions use* **#0** *(for an Observed deferred assertion) or* **final** *(for a final deferred assertion in* the Postponed region*) after the verification directive.*[43]
- *Deferral: Reporting is delayed rather than being reported immediately.*
- *Action block limitations: Action blocks may only contain a single subroutine call.*
- *Use outside procedures: A deferred assertion may be used as a module_common_item.*

Example:

```
module priority_encoder (); ///ch4/4.6/priority_encoder.sv
    bit[3:0] sel=4'b0010;
    bit enb, ready;
    bit[1:0] code;
    bit go;
    always_comb begin : ac1
        $display("ac1a: enb=%b, go=%b, ready=%b", enb, go, ready);
        go=enb;
        $display("ac1b: enb=%b, go=%b, ready=%b", enb, go, ready);
    end : ac1
```

[43] See **4.1 Systemverilog scheduling semantics for Assertions**

```
always_comb   begin : p1
        if (go || ready) begin : enb_on
            if (sel[0]) code = "00";
            else if (sel[1]) code = "01";
            else if (sel[2]) code = "10";
            else if (sel[3]) code = "11";
            else             code = 3'bxx;
        end : enb_on
        else
            code = "00";
    end : p1

    always_comb begin : b1
        a1: if ((go || ready ) && sel[1] ) assert (code =="01") else
                $error ("a1:     priority encoding error");  // Line 27
        a1_dfr: if ((go || ready ) && sel[1] ) assert #0 (code =="01") else
                $error ("a1_dfr: priority encoding error");    // Line 29
    end : b1

    initial begin : it1
       #2 enb=1; $display("it1a: enb=%b, go=%b, ready=%b", enb, go, ready);
       #0 enb=0; $display("it1b: enb=%b, go=%b, ready=%b", enb, go, ready);
       ready=1; $display("it1c: enb=%b, go=%b, ready=%b", enb, go, ready);
       #5 ready=0;
    end : it1
endmodule : priority_encoder
```

Immediate assertions

```
ac1a: enb=1, go=0, ready=0
# ac1b: enb=1, go=1, ready=0
# ** Error: a1:     priority encoding error
#    Time: 2 ns  Scope: priority_encoder.b1.a1 File: priority_encoder.sv Line: 27
# ** Error: a1:     priority encoding error
#    Time: 2 ns  Scope: priority_encoder.b1.a1 File: priority_encoder.sv Line: 27
# it1b: enb=0, go=1, ready=0
# it1c: enb=0, go=1, ready=1
# ** Error: a1:     priority encoding error
#    Time: 2 ns  Scope: priority_encoder.b1.a1 File: priority_encoder.sv Line: 27
# ac1a: enb=0, go=1, ready=1
# ac1b: enb=0, go=0, ready=1
# ** Error: a1:     priority encoding error
#    Time: 2 ns  Scope: priority_encoder.b1.a1 File: priority_encoder.sv Line: 27

# ** Error: a1_dfr: priority encoding error
#    Time: 2 ns  Scope: priority_encoder.b1.a1_dfr File: priority_encoder.sv Line: 29
# ac1a: enb=0, go=0, ready=0
# ac1b: enb=0, go=0, ready=0
```

Simple immediate assertion report

deferred immediate assertion report

Figure 4.6.2-1 Model for a priority encoder with an enable

Deferred assertions were originally designed to be a glitch-protected variant of immediate assertions. But since deferred assertions mature in the Observed region (see 4.1), which is an iterative region, it is still possible for testbenches to change values after their results are reported. Thus, a single deferred assertion may execute and mature multiple times in a single time step, and all executions except the final are logically glitches, reports of results on intermediate combinational values. To solve this issue, a new subclass of deferred assertions, *final assertions* was introduced, as that would make them truly glitch-free. In the case of a final deferred assertion the action block subroutine is called in the Postponed region. For example,

```
   assert final (out_data==mem[cache_addr] else $display(“Error in fetch from cache”);
```

Deferred `assume and cover` and `final assume and cover` statements are also available. These are scheduled in the same way as the deferred and final assertions described above. A deferred `cover` is useful to avoid crediting tests for covering a condition that is only met in passing by glitched values.

4.6.2.1 Deferred assertion reporting

[1] When a deferred assertion declared with `assert #0` *passes or fails, the action block is not executed immediately. Instead, the action block subroutine call (or* `$error`*, if an* `assert` *or* `assume` *fails and no action_block is present) and the current values of its input arguments are placed in a deferred assertion report queue associated with the currently executing process. Such a call is said to be a pending assertion report.* See Figure 4.6.2.1 for a view of the queue for the deferred action block.

If a deferred assertion flush point (see below and Appendix B) is reached in a process, its deferred assertion report queue is cleared. Any pending assertion reports will not be executed. In the Observed region (for deferred assertion), or Postponed region (for final assertion) of each simulation time step, each pending assertion report that has not been flushed from its queue shall mature, or be confirmed for reporting. Once a report matures, it may no longer be flushed. Then the associated subroutine call (or `$error`, if the assertion fails and no action block is present (see 4.1.3) is executed in the Observed region for deferred assertion, or in the Postponed region for the final assertion, and the pending assertion report is cleared from the appropriate process's deferred assertion report queue.

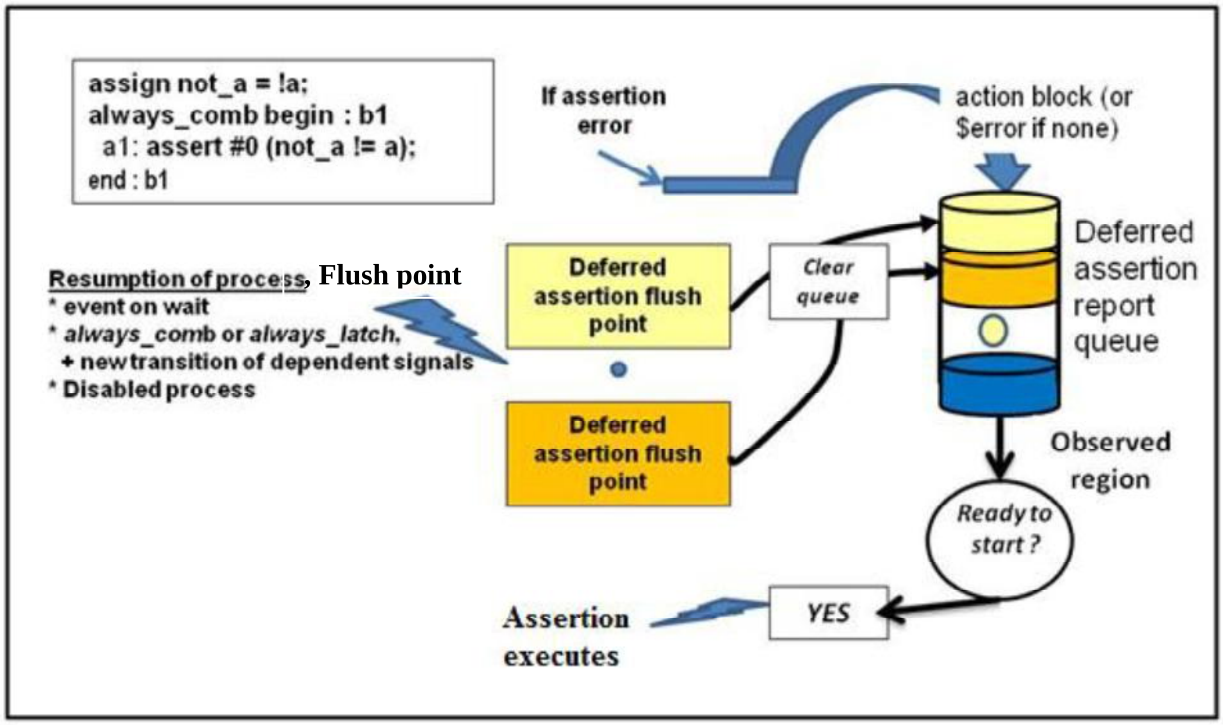
assign not_a = !a;
always_comb begin : b1
a1: assert #0 (not_a != a);
end : b1
If assertion error
action block (or $error if none)
Resumption of process, Flush point
* event on wait
* always_comb or always_latch, + new transition of dependent signals
* Disabled process
Deferred assertion flush point
Clear queue
Deferred assertion flush point
Deferred assertion report queue
Observed region
Ready to start ?
YES
Assertion executes

Figure 4.6.2.1 Deferred action block reporting and queue