

Motivation

Immediate assertions may report false failures when 0-width or short finite-time glitches occur in assignments to variables in always blocks or due to races in the design (see Mantis #1833). For example, the following implementation of a MUX gate may execute the always block twice when a changes. The assertion may incorrectly fire on the first execution of the always block, before the change to not_a has been applied:

```
assign not_a = !a;
always_comb begin
    out = a && x || not_a && y;
    assert (out == a ? x : y);
end
```

Concurrent assertions do not suffer from this problem, but there are many situations where a concurrent assertion cannot be used in place of an immediate assertion:

- Concurrent assertions cannot be used inside functions.
- Concurrent assertions in always blocks cannot report on intermediate values of variables when assigned more than once in sequential code in an always block.
- Concurrent assertions cannot be used in procedural loops and report on values computed in each iteration (this is covered in some cases by Mantis 1995.)
- Concurrent assertions cannot appear in a context without a defined clock.

The proposal is to create a new type of immediate assertion known as a *deferred assertion*. Deferred assertions can be used in any place an immediate assertion is permitted. When a deferred assertion fails in simulation, rather than being reported immediately, the reporting of the failure is deferred until the Observed region. While the Postponed region is also a candidate for the execution of deferred assertion reports, it is risky to allow execution of action block code in the Postponed region. Currently SystemVerilog has no such requirement. The Observed region is the point at which the deferred assertion reports are deemed “mature”. Once mature, any associated action block code shall execute in the following Reactive region. This is consistent with concurrent assertions, and we would like to have the behavior of deferred assertions be as close to concurrent assertions as possible. (In particular note the similarities between the handling of deferred assertion reports and the handling of match_item subroutines in sequences.)

Each process containing deferred assertions has an associated deferred assertion report queue. All pending assertion reports associated with a given process are placed on its deferred assertion report queue. If a simulator advances to a point at which a pending assertion report matures, then the appropriate action block code is executed in the Reactive region. If a process containing a deferred assertion is reevaluated for any reason, the deferred assertion report queue is implicitly flushed. Thus glitches that cause temporary “bad state” are ignored by deferred assertions.

The earlier concept of explicit flushing was removed due to various problematic scenarios. The most common scenario would be overly aggressive flushing of all deferred assertion reports spawned by a process. It is more useful to allow targeted flushing of individual deferred assertions via the existing disable statement. Using disable, a user can precisely control which assertions are explicitly flushed, and not worry about flushing unintended assertions (perhaps buried down deep under a big subroutine call graph).

Another reason to remove explicit flushing was simply due to the complexity it wrought. Since explicit flushing has been shown above to be of dubious value, the complexity isn't worth the trouble at this early stage in the life of deferred assertions.

Modify Clause 9 as follows:

9.6.2 Disable statement

The *disable* statement provides the ability to terminate the activity associated with concurrently active processes, while maintaining the structured nature of procedural descriptions. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. It is useful for handling exception conditions such as hardware interrupts and global resets. The disable statement can also be used to terminate execution of a labeled statement, including a deferred assertion (See clause 16.4. (editor correct reference)).

Modify Clause 16 as follows:

Modify Syntax 16-1

IF Proposal 1729 has been accepted

CHANGE (from modified text of 1729)

```
immediate_assertion_statement ::=
    immediate_assert_statement
    | immediate_assume_statement
    | immediate_cover_statement

immediate_assert_statement ::=
    assert ( expression ) action_block
immediate_assume_statement ::=
    assume ( expression ) action_block
immediate_cover_statement ::=
    cover ( expression ) statement_or_null
```

TO

```
immediate_assertion_statement ::=
    simple_immediate_assertion_statement
    | deferred_immediate_assertion_statement

simple_immediate_assertion_statement ::=
    simple_immediate_assert_statement
```

| simple_immediate_assume_statement
| simple_immediate_cover_statement

simple_immediate_assert_statement ::=
 assert (expression) action_block
simple_immediate_assume_statement ::=
 assume (expression) action_block
simple_immediate_cover_statement ::=
 cover (expression) statement_or_null

deferred_immediate_assertion_statement ::==
 deferred_immediate_assert_statement
 | deferred_immediate_assume_statement
 | deferred_immediate_cover_statement

deferred_immediate_assert_statement ::==
 assert #0 (expression) action_block
deferred_immediate_assume_statement ::==
 assume #0 (expression) action_block
deferred_immediate_cover_statement ::==
 cover #0 (expression) action_block

ELSE

CHANGE

immediate_assert_statement ::=
 assert (expression) action_block

TO

immediate_assert_statement ::=
 simple_immediate_assertion_statement
 | deferred_immediate_assertion_statement

simple_immediate_assertion_statement ::==
 assert (expression) action_block

deferred_immediate_assertion_statement ::==
 assert #0 (expression) action_block

ENDIF

16.3 Immediate Assertions

CHANGE

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is nontemporal and is interpreted the same way as an expression in the condition of a procedural `if` statement. In other words, if the expression evaluates to `x`, `z`, or `0`, then it is interpreted as being false, and the assertion is said to fail. Otherwise, the expression is interpreted as being true, and the assertion is said to pass.

TO

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is nontemporal and is interpreted the same way as an expression in the condition of a procedural `if` statement. If the expression evaluates to `x`, `z`, or `0`, then it is interpreted as being false, and the assertion is said to fail. Otherwise, the expression is interpreted as being true, and the assertion is said to pass.

There are two types of immediate assertions, *simple immediate assertions* and *deferred immediate assertions*. In a simple immediate assertion, pass and fail actions take place immediately upon assertion evaluation. In a deferred immediate assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or “glitch” values. Deferred immediate assertions are described in detail in 16.4 ([editor correct reference](#)).

Add to chapter 16, just after Subclause on Immediate Assertions

(Using 16.4 as placeholder, editor please number sections correctly.)

16.4 Deferred assertions

IF proposal 1729 is accepted

```
immediate_assertion_statement ::=
    simple_immediate_assertion_statement
  | deferred_immediate_assertion_statement

deferred_immediate_assertion_statement ::=
    deferred_immediate_assert_statement
  | deferred_immediate_assume_statement
  | deferred_immediate_cover_statement
```

```
deferred_immediate_assert_statement ::=
    assert #0 ( expression ) action_block

deferred_immediate_assume_statement ::=
    assume #0 ( expression ) action_block

deferred_immediate_cover_statement ::=
    cover #0 ( expression ) action_block
```

Syntax 16-(editor insert number): Deferred assertion syntax (excerpt from Annex A)

ELSE

```
deferred_immediate_assertion_statement ::=
    assert #0 ( expression ) action_block
```

Syntax 16-(editor insert number): Deferred assertion syntax (excerpt from Annex A)

ENDIF

Deferred assertions are a kind of immediate assertion. They can be used to suppress false reports that occur due to glitching activity on combinational inputs to immediate assertions. 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.

A deferred assertion is similar to a simple immediate assertion, but with several key differences:

- Syntax: Deferred assertions use #0 after the verification directive .
- 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*.

Deferred assertion syntax is similar to simple immediate assertion syntax, with the difference being the specification of a #0 after the keyword:

```
assert #0 (expression) action_block
```

As with all immediate assertions, a deferred assertion's expression is evaluated at the time the deferred assertion statement is processed. However, in order to facilitate glitch avoidance, the reporting or action blocks are scheduled at a later point in the current time step.

The pass and fail statements in a deferred assertion's *action_block*, if present, shall each consist of a single subroutine call. The subroutine can be a task, task method, void function, void function method, or system task. The subroutine shall be scheduled in the Reactive region. A subroutine argument may be passed by value

as an `input` or passed by reference as a `ref` or `const ref`. Actual argument expressions that are passed by value use the values of the underlying variables at the instant the deferred assertion expression was evaluated. Actual argument expressions that are passed by reference use or assign the current values of the underlying variables in the Reactive region. It shall be an error to pass automatic or dynamic variables as actuals to a `ref` or `const ref` formal. The requirement of a single subroutine call implies that no `begin-end` construct shall surround the `pass` or `fail` statements, as `begin` is itself a statement which is not a subroutine call.

Deferred assertions may also be used outside procedural code, as a *module_common_item*. This is explained in more detail in 16.4.3.

IF proposal 1729 is accepted, also add to this section:

In addition to deferred `assert` statements, deferred `assume` and `cover` statements are also defined. Other than the deferred evaluation as described in this section, these `assume` and `cover` statements behave the same way as the simple immediate `assume` and `cover` statements described in 16.3 (editor correct reference). A deferred `assume` will often be useful in cases where a combinational condition is checked in a function, but needs to be used as an assumption rather than a proof target by formal tools. A deferred `cover` is useful to avoid crediting tests for covering a condition that is only met in passing by glitched values.

ENDIF

16.4.1 Deferred assertion reporting

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*.

If a *deferred assertion flush point* (see 16.4.2) is reached in a process, its deferred assertion report queue is cleared. Any pending assertion reports will not be executed.

In the Observed region 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) is executed in the Reactive region, and the pending assertion report is cleared from the appropriate process's deferred assertion report queue.

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, as the new passage in the Active region may re-execute some of the deferred assertions with different reported results. In general, 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.

16.4.2 Deferred assertion flush points

A process is defined to have reached a deferred assertion flush point if any of the following occur:

- The process, having been suspended earlier due to reaching an event control or wait statement, resumes execution.
- The process was declared by an `always_comb` or `always_latch`, and its execution is resumed due to a transition on one of its dependent signals.
- The outermost scope of the process is disabled by a disable statement (see 16.4.4)

The following example shows how deferred assertions might be used to avoid undesired reports of a failure due to transitional combinational values in a single simulation time step:

```
assign not_a = !a;
always_comb begin : b1
    a1: assert (not_a != a);
    a2: assert #0 (not_a != a); // Should pass once values have settled
end
```

When `a` changes, a simulator could evaluate assertions `a1` and `a2` twice -- once for the change in `a` and once for the change in `not_a` after the evaluation of the continuous assignment. A failure could thus be reported during the first execution of `a1`. The failure during the first execution of `a2` will be scheduled on the process's deferred assertion report queue. When `not_a` changes, the deferred assertion queue is flushed due to the activation of `b1`, so no failure of `a2` will be reported.

This example illustrates the behavior of deferred assertions in the presence of time delays:

```
always @(a or b) begin : b1
    a3: assert #0 (a == b) rptobj.success(0) else rptobj.error(0, a, b);
    #1;
    a4: assert #0 (a == b) rptobj.success(1) else rptobj.error(1, a, b);
end
```

In this case, due to the time delay in the middle of the procedure, an Observed region will always be reached after the execution of `a3` and before a flush point. Thus any passes or failures of `a3` will always be reported. For `a4`, during cycles where either `a` or `b` changes after it has been executed, failures will be flushed and never reported. In general, deferred assertions must be used carefully when mixed with time delays.

IF Proposal 1729 has been accepted, also add:

The following example illustrates a typical use of a deferred `cover` statement:

```
assign a = ...;
assign b = ...;
always_comb begin : b1
    c1: cover (b != a);
    c2: cover #0 (b != a);
end
```

In this example, we want to make sure some test is covering the case where `a` and `b` have different values. Due to the arbitrary order of the assignments in the simulator, it might be the case that in a cycle where both variables are being assigned the same value, `b1` executes while `a` has been assigned but `b` still holds its previous value. Thus `c1` will be triggered, but this is actually a glitch, and probably not a useful piece of coverage information. In the case of `c2`, this coverage will get added to the deferred report queue, but when

b1 is executed the next time (after b has also been assigned its new value), that coverage point will be flushed, and c2 will correctly not get reported as having been covered during that time step.

ENDIF

16.4.3 Deferred assertions outside procedural code

A deferred assertion statement may also appear outside procedural code, used as a *module_common_item*. In such cases, it is treated as if it were contained in an `always_comb` procedure. For example:

```
module m (input foo, bar);
    a1: assert #0 (foo == bar);
endmodule
```

This is equivalent to:

```
module m (input foo, bar);
    always_comb begin
        a1: assert #0 (foo == bar);
    end
endmodule
```

16.4.4 Disabling deferred assertions

The `disable` statement shall interact with deferred assertions as follows:

- A specific deferred assertion may be disabled. Any pending assertion reports for that assertion are cancelled.
- When a `disable` is applied to the outermost scope of a procedure that has an active deferred assertion queue, in addition to normal disable activities (See 9.6.2), the deferred assertion report queue is flushed and all pending assertion reports on the queue are cleared.

Disabling a task or a non-outermost scope of a procedure does not cause flushing of any pending reports.

The following example illustrates how user code can explicitly flush a pending assertion report. In this case, failures of a1 are only reported in time steps where `bad_val_ok` does not settle at a value of 1.

```
always @(bad_val or bad_val_ok) begin : b1
    a1: assert #0 (bad_val) else $fatal("Sorry");
    if (bad_val_ok) begin
        disable a1;
    end
end
```

The following example illustrates how user code can explicitly flush all pending assertion reports on the deferred assertion queue of process b2:

```

always @(a or b or c) begin : b2
  if (c == 8'hff) begin
    a2: assert #0 (a && b);
  end else begin
    a3: assert #0 (a || b);
  end
end

always @(clear_b2) begin : b3
  disable b2;
end

```

16.4.5 Deferred assertions and multiple processes

As described in the above subclauses, deferred assertions are inherently associated with the process in which they are executed. This means that a deferred assertion within a function may be executed several times due to the function being called by several different processes, and each of these different process executions is independent. The example below illustrates this situation.

```

module fsm(...);
function bit foo(int a, int b)
  ...
  a1: assert #0 (a == b);
  ...
end
...
always_comb begin : b1
  some_stuff = foo(x,y) ? ...
  ...
end
always_comb begin : b2
  other_stuff = foo(z,w) ? ...
  ...
end
endmodule

```

In this case, there are two different processes which may call assertion a1: b1 and b2. Suppose simulation executes the following scenario in the first passage through the Active region of each time step:

- In time step 1, b1 executes with $x \neq y$, and b2 executes with $z \neq w$.
- In time step 2, b1 executes with $x \neq y$, then again with $x = y$.
- In time step 3, b1 executes with $x \neq y$, then b2 executes with $z = w$.

In the first time step, since a1 fails independently for processes b1 and b2, its failure is reported twice.

In the second time step, the failure of a1 in process b1 is flushed when the process is re-triggered, and since the final execution passes, no failure is reported.

In the third time step, the failure in process b1 does not see a flush point, so that failure is reported. In process b2, the assertion passes, so no failure is reported from that process.

MODIFY 19.10 Assertion control system tasks

CHANGE

SystemVerilog provides three system tasks to control assertions.

- `$assertoff` shall stop the checking of all specified assertions until a subsequent `$asserton`. An assertion that is already executing, including execution of the pass or fail statement, is not affected.
- `$assertkill` shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent `$asserton`.
- `$asserton` shall reenable the execution of all specified assertions.

TO

SystemVerilog provides three system tasks to control assertions.

- `$assertoff` shall stop the checking of all specified assertions until a subsequent `$asserton`. An assertion that is already executing, including execution of the pass or fail statement, is not affected. In the case of a deferred assertion (see 16.4 editor correct reference), currently queued reports are not flushed and may still mature, though further checking is prevented until the `$asserton`.
- `$assertkill` shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent `$asserton`. This also flushes any queued pending reports of deferred assertions (see 16.4 editor correct reference) that have not yet matured.
- `$asserton` shall reenable the execution of all specified assertions.

ADD TO CHAPTER 38, AFTER 38.5.2 Assertion Control

38.5.3 VPI Functions and Deferred Assertions

Deferred assertions (see 16.4 editor correct reference) may be in a *pending* state where the assertion has executed, but been placed in a deferred assertion report queue. For any VPI function, if it discards current attempts in progress, that also means it flushes any pending instances of that assertion that have not yet matured from the report queues. If a VPI function does not interfere with current attempts, that also means it does not affect or flush any report queues.

For example, since `VPIAssertionReset` discards all current attempts in progress for the targeted assertion, it flushes any pending reports for that assertion. However, `VPIAssertionDisable` disables the starting of any new attempts without affecting existing attempts, so any pending reports from the disabled assertion that are already queued may still mature and be reported.

MODIFY A.1.4

CHANGE

```
module_common_item ::=  
  module_or_generate_item_declaration  
  | interface_instantiation  
  | program_instantiation  
  | concurrent_assertion_item
```

TO

```
module_common_item ::=  
  module_or_generate_item_declaration  
  | interface_instantiation  
  | program_instantiation  
  | concurrent_assertion_item  
  | deferred_immediate_assertion_statement
```

Modify Clause A.6.10

IF proposal 1729 has been accepted

CHANGE

```
immediate_assertion_statement ::=  
  immediate_assert_statement  
  | immediate_assume_statement  
  | immediate_cover_statement  
  
immediate_assert_statement ::=  
  assert ( expression ) action_block  
  
immediate_assume_statement ::=  
  assume ( expression ) action_block  
  
immediate_cover_statement ::=  
  cover ( expression ) statement_or_null
```

TO

```
immediate_assertion_statement ::=  
  simple_immediate_assertion_statement  
  | deferred_immediate_assertion_statement  
  
simple_immediate_assertion_statement ::=  
  simple_immediate_assert_statement  
  | simple_immediate_assume_statement
```

```
    | simple_immediate_cover_statement

simple_immediate_assert_statement ::=
    assert ( expression ) action_block

simple_immediate_assume_statement ::=
    assume ( expression ) action_block

simple_immediate_cover_statement ::=
    cover ( expression ) statement_or_null

deferred_immediate_assertion_statement ::=
    deferred_immediate_assert_statement
    | deferred_immediate_assume_statement
    | deferred_immediate_cover_statement

deferred_immediate_assert_statement ::=
    assert #0 ( expression ) action_block

deferred_immediate_assume_statement ::=
    assume #0 ( expression ) action_block

deferred_immediate_cover_statement ::=
    cover #0 ( expression ) action_block
```

ELSE

CHANGE

```
immediate_assert_statement ::=
    assert ( expression ) action_block
```

TO

```
immediate_assert_statement ::=
    simple_immediate_assertion_statement
    | deferred_immediate_assertion_statement

simple_immediate_assertion_statement ::=
    assert ( expression ) action_block

deferred_immediate_assertion_statement ::=
    assert #0 ( expression ) action_block
```

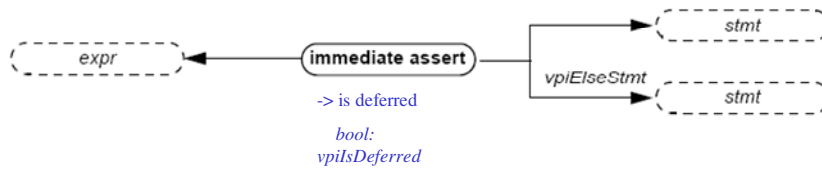
ENDIF

36.47 Sequence expression

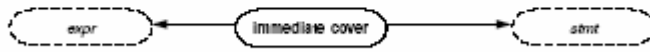
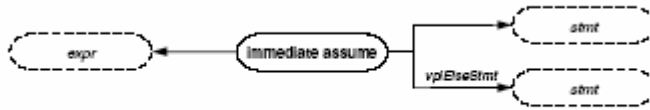
REPLACE



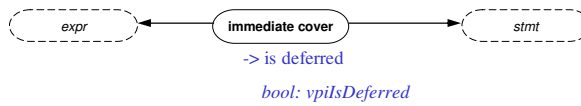
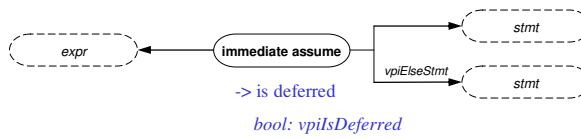
WITH



If proposal 1729 has been accepted, REPLACE



WITH



ENDIF

Annex N, REPLACE

```
#define vpiMethod 645
#define vpiIsClockInferred 649
```

WITH

```
#define vpiMethod 645  
#define vpiIsClockInferred 649  
#define vpiIsDeferred editor to insert number
```