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 will execute the **always** block twice when **a** changes. The assertion will incorrectly fire on the first execution of the always block:

```
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.
- Immediate assertions can be placed in procedural code, but not in structural scopes, so the same combinational checker cannot be used in both contexts.
- 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 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)).

Add to chapter 16, just after Subclause on Immediate Assertions

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

16.4 Deferred Assertions

---

deferred_immediate_assert_statement ::==
    assert #0 (expression) action_block
    | **assert** @ (event_expression) (expression) action_block

---

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

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

Deferred assertion syntax is similar to immediate assertion syntax, with the difference being the specification of a #0 delay control or an event control after the **assert** keyword.

```
assert  [ #0 | event_control ] (expression) action_block
```

A deferred assertion is similar to an immediate assertion, but with one key difference. As with immediate assertions, a deferred assertion's expression is evaluated at the time the deferred assertion statement is processed. However, the pass and fail reporting is scheduled at a later point in the simulator's operation.

The pass and fail statements in an *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. The subroutine shall not contain any `output` or `inout` ports. 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 the current values of the underlying variables (from 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.

16.4.1 Simple (#0) Deferred Assertions

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 the assertion 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 16.4.1) 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, one or more pending assertion reports may mature. In this case, the associated subroutine call (or $error) is executed in the Reactive region, and the pending assertion report is cleared from the appropriate process's deferred assertion report queue.

16.4.2 Deferred Assertion Flush Points

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

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

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 will 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
    a5: assert #0 (a == b) rptobj.success(0) else rptobj.error(0, a, b);
    #1;
```

```
    a6: 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 `a5` and before a flush point. Thus any passes or failures of `a5` will always be reported. For `a6`, 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.

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` block. For example:

```
module m (input foo, bar);
    a99: assert #0 (foo == bar);
endmodule
```

This is equivalent to:

```
module m (input foo, bar);
    always_comb begin
        a99: 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 are cancelled.
- When a `disable` is applied to a procedure that has an active deferred assertion queue, in addition to normal disable activities (See 9.6), the deferred assertion report queue is flushed and all pending assertion reports on the queue are cleared.

The following example illustrates how user code can explicitly flush a pending assertion report. In this case, failures of `a4` 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
    a4: assert #0 (bad_val) else $fatal("Sorry");
    if (bad_val_ok)
        disable a4;
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
        a5: assert #0 (a && b);
    end else begin
        a6: assert #0 (a || b);
    end
end

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

16.4.5 Deferred Assertions with Event Controls

The event control portion of the deferred assertion syntax allows for the settling of glitches with non-zero delay. Such glitches might be caused by gate level logic feeding an RTL module. This example shows the use of an event control in a deferred assertion statement.

When a deferred assertion contains event control syntax, the scheduling is more complex than simple *#0* deferred assertions. The assertion is evaluated when encountered in the code, but its pass/fail blocks are added to a *pre-scheduled queue* associated with the current process and the designated event. If a flush point occurs in the process or the assertion is disabled before the event is activated, the pre-scheduled deferred assertions shall be flushed as described earlier. When the event occurs, the action blocks in the pre-scheduled queues (for all processes that have queues associated with that event) shall be scheduled to mature in the Observed region of the current time step, and sever their connections with the process, so they may no longer be flushed.

Here is a typical situation in which a deferred assertion with event controls might be used:

```
module fsm(...);
function bit foo(…)
    ...
    a42: assert @(posedge clk) (a == b);
    ...
end
...
always_comb begin : b1
    next_state = foo(x,y) ? ...
    ...
end
endmodule
```

In this case, deferred assertion `a42` is associated with the design's clock as its event control; the designer wants to ensure that the assertion can only fail if the condition is failing at a positive clock edge, rather than at an abitrary between-clocks glitch. (Since it is within a function, a concurrent assertion cannot be used here.) When the assertion is evaluated during process `b1`, if `a` and `b` are not equal, the default `$error` call will be pre-scheduled to be scheduled at the next positive edge of `clk`. If `b1` is activated again before that positive edge, the pre-scheduled queue will be flushed, and the error will not be reported unless a42 fails again. Otherwise, upon the next positive edge, this report will be scheduled to mature in the Observed region of the time step where the positive edge occurs. Note that if function `foo` is called again later by process `b1` in the

time step where this edge happened, a new `$error` report may be pre-scheduled for the following edge—it will not flush or modify the report that has already been confirmed by the current edge.

16.4.6 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)
    ...
    a56: assert #0 (a == b);
    ...
end
...
always_comb begin : b1
    next_state = 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 `a56`: `b1` and `b2`. Suppose simulation executes the following scenario:

- In time step 1, `b1` is triggered with x!=y, and b2 is triggered with z!=w.
- In time step 2, `b1` is triggered with x!=y, then again with x==y.
- In time step 3, `b1` is triggered with x!=y, then b2 is triggered with z==w.

In the first time step, since `a56` fails independently for processes `b1` and `b2`, its failure is reported twice.

In the second time step, the failure of `a56` 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 Clause 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_assert_statement

Modify Clause A.6.10

CHANGE

immediate_assert_statement ::=
assert (expression) action_block

TO

immediate_assert_statement ::=
simple_immediate_assert_statement
| deferred_immediate_assert_statement

simple_immediate_assert_statement ::==
assert (expression) action_block

deferred_immediate_assert_statement ::==
assert #0 (expression) action_block
| assert @ (event_expression) (expression) action_block