Motivation

Immediate assertions may report false failures when 0-width 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 with no defined clock.

The proposal is to create a new type of immediate assertion, "deferred immediate assertions", which can be used in any place an immediate assertion is permitted. When a deferred immediate assertion fails in simulation, rather than being reported immediately, the reporting of the failure is deferred until the postponed region. The postponed region is chosen because it is guaranteed to be executed only once per time step, after all procedural code has executed and 0-delay signal values settled, and is thus ideal for glitch prevention. By default, if the process is rescheduled due to another event trigger in the same time step, the deferred assertion failure queue is flushed. Thus glitches are prevented.

Note: This current proposal version does not define the grammar changes, appendix additions, etc. Those will be added after we have converged on the main proposal.

Add to chapter 16:

16.3.1 Deferred Immediate Assertions

A deferred immediate assertion statement is a nontemporal test of an expression performed when the statement is executed in the procedural code. An immediate assertion statement is a deferred immediate assertion statement if specified using '#0' or an event control after the assert keyword.

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

A deferred immediate assertion differs from other immediate assertions in the following ways:

- The *action_block,* if it is present, shall only contain calls to constant functions, tasks, or system functions.
- When the assertion fails, the action block is not executed immediately. Instead, the action block function call (or default `$error` call) and the current values of its arguments are placed in a *deferred assertion queue* associated with the current process.
- If a *deferred assertion flush point* (see section 16.3.1.1) is reached in a process, its deferred assertion queue is cleared, so any deferred assertion failures until that point will not be reported.
- In the postponed region of each simulation time step, action blocks and default `$error` calls in 0-delay deferred assertion queues, or in deferred assertion queues whose event control is active in the current time step, are executed, and the queues are cleared. If a deferred assertion statement contained an event control which was not activated in the current time step, its failure is ignored, and its queue is cleared.

Thus, any given failure of a deferred immediate assertion might be ignored if the procedure in which it occurred reaches a deferred assertion flush point or if it is controlled by an event not relevant to the current time step. This functionality is often useful to prevent 0-width "glitches" that are otherwise reported on transitional combinational signal values by immediate assertions.

16.13.1.1 Deferred Assertion Flush Points

A procedure may be in one of two *flush states* : *implicit flushing*, the default, or *explicit flushing*. In an implicit flushing state, 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 statement, resumes execution due to the occurrence of an event.
- The procedure was declared by an `always_comb` statement, and its execution is resumed due to a transition on one of its dependent signals.
- A `$deferred_assertion_flush()` system function call is executed. The use of this function call moves the procedure to an explicit flushing state.

A procedure shall enter an explicit flushing state if a `$deferred_assertion_explicit()` or a `$deferred_assertion_flush()` system call is executed. Once a procedure is in an explicit flushing state, only a `$deferred_assertion_flush()` creates a flush point. If a procedure is put in an explicit flushing state using `$deferred_assertion_explicit()` and it does not contain any `$deferred_assertion_flush()` calls, deferred immediate assertions in that procedure behave just like ordinary immediate assertions, except that their reporting is delayed until the postponed region.

16.13.1.2 Deferred Assertions Outside Procedural Code

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

Formatted: Indent: First line: 0.5"
Deleted: defer.
Deleted: contain a single
Deleted: a
Deleted: the
Deleted: ¶
Formatted: Normal

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

This is equivalent to:

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

16.13.1.3 Examples of Deferred Immediate Assertions

The following examples illustrate the differences between ordinary and deferred immediate assertions. The first 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
  #0
  a3: assert #0 (not_a == a); // Should fail once values have settled
end
```

Most simulators will execute the code in block b1 above twice when a changes, once due to the change in a and a second time due to the change in not_a. A failure will be reported during the first execution of a1, but the failure during the first execution of a2 will end up on the deferred assertion queue. When not_a changes, the deferred assertion queue is flushed, so no failure of a2 will be reported. But in its final execution, a3 will fail, and since no flush point is reached in the process before the postponed region, this failure will be reported. Note that the #0 time delay does not change any of this behavior, since a time delay does not result in a flush point.

This example illustrates a use of explicit flushing:

```
always @(bad_val or bad_val_ok) begin : b1
  a4: assert #0 (bad_val);
  if (bad_val_ok)
      $deferred_assertion_flush();
end
```

In this case, any failure of a4 is only reported in time steps where bad_val_ok does not settle at a value of 1. Here is another example of explicit flushing:

```
// not_a and a are inputs, but we don't trust supplier
always_@(p or q or r or s) begin : b1
  $deferred_assertion_explicit();
  ...
```

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Times New Roman

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Indent: First line: 0.5"

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Normal

Deleted: 2

Deleted: defer

Deleted: defer

Deleted: defer

Formatted: Font: Not Bold, Complex Script Font: Not Bold

```
  if (expecting_a) begin
     @(not_a or a) begin
          $deferred_assertion_flush();
          a3: assert #0 (not_a != a);
     end
  end
end
```

Here we assume we have a large block of combinational code, but our deferred assertion is controlled by an event control on an inner block. When that inner block is activated, if the assertion fails, we don't want a later activation of the outer `always` block to cause the assertion failure to be discarded. On the other hand, if that inner block is activated multiple times, we don't want to see a failure on the intermediate values if the correct values of a and not_a are ultimately assigned.

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

```
always @(a or b) begin : b1
  a5: assert #0 (a == b);
  #1
  a6: assert #0 (a == b);
end
```

In this case, due to the time delay in the middle of the procedure, a postponed region will always be reached after the execution of `a5` and before a flush point. Thus any failure of `a5` will always be reported. For `a6`, during cycles where either `a` or `b` changes after it has been executed, its failures will be flushed and never reported. In general, deferred immediate assertions must be used very carefully when mixed with time delays.

This final example shows the use of event control in a deferred immediate assertion statement.

```
module mymod(clk1, rst1, ...);
...
function int myfunc(a,not_a,b,c) begin
      a1: assert @(clk1) (not_a != a);
      ...
end
```

Here the assertion `a1` is checking the values of `a` and `not_a` any time `myfunc` is called during a time step when a `clk1` edge occurs. If a process that calls the function is invoked multiple times, only the values in the final execution of the process might cause an assertion failure. If the function is called during an intermediate time step when there is no transition on the clock, assertion failures are ignored.

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Indent: First line: 0.5"

Formatted: Indent: First line: 0.17"

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: (Default) Courier New, Complex Script Font: Courier New

Deleted: e next e

Deleted: `defer`

Deleted: `defer`

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: Bold, Complex Script Font: Bold

Formatted: Font: (Default) Courier New, Complex Script Font: Courier New

Formatted: Font: (Default) Courier New, Complex Script Font: Courier New

Formatted: Font: (Default) Courier New, Complex Script Font: Courier New