5. Scheduling semantics

5.1 Execution of a model

The balance of the sections of this standard describe the behavior of each of the elements of the language. This section gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events.

The elements that make up the Verilog HDL can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. An HDL has to be a parallel programming language. The execution of certain language constructs is defined by parallel execution of blocks or processes. It is important to understand what execution order is guaranteed to the user, and what execution order is indeterminate.

Although the Verilog HDL is used for more than simulation, the semantics of the language are defined for simulation, and everything else is abstracted from this base definition.

5.2 Event simulation

The Verilog HDL is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this section to provide a context to describe the meaning and valid interpretation of Verilog HDL constructs. These resulting definitions provide the standard Verilog reference model for simulation, which all compliant simulators shall implement. Note, though, that there is a great deal of choice in the definitions that follow, and differences in some details of execution are to be expected between different simulators. In addition, Verilog HDL simulators are free to use different algorithms than those described in this section, provided the user-visible effect is consistent with the reference model.

A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include primitives, modules, initial and always procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements.

Every change in value of a net or variable in the circuit being simulated, as well as the named event, is considered an *update event*.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated. The term *time* is used interchangeably with simulation time in this section.

Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an *event queue*, ordered by simulation time. Putting an event on the queue is called *scheduling an event*.

5.3 The stratified event queue

The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions but are only removed from the *active* region.

1) Events that occur at the current simulation time and can be processed in any order. These are the *active* events.
2) Events that occur at the current simulation time, but that shall be processed after all the active events are processed. These are the *inactive* events.

3) Events that have been evaluated during some previous simulation time, but that shall be assigned at this simulation time after all the active and inactive events are processed. These are the *non blocking assign update* events.

4) Events that shall be processed after all the active, inactive, and non blocking assign update events are processed. These are the *monitor* events.

5) Events that occur at some future simulation time. These are the *future* events. Future events are divided into *future inactive events*, and *future non blocking assignment update events*.

The processing of all the active events is called a *simulation cycle*.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL.

An *explicit zero delay* (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A non blocking assignment (see 9.2.2) creates a non blocking assign update event, scheduled for current or a later simulation time.

The **$monitor** and **$strobe** system tasks (see 17.1) create monitor events for their arguments. These events are continuously re-enabled in every successive time step. The monitor events are unique in that they cannot create any other events.

The call back procedures scheduled with PLI routines such as `tf_synchronize()` (see 25.58) or `vpi_register_cb(cb_readwrite)` (see 27.33) shall be treated as inactive events.

5.4 The Verilog simulation reference model

In all the examples that follow, T refers to the current simulation time, and all events are held in the event queue, ordered by simulation time.

```
while (there are events) {
        if (no active events) {
                if (there are inactive events) {
                        activate all inactive events;
                } else if (there are non blocking assign update events) {
                        activate all non blocking assign update events;
                } else if (there are monitor events) {
                        activate all monitor events;
                } else {
                        advance T to the next event time;
                        activate all inactive events for time T;
                }
        }
        E = any active event;
        if (E is an update event) {
                update the modified object;
                add evaluation events for sensitive processes to event queue;

        } else { /* shall be an evaluation event */
                evaluate the process;
                add update events to the event queue;
        }
}
```

5.4.1 Determinism

This standard guarantees a certain scheduling order.

1) Statements within a `begin-end` block shall be executed in the order in which they appear in that `begin-end` block. Execution of statements in a particular `begin-end` block can be suspended in favor of other processes in the model; however, in no case shall the statements in a `begin-end` block be executed in any order other than that in which they appear in the source.

2) Non blocking assignments shall be performed in the order the statements were executed. Consider the following example:

```
initial begin
    a <= 0;
    a <= 1;
end
```

When this block is executed, there will be two events added to the non blocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time step `1`, the variable `a` will be assigned `0` and then `1`.

5.4.2 Nondeterminism

One source of nondeterminism is the fact that active events can be taken off the queue and processed in any order. Another source of nondeterminism is that statements without time-control constructs in behavioral blocks do not have to be executed as one event. Time control statements are the # expression and @ expression constructs (see 9.7). At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.

5.5 Race conditions

Because the execution of expression evaluation and net update events may be intermingled, race conditions are possible:

```
assign p = q;
initial begin
    q = 1;
    #1 q = 0;
    $display(p);
end
```

The simulator is correct in displaying either a `1` or a `0`. The assignment of `0` to `q` enables an update event for `p`. The simulator may either continue and execute the $display task or execute the update for `p`, followed by the $display task.

5.6 Scheduling implication of assignments

Assignments are translated into processes and events as follows.

5.6.1 Continuous assignment

A continuous assignment statement (Section 6) corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

5.6.2 Procedural continuous assignment

A procedural continuous assignment (which are the **assign** or **force** statement; see 9.3) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A **deassign** or a **release** statement deactivates any corresponding **assign** or **force** statement(s).

5.6.3 Blocking assignment

A blocking assignment statement (see 9.2.1) with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

5.6.4 Non blocking assignment

A non blocking assignment statement (see 9.2.2) always computes the updated value and schedules the update as a non blocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

5.6.5 Switch (transistor) processing

The event-driven simulation algorithm described in 5.4 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bi-directional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net, because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events.

Further refinement is required when some transistors have gate value `x`. A conceptually simple technique is to solve the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response `x`.

5.6.6 Port connections

Ports connect processes through implicit continuous assignment statements or implicit bidirectional connections. Bidirectional connections are analogous to an always-enabled tran connection between the two nets, but without any strength reduction. Port connection rules require that a value receiver be a net or a structural net expression.

Ports can always be represented as declared objects connected as follows:

If an input port, then a continuous assignment from an outside expression to a local (input) net
If an output port, then a continuous assignment from a local output expression to an outside net
If an inout, then a nonstrength-reducing transistor connecting the local net to an outside net

5.6.7 Functions and tasks

Task and function parameter passing is by value, and it copies in on invocation and copies out on return. The copy out on the return function behaves in the same manner as does any blocking assignment.