Section 5

Scheduling semantics

5.1 Execution of a model

The balance of the sections of this specification 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.

Verilog is a hardware description language (HDL). The elements of the language can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. A hardware description language must 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 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 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 must implement. Note, though, that there is a great deal of choice in the definitions which 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 here, provided the user visible effect is consistent with the reference model.

A design consists of connected threads of execution or processes. Processes are objects which can be evaluated, which may have state, and can respond to changes on their inputs, producing 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 register 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 which 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 the following discussion.

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 which occur at the current simulation time and can be processed in any order. These are the *active* events.

2) Events which occur at the current simulation time, but which must be processed after all the active events are processed. These are the *inactive* events.

3) Events which have been evaluated during some previous simulation time, but must 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 which must be processed after all the active, inactive, and non-blocking assign update events are processed. These are the *monitor* events.

5) Events which 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 non-determinism in the language.

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 (section 9.2.2) creates a non-blocking assign update event, scheduled for cuurent or a later simulation time.

The **$monitor** and **$strobe** system tasks (section 14.1) 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 can not create any other events.

5.4 The verilog simulation reference model

In all the examples which 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 { /* must be an evaluation event */
            evaluate the process;
            add update events to the event queue;
        }
    }
```

5.4.1 Determinism

The Verilog simulation reference standard guarantees 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 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 Non-determinism

One source of non-determinism is the fact that active events can be taken off the queue and processed in any order. Another source of non-determinism 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 (section 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 non-deterministic, 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 here. 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** statements, section 9.3) corresponds to a process which 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 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 resumed, (or immediately, were there no delay 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 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 non-zero. 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 above depends on uni-directional signal flow, and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

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 which model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing must consider all the devices in a channel 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, and can process a subset of tran connected events at a particular time, intermingled with execution of other active events.

Further refinement is required when some transistors have gate value `x`. A conceptually simple technique is to repeatedly solve the network with these transistors set to all possible combinations of fully conducting and non-conducting. 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, however, 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 input port, then a continuous assignment from outside expression to local (input) net

if output port, then a continuous assignment from local output expression to outside net

if inout, then a non strength reducing transistor connecting local net to outside net

Port Connection Rules (section 12.3.7) is the definitive section for port connection rules. Modules can have the following declaration:

module `foo (.a(p), .b(p));`

which makes `a` and `b` the external names of the ports, and `p` the internal name of the port. This means `a`, `b`, and `p` are connected bidirectionally, and hence always have the same value.

5.6.7 Functions and tasks

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