A Note on the IEEE Verilog Simulation Cycle

David M. Russinoff

September 28, 2005

Abstract

The IEEE Verilog Standard contains a number of ambiguities and inconsistencies with respect to the semantics of event scheduling, creating difficulties for the programmer in predicting the behavior of a compliant simulator. In this note, we bring some of these issues to light and attempt to resolve them by outlining an abstract formulation of the Verilog simulation cycle, aimed at clarifying the intent of the Verilog Standard Committee. We also observe that the degree of freedom allowed by the Standard in the interleaving of concurrent processes is impractical, and if fully exercised, would inevitably lead to race conditions and unpredicable results. Consequently, this aspect of the specification has been essentially ignored by tacit agreement between implementors and users. As a remedy, we propose to modify the specification of the simulation cycle by imposing a simple restriction on the nondeterministic selection of active events. The suggested restriction would allow the programmer to eliminate race conditions without inhibiting compiler optimization.

Introduction

The Verilog simulation cycle—a model of the nondeterministic procedure by which concurrent processes are scheduled and executed—is central to the semantics of the language as specified by IEEE Standard 1384 [4]. Its definition, however, is poorly understood by the Verilog user community, especially with regard to two questions:

(1) When does a process become *enabled*, i.e., eligible for execution?

(2) Under what circumstances may the simulator suspend the execution of one enabled process in order to pass control to another?

Confusion surrounding the first question may be fairly attributed to ambiguities and inconsistencies in the Standard, as recounted below. With respect to the second, however, the document is quite clear (pp. 65-66):

> The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL ... At any time while evaluating a behavior 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.

But commercial Verilog implementations are generally quite conservative in their exercise of this freedom. Moreover, programmers often rely on the simulator's restraint in this regard, assuming, for example, that control will never be passed from one process to another until the first has either terminated or been halted by a timing control. Simulators usually, but not always, conform to this assumption, which may lead to unexpected results. In any case, the effectiveness of a language standard is compromised when it is supplanted by implicit convention.

The purpose of this note is twofold. First, we present a formulation of the simulation cycle, restricting our attention to a small but practical subset of Verilog, in which we attempt to clarify several points that are left ambiguous by the Standard. We are aware of one previous similar effort, by Mike Gordon [3], on behalf of the formal methods community. We find Gordon's work to be useful as a point of departure, but flawed in its treatment of certain aspects of event scheduling and delay.

Second, we suggest a practical restriction of the nondeterministic selection of enabled events, intended to narrow the gap between the Standard and prevailing implementation policies while still providing opportunities for compiler optimization through transfer of control between enabled processes. We also formulate a simple coding guideline that eliminates race conditions and guarantees predictable behavior in the presence of this restriction.

Events

Of the classes of processes supported by the Standard, we consider only the two that are used most commonly by designers of register-transfer logic (RTL): the *procedural block* and the *continuous assignment* . A process may assign values to signals of a variety of types, but for simplicity, we shall assume that all signals defined by procedural blocks and continuous assignments are declared as *registers* and *wires*, respectively, and that every signal is a scalar, i.e., assumes only single-bit values.

A procedural block consists of either of the two keywords `initial` and `always` followed by a procedural statement, which may be a compound statement consisting of a sequence of statements bracketed by the keywords `begin` and `end` . Of particular interest are the *blocking* and *nonblocking assignment* statements,

$$v \texttt{ = \#}\delta \;\; E;$$

and

$$v \texttt{ <= \#}\delta \;\; E;$$

respectively, where v is a register, E is an arbitrary expression, and `#`δ is an optional indicator of a delay of $\delta \geq 0$ time units. Any procedural statement may be preceded by a *timing control* of any of three types:

(1) a *delay indicator*, `#`δ, where δ is a natural number;

(2) an *edge-sensitive control*, either `@(`v`)`, `@(`v_1 `or ... or` v_k`)`, `@(posedge` v`)`, or `@(negedge` v`)`, where v and v_i, $i = 1, \ldots, k$, are signals;

(3) a *level-sensitive control*, `wait(`E`)`, where E is an expression.

A continuous assignment has the form of a single statement,

$$\texttt{assign \#}\delta\ v\ \texttt{=}\ E\texttt{;}$$

where v is a wire and again, the delay indicator is optional.

The state of a procedural block comprises several components:

(1) a *program counter* (PC), which may point either to a statement, referred to as the *current statement* of the block, or to a timing control;

(2) an *active* bit;

(3) in the case of an inactive block that has not been terminated, the *time* at which the block is scheduled to resume;

(4) in the event that the current statement is a delayed blocking assignment that has already been evaluated, a *pending assignment value*.

Following Gordon [3], we distinguish between *active* and *enabled* processes. By definition, a procedural block is enabled if and only if it is active and its PC does not point to a timing control. A continuous assignment is considered to be continuously active, but may or may not be enabled, as determined by the value of an explicit *enabled* bit associated with the process.

During the course of simulation, a set of *pending updates* is maintained. Each update is identified as either nonblocking or continuous (according to the type of assignment statement from which is was generated), and consists of an active bit, a value, a signal to which the value is to be assigned, and a scheduled time of assignment. An assignment update is enabled if and only if it is active.

All processes (procedural blocks and continuous assignments) and assignment updates (nonblocking and continuous) are referred to as *events*.

Execution

The state of a simulation consists of

(1) a nonnegative-integer-valued global variable C, representing the current time;

(2) the current value of each signal;

(3) the state of each process as defined above;

(4) the set of all pending updates.

The period between successive increments of C is called a *simulation cycle*. At each step within a simulation cycle, an enabled event (if any exists) is selected for execution. In this section, we define, for each event type, the effect of an execution step on the simulation state.

In the case of a procedural block, execution is determined by the current statement as follows:

(1) v `=` E : Set the value of v to the result of evaluating E.

(2) v `=` `#`δ E : If there is a pending assignment value associated with the state of this block, update v with that value. Otherwise, evaluate E, set the pending assignment value to the result, clear the active bit, and schedule the process to resume at time $C + \delta$.

(3) v `<=` `#`δ E : Add an inactive nonblocking pending update for v, using the value of E, with scheduled time of assignment $C + \delta$, overriding any other pending update for v that was previously scheduled for time $C + \delta$.

(4) v `<=` E : This is equivalent to v `<=` `#0` E.

Except for Case (2) with no pending value, the PC is then incremented or otherwise adjusted according to any indicated sequence control construct. If this leaves the PC pointing to

(a) a delay indicator `#`δ, then the PC is adjusted further to point to the following statement; the process becomes inactive and is scheduled to resume at time $C + \delta$;

(b) a level-sensitive control `wait(`E`)`, then E is evaluated and if the result is nonzero, then the PC is incremented.

If the selected enabled event is a continuous assignment, then execution proceeds as follows, according to whether or not a delay is indicated:

(1) `assign` v `=` E : Set the value of v to the result of evaluating E.

(1) `assign` `#`δ v `=` E : Add an inactive continuous pending update for v using the result of evaluating E and time $C + \delta$. Delete any existing pending update for v.

In either case, the enabled bit of the continuous assignment is then cleared. Note that the scheduling of a delayed continuous assignment, unlike that of a nonblocking assignment, has the effect of overriding any previously scheduled assignment to the same wire, regardless of the time for which it was scheduled. This behavior is characteristic of *inertial delay* .

Finally, if the selected event is a pending update, then the signal is updated with the indicated value and the event is retired.

When the execution of an event changes the value of a signal, and the PC of some active procedural block points to a timing control, that timing control is said to *fire* under the following conditions:

(1) `@(`v`)` or `@(`v_1 `or` ... `or` v_k`)` : v or some v_i changes value;

(2) `@(posedge` v`)` : the value of v changes to 1;

(3) `@(negedge` v`)` : the value of v changes to 0;

(4) `wait(`E`)` : the value of E changes to 1.

Similarly, a disabled continuous assignment fires whenever its right-hand side changes value.

The Simulation Cycle

In the initial state of a simulation, $C = 0$, signal values are initialized as indicated, every procedural block is active with PC reset to its initial statement or timing control, every continuous assignment is disabled, and there are no pending updates. If any procedural block begins with a delay indicator `#`δ, then its PC is adjusted to point to the following statement, its active bit is cleared, and it is scheduled to resume at time δ. Simulation proceeds as follows:

(1) If there are no enabled events, go to (3); otherwise, select (nondeterministically) an enabled event and execute it.

(2) Increment the PC of each active process whose current instruction is a timing control that fires. Set the enabled bit of any continuous assignment that fires. Go to (1).

(3) If there are no inactive processes or pending continuous assignment updates scheduled for time C, then go to (4). Otherwise, activate all such events and go to (1).

(4) If there are no pending nonblocking assignments scheduled for time C, then go to (5). Otherwise, activate and execute all such assignments and go to (2).

(5) Increase C to the earliest time at which some input changes or for which some event is scheduled. Update the inputs and go to (2).

This formulation of the algorithm reflects a number of decisions concerning the resolution of ambiguities in the Standard. For example, our treatment of continuous assignment is at odds with following prescription (p. 71):

> Assignments on [wires] shall be continuous and automatic. This means that that whenever an operand in the right-hand side expression changes value, the whole right-hand side shall be evaluated and if the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

This suggests that a continuous assignment must be executed immediately whenever it becomes enabled, taking precedence over any other process (even one that has already begun execution). This would require, for example, the following program to terminate with the values `x` = `y` = `z` = 1:

```
module MOD1;
  wire x;
  reg y, z;

  assign x = y;

  initial begin
     y = 0;
  #1 y = 1;
     z = x;
  end
```

```
endmodule
```

On the other hand, we read elsewhere (p. 67) that a continuous assignment is to be treated as any other event:

> 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.

This clearly implies that an enabled continuous assignment need not be executed immediately (p. 66):

> One source of nondeterminism is the fact that active events can be taken off the queue and processed in any order.

Thus, the alternative result `x` = `y` = 1, `z` = 0 must be allowed. Since this is indeed the observed outcome of a VCS trace of this program, we elected to ignore the contradictory "continuous and automatic" directive.

Further confusion surrounds the execution of nonblocking assignment updates. On the one hand, it is clear that for a given simulation cycle, these events are not to be activated until all other events scheduled for the same time have been executed. What is not clear is how these events, once activated, are to be prioritized relative to any other events that may be generated by their execution. Consider, for example, the following code:

```
module MOD2;
 reg x, y;

 initial begin
   x <= 1;
   y <= 1;
 end

 always @(x) y = 0;

endmodule
```

After the `initial` block is executed and the two resulting nonblocking assignment updates are activated, the first of the two is executed, and this enables the `always` block. Now, which is executed next: the nonblocking assignment update to `y`, or the blocking assignment? This would depend on one's interpretation of the directive (p. 65) that nonblocking assignment update events "shall be assigned ... after all the active and inactive events have been processed." In Gordon's model [3], this statement has been given the strictest possible interpretation: any event generated by a nonblocking update must be executed before any remaining nonblocking updates. But this policy is in direct conflict with a later directive (p. 122):

> When the simulator activates the *nonblocking assign update events*, the simulator updates the left-hand side of each nonblocking assignment statement ... Nonblocking assignment events can create blocking assignment events.

These blocking assignment events shall be processed after the scheduled nonblocking events.

This last statement also contradicts the one quoted above concerning nondeterminism, i.e., "active events can be taken off the queue and processed in any order." However, it is consistent with the VCS trace of `MOD2`, in which the final value of `y` is 0, and is therefore reflected in our model.

Restricted Nondeterminism

In this section, we investigate the consequences of the nondeterministic nature of the simulation cycle. As a practical matter, we are especially interested in an even more limited class of Verilog programs, typical of those used to model synchronous sequential RTL. Thus, we shall henceforth further restrict our attention to programs consisting only of `always` blocks and continuous assignments, in which the following conditions are satisfied:

(1) Nonblocking assignments occur only within blocks of the form

`always @(posedge` *clk* `) begin` ... `end`

where *clk* is an input.

(2) Blocking assignments occur only within blocks of the form

`always @* begin` ... `end`

where `*`, by convention, represents the disjunction of all signals appearing in the right-hand side of some assignment within the block, called the *sensitivity list* of the block.

(3) No other timing controls (in particular, no delays) appear anywhere in the program.

(4) No signal is assigned values by two distinct processes.

(5) The program does not contain any set of blocking or continuous assignments v_0 = E_0, ..., v_k = E_k, such that v_i occurs in E_{i-1} for $i = 1, \ldots, k$ and $v_k = v_0$.

Such programs will be considered *standard* Verilog. It is not difficult to construct a standard program that exhibits the same unpredictability that was seen in our earlier example `MOD1`:

```
module MOD3(input clk);
 reg w = 0, x = 0, y = 0, a = 0;
 wire z = 0;

 always @* begin
   x = ~a;
   y = z;
```

```
    x = a;
    w = 0;
  end

  assign z = x;

  always @(posedge clk) a <= y;

endmodule
```

If the first procedural block is interrupted in order to execute the continuous assignment to `z` in response to each assignment to `x`, then the value of `a` will alternate on successive clock cycles; otherwise, `a` will remain constant. Some commercial Verilog compilers, such as VCS [1], do interrupt executing processes in some situations in order to execute a continuous assignment that has just become enabled, and are therefore liable to exhibit a race condition here. On the other hand, we are told that no existing implementation will ever interrupt one procedural block in favor of another [2]. Hence the following variation is more predictable:

```
module MOD4(input clk);
  reg w = 0, x = 0, y = 0, z = 0, a = 0;

  always @* begin
    x = ~a;
    y = z;
    x = a;
    w = 0;
  end

  always @* z = x;

  always @(posedge clk) a <= y;

endmodule
```

Of course, it might be argued that both of the above contain circularities that constitute "bad programming practice", and that their author therefore deserves whatever results are produced. In contrast, the following exhibits only data dependencies of the sort commonly found in conventional RTL designs:

```
module MOD5(input clk);
  reg w = 0, x = 0, y = 0, z = 0, a = 1;

  always @* begin
    x = ~a;
    y = x;
  end
```

```
  always @* begin
    z = y;
    w = x;
  end

  always @(posedge clk) a <= z;

endmodule
```

It may well be that every existing Verilog implementation would treat each block of this program as atomic, thereby producing an execution in which the value of `a` alternates on successive cycles as expected. However, the Standard clearly provides for a simulation in which the blocks are interleaved, updating the signals in the order `a`, `x`, `z`, `y`, `w`, which leads to a different result, with the value of `a` settling at 0 after the first cycle. Consequently, common design practice must rely on the presumed behavior of commercial simulators to prevent race conditions, while ignoring the freedom encouraged by the Standard.

Here we propose a minor modification of the Standard, a simple restriction on the nondeterministic selection of enabled processes that would eliminate the anomalous behavior described above. This restriction would still allow a sufficient degree of freedom to provide for compiler optimization, and is in fact consistent with all major commercial implementations:

> Execution of an enabled process P may be suspended in order to pass control to another enabled event, but only if that event is a process Q, and only immediately after Q becomes enabled. Once the execution of Q is terminated or otherwise disabled, control must be passed directly back to P.

In order to modify our simulation algorithm to accommodate this restriction, we introduce an additional data structure: an *execution stack* of enabled events. When a process or other event is selected for execution, it is pushed onto the stack (which is empty in the initial state). When the event at the top of the stack terminates or becomes disabled, it is popped from the stack. The revised simulation cycle is as follows:

(1) If the execution stack is not empty, go to (2). If there is no enabled event, go to (4). Otherwise, select an enabled event and push it onto the stack.

(2) Execute the event at the top of the stack. Unless that event is a procedural block that remains enabled, pop it from the stack. Increment the PC of each disabled active process that points to a timing control that fires, set the enabled bit of any continuous assignment that fires, and go (nondeterministically) to (3) or to (1).

(3) If any process was enabled by the last execution step, then select one such process and push it onto the stack. Go to (1).

(4) If there are no inactive processes or pending continuous assignment updates scheduled for time C, then go to (5). Otherwise, activate all such events and go to (1).

(5) If there are no pending nonblocking assignments scheduled for time C, then go to (7). Otherwise, execute all such assignments.

(6) Increment the PC of each disabled active process that points to a timing control that fires, set the enabled bit of any continuous assignment that fires, and go to (1).

(7) Increase C to the earliest time at which either some input changes or some process is scheduled to resume. Update the inputs and activate all such processes. Go to (6).

Consider the effect of this restriction on the behavior of `MOD5`. If the execution of `x = a` results in a new value of x, then control may be passed to the newly enabled second block. But if this occurs, then the second block must be executed to termination. The first block is then resumed. As it terminates, the second block is enabled once again and consequently re-executed. Thus, the race condition is eliminated; the end result is as if the first block had never been interrupted.

`MOD5` belongs to a class of *acyclic* standard programs, as defined below, which are well-behaved under our restricted simulation cycle. Our goal is to show that if a continuous or blocking assignment v = E of such a program is executed under certain conditions during a simulation cycle, then at the end of the cycle, the value of v is the same as that of E.

First, given two distinct processes P and Q, we define P to be *dependent* on Q if any signal in the sensitivity list of P is assigned by a statement in Q. A procedural block P is *dependent* on itself if it contains two assignments v = E and v' = E' such that v' occurs in E and the assignment to v precedes the assignment to v' with respect to program order. An *acyclic* program is one that contains no set of processes $P_0, \ldots, P_k$ such that $P_0 = P_k$ and for $i = 1, \ldots, k$, P_i is dependent on P_{i-1}.

In the three examples in this section, no process is dependent on itself. `MOD3` and `MOD4` both contain mutually dependent processes; only `MOD5` is acyclic.

Note that when a process is pushed onto the stack, it must be dependent on every other process on the stack. During the execution of an acyclic program, therefore, no process on the stack can be dependent on the process at the top. Now let v = E be an assignment occurring in a process P, and suppose that some signal $v' \neq v$ that occurs in E changes value. If P itself is the process that produces this change (i.e., P is at the top of the stack), then the assignment to v must appear later in P than the assignment to v'. Otherwise, P must now be enabled and cannot already be on the stack; hence, P will eventually be selected for execution later during the current cycle. Thus, we have the following result, which precludes race conditions of the sort that we observed above:

> *Let v = E be a continuous or blocking assignment that appears in a process P of an acyclic standard program. Suppose that during a given simulation cycle, this assignment is the last assignment to v to be executed during the final execution of P. Then the final value of v for that cycle coincides with the final value of E.*

Conclusion

Naturally, as the product of a committee, the IEEE Verilog Standard represents a divergent set of views. It is important, however, that any resultant inconsistencies, such as those discussed above, be resolved if the Standard is to serve its intended purpose as an unambiguous guide for implementors and users.

Of further concern is the impractical degree of freedom that is permitted in the interleaving of processes and the resulting unpredictable behavior. Our final example, `MOD5` of the preceding section, suggests that it may be difficult to write a Verilog program that is both portable, as currently defined by the Standard, and useful. If this situation is to be remedied, prevailing implementation policies and programming practices must be considered more closely.

This note is written with the hope that our observations and suggestions may be of use in the drafting of a future version of the Standard, especially in a more complete formulation of the simulation cycle than the outline presented here. Ultimately, this might benefit not only the implementor or experienced user of Verilog, but also the novice who, like the author, is merely interested in gaining an understanding of its basic features.

References

[1] Bui, Dinh K. (Synopsys, Inc.), Private communication, July, 2004.

[2] Cummings, Clifford E. (Sunburst Design, Inc.), Private communication, July, 2004.

[3] Gordon, Mike, "The Semantic Challenge of Verilog", Tenth Annual IEEE Symposium on Logics in Computer Science, 1995.

[4] Institue for Electrical and Electronic Engineers, *IEEE Standard Verilog Hardware Description Language*, Standard 1364, 2001.