Achieving Determinism in SystemVerilog 3.1 Scheduling Semantics

Phil Moorby, Arturo Salz, Peter Flake, Surrendra Dudani, Tom Fitzpatrick
Synopsys, Inc.

Abstract

The SystemVerilog 3.1 language initiative includes the requirement to bring together design, testbench, and assertion-based descriptions into a consistent, backward compatible, evolutionary language standard that achieves determinism and common semantics across the spectrum of design and verification tools. This paper describes the algorithm proposed for the new scheduling semantics that meets these demanding requirements. This algorithm achieves predictable and consistent results across design and verification tools. This proposal has been donated to the Accellera SystemVerilog 3.1 standardization technical committees. Standardization of this new algorithm will extend these benefits to users of other tools.

1. Introduction

The increasing complexity of today's designs is turning the verification process into the most critical bottleneck in the design flow. In addition, the changes taking place in the design of System-on-Chips are making traditional verification methodologies obsolete. To address these problems, the SystemVerilog 3.1 initiative strives to increase the verification capabilities of Verilog substantially by incorporating testbench features and assertions in the same language.

Assertion-based verification is gaining considerable momentum, as promises to improve the effectiveness of today's simulation-based verification methodologies by incorporating knowledge of the designer's intent in the verification process. Several proprietary assertion specification formats exist today. Unfortunately, all of these formats are inherently different from Verilog, and this makes it difficult for designers to adopt them in their designs. With the addition of assertions in the SystemVerilog standard, the industry has taken the first step toward establishing a standard mechanism that allows assertions to be specified once, and used in different tools, from simulation to formal verification.

A critical problem of adding new and more abstract features to an event-based language is dealing with the race conditions that arise from the interactions between new and existing constructs. A simulation race occurs when multiple events happen at the same time and the outcome depends on the order of the events. Races in Verilog have always been of two kinds: the real ones in the hardware, and the ones that are purely an artifact of the timing abstractions used to model concurrency. Verification tools should assist in discovering the former while minimizing the latter. Towards that end, our proposed algorithm aims to separate the execution of verification and design code into three distinct categories: design, assertions, and testbench.

One of the most misunderstood aspects of Verilog is its event-driven simulation algorithm, and its potential for race conditions. As a design moves into the system verification phase, several purported safe-coding styles may come together, and cause verification engineers to spend many frustrating hours debugging obscure simulation race conditions. With the advent of reusable verification and implementation IP, often from different providers, the problem of race conditions between the design and the testbench is often exacerbated.

Races between the design and the testbench are usually unrelated to the physical races that may be present in the hardware [5]. More commonly, races between the design and the testbench are introduced by the simulation algorithm in what may be a correctly working system. Engineers avoid some of these races by writing parts of the testbench in C and making use of various PLI calls to synchronize their execution. A common technique is to synchronize execution of testbench code to the end of the time-slot, at a point when the testbench inputs are stable and the testbench code can compute a

The authors want to make special mention to the many constructive discussions with the members of the Accellera SystemVerilog 3.1 standardization technical committees, and in particular Jay Lawrence, Joao Geada, David Smith, Dennis Brophy, Clifford Cummings, Matt Maidment, Neil Korpusik, Taj Singh, Mehdi Mohtashemi, Bassam Tabbara, and Doug Murphy.

response. Another technique is to synchronize to the beginning of the time-slot in order to sample and save the values of various design variables before they are modified. Formalizing these synchronization semantics in SystemVerilog's scheduling algorithm allows the same functionality to be written directly in SystemVerilog without resorting to a C interface for special semantics.

To fully support clear and predictable interactions, the proposed algorithm divides a single time slot into multiple ordered regions of executions. New regions allow properties and checkers to sample data when the design under test is in a stable state. This enables assertions to be safely evaluated, and testbenches to react to both properties and checkers with zero delay, all in a predictable manner. In the proposed algorithm, this determinism is accomplished by adding three new execution regions to the Verilog scheduling algorithm: preponed, observe, and reactive. This same mechanism also allows for non-zero delays in the design, clock propagation, and/or stimulus and response code to be mixed freely and consistently with cycle accurate descriptions.

2. A brief history of simulation races

Over the past few decades, there have been various implementation approaches to the simulation of digital hardware systems, and the event-directed simulation technique has emerged as the most widely used. The major underlying motivation for inventing new techniques has been the need to increase simulation performance. When the first version of Verilog was created in 1984/85, gate-level primitives were the dominant form of modeling digital logic [2]. There were many kinds of gate-level simulators in use 15 years ago, and all of them had problems with simulation races, especially when zero-delay networks were used in the modeling.

Event-directed simulation algorithms model hardware timing approximately, ranging from accurate primitive and interconnect delays for ASIC sign-off simulation to zero-delay for verifying the functionality while leaving the timing analysis to other tools. The use of zero delay modeling has become one of the most popular techniques in the simulation of very large digital systems. An extreme form of zero-delay simulation is cycle-simulation, in which all the zero-delay combinational logic is levelized into a statically defined evaluation order, and the state variables are updated by performing a simple next-state sequence control; the clock signals are abstracted out. Cycle-simulators promise significant performance benefits because various kinds of optimization techniques open up to the simulation implementors. However, cycle-simulators suffer from serious race problems that are very hard for users to avoid. Most of today's leading-edge digital simulators use a combination of techniques, including event directed and cycle-simulation.

In a Verilog simulator, when a variable is updated with a new value, the simulator evaluates the primitives and processes that are sensitive to the changing value. The order in which the primitives and processes are evaluated is not defined in the language standard. Also, if these evaluations schedule other events with zero delay, the order of these new events is also not defined. So what execution order can the user rely on to predict the outcome of potentially racy situations?

To illustrate a simple zero-delay potentially racy situation, consider the circuit in figure 1. This is a two-stage shift register that uses D-type flip-flops and a common clock signal. In Verilog, there are several ways to model the flip-flops. These flip-flop networks require careful placement of non-zero delays to create predictable, race-free models.

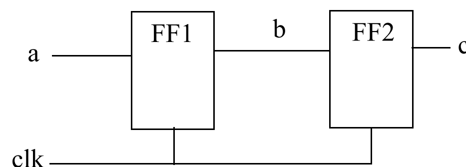

Figure 1. A 2-stage shift register

Verilog introduced more convenient methods for coding flip-flops. User defined primitives (UDP's) was one method that focussed on simulation performance and flexibility in modeling the logic, and allowed a flip-flop to have zero-delay. In the example in figure 1, if the flip-flops are implemented with UDP's, a clock signal on **clk** will cause both flip-flops to be evaluated in some unspecified order. It is therefore important in the evaluation of **FF1** that, if **b** is to change value, the update of **b** happens after the evaluation of **FF2**, otherwise the sampling of **b** by **FF2** will incorrectly read the new value of **b**. In this network structure, the correct behavior is guaranteed because the writing to **b** will occur in a separately scheduled event after both flip-flops have sampled their data inputs. Care does need to be taken, however, not to introduce extra logic in the clock fanout. For instance, a zero-delay gate in the clock fanout leading to **FF2** will introduce an unpredictable race.

A conceptual waveform view of the correct sequence of actions that must be taken is illustrated in figure 2.

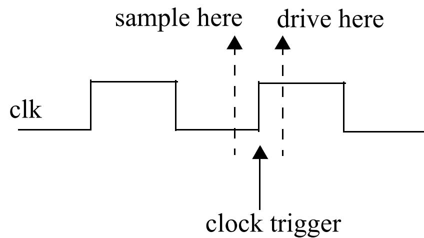

Figure 2. The data sampling and driving sequence in the shift register

3. Eliminating design races with NBA's

Another, more popular, method of modeling flip-flops is with always statements [3], and one convenient way to write the shift register example is

```
always @(posedge clk) begin
   c = b;
   b = a;
end
```

Here, the sequence of assignments guarantees that the order of reading and writing the variables produces the correct result. This type of coding became popular because it had the advantage of being easily understood and synthesizable, even when the right-hand-side expressions became complex and included function calls. A problem, however, that developed is that the flip-flops were coded separately:

```
always @(posedge clk) c = b;
always @(posedge clk) b = a;
```

Now a simulation race has been introduced because the evaluation order of the processes on the clock fanout is unspecified, even though the code is interpreted correctly for synthesis. In simulation, it is possible for the assignment to **b** in the second always statement to be completed before the assignment in the first always statement reads the value of **b**. An early workaround to this problem was to insert temporary variables, as in

```
assign #1 c = cTemp;
always @(posedge clk) cTemp = b;
assign #1 b = bTemp;
always @(posedge clk) bTemp = a;
```

Ugly, but it worked, and gave the same simulation results as the synthesized code. It was not long before those users, who insist on modeling exclusively with zero-delay, tried substituting #0 for the continuous assignments. This actually also works, only because of the typical event ordering techniques used in event-directed simulators. Eventually, in order to preserve this coding style, the semantics of #0 was written into the IEEE Verilog standard.

Inserting temporary variables, and especially extra delays, leads to code bloat and degrades simulation performance. So this problem was the compelling reason to introduce non-blocking assignments (NBA's) into Verilog. NBA's provide a well defined region in the time slot, after all the design clock signals have propagated and clock triggered processes have executed, but before time advances. The mechanism also allows the simulator to optimize the execution. The coding with NBA's is simply

```
always @(posedge clk) c <= b;
always @(posedge clk) b <= a;
```

The simulator will implicitly save the right-hand-side values, and suspend updating the left-hand-side variables until both processes have read the right-hand-side variables. The updates of the left-hand-side variables are then predictably applied with zero-delay in the NBA region of the time slot.

4. Assertions

When used correctly and consistently, NBA's have been largely successful in eliminating many simulation races. Nonetheless, the addition of NBA's to Verilog entailed the creation of the NBA region in the simulator's event scheduler. To show that additional event ordering is required for assertions [7], we will analyze two types of assertions: *continuous invariant assertions* and *clocked assertions*.

Continuous invariant assertions describe a Boolean expression that must be true at all times. These assertions are sometimes called combinational assertions. An example of an invariant assertion that ensures non-overlapping clocks is

```
(clk1 && clk2) == 0
```

When the variables in the expression change in the same time slot, which will happen in a zero-delay simulation model, a "false firing" may occur due to a race. For example, in the same simulation time slot, **clk1** may change to 1 before **clk2** changes to 0. With the correct delays, of course, this condition should not occur. However, users demand zero-delay modeling for RTL simulation. Thus, the reading of the variables and the evaluation of the expression must wait until all potential value changes on the variables have completed. Since the variables could be driven with NBA assignments, the invariant evaluation must be done after the NBA

region, at a point when the design is stable. Yet, there is no mechanism to do this in Verilog. Instead, the PLI read-only synchronization callback [4] is used by add-on tools to get to the end of time slot region, but as we shall see this is a limited mechanism.

The proposal for SystemVerilog 3.1 defines a new region called the *observe* region, in which invariant assertions can be safely evaluated.

Clocked assertions are synchronized to some clock signal. To illustrate why it is necessary to be clear about the scheduling semantics with clocked assertions, we re-code the shift register shown in figure 1 as

```
assign #0 gclk = clk;
always @(posedge gclk) b = a;
always @(posedge clk) c = b;
```

According to the IEEE Verilog standard this coding is guaranteed to simulate correctly and is also code that synthesizes correctly. The first always statement will execute after the second one because of the semantics of the #0 delay in the clock line.

We now specify a temporal assertion about this design with the following SystemVerilog code

```
sequence @(posedge clk) sa = (!a ; a ; !a);
sequence @(posedge clk) sc = (!c ; c ; !c);
property p = (sa  => [2]sc);
assert (p) pass_statement; else fail_statement;
```

This group of concurrent statements asserts that a clocked 010 data sequence on the input **a** implies that the output **c** should experience a clocked 010 sequence, 2 cycles later. The important questions that must be answered about the semantics of these statements are

1) Which values of **a** and **c** should be used?
2) When does the property **p** get evaluated?
3) When do the pass/fail statements execute?

The first question is straightforward. The values of **a** and **c** must be *sampled* before any events can change their value in the time slot in which the clock triggers. The proposal for SystemVerilog is for this sampling to occur within a *preponed* region that exists at the beginning of the time slot. Simulator implementations may optimize the actual sampling mechanism to take place elsewhere, as long as the semantics are preserved.

The answer to the second question is flexible. Since the sampled values of **a** and **c** do not change throughout the time slot, the property may be evaluated anytime after detecting the clock trigger and before the end of the observe region, along with the invariant assertions. The only requirement is that the outcome of the assertion be predictable and available by the end of the observe region. It is interesting to note that various properties or sequences clocked by the same clock may exhibit ordering requirements. For example, **sa** and **sc** must be evaluated prior to **p**.

To answer the third question, we must consider what kinds of actions are allowed in the pass/fail statement. It is a requirement, that code executed by assertions must not be allowed to change the state of the design. In our proposal, we regard the pass/fail statements as reactive testbench code, and create a new region called the *reactive* region in which this code is executed. Events scheduled into the reactive region are executed after the observe region, and before time advances.

5. Design and testbench interactions

To incorporate assertions into SystemVerilog, we proposed extending the scheduling algorithm to include three additional execution regions: preponed, observe, and reactive. This section examines if the existing regions are sufficient to satisfy the demands of the testbench.

The main function of a testbench is to generate stimulus for the design, and check the results in order to verify that the design conforms to its specifications. A testbench typically includes numerous models, often written at various levels of abstraction, and sometimes requiring special synchronization semantics. Typically, some models need to synchronize to a stable point in the time slot [6]. For example, models of large pass transistor networks, whose repeated evaluation can severely degrade performance, and asynchronous memory models that can trigger special code when certain memory locations are accessed. Thus, it is at the testbench level that many different methodologies come together and exacerbate the need for predictable behavior. We illustrate this with a simple example.

The design example shown in figure 3 consists of an arbiter with two serial channels: channel1 and channel2. The number of channels can easily be increased, but we illustrate just two in the example. Each channel is driven by its own independent clock and is capable of buffering one message or data packet. When a full message is received, the channel raises its request line to gain

access to some common resource, which could be a parallel bus, another serial channel, or some other device

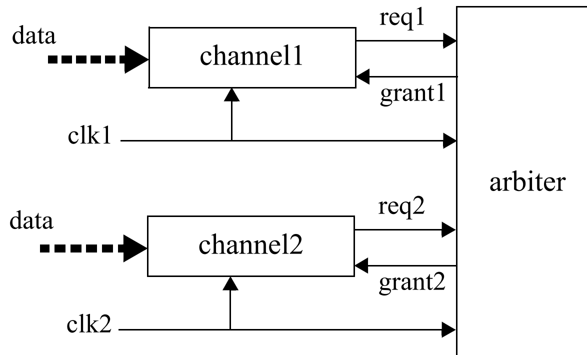

Figure 3. The 2-channel arbiter example

The role of the arbiter is to grant access to the shared resource by raising the corresponding grant signal and keeping it that way until the request is removed. In addition, the arbiter must enforce a policy for granting access to the shared resource. The hardware that implements the arbiter will certainly need special circuitry to synchronize the various clocks. That hardware can take several forms depending on the arbiter's policy: a simple daisy chain for a fixed priority policy, or a more complex synchronizer that implements a round robin or load balancing policy. Regardless of the policy in use, a testbench for the arbiter needs to ensure that the arbiter correctly implements the policy under various scenarios. Thus, to ensure that various clocking conditions are covered, the testbench must maintain some coverage metrics.

A testbench will not replicate the synchronization hardware; instead, it will implement the policy checker at a higher level of abstraction. This is simpler and desirable because a higher level description that verifies the synchronization avoids repeating in the testbench the same bugs as in the implementation.

A simplified testbench for this 2-channel arbiter can be written as:

```
// sample requests with corresponding clocks
always @(posedge clk1) arbReq1 <= req1;
always @(posedge clk2) arbReq2 <= req2;

// arbiter testbench
always @(arbReq1 or arbReq2) begin
   case ({arbReq2, arbReq1})
      ...        // policy checker
   endcase
end
```

The case statement analyzes the current state of both request lines simultaneously, in an attempt to catch the various clocking conditions. Note that correct modeling of the arbiter protocol requires that the case statement execute only once per time slot. However, because **clk1** and **clk2** may trigger at arbitrary times during the execution of the time slot, that statement may be executed multiple times. If the arbiter testbench were written in C, this race condition could easily be overcome by delaying execution of the testbench until all clocks have triggered [6]. But, until now, this testbench could not be written in standard Verilog code.

It's important to note that if the two clocks do not trigger at the same time during the test then it is still the responsibility of the testbench to report that fact as a coverage hole. Therefore, even to maintain a simple coverage metric (i.e., how many channels triggered simultaneously), a testbench written in standard Verilog code could misbehave and yield incorrect coverage metrics.

The new scheduling semantics has the reactive region in which a SystemVerilog testbench can execute the example code in a predictable manner.

The example above can be made more interesting if the request lines themselves are not the outputs of some models, but are instead generated by a set of assertions that model the system via state machine transitions of an abstract system.

```
property p1 = @(posedge clk1)
   ((!reset ;[2]idle) => (!busy;[1:15] rdy1));[1]
property p2 = @(posedge clk2) ...

always @(p1 or p2) begin
   arbReq1 = p1;   // get property status
   arbReq2 = p2;
   // arbiter testbench
   case ({arbReq2, arbReq1})
     ...
   endcase
end
```

This example will not work correctly unless the assertions execute before the arbiter code. This is a key observation because it introduces an ordering constraint on the execution of SystemVerilog assertions and testbench code. This ordering constraint enforces the need for a separate reactive region that can react to the outcome of the assertions. Since assertions are forbidden from modifying the state of the system as a side effect of their execution, they delay execution of pass/fail state-

1. This is proposed syntax that is subject to change.

ments until the reactive region in which the rest of the testbench executes.

6. The complete new scheduling algorithm

We have introduced three new regions to the existing Verilog standard regions for evaluating assertions and executing testbench code. This section describes the algorithm that compliant SystemVerilog simulators should follow. The semantics of executing Verilog code is actually extremely complex, and a user does not need to understand every detail of how a simulator is implemented. So to produce sufficient semantic content, we need to understand the major principles guiding how detailed the semantics should become. They are:

1) Provide the user with sufficient conceptual information so that it is possible to predict the behavior of well-constructed code consistently across design and verification tools.

2) Provide the tool implementors sufficient information so that tool performance can be maximized whilst maintaining the predictable behavior of well-constructed code.

3) Provide an event-scheduling framework that sufficiently defines PLI callback points so that writers of PLI code can reliably predict which simulation events have executed.

The proposed scheduling algorithm for SystemVerilog is an extension to the IEEE Verilog 1364-2001 standard [1, section 5]. A time slot is divided into a set of ordered regions, where each region is a container for a particular set of simulation events. The order in which events are executed in any given region is undefined by the semantics. A flow diagram showing conceptually the order of executing the regions is depicted in figure 4.

The active, inactive, NBA, observe and reactive regions are known as the *iterative* regions.

The preponed and postponed regions are for registering PLI callbacks.

The preponed region enables PLI code to access simulation data in a time slot before a variable can change state or process executes. The preponed region is also the region where sampling of steady state data takes place.

The postponed region enables PLI code to be suspended until after all the iterative regions have completed. This is where the Verilog standard tf_rosynchronize and cbReadOnlySynch calls are made.

The active, inactive, and NBA regions are standard and unchanged from the 2001 standard.

The observe region is where invariant assertions may be evaluated. It is also the last region in which clocked assertions may be evaluated.

The reactive region is where testbench code is executed, including the pass/fail statements of assertions.

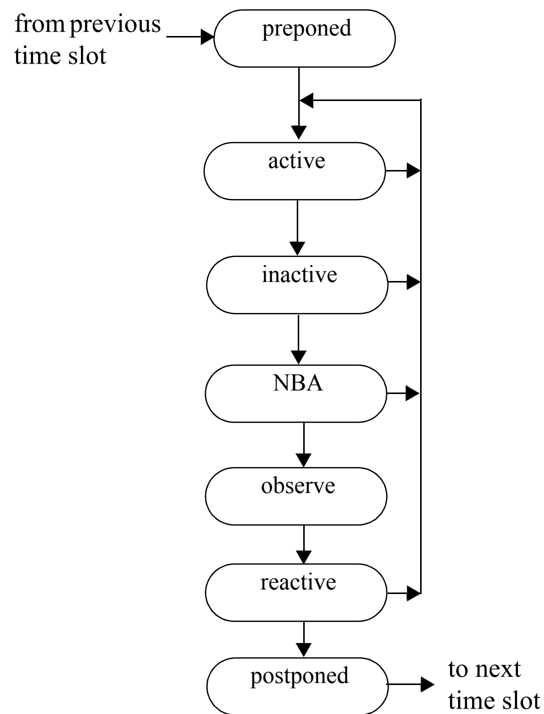

Figure 4. The SystemVerilog flow of event regions within a time slot

7. Conclusions

By unifying testbench features and assertions together with design into one language, SystemVerilog aims to substantially reduce the verification bottleneck. Standardizing the semantics of these verification-specific features will assure users that the functionality verified by assertions via simulation will provide the same results as proving those assertions with formal tools. By embedding the verification code, as assertions, directly in-line with the design, it can be used throughout the verification process by multi-disciplined teams. This integration of testbench and assertions allows complementary verification methodologies that seamlessly bring formal verification and simulation together in a way not previously possible.

To guarantee predictability and consistency between design, testbench and assertions, SystemVerilog 3.1 requires a unified and deterministic scheduling algorithm. We have presented a new algorithm that extends the Verilog standard algorithm in the form of ordered execution regions. The regions are organized to allow deterministic simulation results that correspond precisely between event and formal semantics. Backward compatibility is assured by the addition of three new regions to the scheduler: a preponed region, an observe region, and a reactive region.

The scheduling semantics formalizes the execution regions, and enables access to them from within the SystemVerilog language. This enables functionality to be coded directly in SystemVerilog, without resorting to the PLI.

Using one language for both design and verification lessens the need for verification environments designed using combinations of Verilog and PLI, enables the same code to be used consistently across all design and verification tools, and also brings about significant performance and productivity improvements. The proposed scheduling algorithm makes all this possible by adding three extra regions of execution to the standard Verilog scheduler.

8. References

[1] "IEEE Standard Verilog Hardware Description Language", IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001.

[2] Donald E. Thomas, Philip R. Moorby, "The Verilog Hardware Description Language", Fifth Edition, Kluwer Academic Publishers, 2002.

[3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!", SNUG-2000.

[4] Stuart Sutherland, "The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface", Second Edition, Kluwer Academic Publishers, 2002.

[5] Janick Bergeron, "Writing Testbenches, Functional Verification of HDL Models," Kluwer Academic Publishers, 2000.

[6] Lee Tatistcheff, Charles Dawson, David Roberts, Rohit Rana, "The Facts and Fallacies of Verilog Event Scheduling: Is the IEEE 1364 Standard Right or Wrong?". DVCon 2002.

[7] L. Bening, H. Foster, "Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog", Kluwer Academic Publishers, 2001.