



World Class Verilog & SystemVerilog Training

SystemVerilog Event Regions, Race Avoidance & Guidelines

Clifford E. Cummings

Sunburst Design, Inc.
cliffc@sunburst-design.com

Arturo Salz

Synopsys
Arturo.Salz@synopsys.com

ABSTRACT

The IEEE1800 SystemVerilog Standard includes new event regions primarily added to reduce race conditions between verification code and SystemVerilog designs. The new regions also facilitate race-free Assertion Based Verification (ABV).

This paper details common Verilog verification strategies and how the new event regions facilitate construction of race-free testbenches using new SystemVerilog capabilities. An in-depth explanation of SystemVerilog event regions is included to help understand how race-reduction goals have been met. Important design & testbench coding guidelines are also included.

Table of Contents

1	Introduction.....	4
1.1	Who wrote the SystemVerilog Standard?.....	4
1.2	Watch this space!.....	4
2	Event Regions - Verilog-2001 -vs- SystemVerilog.....	5
2.1	What is a Race?.....	6
2.2	SystemVerilog Event Regions.....	11
2.2.1	Preponed Events Region.....	11
	The Active Region Set (Active - Inactive - NBA).....	12
2.2.3	Active Events Region.....	13
2.2.4	Inactive Events Region.....	13
2.2.5	Nonblocking Assignment Events Region (NBA).....	13
2.2.6	Observed Region.....	14
	The Reactive Region Set (Reactive - Re-Inactive - Re-NBA).....	14
2.2.8	Reactive Events Region.....	15
2.2.9	Re-Inactive Events Region.....	15
2.2.10	Re-NBA Events Region.....	16
2.2.11	Postponed Region.....	17
2.2.12	Preponed - Postponed Naming.....	17
2.2.13	#1step - what it really means.....	17
3	Time-0 events, races & race-avoidance.....	22
3.1	Synchronous or asynchronous resets?.....	22
3.2	Clock startup issues.....	22
3.3	Startup delays.....	22
3.4	Input Initialization.....	23
3.5	Capturing startup vectors.....	23
3.6	Functional verification startup matrix.....	24
3.7	4-state variables.....	25
3.8	2-state variables.....	25
4	Sunburst Design recommended and non-recommended clock oscillators.....	26
4.1	Clock oscillators - put them in the top module.....	29
5	How clocking block cycles & delays work.....	30
5.1	Clocking block and non-clocking block timing of the same signals.....	30
5.2	Clocking blocks in SystemVerilog interfaces.....	35
5.2.1	Interface modports and clocking blocks.....	36
5.3	Could clocking blocks lead to a better Verilog?.....	36
6	When to apply stimulus.....	39
6.1	Verilog-2001 designs and testbenches.....	39
6.2	SystemVerilog designs and testbenches.....	39
7	Summary & Conclusions.....	40
8	Acknowledgements.....	40
9	References.....	40
10	Revision Changes.....	41
10.1	Revision 1.1 (November 2007) - What Changed?.....	41

10.2	Revision 1.2 (December 2007) - What Changed?	42
11	Author & Contact Information	42

Table of Examples

Example 1 - Scheduling of #0 Re-Inactive testbench event	16
Example 2 - Simple test used to show sampling and #1step sample delays	19
Example 3 - 2-state clock oscillator with no time-0 event	25
Example 4 - CYCLE and `timescale definitions used for clock oscillators	26
Example 5 - clkgen1 clock oscillator	26
Example 6 - clkgen2 clock oscillator	26
Example 7 - clkgen3 clock oscillator	27
Example 8 - clkgen3a clock oscillator	27
Example 9 - clkgen4 clock oscillator	28
Example 10 - clkgen5 clock oscillator (not recommended)	28
Example 11 - clkgen6 clock oscillator (not recommended)	28
Example 12 - program block with mixed non-clocking and clocking delay assignments	31
Example 13 - Equivalent cycle delays using ##-notation and @(cb1) notation	35
Example 14 - Example interface code with two clocking blocks	35
Example 15 - interface code with modports that reference clocking blocks	36
Example 16 - clocking block to apply stimulus on the negedge of the clk	39

Table of Figures

Figure 1 - Verilog-2001 event regions	5
Figure 2 - Basic NAND S-R latch & De Morgan's equivalent	6
Figure 3 - Sunburst Design's - 8 Coding Guidelines to Avoid Verilog Race Conditions	7
Figure 4 - SystemVerilog-2005 event regions with PLI regions shown	8
Figure 5 - SystemVerilog-2005 event regions with PLI regions omitted	9
Figure 6 - Simplified event regions diagram	9
Figure 7 - Sampling mechanism block diagram	12
Figure 8 - Active region set	12
Figure 9 - Reactive region set	14
Figure 10 - #1step defined by smallest `timescale & timeprecision	18
Figure 11 - How events are scheduled and sampled for the code in Example 2	20
Figure 12 - Example 12 event scheduling from time 0ns to time 7ns	32
Figure 13 - Example 12 event scheduling from time 7ns to time 15ns	32
Figure 14 - Example 12 event scheduling from time 15ns to time 20ns	33
Figure 15 - Example 12 event scheduling from time 20ns to time 27ns	33
Figure 16 - Example 12 event scheduling from time 27ns to time 35ns	34
Figure 17 - Example 12 event scheduling completion at time 35ns	34
Figure 18 - Example 12 output waveforms	35
Figure 19 - A 2-stage shift register of flip flops	37

1 Introduction

New SystemVerilog event regions have been added to help eliminate race conditions that could occur between design modules and verification environments.

SystemVerilog is fully RTL backward compatible with Verilog. This means that SystemVerilog is fully race-backward compatible with Verilog!!

Understanding SystemVerilog event regions and fundamental coding guidelines can help eliminate race conditions from SystemVerilog designs, testbenches and the interaction between the design and the enhanced SystemVerilog Hardware Verification Language (HVL).

1.1 Who wrote the SystemVerilog Standard?

The content of the SystemVerilog standard is the culmination of efforts of many talented engineers working primarily on one or more of the following subcommittees with the designated responsibilities:

- SV-AC - Assertions Committee - largely responsible for the definition of the SystemVerilog assertion and functional coverage constructs.
- SV-BC - Basic & Design Committee - largely responsible for RTL enhancements and errata updates to the SystemVerilog standard.
- SV-CC - C-Interface Committee - largely responsible for C-language Direct Programming Interface (DPI) and VPI enhancements to the SystemVerilog standard.
- SV-EC - Enhancements & Testbench Committee - largely responsible for maintaining and extending the System Verilog language for testbench support.

The fact that the SystemVerilog standard is the work of many talented individuals is both *good* and *bad*. *The good* is that multiple very talented experts contributed to the final functionality described in the standard. *The bad* is that occasionally the standard has the appearance of a "standard-by-committee," and various parts of the standard look like they were written by different people, because they were! Over time, the standard will be updated and clarified and in the process will become more consistent across the different disciplines described in the standard.

1.2 Watch this space!

It should be noted that although the IEEE 1800 SystemVerilog Standard[11] was ratified in 2005, there are some ambiguities that are currently being clarified by the IEEE SystemVerilog committee. Those clarifications might change some of the restrictions of programs, clocking blocks, event regions, as well as some of the behavior described in this paper. As updates become available, this paper will be updated to reflect those changes and will be posted to the **www.sunburst-design.com** web page.

Check back at the **www.sunburst-design.com/papers** web page to see if there have been important updates to this paper.

2 Event Regions - Verilog-2001 -vs- SystemVerilog

First we need to introduce a couple of definitions, *simulation time* and *time slot*.

"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 system description being simulated. The term *time* is used interchangeably with simulation time.[9]."

A *time slot* encompasses all simulation activity that is processed in the event regions for each simulation time. All simulation activity for a particular *simulation time* is executed until no further simulation activity remains for that *time slot*, that is, without advancing the *simulation time*. Note that execution of simulation events within a *time slot* may require multiple iterations through the simulation event regions for that same *time slot*. The IEEE Std 1800-2005 standard sometimes referred to a *time slot* as a *timestep*, but the term *timestep* has been removed from the P1800-2008 Draft Standard.

The Verilog event regions allow scheduling of Verilog simulation statements that model actual hardware behavior.

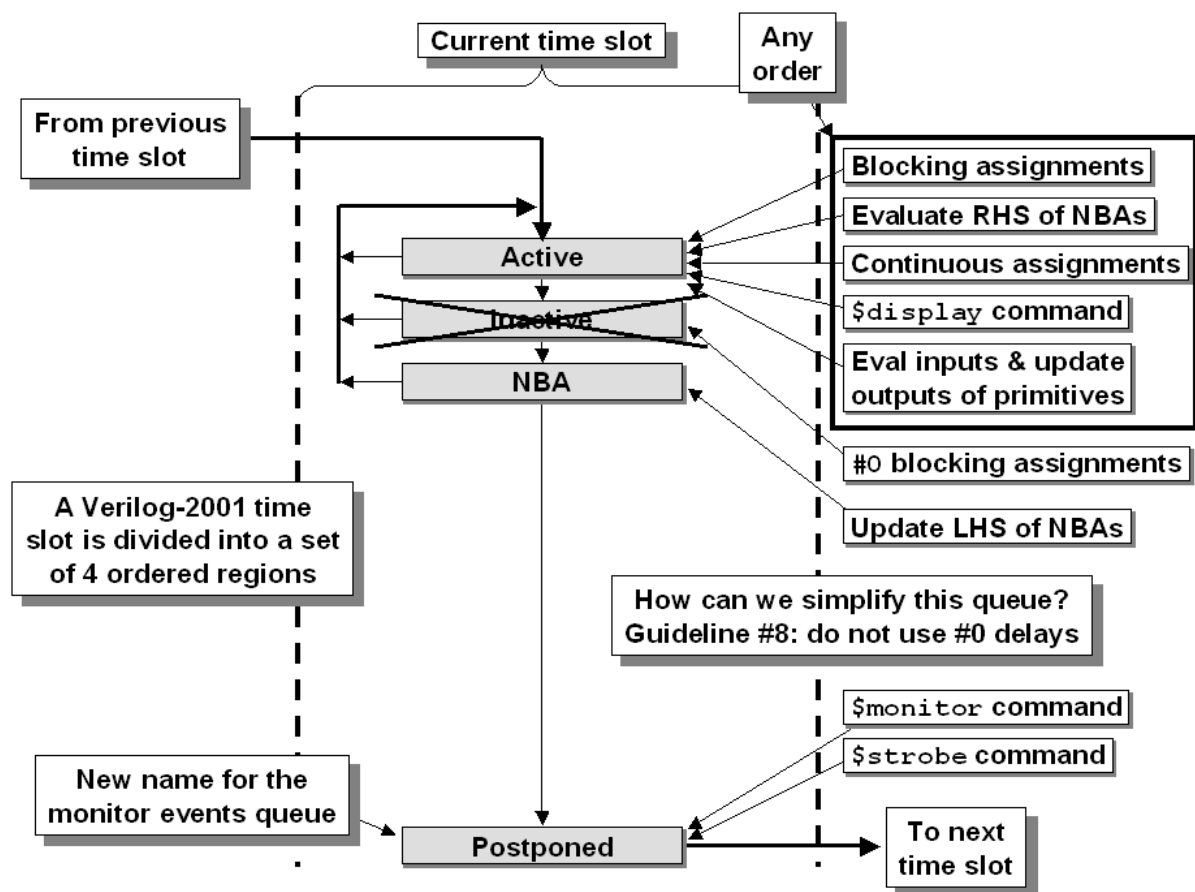


Figure 1 - Verilog-2001 event regions

The IEEE Std 1364-2001 divided the Verilog event regions into four ordered regions: *Active* events, *Inactive* events, *Nonblocking Assign Update* events, and *Monitor* events. The Verilog-2001 standard further defined *Future Inactive* events and *Future Nonblocking Assignment Update* events, which were nothing more than the *Inactive* and *Nonblocking Assign Update* event regions pre-built for a future time slot.

2.1 What is a Race?

A **race condition** is a flaw in a system or process that is characterized by an output that exhibits an unexpected dependence on the relative timing or ordering of events. The term originates with the idea of two signals racing each other attempting to influence the output first.

When simulating a design with Verilog (or any other event-driven simulation language), engineers are often faced with two fundamentally different types of race conditions, *hardware races* and *simulation induced* races.

Hardware races typically occur in combinational logic due to the physical nature of their electronics. For example, when the inputs to a logic gate change state, a finite delay transpires before the change is reflected on the gate's output. Therefore, for a brief time, the output may change to an unwanted state before settling to its final state. Other than wasting power, such transient races typically do not affect the correct operation of the system, except when they control a clock or an asynchronous reset signal. Another well known hardware race is the illegal state of a basic NAND S-R latch shown in Figure 2.

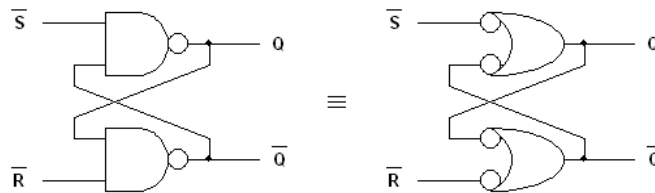


Figure 2 - Basic NAND S-R latch & De Morgan's equivalent

When the two inputs of the latch are set to logic 0 simultaneously, both NAND outputs are forced to logic 1, thereby overriding the feedback latching action. Upon leaving this state, whichever input stays at logic 1 longer will control the state of the latch. But, if both inputs transition to logic 1 simultaneously, the result is a race condition, and the final state of the latch is unpredictable. While hardware races are intrinsic to the physics of the system, proper design techniques and tools effectively eliminate the problems associated with them, and they are not the focus of this paper.

Simulation-induced races are not intrinsic to the design or its physics, but are a natural, although undesirable, consequence of the event-driven simulation algorithm used by Verilog. Because, the simulator processes events one at a time, it unavoidably serializes the events that occur in the same time slot. Hence, the design activity that in the actual hardware takes place concurrently is

instead modeled as a set of ordered actions by the simulator. This modeling deviation from the actual hardware induces additional races that are not present in the design, but are an artifact of the simulator. These races can cause the simulator to simulate a faulty design when in fact the design is correct, or more dangerously, simulate a seemingly correct design when in fact the design is flawed. Frequently, this latter type of a race occurs because the designer's code relies, often unwittingly, on the specific ordering of the simulation algorithm. It is precisely for this reason that Verilog specifies that a particular event region must be processed in an arbitrary order, but every implementation will exhibit a certain order.

There are several well established and well understood RTL coding guidelines that have been known for years. These were detailed in a paper describing Verilog nonblocking assignments, and, when followed, these guidelines will remove 90-100% of the Verilog race conditions induced on Verilog RTL designs[3]. These guidelines are shown in Figure 3.

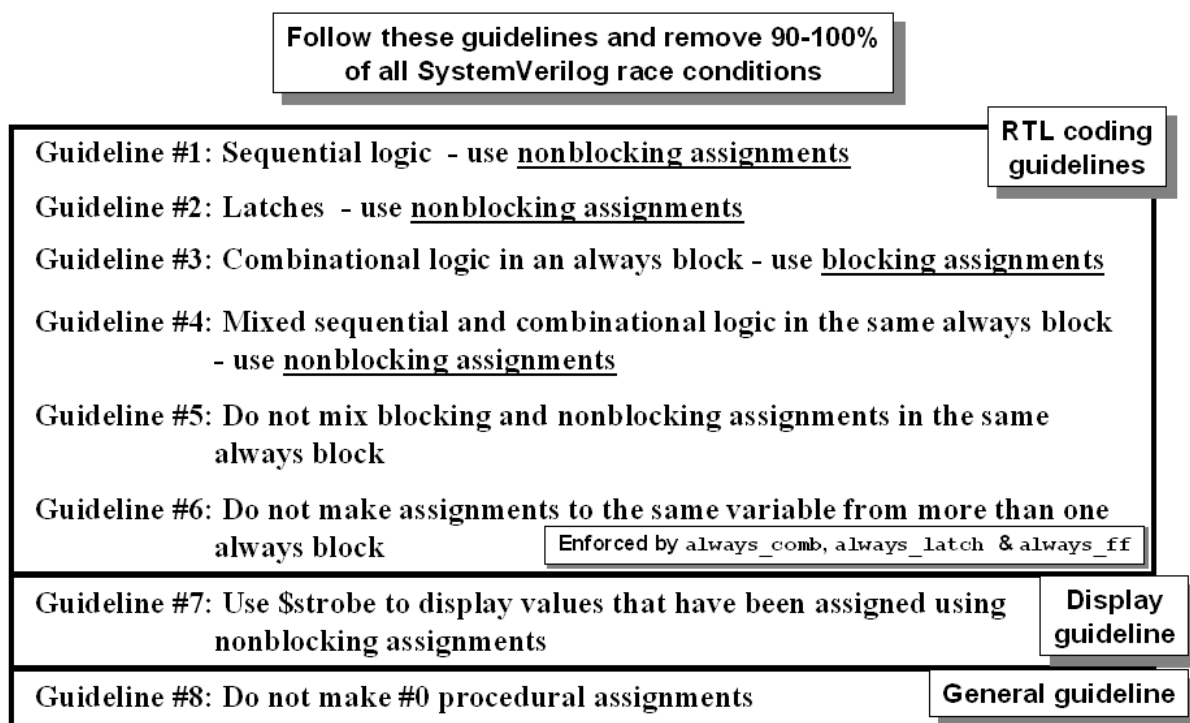


Figure 3 - Sunburst Design's - 8 Coding Guidelines to Avoid Verilog Race Conditions

Updates to IEEE Std 1800-2005¹ divide the SystemVerilog time slot into 17 ordered regions, nine ordered regions for execution of SystemVerilog statements and eight ordered regions for execution of PLI code (note some PLI commands can also be executed in the SystemVerilog statement regions). The new regions were specifically designed to support new constructs that

¹ See Mantis item number 890 in the Verilog & SystemVerilog Database for Tracking Bugs[19]

prevent additional simulation-induced races from being created between the RTL design and the new verification constructs.

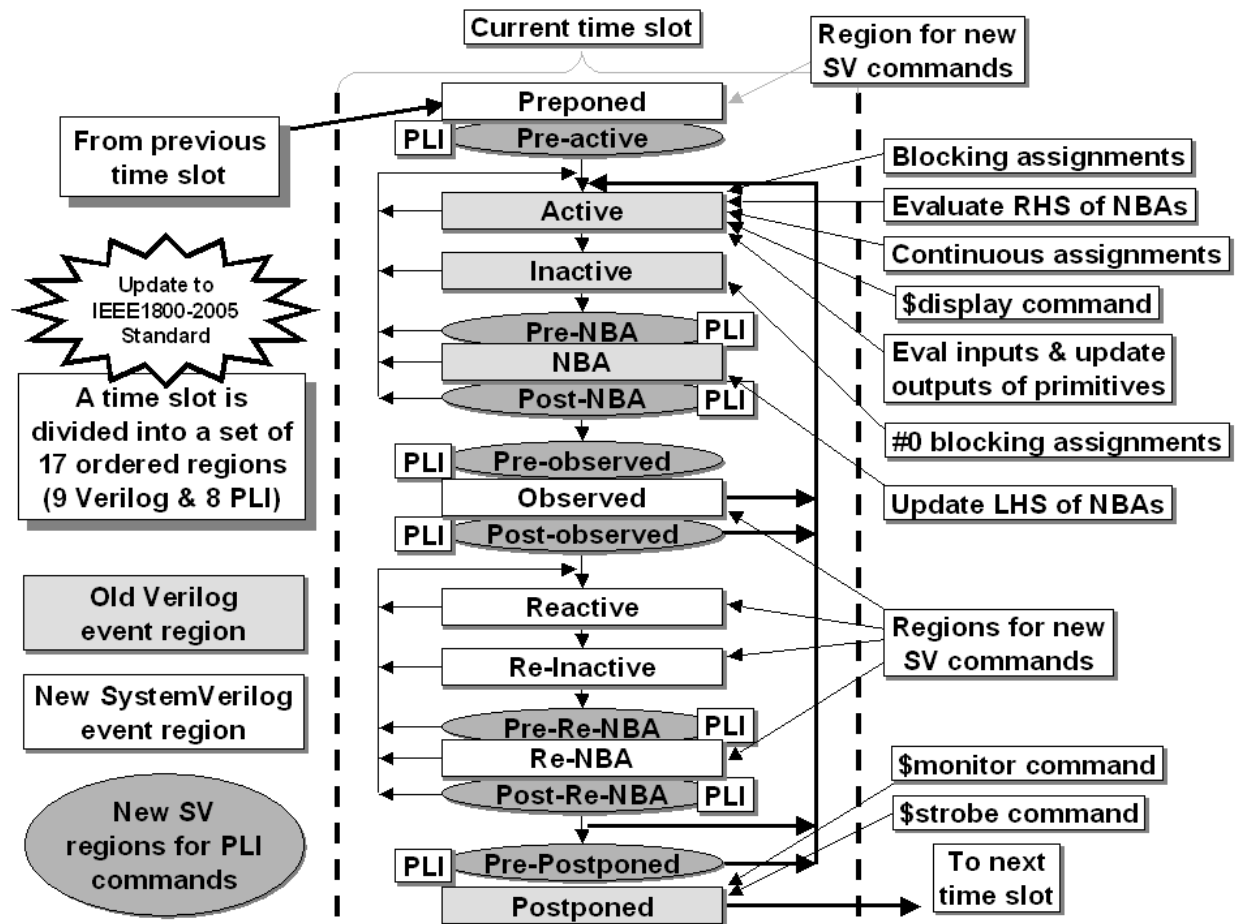


Figure 4 - SystemVerilog-2005 event regions with PLI regions shown

The timing and execution of PLI events is beyond the scope of this paper, so the modified SystemVerilog event-region diagram with PLI regions removed is shown in Figure 5.

Updates to IEEE Std 1800-2005 divide the SystemVerilog design and HVL event regions into nine ordered regions for execution of SystemVerilog statements: *Preponed* events, the RTL regions in the *Active region set* (*Active* events, *Inactive* events, *NBA* events), *Observed* events, the three verification regions in the *Reactive region set* (*Reactive* events, *Re-Inactive* events and *Re-NBA* events) and *Postponed* events. Each of these event regions may exist for each simulation time slot.

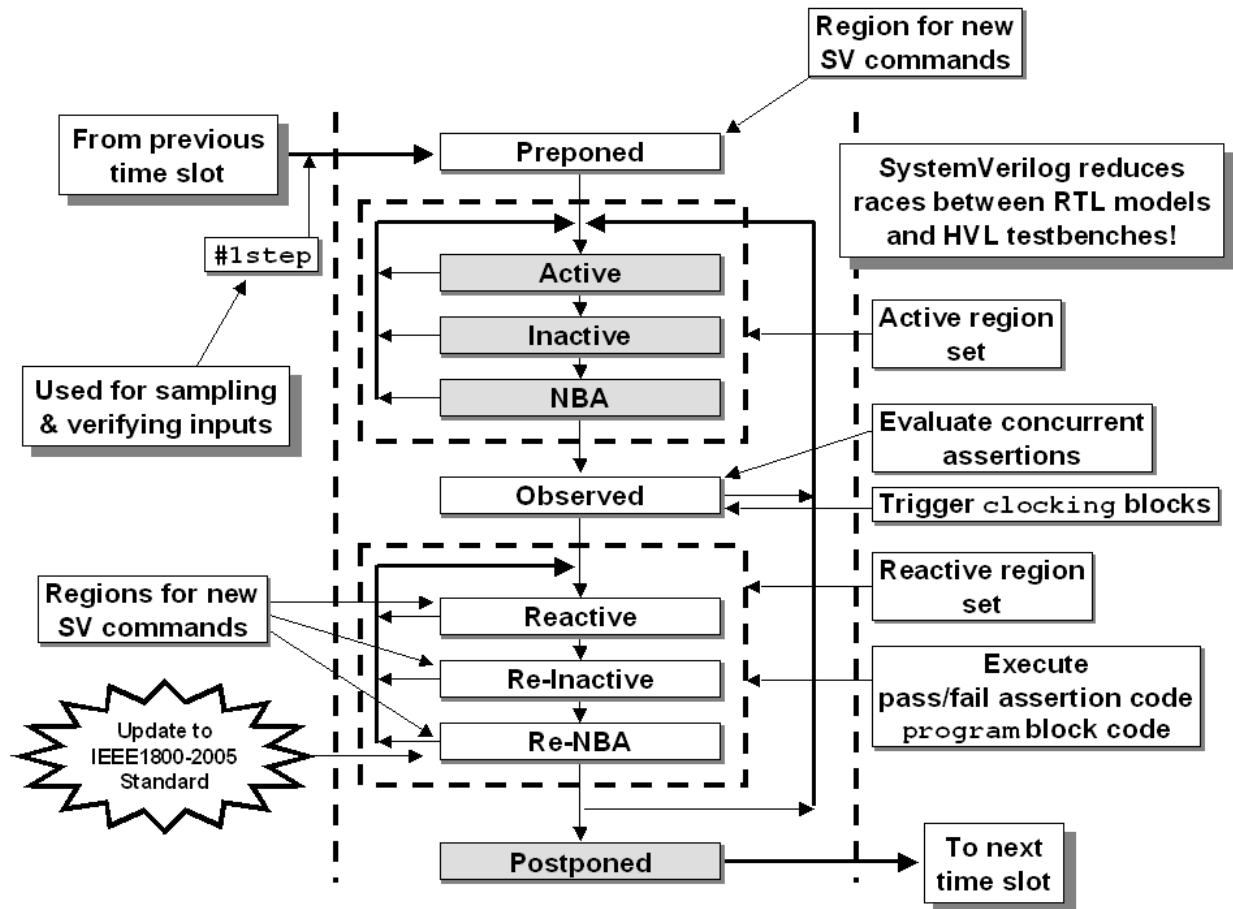


Figure 5 - SystemVerilog-2005 event regions with PLI regions omitted

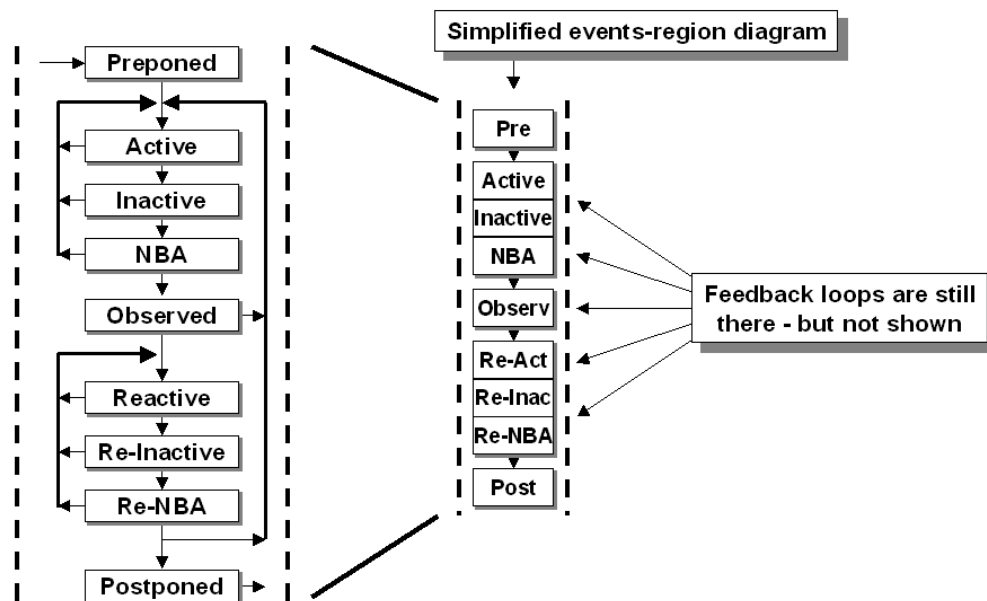


Figure 6 - Simplified event regions diagram

To help demonstrate how example-code is scheduled using the new SystemVerilog event regions, the simplified diagram in Figure 6 will be used later in this paper.

Before describing the activities that take place in each of these event regions, there are a number of frequently asked questions related to the new SystemVerilog event regions that are often posed when these event regions are viewed for the first time. It is useful to dispel the concerns before describing the event region activities so that an engineer is not distracted during the description of event region activities:

FAQ #1: Is the new event scheduling backward compatible with Verilog-2001?

ANSWER: Yes. The same even regions in their same respective scheduling order are still part of the new event region definitions so all existing code will execute the same as before (minor exception noted in FAQ #4).

FAQ #2: With so many new event regions in the SystemVerilog standard, will simulations run slower?

ANSWER: No. Vendors were quick to point out that all Verilog simulation events had to be scheduled even before the SystemVerilog scheduling modifications were made. The enhanced event regions simply give a better definition to describe where all of the events are executed, including an enhanced description of where specific PLI events should be scheduled. This definition was previously incomplete in the IEEE Verilog Standard but the events were already being scheduled as described in the IEEE Std 1800-2005. Now the event scheduling is better defined.

FAQ #3: Does the new SystemVerilog event scheduling standard complicate the task of scheduling simulation events? Are vendors willing to make these potentially difficult changes to event scheduling?

ANSWER: The new SystemVerilog event scheduling actually solves problems related to RTL-HVL race conditions that existed when coupling a Verilog design to a non-Verilog hardware verification language (HVL) or a C testbench through the PLI. Vendors welcomed the new scheduling definitions because they solved some of the persistent race conditions that frequently came up when coupling third party HVLs to Verilog designs. Since the HVL commands are now scheduled in specific event regions, they no longer have to be implemented through a PLI interface, which actually accelerates simulations that use SystemVerilog's HVL.

FAQ #4: Are there any differences in event scheduling or timing between the Verilog IEEE Std 1364-2001 and the SystemVerilog IEEE Std 1800-2005?

ANSWER: Yes. There is one minor but important timing difference. In Verilog-2001, variables that were initialized when declared were scheduled to execute at time-0 as if they had been assigned in an **initial** block, in a non-deterministic order and they would cause a time-0 event. There were no guarantees about time-0 startup of **always** blocks, **initial** blocks, continuous assignments and variable initializations so this was always a potential source for time-0 race

conditions. In SystemVerilog, variables declared with initialization assignments are guaranteed to be initialized before simulation starts at time-0; therefore, variable initialization does not cause any time-0 events. The ramifications of this change are discussed in greater detail in section 4.

2.2 SystemVerilog Event Regions

Below is a brief description of the activities that are scheduled into each SystemVerilog event region. More details are included in appropriate sections later in this paper and even more details are available in the IEEE Std 1800-2005[11] and in the Mantis-890 updates to SystemVerilog scheduling[19]. A quick overview of race-free RTL-design, verification and assertion activity can be summarized as follows:

Regions that are designed to implement **correct RTL functionality**:

- Active regions (Active, Inactive and NBA regions - but avoid Inactive region events).

Regions that are designed to implement **correct verification execution**:

- Preponed, Reactive regions (Reactive, Re-Inactive, Re-NBA) and Postponed regions.

Regions that are designed to implement **concurrent assertion checking**:

- Preponed, Observed, and Reactive regions.

Region that should be avoided:

- Inactive region.

Certainly verification code can be executed in the regions devoted to RTL functionality. Prior to the implementation of SystemVerilog, engineers doing Verilog-2001 verification implemented clever methods to ensure race-free operation between RTL designs and the pseudo-RTL testbenches. The new SystemVerilog regions just make it easier to implement a race-free verification environment.

2.2.1 Preponed Events Region

The stated function of this region is to sample values that are used by concurrent assertions. The Preponed region is executed only once in each time slot, immediately after advancing simulation time (there is no feedback path to re-execute the Preponed region).

There is some doubt as to whether an implementation actually must perform the sampling in the Preponed region or if the sampling may be done in the Postponed region of the previous time slot. Because both, Postponed and Preponed are read-only regions, the actual signal values are the same in any two contiguous Postponed-Preponed regions, thus, it is not observable in which region the simulator actually samples a value – the only value that is different is the simulation time.

According to the IEEE Std 1800-2005 - **17.3 Concurrent assertions overview**

"The values of variables used in assertions are sampled in the Preponed region of a time slot, and the assertions are evaluated during the Observed region. ..."

Sampled values are always defined with respect to a clocking expression. Therefore, it is only necessary to sample values in the Preponed region of the time slot in which the clocking expression is triggered, and not in every time slot. When processing in the Preponed region, how does the simulator know that a clock will be triggered later during the processing of that particular time slot? The answer is that the simulator does not need to know about any future events, it only needs to ensure that the values present in the Preponed region are available to the sampling constructs when the clocking expression is actually triggered while processing the latter regions. The simulator can accomplish this by maintaining two values for each sampled signal, its current value and its value when the Preponed region was processed. This way, when the sampling clock is triggered, the sampling construct simply uses the value corresponding to the Preponed region. While many optimizations are available to the simulator – including but not limited to peeking in the event queue for potential clocking events – the sampling mechanism can be illustrated as an intra-region (or a time slot) delay gate, as shown in Figure 7.

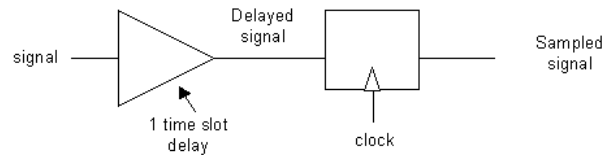


Figure 7 - Sampling mechanism block diagram

By changing the delay value, the above mechanism can be used to sample a signal an arbitrary amount of time before the corresponding clocking expression.

Assertions and **#1step** sampling are discussed in section 2.2.13 .

2.2.2 The Active Region Set (Active - Inactive - NBA)

The active region set is used to schedule blocking assignments and nonblocking assignments included in **module** code. Any task or function called from a **module** is also scheduled into the active region set. The intended purpose of the active region set is to schedule RTL and behavioral code activity. Testbench code can also be written as module code and indeed preexisting Verilog testbench code was written as module code, but SystemVerilog users are encouraged to place future testbench code into programs to isolate RTL design code execution from testbench code execution.

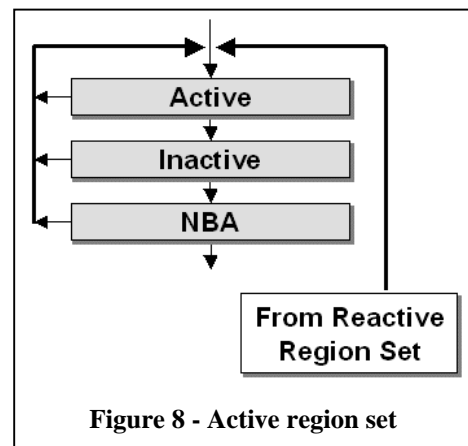


Figure 8 - Active region set

Active region set activities are described in sections 2.2.3, 2.2.4, and 2.2.5.

2.2.3 Active Events Region

The Active Events Region, also commonly called the Active region, is part of the Active Region Set. The principal function of this region is to evaluate and execute all current **module** activity. Including (in any order²):

- Execute all **module** blocking assignments.
- Evaluate the Right-Hand-Side (RHS) of all nonblocking assignments and schedule updates into the NBA region.
- Execute all **module** continuous assignments
- Evaluate inputs and update outputs of Verilog primitives.
- Execute the **\$display** and **\$finish** commands.

Sunburst Design Race Avoidance Guideline #3 dictates that all RTL combinational logic modeled using an **always** block should be coded using blocking assignments to ensure that combinational logic executes in the Active region and correctly models real combinational hardware behavior.

2.2.4 Inactive Events Region

The Inactive Events Region, also commonly called the Inactive region, is part of the Active Region Set. This region is where **#0** blocking assignments are scheduled and per Sunburst Design Race Avoidance Guideline #8, engineers should not make **#0** RTL procedural assignments³. Most engineers that continue to use **#0** assignments are trying to defeat a race condition that might exist in their code due to assignments made to the same variable from more than one **always** block, which is a violation of Sunburst Design Race Avoidance Guideline #6.

Engineers that follow good coding practices will have no need for **#0** RTL assignments and hence, the Inactive region is unused in the rest of the event scheduling examples in this paper.

2.2.5 Nonblocking Assignment Events Region (NBA)

The Nonblocking Assignment Events Region, also commonly called the NBA region, is part of the Active Region Set. The principal function of this region is to execute the updates to the Left-Hand-Side (LHS) variables that were scheduled in the Active region for all currently executing nonblocking assignments.

Sunburst Design Race Avoidance Guideline #1 dictates that all RTL clocked logic modeled using an **always** block should be coded using nonblocking assignments to ensure that the sequential logic will execute in the NBA region and correctly model the pipelined nature of sequential elements.

² Statements listed between begin-end do execute in the order listed

³ Historically, the Inactive region was necessary in the early days of Verilog, before the NBA region was added (circa 1989). Proper usage of the NBA region makes the Inactive region unnecessary.

Sunburst Design Race Avoidance Guidelines #2 and #4 also address RTL coding using nonblocking assignments to correctly model latches and sequential logic with simple combinational input logic. See references [3] and [5] for examples and details.

2.2.6 Observed Region

The principal function of this region is to evaluate the concurrent assertions using the values sampled in the Preponed region.

Assertions that execute a pass or fail action block, actually schedule a process associated with the pass and fail code into the Reactive regions, not in the Observed region. This is because concurrent assertions are designed to behave strictly as monitors, thus, they are not allowed to modify the state of the design. But, if assertions cannot schedule any Active region events, why is there a feedback path from the Observed region to the Active region?

The answer is a bit subtle. That feedback path would be taken when an **expect** statement (IEEE Std 1800-2005, section 17.16) is written in a **module** scope. The LRM states:

The statement following the **expect** is scheduled to execute after processing the Observe region in which the property completes its evaluation.

Hence, if the **expect** statement is incorporated within a **module**, the corresponding process is added to the Active region and will be executed immediately following the Observed region processing. Conversely, if the expect statement is incorporated in a **program** block, the corresponding process is scheduled into the Reactive region.

The utilization of this feedback path should be rare. The **expect** statement is typically used by a testbench that reacts to the disposition of an assertion. We are unaware of a methodology that would place such statements in a **module**. One reason for doing so would be to synthesize the assertion in order for the design to react to the assertion. However, there are currently no synthesis tools that support such a mechanism.

2.2.7 The Reactive Region Set (Reactive - Re-Inactive - Re-NBA)

The reactive region set is used to schedule blocking assignments, **#0** blocking assignments and nonblocking assignments included in **program** code. Any task or function called from a **program** is also scheduled into the reactive set event regions. The intended purpose of the reactive region set is to schedule testbench stimulus drivers and testbench verification checking in the same time slot after RTL code has settled to a semi-steady state. Of course, testbench stimulus could cause additional combinational logic update activity to occur in the same time slot. Testbench code can be

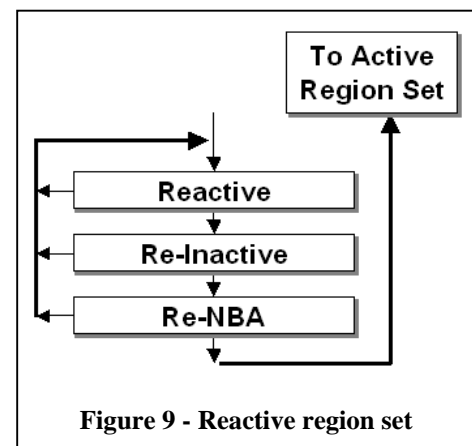


Figure 9 - Reactive region set

written as module code and indeed preexisting testbench code was written as module code, but SystemVerilog users are encouraged to place future testbench code into programs to isolate RTL designs from testbench execution.

Reactive region set activities are described in sections 2.2.8, 2.2.9, and 2.2.10.

2.2.8 Reactive Events Region

The Reactive Events Region, also commonly called the Reactive region, is part of the Reactive Region Set. The Reactive region is the reactive region set dual of the corresponding Active region in the same time slot. The principal function of this region is to evaluate and execute all current **program** activity. Including (in any order⁴):

- Execute all **program** blocking assignments.
- Execute the pass/fail code from concurrent assertions.
- Evaluate the Right-Hand-Side (RHS) of all **program** nonblocking assignments and schedule updates into the Re-NBA region.
- Execute all **program** continuous assignments
- Execute the **\$exit** and implicit **\$exit** commands.

This region is used to execute the verification processes spawned by **program** blocks. Because the Reactive region is located towards the end of the time slot, a process that executes at this point in the simulation has access to three key pieces of information:

1. The current set of steady-state Active region set values – at the start of the current time slot.
2. The next set of steady-state Active region set values, after clock and signal propagation.
3. The disposition of all concurrent assertions triggered in this time slot.

All this information enables more powerful and flexible verification techniques without forcing users to resort to specialized synchronization code.

The processes that execute when processing the Reactive region typically *drive back* the stimulus into the design.

2.2.9 Re-Inactive Events Region

The Re-Inactive Events Region, also commonly called the Re-Inactive region, is part of the Reactive Region Set. The Re-Inactive region is the reactive region set dual of the corresponding Inactive region in the same time slot.

As currently defined, the Re-Inactive region iterates with the Reactive region until all Reactive/Re-Inactive events have completed. Then, within the same time slot, the RTL regions (Active-Inactive-NBA) will re-trigger if the **program** execution scheduled any events in those regions in the same time slot.

⁴ Statements listed between begin-end do execute in the order listed

Events are scheduled into the Re-Inactive region by executing a **#0** in a **program** process. This region is the dual of the Inactive RTL region, which we recommend not using. However, that recommendation does not apply when dealing with verification code, where it is often useful (and harmless) to add some determinism to the scheduler. For example, when forking background processes, it is often very useful to allow the newly created subprocesses a chance to start executing before continuing the execution of the parent process. This is easily accomplished with the following code:

```
program test;
  initial begin
    fork
      process1;
      process2;
      process3;
    join_none

    #0;
    // parent process continues
  end
endprogram
```

Example 1 - Scheduling of #0 Re-Inactive testbench event

An interesting side note is that if the **program** scheduled more RTL events to execute in the same time slot, all of those events will be in the NBA region and the Active region and Inactive regions will both be empty when execution resumes from the Reactive-loop regions back to the RTL regions. It is possible and even likely that the second pass through the NBA region (to execute the stimulus drives placed in the NBA region) could trigger additional RTL combinational logic activity and cause additional Active region events to execute before completing the time slot. However, that will only be the case if the testbench drives stimulus into the design with zero delay. Some engineers find it useful to drive stimulus with some delay away from the clock edge that typically triggers the **program** processes (see the clocking block example in section 6.2).

2.2.10 Re-NBA Events Region

The Re-NBA Events Region, also commonly called the Re-NBA region, is part of the Reactive Region Set. The Re-NBA region is the reactive region set dual of the corresponding NBA region in the same time slot. The principal function of this region is to execute the updates to the Left-Hand-Side (LHS) variables that were scheduled in the Re-Active region for all currently executing nonblocking assignments that were evaluated in the Reactive region.

As currently defined, the Re-NBA region iterates with the Reactive and Re-Inactive regions until all Reactive region set events have completed. Then, if **program** execution scheduled any Reactive region events that could trigger Active region set events in the same time slot, the Active set regions (Active-Inactive-NBA) will re-trigger and iterate until the Active region set events have completed.

2.2.11 Postponed Region

The principal function of this region is to execute the **\$strobe** and **\$monitor** commands that will show the final updated values for the current time slot. This region is also used to collect functional coverage for items that use strobe sampling.

There is no feedback path from the Postponed region back into the RTL or Reactive-loop regions, so the values displayed and the coverage collected will be the final values for that time slot. This generally is the same behavior that was defined for the *Monitor* region of the IEEE Std 1364-2001.

2.2.12 Preponed - Postponed Naming

Yes, believe it or not, *prepone* is a real word![17] A quick Google search of the word *prepone* reveals that it is a commonly used word in South Asia, which explains why many of our engineering friends from India are very comfortable with the term.

The word means "to place in front of, to schedule for an earlier time[17]." As an example, one can: (1) hold a meeting as scheduled, (2) delay, or postpone, the start of a meeting until a later time, or (3) move up the schedule of, or prepone, the start of a meeting.

There was an appropriate amount of groaning made by the SystemVerilog committee members when this event region was named, but nobody was willing to accept any of the alternatives so the name stuck. For SystemVerilog users, *preponed-postponed* can be thought of as the *begin-end* regions that surround and encapsulate each time slot. Enough said!

2.2.13 #1step - what it really means

#1step is a new and important capability added to SystemVerilog intended primarily for sampling data in assertions and verification code.

First let's start with the definition of "global time precision" because that definition is used to define **#1step**.

From the IEEE Std 1800-2005 - **19.10 Time unit and precision**

The global time precision is the minimum of all the time precision statements and the smallest time precision argument of all the **`timescale** compiler directives (known as the precision of the time unit of the simulation in 19.8 of IEEE Std 1364) in the design. The **step** time unit is equal to the global time precision.

Given this definition, and using the three **`timescale** and one **timeprecision** keywords from the models shown in Figure 10, the global time precision is defined to be **1fs**, which is

equal to the **#1step** delay. That is, this is the smallest unit of time for which the simulator will schedule an event – there can be no activity in between **#1step** delays.

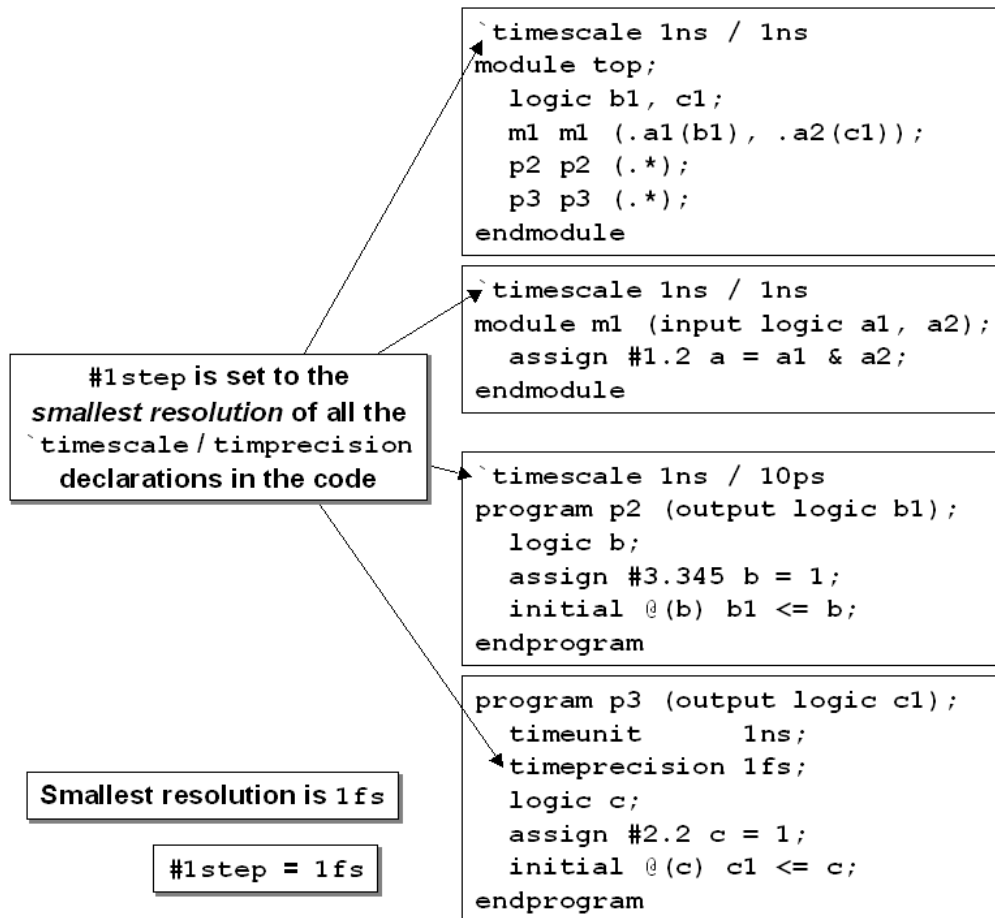


Figure 10 - #1step defined by smallest ``timescale` & `timeprecision`

The IEEE SystemVerilog standard elaborates on the definition of **#1step** as being "conceptually" identical to taking data samples in the Preponed region ...

From the IEEE Std 1800-2005 - **9.3 The stratified event scheduler**

The new **#1step** sampling delay provides the ability to sample data immediately before entering the current time slot and is a preferred construct over other equivalent constructs because it allows the **#1step** time delay to be parameterized. This **#1step** construct is a conceptual mechanism that provides a method for defining when sampling takes place and does not require that an event be created in this previous time slot. Conceptually, this **#1step** sampling is identical to taking the data samples in the Preponed region of the current time slot.

... and further elaborates that the default sample point for **clocking** blocks is **#1step**, which is really the same as sampling the data values in the Postponed region of the actual preceding time slot, because those data values will not have changed between the preceding Postponed region and the current Preponed region. In fact, the sampled value may be the values last sampled 100's of time units earlier if there have been no events during that time span.

From the IEEE Std 1800-2005 - **15.2 Clocking block declaration**

Unless otherwise specified, the default **input** skew is **1step** and the default **output** skew is **0**. A **step** is a special time unit whose value is defined in 19.10. A **1step** input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region). Unlike other time units, which represent physical units, a step cannot be used to set or modify either the precision or the time unit.

The SV-AC subcommittee authored the description of assertion sampling in clause 17.3 of the IEEE SystemVerilog standard as shown below:

From the IEEE Std 1800-2005 - **17.3 Concurrent assertions overview**

The values of variables used in assertions are sampled in the Preponed region of a time slot, and the assertions are evaluated during the Observe region. If a variable used in an assertion is a **clocking** block input variable, the variable must be sampled by the **clocking** block with **#1step** sampling. Any other type of sampling for the **clocking** block variable shall result in an error. The assertion using the **clocking** block variable shall not do its own sampling on the variable, but rather use the sampled value produced by the **clocking** block.

In reality, all values used by assertions, whether sampled with **clocking** block timing or without **clocking** block timing, all occur at **#1step** before the current time slot, but since the value of all of the signals that were sampled **#1step** before the current time slot are exactly the same as the values at the beginning of the time slot, it really does not matter if the signals were sampled **#1step** before the current time slot or in the Preponed region of the current time slot.

```
`timescale 1ns/100ps
module test;
    logic [7:0] d;

    initial begin
        d = 8'h33;
        #5 d = 8'hAA;
    end
endmodule
```

Example 2 - Simple test used to show sampling and #1step sample delays

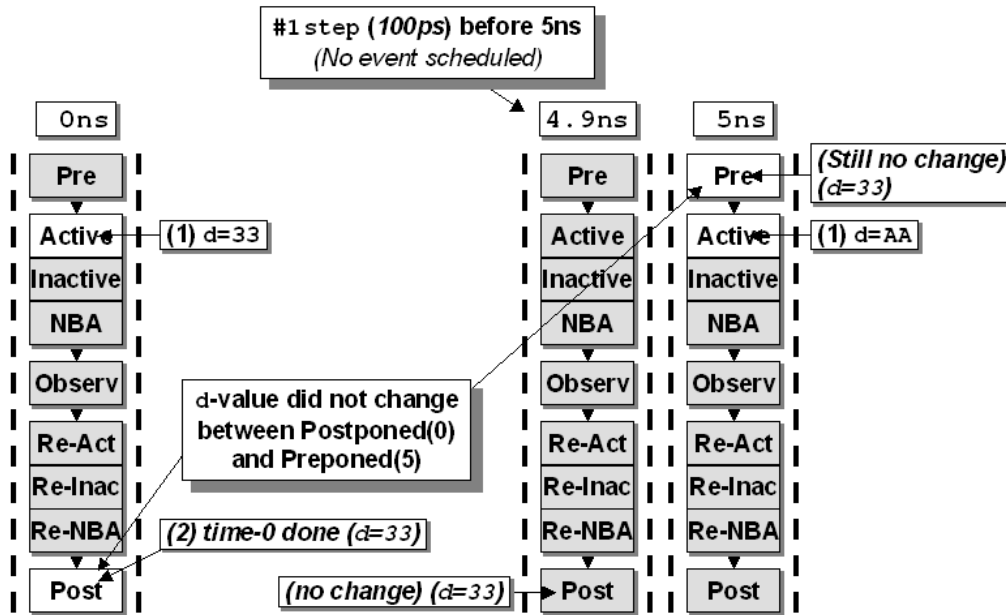


Figure 11 - How events are scheduled and sampled for the code in Example 2

Remember. The IEEE Std 1800-2005 was written by multiple teams of very talented engineers with different specialties. If one perfect and extremely talented engineer had written the entire IEEE Std 1800-2005 document, the descriptions might have been more consistent across the entire document. Whether the assertion values are sampled at **#1step** before the current time slot or in the Preponed region of the current time slot, it will not matter to the end user because the results will be the same.

We have been asked if there is a preferable way to sample the signals, in the previous Postponed region or in the current Preponed region? To the end-user, the answer is no, except that a simulator may run more efficiently using one form or the other.

We have been asked if it would be better to make all of the sampling descriptions in the IEEE SystemVerilog standard the same. We believe the answer is yes - users would be less confused about **#1step** and sampling if all of the IEEE SystemVerilog standard descriptions were consistent. The apparent contradiction arises because the two sampling constructs target different functionality. The clocking block targets verification software, hence it includes a more general sampling mechanism that allows the specification of input sampling skews as arbitrary time delays. Whereas concurrent assertions are intended to be consistent with both formal tools and hardware accelerators, neither of which implements time delays. Hence, assertions restrict sampling to **#1step** of the last time slot exclusively, which, as we saw before, is functionally equivalent to sampling in the Preponed region.

We have been asked what happens at time-0 with respect to **#1step** sampling? Note that time-0 initialization and activity is poorly described for many situations, not just sampling. The working model used to illustrate the time-0 sampling functionality can be used to explain this behavior. At time 0 all 4-state variables are initialized either to an explicit user-assigned value in the

declaration or to their default value of **X**. If the clocking event triggers at time 0, then a sample of the delayed signal value is taken (a virtual pre-time-0 Postponed sampling). Hence, since the delayed signal is always **X** at time 0, a sample at time 0 will yield **X** (or the un-initialized value).

We have been asked if **#1step** can be used as a delay elsewhere in a design or verification environment. The answer is yes, but current implementations only seem to support **#1step** in a sampling context, such as a **clocking** block. Besides the limitations of current implementations, the question we would ask is, why would someone want to use **#1step** as a delay elsewhere in the design or verification environment? We believe that usage of **#1step** as a general purpose delay should be highly discouraged.

Engineers have noted that if all the timeunits and timeprecisions in the compiled files are the same, that a **#1** delay is equivalent to a **#1step** delay. Although this may be true for the current set of design files, if a new design file with different timeprecision is added to the list of compiled files, the **#1step** would change while the **#1** delay would remain unchanged – the **#1step** is a function of the *global* precision while the **#1** delay is a function of the *local* precision only.

Occasionally, engineers add **#1** delays to the RHS of sequential nonblocking assignments and might consider replacing these delays with **#1step** delays. We have heard three reasons why engineers add the **#1** delays:

- (1) it fixes problems with nonblocking assignments (WRONG! - these engineers do not understand how nonblocking assignments work and they probably tried to do a **\$display** on a signal that they just updated using a nonblocking assignment and did not see the expected values - hence the Sunburst Design coding guideline #7).
- (2) to see **clk-to-q** delays in a waveform display for debugging (reasonable - but **#1step** would generally not show these tiny delays in a waveform display)
- (3) to fix hold-time violations when an RTL block drives a gate-level block with timing in a mixed RTL-gates simulation. **#1** delays fix most of these problems because most contemporary ASIC family hold times are well less than 1ns. The **#1step** would generally violate any non-zero hold times in these situations.

We have been asked if it is possible to use fractional **#1.2step** and non-unit **#2step** delays? Although the current IEEE SystemVerilog standard does not prohibit either of these possibilities, the decimal portion of **#1.2step** would be rounded away (remember, step equals the smallest time precision), and using a non-unit step delay is like tinkering with simulation-precision delta delays, and we cannot think of any good reason to try either of these non-unit step delays. Members of the SystemVerilog standards group appear to be united in their distaste for non-unit step delays and a future version of the IEEE SystemVerilog 1800 standard may prohibit or restrict these non-unit step capabilities.

That having been said, having the **#1step** based on the global precision (even though there was nothing like it in Verilog) is exceptionally useful in assertions and verification. We can think of no good methodology that would include **#1step** delays in RTL design code.

Guideline: Only use **#1step** delays in clocking blocks. Assertions already use **#1step** delay-equivalents for sampling the specified assertion signals.

Guideline: Never use non-unit step delays, such as **#1.2step** or **#2step**. The non-unit step delays are not currently supported by any SystemVerilog implementation and they may be prohibited in future version of the IEEE SystemVerilog 1800 standard.

3 Time-0 events, races & race-avoidance

One of the problem areas for simulation race conditions occurs at the beginning of a simulation, at time-0. To avoid these race conditions, one needs to understand which events trigger at time-0 and how events are ordered at time-0.

At the commencement of developing the guidelines for this paper, the authors held two very different views on how tests should startup at time-0. At the heart of the disagreement lies the very nature of time-0 events. Should the guidelines attempt to introduce more determinism to what is fundamentally a random process? Or, should the guidelines make time-0 simulation as pessimistic as possible? Will adding more determinism simply lull engineers into a false sense of security, or will it help them find potential problems?

After collaborating on the development of this paper, the authors have concluded that there are different circumstances that may warrant different approaches to the initialization of clocks and resets during a simulation. Engineers must carefully consider and test the following scenarios if they are applicable to their designs.

3.1 Synchronous or asynchronous resets?

Does the design use synchronous or asynchronous resets? Are all the resets in the design or system synchronous or asynchronous or a mixture of both? What will happen if reset is asserted at time zero? What will happen if reset is not asserted until some later time? What will happen to the design if somebody resets the design multiple times in rapid succession (this is a common hardware failure mode). Additional reset issues are detailed in (Cummings-Mills-Golson - 2002[2]).

3.2 Clock startup issues

Does the design clock startup immediately or after a delay? Should the testbench be setup to test a delayed clock startup? Should the direction of the first clock edge (posedge or negedge) be guaranteed or randomized?

3.3 Startup delays

Can your RTL testbench ignore delays in clock and reset startup? Can your gate-level testbench ignore the same delays in clock and reset startup? Can you test these scenarios in the actual hardware?

3.4 Input Initialization

Should inputs be tested with uninitialized-X values at time-0? Should bi-directional ports be tested with non-driven-Z values at time-0? Should inputs be initialized or driven to specific values at time-0? Should inputs be randomly initialized to state-values at time-0?

3.5 Capturing startup vectors

Are testbench vectors going to be captured and used on a chip-tester? Can the tester be setup to ignore chip outputs during chip-initialization? Does your RTL and gate-level testbench ignore the same chip outputs?

3.6 Functional verification startup matrix

A good test plan will indicate which parts of the startup functionality described in the following matrix should be verified.

Clock, Reset & Input Startup Functionality	Yes	No
(1) Reset - test assuming all synchronous resets?		
(2) Reset - test assuming all asynchronous resets?		
(3) Reset - test assuming mixed system with synchronous & asynchronous resets?		
(4) Reset - test first reset assertion at time-0?		
(5) Reset - test first reset assertion after random delays		
(6) Reset - test for multiple rapid-fire resets in a row?		
(7) Is first clock edge guaranteed to be always a posedge clk?		
(8) Is first clock edge guaranteed to be always a negedge clk?		
(9) Should the first clock edge direction be randomized?		
(10) Clock startup - test immediate clock startup w/posedge clk?		
(11) Clock startup - test immediate clock startup w/negedge clk?		
(12) Clock startup - test fixed-delay clock startup w/posedge clk?		
(13) Clock startup - test fixed-delay clock startup w/negedge clk?		
(14) Clock startup - test random-delay clock startup w/posedge clk?		
(15) Clock startup - test random-delay clock startup w/negedge clk?		
(16) Can the RTL testbench ignore side effects from delayed clock startup?		
(17) Can the RTL testbench ignore side effects from delayed reset assertion?		
(18) Can the Gate-level testbench ignore side effects from delayed clock startup?		
(19) Can the Gate-level testbench ignore side effects from delayed reset assertion?		
(20) Are testbench vectors going to be captured for use on a chip tester?		
(21) Can the chip tester ignore chip-outputs during initialization?		
(22) Does the RTL testbench ignore the same outputs during initialization?		
(23) Does the Gate-level testbench ignore the same outputs during initialization?		
(24) Should inputs be tested with uninitialized-X values at time-0?		
(25) Should bi-directional ports be tested with non-driven-Z values at time-0?		
(26) Should inputs be initialized or driven to specific values at time-0?		
(27) Should inputs be randomly initialized to state-values at time-0?		

Remember: a fundamental guideline of testbench development is that you want to use the same testbench for every phase of the design, from behavioral concept to gates-implementation to chip-tester vectors. Why?

- (1) Because if different testbenches are setup with slightly different parameters to accommodate different phases of the design then they are not testing the same functionality. You want to prove that the design you tested before implementation is the same as the design you tested after implementation and using different testbenches to verify pre- and post-synthesis versions of the design does not accomplish this goal.
- (2) It is hard enough to create one great testbench. You certainly don't want to duplicate that effort for different phases of the design.

3.7 4-state variables

Uninitialized 4-state variables start at time-0 as unknown (**x**) values. Non-driven 4-state nets start at time-0 as floating, high impedance (**z**) values.

Verilog-2001 allowed initialization of variables in the declaration but these would create time-0 events. SystemVerilog changed the way that initialization of variables are scheduled at time-0. SystemVerilog variables initialized in a declaration do not cause a time-0 event.

3.8 2-state variables

Uninitialized 2-state variables start at time-0 with a value of 0. Currently, there are no 2-state net types in SystemVerilog.

This means that the following clock oscillator should not cause a time-0 event, because the 2-state **clk** variable will be initialized to 0 before simulation starts and the first nonblocking assignment to **clk** will also be 0 (no **negedge** transition at time-0).

```
`define CYCLE 10
`timescale 1ns/1ns

module clkgen2a;
    bit clk; // clk is initialized to 0

    initial begin
        clk <= '0; // no time-0 event
        forever #(`CYCLE/2) clk = ~clk;
    end
endmodule
```

Example 3 - 2-state clock oscillator with no time-0 event

4 Sunburst Design recommended and non-recommended clock oscillators

For all of the remaining clock oscillator examples, the following **CYCLE** definition and **`timescale** are used. Of course, both of these declarations could be modified by the verification engineer.

```
`define CYCLE 10
`timescale 1ns/1ns
```

Example 4 - CYCLE and `timescale definitions used for clock oscillators

Recommended 4-state clock oscillator:

```
module clkgen1;
  logic clk; // clk is initialized to X

  initial begin
    clk <= '0; // time-0 no-race negedge clk
    forever #(`CYCLE/2) clk = ~clk;
  end
endmodule
```

Example 5 - clkgen1 clock oscillator

The **clkgen1** clock oscillator of Example 5 transitions from **x->0** at time-0 after all procedural blocks have become active. The resulting deterministic **negedge clk** event has no races with other initializations taking place at time-0.

Recommended 2-state clock oscillator:

```
module clkgen2;
  bit clk = 1; // clk is initialized to 1

  initial begin
    clk <= '0; // time-0 no-race negedge clk
    forever #(`CYCLE/2) clk = ~clk;
  end
endmodule
```

Example 6 - clkgen2 clock oscillator

The **clkgen2** clock oscillator of Example 6 transitions from the initialized **1->0** at time-0 after all procedural blocks have become active. The resulting deterministic **negedge clk** event has no races with other initializations taking place at time-0.

Cliff-Note - Caution: when this paper was published, none of the major SystemVerilog vendors tested had implemented the correct initialization of initialized variable declarations, and all vendors were triggering a time-0 **posedge clk** event from the above **bit clk = '1;** declaration. This is wrong and it may cause all procedural blocks that are sensitive to a **posedge**

clk event to trigger at time-0, which was not intended. For this reason, until vendors correctly initialize variable declarations, Sunburst Design recommends using the 4-state clock oscillators and avoid the 2-state clock oscillators.

Cliff-Note - Caution - Rev 1.1 Update: I have re-tested this feature using Mentor's QuestaSim product and have found that QuestaSim does do the correct pre-time-0 initialization and this has been working since at least ModelSim version 6.2c (the earliest version that I had access to). If you know of other simulators that have implemented this pre-time-0 initialization, please email Cliff so that updates can be made to this paper.

Using the **initial-forever** coding style allows a verification engineer to do interesting clock-startup testing. In **clkgen3** of Example 7, the clock remains unknown for a predefined **START_DLY** before it starts running.

```
module clkgen3;
  parameter START_DLY = 24;
  logic clk; // clk is initialized to X

  initial begin // clk IS INITIALIZED after a delay
    #START_DLY clk <= '0; // first edge after a delay
    forever #(`CYCLE/2) clk = ~clk;
  end
endmodule
```

Example 7 - clkgen3 clock oscillator

In **clkgen3a** of Example 8, the clock is executes a **negedge clk** at time-0 and then delays for a predefined **START_DLY** before it starts running.

```
module clkgen3a;
  parameter START_DLY = 24;
  logic clk; // clk is initialized to X

  initial begin // clk starts after a delay
    clk <= '0; // time-0 no-race negedge clk
    #START_DLY forever #(`CYCLE/2) clk = ~clk;
  end
endmodule
```

Example 8 - clkgen3a clock oscillator

In **clkgen4** of Example 9, the 2-state clock initializes to 0 without causing a time-0 event and then remains unknown for a predefined **START_DLY** before it starts running.

```

module clkgen4;
  parameter START_DLY = 24;
  bit clk; // no time-0 event

  initial begin // clk starts after a delay
    #START_DLY forever #(`CYCLE/2) clk = ~clk;
  end
endmodule

```

Example 9 - clkgen4 clock oscillator

There are two additional commonly used SystemVerilog clock oscillators that take advantage of interesting SystemVerilog features.

The Example 10, **clkgen5** clock oscillator relies on the fact that an uninitialized **bit** type is always initialized to 0. Then the included **always** block continuously re-assigns the **clk** variable every half clock cycle.

```

module clkgen5;
  bit clk; // no time-0 event

  // No time-0 event possible
  // clk always starts after a half-CYCLE
  always #(`CYCLE/2) clk = ~clk;
endmodule

```

Example 10 - clkgen5 clock oscillator (not recommended)

The Example 10, **clkgen6** clock oscillator also relies on the fact that an uninitialized **bit** type is always initialized to 0. Then the included **always** block continuously increments the 1-bit **clk** variable every half clock cycle, causing the **clk** bit to wrap to 0 after incrementing to 1; therefore, the **clk** bit oscillates.

```

module clkgen6;
  bit clk; // no time-0 event

  // No time-0 event possible
  // clk always starts after a half-CYCLE
  always #(`CYCLE/2) clk++;
endmodule

```

Example 11 - clkgen6 clock oscillator (not recommended)

Both of these clock oscillators work fine but neither one will execute a time-0 negedge clock edge (which may or may not be important to the verification plan). This means that the first half clock cycle will pass before a clock-triggering event will occur. Neither of these clock oscillators can have a delayed startup because the **always** block starts unconditionally at time-0. For these

reasons, Sunburst Design does not recommend using either of these clock oscillators even though both will work fine in a simulation.

4.1 Clock oscillators - put them in the top module

You may get different simulation behavior if you put the clock oscillator in the top-level **module** than if you put it in a **program** block.

A clock edge typically initiates all of the activity of a time slot. In Verilog-2001 designs a clock-edge (typically a **posedge clk**) triggers all of the **posedge clk**-sensitive **always** blocks, the blocks that coded according to Sunburst Design's 8 coding Guidelines to Avoid Verilog Race Conditions[3] will all be coded using nonblocking assignments.

The right-hand-side (RHS) of these nonblocking assignments will be saved and the update will be scheduled in the NBA region. Since no other active events are scheduled to execute in the current time slot (at least not yet), all of the NBA events will be activated (placed into the Active region) and executed.

Each updated NBA event may subsequently trigger additional combinational events (**module inputs** or **inouts**, primitive inputs, continuous assignments and combinational **always** blocks), each of which can add additional events to the Active region once all of the previously activated nonblocking assignment events are processed. When these combinational events are executed, they can trigger additional combinational events, and, just like real combinational hardware, can cause ripple events within each clock cycle. And just like real hardware, the ripple of combinational events will not activate edge-triggered clocked logic.

The interesting observation from the preceding description is that a single Active region event (a **posedge** clock assigned using a blocking assignment in the testbench) first triggers and schedules all of the nonblocking events, which in turn trigger the combinational logic events for the same time slot, until all the combinational events have settled to a quiescent state.

Putting a clock oscillator in a **program** block actually schedules a nonblocking clock assignment to the RTL design but might also execute additional **program** block activity before the RTL clock edge is triggered.

Guideline: put the clock oscillator in the top-level **module**, not in a **program** block.

5 How clocking block cycles & delays work

Signals listed in a clocking block will be subject to the specified clocking block timing if the signals are referenced with the clocking block prefix. If the signals are referenced without the clocking block prefix then they refer to the original signals (not the clocking block signals) that are NOT subject to the clocking block timing.

In general to gain access to the synchronous timing of a **clocking** block, the signals should use the prefixed **clocking** block name with the signals. SystemVerilog interfaces and modports offer a practical mechanism to avoid accidentally mixing the two; this is described in section 5.2.

There are times when a verification engineer may want to assign to the same RTL signals without using the predefined **clocking** block delays. One such use might be to initialize some synchronous RTL signals at time-0 to known or randomized values.

5.1 Clocking block and non-clocking block timing of the same signals

Any signal declared as a **module** / **program** / **interface** signal or port that is also declared as an **output** (or **inout**) in a **clocking** block can be assigned using either a regular (asynchronous) delay or a **clocking** block cycle-based delay. The example below illustrates both uses.

```
`timescale 1ns / 1ns
module clk_blk1a;
    logic d, clk;

    initial begin
        clk <= '0;
        forever #5 clk = ~clk;
    end

    blk1a t1 (.*);
endmodule

program blk1a (output logic d,
               input      clk);

    // program clocking block
    default clocking cb1 @(posedge clk);
        output #2 d;
    endclocking

    initial begin
        $timeformat(-9,0,"ns",10);
        $monitor("%t: d=%b  clk=%b", $time, d, clk);
    end
end
```

```

initial begin
    d <= '1; // 0ns: clk=0 d=1
    ##1 cb1.d <= '0; // 5ns: clk=1
                        // 7ns: d=0
                        // 10ns: clk=0
    ##1 cb1.d <= '1; // 15ns: clk=1
                        // 17ns: d=1
                        // 20ns: clk=0
    ##1 cb1.d <= '0; // 25ns: clk=1
                        // 27ns: d=0
                        // 30ns: clk=0
    ##1 $finish; // 35ns: clk=1
end
endprogram

```

Example 12 - program block with mixed non-clocking and clocking delay assignments

In the preceding example, all occurrences of **##1** are equivalent to **@(posedge clk)**. All assignments to the variable **d** happen immediately after the preceding event or time delay, just like any other assignment in Verilog. The assignments in this example proceed as follows:

- The clock oscillator makes its first assignment at time-0 using a nonblocking assignment. All subsequent clock assignments are done using blocking assignments and they all execute in the Active region. The **posedge clk** assignments trigger **program** block activity.
- The first assignment to the **d** variable happens at the beginning of the program block. The **d**-variable is evaluated in the Reactive region and then scheduled to be updated in the same time slot (time-0) Re-NBA region. See the left side of Figure 12 for the corresponding event scheduling.
- The second assignment to the **d** variable happens after **##1** (posedge clk at time 5ns) plus an additional delay of 2ns (time 7ns) because the **d**-variable is prefixed with a clocking block specifier as noted by the assignment **cb1.d <= '0;**. Although the assignment is made using the **<=** token, this is not a nonblocking assignment but is a *clocking drive* assignment that also happens to schedule the update into the Re-NBA region of a future time slot, as if it were a nonblocking assignment with RHS delay. In this case, the assignment is equivalent to: **@(posedge clk) d <= #2 '0;**. See Figure 12 for the corresponding event scheduling.

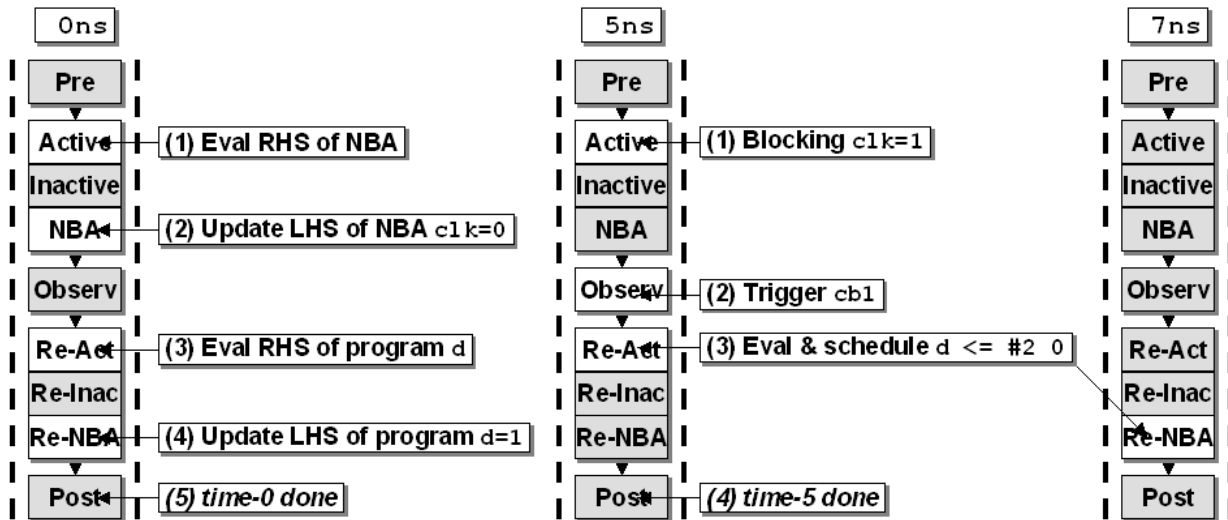


Figure 12 - Example 12 event scheduling from time 0ns to time 7ns

- The third assignment to the `d` variable (a clocking drive assignment) happens after `##1` (posedge `clk` at time 15ns) plus an additional delay of 2ns (time 17ns) because the `d`-variable is prefixed with a clocking block specifier as noted by the assignment `cb1.d <= '1;`. In this case, the assignment is equivalent to: `@(posedge clk) d <= #2 '1;`. See Figure 13 and Figure 14 for the corresponding event scheduling.

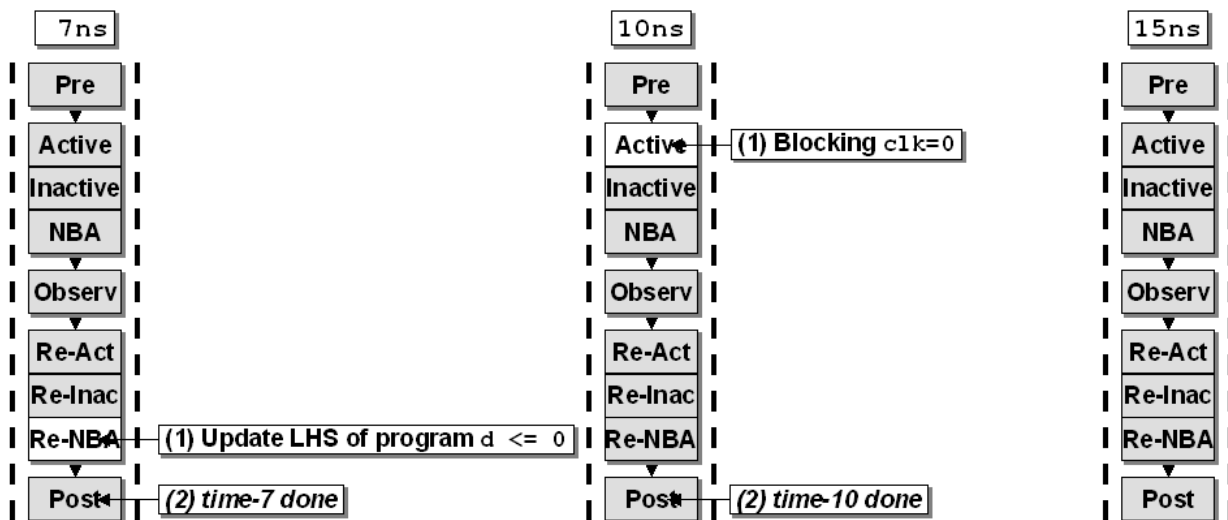


Figure 13 - Example 12 event scheduling from time 7ns to time 15ns

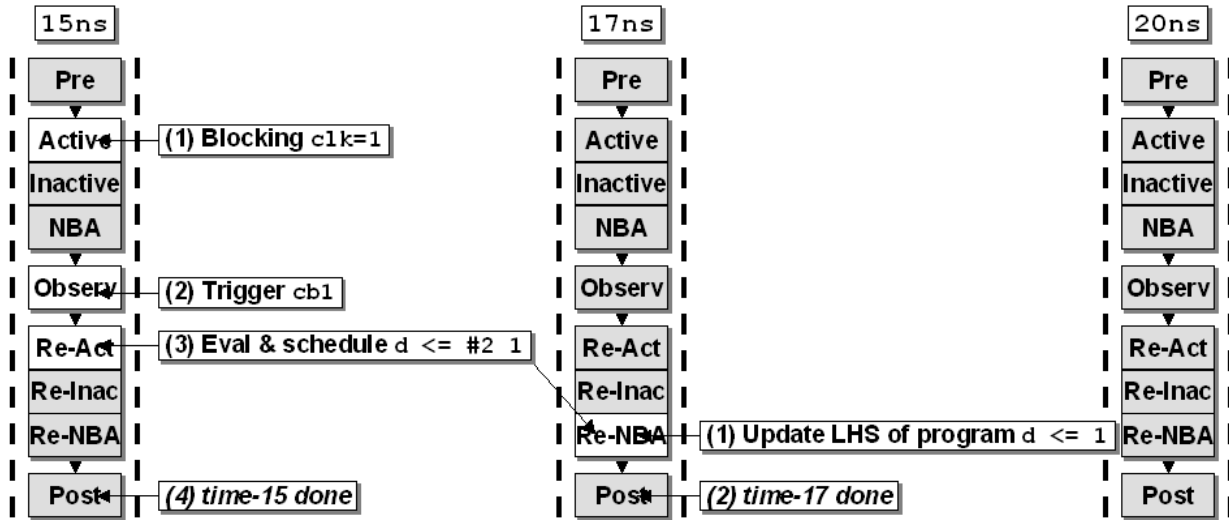


Figure 14 - Example 12 event scheduling from time 15ns to time 20ns

The fourth assignment to the `d` variable (also a clocking drive assignment) happens after `##1` (posedge `clk` at time 25ns) plus an additional delay of 2ns (time 27ns) because the `d`-variable is prefixed with a clocking block specifier as noted by the assignment `cb1.d <= '0;`. In this case, the assignment is equivalent to: `@(posedge clk) d <= #2 '0;`. See Figure 15 and Figure 16 for the corresponding event scheduling.

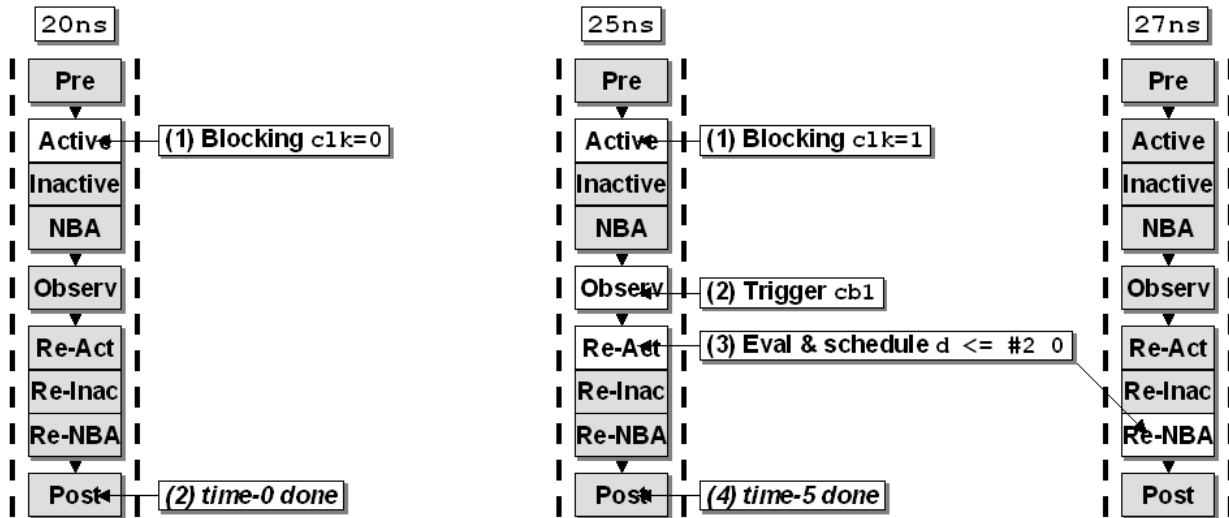


Figure 15 - Example 12 event scheduling from time 20ns to time 27ns

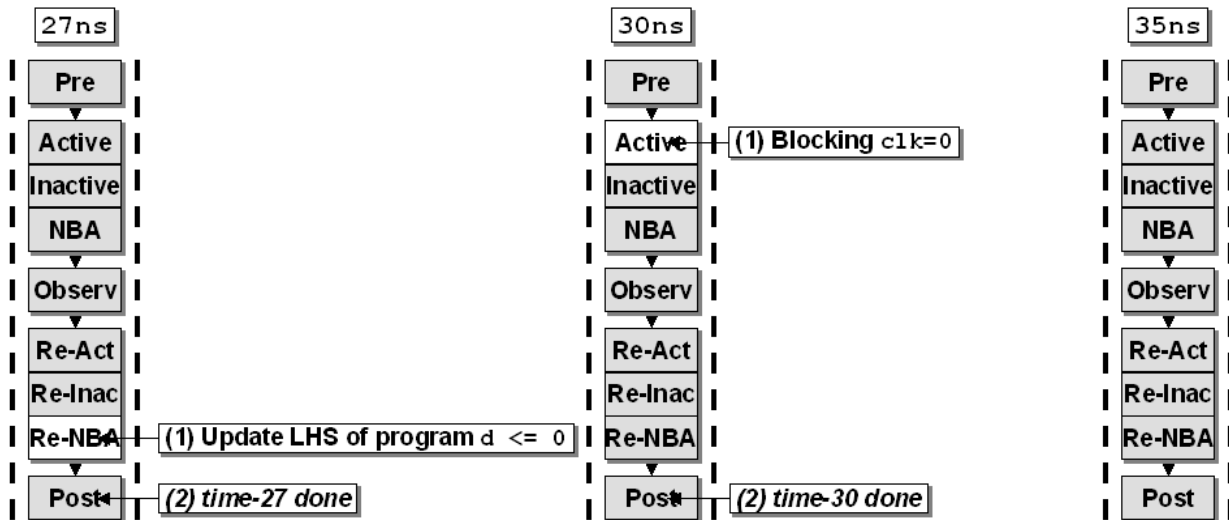


Figure 16 - Example 12 event scheduling from time 27ns to time 35ns

After another ##1 delay (posedge clk at time 35ns) the `$finish` command is executed and the simulation terminates. See Figure 17 for the corresponding event scheduling.

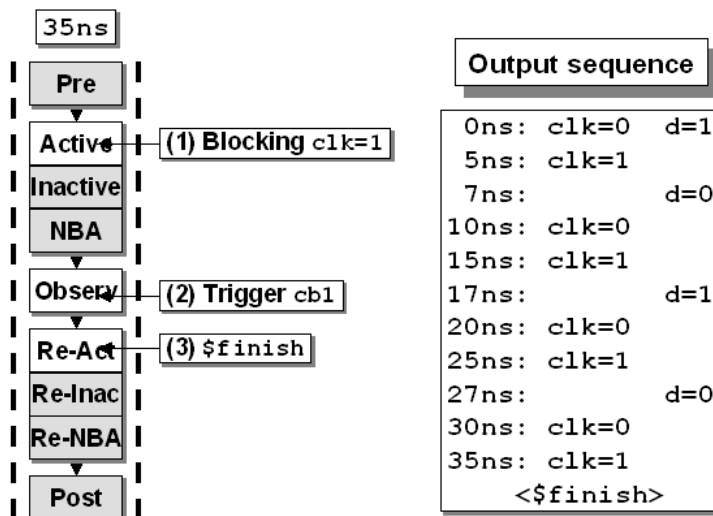


Figure 17 - Example 12 event scheduling completion at time 35ns

The output waveforms for Example 12 are shown in Figure 18.

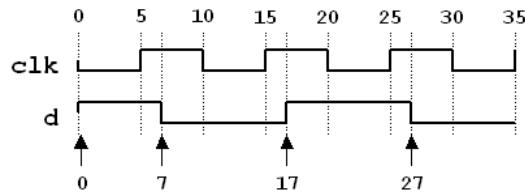


Figure 18 - Example 12 output waveforms

The **##** notation of Example 12 is a convenient shorthand to specify cycle delays. In order to use this notation, a **default clocking** block must be defined, as there was in Example 12. In the absence of a **default clocking** block declaration, an existing **clocking** block can be made into the **default clocking**. This is shown in Example 13.

<pre> clocking cb1 @(posedge clk); endclocking default clocking cb1; initial begin ... ##1 d <= '0; ##5 d <= '1; ... end </pre>	<pre> initial begin ... @(cb1); d <= '0; repeat(5) @(cb1); d <= '1; ... end </pre>
--	---

Example 13 - Equivalent cycle delays using **##**-notation and **@(cb1)** notation

5.2 Clocking blocks in SystemVerilog interfaces

Clocking blocks within SystemVerilog interfaces are useful to specify the timing of the interface signals. An interface can contain one or more **clocking** blocks, as shown in Example 14.

```

interface ifc (input clk, input clk2);
    wire reset;
    wire request;
    wire data;

    clocking cb @(posedge clk);
        output reset;
        input request;
        input data;
    endclocking

    clocking cbDR @(posedge clk2);
        output reset;
        input request;
    endclocking
endinterface

```

Example 14 - Example interface code with two clocking blocks

The signals within each **clocking** block are sampled and driven synchronously with respect to the clock of their corresponding **clocking** block.

5.2.1 Interface modports and clocking blocks

When used in conjunction with an interface **modport**, **clocking** blocks can be used to define the proper direction and timing behavior of each **modport**. In the **interface** definition every **modport** can have a different **clocking** block. In addition, a particular **modport** can include both synchronous signals (**clocking** blocks) as well as asynchronous signals.

```
interface ifc;
    ....
    modport tb      (clocking cbDR, output reset);
    modport checker (clocking cb);
    modport dut      (output request, input reset, data);
endinterface: ifc
```

Example 15 - interface code with modports that reference clocking blocks

Example 15 defines two **modports** **tb** and **dut**. These **modports** establish different access mechanisms. For example, an RTL design can use the **dut modport** while a testbench uses the **tb modport**. The RTL design can read and write all of its inputs and outputs directly, with no timing restrictions. However, the testbench that uses the **tb modport** will only be able to access the **request** or **reset** signals synchronously, through the **clocking** block, while the **reset** signal will be accessible asynchronously.

5.3 Could clocking blocks lead to a better Verilog?

To understand the motivation behind clocking blocks, it is useful to study the genesis of nonblocking assignments in Verilog.

When a signal changes value, the simulator schedules an event for each process that is sensitive to the changing value. Next, when the newly scheduled events are processed, the executing processes may schedule other events with zero delay. Since the order in which an event region is processed is undefined, the order of these new events is also undefined. Thus, users cannot predict the outcome of a potential race.

We illustrate the problem with the two-stage D-flip-flop shift register shown in Figure 19. The flip-flops have a common clock and use 0 delays.

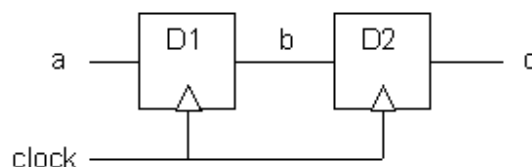


Figure 19 – A 2-stage shift register of flip flops

When the clock signal is triggered, the processes associated with both flip-flops, D1 and D2, are scheduled in the Active region. Next, the two flip-flop processes are executed in an unspecified order. If the flip-flops use blocking assignments to update their outputs then the shift register will work properly only when the D2 executes after D1. If D1 executes first, the update to signal *b* will cause D2 to latch the *next* value. Interestingly, the reason why the simulation model breaks down is because it fails to model the fundamental principle that hardware signals exhibit inertia and cannot change instantaneously.

The solution to this problem has taken several forms. At first, engineers fixed this problem by adding delays to all memory elements. This was only partly satisfactory because the delay information became an artifact to fix the simulator. Next, engineers solved this problem adding explicit temporary variables, unit delays, an eventually #0 delays – a coding style that led to the standardization of the now dreaded Inactive region. Ultimately, this problem was solved by adding nonblocking assignments to the Verilog standard, plus a few methodologies such as the one presented in this paper. Interestingly, the problem was fixed not by adding a construct that could model the intrinsic inertia of the hardware, but by building into Verilog essentially the same basic paradigm (albeit much more efficiently) that engineers had been attempting: store the Right-Hand-Side value into a temporary and delay the update of Left-Hand-Side until a later time – when the NBA region is processed.

One fundamental problem with the NBA solution is that it solves the problem by attempting to move the update past the design's clock propagation phase. And, it has been successful because it works exceedingly well for many popular clocking paradigms. However, the NBA mechanism does break down when the clock signals themselves are triggered in succeeding event regions. For example, if the clock signal of flip-flop D2 in Figure 19 is assigned with a nonblocking assignment, the shift register will once again malfunction. And, indeed designers have run into this problem when not being careful that their clock-gating adheres to the existing methodology. The basic problem is that any mechanism that attempts to be the last one, while allowing zero delay assignments will fail at some point. It is impossible to predict which process will be last in an unbounded relaxation loop.

The new event regions in SystemVerilog suggest a different approach is possible. One simple alternative is to allow users to explicitly retrieve the value present on a signal at the start of the time slot, that is, the value of the signal while processing the Preponed region (or, as discussed, the value during the Postponed region of the previous time slot). Thus, rather than attempt to delay the update until after the *last* clock trigger (which is unknown), we just use the signal value that exists at the *start* of the time slot (which is well defined). To illustrate the idea, we will make this new mechanism a method available to any signal, and will call it **preponed**. This new construct can be used to code the flip-flop thus:

```
always @(posedge clk)
    q = d.preponed() ;
```

This construct can model any type of edge-triggered primitive by more closely emulating the inertia of the underlying hardware being described. It will correctly handle any mix of intra-region clock triggers: parts of the clock tree can be triggered with blocking assignments, while others with nonblocking assignments, either during the first pass through the Active region or during any other pass (in any arbitrary delta-cycle). That is, edge-triggered logic truly becomes independent of the simulation order. This is just a generalization of the mechanism that allows assertions to be evaluated in the Observed region, but using the values sampled in the Preponed region. Interestingly, if this mechanism was implemented then the 8 Sunburst Design coding guidelines shown in Figure 10 could be reduced to just 4:

1. All constructs use blocking assignments – Don't use nonblocking assignments.
2. Use the preponed method on the RHS of edge-triggered sequential constructs.
3. Do not make assignments to the same variable from more than one always block.
4. Do not make #0 procedural assignments.

Hence, a simple change in the way we think about modeling latency would simplify the use model, the simulator implementation, and eliminate additional software induced races.

Although this is an interesting observation of what might have been, it does not exist in the SystemVerilog language.

6 When to apply stimulus

There are two common ways to apply stimulus to a design: (1) apply stimulus on the active clock edge, and (2) apply stimulus on the inactive clock edge.

6.1 Verilog-2001 designs and testbenches

For Verilog-2001 designs and testbenches, applying stimulus on the active clock edge required that all stimulus be applied using nonblocking assignments to avoid race conditions with the registered logic of the design. In effect, applying stimulus using nonblocking assignments on the active clock edge mimicked the behavior of a 0-delay RTL register transfer from testbench to design.

Although this testbench coding style does work, it means that the testbench has to be modified to work with a gate-level version of the design with registers that have hold times. Because the nonblocking stimulus changes all DUT inputs in zero time, hold time violations can be a common problem with this verification strategy.

The aforementioned hold time problem caused many engineers to apply stimulus using nonblocking assignments with RHS #1 delays. This has been shown to be a potential source for serious simulator performance degradation[5].

The Sunburst Design Verilog testbench methodology typically assigned stimulus on the inactive clock edge, typically far away from setup and hold times for the gate-level devices. This permitted the use of efficient blocking assignments and the only activity after application of stimulus on the inactive clock edge would be input combinational logic activity.

6.2 SystemVerilog designs and testbenches

Synchronous testbenches can use the `clocking` block to easily enforce either testbench methodology.

```
clocking sb @(posedge clk);
    default output negedge;
    input in1, in2;
    output out1, out2;
endclocking
```

Example 16 - clocking block to apply stimulus on the negedge of the clk

The `clocking` block of Example 16 implements the methodology that applies stimulus on the `negedge` of a clock. The `clocking` block inputs `sb.in1` and `sb.in2` represent the values of signals `in1` and `in2` sampled on the positive edge of the clock. The outputs `sb.out1` and `sb.out2` are synchronous with the opposite edge of the clock, thus, no matter when during the cycle these signals are assigned a new value, the clocking block output always drives the output signals (`out1` and `out2`) on the negative edge of the clock.

Does the triggering of combinational input logic force the simulator to process more events and degrade simulation performance? No. The combinational events must ripple to a quiescent state, which can either happen and settle before the active clock edge triggers the registered logic, or it can trigger the combinational logic between active clock edges.

Applying stimulus on the inactive clock edge typically means the same testbench can be used on 0-delay RTL models and gate-level simulations with back-annotated delays without modification to the testbench.

7 Summary & Conclusions

The new SystemVerilog event regions were added to the SystemVerilog standard to remove race conditions that could be created between the RTL design and the high-level HVL verification environment. These new regions guarantee predictability and consistency between design, testbenches, and assertions.

Using the new SystemVerilog **clocking** block constructs will facilitate the specification of assertion, cycle operations, and synchronous interfaces.

For RTL coding, engineers should continue to follow the Sunburst Design - 8 coding Guidelines to Avoid Verilog Race Conditions[3]. These coding guidelines still apply to SystemVerilog RTL designs, and if RTL designers use the new **always_comb**, **always_latch** and **always_ff** procedural blocks, guideline #6 will be enforced automatically by the SystemVerilog compiler.

8 Acknowledgements

The authors express their gratitude to Stu Sutherland for his timely feedback for a paper that was submitted to him for his review on very short notice.

9 References

- [1] Chris Spear, "SystemVerilog for Verification", Springer, www.springeronline.com, 2006
- [2] Clifford E. Cummings, Don Mills and Steve Golson, "Asynchronous & Synchronous Reset Design Techniques - Part Deux," *Boston, SNUG 2003 (Synopsys Users Group Conference, Boston, MA, 2003)*, September 2003. Also available at www.sunburst-design.com/papers
- [3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *San Jose, SNUG 2000 (Synopsys Users Group Conference, San Jose, CA, 2000)*, March 2000. Also available at www.sunburst-design.com/papers
- [4] Clifford E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," *San Jose, SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001)*, March 2001. Also available at www.sunburst-design.com/papers
- [5] Clifford E. Cummings, "Verilog Nonblocking Assignments With Delays, Myths & Mysteries," *Boston, SNUG 2002 (Synopsys Users Group Conference, Boston, MA, 2002)*, September 2002. Also available at www.sunburst-design.com/papers

- [6] Clifford E. Cummings and Lionel Bening, "SystemVerilog 2-State Simulation Performance and Verification Advantages," *Boston, SNUG 2004 (Synopsys Users Group Conference, Boston, MA, 2004)*, September 2004. Also available at www.sunburst-design.com/papers
- [7] Dave Rich (member of IEEE 1800-2005 Standards Group), personal communication
- [8] Doug Warmke (member of IEEE 1800-2005 Standards Group), personal communication.
- [9] IEEE P1800/D4 Draft Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language (working draft of the SystemVerilog Standards Group).
- [10] "IEEE Standard Verilog Hardware Description Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001
- [11] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005
- [12] Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale, "Verification Methodology Manual for SystemVerilog," (VMM), Springer, www.springeronline.com, 2005.
- [13] Jay Lawrence (member of IEEE 1800-2005 Standards Group), personal communication.
- [14] Jonathan Bromley, personal communication.
- [15] Lionel Bening, "A Two-State Methodology for RTL Logic Simulation," *Proc. 36th Design Automation Conf.*, June, 1999.
- [16] Lionel Bening, Kenneth Chaney, *Generation of Reproducible Random Initial States in RTL Simulators*, Hewlett-Packard patent US6061819, May, 2000.
- [17] "prepone." Webster's New Millennium™ Dictionary of English, Preview Edition (v 0.9.6). Lexico Publishing Group, LLC. 13 Sep. 2006.
<[Dictionary.com http://dictionary.reference.com/search?q=prepone&x=0&y=0](http://dictionary.reference.com/search?q=prepone&x=0&y=0)>
- [18] Shalom Bresticker (member of IEEE 1800-2005 Standards Group), personal communication.
- [19] Verilog & SystemVerilog Database for Tracking Bugs, Clarifications & Enhancement Requests, www.eda.org/svdb (Login: guest / Password: guest). Also known as the SystemVerilog Mantis Database.

10 Revision Changes

This section includes descriptions of revisions that have taken place since the original paper was presented.

An updated version of this paper can be downloaded from the web site:

www.sunburst-design.com/papers

(Data accurate as of December 5, 2007)

10.1 Revision 1.1 (November 2007) - What Changed?

In April 2007, the SystemVerilog Standards Group updated the SystemVerilog event regions and the behavior of clocking blocks with respect to the new event regions. These changes were incorporated into Mantis item number 890[19].

Mantis 890 added the Re-NBA region (see section 2.2.10) and formally defined the behavior of the Reactive set of event regions (see sections 2.2.7 - 2.2.10) that were put in place to be the testbench dual of the RTL Active set of event regions (see section 2.2.2). These changes forced multiple sections of this paper to be updated to reflect the new program and clocking block event scheduling. Rev 1.0 of this paper should be discarded as obsolete and replaced by this version of the paper.

10.2 Revision 1.2 (December 2007) - What Changed?

My colleague, Jonathan Bromley, found a typo in Example 13. The **##5** is equivalent to **repeat (5) @(cb1);** (with semi-colon after the **@(cb1)**). My thanks to Jonathan for catching this mistake.

11 Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 25 years of ASIC, FPGA and system design experience and 15 years of Verilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 90 SystemVerilog seminars and training classes in the past five years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera Verilog, Verilog Synthesis, SystemVerilog committee, and has presented some 40 papers on Verilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

Arturo Salz received his PhD in Electrical Engineering from Stanford University in 1991. His research focused on RISC architectures and incremental CAD tools, which lead to the creation of the timing simulator IRSIM.

Dr. Salz has worked on video processors and hardware accelerators at Bell Labs, and he has consulted on verification for numerous companies. In 1988 Dr. Salz co-founded Systems Science, which developed the Vera hardware verification language.

Dr. Salz is currently a Synopsys scientist where he works on verification. He served as champion of the Accellera SystemVerilog Enhancements Committee (SV-EC), and is currently an active participant in the SystemVerilog IEEE standard.

Email address: salz@synopsys.com