# Taming Testbench Timing:
## Time's Up for Clocking Block Confusions

Jonathan Bromley
Kevin Johnston

Verilab
Austin, Tx

www.verilab.com

**ABSTRACT**

The *clocking block* feature was designed to provide SystemVerilog verification environments with a versatile and well-structured way to access synchronous signals in a DUT or test harness. In practice, though, the use of clocking blocks has proved to be surprisingly error-prone, despite nearly a decade of application experience since they were first standardized.

This paper reviews the key features and purpose of clocking blocks and then examines why they continue to be a source of confusion and unexpected behavior for many verification engineers. Drawing from the authors' project and mentoring experience, it highlights typical usage errors and how to avoid them. We clarify the internal behavior of clocking blocks to help engineers understand the reasons behind common problems, and show best-practice techniques that allow clocking blocks to be used productively and with confidence. Finally, we consider some areas that may cause portability problems and indicate how to avoid them.

# Table of Contents

## Table of Code Examples



## Table of Figures



## Table of Guidelines

# 1. Introduction

The *clocking block* feature was designed to provide SystemVerilog verification environments with a versatile and well-structured way to access synchronous signals in a device under test (DUT) or test harness. In practice, though, the use of clocking blocks is surprisingly error-prone. Experienced engineers often meet with unexpected or counter-intuitive behaviors, unfamiliar results from apparently familiar syntax, and portability issues when moving from one vendor's tools to another. The authors have heard these cries of despair, and more:

- Cause more timing hazards than they solve
- Confusing to use, especially alongside modports and virtual interfaces
- I can't figure out when to use them and when not to use them
- Why is it behaving like THAT?
- What's this ##1 delay anyway?
- How do I see what it's doing in the waveform viewer?

After reviewing the key features and purpose of clocking blocks, this paper explains how to use them effectively while avoiding the worst pitfalls. It shows how careful use of interface modports can dramatically reduce the risk of timing problems and unexpected behaviors, discusses some powerful but less well-known features of clocking blocks, and provides practical examples of how to make good use of clocking block timing control features in your testbench code. We provide background information on the internal behaviour of clocking blocks, aiding engineers' understanding so that they are less likely to fall foul of common problems. Finally, we examine some areas that may cause portability problems and show how these areas can be easily avoided.

# 2. Synchronous Timing in RTL

Every designer working at the register transfer level (RTL) is thoroughly familiar with the notion of synchronous timing, but it is not something we should take for granted. Taken at face value, it represents a rather strange state of affairs:

- a clock event defines a moment in (simulated) time;
- at that moment in time, we sample the values of various storage elements;
- *and at that same moment in time, we update the storage elements to their new values*.

How can this possibly work? If the storage elements (registers) are sampled and updated at the same moment, how can the sampled values be reliable?

In physical hardware, of course, sampling and updating are not truly simultaneous, but are separated by the registers' input setup and output hold times. When working at the RTL abstraction, though, the intervening time is zero and we must take special steps to ensure that all registers are sampled *before* any are updated.

Maintaining this determinism of sampling before update is vitally important to any synchronous design. As discussed in [1] and elsewhere, the SystemVerilog language provides numerous opportunities to break this determinism. Nevertheless, by following a few simple rules (easily enforced by various linting and formal design checking tools) a designer can be sure to "play safe" and stay out of trouble. VHDL users generally have an easier time of it because the

language enforces delta-cycle delays like those introduced by nonblocking assignment in SystemVerilog, but there is a catch: you must take very great care when modeling clock gating and clock divider logic, because VHDL's unavoidable delta cycle delays are usually *not* appropriate in those situations.

### *Timing Annotation Brings Additional Problems*

Notwithstanding the recent advances in static timing analysis tools, many design teams like to perform some timing-annotated gate level simulation of the design after place and route is complete. In gate level simulation there may be nonzero skews between the clocks applied to various registers that, in RTL simulation, were truly synchronous with one another. In such situations it is important that the testbench should respect the setup, hold and clock-to-output times introduced by timing backannotation.

### *A Clocking Block Is Not A Time Machine*

Any testbench that aims to provide synchronous cycle-by-cycle stimulus and monitoring must, of course, be aware of these timing issues. The key questions are:

- *when should values be sampled*?
- *when should stimulus be driven*?

These questions can only be answered *relative to the moment of a clock edge*, because a testbench operating synchronously has no choice but to use the clock as its timing reference.

At the moment of a clock edge, our testbench can observe the current value of signals to be sampled. As we shall soon see, it can also have access to previous values of those signals. But it has no way to know the values of signals as they will be in the future, at times later than the current clock edge.

On the other hand, stimulus can be set up to change an input signal's value immediately, or at some future time; but the testbench has no way to alter the past value of a signal, since it cannot influence simulation activity that has already happened.

There is a straightforward stratagem for choosing appropriate sampling and drive timings relative to the clock, which works without change both in zero-delay RTL and for timing-annotated simuations:

- sample signals one setup time before the clock event;
- update signals one clock-to-output delay after the clock event.

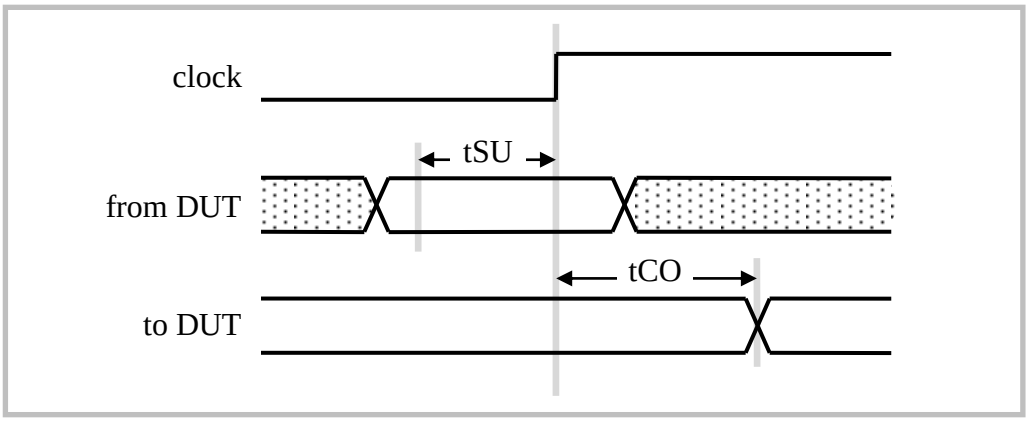


**Figure 2-1: Synchronous sampling and driving**

It is not especially difficult to achieve these objectives using standard SystemVerilog constructs. At the clock edge, the testbench can capture a delayed version of signals it wishes to sample, and can schedule future driving values. In both cases this can be achieved using nonblocking assignment, as indicated in the code fragment below:

```
// signal from_DUT is sampled by the testbench.
// Signal to_DUT is driven by the testbench.
// tSU and tCO are parameters.
always @from_DUT
  delayed_from_DUT <= #tSU from_DUT;
always @(posedge clock) begin
  TB_observation = delayed_from_DUT;
  // testbench checks observation, computes new stimulus…
  to_DUT <= #tCO TB_stimulus;
end
```

**Code Example 2-1: Synchronous timing using standard SystemVerilog**

Although this code example is straightforward, it is clear that it would not scale well to non-trivial problems. Each signal must be individually delayed using procedural code, and – in the case of the driven signals – each drive operation must be carefully annotated with the necessary drive delay. This pattern of code is clumsy and error prone. Clocking blocks provide similar but more flexible functionality, and provide a single place in which the necessary time delays can be specified once and for all, with no duplication.

## 3. Basic Use of Clocking Blocks

We will now consider how a clocking block can provide the same functionality as that shown in Code Example 2-1. Before we begin that discussion, however, it is very important to note that *clocking blocks should not normally be used in isolation* in this way. To get the best possible productivity and reliability gains from clocking blocks, they should be used together with other design and testbench constructs, as discussed later in this paper. The simplified usage in this section is intended only for illustration of clocking block syntax and functionality.

A clocking block must be part of a module or interface.[1] The clocking block specifies, in one single construct:

- the clock event that provides a synchronization reference for DUT and testbench;
- the set of signals that will be sampled and driven by the testbench;
- the timing, relative to the clock event, that the testbench uses to drive and sample those signals.

As we shall see, centralizing the specification of these features in a clocking block greatly reduces coding effort in the testbench.

The code example below shows a clocking block set up to provide the same timing functionality as the SystemVerilog code in Code Example 2-1.

```
// Signal from_DUT is sampled by the testbench.
// Signal to_DUT is driven by the testbench.
clocking simple_CB @(posedge clock);
  input  #tSU from_DUT;
  output #tCO  to_DUT;
endclocking
```

**Code Example 3-1: Basic clocking block declaration**

Before we move on to examine how this affects testbench sampling and stimulus code, it is appropriate to establish some nomenclature. The terms we use here match the SystemVerilog standard Language Reference Manual (LRM) [2], but it is important to note that the wording of relevant parts of the standard was changed significantly between 2005 and 2009 versions and we have chosen to use the more consistent 2009 naming.

***Clocking event:*** The event specification used to synchronize the clocking block, `@(posedge clock)`, is known as the *clocking event*. All clocking block timing is locked to this clocking event, and testbench code that uses the clocking event should normally execute only at the moment of the clocking event (although this can be a tricky issue, and we discuss it in more detail later).

***Clocking signal:*** Signals sampled and driven by the clocking block, `from_DUT` and `to_DUT` in this case, are known as *clocking signals*, although this term is used somewhat inconsistently in the LRM. As we shall see later, they can be nets (`wire`) or variables (`reg`, `logic`) without restriction.

***Clocking skew:*** The parameterized delay values, `#tSU` and `#tCO` in our example, are known as *clocking skews*. They specify the moment (relative to the clock edge) at which input and output clocking signals are to be sampled or driven respectively. The numeric values must be *constant expressions*; in other words, they must be derived only from parameter values, not from variables whose value can change during the simulation run.

***Clockvar:*** Probably the most important piece of terminology, though, is the *clockvar*. Clockvars appear only within clocking blocks. For each clocking input or output signal, the

[1] Strictly, a clocking block can also appear as part of a program. However, this can have unfortunate side-effects and is not recommended in practice. A clocking block can also appear in a checker, but that is a specialized construct generally used only by assertion IP providers, and we do not discuss it further here.

clocking block has a corresponding clockvar. The name of the clockvar is always a two-part dotted name, of the form *clocking_block_name*`.`*clockvar_name*. For example, our clocking block in Code Example 3-1 has two clockvars, `simple_CB.to_DUT` and `simple_CB.from_DUT`. By default each clockvar has the same name as the corresponding clocking signal, but this can be controlled in a more flexible way as described in section 6 below.

### *The Clockvar Is Not the Signal*

Because each clockvar's name is usually the same as the clocking signal's name, it is very tempting to think of them as being one and the same thing. However, we must resist that temptation. It is the authors' experience that confusion between clockvar and clocking signal is the most important single cause of mistakes in the use of clocking blocks. This observation leads us to our first guideline:

**Guideline #1:**

***When using a clocking block, the testbench must access only its clockvars and should never access the clocking signals directly.***

The diagram below shows, in outline, how the clockvars and clocking signals are distinct.

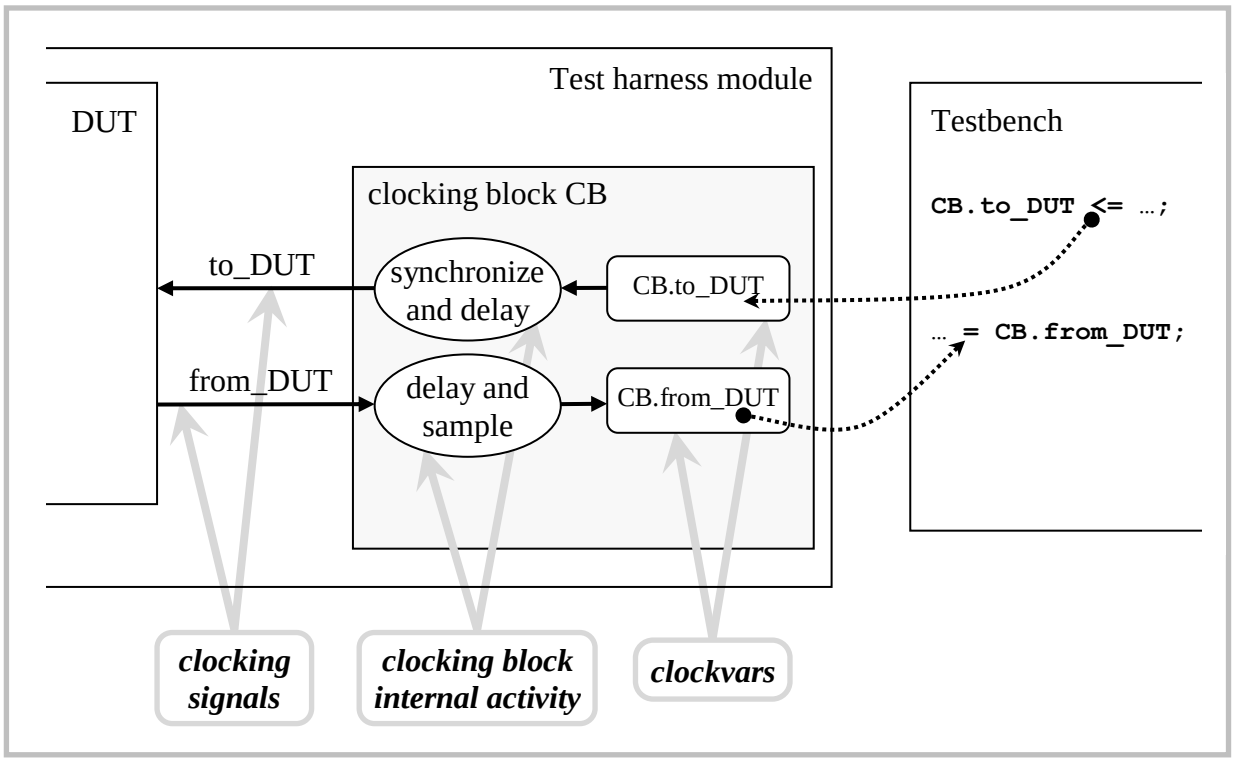


**Figure 3-1: Clockvars and clocking signals**

This diagram also shows that a clocking block's specified timing is effective only when access is made through its clockvars. If a testbench makes access directly to the clocking signals, that access is subverting the clocking block by "going around the back". Such "around the back" access can have disastrous consequences for testbench timing, allowing the testbench to drive and sample signal values in a way that encourages races with RTL code execution.

### *Using the Clocking Block in a Cycle Oriented Testbench*

With sample and drive timing now permanently established for us by the clocking block, our testbench using that arrangement will have synchronous timing that is reliably correct.

The testbench is free to read input clockvars, and write output clockvars, at any time. However, it is strongly recommended that testbench code using clockvars should take care to execute only at the moment of the clocking block's clock event. Constraining the testbench to execute only at clock events may seem unnecessarily restrictive, but in fact it is not. Bear in mind:

- Other parts of the testbench, using different clocking blocks synchronized to different events, are free to execute on those other clock events as required.
- Using clocking blocks encourages us to consider the testbench's behavior in a cycle-by-cycle manner, rather than working in terms of simulation time. This is a valuable discipline for testbench code that is working with a synchronous interface to the DUT.
- Thanks to the clocking block's input and output skews, the testbench code has no need to do anything in the intervals between clock events. All testbench activity relating to a given synchronous interface can (and should) take place exactly on the clock event. Detailed signal timing relative to that event is handled automatically by the clocking block.

There is a further consideration that we ignore at our peril. Testbench code that uses a clocking block should *never* synchronize itself to the raw clock event such as `@(posedge clock)`. This would give rise to unexpected behavior of input sampling. Instead, testbench code should wait for the event created by the clocking block itself. This is achieved by using the name of the clocking block as if it were an event, as shown in the first part of the following example:

```
// GOOD testbench procedural code – Wait on clocking block event
@ simple_CB
  valid_result = simple_CB.from_DUT;
...
...
// BAD testbench procedural code – Wait on raw event
@(posedge clock)  // DON'T DO THIS
  unreliable_result = simple_CB.from_DUT;
```

**Code Example 3-2: Synchronizing to a clocking block's clock event**

Later in the paper we will justify this advice with reference to the internal scheduling of clocking block behavior, and show in detail how things can go wrong if you do not follow this guidance. Meanwhile, the requirement can be summarized thus:

**Guideline #2:**

***Testbench code should synchronize itself to a clocking block's clock event by waiting for the clocking block's own named event, NOT by waiting for the raw clock event.***

### *Driving Signals Through a Clocking Block*

Code Example 3-2 showed how your testbench can wait for the clocking block's event, and then inspect the value of monitored signals by reading its input clockvars. The sampled values your

testbench will obtain are the values of the input clocking signals *as they were at a time before the clock event*, with the actual time specified by the clocking skew. The testbench can then use those "pre-clock" values to compute appropriate stimulus for the DUT.

Bear in mind that the testbench code is running at the moment of the clock event, and the newly computed stimulus value is then available at that moment. This remains true even if the testbench needed to execute thousands of lines of code to compute the stimulus: all that code has executed in zero simulated time, exactly at the moment of the clocking block's event. Nevertheless, it is too late for the stimulus to be useful on the current clock edge. The clock event has already happened, and we do not wish to risk hold time violations by applying new stimulus immediately. Instead, we apply our new stimulus some time after the clock event, so that it is stable and ready for the *next* clock. This timing is shown in Figure 9-1.

This synchronous cycle-by-cycle timing is automatically provided when you use a clocking block's output clockvars. We write to those clockvars using syntax that looks very similar to nonblocking assignment, although the mechanism it uses is rather different. The correct name for it is *clocking drive*. Although you write to the clockvar at exactly the moment of a clock event with no delay, the clocking block automatically delays your new value, applying it to the clocking signal after the skew you specified.

```
// Testbench procedural code
@ simple_CB;                          // Synchronize with the clock
valid_result = simple_CB.from_DUT;    // Get pre-clock values
... // compute new stimulus value (lots of code, runs in zero time)
simple_CB.to_DUT <= new_stimulus;     // Clocking drive
```

**Code Example 3-3: Writing stimulus values using clocking drive**

According to the SystemVerilog LRM, it is illegal to write to an output clockvar using the regular assignment operator =. However, early implementations of clocking blocks in some tools did not properly enforce this restriction, so it is appropriate to add a further guideline as a safety net and to help you avoid hard-to-understand syntax errors:

**Guideline #3:**

***Write to output clockvars using the clocking drive operator*** <=.
***Never try to write an output clockvar using simple assignment*** =.

### *Don't Expect Changes to the Original Signal's Behavior*

A common error among beginning users of clocking blocks is to expect the clocking block to affect the clocking signals themselves, as if the clocking block were somehow annotating the existing signals with a delay value. For example, a user might be surprised to find that the timing of value-changes on the original signal `from_DUT` in Code Example 3-1 is not in any way affected by the clocking block. The clocking block's timing skews do not affect what happens if you access the original signals directly. Timing skews are relevant only when the testbench makes access through the clocking block's clockvars. ***Guideline #1*** applies.

## 4. Saints and Sinners: Zero-Time Skews

In section 3 we described how clocking blocks allow our testbench, executing at the moment of a clock event, to sample signal values as they were some skew time before the clock, and to schedule an update on signals at some skew time after the clock. We believe that it is easier, for tutorial purposes, to describe and understand the operation of clocking blocks when nonzero skew values are used. However, there is an important and useful option to specify zero times for these skews. Indeed, the *most* useful of the options for sampling is in fact the default if you don't specify a skew at all.

### *Just-In-Time Sampling with* `input #1step`

The default skew for clocking block input sampling, if no skew is explicitly specified, is `input #1step`. This rather curious and unfamiliar delay specification indicates that the signal's value is to be sampled *just before* the simulation time at which the clock event occurs. To be a little more precise, "just before" means that the signal is sampled *before* any activity at the time of the clock event, but *after* any earlier simulation activity. For example, suppose our clock event occurs at time 15ns. Suppose, too, that there was some simulation activity (code execution, signal value changes, etc) at time 14.5ns, and nothing happened between those two times. `input#1step` sampling requires the simulator to report the value of our clocking block input signal as it was during that idle time between 14.5ns and 15.0ns. Because there is no simulation activity between those two times, our signal remains unchanged during that interval. Consequently, it makes no difference precisely when the signal is sampled in that time. The LRM actually specifies that it should be sampled at 14.5ns, but *after* all simulation activity has completed at that time – in just the same way as the `$strobe` system task reports the settled values of its arguments at the end of a timestep. Because nothing happens in the intervening time, this definition is exactly equivalent to sampling at 15.0ns but *before* any simulation activity begins.

The foregoing discussion may seem academic, but it has special significance if you also use any assertions clocked by the same clock event. `input#1step` yields exactly the same value as you would obtain by using the `$sampled` system function at the moment of the clock event. This is *the same value that is sampled for the signal if it appears in a SVA property or sequence*. Consistency with assertions' sampling behavior is a strong reason for using `input#1step` whenever possible.

**Guideline #4:**

> ***Use*** `input#1step` ***unless you have a special reason to do otherwise. It guarantees that your testbench sees sampled values that are consistent with the values observed by your SystemVerilog assertions, properties and sequences.***

To conclude our discussion of `input#1step` we mention some potential traps:

- There must be no whitespace between `1` and `step` in the syntax, and you cannot use any number other than `1` together with the `step` time unit. Think of `1step` as a keyword.
- The use of `#1step` in any other context, for example as a procedural delay, is not well defined in the SystemVerilog LRM and is not supported by all tools. Use `#1step` *only* as a clocking block input skew specification, and nowhere else.

### *Zero-Delay Drive with `output#0`*

The default output skew for a clocking block is `output#0`. This skew specification causes the driven signal to be updated during execution of the Re-NBA scheduling region (see clause 4.4 of [2] for details) of the same simulation timestep in which the clock event occurred. This behavior is closely similar to the effect of an ordinary nonblocking assignment (NBA) executed on the clock event, but it will happen after all such NBAs have completed.

The authors generally recommend the use of non-zero output skews, modeling the effect of clock-to-output delay for testbench stimulus. Such delays make waveforms easier to interpret (it becomes obvious that signals were updated *after* a clock event). More importantly, though, the delay makes your testbench robust against the effects of non-zero clock network delays that may arise in gate-level simulation.

**Guideline #5:**

***Use non-zero output skew values in your clocking blocks to make waveform displays clearer, and to avoid problems caused by clock network delays in gate level simulation.***

### *Looks Good, Behaves Badly: `input#0` and Why You Should Avoid It*

Clocking blocks offer a special case for input sampling: `input#0`. This sampling delay specification causes the clocking block to sample an input signal in the Observed region of the SystemVerilog scheduler (see clauses 4.4 and 14.4 of [2]). The Observed region occurs very late in the simulator's processing of events at a given time, even later than the NBA region in which nonblocking assignments are processed, and therefore `input#0` sampling can see the *new* values of register outputs *after* they have been updated by nonblocking assignments executed at the clock edge. In other words, `input#0` sampling allows our testbench a "sneak preview" of the value of a signal *after* the effects of the current clock. Normally your testbench would not be able to see that new value until the *next* clock.

Although this mechanism looks attractive at first, it is (in the authors' opinion) deeply flawed and should be carefully avoided. A primary reason for using clocking blocks is that it enables your testbench to apply appropriate sampling and driving delays, making the testbench immune to the effects of nonzero delays in the DUT. `input#0` sampling, however, offers no such security. As soon as the DUT has nonzero clock-to-output delays, thanks to timing backannotation or even manually-coded modeled delays, the `input#0` mechanism fails to work as expected. For this reason we regard it as too fragile for serious use.

**Guideline #6:**

***Never use `input#0` in your clocking blocks.***

## 5. Using the Clocking Block's Named Event

We have already noted, in Guideline #2, that testbench code should synchronize itself to the named event created by a clocking block, and not to the actual clock event. We will now explain the two critical reasons for this guideline.

### *Decoupling Testbench from Signal Details*

The name of a clocking block (and, therefore, of its named event) can be chosen to reflect the testbench-facing effect of the clock. For example:

```
clocking video_pixel_clk @(negedge vClk);
```

**Code Example 5-1: Renaming the clock event**

In this case, the clock is falling-edge active, and has a cryptic name. The clocking block hides the clock's edge polarity from the testbench. If the testbench carefully uses `@video_pixel_clock` everywhere, instead of `@(negedge vClk)`, then future changes to the clock's name or polarity can be accommodated simply by changing our clocking block's definition. There will be no need to scan hundreds of lines of testbench code to find all the offending `negedge` tests.

This name decoupling is useful, but it is not the dominant reason for our ***Guideline #2***. That is…

### *Assuring Race-free Reading of Input Clockvars*

An input clockvar is a snapshot of the corresponding signal's value, sampled at an appropriate moment. Consequently, we must pay attention to precisely *when* that snapshot is updated. Obviously the update takes place as a result of the clock event. Suppose, then, that we have some code that waits for that same clock event and then reads the clockvar:

```
clocking cb @(posedge clk);
  input from_DUT;  // clockvar updated as a result of (posedge clk)
...
...
initial begin
  @(posedge clk);       // USER ERROR!
  if (cb.from_DUT) ...  // clockvar is read on (posedge clk)
```

**Code Example 5-2: Race between reading and updating a clockvar**

It is clear that we have the potential for a race condition here. There are two actions both scheduled to run as a result of the clock event:

- Internal operations of the clocking block cause the input clockvar `cb.from_DUT` to be updated with its new sampled value.
- External code, triggered by the same clock event, looks at the value of `cb.from_DUT`.

However, there is no need for this race condition to be a problem. If your external (testbench) code waits for the clocking block's named event, `@cb`, then all is well. As described in clause 14.13 of [2], the clocking block's internal scheduling guarantees that all input clockvars have been updated before this event occurs, and therefore there is no race condition; your testbench code will reliably see the newly updated values of input clockvars.

## 6. Funky Features

Clocking blocks offer a number of lesser-known but potentially useful features. Full details can of course be found in the SystemVerilog LRM. Here we merely highlight the possibilities, provide some simple examples and offer guidelines for safe use of these features.

### *Inactive-Edge Skew*

Input or output skew is normally specified as a time using `#N` or `#1step` syntax. However, it is also possible to specify the inactive ede of a clock, as in the following example:

```
clocking edge_CB @(posedge clk);
  output negedge some_signal;
  output negedge #3 later_signal;
...
...
initial begin
  @edge_CB
  edge_CB.some_signal <= new_value;
```

**Code Example 6-1: Opposite-edge output skew**

Here, the clocking drive clearly takes place at the moment of the clock's rising edge, but `some_signal` will be updated on the following falling edge – giving a delay of about half a clock cycle. Of course, if the clocking event were `(negedge clk)` then it would be appropriate to use `posedge` as the opposite-edge skew specification.

Our example also shows that a time delay can be added to the opposite-edge skew, further delaying the update of `later_signal` to 3 time units after the falling edge.

Unfortunately, the LRM's description of edge skew specifiers has some ambiguity and lack of clarity. Consequently we avoid it in our own work, and we encourage others to avoid it.

**Guideline #7:**

***Avoid the use of edge specifiers to determine clocking block skew.***

### *Default Clocking*

A clocking block can be used to provide the default clock event for a module, interface, program or checker[2]. This can be done in either of two ways:

- The keyword `default` can appear as a prefix to the clocking block's definition, in which case that clocking block becomes the default.
- If a design element contains more than one clocking block, it may be clearer to specify the default quite separately, using the isolated default specification `default clocking` *`name_of_existing_clocking_block;`*

Specifying a default clocking has the useful effect that any SVA assertions, properties and sequences use it as their clock by default. For applications where a single clock event controls many assertions, this can be a helpful aid to clarity and brevity. Of course, it remains possible to specify another clock explicitly for some of your assertions if necessary.

This application of `default clocking` is so convenient that it is often found immediately before a block of assertions code, even in situations where the clocking block itself is not likely to be used. In that case, you would write an empty clocking block similar to this example:

[2] The `checker`, a new SystemVerilog construct that encapsulates a collection of assertions and related code for more convenient reuse, appeared in the 2009 revision of the language.

```
default clocking any_name @(posedge sysclk); endclocking
```

**Code Example 6-2: Default clocking**

Note that this clocking block's name is probably not used elsewhere, and indeed in that situation the name is optional.

### *Procedural Cycle Delay*

If a default clocking is in effect, a new option is available in procedural code: the use of ##N as a cycle delay. It is functionally equivalent to the procedural code fragment

```
repeat(N) @(the_default_clocking_block);
```

but it is more convenient because it is not a separate procedural statement; it is a delay prefix just like #N or @(posedge clk). As with #N delays, the delay value is calculated at runtime and does not need to be a constant. However, if it is anything other than a simple number or identifier, it must be enclosed in parentheses (again, this matches the syntax for #N).

As a special case, ##0 can be used. It causes execution to wait for the next clocking event, *unless the clocking event has already occurred at the current moment of simulation time* in which case there is no delay. This can be useful at the beginning of the body of a procedural loop, where you may already have executed a ##1 delay at the end of a previous iteration and you wish to do further work at the same clock edge – but you need to be sure that the first iteration of the loop correctly synchronizes with the clock event. Its behaviour is closely similar to the sync temporal expression in the ***e*** language.

### *Cycle Delays and Clocking Drive*

##N delays can also be used to postpone the effect of a clocking drive, using syntax very similar to intra-assignment delay in NBAs:

```
default clocking C;
...
  C.to_DUT <= ##5 new_value;      // example 1 – postponed drive
  ##10 C.to_DUT <= second_value;  // example 2 – delayed execution
  D.outvar <= ##3 third_value;    // example 3 – different timing
```

**Code Example 6-3: Delayed clocking drives using ## delay**

Just as with time delays in regular NBAs, these different forms have very different effects. In example 1, the clocking drive immediately evaluates the new_value expression, schedules it for application to the to_DUT clockvar five cycles later, and then proceeds without delaying execution. The second example, though, uses ##10 as a prefix delay, stalling execution for 10 cycles before evaluating the second_value expression and applying it immediately to the clockvar.

There is plentiful scope for confusion here. In the first (postponed drive) example, the delay is counted in cycles of the target clockvar's clocking block. This would work correctly even if there were no default clocking. Indeed, in our third example the ##3 delay counts cycles of clocking block D, even though there is a different default clocking in effect. By contrast, a prefix delay such as the ##10 in our second example *must* count cycles of the default clocking, and would be illegal if there were no default clocking in effect.

In the 2005 (and earlier) versions of SystemVerilog, a cycle delay prefix on a clocking drive would count cycles of its target clockvar's clocking block. This was changed for the 2009 revision because of a worrying risk:

```
default clocking C;
...
  ##3  D.to_DUT <= val;  // SV2005: 3 cycles of clocking D
                         // SV2009: 3 cycles of default clocking C
  // But when we add a semicolon...
  ##3; D.to_DUT <= val;  // SV2005: 3 cycles of default clocking C
                         // SV2009: 3 cycles of default clocking C
```

**Code Example 6-4: Hard-to-debug effect of semicolon under the 2005 rules**

Under the 2005 rules, adding the semicolon changes the meaning of `##3` in a way that is subtle and hard to debug. The 2009 rules remove that risk, at the expense of a small loss of convenience, and some asymmetry between the different uses of ## in clocking drive.

After much experimentation and not a little frustration, the authors have concluded that it is safer to avoid entirely the use of ## procedural delays. Instead, we prefer to provide a utility task to wait for a specified number of clock cycles. This task can be used in procedural code with no risk of ambiguity. We will describe this approach in more detail in section 8.

### *A Clocking Block Can Drive Nets or Variables*

Clocking block signals can be either nets or variables. The clocking block automatically modifies its own internal structure if necessary, adding a continuous-assignment driver when the target signal is a net. This feature is a major convenience to testbench code, because it means the testbench does not need to concern itself with choosing the right sort of driver – it merely makes clocking drives to the appropriate clockvar. Note, however, the restriction described in ***Guideline #8*** below.

If a clocking block output or inout drives a net, the driven value will automatically initialize to `'z` (all high-impedance). In this way, the clocking block does not affect the net until the corresponding clockvar is first written. This is convenient because it means that your testbench can choose dynamically, at runtime, whether it should act as a passive monitor or an active driver on the target signal. Your static test harness structure, including the clocking block, can remain unchanged in both cases.

### *Signal Name Aliasing*

The clocking blocks we have discussed thus far use very simple input, output and inout specifications that simply name a signal (identifier). The corresponding clockvar has the same name as the signal.

As a more flexible alternative, a named clockvar can be associated with any expression. This allows you to alias signal names across a clocking block, and to give a single clockvar name for a complicated signal expression that may involve concatenations or hierarchical expressions. This feature works for input, output and inout clockvars, just as it does for all kinds of module ports.

Useful examples can be found in clause 14.5 of [2].

### *Complex Clock Event Expressions*

The clock event is not limited to a simple edge specification. Any valid SystemVerilog event expression is acceptable. For example:

```
clocking complex_CB @(posedge TB.clk iff DUT.clk_enable);
```

Note that any complex clock event expression should always be enclosed in parentheses, as shown in our example.

### *Multiple Clocking Blocks for a Signal*

It is sometimes necessary to sample or drive a signal on more than one clocking event. A good example might be the data ports of a DDR memory, which must be driven or sampled on both rising and falling edges of a clock. Clocking blocks offer special support for this requirement. A variable can be the output expression of two or more clocking blocks, one for each clock event of interest, and at any moment the value driven is determined by the clocking block that had the most recent clock event. This usage is described in clause 14.16.2 of [2], with a specific example for a DDR-like signal. However, it is important to note that this mechanism works only if the clocking signal being driven by multiple clocking blocks is a *variable* (`reg`, `logic` etc). It does not work correctly if the clocking blocks are driving a *net* (`wire` etc).

**Guideline #8:**

***When a signal is driven by more than one clocking block output, that signal should be a variable.***

Any signal can provide an input for more than one clocking block. In that case, each clocking block samples the signal according to its own clock event. The various input clockvars are completely independent, and each behaves just as it would if the signal were connected to only that clocking block.

### *`inout` Clocking Signals*

Clocking blocks can have `input`, `output` and `inout` signals and associated clockvars. The `input` and `output` directions are straightforward, but `inout` is a little surprising when compared with `inout` ports of a module. It is best to think of a clocking block `inout` declaration as being an abbreviation for two separate declarations, identical except that one is an `input` and the other an `output`. Reading the corresponding clockvar has the effect of reading the `input`, whereas driving the clockvar has the effect of driving the `output`.

The behavior described above matches the defined behavior that input clockvars can only be read, never written – and output clockvars can be driven but not read. Unfortunately, though, it gives rise to a small problem concerned with waveform displays in a simulator, discussed more fully in section 7 below.

## 7. Clocking Blocks, Visibility and Debugging

Access to clockvars from testbench code is strictly limited to the appropriate direction as specified in the clocking block. Testbench code can read an `input` clockvar, but cannot write to it. Perhaps more surprisingly, testbench code can drive (write to) an `output` clockvar, *but*

*cannot read that clockvar to inspect its value*. This can be frustrating to beginning users, who may be tempted to try to read an output clockvar to debug what their testbench has written to it:

```
cb.outSig <= something;  // write to an output clockvar
...
$display("Wrote 'b%b to cb.outSig", cb.outSig);  // ILLEGAL
```

The well-intentioned `$display` is attempting to read `cb.outSig` but this read operation is illegal on an `output` clockvar, and will yield a syntax error in any standards-compliant tool.

However, given an `output` clocking signal, most simulators will allow you to see the corresponding clockvar in their wave viewer, using vendor-specific or VPI-based means to do the apparently illegal read operation. This appears to be contrary to the LRM, because the output clockvar cannot be read. However, it makes good sense to be able to visualize the *projected* signal value, as written to the clockvar, in addition to the *actual* value of the signal, affected both by the clockvar (after the output skew delay) and perhaps other drivers on the same signal. Without help from the tool's waveform viewer, user code must resort to reading of the signal rather than the clockvar. Such a read operation will show the signal's value as controlled by the clocking block's synchronization and skew functionality, and does *not* directly show the value written to the clockvar:

```
cb.outSig <= something;  // write to an output clockvar
...
$display("Wrote 'b%b to cb.outSig", outSig);  // read signal itself
```

For a clocking `inout` signal, things are even more tricky for debug visualization because there are in fact two distinct clockvars – one for the output and one for the input direction. As pointed out in section 6, when user code reads an `inout` clockvar it is in fact reading a corresponding `input` clockvar; by contrast, when user code drives an `inout` clockvar it is in fact driving a corresponding `output` clockvar. To display the full relationships between the actual signal and those two clockvars, the simulator's waveform viewer must provide some way to display the two clockvars separately.

## 8. Testbench Structure and Interaction with Classes

We have seen that a clocking block can provide everything that a testbench needs to deal with a set of synchronous signals: a testbench-friendly version of the clock event `@cb_name`, and clockvars allowing the testbench to drive and sample the signals synchronously using a cycle-based methodology. Now we must consider how best to expose that clocking block to the testbench.

It is important to recognize that any advanced SystemVerilog testbench today is likely to be based on classes. Classes that interact directly with signals – drivers, monitors and the like – generally form part of a library of reusable protocol-oriented verification components, and therefore will be defined in SystemVerilog `package` constructs. This brings its own special challenges: code in a SystemVerilog package is not permitted to make hierarchical (cross-module) references to other parts of the testbench or design, because that would defeat its reusability. Instead, references out of the package must be made using dynamic constructs: *virtual interfaces* or class handles. Although one of the authors has previously argued against

their use, virtual interfaces remain the conventional standard way to link classes to other parts of a testbench and we follow that convention here. Interested readers can consult [3] for details of alternative approaches.

If we are to use virtual interfaces to hook our class-type verification objects to static parts of the testbench, it follows that our clocking blocks must be created within an interface. The class-based testbench is then given a pointer or reference, of virtual interface type, to that interface. For example, suppose we have a synchronous bus with `VALID`, `READY` and `DATA` signals. An interface suitable for linking that bus to part of a testbench might be coded thus:

```
interface syncBus_intf   // Bus signals visible through ports
    (input CLOCK, RESET_async_n,
    inout VALID, inout READY, inout [31:0] DATA);
  // Use this clocking block for TB code that stimulates the bus:
  clocking master_cb @(posedge CLOCK);
    default input #1step output #1;
    input READY;
    output VALID, DATA;
  endclocking
  // Use this clocking block for passive monitoring TB code
  default clocking passive_cb @(posedge CLOCK);
    default input #1step;
    input READY, VALID, DATA;
  endclocking
  // One modport for each testbench role
  modport master_mp(clocking master_cb, input RESET_async_n);
  modport passive_mp(clocking passive_cb, input RESET_async_n);
  // This is a good place for bus protocol assertion checks
  p_data_stable_throughout_VALID: assert property (
    VALID && !READY |=> VALID && $stable(DATA);
  );
endinterface
```

**Code Example 8-1: Example signal interface for a simple synchronous bus**

Although it is simple and incomplete, this example shows some relevant and useful features:

- The physical signals of interest are brought in through ports. Most of these are `inout` ports so that the testbench can choose dynamically whether or not to drive them. Only clock and reset, which are stimulated by other unrelated testbench code, are `input` ports.
- The interface contains one or more clocking blocks. Each such clocking block provides timing and in/out direction behaviour corresponding to a specific testbench component's role – active or passive, master or slave, etc.
- Each clocking block is exposed to users of the interface by means of a `modport`. This modport exposes *only* the clocking block, and any signals that are truly asynchronous and therefore should not go through the clocking block. Section 9 explains in detail why asynchronous signals should never be driven or sampled through a clocking block.
- This interface will act as your testbench's hook to each set of bus signals that follow this protocol. As such, it is an ideal place for assertions that check protocol validity.

Thanks to the modport, a correctly hooked-up testbench can see *only* the clocking block and asynchronous signals. This restriction greatly reduces the risk of a testbench making inappropriate access "round the back", bypassing the clocking block and violating *Guideline #1*. However, to get the benefit of this restriction you must ensure that the testbench cannot get to the interface directly, but only via the modport. This is easily achieved by declaring a virtual interface data type that can reference only the modport:

```
typedef virtual syncBus_intf.master_mp  sb_master_vi;
typedef virtual syncBus_intf.passive_mp sb_passive_vi;
```

This data type can and should be declared in the same package that contains the relevant verification component classes. The package and interface should then be compiled together, ready for use elsewhere in the testbench.

**Guideline #9:**

***Declare your clocking block in an interface.***
***Expose the clocking block, and any asynchronous signals that are directly related to it, through a modport of the interface.***
***In your verification code, declare a virtual interface data type that can reference that modport.***

### *Potential Misuse of Modports*

It is important to note, when using a modport to expose a clocking block, that the testbench-facing signal directions are specified in the clocking block and not in the modport. The authors have seen code in which a testbench-facing modport exposes not only a clocking block but also some of the signals in that clocking block. This is highly undesirable, because it allows the testbench to make inappropriate direct access to those signals. Such accesses would violate ***Guideline #1***. The following code example, intended as a possible modification of the interface in Code Example 8-1, shows an example of this unfortunate error. It is wrong to specify signal directions in a modport when that modport already exposes a clocking block that controls access to those signals. Only the asynchronous signals should be listed in the modport itself.

```
modport master_mp(
 clocking master_cb, input RESET_async_n,
 input READY, output VALID, DATA);  // WRONG – exposes raw signals
```

**Code Example 8-2: Erroneous use of modport direction with a clocking block**

**Guideline #10:**

***Use your clocking block to establish signal directions with respect to the testbench.***
***Do not add the raw signals to a testbench-facing interface's modport.***

### *Using the Modports in Verification Code*

With all this structure created, we can now create reusable verification components that make use of the interface and clocking block. Here we use code fragments from a UVM monitor

component as an example, but the same principles apply to verification components in any class-based methodology.

```
class syncBus_monitor extends uvm_monitor;
  sb_passive_vi sigs;  // pointer to the appropriate modport
  .... // other variable declarations and methods
  ....
  // This task will be called from the run_phase method.
  task synchronous_monitoring();
    forever @(sigs.passive_cb) begin
      // Do this code on every clock event
      if (sigs.passive_cb.VALID && sigs.passive_cb.READY) begin
        result_data = sigs.passive_cb.DATA;
        ... // construct a transaction object and
        ... // send it to other parts of the testbench
      end
    end
  endtask : synchronous_monitoring
  ....
  // Utility task to wait until reset has gone false
  task wait_for_end_of_reset();
    wait (sigs. RESET_async_n == 1'b1);
  endtask : wait_for_end_of_reset
  ....
```

**Code Example 8-3: Reaching clockvars through a virtual interface reference**

It is interesting to note that each access to a clockvar such as `sigs.passive_CB.DATA` uses a three-part dotted name. The first part, `sigs`, is the virtual interface variable that points to our modport, and it gives us access directly to the modport `passive_mp` within some instance of the interface. The second part, `passive_cb`, gets us into the clocking block, and the third part, `DATA`, is the name of the clockvar we want to access. Unfortunately there is no way to avoid these three-part dotted names in this situation. Note, however, that we use two-part dotted names in two specific situations: to denote the clocking block's event `sigs.passive_cb`, and to access asynchronous signals such as `sigs.RESET_async_n` that do not make use of the clocking block.

**Guideline #11:**

***Clocking blocks should usually be accessed through a virtual interface variable pointing to a modport of the clocking block's enclosing interface.***

***In that situation, each clockvar must be accessed using the three-part dotted name*** *`virtual_interface.clocking_block.clockvar`*

### *Creating Cycle Delays in Verification Code*

When class-based verification code accesses a clocking block through a virtual interface, as described in this section, it becomes impossible for it to perform `##N` procedural delays. This limitation arises because `##N` used as a procedural delay will always delay by `N` cycles of the

*default clocking* that is currently in effect, as described in section 6. Default clocking is established by lexical scope, not by virtual interface connection. Consequently, the class-based testbench code has no access to the interface's default clocking, and cannot safely use `##N` procedural delays. It is this issue that led us to recommend, in section 6, that cycle delays in testbench code should be implemented as calls to a delay task such as this, which is intended to be an extension of Code Example 8-3:

```
// Utility task to wait for a number of clock cycles.
// This is a method of class syncBus_monitor.
task wait_clocks(int N = 1);
  if (N<0) N = 0;                 // protect against user error!
  repeat(N) @(sigs.passive_cb);  // wait for N clocks
endtask : wait_clocks
....
```

**Code Example 8-4: Implementing cycle delay as a class method**

## 9. Asynchronous Testbench Activity

In this section we examine the impact of clocking blocks on two different kinds of asynchronous behavior in a testbench.

### *Asynchronous Access to Clockvars*

So far, we have described how a testbench reads and writes a clocking block's clockvars in procedural code running at the moment of a clock event. This is the pattern of use that clocking blocks were designed for. However, it is also possible for a testbench to make access to a clocking block's clockvars at some time other than its clock event. The resulting behavior is defined in the LRM, but is often confusing and unexpected.

- If your testbench reads a clockvar at some time between clock events, it will see the value that was sampled at the moment of the previous clock event.
- If your testbench drives (writes to) a clockvar at some time between clock events, the drive will take effect, with the appropriate defined skew, on the next clock event.

In other words, clocking blocks force your testbench to read and write clocking signals in a clock-synchronous manner. If you need truly asynchronous access to a signal, that signal should *not* be accessed through a clocking block. Instead, access the signal directly. It is usually convenient to do this through a modport of the relevant interface, as we did for signal `RESET_async_n` in Code Example 8-1 and Code Example 8-3.

### *Asynchronous (Loopback) Testbench Response to DUT Activity*

Sometimes a testbench needs to take a DUT's output and use it to calculate DUT stimulus. In this case, the clocking block's timing is not necessarily ideal, as it introduces at least one clock cycle of delay between computing a new stimulus value and that value appearing at the DUT.

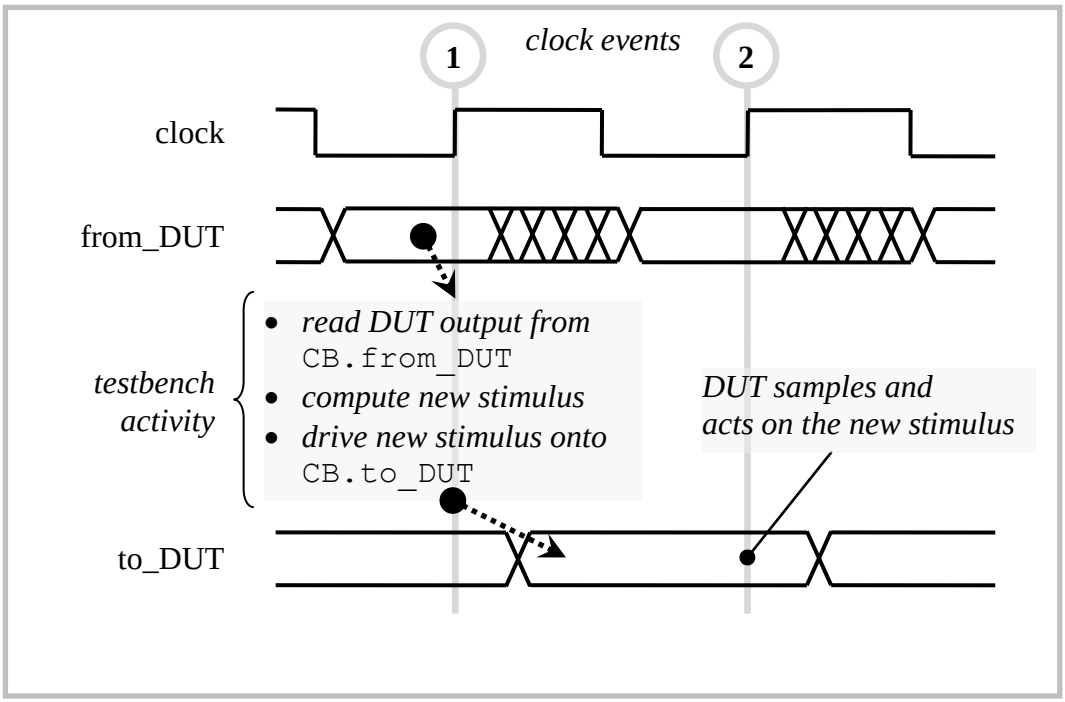


**Figure 9-1: Synchronous sampling and driving**

In this diagram we can see that the testbench, executing at the moment of a clock event:

- picks up DUT output values at clock #1 by reading input clockvars– thereby getting the values as they were just before clock #1
- uses those values to compute new stimulus, using code that executes in zero simulated time at the moment of clock #1
- drives the new stimulus on to output clockvars, so that the stimulus signals update to their new value shortly after clock #1

The DUT then sees the new stimulus value at clock #2. In other words, clocking blocks force the testbench to behave rather like a piece of RTL with a one-clock register delay. Suppose, though, that your testbench should implement configurable loopback, or other combinational feedback, from the DUT's output to input. We cannot achieve this through a clocking block!

Fortunately, there is a straightforward solution: Provide some asynchronous (combinational) feedback logic in the conventional SystemVerilog testbench, and provide synchronous control signals allowing your testbench to manage the feedback clock cycle by clock cycle. This technique allows your testbench to provide stimulus that responds combinatorially to DUT output behavior, while preserving the benefits of clocking blocks.

## 10. SV2005/SV2009 Differences

The SystemVerilog LRM definition of clocking blocks was extensively reworked for the 2009 revision [4]. This section summarizes those changes and provides coding guidelines that help to ensure that your code is not adversely impacted by the changes or back-compatibility concerns.

There were two changes that introduced potential backward compatibility issues:

- In the 2005 standard, ##(N) as a prefix delay to a clocking drive statement would count cycles of that clocking drive's clock. In the 2009 standard it instead counts cycles of the default clocking, just as it would if used as a prefix for any other kind of statement. See section 6 of this paper.
- If your procedural code executes more than one clocking drive to the same clockvar, taking effect at the same clock event, then the most recent of those drives will take effect. Each new clocking drive overwrites the effect of any previous clocking drive to the same clockvar at the same clock event. However, in the 2005 standard, the behavior was different: multiple drives would cause the target signal to be driven to `'x`. The new behavior is generally more flexible and useful. However, to avoid any possible compatibility issues with older releases of simulation tools, it is prudent to try to avoid multiple writes to a clockvar within the same clock cycle.

The numerous other changes were mostly clarifications intended to make the language definition more robust and to ensure portability across simulators. The interaction between clocking blocks and SystemVerilog's event scheduling model is now defined in detail for anyone who is interested. Fortunately for the average user, this arcane information is primarily a concern for implementers rather than for users. However, it did introduce some important new flexibility in the use of clocking blocks. In particular, it now makes perfect sense for testbench code *running in a module or an interface* to drive and read the clockvars of a clocking block. Previously, it was necessary to put your testbench code into a *program block* if you wished to make safe use of clocking blocks.

### *LRM Errata Outstanding on Clocking Blocks*

At the time of writing, there are at least seven distinct bugs filed against the SystemVerilog LRM's description of clocking blocks and how to use them. Some of those issues have already been touched upon here. In practical cases the conservative advice offered in this paper will help you to avoid any problems due to these issues, because they are largely concerned with language definition details that are of interest to tool implementers rather than to users.

## 11. Summary of Guidelines and Conclusions

Clocking blocks provide a powerful and useful mechanism to manage synchronous timing relationships between testbench and DUT. However, they can easily give rise to unexpected and inconvenient behaviors if used improperly. The guidelines in this paper – gathered below for convenience – provide a framework for using clocking blocks in a safe, portable and helpful manner.

1. *When using a clocking block, the testbench must access only its clockvars and should never access the clocking signals directly.*
2. *Testbench code should synchronize itself to a clocking block's clock event by waiting for the clocking block's own named event, NOT by waiting for the raw clock event.*
3. *Write to output clockvars using the clocking drive operator* <=.
   *Never try to write an output clockvar using simple assignment* =.
4. *Use* `input#1step` *unless you have a special reason to do otherwise. It guarantees that your testbench sees sampled values that are consistent with the values observed by your SystemVerilog assertions, properties and sequences.*

5. *Use non-zero output skew values in your clocking blocks to make waveform displays clearer, and to avoid problems caused by clock network delays in gate level simulation.*
6. *Never use* `input#0` *in your clocking blocks.*
7. *Avoid the use of edge specifiers to determine clocking block skew.*
8. *When a signal is driven by more than one clocking block output to model DDR or similar multi-clock behavior, that signal should be a variable.*
9. *Declare your clocking block in an interface. Expose the clocking block, and any asynchronous signals that are directly related to it, through a modport of the interface. In your verification code, declare a virtual interface data type that can reference that modport.*
10. *Use your clocking block to establish signal directions with respect to the testbench. Do not add the raw signals to a testbench-facing interface's modport.*
11. *Clocking blocks should usually be accessed through a virtual interface variable pointing to a modport of the clocking block's enclosing interface.*
    *In that situation, each clockvar must be accessed using the three-part dotted name* `virtual_interface.clocking_block.clockvar`

## 12. Acknowledgements

The authors wish to thank their colleagues at Verilab for the many interesting and helpful discussions that led to the writing of this paper. They also wish to express their thanks to the SystemVerilog testbench extensions committee (SV-EC), whose efforts over the period 2005-2009 brought the LRM definition of clocking blocks to its current state. We are grateful to Stu Sutherland of Sutherland HDL for his reviews.

## 13. References

[1] C. Cummings, "Nonblocking Assignments in Verilog Synthesis," in *SNUG*, San Jose, 2000.

[2] IEEE, Standard 1800-2009 for SystemVerilog Hardware Design and Verification Language, Piscataway, NJ: IEEE, 2009.

[3] J. Bromley and D. Rich, "Abstract BFMs Outshine Virtual Interfaces," in *DVCon*, San Jose, 2008.

[4] SystemVerilog Testbench Enhancements Committee (SV-EC), "EDA.org Mantis bug tracker issue 890," 2005-2009. [Online]. Available: http://www.eda.org/svdb/view.php?id=890.

[5] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.1 User's Guide," Accellera, 2011.