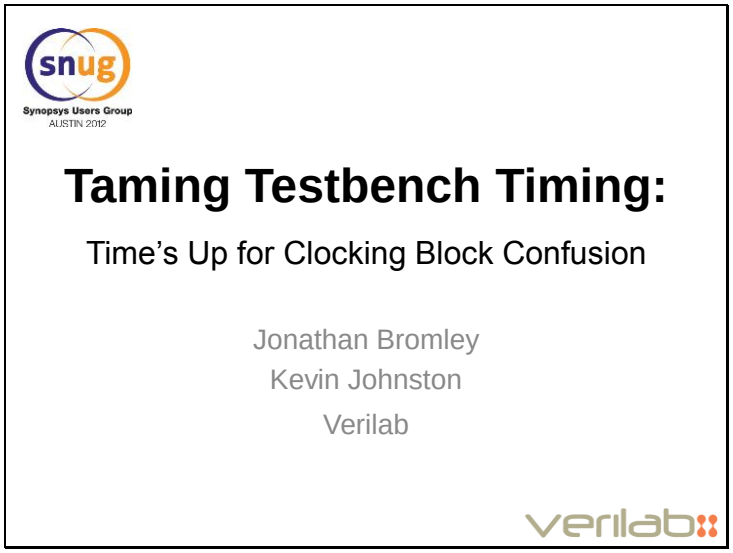
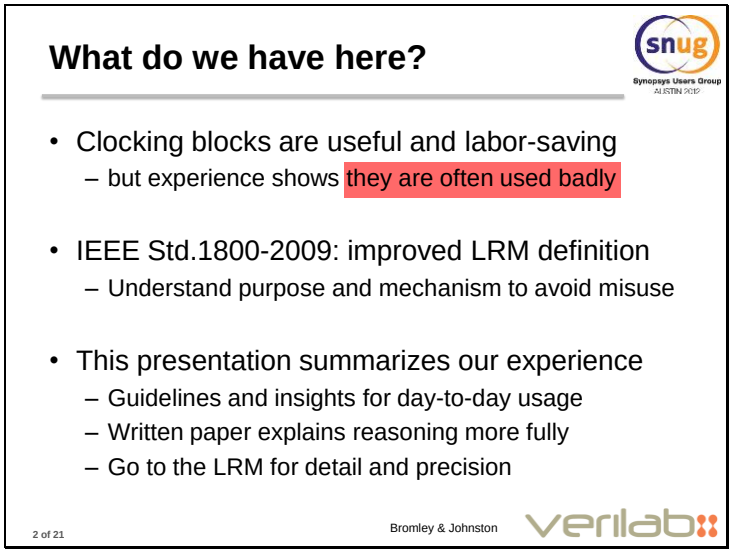
Slide 1



This paper was presented at SNUG Austin (28 September 2012) where it received Best Paper award based on audience voting.

Slide 2

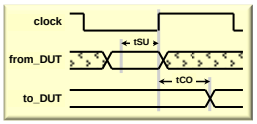


This paper and presentation is a distillation of many years' accumulated experience of using, and teaching about, clocking blocks in SystemVerilog. It's a useful and very widely used language feature, but one that (in our experience) is often abused. We suspect this may be, at least in part, because clocking blocks are concerned with low-level interconnection between your testbench (TB) and device-under-test (DUT), an area of verification that is typically coded just once at the start of a project. Consequently, many engineers get to code clocking blocks only occasionally, and don't get as much practice in their use as with many other language constructs.

Slide 3

Testbench sampling and driving

- **on active clock** – fragile if gated clock moves around w.r.t. data edges
- **any other time** – it's a hack! – very difficult in presence of gated/variable clocks, startup, …
- **correct answer:**
  - sample BEFORE the clock edge
  - drive AFTER the clock edge



3 of 21

Bromley & Johnston

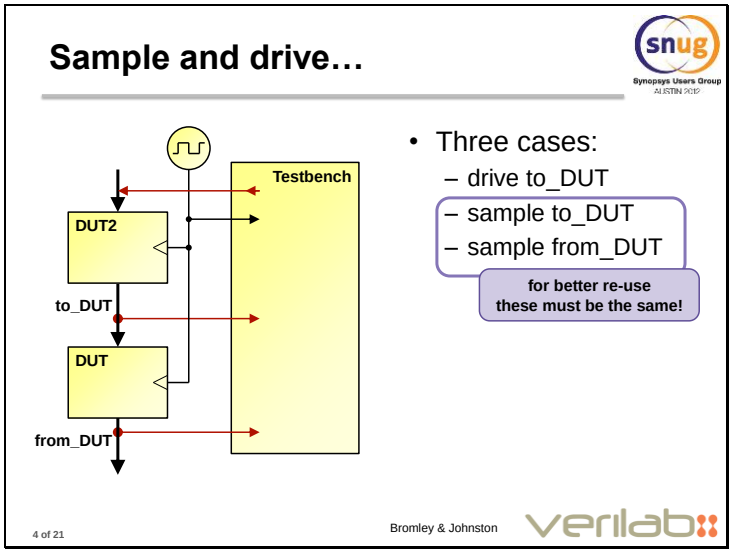
Clocking blocks are all about synchronous interaction between TB and DUT. The central concern is to choose when the TB samples and drives DUT signals relative to the active clock edge.

If you sample and drive on the clock edge, there's always a risk of race conditions between activity in the TB and activity in the DUT – especially if there are any gated clocks between TB and DUT that may introduce skew between the TB's timing and the DUT's.

Shifting the sample and drive time to some moment away from the clock edge is likely to be a hack. We see plenty of engineers trying to do it on the inactive clock edge, but that doesn't make much sense if you have gated clocks – the falling clock edge may happen a VERY long time before the next upcoming active edge!

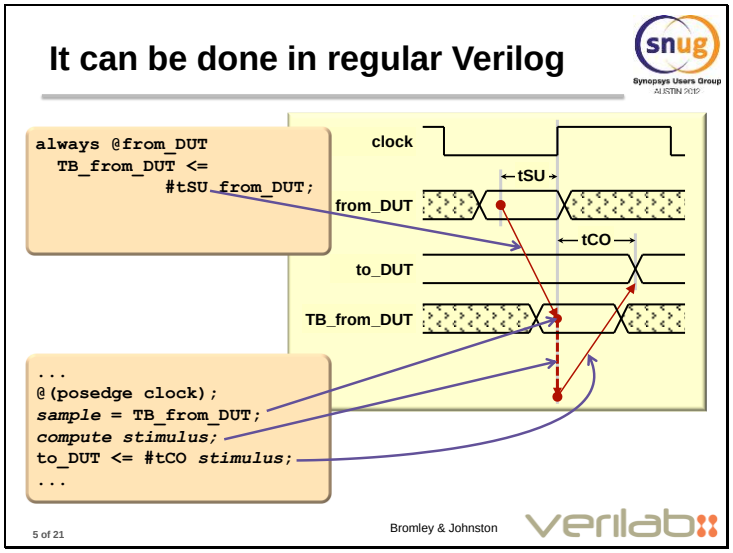
The best approach is for the TB to do everything on the active clock edge, but sample DUT signals' values as they were before the clock, and drive DUT signals some time after the clock – mimicking the external logic's clock to output delay.

Slide 4



If we take a closer look at the sampling and driving problem, we can see that there are three separate cases to consider: driving signals from TB into the DUT, sampling those DUT input signals back into the TB, and sampling DUT outputs. However, we must recognize that sampling DUT inputs and sampling DUT outputs is really the same problem, because any given DUT's inputs are not necessarily driven by the TB. If we reuse our environment in a bigger verification problem, the DUT's stimulus may very well come from some upstream DUT subsystem. So, in reality, we have only two problems: sampling and driving.

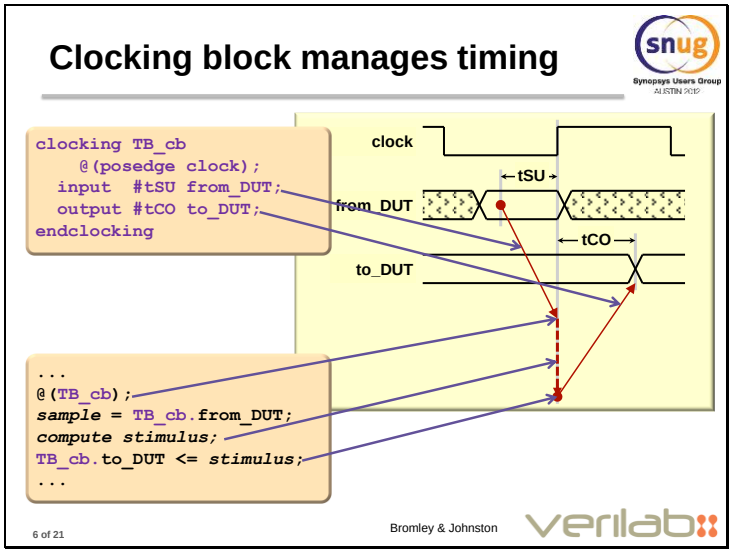
Slide 5



For any chosen signal, we can do the required timing easily enough in conventional SystemVerilog code. As we've already discussed, we will synchronize TB execution to the active clock edge. At that moment, we want to sample from_DUT as it was some time before the clock edge. But back at that sampling time, we had no way to know that the clock edge is coming! So our best option is to create a delayed version of the signal, and allow the TB to use that delayed signal. Not too hard, just a little messy.

Then the TB starts work at the moment of the clock edge, computing some interesting new stimulus and driving it out to the DUT – but *with a delay*. Again, not too difficult… but a little messy, especially given that you need to do that delay on *every* drive to the DUT. Fortunately, clocking blocks package up a neat solution to this problem for us, as we'll see on the next slide.

Slide 6



Here are the same signals again, but this time we use a clocking block to implement those time delays.

At the top left you can see how we might code the clocking block. We would write this code as part of a module or interface, but we'll look at that in more detail later. Note the input and output delay specifications. Note, too, that the input and output directions are specified *from the point of view of the TB* – "input" means an input *from* the DUT *into* the testbench. The clocking block's input/output directions have nothing to do with whether the signal is a DUT input or output.

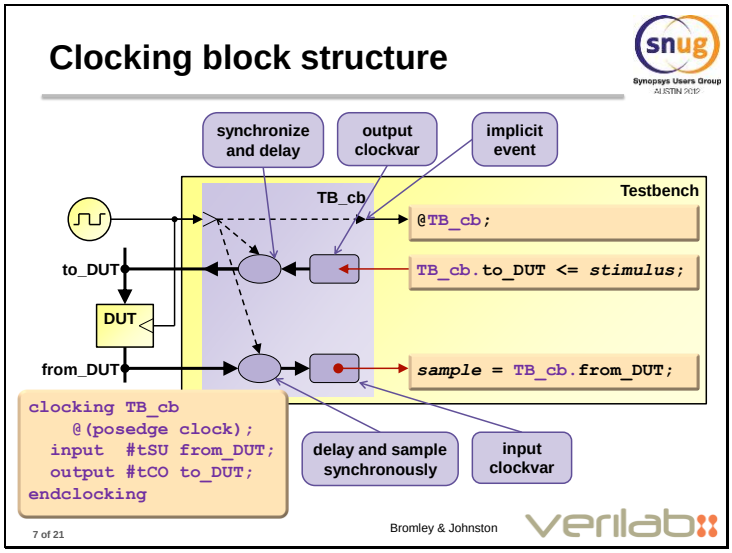
At the lower left we see how the TB code interacts with our clocking block (CB). First we want to synchronize to the active clock. Note we DON'T wait for (posedge clock), but instead we wait for the CB itself – we'll come back to this critically important point later. Next we pick up signal values from the DUT – but we read those signals through the clocking block. Note the dotted name "TB_cb.from_DUT". Note, too, that we're executing this code at the exact moment of the clock edge - but thanks to the CB, we're seeing the DUT signal's value as it was some time before.

Next, our TB goes to work computing the new stimulus value. There may be hundreds or even thousands of lines of code here, but all that code executes in zero simulated time, at the exact moment of the clock edge.

Finally we write the newly calculated stimulus out to the DUT, again going through the CB so that it can take responsibility for delaying the drive appropriately. The TB then goes on its way (likely looping back to wait for the next active clock).

The key thing to note here is that the TB only ever waits for the clocking block, so it's operating strictly cycle-by-cycle. All other timing details are handled by the clocking block. Those pesky timing skew details are described just once, in the clocking block, and your TB has no need to be concerned with them again.

Slide 7



This slide tries to show something of the internal structure of a clocking block. We've used the same simple example from the previous slide.

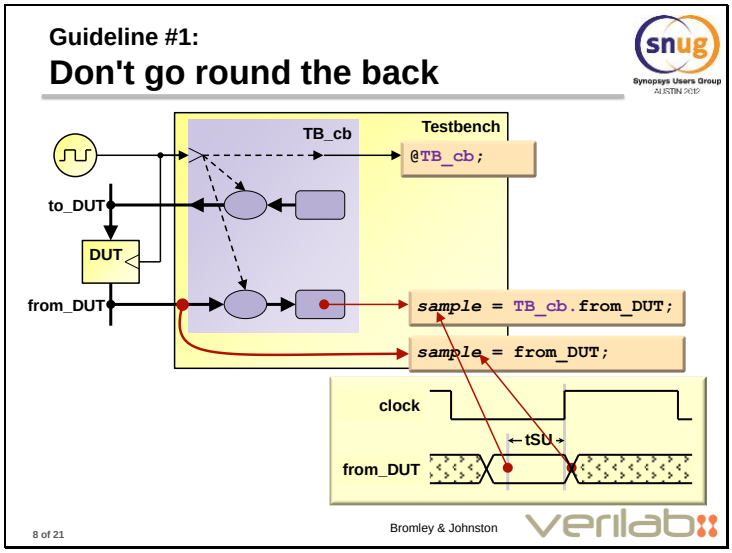
The clocking block is synchronized to the same active clock edge that your DUT and testbench care about. First, the clocking block creates its own version of the active clock edge event (we'll soon say more about why that is so important). Our TB can, and should, wait for that clocking block event when synchronizing to the active clock.

Next, let's think about how DUT signals are driven from the TB. When we write through the clocking block, we're in reality writing to the *output clockvar* "TB_cb.to_DUT". This value is then synchronized and delayed by the clocking block's internal structures, so that it appears on DUT inputs at the specified skew time after the clock.

Finally, what about inputs to the testbench? They are delayed and synchronously sampled by the clocking block's internal structures, and the sampled result is available on the *input clockvar* that your testbench actually reads.

I've been deliberately vague about the exact mechanisms here, because I don't want you to start trying to reproduce it in standard Verilog. The timing semantics of clocking blocks are quite funky, and would be difficult or perhaps impossible to replicate in standard Verilog code. It really *is* important, though, to be aware that *clockvars* are distinct structures within the clocking block, and it is the clockvars – *not* the actual signals – that will be accessed from your TB code.

Slide 8



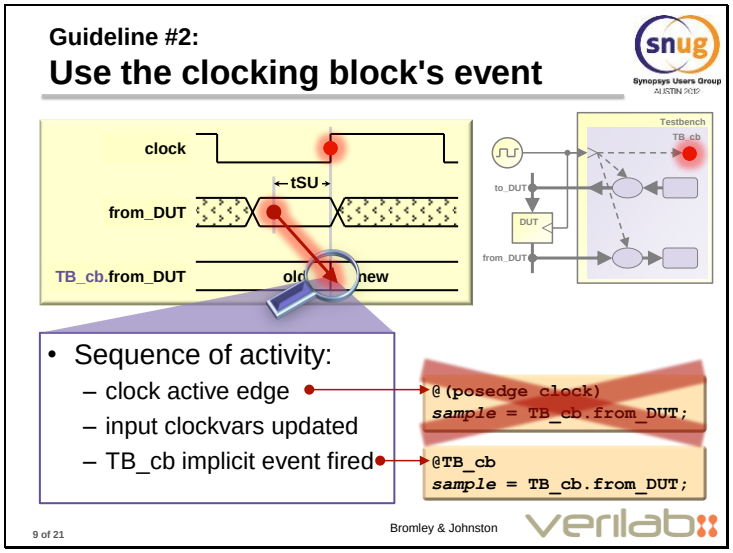
OK, so that's the background to how clocking blocks work and why they're useful. Now let's get started on some practical guidelines for using them effectively.

Our first guideline deals with what I suspect is the most common error I see with clocking blocks in practice. Here's a clocking block being correctly used to sample values from some DUT signal. But what happens if the testbench code tries to go around behind the clocking block, as indicated by the red curved arrow? I see this done very commonly, and I'm not really sure I know why people do it – surely if you bother to write a clocking block, you may as well make use of it? Anyway, its effect is that we pick up the DUT output *exactly at the moment it's changing*! This is hardly a good recipe for getting repeatable, race-free behaviour.

IF YOU USE A CLOCKING BLOCK AT ALL, THEN NEVER GO ROUND THE BACK OF IT. This is Guideline #1 in the full written paper, which gives more detailed reasoning for why round-the-back access is such a bad idea.

A similar story applies to driving through a clocking block – use it, don't cheat your way round it!

Slide 9



Here's the second major error we see often: using the wrong clock event to synchronize a TB. This can have really bad effects on the input, or sampling, side of the clocking block. Here's a closer look at what happens.

First, of course, the real physical clock ticks.

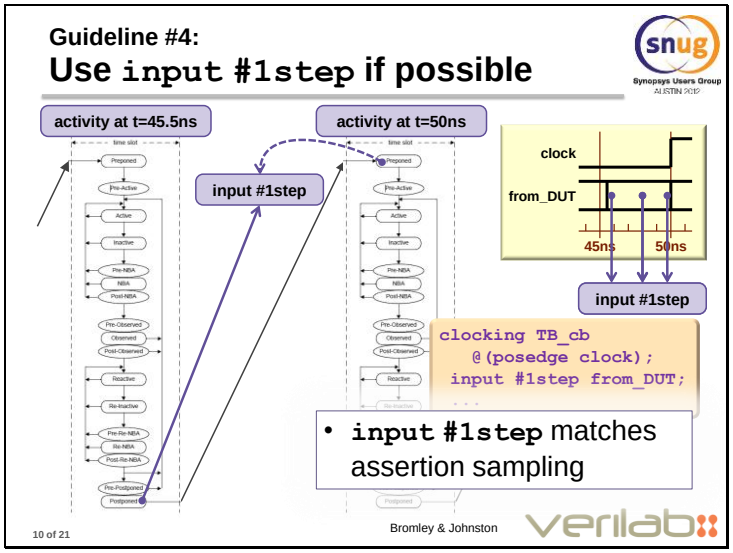
That causes the clocking block to update its input clockvars. The clocking block correctly picks up the signal values as they were at an earlier time, specified by the input skew. Note especially that the clockvar updates right on the clock edge.

The problem arises if we try to use that raw "posedge clock" to synchronise the TB: it looks like there's a race between the clocking block updating its clockvars, and the TB reading them. Clearly this is bad.

However, there is no need for this to be a problem. The next step in the clocking block's internal activity is to fire its own named event, TB_cb. Although this happens in the same simulation timestep as the clock edge, it's guaranteed to happen *after* all clockvars have been updated by the simulator's activity.

If we synchronize our TB to that event, we can then read those input clockvars with confidence, knowing that they have definitely been updated already.

Slide 10



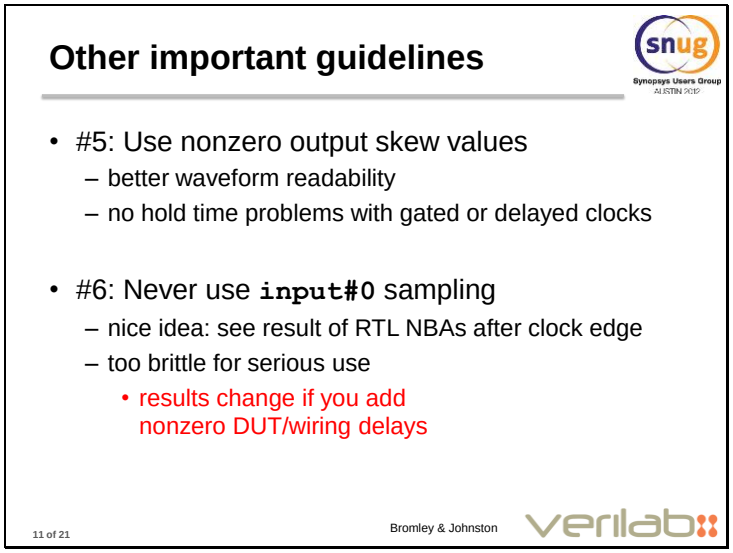
In previous slides we have discussed the use of input sampling that is offset by some non-zero time from the clock edge, such as "input #2ns", because we believe that doing so makes it easier to describe and understand the operation of clocking blocks. However, there is another possibility: "input #1step", which samples a signal *just before* the clock edge. This slide tries to clarify the precise meaning of "just before" in this context.

At any moment of simulation time where there is some activity (code execution, signal value changes, etc), the simulator goes through a sequence of activity known as a *timeslot*. The two flow-chart-like diagrams on this slide are taken from the IEEE SystemVerilog language reference manual (LRM) and are the official description of how a timeslot does its work. In our example, there is an active clock edge at time 50ns and so of course there is a simulation tim slot at that time. We also suppose that, because of some other activity, there is a previous timeslot at 45.5ns – and there is no activity whatsoever between these two timeslots.

At the very beginning of each timeslot there is a Preponed region, in which the simulator can sample signal values as they were at the start of the timeslot. No signal changes or code execution can occur in this Preponed region. Similarly, at the end of the timeslot there is a Postponed region in which the final values of signals at the end of the timeslot can be sampled. Between these two regions there is… how should we say… A Lot Of Complicated Stuff, the whole of the simulator's activity at that moment of simulation time. See clause 4 of the SystemVerilog LRM (IEEE1800-2009) if you are truly determined to learn all the ugly details.

Specifying "input #1step" in a clocking block is defined to cause a signal to be sampled in the Postponed region of the immediately previous timeslot. However, as you can see from the three violet arrows on our timing diagram, that is sure to give the same result as sampling in between the previous and current timeslot, and – more interestingly – the same result as sampling in the Preponed region of the clock edge's timeslot. This is especially important because SystemVerilog assertions do their signal sampling in their clock event's Preponed region.

The overall result of this complicated business is that "input #1step" through a clocking block will provide your TB with the same signal values as are used by any assertions that have the same clock. This is a very useful piece of consistency, and we strongly recommend the use of input #1step sampling unless you have a special reason to do otherwise.

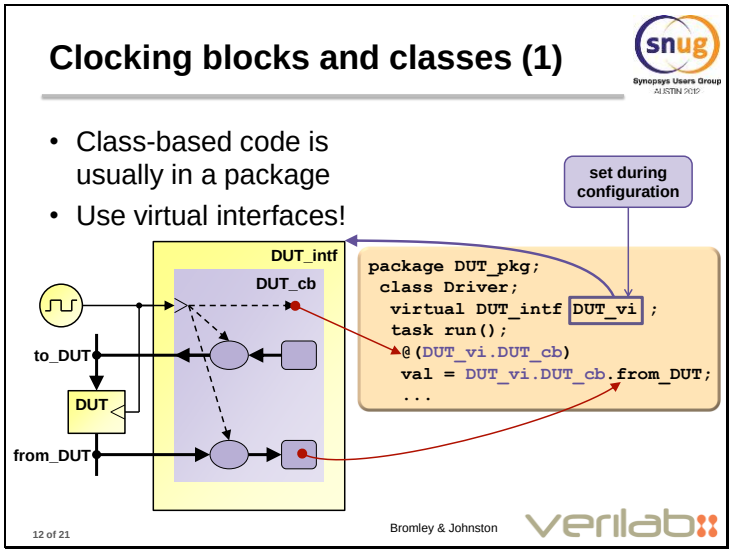


Some important additional guidelines, described in more detail in the paper:

Use nonzero output skew values in your clocking blocks. The visible clock-to-output delay makes waveform displays much easier to interpret. Furthermore, it makes your TB much more robust against issues that may be caused (especially in timing-annotated gate level simulation) by clock skew between a reference clock seen by the TB and gated or buffered clocks presented to individual flipflops within the DUT.

input#0 sampling is, at face value, an interesting feature of clocking blocks. It causes the clocking block to sample a DUT signal at a rather late stage in the simulator's processing of a timeslot (see previous slide). In particular, the sampling happens *after* the updating of signals that were written by zero-delay nonblocking (<=) assignments in the RTL code. This gives your TB access to a register's newly updated (post-clock) value, rather than the register's value as it was just before the clock. However, this useful feature comes at an unacceptably high price. If there is *any* nonzero time delay in that register's update, either because of a delayed nonblocking assignment or wiring delay in the RTL code or because of timing backannotation in gate level simulation, input#0 sampling will revert to giving you the pre-clock value of the signal. The resulting uncertainty and fragility of the TB is, for the authors, completely unacceptable.

Slide 12



Our next concern is to investigate how clocking blocks should interact with a TB written using SystemVerilog classes.

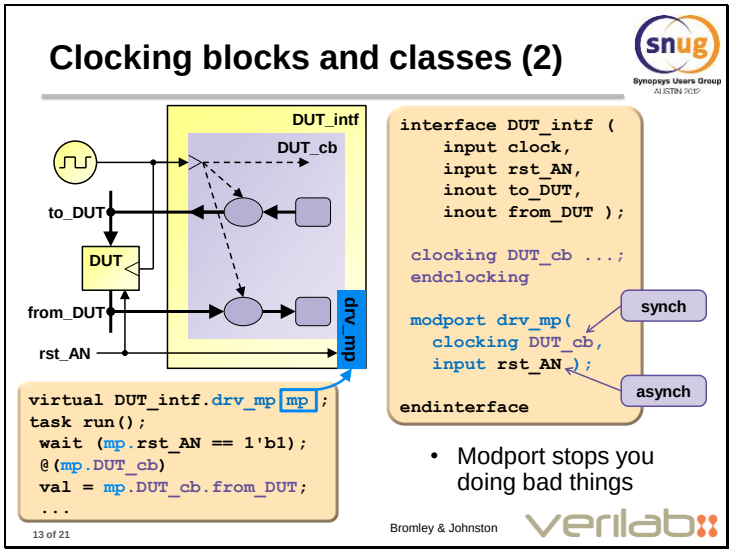
Because clocking blocks inevitably connect directly to signals in the DUT or its immediate surroundings, they are almost always used in low-level or protocol-specific verification components such as a VMM transactor or a UVM agent. Because such components should be designed for reuse, the SystemVerilog classes that they comprise will normally be defined in a SystemVerilog *package.* Code in a package is not permitted to reach out to other parts of the simulation by cross-module reference (XMR), and so there is no way for our Driver class – or any other class written in a package – to be able to make direct access to the clocking block it needs to work with.

The conventional solution to this problem uses *virtual interfaces* and it's shown in outline here. The clocking block should be declared as part of a SystemVerilog *interface* that is designed to hook to some set of DUT signals. Class-based code in the package is then given a *virtual interface* variable of the right data type to be able to reference (point to) any chosen instance of the DUT-signal interface. Setting up the value of that virtual interface variable is a well-known problem with standard solutions in the published methodologies such as UVM.

Once the class has access to such a virtual interface variable, procedural code in the class can reach through that variable to get to the clocking block, as shown in our code fragment.

However, we need just a little more sophistication to get the optimum solution…

Slide 13



Here we have expanded the example to include an additional protocol signal *rst_AN* that is truly asynchronous. Naturally, we want to bring that signal through the interface too, so that our interface encapsulates the whole set of signals required for the protocol. However, it is *never* appropriate to pass asynchronous signals of this kind through a clocking block.

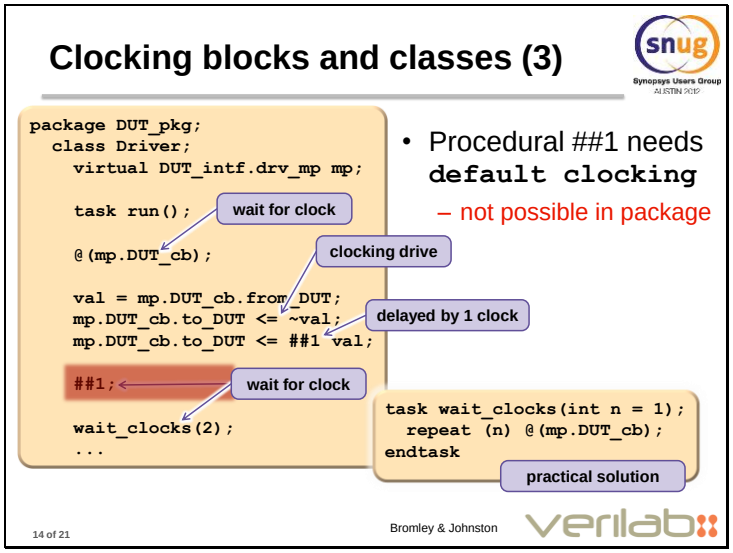
The preferred architecture here is to declare a *modport* within the interface. In the class-based reusable verification code (shown at the lower left) our virtual interface variable is now given the correct data type for the modport, not for the whole interface. This will prevent (or, at least, strongly discourage) our users from trying to access the interface in an inappropriate way, because the modport allows us to restrict which features of the interface are accessible. Look carefully at the modport declaration: it exposes the clocking block, the asynchronous signal(s), and *nothing else*. Directionality of synchronous signals with respect to the testbench is established in the clocking block, not in the modport.

Synchronous signals do not appear in the modport list. Instead, TB code must access them via the clocking block and its clockvars. In this way, Guideline#1 (don't go round the back of a clocking block) is automatically enforced.

The clock signal itself does not appear in the modport list – instead, TB code must synchronize itself to the clocking block's implicit event. This enforces Guideline#2 (synchronize to the clocking block, not the clock event).

Setting the virtual interface's data type to be that of the modport forces our users to point to the modport and not to the whole interface, so that it becomes harder for users to subvert these guidelines inadvertently.

Slide 14

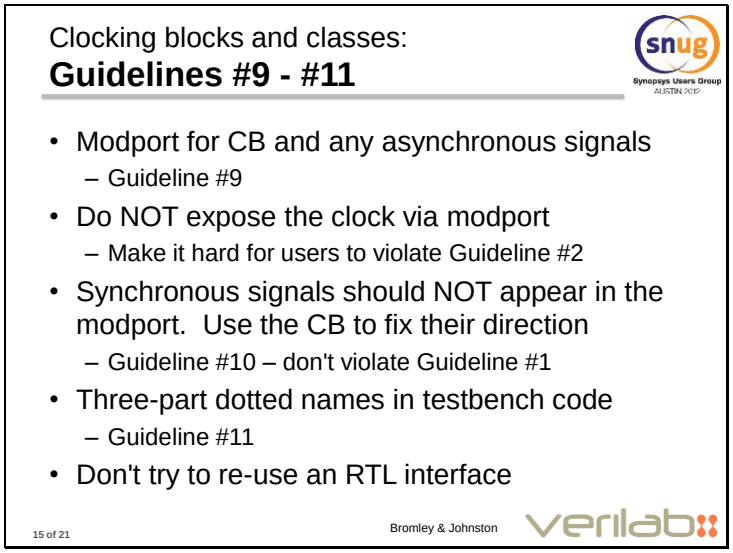


There is a final subtlety about the use of clocking blocks with classes in a package. It concerns the *##N* cycle delay construct, a feature of clocking blocks that we have not yet mentioned.

When TB code writes to an output clockvar using the <= operator, it is possible to add an intra-assignment cycle delay ##N (see callout "delayed by 1 clock" in the slide). Rather like intra-assignment delays in regular nonblocking assignment, this construct does not delay execution of the code at all; but it causes the clocking block's output signal update to be postponed for a further N ticks of the clocking block's clock event. This coding construct *is* available, and useful, for code in a class or package.

It is also possible to use ##N as a procedural delay, just as you might use "#5ns" in ordinary Verilog. However, this delay construct counts cycles of the *default clocking* (discussed in more detail in the full paper). As far as we are concerned here, this is simply impossible because default clocking can be provided only in the same lexical scope as the clocking block itself, i.e. within the interface that contains the clocking block. So it is not available to code in a package. Consequently, we recommend that you should create a simple task that waits for a provided number of clock cycles, as indicated in the lower right of the slide. This task is then readily available for use in your class's procedural code.

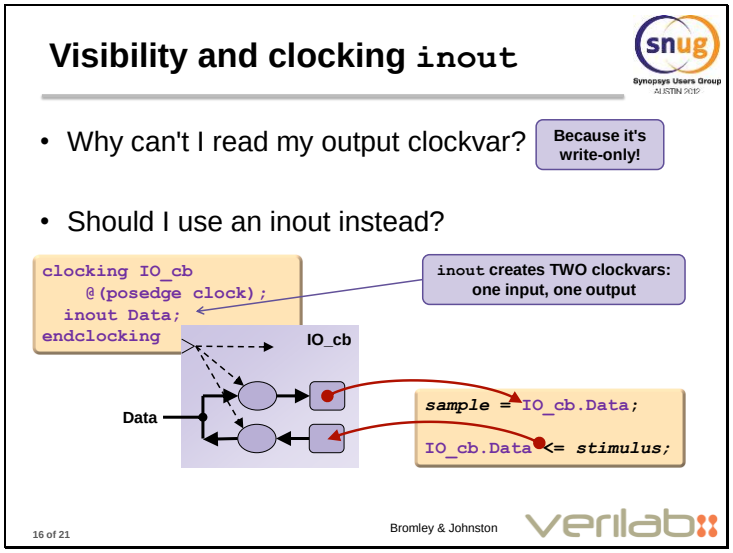
Slide 15



This slide summarizes our guidelines concerning the use of clocking blocks with classes. Once again, full details can be found in the published paper that accompanies these slides.

It also adds a further piece of advice taken from our experience. If your RTL (design) code makes use of interfaces for interconnect between design modules, it may be very tempting to try to re-use those interfaces to contain testbench-facing clocking blocks and modports. We believe this is inappropriate. The coding requirements for design-oriented and testbench-oriented interfaces are so different that they can almost never be reconciled in one and the same piece of code. Keep your verification IP and your design IP separate.

Slide 16



It is fairly obvious that an *input* clockvar is read-only, and cannot be written or modified by your own code. It can be quite surprising, though, to find that an *output* clockvar is strictly write-only, and there is no way to read back its value in your testbench code.

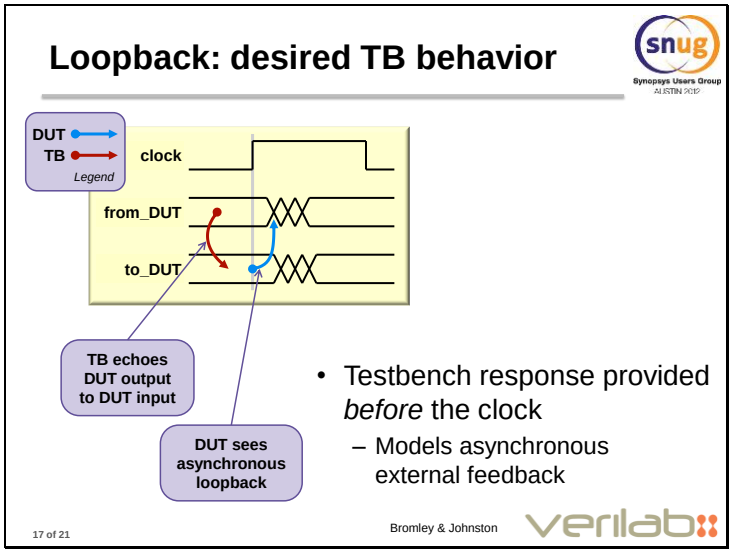
What does this mean?

An output clockvar contains the *projected* value that the clocking block will drive on to the relevant signal, at the appropriate time after each clock event. Reading this clockvar is definitely not the same thing as reading the signal itself! However, many users try to read their output clockvars, perhaps as a way to check the value that has been placed there by testbench code execution. This is illegal according to the SystemVerilog standard, and should be rejected by any conforming simulator (although most simulators will allow you to display the clockvar in a waveform viewer).

Frustrated by this, some users try to create *inout* clockvars so that they can read the value back. However, this does not come close to providing you with the clockvar's value. In fact, an *inout* declaration in a clocking block provides you with TWO clockvars – one input, one output – having the same name. Writing to that name in fact writes to the output clockvar. Reading the name in fact reads the corresponding input clockvar.

inout clockvars are very useful as a way to allow the TB to work with bidirectional bus signals, and to allow the TB to choose dynamically whether it will drive a signal or merely monitor it passively. If the signal in question is a net rather than a variable, then the output side of the inout will initialize to driving 'z and will have no effect on the target signal until it is written with some other value. In this way, the TB can choose dynamically (for example, as part of UVM configuration) whether it is to be an active driver or a passive monitor on the signal.

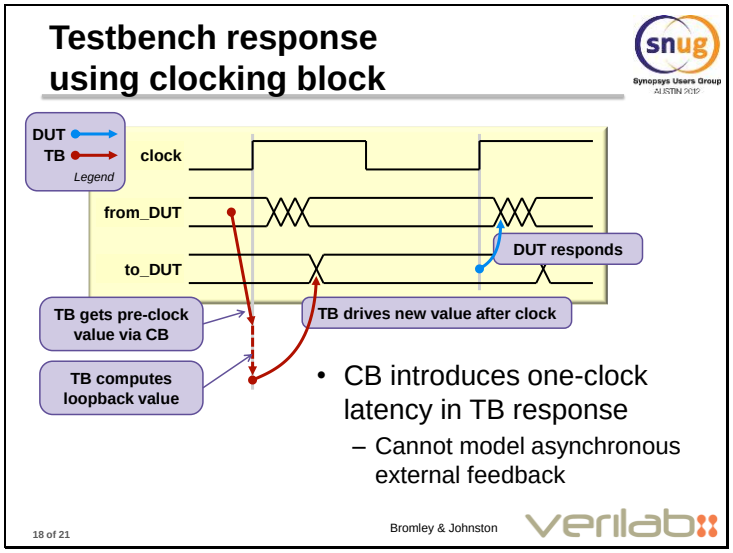
Slide 17



Finally, we take a brief look at a verification scenario to which clocking blocks are perhaps not ideally suited: the provision of asynchronous loopback or feedback from a DUT's output to its input.

This timing diagram shows what is typically required: the DUT's output *from_DUT* (or some combinational function based on it) must be echoed back to the DUT's input *to_DUT*. In this way, the DUT's output just before a clock edge can be used to provide stimulus that will influence the DUT's behavior on that same clock edge. As we shall shortly see, clocking blocks make that a little difficult to achieve.

Slide 18



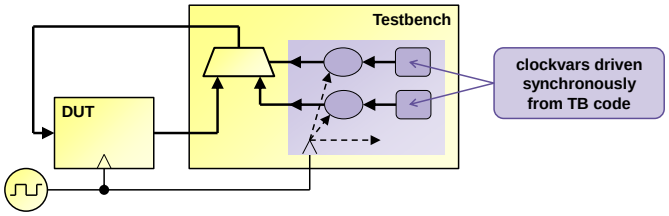
Here's a naive attempt to get the same asynchronous-loopback effect, using a TB that reaches the DUT's signals through a clocking block. Precisely because the TB performs all its activity at the exact moment of the clock edge, it is impossible for the TB to provide combinatorial (asynchronous) feedback. The intervening clocking block causes our TB to behave in much the same way as a piece of synchronous RTL, introducing a one-clock latency.

Fortunately, this problem can be solved in a reasonably flexible way by adding some additional structures in the test harness around our DUT.

Slide 19

Achieving asynchronous response

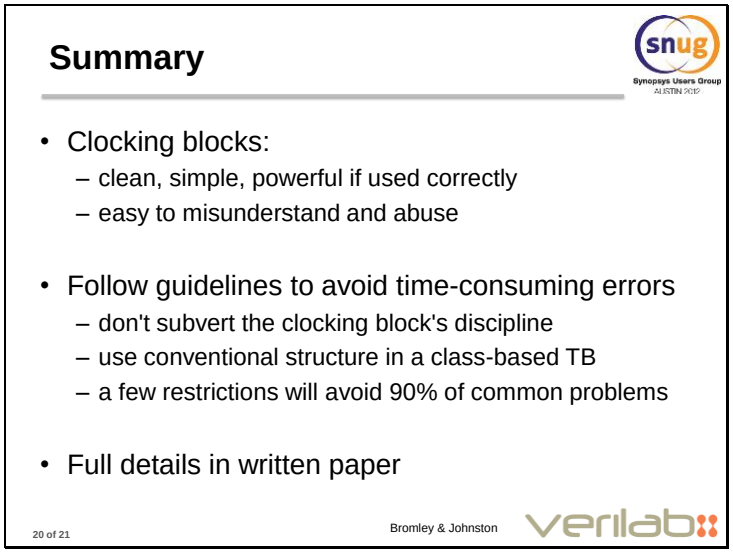
- **Cannot** be done directly
  - Don't access clockvars asynchronously
- Consider async feedback mux, controlled synchronously through the clocking block



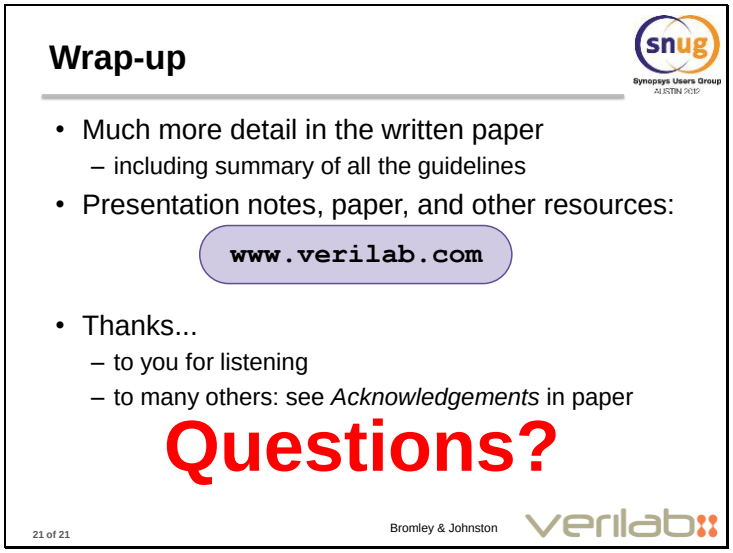
Bromley & Johnston

Here is a sketch of our recommended solution. Our testbench (or, perhaps, our verification component's interface) includes a feedback multiplexer, written as purely combinational Verilog. This multiplexer provides a configurable asynchronous path from DUT output to DUT input, all the while preserving the benefits of a strictly synchronous view of things from the TB side. The key is to provide some additional outputs from the clocking block that can be used to manipulate or configure the combinational feedback path.

Slide 20



In summary: the misuse of clocking blocks, which we see so often, can readily be avoided by following a few simple guidelines or – better! – by gaining a straightforward but robust understanding of clocking blocks' internal operation. Our full paper includes much more detail about these topics, and includes detailed references to the SystemVerilog standard's definition of clocking blocks.



In the presentation we had too little time to mention a whole bunch of useful clocking block features… clocking expressions, assertions, multi-clock (DDR). That, and much more detail on the reasoning behind our guidelines, is in the written paper.