



Applying Stimulus & Sampling Outputs - UVM Verification Testing Techniques

Clifford E. Cummings

Sunburst Design, Inc.
Provo, Utah, USA

www.sunburst-design.com

ABSTRACT

When should testbench stimulus vectors be applied to a design? When should design outputs be verified by the testbench? How should UVM drive and sample DUT signals?

The details of driving stimulus and sampling outputs is one of the most ad hoc habits of many verification engineers, and little thought has been given to the best usage strategies and reasons for using them.

This paper will detail fundamental techniques that have been proven to work with all phases of design verification. Discussed in this paper are the tradeoffs of three different techniques to apply stimulus. Also discussed in this paper are the best techniques for sampling DUT outputs. Finally, useful new SystemVerilog verification features such as clocking blocks and #1step sampling will be discussed. This paper also explains why the addition of the SystemVerilog program keyword was a bad idea and why it should not be used.

Table of Contents

1. Introduction	4
1.1 Introduction to UVM methodologies.....	4
1.2 Introduction to terminology.....	4
1.3 Assessing knowledge.....	5
2. Time-0 race conditions.....	5
2.1 Time-0 potential problem	5
2.2 Time-0 initial and always block behavior	8
2.3 Time-0 stimulus assignments	8
3. Verification goal	8
4. Stimulus Timing	9
5. Driving stimulus on the active clock edge - Avoid this.....	9
5.1 Testbench blocking assignments.....	10
5.2 I/O pads in the RTL model	10
5.3 Gate-level simulations with setup and hold delays	11
5.4 Waveform display debugging	11
6. Driving stimulus on the inactive clock edge - Avoid this.....	12
6.1 Inactive-clock stimulus problems	12
7. Driving stimulus using time budgeting - Use this.....	14
8. Verification Timing	15
8.1 Sampling outputs on the active clock edge	15
8.2 Sampling outputs just before the next active clock edge.....	16
9. Clocking Blocks.....	16
9.1 Clocking Block Default Timing.....	17
9.2 #1step Sampling.....	17
9.3 #0 Drive Times.....	17
9.4 Example UVM clocking block.....	18
9.5 Stimulus using clocking drives	19
9.6 Why drive signals at time-0?	20
9.7 Asynchronous control inputs.....	21
9.8 Interface modports and testbenches	25
10. Death to the SystemVerilog program!	25
10.1 Cliff's confession	27
11. Conclusions.....	27

12. References.....	27
---------------------	----

Table of Figures

Figure 1 - VCS simulation race-condition output of Example 1	6
Figure 2 - Simulator-B simulation race-condition output of Example 1.....	6
Figure 3 - VCS simulation NO-race-condition output of Example 2	8
Figure 4 - Simulator-B simulation NO-race-condition output of Example 2.....	8
Figure 5 - Example of real hardware timing.....	9
Figure 6 - Example of applying stimulus on the active clock edge - prone to testbench race conditions	10
Figure 7 - Asymmetrical stimulus clock - change stimulus on the negedge of (stim) clk.....	12
Figure 8 - Stimulus generation for dual-clock logic - change stimulus on the negedge of vclk	13
Figure 9 - Synthesis constraint settings using time budgeting.....	14
Figure 10 - Example clockvars driven using the clocking block name.....	15
Figure 11 - Four asynchronous reset signal scenarios	21
Figure 12 - Asynchronous mid-cycle, sub-cycle reset pulse<insert common code example here>	23
Figure 13 - SystemVerilog module and program event scheduling	26

Table of Examples

Example 1 - Time-0 blocking assignment - race condition	6
Example 2 - Time-0 nonblocking assignment - NO race condition.....	7
Example 3 - SystemVerilog code to generate asymmetrical stimulus clock.....	13
Example 4 - SystemVerilog code to generate dual-clock logic stimulus clock.....	14
Example 5 - Clocking block format using 20% time budget to drive stimulus	14
Example 6 - Program counter DUT code.....	18
Example 7 - CYCLE.sv file	18
Example 8 - DUT interface with clocking block	19
Example 9 - tb_driver with initialize() task (no clocking block timing) and drive_tr() ask (uses clocking block timing)	20
Example 10 - sample_dut task checks async reset at beginning and end of the cycle	22
Example 11 - tb_monitor checks async reset at beginning and end of the cycle.....	23
Example 12 - DUT interface with sticky-bit code to save reset short-pulse reset condition	24
Example 13 - tb_monitor modified to test the sticky-bit reset_n version of the rst_n asynchronous reset.....	25

1. Introduction

Although much is known about design verification, little has been written about the strategies that can and should be used to apply stimulus vectors and sample DUT outputs for design validation.

Troublesome verification issues include: Time-0 simulation race conditions, how to apply stimulus to reduce simulation race conditions, stimulus techniques that do not require changing the testbench when timing delays and timing checks are added to the simulation, a consistent way to verify the design outputs, and how to handle asynchronous control signals in verification.

This paper will show common stimulus generation techniques and present guidelines for Best Known Practices. The best technique will also be shown within a common UVM driver in Section 9.5

This paper will also show the Best Known Practices for sampling DUT outputs and the best technique will be incorporated into a common UVM monitor as shown in Section 9.7

This paper will also detail problems related to time-0 simulation issues and how to avoid the problems.

Although this paper describes when to sample DUT outputs for verification purposes, it does not describe how to build a verification scoreboard for the sampled outputs. A paper describing UVM scoreboard architectures can be seen in [2].

1.1 Introduction to UVM methodologies

The testbench techniques described in this paper cover both SystemVerilog and UVM approaches. UVM verification environments typically connect a DUT to an **interface** that includes a **clocking** block. The handle of this **interface** is typically stored in a **uvm_config_db** that is accessed as a **virtual interface** by the UVM driver and monitor, or accessed by the UVM agent and the agent copies the **virtual interface** handle to the driver and monitor that are built by the agent. The driver then uses **clocking** drives that accesses the **clocking** block in the real **interface** to drive stimulus and the monitor uses clocking samples that again accesses the **clocking** block in the real **interface** to sample DUT outputs.

The use of **clocking** blocks for testbenches is described in Section 9 of this paper.

Examples of **interfaces** that are used in a UVM testbench are shown in Example 8 and in Example 12. These **interfaces** are typically instantiated in a top-level module and the **interface** handles stored in a **uvm_config_db** for retrieval by the UVM testbench classes.

1.2 Introduction to terminology

There are some terms used in this paper that might not be familiar to some verification engineers and cause undue confusion. Below are descriptions of a couple of terms that could help verification engineers better understand the concepts discussed in this paper.

clockvar is the term used to describe the signals declared in a **clocking** block as described by the SystemVerilog Standard [5] and by Bromley and Johnston [7].

clocking signal is the name of the signals declared in the **clocking** block and generally, a clocking signal and the clockvar names are the same [5].

clocking drive refers to stimulus that is driven using **clocking** block timing defined for a clockvar. The clocking drive operation makes assignments to clockvars using the **clocking** block name and the assignments are made using the clocking drive operator (**<=**), which is the same operator that is used for nonblocking assignments [5].

1.3 Assessing knowledge

When assessing skills of new college graduates, job candidates or even your current verification engineers, I suggest the following assessment scale: 40% credit for properly starting up a verification test. 40% credit for properly shutting down a verification test. 20% credit for properly testing the DUT after the testbench has started. Almost anybody can get the middle part of a test to work correctly but properly starting and terminating a test is where real verification skill is required. My experience has shown that a large portion of the test debug time is related to the start-up and shut-down of a test. Talented engineers can avoid these prolonged debug issues and those are the skills that will be shown in this paper.

2. Time-0 race conditions

Before starting to develop a SystemVerilog or UVM testbench, an engineer needs to consider what happens at time-0 during a simulation.

Time-0 is a tricky place in Verilog and SystemVerilog simulations. It is easy to experience lengthy race-condition debugging issues at time-0. If you follow a couple of simple guidelines, it is just as easy to avoid 100% of the time-0 race conditions.

2.1 Time-0 potential problem

One of the potential problems related to time-0 race conditions is that the IEEE Verilog and SystemVerilog Standards require all procedural blocks (**initial** blocks and **always** blocks) to start execution at the beginning of the simulation at time-0 but there is no defined order of execution of these blocks at time-0. The SystemVerilog code of Example 1 has a time-0 race condition:

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

module initial_always1;
    logic clk;

    initial $timeformat(-9,0,"ns",6);

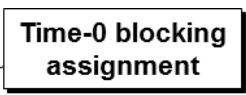
    initial @(negedge clk) $display("%t: initial #1 negedge clk", $time);

    always begin
        @(negedge clk) $display("%t: always #1 negedge clk", $time);
        wait(0);
    end

    initial begin
        clk = '0;
        forever #(`CYCLE/2) clk = ~clk;
    end

    initial @(negedge clk) $display("%t: initial #2 negedge clk", $time);

    always begin
        @(negedge clk) $display("%t: always #2 negedge clk", $time);
        wait(0);
    end
end
```



```

initial begin
    repeat(2) @(negedge clk);
    FINISH();
end

task FINISH();
    @(posedge clk);
    $display("%t: FINISH\n\n", $time);
    $finish;
endtask
endmodule

```

Example 1 - Time-0 blocking assignment - race condition

Note that the clock oscillator in Example 1 has a time-0 `negedge clk` assignment using a blocking assignment. The example also has an `initial` block (`initial #1`) and `always` block (`always #1`) that trigger on the `negedge clk` positioned in the code before the clock oscillator and another `initial` block (`initial #2`) and `always` block (`always #2`) that trigger on the `negedge clk` positioned in the code after the clock oscillator. If the clock oscillator starts *before* the `initial` and `always` blocks are active, those blocks will not trigger until one cycle after the simulation starts. If the clock oscillator starts *after* the `initial` and `always` blocks are active, those blocks will trigger at time-0.

When VCS runs this simulation, the output is shown in Figure 1. Note that all of the blocks triggered at time-0 except for the `initial` block that was placed after the clock oscillator. This is perfectly legal behavior for Verilog and SystemVerilog simulators.

```

0ns: initial #1 negedge clk
0ns: always #1 negedge clk
0ns: always #2 negedge clk
10ns: initial #2 negedge clk
25ns: FINISH

```

Figure 1 - VCS simulation race-condition output of Example 1

When another simulator ("Simulator-B") runs this simulation, the output is shown in Figure 2. Note that the blocks that preceded the clock oscillator triggered at time-0 while the blocks that were positioned after the clock oscillator did not trigger until one cycle later. This too is perfectly legal behavior for Verilog and SystemVerilog simulators.

```

0ns: always #1 negedge clk
0ns: initial #1 negedge clk
10ns: always #2 negedge clk
10ns: initial #2 negedge clk
25ns: FINISH

```

Figure 2 - Simulator-B simulation race-condition output of Example 1

Using blocking assignments at time-0 frequently causes a race condition. This race condition can be avoided by using nonblocking assignments at time-0.

The modified SystemVerilog code of Example 2 uses a nonblocking assignment for the first assignment in the clock oscillator, which removes the time-0 race condition.

```

`define CYCLE 10
`timescale 1ns/1ns

module initial_always2;
  logic clk;

  initial $timeformat(-9,0,"ns",6);

  initial @(negedge clk) $display("%t: initial #1 negedge clk", $time);

  always begin
    @(negedge clk) $display("%t: always #1 negedge clk", $time);
    wait(0);
  end

  initial begin
    clk <= '0;
    forever #(`CYCLE/2) clk = ~clk;
  end

  initial @(negedge clk) $display("%t: initial #2 negedge clk", $time);

  always begin
    @(negedge clk) $display("%t: always #2 negedge clk", $time);
    wait(0);
  end

  initial begin
    repeat(2) @(negedge clk);
    FINISH();
  end

  task FINISH();
    @(posedge clk);
    $display("%t: FINISH\n\n", $time);
    $finish;
  endtask
endmodule

```



Example 2 - Time-0 nonblocking assignment - NO race condition

Note that the clock oscillator in Example 2 has a time-0 **negedge clk** assignment that now uses a nonblocking assignment. The example still has an **initial** block (**initial #1**) and **always** block (**always #1**) that trigger on the **negedge clk** positioned in the code before the clock oscillator and another **initial** block (**initial #2**) and **always** block (**always #2**) that trigger on the **negedge clk** positioned in the code after the clock oscillator. Even if the clock oscillator starts *before* the **initial** and **always** blocks are active, the clock assignment will not complete until after the other blocks have become active.

When VCS runs this simulation, the output is shown in Figure 3. Note that all of the blocks triggered at time-0. The time-0 race condition has been removed.

```

0ns: initial #1 negedge clk
0ns: always #1 negedge clk
0ns: initial #2 negedge clk
0ns: always #2 negedge clk
15ns: FINISH

```

Figure 3 - VCS simulation NO-race-condition output of Example 2

When another simulator ("Simulator-B") runs this simulation, the output is shown in Figure 4. Note that even though the code has executed in a slightly different order, again all of the blocks triggered at time-0. The time-0 race condition has been removed, and both simulations give the same result.

It is also worth noting that the **FINISH** command executed one cycle earlier using both simulators because the **initial** block with the **FINISH** command was also now active at time-0.

```

0ns: always #2 negedge clk
0ns: initial #2 negedge clk
0ns: always #1 negedge clk
0ns: initial #1 negedge clk
15ns: FINISH

```

Figure 4 - Simulator-B simulation NO-race-condition output of Example 2

2.2 Time-0 initial and always block behavior

Although all of the simulation results shown in Section 2.1 are legal, in practice the major simulation vendors frequently start up **always** blocks before **initial** blocks at time-0. This behavior is not guaranteed by the IEEE Verilog and SystemVerilog Standards, but this, and the fact that most testbenches drive stimulus across **module** ports, is why most testbenches work correctly at time-0. RTL designs are typically coded using **always** blocks and testbenches are typically coded using **initial** blocks, so the RTL designs typically become active at time-0 before the **initial** blocks send the first stimulus.

This is a point of confusion for most new Verilog users because it sounds like an **initial** block should execute first at time-0, but this is not what happens. A more accurate name for the **initial** block would have been a **run_once** block!

2.3 Time-0 stimulus assignments

The best guideline to follow to avoid time-0 race conditions is to make all time-0 stimulus assignments using nonblocking assignments. After time-0, all other stimulus assignments can be made using blocking assignments if stimulus is driven using the time budgeting technique described in Section 7.

This is the race-free stimulus driving technique that I have used successfully for more than a 10 years.

3. Verification goal

When building a testbench, engineers need to ask these questions:

1. When should test vectors be driven?
2. When outputs should be sampled.

The goal is to construct a testbench that can be used for behavioral models, for 0-delay RTL designs and for gate-level simulations that include backannotated SDF timing with no modification to the

testbench. Engineers certainly do not want to maintain two or more separate testbenches due to poor testbench planning and timing issues related to different DUT implementations.

4. Stimulus Timing

When should stimulus vectors be applied to a design? Is the stimulus timing realistic when compared to real-world design constraints?

There are three primary stimulus generation techniques that have been used for decades by engineers responsible for building verification environments: (1) apply stimulus on the active clock edge, (2) apply stimulus on the inactive clock edge, and (3) apply stimulus using Time Budgeting techniques.

These techniques, along with new SystemVerilog clocking drive techniques, are described with their advantages and potential pitfalls in the following sections.

5. Driving stimulus on the active clock edge - Avoid this

In theory, driving stimulus vectors on the active clock edge should work with 0-delay RTL models and many engineers commonly use this technique. I highly discourage this practice for reasons that are described later in this section, but if this technique is used, verification engineers need to understand the limitations and potential pitfalls of this technique.

It is frequently claimed that applying stimulus on an active clock edge more closely replicates actual hardware behavior, but this is not true. In real hardware, input data frequently changes nanoseconds after an active clock edge is observed, as shown in Figure 5.

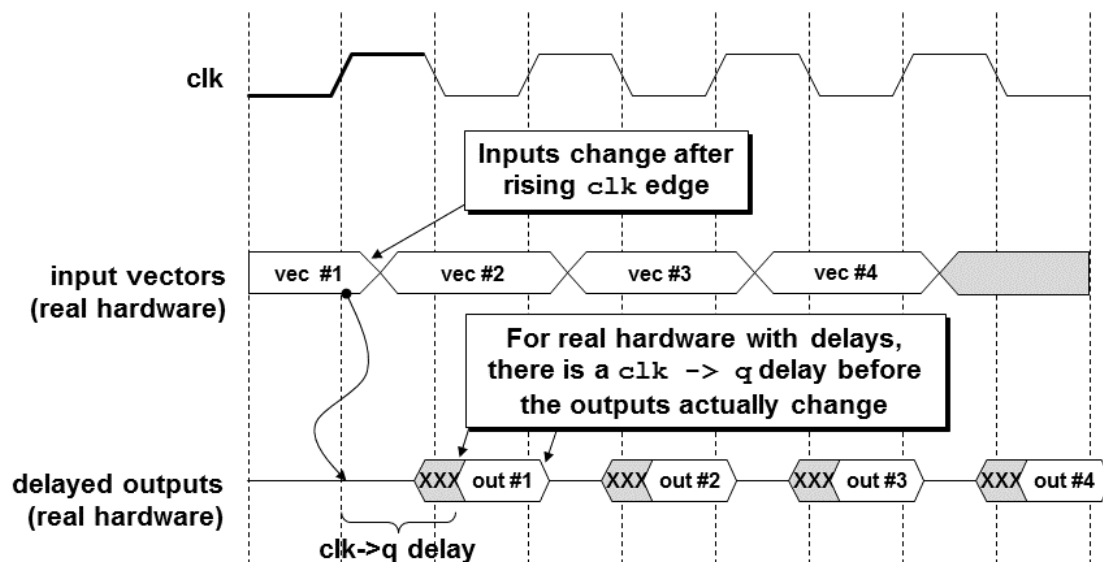


Figure 5 - Example of real hardware timing

The non-recommended practice of applying testbench stimulus on the active clock edge would be a setup or hold time violation in real hardware, and prone to simulation race conditions as shown in Figure 6.

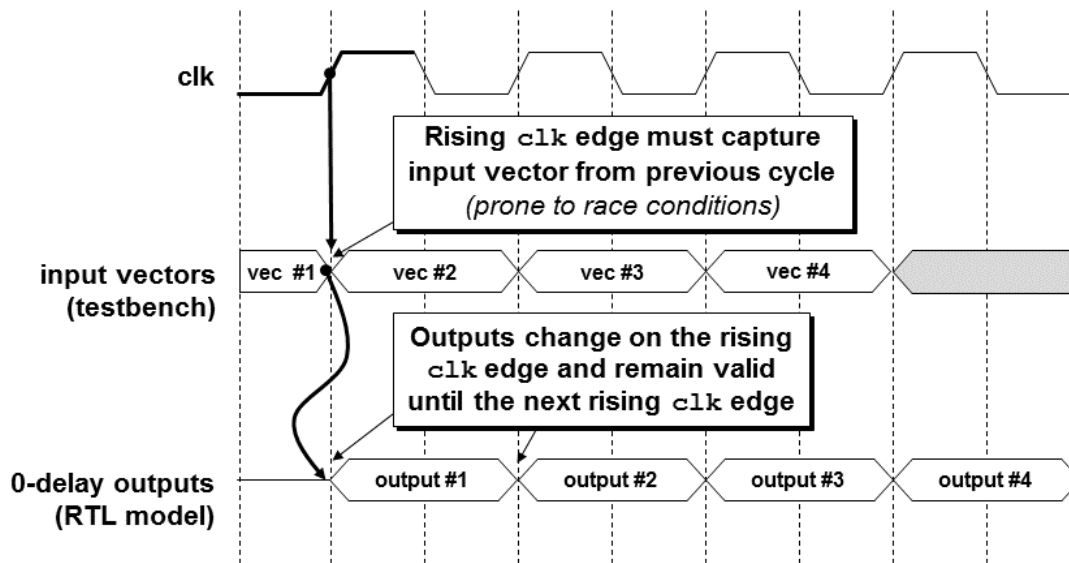


Figure 6 - Example of applying stimulus on the active clock edge - prone to testbench race conditions

For Verilog verification, a key to using the active clock-edge stimulus technique is to drive all stimulus from the testbench using nonblocking assignments, which should guarantee that design inputs change after the exact same active clock edge has been used to sample signals from the previous cycle. You do not want to change inputs on a clock edge and have the same clock edge capture the signals that just changed. Real hardware does not behave that way.

For SystemVerilog verification, a key to using the active clock-edge stimulus technique is to drive the stimulus using one of the following: nonblocking assignments from modules, any type of assignment from a **program**, or driving assignments from a **clocking** block originating from either a **module**, a **class** or a **program**.

In practice, there are many situations where applying stimulus on the active clock edge can cause unnecessary, time-consuming debug difficulties related to stimulus-design race conditions. Below are some of the potential race conditions that can occur.

So what are the potential problems and mistakes associated with applying stimulus on the active clock edge?

5.1 Testbench blocking assignments

The first potential problem is a rather obvious mistake that happens frequently but can be detected and corrected quickly. The simple mistake is that the testbench-writer used a blocking assignment from a **module**-based testbench to apply stimulus, and some of the stimulus inputs changed before the active clock edge had a chance to sample the previous DUT inputs. Although simple and obvious, this mistake still happens frequently, especially with new users of Verilog and SystemVerilog.

5.2 I/O pads in the RTL model

Even when doing 0-delay RTL modeling, it is not uncommon for engineers to instantiate I/O pads in the top-level module to communicate with the rest of the design. If the data path I/O pads have short delays and the clock-tree I/O pads have slightly longer delays, the previous input data values will be changed by the testbench before they are sampled at the DUT inputs on the active clock edge. This is not how the real hardware will work.

This problem can be avoided by adding right-hand-side (RHS) delays to the stimulus nonblocking assignments, effectively delaying the change of the data inputs until after the active clock edge, which closely replicates the actual behavior of real hardware. Adding RHS delays to stimulus assignments is somewhat of a coding nuisance, and could cause simulations to run a little slower, but it does solve the problem and is described in [3].

Using SystemVerilog clocking blocks and stimulus clocking drives can also place delays on the stimulus data, thus localizing the delays into a common `clocking` block, but be aware that the default `clocking` block delay is 0, so a non-zero default `clocking` block output will be necessary to add the equivalent delays.

5.3 Gate-level simulations with setup and hold delays

One of the major problems related to applying stimulus on the active clock edge manifests itself in gate-level simulations with backannotated delays, including setup and hold time checking. A gate-level design with actual clock tree logic typically introduces more delay into the clock path than is introduced into the data path. In an actual hardware design this typically is not a problem, because there are actual clock-to-q delays on the driven signals to a design, but when an actual design is driven by a testbench that changes both the clock and data stimulus signals at the same time, it can cause the data signals to change on the inputs of registered devices hundreds of picoseconds to even a couple of nanoseconds before the active clock edge traverses the clock tree to the clock input of the registers. The differential in actual data path versus clock path delays is frequent enough to cause a setup or hold time violation for the gate-level design interacting with the 0-delay testbench. These violations typically cause simulation X's to be propagated throughout the gate-level design, causing verification to fail.

Again, to avoid this problem when applying stimulus on the active clock edge, the verification engineer either needs to add RHS delays to the stimulus nonblocking assignments, or add clocking drive delays to a SystemVerilog testbench.

An important goal of testbench development is to use the same testbench for both 0-delay RTL designs and gate-level designs that include delays and timing checks. Using active clock stimulus (typically on the posedge clock) violates this goal.

Guideline: do not apply testbench stimulus on the active clock edge.

5.4 Waveform display debugging

When stimulus inputs change on the active clock edge, debugging a 0-delay RTL design in a waveform display can be confusing to design engineers. The confusion arises because in a waveform display, the input changes coincident with the rising clock edge and the registered output, as was shown in Figure 6. Although the results are correct for a 0-delay RTL simulation, any hardware engineer with real-world experience will find it strange that the inputs changed coincident with the rising clock edge and it will appear that the inputs have potentially violated real setup and hold times. This is one reason that engineers frequently add `#1` delays to the RHS of nonblocking assignments, so they can see a clk-to-q delay in the waveform display. A technique describing the use of `#1` clk-to-q delays is described in [3].

If careful, a verification engineer can make the active-clock stimulus technique work, but there are fewer chances for errors if stimulus is applied some time after the active clock edge. I have found that engineers make fewer mistakes and spend less time debugging stimulus-simulation race conditions by applying the stimulus away from the active clock edge.

6. Driving stimulus on the inactive clock edge - Avoid this

To avoid all of the problems related to changing inputs on the active clock edge and to ensure that the same stimulus vectors can be used for 0-delay RTL simulations and gate-level simulations with backannotated timing, for more than 10 years I used the technique of applying input stimulus vectors on the inactive clock edge (typically the `negedge clk`).

Applying vectors on the inactive clock edge accomplishes the following goals:

1. Stimulus can be applied using either blocking or nonblocking assignments from either a `module` or a `program`. The active clock captures the inputs and since the next inputs are not placed on the DUT inputs until the next inactive clock edge, there is never a simulation race condition related to the proper or improper use of `modules`, `programs`, blocking or nonblocking assignments.
2. 0-delay RTL models with top-level I/O pads and accompanying delays never cause problems. DUT inputs change far away from propagated clocks.
3. Gate-level simulations with delays (including setup and hold time checks) are never a problem, except when the input combinational data paths are longer than half of the clock period. This is addressed in Section 7.
4. Waveform displays of input stimulus are easy to understand. When the stimulus changes on the inactive clock edge, any input combinational logic will react immediately and setup on the register inputs of the design. The next active clock edge will then capture the stimulus and settle immediately (0-delay RTL models) or shortly thereafter (gate-level models with delays), which is highly intuitive to most hardware design engineers.

6.1 Inactive-clock stimulus problems

So what are the potential problems associated with applying stimulus on the inactive clock edge?

The most frequent problem associated with applying stimulus on the inactive clock edge is that there is now only half of a clock cycle for the primary DUT data inputs to propagate to the input registers of the design and meet the register setup time when running gate-level simulations with backannotated delays.

There are two Verilog approaches to address this potential problem. Both approaches use time-budgeting techniques to address the issues.

The first is to build a clock oscillator with a short high pulse width and a long low pulse width. The short high pulse width is the time budgeted from the beginning of the cycle until when the stimulus inputs will change. The low pulse width is the time budgeted for the stimulus inputs to propagate through the DUT input combinatorial logic. In the following example, it is assumed that the clock period is 10ns and that the stimulus can change 2ns after the active clock edge.

As shown in Figure 7, the stimulus vectors should be applied at time-0 and on each `negedge` of the (stim) `clk`.

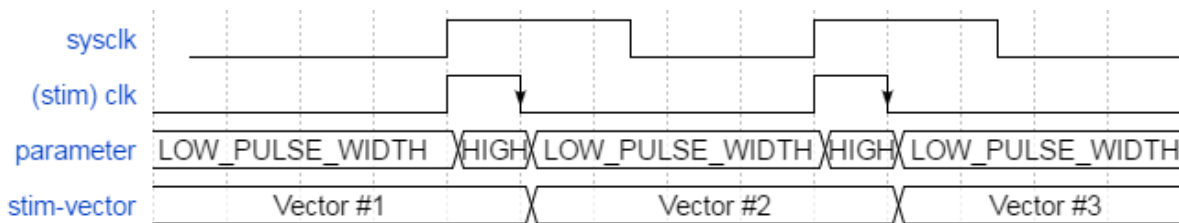


Figure 7 - Asymmetrical stimulus clock - change stimulus on the negedge of (stim) `clk`

The corresponding SystemVerilog code to generate this asymmetrical clock is shown in Example 3.

```
`define CYCLE 10
`timescale 1ns/100ps // Use appropriate timescale resolution

...
parameter HIGH_PULSE_WIDTH = 2; // Choose an appropriate clk-to-q delay
parameter LOW_PULSE_WIDTH = `CYCLE - HIGH_PULSE_WIDTH;
...
initial begin // virtual stimulus clock
    clk <= '0;
    forever begin
        #( LOW_PULSE_WIDTH); clk = '1;
        #(HIGH_PULSE_WIDTH); clk = '0;
    end
end
...
```

Example 3 - SystemVerilog code to generate asymmetrical stimulus clock

This technique works fine as long as only the active system clock edge is used by the DUT and that there are no transparent latches enabled by the system clock or flip-flops that trigger on the negative edge clock.

If both clock edges are separately used to trigger flip-flops or if the clock level is used in some type of latching logic configuration, then I recommend using a 2X virtual clock with skewed duty cycle to accomplish the same goal.

As shown in Figure 7, the stimulus vectors should be applied at time-0 and on each **negedge** of the **vclk**

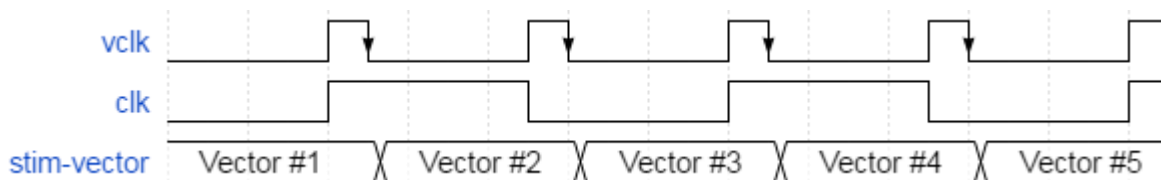


Figure 8 - Stimulus generation for dual-clock logic - change stimulus on the negedge of vclk

The corresponding SystemVerilog code to generate this dual-clock logic stimulus clock is shown in Example 4. The rising virtual clock edge is used to toggle the DUT clk.

```
`define VCYCLE 5 // Actual clock cycle is 10ns
`timescale 1ns/100ps // Use appropriate timescale resolution

...
parameter HIGH_PULSE_WIDTH = 1;
parameter LOW_PULSE_WIDTH = `VCYCLE - HIGH_PULSE_WIDTH;
...
initial begin // virtual stimulus clock
    vclk <= '0;
    forever begin
        #( LOW_PULSE_WIDTH); vclk = '1; // This example: 4ns
        #(HIGH_PULSE_WIDTH); vclk = '0; // This example: 1ns
    end
end
...
```

```

    end
end

initial begin // actual design clock
    clk <= '0;
    forever @(posedge vclk) clk = ~clk;
end
...

```

Example 4 - SystemVerilog code to generate dual-clock logic stimulus clock

7. Driving stimulus using time budgeting - Use this

The concept of time budgeting is a technique that has long been used in synthesis and was described in a 1997 SNUG paper by Anna Ekstrandh and Wayne Bell [1]. A recommended synthesis technique is to register all module outputs and only allow combinatorial logic on the module inputs so that the synthesis compiler would use most of the clock cycle to meet combinational input constraints and very little of the clock cycle would be required to constrain the clk-to-q output of the **module**, as shown in Figure 9.

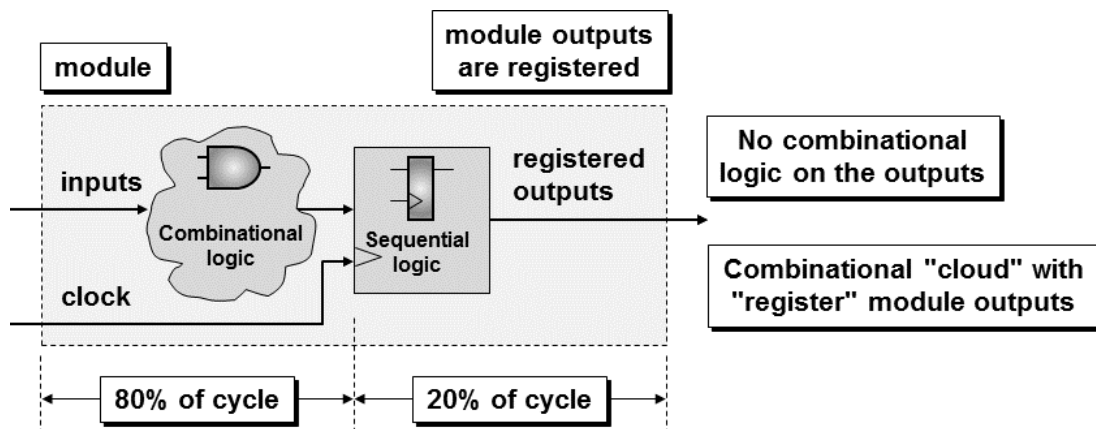


Figure 9 - Synthesis constraint settings using time budgeting

In the example of Figure 9, a synthesis tool would be instructed to allocate a small percentage of the clock cycle to meet register-to-module-outputs timing (20% of the cycle in this example) and allocate the majority of the clock cycle to meet inputs-to-registered-logic timing (80% of the cycle in this example).

The same concept can be used to drive stimulus into a Verilog or SystemVerilog design.

SystemVerilog provides a **clocking** block to help define the time budgets of stimulus drives and output samples. Note that the **clocking** block can be placed in a **module**, **program** or **interface**.

```

clocking cb1 @(posedge clk)
    default input #1step output (`CYCLE * 0.2);
    input <list of all inputs> ;
    output <list of all outputs>;
endclocking

```

Example 5 - Clocking block format using 20% time budget to drive stimulus

Remember that testbench outputs are the stimulus that is driven into the DUT and testbench inputs are the DUT signals that were sampled and passed to the testbench.

To drive signals using the specified **clocking** block delays, the **clocking** block name must be used with **clocking** drives to use the specified timing as shown in Figure 10.

```
task drive_tr (transl tr);
  @vif.cb1;
  vif.cb1.din    <= tr.din;
  vif.cb1.ld     <= tr.ld;
  vif.cb1.inc    <= tr.inc;
  vif.cb1.rst_n <= tr.rst_n;
endtask
```

Figure 10 - Example clockvars driven using the clocking block name

The assignments in Figure 10 contain three parts on the left-side of the clocking drive operator. The first part is the handle to the **virtual interface** (**vif** in this example). The second part is the name of the **clocking** block in the **interface** (**cb1** in this example). The third part is the name of the signals in the **interface** that are to be driven using the **clocking** block timing (**din**, **ld**, **inc**, and **rst_n** in this example). These 3-part-reference signals are referred to as clockvars as described in the SystemVerilog Standard and in a paper by Bromley and Johnston [7].

8. Verification Timing

When should design outputs be verified by the testbench? How can a verification strategy be formulated to work with both 0-delay RTL models and gate-level models with delays?

There are two primary verification timing techniques that have been used for decades by engineers responsible for building verification environments: (1) sample design outputs on the active clock edge, (2) sample design outputs just before the next active clock edge.

These techniques, along with new SystemVerilog **clocking** block sampling techniques, are described with their advantages and potential pitfalls in this section.

8.1 Sampling outputs on the active clock edge

In theory, sampling design outputs on the active clock edge should work with 0-delay RTL models and some engineers do use this technique. I generally discourage this practice for reasons that will be described later, but if this technique is used, verification engineers need to understand the limitations and potential pitfalls of sampling on the active clock edge.

For Verilog verification, a key to sampling outputs on the active clock-edge is that the input stimulus has to be driven using nonblocking assignments. The theory behind this technique is that stimulus driven with nonblocking assignments will not cause design outputs to be updated until the Verilog nonblocking assignment event region, which in theory means that the old design outputs should still be valid and available to be sampled for verification testing until the new stimulus has been clocked into the design. The design outputs that are being sampled were clocked on the previous rising clock edge and should have settled to their final value almost a full clock cycle earlier.

For SystemVerilog verification, a key to using the active clock-edge stimulus technique is to drive the stimulus using one of the following: nonblocking assignments from modules, any type of assignment from a **program** block, or driving assignments from a **clocking** block originating from either a **module**, a **class** or a **program**.

8.2 Sampling outputs just before the next active clock edge

The best place to sample DUT outputs is at the end of the cycle just before the next active clock edge and SystemVerilog introduced a new type of sampling delay called the `#1step` to help accomplish this goal. The use of the `#1step` sampling delay is describe in Section 9.2

9. Clocking Blocks

Clocking blocks play an important role in controlling stimulus driving and output sampling timing in a testbench, especially in a UVM testbench environment.

An excellent paper by Jonathan Bromley and Kevin Johnston [7] goes into great detail on how the SystemVerilog `clocking` block works and some of its lesser known capabilities and quirks. The reader is encouraged to read that entire paper for a greater understanding of `clocking` blocks.

Bromley and Johnston also shared 11 guidelines in their paper, most of which I strongly agree with but there are a couple of exceptions or further clarifications that I will make in this section.

The Bromley and Johnston guidelines are:

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

**** I mostly agree but an important exception is described in Section 9.5**

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.

**** I agree - follow this guideline.**

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

**** I agree - further clarification is described at the end of Section 9.5**

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.

**** I agree - an additional important reason is described in Section 9.2 .**

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.

**** I agree - an addition to this guideline is described in Section 9.3**

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

**** I agree - follow this guideline.**

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

**** I agree - follow this guideline**

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.

**** I agree - follow this 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.

**** I mostly agree - I will comment on the modport portion of this guideline in Section 9.8**

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.

**** I agree - follow this 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`

**** I mostly agree - I will comment on the modport portion of this guideline in Section 9.8**

In addition to guideline #3, I also discuss at the end of Section 9.7 the proper coding style for sampling DUT outputs using a clocking block.

9.1 Clocking Block Default Timing

Clocking block default timing values are `#1step` for sampling DUT signals and `#0` for driving DUT stimulus. The `#1step` sample time should almost always be used. The `#0` drive time should NEVER be used.

9.2 #1step Sampling

The best time to sample DUT outputs is at the end of the cycle, just before the next active clock edge changes the outputs of registered logic. The `#1step` input sample time specified in a clocking block elegantly accomplishes this goal.

In Verilog testbenches that did not have the `#1step` sample time, I used to wait for almost a full cycle and then sample the signal one time unit before the next active clock edge, using `#(`CYCLE-1)`. This worked fine unless the `CYCLE` delay was relatively short, such as a 2ns `CYCLE` delay, in which case `CYCLE-1` would be half of the `CYCLE`. For faster clock cycles, I would have to sample using either `#(`CYCLE-0.1)` or perhaps even `#(`CYCLE-0.01)`. The `#1step` gives the smallest resolution delay before the next active clock edge so engineers don't have to worry about relative clock speeds. With this added observation, this agrees with Bromley & Johnston Guideline #4.

It has been argued by some engineers that the best place to sample the DUT outputs is one setup-time delay before the active clock edge, assuming that the propagation of signals need to settle and be ready before the setup time requirement of the clocked logic. Although it is true that actual signals must be stable for the duration of the setup time before the active clock edge, functional simulation is not the place to prove that this requirement is being met. Functional and gate-sims with backannotated timing delays should be used to prove that the design is functionally correct and Static Timing Analysis Tools (STA) should be used to prove that all timing, including setup and hold times, are being met. Verification engineers should not be required to periodically ensure that maximum setup times are specified in the testbench.

9.3 #0 Drive Times

Driving stimulus to the DUT at `#0` after the active clock edge is possibly the worst place to drive stimulus. Unfortunately this is the default stimulus drive time of a `clocking` block. In real hardware, nobody tries to change the inputs of clocked logic exactly on the active clock edge and yet that is the precise behavior of the default `#0` drive time.

With few exceptions, in real hardware, changing the inputs exactly on the active clock edge will violate setup times and/or hold times and could frequently cause metastable values to be generated at the output of the clocked logic.

Engineers should set the drive time of a **clocking** block to be 20% of the clock cycle, allowing 80% of the clock cycle for input combinational delays to the DUT. If more input combinational settling is required, the **clocking** block could be set to 10% of the clock cycle to allow 90% of the cycle of input settling. Figure 9 shows that DUT combinational inputs could require most of the clock cycle to process testbench stimulus.

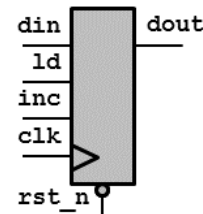
This agrees with Bromley & Johnston Guideline #5, noting that the delay should generally be 10%-20% of the clock cycle to allow DUT input combinational settling time.

9.4 Example UVM clocking block

To better understand some of the concepts of using **clocking** blocks, consider a simple program counter with asynchronous low-true reset, along with synchronous load and increment control signals as shown Example 6.

```
module pcnt (
    output logic [15:0] dout,
    input      [15:0] din,
    input      ld, inc, clk, rst_n);

    always_ff @(posedge clk, negedge rst_n)
        if (!rst_n) dout <= '0;
        else if (ld) dout <= din;
        else if (inc) dout <= dout + 1;
endmodule
```



Example 6 - Program counter DUT code

The **CYCLE** definition and clocking block drive time (**Tdrive**) definitions are kept in the file **CYCLE.sv**

```
`ifndef CYCLE
`define CYCLE 10
`endif
`ifndef Tdrive
`define Tdrive #(0.2*`CYCLE)
`endif
`timescale 1ns/1ns
```

Example 7 - CYCLE.sv file

The **interface** with **clocking** block used in the corresponding UVM testbench is shown in Example 8.

```
`include "CYCLE.sv"

interface dut_if (input clk);
    logic [15:0] dout;
    logic [15:0] din;
    logic      ld, inc, rst_n;
```

```

clocking cb1 @(posedge clk);
default input #1step output `Tdrive;
input  dout;
output din;
output ld, inc, rst_n;
endclocking
endinterface

```

Example 8 - DUT interface with clocking block

The next section shows how the UVM testbench driver is setup to drive stimulus to this design.

9.5 Stimulus using clocking drives

Bromley & Johnston Guideline #1 states:

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

Although this is generally a good guideline, there is one very important exception to this guideline that I do in all of my testbenches, and that exception occurs at time-0. In my UVM driver components, I always include an `initialize()` task that makes direct, non-clocking block signal assignments at time-0, and that `initialize()` task is called at the beginning of the `run_phase()` and then a `forever` loop executes a `drive_tr()` method or methods, all of which exclusively make assignments to the clockvars (signals with `clocking` block timing) after time-0. This allows my testbenches to initialize all signals with fixed or random values at time zero and then take advantage of `clocking` block controlled assignments to the same signals throughout the rest of the simulation.

The `tb_driver` in Example 9 includes the `initialize()` task and shows that `initialize()` is called just before entering the `forever` loop. For informational purposes, the `virtual dut_if` is set by the `tb_agent` (not shown) for this example.

```

class tb_driver extends uvm_driver #(transl);
  `uvm_component_utils(tb_driver)

  virtual dut_if vif;

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    transl tr;
    initialize();
    forever begin
      seq_item_port.get_next_item(tr);
      drive_tr(tr);
      seq_item_port.item_done();
    end
  endtask

  task initialize(); // @0 - Does not use clocking block
    vif.rst_n <= '0;
    vif.ld    <= '1;
    vif.inc   <= '1;
    vif.din   <= '1;
  endtask

```

```

endtask

task drive_tr (trans1 tr);
  @vif.cb1;
  vif.cb1.din    <= tr.din;
  vif.cb1.ld     <= tr.ld;
  vif.cb1.inc    <= tr.inc;
  vif.cb1.rst_n <= tr.rst_n;
endtask
endclass

```

Example 9 - tb_driver with initialize() task (no clocking block timing) and drive_tr() task (uses clocking block timing)

In his testbench book [6], Janick Bergeron similarly states that stimulus should not be assigned at time-0, but again, I have found it useful to make time-0 stimulus assignments using nonblocking assignments. In a personal conversation with Janick regarding this exception, Janick somewhat-accurately stated that using nonblocking assignments at time-0 did not violate his guideline as a nonblocking assignment executes at a later stage of the time-0 event regions.

Note that the `drive_tr()` task of Example 9 uses notations similar to: `vif.cb1.din <= tr.din`

Bromley & Johnston Guideline #3 states:

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

The added clarification is that the clocking drive operator (`<=`) is required whenever driving a clockvar (a signal that includes the `clocking` block name) and using the simple blocking assignment operator (`=`) is illegal. The SystemVerilog compiler will enforce Bromley & Johnston Guideline #3.

9.6 Why drive signals at time-0?

One frequently asked question is, why even drive DUT input signals at time-0? Why not allow DUT inputs to remain uninitialized at `x` at time-0 and then use clocking drives after the first testbench active clock edge?

It should be noted that many successful testbenches never drive DUT input signals at time-0 and these testbenches work just fine.

As noted in Section 2, time-0 is a tricky place in Verilog and SystemVerilog simulations. Input signals that are allowed to be X at time-0 have the potential to cause a pre- and post-synthesis simulation mismatches [4]. Any uninitialized input signal tested by a procedural `if`-statement will fail the `if`-test and always take the `else` branch. Any uninitialized input signal tested by a `casex` statement will always execute the first `casex`-item-statement. These are well known X-optimism examples, and have caused companies to have costly re-spins in their ASIC designs.

At time-0, my `initialize()` task is frequently written to reset the device, while simultaneously setting inputs to either all 1's or random values, and to set load-control input signals to attempt to load values into my DUT. I do this to ensure that reset properly clears the required register values and has priority over other loading control signals at time-0.

Another advantage to doing the time-0 assignments becomes visible in the waveform display. I generally do not like to see "red" signals at time-0, except for uninitialized and non-reset outputs. When I see "red" at time-0, I quickly analyze the red signals to make sure that their values are indeed unknown at time-0. I do not want to waste time analyzing uninitialized DUT inputs.

9.7 Asynchronous control inputs

Unlocked reference models and prediction functions typically take inputs sampled on clock edges to predict what the actual DUT output values should be. Typically inputs sampled on the active clock edge, such as the `posedge clk` are the only values required to predict the correct outputs. The exception to this rule is asynchronous control signals.

Consider the example of the asynchronous reset signal. Figure 11 shows four asynchronous reset scenarios that might have to be considered when predicting the DUT output.

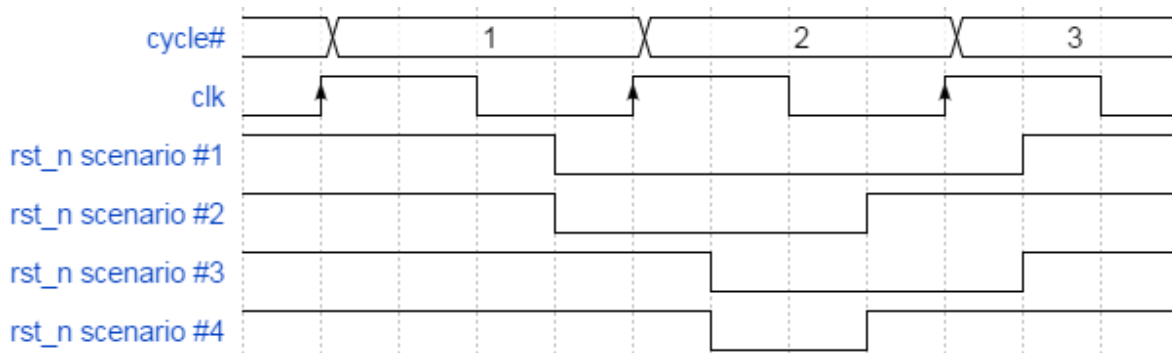


Figure 11 - Four asynchronous reset signal scenarios

- In the first asynchronous reset scenario, the predicted DUT outputs would need to be reset in cycles 1-3.
- In the second asynchronous reset scenario, the predicted DUT outputs would need to be reset in cycles 1-2.
- In the third asynchronous reset scenario, the predicted DUT outputs would need to be reset in cycles 2-3.
- And in the fourth asynchronous reset scenario, the predicted DUT outputs would only need to be reset in cycle 2.

In the first three asynchronous reset scenarios, it is clear that the reset signal needs to be sampled both at the beginning and at the end of the cycle since a reset at any time during the cycle should cause the DUT outputs to be reset.

If the reset signal is active when the inputs are sampled on the active clock edge, the predicted outputs can be reliably calculated. If reset is not active on the active clock edge, the predicted outputs cannot be guaranteed to not be reset later in the same cycle, which is why a reset signal that was not active when the inputs were sampled, needs to be re-sampled on the next active clock edge to see if it has been asserted during the cycle.

Example 10 shows part of the `sample_dut` task that is called by the `tb_monitor` (the full `tb_monitor` code is shown in Example 11). This UVM testbench is setup so that the `sample_dut` task is always synchronized to the `posedge clk`, so when called it first samples all of the inputs, including the asynchronous `rst_n` input, then re-synchronizes to the next clocking block sample signal from the virtual interface, `@vif.cb1`, which in this example re-synchronizes to the `posedge clk`, then samples the outputs `#1step` before that clocking block sample signal and then re-samples the asynchronous `rst_n` to see if it is low-true asserted on this `posedge clk` edge, and if asserted, the `rst_n` signal that will eventually be passed to the testbench predictor is set to 0, otherwise it keeps

its previous value, which might have been low-true asserted at the beginning of the cycle.

```
task sample_dut (output trans1 tr);
    trans1 t = trans1::type_id::create("t");
    //-----
    // Sample DUT synchronous inputs on posedge clk.

    ...

    @vif.cb1;
    if (!vif.rst_n) t.rst_n = '0; // async reset
    t.dout = vif.cb1.dout;
    //-----
    tr = t;
endtask
```

Example 10 - sample_dut task checks async reset at beginning and end of the cycle

Note that the `sample_dut` task uses:

```
@vif.cb1 ... t.dout = vif.cb1.dout
```

instead of:

```
@(posedge clk) ... t.dout = vif.cb1.dout
```

which agrees with Bromley & Johnston Guideline #2. `@cb1.vif` allows outputs to be sampled **#1step** before the `posedge clk`, while the `@(posedge clk)` gives race-condition results and appears to cause at least two simulators to sample the output one clock cycle earlier (the `vif.cb1.dout` sampling does not appear to recognize the current clock edge and appears to sample one clock edge earlier - this is just an observation and may not be consistent between current simulators or consistent with the future behavior of simulators).

The `tb_monitor` code of Example 11 shows the full UVM monitor example including the full `sample_dut()` code.

```
class tb_monitor extends uvm_monitor;
    `uvm_component_utils(tb_monitor)

    virtual dut_if vif;

    uvm_analysis_port #(trans1) aport;

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        aport = new("aport", this); // build the analysis port
    endfunction

    task run_phase(uvm_phase phase);
        trans1 tr;
        tr = trans1::type_id::create("tr");
        //-----
        forever begin
```

```

        sample_dut(tr);
        apert.write(tr);
    end
endtask

//-----
// sample_dut assumed to be synced to posedge clk
// except for first sample at time=0
//-----
task sample_dut (output trans1 tr);
    trans1 t = trans1::type_id::create("t");
    //-----
    // Sample DUT synchronous inputs on posedge clk.
    // DUT inputs should have been valid for most
    // of the previous clock cycle
    //-----
    t.din    = vif.din;
    t.ld     = vif.ld;
    t.inc    = vif.inc;
    t.rst_n  = vif.rst_n;
    //-----
    // Wait for posedge clk and sample outputs #1step before.
    // Also re-sample and check async control input signals
    //-----
    @vif.cb1;
    if (!vif.rst_n) t.rst_n = '0; // async reset
    t.dout = vif.cb1.dout;
    //-----
    tr = t;
endtask
endclass

```

Example 11 - tb_monitor checks async reset at beginning and end of the cycle

This technique is generally good enough for testing purposes because the verification engineer typically does not generate sub-cycle asynchronous control pulses when generating stimulus.

If there is the possibility of generating sub-cycle asynchronous control pulses either from the stimulus source or from another sub-block connected to this DUT block, then a **sticky-bit** technique will be required to capture the asynchronous control signal activity to pass to the testbench predictor. An example of this scenario is **rst_n scenario #4** as shown in Figure 11, and again, isolated as shown in Figure 12.

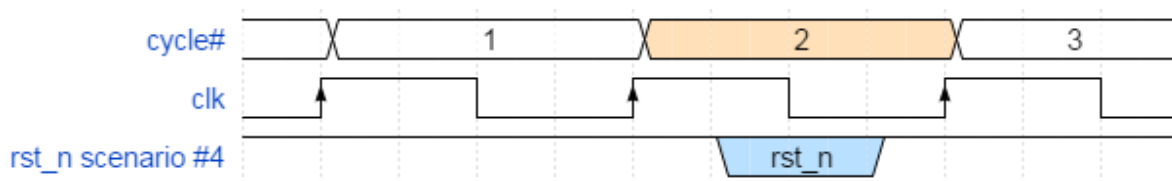


Figure 12 - Asynchronous mid-cycle, sub-cycle reset pulse<insert common code example here>

As shown in Figure 12, the reset pulse cannot be detected on either the rising edge of cycle #2 or on the rising edge of cycle #3. It will be necessary to capture and hold the reset condition of cycle #2 and sample that condition **#1step** before cycle #3.

The **sticky-bit** code can be easily added to the `dut_if.sv` file as shown in Example 12. A simple **always** block captures any active (**negedge**) transition on the `rst_n` signal and assigns **0** to the **sticky reset_n** signal that is used in the `tb_monitor` component shown in Example 13. At the next **posedge clk**, either the `reset_n` signal is still assigned to **0** (if `rst_n` is still active low) or is set to **1** to clear the active `reset_n` condition (if `rst_n` was deasserted before the end of the cycle). The only other DUT interface requirements to make this technique work are to declare the **sticky logic reset_n** signal in the declarations portion of the interface, and to add the `reset_n` signal as an **input** in the **clocking** block, to allow the `tb_monitor` to sample the signal `#1step` before the next **posedge clk**, which is used to deactivate the **sticky-bit** by setting it to a **1**.

```
`include "CYCLE.sv"

interface dut_if (input clk);
    logic [15:0] dout;
    logic [15:0] din;
    logic        ld, inc, rst_n;
    logic        reset_n;

    //-----
    // Sticky reset_n signal to capture short rst_n pulses
    //-----
    always_ff @(posedge clk, negedge rst_n)
        if (!rst_n) reset_n <= '0;
        else        reset_n <= '1;

    clocking cb1 @(posedge clk);
        default input #1step output `Tdrive;
        input  dout;
        output din;
        output ld, inc, rst_n;
        input  reset_n;
    endclocking
endinterface
```

Example 12 - DUT interface with sticky-bit code to save reset short-pulse reset condition

The `tb_monitor` also needs to be slightly modified to sample the `vif.cb1.reset_n` signal and assign those values as appropriate to the `t.rst_n` signal in the transaction before writing the transaction to the analysis port. The modified `tb_monitor` code is shown in Example 13.

```
class tb_monitor extends uvm_monitor;
    `uvm_component_utils(tb_monitor)

    ...

    task run_phase(uvm_phase phase);
        trans1 tr;
        tr = trans1::type_id::create("tr");
        //-----
        forever begin
            sample_dut(tr);
            aprot.write(tr);
        end
    endtask
endclass
```



```

task sample_dut (output trans1 tr);
    trans1 t = trans1::type_id::create("t");
    t.din      = vif.din;
    t.ld       = vif.ld;
    t.inc      = vif.inc;
    t.rst_n    = vif.rst_n;
    //-----
    // ...
    // Sample the sticky-bit reset_n to update rst_n if needed
    //-----
    @vif.cb1;
    if (!vif.cb1.reset_n) t.rst_n = '0; // async reset
    t.dout = vif.cb1.dout;
    //-----
    tr = t;
endtask
endclass

```

Example 13 - tb_monitor modified to test the sticky-bit reset_n version of the rst_n asynchronous reset

Note that the `sample_dut()` task of Example 13 uses notations similar to:
`t.dout = vif.cb1.dout;`

The proper coding for sampling DUT outputs using a `clocking` block requires the use of the clocking sample operator (`=`) and using a nonblocking assignment operator (`<=`) for clocking sample operations is illegal. The SystemVerilog compiler will catch this mistake.

9.8 Interface modports and testbenches

Bromley & Johnston Guidelines #9 and #11 recommend using `modport` versions of the clocking signals. These guidelines are fine and even add a small amount of additional checking to the signals being driven and sampled, but I generally find the use of `modports` in testbench `interfaces` to be a level of complexity that is generally not needed. Verification engineers are encouraged to use or not use `modports` at their discretion.

10. Death to the SystemVerilog program!

SystemVerilog-2005 added a new testbench construct called a `program`. The primary reason for adding the `program` construct to SystemVerilog was to help avoid stimulus-DUT race conditions when stimulus was driven on the active clock edge, which should never be done!

The idea was that the RTL code would be captured within modules while the testbench stimulus generation would be captured within programs.

The perceived benefit of this approach was that if stimulus was driven on the active clock edge, an active clock edge would first execute the RTL code in the active region of the event schedule (shown in the upper half of Figure 13) and allow the RTL to settle to a semi-final value before new stimulus was sent to the DUT. This means that the RTL would sample all inputs that had setup on the registers before the testbench changed those inputs for the next cycle.

After the RTL had settled to a semi-final state in the active region, testbench `program` code would then drive new stimulus to the DUT during the reactive region of the same time slot (shown in the lower half of Figure 13). After the `program` code had calculated the appropriate stimulus to send to the DUT, those stimulus values would then be sent back into the DUT and any DUT input

combinational logic would recalculate the inputs to the DUT registers and these values would remain on the DUT inputs until the next active clock edge.

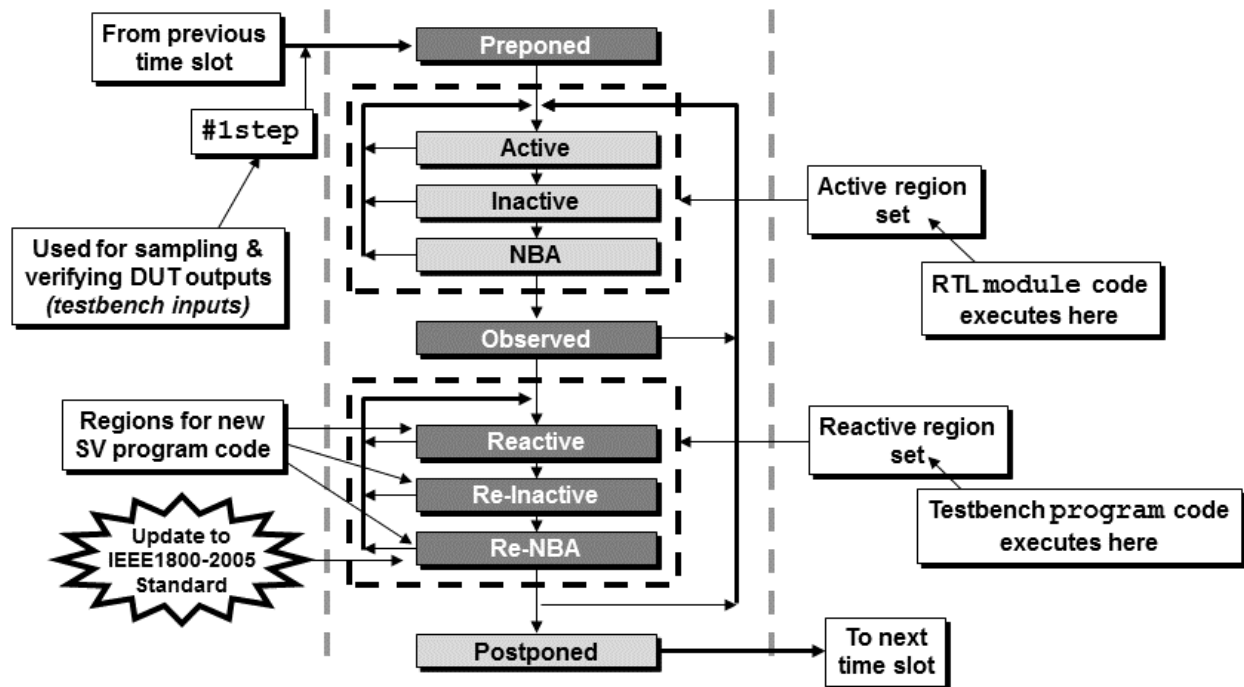


Figure 13 - SystemVerilog module and program event scheduling

If a verification engineer drives stimulus on the active clock edge, the **program** testbench scheduling could prove useful to avoid race conditions where the RTL might partially calculate a final value, the stimulus arrives before the RTL is done calculating values, and the stimulus changes some of the inputs that have not yet been registered. So the **program** essentially made it possible to drive stimulus on the active clock edge and avoid RTL-stimulus race conditions, but as has been previously discussed, stimulus should NOT be driven on the active clock edge, and hence there is no need for the use of the **program** block.

As long as the verification engineer does not drive stimulus on the active clock edge, there is no RTL-stimulus race condition and a **program** is not needed.

The SystemVerilog Standard also introduced a number of confusing coding restrictions associated with **program** usage, such as:

A **program** can only use **initial** procedures but not **always** procedures.

A **program** can hierarchically reference **module** signals but a **module** cannot hierarchically reference **program** signals.

A **program** can call **module** tasks and functions while a **module** cannot call **program** tasks and functions.

Simulators have not always (and still may not) consistently check and execute **program** code.

The SystemVerilog **program** statement should just die and never be used in your code!

10.1 Cliff's confession

Despite personal reservations, I voted to include programs into the SV2015 standard. In a separate conference call with committee members who were advocating the inclusion of programs to SystemVerilog, I described my technique of avoiding RTL-stimulus race conditions by explaining how I never drive stimulus on the active clock edge. Those advocating inclusion of the **program** acknowledged that I had a good technique but that if I allowed programs to be added to SystemVerilog, it would be easier for engineers to drive race-free stimulus without being required to understand my technique. Based on that argument, I voted in favor of adding programs, a vote that I now regret.

If I could remove programs from the SystemVerilog language I would, but due to backward compatible coding reasons, I cannot remove them.

All I can do is give my strongest recommendation to verification engineers:

Guideline: Never use or quit using SystemVerilog programs!

11. Conclusions

Time-0 is a tricky place in Verilog and SystemVerilog simulations. To avoid time-0 race conditions, create an **initialize()** task and assign all inputs at time-0 using nonblocking assignments (see Section 2.)

There are three timing values that need to be properly considered to generate robust, race-free testbenches: (1) When to drive stimulus, (2) When to sample DUT inputs, (3) When to sample DUT outputs.

This paper has shown a robust technique of driving stimulus using time budgeting to ensure that stimulus is driven safely after the active clock edge. A time budget of waiting 20% of the clock cycle after the active clock edge was recommended, and that value was used in a **clocking** block. Other values could be used based on the combinational logic input delay of the DUT.

DUT inputs should be sampled on the active clock edge because that is where the DUT will sample those same inputs. The exception is asynchronous control signals that must frequently be re-sampled at the end of the cycle, and if asynchronous control signals could be sub-cycle pulses, a **sticky-bit** technique may be employed to capture those glitching asynchronous control signals.

Outputs should be sampled at the last possible moment before the next active clock edge. This is accomplished by using the **#1step** sample time within a clocking block. This was shown in Section 9.2

Finally, the well-intentioned SystemVerilog **program** enhancement only offers value if you are trying to apply stimulus on the active clock edge, which you should never do! The **program** has a number of annoying restrictions when interacting with a **module** and just adds confusion to how SystemVerilog events are scheduled. The SystemVerilog **program** should never be used! The SystemVerilog **program** should just die!

12. References

- [1] Anna Ekstrandh, Wayne Bell, "Evolvable Makefiles and Scripts for Synthesis," *SNUG (Synopsys Users Group) 1997 Proceedings*, February 1997.
- [2] Clifford E. Cummings, "OVM/UVM Scoreboards - Fundamental Architectures," SNUG (Synopsys Users Group) 2013 (Silicon Valley, CA). Also available at www.sunburst-design.com/papers

- [3] Clifford E. Cummings, "Verilog Nonblocking Assignments With Delays, Myths & Mysteries," SNUG (Synopsys Users Group) 2002 (Boston, MA). Also available at www.sunburst-design.com/papers
- [4] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," SNUG (Synopsys Users Group) 1999 (San Jose, CA).
Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers
- [5] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2012
- [6] Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models, 2nd Edition, Springer Science+Business Media, Inc., 2003. ISBN: 1-4020-7401-8
- [7] Jonathan Bromley and Keven Johnston, "Taming Testbench Timing: Time's Up for Clocking Block Confusions," SNUG (Synopsys Users Group) 2012 (Austin, TX). Also available at www.verilab.com/resources/papers-and-presentations/#snug2012clock