Sub-cycle Functional Timing Verification Using SystemVerilog Assertions

Anders Nordstrom

Verilab Inc.
Ottawa, Canada

andersn@verilab.com
www.verilab.com

ABSTRACT

This paper describes a novel, more complete approach to functional verification of sub-cycle timing using SystemVerilog assertions in an OVM verification environment. This approach found many bugs otherwise missed in OVM-only simulations.
This functional sub-cycle timing behaviour includes maintaining fixed delays and phase relationships between inputs and outputs and ensuring there are no glitches on clocks or delayed signals. SystemVerilog assertions are evaluated on successive occurrences of an event or timing expression. This presents a challenge for sub-cycle timing verification, where there is no obvious reference clock suitable for triggering the assertions. Assertions sample their expressions in the preponed region of the simulation time slot, but the requirements called for sampling both before and after each triggering point. Examples of assertions showing how to overcome this and many other issues will be shown along with recommendations on how to write assertions for functional timing verification.

Copyright © 2013 Verilab & SNUG

Table of Contents

Table of Figures

1. Introduction

Simulation using SystemVerilog along with a class library such as OVM or UVM is commonly used to verify the functionality of a chip, and Static Timing Analysis (STA) is used to verify implementation timing. However, when timing behaviour, such as varying signal delay and phase relationships, is part of the RTL functionality to verify, this approach is not so straightforward. Verifying timing where delays vary throughout the simulation and are modeled in fractions of clock cycles is different than the type of timing verified in gate-level simulation or STA. This is not easily verified in RTL simulations since they do not typically model delays. The typical behaviour of RTL where, if data is stable and has the correct value just before the clock edge, the data will be captured and propagated is not sufficient for this case and needs to be extended.

Any solution needs to address two aspects of the verification; firstly, how to introduce and vary the delays and secondly, how to verify that signal timing is correct in the presence of delays. The timing verification aspect is covered here. The control aspect is beyond the scope of this paper and is described in detail in [5].

SystemVerilog provides several mechanisms for verifying timing such as Verilog timing checks (e.g. **`$setup()`**), simulation timestamps using **`$time`** and SystemVerilog Assertions (SVA). After investigating each option, assertions were chosen. While assertions provide the most efficient solution to the functional timing verification problem, using them for sub-cycle verification presents additional issues including:

- Integration difficulties
- Simulation behaviours
- Jitter on signals
- Lack of clock

While addressing these issues a set of recommendations, shown throughout the paper, were developed. The verification approach using assertions proved to be successful and found many bugs missed in regression simulation without assertions.

This paper describes and provides examples of the functional sub-cycle timing verification of a DDR3 device using SystemVerilog assertions, although they are applicable to any system where variable functional timing needs to be verified.

2. The System Under Verification

The system under verification is a DDR3 Dual In-line Memory Module (DIMM). It consists of nine byte-lanes of memory and the DUT. It is a DDR3[1] Register Phase Locked Loop (RPLL)[2] like device which re-times and re-drives all required DDR3 signals.

The DIMM is a long and narrow module and the track length, and hence delay, for each byte-lane is different. The DUT needs to insert delay in order for the DDR3 timing requirements to be met. For example, the delay on **`DQ0`** and **`DQ31`** in Figure 1 are significantly different so the DUT must sample and drive them at different times. In addition, delays inside the DUT vary with process, voltage and temperature (PVT) and these also needs to be compensated for. The physical interface (PHY) of the DUT contains several Delay Locked Loops (DLLs), Phase Locked Loops (PLLs) and control algorithms to control the launch and sample times of signals on the DDR3 interface within a tight range.

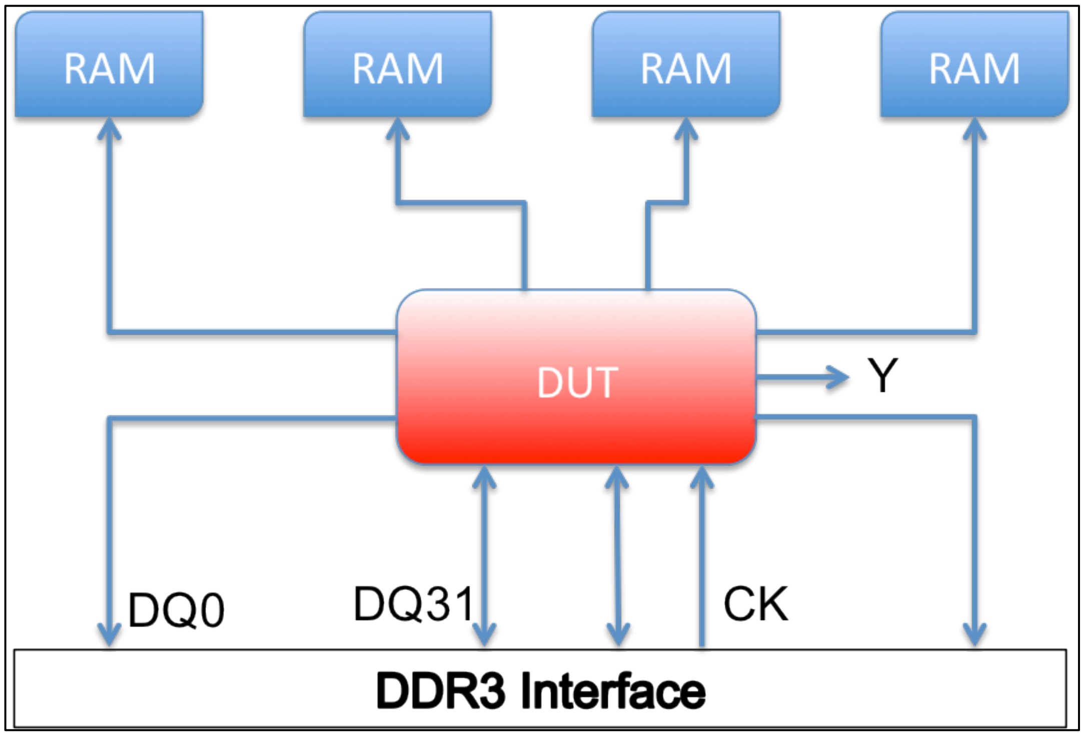


Figure 1: DIMM with Device Under Test

3. Verification Challenges

In addition to generating DDR3 transactions and checking responses in order to verify the data and control path functionality of the DUT, the verification environment has to insert delays to model board, pad and loop control delays in DLLs. These delays are varied to model PVT variations, which the DLL control algorithms are to compensate for. Signal B in Figure 2 has a programmed phase delay with respect to signal A. The delay on A varies with PVT but the phase delay must be kept constant. The DLL has control algorithms that use a reference PVT delay to compensate for the delay variation. The delay variations are done using OVM analysis port connections and are described in [5].

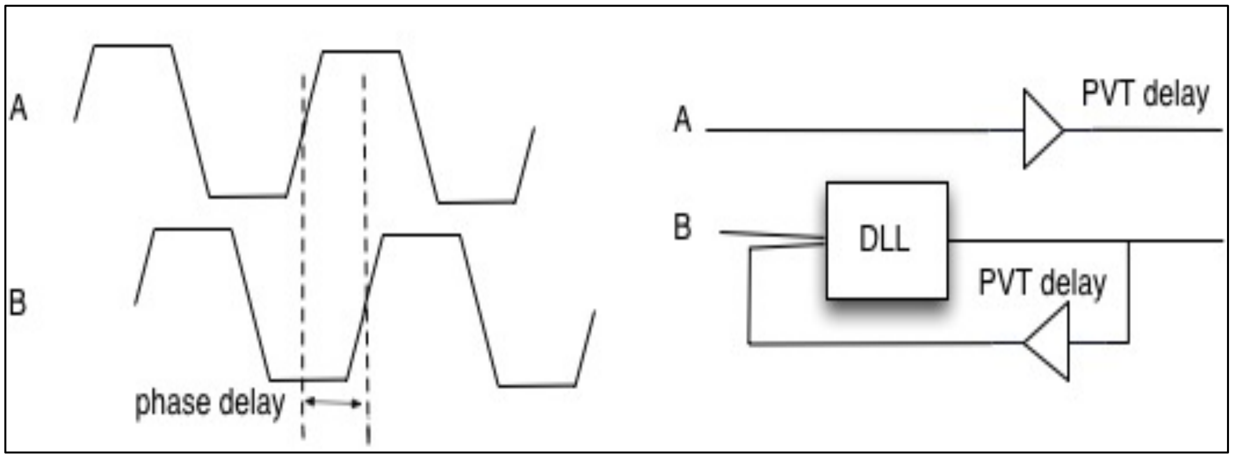


Figure 2: DLL with feedback delay

The DUT has multiple clock domains and the DDR3 interface operates at DDR3-800, DDR3-1066, DDR3-1333 and DDR3-1600 speeds. The required delay or phase relationships depend on frequency and are programmed in DUT registers. The phase relationships vary from 0 to a full clock period, hence the timing checks cannot be performed at a predetermined reference clock. The main functional timing verification requirement is to check that all programmed phase delays are correctly maintained by the DLLs. Checks that need to be implemented includes verifying:

- Input clock to output clock phase is held at the programmed value
- Control signals are driven at the correct time with respect to input clock
- No glitches on internally generated or delayed clocks
- Control signals are asserted long enough despite jitter introduced by DLLs

4. Verification Approaches and Adopted Solution

Three approaches for verifying the sub-cycle functional timing behaviour of the DUT were considered.

- Extend the existing OVM environment
- Use Verilog timing checks
- Use SystemVerilog assertions

Extend OVM Environment

The first solution considered was to extend the already existing OVM test environment. It generates sequences of DDR3 transactions and checks that correct data is returned. Time stamps on transactions are captured by monitors so transaction timing can be verified. Capturing when individual signals change may be done at the interface level and then passed to the monitors for checking. However, verifying signal behaviour over multiple interfaces is easier done closer to the DUT than in the verification environment so further options needed to be investigated.

Verilog Timing Checks

SystemVerilog has built in timing checks such as `$setup()`, `$hold()` and `$width()`. They are normally used to check gate-level implementation timing in simulation. They may also be used at the RTL level, for example, to check that there are no glitches on a generated clock. An example of doing this using `$width()` is shown in Figure 3.

```
signal_width_check c_glitch (phy_clk);

module signal_width_check (input wire signal2check);

  specify
    specparam min_width = 1250;
    $width(posedge signal2check, min_width);
    $width(negedge signal2check, min_width);
  endspecify

endmodule // signal_width_check
```

Figure 3: Verilog timing checks

Timing checks are a workable solution but have limitations. They must be placed inside a specify block in a module which may be bound to a design module. The **`min_width`** variable in Figure 3 must be a **`specparam`** which makes it harder to change its value than if a variable in an assertion was used. As well, there is no means of changing the output message, shown in Figure 4, from the timing check.

```
Warning!  Timing violation
          $width( negedge signal2check:73511332340 FS,  : 73512580250 FS,
1260 : 1260 PS );
           File: /home/andersn/phy_assert.sv, line = 576
          Scope: harness.top.core.ddr_phy.phy_assert.c_glitch
           Time: 73512580250 FS
```

Figure 4: Timing check error message

SystemVerilog Assertions

Assertions provide much more flexibility. They may use a customizable **`OVM_ERROR`** macro, making the assertion error message, shown in Figure 5, easier to understand and debug than the standard message from the timing check.

```
OVM_ERROR @ 73362.640 ns: reporter [SVA phy_assert] h_clk high or low period
too short, got (1.250 ns) expected (1.260 ns)
```

Figure 5: Assertion error message

Another requirement which is more easily accomplished with assertions than with timing checks is controlling when to disable checks; for example during reset or device programming. Further, the verification team had experience using assertions but not using Verilog timing checks so adoption was easier. Timing checks may be used in some cases but the limitations noted above lead to assertions being chosen for the functional timing checks.

5. Assertions in an OVM Environment

Adding assertions to an existing OVM environment requires consideration of:

- Where to put the assertions
- How to access out-of-module signals in assertions
- How to exclude assertions
- How to OVM testbench configuration information in assertions

Several solutions to these questions were considered and tried. They are described in this section along with recommendations and descriptions of the approaches used.

Location of Assertions

Where is the best location to add assertions? The assertions verify DUT I/O signal relationships as well as internal signal behaviours. Adding them to RTL modules is possible for assertions verifying internal signals of the module but it is not the best general solution. Protocol assertions are often added inside existing SystemVerilog interfaces. Interfaces provide good encapsulation when checks only involve signals inside the interface, but this was not the case here. Instead, the assertions were placed inside dedicated interfaces in separate files that were bound to RTL modules using the SystemVerilog **bind** statement. Verilog modules can be bound to modules only, but SystemVerilog interfaces can be bound both to modules and interfaces. Similarly, interfaces can be instantiated in both interfaces and module but modules can only be instantiated in modules as illustrated in Figure 6.

	Bind to	**Instantiate in**
module	module	module
interface	module / interface	module / interface

Figure 6: Module to interface comparison

The assertions could reside in interfaces or in modules but interfaces were selected due to the greater flexibility when reusing the assertions on future projects.

Recommendation 1: Put assertions inside interfaces in separate files.

Accessing signals in assertions

The use of **bind** is well described in [6] but the main example is using **(.*)** to connect signals whose name match the ones in the module bound to. In several cases the assertions need signals outside of the module the assertion interface was bound to. These signals can either be added to the **bind** statement port map and connected using the hierarchical name or they can be cross-module references to signals used inside the assertion interface.
In the example below the signal **c_hclk** is not present in the module **ddr_phy** but in a lower level module **ckgen** instantiated in **ddr_phy**. By adding **c_hclk** to the bind statement port map, the signal becomes accessible to assertions inside the interface by the name **c_hclk**. The hierarchical connection is made in the **bind** statement itself as shown in Figure 7.

```
bind ddr_phy ddr_assert my_assert (.c_hclk (ckgen_inst.c_hclk), .*);

interface ddr_assert (input      c_hclk,
                      ....
                      input      rst_n);
```

Figure 7: Bind with hierarchical signal connection

This allows the **ddr_assert** interface to be bound to modules where **c_hclk** is connected to different signals without any changes to the assertion itself, only the connection in the bind statement needs to be changed.
In the example in Figure 8, the signals **c_reg_ckq_off** and **c_phy_ckdb_off** are used directly inside the interface by specifying their full hierarchical name. Hence, they are not being added to the bind statement.

```
`define OFFSET_REG harness.card_inst.dut_inst.inst_core.inst_phy.reg_inst

interface ddr_assert (input       c_hclk,
                      input [1:0] rc10,
                      input [7:0] ckq_off, ckdb_off);

always @(*)
      case (rc10[1:0])
        2'b00: begin      //DDR800
           ckq_off =      `OFFSET_REG.c_reg_ckq_off;
           ckdb_off =     `OFFSET_REG.c_phy_ckdb_off;
        ...
      endcase
...
endinterface
```

Figure 8: Cross module references inside assertion interface

Using cross-module references may seem convenient and make the bind statement less complex and easier to write but the assertions become less re-usable. All signals used in the assertion interface are not readily available in a waveform viewer since the signals are not declared in the interface, hence an extra step adding them is required for debugging.

Recommendation 2: Connect all signals explicitly in the bind statement, if they are not automatically connected by **`.*`** in the bind port map.

Excluding Assertions

Excluding all or a group of assertions is often necessary in some or all testcases. SystemVerilog has many options for this. Some, however lead to unwanted behaviour. In this context, excluding refers to the unconditional disabling of assertions whereas disabling assertions refers to the conditional disabling using a **`disable iff()`** statement in the assertion declaration.

Most simulators have a **"`-assert disable`"** compilation option. This option disables all assertions but may cause unexpected errors. Disabling all assertions in causes a pointer null dereference error for the code in Figure 9 because the **`$cast`** is not executed, and **`this.my_parent`** stays null. The assertion is disabled and doesn't fail, and the expected error message in the assertion is not printed.

```
assert($cast(this.my_parent, get_parent() ))
       else `DUT_ERROR(this, $sformatf(
             "Host Agent must be the parent of this component."));
```

Figure 9: Assert $cast error check

Similarly, the following call to randomize is not executed because all assertions are disabled.

```
assert(mytop.cfg.dimm_config.randomize(clock13));
```

Figure 10: Assert randomize error check

As we have seen in the examples above, using assertions for testbench error checking may have unexpected side effects. A better way to check for the randomization errors shown in Figure 10 is shown in Figure 11 . Some simulators allows for more flexibility through a **"`-assert disa-`**

ble_file=<file_name>" compilation option. However, the testbench needed to run on multiple simulators so simulator specific options were avoided.

```
if(!mytop.cfg.dimm_config.randomize(clock13))
   ovm_report_error("Randomization Failed");
```

Figure 11: if randomize error check

Recommendation 3: Use assertions to check DUT signals only, never for testbench error checks.

The assertions in an interface are bound to the DUT using a **bind** statement. Another method for excluding a set of assertions is using an **`ifdef** statement around the bind statement. By defining **USE_PHY_ASSERTIONS** on the command line or in individual testcases the assertions in the bound interface are included, otherwise they are excluded.

```
`ifdef USE_PHY_ASSERTIONS
   bind phy_dll phy_dll_assert my_phy_dll_assert (.*);
`endif
```

Figure 12: `ifdef around bind

Although using an **`ifdef** around the **bind** statement works, it can have side effects. When the assertion interface is not bound to a design module it becomes an unconnected top level interface and its inputs will have unknown values. This is not a problem for assertions using a clock as a trigger but when the clocking expression is a combinatorial expression as shown in Figure 13, an assertion not bound to the design may still fail.

```
always @(posedge sigo) sigo_posedge_time = $time;
always @(negedge sigo) sigo_negedge_time = $time;

wire signed [31:0] period_delta =
      ((sigo_negedge_time - sigo_posedge_time)<0) ?
        (sigo_posedge_time - sigo_negedge_time) :
        (sigo_negedge_time - sigo_posedge_time);

a_dll_sigo_glich: assert property (@(period_delta) disable iff(reset)
                                   (period_delta > sdll_width))
                  else ovm_report_error("SVA phy_assert",
  $psprintf("sdll %m sigo high or low period too short.
    Got %d Expected %d", $sampled(period_delta), sdll_width));
```

Figure 13: Assertion with combinatorial clocking expression

The clocking expression (**period_delta**) for the assertion **a_dll_sigo_glitch** has a glitch at time 0 which trigger the assertion and it fails with the error shown in Figure 14.

```
OVM_ERROR @ 0.000 ns: reporter [SVA phy_dll_assert] sdll phy_dll_assert sigo
high or low period too short. Got 0 Expected x
```

Figure 14: Unbound assertion error

A more robust solution is to put the **bind** statement and the interface in the same file and put the **`ifdef** around both, as shown in Figure 15 so they are included and excluded together.

```
`ifdef USE_PHY_ASSERTIONS
bind phy_dll phy_dll_assert my_phy_dll_assert (.*);

interface phy_dll_assert (input clk, input sigo, input [3:0] phy_rc10);
…
endinterface // phy_dll_assert
`endif
```

Figure 15: `ifdef around bind and interface

Recommendation 4: Use the same **`ifdef** around the **bind** statement and **interface** declaration

The SystemVerilog assertion control system tasks **$asserton()** and **$assertoff()** allows assertions bound to a specified module to be turned on or off during simulation. Including the code in Figure 16 in a testcase will turn off all assertions bound to **inst_phy** for that testcase.

```
initial begin
      $assertoff(0, `PHY_ASSERT_PATH.inst_phy );
      $display ("Phy assertions disabled for test foo ...");
...
      run_test();
   end
```

Figure 16: Assertion control using $asserton() task

Recommendation 5: Use **$assertoff()** to turn off assertions for specific testcases

Using OVM Configurations

The DDR3 interface operates at 4 speeds, corresponding to the frequencies 400MHz, 533MHz, 666MHz and 800MHz. There are configuration bits in the test environment that are randomized at build time to set the operating speed of the OVM driver component. The DDR3 interface in the DUT also needs to know what speed to operate at. A DUT register is written by a DDR3 RC10 control word [2] write as part of the initialization procedure. Since the functional timing is dependent on the speed, the assertions need to know the current speed or clock period as well. The interface containing the assertions is bound to a design module, and does not have a handle to the configuration in the class-based OVM environment so it can not access the speed configuration bits.
The assertion interface can determine the clock period (**sim_period)** by a direct reference to the **rc10** register in the DUT as shown in Figure 17.

```
bind phy ddr_assert my_assert(.rc10(reg_inst.rc10), .*);

interface ddr_assert(input [1:0] rc10);

time sim_period;
always @(*)
      case(rc10[1:0])
        2'b00:   sim_period = 2500ps;                       //DDR3-800
        2'b01:   sim_period = 1876ps;                       //DDR3-1066
        2'b10:   sim_period = 1501ps;                       //DDR3-1333
        2'b11:   sim_period = 1250ps;                       //DDR3-1600
      endcase
…
endinterface
```

Figure 17: Direct DUT register reference

The assertions will thus always use the same speed configuration as the DUT. However, if the programming of DUT registers has a bug and the `rc10` register is not updated when the testbench is initializing the DUT, the assertions may miss failures or report false failures. A better solution to ensure that the assertions and the test environment use the same timing setup is to have both use the same configuration information. The assertion interface does not have a handle to the OVM configuration class but the configuration class knows the hierarchical path to the assertion interface since it is static.
The simulation clock period (`sim_period`) is randomized in the timing configuration class (`timing_cfg`). Once randomized, the `sim_period` variable in the assertion interface is set in the `post_randomize()` function of the class as shown in Figure 18.

```
class timing_cfg extends ovm_transaction;
rand int sim_period;
…
constraint sim_period_cons {
         (speed == DDR_800)   -> sim_period == 2500;
         (speed == DDR_1066)  -> sim_period == 1876;
         (speed == DDR_1333)  -> sim_period == 1501;
         (speed == DDR_1600)  -> sim_period == 1250;
      }
…
function void post_randomize();
   harness.card_inst.top.inst_core.inst_phy.my_assert.sim_period =
   this.sim_period;
endfunction : post_randomize
…
endclass
```

Figure 18: Assertion variable set by configuration class

This ensures that the test environment and the assertion module use the same timing parameters.

Recommendation 6: When assertions depend on testbench configurations, set assertion parameters from the environment instead of accessing DUT registers.

6. Assertions for Functional Timing Verification

Writing assertions to verify functional timing of essentially asynchronous signals requires considerations not normally encountered when writing functional verification assertions. Sub-cycle timing refers to the fact that the delays between signals are fractions of the clock period and the signals change at any time with respect to clocks. Considerations include:

- Lack of reference clock
- Assertions sample signals in preponed region of time step
- Uneven simulator support
- Jitter on observed signals

The SystemVerilog LRM [3] states that concurrent assertions are based on clock semantics and use sampled values of variables. This provides a common semantic for the meaning of assertions and allows them to be used with different tools. This however, presents a challenge for sub-cycle timing verification, where there is no obvious reference clock suitable for triggering the assertions. A further difficulty arises because assertions use values sampled in the preponed region of the simulation time step but the assertion is evaluated in the observed region. If the signal used for triggering the assertion is the same one as being checked, the assertion will use the value before the assertion was triggered.

One solution is to use sequences as clocking expressions in assertions. Sequences ends in the observed region of the simulation time step so the values for variables in the assertion will be sampled in the observed region. Another option is to use immediate assertions in procedural code. Note that not all simulators sample values in the observed region of the time step when a sequence is used as a trigger so the method is not completely general.

7. Assertion Examples

The following assertion examples shows how a number of the verification requirements are implemented and how the considerations in section 6 are addressed.

One verification requirement was to check that "mux inputs to DLL's are low immediately following a mux select change". This could simply be written as shown in Figure 19.

```
a_mux_sel: assert property (@(posedge mux_sel)
                            (sigo0==0 && sigo1==0));
```

Figure 19: Concurrent assertion

This will however not meet the verification requirement since the values of **`sigo0`** and **`sigo1`** are sampled before the rising edge of **`mux_sel`**. One solution is to use immediate assertions. They are placed in procedural code so they execute after the rising edge of **`mux_sel`** as shown in Figure 20. In this case the values of **`sigo0`** and **`sigo1`** are checked after the rising edge of **`mux_sel`** which meets the verification requirement.

```
always @(posedge mux_sel)
  a_mux_sel: assert (sigo0==0 && sigo1==0);
```

Figure 20: Immediate assertion

Signal Jitter Impact

If a reference clock exists but the signal being checked has jitter, the assertions need to be written differently. Even an assertion as seemingly simple as checking "a high pulse on **`DQSEN`** must be 3 clock periods", illustrated by the waveform in Figure 21, becomes quite complex.

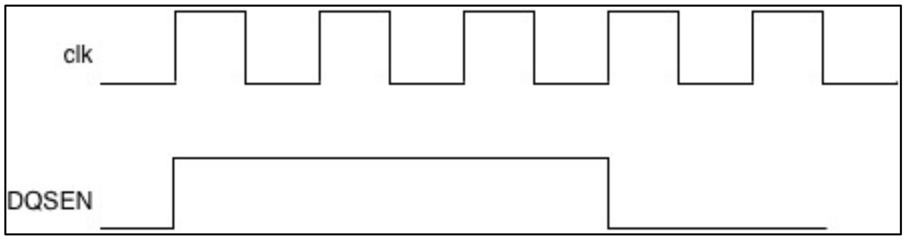


Figure 21: DQSEN timing requirement

The assertion in Figure 22 works in the case shown in the waveform above but it will not work if there is jitter on the DQSEN signal.

```
a_dqsen_length: assert property (@(posedge clk)
                                $rose(DQSEN) |-> DQSEN[*2]);
```

Figure 22: Clocked DQSEN assertion

Taking jitter into account, the check becomes "a high pulse on **DQSEN** must be 3 clock periods ±jitter". If the clock period is 2.5ns and the jitter is 100ps then **DQSEN** needs to be asserted for at least 7.4ns but not more than 7.6ns. The **a_dqsen_length** assertion in Figure 22 will incorrectly fail **DQSEN_short** and incorrectly pass **DQSEN_long** in Figure 23.

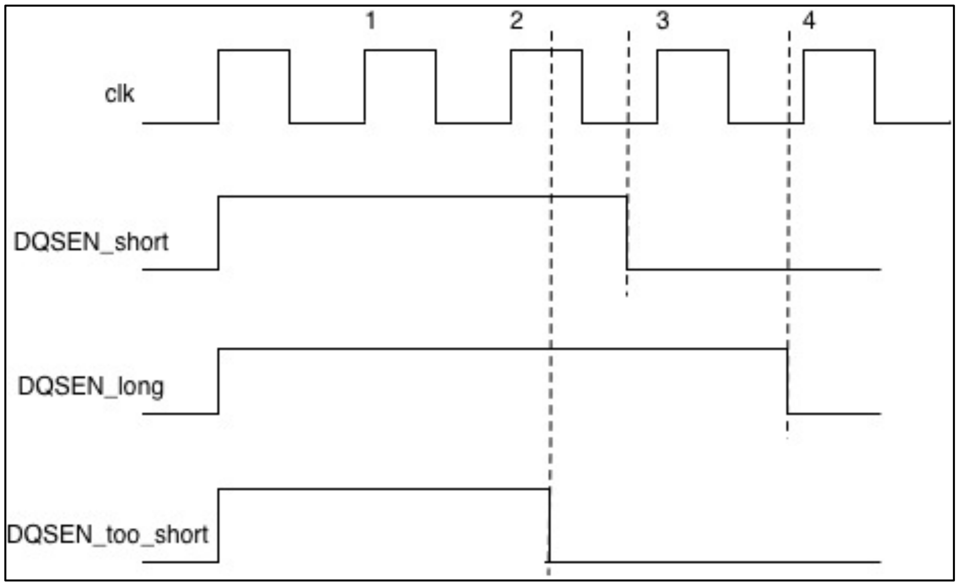


Figure 23: DQSEN with jitter

It becomes necessary to measure the actual time **DQSEN** is asserted by capturing simulation time at rising and falling edges of **DQSEN**. The updated assertion is shown Figure 24.

```
parameter dqsen_length = 7500ps;
parameter dqsen_jitter = 100ps;

always @(posedge DQSEN) dqsen_postime = $time;
always @(negedge DQSEN) dqsen_negtime = $time;

sequence dqsen_length_seq;
   @(posedge DQSEN) 1;
endsequence

a_dqsen_length:
      assert property (@(dqsen_length_seq) disable iff(!reset_b)
            (dqsen_postime - dqsen_negtime) >
            (dqsen_length - dqsen_jitter) &&
            (dqsen_postime - dqsen_negtime) <
            (dqsen_length + dqsen_jitter));
```

Figure 24: DQSEN assertion with jitter

A sequence is used as a clocking expression so that the assertion is evaluated after `dqsen_postime` is updated.

Recommendation 7: Measure signal transition times instead of relying on a reference clock if jitter is modeled.

Incorrect Error Messages

Concurrent assertions use signal values in the preponed region of the timestep and it may lead to incorrect and confusing error messages when an assertion fails. For example, the assertion `a_DBODT_delay_max` in Figure 25 checks that the delay from rising edge of `CLK` to a change in `DBODT` is less than `ck2db_max`.

```
always @ (posedge CLK)    db_start_time  = $time;
always @ (DBODT)          DBODT_delay    = $time - db_start_time;

a_DBODT_delay_max: assert property (@(DBODT_delay)
                                   DBODT_delay <= ck2db_max)
                   else ovm_report_error("ddr_assert",
                $psprintf("DBODT_delay (%t) is longer than ck2db_max (%t)",
                         DBODT_delay, ck2db_max));
```

Figure 25: Delay check assertion

In one instance, the assertion fails with the error message in Figure 26. At first, the time values in the message does not make sense but the waveform in Figure 27 shows that `DBODT_delay` changes from 1250ps to 1048ps on the rising edge of `DBODT`.

```
OVM_ERROR @ 74652.923 ns: reporter [ddr_assert] DBODT_delay (1.048 ns) is
longer than ck2db_max (1.109 ns)
```

Figure 26: Incorrect error message

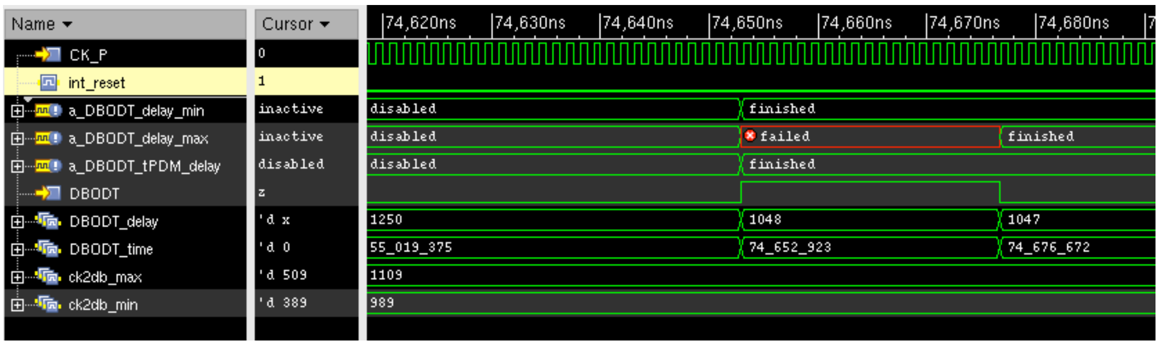


Figure 27: Delay check waveform

The assertion checks the value in the preponed region (1250ps) and hence fails, but the **`$psprintf()`** function use the value (1048ps) in the observed region where the assertion is executed. One solution is to use the **`$sampled()`** function in the **`$psprintf()`** call so that it displays the value that was sampled by the assertion as shown in Figure 28.

```
a_DBODT_delay_max: assert property (@(DBODT_delay)
                                  DBODT_delay <= ck2db_max)
                   else ovm_report_error("ddr_assert",$psprintf(
                   "DBODT_delay (%t) is longer than ck2db_max (%t)",
                   $sampled(DBODT_delay), ck2db_max));
```

Figure 28: Corrected delay check assertion

Recommendation 8: Use **`$sampled()`** to display the correct value in error messages when the value checked is the same as the assertion trigger.

Uneven Simulator Support

A central verification requirement of the DUT is to ensure that DLL outputs don't have glitches shorter than 3/8 of the clock period. Using multiple clocking events in a property and sampling **`$time`** to calculate the pulse width as shown in Figure 29 and in [4] appear to be a good solution.

```
property p_no_dll_glitch (start_event, end_event, min_delay);
time start_time;

@(start_event) (1, start_time = $time()) |=>
@(end_event) (($time – start_time) > min_delay);
endproperty
```

Figure 29: Multiple clocking event glitch check property

However, the property does not work reliably in all major simulators so an alternative solution is required. Both the multiple clocking events and the use of **`$time`** appear to be causing false failures. If the sampling of **`$time`** at signal edges is moved outside the property and only the comparison of sampled versus required time is done in the property, the simulator difference is avoided. The updated property is shown in Figure 30. Note: **`period_delta`** needs to be declared signed in order to handle both cases where either the posedge or negedge appear first.

```
always @(posedge sigo) posedge_time = $time;
always @(negedge sigo) negedge_time = $time;

wire signed [31:0] period_delta = ((negedge_time - posedge_time) < 0) ?
                                   (posedge_time - negedge_time) :
                                   (negedge_time - posedge_time);

a_no_dll_glich: assert property (@(period_delta)
                                 (period_delta > sdll_width))
                else ovm_report_error("SVA phy_sdll_assert",
                      $psprintf("sdll %m sigo high or low period too short.
                                 Got %d Expected %d",
                                 $sampled(period_delta), sdll_width));
```

Figure 30: Glitch check with $time in always block

Recommendation 9: Avoid using **`$time`** directly in assertions due to differences in simulator support

Clock Jitter Impact

The DUT has an input clock (**`CK`**) and an output clock (**`Y0`**) which has a programmable offset, referred to as **`tSTAOFF`** in [2]. It specifies how long time after **`CK'`**s rising edge, **`Y0`** should have a rising edge, see Figure 31. The offset is programmable in a register in the chip and is referred to as **`ck2y`** in the following examples. When comparing the phase of two clocks with jitter, the jitter may cause the order of the edges to change which leads to more cases to verify. The examples in this section shows the progressively more complete assertions required to verify **`tSTAOFF`** timing in the DUT in the presence of jitter.

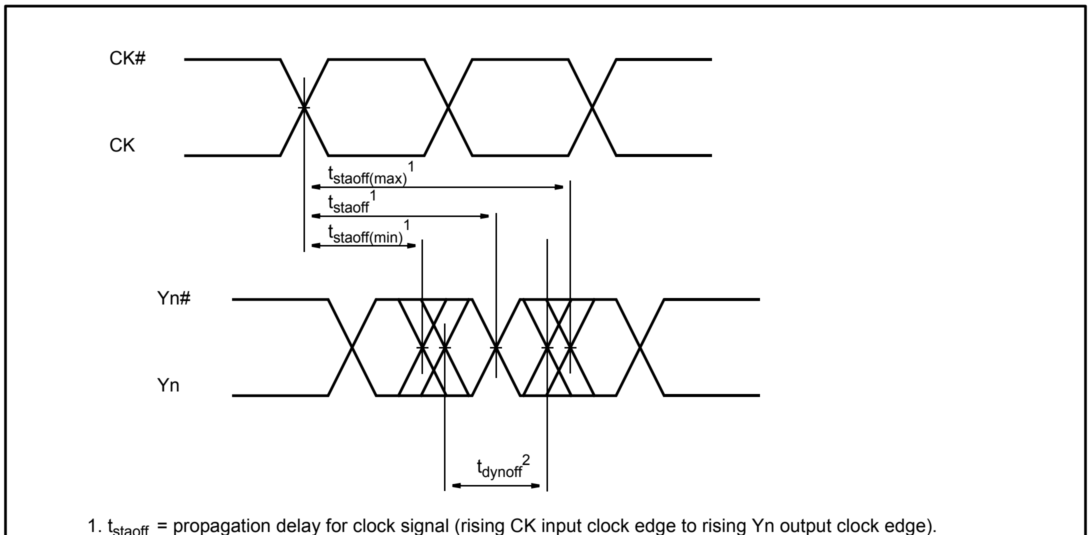

CK#
CK
$t_{staoff(max)}$[1]
t_{staoff}[1]
$t_{staoff(min)}$[1]
Yn#
Yn
t_{dynoff}[2]
1. t_{staoff} = propagation delay for clock signal (rising CK input clock edge to rising Yn output clock edge).

Figure 31: tSTAOFF definition from [2]

The most basic assertion, shown in Figure 32, needs to verify that the **`Y0`** clock has the offset that is programmed in the **`ck2y`** register.

```
always @ (posedge CK) start_time = $time;

a_tSTAOFF: assert property (@posedge Y0) disable iff (reset)
                           ($time – start_time) == ck2y
           else ovm_report_error("SVA phy_assert",
           $psprintf("tSTAOFF (%t) is different than programmed value (%t)“,
                          ($time – start_time), ck2y);
```

Figure 32: Clock offset assertion

However, using `$time` directly in a property doesn't work reliably in all simulators so moving it to an always block as shown in Figure 33 makes the assertion more portable.

```
always @ (posedge CK) begin
   start_time = $time;
   @ (posedge Y0);
   tSTAOFF = $time – start_time;
end

a_tSTAOFF: assert property (@(tSTAOFF) disable iff (reset)
                           (tSTAOFF == ck2y))
           else ovm_report_error("SVA phy_assert",
           $psprintf("tSTAOFF (%t) is different than programmed value (%t)“,
                          $sampled(tSTAOFF), ck2y);
```

Figure 33: Clock offset assertion with $time in always block

The output clock `Y0` has jitter caused by the DLLs and it is model by the delay variation techniques described in [5]. Jitter can make the delay longer or shorter which may push the `Y0` clock's edge before or after the reference edge of `CK`. The delay variation also means we can not compare the actual delay directly to the programmed delay value `ck2y`. The assertion needs to be changed, taking jitter into account and compare the delay to a range `[ck2y_min : ck2y_max]` of values instead.

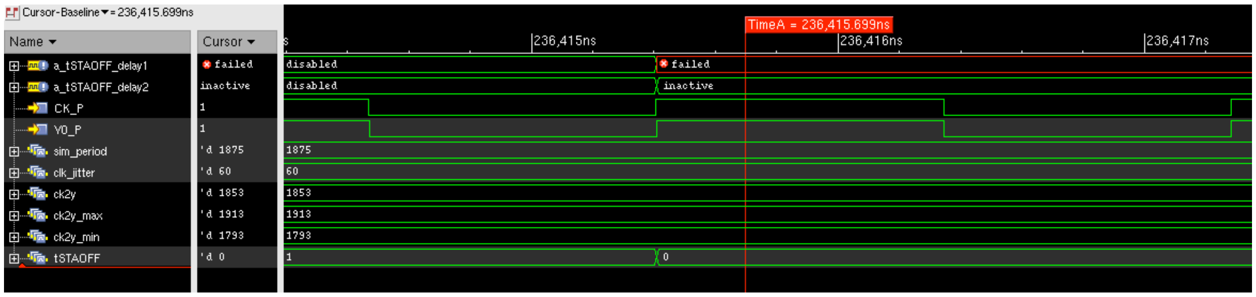
```
ck2y_min = ck2y – jitter;
ck2y_max = ck2y + jitter;

a_tSTAOFF: assert property (@(tSTAOFF) disable iff (reset)
                           (tSTAOFF >= ck2y_min) && (tSTAOFF <= ck2y_max))
           else ovm_report_error("SVA phy_assert",
                $psprintf("tSTAOFF (%t) is not between ck2y_min (%t)
                and  ck2y_max (%t)", $sampled(tSTAOFF), ck2y_min, ck2y_max));
```

Figure 34: Clock offset assertion with jitter

The changed assertion in Figure 34 passes in most cases but it fail with the error message in Figure 35 when the two clocks are in phase.

```
OVM_ERROR @ 236415.407 ns: reporter [SVA phy_assert] tSTAOFF (0.000 ns) is
not between ck2y_min (1.793 ns) and ck2y_max (1.913 ns)
```

Figure 35: Clocks in phase error



```
Cursor-Baseline = 236,415.699ns          TimeA = 236,415.699ns
Name               Cursor      |236,415ns            |236,416ns            |236,417ns
a_tSTAOFF_delay1   failed      disabled  failed
a_tSTAOFF_delay2   inactive    disabled  inactive
CK_P               1
Y0_P               1
sim_period         'd 1875     1875
clk_jitter         'd 60       60
ck2y               'd 1853     1853
ck2y_max           'd 1913     1913
ck2y_min           'd 1793     1793
tSTAOFF            'd 0        1  0
```

The programmed delay in this example is 1853ps which is close to the clock period, 1875ps. Adding 60ps of jitter to 1853ps moves the upper limit (**`ck2y_max`**) past the next rising edge of **`CK`**. The calculated offset **`tSTAOFF`** may be greater than the clock period. This is still in the allowed range between **`ck2y_min`** and **`ck2y_max`** illustrated by the two green lines A and B in Figure 36.

It turns out that there are two cases than needs to be considered:

- Programmed offset greater than the jitter (**`ck2y > jitter`**)
- Programmed offset smaller than the jitter (**`ck2y < jitter`**)

The case in Figure 36 can only occur when **`ck2y`** is greater than the clock jitter and the assertion needs to be updated. The value of **`tSTAOFF`** needs to fall either between **`ck2y_min`** and **`ck2y_max`** if **`ck2y`** plus jitter is less than the clock period. If **`ck2y`** plus jitter is greater than the clock period, the edge on **`Y0`** now occur after the edge on **`CK`**.

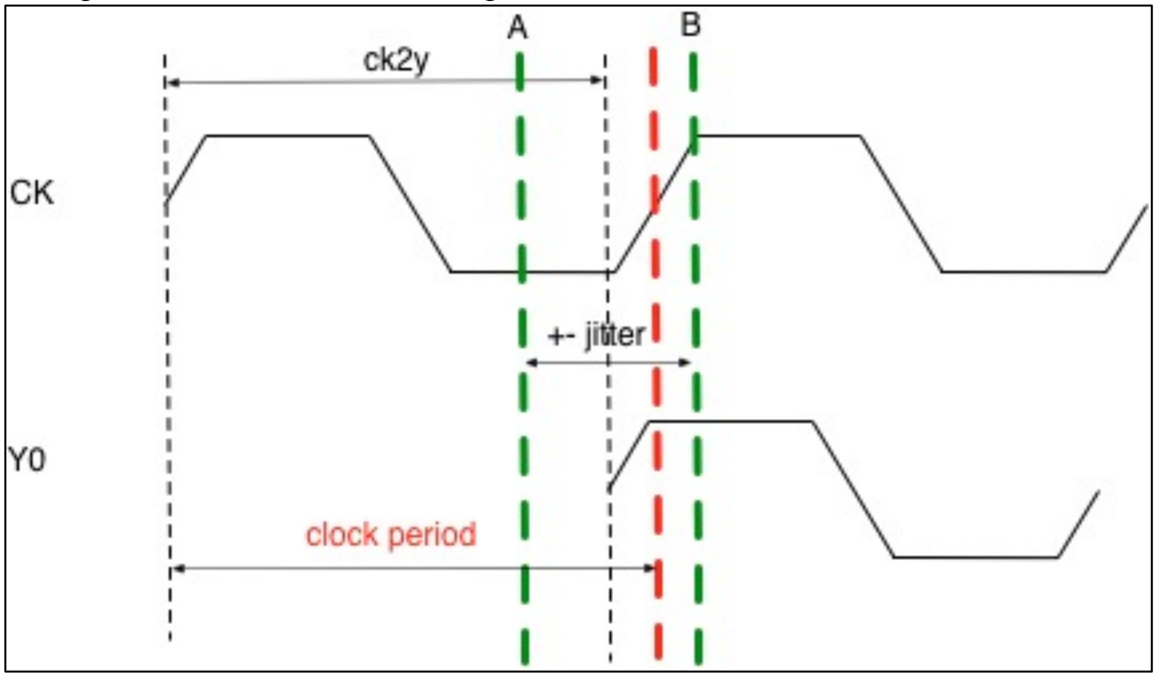
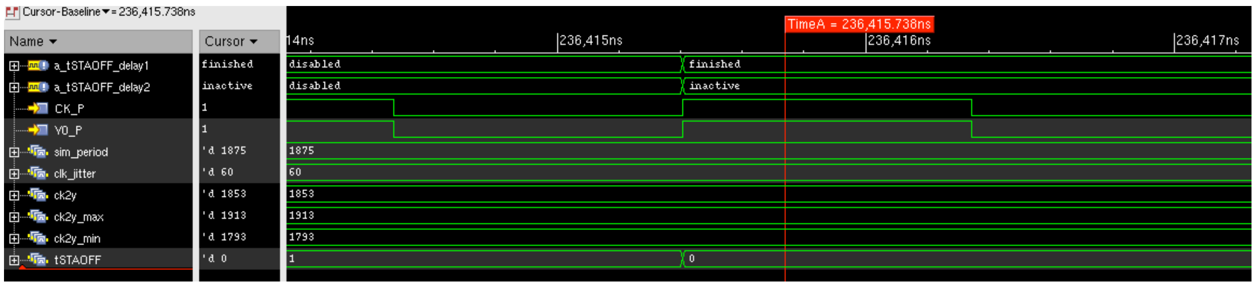


Figure 36: tSTAOFF with jitter, case 1

The assertion checking these two cases is shown in Figure 37. Note the exclusive OR operator "^" between the cases since only one comparison must be true.

```
a_tSTAOFF_delay1: assert property (@(tSTAOFF) disable iff(reset)
                  ((ck2y > jitter) |->
                  ((tSTAOFF > ck2y_min) && (tSTAOFF < ck2y_max)) ^
          (((tSTAOFF + ck2y) > ck2y_min) && ((tSTAOFF + ck2y) < ck2y_max)) ))
                else ovm_report_error("SVA phy_assert",
     $psprintf("tSTAOFF (%t) is not between ck2y_min (%t) and ck2y_max (%t)",
                                   $sampled(tSTAOFF), ck2y_min, ck2y_max));
```

Figure 37: tSTAOFF assertion with jitter (1)



The case where **ck2y < jitter** also need to be checked and this is easiest done with a separate assertion. The assertion needs to take the two cases into account where the actual delay falls between zero and **ck2y_max** i.e. between the rising edge of **CK** and the green line B in Figure 38. When **ck2y_min** is negative, it is the same as checking that the measured delay is **ck2y_min** before the next rising edge of **CK** since it is a periodic signal, i.e. between line A and the next rising edge of **CK** in Figure 38.

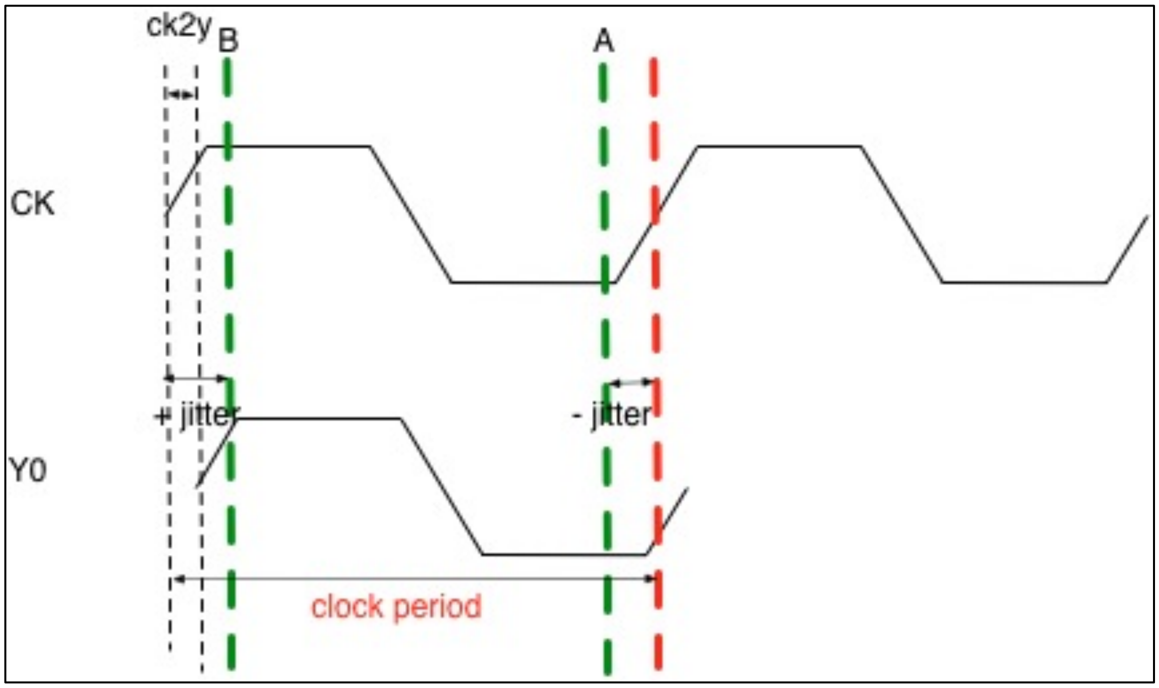


Figure 38: tSTAOFF with jitter, case 2

The new assertion in Figure 39 is similar to the one covering the case where **ck2y > jitter** but the ranges are calculated differently.

```
a_tSTAOFF_delay2: assert property (@(tSTAOFF) disable iff(reset)
                              ((ck2y <= jitter) |->
                      ((tSTAOFF >= 0) && (tSTAOFF < ck2y_max)) ^
           ((tSTAOFF > (sim_period - ck2y_min)) && (tSTAOFF < sim_period)) ))
               else ovm_report_error("SVA phy_assert",
               $psprintf("tSTAOFF (%t) is not between 0 and ck2y_max (%t) or
                  ck2y_min (%t) and sim_period (%t)",
                  $sampled(tSTAOFF), ck2y_max, ck2y_min, sim_period));
```

Figure 39: tSTAOFF assertion with jitter (2)

Recommendation 10: Carefully consider the impact of jitter on clock phases and write separate assertions for each case.

8. Conclusions and Recommendations

Several issues where encountered and resolved when developing assertions for sub-cycle timing verification and led to a number of recommendations being developed. SystemVerilog assertions were successfully used to verify the timing relationships of the PHY section of the DUT. A number of critical RTL bugs where found by assertions when used in combination with [5] that were missed in the OVM-only regression.

Examples of bugs found includes glitches and unknown values on clocks generated by DLLs and phase relationships being violated with respect to programmed values when delays were varied. One specific bug in the clock phase circuit found by the assertion in Figure 37 is shown in the waveform in Figure 41. The assertion failed with the error message in Figure 40. The assertion failed 9 times but the simulation ran to completion with no other errors. This means it would most likely have been missed in a simulation without assertions.

```
OVM_ERROR @ 85452.513 ns: reporter [SVA phy_assert] tSTAOFF (2.276 ns) is not between
ck2y_min (1.063 ns) and ck2y_max (1.183 ns)
```

Figure 40: Missing clock pulse assertion error

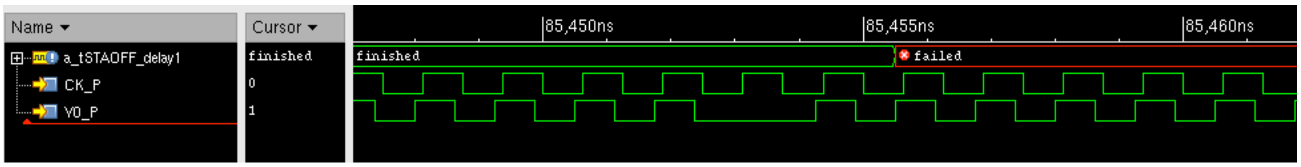


Figure 41: Missing clock pulse bug

Recommendations

1. Put assertions inside interfaces in separate files.
2. Connect all signals explicitly in the bind statement, if they are not automatically connected by `.*` in the bind port map.
3. Use assertions to check DUT signals only, never for testbench error checks.
4. Use the same `` `ifdef `` around the **`bind`** statement and **`interface`** declaration

5. Use **`$assertoff()`** to turn off assertions for specific testcases
6. When assertions depend on testbench configurations, set assertion parameters from the environment instead of accessing DUT registers.
7. Measure signal transition times instead of relying on a reference clock if jitter is modeled.
8. Use **`$sampled()`** to display the correct value in error messages when the value checked is the same as the assertion trigger.
9. Avoid using **`$time`** directly in assertions due to differences in simulator support
10. Carefully consider the impact of jitter on clock phases and write separate assertions for each case.

9. References

[1] JEDEC JESD79-3 DDR3 SDRAM Specification

[2] JEDEC SSTE32882 Registering Clock Driver Item #104 with Parity and Quad Chip Selects for DDR3 RDIMM Applications Specification

[3] IEEE 1800-2009 SystemVerilog Language Reference Manual

[4] Using SystemVerilog Assertions in Gate-Level Verification Environments, Mark Litterick, DVCon 2006

[5] Run-time Configuration of a Verification Environment – a Novel Use of the OVM/UVM Analysis Pattern, Paul Marriott, DVCon 2013

[6] SystemVerilog Assertions Design Tricks and SVA Bind Files, Cliff Cummings, SNUG 2009