SystemVerilog Assertions for Clock-Domain-Crossing Data Paths

Don Mills
Microchip Technology
2355 W. Chandler Blvd.
Chandler, AZ 85224
don.mills@microchip.com
mills@lcdm-eng.com

Abstract - SystemVerilog Assertions include capabilities to track data signals that cross clock domains. However, there can be a significant GOTCHA when using SVAssertions across clock domains verses the actual signal activity. This is due to the difference between event driven signal activity and SVAssertion cycle based monitoring. Depending on from/to clock frequencies, the receiving side could receive and process the data while the sending side SVAssertion is still being processed. By the time the SVAssertion transitions to the new clock domain, the data could be gone. This paper will provide guidelines for using SVAssertions for Clock-Domain-Crossing (CDC) data paths.

1.0 INTRODUCTION

The modeling of data paths crossing from one clock domain to another clock domain is a common topic for papers and articles in the digital design world of today. Many in-depth papers have been dedicated to the techniques and guidelines to manage the data path crossings [1][2][3][4]. A brief review of the basics of Clock Domain Crossing (CDC) will be presented in the following section. For in-depth details on CDC concepts and techniques, refer to the papers just previously sited. To support CDC, there are Electronic Design Automation (EDA) tools specifically directed towards CDC that provides features to analyze designs for CDC data paths and associated design practices. These features include monitoring for synchronizers on scalar data paths, checking if synchronizers were added to data buses (a bad thing to do) and checking if synchronized data paths converge at later pipe stages in the clock domain. Note this list is not all-inclusive. In addition to EDA CDC tools, the SystemVerilog Hardware Description Language provides assertions which can be used to track data as it crosses clock domains. Because the SystemVerilog assertion simulation model is cycle-based while the RTL simulation model is event driven, writing assertions to match the data as it crosses clock domains can be a little tricky. Assertions can be written that appear to be working but then fail if clock frequencies vary. Care must be taken to generate assertions that can work for all corner conditions.

SystemVerilog assertions applied to clock domain crossing has been highlighted as part of at least one paper [5] by Litterick, and is part of SV training classes as a means to monitor data as it passes between clock domains. The Litterick paper does show an example of using SystemVerilog assertions but does not cover the various corner conditions. This paper will explain and diagram a common approach for CDC assertions that is often used and show how and where it breaks down, followed by providing a SystemVerilog Assertions approach that will work for all corner conditions.

2.0 CDC BASICS

Clock Domain Crossing (CDC) is defined as a signal from a flip-flop clocked by one clock which is then captured by a flip-flop clocked by a different clock as shown in Figure 1.

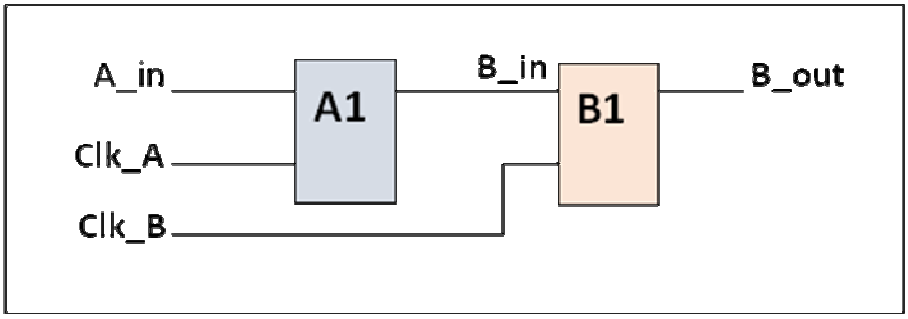


Figure 1 – CDC Diagram between flip-flops **A1** and **B1**

Figure 1 shows signal **B_in** from the **Clk_A** clock domain is captured by flip-flop **B1** in the **Clk_B** clock domain. Based on the clock relationship between **Clk_A** and **Clk_B**, the capturing of signal **B_in** by flip-flop **B1** can cause flip-flop **B1** to go metastable. This metastable state occurs when signal **B_in** is changing values within this setup time or the hold time window of flip-flop **B1** as shown in Figures 2 and 3.

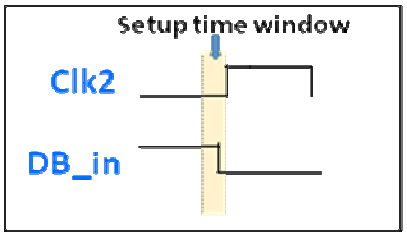


Figure 2 – Metastable Diagram – Setup time violation

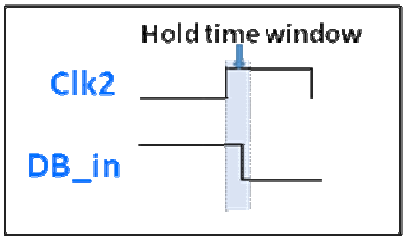


Figure 3 – Metastable Diagram – Hold time violation

Diagrams Figure 2 and Figure 3 show setup and hold time violations which will result in the flip-flop going into a metastable state. If a metastable result is left unchecked, it can propagate incorrect state values throughout the circuit that comes after the flip-flop. See [1] (Cummings 2008) section 2.1 for details and explanations of this phenomenon.

The most common method used to mitigate metastability propagation is to add a second flip-flop on the receiving side as shown in Figure 4. Sometimes, a third or even a forth (or more) synchronizing flip-flop is added to filter the metastable state. Statistical analysis of Mean Time Between Failure (MTBF) of the flip-flips used as synchronizers combined with the end use of the product should be the basis to determine if more than two flip-flops are required. The MTBF that is being considered here for analysis is the time is takes a flip-flop in a metastable state to resolve to a known state of either high or low.

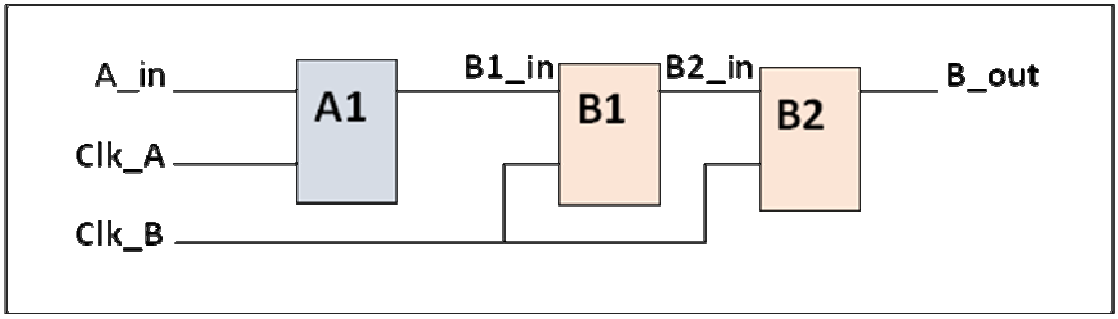


Figure 4 – Flip-flops **B1** & **B2** comprise a two stage Synchronizer

One of the design guidelines for synchronizer flip-flops such as **B1** and **B2** in Figure 4 is that the two flip-flops, should be placed close together during layout. Functionally, there should not be any combinational logic in the data path between the **B1** and **B2**, allowing for maximum metastability resolution time. Most companies (or libraries) have predefined, laid-out synchronizer macros which are hand-instantiated in RTL. The purpose of the second flip-flop (**B2** in Figure 4) is to filter (or block) a metastable signal from the first stage flip-flop from propagating to the circuit that follows. This implies, of course, that the metastability will resolve within the clock period minus other timing items such as **B1** propagation and **B2** setup. As noted in the previous paragraph, analysis of MTBF of these synchronizer flip-flops is needed to determine how many flip-flops are actually required. In most situations, two flip-flop synchronizers are sufficient.

Most EDA companies provide CDC analysis programs that check if synchronizers are in place as needed. Additionally, these analysis programs will monitor for a number of other CDC-type failure conditions based on CDC data paths. These tools are necessary because of the difficulty to observe and analyze CDC issues manually. It is important to note that during RTL simulation, CDC metastability does not exist since there are no setup and hold time timing constraints in RTL models for synthesis. These constraints are strictly gate-level or behavioral model conditions. There are papers that provide techniques showing how to model a two-stage synchronizer for RTL simulation that randomly delays its output to either two or three clock cycles[1][5]. These techniques provide a way to model the extra delay caused in the real world by metastability occurring on the first flip-flop stage. In an actual chip when the first stage goes metastable and then settles, it will resolve to either a 1 or a 0 state. This resolved state will be either to the old-original value or the new-updated value. When resolving to the old-original value, the first stage will clock in the new value on the next clock (clock 2). A third clock then clocks the new value into the second stage flip-flop. In other words, if stage 1 does not go metastable, it takes two clocks for the CDC data to pass through the two flip-flops. If stage 1 does go metastable, then it could take three clocks to pass the CDC data through the two flip-flops, based on how stage 1 resolves. These techniques can be scaled to apply to synchronizers that have more than two stages.

3.0 SVA Assertions Basics

SystemVerilog has two types of assertions: immediate assertions and concurrent assertions. Both types of assertions are used to perform tests on a design whenever the assertion is called or executed. When an assertion test is completed, a pass or fail statement from the assertion can be executed. Assertions provide a mechanism for the continuous monitoring of signals and conditions across all simulation regression tests. This section reviews basics of SystemVerilog concurrent assertions. Concurrent assertions will be used for monitoring signals across clock domains and through synchronizers.

3.1 Immediate Assertion

Immediate assertions execute in zero simulation time and can be used to monitor for illegal conditions such as an unknown in the condition of an "if" statement [6]. While this is important, it is not the focus of this paper.

3.2 Concurrent Assertions

Concurrent assertions use a clock or some other repetitive signal (referred to hereafter as the property clock) to trigger the concurrent assertion evaluation. The primary difference between immediate and concurrent assertions is

that concurrent assertions evaluate conditions over time, whereas an immediate assertion tests (start and finish) at the point in time when the assertion is called. The syntax for a concurrent assertion directive is:

```
assert property (property_expr) [pass_statement;] [else fail_statement;]
```

The argument to **`assert property`** is a *property expression*. A *property expression* is comprised of a clock specification and sequences of Boolean expressions tested over time, as well as property and sequence operators. The expressions are evaluated on the clock edge per the clock specification. It is important to note relative to this paper, that the value used per a clock edge is the value sampled in the preponed region of the time step. The sequence of Boolean expressions can be spread over multiple clock cycles by using the **##** cycle delay operator between each expression. This is a very simplistic view of assertions. Other features and syntax relative to CDC will be discussed in the following sections.

3.2.1 Property Implication Operator

Concurrent assertions are activated to start a thread each clock cycle throughout simulation. These assertions will run concurrent with the design functionality. To prevent enormous amounts of unneeded concurrent threads that are "don't cares" from consuming simulation time, concurrent assertions use the implication operator to determine if its sequences should be evaluated and consume simulation resources. For example, an assertion with a sequence that takes twelve clock cycles to execute could possibly have twelve concurrent threads running at the same time, each thread starting on each subsequent clock cycle.

Consider a simple test to show that a **`req`** is followed by a **`grant`**. In Example 1, a property is coded to test this simple sequence without using an implication operator. The **`bus_req_prop`** property triggers "to start" on each cycle from cycle **0** through cycle **10**. Only the thread that starts at cycle **1** passes, the remaining threads fail and could be counted in the total fail count for the simulation.

```
property bus_req_prop;
  @(posedge clk) req ##[1:5] grant;
endproperty:bus_req_prop

assert property (bus_req_prop);
```

Example 1 – bus_req_prop code for Figure 5

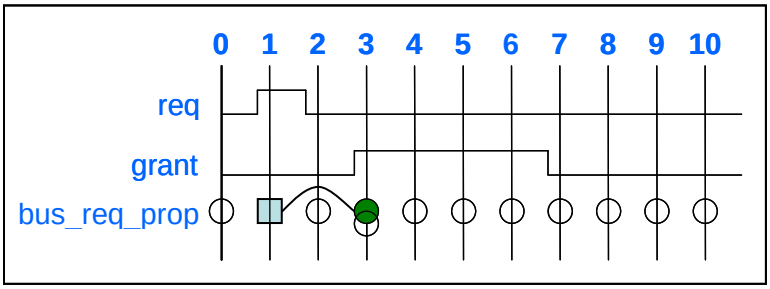


Figure 5 – Sequence and Property Pass/Fail for all the threads between Cycle **0** and Cycle **10**

In Figure 5, cycle **1** has neither a pass nor a fail because there are no threads ending at that point in time. Cycle **3** has both a pass and a fail due to two separate threads ending at that cycle. The thread that started at cycle 1 successfully completed at cycle **3** with a pass. The thread that started at cycle **3** ended immediately with a fail, because **`req`** is false at that cycle.

The behavior of the assertion diagramed in Figure 5 is not practical due to assertion failures occurring almost every cycle. More important is that failures are on conditions that are "don't cares". If **`req`** is not active, the sequence should not be tested. To make assertions usable, the assertion property needs to be modeled so that it will only test during expected event cycles and be idle during "don't care" cycles. SystemVerilog properties make this possible by using the implication operator. Typically, assertions property expressions are specified with an

implication operator. There are two implication operators: overlapping **|->** and non-overlapping **|=>**. An implication operator tells the property not to evaluate the sequence expression following the implication operator unless the condition before the operator is true. The expression to the left of the implication operator is called the antecedent and the expression following the implication operator is called the consequent. The difference between the two implication operators is when the consequent testing begins. An overlapping implication operator begins testing the consequent in the same time step that the antecedent passes true. The non-overlapping implication operator delays one clock cycle between the antecedent testing true and the consequent test starting.

The **req/grant** code in Example 1 should only test the sequence when **req** is true. For the clock cycles where **req** is false, the assertion is a "don't care" and the sequence should not to be evaluated. In Example 2, the implication operator is used to prevent or guard the consequent expression from testing when **req** is false. The assertion does not fail; it simply does not run and returns a vacuous success.

```
property bus_req_prop;
  @(posedge clk) req |-> ##[1:5] grant;
endproperty:bus_req_prop

assert property (bus_req_prop);
```

Example 2 – bus_req_prop code with implication operator for Figure 6

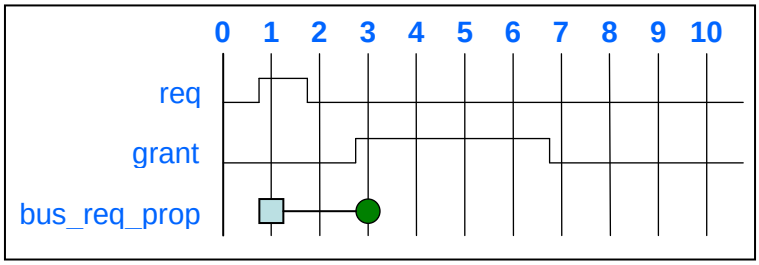


Figure 6 – Property Pass/Fail for all the threads between Cycle **0** and Cycle **10** for code example 2.
(Vacuous Successes are not noted.)

Figure 6 shows that by utilizing an implication operator, only one thread starts and completes. A single "pass" is recorded for the simulation.

A number of papers have been written that focus on syntax and utilizing various features of SystemVerilog assertions [6][7][8][9]. Please refer to these papers (and others found in "google land"), in addition to the IEEE 1800-2012 standard [10], for more details on the various ways to apply assertions to designs.

3.3 Concurrent Assertions with Multiple Clocks

The application of assertions for this paper is to apply assertion testing with multiple clocks. SystemVerilog assertions provide a number of ways to transition between multiple clocks. This paper will focus on the operators ##0, ##1, |-> and |=> to transition between clocks.

4.0 SVAssertions To Monitor CDC

SystemVerilog assertions have been promoted as a means to monitor and verify data as it passes across clock domains. Figure 7 is used as template for discussing CDC along with the RTL code in Example 3.

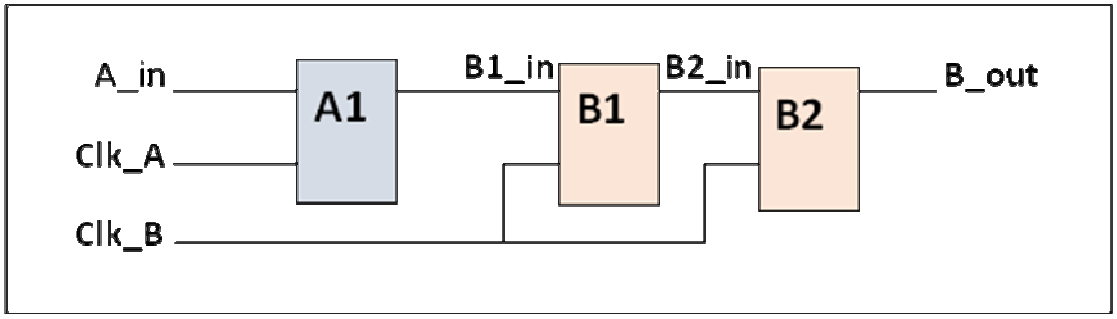


Figure 7 – Clock Domain Crossing of flip-flip A1 output to flip-flop B1 input

```
always_ff @(posedge Clk_A)
   B1_in <= A_in;

always_ff @(posedge Clk_B)
   begin
      B2_in <= B1_in;
      B_out <= B2_in;
   end

property CDC_prop1;
   @(posedge Clk_A) $rose(A_in) |=> $rose(B1_in)
   ##1
   @(posedge Clk_B) 1'b1 ##[2:3] $rose(B_out)
endproperty:CDC_prop1

assert property (CDC_prop1);
```

Example 3 – Simple RTL Code snippet and Property for Figure 7

This code snippet represents the commonly used technique for RTL coding of CDC and a simplified SVAssertion property to monitor CDC for the design in Figure 7. This property only monitors for a rising edge of **A_in**. A full model which monitors for any transition on **A_in** will be shown in a later section. The following steps describe each part of **CDC_prop1**:

1. **@(posedge Clk_A) $rose(A_in)** The antecedent waits for a **$rose** transition on **A_in.** When this condition is true, the consequent of the implication operator begins testing.

2. **|=> $rose(B1_in)** A non-overlapping implication operator is used to model the clock delay through the flip-flop. The output of the flip-flop is then tested for a rising edge in the **Clk_A** domain.

3. **##1** As noted in section 3.3, this operator is used to transition to another clock source.

4. **@(posedge Clk_B) 1'b1 ##[2:3] $rose(B_out)** Using **Clk_B**, wait two or three clock cycles for the output to show a rising edge as the data passes through the two synchronizers (**B1** and **B2**). The output, **$rose(B_out)**, will arrive in two **Clk_B** cycles if the first stage (flip-flip **B1**) does not go metastable. Three **Clk_B** cycles are needed if flip-flip **B1** does go metastable.

A_in is clocked into the flip-flop **A1.** The output of **A1** becomes flip-flop **B1** input labeled **B1_in**. **B1_in** passes through the two synchronizers (**B1** and **B2**), resulting in output signal **B_out**.

The property **CDC_prop1** will monitor for a rising edge of **A_in** followed by testing the **$rose(B1_in)** after the non-overlapping implication operator in the **Clk_A** domain. It is crucial to point out that the **$rose(B1_in)**

will not test true until one clock following flip-flop **A1** going high. This is due to SystemVerilog assertions sampling in the preponed region [10] of a time step. That is, SystemVerilog assertions use the value of a signal prior to a clock changing the signal.

The property transitions to the **Clk_B** domain via the **##1** and waits the required 2 to 3 cycles while the signal passes through the synchronizers. Two cycles are required in the following two cases: first, if the first stage of the synchronizer does not go metastable; second, if it does go metastable and settles to the new data value. Three cycles are needed when the first stage does go metastable and settles to the old-original data value.

Finally, the output of the synchronizers is tested for the transition of **B_out** using the **$rose** function. A more complete model will manage transitioning of the input to high or low, and verify that result out is the same value that was clocked in. The more complete model will be discussed in the next section.

SystemVerilog Assertions has two ways of waiting for the 2 to 3 cycles needed for the two synchronizers. The approach shown above uses the **##[2:3]** cycle range. The property then delays the cycle min-max range time specified, while looking for the next condition to test true. The other approach utilizes a **[*2:3]** repetition. This approach requires the test condition prior to the repetition to remain true throughout the delay cycles. If the property uses a **1'b1** as in step 4 above, the "remain true" condition will hold true through the repetition. From this perspective, both modes of delay provide the same result. However, when testing for the change on the signal, the repetition does not work. The recommendation of this paper will be to test for a change on the signal as the signal transitions from one clock domain to another.

If the condition before the delay uses the CDC data value as shown in examples below, the cycle range will do a single data value sample during the time step prior to the cycle range starting, while the repetition will require that the data remains true throughout the cycle. Due to the asynchronous nature of CDC data signals, the property cannot guarantee the number of cycles that the data will be present in order for the repetition operator to be successful. This is a second reason to not use repetition for CDC properties.

The property in code Example 3 may look right. It covers all the transitions of the three flip-flops used for CDC. Unfortunately, this property does not work under all the various permutations of source/destination clock frequencies. If this property is used, it could give false negatives for CDC signals. The problem is that RTL simulations use event driven simulators, causing actions to occur as events are scheduled. Simulations of SystemVerilog Assertions use cycle-based sampling and samples from the preponed region of the specified clock transaction. Applying event- and cycle-based simulation to data crossing clock domains means the assertion monitoring involves two unique event driven clock regions and two unique cycle-based clock regions. Ultimately, the issue simplifies to when the event driven signals cross from the source clock domain to the destination clock domain versus when the cycle-based assertions makes this same transition. Adding to this complexity is the variation of to/from clock frequencies.

4.2 A More Complete Property Model

Before the discussion continues on how to model assertions across clock domains, consider a more complete property model. The model must be able to test for either rising or falling conditions on the input and then test for this new value as the value transfers from the source flip-flop in the originating clock domain through synchronizers in the destination clock domain.

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
                   ($changed(B1_in) && (B1_in === v_temp))
  ##1
  @(posedge Clk_B) ($changed(B1_in) && (B1_in === v_temp))
  ##[2:3]          ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop1
```

Example 4 – Simple RTL Code snippet and Property for Figure 7

In Example 4, the **$changed** function is used to test for changes on **A_in** and a local variable, **v_temp**, is introduced to the property to store the value of **A_in** when it changes values. This local variable is used to verify the sampled changed value of **A_in** when it appears in each of the flip-flop stage outputs.

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=> 1'b1
  ##1
  @(posedge Clk_B)  $changed(B1_in)
  ##[2:3]          ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop1
```

Example 5 – Simple RTL Code snippet and Property for Figure 7

Example 5 is a variation of Example 4, as it uses a "pass" place holder after the implication operator and drops the temp variable test in the **Clk_B** domain prior to waiting for the **##[2:3]** cycle delay for the synchronizers. The final output does the complete test of checking for a change of **B_out,** and that **B_out** is the correct value.

Example 6 is another variation which drops the checking of **B1_in** before the **##[2:3]** cycle delay and uses a "pass" instead. The final test at the end of the property is all that is actually required. This approach is NOT recommended because it will not be accurate under all corner conditions when running gate level simulation (GLS).

```
property CDC_prop2;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=> 1'b1
  ##1
  @(posedge Clk_B) 1'b1
  ##[2:3]          ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop2
```

Example 6 – Property that only tests the beginning and the end of a CDC transaction

The properties shown in Example 5 and 6 are used below to highlight the event versus cycle-based clocking problem with SystemVerilog assertions.

The following code is an extended model for Figure 7. This code has an option to simulate with either a two- or a three-cycle delay through the synchronizers. There are synchronizer models shown in papers [1] that randomized the two- or three-cycle delay, but that feature is not needed for the tests used for this paper.

```
always_ff @(posedge Clk_A)
  B1_in <= A_in;

always_comb //add extra FF cycle delay if dly is true
  if (dly) B2_in = B1_dly;
  else     B2_in = B1_out;

always_ff @(posedge Clk_B)
  begin
    B1_out <= B1_in;
    B1_dly <= B1_out;

    B_out  <= B2_in; //from the always_comb block above
  end

property CDC_prop1;
```

```
    logic v_temp;
    @(posedge Clk_A)($changed(A_in), v_temp = A_in) |=> 1'b1
    ##1
    @(posedge Clk_B) $changed(B1_in) ##[2:3]
                     ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1

  property CDC_prop2;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=> 1'b1
    ##1
    @(posedge Clk_B) 1'b1  ##[2:3]
                      ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop2

  apCDC_prop1: assert property (CDC_prop1);
  apCDC_prop2: assert property (CDC_prop2);
```

Example 7 – Primary body of code used for simulation tests

Consider the clock frequency variations for **Clk_A** and **Clk_B** in the following screen shots. The following figures will only display **CDC_prop1** because the results for both properties are identical. In cases where **CDC_prop2** is different, it will be noted shown.

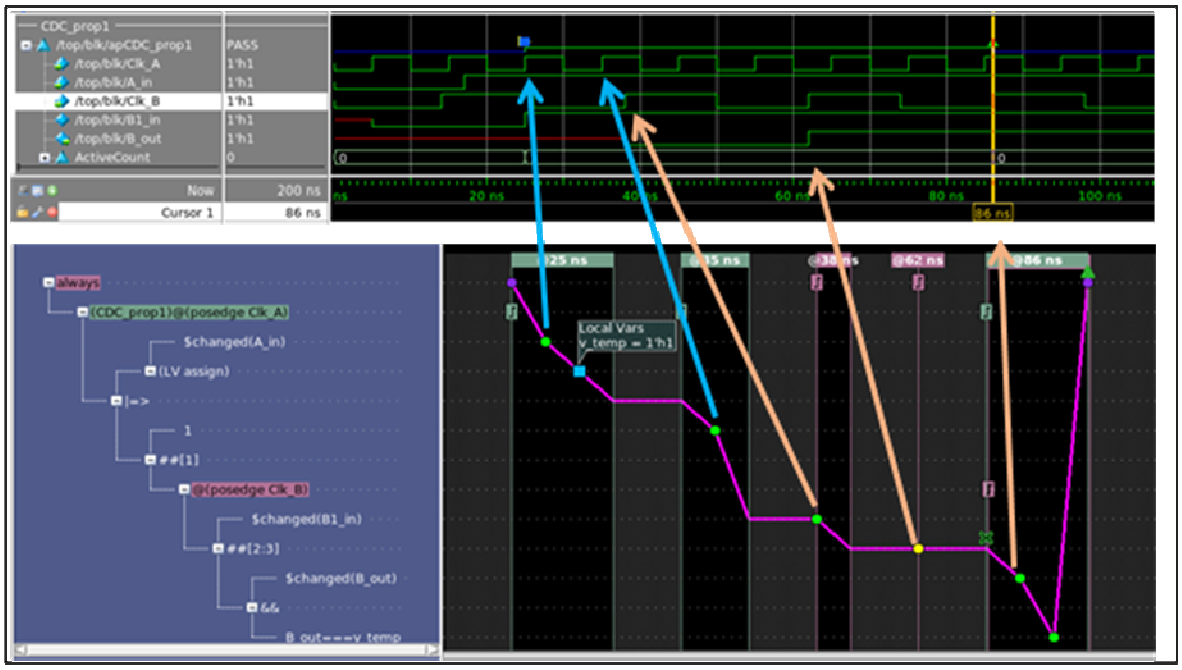


Figure 8 – Passing Assertions using Example 7 code. Two **Clk_A** rising edges before a **Clk_B** rising edge

In Figure 8, the assertion passes and all is well! The upper panel in Figure 8 is a wave view of **CDC_prop**1 showing it's starting and ending points and the signals associated with the assertion property. The lower panel displays each component of the assertion in a tree form on the left. Each line of this panel represents a test or a transition. The vertical columns on the right hand side of the panel represent time points where test results of individual conditions of the property are noted. In the first column at time **25ns**, the assertion starts (purple dot), **$changed (A_in)** tests true (green dot) and then **v_temp** is assigned a value (blue square.) Next, the non-overlapping implication operator transition the property to the next time step column of **45n**s where a true (1) is tested. This is followed by the transition to the **Clk_B** domain. Note the time for the columns are highlighted with a different col-

or denoting a different clock domain. The test for **$changed(B1_in)**tests true in column **38ns** (green dot). This is then followed by two clock delays at **62ns** and **86ns**. Column **86ns** represent 2 cycles of the **##[2:3]** cycle range so **$changed(B_out)** and **(B_out === v_temp)**are tested (green dots). Since both of those conditions pass true, the assertion passes.

The arrows map the lower panel to the wave window. The two blue arrows show the assertion samples in the **Clk**_A domain. The three tan arrows show the assertion samples in the **Clk_B** domain. In this simulation example, there was no delay applied to the synchronizers, i.e., the data passed through the synchronizers in two clock cycles. Adding an additional stage delay to model metastability does not change the results for this case. Note that the two **Clk_A** samples occurred prior to any **Clk_B** rising edges. This is significant! This is not be the case in the next example.

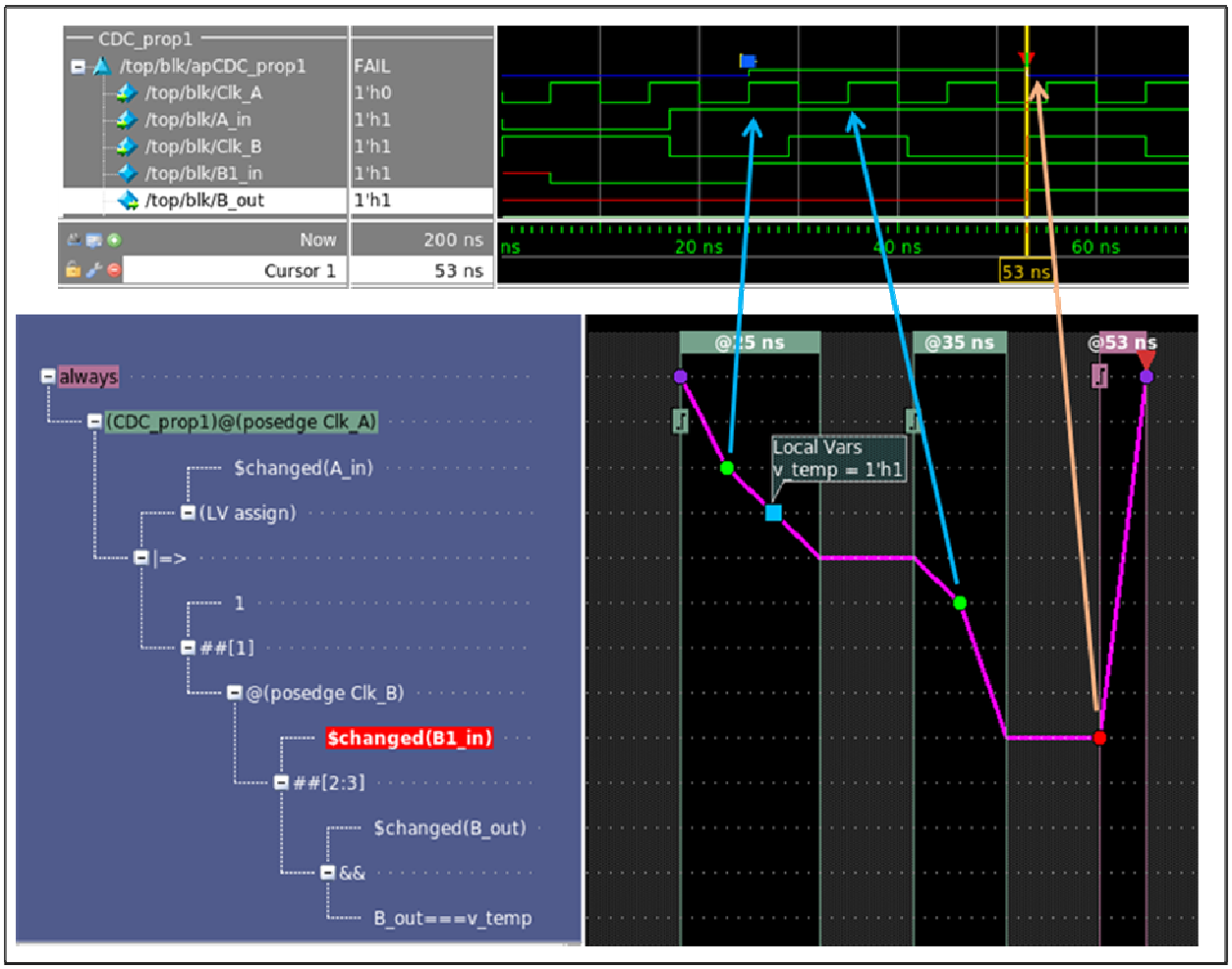


Figure 9 – Assertion fails using Example 7 code. Only one **Clk_A** rising edge before a **Clk_B** rising edge

The same assertion that passed in Figure 8 now fails in Figure 9. The assertion sampling does not cross over into the **Clk_B** domain until a second clock in **Clk_A** has occurred. This is due the property requiring a test in the Clk_A clock domain following non-overlapping implication operator. While the property is stuck in the Clk_A domain, the actual data signal has moved on the Clk_B domain at time **25ns**. The next Clk_B at time **29ns** samples **B1_in**. The property **CDC_prop1** fails at time **53ns** because there was no change on the value of **B_in** between the sampling at time **29ns** and at time **53ns** in the **Clk_B** domain. (**B_i**n is sampled high at both **29ns** and **53n**s.) What happened is the event simulation mode moves on to the new clock domain across the CDC, while the assertion stays in the old clock domain until all the property conditions have been satisfied. This example failed

almost immediately, whereas **CDC_prop2** for the same simulation does not fail until the end of the **Clk_B** part of the property as shown in Figure 10.

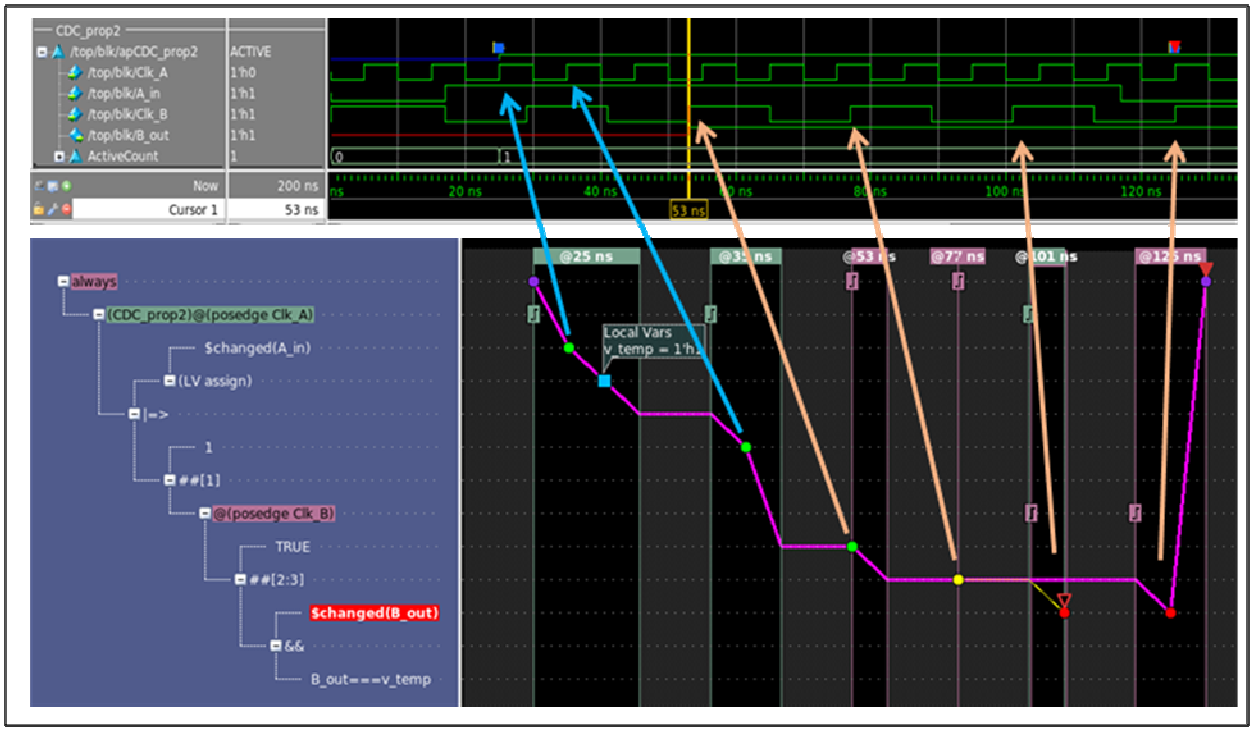


Figure 10 – Assertion fails using Example 7 code. Only one **Clk_A** rising edge before a **Clk_B** rising edge

CDC_prop2 fails because there is no change on **B_out** between samples at time **101ns** and **125ns** (red dots.) Note that at column **101ns** which is cycle 2 of the **##[2:3]**cycle range, **$changed(B_out)** tests false (red dot). Column **125ns** is cycle 3 of the **##[2:3]**cycle range and since **$changed(B_out)**again tests false (red dot), the property fails. Both **CDC_prop1** and **CDC_prop2** fail, but **CDC_prop2** failure does not show up until cycles later in the assertion due to no intermediate tests of the signal as it transitions through the design. This property should have shown a pass at time **77ns**.

The observation between Figure 8 that passes and Figures 9 and 10 that fail is the source/destination clock frequency differences and where the destination sampling edge line up relative to the source clock edges. To fix property so it tracks the CDC data flow properly, the CDC property must be modified to not test in **Clk_A** domain after the implication operator. Appendix C shows a number of variations that were tested with all the test variations listed in Appendix B. If the input-to-output of flip-flop **A1** (Figure 7) needs to be tested with an assertion, then a separate assertion should be used. Do not test the input-to-output of **A1** in the same assertion that is used to test the data path across the clock domains.

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |->
  ##1  // this is not needed
  @(posedge Clk_B) $changed(B1_in) ##[2:3]
                  ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop1

property CDC_prop2;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |->
  ##1  // this is not needed
  @(posedge Clk_B) 1'b1  ##[2:3]
                   ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop2
```

Example 8 – Properties that will work for CDC

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
  @(posedge Clk_B)  $changed(B1_in) ##[2:3]
                   ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop1

property CDC_prop2;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
  @(posedge Clk_B) 1'b1  ##[2:3]
                   ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop2
```

Example 9 – Different way to code Properties that work for CDC

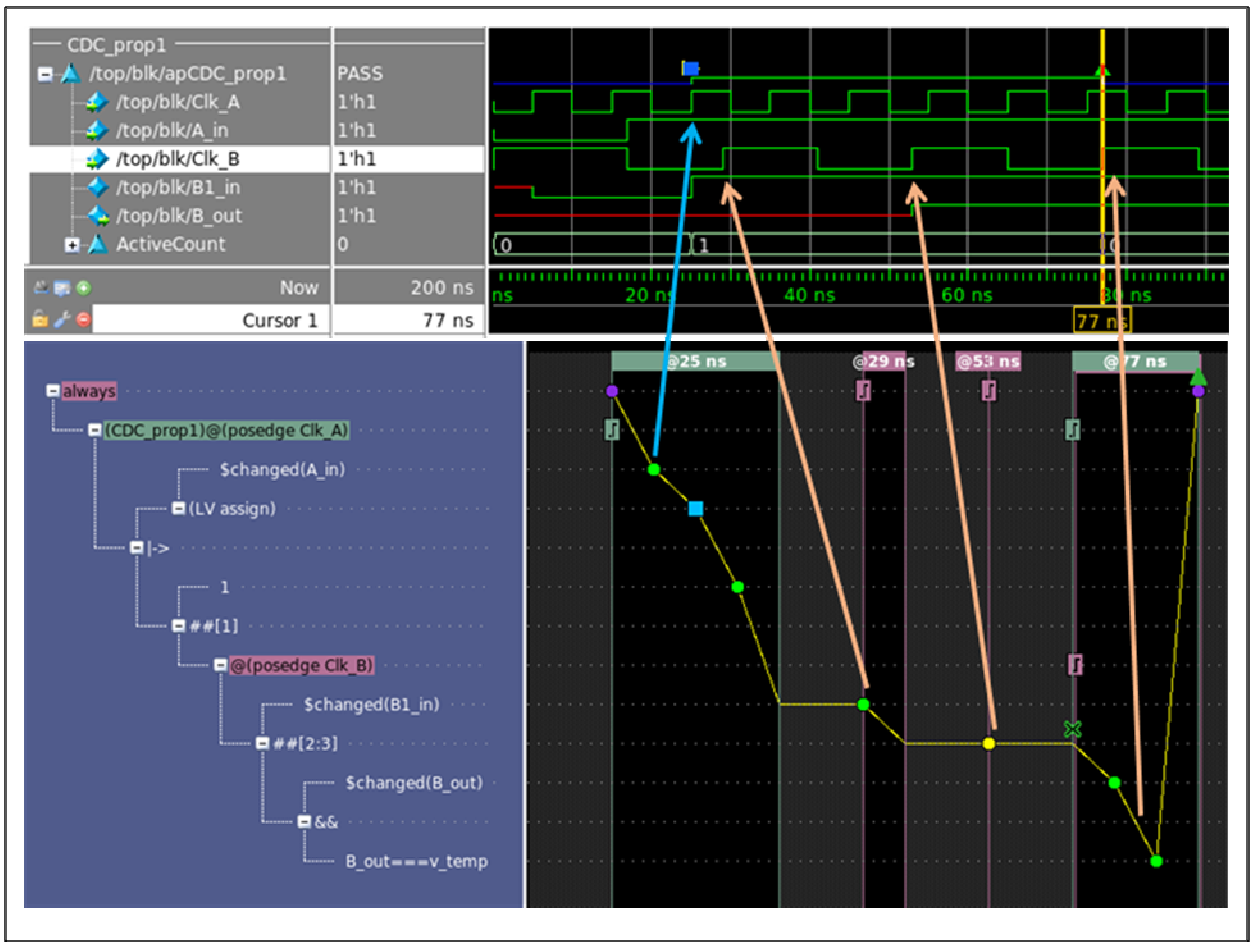


Figure 11 – Same stimulus used for Figure 9. Now the Assertion passes using the new property model.

In Figure 11, the first column of the lower panel at time **25ns**, the assertion starts (purple dot), **$changed (A_in)** tests true (green dot) and then **v_temp** is assigned a value (blue square.) Next, while still in the 25ns column, the overlapping implication operator and the ##1 transition the property to the **Clk_B** domain. The property is immediately testing for **$changed(B1_in)** which tests true in column **29ns** (green dot). This is then followed by two clock delays at **53ns** and **77ns**. Column **77ns** represent 2 cycles of the **##[2:3]** cycle range so **$changed(B_out)** and **(B_out === v_temp)** are tested (green dots). Since both of those conditions pass true, the assertion passes.

4.3 Variations of CDC Operators

Both types of implication operators (**|->** & **|=>**) can be used to traverse between clock domains. Additionally, both **##0** and **##1** operators can also be used. The only time there will be a difference in functionality between overlapping and non-overlapping or the **##0** and **##1** will be when the destination clock lines up perfectly with the source clock. Refer to the IEEE 1800-2012 standard [10] for a detailed explanation of this concept. This is a subject for another paper.

4.4 Modeling for Both RLT and GLS

Examples 8 & 9 shows the variations of the two models used for this paper. Appendix C shows the full models that was used. It is critical to note that only **CDC_prop1** in either Example 8 or 9 will work properly for GLS. The difference between RTL and GLS property modeling is an occurrence of a destination clock happening just after a source clock. In RTL, the simulation will transition to the destination clock domain. But in GLS, this "close" destination clock would be occurring during the clock-q delay of the source flip-flop. Thus the data sampling in the destination clock domain is one clock later in GLS than in RTL.

To fully make CDC_prop1 compatible for both RTL and GLS, an additional variable delay is required. The additional delay is highlighted in Example 10.

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |->
  @(posedge Clk_B)
       ##[0:1]  $changed(B1_in)
       ##[2:3] ($changed(B_out) && (B_out === v_temp));
endproperty:CDC_prop1
```

Example 10 – Property that will work for CDC with both RTL and GLS

The code in Example 10 works for signals crossing from one clock domain to another with the final value being verified in the destination clock domain.

If the verification intent is to monitor the a data path from source clock domain to a destination clock domain and then loop back to the source clock domain, then the assertion just adds an additional CDC transition stage as shown in Example 11.

```
property CDC_prop1;
  logic v_temp;
  @(posedge Clk_A) ($changed(A_in), v_temp = A_in)
  |->
  @(posedge Clk_B)
       ##[0:1]  ($changed(B1_in) && (B1_in === v_temp))
       ##[2:3]  //wait for synchronizers
  ##0
  @(posedge Clk_A)
       ##[0:1] ($changed(B_out) && (B_out === v_temp))
       ##[2:3] ($changed(A_out) && (A_out === v_temp));
endproperty:CDC_prop1
```

Example 11 – Property that will work for CDC with both RTL and GLS

The code in Example 11 works for signals crossing from one clock domain to another and then back to the source clock domain. There is often a more content to this loopback path but the above code will provide a starting point for anyone applying SystemVerilog assertions to such a CDC loop back data path.

5.0 CONCLUSIONS

There are many papers published that discuss CDC, but few papers focus on assertions for CDC. It is easy to write assertions that only work some of the time for CDC data paths. The objective of this paper was to present a solution that will work when applying assertions to all CDC data paths regardless of source/destination clock frequency or clock edge relationships. Using the properties presented in Examples 10 & 11 as models, assertions can be written that work for both RTL and GLS. These assertions are not trivial and require testing across to ensure correctness.

REFERENCES

[1] Clifford E. Cummings, "*Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog*," SNUG 2008, Boston, MA

[2] Clifford E. Cummings, "*Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*," SNUG 2001, San Jose, CA

[3] Frank Herman Behrens, Walter Soto Encinas Junior, "*Clock Domain Crossing Check Based on Assertions A Case study in IP Design*," www.lbd.dcc.ufmg.br/colecoes/wcas/2011/0017.pdf

[4] Shubhyant Chaturvedi, "*Staic Analysis of Asynchronous Clock Domain Crossing*," DATE 2012

[5] Mark Litterick, "*Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertions*," DVCon 2006

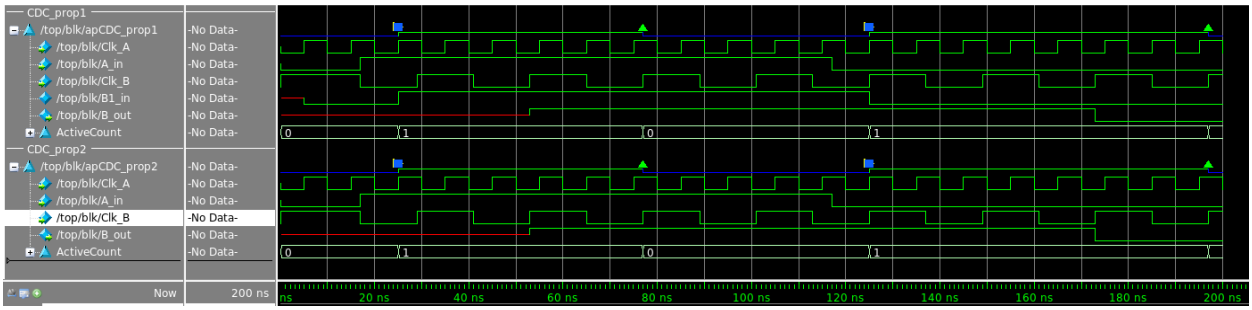
[6] Don Mills, "*Being Assertive With Your X (SystemVerilog Assertions for Dummies),*" SNUG 2004

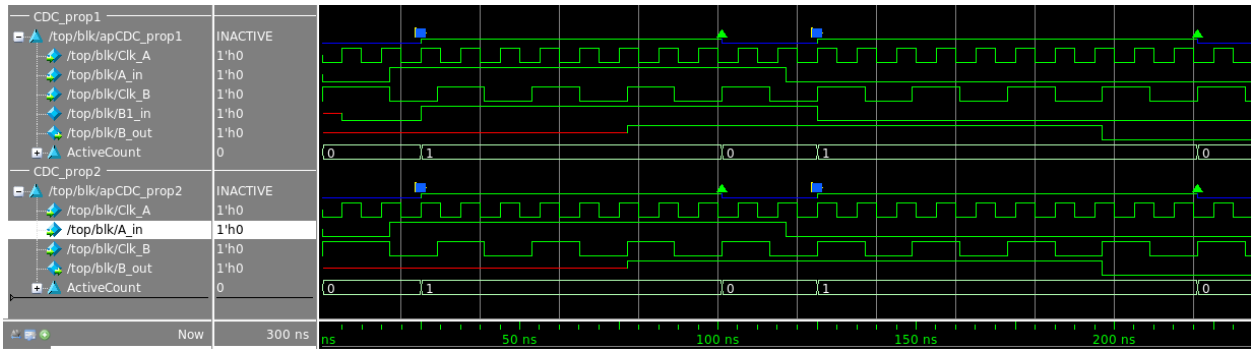
[7] Don Mills, "*If Chained Implications Weren't so Hard, They'd be Easy*", SNUG 2009
[8] Don Mills, Stu Sutherland, "*Assertions are for Design Engineers Too*", SNUG 2006
[9] Clifford E. Cummings, "*SystemVerilog Assertions, Design Tricks and SVA Bind Files*", SNUG 2009
[10] *"1800-2012 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2013. ISBN: 978-0-7381-8110-3 (PDF), 978-0-7381-8111-0 (print).

[11] Don Mills, Clifford E. Cummings "*Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?*" SNUG 2002, San Jose, CA
[12] Don Mills, Clifford E. Cummings, Steve Golson "*Asynchronous & Synchronous Reset Design Techniques – Part Deux,*" SNUG 2003, Boston, MA

APPENDIX A

Additional screen shots of the assertions used for this paper are shown below. These screen shots show **A_in** transitioning from low to high and then later from high to low. Both transitions test successfully for both properties in both panels. The delay through the synchronizers in the first panel is 2 clock cycles and the delay for the synchronizers in the second panel is three clock cycles.





APPENDIX B

Full code used for tests for this paper with the “bad” properties.

```
module blk
  (input  var logic A_in, Clk_A, Clk_B,
   output var logic B_out);

  logic B1_in;
  logic B1_out;
  logic B1_dly;
  logic B2_in;
  enum  bit {FALSE, TRUE} temp;
  bit   dly;

  initial
    if ($value$plusargs ("dly=%b", dly))
      $display("dly = ", dly);
    else
      $error ("Error: no value for dly input");

  always_ff @(posedge Clk_A)
    B1_in <= A_in;

  always_comb
    if (dly) B2_in = B1_dly;
    else     B2_in = B1_out;

  always_ff @(posedge Clk_B)
    begin
      B1_out <= B1_in;
      B1_dly <= B1_out;

      B_out  <= B2_in;
    end

  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
    ##1
    @(posedge Clk_B) $changed(B1_in)
    ##[2:3]          ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1

  property CDC_prop2;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
    ##1
    @(posedge Clk_B) TRUE
    ##[2:3]          ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop2

  apCDC_prop1: assert property (CDC_prop1);
  apCDC_prop2: assert property (CDC_prop2);
endmodule:blk
```

Top level model

```
module top;
  logic A_in;
  logic Clk_A;
  logic Clk_B;
  logic B_out;

  blk blk (.*);

  testbench TB (.*);

endmodule:top
```

The following four code blocks are the four tests used to test the code blk above. These tests differ primarily by changing the clock periods for Clk_A and Clk_B.

```
//test1
module testbench
  (output var bit Clk_A,
   output var bit Clk_B,
   output var bit A_in);

  localparam int A = 5;
  localparam int B = 12;

   initial begin
     Clk_A <= 0;
     Clk_B <= 0;
     fork
       forever #A Clk_A = ~Clk_A;
       #(A/2) forever #B Clk_B = ~Clk_B;
     join_none
   end

   initial begin
                $assertoff;
                A_in = 0;
     #(A * 2)   $asserton;
     #(A+(A/2)) A_in = 1;
     #(A * 5)  A_in = 0;
  end

endmodule:testbench
```

```
//test2
module testbench
  (output var bit Clk_A,
   output var bit Clk_B,
   output var bit A_in);

  localparam int A = 5;
  localparam int B = 12;

   initial begin
     Clk_A <= 0;
     Clk_B <= 1;
     fork
                  forever #A Clk_A = ~Clk_A;
      #A          forever #B Clk_B = ~Clk_B;
     join_none
   end

   initial begin
                $assertoff;
                A_in = 0;
     #(A * 2)   $asserton;
     #(A+(A/2)) A_in = 1;
     #(A * 5)  A_in = 0;
  end

endmodule:testbench
```

```
//test3
module testbench
  (output var bit Clk_A,
   output var bit Clk_B,
   output var bit A_in);

  localparam int A = 24;
  localparam int B = 5;

   initial begin
     Clk_A <= 0;
     Clk_B <= 0;
     fork
       forever #A Clk_A = ~Clk_A;
       #(A/2) forever #B Clk_B = ~Clk_B;
     join_none
   end

   initial begin
                A_in = 0;
     #(A * 2);
     #(A+(A/2)) A_in = 1;
     #(A * 10)  A_in = 0;
  end

endmodule:testbench
```

```
//test4
module testbench
  (output var bit Clk_A,
   output var bit Clk_B,
   output var bit A_in);

  localparam int A = 24;
  localparam int B = 5;

   initial begin
     Clk_A <= 0;
     Clk_B <= 0;
     fork
                  forever #A Clk_A = ~Clk_A;
       #(A/2 + A) forever #B Clk_B = ~Clk_B;
     join_none
   end

   initial begin
                A_in = 0;
     #(A * 2);
     #(A+(A/2)) A_in = 1;
     #(A * 10)  A_in = 0;
  end
  endmodule:testbench
```

APPENDIX C

This section includes notes for the different property variations used when testing for this paper.

```
passes
  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |->
    ##1
    @(posedge Clk_B) $changed(B1_in) ##[2:3]
                    ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1


passes
  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |->
    @(posedge Clk_B)  $changed(B1_in) ##[2:3]
                     ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1


passes
  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
    @(posedge Clk_B)  $changed(B1_in) ##[2:3]
                     ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1

*****************************************************************

fails
  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=> TRUE
    ##1
    @(posedge Clk_B)  $changed(B1_in) ##[2:3]
                     ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1

(run0: dly=0
 Run1: dly=1)

//            prop1      prop 2
// test 1 -
//   run0     pass       pass
//   run1     pass       pass
//
// test 2 -
//   run0     fail       fail
```

```
//   run1     fail       pass
//
// test 3 -
//   run0     fail       fail
//   run1     fail       fail
//
// test 4 -
//   run0     fail       fail
//   run1     fail       fail


*****************************************************************

fails
  property CDC_prop1;
    logic v_temp;
    @(posedge Clk_A) ($changed(A_in), v_temp = A_in) |=>
                     ($changed(B1_in) && (B1_in === v_temp))
    ##1
    @(posedge Clk_B)  $changed(B1_in) ##[2:3]
                     ($changed(B_out) && (B_out === v_temp));
  endproperty:CDC_prop1

(run0: dly=0
 Run1: dly=1)

//            prop1      prop 2
// test 1 -
//   run0     pass       pass
//   run1     pass       pass
//
// test 2 -
//   run0     fail       fail
//   run1     fail       pass
//
// test 3 -
//   run0     fail       fail
//   run1     fail       fail
//
// test 4 -
//   run0     fail       fail
//   run1     fail       fail
```