Verification of Clock Domain Crossing Jitter and Metastability Tolerance using Emulation

Ashish Hari
Mentor Graphics Corp.
ashish_hari@mentor.com

Suresh Krishnamurthy
Mentor Graphics Corp.
k_suresh@mentor.com

Amit Jain
Mentor Graphics Corp.
amit_jain@mentor.com

Yogesh Badaya
Mentor Graphics Corp.
yogesh_badaya@mentor.com

ABSTRACT

Advances in technology are leading to the creation of complex SoCs. Design sizes are inching towards multi-billion gates and verification of these designs is already a big challenge. To add to the complexities of verification, these designs use power management strategies to reduce power consumption, plus they incorporate multiple IO interfaces, cores and peripherals. As a result, these designs have many asynchronous clocks.

Validation of designs with asynchronous clocks requires clock domain crossing (CDC) verification. CDC verification tools perform structural analysis of the design, which identifies the boundaries of the clock domain crossing signals. Designers add synchronizers at these boundaries whenever necessary. Synchronizers mitigate the impact of metastability on the design, but they do not guarantee that data output from the synchronizers are always correct.

In particular, CDC jitter occurs when an unpredictable delay is introduced at a synchronizer output and the receiving flop goes metastable. As a result, the flop's output might settle to a wrong value for a clock cycle. The design logic must tolerate unpredictable delays caused by CDC jitter. But, verifying design functionality in the presence of CDC jitter is tricky. Typically, simulators and formal verification engines do not model metastability completely and accurately.

Certain simulation and formal verification modeling techniques have been used to introduce metastability when checking the design for functional correctness. While they do a reasonable job modeling metastability and assisting verification, they slow simulation and formal verification significantly. So, they cannot be used with large designs.

In this paper, we propose a methodology to verify that a design tolerates CDC jitter. The methodology is based on emulation, so our metastability modeling does not compromise accuracy, performance or the debugging features of the verification tools.

INTRODUCTION

A *clock domain crossing* (CDC) occurs when a signal generated in one clock domain is sampled in another asynchronous clock domain. Here, the relationship between the *transmit clock* (the clock on which a value is generated) and the *receive clock* (the clock on which that value is sampled) is asynchronous. A receive register might experience setup and hold timing violations, in which case, the register could *go metastable*. To avoid propagating metastable values to downstream logic, such crossings should include CDC *synchronizers*.

Synchronizers reduce the probability of metastable values flowing into the design. However, having a synchronizer in a crossing does not guarantee that the synchronizer's output has a predictable value during any cycle that its receive register goes metastable. The synchronizer's output value is delayed by one cycle, is advanced by one cycle or is correct. Functional verification of a design must verify that the design behaves correctly in the presence of unpredictable synchronizer output values.

Figure 1 shows how simulation behavior can differ from silicon behavior. Simulation behavior is predictable, whereas silicon behavior can be unpredictable when setup/hold timing constraints are not satisfied. Verifying design tolerance in the presence of this unpredictability is crucial before moving to silicon.

In the following sections we first present the existing simulation and formal methods of verifying design tolerance of metastability effects. We show the remaining challenges of using these methods. Then, we present our metastability model and emulation-based verification methodology that overcomes the limitations of the existing methods.

HIGH-LEVEL REQUIREMENTS FOR VERIFYING METASTABILITY TOLERANCE

Conventional simulation does not model metastability effects. To verify a design's tolerance of metastability affects, a verification methodology:

- Must have a logic model that intelligently accounts for metastability effects at any register.
- Must use these models at all appropriate points in the design.
- Must include a debug capability to help debug functional failures in the presence of many injections from the metastability models.

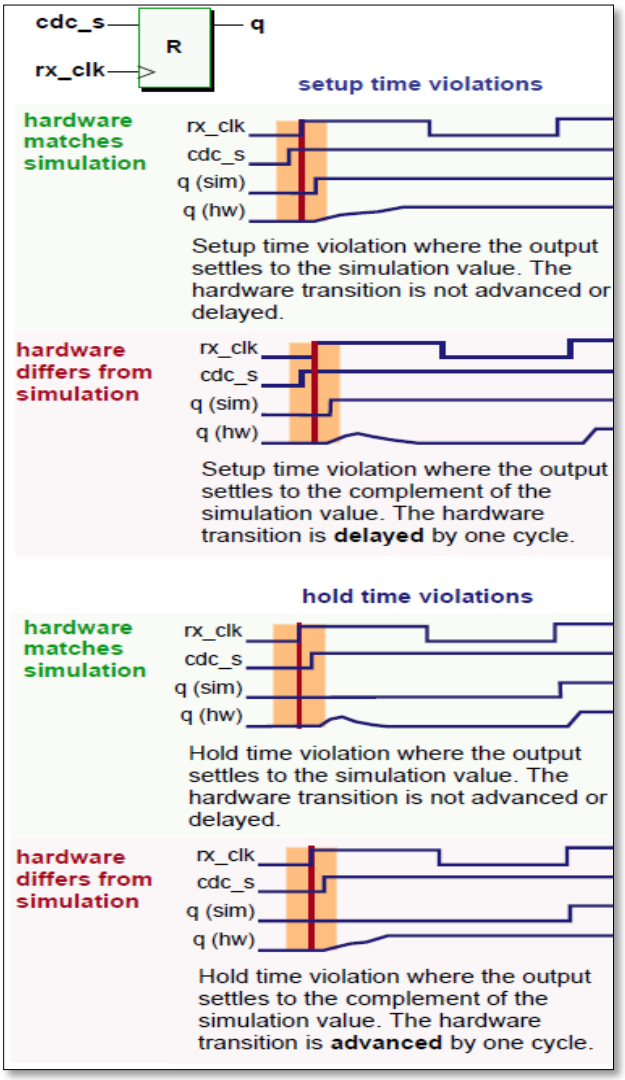

***Figure 1*: Simulation behavior can differ from silicon behavior when setup/hold violations occur**

Requirements for the Model

To model metastability in simulation, a metastability injection model must do the following.

- Introduce random delays

  As shown in *Figure 1*, when a register is metastable, its output is delayed by one cycle, is advanced by one cycle or is correct. The metastability injection model must be able to introduce this behavior randomly when sampling an asynchronously-clocked register.

- Model metastability independently

  In silicon, metastability might occur at any register that samples values from an asynchronous clock domain. Furthermore, the metastable behavior of a register is independent of other registers' behaviors. For example, if the same value is sampled at two different registers, the registers' metastability effects must be modeled independently. In particular, the metastability injection model must independently introduce metastability at each bit of a clock domain crossing's receive register.

- Model metastability accurately

  In silicon, metastability occurs only if: 1) the data value changes, 2) the data value is sampled and 3) the setup/hold time constraints are violated. The metastability injection model must introduce metastability only when these events happen. A model that randomly injects metastability effects at any time point does not accurately model metastability effects in silicon.

- Model metastability completely

  A register that samples values from another clock domain can experience metastability whether or not the crossing includes a synchronizer and regardless of the synchronizer type. To model metastability completely, a metastability injection model must cover these situations.

Requirements for the Methodology

In addition to the metastability injection model, a modeling methodology is required to apply these models to the appropriate points in the design. The model should be convenient and easy-to-use, for example:

- The model can be integrated seamlessly into the design under simulation. Design instrumentation is minimal and does not modify design files.
- The model is instrumented at all clock domain crossings. Metastability can be injected independently at every CDC path.
- Each instance of the model can be turned on and off independently—to ensure flexibility.

Requirements for the Debugging Environment

Debugging functional errors resulting from metastability injection is difficult. So, complex metastability models not only inject metastability, but they also collect debug information. For example, debug details might include information about the metastability injection cycles. Additional controls such as adjusting the metastability window and tuning the model parameters reduce false failures and assist in effective debugging.

CURRENT MODELS AND METHODOLOGY

In this section, we present some common metastability injection models and methods.

Clock Jitter Model

A *clock jitter model* uses jitter in clocks to inject random delay in the design. The model changes the locations of clock edges randomly—and therefore causes values to be sampled as advanced, delayed or normal values—which means the model satisfies the random delay requirement for a metastability injection model. Two types of clock jitter models are:

1. Clock jitter at primary clocks

   Random jitter can be introduced at the primary clocks (*Figure 2a*). Such a model introduces minor changes and has little impact on the design. But, this approach does not model metastability independently at each receive register.

2. Clock jitter at synchronizer clocks

   A more accurate approach models metastability independently by introducing jitter on each receiving register clock (*Figure 2b*). For crossings with synchronizers, jitter is applied to clocks internal to the synchronizers. Such a model is not easy to implement for unsynchronized crossings and cases where the synchronizer is not a predefined custom cell (i.e., a *custom synchronizer*).

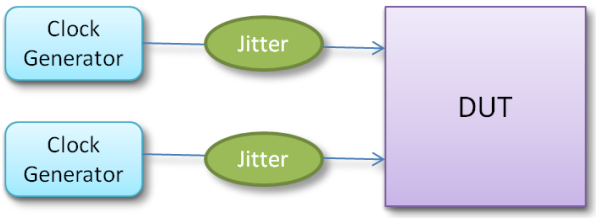

***Figure 2a*: Clock jitter at primary clocks**

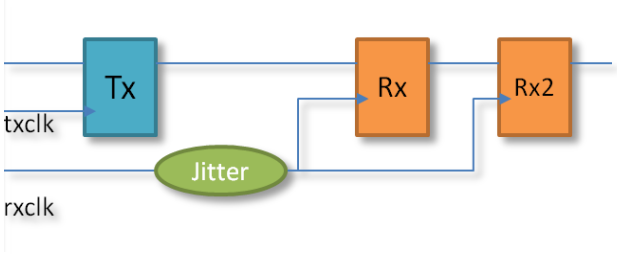

***Figure 2b*: Clock jitter at a synchronizer clock**

3-Flop Model

The *3-flop model* replaces 2-flop synchronizers in the design by 3-flop cells that randomly generate advanced, delayed or normal output values (*Figure 3a*). This model can be used for specific 2-flop synchronizers, but typically not for all crossings where a model is required.

A critical issue is that this 3-flop model can generate an output sequence that is impossible in silicon: values might get skewed by 2 clock cycles. Some methodologies modify the 3-flop model to only delay values by 1 cycle (*Figure 3b*). However, this approach cannot model the effects of advancing cycles (which can happen for hold violations).

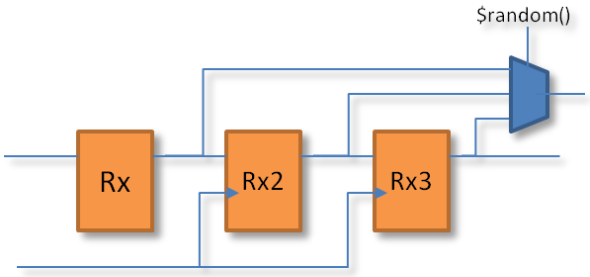

***Figure 3a*: 3-flop model for setup/hold violations**

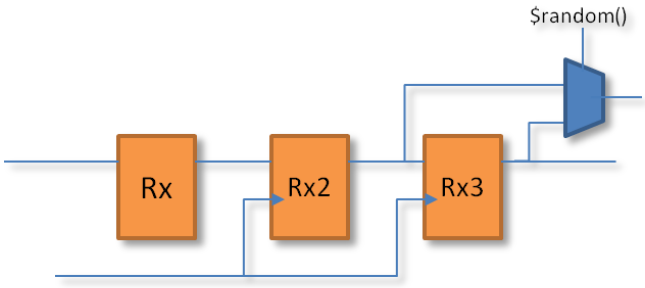

***Figure 3b*: 3-flop model for setup violations only**

Delayed 2-Register Model

Another approach delays the input of a 2-flop synchronizer randomly (*Figure 4*). As for the 3-flop model, this model cannot replicate advance-cycle effects and the model only works for clock domain crossings synchronized by 2-flop synchronization.

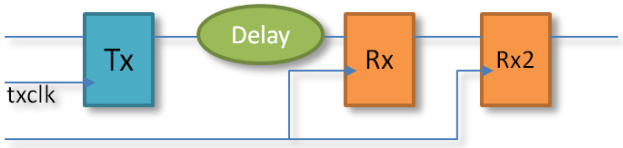

***Figure 4*: Delayed 2-register CDC jitter model**

Accurate Metastability Model

The *accurate metastability model* shown in *Figure 5* is the model we use throughout the rest of this paper. This model handles metastability effects in the most accurate way. It satisfies our requirements for a good metastability model. In particular, the accurate metastability model:

- Decides when to inject metastability

  The model injects metastability only if: 1) transmitting and receiving clocks are aligned enough to violate setup/hold time constraints; 2) the sampled data are

changing; and 3) the register is actually sampling the data.

- Decides what value to inject

  The model randomly injects values with and without a delay. Since values are injected at the outputs of the receive registers, the model produces both advance and delay effects.

- Satisfies our model requirements

  The model works directly on receive registers, so it can be used for unsynchronized crossings as well as synchronized crossings. The synchronized crossings can be synchronized by any method without impacting the model's functionality. The model can be applied at each point independently of the other injection points.

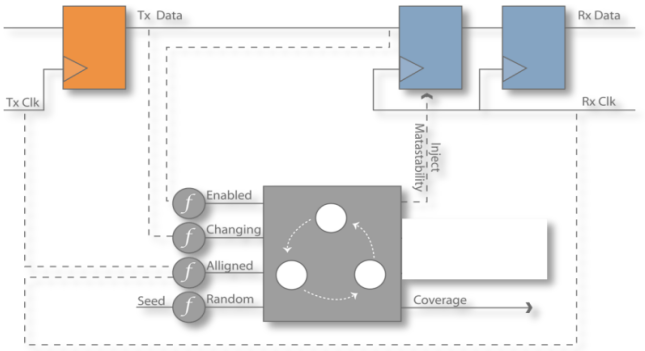

***Figure 5*: Accurate metastability model**

METHODOLOGY IN A TRADITIONAL FRAMEWORK

The models and methods described in the previous section are *simulation based*. That is, a design is modified to appropriately use the models and then the modified design is verified with traditional simulation.

In this section, we present our methodology within this traditional framework. However, our flow uses our version of the accurate metastability model, which is *formal friendly* (i.e. the model is compatible with formal methods). For this paper, we present a simulation-based flow. But a parallel, formal-based verification flow works as well.

Verification Flow

Our verification flow has the following steps based on the flow shown in *Figure 6*:

1. **Analyze the design's CDC structure**

   Structural CDC analysis identifies all CDC signals in the design. For each CDC signal, structural CDC analysis identifies whether the crossing has a good, a bad or a missing synchronizer.

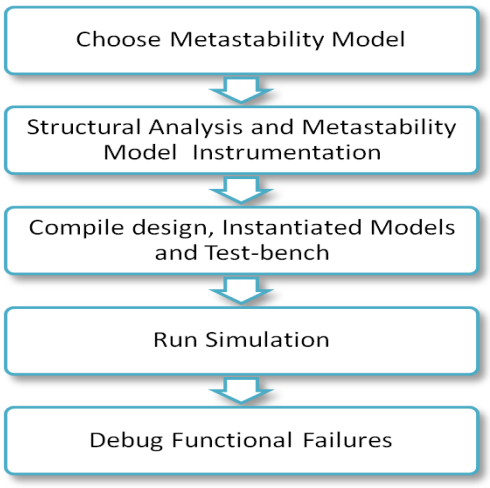

***Figure 6*: Simulation-based CDC-jitter verification flow**

2. **Instrument the design with instances of the metastability models**

   Use the accurate metastability model described previously. Using the results of structural CDC analysis, identify all points where metastability can occur. Connect an instance of the metastability model to the logic surrounding the receive register.

   This design instrumentation uses SystemVerilog *bind* constructs generated in separate files that can be simulated with the original design files (without the need for altering the original design). A Verilog *force* statement injects value based on the metastability algorithm.

3. **Simulate the design with the model instances**

   Run the standard design verification flow with the files created to instrument the design.

   Each instance of our model can be independently enabled and disabled, so each injection point is separately controlled. In particular, metastability injection can be controlled across the entire design.

4. **Debug and resolve any functional issues**

   If functional errors occur because of metastability injections, a robust debug environment is crucial for identifying their associated causes. Debugging such issues is difficult because the source of a problem could propagate through many metastability injection points before being detected as a test error. Discovering which injection point or which multiple injection points caused the failure can be quite tricky.

   Since the models are defined in standard SystemVerilog, a designer can use any preferred debug tool.

Our model collects information to help designers debug issues resulting from metastability injection, including the following:

- Details on the points where metastability caused an injection of delay or advance.
- Details on setup/hold violations and information about data changing or being sampled during this period.

Our model also controls metastability injection in the following ways:

- Metastability injection can be independently enabled and disabled to ease debugging.
- Metastability windows can be controlled: the setup/hold time conditions can be specified separately for each model instance.

Limitations of the Flow

The above verification flow works well for simulation (and for formal verification). But, CDC verification must be run on the whole design. So, the methodology has limitations for large designs:

- Introducing metastability models in a design degrades standard simulation performance by 3X to 4X. This is okay for smaller designs—but for larger and larger designs, this slow-down makes the traditional flow impractical.
- Because of simulation performance and capacity limitations, simulation with metastability injection of large designs can be done only for small testbenches. Or, only a small number of cycles can be verified,
- Formal methods work well only for small, block-level designs—not for the typical system-level designs.

PROPOSED VERIFICATION FLOW BASED ON EMULATION

Emulation-based verification is typically used for very large designs because it provides a significant run-time performance gain—often 1000X—over simulation-based flows. In particular, using emulation might overcome the limitations of our simulation based flow. Here, we propose modifications to our simulation-based flow to support metastability tolerance verification on an emulator.

Emulation-friendly Metastability Injection Model

For this work, we chose the accurate metastability model and targeted virtual emulation environments based on the SCE-MI 2.0 standard, which is supported by all emulation vendors.

When to inject metastability

SCE-MI 2.0 compliant emulation platforms provide full simulation interoperability. They support simulation style clock generation on the emulator—including accurately tracking model simulation time inside the emulator. In particular, these emulators provide *clock proximity detection* based on simulation time.

Clock proximity detection is a key feature of the accurate metastability model. It first measures the distance in model time between transmit clock arrival and receive clock arrival. It then compares these times with the appropriate setup/hold time parameters and determines whether or not the edges are too close. Clock proximity detection does require arithmetic logic involving large values—but this logic is shared by CDC points that have a common transmit-receive clock pair. So, the impact of adding this logic is minimized.

Other checks for determining when to inject metastability are whether or not data are changing and are sampled. We borrowed these straightforward checks from the simulation model.

Randomizing metastability values

When the model encounters a metastability point, it must randomly decide whether to let the data pass as is, or to invert it. This decision is random in multiple dimensions:

- For each transmit-receive clock pair, the associated CDC points should show random (i.e., un-correlated) metastability decisions.
- For a specific receive register, random metastability behavior should be observed over time as well as over different types of metastability violations (such as setup violations, hold violations, violations when data are rising and violations when data falling).

Simulation uses a *$random* call to make this decision. In emulation, *$random* can be implemented as a stream of random values coming in from the workstation. But, this implementation has a high cost in terms of the logic needed to distribute the random values to different CDC points, and in the extra runtime needed to transfer the random values. Although these problems are not insurmountable, the present work has chosen to use an *LFSR-based random sequence generator*, which solves the problem within the emulator.

The generator uses simple LFSR logic to model an appropriate number of bits that are shared. Each CDC point sources a specific bit in the sequence. The bit distribution is static and is chosen randomly. The number of bits in the LFSR sequence is increased based on the maximum number of CDC points within a transmit-receive boundary. Different transmit-receive boundaries cause metastability at different points in time, so they can share LFSR bits.

How to inject metastability

Most emulation environments support *force/release* mechanisms that alter design registers to put the design into a state that is interesting from a verification standpoint. The present work uses this mechanism to modify receive registers where necessary to inject values.

Modified Verification Flow

The CDC analysis and metastability model insertion process is largely the same for both simulation and emulation. The present work uses these same tools. However, existing tools in this area were designed with simulation environments in mind. So, such tools most likely must be re-architected to scale up to handle very large emulation-sized designs.

That said, the basic principles behind how metastability insertion works is not different. The design is fed to the native emulation compile steps. The flow uses our emulation-friendly metastability injection model and beyond that point, all use-model steps—how to run the emulator, how to debug and so on—follow those provided by the native emulation environment (*Figure 7*).

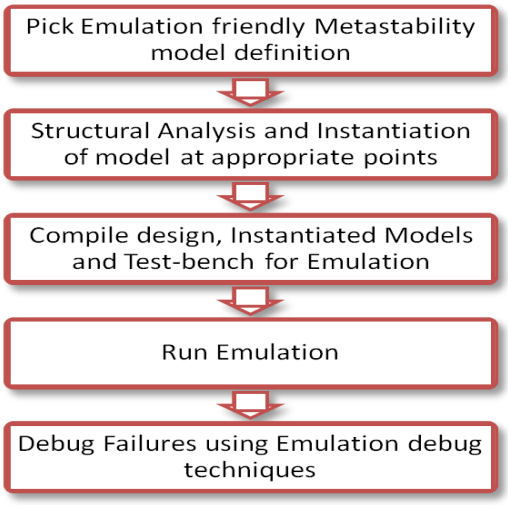

***Figure 7*: Emulation-based verification flow**

Debug Methodology

Functional problems caused by CDC metastability injection exhibit themselves in a manner no different from other functional bugs in the RTL. All of the debugging techniques of the native emulation platform are available.

Today's emulators support sophisticated debug methodologies. Almost all of the powerful simulation-like debug features are available with the virtual emulation environments (especially the SCE-MI 2.0 compliant emulators). These features include: model simulation time recognition for interactive run control and waveform display, full trace of waveforms, time- or space-selective trace of waveforms, waveform viewers, source code editors, path browsers and schematics browsers.

Emulation has its own traditional debug techniques which are useful for large designs. For example, emulators can run large numbers of cycles under trigger-based debug, which allows the on-the-fly creation of trigger state machines to look for complex sequences of events in the design execution.

Modern emulation environments also support some of the newer verification paradigms, such as assertion-based verification. Here, functional assertion points are compiled into the emulator as part of the model. Violations are reported immediately—just as with simulation. These assertions can be collected in a log file for post-process analysis or emulation can stop at the violation. Waveforms of assertion failures can be captured and debugged.

Assertion-based capabilities are especially relevant to the current work. Problems introduced by CDC injections can show up a long time after injection and in a different part of the design. Assertions used effectively can cause violations to manifest sooner and much closer to the point of the CDC injection, which makes debug easier and more productive.

The emulation solution supports the enabling and disabling of individual metastability injection points. In addition, virtual emulation is repeatable and is much faster than simulation methods. These features support an effective *elimination strategy* that narrows analysis down to specific culprit CDC effects. Once a problem is detected, CDC injection points are disabled based on certain classifications (such as injection points that are not in specified design scopes and injection points that are outside of specific transmit-receive clock pairs). Then, emulation tests are repeated to see whether or not the problem persists. This strategy narrows the possible candidate CDC points that caused the problem, which reduces the problem's degrees of freedom during debug.

Summary of Benefits

- Emulation handles very large design capacities (on the order of a billion gates). It is more effective for CDC verification of complete chips.
- Emulation is much faster than simulation. It can run much longer simulations, which increases the possibility of exposing more CDC problems.
- Emulation can stimulate the design with realistic stimulus traffic, so CDC verification can model relevant activity.
- Emulation can connect to more realistic environments (such as static in-circuit targets). Some emulators also can capture real driver traffic for some standard interfaces (Ethernet, PCI, and so on) and feed it to models on the emulator.

- Emulation continues to extend itself to support advanced verification methodologies, including assertion-based coverage, SystemVerilog UVM/OVM and SystemC TLM. These new methodologies support CDC verification in a variety of emulation environments.

BENCHMARK DATA AND RESULTS

Experiments are done on two designs with the following characteristics.

	Design 1	Design 2
Crossing	1100	6071
Design Flops	17603	66724
Number of Clocks	8	38

Following are the results for the designs. Both designs exhibit about 1000X performance improvement when compared with simulation.

	Design 1	Design 2
Performance Gain (compared against s/w simulation)	1050X	950X
Area penalty (compared against Emulation area requirement without CDC metastability Instrumentation)	9%	12%

Emulation has a proven track record for maintaining the performance levels even as design sizes grow, whereas simulation performance suffers significantly from design size growth.

We ran these designs as a proof-of-concept for our methodology. They do indeed illustrate key advantages of our methodology. CDC verification can be performed on emulation and with significant performance gains! The results also show that CDC metastability verification has an adverse impact on simulation performance—even for small designs. For this reason, we observed very good performance gains even on our small experimental designs.

LIMITATIONS

Emulation-based methodology can work only with emulation-friendly verification environments. Those environments that use behavioral models for testbenches or use non-RTL models inside the DUT are not suitable for emulation.

Our solution does not work if the emulation environment involves traditional dynamic in-circuit targets.

Emulation also does not support some elaborate statistical information—such as which CDC points were hit and which types of violations occurred—that is supported by our simulation model.

CONCLUSION AND FUTURE WORK

Ever-increasing design sizes and shrinking verification schedules are making emulation technology more popular. Emulation technologies are increasingly easier to use and they provide the capability to enter the verification flow earlier in the development cycle. CDC verification using emulation is a viable option that drastically reduces verification time and leads to comprehensive CDC robustness.

Emulation allows CDC verification to be executed in a context closer to real systems. Emulation-based CDC verification can cover the entire SoC; it can apply realistic stimulus; and it can run for a sufficiently long time. At the same time, it provides the ability to effectively debug functional errors.

Future work in this area is to evolve our emulation-based CDC verification paradigm as emulation technologies and methodologies themselves evolve. We want to merge the CDC analysis and metastability insertion phases with the emulation clock analysis phase. This would make the modeling methodology naturally scalable to handle design sizes encountered by emulation methodologies. Plus, more work is needed to reduce the impact of CDC instrumentation logic on capacity.

REFERENCES

[1] Tai Ly, Neil Hand, Chris Ka-Kei Kwok, *Formally verifying clock domain crossing jitter using assertion-based verification.*

[2] Chris Kwok, et al, *"Using Assertion-Based Verification to Verify Clock Domain Crossing Signals"*, DVCon, February 2003

[3] Mark Litterick, "*Pragmatic Simulation-Based Verification of Clock Domain Crossing and Jitter using System Verilog Assertions",* DVCon, 2006

[4] Alex Genusov, et al, *"Verification of Skew and Jitter Tolerance and Compensation in High-Speed Interfaces",* DVCon, February 2003

[5] A. Saha, K. Suresh, A. Jain, V. Kulshrestha, S. Gupta, "An Acceleratable OVM Methodology Based on SCE-MI 2," DVCon, 2008

[6] H. van der Schoot, J. Bergeron, "Transaction-Level Functional Coverage in SystemVerilog," DVCon, 2006

[7] Hans Van der Schoot, et al, "Off To The Races With Your Accelerated SystemVerilog *Testbench"* , DVCon, 2011