Comprehensive and Automated Static Tool Based Strategies for the Detection and Resolution of Reset Domain Crossings

Yossi Mirsky B.Sc, M.Sc, MBA
Intel Corporation (joseph.mirsky@intel.com)
9 Hamarpe Street, Jerusalem, Israel, 977740

***Abstract*- Unlike CDC, there is little awareness in the industry of the dangers of Reset Domain Crossings (RDC). Inside the same clock domain with two asynchronous resets, the asynchronous assertion of the transmit FF's reset can generate data changes in the receive reset domain that violate setup and hold timing considerations in addition to inducing metastability. RDC issues are not covered in standard verification flows, and create random, illusive silicon bugs which are difficult to detect and diagnose in the lab. A simple static tool-based methodology for detecting RDC is presented, and strategies for avoiding or resolving RDC issues are explored.**

I. INTRODUCTION

In today's complex, multiple clock system-on-chip (SoC) designs, the dangers of unsynchronized data crossings between clock domains are well known in the industry. Clock domains are defined as a collection of sequential elements driven by a clock with the same frequency and phase. Data that cross between asynchronous clock domains may arrive in the receive domain close to the rising edge of the receive domain's clock, violating setup and hold timing requirements. This is known as a clock domain crossing (CDC) and there is a statistically significant possibility that such crossings can drive the receive domain's flip-flops (FF) into a metastable state, potentially producing functional bugs and chip failures. Metastability refers to the state where a sequential element's output remains in an undefined logic value state between '0' and '1' for an unknown amount of time before settling to a defined logic value. As a result, clock domain crossing tools have become a standard part of front-end verification flows throughout the industry, and a plethora of synchronization schemes and strategies have been developed to either prevent or address CDCs [1].

However, there is much less awareness in the industry that metastability can also be induced inside the same clock domain by employing multiple asynchronous resets. This problem is known as reset domain crossings (RDC) [2], and as a design's size, complexity, and the number of asynchronous resets grow (Intellectual Properties (IP), power domains, etc.), the chances of encountering an RDC bug in the silicon increases exponentially. Worse, due to their random nature, they can create insidious chip failing bugs that are hard to detect, reproduce, and debug in the lab.

Due to several RDC bugs discovered in our department's silicon, strategies for addressing RDC issues were developed and an automated verification flow based on an industry standard tool was deployed[1]. The purpose of this paper is to impress upon the reader the critical importance of checking for RDC issues in today's SOCs. This paper will first describe in depth what RDC bugs are and why they are so dangerous. Then strategies and methods for mitigating or avoiding RDC issues will be explored. An automated tool flow and methodology for using static checks at the RTL stage to detect and resolve RDC issues [3] will be described and the paper will conclude with a report of real world user experience using such an RDC flow in our department.

[1] Due to Intel corporate policy, I am prohibited from expressly mentioning the tool vendor.

II. Reset Domain Crossings

A. *Asynchronous Resets*

Before explaining RDC's it is important to briefly review reset implementation schemes and define "asynchronous resets" for this paper. In modern designs today, there are two broad categories of resets: synchronous and asynchronous. The behavior, implementation, advantages, disadvantages and use cases for each has been addressed in depth in other articles [4]. In brief, synchronous resets are only asserted or de–asserted synchronously with the clock of the flip-flop being affected by the reset. As such, a synchronous reset will only affect the FF's state on the active edge of the clock, and static timing analysis will take these requirements into consideration when building the reset tree. On the other hand, asynchronous resets can be asserted at any point during the clock cycle, regardless of setup and hold timing considerations, inducing the receiving flip-flops to immediately output their reset value (generally '0'). Since async (asynchronous) resets arrive asynchronously to the clock it is not necessary to balance the reset tree, which can provide a tremendous advantage in terms of back-end implementation complexity, area, power, and timing considerations. As a result, asynchronous resets have become the preferred and common choice for reset implementation.

However, while async resets can be asserted asynchronously, it is imperative that the de-assertion reach sequential elements in a synchronous fashion. This is crucial for two reasons: Firstly, due to variable propagation delay, an asynchronous reset de-asserted close to the rising clock edge might result in sequential elements resuming function and sampling data in different clock cycles. Secondly, even worse, an asynchronous de-assertion may violate reset recovery time, which is the minimum amount of time required between reset de-assertion and the next rising clock edge. The FF will not have enough time to properly capture/sample data before the next rising clock edge, risking putting the output of the FF into a metastable state.

The two concerns above are not relevant during reset assertion, as in any case the designer's assumption is that the reset FF's outputs are no longer functionally active/relevant. The reset will be asserted for multiple clock cycles, and therefore temporary metastable values can cause no harm.

As a result, failing to ensure synchronous de-assertion of an asynchronous reset can wreak havoc on the design and introduce major functional bugs. Instead of exiting the reset state clean and reinitialized, the asynchronous de-assertion can introduce unpredictable clock cycle behavior and metastable values. As a result, it has become common practice to use reset synchronizers such as the one illustrated in Fig. 1 to ensure asynchronous reset de-assertions arrive at the affected FFs in a synchronous fashion. As such, in this paper when referring to the term async reset, nonetheless it is implied that the reset de-asserts synchronously.

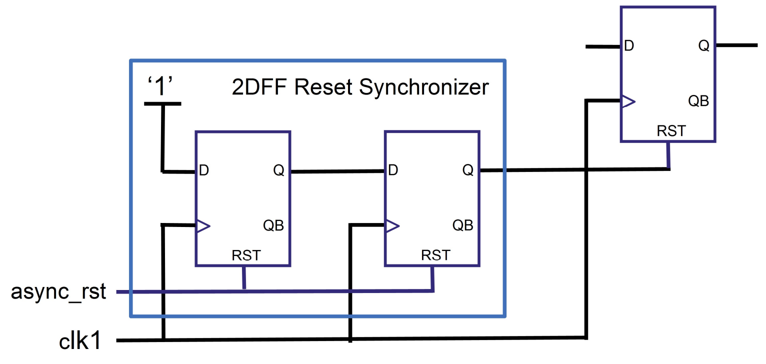

Figure 1. Reset Synchronizer.

B. *Reset Domain Crossings*

In the past, many chips had only one reset to reinitialize the design. However, today's complex designs and SoCs demand multiple resets to target specific IPs and power domains in order to minimize power. Furthermore, demand for improved functionality has increased the number of resets to enable special software and firmware level resets. As mentioned above, due to area, timing and power considerations beyond the scope of this article, most resets used in modern designs are asynchronous resets.

This prevalent use of numerous asynchronous resets lays the foundation for a new class of bugs known as RDC [4]. Similar to CDC's clock domains, we can divide the design into reset domains, where each domain can be reset independently from those of other reset domains. Connectivity between each reset domain permits the transmission of data, hence the term reset domain crossing. It is these crossings which make it possible for an asynchronous reset asserted in one reset domain to create asynchronous data changes in a receiving reset domain. Similar to CDCs, since such changes occur without any consideration for setup and hold time requirements, they can potentially induce the FFs in the receive reset domain into metastability.

Based on the simple RDC schematic in Fig. 2a, the asynchronous assertion of reset rst1 will drive low the logic values on the output q1 and the input d2 in an asynchronous fashion. If concurrently rst2 is in a de-asserted state, this asynchronous datum change may violate the setup and hold timing conditions, and FF2 may enter a metastable state (as seen in clock cycle 6 of the waveform in Fig. 2b). The red arrow in Fig. 2a shows the timing path from rst1 to d2 that static timing analysis (STA) will not take into account and thus may be violated. Consequently, RDCs can induce metastability downstream in the sequential elements of the destination reset domain with the same potentially serious consequences as those seen in clock domain crossings or asynchronous reset de-assertions. Note, the reset de-assertion in clock cycle 2 occurs well before the next rising clock edge, and therefore does not create metastability.

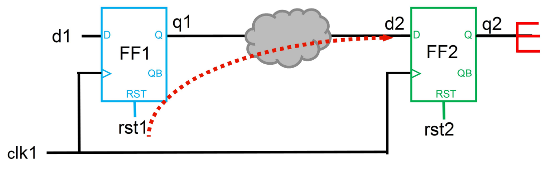

Figure 2a. Simple RDC.

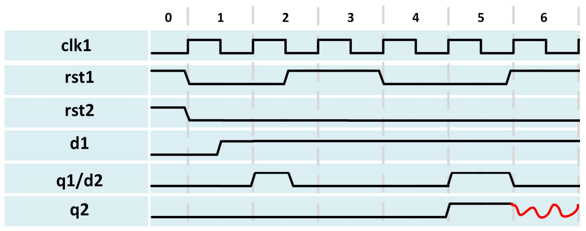

Figure 2b. Waveform of RDC metastability.

The asynchronous datum change is particularly dangerous in cases where FF2 has a fanout greater than one, since the metastable value will propagate to multiple sequential elements and eventually, unpredictably, settle into a random pattern of ones and zeros. This has serious potential for corrupting state machines and causing the design to enter illegal states that it was not designed for.

It is important to note that the potential for reset domain crossings bugs remains constant regardless of whether or not the source and destination reset domains are in the same clock domain. It is also worth noting that typically, FFs have a reset value of '0'. RDC's can only create metastability if the asynchronous reset assertion results in a change in logic value at d1; this is only possible if FF1's output was high ('1') just prior to the assertion of rst1. If the output was low ('0'), the reset assertion will not cause any data change in the receive reset domain. In certain cases, this characteristic can be utilized to tailor specific synchronization schemes or design adjustments to prevent RDC issues.

C. RDC versus CDC

In RDCs, the asynchronous change in value will only occasionally violate setup and hold, and even then will not always cause functional bugs due to metastability. Similar to CDCs, the random and sporadic nature of metastability propagation and settling makes RDC bugs incredibly elusive and insidious. They are incredibly difficult to catch, diagnose or reproduce in the lab during silicon validation (SV), and the bug may sneak past SV only to be discovered by a customer.

There are fundamental similarities between the potential design bugs due to either CDCs or RDCs. In both cases, one must divide the design into domains (clock/reset) that are asynchronous to one another, and an asynchronous event (clock change/reset assertion) in the transmit domain can induce an asynchronous data change in the receive domain that violates setup and hold timing considerations.

Both CDCs and RDC require specialized tools/flows to detect and clean design from possible issues. Similar to CDCs, attempts to properly manually detect RDCs through a code review is practically impossible and simulations involving x-injection, formal assertions, etc., are neither scalable nor robust enough to provide full coverage for an entire design. Static timing analysis is also ineffective as such tools can only address timing requirements for a predetermined clock cycle and length, and are unable to take into account asynchronous changes (without producing a tremendous amount of noise and false violations). Therefore, an automated static tool based approach to RDCs is required.

CDC tools themselves fail entirely to address RDC issues as they only examine the crossings between asynchronous clock domains, ignoring reset domains. If the crossing between the two reset domains coincides with the crossing between two clock domains, then in all likelihood standard CDC static checks will flag this crossing. Therefore, the greatest danger of RDCs lies specifically in cases where both reset domains are in the *same* clock domain, since current industry standard CDC tools and verification practices have no methodology for catching these issues. RDCs and CDCs tools also differ as new categories of constraints are required in RDC tools to clean false warnings (such a defining the order of reset assertions and dependencies).

III. EXAMPLES OF RESET DOMAIN CROSSINGS

A. *Asynchronous Clock Domains*

Fig. 2a above illustrated the simplest and most common RDC structure. However, it is important to note that there are many different complex and varied reset domain crossing schemes, a few of which will be discussed below as instructive examples. As previously mentioned, RDCs can overlap with CDCs where different transmit and receive reset domains can correspond with different asynchronous clock domains as well. For example, in Fig. 3, FF1 is both in clock (clk) domain clk1, and reset domain async rst1, whereas FF2 is in clock domain clk2 and reset domain async rst2. While in most cases this crossing will be flagged in CDC tools as a CDC issue, it is not safe to rely on this, as it may be legitimately waived from a CDC perspective despite remaining a serious RDC bug. As a result, it is important that these clock domain crossing RDC structures be searched for in RDC tools as well.

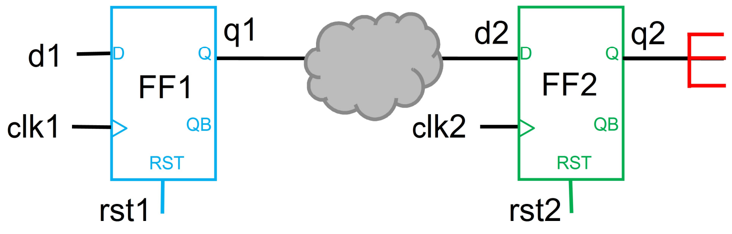

Figure 3. RDC in 2 clock domains.

B. *No Receive Reset Domain*

In some cases, the receive FF may be at risk of entering a metastable state even if it is not part of a true reset domain. For example, in the schematic in Fig. 4, a sequential element in the receive domain either does not possess a reset pin or its clear/reset pins are undriven. As a result, FF2 is essentially always on, and can never be reset or initialized. This is functionally analogous to a sequential element in a receive reset domain whose reset is currently de-assertStied; therefore, FF2 is still equally vulnerable to asynchronous datum changes created by the assertion of async rst1.

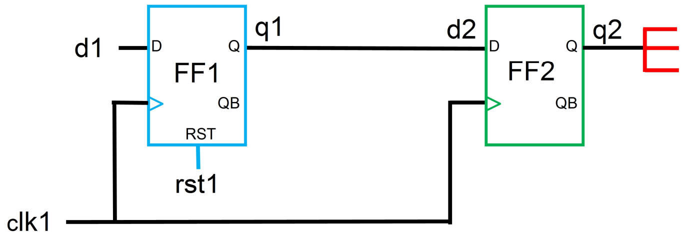

Figure 4. Receive domain with no set/reset.

C. *Same Clock and Reset Domain*

RDC issues can arise even if the source and receive sequential elements are both in the same clock domain *and* the same reset domain. Such a scenario exists if the reset of the receive FF is affected by another signal known as the side-input shown in Fig. 5. If the side_input signal is driven by a different, asynchronous clock or reset domain, then any trigger from side_input's source domain which drives the signal low can potentially lead to RDC bugs.

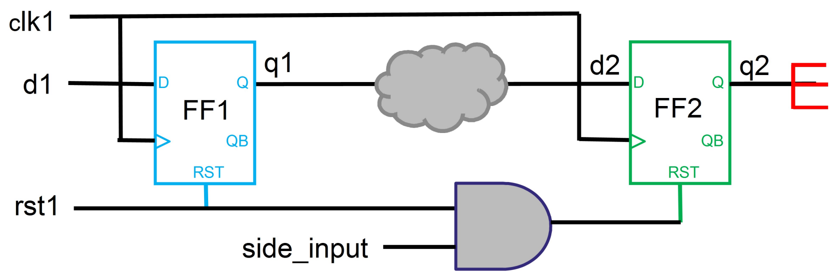

Figure 5. Side-input signal to the same reset and clock domain.

D. *Multiple Resets*

When defining clock domains for CDC tools, each domain's sequential elements are associated with a single particular clock frequency and/or phase. If the design possesses multiple functional modes which require modelling different clocks for a particular clock domain, then multiple CDC runs are required. However, such clear and clean definitions of associating one reset per sequential element are not possible regarding reset domains. It is both a common and legitimate coding/design practice for the conditions of driving a FF's reset pin high to be dependent on the concurrent assertion or de-assertion of several async resets. This can frustrate any attempts to divide the design into simple, unique, non-overlapping reset domains and greatly complicates the RDC analysis.

Fig. 6 illustrates an example of such a case. Assuming an active high reset, each flip-flop, FF1 and FF2, may receive '1' on their RST pins based on the output of combinational logic from two independent async resets. Before RDC checks can be performed, the tool must analyze the RDC structure for all possible reset assertion scenarios, and this is done by pairing all combinations of transmit and receive reset domains for consideration. Then each pair can be analyzed for the presence of RDC issues.

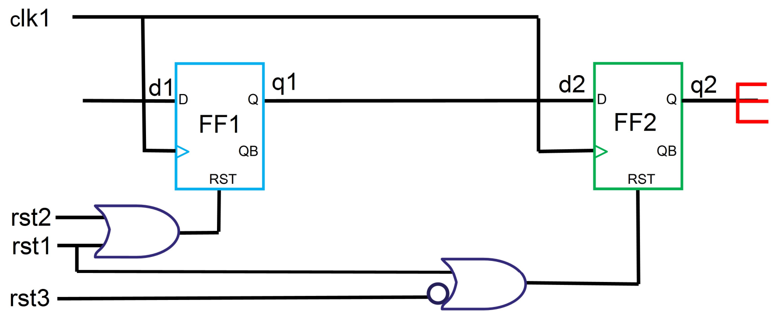

Figure 6. Multiple, dependent async resets.

Based on the schematic in Fig. 6, listed in Table 1 are four reset assertion scenarios involving 3 async resets: rst1, rst2, and rst3. In the first scenario, rst1 is both the transmit and the receive reset domain. In this case, there is no possibility of an RDC issue; once rst1 is asserted (high) FF2 will be reset regardless of the state of the other 2 resets. Similarly, in the second scenario where rst1 is the transmit reset domain and rst3 is the receive reset domain, the assertion of rst1 will ensure that FF2 is not active. In the third and fourth scenarios, rst2 is the transmit domain and rst1 or rst3 (respectively) are the receive domains. RDC tools will identify that if rst2 was low and then asynchronously goes high, then there is a potential for RDC issues in the third scenario (if rst1 at the time was low) and in the fourth scenario (if rst3 at the time was high).

TABLE 1
MULTIPLE RESETS

Source Domain Reset	Receive Domain Reset	Potential RDC Issue?
rst1	rst1	No
rst1	rst3	No
rst2 (‘0’→’1’)	rst1 (‘0’)	Yes
rst2 (‘0’→’1’)	rst3 (‘1’)	Yes

This simple example demonstrates how multiple reset dependencies can add great complexity to RDC analysis. As a result, dynamic verification methods such as simulation or formal approaches are ineffective because it is very difficult to achieve 100% coverage of all possible reset combinations and scenarios. Therefore, static tools have an advantage given that the user can be certain that no potential RDC crossings were missed. The disadvantage is that simple static RDC checks tend to produce a lot of noise; this issue will be addressed in the next section.

IV. MITIGATING AND AVOIDING RDC ISSUES

The best way to resolve RDC issues is to avoid them in the first place by using synchronous resets. RDC tools will not report RDC warnings when the source reset domain is synchronous. However, as mentioned above, often in today’s modern designs this is not a realistic option. While static based RDC tools are quick, easy to set up, and simple to use and debug, the main disadvantage is that large designs/SoCs may contain millions of reset domain crossings between sequential elements. This section will present alternative design methodologies, architectural modifications, specialized constraints and dynamic waiver generation to resolve or avoid RDC issues. The main purpose of the strategies outlined in this section are to leverage the RDC tool’s capabilities to quickly and effectively remove the severe noise from numerous false RDC violations so users can focus on real bugs. The solutions will progress from the “lighter” suggestions which will entail constraints or minimal design changes, before progressing to major design/architectural interventions and finally, as a last resort, waivers.

A. Same Reset Domain

Similar to CDCs, different resets can be defined as sharing the same reset domain. This will enable RDC tools to remove many false violations, without the need for the overhead of adding synchronizers and qualifiers into the design. Modern RDC tools provide specialized constraints to enable users to define resets that are part of the same reset domain. Therefore, by far the best methodology for reducing the impact of RDC issues is to plan the chip from the outset with as few reset domains as possible. The most basic definition of reset domain has been defined in the past to include the following four characteristics [5]:

1) *Type - Synchronous or asynchronous to the domains clock.*
2) *Polarity – active high or active low.*
3) *FF reset value – set ‘1’, reset ‘0’.*
4) *Signal – primary top reset name.*

However, in modern RDC tools, the definition of the same reset domain has been expanded to include three more categories. These categories are generally created by functionality intentionally implemented in the design, and not by reset signals driven by the same hardcoded reset pin from the top.

1) *If two or more resets are, by design, always asserted together, then they can be defined as the same reset domain. There can be a variety of scenarios, including primary inputs/resets from the top, which are triggered simultaneously by an external reset, firmware, software, global resets, etc.*
2) *If there is functionality in the design, at any level, which ensures that triggering one reset will always produce the immediate assertion of another reset and vice versa, then both resets can be considered part of the same reset domain. For example with 2 resets, rst1 and rst2, if asserting rst1 will always*

trigger the immediate assertion of rst2, and the assertion of rst2 will always trigger the immediate assertion of rst1, then rst1 and rst2 are part of the same reset domain.

3) *If the assertion of a reset will cause the delayed assertion of another reset and vice versa, then the designer must assess the extent of the delay's impact on the functionality of the receive reset domain. If the designer can definitively determine that in the interim, between the assertion of the first and the second reset, no metastable values or incorrect logic values will have a chance to propagate out of the receive reset domain to the rest of the design (and potentially create bugs), then these two resets can be defined as sharing the same reset domain. In other words two resets, rst1 and rst2, can be defined as sharing the same reset domain if the following two conditions are met:*
    a) *Asserting rst1 will initiate the delayed assertion of rst2, and despite the delayed assertion of rst2, no incorrect data can/will propagate into the rest of the design.*
    b) *Asserting rst2 will initiate the delayed assertion of rst1, and despite the delayed assertion of rst1, no incorrect data can/will propagate into the rest of the design.*

B. Reset Order

In many cases, it is not possible to define 2 resets as sharing the same reset domain. While the assertion of one reset will trigger the assertion of another, vice versa may not always be true. This is particularly common with pairs of resets which may include shallow, powerful, global resets such as power-on-reset (PoR) and deeper, more localized, and specific resets. For example, the assertion of PoR may trigger the immediate assertion of all other resets in the design, however the opposite is certainly not true. The assertion of the deep localized reset for a particular block will *not* always trigger the assertion of PoR. In such cases RDC tools provide constraints for defining the order of reset assertion. One must capture for the RDC tool the order and dependency of reset assertions in the design. For instance, if the following reset order constraint is defined power_on_good→deep_reset (where "→" is the direction of reset assertion), then the RDC tool will automatically filter out all reset domain crossings with power_on_good as the transmit reset domain and deep_reset as the receive reset domain. However, in the opposite direction, the tool will still report RDC crossings with deep_reset as the transmit domain and power_on_good as the receive domain. Caution is advised in defining reset orders as the effects can be far reaching, and static tools cannot verify the accuracy of the user's constraint.

It is important to note that the RDC tool provides two methodologies for defining reset order, each with its own nuance. One can either define the reset order, or request that specific RDC paths between two reset domains be filtered out of the results (similar to CDC false path constraints). To illustrate the point, here is an example:

1) *Reset order constraints define a relationship of resets for the tool. For example, given the constraints rst1→rst2 and rst2→rst3, the tool can conclude by itself that rst1→rst3 i.e. without explicitly defining it. RDCs with rst1 as the transmit domain and rst3 as the receive domain will not be reported.*
2) *Reset filter path constraints will only remove the results of the explicitly defined set of resets. For example, given the constraints rst1→rst2 and rst2→rst3 RDCs with rst1 as the transmit domain and rst3 as the receive domain will still be reported. Should the user desire to filter out rst3 as well, then using the research filter path constraint, one will have to explicitly define rst1→rst3 or rst1→rst2 & rst3.*

It is critically important that these two reset order-related constraints be used correctly as they have distinct use cases, and incorrect handling can hide real RDC bugs. For example, the user has three power domains (1, 2, and 3) each associated with their own respective reset domains (rst1, rst2, & rst3). Examine the scenario depicted in Fig. 7, where the second power domain is powered down, but the first and third power domains remain on. If it is known that during the standard operation mode, power domain 2 will always be powered down, then it is desirable to remove all RDC violations between the second and the first and third power domains. However, were one to use the reset order constraint (i.e. rst1→ rst2 and rst2→ rst3), the RDC tool would derive automatically the additional reset order constraint rst1→ rst3. This would hide potentially serious bugs as all the RDCs from the first to the third power domains would not be checked. Instead, in such a case one should only use the reset filter path constraints (i.e. rst1→ rst2 and rst2→ rst3) which would not be extended to affect any other reset/power domains.

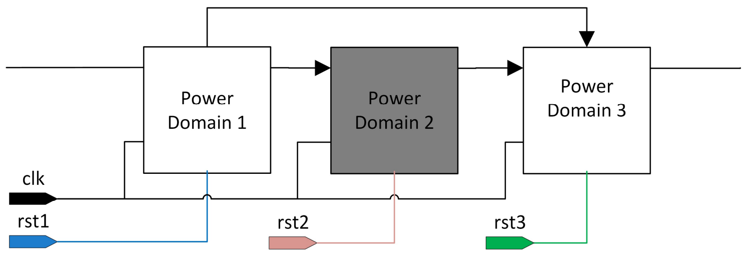

Figure 7. RDC crossings with power domain 2 powered down.

C. Synchronizers

The strategies mentioned above are by far the broadest and overall most effective for addressing RDCs, but it is usually not possible to modify the reset/functional architecture of the design to the point where it is completely RDC clean. There are then two main strategies which can be employed to avoid metastability in the RDC crossings: synchronizers and qualifiers. It is standard practice in the industry to resolve CDCs with synchronization schemes whose purpose is to protect functional logic and ensure that only stable valid logic values are sampled. This can be achieved through a variety of structures including FIFOs, DMUX bus synchronizers and the classic back-to-back 2DFF synchronizer. Due to the similarity of the source of CDC and RDC bugs (asynchronous data changes producing metastability), many of the same industry standard synchronizer structures can be implemented in the design to prevent RDC bugs.

For example, a back-to-back flip-flop (shown in Fig. 8) is inserted on the data signal between the source domain and the receive domain's sequential element. The 2DFF must be connected to the same clock and reset as the receive domain. Upon the asynchronous assertion of the transmit domain’s reset, the datum sampled by the first FF in the synchronizer may change too close to the rising clock edge, thus violating setup and hold time. This may cause a metastable output on the output of the first FF, however the synchronizer structure ensures a delay of two clock cycles for the metastable value to settle (Whether ‘1’ or ‘0’). Only then is the signal sampled by functional logic, thus preventing an RDC bug. As in CDC, one can perform statistical analysis to determine if the depth of the back-to-back FF is enough to avoid RDC issues, or if larger sizes (e.g. 3DFF, 4DFF, etc.) are required. It is important to note that while the RDC static tool can detect synchronizer schemes automatically in one’s design, the recommended practice is to turn this feature off and permit the tool to use synchronizers implemented with the project’s specific library of synchronizer cells.

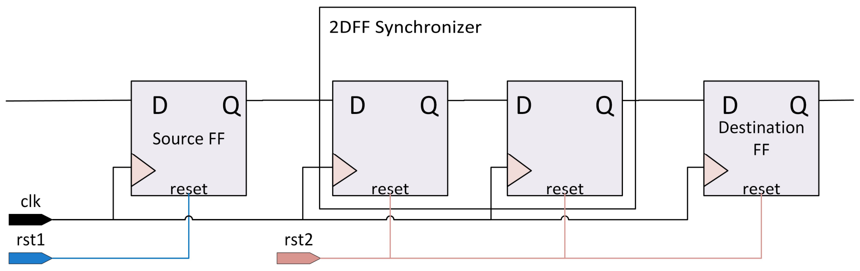

Figure 8. 2DFF RDC synchronizer.

D. Qualifiers

While placing synchronizers at every RDC crossing is a simple solution, it can be costly in terms of chip speed/performance, cell count, and congestion. An alternative method is to implement qualifiers to gate or block asynchronous data changes from propagating into the receive domain. Qualifiers require a signal to indicate that a reset is imminent, *before* the reset itself is actually asserted. To prevent the qualifier signal itself from introducing asynchronous data changes, the qualifier signal itself must be sampled by a sequential element in the clock and reset receive domain prior to propagating and blocking data signals in the receive domain. In many cases, this will require the use of synchronizers such as a 2DFF to ensure that the qualifier signal is synchronous and stable as can be seen in Fig. 9 below.

The qualifier signal must be generated sufficiently in advance to ensure that it will propagate through the design, and be sampled in the receive domain at least a full clock cycle in advance of the asynchronous reset assertion. This will ensure that the potential async data changes are blocked before they can arrive at the receive reset domain's FFs. Generating such a qualifier signal may involve significant architecture modifications. However, many designs already create such "pre-indication" signals in order to ensure a smooth and clean shutdown prior to reset assertion. In a typical design there is a Clock-and-Reset (CAR) module where the design's clocks and resets are generated and controlled. Since this module contains the logic which controls reset assertion, it is easiest to generate the qualifier signals from the CAR, as depicted in Fig. 9. From the CAR, the same qualifier signals can be synchronized as many times as necessary and distributed across the design to different clock and reset domains.

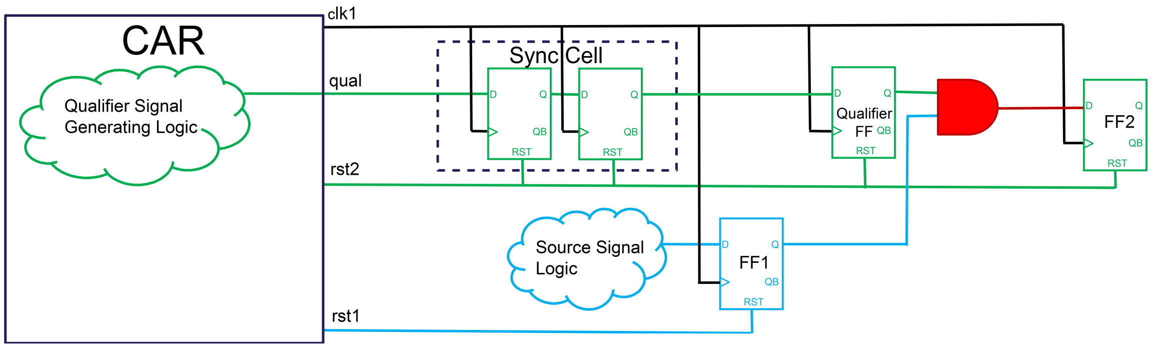

Figure 9. A qualifier signal generated in the CAR module and synchronized.

A typical example of a qualifier implementation employs a 2 input AND gate and can be seen below in Fig. 10. The AND gate is placed on the signal between the output (Q) of FF1 from the transmit reset domain and input (D) of FF2 of the receive reset domain. The second input of the AND gate is driven by the qualifier signal itself. As long as the qualifier is high ('1'), all signals from the transmit reset domain can propagate freely to the receive reset domain. If rst1 is about to be asserted while rst2 is de-asserted, the qualifier signal will be driven low, gating any value changes, asynchronous or otherwise, from reaching the receive reset domain. This ensures that no bugs due to metastability can occur.

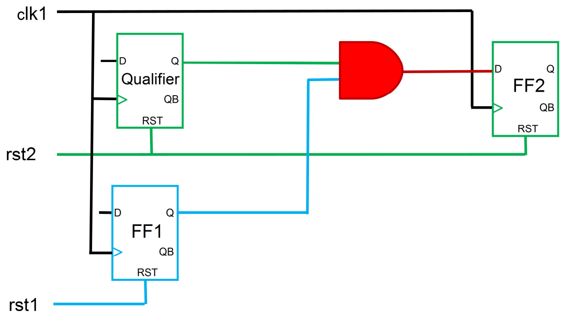

Figure 10. Data Qualifier.

There are other methods of implementing qualifiers. Fig. 11 illustrates a clock gating qualifier which will freeze the clock driving the receive FF when the qualifier goes low. Without a rising edge clock signal to trigger sampling, FF2 cannot possibly sample an asynchronous signal change or violate setup and hold timing restrictions. The RDC tool is capable of repurposing the existing power saving latch based clock gating structures as RDC qualifiers.

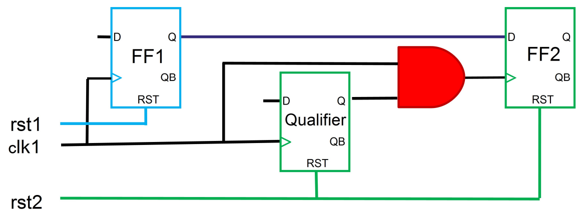

Figure 11. Clock gating qualifier.

Another example of an RDC qualifier implementation is similar in concept to the data gating example above. However, in this case, the output of the AND gate is inverted, and used to toggle the enable pin of the receive domain's FF (Figure 12). The output of FF1 doesn't just drive the qualifier's AND gate, it is also connected directly to the input of FF2. Thus the FF will be disabled only if both conditions are simultaneously met:

1) *The qualifier signal goes high to signal an imminent reset assertion.*
2) *The output of FF1 is currently not at its reset value (i.e. output currently '1'). This avoids disabling FFs when unnecessary since while FF1's output equals the reset value, no asynchronous datum change can occur upon reset assertion.*

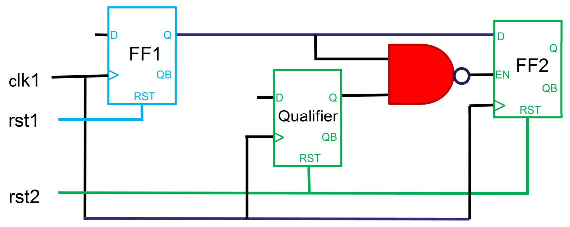

Figure 12. Enable qualifier.

It is important to note that while one qualifier signal per pair of transmit/receive reset domains is sufficient, the gating logic (i.e. AND gate) will need to be placed for each and every RDC crossing. While most qualifiers are implemented using AND gates, the static RDC tool supports using OR gates as well, and can be instructed whether the qualifier logic is active high or active low. Additionally, the qualifier constraints support defining the name of the qualifier signal as well as defining the specific transmit and receive clock and reset domains affected. This enables the user to carefully limit the extent of the qualifier's influence.

In general, qualifiers must be used carefully, since usually RDC checks will accept such qualifier constraints "as-is" and remove RDC violations from the reports without verifying the qualifier's true, dynamic, functional impact. Some RDC tools allow users to define the specific condition or logic expression for qualifier assertion to limit the scenarios where the qualifier is active during RDC analysis. Other RDC tools provide advanced checks capable of performing additional dynamic or formal analysis to verify that the user-defined qualifier constraints are correct and the qualifier is functionally capable of gating the asynchronous data change.

E. Waivers

Similar to the standard CDC and LINT static RTL based verification tools, the RDC tool provides an advanced waiver mechanism for waiving RDC warnings by message text, module, source or receive reset and clock domain, wildcards, etc. The tool also provides an advance TCL based interface to dynamically generate waivers, in order to take into account characteristics beyond those listed in the basic RDC text warnings. This dynamic waiver can take into account signal names and sequential elements on the RDC violation's fan-in or fan-out, and has proven extremely useful in waiving false violations.

For example, our designs make use of standard pulse-to-toggle and toggle-to-pulse synchronization schemes to transmit signals across clock domains. The structure (Fig. 13 below) first turns the pulse event from the transmit clock domain into a toggle signal before it can be safely synchronized by a 2DFF synchronizer and converted from a toggle back to a pulse in the receive clock domain. The tool reports an RDC crossing between the FF that generates the pulse signal in the first reset domain and the pulse-to-toggle logic in the second reset domain. This is not a true violation as the 2DFF after the pulse-to-toggle logic will ensure that no metastable values propagate into the second reset domain. However, since the 2DFF appears after the RDC crossing and not in between, it is not recognized as a valid RDC synchronization scheme.

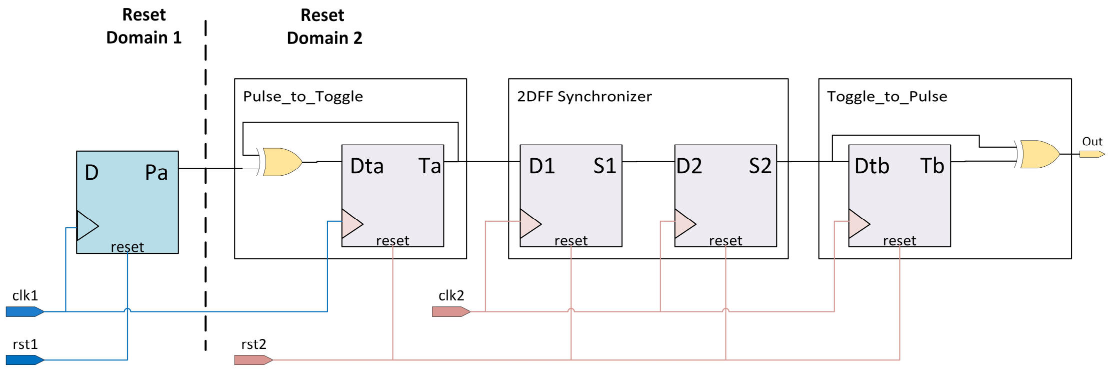

Figure 13. Pulse to toggle and toggle to pulse synchronizer.

There is no easy fix for this issue, as to avoid bugs and glitches the pulse-to-toggle generating logic must be associated with clk1. At the same time, to avoid false toggles, the reset must be associated with the receive reset domain (rst2). The static RDC tool found thousands of violations in our design related to this structure; however, there is no easy way to globally waive this category of violations. The message text itself was not enough as the pulse-to-toggle was a standard library cell which was not exclusively used in clock domain crossing schemes. The solution in this case was to use the dynamic waiver abilities of the tool, and through TCL to automatically generate waivers for cases where a 2DFF synchronizer was located on the fanout of the pulse-to-toggle. This illustrates how dynamic waivers can save many man hours of engineers' time RDC cleaning the design.

V. Setup and Running RDC Static Tools

A. Initial Setup

Industry standard CDC/LINT-like tools are now available that can detect and analyze RDC issues in one's design. They are static, RTL based tools, usually with a large capacity, which enable running large designs flat in one run. This is key, as most RDC issues are found in the connectivity between blocks/clusters, and fewer are located internally inside the blocks and clusters themselves. Therefore, running RDC checks full chip flat is most beneficial, though most vendors also provide top-down and bottom-up hierarchical capabilities/flows. There are two ways one can introduce an RDC tool into a project's standard verification flow:

1) *Several vendors' platform level CDC/LINT tools already support RDC verification. If one's flow already employs such a tool, then in most cases all that is required is to enter the relevant reset definition/constraints described above, along with the applicable RDC rules to your rule list/methodology.*
2) *If one's current CDC/LINT tools does not support RDC check, it is still relatively quick and easy to perform RDC checks through a separate tool in parallel. This will require adapting the file list, clock and reset definitions, and functional mode constraints from your current CDC/LINT flow. Primarily, this involves modifying the constraint syntax, but in some cases there are slight differences in the way tools interpret the clock/reset tree which will require some name changes and new constraints to overcome. It shouldn't be necessary to convert or translate the CDC waivers, stable signal declarations and other such constraints from the CDC/LINT tools' environment, as they generally have no impact on RDC issues.*

B. *Tool Flow*

Running an RDC tool in many ways mirrors CDC flows. Initially, the design is read into the tool in the form of RTL file lists, known clock and reset definitions (sync, async), etc., and a report is produced. Next, users must incorporate known architectural and design information in the form of constraints to define reset domains, reset orders and reset filter paths. Generally, the definition phase requires several iterations until all of the resets have been defined and all of the constraints have been captured. Through the tool's schematic view and the graphic user interface (GUI), qualifiers can be defined and waivers added to the violation database to remove false warnings. RDC issues can be debugged interactively and real bugs can be resolved by adding synchronizers or making architectural changes. The RDC flow can be easily automated to run for every subsequent RTL release to track fixes and monitor for any new RDC bugs that may have subsequently been introduced.

VI. REAL WORLD RESULTS

In the past, our division encountered several serious RDC bugs while testing our silicon. At the time, no automated tools were known to check for RDC issues, and manual methods such as GREPing RTL were found to be woefully ineffective. In 2014, we had a large project which was basically clean from a LINT, CDC, and simulation perspective, and close to tape out. This network chip required about 2.5 million FFs, or approximately 52 million NAND equivalent gates, and possessed 13 clock domains and 8 reset domains.

Intel's Plan of Record (POR) static LINT and CDC tools did not support RDC checks, so as a trial we added a second RDC tool in parallel. Initial setup based on our existing CDC flow was relatively quick and easy. Initially, noise levels of false RDC violations were difficult to manage, but by deploying the strategies discussed above, message counts were brought down to manageable levels.

Our RDC flow found several serious RDC bugs which were officially documented as high risk issues that required RTL or design fixes. Throughout the project's lifecycle, there were also many undocumented "on the fly" RTL fixes by the designers. User feedback throughout the trial found the RDC checks to be very accurate at catching real, or potentially real, issues, without many false violations. Users also found the tool a robust and user friendly platform for analyzing and debugging RDC issues and iterating to verify design changes and waivers.

Since the initial trial, RDC tools have continued to improve with many enhancements such as new constraints and capabilities to enable the recognition of synchronization and qualifier schemes. The recent edition of qualifiers to the tool was extremely beneficial, as our designs already generated pre-indication signals to ensure the proper and clean completion of transmission of network packets before reset assertion. This made it extremely easy to leverage the pre-indication signals and resolve many RDC issues with little effort at a low cost.

In parallel to our first RDC trial, a smaller project in our division, also in the design stage, declined to join the RDC trial due to time constraints. Both projects taped out, and an RDC bug that could have easily been detected was found in the silicon of the project that was not checked for RDC issues. As a result, RDC checks have now become POR in our division for all projects as part of signoff verification. In Intel and throughout the industry, RDC checks are gaining recognition, awareness and traction towards becoming an accepted part of the standard verification flow and methodology.

ACKNOWLEDGMENTS

Thank you to Intel's former LAD engineering department (now part of DCG) for allowing the design automation team to think outside the box, explore, and pioneer new tools and flows which made this paper possible. Also thank you to Intel for funding this trip to DVCon so I can present this paper. A special thank you to my wonderful wife, Shoshana, for as always editing my paper.

REFERENCES

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

[2] Arjun Pal Chowdhury, Neha Agarwal and Ankush Sethi, “Dealing with SoC metastability problems due to Reset Domain Crossing”, Embedded.com, November 10, 2013. Available: http://www.embedded.com/design/programming-languages-and-tools/4424093/Dealing-with-SoC-metastability-problems-due-to-Reset-Domain-Crossing- [Accessed 7 11 2016].

[3] Yossi Mirsky, “Using Static Checks in Automated Flows to Catch Reset Domain Crossings, A Growing Issue Within Today’s Complex Designs”, SNUG Israel 2016.

[4] Clifford E. Cummings, Don Mills, and Steve Golson, “Asynchronous & Synchronous Reset Design Techniques - Part Deux”, SNUG Boston, 2003.

[5] Chris Kwok, Priya Viswanathan, Ping Yeung, “Addressing the Challenges of Reset Verification in SoC Designs”, DVCon US, 2015.