Preventing Chip-Killing Glitches on CDC Paths with Automated Formal Analysis

Jackie Hsiung
Mediatek Inc.
Hsinchu City, Taiwan
jackie.hsiung@mediatek.com

Ashish Hari
Mentor, A Siemens Business
Noida, India
ashish_hari@mentor.com

Sulabh Kumar Khare
Mentor, A Siemens Business
Noida, India
sulabh-kumar_khare@mentor.com

Abstract: As design sizes continue to grow and geometries continue to shrink, more and more SoCs are at risk of falling prey to the silent chip-killer — glitch defects on asynchronous paths. Glitches detected on a SoC at the last minute can cause panic and waste months of verification cycles. Worse, if it's detected post manufacturing, it can lead to very costly recalls.

Glitches are undesired transitions that occur before the signal settles to its intended value. These glitches, which are pulses of very short duration, may be captured by a register when crossing clock domain, thereby causing a functional failure. Clock domain crossing (CDC) paths with multiplexer or combinational logic are often prone to glitch defects that can be introduced during the synthesis process. Since CDC verification is usually done at the RTL, such glitch defects can be missed and lead to logic failure resulting in costly chip respins. These defects are tricky and can often escape undetected. These cannot be accurately identified in gate-level simulation as simulators have difficulty sampling the glitch since it is hard to create a pattern to cover glitch conditions and an existing glitch in a waveform seldom hits the clock edge on asynchronous paths. Similarly, static CDC verification checks that warn for combinational logic in a data path do not work well on gate-level designs as these checks would be very noisy. At the gate level every CDC path is bit blasted and any multiplexer or combinational logic is synthesized as OAI (or-and-invert) or AOI (and-or-invert) logic, and it would take considerable effort to reach a select few real glitch paths out of over a million CDC paths using this approach.

In this paper we present a unique method based on automatic formal analysis to zero-in on real glitch paths that can kill your chip if not addressed early. This paper further addresses the challenges involved in fixing the glitch paths once these are identified. Appropriate debug aids based on combinational expression analysis of the data path logic are presented to quickly zoom to the logic that's responsible for introducing the glitch. We walk through a case study on a real design utilizing the combination of formal techniques and a structured glitch resolution methodology to significantly improve glitch detection accuracy, utilize advanced debug techniques to identify and fix glitches, and save crucial verification cycles late in the design tape-out phase.

I. INTRODUCTION

Glitches are unwanted spikes in signals that can propagate through the combinational logic. At times, signals display an incorrect intermediate value before reaching the final, steady state value; this results in a glitch. Fig 1 shows glitches in combinational logic. A signal may temporarily change its value while it is supposed to remain static at a logic 1 or 0 value, or it may oscillate while changing value from 0 to 1 or 1 to 0 before reaching the final value.

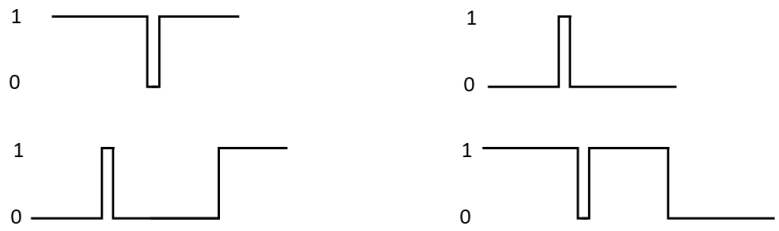

Fig1: Examples of glitches on combinational logic

In combinational logic, a glitch can occur if there are multiple paths with opposite polarity converging at an AND or OR gate and the path delays from the source of the signal to the converging point do not match. Glitches can occasionally cause functional errors that may lead to chip failures. Typically, glitches can be introduced during synthesis due to glitch prone implementation of combinational logic. Fig 2 shows a simplified synthesis flow; it takes an RTL description of the design in an HDL language like Verilog or VHDL with design constraints and converts it to a gate-level netlist. The synthesis process performs some optimizations to satisfy the design constraints and meet power, area, and frequency requirements. During the synthesis process, it may implement basic RTL elements like multiplexers in a way which is prone to glitches.

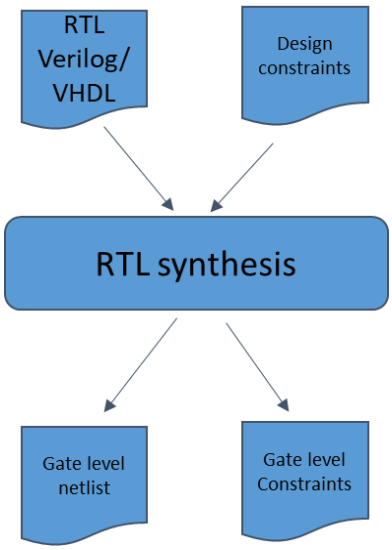

Fig 2: Synthesis may produce glitch prone logic when it converts RTL to a gate-level netlist

A glitch generated in one clock domain can be captured by a register in another clock domain if the glitch passes through a clock domain crossing (CDC) path. Some unsynchronized paths at the RTL may be deemed safe due to stable signals, such as configuration registers. In other cases, some synchronized CDC paths which use a data-mux type of synchronizer are structurally verified at the RTL and considered good CDC paths. There are other types of complex synchronizers, such as FIFO and handshake, that include a multiplexer or combinational logic at the CDC path in RTL and are considered good CDC paths with combinational logic because the protocol involved in the data transfer is followed. A CDC path that is completely glitch safe at the RTL can run into chip-killing glitch defects post synthesis if the combinational logic and multiplexer logic in crossing paths is not synthesized in a glitch-free manner. The challenge with glitch problems is that they manifest very late in the design cycle yet often are the culprit behind crucial silicon re-spins. This problem becomes more acute as SoCs move towards lower geometries and higher performance. The real challenge is to identify CDC paths with potential glitches from among the huge number of CDC paths, typically over a million in a netlist, and then provide sufficient guidance to the verification engineer to quickly address them.

In this paper, we first examine the glitch problem in CDC paths then discuss the various attempts made to identify glitch paths in the netlist and the challenges faced in each method. We also talk about inherent limitations of each method. We propose a new solution based on automatic formal analysis and show how it can be employed practically in large SoCs. The proposed methodology was applied for complete glitch verification on a set of real SoCs. We illustrate a real glitch scenario in a design and summarize the results obtained using our proposed methodology.

II. The Problem of Glitches on CDC Paths in Netlists

As an example of how a glitch can be introduced in a CDC path, consider the logic in Fig 3, which is an example of a synchronized CDC path in RTL. It is a data-mux synchronizer in which a control signal is synchronized in the receiver domain and data is transferred from the transmitter to the receiver domain when the multiplexer is enabled by a synchronized control signal. When the multiplexer is not enabled by the control signal, the receiver holds the data. Structurally the schematic in Fig 3 satisfies all requirements of the synchronizer.

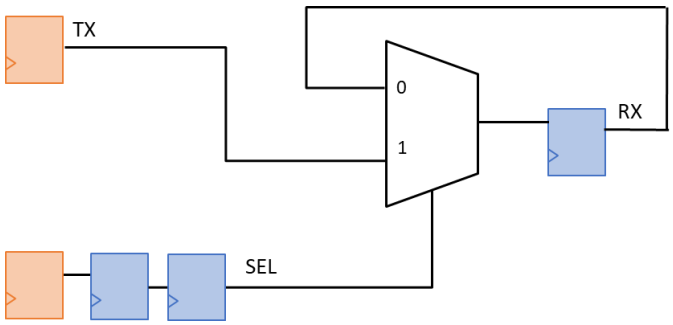

Fig 3 : CDC path in RTL is glitch free

Every CDC synchronizer has a protocol, and the data-mux synchronizer has a protocol that the transmit data should not change when the multiplexer is enabled. Typically at the RTL, in addition to structural verification, the protocol of the synchronizer is validated using simulation or formal methods to ensure that data is transferred safely across the CDC path. Thus, the schematic in Fig 3 represents a completely valid CDC synchronizer at the RTL when the protocol for synchronizer is followed.

Fig 4 shows the schematic of the same CDC crossing after synthesis with combinational logic. The synthesis tool creates such logic while mapping the multiplexer to gates. The combinational logic is equivalent to the multiplexer in functionality, but it has the potential to glitch under certain conditions. For example when the SEL is 0 and RX is 1, the RX should hold its value and the output of the OR gate should always stay 1. But in this schematic, when the TX changes its value from 1 to 0, there is a possibility of delay through the inverter and the output signal may momentarily become 0. So even when SEL is 0 the output of the OR gate can create glitches for TX value changes. If the RX register in the receiving clock domain samples the glitch, it may cause a CDC bug. This kind of bug will be missed in the CDC verification. If the designer is lucky, it might be detected in gate-level simulation; if unlucky, it may escape to silicon and cause chip failure.

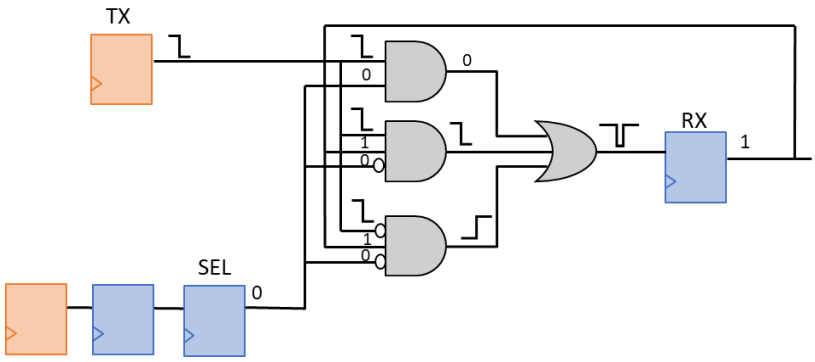

Fig 4: Static-0 glitch introduced in CDC path after synthesis

The glitch prone logic can be fixed by adding extra combinational logic, as shown in Fig 5. With the additional logic, whenever the transmitter and receiver are 0 it makes sure that the output is always 0, and it prevents the transmitter from making any glitches at the OR gate output. This is typically done by an engineering change order (ECO). ECOs are the process of inserting a logic change directly into the netlist after it has already been processed by an automatic tool. ECOs are usually done to save time before the chip masks are made because they avoid the need for full ASIC logic synthesis, technology mapping, place and route, and timing verification.

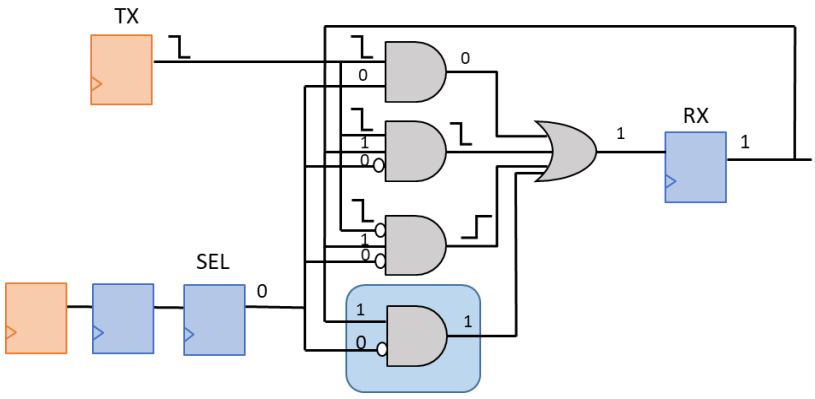

Fig 5: Additional logic added to stop glitch generation

Another example of glitch prone logic on a CDC path are unsynchronized paths. An unsynchronized path at the RTL may be considered valid if it has stable signals, such as a configuration register on the transmitter side. A combinational logic may be allowed on the unsynchronized path since it is guaranteed that the configuration register will not change its value once s et. If the combinational logic involved a controlling signal from the transmitter domain and if it is synthesized to a glitch prone logic, any change in the control signal can create a glitch at the output of the combinational logic which may be captured by a receiver register in another clock domain.

III. Challenges Detecting Glitch Prone CDC on a Netlist

As discussed in the previous section, glitches can be introduced on good RTL paths which were completely CDC clean and had good functionality during the synthesis process. Verification engineers have attempted to identify the presence of combinational logic capable of inducing the glitches in signals traversing between clock domains in a variety of ways. Each such method has some limitations in terms of the accuracy it provides and effort required to identify the true glitch paths. Below are some examples of methods tried by engineers and the challenges faced.

- **Simulation to capture glitch**: For each of the clock domain crossing paths with combinational logic, the receiver value can be sampled in a gate-level simulation. Sampled values then can be used to identify the presence of glitch logic in the clock domain crossing path. Due to the asynchronous nature of signal paths crossing clock domains, many simulators have difficulty sampling the signal paths at the times when glitches may be present. Even if the simulator catches a glitch path and a waveform is present, it is difficult to manually identify the source of the glitch path. There are typically over a million CDC paths in the netlist and if an ECO is required at the netlist level to resolve the glitch with additional logic or stop the propagation of the glitch, it is necessary to know all the glitch sources and the exact point of glitch origin.

- **Assertions to identify glitch:** Some verification engineers have elected to write assertions to check if the simulators are sampling the signal paths at the correct time, but the results of sampling checks are often not accurately captured by the simulator. Assertions also have some semantic limitations when it comes to asynchronous clocks, and all simulators may not handle these kind of assertions. So writing assertions may itself become a difficult task.

- **Static checks at the netlist level:** Verification engineers have also performed static checks on the circuit design to identify portions of the circuit that may be susceptible to inducing glitches in signals traversing a clock domain crossing. These static checks typically include flagging every clock domain crossing that contains combinational logic in its path for manual inspection by the verification engineers. While manual inspection can identify combinational logic capable of inducing signal glitches, the procedure is error-prone and time-consuming. This becomes a mammoth effort at the gate-level because every data path with mux logic is now transformed as a path with significant combinational logic. The presence of design for test (DFT) and power logic also makes the relatively simpler RTL data paths more complicated.

- **Preventive steps at the RTL:** Preventive steps at the RTL that mark data paths with mux logic as “don’t touch” during synthesis are often used and significantly reduce glitches that propagate to the netlist. However, a combination of factors and processes in the design cycle inevitably leads to a few glitch errors manifesting during synthesis or escaping detection until late in the design cycle. The problem of ensuring that the design being taped out is glitch proof still remains to be addressed.

With the current challenges, an approach than can help zero-in on the real glitch paths out of the millions of CDC paths is highly desirable and can significantly reduce the verification effort late in the design cycle and eliminate any risk of silicon respin on account of glitches on asynchronous paths.

IV. Automatic Formal Based Glitch Detection Methodology

Fig 6 shows the steps in our proposed automatic, formal-based glitch detection methodology. The proposed method utilizes a combination of structural CDC analysis, expression analysis, and formal analysis to prune and prove the real glitches in the design at the gate level. This approach works on gate-level designs and is hence fool proof. The detected real glitches are very few and can be investigated quickly utilizing the visualization aids available. The approach has various stages that are interlinked, and the entire methodology is automated.

The first stage is complete static CDC analysis of paths to prune out paths that do not contain any combinational log ic at the gate level. These are regarded as glitch-free paths. This requires a dedicated CDC solution that can understand RTL setups and automatically convert them to the gate level and that can also infer additional information about the DFT constraints at the gate level.

The next step is to do a comprehensive expression analysis of the combinational logic tree in the data path to identify potential glitch candidates that can cause the glitch to propagate. This further prunes the list of glitch candidates; however the real challenge is to identify the scenarios under which the glitch really propagates. To accomplish this, we utilize formal engines to verify the glitch propagation condition and conclusively give counter examples of scenarios under which the glitch will propagate. This results in glitches being detected that have been proven to exist and propagate during design operation. The counter example helps the verification engineer convince the IP team of the exact scenario under which the glitch can propagate and cause a silicon failure.

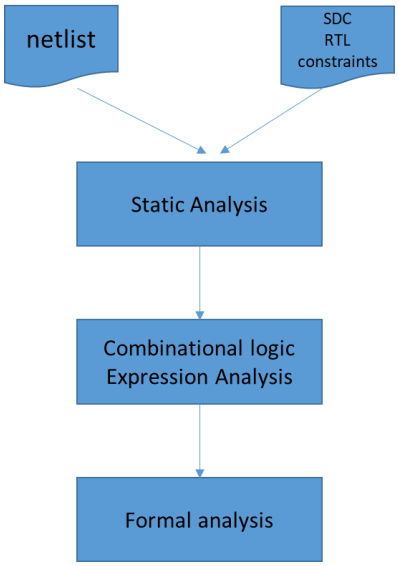

Fig 6: Proposed automatic formal-based glitch analysis flow

The combinational logic on the data path of glitch prone paths can be extremely deep. So for the select few paths that are glitch prone, comprehensive expression analysis helps result in identifying the exact location at which the signal and its complementary term are expected to converge, resulting in a glitch during real design operation. This expression analysis technique coupled with GUI visualization substantially reduces the glitch investigation and resolution effort.

The method was critically analyzed on the following factors:

- **Quality of results and minimal noise** - At the netlist level, verification cycles are already stretched so quality of results have a critical impact on meeting tape-out schedules. The proposed formal-based approach for glitch verification always resulted in less than a few hundred glitches for millions of paths thereby reducing the verification effort. Utilizing this formal approach eliminated noise in results; as well, very few reported glitches were deemed invalid (and these were due to some missing constraints in setups).

- **Ability to run very large SoCs** – The hierarchical CDC verification approach was utilized to propagate constraints from the top run to the block level, complete the CDC and glitch analysis on blocks, and then do glitch analysis at the top level. In addition, constraints from the RTL setup were transformed and reused to quickly bring up the CDC solution at the gate level to get accurate results quickly. It was essential for the glitch verification solution to automatically consume and transform RTL directives to gate-level directives. Fig 7 and 8 show the hierarchical approach followed to run very large SoCs.

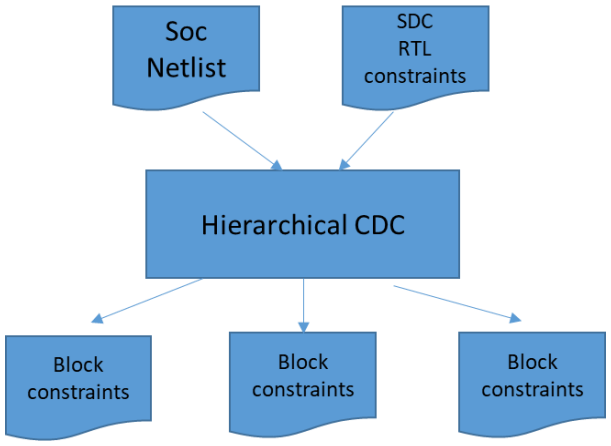

Fig 7: Hierarchical CDC runs to generate constraints for blocks (partitions)

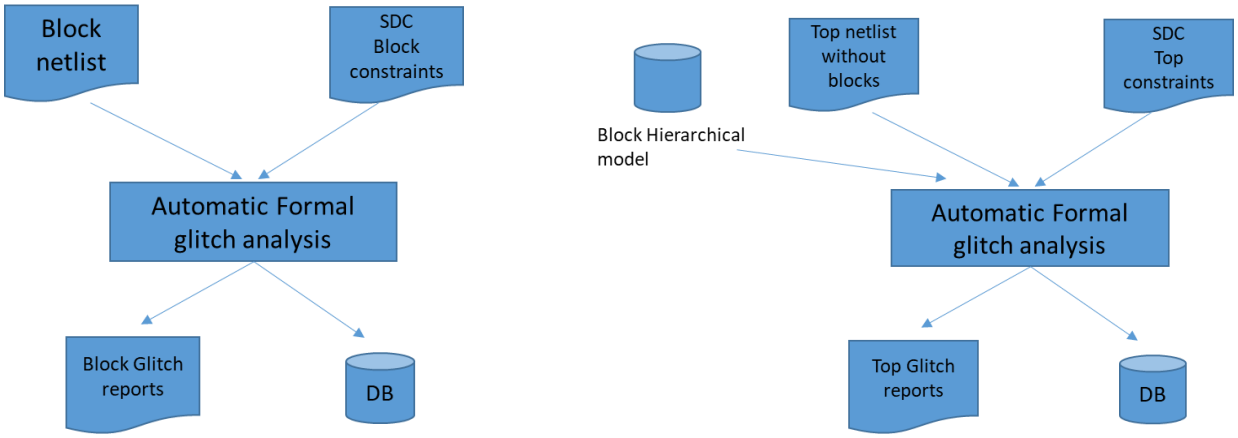

Fig 8: Formal glitch analysis can be done on blocks and top level separately

- **Ease of debug** – Due to deep combinational logic on glitch prone paths, it was essential to zero-in on the exact area where the glitch needs to be addressed. This required review of schematics, and the time to review each proven glitch path was noted. This was generally in the order of a few minutes due to additional guidance on convergence points in the data paths and the ability to use expression analysis to identify the exact paths that represent the complex glitch scenario. Paths with glitch signals and the source of glitches are highlighted in the schematic as shown in Fig 9. The precision of debug information helps reduce the time to reach the root cause of the issue to just a few minutes, down from many hours without this information.

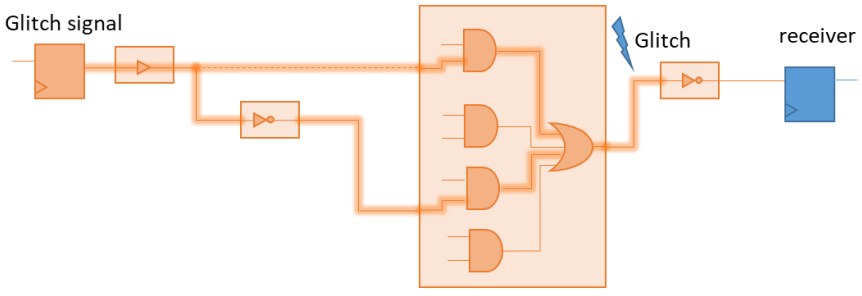

Fig 9: Highlight glitch signal and its converging points

- **Reduction in verification cycles** – Utilizing the proposed approach significantly reduces the verification cycles as glitch verification can be performed with existing RTL setups and the results pinpoint the exact glitch signal and the point where the glitch was introduced. With this approach, no simulation setup and test vectors are required, and it is much faster than simulation to get the results.

V. CASE-STUDY

We will now illustrate the proposed methodology with an example of a real glitch which was observed in gate-level simulation of one of the SoCs. Fig 10 shows the details of the glitch: a FIFO's output selection logic was creating a glitch, and it was captured in an asynchronous domain leading to functional issues. The CDC crossing path was a valid FIFO path at the RTL, and it was signed off for CDC. The glitch scenario was identified in a gate-level simulation of the netlist. It was clear from the simulation results that the glitch scenario was valid, but the source of the glitch was not clear from the simulation waveform. The glitch needed to be fixed at the source. Also it was essential to identify all such cases in the design if they existed. For each glitch path we needed to make a decision on how to guard against them before tape-out. Fig 11 shows the analysis of the netlist glitch with RTL. A case statement from the Verilog RTL, which was used to select FIFO output, was synthesized into glitch prone AOI (and-or-invert) logic by the synthesis.

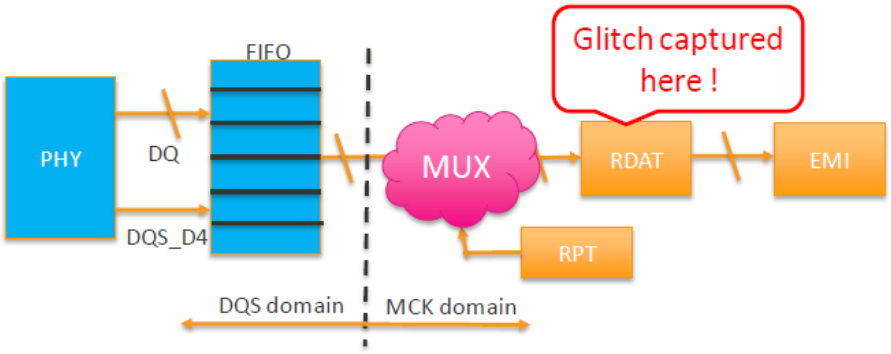

Fig 10: Real glitch scenario in a SoC

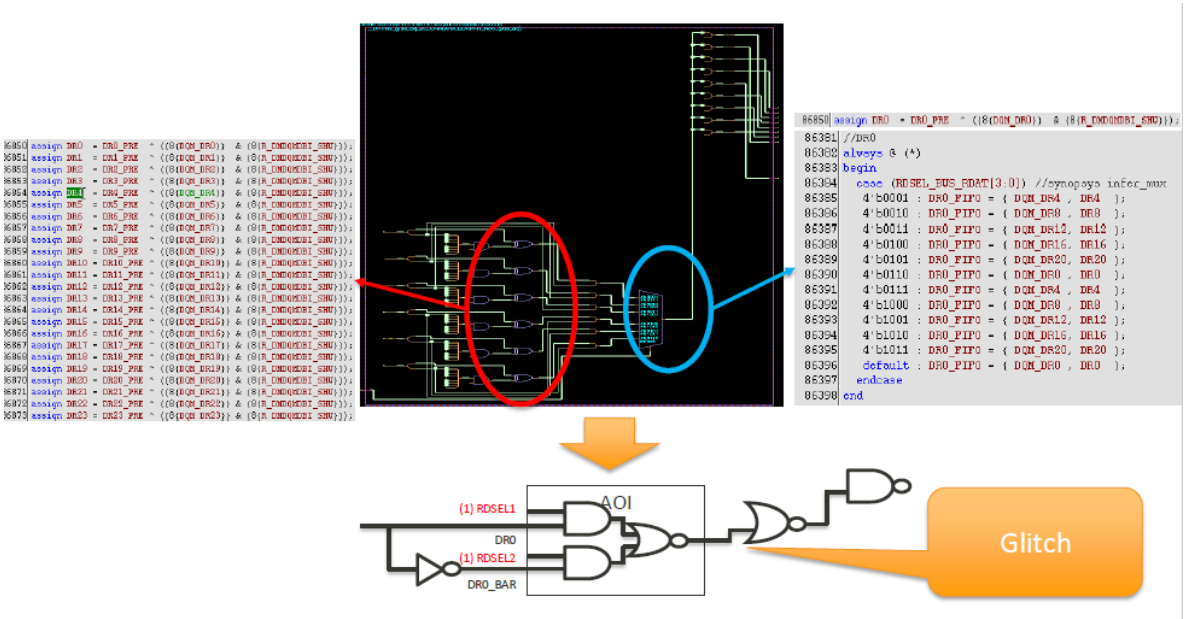

Fig 11: RTL to gate-level analysis of the real glitch scenario

Using the proposed methodology on the SoC, we were able to identify the glitch case with complete details of the source of the glitch. With the exact location of the glitch source available it was easy to make an ECO in the netlist to ensure that the glitch is prevented and does not enter into another domain. Additional glitch paths were identified by the automatic formal-based analysis, which were further reviewed and fixed.

After completing glitch analysis and verification successfully in one of the SoCs, we used the proposed methodology on a set of other real SoC netlists with sizes ranging from 1 million to 100 million gates. Fig 12 shows the details of the methodology used on SoCs. The hierarchical approach helped to make sure that the constraints for large partitions were generated correctly, and we have separate reports and debug databases available for teams to review the results. We captured the total number of glitch source paths that contained combinational logic at the gate level. The number of glitch sources reported was minimal compared to the total number of CDC paths at the gate level. The reason for low noise is the pruning of paths during various stages of the glitch identification methodology.

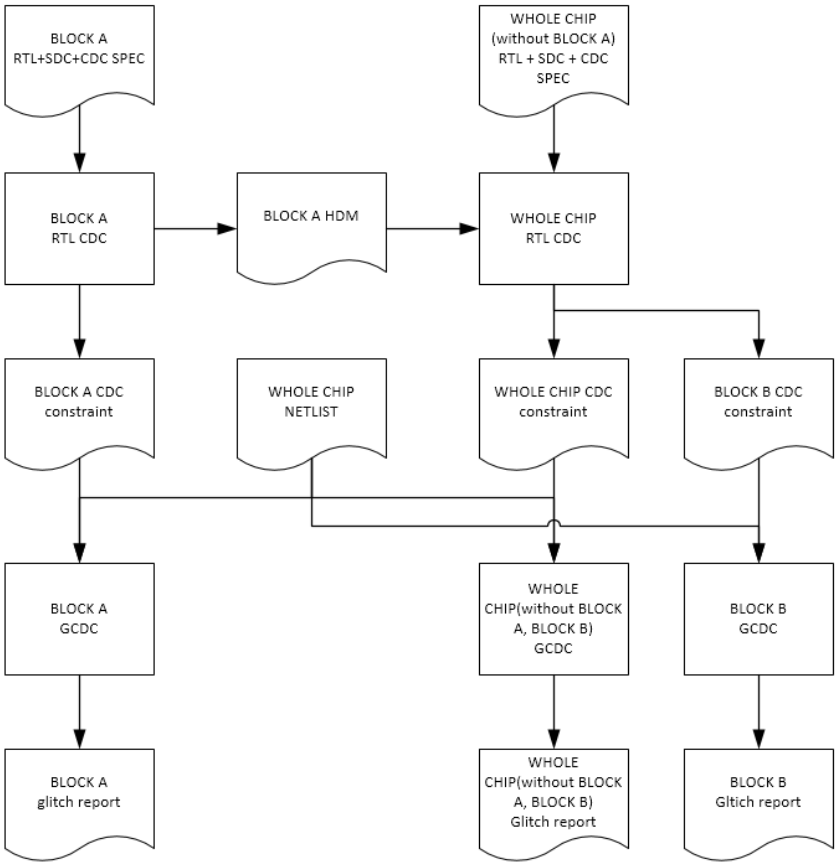

Fig 12: Mediatek glitch verification flow

Table 1 shows the details of the multiple SOCs and glitch analysis results. Glitch analysis noise was much less, and we were able to complete glitch analysis within the schedule for tape-out. Out of the detected glitch sources, two real glitch issues were identified. In Project A, the glitch issue was found on a data-mux crossing, and an ECO was done on 288 paths to block the glitch's propagation. In Project C, the glitch issue was identified on a missing synchronizer crossing. This case was considered safe at the RTL, but in the gate-level netlist, the combinational logic synthesized was glitch prone. In other cases the glitch source and the glitch paths were considered safe after debug and waived. The proposed automatic formal analysis based glitch detection methodology saved valuable time for us late in the verification process.

Table 1: Glitch results on various SoCs

S.No.	Project Name	Clock Domains	Glitch Source (Sum of all partitions)	Run time (Sum of all partitions/Maximum for a partition) hrs	Glitch Result
1	Project A	1155	1848	93/31	data-mux glitch found in one module (288 paths ECO)
2	Project B	1028	1639	87/38	All waived
3	Project C	641	1241	69/23	no-sync glitch found in one module (166 paths ECO)
4	Project D	823	1487	53/6	All waived
5	Project E	828	1534	77/29	All waived
6	Project F	754	2846	103/34	All waived
7	Project G	963	2262	84/32	All waived

VI. CONCLUSION

The proposed automatic formal-based technique significantly improves the quality of results for glitch detection, and it reduces the time required to identify glitches on CDC paths. This methodology identifies a few relevant proven glitches out of millions of CDC paths and helps users focus on verifying them effectively. The generation of counter examples under which glitches would propagate makes it obvious that reported issues are genuine and pose serious threats to our next SoC. Advanced debug techniques that highlight the glitch introduction point help in fixing the glitches efficiently.

Validation on real SoCs confirms that the proposed flow and techniques are practical and must be applied on modern designs as a signoff methodology to prevent chip-killing glitches before tape-out.

REFERENCES

[1] Anwesha Choudhury, Ashish Hari, “Accelerating CDC Verification Closure on Gate-Level Designs”, DVCON 2017

[2] Clifford E. Cummings, “Clock Domain Crossing (CDC) Design & Verification Techniques Using System Verilog”, SNUG-2008.

[3] Ping Yeung, “Five Steps to Quality CDC Verification,” Mentor Graphics, advanced verification white paper.

[4] M. Litterick, “Full Flow Clock Domain Crossing – From Source to Si”, DVCON 2016