Covering All the bases: Coverage-driven Formal Verification Sign-off of Pipelined Error Detection Filter

Harbaksh Gupta, Anshul Jain
{harbaksh.gupta, anshul.jain}@intel.com

***Abstract*- The erratic bit failure phenomenon in advanced on-chip cache memory poses significant challenges for modern System-on-Chips (SoCs). While efforts have been made to log, correct, and recover from such failures, simulating all possible combinations and sequences of errors remains practically impossible. This paper proposes a formal verification (FV) methodology that complements or replaces traditional simulation approaches for select hardware designs. By leveraging FV's capability to cover 100% of the design's state space and produce exhaustive proofs, our methodology addresses the complex nature of deeply pipelined hardware designs with multiple interfaces, events, handshakes, and dependencies. We introduce a comprehensive formal verification methodology to enhance confidence in the quality of FV efforts and achieve formal sign-off. The effectiveness of our methodology is demonstrated through a case study where we applied our methodology to formally sign-off an error detection filter of a flagship server CPU SoC.**

I. Introduction

Erratic bit failure phenomenon is one of the widely reported issue in advanced flash memories used for implementing on-chip cache memory [1]. Modern SoCs invest great efforts in logging, correcting, and recovering from such bit failures. These failures can occur due to various factors such as aging, environmental conditions, and variations in manufacturing processes. As the word "erratic" suggest, simulating all possible combinations and sequence of errors is practically impossible. DV would not only require many test vectors and sizeable compute to simulate errors, but it will also fall short of providing absolute confidence in verification effort.

Formal Verification (FV) has become a mainstream verification methodology, and advanced to the stage that it can complement or even replace simulation effort for select hardware designs. Today, FV is widely recognized for its capability to cover 100% state space of the design and produce exhaustive proofs. For complex hardware designs involving many interfaces, many events (including errors), many handshakes and dependencies, a deterministic measure of the design state space being simulated and checked is necessary for high-quality formal sign-off.

In this paper, we would like to share our formal methodology for verifying deeply pipelined hardware designs which involves filtering different types of correctable and un-correctable erratic errors. We will discuss how we leveraged formal coverage analysis to enhance the confidence in quality of formal verification effort and achieving formal sign-off. We will also discuss the effectiveness of our methodology using a case study related work.

II. Related Work

Application of formal verification has matured many-folds in past one decade. Formal verification engineers/researchers/enthusiasts as well as EDA companies have been churning out new ideas, implementations, and utilities for pushing the boundaries of FV and its application. We leveraged some of such methods and developments in our work and would like to list them in this section.

A. Floating Pulse

Floating pulse[2] is a well-known powerful formal technique where the formal tool can assert a single pulse at some arbitrary time after reset, and on that pulse do something special. Based on the purpose of use, special thing can be used to tag a certain transaction, inserting a barrier, switching modes etc.

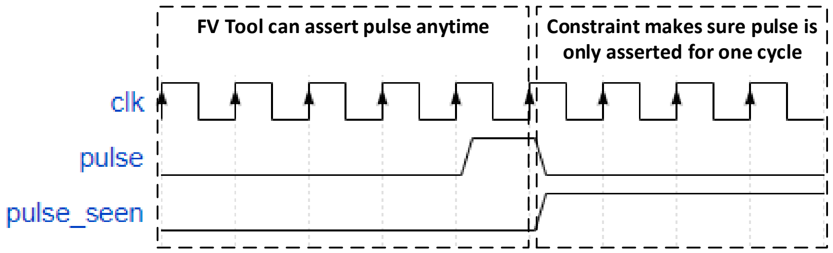
FV Tool can assert pulse anytime
Constraint makes sure pulse is only asserted for one cycle
clk
pulse
pulse_seen

Figure 1: Floating Pulse Behavior

For our work described in this paper, we have used floating pulse to quiesce the incoming traffic at a random time, wait for a specific duration and then compare the design outputs with the predicted outputs. Effectiveness of this technique helps simplify formal reference modelling by achieving cycle independence in a deeply pipelined design.

B. Symbolic Variable

Use of symbolic variables is an efficient formal verification technique for leveraging design symmetries to keep implementation of checkers simple and manage proof complexity. A symbolic variable enables checking of the entire symmetric logic by checking only one unit of it.

Symbolic variables can take any legal value and are kept stable during a formal run.[3] For example, when verifying control logic that reads/writes to a memory, it is sufficient to verify the behavior for one address if the control logic behaves identically across all addresses and all addresses are independent of each other.

In our design, error information gets logged in a Content Addressable Memory (CAM)[4]. We used symbolic variable for tracking sequence of transaction on a symbolic address of the CAM. Since a symbolic variable can take any legal value, transactions for every possible address were covered. Hence, achieved exhaustive coverage of addresses.

C. Formal Coverage

Formal coverage is a powerful way to gauge the quality of formal testbench used to verify a design. In our work, we used reachability coverage, observability coverage and formal coverage metrics to analyze the quality of verification effort. Three coverage metrics are described below:

1. *Reachability Coverage*: This metric determines the extent to which different parts of the design are reachable under a given set of constraints. Reachability coverage helps identify whether all the intended behaviors and states of the design can be reached during the formal verification process. It ensures that the constraints and properties specified for the design exercise all the relevant functionalities.
2. *Observability Coverage*: Observability coverage measures the exhaustiveness of the checkers or assertions implemented in the testbench. It determines the extent to which the design is being observed and verified for specific properties. Observability coverage helps ensure that the checkers adequately capture the desired properties and behaviors of the design.
3. *Formal Coverage*: Formal coverage provides a consolidated view of both reachability and observability coverage. It combines the metrics of reachability and observability to assess the overall verification progress. In order for a design to be considered completely verified with formal methods, it must exhibit full reachability (all parts of the design are reachable) and full observability (all relevant properties are observed and checked).[5]

These metrics collectively help evaluate the quality and completeness of the formal testbench and guide the verification engineer in identifying areas that may require additional coverage or refinement. In our work, formal coverage was enabled at the early stage of the verification cycle to track how close we are to the goal of formal sign-off.

III. Formal Verification Methodology

A. Design Details

The Error Detection Filter is a specialized memory structure designed to manage and track errors in high volume manufacturing environments. This is particularly useful for detecting and managing errors in L3 cache memory, where "erratic bit failures" can cause correctable ECC errors.

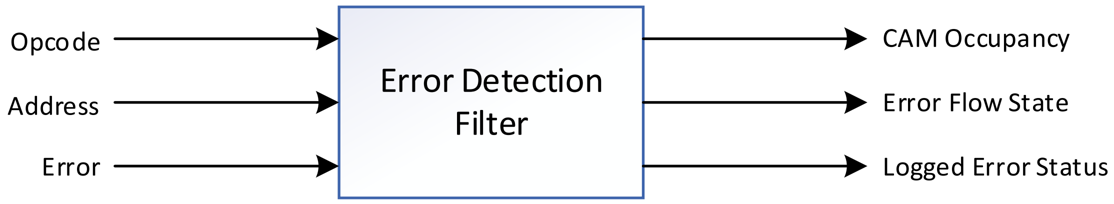

Figure 2: Block Diagram

Key features of an Error Detection Filter include:

1. N entry CAM array: The filter has an N entry CAM array to log errors for unique addresses, providing an efficient way to track and manage correctable errors in the L3 cache.
2. Tracking correctable errors (CE): The Error Detection Filter tracks ECC errors that can be corrected, allowing for more efficient error management and prevention of further issues.
3. Logging defects in DFT (Design for Test) mode: The filter logs defects found during DFT mode [6], which is used in high volume manufacturing to detect and prevent potential issues before they arise in the final product.
4. Managing errors on a per set and way basis: The Error Detection Filter logs errors for unique addresses within the L3 cache, allowing for better analysis and understanding of the issues that arise.
5. Filtering unique correctable errors: The Error Detection Filter is designed to filter up to N unique correctable errors that may occur due to erratic bit failures, helping to maintain the integrity and performance of the L3 cache memory.

Out of reset, all the entries in the CAM are initialized to “NO_ERR”. When a correctable error (CE) of read type is received for an address for the first time, one of the CAM entries is identified to log errors on that particular address and the state for that entry moves from NO_ERR to SOFT_ERR state. After receiving a write to the same address, the state moves from SOFT_ERR state to REPLACE state. This means that there is a replace of the data at that address. If another corrected error is then detected for this entry, the state will be updated to HARD_ERR state. HARD_ERR is a terminal state, which means we exit from HARD_ERR state only on cold reset.

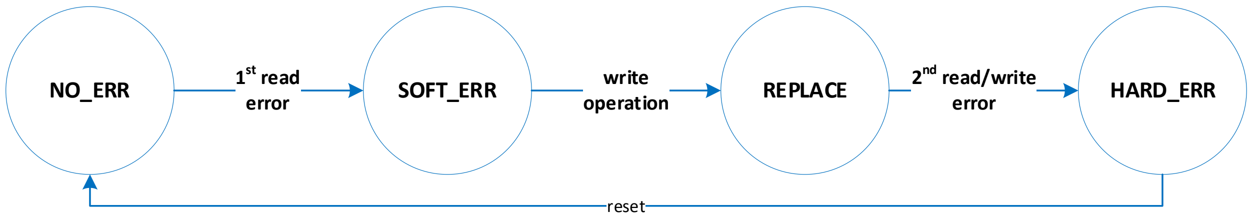

Figure 3: State transition for CAM entries

B. *Verification Challenge*

Error detection filter design has been a challenging area for several generations because of the repeated bug-escapes from IP and SOC DV environments to Silicon. Thorough analysis of bugs found in the past pointed towards following two main contributing factors:

1. *Handling multiple types of incoming errors*: In complex systems, various types of errors can occur simultaneously or at different points in time. These errors can arrive at different pipeline stages, which adds complexity to the error detection filter design.
2. *Variable latencies*: In some cases, the time taken to process and log the errors is not constant. This variability can be attributed to dependencies between different errors, which can make it challenging to design a robust error detection filter.

Creating a simulation environment that can effectively test all possible combinations and sequences of events is difficult and impractical. This is because it would require generating exhaustive stimuli for all potential test vectors, including register programming. Formal Verification can address the challenges faced in error detection filter verification through its breadth-first exhaustive search approach and comprehensive coverage. FV can ensure that no bugs are left undetected.

C. *Implementation Complexities and Formal Verification Solutions*

In this section we will cover four most crucial implementation complexities and describe the solution developed to overcome them in an efficient manner.

1. *Multiple input streams of errors spread across different pipe stages*: The DUT handles multiple types of incoming errors that arrive through different streams of inputs and at different pipeline stages. Misaligned pipeline stages can pose a challenge in efficiently processing and managing errors. This misalignment can lead to incorrect error detection or handling, negatively impacting the system's overall performance and reliability. Creating an accurate reference model is crucial for proper error detection and handling, which requires sampling the data at the correct time and pipeline stage to make well-informed decisions. To address this issue, a solution is to align the pipeline stages at a common point before feeding the data to the FV reference model. The process of aligning pipeline stages involves synchronizing the data from different stages to a single reference point or stage. This ensures that all incoming errors are processed consistently and accurately, regardless of the stage at which they initially arrived. Once the pipeline stages are aligned, the data can be fed into the FV reference model for comprehensive error analysis and detection. By implementing this pipeline alignment strategy, the problem of misaligned pipeline stages is greatly reduced. This results in accurate error detection and simplifies the creation of the reference model, as the data is consistently processed and organized, allowing for a more streamlined approach to error management.

```
always @(posedge fv_clk) begin
   if (en_err_log && en_clk) begin
      // error info received @stage x18
      fv_tag_err_x19 <= tag_err_x18;
      fv_tag_err_x20 <= fv_tag_err_x19;
      fv_tag_err_x21 <= fv_tag_err_x20;
      fv_tag_err_x22 <= fv_tag_err_x21;
      fv_tag_err_x23 <= fv_tag_err_x22;
      fv_tag_err_x24 <= fv_tag_err_x23;
      fv_tag_err_x25 <= fv_tag_err_x24;
   end
end
always @(posedge fv_clk) begin
   // opcode info received @stage x20
   if (pkt_x20.vld && en_clk) fv_pkt_x21.op <= pkt_x20.op;
   if (fv_pkt_x21.vld && en_clk) fv_pkt_x22.op <= fv_pkt_x21.op;
   if (fv_pkt_x22.vld && en_clk) fv_pkt_x23.op <= fv_pkt_x22.op;
```

```
    if (fv_pkt_x23.vld && en_clk) fv_pkt_x24.op <= fv_pkt_x23.op;
    if (fv_pkt_x24.vld && en_clk) fv_pkt_x25.op <= fv_pkt_x24.op;
end
```

Figure 4: Auxiliary code to handle misaligned pipe stages

2. *Multiple CAM entries*: In the design under test, there are multiple entries in the CAM. Writing a reference model for each individual entry in the design can be time-consuming and inconvenient, especially considering that every entry in the CAM is symmetric in nature. This symmetry can be leveraged to simplify the process of creating the reference model. Instead of writing a reference model for each entry, a symbolic variable can be used to track the sequence of transactions on a symbolic address of the CAM. Symbolic variables can take on any legal value, which makes them extremely useful in this context. By using a symbolic variable for tracking transactions, the reference model can cover transactions for every possible address in the CAM. This approach effectively exploits the symmetry of the CAM entries, reducing the need to create multiple reference models for each individual entry, and streamlining the overall design and verification process. This approach ensures that transactions for every possible address are covered, enhancing the efficiency verification process.

3. *Variable output latencies*: Variable latencies in a design can pose a challenge when it comes to accurately verifying the system's behavior. One technique to address this issue is the use of quiesce checking. Quiesce checking helps manage variable latencies by temporarily halting incoming traffic at a random time, waiting for a specific duration, and then comparing the design outputs with the predicted outputs. In this approach, a floating pulse is used to quiesce the incoming traffic. The floating pulse is applied randomly, causing the system to pause or "quiesce" temporarily. During this quiescent period, the system stabilizes, allowing for accurate comparisons between the design outputs and the predicted outputs. Once the specific waiting duration has passed, the design outputs are compared with the predicted outputs to verify the system's behavior. This comparison helps identify any discrepancies, which can then be addressed to improve the overall design. The effectiveness of quiesce checking lies in its ability to simplify formal reference modeling by achieving cycle independence in a deeply pipelined design. This cycle independence allows the system to be evaluated and verified without being affected by variable latencies, ensuring accurate and reliable results.

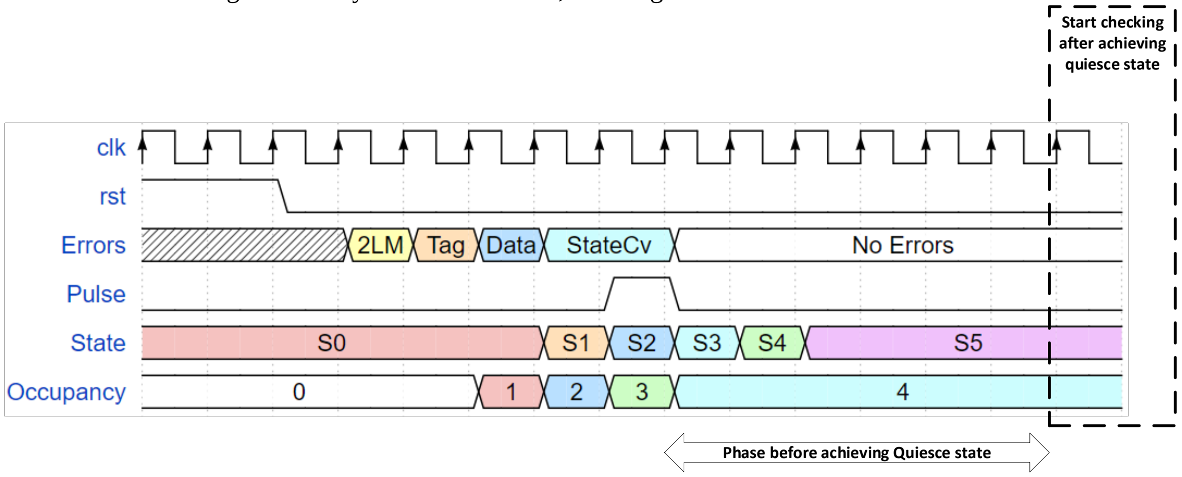
Start checking after achieving quiesce state
clk
rst
Errors
2LM
Tag
Data
StateCv
No Errors
Pulse
State
S0
S1
S2
S3
S4
S5
Occupancy
0
1
2
3
4
Phase before achieving Quiesce state

Figure 5: Waveform showing use of quiesce method

IV. Results

Sign-off capable formal verification environment was implemented form the scratch. Novel and straightforward methods and techniques were devised for implementing different components of formal verification environment. For reference, table 1 captures the statistical details of the design under test (DUT). A total of two formal verification engineers were engaged for 8 engineering weeks to build the formal verification solution and achieve formal sign-off.

Design Statistics for Hardware Configuration with 32 CAM entries (N=32)

Flops	1,367
Latches	11
Gates	47,739
Nets	50,778
Ports	36
RTL Lines	2,989

Formal verification analysis helped find four new bugs in a simulation clean design, most importantly the "last bug" of the design. A couple of bugs were identified to be existing in previous generations of the IP as well. These bugs were extremely complex to uncover due to the nature of sequence of the events required to activate them. Using this approach, we were able to suggest two enhancements which simplified the design implementation and made it more efficient w.r.t area and performance.

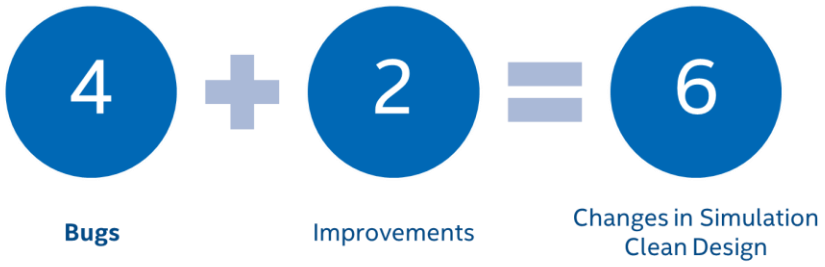

Figure 6: Number of design changes triggered by FV

A. Bug Description

In one of the design bugs found through formal verification, "error flow" state machine did not transition correctly from "soft error" to "replace" state when "write operation" arrives immediately (in the next cycle) after "read error" for same address(represented by set and way).

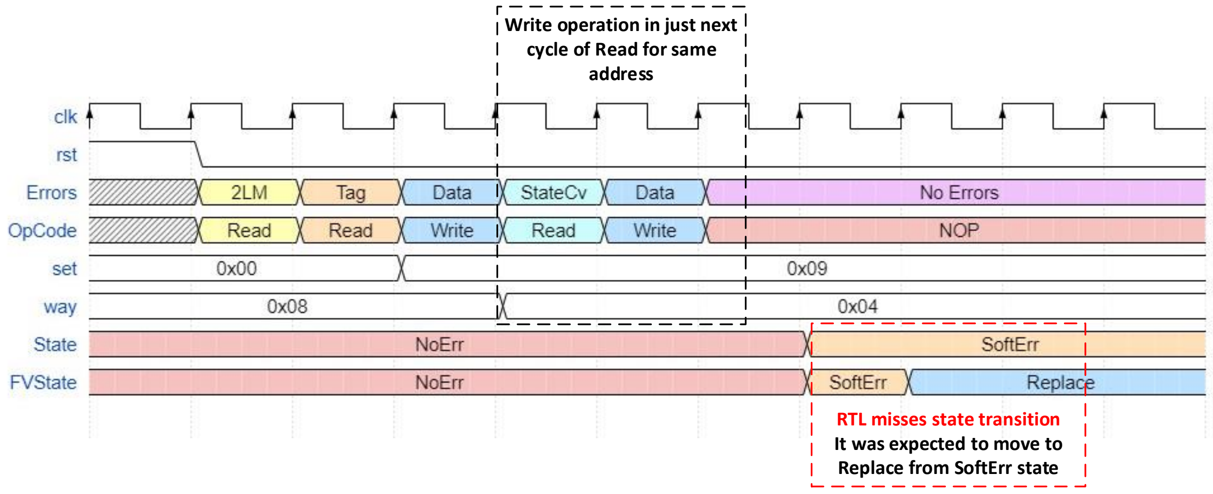
Write operation in just next cycle of Read for same address
clk
rst
Errors
2LM
Tag
Data
StateCv
Data
No Errors
OpCode
Read
Read
Write
Read
Write
NOP
set
0x00
0x09
way
0x08
0x04
State
NoErr
SoftErr
FVState
NoErr
SoftErr
Replace
RTL misses state transition
It was expected to move to Replace from SoftErr state

Figure 7: Waveform showing bug scenario

The root-cause of the bug lies in erroneous way of matching pipeline latencies. In the waveform shown in figure above, even though “write operation” and “read error” was received for the same address (set 0x09, way 0x04), design pipeline captured the “write operation” before the address (set, way) for the write operation be updated. This lead to mapping the “write operation” to incorrect address and corrupting the “error flow” state of address (set=0x09, way=0x04).

B. *Formal Coverage*

We collected formal coverage numbers on weekly basis throughout the implementation of FV testplan. This helped in active tracking the execution progress and identify waivers, but also helped gain confidence that we have found the “last bug” in the design. As you can see in figure 12, constant improvements in coverage metrics shows how the design state space was covered through a well-defined process that helped us achieve exhaustive proofs and high-quality formal sign-off.

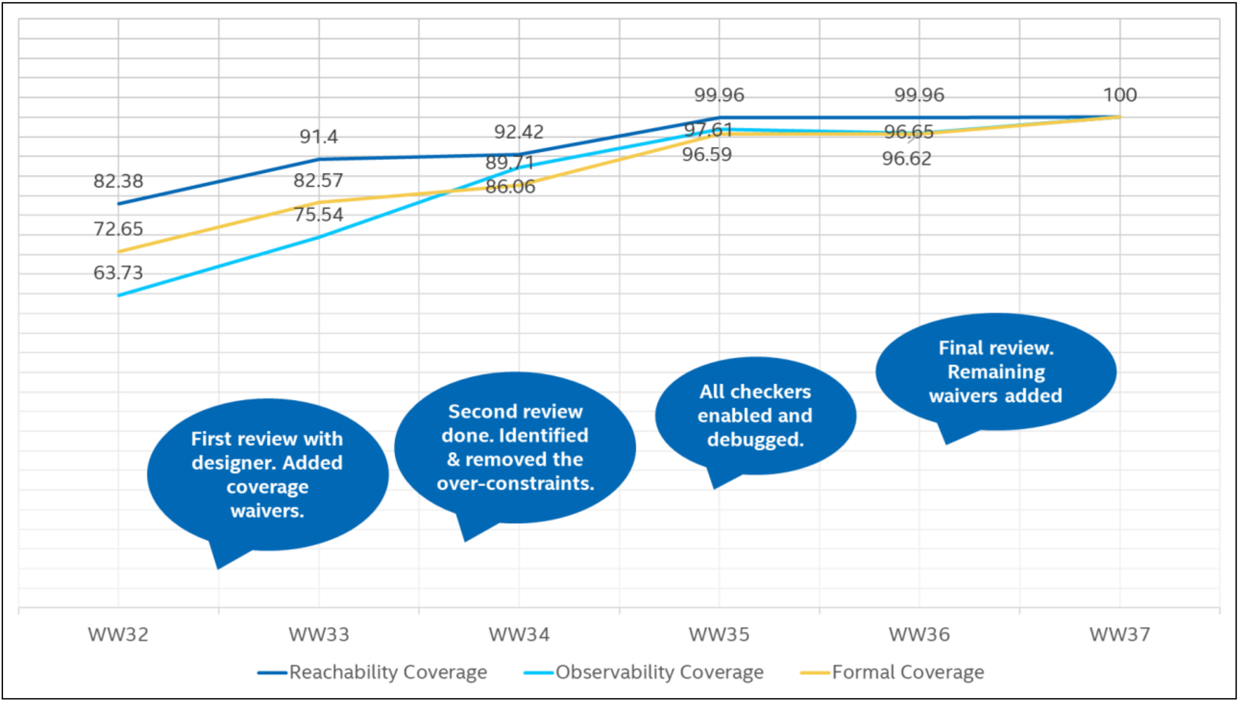

Figure 8: Coverage trend during FV lifecycle

IV. Conclusions

Coverage-driven Formal Property Verification on error detection filter enabled the guarantee of “0 bugs left” on a simulation clean design which had a history of repeated bug escapes from pre-silicon IP & SOC DV analysis. The decision to deploy formal verification on a complex design involving erratic bit errors and deep pipeline proved to be a successful strategy to gain full confidence on the quality of implementation. High-quality formal sign-off of error detection filter design not only helped prevent re-spins and save costs, but also paved way for novel methods for verifying similar class of designs and defined a sound strategy for formal sign-off.

ACKNOWLEDGMENT

Thanks to Praveen Khemalapure and Sudarshan Sridharan Amruthur from Design Team for the support and FVCTO Team for the motivation and the valuable feedback.

REFERENCES

[1] M. Agostinelli, J. Hicks, J. Xu, B. Woolery, K. Mistry*, K. Zhang*, S. Jacobs, J. Jopling, W. Yang, B. Lee# , T. Raz+ , M. Mehalel- , P. Kolar*, Y. Wang*, J. Sandford*, D. Pivin, C. Peterson, M. DiBattista, S. Pae, M. Jones, S. Johnson and G. Subramanian et al, "Erratic Fluctuations of SRAM Cache Vmin at the 90nm Process Technology Node", IEEE InternationalElectron Devices Meeting, IEDM Technical Digest, 2005

[2] T. Patel et al, "Using Formal Sign-Off to Deliver Bug-Free IPs", Oski Decoding Formal Club, Dec 2019

[3] I. Tripathi, A. Saxena, A. Verma, P. Aggarwal et al, "The Process and Proof for Formal Sign-off A Live Case Study", DVCON US 2016

[4] X. Fan, A. Ghonem, and T. Gemmeke et al, "Content-Addressable Memory – Overview and Outlook of an Enabler for Modern Day Applications" ANALOG 2018; 16th GMM/ITG-Symposium, Sep 2018

[5] "Formal Verification: An Essential Toolkit for Modern VLSI Design" by Erik Seligman, Tom Schubert, M V Achutha Kiran Kumar, Elsevier Publications, 2016.

[6] Wang D, Hu Y, Li HW et al, "Design-for-testability features and test implementation of a giga hertz general purpose microprocessor ", Journal of Computer Science and Technology23(6): 1037–1046 Nov. 2008