Formal Verification Approach to Verifying Stream Decoders: Methodology & Findings

Abhishek Asi, Anshul Jain
Intel Corporation
{abhishek.asi, anshul.jain}@intel.com

***Abstract*- This paper investigates a novel formal verification approach for validating the functionality of stream decoders, essential components in modern SoCs responsible for interpreting encoded data. Stream decoders are crucial for tasks such as compression, decompression, error-correction, and secure transmission. Traditional methodologies, including simulation and testing, struggle to address the complexities of evolving formats and standards. Formal verification, a systematic method, offers mathematical proof of design correctness, ensuring reliable operation in all situations. In this paper, we present our formal verification methodology for exhaustive verification of stream decoders and demonstrate the superiority of formal verification in ensuring the reliability and correctness of stream decoders, advocating for its adoption as a rigorous validation tool in complex digital systems.**

I. Introduction

Modern SoCs employ a wide variety of sub-systems to accelerate the computation using specialized hardware by offloading the CPU from certain computation tasks such as communication, encryption, compression, decompression, decryption, multimedia transfer, etc. In all such specialized hardware sub-systems, a stream decoder is an essential component that interprets or converts a stream of encoded data into a format that can be more easily processed or understood by other components of the system. Stream decoders are particularly significant in scenarios where data is transmitted in a specific format to achieve goals like compression, decompression, error-correction, or secure transmission.

As such, the correct functionality of stream decoders is not merely desirable, but imperative. Errors in decoding can result in significant data corruption, misinterpretation, or even total loss of information. The traditional methodologies to validate decoders have largely been based on extensive simulation and testing. However, given the increasing complexity of stream decoders due to evolving formats and standards, conventional methods are beginning to show their limitations. Simulation-based approaches, while effective to a degree, can often miss corner-case scenarios or subtle, intermittent issues, which could become catastrophic in real-world deployments.

Formal verification [1][2] is a systematic approach that offers exhaustive validation of a design against its specification. Instead of relying on extensive test vectors or simulation scenarios, formal verification mathematically proves the correctness of a design, ensuring its operation in all conceivable situations. This paper delves deep into an innovative formal verification approach tailored for stream decoders, elucidating the methodology employed and presenting findings that underscore its effectiveness. Through this exploration, we aim to foster confidence in formal verification as a robust and rigorous tool for ensuring the correctness and reliability of stream decoders in diverse digital systems.

II. Problem Statement

In the current technological landscape, data-centric computation is integral to modern computing. It addresses the challenges posed by the volume, velocity, and variety of data generated today. In this paradigm, the efficiency of data access and data processing is prioritized to improve performance and scalability. File streams play a crucial role in data-centric computation by enabling efficient reading and writing of data to and from files. A file stream refers to an abstraction used in software to read from or write to files on a storage device. It is a sequence of bytes that represents the data contained in a file, which can be processed sequentially. They provide a seamless way to handle large datasets, facilitating data processing and analysis without loading everything into memory at once.

A file stream decoder in the context of digital design and computation is a component that interprets and converts encoded data from a stream of files into a format that is usable by computer systems or applications. This process is

essential when dealing with data that has been compressed or stored in a non-native format. Stream decoders present a unique verification challenge due to their extensive input possibilities. Verifying a decoder design through traditional simulation techniques faces two primary obstacles: first, generating non-corrupt and corrupt files for directed functionality tests, which is difficult due to the potential for missed corner case scenarios that only a unique file could trigger; and second, ensuring the created tests are sufficient for sign-off.

We would like to use the ZSTD [3][4] compression algorithm as an example to illustrate the problem statement further. A ZSTD compressed file consists of one or more frames. Each frame can be either a ZSTD frame or a skippable frame. A ZSTD frame contains compressed data, whereas a skippable frame contains custom user metadata. Figure 1 shows examples of ZSTD compressed files and the format of different types of frames. Most design implementations use "EOF" aka end-of-file indication to convey the end of the incoming input file.

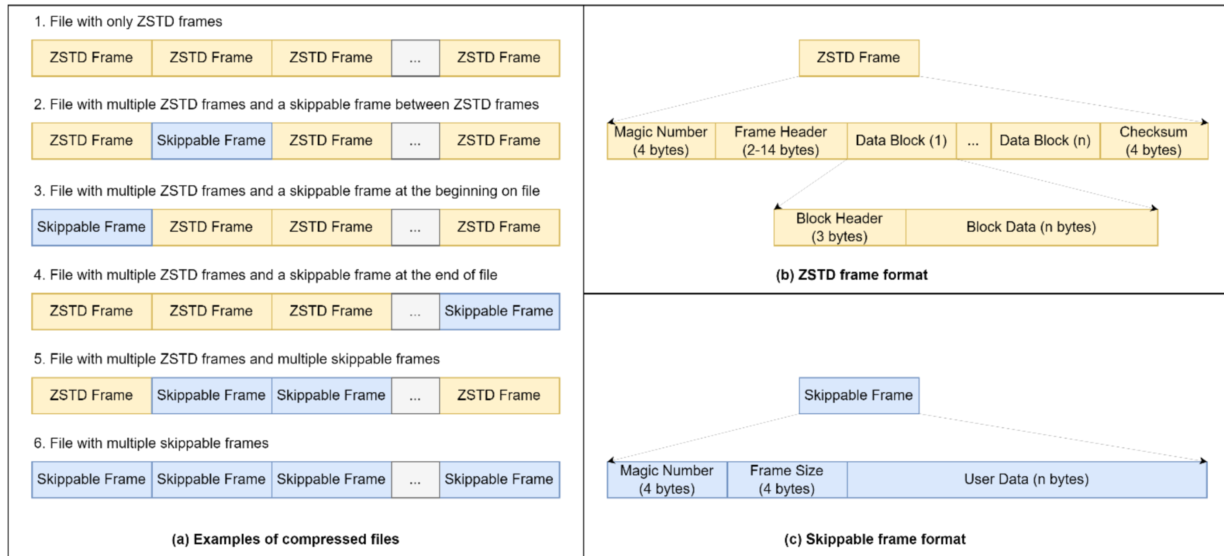

Figure 1: ZSTD Compressed File Formats

Stream decoders process the input file stream to extract the block data packed in ZSTD frames as shown in figure 2. Depending on the data bus, the stream decoder would receive N bytes of compressed file per cycle and send out M bytes of block data per cycle. The decoder looks for a "magic number" when it starts processing the incoming bytes to locate the start of a ZSTD frame.

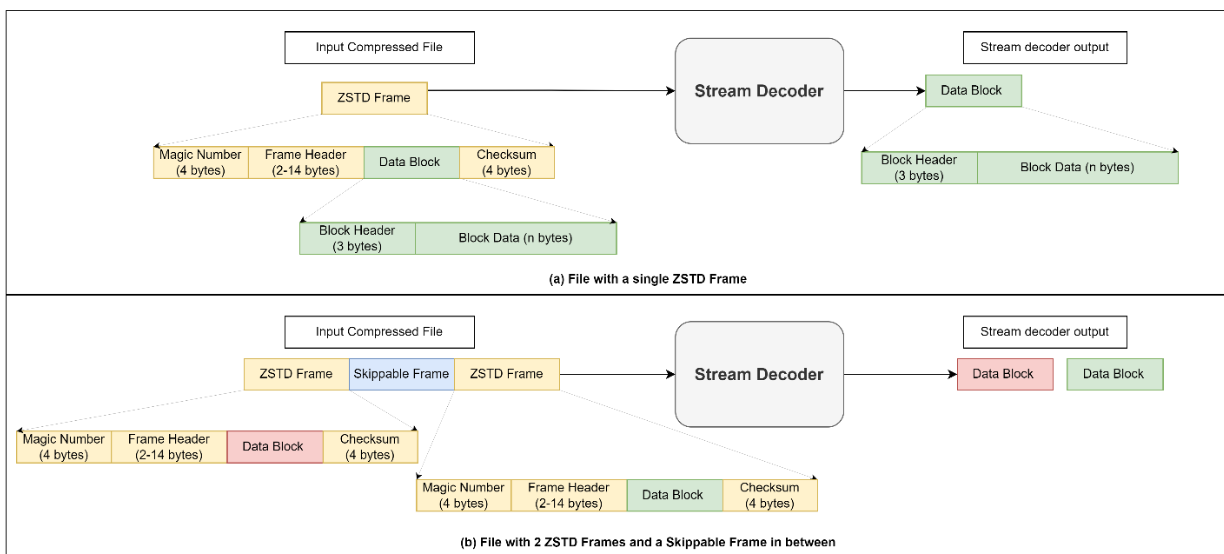

Figure 2: Basic Functionality of Stream Decoder

Simulating all possible ZSTD compressed files for full coverage would require randomizing various components and parameters of the ZSTD format. Here is a list of key variables to randomize:

Variable	Sub-variable	Range
Frame Header	Frame Header Descriptor	4 flags, one each for – frame content size, single segment, checksum, and dictionary
	Window Descriptor	0 to 2^{64} - 1 bytes (16 Exabytes)
	Dictionary ID	4 bytes can represent an ID 0-4294967295
	Frame Content Size	0 - 2^{64} – 1
Data Blocks	Number of Blocks	1- ∞
	Block Size	0 - 2^{21} – 1
	Block Content	Arbitrary
Content Checksum	Optional	Arbitrary
Skippable Frames	User Data	Arbitrary
File Structure	Number of Frames	1- ∞
	Types of Frames	ZSTD only, skippable only, both
	Frame Order	Different arrangements of ZSTD and skippable

Given the complexity and range of these variables, randomizing them to achieve full coverage is a substantial task. This approach requires a significant computation resource, as the number of possible combinations is extremely large. For practical purposes, simulation can focus on only a representative subset of these variables – leaving holes in coverage.

Apart from the basic functionality of extracting the data blocks from the compressed files, stream decoders are also responsible for detecting corrupt compressed files, reporting them, and recovering themselves to normal state for processing the next compressed file. Figure 3 provides examples of some corrupt compressed files.

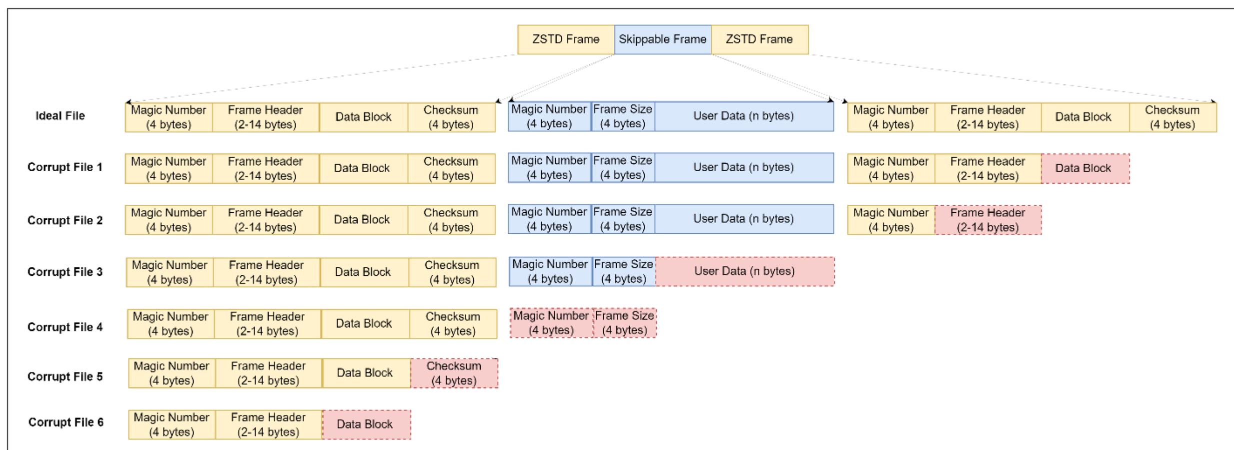

Figure 3: Corrupt file examples

The following examples list some commonly encountered corrupt compressed files in real scenarios.
Corrupt file Example #1: The number of compressed block data bytes received by the stream decoder with the end of file indication is less than the number mentioned in the block header.
Corrupt file Example #2: The number of frame header bytes received by the stream decoder with the end of file indication is less than the header size mentioned in the frame header.
Corrupt file Example #3: The number of skippable data bytes received by the stream decoder with the end of file indication is less than the number mentioned in the skippable frame size field after the magic number.
Corrupt file Example #4: In such a corrupt file, the skippable frame is detected by decoding the magic number (4 bytes) but the frame size field, which is 4 bytes size, is incomplete with the end of file indication.

Corrupt file Example #5: In such a corrupt file, the checksum flag was set for a ZSTD frame, but checksum bytes were insufficient or absent with the end-of-file indication.
Corrupt file Example #6: Each block header within the data block of a ZSTD frame has a bit that denotes whether it is the last block in a frame. If this "last block bit" in the last block of the frame is not set, then it is a corrupt file.

Apart from these examples, there are numerous other types of corruption possible in the input compressed file. Simulating the design with corrupt files involves creating scenarios where the standards are violated including but not limited to altering bits of compressed data, frame header, block size, checksum, or even magic number. Such a wide negative space to verify the error-handling functionality of the stream decoder is impractical. Creating a comprehensive simulation that accurately represents all potential corruption scenarios in ZSTD compressed files is a complex task. It requires not only an in-depth understanding of the ZSTD format but also of how different types of errors and corruptions can manifest in compressed data.

III. METHODOLOGY

In this section, we explore formal property verification solutions to verify stream decoders. Formal technology, known for its breadth-first search of design space, can explore the entire input space, which is not feasible with traditional testing methods. This means that stream decoders can theoretically be verified for every possible variation of a ZSTD file (both corrupt and uncorrupted). However, systematic solutions are required for efficient and effective application of formal property verification in checking stream decoders. We offer three formal verification solutions in this paper and compare them to show which solution is more suitable for different implementation stream decoder-type designs.

Solution #1: Vault & Verify

This is a straightforward approach to verifying the functionality of stream decoder devices under test (DUTs). This technique involves a two-part phase.

1. **Vault**: In this initial phase, the entire compressed file is securely stored in a storage element, typically a shift register. This step ensures that a complete and unaltered version of the original compressed data is retained for subsequent verification.

2. **Verify**: The second phase involves recording the outputs produced by the DUT as it processes the entire compressed file. These outputs, alongside the originally stored compressed file, are then fed into a Formal Verification (FV) model. The FV model runs exhaustive checks based on the algorithmic standards and flags any discrepancies.

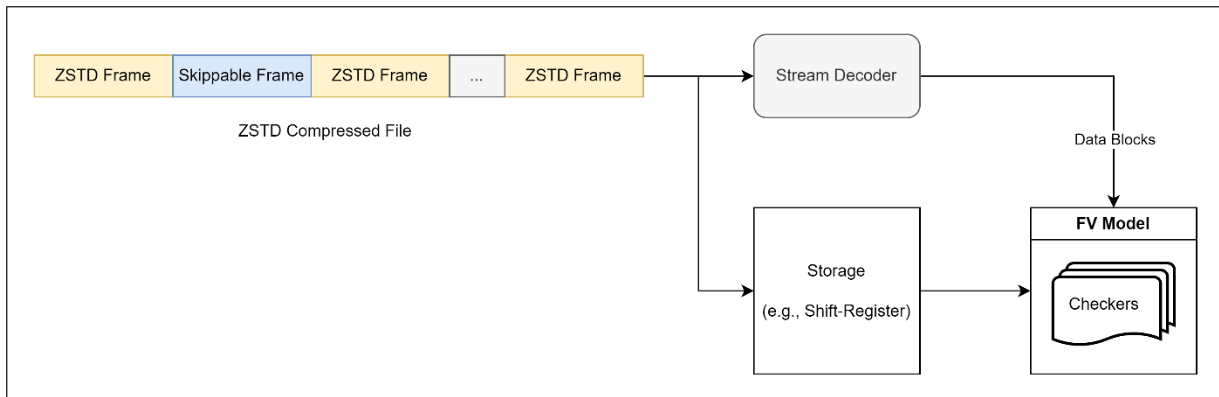

Figure 4: Solution #1 Vault &Verify

The "Vault & Verify" approach, while robust for verifying stream decoder devices under test (DUTs), does have some limitations and drawbacks:

1. **Complexity in Handling Compressed Files**: Storing an entire compressed file, especially if it's large, requires significant memory resources. When dealing with extremely large data sets or streams, the "Vault" part of the process can become complex and unwieldy. This can add exponential complexity for the formal tools to manage. It could also lead to performance bottlenecks.
2. **Coverage Scalability**: Scaling this approach to all possible file sizes is not feasible. Since formal property verification tools do not allow defining dynamic memory storage, the volume of file size will always be limited to a max value.
3. **Incapable of Detecting Hangs**: While effective for verifying the accuracy of outputs from a DUT, this technique typically does not have the capability to detect hangs or freezes in the DUT. It doesn't continuously monitor the DUT's operational state. Therefore, it might miss real-time issues like hangs or response delays that occur during the processing of compressed files.

In summary, while the "Vault & Verify" approach offers thorough verification through a detailed assessment of the DUT's fidelity in reproducing the original data from its compressed form, its resource/complexity intensity can pose significant challenges, particularly in dealing with large files.

Solution #2: Quiesce Check

In this solution, Quiesce Formal Checking (QFC) methodology [8] is explored for a stream decoder design. Logic bugs in hardware systems often take time to become observable. They appear after a delay or "warm-up" period. This means that the bug only becomes active after certain events or conditions have happened within the system. When such bugs get activated, they may not cause immediate problems, but over time, they can lead to significant design flaws that can be detected by high-level end-to-end checkers.

As shown in Figure 5, imagine a design with two events: event A at cycle 3 and event B at cycle 5. These events act like a "warm-up" for a hidden bug, making it active. However, the bug doesn't cause problems until cycle 10. Even after the bug is active, it may take time for it to cause noticeable problems. This makes sophisticated bugs hard to find and fix because they don't show up right away and can be hard to trace back to their root cause. High-level end-to-end checkers can help find these bugs by looking for unusual behavior in the system.

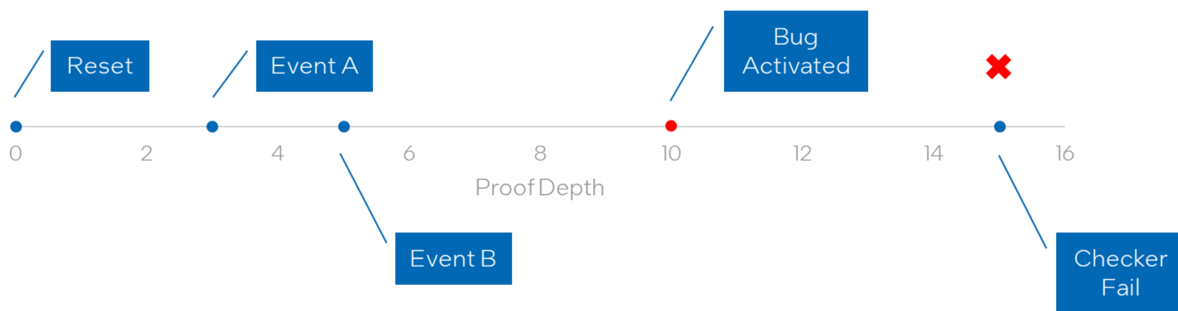

Figure 5: "Warm-up" for a hidden bug.

In the case of stream decoders, we can see a similar “warm-up” period for the input data byte corruption detection logic to report a soft error, hence, any bug in this corruption detection logic when an illegal input sequence is received at the input, bug activation, surfaces after a certain delay only, depending on the current state of the decoder and processing speed of the decoder. Thus, QFC seems the right approach to detect hangs, verify the error detection logic, or verify the absence of inconsistencies in the error detection logic of the decoder.

To implement QFC methodology in stream decoder designs, one needs to identify critical signals that must have consistent behavior, for example, let’s consider an “empty” signal output by a stream decoder. The “empty” signal denotes that the decoder has no pending input data bytes to be processed. This empty signal must be asserted after reset. On receiving the “quiesce” command [6][7], the decoder must process the pending input data bytes and

eventually within finite cycles assert this "empty" signal again. Thus, using the QFC methodology verifying the consistency of such signals is possible.

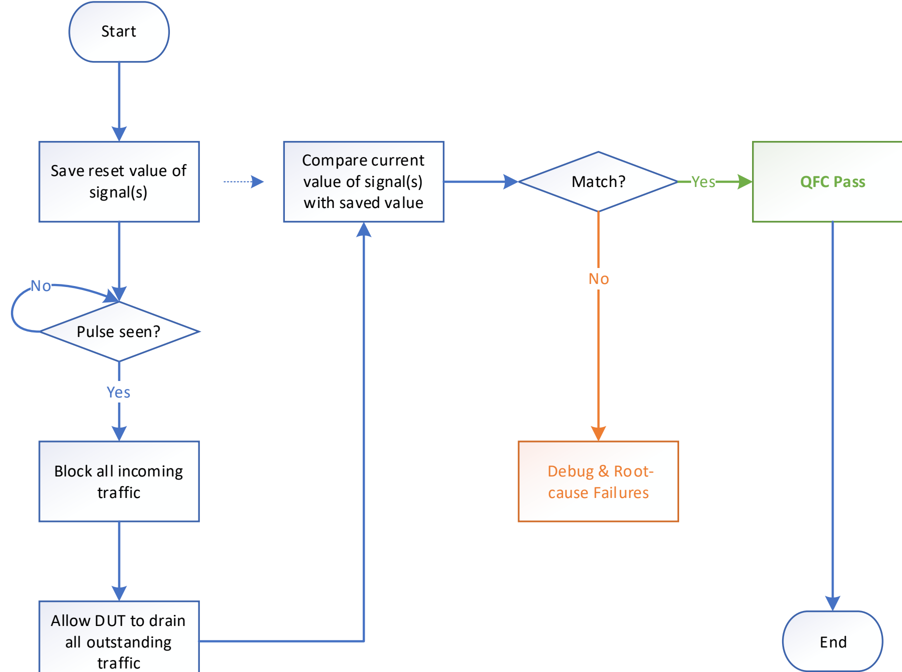

Figure 6: QFC Framework

QFC is an efficient method to check for hangs in the DUT which overcomes the scalability issues of the Vault & Verify approach, however, it is incapable of verifying the correctness of design behavior and is in-exhaustive. Therefore, it can be used to complement the overall verification strategy but cannot be the main workhorse for exhaustive verification of the stream decoders.

Solution #3: Instant Inspection

In this approach, we dissect the verification goals into the following five major categories which allows us to build a verification environment capable of comprehensive assessment of the stream decoder while overcoming the shortcomings of the previously discussed solutions. Instead of analyzing the entire compressed file, we analyze a sliding window (of a much smaller width compared to the entire file) of the file to predict DUT's behavior on various fronts. In this approach, the width of the sliding window is a function of file format. For example, in the case of a ZSTD compressed file, the sliding window width should be 14 bytes as that is the largest chunk of control information available in contiguous form.

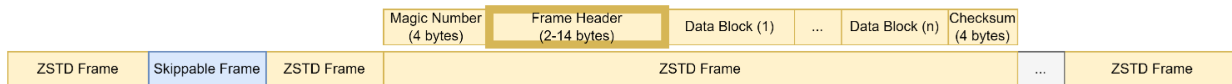

Figure 7: Sliding window width in ZSTD Stream Decoder

1. **Status Flag Verification-** Most of the DUTs have output flags to denote certain events. For example, "event start flag – asserted when an event occurs" or "job done flag – asserted by DUT to denote it is done with decoding the input file bytes". These flags can be verified by storing sliding window-sized bytes in the FV model and creating checkers based on the information extracted from the stored bytes.
2. **Data Integrity-** DUTs forward data decoded from the input file to the output, and it is imperative to ensure the integrity of this forwarded data. To ensure the integrity of the decoded data at the output of DUTs Formal Scoreboard [5], a utility provided by formal verification tool vendors can be utilized. In the case of the ZSTD stream decoder, the 14 bytes stored can be decoded and sent to the Scoreboard instantaneously as it is received at the input.

3. **Data Ordering-** The order of data bytes of any file is important to reproduce the file when needed. Hence, the order of the decoded file forwarded by DUT must be maintained to avoid corruption in the decoded data being forwarded. To ensure this, checkers can be implemented using techniques such as Wolper Technique [9], Data Coloring Techniques, etc.
4. **Latency Verification-** The decoding algorithm inside the DUT is usually implemented using a finite state machine (FSM) and it is important to ensure that the DUT is not stuck in a particular state leading to hangs/deadlocks. The 14 bytes stored by the formal model in the case of ZSTD Stream Decoder can be decoded to identify the state the DUT is in currently and implement liveliness checkers using the decoded information to ensure the forward progress of the DUT.
5. **Error Handling-** The sliding window-sized bytes stored in the formal model can be decoded by the formal model almost as instantly as the DUT and checkers can be implemented using the specification for the error conditions using the decoded bytes. This helps in detecting corruption in the incoming file quickly and aids in scaling for all possible file sizes.

Since the **Instant Inspection** approach relies on inspecting a sliding window of the compressed file, therefore it requires an accurate and detailed description of the intermediate checkpoints often documented in microarchitectural specifications created for aiding design implementation. Creating an exhaustive microarchitecture specification that accounts for all possible illegal inputs is complex. Considering the vast amount of data a decoder processes, there is a significant challenge in ensuring the microarchitectural specification is complete and unambiguous. In most cases we have encountered, microarchitectural specification was found to have many gaps.

To tackle this challenge, we devised a **hypothesis-based method** to implement the **Instant Inspection** solution. In this method, a formal specification was developed based on the compressed file format and decoding algorithm instead of relying solely on incomplete and ambiguous microarchitectural specification. First, a formal property is defined and a hypothesis of events at the interface signals is developed. On executing the property, if it does not fail, it means that the hypothesis is correct, and the design implementation adheres to it. But if it fails, it means we need to verify if the hypothesis is in accordance with the decoding algorithm, if not, the hypothesis must be redefined. If the hypothesis is in accordance with the decoding algorithm, then the failing scenario must be debugged, discussed with designer and micro-architect for the validity of failure. If the failure is a valid scenario, the design needs to be fixed. If not, then microarchitectural specification should be updated and the hypothesis should be refined.

In some cases, it might be difficult to visualize the scenario and comment on the validity of failure. In such cases, we can leverage the simulation environment to simulate the design with the compressed file seen in the formal verification counterexample. It can be used to extend the trace up to the end-of-file to identify if the failure is real or false.

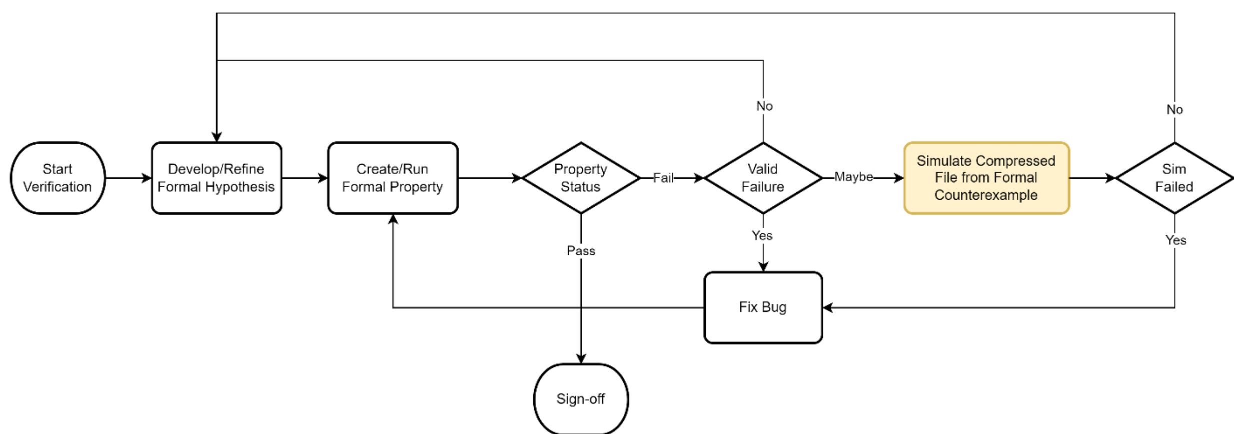

Figure 8: Hypothesis Based Property Verification

To summarize all three approaches in a comparative form, we rate them on three parameters – scope of verification (bug-hunting vs sign-off), ease of verification environment implementation, and formal complexity as shown in the

figure below. Instant Inspection is more complex to implement, however, it helps cover the design behavior exhaustively while managing formal complexity well.

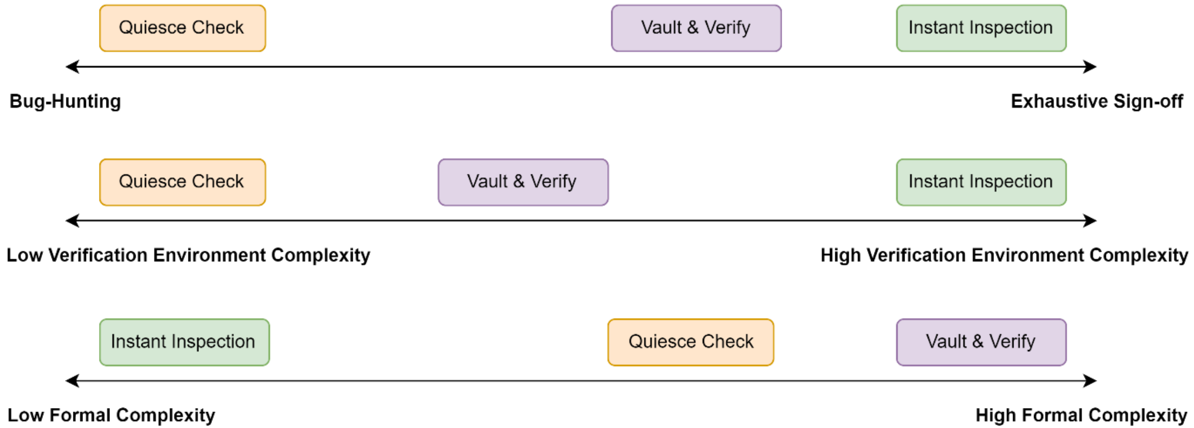

Figure 9: Comparative View of Suggested Solutions

IV. Case Study – Decompression Stream Decoder

In this paper, we focus on the challenges of verifying a class of stream decoders used for processing files compressed using ZSTD that is used in Intel's Xeon CPUs targeted for data-center applications. Like most of the stream decoders, the ZSTD stream decoder processes frames of ZSTD compressed files and forwards the payload associated with the frame.

Block Diagram

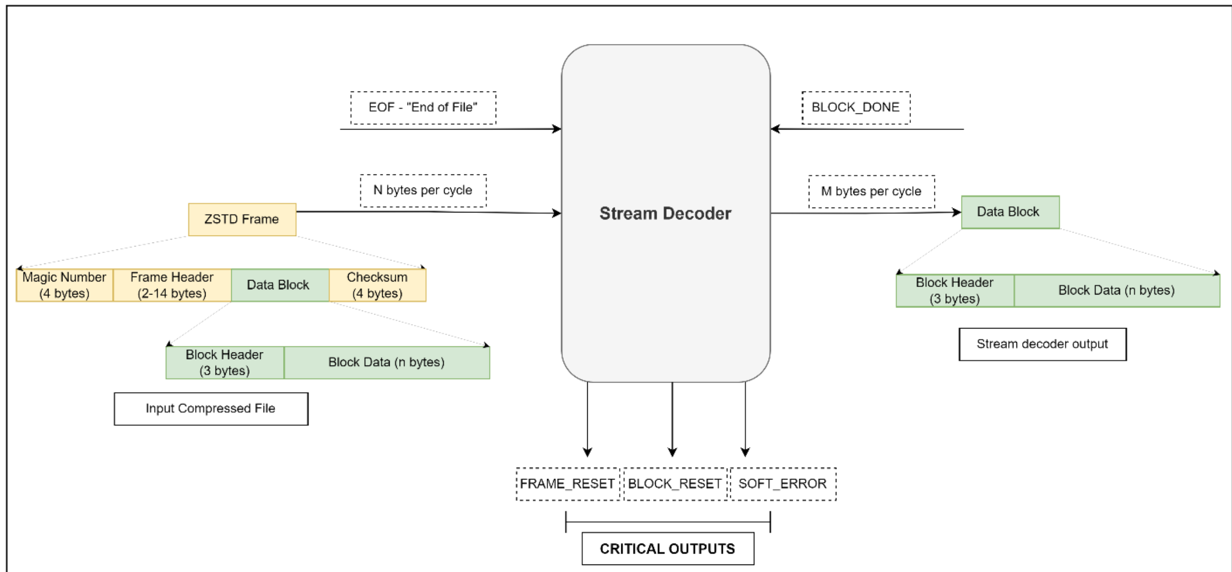

Figure 10: ZSTD Stream Decoder Design

Design Details

ZSTD Stream Decoder design implementation processes the incoming compressed file to extract "data blocks" from the ZSTD frames. It uses the "frame header" to understand how to decode the data block and the checksum to ensure data integrity. Extracted "data blocks" are forwarded to the subsequent logic that performs decompression operation.

ZSTD Stream Decoder design implementation uses certain (input and output) control flags to manage the movement of information in the compressed file. These flags are described below:

1. EOF: This input flag signifies the end of the incoming compressed file. In the context of our stream decoder, EOF indicates that the decoder should expect no more data after this point for the current file.
2. BLOCK_DONE: This input flag indicates an acknowledgment by subsequent decompression logic that it is done decompressing the previously sent "data block" and is ready to receive the next "data block". The stream decoder begins processing subsequent blocks only after receiving this acknowledgment.
3. FRAME_RESET: This output flag indicates that the decoder is resetting its current state to prepare for a fresh start, ensuring that no residual data from the previous frame affects the decoding of the new one.
4. BLOCK_RESET: This output flag indicates that the decoder is starting to decode a new block within the same frame. It clears any settings or temporary data that were specific to the previously decoded block.
5. SOFT_ERROR: This output flag is a critical feature of a stream decoder as it flags corruption in the input compressed file and reports it to the firmware. Any corrupt file if goes undetected can have many consequences like a deadlock or security issue.

In terms of design size, the ZSTD stream decoder was a 90-thousand gate design with a total of 50 IO ports.

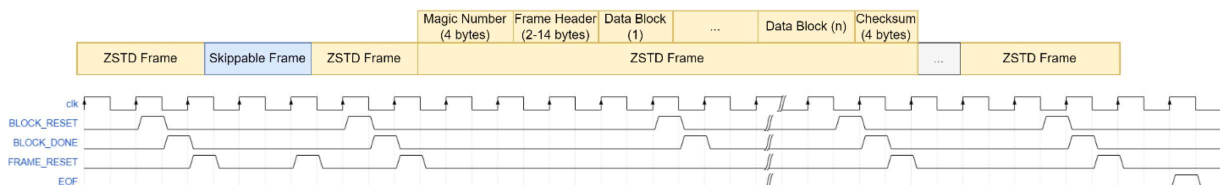

Figure 11: Sample Design Waveform

V. RESULTS

In our comprehensive study, we identified a total of 36 bugs in the system, highlighting the efficacy of our **Instant Inspection** approach in uncovering a variety of elusive design issues. These bugs were not only numerous but also represented a spectrum of challenging corner case scenarios, demonstrating the depth and thoroughness of our analysis. Table 2 encapsulates the distribution of bugs across critical outputs and internal counters. This tabulation not only provides a quantitative summary of our findings but also serves as a testament to the comprehensive nature of our investigation.

Table 1: Categories & Number of Bugs Found

Category of bugs	Number of bugs
Soft error-related issues	15
Frame/Block reset related issues	3
Deadlocks	2
Counter overflow/underflow	4
Incorrect FSM transition	5
Data Integrity	3
Microarchitecture Specification	4

A significant observation was the nature of these bugs: they were typically hard to pinpoint through conventional testing methods. This difficulty stems from the inherent complexity of designing directed tests that could precisely replicate the specific conditions under which these bugs manifest. To address this challenge, our simulation team leveraged the bug scenarios unearthed by Formal Property Verification (FPV) to develop targeted tests. This innovative strategy enabled us to recreate these bugs within a controlled simulation environment, thereby validating their existence and understanding their behavior in detail.

In addition to these quantitative results, our study also presents several intriguing corner-case scenarios that were unearthed through our methodology. These examples serve to illustrate the complexity of the bugs we encountered and the sophistication of our approach to detecting them. We would like to highlight some of the interesting bugs found as follows.

Bug #1: Block Reset erroneously asserted.

A hypothesis was formulated that **"the "BLOCK_RESET" signal must be asserted whenever the decoder has completed processing the current block and starts processing the next block"**. This hypothesis went through multiple failures (shown in Table 2) to finally detect a buggy case for a specific compressed file.

Table 2: Example of how Hypothesis-based Method Formulated for "BLOCK_RESET" Flag.

Version	Hypothesis	Outcome	Root cause
1.	Whenever the ZSTD Frame Magic Number is seen, the FV model starts tracking the frame and stops tracking it when the last block of this frame is completely processed. When ZSTD Frame Header bytes are decoded, the incoming bytes after those bytes must be of a Block, so the FV model starts tracking the Block and de-asserts the tracking once the data bytes of that block are forwarded. Assert "BLOCK_RESET" a cycle after whenever the Block tracking de-asserts.	False Failure	False failure as the hypothesis on tracking of Block was insufficient. It did not include the fact **that the decoder must receive an acknowledgment "BLOCK_DONE" from the subsequent decompression logic before moving to the next block and assert "BLOCK_RESET"**
2.	Whenever the ZSTD Frame Magic Number is seen, the FV model starts tracking the frame and stops tracking it when the last block of this frame is completely processed. When ZSTD Frame Header bytes are decoded, the incoming bytes after those bytes must be of a Block, so the FV model starts tracking the Block and de-asserts the tracking once the data bytes of that block are forwarded and **"BLOCK_DONE" acknowledgment is received.** Assert "BLOCK_RESET" a cycle after whenever the Block tracking de-asserts.	False Failure	False failure as the design implementation did not intend a "BLOCK_RESET" when the current block is the last block of the decompressed input file. **This detail was missed in the microarchitectural specification and was uncovered through this false failure.**
3.	Whenever the ZSTD Frame Magic Number is seen, the FV model starts tracking the frame and stops tracking it when the last block of this frame is completely processed. When ZSTD Frame Header bytes are decoded, the incoming bytes after those bytes must be of a Block, so the FV model starts tracking the Block and de-asserts the tracking once the data bytes of that block are forwarded and "BLOCK_DONE" acknowledgment is received. Assert "BLOCK_RESET" a cycle after whenever the Block tracking de-asserts, **only if the block is not the last block of the input file**.	**Real Failure**	**RTL Bug was found**. Between two ZSTD Frame Blocks, the Block size counter internal to the design implementation did not reset properly and a Block reset for the second Block was seen earlier than expected. **RTL bug was fixed after verifying in the simulation environment.**
4.	Whenever the ZSTD Frame Magic Number is seen, the FV model starts tracking the frame and stops tracking it when the last block of this frame is completely processed.	Pass	

	When ZSTD Frame Header bytes are decoded, the incoming bytes after those bytes must be of a Block, so the FV model starts tracking the Block and de-asserts the tracking once the data bytes of that block are forwarded and "BLOCK_DONE" acknowledgment is received. Assert "BLOCK_RESET" a cycle after whenever the tracking de-asserts, only if the block is not the last block of the input file.		

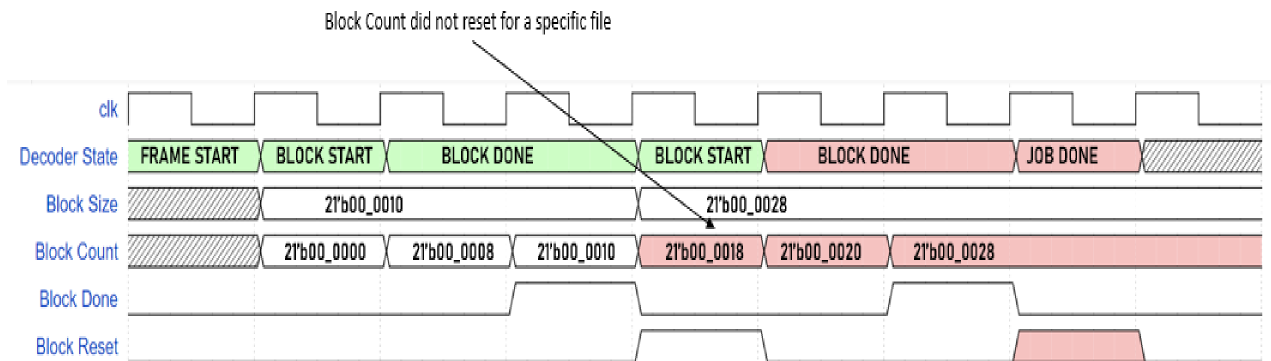

Figure 12: Block reset erroneously asserted waveform.

<u>Bug #2: Deadlock Due to Incorrect Block Decoding</u>

A **hypothesis** was formulated with the rule that **"the stream decoder must eventually move to IDLE state gracefully whatever be the compressed input file (corrupt or non-corrupt)".** In this case, when EOF is received and if the second last block's size is greater than equal to 8 (in this case it was equal to 8), the decoder should have moved to the "BLK_END" state as shown in green in the state diagram in Figure 13, instead, it moved to "BLK_START", shown in red, which in turn did not update the "BLK SIZE" value for the last block received by decoder leading to subsequent incorrect comparison, resulting in a deadlock. This deadlock happens only if the second last block's size is 8 bytes.

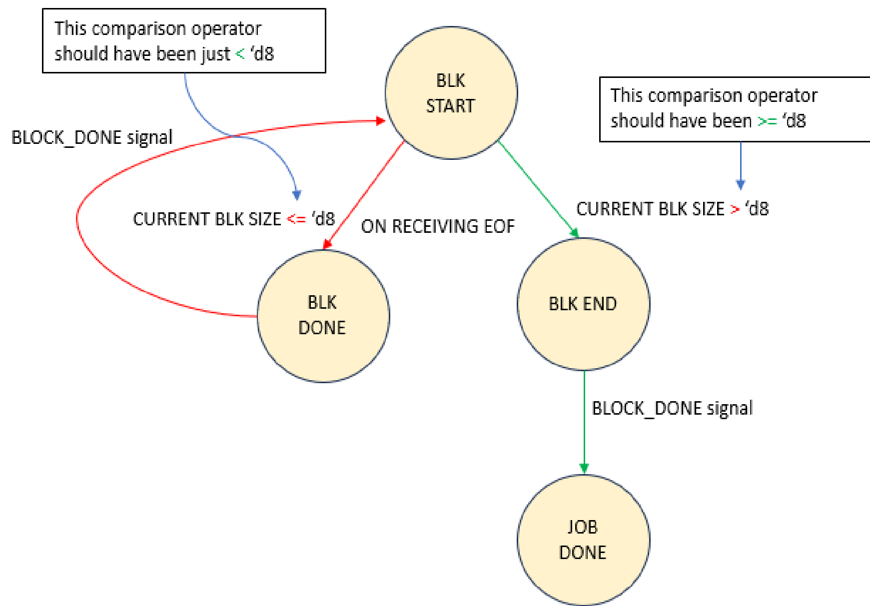

Figure 13: Deadlock due to incorrect comparison

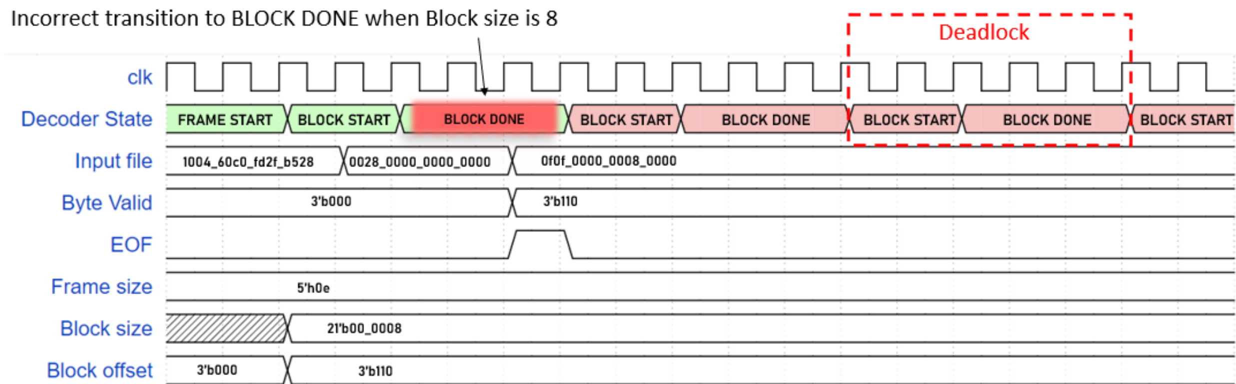

Figure 14: Deadlock due to incorrect Block decoding waveform.

Bug #3: Unable to Detect Corrupt File

This bug was identified in the error detection system of the Stream Decoder. Specifically, the decoder was unable to detect a corrupt (compressed) file. Our **hypothesis** was that “**the stream decoder must report an error if a compressed file had insufficient bytes to decode a frame (ZSTD and Skippable)**”. In the case of an unusually corrupt (compressed) file that ended with a Skippable frame with very few bytes of user data (less than equal to one byte), design implementation was unable to detect that a Skippable frame should have a minimum of 8 bytes (4 bytes of magic number + 4 bytes of frame size) and continued decoding the corrupt file.

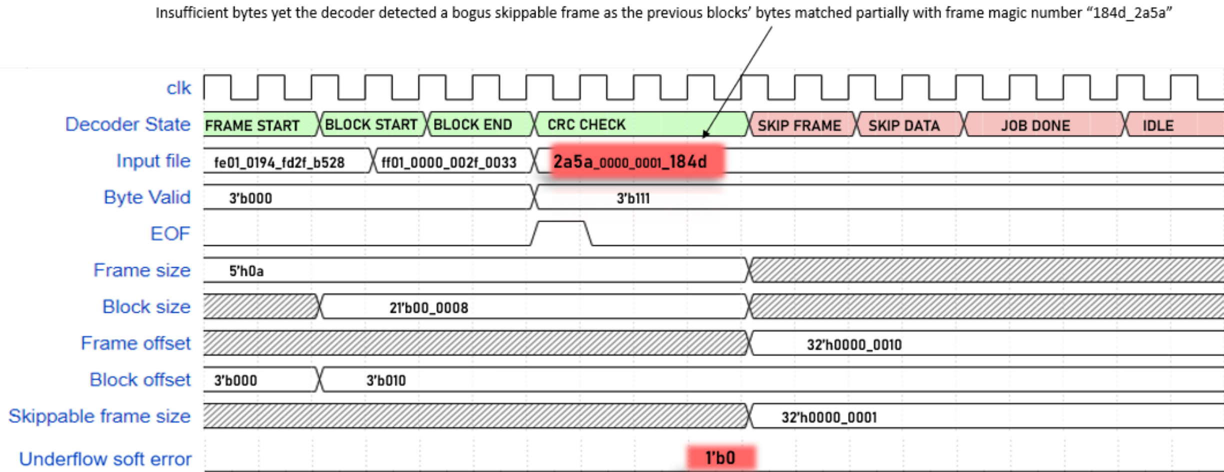

Figure 15: Bogus skippable frame detection waveform.

Bug #4: Unreported Block Underflow Error

This bug was identified in the error reporting system of the Stream Decoder. Specifically, the decoder successfully detected a block underflow error but failed to report it to the firmware via the error reporting register. If this bug had gone unnoticed, the firmware would have missed such underflow errors, potentially leading to undetected file corruption. As such, the microarchitectural specification document did not require such an error to be reported to firmware. However, our **hypothesis** was that "**all errors identified by the stream decoder must be reported to firmware**" Our hypothesis led us to (1) finding this issue in the design implementation and (2) flagging incompleteness of the microarchitectural specification. Our approach highlights the necessity of a comprehensive understanding of the design intent and the potential limitations of relying exclusively on the microarchitecture specification for stream decoder designs.

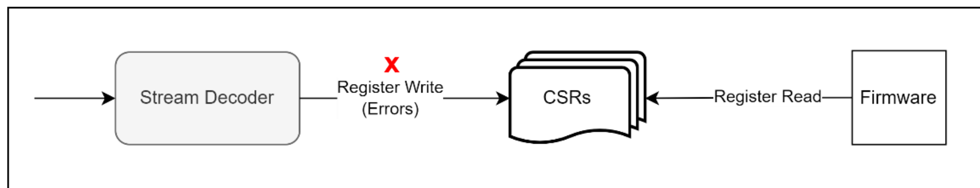

Figure 16: Unreported Block Underflow Error

To summarize, the results of our study demonstrate a significant advancement in bug detection methodologies, particularly in identifying and understanding complex, hard-to-find bugs in critical systems. The breadth and depth of bugs uncovered affirm the robustness and effectiveness of our approach, paving the way for more reliable and error-resistant system designs.

VI. Conclusion

In conclusion, this paper has demonstrated the effectiveness of a formal verification approach specifically designed for stream decoders in modern SoCs. As traditional methodologies, such as simulation and testing, struggle to address the increasing complexity of stream decoders due to evolving formats and standards, formal verification emerges as a powerful alternative. By mathematically proving the correctness of a design, formal verification ensures reliable operation in all conceivable situations, addressing potential corner-case scenarios and subtle issues that may be missed by conventional methods. The presented methodology and findings in this paper aim to bolster confidence in formal verification as a robust and rigorous tool for guaranteeing the accuracy and dependability of stream decoders in a wide range of digital systems. This adoption of formal verification can lead to improved system performance, reduced risk of data corruption, and ultimately, enhanced user experiences across various applications.

Acknowledgment

Thanks to our management for their support, Qumrul Ahsan from the dynamic simulation team, Virginia Bao, Mark Yarch, and Peter Graniello from the design team for helping in development efforts.

References

[1] E. Seligman, T. Schubert and M.V.A. Kiran Kumar, Formal Verification: An Essential Toolkit for Modern VLSI Design, 2015.
[2] M Achutha KiranKumar, et al., “Making Formal Property Verification Mainstream: An Intel® Graphics Experience,” DVCon 2017
[3] ZSTD Compression format standard - https://www.rfc-editor.org/rfc/rfc8478
[4] ZSTD Github repository - https://github.com/facebook/zstd
[5] OIL check of PCIe with Formal Verification, Vedprakash Mishra, Carlston Lim, Zhi Feng Lee, Jian Zhong Wang, Anshul Jain and Achutha KiranKumar V M, DVCon India 2022
[6] IBM, “Quiesce command”. IBM Documentation, https://www.ibm.com/docs/en/db2/11.5?topic=commands-quiesce, Jan 2023.
[7] IBM, “Unquiesce command”. IBM Documentation, https://www.ibm.com/docs/en/db2/11.5?topic=commands-unquiesce, Jan 2023
[8] Quiescent Formal Checks (QFC) for Detecting Deep Design Bugs – Sooner and Faster, Somesh et sl., DVCon India 2023
[9] Pierre Wolper, “Expressing interesting properties of program in propositional temporal logic” 13th ACM SIGACTSIGPLAN symposium on Principles of programming languages, pages 184-193. ACM Press, 1986.